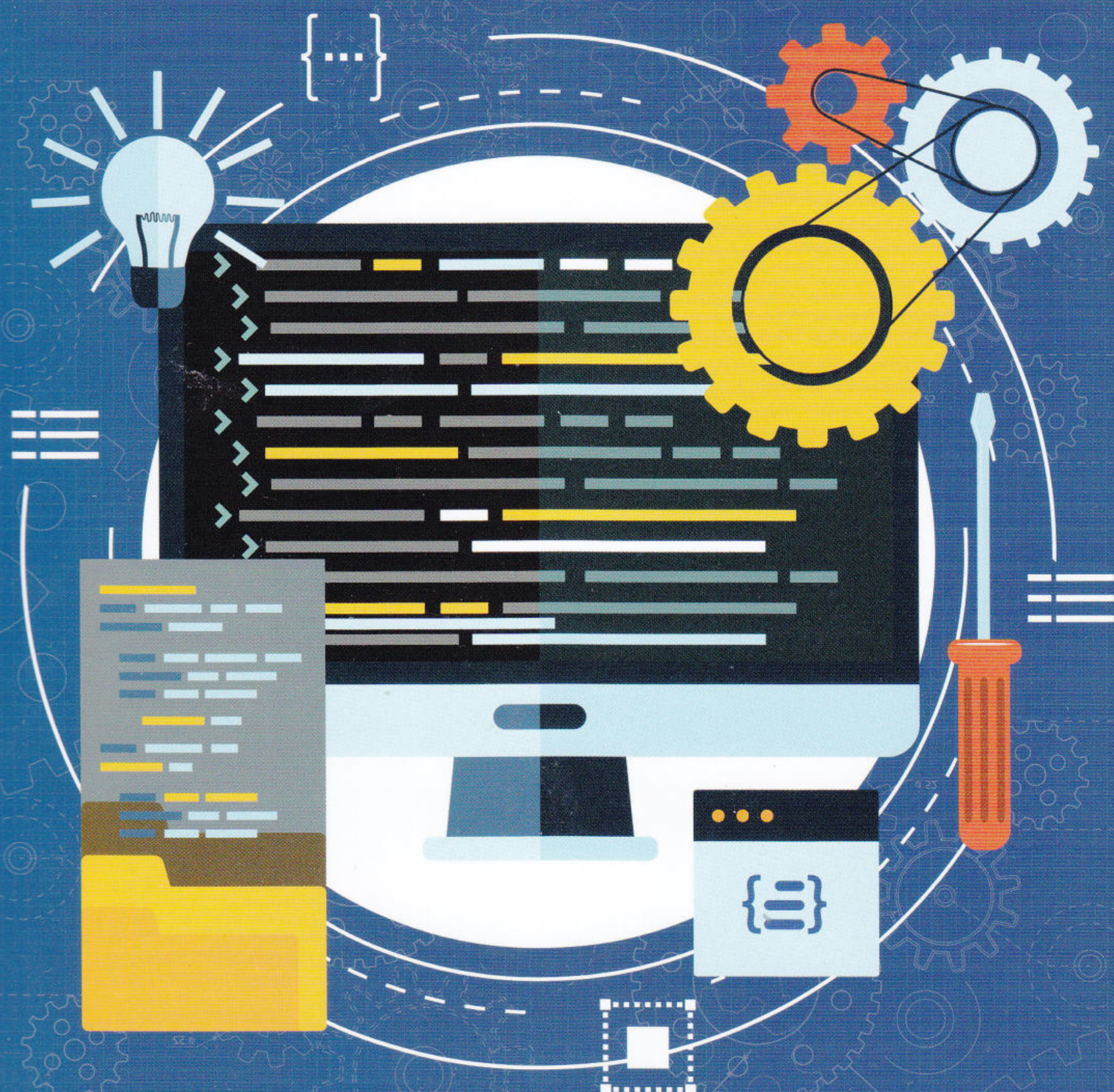
MAGYARY GYULA

> EMELT SZINTŰ INFORMATIKA ÉRETTSÉGI 2.

PYTHON LÉPÉSRŐL LÉPÉSRE

OBJEKTUMORIENTÁLT PROGRAMOZÁSSAL ÉS WINDOWS ALKALMAZÁS KÉSZÍTÉSÉVEL KIEGÉSZÍTVE



TARTALOMJEGYZÉK

ELŐSZÓ 7

I. ISMERKEDÉS A PROGRAMOZÁSSAL 9
1. Parancsok 9
2. Program 11
3. Eljárás 13
4. Változók 15
5. Elágazás 17
6. Számláló ciklus 19
7. Változók használata 21
8. Véletlenszám 25
9. Lista 27
10. Feltételes ciklus 30
11. Rekurzió 32

II. A PROGRAMOZÁS ALAPJAI 35
1. Egy program általános felépítése 35
2. Algoritmus 36
3. Programnyelvek 39
4. Hibák 42
5. A programfejlesztés lépései 44

III. PYTHON ALAPOK 45
1. "Helló világ!" 45
2. Alapműveletek 46
3. Változók 48
4. Program 51
5. Adatbekérés 53
6. Formázott kiíratás 55
7. Véletlenszámok 58
8. Megjegyzések a programban 60
9. Logikai változó, logikai érték 62
10. Gyakorlófeladatok 66

IV. ALAPVETŐ VEZÉRLÉSI SZERKEZETEK 69
1. Sorrendi végrehajtás 69
2. Feltételes elágazás – alapok 69
3. Elágazás több irányba (elif) 74
4. Elágazás összetett feltétellel (not, and, or) 76
5. Karakterek és karakterláncok a feltételben 83
6. Teljes lefedés elve 87
7. Ciklusok 88
8. Számláló ciklus 88
9. Ciklusok egymásba ágyazása 93
10. Feltételes ciklusok 95
11. Alprogramok 100
12. Eljárások 102
13. Függvények 107
14. Rekurzió 110
15. Moduláris programozás és tesztelés 113

V. ÖSSZETETT ADATSZERKEZETEK 119
1. Lista 119
2. Lista bejárása (for) 125
3. A karakterlánc mint lista 127
4. További műveletek karakterláncokkal 132
5. Listában a lista, többdimenziós lista 136
6. Listák az eljárásokban és függvényekben 139
7. Szótárak 144
8. Rekordok 149
9. Egyszerű szöveges fájlok 155
10. Fájlok beolvasása 158
11. Kiírás fájlba 164

VI. PROGRAMOZÁSI TÉTELEK 169
1. Bevezetés 169
2. Sorozatszámítás 170
3. Összesítés 172
4. Megszámlálás 175
5. Maximumkiválasztás 178
6. Eldöntés 180
7. Lineáris keresés 184
8. Kiválogatás 186
9. Rendezés 188
10. Feladatsorok megoldása 194

VII. OBJEKTUMORIENTÁLT PROGRAMOZÁS 205
1. Objektum és osztály 205
2. Az objektum részei 206
3. Különleges metódusok és változók 209
4. Objektumosztály tervezése 212
5. Objektumosztály kódolása 214
6. Objektumlista és a control objektum 219
7. Az objektumorientált programozás további fogalmai 222

VIII. GRAFIKUS ALKALMAZÁS KÉSZÍTÉSE 225
1. Egyszerű grafikus alkalmazás létrehozása 225
2. Jelölőnégyzet, rádiógomb, szövegdoboz, menü 232
3. Képmező, listamező, időzítés 239
4. Rajzolás 242

IX. KIEGÉSZÍTÉSEK, ÖSSZEFOGLALÁS 247
1. Alapok 247
2. A teknőcgrafika fontosabb utasításai 249
3. Változók 250
4. Vezérlési szerkezetek 253
5. Összetett változók, adatszerkezetek 258
6. Programozási tételek 262
7. Grafikus alkalmazások 266
8. Objektumorientált programozás 272

FÜGGELÉK 275

ELŐSZÓ

A könyvet elsősorban a közismereti, emelt szintű informatika érettségire készülőknek ajánlom, mivel az ismeretek, feladatok főleg ezekhez a követelményekhez igazodnak. A kiegészítő témákkal együtt (objektumorientált programozás alapjai, Windows alkalmazások készítése) azonban hasznos lehet informatikai szakképzésekhez, felsőfokú tanulmányok alapozó tárgyaihoz, vagy önképzéshez is.

Programozási nyelvnek a Pythont választottam. Egyszerűsített utasításkészlete, deklarációmentes változói, tömör, de logikus programépítése miatt a fiatalabb generációk kedvence, mégpedig teljes joggal. Előbb említett tulajdonságai miatt alkalmas első programnyelvként történő elsajátításra. Ugyanakkor a programnyelv teljes körű abban az értelemben, hogy alkalmas akár hálózati alkalmazások írására, vagy weblapgenerátorként (mint a PHP), stb.

A könyvet alapvetően egy lehetséges útnak szántam, amit az olvasó végigjárhat. Ehhez módszerként a mintafeladatokon keresztüli bemutatást választottam, lépésről lépésre haladva.

Aki most kezd el a programozással foglalkozni, kezdje az I. fejezettel, ami egy könnyed, rajzos bevezetés, pontos fogalmak nélküli ismerkedés a Pythonnal. Aki már tisztában van az olyan alapfogalmakkal, mint változó, ciklus, elágazás, átugorhatja ezt a fejezetet.

A II. fejezet egy rövid áttekintés. Milyen programnyelvek vannak? Milyen feladatai vannak egy szoftverfejlesztőnek? Mi az algoritmus? Stb.

A Python nyelvi elemeivel a III–V. fejezetek foglalkoznak, elsajátíthatóság szerinti sorrendben. Kivételt képez az objektumorientált programozás, ami a VII. fejezetben kapott helyet, hogy ne keveredjen a közismereti emelt szintű érettségire elsajátítandó témakörök közé.

A programozási tételeket (VI. fejezet) átírtam Python nyelvhez közelebb állóra. A listák indexelése 0-tól kezdődik, és kiegészítettem olyan változatokkal, amik nincsenek benne az algoritmusgyűjteményekben, de érettségin már előkerültek (pl. összetett rendezés). A VI. fejezet vége kifejezetten az érettségire összpontosít.

A grafikus alkalmazások fejlesztése (VIII. fejezet) jelenleg nem közismereti érettségi témakör, mégis hasznos, ha élvezhető, netán eladható alkalmazást szeretnénk készíteni. Igyekeztem a leggyakrabban előforduló problémákat érdekes feladatokon keresztül bemutatni.

A könyvet egy olyan áttekintő fejezet (IX. Kiegészítések, összefoglalás) zárja, amihez fordulhatunk emlékeztetőként akkor is, amikor már értjük az alapokat, de még nincs minden információ a fejünkben.

Terjedelmi okokból a könyv kevés teljes programot tartalmaz. A legfontosabbak a Függelékben megtalálhatók. A könyvhöz tartozó további programokat mellékletként lehet letölteni a kiadó honlapjáról:

https://galaktikabolt.hu/emelt-szintu-informatika-erettsegi-2-python-lepesrol-lepesre/

A könyv írása alatt a Python 3.8.0 változata volt a gépemen. A Python fejlesztői környezete Windows és Linux alapon is szerepel az érettségire választható szoftverek listáján, de a PyCharmot választók számára sem jelenthet gondot a könyv használata.

A Python tanulásához segítségemre volt Gérard Swinnen Tanuljunk meg programozni Python nyelven című, GNU Szabad Dokumentációs Licenc alá eső műve, ami pdf formátumban forog közkézen az interneten. 15 éves dokumentáció, ezért pár dolog már nem úgy működik, ahogyan ott olvasható.

A könyv megírása közben fontos segítségem volt a Python hivatalos, angol nyelvű webes dokumentációja, ami jelenleg itt található:http://docs.python.org, valamint egy magyar nyelvű webes leírás: http://nyelvek.inf.elte.hu/leirasok/Python.

A feladatokhoz elsősorban az Oktatási Hivatal honlapjáról letölthető érettségi feladatsorokat tanulmányoztam: https://www.oktatas.hu/kozneveles/erettsegi/feladatsorok/.

Jó utat kívánok a tanuláshoz minden olvasónak!

I. ISMERKEDÉS A PROGRAMOZÁSSAL

1. PARANCSOK

A számítógép parancsaink, utasításaink szerint cselekszik. Ezért a programozó parancsokat, utasításokat ad a számítógépnek, mikor programot ír. Kezdjük a programozással történő ismerkedést a parancsokkal.

A Python indításakor egy ablakot kapunk, ami egy parancsértelmező. Ahol villog a kurzor, oda írhatjuk be a parancsunkat, amit a Python végrehajt. Úgy adjuk a parancsértelmező tudtára, hogy befejeztük a parancs beírását, hogy megnyomjuk az **Enter** billentyűt.

1. mintafeladat – parancsok használata

1. A parancssor indításához kattintsunk a **Start** menüben a **P** betűnél található **Python x.x/(IDLE Python x.x 64-bit)** menüpontra. Megjelenik a parancsértelmező ablaka.
2. A **>>>** -tól jobbra, ahol a kurzor villog, írhatjuk be a parancsunkat.

Először ki kell adnunk egy parancsot, hogy elérhetőek legyenek a rajzoló utasítások.

3. Írjuk be a következő parancsot, majd nyomjuk meg az **Enter** billentyűt:

```
from turtle import *
```

Python 3.8.0 Shell

File Edit Shell Debug Options Window Help

```
Python 3.8.0 (tags/v3.8.0:fa919fd, Oct 14 2019, 19:37:50) [MSC v.1916 64 bit (AM
D64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> from turtle import *
>>> forwad(400)
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    forwad(400)
NameError: name 'forwad' is not defined
>>> forward(400)
>>> left(90)
>>> forward(100)
>>> |
```

Ln: 12 Col: 4

4. Ha nem látunk semmilyen piros szöveget a parancsértelmező ablakban, akkor jól írtuk be az utasítást. Az esetleges piros szöveget hibaüzenetnek hívják. Valószínűleg elgépeltünk valamit. Ilyenkor írjuk be újra a parancsot, majd nyomjuk meg az **Enter** billentyűt. (A képen szándékosan kihagytam egy r betűt a második parancsnál, hogy látható legyen a példa erre az esetre is.)

A rajzoláshoz képzeljünk egy teknőcöt, aminek a feje jelenleg jobbra néz. A teknőc a hasán hord egy ecsetet, ami most hozzáér a papírhoz.

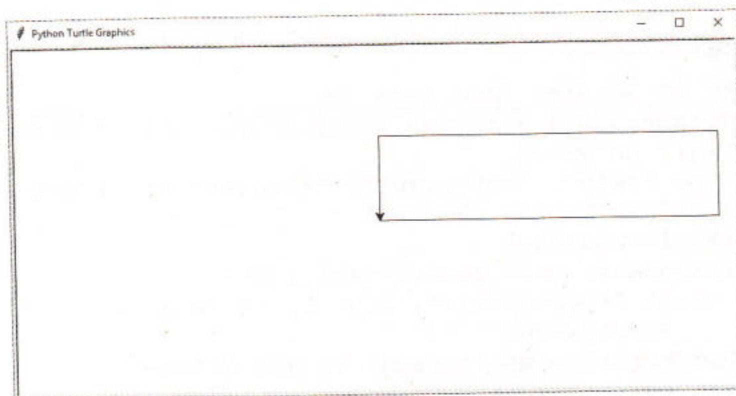
5. Húzzunk egy egyenes vonalat 400 képpont hosszúságban. Utasítsuk a teknőcöt, hogy menjen előre (a feje irányában) 400 képpontot. Ehhez írjuk be a következő parancsot, majd nyomjuk meg az **Enter** billentyűt:
```
forward(400)
```

Megjelenik egy grafikus ablak (*Python Turtle Graphics* a neve), és a középponttól indulóan ott lesz rajta az egyenes vonal. A nyíl jelképezi a teknőcöt. Amerre mutat, arra áll a teknőc feje. Szeretnénk felfelé folytatni a rajzolást.

6. Fordítsuk el a teknőcöt 90 fokkal balra. Ehhez adjuk ki (gépeljük be, majd üssünk **Entert**) a következő parancsot:
```
left(90)
```
7. Majd utasítsuk a teknőcöt, hogy ismét menjen előre, most elegendő lesz 100 egységgel. Ehhez adjuk ki a következő parancsot[1]:
```
forward(100)
```

A parancsok mögött, a zárójelek között található számokat (vagy betűket) paramétereknek nevezik. A paraméterek pontosítják a parancs tevékenységét. Például: a parancs lehet az, hogy menj előre, a pontosítás pedig az, hogy 100-zal.

Próbáljuk meg önállóan kiegészíteni a programot úgy, hogy egy teljes téglalapot kapjunk, aminek az alsó és jobb oldalát már megrajzoltuk az előbb. Csak akkor olvassuk tovább, ha készen vagyunk, vagy semmi ötletünk nincs. Íme, a kiadandó utasítások:




```
left(90)
forward(400)
left(90)
forward(100)
```

1 Nálam csak akkor vált láthatóvá az új vonal, amikor kicsit jobban széthúztam a grafikus ablakot.

2. PROGRAM

Jó lenne, ha meg tudnánk őrizni az előző, téglalapot rajzoló parancsokat, hogy a megfelelő sorrendben kiadva őket, lehetőleg nem egyesével, kényelmesen előhívhassuk egy fájlból.

2. mintafeladat – program készítése, futtatása

1. A Python Shell ablakban (ahol eddig dolgoztunk), kattintsunk a **File/New File** menüpontra.
2. Megjelenik egy új ablak. Nevezzük a továbbiakban *szerkesztőablaknak*. Itt gépeljük be, javítjuk, futtatjuk, mentjük, stb. a programunkat.
3. Jelenítsük meg a sorok számozását, mert így áttekinthetőbb lesz a munka. Ehhez kattintsunk az **Options/Show Line Numbers** menüpontra.
4. Gépeljük be a parancsokat pontosan a mellékelt ábra szerint látható módon. A sorok végén nyomjuk meg az **Enter** billentyűt.
5. Mentsük el egy fájlba. Kattintsunk a **File/Save As...** menüpontra. Megjelenik a Mentés másként ablak, ahol a Windows alkalmazásoknál megszokott módon kiválaszthatjuk a fájl helyét, és megadhatjuk a nevét. Legyen a **Fájl neve** most **feladatI_01**. (Végül a **Mentés** gombra kell kattintani, hogy a fájl ténylegesen a kiválasztott mappába kerüljön.)

```
1 from turtle import *
2 forward(400)
3 left(90)
4 forward(100)
5 left(90)
6 forward(400)
7 left(90)
8 forward(100)
```

Amit így elkészítettünk, az egy **program**. A program utasításokból áll. Eddig a parancs elnevezést használtuk, mert egyesével hajtottuk végre őket. Ha egy program részei, inkább az utasítás elnevezés a megszokott. A program **futása** a benne található utasítások automatikus egymás utáni végrehajtását jelenti, alapvetően a beírás sorrendjében.

6. Futtassuk a programot, azaz hajtsuk végre a benne található utasításokat. Ehhez nyomjuk meg az **F5** billentyűt.
7. A téglalapot a *Python Turtle Graphics* ablakban fogjuk látni.
8. Zárjuk be a *feladatI_01* ablakot az X-re kattintva a jobb felső sarokban.
9. Győződjünk meg róla, hogy a programunk nem veszett el, azaz nyissuk meg újra. Kattintsunk a **File/Open...** menüpontra. Megjelenik a Megnyitás ablak, ahol a Windows alkalmazásoknál megszokott módon kiválaszthatjuk a fájl helyét és nevét. Kattintsunk duplán a **feladatI_01** nevű fájlra.

10. Megnyílik a szerkesztőablak, és benne látható a programunk.

Változtassuk meg a programot úgy, hogy kitöltse a téglalap belsejét pirosra (red), és a határolóvonalak aranyszínűek (gold) legyenek.

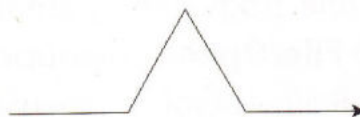
11. Gépeljük be a mellékelt programot sorról sorra, pontosan. Mindegyik sor végén **Entert** ütve.
12. A 2. sor **reset** utasításával letöröljük a grafikus ablakot, és alaphelyzetbe állítjuk a teknőcöt.
13. A **color** utasítással állítható be a vonal, határolóvonal színe, amit a teknőc ecsetje húz (3. sor).
14. A **fillcolor** utasítással az alakzat (most téglalap) belsejének kitöltő színét lehet beállítani (4. sor).
15. A **begin_fill** (5. sor) és **end_fill** (13. sor) utasítások között leírt alakzat belseje lesz kitöltve pirossal.

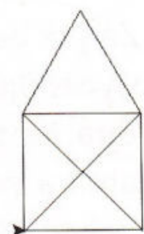
```
 1 from turtle import *
 2 reset()
 3 color("gold")
 4 fillcolor("red")
 5 begin_fill()
 6 forward(400)
 7 left(90)
 8 forward(100)
 9 left(90)
10 forward(400)
11 left(90)
12 forward(100)
13 end_fill()
```

16. Mentsük el a programot. Kattintsunk a **File/Save As...** menüpontra. Megjelenik a **Mentés másként** ablak, ahol a Windows alkalmazásoknál megszokott módon kiválaszthatjuk a fájl helyét, és megadhatjuk a nevét. Legyen a **Fájl neve** most **feladatl_02**.
17. Futtassuk a programot (**F5**). Az eredményt a *Python Turtle Graphics* ablakban fogjuk látni.

Feladatok

1. Rajzoljuk meg a bal oldalon látható ábrát. A szakaszok egyforma hosszúak, 100 képpontosak.
2. Rajzoljuk meg a középső ábrán látható rajzot. A szakaszok egyforma hosszúak, 100 képpontosak. A középső háromszög szabályos.
3. Rajzoljuk meg a mellékelt házikót egyetlen vonallal. Nem emelhetjük fel az ecsetet, és nem is mehetünk kétszer végig egy vonalon. A tető szabályos háromszög. A szakaszok egyforma hosszúak, 100 képpontosak. Aki nem tudná kiszámítani az átlók hosszát, az a lábjegyzetben megnézheti[2].





2 141 képpont.

3. ELJÁRÁS

Több, ugyanolyan méretű téglalapot szeretnénk rajzolni, de nem szeretnénk megismételni az összes utasítást, amik egy téglalapot rajzolnak. Több utasítást együtt lehet kezelni, ha egy eljárásba helyezzük őket. Az eljárásnak nevet adunk. Utána már elég ezt a nevet beírni, és az összes benne található utasítás a megadott sorrendben végrehajtódik.

3. mintafeladat - kód újra hasznosítása eljárással

Módosítsuk az előző programot (feladatl_02.py) úgy, hogy három téglalapot rajzoljon egymás alá piros, fehér és zöld színekben, így egy magyar zászlót kapjunk.

1. Kattintsunk a **File/Open...** menüpontra. Megjelenik a Megnyitás ablak, ahol a Windows alkalmazásoknál megszokott módon kiválaszthatjuk a fájl helyét és nevét. Kattintsunk duplán a **feladatl_02** nevű fájlra.
2. Egészítsük ki a programot a mellékelt ábrán látható módon.
3. A 3. sorban kezdődik az eljárás. A **def** utasítás vezeti be, amit az eljárás neve követ (teglalap), amit szabadon megválaszthatunk[3]. Kötelező a zárójelpár, majd utána a kettőspont.
4. Ne lepődjünk meg, ha a 3. sor végén **Entert** ütve, a következő sor négy karakterrel beljebb kezdődik. A beljebb írt utasítások tartoznak az eljáráshoz (4–12. sor). Ezek rajzolják meg a téglalapot. (Ha valamiért nem állna beljebb a kurzor a továbbiakban, nyomjuk meg a **Tab** billentyűt, az pontosan 4 szóközt fog lerakni. Természetesen megnyomhatjuk négyszer a **Szóköz** billentyűt is helyette.)
5. Az eljáráson kívüli részt főprogramnak nevezik (14–29. sor).

```
 1 from turtle import *
 2
 3 def teglalap():
 4     begin_fill()
 5     forward(400)
 6     left(90)
 7     forward(100)
 8     left(90)
 9     forward(400)
10     left(90)
11     forward(100)
12     end_fill()
13
14 reset()
15 color("gold")
16 fillcolor("red")
17 teglalap()
18 up()
19 forward(100)
20 left(90)
21 down()
22 fillcolor("white")
23 teglalap()
24 up()
25 forward(100)
26 left(90)
27 down()
28 fillcolor("green")
29 teglalap()
```

3 Bizonyos korlátozásokkal. Például nem egyezhet meg a név a Python valamelyik utasításával.

6. Az eljárást három helyen hívjuk meg (17., 23., 29. sorok). A meghívás azt jelenti, hogy az utasítások végrehajtása az eljárásban lévőkkel folytatódik. Tehát például a 16. sor után a 17., majd a 3–12. sor, és csak utána a 18. sor.
7. Mindegyik eljárás hívás előtt beállítjuk a kitöltés színét rendre pirosra (16. sor), fehérre (22. sor), végül zöldre (28. sor).
8. Mikor az első téglalap elkészült, a bal alsó sarkában állunk. Mivel a következő téglalap is a saját bal alsó sarkából indul, lejjebb kell mennünk 100 képponttal (19., 25. sor).
9. Ehhez felemeljük az ecsetet, hogy ne rajzoljon a teknőc megfelelő helyzetbe mozgatása közben, az **up** utasítással (18., 24. sor)[4].
10. Ha a teknőc a megfelelő helyzetben van, lerakjuk az ecsetet a **down** utasítással, hogy tudjon rajzolni (21. és 27. sor).
11. A téglalap rajzolása lefelé álló teknőccel végződik, viszont jobbra mutatóval kezdődik. Ezért minden új téglalap rajzolása előtt el kell fordítani balra 90 fokkal (20., 26. sor).
12. Mentsük el a programot. Kattintsunk a **File/Save As...** menüpontra. Legyen a **Fájl neve** most **feladatl_03**.
13. Futtassuk a programot (**F5**). Az eredményt a *Python Turtle Graphics* ablakban fogjuk látni.

Feladatok

1. Változtassuk meg a *feladatl_03* programot úgy, hogy a német zászlót jelenítse meg (fekete-piros-sárga).
2. Változtassuk meg a *feladatl_03* programot úgy, hogy kétszínű, vízszintesen csíkos zászlókat tudjon megjeleníteni. A teljes zászló mérete maradjon meg. Tesztelhetjük például a lengyel zászlón (fehér-piros).
3. Változtassuk meg a *feladatl_03* programot úgy, hogy háromszínű, függőlegesen csíkos zászlókat tudjon megjeleníteni. A teljes zászló mérete maradjon meg. Tesztelhetjük például az ír zászlón (zöld-fehér-narancs).

---

4 Nem kellett volna felemelni az ecsetet, mert ugyanazon a vonalon haladt volna végig, amit utána úgyis megrajzolunk, de nem árt megismerkedni az **up** és **down** utasításokkal, mert más feladatok megkövetelhetik a használatát.

4. VÁLTOZÓK

Jó lenne egy olyan általános eljárást írni, ami minden vízszintesen háromcsíkos zászlót meg tud jeleníteni anélkül, hogy mindegyik esetben módosítani kellene a kódot. Az eljárásokhoz készíthetünk paramétereket. (Ha az utasításoknak van, miért ne lehetne az eljárásoknak is.) A paraméterekben megadhatnánk, milyen színűek legyenek a csíkok. Már csak az a kérdés, hogyan jut el az eljárás paraméterétől az utasításig a szín megnevezése.

Ehhez a változók lesznek a segítségünkre. **A változó átmeneti tárolóhely, amire a nevével hivatkozhatunk**. Elképzelhetjük úgy, mint egy felcímkézett fiókot. Amit odateszünk, az ott marad. Akárhányszor megnézhetjük. Módosíthatjuk is, de akkor legközelebb a módosított értéket látjuk ott. A tárolóhely a Pythonban az első használatkor automatikusan létrejön. (Lefoglaljuk és felcímkézzük a fiókot.) Egészen addig elérhető, míg fut a program.

4. mintafeladat – változók

Készítsünk egy *trikolor* nevű eljárást, ami három paraméterként megkapja a kirajzolandó zászló csíkjainak színét, és megjeleníti az ennek megfelelő zászlót. (Folytassuk az előző feladatot – feladatI_03.py)

1. Nyissuk meg a **feladatI_03** nevű fájlt.
2. Egészítsük ki a programot a mellékelt ábrán látható módon. Az 1–13. sorokban nincs változás.
3. Az előző program 16–29. soraiból készítjük el a *trikolor* eljárást, kiegészítve még egy sorral, ami a végén a teknőcöt a kiindulási irányba állítja (29. sor, ebben a programban).

```
14 def trikolor(f, k, a):
15     fillcolor(f)
16     teglalap()
17     up()
18     forward(100)
19     left(90)
20     down()
21     fillcolor(k)
22     teglalap()
23     up()
24     forward(100)
25     left(90)
26     down()
27     fillcolor(a)
28     teglalap()
29     left(90)

31 reset()
32 color("gold")
33 trikolor("red","yellow","black")
34 up()
35 goto(-420,0)
36 down()
37 trikolor("red","white","green")
```

4. A trikolor eljárásnév utáni zárójelben található a három változó: *f, k, a*. Az *f*-be helyettesítődik a felső téglalap színének neve (14. sor), amikor meghívjuk az eljárást (például a 33. sorból).

5. Amikor elérjük a kitöltő színt megadó **fillcolor** utasítást a felső téglalap megrajzolása előtt (15. sor), paraméterként az f-et adjuk meg. Így az f-ben tárolt szín nevével hajtjuk végre a **fillcolor** utasítást.
6. A középső téglalap színének neve az eljárás második paramétereként a *k* változóba kerül (14. sor). Majd a szín megadásakor a **fillcolor** paraméterébe helyettesítődik (21. sor). Ugyanez a helyzet az alsó téglalap színével is (14., 27. sor).
7. A főprogramból kétszer hívjuk meg a *trikolor* eljárást, tehát két zászlót rajzolunk (33. és 37. sor).
8. Célszerű, hogy a két zászló ne fedje egymást. Ezért a második zászló rajzolását a -420, 0 koordinátájú pontokból kezdjük. A teknőc a **goto** utasítással kerül ebbe a pontba (35. sor).
9. Mentsük a fájlt **feladatl_04** néven, és futtassuk a programot.

A változóknak több típusa van. A szöveges típusnál a *tartalmat* (tehát *nem* a változó nevét), idézőjelpár vagy aposztrófok közé kell írni. Például: "gold" vagy 'gold'. Számoknál semmilyen jelre nincs szükség.

Gondoljuk meg, hány soros lenne a programunk eljárások nélkül. Még egyszer kellettek volna a 15–29. sorok. Ez eddig +15 sor. A téglalap rajzoló még ötször (összesen hat téglalap van a két zászlóban). Ez újabb 5-ször 9, azaz 45 sor lett volna. Igaz, nem kellett volna létrehozni és meghívni az eljárásokat, ami miatt 7 sorral kevesebb lett volna. Összesen 60 – 7 = 53 sort takarítottunk meg azzal, hogy eljárásokat használtunk a feladatban. Nem is beszélve az áttekinthetőségről, ami előnyös, ha a programot módosítani akarjuk.

Feladatok

1. Bővítsük a *feladatl_04* programot egy *dikolor* nevű eljárással, ami kétszínű, vízszintesen csíkozott zászlókat képes megjeleníteni. A zászlók mérete egyezzen meg a háromcsíkosakéval. Az eljárásnak két paramétere lesz a felső és az alsó téglalap színe. Tesztelhetjük például a lengyel zászlón (fehér-piros). Felhasználhatók előző feladatok kódrészletei.
2. Bővítsük az *előző (gyakl_04_1)* feladatot egy *ftrikolor* nevű eljárással, ami háromszínű, függőlegesen csíkozott zászlókat képes megjeleníteni. A zászlók mérete egyezzen meg a vízszintesen három csíkosakéval. Az eljárásnak három paramétere lesz a bal, a középső és a jobb téglalap színe. Tesztelhetjük például az ír zászlón (zöld-fehér-narancs). Felhasználhatók előző feladatok kódrészletei.

5. ELÁGAZÁS

Jó lenne egy olyan eljárás, aminek elég megadni az ország nevét paraméterként, és kirajzolja a zászlaját, ha az vízszintes trikolor zászló.

Megkap az eljárás egy paramétert, és a kapott ország neve alapján az eljárást többféleképpen kell folytatni. Mást kell tennie a programnak a különböző esetekben. Erre a célra szolgál az **elágazás**, ahol feltételtől függően más és más utasítások hajthatók végre.

5. mintafeladat – elágazás

Írjunk egy *zaszlo* nevű eljárást, aminek paramétere egy olyan ország neve, aminek a zászlaja vízszintes trikolor. Egyelőre dolgozzunk csak három nemzettel: magyar: piros, fehér, zöld; osztrák: piros, fehér, piros; német: fekete, piros, sárga; holland: piros, fehér, kék. Ha nem a felsorolt országok valamelyikét adják meg, jelenjen meg egy fehér, fehér, fehér zászló. Ez fogja jelezni a helytelen megadást.

```
31 def zaszlo(nemzet):
32     if nemzet == "magyar":
33         trikolor("red","white","green")
34     elif nemzet == "osztrak":
35         trikolor("red","white","red")
36     elif nemzet == "nemet":
37         trikolor("black","red","yellow")
38     elif nemzet == "holland":
39         trikolor("red","white","blue")
40     else:
41         trikolor("white","white","white")
```

```
42
43 reset()
44 color("gold")
45 zaszlo("magyar")
46 up()
47 goto(-420,0)
48 down()
49 zaszlo("nemet")
```

1. Nyissuk meg a **feladatl_04** nevű fájlt.
2. Egészítsük ki a programot a mellékelt ábrán látható módon. Az 1–30. sorokban nincs változás.
3. A 31. sorban létrehozzuk a *zaszlo* nevű eljárást, aminek egy paramétere van, a *nemzet* nevű változó. (A változó neve nem feltétlenül egy betű, mint az előző példában.)
4. A 32. sorban kezdődik az elágazás az **if** (ha) utasítással. Mögötte megadjuk a feltételt. A == azt jelenti, hogy *megegyezik-e* (azaz *egyenlő-e*). Az egész sor

649403
...GHY KÖNYV...

"lefordítva": *Ha a nemzet* változóban *magyar* van[5]. A sort kettősponttal kell zárni, mint az eljárásoknál.

5. A következő sortól (33.) kezdődően lehet megadni, mi történjen, ha a *nemzet* változóban magyar van. Meghívjuk a trikolor eljárást a magyar zászlónak megfelelő paraméterekkel. Az utasításnak beljebb kell kezdődnie 4 karakterrel. Innen tudja a Python, hogy azt csak a feltétel teljesülése esetén kell végrehajtani. A beljebb kezdett utasítások végrehajtása után a program átugorja az elágazás többi sorát, és utána folytatja a program végrehajtását. (Esetünkben az elágazás vége egyben az eljárás vége is, tehát a főprogram következő sorával folytatódik az utasítások végrehajtása.)
6. Ha nem magyar volt a *nemzet* változóban, a program futása átugorja a 33. sort, és a 34.-kel folytatja. Itt az **elif** utasítás mögött újabb feltétel adható meg. Ha a *nemzet* változóban *osztrák* van megadva, akkor a 35. sorban található módon meghívjuk a *trikolor* eljárást, az osztrák zászlónak megfelelő színeket rakva a paraméterekbe.
7. Ezt így folytatjuk a német (36–37. sor) és holland (38–39. sor) zászlókkal.
8. Az elágazás az **else** (különben) utasítással zárul (40. sor). Itt már nincs feltétel, hiszen az összes eddig felsorolttól különböző eset történését tudjuk itt megadni. Ez jelen esetben a fehér, fehér, fehér zászló megrajzolása (41. sor).
9. A főprogramot is módosítani kell. A *trikolor* utasításokat cseréljük *zaszlo*-ra, paraméterként a nemzet nevét adva meg (45. és 49. sor).
10. Mentsük a fájlt **feladatl_05** néven, és futtassuk a programot.

Annyi **elif** sort használhatunk, amennyire szükség van. Ha csak egy feltételünk van, egy sem kell belőle, így az **if** után **else** következik. Az **elif** az else+if rövidítése, így jelentése *"különben ha"*. Az **else** használata sem kötelező.

A feltételek megadása során használhatjuk a <, >, <=, >= jeleket a megszokott módon. A **nem egyenlő** jele a Pythonban a **!=** .

Feladatok

1. Egészítsük ki úgy a *feladatl_05* programot, hogy felismerje a jemeni (piros-fehér-fekete) és az orosz (fehér-kék-piros) zászlót is.
2. Egészítsük ki úgy az előző *(gyakl_05_1)* programot, hogy felismerje a lengyel (fehér-piros), a monacói (piros-fehér), a nigériai (függőlegesen zöld-fehér-zöld) és az olasz (függőlegesen zöld-fehér-piros) zászlókat is.

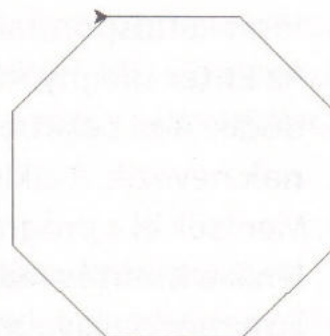
---

5 Vegyük észre az = és a == közötti különbséget. Az = jel az értékadást jelenti. A bal oldalon változó értéke a továbbiakban a jobb oldalon álló kifejezéssel egyenlő. Például b = 2 azt jelenti, hogy a b változó értéke a továbbiakban 2. Ezzel szemben a == az összehasonlítás jele. Például b == 2 azt jelenti, b értéke egyenlő-e 2-vel. Ettől a b értéke nem változik meg 2-re.

6. SZÁMLÁLÓ CIKLUS

6. mintafeladat – programrészlet ismétlése ciklussal

Írjunk programot, ami egy szabályos nyolcszöget rajzol, aminek az oldalai 100 képpont hosszúságúak.



1. Kezdjük a kísérletezést a *parancssorban*, tehát hívjuk elő a parancsértelmező ablakát. A >>> -tól jobbra, ahol a kurzor villog, írhatjuk be a parancsunkat. Ne felejtsük minden parancs után megnyomni az **Enter** billentyűt.
2. Először is töröljük le a grafikus ablakot, és állítsuk alaphelyzetbe a teknőcöt a következő utasítással:
   ```
   reset()
   ```
3 Rajzoljunk egy 100 képpont hosszúságú vonalat. Ehhez adjuk ki a következő parancsot:
   ```
   forward(100)
   ```
4. El kell fordulni 45 fokkal, hogy majd elkezdhessük rajzolni a következő oldalt. Tanuljunk meg most jobbra fordulni, amihez a következő parancsot használjuk:
   ```
   right(45)
   ```
5. Ismét rajzoljunk egy 100 képpont hosszúságú vonalat:
   ```
   forward(100)
   ```
6. Utána ismét el kell fordulni 45 fokot jobbra:
   ```
   right(45)
   ```

És jól láthatóan ez így folytatódna még hatszor. Szerencsés lenne, ha nem kellene beírni nyolcszor a két utasítást, hanem arra utasítani a parancsértelmezőt, hogy *ismételje* meg a két utasítást nyolcszor. **Ismételt utasítás végrehajtásra** a programozásban a **ciklusokat** használjuk. A ciklusokon belül a **számláló ciklusnak** előre megadható, *hányszor ismételjen*.
Ehhez érdemes lesz már programot írni.

7. Nyissunk egy új programot. A Python Shell ablakban kattintsunk a **File/New File** menüpontra.

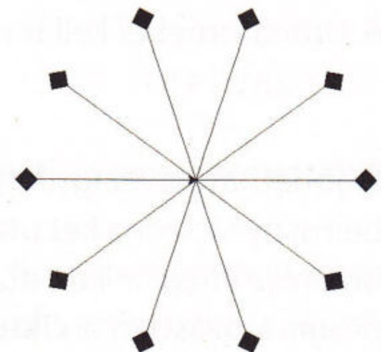
8. Gépeljük be a parancsokat pontosan a mellékelt ábra szerint látható módon. A sorok végén nyomjuk meg az **Enter** billentyűt.

```
1 from turtle import *
2 reset()
3 for i in range(8):
4     forward(100)
5     right(45)
```

9. A 3. sorban található a számláló ciklus. A **for** utasítás vezeti be. Az i-t **ciklusváltozónak**[6] hívják, és az értéke 0-tól 7-ig változik esetünkben, minden egyes ismétléskor eggyel növekedve. (Ebben a programban a ciklusváltozót nem használjuk fel a továbbiakban, de attól még itt meg kell adni.) Az ismétlések számát a **range** utasítás mögötti paraméter tartalmazza (8). A számláló ciklus utasítását kettősponttal zárjuk.
10. Az **Enter** megnyomása után a következő sor négy karakterrel beljebb kezdődik. Ami beljebb van, azt ismétli a ciklus. Az ismétlendő részt **ciklusmagnak** nevezik. A ciklusmagban most a **forward** és **right** utasítások vannak.
11. Mentsük el a programot. Kattintsunk a **File/Save As...** menüpontra. Megjelenik a **Mentés másként** ablak, ahol a Windows alkalmazásoknál megszokott módon kiválaszthatjuk a fájl helyét, és megadhatjuk a nevét. Legyen a **Fájl neve** most **feladatl_06**.
12. Futtassuk a programot (**F5**). A nyolcszöget a *Python Turtle Graphics* ablakban fogjuk látni.

Feladatok

1. Változtassuk meg úgy az előző *(feladatl_06)* programot, hogy szabályos hatszöget rajzoljon.
2. Rajzoljunk csempézett falat 20 képpont oldalú négyzetekből. Legyen 25 oszlop és 12 sor. Minél kevesebb utasításból oldjuk meg a feladatot, azaz ahol lehet, használjunk ciklust. Ha ötletre van szükség, a lábjegyzetben olvasható[7].
3. Rajzoljuk meg az ábrán látható csillagot. A sugárirányú vonalak hossza 200 egység, a végükön található négyzetek oldalhossza 20 egység.



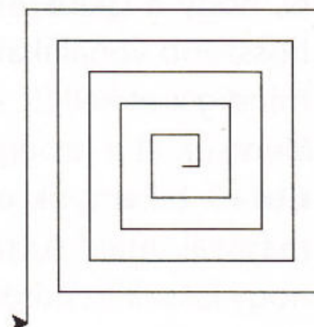
6 Bármilyen más nevet is választhatunk ciklusváltozónak. Az i, j, k betűket szokták többnyire használni ciklusokhoz.

7 Egy ciklus, ami négyzetet rajzolja. Egy másik ciklus egy sornyi négyzetet rajzol. A harmadik ciklus pedig megismétli a sorokat.

7. VÁLTOZÓK HASZNÁLATA

7. mintafeladat – ismerkedés a változók használatával

1. Kezdjük a kísérletezést a *parancssorban,* tehát hívjuk elő a parancsértelmező ablakát. A **>>>** -tól jobbra, ahol a kurzor villog, írhatjuk be a parancsunkat. Ne felejtsük minden parancs után megnyomni az **Enter** billentyűt.
2. A változóknak lehet értéket adni *x = 10* formában. Ilyenkor az x változóba 10 kerül. Úgy szoktuk kiolvasni: *x legyen egyenlő 10-zel.* Mindig a bal oldalt lehet egyenlővé tenni a jobb oldallal, tehát a bal oldalra kell kerülnie, ami változni fog. Írjuk be:
   ```
   x = 10
   ```
3. Mivel nincs hibaüzenet, a parancsértelmező elfogadta az értékadást.
4. A változó értékét kiírathatjuk a parancssorban, ha beírjuk a nevét. Tegyük meg:
   ```
   x
   ```
5. Megjelenik alatta egy kék számmal kiírva a változó értéke, a 10.
6. Növeljük meg az x értékét 20-szal. Ehhez a következő parancs szükséges:
   ```
   x += 20
   ```
7. Ellenőrizzük, tényleg növekedett-e x értéke. Írassuk ki, mennyi most:
   ```
   x
   ```
8. Megjelenik a 30 kékkel kiírva. Eddig rendben vagyunk.
9. Csökkentsük x értékét 15-tel. Ehhez a következő parancs szükséges:
   ```
   x -= 15
   ```
10. Ellenőrizzük, tényleg csökkent-e x értéke. Írassuk ki, most mennyi:
    ```
    x
    ```
11. Megjelenik a 15 kékkel kiírva, tehát tényleg lehet így csökkenteni az x értékét.
12. Végül számoljuk ki, mennyi lesz az x változó 10-szerese, ha még 10-et hozzá is adunk. Tegyük úgy, hogy az x értékét nem változtatjuk meg. Ehhez a következő parancsot kell kiadni:
    ```
    10 * x + 10
    ```
13. Meg is jelenik a következő sorban a helyes végeredmény, a 160.

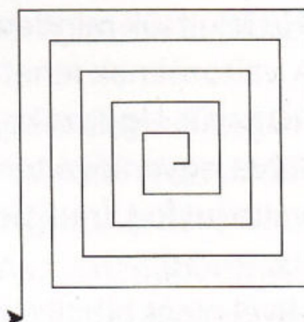


Ha szeretnénk megrajzolni a mellékelt ábrát, hamar észrevehetjük, hogy a rövid megoldáshoz ciklust kell alkalmazni, ami mindig azt fogja ismételni, hogy menj előre, majd fordulj balra

(ha belülről indulunk). Az is biztos, hogy a ciklusnak húszszor kell lefutnia, mivel 20 szakaszból áll össze a rajz. Az elfordulással nincs is baj, hiszen az mindig 90 fokos, azonban mindig növekszik a hosszúság, amivel előre kell menni. Ennek megoldásához kicsit többet kell tudni a változókról, mint eddig. Három megoldását is megnézzük a feladatnak, mert mindegyikből lehet tanulni.

8. mintafeladat – változók használata programban

Készítsünk programot, ami megrajzolja a mellékelt alakzatot.



Első változat – értékadás, növekmény

1. Nyissunk egy új programot (**File/New File**), majd mentsük el (**File/Save As...**) rögtön **feladatl_07_1** néven.
2. Az x változóba kezdetben 10 kerül (3. sor). Enynyivel fog majd először előremenni a teknőc.
3. Utána létrehozzuk a **for** ciklust az ismétléshez az előző példában már megismert módon (4. sor). A **range** zárójelei között kell szerepelnie az ismétlések számának, ami most 20.
4. A ciklus magja 4 karakterrel beljebb kezdődik, ezt fogja ismételni a ciklus. Most három sorból áll: előremegy x egységet (először 10-et) a **forward** utasítás miatt (5. sor), majd elfordul balra 90 fokot a **left** utasítás által (6. sor).
5. Az *x+=10* annyit jelent, hogy az x változó értékét 10-zel növeljük (7. sor). A ciklus minden egyes lefutásának végén az x 10-zel nagyobb lesz: 10, 20, 30, 40... 200. Így érjük el, hogy a ciklus minden lefutásakor egyre hosszabb vonalakat húz, hiszen előremenni mindig x-et kell (5. sor).
6. Mentsük el a programot az eredeti nevén a **Ctrl+S** billentyűk egyszerre történő lenyomásával, majd futtassuk a programot (**F5**), hogy lássuk az eredményt.

```
1 from turtle import *
2
3 x = 10
4 for i in range(20):
5     forward(x)
6     left(90)
7     x+=10
```

Második változat – ciklusváltozó transzformációja

1. Nyissunk egy új programot (**File/New File**), majd mentsük el rögtön **feladatI_07_2** néven (**File/Save As...**).
2. Kihasználhatjuk, hogy a ciklusváltozóban, tehát az i-ben a számok 0-tól 19-ig változnak. Minden lefutáskor eggyel növekedve. i = 0, 1, 2, ..., 19. Ebből fogjuk előállítani, mennyit kell előremenni a **forward** utasítással (4. sor).

```
1 from turtle import *
2
3 for i in range(20):
4     forward(10*i+10)
5     left(90)
```

3. Ha az előző megoldáshoz hasonlóan az előre lépést mindig 10-zel akarjuk növelni, akkor a ciklusváltozónak a 10-szeresét kell venni: 0, 10, 20, ..., 190.
4. Akkor viszont először nem rajzolna semmit, hiszen 10 * 0 = 0. Ha hozzáadunk még 10-et, a probléma megoldódik: 10, 20, 30, ..., 200. Így lesz végül 10 * i + 10, amit előre kell lépni, ez fog szerepelni a **forward** utasítás paraméterében.
5. Mentsük el a programot az eredeti nevén a **Ctrl+S** billentyűk egyszerre történő lenyomásával, majd futtassuk a programot (**F5**), hogy lássuk az eredményt.

Harmadik változat – ciklusváltozó részletes beállítása

1. Nyissunk egy új programot (**File/New File**), majd mentsük el rögtön **feladatI_07_3** néven (**File/Save As...**).
2. A ciklusváltozó (a példában i) nem csak egyesével változtatható. Beállítható a kezdőértéke és a növekménye is a **range** zárójelei közt (3. sor).
3. Ha három paramétert adunk meg, akkor első a kezdőérték (10), a második az az érték, amit a ciklusváltozó már nem érhet el (210), a harmadik pedig a növekmény (10), amivel a ciklus minden egyes lefutásakor növekszik a ciklusváltozó.
4. Így esetünkben 10-zel indulunk, 200-ig megyünk (210 már nem lehet), és 10-esével. Tehát i: 10, 20, 30, ..., 200. Ez mindig pont annyi, amennyivel előre kell menni, így a **forward** utasítás paramétereként az i közvetlenül megadható (4. sor).

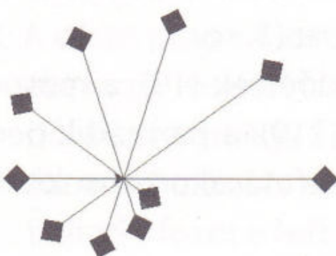
```
1 from turtle import *
2
3 for i in range(10,210,10)
4     forward(i)
5     left(90)
```

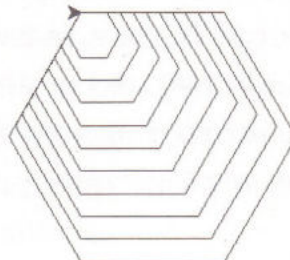
5. Mentsük el a programot az eredeti nevén a **Ctrl+S** billentyűk egyszerre történő lenyomásával, majd futtassuk a programot (**F5**), hogy lássuk az eredményt.

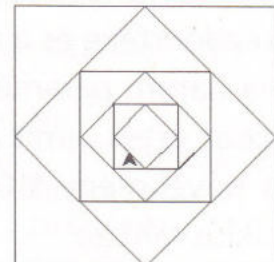
A **for** ciklusban a **range** megadható egy-, két- vagy három paraméterrel. Ha egy paramétert adunk meg, akkor az az lesz, amit nem szabad elérnie a ciklusváltozónak. Ilyenkor a számlálás 0-tól indul, és 1-esével történik. Ha kettőt adunk meg, akkor az első a kezdőérték, a második amit már nem szabad elérni a ciklusváltozónak. Ilyenkor a számlálás 1-esével történik. Ha hármat adunk meg, akkor az előző kettő után még a növekmény is megadható. A **for** ciklussal lehet visszafelé is számolni, akkor a növekmény negatív, a kezdőérték pedig nagyobb, mint ameddig számolnia kell. Például a range (200, 0, -10) visszaszámol 200-tól 10-ig 10-esével.

Feladatok

1. Módosítsuk a szögletes csigavonalat rajzoló programot *(feladatl_07_1)* úgy, hogy jobbra csavarodjon belülről kifelé.
2. Rajzoljuk meg a bal oldali ábrán látható alakzatot, minél kevesebb utasítást használva.
3. Rajzoljuk meg a középső ábrán látható alakzatot, minél kevesebb utasítást használva.
4. Rajzoljuk meg a jobb oldali ábrán látható alakzatot, minél kevesebb utasítást használva. Ha nem tudjuk kiszámítani az átlós vonalak hosszát, a lábjegyzetben van segítség[8].







8 Egy négyzet átlóját megkaphatjuk, ha oldalának hosszát megszorozzuk 1,41-dal.

8. VÉLETLENSZÁM

A számítógép alkalmas rá, hogy hasonlóan viselkedjen, mint egy dobókocka, vagy egy feldobott érme, amik véletlenszerűen mutatnak 1...6 számokat, vagy fejet, illetve írást. Megfelelő utasítással a Python is képes véletlenszámokat létrehozni, amiket aztán közvetlenül felhasználhatunk, vagy megfeleltethetünk valamilyen más kijelzési módnak, például színeknek vagy vonalhosszaknak.

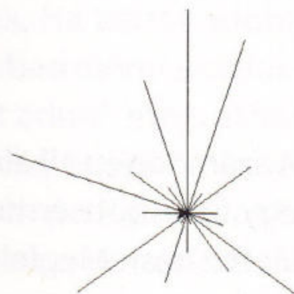
9. mintafeladat – ismerkedés a véletlenszámokkal

1. Kezdjük a kísérletezést a *parancssorban*, tehát hívjuk elő a parancsértelmező ablakát. A >>> -tól jobbra, ahol a kurzor villog, írhatjuk be a parancsunkat. Ne felejtsük minden parancs után megnyomni az **Enter** billentyűt.
2. A Pythonban véletlen egész számokat a **randrange** paranccsal készíthetünk. Ez a parancs sem része az alap utasításkészletnek (akárcsak a teknőc mozgatás), ezért be kell tölteni. Ehhez írjuk be:
   ```
   from random import randrange
   ```
3. A randrange parancsot használhatjuk egy paraméter megadásával. Azt fogja a paraméter megadni, hányféle értéket vehessen fel a véletlenszám, 0-tól kezdődően. Ha például dobókockát akarunk utánozni, 6-féle érték közül szeretnénk egyet kapni, tehát ezt kell beírni:
   ```
   randrange(6)
   ```
4. Kapunk egy számot 0 és 5 között. Írjuk be még párszor, mindegyik után **Entert** nyomva. Akár minden alkalommal eltérő számok jöhetnek, de mindig 0 és 5 között lesznek, ami tényleg 6-féle.
5. Ha a *randrange(6)*-hoz hozzáadunk 1-et, pont a dobókockának megfelelő számokat kapjuk. 0...5 +1 → 1...6. Írjuk is be:
   ```
   randrange(6) + 1
   ```
6. Írjuk be még párszor, mindegyik után **Entert** nyomva, hogy meggyőződjünk a helyességéről.
7. Ugyanerre az eredményre jutunk, ha kétparaméteres formában megadjuk a létrehozandó szám tartományát (intervallumát). Ilyenkor az első paraméter a legkisebb lehetséges érték, a második paraméter a legkisebb már nem lehetséges érték[9]. Írjuk be, és próbáljuk ki néhányszor:
   ```
   randrange(1,7)
   ```

---

9 A randrange is ugyanolyan módokon paraméterezhető, mint a range, tehát van egy-, két- és háromparaméteres lehetőség. Lásd 7. témakör.

10. mintafeladat – véletlenszámok felhasználása programban



Készítsük el az ábrán látható csillagszerű alakzatot, ha tudjuk, hogy ágai véletlenszerűen 0...219 képpont hosszúak lehetnek, és 20 db ága van.

1. Nyissunk egy új programot (**File/New File**), majd mentsük el rögtön **feladatl_08_1** néven (**File/Save As...**).
2. A Pythonban véletlen egész számokat a **randrange** utasítással készíthetünk. Ez az utasítás sem része az alap utasításkészletnek (akárcsak a teknőc mozgatás), ezért be kell tölteni a 2. sorban látható módon.
3. Mivel húsz ágat kell rajzolni, **for** ciklust fogunk használni, ami 20-szor fog lefutni (4. sor).
4. A ciklusmag, amit a ciklus ismétel, most négy sorból áll, amik beljebb kezdődnek (5–8. sor).
5. Az x változóba fogjuk tölteni a véletlenszámot (5. sor). A **randrange** utasítás első paramétere megadja a legkisebb lehetséges véletlenszámot (0), a második paramétere pedig azt a legkisebb számot, amit már nem állíthat elő (200). Úgy is fogalmazhatunk, hogy egy intervallumot adunk meg, ahonnan a véletlenszámok kikerülhetnek. Az alsó határt felvehetik, a felső határt már nem. (Alulról zárt, felülről nyílt intervallum.)

```
1 from turtle import *
2 from random import randrange
3
4 for i in range(20):
5     x = randrange(0,200)
6     forward(x)
7     backward(x)
8     left(18)
```

6. Az x-ben található véletlen értékkel előrevisszük a teknőcöt (**forward**), ezzel húzunk egy vonalat.
7. Mivel a következő ág megint a középpontból indul, vissza kell vinni oda a teknőcöt. Erre a célra szolgál a hátrálás parancsa, a **backward**.
8. Mivel 20 ág lesz, és a teljes kör 360 fok, így két ág között 360 fok : 20 = 18 fok lesz. Ennyivel fordítjuk el a teknőcöt balra a **left** utasítással (8. sor).
9. Mentsük el a programot az eredeti nevén a **Ctrl+S** billentyűk egyszerre történő lenyomásával, majd futtassuk a programot (**F5**), hogy lássuk az eredményt.

Feladatok

1. Írjunk programot, ami mindig 25 egységet lép előre, de minden lépés után a teknőc véletlenszerűen elfordul balra 90 fokot, vagy jobbra 90 fokot, vagy 180 fokot, de az is megtörténhet, hogy ugyanarra néz továbbra is, mint előzőleg.
2. Helyezzünk el 50 darab kicsi csillagokat véletlenszerűen a képernyőn. A csillagok sugarai 5 képpontosak, és 10 darabból áll össze egy csillag. A képernyőterület, ahová rajzolunk, vízszintesen -300-tól +300-ig, függőlegesen -200-tól +200-ig terjedhet. A háttérszínt a bgcolor("black") utasítással változtathatjuk feketére.

9. LISTA

A listák sorszámozott változókat tartalmaznak. Egyszerűbbé teszik a programot, ha sok változót kell használni.

11. mintafeladat – ismerkedés a listákkal

1. Kezdjük a kísérletezést a *parancssorban,* tehát hívjuk elő a parancsértelmező ablakát. A >>> -tól jobbra, ahol a kurzor villog, írhatjuk be a parancsunkat. Ne felejtsük minden parancs után megnyomni az **Enter** billentyűt.
2. A lista is egy változó, tehát nevet adunk neki (szin). Ha az értékeit is meg akarjuk adni, = jelet írunk mögé (legyen egyenlő), és az egyenlőségjel után szögletes zárójelek között felsoroljuk az elemeit, egymástól vesszővel elválasztva. Ha az elemek szövegek, idézőjelek közé kell őket helyezni. Adjunk meg egy listát három elemmel:
   ```
   szin = ["green", "red", "blue"]
   ```
3. Mivel nincs hibaüzenet, a parancsértelmező elfogadta az értékadást.
4. A teljes listát kiírathatjuk a parancssorban, ha beírjuk a nevét. Tegyük meg:
   ```
   szin
   ```
5. Megjelenik a szögletes zárójelek közötti rész a zárójelekkel együtt.
6. Az egyes elemekre a sorszámukkal hivatkozhatunk. A sorszámot indexnek is hívjuk. Az első elem sorszáma a 0. Hogy ne legyen ebből keveredés, gyakran 0. elemnek nevezik. A 0. elemre úgy hivatkozunk, hogy beírjuk a lista nevét, majd szögletes zárójelek között a sorszámát (indexét). Írjuk be tehát:
   ```
   szin[0]
   ```

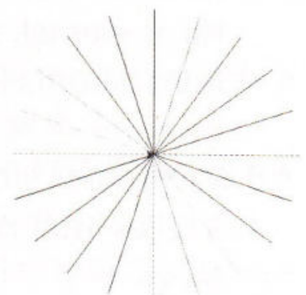
7. Ki is írja a parancsértelmező, hogy 'green'. Próbáljuk most az 1. elemet:
   ```
   szin[1]
   ```
8. Kiírja a parancsértelmező, hogy 'red'. Próbáljuk a 2. elemet:
   ```
   szin[2]
   ```
9. Kiírja a parancsértelmező, hogy 'blue'. Próbáljuk a 3. elemet:
   ```
   szin[3]
   ```
10. Hibaüzenetet kapunk, mert ilyen elem nem létezik (IndexError: list index out of range).
11. Tegyük elérhetővé a **randrange** utasítást:
    ```
    from random import randrange
    ```
12. Hozzunk létre egy 0…2 véletlenszámot egy y változóban:
    ```
    y = randrange(0,3)
    ```
13. Ha most ezt az y változót használjuk a lista indexeként, egy *véletlenszerű színt kapunk* (a listában szereplők közül):
    ```
    szin[y]
    ```

A 12. és 13. lépések mutatják, hogyan lehet egy véletlenszámból egy lista segítségével véletlen színt kapni. Ezt majd a programban tovább vezetjük, mert beállítjuk vele a festés színét, és így véletlenszerűen színezett vonalakat kapunk.

12. mintafeladat – lista felhasználása programban

Készítsük el az ábrán látható csillagszerű alakzatot, ha tudjuk, hogy ágai 200 képpont hosszúak, 20 db ága van, és mindegyik ág színe véletlenszerű.

1. Nyissunk egy új programot (**File/New File**), majd mentsük el rögtön **feladatl_09_1** néven (**File/Save As...**).
2. Létrehozunk a színek nevéből egy listát. A lista neve *szin* lesz, és kilenc szín angol nevét tartalmazza. A könnyebb áttekinthetőség kedvéért a listát három sorra törtem. Ezt szabad a Pythonban. Általában sort nem törhetünk, de listát a zárójelek között igen (4–6. sor).

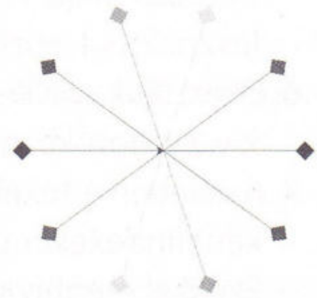


3. Mivel húsz ágat kell rajzolni, **for** ciklust fogunk használni, ami 20-szor fog lefutni (7. sor).
4. A ciklus minden egyes lefutásakor készítünk egy 0 és 8 közötti véletlenszámot (8. sor), amit az y változóban helyezünk el.
5. A lista y. elemére hivatkozunk, hogy kiválasztjuk egy szín nevét. Mivel az y véletlenszám, az így keletkező szín neve is véletlenszerű. A szín neve a *szinnev* változóba töltődik (9. sor).

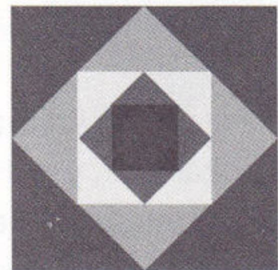
6. Beállítjuk az ecset színét a **color** utasítás segítségével. Paraméterként a *szinnev* változót használjuk, így a beállított szín véletlenszerű lesz (10. sor).
7. Előremegyünk 200 egységet, ezzel rajzolunk egy vonalat az előbb kiválasztott véletlen színnel.
8. Mivel a következő ág megint a középpontból indul, vissza kell vinni oda a teknőcöt. Erre a célra szolgál a hátrálás parancsa, a **backward** (12. sor).
9. Mivel 20 ág lesz, és a teljes kör 360 fok, így két ág között 360 fok : 20 = 18 fok lesz. Ennyivel fordítjuk el a teknőcöt balra a **left** utasítással (13. sor).
10. Mentsük el a programot az eredeti nevén a **Ctrl+S** billentyűk egyszerre történő lenyomásával, majd futtassuk a programot (**F5**), hogy lássuk az eredményt.

```
 1 from turtle import *
 2 from random import randrange
 3
 4 szin = ["green", "red", "yellow",
 5        "blue", "orange", "brown",
 6        "black", "purple","pink"]
 7 for i in range(20):
 8     y = randrange(0,9)
 9     szinnev = szin[y]
10     color(szinnev)
11     forward(200)
12     backward(200)
13     left(18)
```

Feladatok



1. Rajzoljuk meg a mellékelt csillagot úgy, hogy a sugarak és a végükön a négyzetek színezettek legyenek. A sugár és a hozzá tartozó négyzet egyforma színű. A piros, narancs, sárga, zöld, kék színek ismétlődjenek.
2. Bővítsük a zászló rajzoló programunk utolsó változatát *(gyak_05_1)* úgy, hogy véletlenszerűen válasszon országot. Ne módosítsunk a már létező eljárásokon, csak a főprogramon.



3. Rajzoljuk meg a mellékelt alakzatot. A színek kívülről befelé sorrendben piros, narancs, sárga, zöld, kék, lila legyenek.
4. Oldjuk meg az előző feladatot úgy, hogy a színek véletlenszerűen legyenek kiválasztva.

10. FELTÉTELES CIKLUS

13. mintafeladat – feltételes ciklus alkalmazása

Vegyük elő trikolor rajzoló programunkat (feladatI_04), és rajzoltassunk vele olyan zászlót, amiben a színeket véletlenszerűen választjuk ki, de két egymás melletti csík nem lehet ugyanolyan színű. Legyen csak három szín, hogy könnyű legyen a tesztelés (piros, sárga, kék), így gyakran ismétel.

1. Nyissuk meg a **feladatI_04** nevű fájlt, majd mentsük el rögtön **feladatI_10** néven (**File/Save As...**).
2. Mivel véletlenszámokat kell készítenünk, a **randrange** utasítást elérhetővé kell tenni. Írjuk be a program második sorába:
   ```
   from random import randrange
   ```
3. Az eljárásokhoz nem kell nyúlni (teglalap, trikolor), csak a főprogramhoz. (Ezért szeretjük az eljárásokat. Részekre lehet bontani a programot, és nem kell az egészet átnézni egy kis módosításhoz.)
4. Első próbálkozásra változtassuk meg a főprogramot, hogy a mellékelt módon nézzen ki.
5. Létrehozzuk a listát, aminek a segítségével a véletlenszámokból majd véletlen színek lesznek (34. sor).
6. Elkészítjük a véletlenszámokat x, y, z változókban (35–37. sor).

```
32 reset()
33 color("gold")
34 szin = ["red","blue","yellow"]
35 x = randrange(3)
36 y = randrange(3)
37 z = randrange(3)
38 trikolor(szin[x], szin[y], szin[z])
```

7. A véletlen értékeket tartalmazó változókat behelyettesítjük *szin* lista sorszámaként (indexeként): *szin[x], szin[y], szin[z]* már a véletlen színek nevét tartalmazza. Ezekkel meghívjuk a *trikolor* eljárást, ami kirajzolja a megfelelő zászlót.
8. Mentsük el a programot az eredeti nevén a **Ctrl+S** billentyűk egyszerre történő lenyomásával, majd futtassuk a programot (**F5**), hogy lássuk az eredményt. Párszor lefuttatva látni fogjuk, hogy előfordul két azonos színű, szomszédos csík.

*Az első szín száma szabadon megválasztható, tehát x előállításán nem kell módosítanunk. y-nál viszont vizsgálni kell, hogy egyenlő-e az x-szel, és ha igen, új véletlenszámot kell készíteni. És ha ismét egyezik, akkor megint újat, ha az is egyezik, megint újat, stb. Lehetséges, hogy többször is ismételni kell a véletlenszám készítést. Ha ismétlés, akkor ciklust alkalmazunk. Azonban **for** (számláló)*

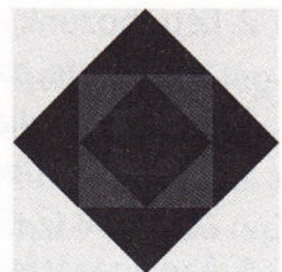
ciklust nem lehet, mert nem tudjuk előre, hányszor kell ismételni. Olyan másfajta ciklusra van szükségünk, ami egy feltételhez tudja kötni, kell-e ismételni vagy sem. A példában ez a feltétel, hogy x és y egyenlő. Nézzük meg, hogy kell egy ilyen ciklust létrehozni.

```
32 reset()
33 color("gold")
34 szin = ["red","blue","yellow"]
35 x = randrange(3)
36 y = randrange(3)
37 while x == y:
38     y = randrange(3)
39 z = randrange(3)
40 while y == z:
41     z = randrange(3)
42 trikolor(szin[x], szin[y], szin[z])
```

9. Változtassuk meg a főprogramot úgy, hogy a mellékelt módon nézzen ki.
10. Az új ciklust a **while** utasítás vezeti be (37. sor).
11. Mögötte megadjuk a ciklusban történő bennmaradás feltételét, tehát amíg a ciklusnak ismételni kell. Esetünkben ez x == y, hiszen addig kell ismételni, míg a leendő szín sorszáma megegyezik az előzőével.
12. A feltétel mögé kettőspontot kell tenni.
13. A következő sor beljebb kezdődik 4 karakterrel. Ez jelzi, melyik utasítást (utasításokat) kell ismételni a ciklusnak. Ez a ciklus csak egy sort, az új y véletlenszám készítését (38. sor).
14. Mikor a ciklusnak vége (x és y nem egyenlő), van két véletlenszámunk, ami nem egyezik.
15. Ugyanez történik a z változóval (39–41. sor).
16. Mentsük el a programot az eredeti nevén a **Ctrl+S** billentyűk egyszerre történő lenyomásával, majd futtassuk a programot (**F5**), hogy lássuk az eredményt.

Mivel a ciklus végrehajtása feltételhez kötődik, **feltételes ciklusnak** nevezik.

Feladatok

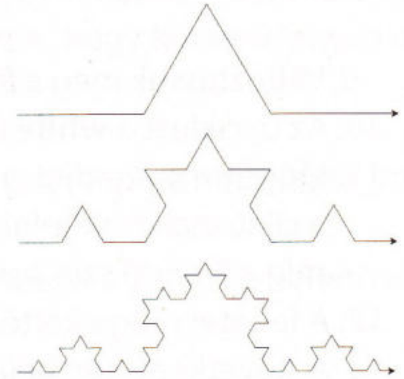


1. Rajzoljuk meg a mellékelt alakzatot. A színek a piros, narancs, sárga, zöld, kék, lila közül kerüljenek ki véletlenszerűen, de két egymás utáni színezés ne lehessen azonos.
2. Készítsük el a bolyongc teknőcöt olyan változatban, hogy egymás után ne léphessen kétszer ugyanabba az irányba. Emlékeztetőül: mindig 25 egységet lép előre, de minden lépés után a teknőc véletlenszerűen elfordul balra 90 fokot, vagy jobbra 90 fokot, vagy 180 fokot, de az is megtörténhet, hogy ugyanarra néz továbbra is, mint előzőleg.

11. REKURZIÓ

Rekurzióról beszélünk, ha *egy eljárás önmagát tartalmazza, hívja* meg. A rekurzió segítségével kiküszöbölhetők a ciklusok, és sok esetben egyszerűbb a program, többet árul el a feladat megoldásának módjáról.

A **fraktálok** olyan *alakzatok*, amik valamilyen módon önmagukat tartalmazzák, önmagukból épülnek fel[10]. Nézzük meg a Koch-fraktált a mellékelt ábrán. A felső ábrán látható alakzat tetten érhető a további alakzatokban.



Fontos jellemzőjük még, hogy a végtelenségig bonyolíthatók. Összetettségüket szintekkel jellemezhetjük. A felső alakzat az 1. szintje, a középső a 2. szintje, az alsó a 3. szintje ugyanannak a Koch-fraktálnak.

A rekurzió és a fraktálok definíciójának összevetésből sejthető, hogy a fraktálokat praktikus lehet rekurzív programokkal megrajzoltatni. A rekurzív programok visszavezetik a rajzolást a legegyszerűbb esetre.

14. mintafeladat – Koch-fraktál rajzolás rekurzív programmal

Készítsünk programot, ami képes megrajzolni a Koch-fraktál bármelyik szintjét[11]. Alkalmazzunk rekurziót, azaz önmagát hívó eljárást.

1. Nyissunk egy új programot (**File/New File**), majd mentsük el rögtön **feladatl_10_1** néven (**File/Save As...**).
2. Létrehozunk egy *koch* nevű eljárást, aminek két paramétere lesz. A *h* az alakzat mérete, az *sz* pedig a fraktál összetettségi szintje.
3. A rekurzív eljárások mindig tartalmaznak egy elágazást, ami szétválasztja a legegyszerűbb alakzatot előállító ágat (5. sor) és a legegyszerűbbre *visszavezető* ágat (7–13. sor).
4. Az elágazás feltételének a szintre kell vonatkoznia (*sz* változó). A legegyszerűbb eset a 0. szint, ezért a feltétel az sz == 0. (4. sor)

10 A fraktáloknak nincs egységes megállapodás szerinti, pontos definíciója.

11 Elvileg bármelyik szintjét, mert a felbontás, a képernyő mérete, a program futásának sebessége korlátozó tényezők.

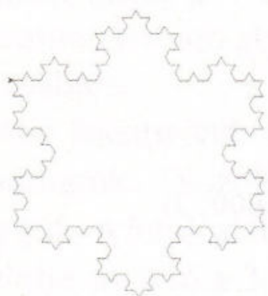
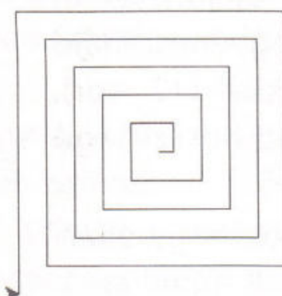
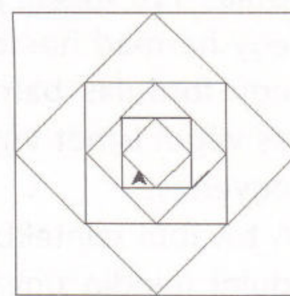
```
 1 from turtle import *
 2 
 3 def koch(h, sz):
 4     if sz == 0:
 5         forward(h)
 6     else:
 7         koch(h/3, sz-1)
 8         left(60)
 9         koch(h/3, sz-1)
10         right(120)
11         koch(h/3, sz-1)
12         left(60)
13         koch(h/3, sz-1)
14 
15 reset()
16 up()
17 goto(-300,0)
18 down()
19 koch(600,3)
```

5. A fraktál 0. szintje egy egyenes, ami egyetlen **forward** utasítással megrajzolható (5. sor).
6. Az egyszerűbb esetre visszavezető ágat az ábrák alapján lehet kitalálni. Az 1. szint alapján rajzolni kellene egy egyenest, aminek a hossza az eredeti hossz harmada, aztán el kell fordulni balra 60 fokot (8. sor), megint egy harmad hosszúságú egyenes jönne, majd egy fordulás 120 fokkal jobbra (10. sor), ismét egy harmad hosszúságú egyenes, majd egy fordulás balra 60 fokkal (12. sor), és végül ismét egy harmad hosszúságú egyenes.
7. A további szintekből látható, hogy elfordulni mindig ugyanúgy kell, mint az 1. szinten, tehát azokon az utasításokon nem kell változtatni (8., 10., 12. sorok).
8. Viszont nem mindig egyeneseket kell rajzolni, hanem az előző szint ábráját. (Az 1. szintnél az előző szint a 0., tehát ebben az esetben az egyeneseket fogja rajzolni.) Így a 7., 9., 11., 13. sorokban meg kell hívni a Koch-fraktált rajzoló eljárást *(koch),* de egy szinttel egyszerűbbet választva *(sz-1),* és a megrajzolandó részalakzat méretének is harmadának kell lennie az eredetihez képest *(h/3).*
9. A főprogramból indítjuk el a rekurzív eljárást. A példában 600 képpont az alakzat mérete, és 3 szint bonyolultságú alakzatot rajzolunk (19. sor).
10. Mentsük el a programot az eredeti nevén a **Ctrl+S** billentyűk egyszerre történő lenyomásával, majd futtassuk a programot (**F5**), hogy lássuk az eredményt.

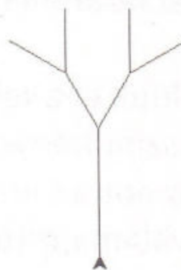
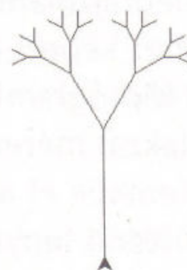
A rekurzió, az önmagát hívó eljárás a 7., 9., 11., 13. sorban figyelhető meg. A legegyszerűbb alakzatot előállító ágban található utasításokat szokás **iniciátornak**, a visszavezető ágban lévőket **generátornak** nevezni. A rekurzív hívásnak a paraméterét úgy kell megválasztani, hogy az elágazásban található változó (sz) egyszer elérje azt az értéket, ami a legegyszerűbb esetet tartalmazza. Ha ez nem történik meg, a programnak sosem lesz vége. Ilyenkor csak a **parancsértelmező újraindítása** segít, amit a **Ctrl+F6** billentyűk együttes lenyomásával érhetünk el.

Feladatok

1. Három Koch-fraktál összeillesztéséből rajzoljuk meg a Koch-pelyhet. A bal oldali ábrán láthatjuk a 3. szintjét.
2. Rajzoljuk meg a középső ábrán látható alakzatot ciklusok nélkül, rekurziót használva.
3. Rajzoljuk meg a jobb oldali ábrán látható alakzatot ciklusok nélkül, rekurziót használva[12].

4. Rajzoljuk meg rekurzív módon az alábbi ábrákon látható fát. Az ábrák rendre a 0., 1., 2., 4. szintet mutatják.

12 Bontsuk két lépésre a feladatot.
1. Egy négyzet megrajzolása legyen rekurzív.
2. A teljes alakzat legyen rekurzív.

II. A PROGRAMOZÁS ALAPJAI

1. EGY PROGRAM ÁLTALÁNOS FELÉPÍTÉSE

Egy számítógépes program a következő részekből épül fel: adatszerkezet, üzleti logika és felhasználói felület. Természetesen egy pár soros programnál ezek nem különülnek el nagyon egymástól. Egy millió sorból álló szoftvernél azonban már más a helyzet.

Adatszerkezet

A program által felhasznált adatokat (legyenek ezek maradandók vagy átmenetiek) tárolni szükséges. Ehhez különböző adatszerkezeteket használunk (változók, fájlok, adatbázisok). Az adatszerkezetek közül a feladatnak megfelelően választhatunk. Minden program alapja a jól megválasztott adatszerkezet, ezért ezek tervezésére, átgondolására különös gondot kell fordítani.

Üzleti logika

Az üzleti logika néha csak pár egyszerű sor, máskor viszont bonyolult és hosszú sorok sokasága. **Alkalmazáslogikának** is hívják. A program működésének lelkét az üzleti logika teszi ki. Itt határozzuk meg, mit is tegyen a program. (Tehát nincs köze a kereskedelemhez.)

Felhasználói felület

A program használója a felhasználói felülettel találkozik. Itt jelennek meg a program üzenetei, eredményei, kérdései. Ez lehet egy megszokott windowsos kialakítás, webes felület vagy kizárólag szöveges megjelenés, stb. Bármelyik is legyen, mindig ügyelni kell az áttekinthetőségre és az egyértelműségre.

A kizárólag szöveges megjelenésű programokat **konzolalkalmazásnak** nevezik.

Érettségire is elvárás, hogy a bekért és kiírt adatokat szöveges magyarázattal kísérjük: milyen adatot kérünk be, mit jelenítünk meg a képernyőn. Közismereti érettségire **csak konzolalkalmazást** kell tudni írni.

A konzolalkalmazások jelentősége, hogy itt a legegyszerűbb kipróbálni a megismerendő szoftvertechnikákat. Néhány esetben nincs is lehetőség vagy szükség grafikus felület kialakítására technikai vagy anyagi okokból.

Főleg webes fejlesztéseknél a kinézet kiemelten fontos szempont, ezért a felhasználói felület fejlesztése sok emberi erőforrást igényel. Gyakran vonnak be webgrafikust.

2. ALGORITMUS

Algoritmus fogalma

Amikor valamilyen feladatot kapunk, a megoldáshoz lépések sorozata vezet. Legyen az egy étel elkészítése vagy egy összeadás elvégzése (akár kézzel, akár géppel), mind egymás után következő lépésekből áll.

Algoritmusnak nevezzük azoknak a lépéseknek a sorozatát, amik egy felmerülő feladat megoldására alkalmasak.

Algoritmusok például az ételreceptek, a lépésről lépésre (step by step) típusú összeszerelési útmutatók, a kémiai kísérletek leírásai vagy a természetes számok prímtényezős felbontásának módszere.

A lépéseket **utasítások**ként fejezzük ki: "Törj fel három tojást!". Ez még akkor is igaz, ha mondatainkat a tisztelet vagy udvariasság jeléül kevésbé direkt módon fogalmazzuk meg.

Algoritmus-leíró eszközök

Az általánosan használt algoritmusokat programozási nyelvtől függetlenül szokás megfogalmazni. Ennek két előnye van. Egyrészt az algoritmus rövidebb, lényegre törőbb, átláthatóbb szokott lenni, mint a kód. Másrészt az algoritmus-leírókat elvileg mindenkinek ismernie kell, míg egy-egy konkrét programnyelvet nem feltétlenül.

- Az algoritmusokat a legkevésbé formális módon mondatokkal fogalmazzuk meg. Ezt nevezik **mondatos leírásnak**. Ilyenek például az ételreceptek. Sok esetben azonban a mondatos leírás nem bizonyul elég precíznek, s ez megakadályozza az algoritmus számítógépre vitelét. Ezért találtak ki olyan algoritmus-leíró eszközöket, amik formálisabbak, azaz a szabályai kötöttebbek.
- **Folyamatábrát** használhatunk rövidebb algoritmusok szemléltetésére. A lépéseket különböző alakú síkidomok jelölik, sorrendjüket nyilak mutatják. Ez látványos, könnyen érthető leírási mód. A grafikus elemek helyfoglalása miatt nagyobb algoritmusok ilyen jellegű ábrázolása problémás.
- Egy másik lehetőség a **mondatszerű leírás**, amit több helyen **pszeudokódnak** vagy **pszeudonyelvnek** hívnak. A mondatos leíráshoz képest annyi a különbség, hogy **előre rögzítik a felhasználható mondatelemeket**.

Léteznek egyéb algoritmus-leíró eszközök, mint például a stuktogram vagy a Jackson-ábra. Ezek megismertetése azonban túlmutat e könyv keretein.

Algoritmus megadása folyamatábrával

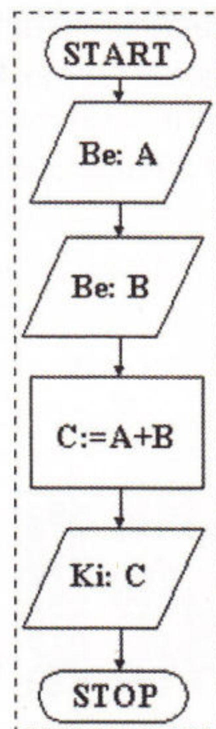
Tekintsünk egy nagyon egyszerű feladatot: kérjünk a felhasználótól két számot, adjuk őket össze, és az eredményt jelenítsük meg.

A feladat megoldásának folyamatábráját láthatjuk a jobb oldalon.

Paralelogrammával jelöljük az **adat be- és kivitelt**, **téglalappal az értékadást**[13], valamint a **műveletvégrehajtást**.

A folyamatábra kötelezően **START-tal kezdődik**, és **STOP-pal fejeződik be**.

Léteznek további folyamatábra-elemek is, amikkel a Vezérlési szerkezetek témakörben fogunk megismerkedni.



13 *c* legyen egyenlő *a* és *b* összegével.

Algoritmus megadása mondatszerű leírással

A megoldás lépéseit rövidítve egymás alá írjuk.

Az összeadó programunk így néz ki mondatszerű leírással:

Be: a
Be: b
c:=a + b
Ki: c

Léteznek további megállapodás szerinti mondatelemek, amikkel a Vezérlési szerkezetek témakörben fogunk megismerkedni.

Algoritmizálás

Az algoritmus kialakítása az algoritmizálás.

Néhány esetben azonnal átlátható, milyen lépéssorozatot kell tennünk. Máskor hosszan töprengünk, és különböző módszereket alkalmazunk. Előfordulhat az is, hogy egy feladatnak nincs megoldása.

A feladat akkor algoritmizálható könnyen, ha alkalmazható rá valamilyen már megismert séma: korábban tanult képletek, típusalgoritmusok, módszerek.

3. PROGRAMNYELVEK

Gépi kód

```
M 2000
>2000 A9 02 8D 19 FF 00 85 49 :
>2008 A9 D6 9D 18 0F 9D 40 0F :
>2010 4C 1D 20 A9 74 9D 18 0F :
>2018 A9 75 9D 40 0F 88 E8 C0 :
>2020 00 D0 D9 60 20 A7 20 A9 :
>2028 03 8D 10 FF A9 00 8D 0F :
>2030 FF A0 0A 20 11 21 AD 0F :
>2038 FF 18 69 01 49 80 8D 0F :
>2040 FF AE 4C 03 DE C4 0F BD :
>2048 C4 0F C9 6D D0 E3 EE 4C :
>2050 03 AD 4C 03 C9 22 D0 D9 :
>2058 A9 21 8D 4C 03 4C B0 20 :
```

Kezdetben a számítógépeket közvetlenül gépi kódban programozták. A program bájtjait egymás után beírták a számítógépbe. A mellékelt ábrán egy Commodore Plus 4 számítógép memóriájában található gépi kódú program látható. A program a 2000 című[14] memóriarekesztől indul. A mindössze 5 egymást követő bájt képes megváltoztatni a képernyő keretszínét. Minden egyes kétjegyű szám 16-os számrendszerben van (pl. A9), így egyetlen bájtot jelent. A képen jól tetten érhető a Neumann-elv, miszerint utasítás és adat ugyanabban a memóriában helyezkedik el: az A9 egy elemi utasítás, az utána következő 02 viszont adat, a sötétvörös szín kódja. A 8D ismét utasítás, majd a 19-ből és FF-ből tevődik össze annak a regiszternek a címe (FF19h), ahová a szín kódját írni kell a keretszín megváltoztatásához.

A gépi kódú programokat megírni hosszadalmasnak és nagy szakértelmet igénylő feladatnak bizonyult. Pontosan kellett érteni a számítógép belső felépítését. Tudni kellett processzor módra "gondolkodni".

A számítógép alapvetően ma is csak gépi kódú programokat tud futtatni. Elsősorban azokat a fájlokat, amiket a Windows alatt **exe** kiterjesztéssel találunk meg. A gépi kódú programokat szokás **binárisoknak** is nevezni.

Assembly

```
D2000
. 2000  A9 02     LDA #$02
. 2002  8D 19 FF  STA $FF19
. 2005  00        BRK
. 2006  85 49     STA $49
. 2008  A9 D6     LDA #$D6
. 200A  9D 18 0F  STA $0F18,X
. 200D  9D 40 0F  STA $0F40,X
. 2010  4C 1D 20  JMP $201D
. 2013  A9 74     LDA #$74
```

Némi könnyebbséget jelentett az assembly megjelenése, ahol a még mindig gépi kódú utasításokat rövid, de jellemző nevekkel helyettesítették. A képen látható gépi kód mellett szerepel az assembly utasítás is: az A9

14 Ejtsd: kettő-nulla-nulla-nulla. Nem kétezer, mivel ez nem egy tízes számrendszerbeli szám, hanem tizenhatos. Abban pedig nincs ezres helyiérték.

utasításnak az LDA (LoaD Accumulator), a 8D-nek az STA (STore the Accumulator) név felel meg.

Az eredetileg gépi kódban vagy assemblyben fejlesztett programok kevés helyet foglalnak, és nagyon gyors működésűek. Fejlesztésük nehézkes, nagy szakértelmet igényel, ezért borzasztó lassú és drága. A gyors számítógépek korában gépi kóddal csak különleges célok megvalósítása érdekében foglalkoznak. Ezek közé tartozik a programok illegális módosítása, feltörése is.

Magas szintű programnyelvek

Az assembly sem bizonyult elég hatékony eszköznek. Olyan programozási lehetőséget kellett kifejleszteni, ami az emberi gondolkozáshoz közelebb áll. Megjelentek a mai programnyelvek elődei, köztük a C, a Pascal és a Basic. Ezekben programozni sokkal könnyebb, mint gépi kódban, mert emberibb gondolkodásmódot igényelnek. A fejlesztő C nyelven megírja a programot, amit **forráskódnak** nevezünk. A forráskódot egy C **fordítóprogram** gépi kódú programmá (binárissá) alakítja.

Az így elkészült bináris nem lesz olyan kicsi és gyors, mintha eredetileg gépi kódban írták volna, mert a fordítóprogram sok felesleges dolgot hozzátesz. Az idők folyamán egyre jobb fordítók készülnek, de tökéleteset sosem fognak tudni létrehozni. Így maradt a választás a kicsi és gyors, vagy az olcsó és pazarló, de azonnali között. A választást pedig a felgyorsult üzleti világ már rég eldöntötte az utóbbi javára. A sebességen egyre gyorsabb és nagyobb kapacitású gépekkel igyekeznek segíteni.

Az emberi gondolkodásmódhoz közelebb álló programnyelveket **magas szintűeknek,** a gépi kódot és az assemblyt **alacsony szintűeknek** nevezzük.

Pár száz soros, egyszerűbb programok megírásához a magas szintű nyelveken történő strukturált programozás ma is elegendő (például programok mikrokontrollerre, kisebb webes scriptek). Érettségin sem várnak el többet.

A forráskódot bizonyos programnyelvek nem alakítják át előzőleg teljes egészében gépi kódú programmá, hanem utasításonként (soronként) hajtják végre. Ezekben az esetekben nem fordítóprogramról, hanem értelmezőprogramról, más néven **interpreterről** beszélünk.

A **Python** értelmezőprogramként működik, így ne is keressük az exe fájlokat, hiszen nem készülnek. Léteznek a Pythonhoz is fordítóprogramok, amiknek a beüzemelése nem egyszerű feladat. Érettségire természetesen nem elvárás az exe fájl készítése.

Objektumorientált programozás

Idővel a piac által igényelt szoftverek terjedelme és bonyolultsága annyira megnövekedett, hogy a hagyományos magas szintű nyelveken, strukturális programozással nem lehetett őket kellően gyorsan, gazdaságosan, bővíthetően megírni. Ezért ki kellett fejleszteni egy új módszert. Ez lett az objektumorientált programozás.

Az objektumorientált programozáshoz a már létező magas szintű programnyelveket (C, Pascal, Basic) fejlesztették tovább. Létrejött a C++, C#, Java, Object Pascal, Delphi, stb.

Az objektumorientált forráskódot is fordítóprogramok alakítják futtatható binárissá. Az így készült programok még a hagyományos strukturált programozással létrehozottaknál is lassabbak és nagyobbak (lásd Windows).

Grafikus fejlesztői környezetek

A piac azonban még gyorsabb és hatékonyabb fejlesztést vár el, egyre összetettebb programokkal. Ehhez készítenek olyan fejlesztői környezeteket, amikben grafikusan, minél kevesebb kódolással lehet létrehozni programokat.

Ilyen például a Visual Studio, ahol a Windows alkalmazásokhoz szükséges gombokat, menüket, stb. az ablakra húzzuk egérrel, s a létrehozásukhoz szükséges kódot a fejlesztői környezet készíti el. Az eseményekhez (például, hogy mi történjen kattintáskor) tartozó programrészeket még nekünk kell megírni. Léteznek már olyan fejlesztői környezetek is, ahol a vezérlési szerkezeteket (pl. ciklus) is grafikusan lehet felvinni és beállítani.

Általánosságban elmondható, minél kényelmesebb egy fejlesztői környezet, annál több helyet foglal, és a belőle fordított program is annál nagyobb és lassabb. Minél kényelmesebb egy fejlesztői környezet, annál automatikusabban készíti a kódot, így kötöttebb a használata, kevésbé támogatja az egyéni, különleges megoldásokat. Gyakran a kód bizonyos részeihez nem is férhetünk hozzá, azt kizárólag a fejlesztői környezet készítheti el.

Üzleti célú és ingyenes fejlesztői környezetek

Az ingyenes fejlesztői környezetek általában kevésbé kidolgozottak, kevesebbet "tudnak", körülményesebben kezelhetők, mint üzleti célú társaik. A könyvben szereplő Python például kizárólag a kódírást támogatja, a Windows alkalmazásokhoz az ablak elemeit sem lehet egérrel elhelyezni.

Az üzleti célú fejlesztői környezetekkel gyorsan lehet haladni. A kód egy részét nem kell megírni, csak megtervezni. A kódot a fejlesztői környezet a terv alapján elkészíti. Legnagyobb problémájuk a borsos ár, ami egy jól menő szoftverfejlesztő cégnek hamar megtérül a befektetett munkaórák alacsonyabb száma miatt.

Gyakran alkalmazott trükk a "beetetés", hogy oktatási célból a fejlesztői környezet ingyenes. Több ismerősöm is pórul járt már így. Megtanult programozni egy ilyen környezetben, majd rájött, hogy eladható szoftvert tud készíteni vele. A szoftvert elkészítette, de az eladáshoz jogtiszta üzleti célú fejlesztői környezetre lett volna szüksége. Ennek árát kisvállalkozóként nem tudta kifizetni. Így az öt év alatt kifejlesztett programját további három év alatt át kellett írnia ingyenes környezetbe.

4. HIBÁK

Ahol ember dolgozik, ott mindig előfordulhatnak hibák, amik a program működését megzavarhatják, vagy akár lehetetlenné is tehetik. A hibák lehetnek szintaktikaiak, illetve szemantikaiak.

Szintaktikai hibák

Szintaktikai, azaz alaki vagy nyelvi hibáról beszélünk, ha a fordító- vagy értelmezőprogram nem fogadja el a kódot.

Ennek oka lehet például:

- Elgépelés: kihagytunk egy betűt egy utasításból.
- Valamelyik zárójelnek nincs meg a nyitó vagy éppen a lezáró párja.

A szintaktikai hibákat általában könnyű megtalálni a kód alapos szemrevételezésével. A fejlesztői környezetek is segítségünkre vannak, megmutatják, hol állt meg a program fordítása. Általában azon a környéken kell keresnünk a hibát. *Hibaüzenetet* is kapunk a fordítótól, ami gyakran segíti a hiba felderítését.

Szemantikai hiba

Szemantikai, azaz tartalmi hiba esetén a program lefordítható vagy értelmezhető, de nem azt csinálja, amit elvárunk tőle.

Előfordulhat, hogy a program lefagy, helytelen eredményeket ad, vagy valamilyen szabálytalan művelet végrehajtása miatt *futási hibával* (runtime error) megáll.

A szemantikai hibákat nehezebb megtalálni, mint a szintaktikaiakat. Alapos kipróbálással, **teszteléssel deríthetjük fel** őket. Egy részük a kódolás közbeni teszteléskor előjön. Azonban előfordulhat, hogy csak évek múlva derül ki egy-egy ilyen meglapuló hiba, amit *bug*nak is szoktak nevezni.

Fejlesztés = kódolás + azonnali tesztelés

Egy program fejlesztése a kód soronkénti beírásából és optimális esetben minden lehetséges részlépésnél a program azonnali teszteléséből, majd a hibák elhárításából áll. Az is egy lehetséges teszt, hogy a programunk még semmi értelemeset nem tesz, de hiba nélkül lefordul.

A tesztelés időigényes folyamat. Mindenki szeretne minél kevesebbet tesztelni, hiszen azért a megrendelő nem fizet, az érettségin nem jár érte pont. Mégsem taktikus megírni száz sort, és csak utána tesztelni, mert egy hiba felderítése sokkal több időt vehet így igénybe, mintha soronként teszteltünk volna.

Kezdők[15] minimum utasításonként teszteljenek. Ha az utasításnak sok paramétere van, akkor két paraméterenként. Később kialakul egy ösztön, ami az azonnali tesztelés és kódolás arányát beállítja a megfelelő szintre. Lesznek programrészletek, ahol sűrűbb tesztelés kell, és lesznek olyanok is, ahol elég a ritkább. (A program átírása előtt gondoskodjunk megfelelő mentésről.)

15 Akik még nem készítettek el 10 ezer működőképes kódsort.

5. A PROGRAMFEJLESZTÉS LÉPÉSEI

1) A **feladatok** alapos **tanulmányozása**, az ügyféligények felmérése.
2) **Tervezés**: Egyszerűbb program (lásd érettségi) esetén ez történhet fejben. Összetett program esetében erre a célra vannak a terv dokumentálását segítő programok.
   a) Eldöntjük, milyen **adatszerkezeteket** fogunk használni.
   b) Lejegyezzük, milyen **algoritmusokat**, képleteket, programozási tételeket (VI. fejezet) vagy egyedi megoldásokat kell alkalmaznunk. A lejegyzés egyszerűbben történhet megjegyzések formájában a leendő kód megalapozásaként, esetleg papíron.
3) Elvégezzük a **kódolást** a célszerűen kiválasztott programnyelven. Kódolás közben folyamatosan elvégezzük az **alapvető teszteléseket** és **hibajavításokat**. Így kiküszöböljük a szintaktikai, valamint az egyszerűbb szemantikai hibákat. Eredményként kapunk egy működő programot, aminek *helyes* működését még ellenőriznünk kell.
4) Alapos **tesztelés és hibajavítás** következik a bugok (mélyen ülő szemantikai hibák) kiküszöbölésére.
5) A program **dokumentálása** a tervezéstől a tesztelésig tart[16].
6) A munka végeztével **az elkészült programot átadjuk**. Szükség esetén az ügyfél számítógépén beüzemeljük.
7) A későbbiekben az ügyfelek esetenként igénylik a szoftver új funkciókkal történő felruházását. Ha utólag kiderül valamilyen hiba, szükséges lehet az ügyfelek gépén található programot újabbra cserélni, azaz **frissíteni**.

A tapasztalatok szerint a legkritikusabb rész az ügyfelekkel történő folyamatos kapcsolattartás. Hiába egy év a fejlesztési idő például. Ha ezalatt nincs kommunikáció, elvesztjük az ügyfél bizalmát. Ráadásul az ügyfél igényei folyamatosan változnak. Ezért az előzőekben tárgyalt lépések sok esetben egymásba csúsznak, ismétlődnek, stb.

Egy jól működő szoftveres cégnél az ügyfél kezelésére külön munkaerőt alkalmaznak. Azonban így is a tipikus szoftverfejlesztői munkának mindössze 20%-a közkedvelt kódolás, 40%-a a tervezés, és 40%-a a legnépszerűtlenebb munka, a tesztelés.

---

16 Érettségire nem elvárás.

III. PYTHON ALAPOK

1. "HELLÓ VILÁG!"

A Python rendelkezik egy parancsértelmezővel, ahol a parancsokat egyenként beírhatjuk és kipróbálhatjuk.

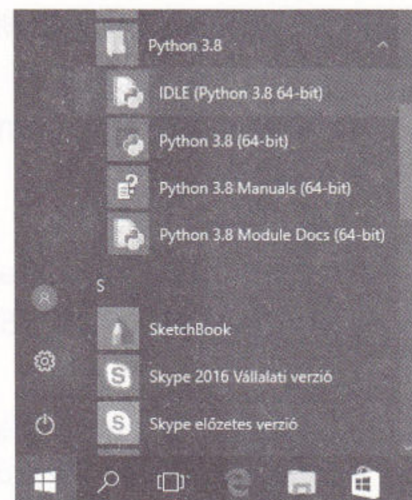
Parancsok begépelése

- Minden parancs végén meg kell nyomni az Enter billentyűt. Ezzel jelezzük, hogy befejeztük a parancs begépelését, kezdődhet a végrehajtása.
- Az utasításokon belül szabadon bánhatunk a szóközökkel. Azaz mindegy, hogy a print parancs után, vagy a zárójel és az idézőjel között hagyunk-e szóközt vagy sem.
- A karakterláncokat (ilyen a Helló világ! a példában) írhatjuk idézőjelek vagy aposztrófok közé is. Nincs különbség. A 'Helló világ!' is ugyanúgy helyes, mint a "Helló világ!". Ne keverjük őket. Például aposztróffal kezdeni és idézőjellel befejezni hibaüzenetet eredményez.

15. mintafeladat – Helló világ!

Hagyományosan, minden programozói tanulmány úgy kezdődik, hogy megtanuljuk kiíratni a képernyőre: "Helló világ!"

1. A Python parancsértelmezőjének indításához kattintsunk a **Start** menüben a **P** betűnél található **Python x.x/(IDLE Python x.x 64-bit)** menüpontra.
2. A >>> -tól jobbra, ahol a kurzor villog, írhatjuk be a parancsunkat.
3. A kiíratás parancsa a **print**. Zárójelek közé kell beírni, mit szeretnénk kiíratni.
4. Ha egy konkrét szöveget, akkor azt még idézőjelek közé is kell tenni.



5. Írjuk be a parancsot:
   print ("Helló világ!"), majd nyomjuk meg az **Enter** billentyűt.
6. A parancsértelmező kiírja a következő sorba kék betűkkel: "Helló világ!"
7. Írjuk be szándékosan rosszul a parancsot:
   prit ("Helló világ!"), majd nyomjuk meg az **Enter** billentyűt.

```
Python 3.8.0 Shell
File Edit Shell Debug Options Window Help
Python 3.8.0 (tags/v3.8.0:fa919fd, Oct 14 201
9, 19:37:50) [MSC v.1916 64 bit (AMD64)] on w
in32
Type "help", "copyright", "credits" or "licen
se()" for more information.
>>> print ("Helló világ!")
Helló világ!
>>> prit ("Helló világ!")
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    prit ("Helló világ!")
NameError: name 'prit' is not defined
>>> |
Ln: 10 Col: 4
```

8. Hibaüzenet jelenik meg (piros színnel kiírva). Ez figyelmeztet minket arra, hogy valamit rosszul írtunk be, vagy adtunk meg. A parancsértelmező megpróbálja behatárolni a hibát: 'prit' is not defined – a prit nincs definiálva, azaz nem ismert.

2. ALAPMŰELETEK

A parancsértelmező egyszerűen használható számológépként. Csak be kell írnunk a print utasításba a matematikában megszokott műveleteket.

16. mintafeladat – alapműveletek

Számoljuk ki, mennyi kerítés szükséges 45 m hosszúságú és 35 m szélességű téglalap alakú telek körbekerítéséhez, ha kihagyunk 5 m-t a kapunak?

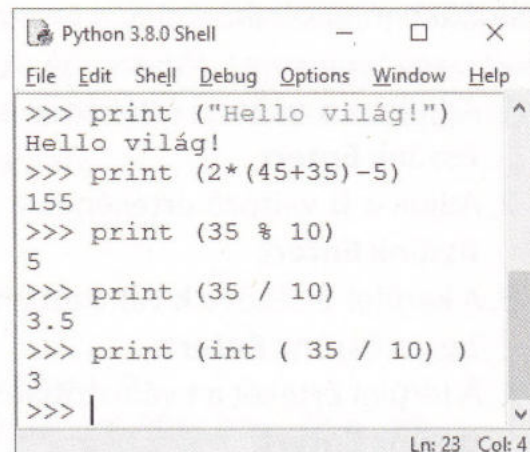
1. Ehhez ki kell számítani a téglalap kerületét, és kivonni belőle a kapu hosszát. A képlet:
   2 • (45 + 35) – 5.
2. A számítástechnikában a szorzás jele a *. Tehát a parancssorba írjuk be print(2 * (45 + 35) – 5), majd üssünk **Entert**[17].

17 Print parancs nélkül, csak a képletet beírva is megkaphatjuk az eredményt, de csak utasításként, a program részeként kötelező lesz a print beírása. Ehhez próbálunk most hozzászokni. A későbbiekben használjuk majd a rövidebb megoldást is.

17. mintafeladat – hányados, maradék, egészrész

Írassuk ki a 35 egyeseinek helyén található számjegyet, majd a tízesek helyén található is!

1. A 35 egyeseinek helyiértékén az 5 áll. Ehhez úgy jutunk, ha a 35-öt elosztjuk maradékosan 10-zel, és az osztás *maradékát* íratjuk ki.
2. Az osztási maradék műveleti jele a **%**. Tehát írjuk be a print(35 % 10) parancsot, majd üssünk **Entert**.
3. A parancsértelmező következő sorában láthatjuk a helyes végeredményt kékkel kiírva.
4. A 35-ben a tízesek helyén a 3 áll. Ehhez úgy juthatunk, ha a 35-öt elosztjuk 10-zel, és a hányadost íratjuk ki.
5. A hányados műveleti jele a **/**. Tehát írjuk be a print(35 / 10) parancsot, majd üssünk **Entert**.
6. Az eredmény helyén most 3.5 jelenik meg. Az értelmező nem maradékos osztást végzett. Ha azonban vesszük az eredmény egészrészét, a helyes eredményhez jutunk.
7. Az egészrész művelet az **int** kulcsszóval érhető el. Zárójelek közé kell tenni, aminek az egészrészét szeretnénk kiszámolni. A következő parancsot kell kiadni:
   ```
   print(int(35 / 10))
   ```
8. Megjelenik a helyes végeredmény (3) a következő sorban.
9. A példában láthattuk, hogy a *tizedestörteket tizedespontokkal* kell megadni.



3. VÁLTOZÓK

A részeredményeket a számológépünkben el szoktuk rakni egy gombnyomással a memóriába, majd amikor szükséges, előhívjuk onnan. Hasonló célt szolgálnak a programnyelvekben a változók. Csak mivel több van belőlük, másképpen használjuk őket.

A változónak (bizonyos korlátozásokkal) tetszőleges nevet adunk, és = segítségével értéket adunk neki. Mintha egy sokfiókos szekrény felhasznált fiókját címkével látnánk el (névadás), és beleraknánk, amit tárolni szeretnénk (érték).

18. mintafeladat – változók használata

Számítsuk ki a 35 m hosszúságú és 25 m szélességű téglalap alakú telek kerületét és területét úgy, hogy az oldalak hosszát, valamint a kerületet és a területet is változókban tároljuk.

1. Adjuk az **a** változó értékének a 35-öt. Írjuk be a parancssorba: a = 35, majd üssünk **Entert**.
2. Adjuk a **b** változó értékének a 25-öt. Írjuk be a parancssorba: b = 25, majd üssünk **Entert**.
3. A kerület értékét a **k** változóban a k = 2 * (a + b) képlettel adhatjuk meg. Írjuk be, és üssünk **Entert**.
4. A terület értékét a **t** változóban a t = a * b képlettel adhatjuk meg. Írjuk be, és üssünk **Entert**.
5. Írassuk ki a kerület, tehát a **k** változó értékét. A print utasításba csak be kell írni a változó nevét: print (k).
6. Megjelenik a kerület értéke (120).
7. Írassuk ki a terület, tehát a **t** változó értékét. A **print** utasításba csak be kell írni a változó nevét: print (t).
8. Megjelenik a terület értéke (875).
9. Növeljük meg az *a* változó értékét 5-tel. Írjuk be a parancssorba:
   ```
   a = a + 5
   ```
10. Ellenőrizzük, mennyi most az *a* értéke. Írassuk ki az értékét:
    ```
    print(a)
    ```
11. Valóban azt írta ki a parancsértelmező, hogy 40.

Elsőre furcsának tűnhet az a = a + 5, hiszen matematikailag értelmetlen. Egy szám nem lehet egyenlő önmaga plusz öttel. A Pythonban azonban az = jelentése *legyen egyenlő,* és nem a bal, valamint a jobb oldal összehasonlítása. (Arra majd az == jel szolgál.) Tehát az a = a + 5 így olvasandó*: a legyen egyenlő a továbbiakban a + 5-tel.* Ha a eddig 35 volt, akkor most 5-tel nagyobb lesz, tehát 40.

12. A Pythonban egy változó értékének növelésének rövidebb módja is van. Növeljük meg ismét a változó értékét 5-tel, de most ehhez ezt írjuk be:
    `a += 5`
13. Ellenőrizzük, mennyi most az *a* értéke. Írassuk ki:
    `print(a)`
14. Valóban azt írta ki a parancsértelmező, hogy 45.
15. Próbáljuk ki, működik-e a rövidített változat, ha a változó értékét csökkenteni akarjuk. Csökkentsük a *b* változó értékét 10-zel. Ehhez írjuk be:
    `b -=10`
16. Ellenőrizzük, mennyi most a *b* értéke. Írassuk ki az értékét:
    `print(b)`
17. Valóban azt írta ki a parancsértelmező, hogy 15.
18. Egy utasításban több értéket is adhatunk, ha a változókat és az értékül kapott kifejezéseket vesszővel elválasztjuk egymástól. Ugyanígy, vesszővel elválasztva, egy **print** utasítással több eredményt is kiírhatunk. Oldjuk meg újra a feladatot így, a jobb oldali ábra alapján.

```
Python 3.8.0 Shell
File Edit Shell Debug Options Window Help
>>> a = 35
>>> b = 25
>>> k = 2 * (a + b)
>>> t = a * b
>>> print (k)
120
>>> print (t)
875
>>>
Ln: 47 Col: 4
```

```
Python 3.8.0 Shell
File Edit Shell Debug Options Window Help
>>> a = a + 5
>>> print(a)
40
>>> a += 5
>>> print(a)
45
>>> b -= 10
>>> print(b)
15
>>> k,t = 2 * (a + b), a * b
>>> print (k,t)
120 675
>>>
Ln: 86 Col: 4
```

Áttekintés

A változó a memória egy lefoglalt darabja, amire a nevével hivatkozhatunk.

A változó nevének megválasztásánál a következőkre ügyeljünk:

- Az angol ábécé valamelyik betűjével kell kezdődnie.
- Az angol ábécé betűit, számokat és _ (aláhúzás jelet) tartalmazhat.
- Nem egyezhet meg egyik parancs nevével sem.

A változókon végrehajtott legfontosabb művelet az értékadás. A bal oldalon lévő változó értéke a továbbiakban egyenlő a jobb oldalon található kifejezéssel. Például b = 25. Amit így kell kiolvasni: *b legyen egyenlő 25.*

Több változónak egyszerre is értéket adhatunk, ha vesszővel elválasztjuk őket: a, b, c = 3, 4, 5.

Az *első használatkor* a Python lefoglalja a *változók helyét a memóriában,* hozzárendeli a változó nevét, amivel elérhetjük, és *meghatározza a változó típusát.* Utóbbira azért van szükség, mert a különböző adatokat eltérő módon érdemes tárolni a memóriában. Ilyen főbb adattípus a *karakterlánc* (karakterek egymás után), amit elterjedten *string*nek is neveznek. Adattípus csoport az *egész számok,* illetve a *valós számok* (lebegőpontos számok), valamint a *logikai értékek*[18]. Utóbbinak az értéke csak *igaz* vagy *hamis* lehet, a Pythonban *True* vagy *False.*

Szám változók esetén működnek a +=, -= *és* *= értékadások, amik a bal oldali változó értékét növelik, csökkentik, szorozzák a jobb oldalon megadott értékkel.

---

18 Sok programnyelvben a változókat használatuk előtt külön utasítással *deklarálni* kell. Ilyen nyelvek például a C++, C# vagy Pascal. Ezek az *erősen típusos nyelvek* csoportjába tartoznak. A Pythonban vagy a Basicben ez automatikusan megtörténik. Az ilyen nyelveket *gyengén típusos*aknak nevezik.

4. PROGRAM

Áttekintés

A gyakran használt, egymást követő parancssorozatokat célszerű lenne nem újra és újra begépelni, hanem egy fájlban tárolni, onnan szükség esetén előhívni, és egyetlen gombnyomással végrehajtani. Ilyen lehet például egy téglalap kerületét és területét kiszámító parancssorozat.

A program alkotórészeit inkább *utasításoknak* szokták nevezni. Tehát ugyanaz a print (3 * 5) parancs, ha a parancsértelmezőbe írjuk, és Entert nyomva azonnal végrehajtjuk, és utasítás, ha egy programnak a része.

Egy feladatot megoldó utasítások megfelelő sorrendben programot alkotnak. A programokban található utasítások végrehajtását futtatásnak nevezik.

A programírás szabályai a Pythonban

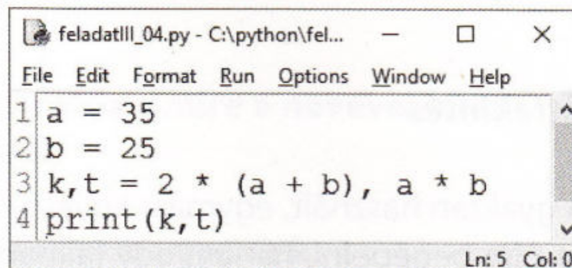
- A Pythonban az utasításokat egymástól a sorvége jel (Enter) választja el. Ezért minden utasítást új sorban kell kezdeni. A sorok végén meg kell nyomni az Enter billentyűt.
- Ha egy utasítás nagyon hosszú, és az áttekinthetőség kedvéért szeretnénk több sorban megjeleníteni, van lehetőség az elválasztásra. Az elválasztójel a \. Amelyik sor végén a parancsértelmező \-et talál, kibővíti a következő sorral, és úgy értelmezi.
- A sorok elején jelentősége van a szóközöknek. Nem mindegy, hol kezdődik az utasítás. Egyelőre kezdjünk minden utasítást a sor elején.

19. mintafeladat – program készítése, mentése, futtatása, megnyitása

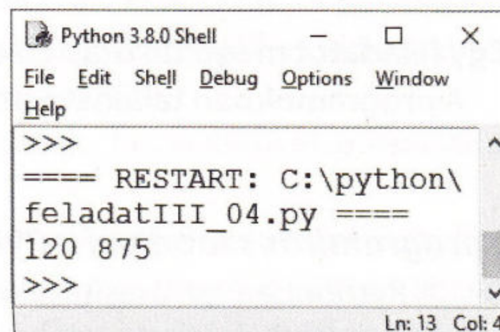
Készítsük el a téglalap kerületét és területét kiszámoló programot. A téglalap oldalai: 35 m és 25 m.

1. A Python Shell ablakban (ahol eddig dolgoztunk), kattintsunk a **File/New File** menüpontra.
2. Megjelenik egy új ablak. Nevezzük a továbbiakban szerkesztőablaknak. A programmal általában a szerkesztőablakban dolgozunk. Itt gépeljük be, javítjuk, futtatjuk, mentjük, stb.

3. Jelenítsük meg a sorok számozását, mert így áttekinthetőbb lesz a munka. Ehhez kattintsunk az **Options/Show Line Numbers** menüpontra.
4. Gépeljük be a programunkat pontosan a mellékelt ábra szerint látható módon. A sorok végén nyomjuk meg az **Enter** billentyűt.



5. Mentsük el a programunkat egy fájlba. Kattintsunk a **File/Save As...** menüpontra. Megjelenik a **Mentés másként** ablak, ahol a Windows alkalmazásoknál megszokott módon kiválaszthatjuk a fájl helyét, és megadhatjuk a nevét. Legyen a **Fájl neve** most **feladatIII_04**. (Végül a **Mentés** gombra kell kattintani, hogy a fájl ténylegesen a kiválasztott mappába kerüljön.)
6. Futtassuk a programot, azaz hajtsuk végre a benne található utasításokat. Ehhez nyomjuk meg az **F5** billentyűt. (Egérimádók választhatják helyette a **Run/Run Module** menüpontot.)



7. Az eredményt a **Python Shell** ablakban fogjuk látni. Kiírja a futtatott fájl (program) nevét, majd alatta megjelennek az eredmények (120 875).
8. Zárjuk be a feladatIII_04 ablakot az X-re kattintva a jobb felső sarokban.
9. Győződjünk meg róla, hogy a programunk nem veszett el, azaz nyissuk meg újra. Kattintsunk a **File/Open...** menüpontra. Megjelenik a Megnyitás ablak, ahol a Windows alkalmazásoknál megszokott módon kiválaszthatjuk a fájl helyét és nevét. Kattintsunk duplán a **feladatIII_04** nevű fájlra.
10. Megnyílik a szerkesztőablak, és benne látható a programunk.

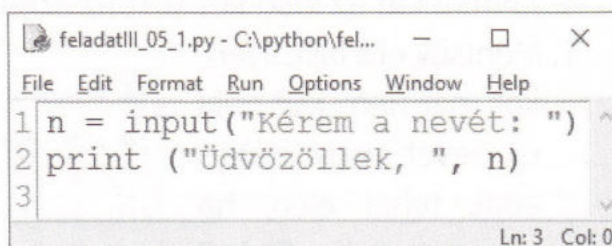
5. ADATBEKÉRÉS

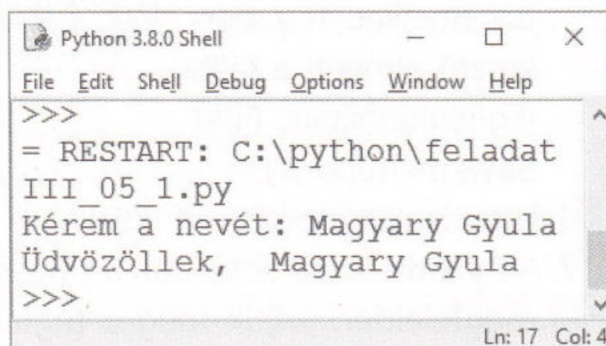
Jó lenne, ha a programjaink adatainak egy részét, például a téglalap oldalait a felhasználó adhatná meg anélkül, hogy át kellene szerkesztenie a programot. Erre a célra szolgálnak az adatbekérő utasítások.

20. mintafeladat – szöveges típusú adat bekérése, gyorsmentés

Kérjük be a felhasználó nevét, majd üdvözöljük a következő szöveggel: "Üdvözöllek, …" Ahol … a felhasználó neve. (Az idézőjeleket nem kell kitenni.)

1. Nyissunk egy új programot (**File/New File**).
2. Az adatbekérés utasítása az **input**. Az utasítás után zárójelpárt alkalmazunk. Illik kiíratni, mit is kérdezünk a felhasználótól. Ezt a zárójelen belül, idézőjelek között adhatjuk meg.
3. Az **input** utasítás elé írjuk annak a változónak a nevét, amiben tárolni akarjuk a bekért értéket. A programunkban ez most az *n*.
4. A kiíratás már ismerős lesz. A **print** utasítás zárójelei közt vesszővel elválasztjuk a megjelenítendő szöveget és a változót.
5. Gépeljük be a mellékelt programot, majd mentsük el **feladatIII_05_1** néven. (**File/Save As…**)
6. Futtassuk a programot (**F5**).
7. A **Python Shell** ablakban a *Kérem a nevét*: kiírás mögött villogni fog a kurzor. Oda írjuk be a nevünket, majd nyomjuk meg az **Enter** billentyűt. Ez lesz a továbbiakban mindig az adatbevitel módja a parancsértelmezőben.
8. A Python kiírja az Üdvözöllek, … szöveget.





Típusátalakítás, konverzió

Láttuk, hogy különböző adattípusok vannak. Az input utasítás karakterláncokat olvas be. A karakterláncokat azonban nem lehet összeszorozni vagy számként összeadni. Ezért a karakterlánc típusú értéket át kell alakítani egész vagy valós számmá. A karakterláncból egészet az **int**, valós számot a **float** utasítással lehet készíteni.

21. mintafeladat – adattípus átalakítása (karakterláncból egész)

Módosítsuk a téglalap kerületet és területet kiszámító programot úgy, hogy a felhasználó adhassa meg a program futása közben a téglalap oldalait.

1. Töltsük be a már elkészített **feladatIII_04** programot (**File/Open...**).
2. Mentsük el új néven, hogy ne írjuk majd véletlenül sem felül az előzőt (**File/Save As..., Fájl neve: feladatIII_05_2.**).
3. Sajnos egy olyan sor, hogy b = input ("Kérem a másik oldalt:") nem lenne jó. Ennek az az oka, hogy az input utasítás alapvetően karakterlánc típusú adatokat olvas be a billentyűzetről, így a Python a *b* változót is karakterlánc típusúra készíti. A karakterláncokat pedig nem lehet összeszorozni, sem számként összeadni. Ilyenkor típusátalakítást (típuskonverziót) kell végeznünk. Az *int* utasítás képes átalakítani egész számmá a karakterláncot. Így a sor helyesen így fog kinézni: b = *int*(input(" Kérem a másik oldalt: ")).
4. Javítsuk át az első két sort az ábrán látható módon.
5. Mentsük el a fájlt. Ilyenkor már nem kell neki új nevet vagy helyet adni, tehát elég, ha megnyomjuk a **Ctrl+S** billentyűket, és a szerkesztő elmenti a fájlt. (Egérimádóknak: **File/Save** menüpont.)
6. Futtassuk a programot (**F5**).
7. A **Python Shell** ablakban a kiírásnak megfelelően adjuk meg a téglalap oldalait, mindegyik után **Enter** billentyűt nyomva.
8. A mellékelthez hasonló ábrát kell látnunk.

feladatIII_05_2.py - C:\python\feladatIII_05_2.py (3.8.0)

File Edit Format Run Options Window Help

```
a = int(input("Kérem az egyik oldalt: "))
b = int(input("Kérem a másik oldalt: "))
k,t = 2 * (a + b), a * b
print(k,t)
```

Ln: 5 Col: 0

Python 3.8.0 Shell

File Edit Shell Debug Options Window Help

```
>>>
= RESTART: C:\python\feladat
III_05_2.py
Kérem az egyik oldalt: 35
Kérem a másik oldalt: 25
120 875
>>>
```

Ln: 22 Col: 4

9. Írjuk át az első két sort úgy, hogy az **int** helyén **float** legyen, tehát törteket is meg lehessen adni.
10. Mentsük el a fájlt. Nyomjuk meg a **Ctrl+S** billentyűket, és a szerkesztő elmenti a fájlt.
11. Futtassuk a programot (**F5**).
12. A Python Shell ablakban adjuk meg a téglalap oldalait, mindegyik után **Enter** billentyűt nyomva. Az értékek legyenek 3.5 és 2.5, tizedesponttal beírva. (Az eredmény: 12.0 és 8.75 lesz.)

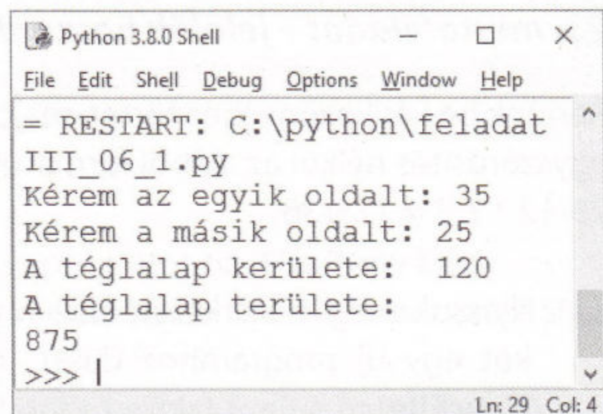
6. FORMÁZOTT KIÍRATÁS

Tagolt kiíratás

A print utasítás alapvetően úgy dolgozik, hogy a vesszővel elválasztott kiírandók közé szóközt tesz, a kiírás végén új sort kezd. Ezek azonban csak az alapbeállítások, amitől néha el kell majd térnünk. Bár az alábbi példát megoldhatnánk másképpen is, az eddigi tudásunk alapján, tágítsuk egy kicsit a látókörünket, mert később nem lehet majd elkerülni a tagolás beállítását.

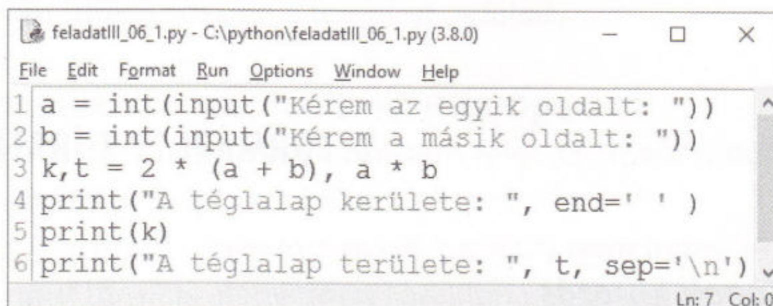
22. mintafeladat – tagolt kiíratás

Módosítsuk az előző programot úgy, hogy a mellékelt képen látható módon jelenjen meg az eredmény kijelzése, és külön print utasításban legyen "A téglalap kerülete:", valamint a 120, viszont egy print utasításban legyen "A téglalap területe:", és a 875.



1. Nyissuk meg a feladatIII_05_2programot (**File/Open...**).
2. Mentsük el **feladatIII_06_1** néven az új programot (**File/Save As...Fájl neve: feladatIII_06_1**).

3. Csak a program végén kell módosítanunk, a print utasításoknál. Az utasítás zárójelein belül, vesszővel elválasztva beállítható, hogy mi történjen a sor végén az **end** paraméter segítségével. Az end = ' ' azt jelenti, hogy szóközzel végződjön a sor (és ne új sorral). Így a print utasítás helyesen: print("A téglalap kerülete: ", end=' ').
4. Az utolsó két megjelenítendő dolgot egy print utasítással kell megjelenítenünk, de ott a szóköz helyett új sort szeretnénk elválasztó karakternek. Az elválasztó karakter paraméterének neve **sep**. Ugyanúgy kell használni, mint az endet használtuk. A sep='\n' azt jelenti, hogy a vesszővel elválasztott kiírandók közé új sor kerüljön. (\n = új sor) Így ez a print utasítás helyesen: print("A téglalap területe: ", t, sep='\n')
5. Végezzük el a szükséges módosításokat a mellékelt ábra szerint.
6. Mentsük a programot. (**Ctrl+S**)
7. Futtassuk a programot, és próbáljuk ki (**F5**).



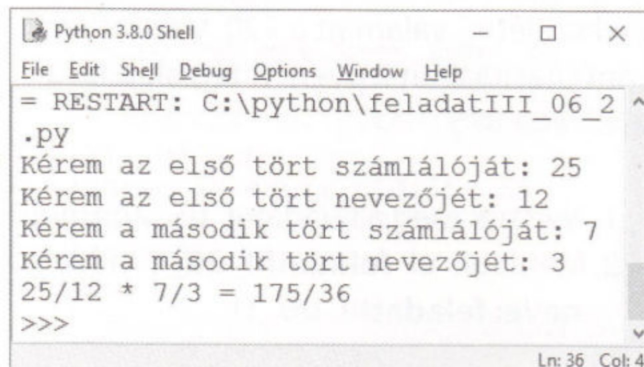
Jelölők használata

Bonyolultabb esetekben a formázott adatkiírást érdemes jelölőkkel (a jelölő neve angolul marker) megvalósítani. A jelölők százalékjellel kezdődnek, *beírhatók a szövegbe,* és a print utasítás egy későbbi részén elegendő megadni az értéküket. Általános célú jelölő a *%s*, leggyakrabban *karakterláncok* helyettesítésére használjuk. Az *egész számokat %d*-vel, a *valós számokat %f*-fel adjuk meg.

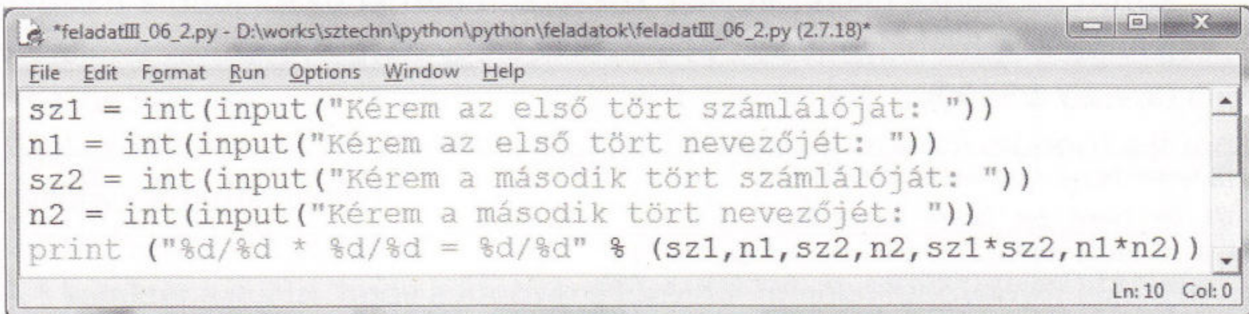
23. mintafeladat – jelölők használata

Kérjünk be két közönséges törtet, majd jelenítsük meg a törteket és a szorzatukat, egyszerűsítés nélkül az alábbi formátumban:
25/12 * 7/3 = 175/36.

1. Nyissuk meg a szerkesztőablakot egy új programhoz (**File/ New File**).
2. Kérjük be egyesével a számlálókat és nevezőket a már megtanult módon az input utasítással (1–4. sor).



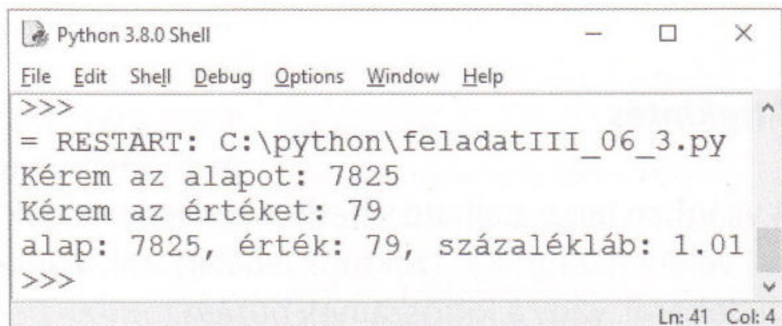
3. Az eredménysor megjelenítéséhez a print utasítást fogjuk használni. Az idézőjelek közötti részben, a számok helyére mindenhol írjuk a %d jelölőt. Így fog kinézni: "%d/%d * %d/%d = %d/%d".



4. A záró idézőjel után írjunk egy % jelet (még mindig a print utasítás zárójelei között vagyunk), majd zárójelek között soroljuk fel balról jobbra haladva a %d-k helyére helyettesítendő értékeket, egymástól vesszővel elválasztva: (sz1,n1,sz2,n2,sz1*sz2,n1*n2).
5. Mentsük a programot **feladatIII_06_2** néven (**File/Save As...**, **Fájl neve: feladatIII_06_2**.).
6. Futtassuk a programot, és próbáljuk ki (**F5**).

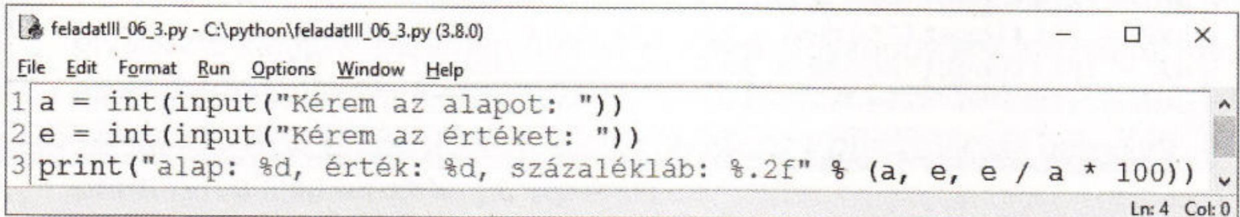
24. mintafeladat – tizedestört megjelenítése adott pontossággal

Készítsünk programot, ami bekér két egész számot (alap, érték), és kiszámítja, hogy az alapnak hány százaléka az érték. Az eredményt két tizedesjegy pontossággal jelenítsük meg a következő minta alapján:
alap: 7825, érték: 79,
százalékláb: 1,01



1. Nyissuk meg a szerkesztőablakot egy új programhoz (**File/New File**).
2. Kérjük be az alapot és az értéket a már megtanult módon az input utasítással (1–2. sor).
3. Az eredménysor megjelenítéséhez a print utasítást fogjuk használni. Az idézőjelek közötti részben, az egész számok helyére mindenhol írjuk a %d jelölőt. A százalékláb tizedestört lesz, így a helyén a %f jelölőt kell használni. A két tizedesjegy pontosságot pedig az f előtti .2 jelzi. Így fog kinézni: "alap: %d, érték: %d, százalékláb: %.2f".

4. A százaléklábat úgy kell kiszámolni, hogy az értéket elosztjuk az alappal, és megszorozzuk 100-zal: e / a * 100
5. A záró idézőjel után írjunk egy % jelet (még mindig a print utasítás zárójelei között vagyunk), majd zárójelek között soroljuk fel balról jobbra haladva a jelölők helyére helyettesítendő értékeket, egymástól vesszővel elválasztva: (a, e, e / a * 100).



6. Mentsük a programot **feladatIII_06_3** néven (**File/Save As...**, **Fájl neve: feladatIII_06_3**.).
7. Futtassuk a programot, és próbáljuk ki (**F5**).

7. VÉLETLENSZÁMOK

Áttekintés

A világban tapasztalható véletlenszerű jelenségek modellezéséhez a számítógéppel véletlenszámokat szoktunk előállíttatni. Ilyenek lehetnek például a dobókocka dobásai, vagy a lottószámok húzása.

A véletlenszámokat a számítógép egy periodikusan változó elektromos jelből veszi. Ennek megfelelően nem valódi véletlen az értéke, csak a felhasználó számára tűnik annak, aki nem rendelkezik azzal az információval, hol is tart éppen a jel.

Véletlen egész számot a randrange utasítással készíthetünk. Az utasítás után zárójelben meg kell adni egy számot. Így legkisebb véletlen egész *0* lesz, a legnagyobb *a megadott számnál eggyel kisebb* szám.

0 és 1 közé eső, véletlen valós számot a random utasítással készíthetünk. A keletkező valós szám nem lesz sosem 1.

Gyakran nem az utasítás által megadott tartományból van szükségünk véletlenszámokra. Ekkor a tartományt utólagos műveletekkel (például összeadással, szorzással) megváltoztatjuk.

Például a dobókocka 1 és 6 közötti véletlen egészeket dobhat, de randrange utasítás csak 0 és valami közötti számokat készít. 1-et úgy kaphatunk 0-ból, ha 1-et hozzáadunk. Ha 0...5 véletlenszámot készíttetünk a randrange(6) utasítással, elég lesz 1-et hozzáadni, és máris a kívánt 1...6 tartományt kapjuk.

A véletlenszámok készítése nem tartozik a Python alaputasításai közé. A programba be kell írni, milyen plusz utasításokat szeretnénk használni. Erre a célra szolgál az import utasítás. A véletlenszámok utasításai a random csoportba (függvénykönyvtárba) tartoznak.

```
from random import *
```

A * karakter azt jelzi, hogy a függvénykönyvtár *minden* függvényét elérhetővé tesszük a továbbiakban. Állhatna a * helyett a randrange, akkor a random() utasítást nem használhatnánk.

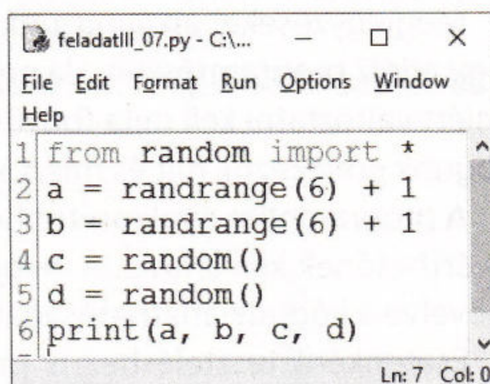
25. mintafeladat – véletlenszámok, függvénykönyvtár csatolása

Modellezzük a dobókockát. Dobjunk vele kétszer. Készítsünk két 0 és 1 közötti véletlen valós számot is.

1. Nyissuk meg a szerkesztőablakot egy új programhoz (**File/New File**).
2. Adjuk meg az 1. sorban, hogy a véletlennel kapcsolatos utasításokat is használni szeretnénk a továbbiakban, azaz csatoljuk a random függvénykönyvtárat.

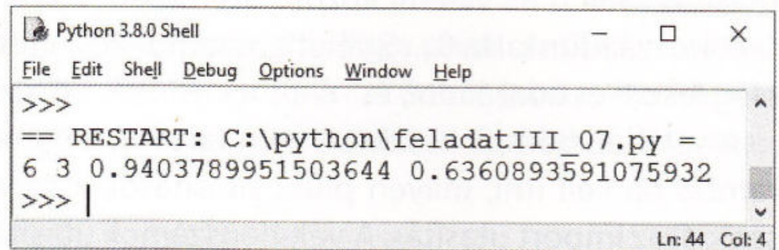
   ```
   from random import *
   ```

3. Véletlen egész számot a randrange utasítással készíthetünk. Paraméterként meg kell adnunk egy egész számot. Az utasítás 0 és a megadott szám között készít egy véletlen egészet, de a megadott számot már nem adhatja ki. A dobókocka hatféle értéket vehet fel, tehát a randrange(6) a helyes választás. Ez egy 0...5 egész számot fog adni eredményül, amiből úgy lesz 1...6, hogy hozzáadunk egyet (2. és 3. sor).



4. 0 és 1 közé eső véletlen valós számot a random utasítással készíthetünk a mellékelt ábrán látható módon (4. és 5. sor). A véletlenszám értéke nem lehet 1.
5. Írassuk ki print utasítással a kapott véletlenszámokat (6. sor).
6. Mentsük a programot feladatIII_07 néven (**File/Save As...**, **Fájl neve: feladatIII_07**).

7. Futtassuk a programot, és próbáljuk ki (**F5**).
8. A mellékelthez hasonló eredményt fogunk kapni. Az egyezés nem lesz számszerű, hiszen az értékek véletlenszerűek.



8. MEGJEGYZÉSEK A PROGRAMBAN

Fogalma

A megjegyzések olyan részletek a kódban, amit a fordító nem kísérel meg futtathatóvá alakítani. Csak a forrásállományban léteznek.

A megjegyzéseket kommenteknek is nevezik. A kód megjegyzésekkel történő ellátását pedig kommentezésnek.

Megjegyzéseket alkalmazunk emlékeztetőként magunknak, például, mire való egy adott programrészlet. Ha nem nyúlunk a programhoz egy évig, és utána valamiért változtatni kell rajta (bővíteni vagy javítani), már nagy valószínűséggel nem fogunk emlékezni, mit és miért kódoltunk.

A programokat gyakran többen fejlesztik. Az általunk írt kódnak más számára is érthetőnek kell lennie. A megjegyzés magyarázó szövegeket tartalmazhat, így növelve a kódunk átláthatóságát.

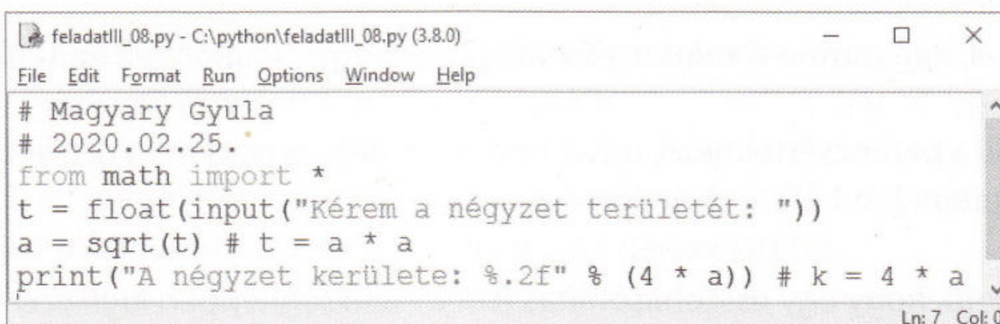
Esetenként teszteléshez is praktikus lehet a megjegyzések használata. Hiba behatárolása történhet úgy is, hogy bizonyos parancsokat megjegyzésbe helyezünk. Ha a programunk ezek nélkül jól működik, akkor a hibát a megjegyzésben kereshetjük.

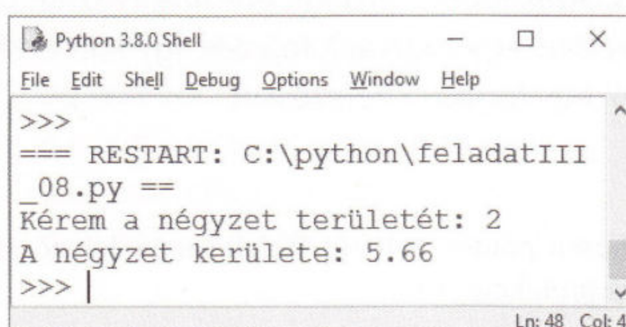
A Pythonban a megjegyzés jele a #. Ettől a karaktertől kezdve a sor végéig (Enter) tarthat a megjegyzésünk. Az itt található szavakat, betűket, kifejezéseket a Python nem értelmezi, így nem kapunk hibaüzenetet.

26. mintafeladat – megjegyzések és matematikai függvény használata

Kérjük be a felhasználótól egy négyzet területét, majd abból számoljuk ki a négyzet kerületét, amit két tizedesjegy pontossággal jelenítsünk meg. Megjegyzésként írjuk be a programba a négyzet területét és kerületét kiszámító képletet abba a sorba, ahol felhasználjuk, valamint a program elejére a nevünket, és külön sorba a program készítésének dátumát.

1. Nyissuk meg a szerkesztőablakot egy új programhoz (**File/New File**).
2. Írjuk be az első sorba a nevünket. Mivel ez megjegyzés, kezdjük a sort #-kal.
3. Írjuk be a második sorba az aktuális dátumot. Mivel ez is megjegyzés, kezdjük ismét a sort #-kal.
4. Adjuk meg a 3. sorban, hogy matematikai utasításokat is használni szeretnénk a továbbiakban, azaz csatoljuk a math függvénykönyvtárat.
   `from math import *`
5. Kérjük be a négyzet területét egy **input** utasítással, majd konvertáljuk valós számmá a **float** utasítás segítségével (4. sor).
6. Számítsuk ki a négyzet oldalát a területéből gyökvonással (**sqrt** függvény). Írjunk az utasítás mögé #-t, majd a négyzet területének kiszámításához szükséges képletet (t = a * a). (5. sor)
7. Írassuk ki a négyzet kerületét a **print** utasítás segítségével. Használjuk fel a két tizedes megjelenítéséhez a **%.2f** jelölőt. Az oldalhosszból (a) a kerületet 4-gyel történő szorzással kapjuk (4 * a) (6. sor).
8. A print utasítás lezárása után írjunk #-ot a megjegyzés nyitásához, majd írjuk be a kerület kiszámításának képletét (k = 4 * a) (6. sor).
9. Mentsük a programot **feladatIII_08** néven (**File/Save As…, Fájl neve:feladatIII_08**).
10. Futtassuk a programot, és próbáljuk ki (**F5**).





9. LOGIKAI VÁLTOZÓ, LOGIKAI ÉRTÉK

A logikai változók értéke igaz vagy hamis lehet. A Pythonban az előbbit a **True**, utóbbit a **False** szóval jelölik, kötelezően nagy kezdőbetűkkel.

Egy logikai változó egy eldöntendő kérdésre képes választ tárolni. Például: A "Beszerezhető vadgesztenye-facsemete." egy eldöntendő kérdés, hiszen vagy igaz (True), vagy hamis (False)[19].

27. mintafeladat – logikai értékek használata, képződése

1. Kísérletezzünk a *parancssorban,* tehát hívjuk elő a parancsértelmező ablakát. A >>> -tól jobbra, ahol a kurzor villog, írhatjuk be a parancsunkat. Ne felejtsük minden parancs után megnyomni az **Enter** billentyűt.
2. Legyen most az A változó igaz (tehát beszerezhető vadgesztenye-facsemete), a B változó értéke hamis. Ehhez adjuk ki a következő parancsokat:
```
A = True
B = False
```
3. Írassuk ki ellenőrzésképpen a két változó értékét:
```
print (A,B)
```
4. A *True False* eredmény jelenik meg, tehát eddig rendben vagyunk.
5. Most adjunk számértéket két változónak, mondjuk, c értéke legyen 50, d értéke 100. (c jelentheti a szavazatok számát arra, hogy egy adott területből park legyen, d arra, hogy parkoló.)
```
c,d = 50,100
```
6. Döntsük el, nagyobb-e c, mint d. (Tehát igaz-e, hogy: "Parkot kell építeni.")
```
print (c > d)
```
7. *False*-t ír ki a parancsértelmező, mivel nem nagyobb most c, mint d. (Nem kell parkot építeni.)

Megfigyelhető, hogy egy összehasonlítás (c > d) szükségképpen *logikai értéket* eredményez. Azokat a kifejezéseket, amik logikai értéket eredményeznek, *logikai kifejezés*eknek hívjuk. Például a c > d egy logikai kifejezés. Így van értelme egy logikai változót egyenlővé tenni egy logikai kifejezéssel.

---

19 Tekintsünk el a kérdés esetleg nem teljesen pontos voltától. Tehát beszerezhető akkor, amikor szükséges, nem szükséges hozzá protekció, stb.

8. Növeljük meg c értékét 100-zal. (További szavazatok érkeztek be, mind a parkra.)
```
c += 100
```
9. Tegyük a B változóba a c > d logikai kifejezés eredményét (igaz-e, hogy: "Parkot kell építeni."):
```
B = (c > d)
```
10. Írassuk ki ellenőrzésképpen a B változó értékét:
```
print (B)
```
11. A B változó értéke *True* lesz, hiszen c = 150, d = 100, így c > d *igaz*. (Az újabb szavazatokat figyelembe véve parkot kell építeni.)

28. mintafeladat – logikai műveletek: ÉS, NEM

A	B	A ÉS B
igaz	igaz	igaz
igaz	hamis	hamis
hamis	igaz	hamis
hamis	hamis	hamis

Az előző feladat folytatása.

A számok között értelmezni szoktunk műveleteket, mint összeadás, kivonás. Ehhez hasonlóan a logikai értékek között is vannak műveletek.

Például tegyük fel, hogy ha "Parkot kell építeni." és "Beszerezhető vadgesztenye-facsemete.", akkor vadgesztenyefákat fognak ültetni. Következtessük ki az előző logikai változókból, lesznek-e vadgesztenyefák az adott területen. Olyan műveletet kell végrehajtani, ami kizárólag akkor ad igaz értéket ("Vadgesztenyefák lesznek a területen."), ha mindkét kiindulási állítás igaz (A: "Beszerezhetők vadgesztenye-facsemeték." B: "Parkot kell építeni."). Ha bármelyik is hamis, az eredménynek hamisnak kell lennie (Ha nem kell parkot építeni, vagy nincsenek beszerezhető vadgesztenye-facsemeték, nem lesznek vadgesztenyefák a területen.). Erre a célra szolgál az ÉS művelet, amit a Pythonban az **and** jelöl.

12. Nézzük meg, mi lesz az A ÉS B művelet eredménye. (Igaz-e, hogy lesznek vadgesztenyefák a területen.) Írjuk be:
```
print (A and B)
```
13. Az eredmény *True*, tehát *igaz*. Így valóban teljesül, hogy igaz ÉS igaz = igaz.
14. Próbáljuk ki közvetlenül is. Írjuk be a parancssorba:
```
print (True and True)
```
15. Természetesen ismét *True*-t, tehát igazat kapunk eredményként.
16. Próbáljuk az összes többi lehetőséget, azaz írjuk be a parancssorba egyesével:
```
print (True and False)
print (False and True)
print (False and False)
```

17. Mindegyik hamist eredményez, tehát az ÉS művelet valóban megfelel a feladatunk célkitűzésének.

Az ÉS műveletet több kiindulási értékkel is van értelme végrehajtani. Például: Egy háromtagú bizottság dönt az új munkaerő felvételéről. Csak akkor veszik fel, ha mindhárman igent mondtak rá.

Egy másik logikai művelet a NEM. Egy változó értékét az ellenkezőjére állítja, tehát az igazból hamisat, a hamisból igazat készít.

A	NEM (A)
igaz	hamis
hamis	igaz

18. Fordítsuk ellenkezőjére az A változó értékét. (A szavazás lebonyolítása alatt megváltoztak a körülmények, és a vadgesztenye-facsemeték már nem beszerezhetők.) Ehhez írjuk be:
```
A = not(A)
```
19. Írassuk ki ellenőrzésképpen az A változó értékét:
```
print (A)
```
20. False-t, tehát hamisat fogunk kapni eredményül.

29. mintafeladat – VAGY művelet

Legyen egy kéttagú felvételi bizottság: Anna és Bea. Ha bármelyikük igent mond, felveszik a jelöltet. Tehát most egy olyan műveletet keresünk, ami csak akkor hamis, ha mindkét kiindulási adat hamis. Minden egyéb esetben igaz. Ez a VAGY művelet, amit a Pythonban az **or** jelöl.

A	B	A VAGY B
igaz	igaz	igaz
igaz	hamis	igaz
hamis	igaz	igaz
hamis	hamis	hamis

1. Kísérletezzünk a *parancssorban,* tehát hívjuk elő a parancsértelmező ablakát. A >>> -tól jobbra, ahol a kurzor villog, írhatjuk be a parancsunkat. Ne felejtsük minden parancs után megnyomni az **Enter** billentyűt.
2. Legyen most az A változó igaz (tehát Anna igent mondott), a B változó értéke hamis (Bea nemet mondott). Ehhez adjuk ki a következő parancsokat:
```
A = True
B = False
```
3. Nézzük meg, mi lesz az A VAGY B művelet eredménye. (Igaz-e, hogy felveszik a jelöltet.) Írjuk be:
```
print (A or B)
```

4. Az eredmény *True*, tehát *igaz*. Így valóban teljesül, hogy igaz VAGY hamis = igaz. (Felveszik a jelöltet.)
5. Próbáljuk ki közvetlenül is. Írjuk be a parancssorba:
```
print (True or False)
```
6. Természetesen ismét *True*-t, tehát igazat kapunk eredményként.
7. Próbáljuk az összes többi lehetőséget, azaz írjuk be a parancssorba egyesével:
```
print (True or True)
print (False or True)
print (False or False)
```
8. Az első kettő igazat, a harmadik hamisat eredményez, tehát a VAGY művelet valóban megfelel a feladatunk célkitűzésének.

Áttekintés

Logikai érték keletkezhet közvetlen megadással, például: A = True, vagy logikai kifejezés eredményeként, például: B = (c > d).

Az ÉS művelet[20] eredménye igaz, ha minden kiindulási érték igaz. Ellenkező esetben hamis.

A NEM[21] művelet a logikai változót az ellenkezőjére fordítja.

A VAGY[22] művelet eredménye hamis, ha minden kiindulási érték hamis. Ellenkező esetben igaz.

20 Másik elnevezése a diszjunkció.

21 Egyéb elnevezései: negáció, tagadás.

22 Másik elnevezése a konjunkció.

10. GYAKORLÓFELADATOK

1. Írjunk programot, ami bekér egy egész számot, 0-t ír ki, ha páros számot adunk meg, és 1-et, ha páratlant. Például 12 esetén 0 jelenik meg, 45 esetén 1.
2. Írjunk programot, ami bekér egy háromjegyű egész számot, és kiírja a szám százasait, tízeseit, egyeseit az alábbi formában: *A 329 egyesei: 9, tízesei: 2, százasai: 3.*
3. Írjunk programot, ami bekér egy kétjegyű egész számot, és kiírja a számot felcserélt számjegyekkel. Például, ha 75-öt adunk meg, 57-et ír ki.
4. Írjunk programot, ami bekér két közönséges törtet, majd összeadja őket (egyszerűsítés nélkül, közös nevezőnek a két nevező szorzatát véve), és az összeadás folyamatát kiírja az alábbi módon:
   11/3 + 4/5 = 55/15 + 12/15 = 67/15.
5. Kérjünk be a felhasználótól három osztályzatot, számítsuk ki az átlagukat, és jelenítsük meg egy tizedes pontossággal.
6. Modellezzünk pénzérme-feldobást, amit háromszor végezzünk el. Jelentse 0 a fejet, és 1 az írást. Az értékeket jelenítsük meg a képernyőn egymás alatt.
7. Sík padlófelületek burkolását Géza mester a felület nagyságánál 20%-kal több felületet lefedő burkolólap felhasználásával vállalja. Készítsünk programot, ami billentyűzetről beolvassa a téglalap alakú helyiség két oldalának hosszúságát, valamint a kiválasztott téglalap alakú burkolólap két oldalának hosszát, majd megadja, hány darab burkolólapra van szüksége Géza mesternek a munka elvégzéséhez. (Minden méret megadásakor méter legyen a mértékegység. Az egésznél pontosabb számokat használjunk.)
8. Próbáljuk ki mindegyik matematikai függvényt parancsmódban. (Lásd Összefoglalás: Változók/Számok kiegészítő műveletei)
9. Kérjük be a felhasználótól egy derékszögű háromszög két befogóját (a, b valós számok). Számítsuk ki az átfogót (c), és jelenítsük meg egy tizedes pontossággal az eredményt[23].
10. Készítsünk egy véletlenszerű időpontot (óra, perc, másodperc), amit ilyen formátumban írunk ki a képernyőre: 21:03:12, 2:12:04.
11. A matematikai inga lengési idejét a $T=2\pi\sqrt{(l/g)}$ képlettel kell kiszámítani, ahol T a lengési idő másodpercben, l a fonál hossza méterben, g = 9,81 m/s² a Földön. Készítsen programot, ami a képlet alapján meghatározza, hogy egy inga P perc alatt hány teljes lengést végez. A P percben megadott időt és az l fonálhosszt a billentyűzetről olvassa be. A teljes lengések számát a képernyőre írja ki.

---

23 Használjuk az $a^2 + b^2 = c^2$ összefüggést. (Pitagorasz-tétel)

12. Egy időben az orosz atomrakétákat csak az elnök (A), a hadügyminiszter (B) és a hadsereg főparancsnoka (C) együttesen tudta elindítani. (1-1-1 táska volt mindegyiknél.) Milyen logikai műveletsorral határozható meg, indítják-e a rakétákat vagy sem. Írjunk rá programot, és írassuk ki az összes esetet. (Lásd mellékelt ábra.)

```
>>>
= RESTART: C:/python/gyakIII_
10_12.py
True True True -> True
True True False -> False
True False True -> False
True False False -> False
False True True -> False
False True False -> False
False False True -> False
False False False -> False
>>>
```

13. Egy felvételi bizottság háromtagú: András (A), Béla (B), Csaba (C). Bármelyikük igent mond a jelöltre, felveszik az iskolába. Milyen logikai műveletsorral határozható meg, felveszik-e a jelöltet vagy sem. Írjunk rá programot, és írassuk ki az összes esetet, mint a 12. feladatnál.
14. Egy jelölt két fordulóban felvételizik a munkahelyre. Az első fordulóban Annával (A) és Beával (B) találkozik. Ha bármelyikük elégedett vele, a következő körbe kerül, ahol a leendő főnökével, Cecíliával (C) találkozik. Ha ő is igent mond, felveszik, ha nem, akkor nem. Milyen logikai műveletsorral határozható meg, felveszik-e a jelöltet vagy sem. Írjunk rá programot, és írassuk ki az összes esetet, mint a 12. feladatnál.

IV. ALAPVETŐ VEZÉRLÉSI SZERKEZETEK

1. SORRENDI VÉGREHAJTÁS

A programozásban általában nem mindegy az utasítások kiadásának sorrendje. Ha ugyanazokat az utasításokat más sorrendben adjuk ki, a program másképpen működik, vagy nem működik. Az 1. "menj előre 5 lépést" és a 2. "menj balra 2 lépést" utasításokkal és felcserélésükkel juthatunk ugyanabba a helyzetbe, ha egy nagy, üres helyen vagyunk. Azonban más lesz a helyzet, ha tőlünk balra 1 lépésre fal van.

A **vezérlési szerkezetek** alkalmazásával dönthetjük el, melyik utasítás melyiket kövesse a program futásakor. Tehát előfordul, hogy az utasítások nem a kódban elfoglalt sorrendjükben kerülnek végrehajtásra.

Az utasítások kódbeli sorrendjének megfelelő sorrendben történő végrehajtását sorrendi végrehajtásnak, szakkifejezéssel **szekvenciának** nevezzük.

A rövid programjainkban eddig is sorrendi végrehajtást követtünk, csak nem neveztük meg.

2. FELTÉTELES ELÁGAZÁS – ALAPOK

Bevezetés

Ha egy feltételtől függően a program többféleképpen folytatódhat, elágazást alkalmazunk[24].

Az elágazásokat **szelekcióknak** is nevezik.

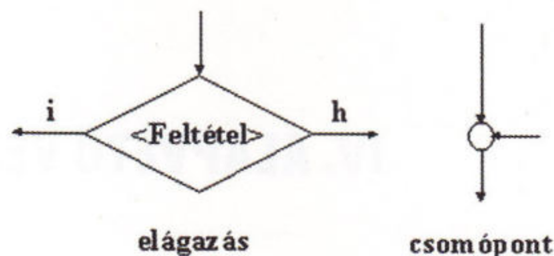
A **feltétel** egy **logikai érték**, tehát vagy **igaz (true)**, vagy **hamis (false)**. Többnyire összehasonlítások[25] formájában adjuk meg. Például az $a > 0$ feltételről egyértelműen eldönthető egy konkrét a esetén, hogy igaz vagy hamis.

---

24 Matematikai megfogalmazásban: az elágazásokat *esetszétválasztások* létrehozására használjuk.

25 Matematikai megfogalmazásban: relációk.

Folyamatábrában a feltételes elágazást egy csúcsán álló rombusszal szemléltetjük. A rombuszba írjuk a feltételt, a belőle induló nyilakra pedig ráírjuk az i, illetve h betűket. Ezzel jelezzük, hogy merre folytatódik a program **igaz ága**, illetve **hamis ága**. A **programágak** találkozhatnak a folyamatábra egy másik helyén, egy **csomópontban**. A csomópontot kis körrel jelöljük.



Elágazás egy irányban

Egyirányú elágazásról beszélhetünk, ha kizárólag igaz ágat hozunk létre a programban[26].

Az egyirányú elágazás szerkezete mondatszerű leírással a következő:

Ha <feltétel> akkor <utasítások>
Elágazás vége

Értelmezése: a *feltétel* teljesülése esetén az *akkor* mögötti utasítások végrehajtódnak. A program végrehajtása ezután az *Elágazás vége* után folytatódik. Ha a *feltétel* nem teljesül, a program végrehajtása közvetlenül az *Elágazás vége* után folytatódik.

Ha az *akkor* után csak egyetlen utasítás van, az *Elágazás vége* elmaradhat.

30. mintafeladat – elágazás egy irányban

Számítsuk ki egy bekért szám abszolútértékét. Írjuk meg az algoritmust, majd kódoljuk.

1. Először bekérjük a felhasználótól a számot az *a* változóba. (Algoritmusban és programban is az 1. sor.)
2. Ha a szám nem negatív, semmit nem kell vele tenni, csak kiíratni. Tehát a *feltételt* meg tudjuk fogalmazni. Akkor kell beavatkozni, azaz elágazni, ha *a<0* (2. sor).
3. A feltétel után a Pythonban : (kettőspont) karaktert kell elhelyezni. Ezt egyébként minden vezérlési szerkezetnél meg kell majd tenni. Tehát itt sem a feltétel, hanem az **if** miatt kell a kettőspont (2. sor).

26 Furcsa lehet a szóhasználat, hiszen hogyan is lehetne elágazni egyetlen irányban. Mégis, a szakirodalomban gyakran így nevezik ezt a szerkezetet, mivel nincs hamis ág.

4. Beírjuk a feltétel teljesülése esetén végrehajtandó utasítást, amivel *a* változó értékét a -1-szeresére változtatjuk (3. sor). A Pythonban ezt külön sorba kell írnunk (az algoritmus-leíróban nem kötelező). Szintén a Pythonban, kötelező a feltétel teljesülése esetén végrehajtandó utasításokat beljebb kezdeni 4 db szóközzel. Hogy ne kelljen nyomogatni négyszer a szóközt, a szerkesztőprogram a tabulátor gomb (Caps Lock feletti, 0 alatti oda-vissza nyíl) megnyomására beszúr 4 szóközt. Tehát nyomjuk meg az utasítás begépelése előtt a **Tab** billentyűt.
5. Írassuk ki az eredményt (4. sor).
6. Mentsük, majd futtassuk a programot. Teszteljük többféle számmal: negatívval, pozitívval, nullával.

A példa algoritmusa

```
(1) Be: a
(2) Ha a < 0 akkor
(3)            a := (-1)*a
(4) Ki: a
```

Elágazás kódolása a Pythonban

```
1 a = float(input("Kérem a számot: "))
2 if a < 0:
3     a = -1 * a
4 print("Az abszolútértéke: ", a)
```

Elágazás két irányban

Kétirányú elágazásról beszélünk, ha egy feltételtől függően a programot kétféleképpen kell folytatni.

Tehát a hamis ágat is használni akarjuk.

Mondatszerű leírása:

Ha <feltétel> akkor <utasítások1>
különben <utasítások2>
Elágazás vége

Értelmezése: a *feltétel* teljesülése esetén az *utasítások1* fognak végrehajtódni, ellenkező esetben az *utasítások2*. Ezután az utasítások végrehajtása az *Elágazás vége* után folytatódik.

31. mintafeladat – elágazás két irányban

Számítsuk ki egy bekért szám négyzetgyökét. Ha a szám negatív, írjuk ki, hogy nem végezhető el a művelet. Írjuk meg az algoritmust, majd kódoljuk.

1. Először bekérjük a felhasználótól a számot az *a* változóba (algoritmusban 1. sor).
2. Ha a szám negatív (a < 0), kiírjuk a képernyőre, hogy a művelet nem elvégezhető (2. sor).
3. Ha a szám pozitív vagy 0, azaz az elágazás különben ágában vagyunk, elvégezzük a gyökvonást, és kiírjuk az eredményt (3. sor).

```
(1) Be: a
(2) Ha a < 0  akkor Ki: "A művelet nem végezhető el."
(3)           különben Ki: négyzetgyök(a)
(4) Elágazás vége
```

4. A Python-kódban először elérhetővé tesszük az **sqrt** függvényt (1. sor).
5. Bekérjük a felhasználótól a számot, és konvertáljuk valós típusra (2. sor).
6. Létrehozzuk az elágazást az a < 0 feltétel alapján (3. sor).
7. A 4. sornak beljebb kell kezdődnie négy szóközzel a Python szintaktikai szabályai szerint, mivel ezeket az utasításokat csak az igaz ágban kell végrehajtani. A szerkesztő magától beljebb kezdi a sort. Ha valamiért mégsem, alkalmazzuk a **Tab** billentyűt.

```
1 from math import sqrt
2 a = float(input("Kérem a számot: "))
3 if a < 0:
4     print ("Nem végezhető el.")
5 else:
6     print (sqrt(a))
```

8. Kiíratjuk, hogy a művelet nem végezhető el.
9. A különben ágat az **else** utasítással kell jeleznünk. Utána kettőspontot kell írni. (5. sor)
10. A 6. sort ismét beljebb kell kezdeni, mert az elágazás hamis ágába fogjuk beírni a gyökvonást. Ismét alkalmazzuk a **Tab** billentyűt, ha szükséges, majd írassuk ki a szám négyzetgyökét.
11. Mentsük, majd futtassuk a programot. Teszteljük többféle számmal: negatívval, pozitívval, nullával.

Összehasonlítás

A következő táblázatból kiolvasható, milyen relációs jeleket használhatunk a feltételek megadásakor.

A Pythonban az = jel *a legyen egyenlőt* jelenti. Az egyenlőség mint összehasonlítás a == jelet kapta, hogy ne lehessen összekeverni a *legyen egyenlő* és az *egyenlő-e* kifejezéseket.

Szokatlan lehet még a *nem egyenlő* kódolása. Algoritmus-leíró nyelvben (és sok más programnyelvben, valamint az Excelben is) a logikus *kisebb vagy nagyobb* <> jellel kódoljuk. A Pythonban a *nem egyenlő* jele !=.

Összehasonlítás neve	Python	Algoritmus
nagyobb	a > b	a > b
kisebb	a < b	a < b
nagyobb vagy egyenlő	a >= b	a >= b
kisebb vagy egyenlő	a <= b	a <= b
egyenlő	a == b	a = b
nem egyenlő	a != b	a <> b

Feladatok

1. Kérjük be a felhasználótól a világítótoronyból látható hajók számát. Ha ez az érték nagyobb, mint három, akkor írjuk ki a képernyőre, hogy "Nehéz torony.", más esetben ne írjunk ki semmit.
2. Kérjünk be a felhasználótól két számot, amely egy közönséges tört számlálója és nevezője. Döntsük el, hogy az így bevitt tört felírható-e egész számként. Ha igen, írjuk ki az értékét egész számként, ha nem, írjuk ki: "Nem egész.".
3. Készítsünk programot, amely bekér a billentyűzetről egy 3 jegyű pozitív egész számot, és eldönti róla, hogy Armstrong-szám-e. A háromjegyű Armstrong-számokra igaz, hogy a számjegyei köbének összege megegyezik az eredeti számmal, például $371 = 3^3+7^3+1^3$. Az eredményt a képernyőre írassuk ki.
4. Készítsünk programot, ami véletlenszerűen fej vagy írást dob. A felhasználótól bekér egy tippet, és kiírja, hogy eltalálta-e vagy sem, amit a gép dobott.
5. Készítsünk programot, ami bekér egy 2011…2016 évszámot, és meghatározza, hogy az szökőév volt-e. A szökőévek 4-gyel oszthatók ebben az évszám-intervallumban.

(1) Be: a
(2) Elágazás
(3)   a > 0 esetén Ki: "pozitív"
(4)   a = 0 esetén Ki: "nulla"
(5)   egyébként Ki: "negatív"
(6) Elágazás vége

5. A Python-kódban először bekérjük a felhasználótól a számot, és konvertáljuk valós típusra (1. sor).
6. Létrehozzuk az elágazást az a > 0 feltétel alapján (2. sor).
7. A 3. sort négy szóközzel kezdi a szerkesztő (ha mégsem, akkor megnyomjuk a **Tab**ot), mert az elágazás igaz ágának beljebb kell kerülnie a Python szintaktikai szabályai szerint.
8. Kiíratjuk, hogy a szám pozitív.
9. Újabb feltétel (a == 0) megadásához az **elif** utasítást kell használni, a feltétel után kettősponttal (4. sor).
10. Az 5. sort beljebb kezdjük a szokásos módon, négy karakterrel. Kiíratjuk, hogy a szám nulla.
11. Az egyébként ágat az **else** utasítással kell jeleznünk. Utána kettőspontot kell írni (6. sor).
12. A 6. sort beljebb kezdjük négy karakterrel, majd írassuk ki, hogy a szám negatív.
13. Mentsük, majd futtassuk a programot. Teszteljük többféle számmal: negatívval, pozitívval, nullával.
14. Az egymásba ágyazott feltételekkel megoldott feladat is megnézhető a a programkód alsó részén..

```
1  a = float(input("Kérek egy számot: "))
2  if a > 0:
3      print("pozitív")
4  elif a == 0:
5      print("nulla")
6  else:
7      print("negatív")
```

```
8  #megoldás feltételek egymásba ágyazásával
9  if a > 0:
10     print("pozitív")
11 else:
12     if a == 0:
13         print("nulla")
14     else:
15         print("negatív")
```

Feladatok

1–2 km	500 Ft
3–5 km	700 Ft
6–10 km	900 Ft
11–20 km	1 400 Ft
21–30 km	2 000 Ft

1. Egy futár az egyes utakra az út hosszától függően kap fizetést az alábbi táblázatnak megfelelően. Kérjünk be a felhasználótól egy 1–30 km közötti távolságot, és határozzuk meg, hogy mekkora díjazás jár érte.
2. Kérjünk be a felhasználótól egy másodfokú egyenlet a, b, c valós együtthatóit. Határozzuk meg az egyenlet megoldását vagy megoldásait, ha vannak. Ha nincsenek, írjuk ki, nincs megoldás. (Feltételezhetjük, hogy *a* nem egyenlő 0-val, azaz valódi másodfokú egyenlettel van dolgunk. Ha szükséges, matematikai emlékeztető a lábjegyzetben[27]. Haladó szinten oldjuk meg a feladatot úgy, hogy a = 0 is engedélyezett.)
3. Készítsünk programot, ami egy dobókockát modellez. A kijelzést úgy oldjuk meg, hogy a karakteres képernyőn jelenjen meg a kocka dobott oldala a mellékelt ábrán látható módon. A program tudjon egyszer véletlenszerűen "dobni".

4. ELÁGAZÁS ÖSSZETETT FELTÉTELLEL (NOT, AND, OR)

Összetett feltételek

Az összetett feltételek részfeltételekből állnak. Például a "7-nél kisebb, és 0-nál nagyobb" egy ilyen összetett feltétel, ami két részfeltételből áll: "7-nél kisebb", valamint "0-nál nagyobb".

Az összetett feltételnek, akárcsak a részfeltételeknek, az értéke kizárólag igaz, illetve hamis lehet, tehát logikai kifejezésekről beszélhetünk.

A részfeltételekből összetett feltétel létrehozása azért fontos, mert az elágazás **if** utasítása mögött egyetlen logikai (igaz/hamis) értéket tud fogadni. Így, ha több feltételünk van, azt nem elegendő egyszerűen egymás mögé írni, hiszen az több logikai értéket eredményezne.

A logikai műveletek segítségével tudunk több logikai értékből egyet készíteni, tehát a részfeltételeket egy összetett feltétellé összekapcsolni.

---

27 Legyen $D= b^2-4ac$. Ha D pozitív, két megoldás van. Ha 0, egy megoldás van. Ha D negatív, nincs megoldás. Ha van megoldás, az $x_{1,2}=(-b\pm\sqrt{D})/2a$ képlettel számolható.

Logikai ÉS

A	B	A ∧ B
igaz	igaz	igaz
igaz	hamis	hamis
hamis	igaz	hamis
hamis	hamis	hamis

Logikai ÉS művelet esetén az eredmény csak akkor igaz, ha minden részfeltétel igaz. Minden egyéb esetben hamis.

Másképpen megfogalmazva: ha az első feltételen a másodikkal szigorítani akarunk, ÉS műveletet használunk.

Két részfeltétel (A és B) esetére érvényes a mellékelt táblázat (igazságtábla). Azonban fontos tudni, hogy kettőnél több részfeltételre is alkalmazható a művelet az előző szöveges meghatározás alapján.

Az ÉS művelet szabályos jele algoritmus-leíró nyelven: ^. Azonban gyakran az ÉS, illetve AND szavakat használják helyette, ha a ^ karakter beírása problémába ütközik. A Pythonban az **and** szót használjuk.

A Python képes értelmezni 1 < a < 8 típusú kétszeres összehasonlítást, ami megfelel az 1 < a and a < 8 összetett feltételnek.

33. mintafeladat – logikai ÉS

Egy vizsga szóbeli és írásbeli részből áll. Az írásbelin 100 pont, a szóbelin 50 pont érhető el. Akkor ment át valaki a vizsgán, ha pontszáma mindkét fordulóban meghaladta a 60%-ot. Írjunk programot, ami bekéri a felhasználótól a két elért pontszámot, és kiírja, átment-e a vizsgázó, vagy megbukott.

```
1 i = int(input("Pontszám az írásbelin: "))
2 sz = int(input("Pontszám a szóbelin: "))
3 iszaz = i / 100
4 szszaz = sz / 50
5 if iszaz > 0.6 and szszaz > 0.6:
6     print("Sikeres vizsga!")
7 else:
8     print("Sikertelen vizsga!")
```

1. Először bekérjük a felhasználótól a pontszámokat az *i* és *sz* változókba (1–2. sor).
2. Kiszámítjuk a százalékokat tizedestört alakban. Ehhez a kapott pontot elosztjuk az elérhető ponttal, külön az írásbeli *(iszaz),* külön a szóbeli *(szszaz)* esetén (3–4. sor).
3. Megfogalmazzuk a két részfeltételt. A kiszámított értékeknek nagyobbnak kell lenni, mint 60%, azaz 0,6: iszaz > 0.6, szszaz > 0.6
4. A vizsga csak mindkét feltétel együttes teljesülésekor lesz sikeres (igaz). Minden más esetben sikertelen (hamis). Ezért az ÉS műveletet kell használnunk (lásd igazságtábla), aminek utasítása az **and** (5. sor). Egy másik meg-

közelítésben: Megvan az írásbelire vonatkozó részfeltétel iszaz > 0.6. Ehhez képest a másik részfeltétellel *szigorítani* szeretnénk, mivel még a szóbelin *is* meg kell felelni. A szigorításnak az ÉS művelet felel meg.

5. A két részfeltétel együttes teljesülésekor kiírjuk, hogy a vizsga sikeres (6. sor).
6. Különben kiírjuk, hogy a vizsga sikertelen (7–8. sor).
7. Futtassuk a programot, és teszteljük.

Logikai VAGY

A	B	A ∨ B
igaz	igaz	igaz
igaz	hamis	igaz
hamis	igaz	igaz
hamis	hamis	hamis

Logikai VAGY művelet esetén az eredmény csak akkor igaz, ha *bármelyik* részfeltétel igaz.

Ebbe beletartozik, hogy egy vagy akár több részfeltétel teljesülése esetén is igaz.

Másképpen megfogalmazva a művelet **hamis, ha minden részfeltétel** hamis. Minden egyéb esetben igaz. Tehát, ha az első feltételen a másodikkal engedni, enyhíteni akarunk, VAGY műveletet használunk.

Két részfeltétel (A és B) esetére érvényes a mellékelt táblázat. Azonban fontos tudni, hogy kettőnél több részfeltételre is alkalmazható a művelet az előző szöveges meghatározás alapján.

A VAGY művelet szabályos jele algoritmus-leíró nyelven a **v** karakter. Azonban gyakran a VAGY, illetve OR szavakat használják helyette. A Pythonban az **or** utasítást használjuk.

34. mintafeladat – logikai VAGY

Egy írásbeli vizsgán három feladat közül választhattak a vizsgázók. Mindegyik feladat 50 pontos volt. Bármelyik feladatot sikerült megoldaniuk legalább 80%-osan, a vizsgájuk eredményes. (A feladatok pontszámai nem adódnak össze.) Kérjük be a három feladatra kapott pontjukat külön-külön, és írjuk ki, megfeleltek-e a vizsgán.

1. Először bekérjük a felhasználótól a pontszámokat az *e, m* és *h* változókba (1–3. sor).
2. Kiszámítjuk a százalékokat tizedestört alakban. Ehhez a kapott pontot elosztjuk az elérhető ponttal, minden feladat esetén külön (4–6. sor).
3. Megfogalmazzuk a részfeltételeket. A kiszámított értékeknek nagyobbnak vagy egyenlőnek kell lenni, mint 80%, azaz 0,8: eszaz >= 0.8, mszaz >= 0.8, hszaz >= 0.8.

4. A vizsga bármelyik részfeltétel teljesülésekor sikeres (igaz). Csak akkor sikertelen (hamis), ha mindegyik részfeltétel hamis. Ez a VAGY műveletnek felel meg (lásd igazságtábla). A VAGY műveletet kell használnunk, aminek utasítása az **or** (7. sor). Egy másik megközelítésben: Megvan az első feladatra vonatkozó részfeltétel eszaz >= 0.8. Ehhez képest a másik részfeltétellel *enyhíteni* szeretnénk, mivel nem kötelező megfelelni az első feladatban a 80%-nak. Az enyhítésnek a VAGY művelet felel meg.
5. A három részfeltétel bármelyikének teljesülésekor kiírjuk, hogy a vizsga sikeres (8. sor).
6. Különben kiírjuk, hogy a vizsga sikertelen (9–10. sor).
7. Futtassuk a programot, és teszteljük.

```
 1 e = int(input("Pontszám az első feladatra: "))
 2 m = int(input("Pontszám a második feladatra: "))
 3 h = int(input("Pontszám a harmadik feladatra: "))
 4 eszaz = e / 50
 5 mszaz = m / 50
 6 hszaz = h / 50
 7 if eszaz >= 0.8 or mszaz >= 0.8 or hszaz >= 0.8:
 8     print("Sikeres vizsga!")
 9 else:
10     print("Sikertelen vizsga!")
```

Logikai NEM

A logikai NEM művelet az igazból hamist, a hamisból igazat hoz létre.

A	¬ A
igaz	hamis
hamis	igaz

Jele algoritmus-leíró nyelven a ¬ jel. Ha problémába ütközik a beírása, NEM, illetve NOT szavakkal helyettesítik. A Pythonban a **not** utasítással kódoljuk.

Szinte lehetetlen hozzá jó és egyszerű feladatot kitalálni, hiszen a feltételt közvetlenül is megfordíthatjuk. Ha például a b > 1 ellentettjét keressük, azt megfogalmazhatjuk b <=1 ként is, nem szükséges a not(b > 1)-et választani. Bonyolultabb esetekben, amikor örülünk, ha egyféleképpen meg tudjuk fogalmazni helyesen a feltételt, jól fog jönni ez a művelet.

6. Az írásbeli eredményét zárójelbe kell tenni, mert az ÉS művelet magasabb rendű, mint a VAGY (9. sor). Ha nem tennénk zárójelbe, a szóbeli és a 3. feladat együttes teljesítése, vagy az 1., vagy a 2. írásbeli feladat teljesítése jelentené a sikeres vizsgát. Tehát helyesen:
   (eszaz > 0.6ormszaz > 0.6 or hszaz > 0.6) and szszaz > 0.6
7. Az előbb megszerkesztett feltétel teljesülésekor kiírjuk, hogy a vizsga sikeres (10. sor).
8. Különben kiírjuk, hogy a vizsga sikertelen (11–12. sor).
9. Futtassuk a programot, és teszteljük.
10. A logikai műveletek nélküli alternatív megoldás is megtekinthető az alábbiakban.

```
 1 e = int(input("1. feladat pontszáma: "))
 2 m = int(input("2. feladat pontszáma: "))
 3 h = int(input("3. feladat pontszáma: "))
 4 sz = int(input("Szóbeli pontszáma: "))
 5 eszaz = e / 50
 6 mszaz = m / 50
 7 hszaz = h / 50
 8 szszaz = sz / 50
 9 if (eszaz>0.6 or mszaz>0.6 or hszaz>0.6) and szszaz>0.6:
10     print("Sikeres vizsga!")
11 else:
12     print("Sikertelen vizsga!")
13 #megoldás logikai műveletek nélkül
14 if szszaz > 0.6:
15     if eszaz > 0.6:
16         print("Sikeres vizsga!")
17     elif mszaz > 0.6:
18         print("Sikeres vizsga!")
19     elif hszaz > 0.6:
20         print("Sikeres vizsga!")
21     else:
22         print("Sikertelen vizsga!")
23 else:
24     print("Sikertelen vizsga!")
```

Feladatok

1. Készítsünk programot, ami bekér három adatot egy telekről: adó, hosszúság, szélesség. A 15 m vagy annál keskenyebb, illetve a 25 m vagy annál rövidebb telkek tulajdonosai 20% adókedvezményben részesülnek. Írjuk ki az adó kedvezménnyel korrigált értékét.
2. Készítsünk programot, amely a billentyűzetről beolvassa három szakasz hosszát (a, b és c), és a képernyőre írja, hogy az adott szakaszokból szerkeszthető-e háromszög. Három szakaszból akkor és csak akkor szerkeszthető háromszög, ha bármely két oldal hosszának összege nagyobb, mint a harmadik oldal hossza[29], és minden oldal hossza nagyobb, mint 0.
3. Készítsünk programot, amely beolvassa egy kocka élét, valamint egy papírlap a és b oldalhosszait, majd meghatározza, hogy a kocka elkészíthető-e a papírból. (Ha szükséges segítség, a lábjegyzetben megtalálható[30].)
4. Készítsünk programot, ami bekér egy 1870...1930 évszámot, és meghatározza, hogy az szökőév volt-e. A szökőévek 4-gyel oszthatók, de 100-zal nem oszthatók ebben az évszám-intervallumban.

29 Háromszög-egyenlőtlenség tétele.

30 Ha a papír rövidebb oldala a kocka élének legalább háromszorosa, a hosszabb oldala a kocka élének legalább négyszerese, akkor a kocka elkészíthető.

5. KARAKTEREK ÉS KARAKTERLÁNCOK A FELTÉTELBEN

Karakterek összehasonlítása

A karakterek ASCII kódjuk, bájt nagyságú számok formájában tárolódnak. A karakterek ASCII kódját úgy határozták meg, hogy ábécésorrendben növekedjenek. A b betű kódja 98, a c betűé 99. Mivel 98 < 99, ezért 'b' < 'c'. A karakterek közvetlenül összehasonlíthatók a már megismert relációs jelekkel. Azaz nem szükséges átváltani ASCII kódra őket. Ha például szeretnénk tudni, hogy egy *kar* nevű változóban az ábécé nagybetűinek valamelyike fordul elő, alkalmazhatjuk a következő feltételt:

```
if 'A' <= kar and kar<='Z':
```

Megkaphatjuk egy karakter ASCII kódját az **ord** utasítással segítségével. Mivel az 'A' kódja 65, a 'Z' kódja 90, az előző feltétellel teljesen egyenértékű a következő:

```
if 65 <= ord(kar) and ord(kar)<= 90:
```

Sajnos ez a jó kis módszer ékezetes betűkre nem működik, mert az ASCII kódokat eredetileg az angol ábécéhez igazították. Így a későbbi bővítésnél az ékezetes betűk a 127 feletti ASCII kódok közé kerültek, tehát az 'a' után nem az 'á' jön, hanem a 'b'.

karakterek	ASCII kódok
0–9	48–57
A–Z	65–90
a–z	97–122

Az **ord** utasítás ellentettje a **chr**. A beírt szám ASCII kódjából előállítja a karaktert. Tehát a chr(65) az A betűt adja eredményül.

36. mintafeladat – karakterek összehasonlítása

Kérjünk be a felhasználótól egy karaktert, és döntsük el, az ábécé nagybetűje-e vagy sem.

1. Először bekérjük a felhasználótól a karaktert a *be* változóba (1. sor).
2. A második sorba beírjuk a feltételt: A *be* változóban található karakter legyen nagyobb vagy egyenlő 'A'-nál, de kisebb vagy egyenlő 'Z'-nél. Az áttekintésben

látottakhoz képest annyi az eltérés, hogy nem írtuk be az **and** utasítást, kihasználva, hogy a Python képes értelmezni a tartományokat is (2. sor).

3. Ha a feltétel teljesül, kiíratjuk, hogy nagybetű, különben, hogy nem nagybetű (3–5. sor).

```
1 be = input("Kérem a karaktert: ")
2 if 'A' <= be <= 'Z':
3     print("Nagybetű!")
4 else:
5     print("Nem nagybetű!")
```

37. mintafeladat – számból karakter

Írjunk programot, ami képez egy véletlenszerű ABC-123 típusú rendszámot.

```
 1 from random import randrange
 2 n1 = randrange(65,91)
 3 n2 = randrange(65,91)
 4 n3 = randrange(65,91)
 5 szj1 = randrange(0,10)
 6 szj2 = randrange(0,10)
 7 szj3 = randrange(0,10)
 8 b1 = chr(n1)
 9 b2 = chr(n2)
10 b3 = chr(n3)
11 print(b1, b2, b3, "-", szj1, szj2, szj3, sep="")
```

1. Elérhetővé tesszük a véletlenszámot készítő utasítást (1. sor).
2. Készítünk 3 db 65 és 90 közé eső véletlen egészet. Ez a tartomány felel meg az ábécé nagybetűinek (2–4. sor).
3. Készítünk 3 db véletlen számjegyet (0–9). Azért nem egy háromjegyűt készítünk, mert akkor mit kezdenénk a 013 és 007 végű rendszámokkal (5–7. sor).
4. A 2–4. sorban található számokból karaktereket képezünk a **chr** utasítás segítségével (8–10. sor).
5. Kiíratjuk egymás után a karaktereket és számjegyeket, ügyelve arra, hogy ne legyen közöttük elválasztójel: sep="" (11. sor).

38. mintafeladat – karakterből szám

Írjunk programot, ami bekér a felhasználótól egy 0–F karaktert, és kiírja a tízes számrendszerben neki megfelelő értéket (A = 10, B = 11, C = 12, D = 13, E = 14, F = 15).

```
1 hexa = input("Kérek egy 0 - F karaktert: ")
2 if hexa <= '9':
3     dec = ord(hexa)-48
4 else:
5     dec = ord(hexa)-55
6 print("Tizes számrendszerben: ", dec)
```

39. mintafeladat – karakterláncok összehasonlítása

Írjunk programot, ami bekér a felhasználótól két szót (csak angol karakterek!), és írjuk ki őket ábécésorrendben.

1. Bekérjük a két karakterláncot a változókba (1–2. sor).
2. Összehasonlítjuk a két változót. Ha az első a kisebb, akkor az van előbb az ábécésorban (3. sor).
3. Kiírjuk a két változó tartalmát a megfelelő sorrendben. Először az elsőt, aztán a másodikat (4. sor).
4. Ellenkező esetben (5. sor).
5. Kiírjuk a két változó tartalmát fordított sorrendben, azaz először a másodikat, aztán az elsőt (6. sor).

```
1 str1 = input("Kérem az első szót: ")
2 str2 = input("Kérem a második szót: ")
3 if str1 < str2:
4     print(str1, str2)
5 else:
6     print(str2, str1)
```

Feladatok

1. Döntsük el egy bekért karakterről, hogy számjegy-e. Eredményként a "Számjegy.", illetve "Nem számjegy." szöveg jelenjen meg.
2. Döntsük el egy bekért karakterről, hogy az angol ábécé betűje-e. Eredményként a "Betű.", illetve a "Nem betű." szöveg jelenjen meg.
3. Kérjünk be három szót. Írjuk ki őket ábécésorrendben.

6. TELJES LEFEDÉS ELVE

Egy programot általában nem egyszer tesztelünk. A változó (változók) többféle lehetséges értékével szoktuk kipróbálni a programot, minden alkalommal külön-külön futtatva. A tesztelendő programot és a változó egy lehetséges értékét **teszt-esetnek** nevezzük. Ha több változónk van, a teszteset létrehozásához mindegyik kap egy-egy konkrét értéket.

A **teljes lefedés elve** szerint a programunkat úgy kell tesztelni (azaz a teszteseteket úgy kell kiválasztani), hogy minden ág kipróbálására sor kerüljön. A mellékelt ábrán látható két, egymásba ágyazott elágazás például három tesztesettel lefedhető. Tegyük fel, hogy ezek a lehetőségek most: a = -5, a = 0, és a = 2.

```
1 if a == 0:
2     ki = "0"
3 else:
4     if a > 0:
5         ki = "+"
6     else:
7         ki = "-"
```

A teljes lefedés elve nem túl szigorú ellenőrzés. A képen látható tesztesetek nem fogják feltárni, ha például a 4. sorban valaki a > 1 feltételt írt be. Pedig a későbbiekben, ha a felhasználó például 1-et ad meg, azt fogja kiírni róla a program, hogy negatív. A teljes lefedés elve szerinti tesztelésből nem következik, hogy a programunk hibátlan. Azonban kitűnő szűrő. Bonyolult elágazásrendszerek esetén a fejlesztők gyakran elfelejtenek a szerintük ritkán előfordulónak látszó ágakba kódot írni, vagy megfelelő kódot írni. Például ennek ellenőrzésére a teljes lefedés elve tökéletes.

Nagyobb programoknál, amiknél a tesztelésről is dokumentáció készül, fel szokták tüntetni, milyen tesztadatokra hogyan reagált a program.

Feladatok

Végezzük el minél több feltételes elágazásrendszert tartalmazó, már elkészült feladatunk tesztelését a teljes lefedés elve szerint. Megjegyzésként írjuk be a programba, milyen értékkel teszteltünk, és mi lett az eredmény.

7. CIKLUSOK

A számítástechnikában gyakran előfordul, hogy egy műveletet vagy műveletsorozatot ismételni kell. Gondoljunk csak például egy gyártósorra egy autó-összeszerelő üzemben, ahol egy robotkar naponta számtalanszor megismétli ugyanazt a mozdulatsort.

A programrészletek ismétlését ciklusok segítségével oldjuk meg.

A ciklusokat szokás *iterációknak* is nevezni.

A ciklusoknak több típusa van. A legalapvetőbbek a számláló és a feltételes ciklusok.

8. SZÁMLÁLÓ CIKLUS

Számláló ciklust alkalmazunk, ha a program írásakor el tudjuk dönteni, hogy hányszor szükséges ismételni a programrészletet.

Például, ha ki akarjuk írni a képernyőre ötvenszer a "Helló világ!" szöveget, célszerű számláló ciklust alkalmaznunk.

Mondatszerű leírás és működés

A számláló ciklus kezdetén meg kell adnunk egy *ciklusváltozót*. A ciklusváltozónak mindig megszámlálható típusúnak kell lennie. Leggyakrabban az egész típusokat használjuk ilyen célra. A programozói szokások alapján a leggyakrabban az i betűt használjuk ciklusváltozóként. Természetesen bármilyen más, például beszédesebb változónevet is alkalmazhatunk. A ciklusváltozónak megadjuk a kezdőértékét, azaz honnan indul a számlálás.

A ciklus addig ismétel, míg a ciklusváltozó értéke el nem éri a végértéket.

A növekmény dönti el, hogy a számolás hányasával történik. Ennek leírásától eltekinthetünk, ha a növekmény 1.

A ciklusmagban található utasítás vagy utasítások ismétlődnek a ciklus minden egyes lefutásakor.

A mondatszerű leírásnál a ciklusmagot kicsivel beljebb kell kezdeni. Így jobban tagolódik a program, és áttekinthetőbb.

A példában a ciklus egytől ötvenig számol egyesével, tehát ötvenszer ismételi meg a ciklusmagban található utasítást. Mikor a ciklusváltozó értéke eléri a végértéket, a ciklusnak vége, és a program a ciklus utáni utasítással folytatódik.

A kezdőérték lehet nagyobb, mint a végérték, ekkor a ciklus visszafelé fog számlálni. Ilyenkor a növekmény negatív, és kötelező megadni.

ciklusváltozó kezdőérték végérték növekmény

```
Ciklus i:=1-től 50-ig 1-esével
    Ki: "Helló világ!"   ← ciklusmag
Ciklus vége

Ki: "Ciklus utáni utasítás."
```

Kódolás

A mellékelt táblázatban láthatjuk, hogyan kódoljuk a Pythonban a számláló ciklust.

A ciklus elején megadjuk a ciklusváltozót (i), majd a **range** utasításban a kezdő- és végértéket, valamint a növekményt. Ha a növekmény 1, elhagyhatjuk, csak két számot adunk meg a **range** utasítás zárójelében. Ha a kezdőérték 0, akkor azt is elhagyhatjuk, így csak egy számot adunk meg.

Ha a range-nek egy paramétere van, az mindig a végértéknél éppen nagyobb szám, másképpen fogalmazva, a ciklus lefutásainak száma. A kezdőérték ilyenkor 0, a növekmény 1.

Ha két paramétere van, akkor az a kezdőérték, és a végértéknél éppen nagyobb szám, ami már nem feltétlenül egyezik meg a ciklus lefutásainak számával. A növekmény ilyenkor 1.

Három paraméter esetén a kezdőérték, és a végértéknél éppen nagyobb szám után a növekményt is megadjuk.

A ciklus paramétereinek megadása után kettőspontot kell tenni, mert vezérlési szerkezet.

A ciklus végét külön nem jelezzük. A ciklusmag vége jelzi a ciklus végét. Az elágazásokhoz hasonlóan a ciklusmagot beljebb kezdjük négy szóközzel (**Tab** billentyű egyszeri megnyomása).

A számláló ciklus megadási módjai		
	Mondatszerű leírás	**Python**
Egyparaméteres, kezdőérték = 0, növekmény = 1	**Ciklus** i:=0-**tól** *100*-**ig** <Ciklusmag> **Ciklus vége**	**for** i in **range** (101): <Ciklusmag>
Kétparaméteres, kezdőértékkel, növekmény = 1	**Ciklus** i:=0-**tól** *100*-**ig** <Ciklusmag> **Ciklus vége**	**for** i in **range** (0,101): <Ciklusmag>
Háromparaméteres, növekménnyel	**Ciklus** i:=0-**tól** *100*-**ig** 10-**esével** <Ciklusmag> **Ciklus vége**	**for** i in **range** (0,101,10): <Ciklusmag>

40. mintafeladat – ismerkedés a számláló ciklussal

Írassuk ki egymás alá ötvenszer: "Helló világ!".

1. Legegyszerűbb módszer, ha a **for** ciklust egy paraméterrel használjuk. Ekkor a **range** mögött elegendő az ismétlések számát megadni, ami 50. Így a program két sorból fog állni. Az elsőben a **for** ciklus. A második a ciklusmag, amibe az ismétlendő kiíratás kerül:

```
for i in range(50):
    print("Helló világ!")
```

Módosítsuk úgy a programot, hogy a "Helló világ!" előtt megjelenjen, hányadik sort írja ki éppen a program.

A ciklusváltozót *(i)* érdemes felhasználnunk. Ezt kétféleképpen is megtehetjük célszerűen:

a) megoldás

2. Maradunk az előbbi ciklusnál. Akkor az *i* változó 0-tól 49-ig változik egyesével. Nekünk viszont 1-től 50-ig kell, mivel a hétköznapi szóhasználatban nincs 0. sor. Ezt úgy érhetjük el, hogy nem *i*-t írjuk ki, hanem *i + 1*-et. Így csak a második sort kell módosítani. Így fog kinézni a program:

```
for i in range(50):
    print("%d. Helló világ!" % (i + 1))
```

b) megoldás

3. A ciklusban kezdőértéket állítunk be, akkor nem kell 1-et hozzáadni a ciklusváltozóhoz. Ilyenkor a ciklus kétparaméteres alakját használjuk. Első paraméter a kezdőérték, ami most 1. Mivel 50-ig számolunk, és egyesével (hiszen nincs megadva növekmény), így az 51 lesz az az érték, amíg már nem kell elszámolni, tehát ez lesz a második paraméter. Ebben az esetben a program így fog kinézni:

```
for i in range(1,51):
    print("%d. Helló világ!" % (i))
```

Most módosítsuk a programot úgy, hogy a sorszámok 10-től 100-ig változzanak 5-ösével.

Ezt is megvalósíthatjuk célszerűen kétféleképpen.

a) megoldás

4. Mivel a növekményt megadjuk, a ciklust háromparaméteres formában írjuk be, így közvetlenül megjeleníthetjük az i-t, mert az tartalmazza a sorszámot. Első paraméter a kezdőérték: 10. Harmadik paraméter a növekmény, ami most 5. Mivel 100-ig számolunk 5-ösével, a 105 lesz az az érték, ameddig már nem kell elszámolni. Ez lesz a második paraméter.

```
for i in range(10,105,5):
    print("%d. Helló világ!" % (i))
```

b) megoldás

5. Az egyparaméteres alakot használjuk, és a műveletekkel előállítjuk a ciklusváltozóból a sorszámokat.
6. Először tudnunk kell, hányszor ismételjen a ciklus. (100 – 10) : 5 + 1 = 19. Tehát 19-szer. Ez lesz megadva a **range** utasítás egyetlen paramétereként. Így a ciklus 0-tól számlál 18-ig.
7. Mivel 5-ösével változik a kiírandó érték, de egyesével számolunk, kelleni fog egy 5-ös szorzó: *5 * i*.
8. Akkor az i így 0-tól 90-ig változna. Ebből úgy lesz 10-től 100-ig, ha hozzáadunk még 10-et. Így jön ki az *5 * i + 10* képlet.
9. A program tehát így fog kinézni:

```
for i in range(19):
    print("%d. Helló világ!" % (5 * i + 10))
```

Feladatok

1. Írassuk ki a képernyőre az első tíz pozitív páros számot!
2. Írassuk ki a képernyőre a 100, 90, 80, ..., 10, 0 számsorozatot!
3. Írassuk ki a képernyőre egy ciklussal:
   12345678901234567890123456789012345678901234567890
   Segítség a lábjegyzetben[32].
4. Írassuk ki a képernyőre két, egymástól független ciklussal:
   00000000011111111112222222222333333333344444444445
   12345678901234567890123456789012345678901234567890
5. Írassuk ki a képernyőre a 3. feladat számsorát függőlegesen egymás alá.
   1
   2
   ...
6. Írassuk ki a képernyőre a 4. feladat számsorát függőlegesen egymás alá.
   01
   02
   ...
   50
7. Írassuk ki a képernyőre a 3. és 5. feladat számsorait egyben.
   12345678901234567890123456789012345678901234567890
   1
   2
   ...
8. Írassuk ki a képernyőre a 4. és 6. feladat számsorait egyben.
   00000000011111111112222222222333333333344444444445
   12345678901234567890123456789012345678901234567890
   01
   02
   ...
   50

32 A ciklusváltozót osztjuk 10-zel, és az osztási maradékot írjuk ki.

9. CIKLUSOK EGYMÁSBA ÁGYAZÁSA

Használhatunk cikluson belül egy másik ciklust. Ezt nevezzük a ciklusok egymásba ágyazásának.

Két egymásba ágyazott ciklust alkalmazunk a téglalapszerű (mátrixos) adatelrendezések kiíratásához, bejárásához valamilyen művelet elvégzése érdekében.

Példaként készítsük el a szorzótáblát 1-től 15-ig (oldal-, illetve fejlécek nélkül az egyszerűség kedvéért).

Szorzótábla kiíratása

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
2 4 6 8 10 12 14 16 18 20 22 24 26 28 30
3 6 9 12 15 18 21 24 27 30 33 36 39 42 45
4 8 12 16 20 24 28 32 36 40 44 48 52 56 60
5 10 15 20 25 30 35 40 45 50 55 60 65 70 75
6 12 18 24 30 36 42 48 54 60 66 72 78 84 90
7 14 21 28 35 42 49 56 63 70 77 84 91 98 105
8 16 24 32 40 48 56 64 72 80 88 96 104 112 120
9 18 27 36 45 54 63 72 81 90 99 108 117 126 135
10 20 30 40 50 60 70 80 90 100 110 120 130 140 150
11 22 33 44 55 66 77 88 99 110 121 132 143 154 165
12 24 36 48 60 72 84 96 108 120 132 144 156 168 180
13 26 39 52 65 78 91 104 117 130 143 156 169 182 195
14 28 42 56 70 84 98 112 126 140 154 168 182 196 210
15 30 45 60 75 90 105 120 135 150 165 180 195 210 225
```

Szorzótábla algoritmusa

```
1)Ciklus i := 1 -től 15 -ig
2)   Ciklus j := 1 -től 15 -ig
3)        Ki: i*j,
4)        Ki: szóköz
5)   Ciklus vége
6)   Ki: új sor
7)Ciklus vége
```

Kezdjük a program elemzését a belső ciklus magjával[33]. A szorzótábla bármely száma két szám (i, j) szorzataként áll elő (3. sor). A kiírt szám után ki kell raknunk egy szóközt, mert egyébként a számok egyetlen számsorrá folynának össze (4. sor). Egy sor kiíratását végzi el a belső ciklus (2–5. sor), hiszen i egy adott értékét minden lehetséges j-vel, sorrendben megszorozza és kiírja. Egy sor kiíratását ismétli a külső ciklus (1–7. sor), így alakul ki az egész táblázat. Az algoritmus 6. sorában az "új sor"-ra azért van szükség, hogy kiíratáskor a szorzótábla minden sora a képernyőn is külön sorban jelenjen meg. Ennek hiányában a szorzótábla minden eleme megjelenne, csak egyetlen sorba kiírva.

```
1 for i in range(1, 16):
2     for j in range(1, 16):
3         print(i * j, end=" ")
4     print("")
```

Jobb oldalon látható a Python-kód. Az algoritmus 3. és 4. sorának a programkód 3. sora, míg az algoritmus 6. sorának a programkód 4. sora felel meg.

33 Az egymásba ágyazott ciklusokat tartalmazó programrészeket érdemes belülről kifelé haladva megérteni.

41. mintafeladat – ciklusok egymásba ágyazása

Készítsük el 1–15-ig a szorzótáblát (fejléc nélkül). Lásd a témakör bevezető ábráját.

1. Gépeljük be az elméletnél látható Python-kódot (1–4. sor). Próbáljuk ki.

Feladatok

1. Készítsünk a képernyőre egy 50 x 50 pontból álló rácsot.
   ....
   ....
   stb.
2. Készítsünk a képernyőre egy 50 soros, soronként eggyel növekvő számú pontrácsot.
   .
   ..
   ...
   stb.
3. Készítsünk a képernyőre egy 50 soros, soronként eggyel csökkenő számú pontrácsot.
   ....
   ....
   ....
   stb.
4. Készítsünk a képernyőre egy 50 x 50-es rácsot, ami sakktáblaszerűen felváltva B és W betűket (black, white) tartalmaz.
   BWBWBWBWBWBWBWBWBWBWBWBWBWBWBWBWBWBWBWBWBWBWBWBWBW
   WBWBWBWBWBWBWBWBWBWBWBWBWBWBWBWBWBWBWBWBWBWBWBWBWB
5. Bővítsük az előző feladatot a sor- és oszlopszámozás megjelenítésével.
   ```
      00000000011111111112222222222333333333344444444445
      12345678901234567890123456789012345678901234567890
   01 BWBWBWBWBWBWBWBWBWBWBWBWBWBWBWBWBWBWBWBWBWBWBWBWBW
   02 WBWBWBWBWBWBWBWBWBWBWBWBWBWBWBWBWBWBWBWBWBWBWBWBWB
   ```

10. FELTÉTELES CIKLUSOK

Bevezetés

Előfordulnak olyan esetek, amikor nem tervezhető előre a ciklus lépésszáma. Tételezzük fel, hogy a felhasználótól adatot kérünk be, majd annak helyességét ellenőrizzük, és csak helyes adatok esetén megy tovább a program, egyébként újra bekérjük az adatot. Ebben az esetben nem tudjuk megmondani előre, hányszor fog rossz adatot megadni a felhasználó.

Egy másik példa: legyen a feladat két szám (35 és 15) legkisebb közös többszörösének meghatározása. Alkalmazzuk ehhez a közvetlen módszert: képezzük mindkét szám többszöröseit (70, 105, illetve 30, 45, 60, 75, 105), majd a legkisebb közös többszörösnél (105) álljunk meg. A többszörösöket úgy képezzük, hogy az előző többszöröshöz hozzáadjuk az eredeti számot (+35, illetve +15). Nem fogunk "előreszaladni" valamelyik szám többszöröseinek meghatározásában, ha mindig a kisebb többszöröst növeljük, hiszen neki van esélye "utolérni" a másikat.

Ebben az esetben a felhasználó által megadott számok és a számok tulajdonságai[34] döntik el, hányszor kell ismételni a növelés műveletét, tehát az nem tervezhető előre.

Szükségünk lesz olyan ciklusokra, amikben az ismétlések száma *feltételhez* kötött. A példában addig kell növelni a kisebbik többszöröst, míg a két többszörös egyenlő nem lesz[35]. Ezeket a ciklusokat nevezzük összefoglaló néven **feltételes ciklusoknak**.

	C	D	
	35	15	
	35	30	+15
	35	45	+15
+35	70	45	
	70	60	+15
	70	75	+15
+35	105	75	
	105	90	+15
	105	105	+15

Minden ciklusnak van eleje és vége, ami közé a ciklusmag kerül. Attól függően, hogy hol vizsgáljuk a feltételt, beszélhetünk **elöl-, illetve hátultesztelő ciklusokról**.

A Python csak elöltesztelő ciklust ismer, így az egyébként praktikusan hátultesztelő ciklussal kódolandó algoritmusokat át kell írni elöltesztelőre.

---

34 prímtényezői

35 Az interneten rengeteg egyszerűbb példát találhatunk a feltételes ciklusokra. Számomra egyik sem volt meggyőző, mert mindegyiket egyszerűen helyettesíteni lehetett számláló ciklussal. S akkor ugye mi szükség a feltételes ciklusra, ha helyettesíthető valami egyszerűbbel. Ezért választottam egy bonyolultabb, de meggyőző példát, ahol a helyettesítés nem oldható meg a strukturált programozás keretein belül ésszerűen számláló ciklussal.

Elöltesztelő ciklus

Folytatva a bevezető példát, készítsük el a legkisebb közös többszörös meghatározásának algoritmusát.

Először bekérjük a felhasználótól a két számot, aminek a legkisebb közös többszörösét szeretnénk meghatározni (1. sor). Majd lemásoljuk az értéküket azokba a segédváltozókba (C, D), amikben a többszörösöket képezni fogjuk (2. sor). Erre azért van szükség, mert az eredeti számokat (A, B) is fel fogjuk használni (5–6. sor).

Mindig a kisebbik többszöröst növeljük (5–7. sor): Ha a D a kisebb, akkor a D-t (5. sor), különben a C-t (6. sor).

Ezt a növelést kell ismételni, azaz egy ciklus magjában elhelyezni (4–8. sor). A ciklus kezdetét a "Ciklus amíg" jelzi (4. sor). Az "amíg" mögött a ciklusban maradás feltétele található. Esetünkben a ciklust addig kell ismételni (azaz a ciklusban maradni), amíg a két többszörös nem egyenlő.

```
1) Be:  A, B
2) C := A
3) D := B
4) Ciklus amíg C ≠ D
5)   Ha C > D akkor D := D + B
6)           különben C := C + A
7)   Elágazás vége
8) Ciklus vége
9) Ki: C
```


Miután a ciklus befejeződik (8. sor), kiírjuk az egyik többszöröst tartalmazó változót (C), amiben az eredmény található (9. sor).

A feltétel megadására az elágazásoknál megismert szabályok érvényesek, azaz csak igaz/hamis értéket adó kifejezés lehet. Összetett feltételeket úgy adhatunk meg, hogy részfeltételeket logikai műveletekkel kapcsolunk össze.

Ha a felhasználó két egyforma számot ad meg, nincs szükség a növelésekre. A ciklusmag egyszer sem hajtódik végre. Általában is megfogalmazható, hogy elöltesztelő ciklust akkor használunk, ha a ciklusmagot nem biztos, hogy végre kell hajtani.

Végül fel kell hívnom a szóban felelő vagy vizsgázó tanulók figyelmét, hogy amikor a vizsgáztató azt kéri, írjon fel egy elöltesztelő ciklust általában, akkor a fenti példából kizárólag a "Ciklus amíg" és "Ciklus vége" sorokat kell feltüntetni, hiszen a többi csak egy konkrét példa része.

Kódolás: A ciklus elejét a **while** kulcsszóval jelöljük. Mögötte a *bennmaradás feltételét* kell megadni, majd egy kettőspontot. A ciklusmagot tagoltan kell beírni, azaz 4 szóközzel beljebb kezdeni (**Tab** billentyű használata).

42. mintafeladat – elöltesztelő ciklus

Kérjünk be két pozitív egész számot, és határozzuk meg a legkisebb közös többszörösüket a Bevezetésben olvasható algoritmus alapján.

```
 1 a = int(input("Kérem az egyik egészet: "))
 2 b = int(input("Kérem a másik egészet: "))
 3 c = a
 4 d = b
 5 while c != d:
 6     if c > d:
 7         d += b
 8     else:
 9         c += a
10 print("A legkisebb közös többszörös: ", c)
```

1. Kérjük be a két egész számot a felhasználótól *a* és *b* változókba (1–2. sor).
2. Az eredeti számokat (a, b) meg kell tartanunk, mert azokat fogjuk hozzáadni az utoljára előállított többszöröshöz, hogy a következő többszöröst kapjuk. Ezért szükségünk lesz még egy *c* és egy *d* változóra, ahol a többszörösöket tároljuk. Az első többszörös maga a szám, ezért c = a és d = b (3–4. sor).
3. Ha a c-ben tárolt többszörös nagyobb (6. sor), akkor a d-ben tárolt többszöröst kell növelni (7. sor).
4. A d += b jelentése: Növeld d-t b-vel, azaz d = d + b (7. sor).
5. Ha d-ben tárolt többszörös nagyobb (8. sor), akkor a c-ben tárolt többszöröst kell növelni (9. sor).
6. A 3–5. pontokban leírt folyamatot addig kell ismételni, míg a két többszörös közössé, tehát egyenlővé nem válik. Tehát feltételes ciklust kell alkalmazni (while), aminek a feltétele c != d.
7. Írassuk ki az eredményt, ami a c és d változóban is jelen lesz, hiszen közös többszörös. A programban a c-t választottuk.

Hátultesztelő ciklus

Olyan programot írunk, amiben osztályzatot kérünk be a felhasználótól. Elképzelhető, hogy a felhasználó hibás adatot ad meg. A bevitt adat ellenőrzésével szeretnénk ezt elkerülni. A hibás adat után *megismételjük* az adatbekérést.

Nem tudjuk előre eldönteni, hogy a felhasználó hányszor fog egymás után hibás adatot bevinni. Az ismétlések száma ismeretlen, tehát feltételes ciklust fogunk alkalmazni. Az adatbekérésnek mindenképpen le kell futnia egyszer, tehát nincs értelme a ciklus elejére helyezni a feltételt. Így hátultesztelő ciklust célszerű használni.

Az algoritmus felépítése: Először elkészítjük az adatbekérő részt (2–3. sor). Hátultesztelő ciklusba ágyazzuk (1–5. sor). Végül megadjuk a ciklusban maradás feltételét (4. sor). Addig kell ismételni az adatbekérést, míg a bekért szám egynél kisebb vagy ötnél nagyobb.

```
1) Ciklus
2)   Ki: „Kérek egy osztályzatot!"
3)   Be: A
4) amíg (A < 1)v(A > 5)
5) Ciklus vége
6) ...
```


Összefoglalva: a hátultesztelő ciklus egy olyan feltételes ciklus, ahol a feltételt a ciklus végén adjuk meg. Ennek megfelelően akkor használjuk, ha **nem tudjuk előre megadni az ismétlések számát**, de **egyszer mindenképpen végre kell hajtani a ciklusmagot**.

A feltétel megadására az elágazásoknál megismert szabályok érvényesek, azaz csak igaz/hamis értéket adó kifejezés lehet. Összetett feltételeket úgy adhatunk meg, hogy a részfeltételeket logikai műveletekkel kapcsoljuk össze.

A hátultesztelő ciklust gyakran alkalmazzuk ellenőrzött adatbevitelre, illetve a program újrakezdésére. Utóbbira példa a dobókockával történő dobás szimulálása többször egymás után bármely billentyű (kivétel a szóköz) megnyomásával. A programnak akkor van vége, mikor a felhasználó szóköz billentyűt nyom.

A Python nem használ hátultesztelő ciklust. A kódot úgy kell alakítani, hogy elöltesztelő ciklust használjon. Ehhez: 1. A ciklusmagban található utasításokat a ciklus elé is be kell írni. 2. A feltételt a ciklus elejére kell helyezni.

43. mintafeladat – hátultesztelő ciklusból elöltesztelő

Addig kérjünk be a felhasználótól egy egész számot, amíg az meg nem felel egy osztályzatnak, tehát 1...5 egész számnak.

1. Kérjük be a felhasználótól az egész számot az a változóba (1. sor).
2. Vizsgáljuk meg egy feltétellel, hogy a szám rossz-e. Rossz a szám, ha kisebb, mint 1 vagy nagyobb, mint 5: a < 1 or a > 5 (2. sor).
4. A feltételhez kötött ismétlés (ciklus) utasítása a while (2. sor).
5. A while-t tartalmazó sor végére :-ot rakunk, hiszen ez egy vezérlési szerkezet (2. sor).
5. Rossz szám esetén ismételni kell az adatbekérést. A ciklus magjában az 1. sort megismételjük (3. sor).
6. Végül a helyesen megadott számot kiírjuk (4. sor). Nyilván egy komolyabb programnak ez lenne a kezdete.

```
1 a = int(input("Kérek egy osztályzatot: "))
2 while a < 1 or a > 5:
3     a = int(input("Kérek egy osztályzatot: "))
4 print("Az ellenőrzött osztályzat: ",a)
```

Végtelen ciklusok

Megadhatunk olyan feltételt, hogy a ciklusból nem lép ki a program. (A mellékelt példában a 3 mindig kisebb, mint a 4.) Addig ismétli a ciklusmagot, míg a programot meg nem állítjuk külső beavatkozással. A szaknyelv erre mondja, hogy végtelen ciklusba került a program.

```
1 while 3 < 4:
2     print ("végtelen ciklus")
```

A példában egyszerűen átlátható, hogy a feltétel végtelen ciklust okoz. Egy összetett programban azonban ez nem szokott ennyire egyértelmű lenni.

Ez különösen zavaró, ha nem szándékosan tesszük, hanem valami hiba következménye. Ilyenkor az ablakban nem történik semmi változás, hiába próbáljuk használni a billentyűzetet vagy egeret az érintett ablakon, nem reagál. A program látszólag lefagyott. Valójában valamit tesz, csak annak esetleg nincs semmilyen jele számunkra.

Fejlesztés közben a lefagyott programot megállíthatjuk a parancsértelmező ablakban a **Shell/Restart Shell** menüvel, vagy a **Ctrl+F6** billentyűkombinációval. Végszükség esetén a Feladatkezelőben állíthatjuk le a teljes parancsértelmezőt.

44. mintafeladat – végtelen ciklus megállítása

Írjunk egy végtelen ciklust tartalmazó rövid programot. Futtassuk, majd állítsuk meg.

```
Python ...
File Edit Shell Debug Options
Window Help
végtelen ciklus
végtelen ciklus
végtelen ciklus
végtelen ciklus
================
======== RESTART:
Shell ==========
=============
>>>
Ln: 56000 Col: 4
```

1. Gépeljük be a Végtelen ciklusok fejezetben olvasható kétsoros programot.
2. Futtassuk (**F5**).
3. Menjünk át a parancsértelmező ablakba (**Python 3.8.0 Shell**).
4. Válasszuk a **Shell/Restart Shell** menüpontot, vagy nyomjuk meg a **Ctrl+F6** billentyűket.
5. Megáll a program futása, és megjelenik a prompt jel >>>. Ez mutatja, hogy az értelmező várja a következő parancsot.

Feladatok

2. feladat	
70	15
55	15
40	15
25	15
10	15
10	5
5	5

1. Bővítsük a legkisebb többszöröst kiszámító programot úgy, hogy előzőleg ellenőrizzük a felhasználó által megadott számokat, és csak akkor engedjük tovább a programot, ha 0-nál nagyobb számot adott meg.
2. Határozzuk meg két pozitív egész szám legnagyobb közös osztóját az alábbi algoritmussal. Ha a két szám egyenlő, akkor megvan a legnagyobb közös osztó. Ha nem egyenlők, a nagyobból kivonjuk a kisebbet, a kisebbet változatlanul leírjuk. (Lásd mellékelt ábra.)
3. Egészítsük ki a fenti programot úgy, hogy a számítást ismételten, több számpárral is legyen módunk elvégezni. A felhasználó a kilépési szándékát úgy jelezze, hogy valamelyik szám 0 vagy negatív. Egyéb ellenőrzés nem kell.
4. Írjunk programot, ami eldönti egy bekért pozitív egész számról, hogy prím-e. (Eratoszthenész szitája)
5. Írjunk programot, ami elvégzi egy pozitív egész szám prímtényezős felbontását.

11. ALPROGRAMOK

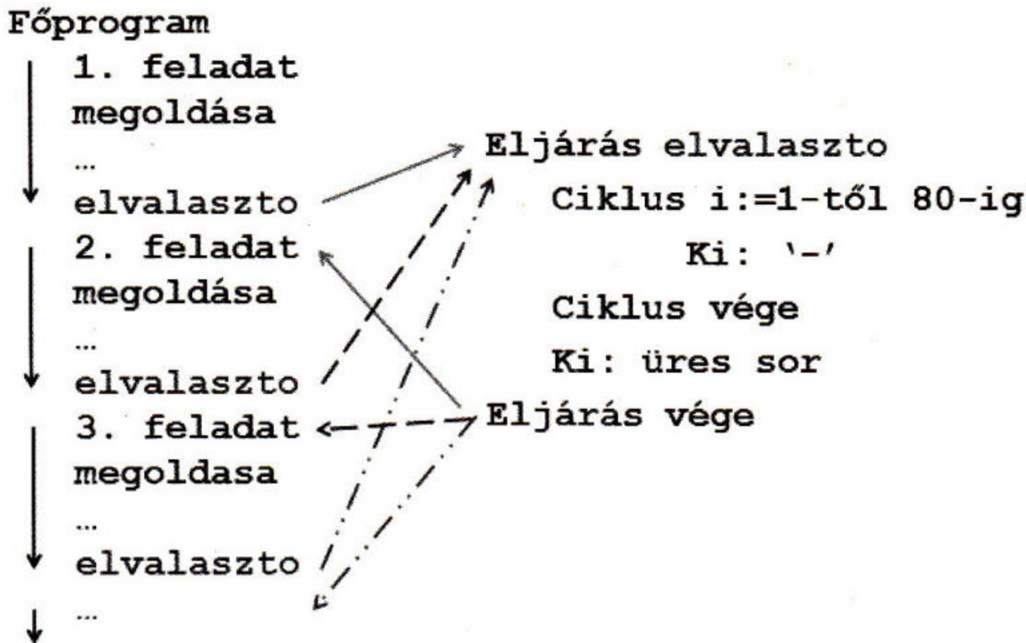
Az **eljárások** és **függvények** alprogramok (más néven részprogramok, szubrutinok).

Az eljárások és függvények segítségével programjainkat részekre bonthatjuk.

Az alprogramot megírjuk, majd névvel látjuk el. A főprogram különböző helyén már csak a nevével hivatkozunk rá, és végrehajtódik.

Az eljárások és függvények használatának egyik előnye a **kódismétlés elkerülése**. A programunk rövidebb, áttekinthetőbb lesz, és elég az alprogramban változtatni, ha a kód hibás, vagy másképpen szeretnénk működtetni a továbbiakban, tehát **egységesen módosítható**. További előny, hogy egy nagyobb program így részekre bontva, több emberrel, **csoportmunkában megoldható**.

Az eljárások és függvények használata megváltoztatja az utasítások alapvetően sorrendi végrehajtását. Az alprogramokon és a főprogramon belül megmarad a sorrendi végrehajtás.[36] A meghíváskor a program futása az alprogram első sorával folytatódik. Az alprogram befejeztével a főprogramban a meghívás utáni sorral folytatódik a végrehajtás.



Tanulmányozzuk a mellékelt ábrát. Legyen egy főprogramunk, ami sok feladatot old meg. A képernyőre kiírásnál az áttekinthetőség kedvéért szeretnénk a feladatokat vonalakkal (és egy üres sorral) elválasztani. Megírjuk tehát a jobb oldalon látható *elvalaszto* nevű eljárást, amit minden feladat végén meghívunk a főprogramból. Gondoljuk át, mennyivel hosszabb lenne a főprogram, ha az alprogramot a meghívás minden egyes helyére behelyettesítenénk. Ha úgy döntünk, hogy dupla vonalsort akarunk alkalmazni, akkor eljárást használva egy helyen kell hozzányúlni a programhoz. Ha nem alkalmaznánk, akkor minimum három helyen. Emellett gyorsabban célt érünk, ha felkérünk valakit az alprogram megírására, hiszen akkor ketten dolgozunk a feladaton. A főprogram és az eljárás megírható anélkül, hogy részletekbe menően tisztában kellene lenni egymás feladatával.

36 A program összes utasításának helyzetét egyszerre nézve változik meg a végrehajtás sorrendje.

12. ELJÁRÁSOK

Az eljárás egy alprogram. Hatását azon keresztül fejti ki, hogy valamilyen tevékenységet hajt végre. Például megjelenít valamit a képernyőn, vagy megváltoztat egy adatot egy fájlban vagy adatbázisban.

```
Eljárás alprogram2
...
Eljárás vége

Eljárás alprogram1
        alprogram2
...
Eljárás vége

Főprogram (main)
        alprogram1
```

Eljárás paraméter nélkül

Az eljárást a meghívás helye előtt kell létrehozni. Tehát, ha a főprogramban használni akarunk egy eljárást, azt a megelőző sorokban kell megírnunk. Ha az alprogram1 nevű eljárásban használni szeretnénk az alprogram2-t, akkor az alprogram2-t az alprogram1 előtt kell létrehoznunk. (Lásd mellékelt ábra.)

Az eljárás létrehozását a **def** kulcsszóval kezdjük. Ezt követi az eljárás neve *(border)*, majd egy üres zárójelpár, ha nincsenek átadandó paraméterek. Végül kettősponttal zárul a sor (1. sor). Az eljárás tartalmi részét egy szinttel beljebb kezdve adjuk meg (2–5. sor). Mint minden Python vezérlési szerkezet esetében, az eljárás végének sincs külön jele. A tagolás mutatja a végét.

```
1  def border():
2      print()
3      for i in range(80):
4          print("-", sep='', end='')
5      print('\n')
6
7  print("1. feladat")
8  border()
9  print("3. feladat")
10 border()
11 print("4. feladat")
12 border()
13 print("6. feladat")
```

Az eljárást a főprogramból a nevével és az üres zárójelpárral hívjuk meg (8., 10., 12. sorok).

Eljárás paraméterátadással

Az eljárások működését paraméterek segítségével befolyásolhatjuk. Az előző példa folytatásaként megadhatjuk, milyen legyen az elválasztójel, és milyen szélességű legyen. A paramétereket a név mögötti zárójelek között adjuk meg (7., 10., 12. sorok).

Az eljárás létrehozásakor a zárójelek között fel kell sorolni a paraméterként használt változók nevét (1. sor – n, s). Ezekbe fognak (alapesetben) behelyettesítődni a főprogram meghívásánál szintén a zárójelek között megadott értékek, mégpedig sorrendben. Az ábrán a 80 (7. és 12. sor), illetve 10 (10. sor) az n változóba, a * (7. és 12. sor), illetve – karakterek (10. sor) az s-be.

```
1  def border(n,s):
2      print()
3      for i in range(n):
4          print(s, sep='', end='')
5      print('\n')
6
7  border(80,'*')
8  print("1. feladat")
9  print("a) feladat")
10 border(10,"-")
11 print("b) feladat")
12 border(80,'*')
13 print("2. feladat")
```

Változók hatóköre

A Pythonban a modulban létrehozott változó csak a modulban használható fel, ha erről másképpen nem rendelkezünk. Hiába hivatkozunk rá a főprogramban, vagy egy másik modulban, hibaüzenetet kapunk. A mellékelt ábrán az *a* változót az eljáráson belül hozzuk létre (3. sor), csak ott használhatjuk fel. Ha a főprogramból megpróbáljuk kiíratni az értékét (11. sor), hibaüzenetet kapunk, mert a változó ott nem létezik.

```
1  def proc():
2      #global a
3      a = 5
4      print(b)
5      c = 3
6      print(c)
7
8  b = 0
9  c = 4
10 proc()
11 print(a)
```

A modulokban létrehozott változókat lokálisnak nevezik. Érvényességi körük (hatókörük) az a modul, ahol létrejöttek.

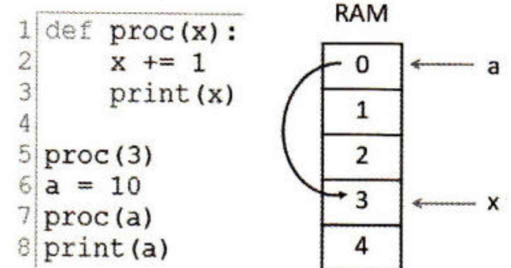
A főprogramban létrehozott változók viszont alapvetően a teljes programban elérhetők. Az ábrán létrehozzuk a *b* változót 0 értékkel (8. sor), amit aztán kiírathatunk az eljárásból (4. sor), mivel ott is elérhető.

Ha létezik ugyanolyan nevű lokális és globális változó, a lokális változó érhető el. Az ábrán ilyen a c változó, amit létrehozunk 4 értékkel a főprogramban (9. sor), és 3 értékkel az eljárásban (5. sor). Az eljáráson belüli kiíratás (11. sor) a 3-at, azaz a lokális változót fogja megjeleníteni. Ha az 5. sort kivesszük az eljárásból, például egy #-ot írunk elé, a c értékeként 4 fog megjelenni a kiíratáskor.

A főprogramban létrehozott változókat globálisnak nevezik. Érvényességi körük a teljes program, ha nincs egy azonos nevű lokális változó.

A Python lehetőséget biztosít rá, hogy létrehozzunk globális változót a modulon belül. Ilyenkor a változó elé a **global** kulcsszót kell írni. Ha az ábrán kivesszük a 2. sorban a #-ot, már nem kapunk hibaüzenetet, és kiírható a főprogramból az *a* változó értéke.

Érték szerinti paraméterátadás



Az előzőekben megismert paraméterátadást érték szerinti átadásnak nevezik, mert a főprogramból csak a változó értéke kerül át az alprogramba.

A program a futásakor lefoglal egy memóriarészt a főprogramban található változónak (az *a* változónak a 0. memóriarészt). Aztán a *proc* eljárás a meghívásakor lefoglal egy másik memóriarészt (az *x* lokális változónak a 3. memóriarészt). Az eljárás meghívása során a 0. memóriarészből áttöltődik a változó tartalma a 3.-ba. Így a két változó innentől kezdve teljesen független életet él. Az *x*-et módosítva az *a* nem változik meg.

Az eljárás befejeztével az eljáráshoz tartozó változó *(x)* megszűnik.

Az eljárást meghívhatjuk úgy is, hogy nem változó, hanem konkrét érték van megadva paraméterként (például az 5. sorban: *proc(3)*). Ebben az esetben az alprogram változójának lefoglalt memóriaterületre a konkrét számérték kerül (a példában 3).

Kövessük végig az ábrán látható példát.

1. Az 5. sorban indul a végrehajtás, mert ez a főprogram első sora. Meghívjuk a *proc* eljárást 3-as paraméterrel.
2. A program végrehajtása az 1. sortól folytatódik: A 3 az x változóba kerül. Az x értéke eggyel növekszik a 2. sorban, tehát 4 lesz. Ezt is fogja kiírni az eljárás a 3. sorból: *4*.
3. Mivel az eljárás végére értünk, a program a főprogram következő sorától (6.) folytatódik.
4. Az *a* változónak a 10 értéket adjuk (6. sor), majd meghívjuk ismét a proc eljárást (7. sor), most az *a* változót használva paraméterül.
5. A program végrehajtása ismét az 1. sortól folytatódik, ahol az *a* változóból átmásolódik a 10 az *x* változóba.
6. A 2. sorban az x értéke eggyel növekszik, így 11 lesz, amit meg is jelenítünk a képernyőn (3. sor).
7. A program végrehajtása a főprogram következő sorától (8.) folytatódik, ahol kiírjuk az a változó értékét. Mivel az a változó értéke nem változott, így 10 lesz a kiírt szám.

45. mintafeladat – eljárás készítése

```
 1 def pn(num):
 2     end = int(num/2)
 3     s = 1
 4     for i in range(2,end+1):
 5         if num % i == 0:
 6             s += i
 7     if s == num:
 8         print("Tökéletes szám")
 9     else:
10         print("Nem tökéletes szám")
11
12 for i in range(2,1001):
13     print(i, end=' ')
14     pn(i)
15 ob = int(input("Kérem a vizsgálandó számot: "))
16 pn(ob)
```

A matematikában tökéletes számoknak nevezik azokat a pozitív egész számokat, amiknek a számnál kisebb pozitív osztóinak összege magát az egész számot adja. Például a 28 tökéletes szám, mert 1+2+4+7+14 = 28. Írjunk eljárást, ami paraméterül kap egy egész számot, majd arról eldönti, hogy tökéletes szám-e vagy sem, és ezt megjeleníti a képernyőn. Ha kész az eljárás, írjunk programot az eljárást felhasználva, ami a 2...1000 számok mindegyikéről kiírja, tökéletes szám-e, majd bekér a felhasználótól egy egész számot, és kiírja, tökéletes szám-e.

1. Az eljárás létrehozásához a **def** utasítást használjuk. Mögötte megadjuk az eljárás nevét *(pn)*, és zárójelek között a paraméter nevét *(num)*, majd beírjuk a vezérlési szerkezetek végén kötelező kettőspontot. A paraméterben fog átadódni a vizsgált szám (1. sor).
2. Az eljárás "lelke" a 4–6. sorok, ahol egy ciklussal végigvizsgáljuk, hogy 2-től kezdve (4. sor – **range** első paramétere) mik a szám osztói. Azért 2-től kezdjük, mert az 1 biztosan osztója mindegyik számnak, így azt nem kell vizsgálni, elég hozzáadni. Ezért indul az osztók összege *(s)* 1-ről (3. sor).
3. Meghatározzuk, meddig kell a ciklusban számlálni. A vizsgált szám feléig elegendő elmenni, hiszen annál nagyobb valódi osztója nem lesz.[37] Az **int** függvény az eredmény egészrészét veszi, amire akkor van szükség, ha a vizsgált szám páratlan, mivel az kettővel osztva törtet adna eredményül, amit nem adhatunk meg a **range**-ben (2. sor).
4. A számláló ciklus végigpörgeti az *i* változóban a lehetséges osztókat (4. sor).
5. Ha a vizsgált szám *(num)* osztható az épp aktuális lehetséges osztóval *(i)*, mert az osztási maradék 0 (5. sor), az eddigi osztók összegét megnöveli az aktuális osztóval (6. sor).

---

37 Igazából elég a vizsgált szám négyzetgyökéig próbálkozni, de ennek belátásától most eltekintünk.

6. Ha az osztók összege *(s)* megegyezik a vizsgált számmal *(num)* (7. sor), akkor kiíratjuk, hogy "Tökéletes szám" (8. sor).
7. Különben (9. sor) kiíratjuk, hogy "Nem tökéletes szám" (10. sor).
8. A főprogramban indítunk egy ciklust, ami a 2-től 1000-ig számlál (12. sor).
9. A ciklus minden egyes lefutásakor kiíratjuk az aktuális számot (13. sor), majd meghívjuk a *pn* eljárást, ami eldönti és kiírja, tökéletes számról van-e szó (14. sor).
10. Bekérjük a vizsgálandó számot a felhasználótól az *ob* változóba (15. sor).
11. Meghívjuk a *pn* eljárást a vizsgálandó számra, hogy megvizsgálja és kiírja, tökéletes számról van-e szó.

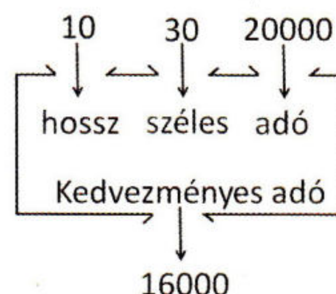
Feladatok

1. Készítsünk eljárást, ami paraméterül kap három egész számot, ami megfelel egy óra, perc és másodperc értéknek, majd kijelzi az időt 21:03:12 (egy másik példa: 4:11:07) formátumban. A kipróbálásához írjunk olyan főprogramot, ami bekéri a felhasználótól a három adatot, és meghívja az eljárást.
2. Készítsünk eljárást, ami paraméterül egy számot kap, ami egy másodpercben mért idő. Ezt kell átváltani óra, perc, másodpercre, majd megjelenítenie 21:03:12 (egy másik példa: 4:11:07) formátumban. Használjuk fel az előző pontban megírt eljárást. A kipróbálásához írjunk olyan főprogramot, ami bekéri a felhasználótól a másodperc adatot, és meghívja az eljárást.
3. A matematikában barátságos számoknak nevezzük azokat a pozitív egész számpárokat, amelyekre teljesül, hogy az egyik szám felírható a másik szám (saját magánál kisebb) osztóinak összegeként, és fordítva. Például a 220 és a 284 barátságos számpár, mert 220 osztói: 1 + 2 + 4 + 5 + 10 + 11 + 20 + 22 + 44 + 55 + 110 = 284, illetve 284 osztói: 1 + 2 + 4 + 71 + 142 = 220. Írjunk eljárást, ami a paraméteréül kapott két számról eldönti, hogy barátságos számpárt képeznek-e, és ezt ki is írja a képernyőre. Teszteléshez további ilyen számpárok: (1184; 1210), (2620;2974), (5020;5564).
4. Készítsünk eljárást, ami kiír a képernyőre egy véletlenszerű számjegyvariációt egy "kombinációs zárhoz" (pl. 34512, vagy 043). A számjegyek számát a paraméterben lehessen meghatározni. A kipróbálásához írjunk olyan főprogramot, ami bekéri a felhasználótól a számjegyvariáció hosszát, és meghívja az eljárást.

13. FÜGGVÉNYEK

A függvények olyan eljárások, amik egyetlen értéket *visszaadnak* a főprogramnak.

Úgy szokták megfogalmazni, hogy a függvénynek visszatérési értéke van. Természetesen a függvény is elvégezheti mindazokat a feladatokat, amiket az eljárás. Azonban különlegességét a visszatérési érték adja.



Szokás a függvényeket mint átalakító gépeket szemléltetni, hasonlóan az alsó tagozatos "mit csinál a gép" típusú feladatokhoz. A gép neve, ami szerencsés esetben utal a feladatára (pl. húsdaráló), lehet a függvény neve. Amit a gépbe rakunk, az a bemenő paraméter (pl. hús). Ami kijön a gépből (darált hús), az eredmény, a visszatérési érték.

```
1) Függvény kedv(h:egész, sz:egész, a:valós) : valós
2)     Ha h<=25 vagy sz<=15 akkor a:=0,8*a
3)     kedv:=a
4) Függvény vége
5) Főprogram
6)     a:=kedv(10,30,20000)
7)     Ki: a
8)     Ki: kedv(20,30,30000)
9)
10)    kedvez(20,25,10000)
```

Visszatérési érték típusa

Mondatszerű leírásnál[38] a függvényen belül a függvény nevével (mindenféle paraméter nélkül) jelöljük meg a visszatérési értéket (3. sor).

A visszatérési érték típusát mondatszerű leírásban a paraméterlista után írjuk :-tal elválasztva (1. sor).

A főprogramban a visszatérési értékkel valamit kezdeni kell: kiíratni (8. sor), egy változóba rakni (6. sor), egy másik függvénnyel vagy eljárással felhasználni, stb. Értelmetlen, és hibaüzenetet okoz önmagában a függvényt használni utasításként (10. sor), hiszen a visszatérési érték létrejön, azzal valaminek történni kell. Olyan lenne, mintha a (húsdaráló) gépünk kimeneti nyílását behegesztenénk.

A függvények kódolása alig tér el az eljárásokétól. A visszatérési értéket a **return** kulcsszó mögé kell írni.

A függvény csak egyetlen értéket adhat vissza.

38 És pár programnyelvben is, mint például a Pascal, de a Pythonban nem!

46. mintafeladat – függvény készítése

Az önkormányzat a 15 m vagy annál keskenyebb, illetve 25 m vagy annál rövidebb telkekre 20% kedvezményt ad az adóból. Írjunk függvényt, ami megkapja a telek szélességét, hosszúságát, és a kedvezmény nélküli adót, visszatérési értékként azt az adót adja, amiben már figyelembe vettük a kedvezményt is, ha volt. Ha nem jár kedvezmény, akkor az eredeti adót adja vissza.

```
1 def red(t, w, l):
2     if l<=25 or w<=15:
3         t = 0.8 * t
4     return t
5
6 tax = int(input("Kérem a kedvezmény nélküli adót: "))
7 length = int(input("Kérem a telek hosszát: "))
8 width = int(input("Kérem a telek szélességét: "))
9 print("A kedvezményes adó: ", red(tax,width,length))
```

1. Az eljárás létrehozásához a **def** utasítást használjuk. Mögötte megadjuk a függvény nevét (*red*), és zárójelek között a paraméterek nevét (*t* – eredeti adó, *w* – szélesség, *l* – hosszúság), majd beírjuk a vezérlési szerkezetek végén kötelező kettőspontot (1. sor).
2. Ha a hosszúság *(l)* kisebb vagy egyenlő 25, vagy (**or**) a szélesség kisebb vagy egyenlő 15 (2. sor), vesszük az eredeti adó *(t)* 80%-át (3. sor). (Ha az eredeti adó 100%, és abból 20% a kedvezmény, akkor marad 80%, amit be kell fizetni.)
3. A **return** kulcsszó mögött megadjuk a visszatérési értéket, ami a kedvezményes adó *(t)*.
4. A kipróbáláshoz bekérjük a főprogramban (6–9. sor) a szükséges adatokat (6–8. sor).
5. A **print** utasításnak arra a helyére, ahová a kiírandó változót szoktuk írni, most a függvény meghívása kerül (red...). Miután a függvény lefutott, a visszatérési értéke, azaz a kedvezményes adó a függvény helyére kerül vissza, így a print utasítás azt fogja kiírni (9. sor).

Feladatok

1. Készítsen függvényt *hetnapja* néven, amely a paraméterként megadott dátumhoz (hónap, nap) megadja, hogy az a hét melyik napjára esik (hétfő, kedd…). Tudjuk, hogy az adott év nem volt szökőév, továbbá azt is, hogy január elseje hétfőre esett. Használja az alábbi algoritmust, ahol a tömbök indexelése 0-val kezdődik.

```
Függvény hetnapja(honap:egesz, nap:egesz): szöveg
    napnev[ ]:= ("vasarnap", "hetfo", "kedd", "szerda", "csutortok", "pentek", "szombat")
    napszam[ ]:= (0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 335)
    napsorszam:= (napszam[honap-1]+nap) MOD 7
    hetnapja:= napnev[napsorszam]
Függvény vége
```

2. N természetes szám faktoriálisa alatt 1-től N-ig a természetes számok szorzatát értjük. Írjunk függvényt, ami paraméterként megkap egy természetes számot, és visszaadja a faktoriálisát. (0 faktoriálisa 1, 1 faktoriálisa 1, 2 faktoriálisa 2, 3 faktoriálisa 6, 4 faktoriálisa 24)
3. A Fibonacci-számsorozat első és második eleme 1. A következő elemet mindig úgy kapjuk, hogy az előző két elemet összeadjuk: 1, 1, 2, 3, 5, 8, 13, ... Készítsünk függvényt, ami meghatározza a bekért sorszámú Fibonacci-számot. Például: ha 7-et adunk meg, 13-at ad vissza.
4. A futár az egyes utakra az út hosszától függően kap fizetést a mellékelt táblázatnak megfelelően. Írjunk függvényt, ami a megadott út hosszának függvényében meghatározza a díjazás mértékét.

1–2 km	500 Ft
3–5 km	700 Ft
6–10 km	900 Ft
11–20 km	1400 Ft
21–30 km	2000 Ft

14. REKURZIÓ

Fogalma

A rekurzív szó jelentése önmagát tartalmazó, önmagára hivatkozó. Ennek alapján a rekurzió alatt olyan eljárásokat, függvényeket értünk, amik önmagukra hivatkoznak.

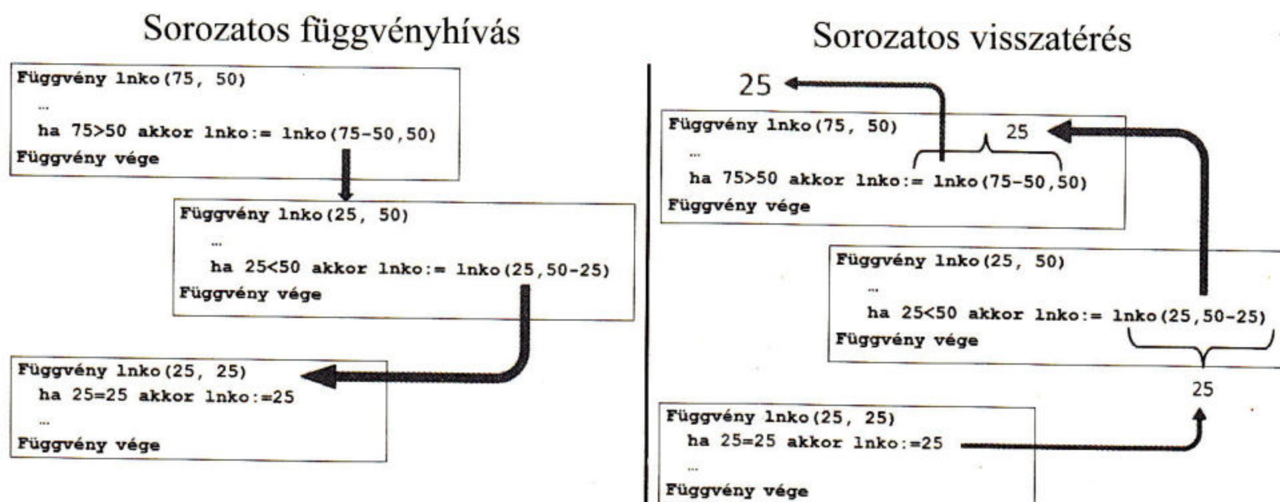
Bár a szakirodalom zömében az előző meghatározás elterjedt valamilyen formában, azért ezt az önmagukat kicsit pontosítanám: *önmaguk egy másik példányára.*

Tanulmányozzuk a mellékelt ábrát, ahol rekurzív algoritmussal keressük a legnagyobb közös osztót. Kövessük nyomon a működését. A 3. és 4. sor végén látható, hogy a függvény önmaga egy újabb példányát hívja meg más bemeneti értékekkel (a program ott folytatódik).

Működése

Nézzük meg a rekurzió működését az előbb látott példán keresztül.

Keressük a 75 és az 50 legnagyobb közös osztóját, így a függvény első meghívásakor ez a két paraméter. Az elágazás három ága közül az utolsó feltétele teljesül (4. sor). Ebben az ágban folytatódik a vezérlés, és újabb függvényhívás következik 75-50 és 50 paraméterekkel. Létrejön a függvény egy újabb példánya, ahol a középső ág feltétele teljesül, így ebben az ágban folytatódik a vezérlés, ahol újabb függvényhívás következik 25 és 50-25 paraméterekkel. A függvény immár harmadik példányának első sorában igaz a feltétel, tehát meg is van a visszatérési érték, a 25, és nincs további függvényhívás.



A függvény harmadik példányát a második példány hozta létre, így a harmadik példány visszatérési értékét is a második példány kapja. Behelyettesítődik a függvényhívás helyén, és íme, már a második függvénynek is van visszatérési értéke. A második példányt az első példány hozta létre, így a második példány visszatérési értékét az első példány veszi át. Az érték behelyettesítődik a függvényhívás helyén. Így már az első függvénynek is van visszatérési értéke, amit végül a főprogramnak ad vissza. A már nem használt függvény példány a paraméter visszaadása után megszűnik.

A most tárgyalt működésből következik, hogy a függvényhívások fázisában az éppen nem futó függvénypéldányt is tárolni kell, hiszen később használjuk. Ez sok hívás esetén memóriaigényes lehet.

Felépítése

Érettségin eddig nem kellett önállóan rekurzív algoritmust írni, de a követelmények szerint akár ez is megtörténhet. Ezért nézzük meg, milyen egy rekurzív függvény felépítése.

A függvénynek hívnia kell önmagát, tehát lesz legalább egy rekurzív hívást tartalmazó rész. A példában kettő is van, a 3. és 4. sorok végén.

A függvénynek egyszer be kell fejeződnie, tehát lesz egy rész, ahol a függvény konkrét visszatérési értéket ad (2. sor vége).

Lesz egy bázisfeltétel (a példában a = b), aminek a segítségével el lehet dönteni, hogy vége van-e a rekurzív hívások sorozatának.

Lesz egy elágazás a bázisfeltétel szerint, aminek egyik ágában a függvényhívás, másik ágában a konkrét értékadás lesz. Persze az értékadást és függvényhívást is megelőzhetik további műveletek.

Végül pedig szükség van egy olyan műveletre, ami minden egyes függvényhívással közelebb visz a rekurzió végéhez. A példában ez az a - b, illetve a b - a a 3. és 4. sorok függvényhívásaiban.

Ha rosszul írjuk meg a rekurzív alprogramot, előfordulhat, hogy végtelen sokszor kellene meghívnia magát. Így olyan sokszor jön létre új példány belőle, hogy megtelik az erre a célra használható memória, és a program hibaüzenettel megáll. Esetleg előtte látszólag sokáig nem tesz semmit, és csak utána áll meg hibaüzenettel.

Rekurzív és iteratív formák

A rekurzió is egyfajta ismétlés, a ciklusok méltó vetélytársának látszik. Amikor a feladatot rekurzió alkalmazása nélkül, ciklusokkal oldjuk meg, iteratív formáról beszélünk.

Ugyanannak a problémának létezhet rekurzív és iteratív megoldása is. Az iteratív megoldás általában gyorsabban fut, és kevésbé memóriaigényes. A rekurzív forma néha rövidebb, és az ember számára jobban megérthető.

A rekurzív függvények alkalmazása nem igényel új kódolási ismereteket.

Feladatok

1. Kódoljuk az alábbi, legnagyobb közös osztó kiszámolására szolgáló függvényt. Próbáljuk is ki.

```
Függvény lnko(a, b : egész számok) : egész szám
   ha a=b akkor lnko := a
   ha a<b akkor lnko := lnko(a, b-a)
ha a>b akkor lnko := lnko(a-b, b)
Függvény vége
```

2. Kódoljuk az alábbi, legkisebb közös többszörös kiszámolására szolgáló függvényt. Használjuk fel az előző feladat függvényét. Próbáljuk is ki.

```
Függvény lkkt(a, b : egész számok) : egész szám
   lkkt := a * b / lnko(a, b)
Függvény vége
```

3. N természetes szám faktoriálisa alatt 1-től N-ig a természetes számok szorzatát értjük. Írjunk rekurzív függvényt, ami paraméterként megkap egy természetes számot, és visszaadja a faktoriálisát. (0 faktoriálisa 1, 1 faktoriálisa 1, 2 faktoriálisa 2, 3 faktoriálisa 6, 4 faktoriálisa 24)
4. A Fibonacci-számsorozat első és második eleme 1. A következő elemet mindig úgy kapjuk, hogy az előző két elemet összeadjuk: 1, 1, 2, 3, 5, 8, 13, ... Készítsünk rekurzív függvényt, ami meghatározza a bekért sorszámú Fibonacci-számot. Például, ha 7-et adunk meg, 13-at ad vissza.

15. MODULÁRIS PROGRAMOZÁS ÉS TESZTELÉS

Az előzőekben láthattuk, hogy egy feladat modulokra bontása eljárások, illetve függvények segítségével áttekinthetőbbé teszi és megrövidíti a kódot, feloszthatóvá teszi a munkát, ráadásul könnyebb a tesztelés is.

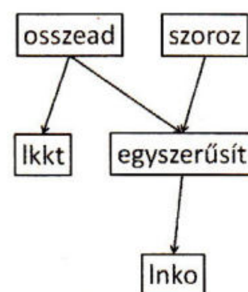
A kérdés csak az, hogyan bonthatunk fel egy összetett feladatot modulokra, majd hogyan illeszthetjük őket ismét össze.

Modulokra bontás a közös részek kiemelésével

Először érdemes megfigyelni a feladatban fellelhető részleges ismétlődéseket. A mellékelt ábrán a feladatok a következők: a megadott tört egyszerűsítése, két tört szorzása, illetve két tört összeadása. A feladathoz rendelkezésünkre áll a legnagyobb közös osztót (lnko), valamint a legkisebb közös többszöröst (lkkt) meghatározó függvény.

1. Egyszerűsítés
   24/32 = 3/4
   24/6 = 4 (ha a tört értéke egész)
2. Tört szorzása
   24/32 * 12/15 = 288/480 = 3/5
   24/32 * 8/3 = 192/96 = 2
3. Tört összeadása (rendelkezésre áll az lkkt függvény)
   24/32 + 8/3 = 72/96 + 256/96 = 328/96 = 41/12
   22/4 + 27/6 = 66/12 + 54/12 = 120/12 = 10

Az ábrán jól látható, hogy a törtek egyszerűsítését minden művelet után el kell végezni. Ezért kár lenne minden művelethez megírni azt. Írjunk egy *egyszerűsít* nevű függvényt, és azt mindhárom esetben felhasználhatjuk.



Az *egyszerűsít* függvény a legnagyobb közös osztót, az összead függvény pedig a legkisebb közös többszöröst meghatározó függvényt használja fel. Így elkészíthetjük a feladatsorunk "függvénytérképét", egy olyan ábrát, ahol látszik, melyik függvény melyiket használja. Ez az ábra mutatja, milyen sorrendben kell a függvényeket elhelyezni a kódban. (A függvényre történő hivatkozás előtt meg kell történnie a deklarációnak.)

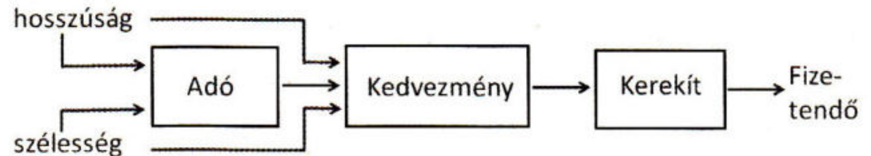
Összefoglalva: Ezzel a módszerrel a közös részeket egy önálló függvénybe (eljárásba) emeljük ki.

Modulokra bontás a feladat szétszedésével

Egyetlen feladat önmagában is túl bonyolult lehet az egyben történő megíráshoz, vagy éppen a teszteléshez.

Általános tanács, hogy a feladat szövegében szorosan összetartozó dolgokat érdemes egyben hagyni, és a kevésbé összetartozók mentén szétbontani.

Tekintsük például az alábbi feladatot:




```
1) Függvény kerekit(fab : egész)
2) Függvény kedv(h: egész, sz: egész, p: egész)
3) Függvény ado...
4)    ...
5)    ado := kerekit(kedv(hossz,szel,f))
6) Függvény vége
7) Főprogram
8)        Ki: ado(40, 20)
```

Az önkormányzat telekadót fog kivetni. Az adót Fabatkában számolják. A 700 négyzetméteres és annál kisebb telkek esetén ez 51 Fabatka négyzetméterenként, az ennél nagyobb telkeknél az első 700 négyzetméterre vonatkozóan szintén 51 Fabatka, 700 négyzetméter felett egészen 1000 négyzetméterig 39 Fabatka a négyzetméterenkénti adó. Az 1000 négyzetméter feletti részért négyzetméterárat nem, csak 200 Fabatka egyösszegű átalányt kell fizetni. A 15 m vagy annál keskenyebb, illetve a 25 m vagy annál rövidebb telkek tulajdonosai 20% adókedvezményben részesülnek. Az adó meghatározásánál 100 Fabatkás kerekítést kell használni (pl. 6238 esetén 6200, 6586 esetén 6600). Írjunk függvényt, ami meghatározza egy telek adóját, ha megadjuk az oldalainak méretét.

A feladatot többször figyelmesen elolvasva rájöhetünk, hogy maga a szöveg is három részre tagolódik. Az elején a legnagyobb rész az adó meghatározásáról szól. Utána következik a kedvezmény megállapítása, majd végül a kerekítés.

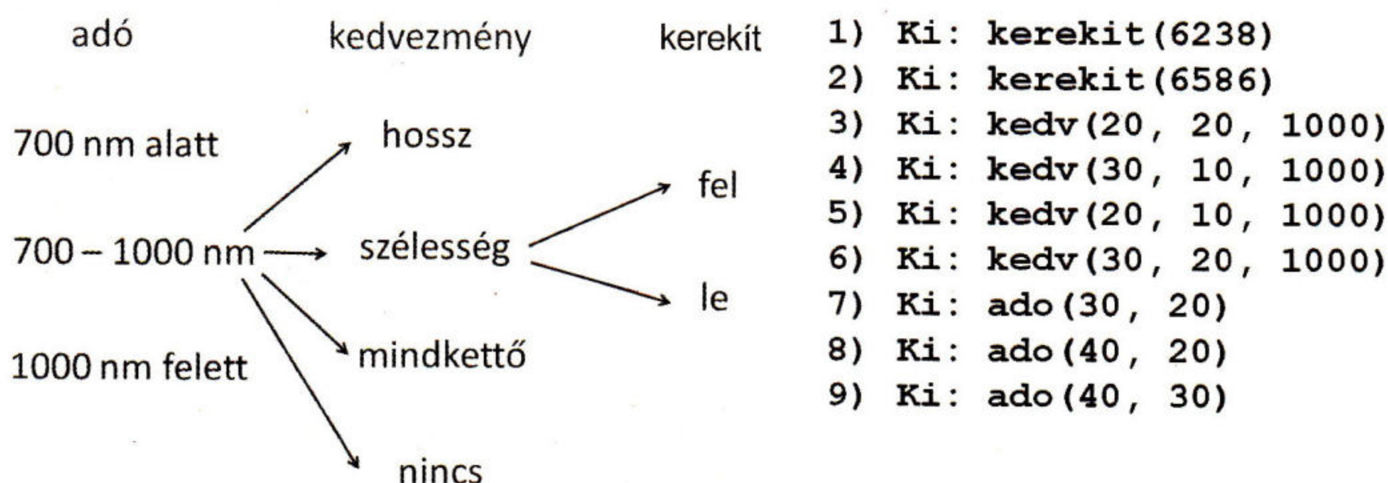
Ennek alapján érdemes megrajzolni a függvények egymásra épülésének térképét, feltüntetve a paramétereket és a visszaadott értékeket. Ennek segítségével már könnyen deklarálhatjuk a függvényeket, határozhatjuk meg a sorrendjüket a kódban (1–3. sor).

Ha a függvényeket külön-külön megírtuk, valahogy meg kell oldani az egymás után történő meghívásukat. Ezt megtehetjük például a függvények egymásba ágyazásával, azaz a kódban előrébb szereplő függvényt meghívjuk a későbbivel (5. sor).

Moduláris és funkcionális tesztelés

A modulokra bontás általában nagyban leegyszerűsíti a tesztelést is. Számoljuk át, hogy a teljes lefedés elve alapján történő teszteléshez hány tesztesetre van szükség modulokra bontva, és hányra egyben a telekadós példában.

A kerekít függvény teszteléséhez két teszteset elegendő, mivel felfelé (2. sor) vagy lefelé (1. sor) kerekíthetünk. A kedvezményt számoló függvényt négy esetből tudjuk tesztelni a feltételek alapján: hosszúság alapján jár kedvezmény (3. sor), szélesség alapján jár kedvezmény (4. sor), mindkettő alapján jár kedvezmény (5. sor), egyik alapján sem jár kedvezmény (6. sor). Végül az adó-függvény három esettel fedhető le: 700 négyzetméter alatt (7. sor), 700 és 1000 négyzetméter között (8. sor), és 1000 négyzetméter felett (9. sor). Ha ehhez hozzávesszük a függvények összeillesztésének tesztelését, összesen 2+4+3+1 = 10 tesztesetet kapunk.



Ha egyben hagytuk volna a feladatot (vagy csak egyben tesztelnénk), ez a szám 2*4*3=24 lenne, hiszen például a 700 négyzetméter alatti telek lehet hossz miatt kedvezményes, szélesség miatt kedvezményes, mindkettő miatt kedvezményes, vagy egyik miatt sem. S mindezek az esetek kerekedhetnek felfelé vagy lefelé. (Lásd bal oldali ábra.) Ami talán ennél is időigényesebb, az összes megfelelő teszteset megtalálása, hiszen honnan tudnánk előre, hogy például az 1000 négyzetméter feletti, kedvezményes hosszúságú telek adója lefelé vagy felfelé fog kerekedni a végén?

Mikor a modulokat önállóan, azaz a többi modultól függetlenül teszteljük, modultesztelésről beszélünk.

Időnként azonban a modulteszt nem elegendő. Egyrészt, mert ritkán történik mindenre kiterjedő alaposságú tesztelés. Másrészt a modulok összeillesztésében is lehet hiba. Ezért általában szükség van a teljes feladat összefüggő tesztjére is.

Az egymással összekapcsolt modulok együttes tesztelését funkcionális tesztelésnek nevezzük.

Mint fentebb láthattuk, a funkcionális tesztelés jóval több esetet jelentene, amit ráadásul gondos tervezéssel kellene kiválasztani. Mindez nagyon időigényes, ezért általában kiválasztanak néhány (szélsőséges esetben egyetlen) jellemző bemeneti paraméterekkel rendelkező tesztesetet, és csak ezekkel (ezzel) végzik el a funkcionális tesztet.

Az adó-kiszámítós példában a 7–9. sorok bármelyikét választhatjuk funkcionális tesztnek. A különbség a moduláris teszteléshez képest az, hogy itt már elvégeztük a függvények egymásba ágyazását, míg a modultesztelésnél csak a kedvezmény és kerekítés nélküli adó kiszámítását teszteljük.

Néhány funkcionális teszt nagyon alapos modulteszteléssel párosítva a legtöbb esetben már elfogadható üzembiztonságot ad.

Természetesen egy érettségin nem lesz időnk arra, hogy ilyen alaposan megtervezzük a tesztelést. A modulteszteket általában a modulok írása közben többé-kevésbé letudjuk, a funkcionális tesztből általában beérjük eggyel az egyes feladatok elkészülte után.

A folyamatosan fejlesztés alatt álló szoftvereket a cégek gyakran automatikusan tesztelik. Mind a modul-, mind a funkcionális teszteket rendszeres időközönként egy program futtatja, s hiba esetén jelzést küld a fejlesztőknek. (A folyamatos fejlesztés miatt a modulok változhatnak.) Így takarítják meg a tesztelésre szánt drága munkaórák egy tekintélyes részét[39].

Végül megemlítendő, hogy a szoftverhibák (majd a különböző javítócsomagok) közvetlen oka a nem megfelelő alaposságú tesztelés, ami mögött egyrészt a szoros határidők állnak, másrészt az a szemlélet, hogy az ügyfél nem fizet a tesztelésért[40]. Az sem mellékes, hogy a szoftverfejlesztők általában nem szeretnek tesztelni. Mára gyakran a szoftvertesztelő nemcsak külön munkakörként, hanem külön foglalkozásként jelenik meg.

---

39 Magyarországi főiskola is alkalmazta a tesztautomatizálást a hallgatók által beadott programok értékelésére. Maga az automatizálás is munkaidőt igényel, így alaposan meg kell gondolni, megéri-e.

40 Az ügyfél igyekszik minél alacsonyabb árat kiharcolni, míg a szoftverfejlesztő cég a magas óradíjban (kevés munkával sok pénzt) érdekelt. Ennek eredménye a kevesebb ráfordított munkaóra. A legegyszerűbb a tesztelési időt megnyirbálni.

Feladatok

1. Az önkormányzat telekadót fog kivetni. Az adót Fabatkában számolják. A 700 négyzetméteres és annál kisebb telkek esetén ez 51 Fabatka négyzetméterenként, az ennél nagyobb telkeknél az első 700 négyzetméterre vonatkozóan szintén 51 Fabatka, 700 négyzetméter felett egészen 1000 négyzetméterig 39 Fabatka a négyzetméterenkénti adó. Az 1000 négyzetméter feletti részért négyzetméterárat nem, csak 200 Fabatka egyösszegű általányt kell fizetni. A 15 m vagy annál keskenyebb, illetve a 25 m vagy annál rövidebb telkek tulajdonosai 20% adókedvezményben részesülnek. Az adó meghatározásánál 100 Fabatkás kerekítést kell használni (pl. 6238 esetén 6200, 6586 esetén 6600). Írjunk függvényt, ami meghatározza egy telek adóját, ha megadjuk az oldalainak méretét. Bontsuk több részfeladatra, függvényekre.)
2. Végezzük el az előző feladat modultesztelését a teljes lefedés elve szerint. A tesztesetek legyenek ott a főprogramban megjegyzésként.
3. Végezzük el a 1. feladat funkcionális tesztelését. A tesztesetek legyenek ott a főprogramban megjegyzésként.
4. Írjunk programot, ami bekéri két közönséges tört számlálóját és nevezőjét. A törteket külön-külön leegyszerűsíti. Ha felírhatók egész számként, az egész értéküket jeleníti meg. A két bekért törtet összeszorozza, illetve összeadja, és az eredményeket leegyszerűsítve, ha lehetséges egész számként megadja. Felhasználható a Rekurzió fejezetnél felírt legnagyobb közös osztó, illetve legkisebb közös többszörös függvény. Az ismétlődéseket emeljük ki függvénybe.
5. Végezzük el az előző feladat modultesztelését a teljes lefedés elve szerint. A tesztesetek legyenek ott a főprogramban megjegyzésként.
6. Végezzük el a 4. feladat funkcionális tesztelését. A tesztesetek legyenek ott a főprogramban megjegyzésként.

V. ÖSSZETETT ADATSZERKEZETEK

1. LISTA

Bevezetés

Ha sok változóra van szükségünk (pl. 100-ra), hosszú időbe telne, mire mindegyiknek saját nevet adnánk. Problémát okoz a nevek megjegyzése, valamint az, hogy a későbbiekben hogyan hivatkozzunk ezekre. Megoldást jelent, ha a sok változónak egyetlen nevet adunk, de mellette sorszámozzuk őket. Például: honap(2). Ezek lesznek a listák.

A lista névvel és sorszámokkal ellátott összetett változó. A lista elemeire a nevével és a sorszámukkal együtt hivatkozunk.

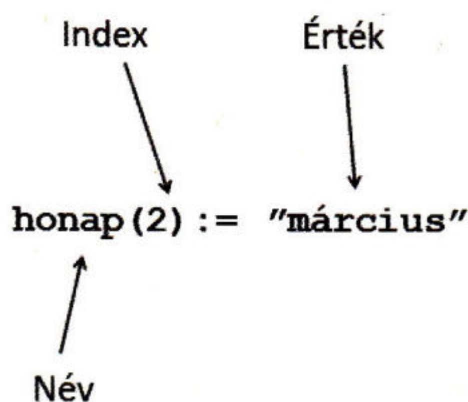
Elképzelhetjük a listákat, mint egy fiókos szekrényt. A szekrénynek nevet adunk, például hónap, a fiókjait pedig megszámozzuk. A fiókok tartalma pedig a változóban tárolt érték lesz.

A listaelem sorszámát **index**nek nevezzük.

Felfoghatunk egy listát egy számozott felsorolásként, aminek neve van: például a *honap* nevű felsorolás (lista), aminek 0. eleme a január, 1. eleme a február, stb.

A Pythonban a lista elemei akár különböző típusúak is lehetnek, sőt akár egy másik lista is lehet listaelem.

Más programnyelvekben és az általános programozás módszertanban létezik a **tömb** fogalma. A fenti példa lehetne akár tömb is. A lista egy sokoldalúbb adatszerkezet: olyan műveleteket hajthatunk végre, amiket egy tömbön nem (hozzáfűzés, törlés, rendezés, stb.), va-



honap

Index	Érték
0	→ január
1	→ február
2	→ március
3	→ április
4	→ május

lamint a tömb elemei csak egyféle típusúak lehetnek. Azokat a tömböket, amik egy indexszel kezelhetők, **vektorok**nak is nevezik.

A listaelem értékét az egyszerű változókkal megegyező módon adhatjuk meg, módosíthatjuk, érhetjük el.

A Pythonban használható a **tuple**, ami egy olyan lista, aminek **nem módosíthatók az elemei**. A listához képest kevesebb helyet foglal a memóriában, és gyorsabb a hozzáférés. Érdemes használni, ha fix elemű a lista (például: hét napjai, hónapok neve)[41].

A listákkal elvégezhető fontosabb műveleteket az Összefoglalás, kiegészítés / Összetett változók, adatszerkezetek fejezetben találjuk, a További listaműveletek táblázatban.

Kódolás

A Pythonban a listák indexelése mindig nullával kezdődik.

Így a 6 elemű lista utolsó tömbindexe 5, és nem 6. Gyakran előforduló hiba a tömb használatakor, hogy a lehetségesnél nagyobb értéket adunk meg indexnek. Ilyenkor a programunk nem fog működni, és hibaüzenetet kapunk: túlindexeltük a listát (IndexError: list index out of range).

Szintén fontos tudni, hogy kör alakú zárójelek helyett szögletes zárójelet kell használni. Más nyelvekben (Basic, Pascal) kör alakú zárójelet használnak, és ez a szokás maradt meg a mondatszerű leírásban.

A tuple-ek indexeit szintén szögletes zárójelek közé írjuk, de a létrehozáskor kör alakú zárójelet használunk:

```
lista:  names = ["András", "Bori", "Csaba"]
        print(names[1])
tuple:  days = ("hétfő", "kedd", "szerda")
        print(days[1])
```

41 Tuple-ek nélkül is le lehet érettségizni, így ez kiegészítő információnak számít az érdeklődők számára. A tuple-ekre ezért nem is készítettem külön feladatokat. Ha valaki gyakorolni szeretné, gondolja meg, hogy a felhasznált lista módosul-e a létrehozása után. Ha nem módosul, alkalmazzon gyakorlásként mindig tuple-t.

47. mintafeladat – ismerkedés a listákkal

1. Álljunk a parancsértelmező ablakba (**Python 3.8.0 Shell**).
2. A prompt jel >>> mögött villog a kurzor. Oda írjuk majd be a parancsokat, mindegyik után megnyomva az **Enter** billentyűt.
3. Készítsünk az első három hónap nevéből egy *honap* nevű listát. A listát szögletes zárójelek közé írjuk, elemeit vesszővel választjuk el egymástól. Azért vannak idézőjelek is, mert karakterláncok az elemek. Írjuk be, majd üssünk **Entert**:
   ```
   honap = ["január", "február", "március"]
   ```
4. Visszakaptuk a prompt jelet, jöhet az újabb parancs. Írassuk ki a lista 2 indexű elemét. Írjuk be, majd üssünk **Entert**:
   ```
   print(honap[2])
   ```
5. Azt írja ki a parancsértelmező, hogy március.
6. Írassuk ki most az egész listát. Írjuk be, majd üssünk **Entert**:
   ```
   print(honap)
   ```
7. Kiíródik az összes elem, zárójelestül, aposztrófokkal, vesszőkkel. Ez így használhatatlan lesz a programban, mert ott általában valamilyen formázott kiíratás az elvárás. (Például: január 23.) Viszont nagyon hasznos lehet tesztelési céllal, hogy lássuk, éppen mi van a listában.
8. Hozzunk létre egy *nev* nevű üres listát. Írjuk be, majd üssünk **Entert**:
   ```
   nev = []
   ```
9. Próbáljunk értéket adni a lista 0 indexű elemének. Írjuk be, majd üssünk **Entert**:
   ```
   nev[0] = "Glázser Bozsó"
   ```
10. Hibaüzenetet kapunk: *IndexError: list assignment index out of range*. Ennek az oka, hogy a lista üres, tehát nincs 0. eleme, így minek is adnánk értéket.
11. Listába (így üres listába is) új elemet felvenni az **append** utasítással lehet. Írjuk be, majd üssünk **Entert**:
    ```
    nev.append("Glázser Bozsó")
    ```
12. Most már, hogy van 0 indexű eleme a listának, kiírathatjuk. Írjuk be, majd üssünk **Entert**:
    ```
    print(nev[0])
    ```
13. Megváltoztathatjuk a listaelemet értékadással. Írjuk be, majd üssünk **Entert**:
    ```
    nev[0] = "Nyomasek Bobó"
    ```
14. Ellenőrzésként ismét írassuk ki a 0 indexű elemet. Ha minden rendben, Nyomasek Bobót fogjuk kapni. Írjuk be, majd üssünk **Entert**:
    ```
    print(nev[0])
    ```
15. Másképpen is létrehozhatunk egy listát (a példában a lista neve *szuletes*). Ha tudjuk, hogy kezdetben 40 elemű lesz, akkor feltöltjük létrehozáskor 40 db [0] elemmel. Írjuk be, majd üssünk **Entert**:
    ```
    szuletes = [0]*40
    ```

16. Adjunk értéket a 29 indexű elemnek. Legyen az értéke 1. Írjuk be, majd üssünk **Entert**:
    `szuletes[29] = 1`
17. Ellenőrzésként írassuk ki a 29 indexű elemet. Ha minden rendben, az 1-et fogjuk kapni. Írjuk be, majd üssünk **Entert**:
    `print(szuletes[29])`
18. Nézzük meg, hogyan lehet egy index maga is listaelem. Írjuk be, majd üssünk **Entert**:
    `print(honap[szuletes[29]])`
19. Eredményként a februárt kapjuk. A szuletes[29] értéke 1, így a honap mögötti zárójelbe az 1 kerül. A kiírt elem tehát *február* lesz:
    `honap[szuletes[29]] → honap[1] → február.`

48. mintafeladat – néhány listaművelet

Egy játékba tippeket küldtek számok formájában, amiket az érkezés sorrendjében jegyeztek le. Az nyer, aki a legkisebb számra tippel *egyedül*. Próbáljunk ki pár listaműveletet rajta.

```
>>> tippek = [3, 12, 1, 8, 5, 8, 1, 2, 1, 4]
>>> tippek.index(2) + 1
8
>>> masolat = tippek.copy()
>>> del tippek[3:5]
>>> tippek
[3, 12, 1, 8, 1, 2, 1, 4]
>>> tippek.sort()
>>> tippek
[1, 1, 1, 2, 3, 4, 8, 12]
>>> tippek.reverse()
>>> masolat
[3, 12, 1, 8, 5, 8, 1, 2, 1, 4]
>>> hivatkozas = tippek
>>> tippek.remove(12)
>>> tippek
[8, 4, 3, 2, 1, 1, 1]
>>> hivatkozas
[8, 4, 3, 2, 1, 1, 1]
```

1. Álljunk a parancsértelmező ablakba (**Python 3.8.0 Shell**).
2. A prompt jel >>> mögött villog a kurzor. Oda írjuk majd be a parancsokat, mindegyik után megnyomva az **Enter** billentyűt.
3. Adjuk meg a következő listát, azaz írjuk be a parancssorba:
   `tippek = [3, 12, 1, 8, 5, 8, 1, 2, 1, 4]`
4. A helyes tipp a 2 volt. Hányadiknak érkezett? Ennek megállapításához használjuk az **index** függvényt, ami az elem sorszámát adja meg. Ehhez még hozzá kell adnunk 1-et, hogy ne a 0-val kezdődjön a számlálás:
   `tippek.index(2) + 1`
5. Készítsünk *masolat* néven a tippek tömbről egy valódi másolatot, azaz a *tippek* és *masolat* tömb egymástól függetlenül módosíthatók legyenek. Erre a célra szolgál a **copy** függvény:
   `masolat = tippek.copy()`

6. A 4. és 5. tippek beadói csaltak, ezért töröljük őket a *tippek* listából. Indexszel megadott elemeket a del paranccsal törölhetünk. A megadott sorszámokhoz képest az index 1-gyel kisebb, így a törlendő tartomány [3:5], hiszen az 5.-et már nem fogjuk törölni:
```
del tippek[3:5]
```
7. Ellenőrizzük a maradék listát, tényleg a megfelelő elemeket töröltük-e (igen):
```
tippek
```
8. Rendezzük a tippeket növekvő sorrendbe, hogy jól látszódjon, melyik a legkisebb, nem ismétlődő szám, mert az a nyerő tipp. Ezt a **sort** függvény segítségével tehetjük meg[42]:
```
tippek.sort()
```
9. Ellenőrizzük a listát, megtörtént-e a rendezés (igen):
```
tippek
```
10. Nézzük meg a listát csökkenő sorrendbe rendezve. Ezt úgy tudjuk megtenni, hogy létezik a **reverse** függvény, ami megfordítja az elemek sorrendjét. Ez az utasítás nem csökkenő sorrendbe rendez. Most azért lesz mégis a lista csökkenő sorrendben, mert előtte növekvő sorrendben volt.
```
tippek.reverse()
```
11. Ellenőrizzük a listát, megtörtént-e a csökkenő rendezés (igen):
```
tippek
```
12. Most ellenőrizzük a *masolat* listát. Ha tényleg független a tippektől, akkor nem szabad rendezettnek lennie, és két elemmel több van benne.
```
masolat
```
13. Készítsünk *hivatkozas* néven a tippek tömbről egy nem valódi másolatot, azaz a *tippek* és *hivatkozas* tömb egymástól függetlenül ne legyenek módosíthatók.
```
hivatkozas = tippek
```
14. Csak 10 alatti számra lehetett tippelni, így töröljük a 12-es számot a tippek listából. Erre a célra szolgál a **remove** parancs:
```
tippek.remove(12)
```
15. Ellenőrizzük mindkét listát, megtörtént-e a 12 eltávolítása (igen):
```
tippek
hivatkozas
```

42 Ékezetes betűket tartalmazó karakterláncokra nem működik megfelelően.

Feladatok

1. Használjunk egy listát a hét napjainak tárolására. A neveik sorrendben tárolódjanak (hétfő, kedd,...).
   a) Hozzuk létre a listát, és töltsük fel a megfelelő értékekkel.
   b) Írassuk ki a 0-s indexű nap nevét.
   c) Írassuk ki a hét harmadik napjának nevét. (A hétköznapi szóhasználatban nem használunk 0. napot, a hétfő a hét első napja.)
   d) Írassuk ki a szombatot, az indexet és a listanevet használva.
   e) Kérjünk be a felhasználótól egy 1...7 számot, amivel megadja, hogy a hét hányadik napját szeretné látni. Írassuk ki a nap nevét.
2. Azt szeretnénk tárolni, hogy egy osztálykarácsony alkalmával ki és kinek adott ajándékot. Ehhez készítsünk listát, aminek indexe mutatja az ajándékozót, és eleme mutatja a megajándékozottat.
   a) Hozzuk létre a listát, és töltsük fel adatokkal, ha a 0. tanuló a 6.-nak adott ajándékot:
   0 → 6, 1 → 5, 2 → 8, 3 → 7, 4 → 10, 5 → 3, 6 → 1, 7 → 2, 8 → 0, 9 → 4, 10 → 9.
   b) Írassuk ki a listát felhasználva, kinek adott ajándékot a 7. tanuló, ilyen formátumban: *A 0. tanuló a 6. tanulónak adott ajándékot.*
   c) Kinek adott ajándékot, aki a 7. tanulótól kapott ajándékot? Írjuk ki a számát a képernyőre. A feladat megoldásához ne használjunk segédváltozót.
   d) A 3. és 4. tanuló elcserélte, hogy kinek ajándékoz. Végezzük el az ehhez szükséges módosításokat a listán. Ne a kezdőértékeken változtassunk. Írassuk ki csere előtt és után, kinek adtak ajándékot.
   e) Nevesítsük a tanulókat egy tömb segítségével, azaz készítsünk egy listát:
   0 – András, 1 – Béla, 2 – Cecília, 3 – Dóra, 4 – Elemér, 5 – Fanni, 6 – Glória, 7 – Hedvig, 8 – Ilona, 9 – József, 10 – Katalin.
   f) Kérjünk be egy 0...10 számot, és írassuk ki névvel, kinek adott ajándékot a bekért számú tanuló, ilyen formában: *Géza Rékát ajándékozta meg.* Ne használjunk segédváltozót.
   g) Kérjünk be egy 0...10 számot, és írassuk ki nevekkel egy láncolat formájában, kinek adott ajándékot, aki a bekért számú tanulótól kapott ajándékot. Pl.: Géza -> Réka -> Béla.

2. LISTA BEJÁRÁSA (FOR)

Gyakori feladat, hogy a lista minden elemével valamilyen műveletet kell végrehajtanunk. Legegyszerűbb esetben formázottan kiíratni, vagy feltölteni véletlenszerű értékekkel. Ilyenkor azt mondjuk, bejárjuk a listát.

A lista bejárásához készítették a Pythonban a **for** ciklust eredeti alakjában.

Ha csak a lista elemeire van szükség, és az indexük nem fontos, alkalmazhatjuk a 3–4. sorban látható módon. Az **in** mögött megadjuk a lista nevét (honap). A ciklus minden egyes lefutásakor a lista egy még sorra nem került eleme (azaz a példában az egyik hónap neve, ami még nem volt az előző lefutások alkalmával) az i változóban lesz. Az ismétlés addig tart, míg a lista minden eleme sorra nem került. Így a példában a 3–4. sor kiírja a képernyőre a listában tárolt hónapok nevét.

Kicsit bonyolultabb a helyzet, ha szükség van az elem indexére is.

Ehhez először ismerkedjünk meg a **len** utasítással, ami megadja a lista elemeinek számát. Az 5. sor eredménye az 5 lesz, hiszen a listában 5 hónap neve található.

Tegyük fel, hogy a hónapok neve előtt szeretnénk megjeleníteni a sorszámukat is. Ehhez szükségünk lesz az elemek indexére is. Ilyen esetekben az **in** mögött az *indexek tartományát* adjuk meg (6. sor). A **len** utasítás megadja a lista elemszámát (a példában 5), így a **range** a 0-tól kezdődően egy egész számokból álló sorozatot hoz létre, amik pont a lista indexei (a példában: 0, 1, 2, 3, 4). A j változóba a ciklus minden egyes lefutásakor egy index kerül. Ezt használjuk fel a 7. sorban előbb a hónap sorszámának (j+1, mert 0. hónapról nem szokás beszélni), majd nevének (honap[j]) kiírására.

```
1 honap = ["január", "február", "március", \
2           "április", "május"]
3 for i in honap:
4     print(i, end=" ")
5 print("\nA tömb mérete: ", len(honap))
6 for j in range(len(honap)):
7     print("%d. hónap: %s" % (j+1, honap[j]))
```

49. mintafeladat – listaelemek bejárása

Készítsünk egy listát az első öt hónap nevével. Jelenítsük meg először a hónapok nevét, egymástól szóközzel elválasztva. Határozzuk meg a lista elemeinek számát. Írassuk ki a hónapok nevét a sorszámukkal együtt. Minden hónap külön sorba kerüljön, ilyen formában: 2. hónap: február.

```
=============== RESTART: C:/python/l
istabejaras.py ===============
január február március április május
A tömb mérete:  5
1. hónap: január
2. hónap: február
3. hónap: március
4. hónap: április
5. hónap: május
```

1. Gépeljük be az elméletnél látható Python-kódot (1–7. sor). Próbáljuk ki.

Feladatok

1. Tároljuk egy listában a hét napjait. Írassuk ki a képernyőre a hét napjait sorrendben, egymás mellé, szóközzel elválasztva.
2. Tároljuk egy listában a hét napjait. Írassunk ki a képernyőre véletlenszerűen 15 db nap nevét egymás mellé, szóközzel elválasztva.
3. Az amőbajáték ábrázolásához háromféle karaktert használunk: 'X', 'O', és '.' Utóbbival az üres helyeket ábrázoljuk. Tároljuk egy listában a felhasznált karaktereket. Írassuk ki a képernyőre az amőbajáték ábrázolásához használt karaktereket egymás mellé.
4. Töltsük fel egy lista 50 elemét egy dobókocka véletlenszerű értékeivel, majd írassuk ki a képernyőre ilyen módon: 1 3 1 4 4 5 6 2... A feltöltés és a kiíratás két külön ciklussal történjen.
5. Töltsük fel egy lista 100 elemét fej vagy írás véletlenszerű értékeivel, majd írassuk ki a képernyőre ilyen módon: *fej írás* írás fej... A feltöltés és a kiíratás két külön ciklussal történjen.

3. A KARAKTERLÁNC MINT LISTA

Fogalma

A karakterlánc (string) egy különleges karakterlistának tekinthető. Ennek megfelelően minden karakterére hivatkozhatunk a karakterlánc nevével és a betű sorszámával. Tárolja például a *nev* nevű karakterlánc azt a nevet, hogy András. Ebben az esetben a nev(2) értéke a d betű lesz. A számozás itt is a 0-tól indul.

nev

0	→	A
1	→	n
2	→	d
3	→	r
4	→	a
5	→	s
6	→	\0

Ki: nev(2) → d

Karakterlánc vége jel

A karakterlánc azért különleges karaktertömb, mert tartalmaz egy lezáró karaktert, amit \0-val jelölünk. Ennek segítségével tudják megállapítani a Python beépített függvényei, hol a karakterlánc vége.

Alapvető műveletek karakterláncokkal

A karakterlánc akárhányadik karaktere lekérdezhető ilyen módon. Például: *Ki: nev(2)*(4. sor).

```
1) w = "isz"
2) hb = ""
3) hb = "h" + w + "ed"  →  hiszed
4) print (w[2])        →      z
5) w[2] = 'i'
```

Ellenben a Pythonban nem változtathatunk meg egyszerű értékadással olyan karaktert, ami része egy karakterláncnak (5. sor). Azonban a kívánt hatást elérhetjük, ha a lecserélendő karaktert először töröljük, majd a helyére beszúrjuk a kívánt karaktert.

Üres karakterláncot a 2. sorban található módon hozhatunk létre.

A karakterláncok összefűzhetők a + jel segítségével (3. sor). A művelet eredménye egy karakterlánc, ami balról jobbra sorrendben egymás után tartalmazza a kiindulási karakterláncokat. A *w* változóban az *isz* karakterlánc tárolódik, így hb = "h" +"isz"+ "ed" = "hiszed".

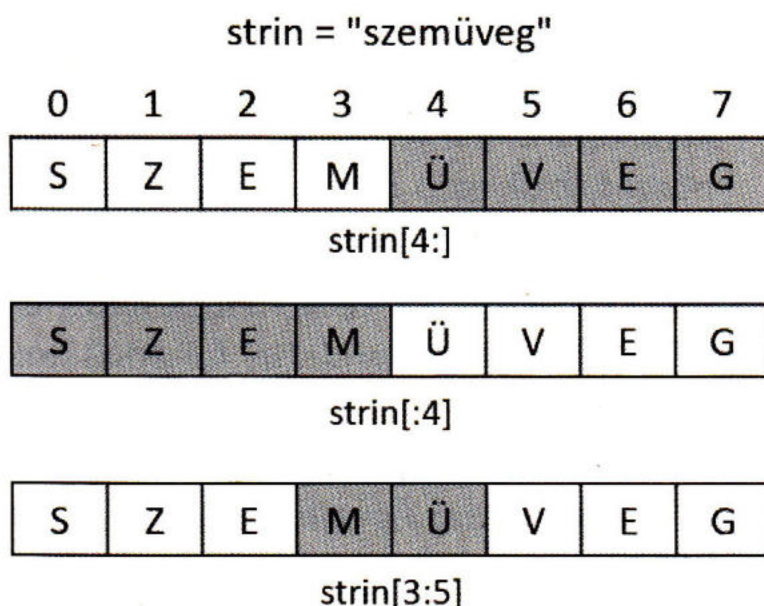
A szövegösszefűzést szaknyelven **konkatenációnak** is hívják.

50. mintafeladat – ismerkedés a karakterláncokkal

1. Álljunk a parancsértelmező ablakba (**Python 3.8.0 Shell**).
2. A prompt jel >>> mögött villog a kurzor. Oda írjuk majd be a parancsokat, mindegyik után megnyomva az **Enter** billentyűt.
3. Készítsünk egy *str1* nevű karakterláncot, ami ezt tartalmazza: *isz*. Írjuk be, majd üssünk **Entert**:
   ```
   str1 = "isz"
   ```
4. Hozzunk létre egy *hb* nevű, üres karakterláncot. Írjuk be, majd üssünk **Entert**:
   ```
   hb = ""
   ```
5. A *hb* karakterláncban fűzzük egymás után a *h* betűt, az *str1* változó tartalmát, majd az *ed* karakterláncot. Az összefűzést a Pythonban a + jel jelöli. A művelet eredménye egy karakterlánc, ami balról jobbra sorrendben egymás után tartalmazza a kiindulási karakterláncokat. Írjuk be, majd üssünk **Entert**:
   ```
   hb = "h" + str1 + "ed"
   ```
6. Írassuk ki a *hb* változó tartalmát. Az *str1* változóban az *isz* karakterlánc tárolódik, így hb = "h" +"isz"+"ed" = "hiszed". Írjuk be, majd üssünk **Entert**:
   ```
   print(hb)
   ```
7. Írassuk ki a *hb* változó 3. karakterét. (Z betűt kell kapnunk eredményként.) Írjuk be, majd üssünk **Entert**:
   ```
   print(hb[3])
   ```
8. Próbáljuk megváltoztatni a *hb* változó 0. karakterét *v* betűre. Írjuk be, majd üssünk **Entert**:
   ```
   hb[0] = "v"
   ```
9. Hibaüzenetet fogunk kapni: *'str' object does not support item assignment*. Sajnos a karakterek nem helyettesíthetők olyan egyszerűen, mint a lista elemei.

Szeletelés

A karakterláncnak hivatkozhatunk a tartományaira a mellékelt ábra alapján. Ilyenkor a szögletes zárójelbe két számot írhatunk kettősponttal elválasztva (pl. strin[3:5]). Az első szám a tartomány kezdő pozícióját jelzi. A második szám a végpozíciónál eggyel nagyobb. Az ü a 4. karakter, helyette az 5-öt kell megadni.



Ha a második számot hagyjuk el, a tartomány a karakterlánc végéig tart (pl. strin[4:]).

Ha az első számot hagyjuk el, a tartomány a karakterlánc elejétől kezdődik (pl. strin[:4]).

Az itt megismert technikát szeletelésnek hívják, és a listákra is végrehajtható.

51. mintafeladat – alapvető karakterlánc műveletek

1. Álljunk a parancsértelmező ablakba (**Python 3.8.0 Shell**).
2. A prompt jel >>> mögött villog a kurzor. Oda írjuk majd be a parancsokat, mindegyik után megnyomva az **Enter** billentyűt.
3. Készítsünk egy *str1* nevű karakterláncot, ami ezt tartalmazza: *hiszed*. Írjuk be, majd üssünk **Entert**:
```
str1 = "hiszed"
```
4. Határozzuk meg a karakterlánc hosszát, azaz hogy hány karakterből áll. (6 lesz a helyes eredmény.) Írjuk be, majd üssünk **Entert**:
```
print(len(str1))
```
5. Írassunk ki a karakterlánc 2. (azaz 1 indexű) karakterétől számítva 3 karaktert. A 3 karakter a harmadik pozícióban végződik. Ennél eggyel nagyobb számot kell megadni. Ezért a kettőspont mögötti szám a 4. (*isz* a helyes eredmény). Írjuk be, majd üssünk **Entert**:
```
print(str1[1:4])
```

6. Töröljük a karakterlánc első (azaz 0 indexű) karakterét. Ezt úgy tehetjük meg, hogy a többi karaktert (0 indexű utániakat, tehát az 1-től kezdve) áttöltjük az eredeti változóba. (Ezt kapjuk: *iszed.*) Írjuk be, majd üssünk **Entert**:
```
str1 = str1[1:]
print(str1)
```
7. Cseréljük ki a karakterlánc negyedik (azaz 3 indexű) karakterét o betűre. Ezt úgy tehetjük meg, hogy a 0.-tól kezdődően kivágjuk az első 3 karaktert (a 3 karakter a 2.-nál végződik, ennél eggyel nagyobbat kell megadni, tehát 3-at), hozzáfűzzük az o betűt, majd a 4. karaktertől kezdve mellé illesztjük a szó maradékát. (Ezt kapjuk: *iszod.*) Írjuk be, majd üssünk **Entert**:
```
str1 = str1[:3]+"o"+str1[4:]
print(str1)
```

Feladatok

1. Kérjünk be egy szót, majd írassuk ki a hosszát.
2. Kérjünk be egy szót, majd írassuk ki a kezdőbetűjét.
3. Kérjünk be egy szót, majd írassuk ki az utolsó betűjét.
4. Kérjünk be egy ötbetűs szót, és írassuk ki a középső három karakterét.
5. Kérjünk be egy budapesti irányítószámot, és írjuk ki, melyik kerületben található. A budapesti irányítószámok középső két karaktere adja az irányítószámot.
6. Készítsünk egy karakterláncot, ami a *leszel* szót tartalmazza. Minden feladatrész után írjuk ki a karakterláncot. Ne használjunk segédváltozót.
   a. Változtassuk meg a kezdőbetűt L-re.
   b. Változtassuk meg az utolsó betűt k-ra.
   c. Töröljük a kezdő- és utolsó betűjét.
   d. Bővítsük a szó végét egy m betűvel.
7. Kérjük be a felhasználó vezeték-, majd utónevét két külön változóba. Fűzzük össze a vezetéknév első karakterét a teljes utónevével. Írassuk ki az így keletkezett karakterláncot csupa kisbetűvel. (Például: Bodor Elek→belek)
8. Kérjük be a felhasználó vezeték-, majd utónevét két külön változóba. Fűzzük össze a vezeték- és utónevének első három karakterét, majd egészítsük ki 01-gyel. Írassuk ki az így keletkezett karakterláncot csupa kisbetűvel. (Például: Kiss Ferenc→kisfer01)
9. Kérjünk be egy szót. Írassuk ki, hogy 'e' betűvel kezdődik-e.
10. Kérjünk be két szót. Írassuk ki őket hosszúság szerint növekvő sorrendben.
11. Kérjünk be két ötbetűs szót, hasonlítsuk össze a három belső karaktert, és írjuk ki azt, amelyik ábécésorrendben előrébb van.

12. Készítsünk programot, ami bekér egy kétjegyű hexadecimális (16-os számrendszerbeli) számot (pl.: A9), és átváltja 10-es számrendszerbe (Az A9-et például 169-re kell váltania).
13. Kérjünk be egy szót egy karakterláncba. Fordítsuk meg a szót, és írassuk ki a képernyőre. (például: rizs → szir)
    a. Elég a képernyőn fordítva megjeleníteni a szót.
    b. A szó a változóban is forduljon meg, ne csak a képernyőre legyen fordítva kiíratva, de használhatunk segédváltozót.
    c. A szó a változóban is forduljon meg, ne használjunk segédváltozót.
14. Készítsünk egy 20 karakterből álló karakterláncot A, C, G, T betűkből véletlenszerűen. Írjuk ki a kész karakterláncot a képernyőre.
15. Készítsünk egy ötelemű listát, ami szavakat tartalmaz, majd írjunk szókitaláló programot, ami:
    a. Kiválaszt egy szót véletlenszerűen.
    b. A kiválasztott szó hosszával megegyező hosszúságú, csupa X-ből álló karaktersorozatot ír a képernyőre.
    c. Bekér a felhasználótól egy karaktert. Ahol ez a karakter egyezik a szóban találhatóval, ott egy új karaktersorozatban az X-ek helyén a kitalált karakter jelenik meg.
    d. Ismételjük a karakterek bekérését, míg ki nem találtuk a szót. (Egyszerűsítésként először feltételezhetjük, a felhasználó nem ad meg már egyszer elfogadott betűt. Aztán mindenképpen oldjuk meg úgy is, hogy megadhat már kitalált betűt.)
16. Írjon programot, amely előállítja a felhasználó által megadott 1...3999 egész római szám alakját. Az átváltás szabályait az alábbi táblázat tartalmazza. (Ha további segítségre van szükség, a lábjegyzetben található[43].) Minél kevesebb elágazást használjunk[44].

	0	1	2	3	4	5	6	7	8	9
Egyesek	-	I	II	III	IV	V	VI	VII	VIII	IX
Tízesek	-	X	XX	XXX	XL	L	LX	LXX	LXXX	XC
Százasok	-	C	CC	CCC	CD	D	DC	DCC	DCCC	CM
Ezresek	-	M	MM	MMM	-	-	-	-	-	-

43 A helyiértékeket külön-külön átváltjuk, és egymás mellé írjuk.

44 A szerzőnek 1 db elágazással sikerült megoldania.

4. TOVÁBBI MŰVELETEK KARAKTERLÁNCOKKAL

Karakterlánc keresése karakterlánc változóban

Karakterláncot kereshetünk egy változóban a **find** függvény segítségével. A függvény eredménye egy egész szám (12–13. sor). Ha megtalálta a karakterláncot, akkor az első karakter helyzetét adja meg a változóban (12. sor). Ha a karakterlánc többször is előfordul, az első előfordulás helyét adja meg. Ha a karakterlánc nem fordul elő a változóban, -1-et ad meg (13. sor). Így a 14–17. sorok alkalmasak annak eldöntésére, tartalmaz-e egy karakterlánc (sz) egy másikat ("kutya").

```
11) sz = "van benne farkas"
12) print(sz.find("farkas"))          10
13) print(sz.find("kutya"))           -1
14) if sz.find("kutya") > -1:
15)     print("Nincs benne kutya")
16) else:
17)     print("Van benne kutya")
```

Gyakori probléma, hogy a **find** utasítás a kis- és nagybetűt megkülönbözteti. Ha a felhasználó keresi az ég szót, nem fogja megtalálni, ha a mondat elején van, ezért nagybetűs. ("Ég a napmelegtől...") Ezt kiküszöbölendő, érdemes a keresendő karakterláncot, és azt is, amiben keresünk, kisbetűssé (vagy nagybetűssé) alakítani. Kisbetűssé alakít a **lower** utasítás, nagybetűssé az **upper**.

52. mintafeladat – keresés karakterláncban

Legyen adva egy karakterlánc: "Ha a távoli reményében nem törődsz a közelivel, biztosan kudarcot vallasz"[45]. Kérjünk be a felhasználótól egy szót, és írassuk ki, szerepel-e a fenti karakterláncban vagy sem. Gondoskodjunk róla, hogy ne számítson, kis- vagy nagybetűt ad meg a felhasználó.

1. Adjuk meg a mondatot egy változóban *(strin)* (1. sor), amit aztán ki is íratunk a felhasználónak emlékeztetőként (2. sor).

```
1 strin = "Ha a távoli reményében nem törőc
2 print(strin)
3 strf = input("Melyik szót keressük? ")
4 strin, strf = strin.lower(), strf.lower()
5 if strin.find(strf) > -1:
6     print("Benne van a szó!")
7 else:
8     print("Nincs benne a szó!")
```

45 Az idézet Lao-cétől származik.

2. Egy másik *(strf)* változóba bekérjük a felhasználótól a keresendő szót (3. sor).
3. Mindkét változó tartalmát kisbetűssé alakítjuk, hogy ne legyen zavaró, ha valaki véletlenül nagybetűt írt, vagy éppen kicsit, ahol nagybetű van az eredeti szövegben. Például a *ha* szót keresi, de csak *Ha* van a szövegben (4. sor).
4. Kerestetjük a **find** függvénnyel a mondatban *(strin)* a szót *(strf)* (5. sor).
5. Ha megtalálható benne, akkor az érték nagyobb lesz, mint -1 (5. sor).
6. Ez esetben kiíratjuk, hogy "Benne van a szó!" (6. sor).
7. Ha nincs benne a szó, akkor -1-et kapunk, ami nem nagyobb, mint -1, tehát a különben ágra kerülünk (7. sor).
8. Kiírjuk, hogy "Nincs benne a szó!" (8. sor).

Karakterlánc darabolása listává

Gyakran az összetartozó adatok egymás mellett helyezkednek el karakterláncként, elválasztva valamilyen karakterrel, például szóközzel vagy pontosvesszővel[46]:

```
ABC-123 fehér Fiat 2019 2000000
DEF-456 szürke Volvo 2018 2500000
...
```

Az első sor jelentheti például, hogy az ABC-123 rendszámú fehér, 2019-es évjáratú Fiat ára 2000000 Ft.

Ezekre az adatokra általában külön-külön van szükségünk. Ha egy autópiac autóiról van szó, például kíváncsiak lehetünk a kocsik összértékre. Ehhez pedig ki kell bányásznunk valahogy az árakat.

A Python képes tetszőleges elválasztó karakter mentén listává bontani a karakterláncot (jelen esetben a teljes sort) a **split** utasítás segítségével. Az utasítás mögötti zárójelben, idézőjelek között megadhatjuk az elválasztójelet. Például:

```
>>> strin = "egy;bigyó"
>>> datas = strin.split(";")
>>> print(datas)
['egy', 'bigyó']
```

Ha a zárójelek közötti helyet üresen hagyjuk, szóközök mentén történik az elválasztás.

46 Például egy szöveges fájlban vannak így tárolva, ahonnan beolvastuk a sort.

Segítségünkre lesz a továbbiakban a **strip** utasítás, aminek segítségével eltávolíthatók a többszörös szóközök. Amiről nem szólnak a leírások, de talán még fontosabb: a sorvége jelektől (chr(13) chr(10)) is megszabadítja a karakterláncokat. Sorvége jelek a szövegfájlokból történő beolvasáskor kerülnek a karakterláncokba. Ezekben az esetekben érdemes leszedni őket:

```
line = line.strip()
```

53. mintafeladat – karakterlánc darabolása listává

Legyen adva egy karakterlánc: "A bölcs megismeri a világot anélkül, hogy kilépne az ajtón."[47]. Írjuk ki a benne található szavakat egymás alá a képernyőn.

```
1 strin = "A bölcs megismeri a világot
2 strin = strin.replace(",","")
3 words = strin.split()
4 for w in words:
5     print(w)
```

1. Adjuk meg a mondatot egy változóban *(strin)* (1. sor).
2. Töröljük a szövegből a vesszőt. Ha nem tennénk, megjelenne a szó mögött. Ennek legegyszerűbb módja, ha helyettesítjük a vesszőt az üres karakterrel ("semmivel"). Erre a célra szolgál a **replace** függvény (2. sor).
3. A **split** utasítással szétbontjuk a mondatot a szóközök mentén szavakra, és a szavakat a *words* listába helyezzük el (3. sor).
4. A *words* listát a szokott módon bejárjuk, azaz minden elemét kiemeljük a *w* változóba (4. sor), amiket aztán kiíratunk a képernyőre (5. sor).

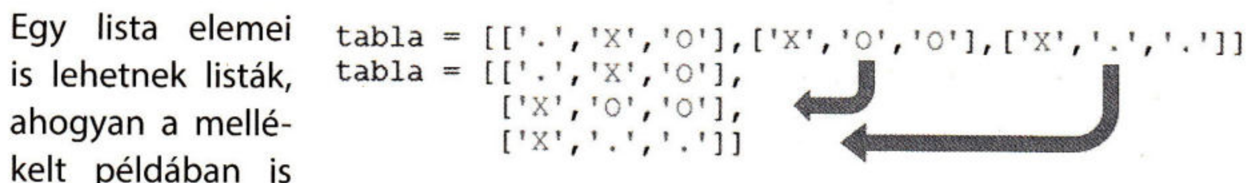
47 Az idézet Lao-cétől származik.

Feladatok

1. Kérjünk be egy szót. Írassuk ki, hogy az angol ábécé valamelyik magánhangzójával kezdődik-e.
2. Egy teszt válaszait karakterláncban tároljuk. Például: ABBCBAAXBC. Ha egy kérdésre nem válaszoltak, vagy a válasz szabálytalan, ott X betű van. Kérjünk be teszt választ, és írjuk ki, minden kérdésre válaszolt-e a vizsgázó.
3. Kérjünk be egy vezetéknévből és keresztnévből álló nevet a felhasználótól, és írassuk ki angolos alakban, azaz a keresztnév legyen elöl, és mögötte a vezetéknév.
4. Adott két időpont, ami ugyanazon a napon van, 21:03:12 (és 4:11:07) alakban. Számítsuk ki a két időpont között eltelt időt, és jelenítsük meg a képernyőn ugyanolyan alakban, mint a bevitel történt. Kerüljük a kódismétlést, alkalmazzunk függvényeket.
5. Adott egy karakterlánc. A karakterlánc elején álló két szám jelölje a megfigyelt farkasok számát. Az első szám a kifejlett példányokat, a második a kölyköket. A sor többi része szöveges üzenet. Határozzuk meg, hány farkast látott az üzenetküldője összesen. A kölykök és a kifejlett példányok számát egyben adjuk meg. Megadtam néhány tesztadatot. Mindegyikre jól kell működnie a programnak. (Nem lehet tudni, hány jegyűek a számok.)
   2/3 ro#s#
   0/11 # ##t
   13/2 sotete

5. LISTÁBAN A LISTA, TÖBBDIMENZIÓS LISTA

Bevezetés



Egy lista elemei is lehetnek listák, ahogyan a mellékelt példában is láthatjuk. A *tabla* lista 0. eleme a ['.' , 'X' , 'O'] lista, annak 1. eleme az 'X'. Mikor egy ilyen elemre hivatkozunk, mindig kívülről befelé haladunk, azaz a nagyobb lista felől a kicsi felé: A tabla lista 0. elemének az 1. eleme, tehát: tabla[0][1].

Ha az ábra szerint a befoglaló lista elemeit egymás alá rakjuk a nyilak szerint, az elemek téglalapszerű elrendezését kapjuk.

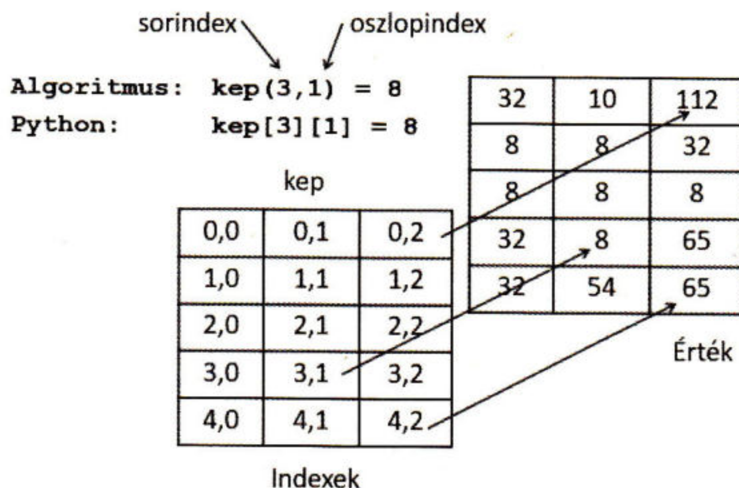
Az elemek téglalapszerű elrendezését kétdimenziós listának vagy mátrixnak nevezhetjük.

Mátrixként nézve az elrendezésre, az **első index** a táblázat **sorát** adja meg, ahol a hivatkozott elem található, 0-val kezdődően. Ezért **sorindexnek** is nevezzük. A **második index** az **oszlopot** határozza meg, szintén 0-tól kezdődően, tehát **oszlopindexnek** hívjuk.

Lehetséges a listában listákat, majd azon belül is további listákat tárolni, így három-, vagy akár többdimenziós listákat is kaphatunk.

A Pythonban a listaelemként szereplő listáknak nem kell egyforma elemszámúaknak lenniük. Így az elemek elrendezése nem kötelezően téglalap. Azonban ezt használjuk a legtöbbször, ezért csak ezzel az esettel foglalkozunk a továbbiakban.

A programozáselméletben, illetve más programnyelvekben az elemek téglalap alakú elrendezését kétdimenziós tömbnek is nevezik. A mondatszerű leírásban az indexeket egyetlen kör alakú zárójelpáron belül, egymástól vesszővel elválasztva tüntetjük fel.



Kódolás szempontjából a fenti átrendezés (sorokra tördelés) elvégezhető, mert a Python a szögletes- vagy kapcsos zárójelek közötti részt akkor is egyben értelmezi, ha közben sor vége volt. Fokozottan ügyeljünk a vesszők és szögletes zárójelek meglétére és számára.

Külön probléma a mátrix elkészítése, ha kezdetben nem adunk értéket az elemeknek. Sajnos a mindenki számára logikus matrix = [[0]*20]*20 rossz megoldást ad. A működő megoldás lényege, hogy a mátrix egy sorát elkészítjük a már megtanult módon, majd az így keletkezett sorral bővítjük (**append**) az üres mátrixot.

54. mintafeladat – kétdimenziós lista

Egy 3 x 3-as amőbatábla tetszőlegesen megadott állapotát tároljuk kétdimenziós listában. Lehetséges értékek: X, O, és jelöljük .-tal az üres mezőket. Jelenítsük meg a táblát a képernyőn.

1. Adjuk meg az elrendezést egy *tabla* nevű listában (1–3. sor). Ügyeljünk rá, hogy az utolsó sor végén nincs vessző a két záró zárójel között, mivel a vessző elválasztójel, és ott már nincs mitől elválasztania az utolsó elemet. Ügyeljünk rá, hogy két nyitó zárójellel kezdődik és két záróval végződik a mátrix. A külső zárójelek a befoglaló ("nagy") lista zárójelei, a belsők az első listaelem nyitó, illetve az utolsó listaelem záró zárójele.
2. Téglalap alakú elrendezések bejárásához a ciklusban ciklus szerkezetet használjuk, mint tettük a szorzótábla kiírásakor (4–5. sor). A 4. sorban található ciklus szalad végig a befoglaló lista elemein, azaz a sorokon, az 5. sorban található ciklus pedig a befoglalt listák elemein, tehát az oszlopokon.

```
1 tabla = [['.','X','O'],
2          ['X','O','O'],
3          ['X','.','.']]
4 for i in range(0,3):
5     for j in range(0,3):
6         print(tabla[i][j], sep="", end="")
7     print()
```

3. A ciklusok mindkét esetben 0-tól 2-ig futnak, mivel a mátrix mindkét irányban 3–3 elemű.
4. Minden elemet kiíratunk. Egy soron belül szóközök és sorvége-jelek nélkül (6. sor).
5. Minden táblázatsor kiíratása után új sort kezdünk (7. sor).

55. mintafeladat – kétdimenziós lista feltöltése

Egy 20 x 20-as amőbatáblát töltsünk fel véletlenszerűen adatokkal. Lehetséges értékek: X, O, és jelöljük .-tal az üres mezőket. Jelenítsük meg a táblát a képernyőn.

```
 1 from random import randrange
 2 table = []
 3 for i in range(0,20):
 4     row = []
 5     for j in range(0,20):
 6         row.append(randrange(0,3))
 7     table.append(row)
 8 fig = ['.', 'O', 'X']
 9 for i in range(0,20):
10     for j in range(0,20):
11         print(fig[table[i][j]], sep="", end="")
12     print()
```

1. Elérhetővé tesszük a **randrange** utasítást, mert nem része az alap utasításkészletnek (1. sor).
2. Létrehozunk egy *table* nevű listát. Ez lesz majd a mátrix (2. sor).
3. A külső ciklus (3. sor) minden egyes lefutása létrehoz egy *row* nevű listát (4. sor), amiben a táblázat egy-egy sora elkészül (5–6. sor), majd a mátrixba töltődik (7. sor).
4. Egy sor úgy készül, hogy feltöltjük 20 db (5. sor), 0–2 közötti véletlen egésszel (6. sor). Csak véletlen számot tudunk készíteni közvetlenül, véletlen karaktert nem. Hogy melyik szám melyik karaktert jelenti, azt még a kijelzés előtt rögzítjük (8. sor).
5. A *fig* listában felsoroljuk a lehetséges karaktereket. Így a nullás indexű lesz a ., az 1-es a O, a 2-es az X (8. sor).
6. A tábla (mátrix) bejárásához a ciklusban ciklus szerkezetet használjuk (9–10. sor). A 9. sorban található ciklus szalad végig a befoglaló lista elemein, azaz a sorokon, az 10. sorban található ciklus pedig a befoglalt listák elemein, tehát az oszlopokon.
7. Minden elemet kiíratunk. Egy soron belül szóközök és sorvége jelek nélkül. Önmagában a table[i][j] a mátrixban tárolt számokat jelenítené meg egymás után sorfolytonosan. Ha ezt a *fig* tömb indexébe helyettesítjük, pont a megfelelő karaktereket kapjuk (11. sor).
8. Minden táblázatsor kiíratása után új sort kezdünk (12. sor).

Feladatok

1. Egy karakter 8 x 8 képpontból áll, amit egy mátrixban tárolunk. Alakítsunk ki a 8 x 8 képpontra egy karaktert (például A betűt). A képpontok karakter méretűek lesznek, és maguk is karakterekből állnak. Készítsük el a definíciót, és jelenítsük meg a képernyőn. A pontok helyett lehetnek szóközök, ha úgy jobban tetszik. (Megismételhetjük más karakterekre is, további gyakorlásként. Meg lehet próbálni "képeket" rajzolni karakterekből, más méretben is. Ennek nagy divatja volt egy időben.)

```
...AA...
..AAAA..
.AA..AA.
AA....AA
AAAAAAAA
AAAAAAAA
AA....AA
AA....AA
```

2. Jelölje K (nagy K betű) a világos királyt, B (nagy B betű) a világos bástyát, és k (kis k betű) a sötét királyt. A bábok állását egy 8 x 8-as mátrixban tároljuk. Legyen a fenti bábok helyzete véletlenszerű. Jelenítsük meg a képernyőn a táblát karakterekkel. Amelyik mezőn nem áll semmi, azt jelölje pont karakter. (Ha már készen van, tegyük úgy életszerűbbé, hogy a két király nem lehet egymás mellett, mert a sakkban nincs királyközelítés.)

6. LISTÁK AZ ELJÁRÁSOKBAN ÉS FÜGGVÉNYEKBEN

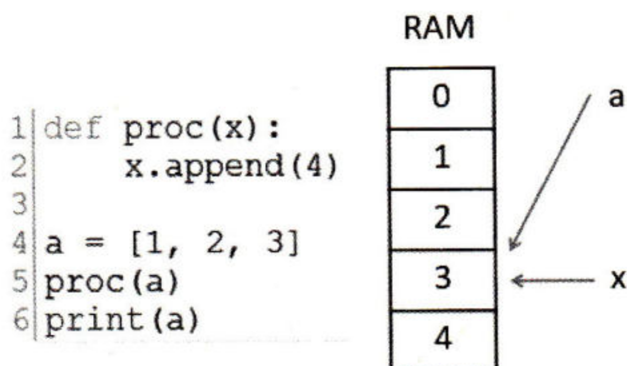
Láttuk, hogy a függvények mindig csak egy adatot adhatnak vissza a főprogramnak. A Python azonban megengedi, hogy az az érték összetett típusú legyen, tipikus esetben lista. Így mégis át tud venni több adatot a főprogram a függvénytől, egy listába ágyazva.

Át lehet adni a listát paraméterként mind az eljárások, mind a függvények esetében. Ehhez csak a nevüket kell beírni. Semmiféle szögletes zárójelre, listaméret megadásra nincs szükség sem az eljárás paraméterénél, sem a meghíváskor. Azonban a paraméterátadás mindig referencia szerint történik, ami eltér az eddig megtanult érték szerintihez képest.

Cím (referencia) szerinti paraméterátadás

A cím szerinti átadásnál a főprogram és az alprogram a változó nevéhez *ugyanazt* a memóriaterületet rendeli. Tehát csak látszólagosan létezik két példányban az adat, ami valójában ugyanaz a memóriaterület két névvel.

A program futásakor először a főprogram változója jön létre, amihez hozzárendelődik egy memóriaterület (az *a* változónak a 3. memóriarész). Aztán az alprogram meghívásakor létrejön egy új név (x), ami ugyanarra a memóriaterületre mutat, mint a főprogrambeli párja. Így *x* értéke ugyanaz lesz, mint *a* értéke. Ha megváltoztatjuk x értékét, az a közös memóriaterületen változik meg, ezért a változás hatással lesz a főprogramunkban használt változó értékére is.



Azért nevezik referencia szerinti átadásnak, azaz hivatkozás szerintinek, mert az alprogramban lévő változónév egy már létező memóriaterületre hivatkozik.

Ennél a paraméterátadásnál nincs értelme a *proc(27)* típusú referencia szerinti meghívásának, azaz ha nem változó van megadva paraméterként. Hibaüzenetet fogunk kapni, mivel nincs hová visszaadni a paramétert.

Kövessük végig az ábrán található kis mintaprogramot.

1. A 4. sorban indul a végrehajtás, mert ez a főprogram első sora. Az *a* változó értéke az [1, 2, 3] lista.
2. Meghívjuk a proc eljárást (5. sor), az *a* változót használva paraméterül.
3. A program végrehajtása az 1. sortól folytatódik, ahol az *a* változó memóriaterületéhez fog csatlakozni a most létrejött *x* változó, ami a korábban létrehozott listát tartalmazza.
4. A 2. sorban az x listához hozzáfűzünk egy új elemet, a 4-et. Mivel csak egy listánk van két névvel, az *a* változó irányából ugyanezt a módosult listát látjuk majd.
5. A program végrehajtása a főprogram következő sorától (6.) folytatódik, ahol kiírjuk az *a* változó értékét. Mivel az *a* változó ugyanaz, mint az *x*, csak más néven, így a kiírt lista: [1, 2, 3, 4] lesz.

Paraméterátadások a Pythonban

Kétféle paraméterátadást ismertünk meg. Az eljárásoknál az érték szerintit, most a referencia szerintit. A Python mindkettőt használja, és nem mi döntünk arról, mikor melyiket. Egyszerű változóknál vagy konkrét értékeknél az érték szerintit, összetett változóknál a referencia szerintit alkalmazza.

Ebben az a logika, hogy az összetett változók érték szerinti átadásához azokból még egy példányt kellene létrehozni, ami már tekintélyes helypazarlás lenne (egy lista tízezer számra tartalmazhat adatot). Emellett lassítaná a programot, amíg egyik listából a másikba a tetemes mennyiségű adatot áttöltenénk.

Ha az eljárás vagy függvény paramétere összetett változó (lista), az átadás referencia szerinti, így a főprogramban található változót az alprogram módosíthatja.

Ha a paraméter egyszerű változó vagy konkrét érték, az átadás érték szerinti, így a főprogramban található változó nem módosul.

Ha egyszerre adunk át listát és egyszerű változót, mindkét átadás referencia szerinti lesz.

A karakterlánc egyszerű változónak számít, azaz érték szerinti átadással kerül az alprogramba.

56. mintafeladat – paraméterátadások referencia szerint

Készítsünk függvényt, ami feltölt egy üres listát véletlenszerű fej vagy írás elemekkel. A függvény paraméterben kapja meg, hány elemű legyen a lista. Készítsünk eljárást, ami megcseréli a paraméterként kapott lista első és utolsó elemét. Készítsünk egy eljárást, ami a paraméterként kapott lista elemeit kiírja a képernyőre, egymástól szóközzel elválasztva. Kapcsoljuk össze a három modult egy főprogrammal: készüljön egy véletlenszerű fej-írás lista, amit először kiíratunk, majd megcseréljük az első és utolsó elemét, végül ismét kiíratjuk.

1. Elérhetővé tesszük a **randrange** utasítást, mert nem része az alap utasításkészletnek (1. sor).
2. A jó tesztelhetőség érdekében először általában a megjelenítő eljárást érdemes elkészíteni (3–6. sor).

```
1 from random import randrange
2
3 def listout(olist):
4     for i in olist:
5         print(i, end=' ')
6     print()
```

3. Az eljárás a **def** kulcsszóval kezdődik. Mögötte megadjuk az eljárás nevét *(listout)*, és zárójelek között a paraméterként átvett lista nevét *(olist)*, majd a vezérlési szerkezeteknél kötelező kettőspontot (3. sor).
4. Az *olist* nevű listát bejárjuk a **for** ciklussal (4–5. sor). Minden lefutáskor a lista egy-egy még sorra nem került eleme az *i* változóba kerül (4. sor), amit megjelenítünk a képernyőn (5. sor).
5. Az eljárás végén új bekezdést kezdünk egy üres **print** utasítással (6. sor).
6. Az eddig elkészült programrészlet máris tesztelhető, ha a következő sorok valamelyikében egy konkrét listával meghívjuk az eljárást. Például írjuk be:
   ```
   listout(["fej","írás"])
   ```
   Majd futtassuk a programot.
7. Következő lépésben általában az adatot készítő modult (most függvényt) érdemes elkészíteni (8–16. sor).
8. A függvény a **def** kulcsszóval kezdődik. Mögötte megadjuk az eljárás nevét *(gen)*, és zárójelek között paraméterként a darabszámot *(c)*, majd a vezérlési szerkezeteknél kötelező kettőspontot (8. sor).

```
 8 def gen(c):
 9     glist = []
10     for i in range(c):
11         rnd = randrange(0,2)
12         if rnd % 2 == 0:
13             glist.append("fej")
14         else:
15             glist.append("írás")
16     return glist
```

9. Készítünk egy új listát *glist* néven (9. sor).
10. Egy számláló ciklussal (**for**) ismételtetjük a következő műveleteket (11–15. sor), ami 0-tól indul, és c-1-ig tart, tehát c db elemünk lesz (10. sor.).
11. A ciklus minden menetében készítünk egy 0...1 véletlen egész számot (11. sor).
12. Ha a véletlenszám páros (kettővel osztva 0-t ad maradékul) (12. sor), a listához hozzáadunk egy új elemet, ami a fejet tartalmazza (13. sor).
13. Különben (14. sor), a listához hozzáadunk egy új elemet, ami az írást tartalmazza (15. sor).
14. Végül visszaadjuk a programnak az elkészített listát (16. sor).
15. Az eddig elkészült programrészlet máris tesztelhető, hiszen már van megjelenítő eljárásunk. A következő sorok valamelyikétől kezdődően hozzunk létre egy üres listát, töltsük bele, amit a *gen* függvény ad, majd jelenítsük meg a *listout* eljárással. Például írjuk be:
    ```
    mlist = []
    mlist = gen(10)
    listout(mlist)
    ```
    Majd futtassuk a programot.
16. Utolsó modulként érdemes az adatmanipuláló eljárással foglalkozni (18–20. sor).
17. Az eljárás a **def** kulcsszóval kezdődik. Mögötte megadjuk az eljárás nevét *(swap)*, és zárójelek között a paraméterként átvett lista nevét *(slist)*, majd a vezérlési szerkezeteknél kötelező kettőspontot (18. sor).

18. Megállapítjuk a lista elemszámát, majd ebből 1-et kivonva az utolsó elem indexét (19. sor).
19. Megcseréljük a 0. és az utolsó elemet (20. sor).
20. Már csak a főprogram véglegesítése van hátra (22–26. sor).

```
18 def swap(slist):
19     n = len(slist) - 1
20     slist[0],slist[n] = slist[n],slist[0]
21
22 mlist = []
23 mlist = gen(10)
24 listout(mlist)
25 swap(mlist)
26 listout(mlist)
```

21. Mivel a paraméterátadás referencia szerint történik listák esetében, *mlist, glist, slist, olist* ugyanazt a listát jelentik, csak különböző nevekkel hivatkoztunk rájuk. Így mikor a csere megtörténik az slist-ben, az mlist-ben is megtörténik egyúttal.
22. A főprogramban készítünk egy üres listát (22. sor, *mlist*), majd feltöltjük 10 adattal a *gen* függvény segítségével (23. sor).
23. A feladat szövegének megfelelően megjelenítjük az eredeti listát a *listout* eljárás meghívásával (24. sor).
24. Megcseréljük az első és utolsó elemet a *swap* eljárás segítségével (25. sor), majd kiírjuk a már megváltozott listát a *listout* eljárással (26. sor).

Feladatok

1. Készítsünk függvényt, ami létrehoz számjegyekből véletlenszerűen egy számzárhoz való kódsorozatot (azaz például a 049 is megengedett). A függvény paraméterben kapja meg, hány elemű legyen a kódsorozat. Készítsünk eljárást, ami eggyel balra tolja az összes számjegyet. Az első számjegy kerüljön a kódsorozat végére. Készítsünk egy eljárást, ami kiírja a képernyőre a kapott kódsorozatot a fentebb megadott alakban (szóközzel elválasztva). Kapcsoljuk össze a három modult egy főprogrammal: készüljön egy véletlenszerű kód, amit először kiíratunk, majd balra toljuk a számjegyeit, végül ismét kiíratjuk a képernyőre.
2. Egy karakter 8 x 8 képpontból áll, amit egy mátrixban tárolunk. Alakítsunk ki a 8 x 8 képpontra egy karaktert (például A betűt). A képpontok karakter méretűek lesznek, és maguk is karakterekből állnak. Készítsük el a 8 x 8 karakter mátrixát a főprogramban. Eljárás segítségével jelenítsük meg a képernyőn. Egy másik eljárás segítségével tükrözzük a karaktert vízszintesen, és jelenítsük meg újra.

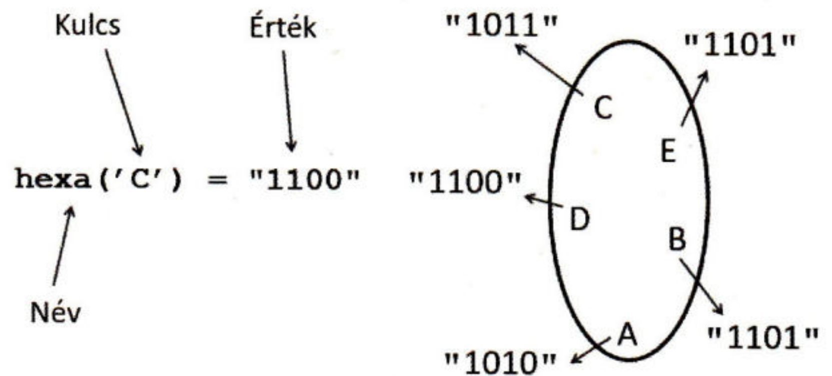
```
...AA...
..AAAA..
.AA..AA.
AA....AA
AAAAAAAA
AAAAAAAA
AA....AA
AA....AA
```

7. SZÓTÁRAK

Áttekintés

Szótár esetén az index szerepét a **kulcs** veszi át. A kulcs **nem feltétlenül szám** típus. Lehet egyetlen karakter, karakterlánc vagy akár lista is.

A szótár *kulcs* → *érték* párokból áll. A kulcs alapján tudunk hozzáférni az értékhez. A kulcsok között keresni gyors művelet, akár a kulcshoz tartozó értéket megkeresni. Fordított irányban (tehát az értékek között keresni, majd abból vissza a kulcsot) a művelet körülményes.



A szótár egy viszonylag fiatal programozási fogalom, ezért több neve is van. Hívják még asszociatív tömbnek, hashtable-nek, stb.

Hasonló a listához annyiban, hogy hivatkozni tudunk valamin (a kulcsokon) keresztül az elemeire. Különböző abban, hogy nem egy sorozat, tehát **az elemeknek nincs sorrendje**. A kulcsok egy halmazszerű elrendezést alkotnak. Nem mondható meg, melyik elemet melyik követi.

A szótár bejárható abban az értelemben, hogy a kulcshalmaz minden tagja beolvasható egy változóba (csak a kulcsok sorrendje ismeretlen), így a hozzájuk tartozó értékek is megkaphatók.

Kódolás

A szótárakat kapcsos zárójelek { } határolják. A kulcs-értékpárok közé vesszőt írunk, a kulcsot az értéktől kettősponttal választjuk el (1–4. sor).

```
1 change = {'0': "0000",'1': "0001", '2': "0010",'3': "0011",
2            '4': "0100",'5': "0101",'6': "0110",'7': "0111",
3            '8': "1000",'9': "1001",'A': "1010",'B': "1011",
4            'C': "1100",'D': "1101",'E': "1110",'F': "1111"}
5 v = input("Számjegy a 16-osban: ")
6 print("Ugyanez a 2-esben: ", change[v])
7 for key in change:
8     print (key, change[key])
9 print(change)
```

A szótár elemeire formailag úgy hivatkozunk, mint a lista elemeire. A szótár neve mögött szögletes zárójelek között megadjuk a kulcsot: change[v] (6. sor).

A szótárt bejárni, tehát összes kulcsát és értékét megjeleníteni **for** ciklussal lehet. A **for** utáni változóba fognak kerülni a kulcsok. A ciklus minden egyes lefutásakor egy olyan kulcs, amire előzőleg még nem került sor. Az **in** mögött kell megadni a szótár nevét (7. sor). Így mind a kulcshoz, mind az értékhez külön hozzá tudunk férni, tehát bármilyen formázott megjelenítést alkalmazhatunk (8. sor).

Tesztelési célokra időnként jól jön, ha a szótárt egyben meg tudjuk jeleníteni. Ezt megvalósíthatjuk egyszerűen a **print** paranccsal (9. sor).

Szótárak összetett kulccsal

A kulcsok lehetnek összetettek, így a kulcs részeit különkülön változtatni tudjuk. A tuple adattípus tökéletesen megfelel az összetett adattípusú kulcsnak. Például: table[(1,2)]='O'. Mivel a tuple-k esetében a zárójelek elhagyhatók, az értékadás így is helyes: table[1,2]='O'. A két kulcsrészt külön változtatva egy mátrixot kaphatunk (2–6. sorok).

```
 1 table = {}
 2 table [5,5] = 'O'
 3 table [4,5] = 'O'
 4 table [5,6] = 'X'
 5 table [4,4] = 'X'
 6 table [5,4] = 'X'
 7 for i in range(10):
 8     for j in range(10):
 9         print(table.get((i,j),'.'), end='')
10     print()
```

Komoly különbség a listából készített mátrixhoz képest, hogy itt nem kell minden közbülső elemet megadni. Ez előnyös lehet, ha csak pár elem van. Nem kell a néhány elem miatt a teljes mátrixnak megfelelő memóriaterületet lefoglalni. Ha van egy 10 x 10-es pályánk, amin csak 5 figura áll, akkor 10 x 10 – 5 = 95 elemnek feleslegesen foglalnánk helyet egy listából készült mátrixban.

```
..........
..........
..........
..........
....XO....
....XOX...
..........
..........
..........
..........
```

Problémát jelent egy ilyen szótár-mátrix bejárása, mert a meg nem adott kulcsokra hivatkozva hibaüzenetet kapunk. Megoldást kínál erre a **get** utasítás, ami lekéri a szótárból a hivatkozott értéket, ha létezik a kulcs. Ellenkező esetben a "hiányt" pótolja egy megadható karakterrel, számmal vagy egy változó értékével (9. sor).

57. mintafeladat – ismerkedés a szótárakkal

1. Álljunk a parancsértelmező ablakba (**Python 3.8.0 Shell**).
2. A prompt jel >>> mögött villog a kurzor. Oda írjuk majd be a parancsokat, mindegyik után megnyomva az **Enter** billentyűt. Ha többsoros parancsot írunk, az utolsó sor végén **kétszer** kell **Entert** használni.
3. Készítsünk egy *change* nevű szótárt, ami tartalmazza a tizenhatos számrendszerbeli A–F számjegyek átírását kettes számrendszerbe. Írjuk be, majd üssünk **Entert**:

```
change = {'A': "1010",'B': "1011",
      'C': "1100",'D': "1101",
 'E': "1110",'F': "1111"}
```

4. Írassuk ki, mi tartozik a D-hez, tehát a 14-hez. Írjuk be, majd üssünk **Entert**:

```
print(change['D'])
```

5. Ha az 1101-et kaptuk, jól dolgoztunk.
6. Kérjünk be a felhasználótól egy A...F betűt egy v változóba. Írjuk be, majd üssünk **Entert**:

```
v = input("Számjegy a 16-osban: ")
```

7. Az Enter megnyomása után megjelenik a kérdés, és a parancsértelmező várakozik a betű megadására. Írjuk be, majd üssünk **Entert**:

```
Számjegy a 16-osban: B
```

8. Írassuk ki a képernyőre az átváltott értéket úgy, hogy bármilyen megadott számjegy esetén működjön (tehát változót használva). Írjuk be, majd üssünk **Entert**:

```
print("Ugyanez a 2-esben: ", change[v])
```

9. Ha a B-t írtuk be, az 1011-et kellett kapnunk.
10. Most írassuk ki a képernyőre az egész szótárt, mert például elfelejtettük, mit is tartalmaz pontosan. Írjuk be, majd üssünk **Entert**:

```
print(change)
```

11. Ezt kell látnunk a képernyőn:
    {'A':'1010', 'B':'1011','C':'1100','D':'1101','E':'1110','F': '1111'}
12. Járjuk be a szótárt, azaz írjuk ki formázottan a kulcs-érték párokat. Soronként egy pár jelenjen meg, a kulcs és az érték között szóköz legyen. A **for** ciklussal járjuk be a szótárt. A változóba fognak kerülni a kulcsok, minden egyes ciklus lefutáskor másik. Írjuk be, majd üssünk **Entert**:

```
for key in change:
```

13. A parancsértelmező észreveszi a kettőspontot a sor végén, ezért négy szóközzel beljebb kezdi a következő sort, és nem ad prompt jelet, hiszen folytatni fogjuk a beírást. Még a kiíratás hiányzik. Írjuk be, majd üssünk *kétszer* **Entert**:

```
print (key, change[key])
```

14. A képernyőn minden kulcs-érték pár meg fog jelenni az alábbi mintarészlet szerint:
    A 1010
    B 1011
    ...

58. mintafeladat – ismerkedés a szótármátrixokkal

1. Álljunk a parancsértelmező ablakba (**Python 3.8.0 Shell**).
2. A prompt jel >>> mögött villog a kurzor. Oda írjuk majd be a parancsokat, mindegyik után megnyomva az **Enter** billentyűt. Ha többsoros parancsot írunk, az utolsó sor végén **kétszer** kell **Entert** használni.
3. Készítsünk egy *table* nevű üres szótárt. Írjuk be, majd üssünk **Entert**:
   ```
   table = {}
   ```
4. Adjuk meg három elemét, mátrixként használva. Minden sor után üssünk **Entert**:
   ```
   table [5,5] = 'O'
   table [4,5] = 'O'
   table [5,6] = 'X'
   ```
5. Írassuk ki a 4. sor, 5. oszlopban található figurát. Írjuk be, majd üssünk **Entert**:
   ```
   print(table[4,5])
   ```
6. Ha jól dolgoztunk, egy O betűt kaptunk válaszként.
7. Írassuk ki a 1. sor, 2. oszlopban található figurát. Persze nincs ott semmi, csak kíváncsiak vagyunk, mi történik. Írjuk be, majd üssünk **Entert**:
   ```
   print(table[1,2])
   ```
8. Hibaüzenetet kapunk, mert nem létezik ilyen kulcs: KeyError: (1, 2)
9. Próbáljuk ki a **get** paranccsal. Ha nem létezik a kulcs, kapjuk eredményként a pont karaktert. Írjuk be, majd üssünk **Entert**:
   ```
   print(table.get((1,2),'.'))
   ```
10. Ha megfelelően gépeltünk, egy pont karaktert kaptunk eredményül.
11. Most próbáljuk ki, mit tesz a **get** parancs, ha létezik a kulcs. Írjuk be, majd üssünk **Entert**:
    ```
    print(table.get((4,5),'.'))
    ```
12. Ha megfelelően gépeltünk, most az O betűt kaptuk eredményül.
13. Nézzük meg, hogy lehet bejárni egy szótármátrixot. Tételezzük fel, hogy a mátrix 10 x 10-es. Két számláló ciklus biztosan kell hozzá egymásba ágyazva, hiszen ezek fogják változtatni a kulcsrészeket. Kezdjük a sorokat bejáró ciklussal. Írjuk be, majd üssünk **Entert**:
    ```
    for i in range(10):
    ```
14. A parancsértelmező észreveszi a kettőspontot a sor végén, ezért négy szóközzel beljebb kezdi a következő sort, és nem ad prompt jelet, hiszen folytatni fogjuk a beírást. Folytassuk a soron belül az oszlopokat bejáró ciklussal. Írjuk be, majd üssünk **Entert**:
    ```
    for j in range(10):
    ```
15. A parancsértelmező észreveszi a kettőspontot a sor végén, ezért még négy szóközzel beljebb kezdi a következő sort, és nem ad prompt jelet, hiszen folytatni fogjuk a beírást. Folytassuk az értékek kiíratásával. Írjuk be, majd üssünk **Entert**:
    ```
    print(table.get((i,j),'.'), end='')
    ```

16. Még hiányzik a sorok végének tördelése. Ehhez egyszer meg kell nyomnunk a visszatörlő billentyűt (többnyire egy hosszú balra mutató nyíl van rajta). Visszaugrik 4 karakternyit a kurzor, jelezvén, hogy amit írunk, az első ciklus magjába kerül. Írjuk be, majd üssünk *kétszer* **Entert**:

```
print()
```

17. Megjelenik a 10 x 10-es mátrix, ami javarészt pontokból áll, középtájon a három figurával.

Feladatok

1. Készítsünk programot, ami átszámol egy tizenhatos számrendszerben megadott számot kettes számrendszerbe. A tizenhatos számrendszerbeli számot a felhasználó adhatja meg.
2. Folytassuk az előző feladatot azzal, hogy megfordítjuk az ott megadott szótárt. A kulcsokból legyenek értékek, és az értékekből kulcsok. Ellenőrzésként írassuk ki az új szótárt a kulcs -> érték alakban. Minden kulcs-érték pár külön sorban legyen.
3. Folytassuk az előző feladatot úgy, hogy képes legyen átszámolni a felhasználó által 8 helyiértéken megadott kettes számrendszerbeli számot tizenhatos számrendszerbe.
4. A mellékelt táblázat[48] azt mutatja, hogy egy hét alatt hány közvetlen repülőgépjárat van az első oszlopban tárolt városból a második oszlopban tárolt városba. A harmadik oszlopban található a járatok száma. Tároljuk szótár segítségével az adatokat, majd jelenítsünk meg egy olyan mátrixot a képernyőn, ami megmutatja az összes lehetséges kapcsolatot.[49] Írjunk 0-t oda, ahol nincs közvetlen repülőjárat.

indul	érkezik	db
1	2	1
1	3	2
2	1	1
5	2	4
1	4	1
2	5	2
4	1	2
3	4	1

48 A szaknyelv az ilyen táblázatot élmátrixnak nevezi. Amennyiben a városokat egy gráf csúcsainak fogjuk fel, a köztük lévő utakat pedig a gráf éleinek, egy ilyen

49 A szaknyelv az ilyen táblázatokat csúcsmátrixnak, ritkábban szomszédsági mátrixnak nevezi, szintén a gráfok miatt.

8. REKORDOK

Fogalma

Rekordban a szorosan összetartozó adatokat tároljuk. Például egy autó adatait: rendszám, évjárat, márka, szín, ár, vagy egy személy adatait: név, születési dátum, gyerekek száma.

A rekord tehát nem feltétlenül azonos típusú elemi adatokból álló összetett adattípus, amiben összetartozó adatokat tárolunk.

A rekord egy adattípus, tehát úgy tekintsünk rá, mint egy sablonra, ami alapján később változókat hozhatunk létre. A rekordtípus önmagában nem alkalmas adattárolásra. Majd a típus alapján létrehozott változók fognak tudni adatot tárolni.

A példában két közönséges tört közötti műveletet tárolunk egy rekordban (1–4. sor). A példában a rekordtípus neve *Frops* (1. sor). A rekord részeit mezőknek nevezzük. Ezzel a névvel utalunk arra, hogy egy nagyobb egység részei, és nem több önálló egyszerű adattípus. Egyszerű törtek közötti bármilyen művelethez szükséges a két tört számlálója és nevezője (sz1, n1, sz2, n2). Ezek egész típusú *mezők* (2. sor). A közöttük lévő műveletet pedig az *op* mezőben tároljuk, ami karakter típusú mező (3. sor).

	int	int	int	int	char
buff	sz1	n1	sz2	n2	op
	↓	↓	↓	↓	↓
	30	12	8	8	+

```
1) Frops = rekord
2)    sz1,n1,sz2,n2 : egész
3)    op : karakter
4) rekord vége
5) buff, temp : Frops
6) buff.sz1 = 30
7) Ki: buff.sz1
8) temp.n1 = buff.n2
9) temp = buff
```

Miután a típust létrehoztuk, konkrét adattárolásra alkalmas változókat deklarálhatunk (5. sor). A példában a változók neve *temp* és *buff*.

A rekord mezőire hivatkozhatunk egyesével: a változó neve után írjuk ponttal elválasztva a mező nevét. A 6. sorban például a *buff* változó *sz1* mezőjének a 30 értéket adjuk. A 7. sorban ugyanennek a változónak az értékét kiíratjuk a képernyőre.

A rekord használatának igazi hatékonyságát a 9. sor sor mutatja. Egy teljes rekord minden mezőjének egyetlen utasítással adhatunk értéket, nem kell mezőnként megtenni. A példában a *buff* rekord tartalmát átrakjuk a *temp* rekordba is. A *buff.sz1* mező tartalmát a *temp.sz1* mezőbe, a *buff.n1* tartalmát a *temp.nev1* mezőbe, stb. Ennek megfelelően ilyen jellegű értékadás csak azonos típusú rekordok között történhet.

	sz1	n1	sz2	n2	op
buff	30	12	8	8	+
⬇	↓	↓	↓	↓	↓
temp					

A rekordok mezői között továbbra is vihetünk át adatokat egyesével (8. sor).

Rekord típusú listákat is létrehoztunk, bennük akár olyan táblázatos elrendezésű adatokat tárolva, ahol az adattípus oszloponként eltérő.

frop	sz1	n1	sz2	n2	op
0	30	12	8	8	+
1	8	20	8	30	*
2	10	4	12	3	+

Kódolás

A Pythonban nincs kimondottan rekord adattípus, mint más nyelvekben. Mégis van lehetőségünk a használatára. Ha az objektumoknak csak az adatmezőit használjuk fel, rekordokat kapunk.

Az objektumokat ugyanúgy használjuk fel, ahogyan a rekordokat. Tehát először létrehozzuk a rekordtípust, ami megfelel az objektumosztálynak, vagy röviden osztálynak (1–2. sor). Majd a rekordtípus, azaz az osztály alapján létrehozzuk a rekordokat, amiket objektumoknak vagy példányoknak nevezünk (3–6. sor).

Mivel a Python gyengén típusos nyelv, nem kell deklarálni előre a mezőket, azok az első használatkor létrejönnek (a példában az *sz1* a 7. sorban) maguktól. Így a rekordtípus (azaz az osztály) létrehozásához csak a nevét (1. sorban *Frops* a név) kell megadni a **class** utasítás mögött, és a **pass** utasítással jelezni, hogy az objektum többi részét (a később tárgyalandó metódusokat) nem használjuk (2. sor).

```
 1 class Frops():
 2     pass
 3 buff = Frops()
 4 frop = []
 5 for i in range(100):
 6     frop.append(Frops())
 7 buff.sz1 = 8
 8 print (buff.sz1)
 9 frop[0] = buff
10 print (2 * frop[0].sz1)
```

Egyetlen rekordot (objektumot, példányt) könnyű létrehozni: megadjuk a rekord nevét (a 3. sorban például *buff*), és egyenlőségjel után odaírjuk, melyik rekordtípusból származik *(Frops)*.

Kicsit bonyolultabb a rekordlista létrehozása. Először létrehozunk egy üres listát (4. sor), majd a rekordtípus alapján hozzáadjuk a listához a rekordot (6. sor) az **append** utasítás segítségével. Mindezt annyiszor ismételjük, ahány elemű listát szeretnénk (5. sor).

A 7. sorban láthatjuk, hogyan kell a rekord egy mezőjének értéket adni. A 9. sorban a *buff* rekord minden mezőjét egyetlen utasítással áttöltjük a *frop[0]* rekordba is. A 8. és 10. sorokban láthatjuk, hogy írathatjuk ki a rekordok egy-egy mezőjét, illetve hogyan használhatjuk fel számolásra (10. sor).

59. mintafeladat – ismerkedés a rekordokkal

1. Álljunk a parancsértelmező ablakba (**Python 3.8.0 Shell**).
2. A prompt jel >>> mögött villog a kurzor. Oda írjuk majd be a parancsokat, mindegyik után megnyomva az **Enter** billentyűt. Ha többsoros parancsot írunk, az utolsó sor végén **kétszer** kell **Entert** használni.
3. Készítsünk egy *Frops* nevű rekordtípust (osztályt), ami ezt tartalmazza. Írjuk be, majd üssünk **Entert**:
```
class Frops():
```
4. A parancsértelmező észreveszi a kettőspontot a sor végén, ezért négy szóközzel beljebb kezdi a következő sort, és nem ad prompt jelet, hiszen folytatni fogjuk a beírást. Még annyi hiányzik a rekordtípus létrehozásához, hogy be kell írnunk a **pass** utasítást, amivel jelezzük, hogy az objektum egyéb jellemzőit most nem állítjuk be. Írjuk be, majd üssünk **Entert**:
```
    pass
```
5. Most visszakerültünk a következő sor elejére, de még nincs prompt jel. A parancsértelmező nem tudja, újabb parancsot akarunk-e beírni, vagy vége az utasításnak. Úgy fejezhetjük be az utasítást, hogy ismét megnyomjuk az **Entert.**
6. Hozzunk létre egy *buff* nevű rekordot (példányt) a *Frops* rekordtípusból. Írjuk be, majd üssünk **Entert**:
```
buff = Frops()
```
7. Adjunk értéket a *buff* rekord *sz1* mezőjének. Az érték legyen 8. (Az *sz1* mező most fog létrejönni automatikusan a művelet végrehajtása közben.) Írjuk be, majd üssünk **Entert**:
```
buff.sz1 = 8
```
8. Írassuk ki a *buff* rekord *sz1* mezőjének tartalmát a képernyőre. Írjuk be, majd üssünk **Entert**:
```
print(buff.sz1)
```

9. Hozzunk létre egy üres, frop nevű listát. Ebből fogunk a következő lépésben rekordlistát készíteni. Írjuk be, majd üssünk **Entert**:

```
frop = []
```

10. Indítsunk ciklust, ami százszor fut le. Írjuk be, majd üssünk **Entert**:

```
for i in range(100):
```

11. A parancsértelmező észreveszi a kettőspontot a sor végén, ezért négy szóközzel beljebb kezdi a következő sort, és nem ad prompt jelet, hiszen folytatni fogjuk a beírást. Adjunk az üres *frop* listához *Frops* típusú rekordokat. Írjuk be, majd üssünk kétszer **Entert**:

```
frop.append(Frops())
```

12. Az előző két pontban létrehoztunk egy 100 elemű rekordlistát. A 0. elemébe töltsük át a *buff* rekord tartalmát. Írjuk be, majd üssünk **Entert**:

```
frop[0] = buff
```

13. Ellenőrizzük, hogy áttöltődött-e. Írjuk ki a rekordlista 0. elemének sz1 mezőjét. Hogy lássuk, hogyan kell műveletet végezni vele, szorozzuk meg kettővel. Írjuk be, majd üssünk **Entert**:

```
print(2*frop[0].sz1)
```

60. mintafeladat – rekordok alkalmazása

a) Készítsünk rekordtípust az alábbi törtműveletet tartalmazó adatsor tárolására *Frops* néven: 8 20 8 30 *. A mezők neve sorrendben *sz1, n1, sz2, n2, op* legyen. Hozzunk létre *frop* néven egy 10 elemű *Frops* típusú rekordlistát.

b) Töltsük fel véletlenszerűen 1...9 számokkal a számlálókat (sz1, sz2) és nevezőket (n1, n2), valamint szorzás és osztás (*, :) jelekkel az *op* mezőt.

c) Írassuk ki a képernyőre ilyen formában az összes rekordot: 8/2 * 8/3 = 64/6. Az eredményt számolással állítsuk elő, egyszerűsíteni nem szükséges.

d) A 8. rekord kerüljön a 9. helyére, a 7. rekord a 8. helyére, stb. A 9. rekord pedig a 0. rekord helyére.

e) Jelenítsük meg ismét a rekordlistát a c) feladatban leírt módon.

1. A program a feladatrészek szövege szerint tagolható részekre.

a) feladat – kezdőlépések, létrehozás

2. Elérhetővé tesszük a **randrange** utasítást, mert nem része az alap utasításkészletnek (1. sor).

```
1 from random import randrange
2
3 class Frops():
4     pass
5 frop = []
6 for i in range(10):
7     frop.append(Frops())
```

3. Létrehozzuk a rekordtípust (osztályt) *Frops* néven a **class** utasítással, majd jelezzük a **pass** utasítással, hogy nem akarunk további beállításokat az osztályhoz (3–4. sor).
4. Létrehozunk egy üres listát *frop* néven, amibe majd a rekordok kerülnek (5. sor).
5. Hozzáadjuk az üres listához a *Frops* típusú rekordokat egyesével (**append**), egy ciklus segítségével, összesen tízszer (6–7. sor).

b) feladat – rekordok feltöltése

6. Feltöltjük egy rekord mezőit (24–31. sor). Ezt tízszer ismételjük, mert ennyi rekordot kell feltölteni. Ehhez ciklust használunk (23. sor).
7. A számokat tartalmazó mezőket (sz1, n1, sz2, n2) feltöltjük 1…9 véletlenszámokkal (24–27. sor).

```
23 for i in range(10):
24     frop[i].sz1 = randrange(1,10)
25     frop[i].n1 = randrange(1,10)
26     frop[i].sz2 = randrange(1,10)
27     frop[i].n2 = randrange(1,10)
28     if randrange(2) == 0:
29         frop[i].op = '*'
30     else:
31         frop[i].op = ':'
32 fropout()
```

8. A műveleti jeleket úgy készítjük, hogy létrehozunk egy 0…1 véletlen egészet (28. sor), majd 0 esetén * (29. sor), 1 esetén : kerül az op mezőbe (31. sor).
9. A végén meghívjuk a **fropout()** eljárást, ami megjeleníti a teljes rekordlistát.

c) és e) feladatok – megjelenítés

10. Mivel a megjelenítés kétszer is előfordul a programban, célszerű neki eljárást írni.
11. Megadjuk az eljárás nevét a **def** utasítás mögött (9. sor).
12. Egy rekordot írnak ki a 11–20. sorok. Ezt ismételjük meg tízszer egy ciklus segítségével (10. sor).

```
 9 def fropout():
10     for i in range(10):
11         if frop[i].op == '*':
12             esz = frop[i].sz1 * frop[i].sz2
13             en = frop[i].n1 * frop[i].n2
14         else:
15             esz = frop[i].sz1 * frop[i].n2
16             en = frop[i].n1 * frop[i].sz2
17         print("%d/%d %s %d/%d = %d/%d" % \
18               (frop[i].sz1, frop[i].n1, \
19                frop[i].op, frop[i].sz2, \
20                frop[i].n2, esz, en))
21     print()
```

13. A 12–16. sorokban meghatározzuk az eredmény számlálóját *(esz)* és nevezőjét *(en)*.
14. Ha a rekordban a műveleti jel (frop[i].op) a szorzásjel (11. sor), akkor az eredmény számlálója a két számláló (sz1, sz2) szorzata, a nevező pedig a két nevező (n1, n2) szorzata.
15. Különben (14. sor) osztás műveletet kell végezni, ami a reciprokkal történő szorzást jelenti (15–16. sor).

16. Végül kiíratjuk az aktuális rekord minden adatát a megfelelő formátumban (17–20. sor), majd kihagyunk egy sort, hogy elkülönítsük a következő megjelenített adatoktól (21. sor).

d) feladat

17. Nem tudjuk azonnal a 9. rekordba áttölteni a 8. tartalmát, mert akkor a 9. tartalma elvész. Ezért létrehozunk *Frops* típusú rekordot, ahová átmenetileg helyezzük. Legyen a neve: *buff* (34. sor).
18. Ebbe a *buff* változóba átmásoljuk a 9. rekord tartalmát (35. sor).

```
34 buff = Frops()
35 buff = frop[9]
36 for i in range(9,-1,-1):
37     frop[i] = frop[i-1]
38 frop[0] = buff
39 fropout()
```

19. A továbbiakban ismételni kell kilencszer (36. sor), hogy az aktuális (tehát az i.) rekordba áttöltjük a nála eggyel kisebb számú (i-1.) rekord tartalmát. Visszafelé kell haladnunk, ezért indulunk 9-től, és megyünk 0-ig -1 lépésközzel. (Ha előre haladnánk, a 0. rekordot írnánk az 1.-be, majd a már megváltozott 1.-t a 2.-ba, stb., így a végén minden rekord teljesen egyforma lenne.)
20. Végül a 0. rekordba betöltjük *buff* változóban tárolt, eredetileg 9. rekord tartalmát. Így most teljes a feladat (38. sor).
21. Nincs más hátra, mint kiíratni ismét a teljes rekordlistát a **fropout()** eljárással (39. sor).

Feladatok

1. Telkek
   a) Készítsünk rekordtípust az alábbi, telkek adatait tartalmazó adatsor tárolására *Sites* néven: 21 15 50. A mezők neve sorrendben *num, wi, lo* legyen (házszám, szélesség, hosszúság). Hozzunk létre *site* néven egy 10 elemű *Sites* típusú rekordlistát.
   b) A házszámok növekedjenek 1-től 10-ig egyesével. Töltsük fel a szélességeket véletlenszerűen 10...30 számokkal, a hosszúságokat véletlenszerű 50...100 számokkal.
   c) Írassuk ki a képernyőre a házszámokat és a telkek területét ilyen formában minden rekordra: *A 9. számú telek területe 750 négyzetméter.* Az eredményt számolással állítsuk elő.
   d) Cseréljük meg az első és utolsó rekordot, a második és utolsó előtti rekordot, stb.
   e) Jelenítsük meg ismét a rekordlistát a c) feladatban leírt módon.

2. Kocsik
   a) Készítsünk rekordtípust az alábbi, kocsik adatait tartalmazó adatsor tárolására: ABC-123 BMW 2016 fekete 10000 (rendszám, márka, évjárat, szín, érték). Hozzunk létre *site* néven egy 10 elemű rekordlistát az autók adatainak tárolására.
   b) A rendszámok legyenek véletlenszerűek, a márka is véletlenszerű a BMW, Fiat, Volvo, Peugeot lehetőségekből, az évjárat egy véletlenszerű 2010...2019 egész szám. A színeket a fekete, kék, fehér, zöld lehetőségekből véletlenszerű választjuk. Az értékek 8000...15000, ezresekre kerekített értékek.
   c) Jelenítsük meg minden autó minden adatát az a) feladatnál látható formában.
   d) Írjuk ki a képernyőre, hány évesek az egyes autók, ilyen formátumban: *rendszám: ABC-123, évjárat: 2016, kor: 4 év.*
   e) Mennyi az autók együttes értéke? Jelenítsük meg ilyen formában: Összérték: 290000.

9. EGYSZERŰ SZÖVEGES FÁJLOK

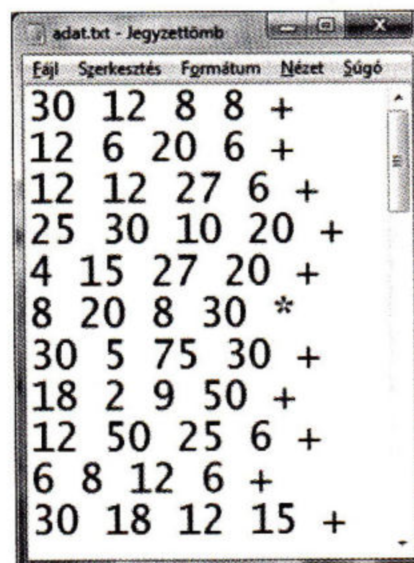
Szöveges fájlok szerkezete

A háttértárolók képesek árammentes állapotban is nagy mennyiségű adat tárolására. A háttértárolókon az összetartozó adatokat (bájtokat) fájlokba szervezik.

Egy fájl összetartozó bájtokból áll, amik a megfelelő sorrendben vannak.

Az egyszerű szöveges fájlban, alapvető esetben, az egy karakter – egy bájt tárolás szokásos. A karakterek ASCII kódja tárolódik a bájtokban, mégpedig olyan sorrendben, ahogy a szavakat alkotják. Például: ABBA – 65, 66, 66, 65 – 41, 42, 42, 41; szó – bájtsorrend 10-es számrendszerben – bájtsorrend 16-os számrendszerben.

Az egyszerű szöveges fájlokat legegyszerűbben a Jegyzettömbbel (Notepad) nézhetjük meg. Hívják még őket textfájloknak, vagy szöveges fájloknak is.



Az egyszerű jelző arra utal, hogy az ilyen szöveget nem lehet úgy formázni, mint egy szövegszerkesztőben. Nem lehetnek benne képek, rajzok, többféle betűtípus, aláhúzás, stb. Így tároljuk általában a naplózásokat (log), és gyakran a programok, szolgáltatások működését befolyásoló paramétereket is. Gyakran a hivatalok így kérik az adatokat, mert ez egyszerű eszközökkel megnyitható. Nem kell például megvenni a drága MS Office-t a hivatali gépekre. Nem utolsó szempont az sem, hogy kevés helyet foglal.

Ha megnyitunk egy egyszerű szöveges fájlt a Jegyzettömbben, és az ablakméretet vízszintesen változtatjuk, azt fogjuk tapasztalni, hogy egy formázás mégiscsak van benne, ez pedig a bekezdés vége, ahol Entert ütött valaki korábban. (A képen lévő fájlban minden sor vége egyben bekezdés vége is.) Utána a szöveg mindig új sorban kezdődik. A Jegyzettömb onnan tudja, hol a bekezdés vége, hogy oda az Enter billentyű megnyomásakor egy 13, 10 (tizenhatos számrendszerben: 0D, 0A) bájtsorozat kerül tárolásra. Mikor a Jegyzettömbben megnyitjuk a fájlt, ahol ezt a bájtsort találja, ott automatikusan új sort kezd.

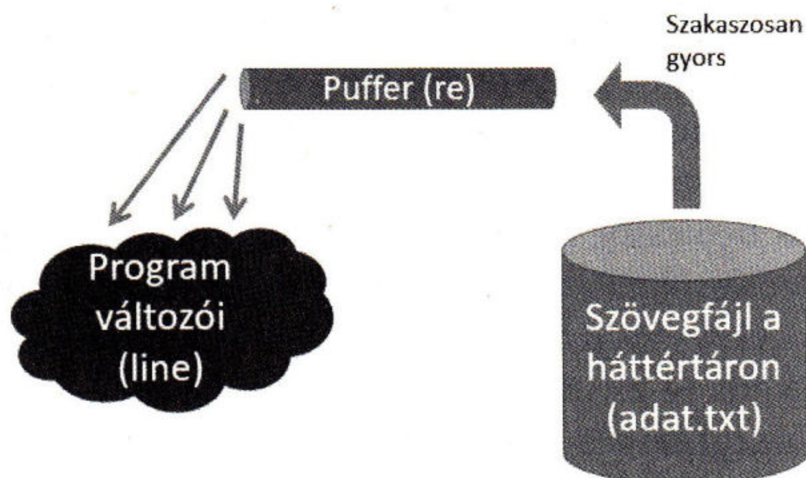
Természetesen a fájlt Jegyzettömbbel megnézve ezek a bájtok nem láthatók. Akkor honnan tudhatjuk, hogy tényleg léteznek? Bizonyos programok képesek bájtról bájtra megjeleníteni egy fájlt. Egy ilyen programmal megnézve a szöveges fájlt a sorvége jelek előtűnnek. Az alábbi ábrán az *adat.txt* fájl első pár sora látható. Keressük meg bátran a sorvége jeleket, és akár a már ismert adatokat is. A második szaggatott vonal jobb szélén láthatjuk a karaktereket, tőle balra a bájtokat tizenhatos számrendszerben.

```
00000000: 33 30 20 31 32 20 38 20|38 20 2B 0D 0A 31 32 20 | 30 12 8 8 +..12
00000010: 36 20 32 30 20 36 20 2B|0D 0A 31 32 20 31 32 20 | 6 20 6 +..12 12
00000020: 32 37 20 36 20 2B 0D 0A|32 35 20 33 30 20 31 30 | 27 6 +..25 30 10
00000030: 20 32 30 20 2B 0D 0A 34|20 31 35 20 32 37 20 32 |  20 +..4 15 27 2
00000040: 30 20 2B 0D 0A 38 20 32|30 20 38 20 33 30 20 2A | 0 +..8 20 8 30 *
00000050: 0D 0A 33 30 20 35 20 37|35 20 33 30 20 2B 0D 0A | ..30 5 75 30 +..
00000060: 31 38 20 32 20 39 20 35|30 20 2B 0D 0A 31 32 20 | 18 2 9 50 +..12
00000070: 35 30 20 32 35 20 36 20|2B 0D 0A 36 20 38 20 31 | 50 25 6 +..6 8 1
00000080: 32 20 36 20 2B 0D 0A 33|30 20 31 38 20 31 32 20 | 2 6 +..30 18 12
```

A fájlkezelés műveletei

A fájlkezelés műveletei legáltalánosabb esetben a következők:

1. **Fájl megnyitása.** Meg kell adnunk, melyik fájlt (fájlnév), melyik helyről (elérési útvonal), és milyen céllal (írás, olvasás, bővítés) akarunk megnyitni. A fájlhoz tartozik a memóriában egy lefoglalt terület, amin keresztül egy szövegfájlt elérhetünk. Ez a terület pufferként szolgál. Például a háttértár felől szakaszosan, de gyorsan érkező adatokat a program nem tudná megfelelő sebességgel feldolgozni, ezért ezeknek az adatoknak valahol várakozni kell. Ugyanez a helyzet a fájlba írásnál is, csak akkor a nyilak iránya fordított. A puffernek azért van jelentősége a mi szempontunkból, mert ha idő előtt megszüntetjük, elveszhetnek azok az adatok, amik még nem jutottak el a végállomásig. A fájl helyét elérési útvonallal kell megadni. Például, ha a d: meghajtón közvetlenül elhelyezkedő works könyvtárban található, akkor a helyes utasítás a re = open("d:\\works\\adat.txt",'r'). Mappaelválasztó jelként tehát mindig \\ -t kell használni. Nem kell megadni elérési útvonalat, azaz elég a fájlnév, ha a megnyitandó szövegfájl *(txt)* ugyanott van, mint a programfájl *(py*, netán a belőle fordított *exe)*. Ezt a megoldást szoktam ajánlani az érettségizőknek. Másolják oda a *txt* fájlt, és nem okoz problémát az elérési út. (Természetesen ez egy üzleti célú programnál nem mindig valósítható meg.)



2. **Az adatok beolvasása fájlból, vagy kiírása fájlba.**

3. Végül használat után a fájlt kötelezően **be kell zárni**, hogy az adatvesztést vagy a fájl esetleges sérülését elkerüljük.

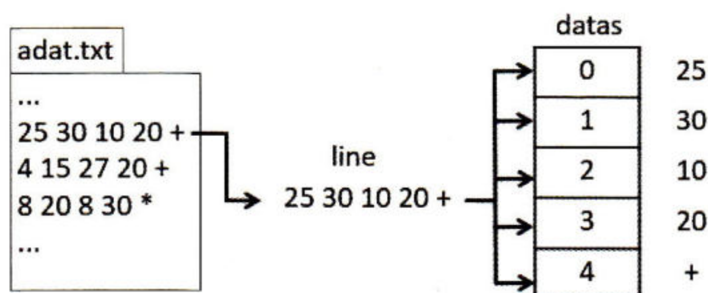
10. FÁJLOK BEOLVASÁSA

Áttekintés

Az olvasás történhet bájtonként, vagy bekezdésenként, vagy beolvashatjuk az egész fájlt egyszerre. Mindig a célszerűség dönti el, melyiket használjuk. Ha egy az egyben meg kell jeleníteni a fájlt, választhatjuk az egész fájl egyszerre történő beolvasását. Ha nem ismerjük a fájl szerkezetét, egészen biztosan bájtonként fogjuk beolvasni. Az egyszerű szöveges fájlokat általában bekezdésenként célszerű beolvasni, ezért a továbbiakban ezzel foglalkozunk részletesen.

A *következő* bekezdés beolvasására a **readline** utasítás szolgál a Pythonban. Sajnos ez az utasítás beolvassa a bekezdés vége (chr(13) chr(10)) jelet is, ami sokszor zavaró, ezért a **strip** utasítást használva levágjuk.

A bekezdéseknek is szokott belső szerkezete lenni, azaz több adatból állhatnak. (A példában [adat.txt] a bekezdések 5 részből álltak.) Az adatokat valamilyen elválasztójel, például szóköz különíti el egymástól. Általában az adatokra külön-külön van szükség, ezért a bekezdéseket lebontjuk az elválasztójelek mentén. Erre a célra szolgál a **split** utasítás, ami egy listába rakja az adatokat, amiket aztán szükség esetén konvertálni kell számmá.



A **readline** utasítás mindig a **következő bekezdést** olvassa be. Emiatt a fájlban található adatokhoz csak az elejétől indulva lehet hozzáférni. Lehet, hogy csak egy bekezdést fogunk kiíratni vagy feldolgozni, de ebben a fajta fájlkezelésben nem lehet közvetlenül hozzáférni az adathoz, végig kell járnunk az előzőeket.

Ha meg kell változtatnunk a **bekezdések sorrendjét**, célszerű az adatokat egy **listába**, vagy ha egy bekezdésben több adat van, **rekordlistába** vagy **mátrixba** helyezni, mert ott már szabadon kezelhetjük őket. A lista a memóriában (RAM) tárolódik, így a műveletek gyorsak vele. Azonban a memória mérete sokkal kisebb, mint a háttértárolóké, így előfordulhat, hogy a fájlból beolvasott adatok nem férnek el a memóriában. Ezért a memóriába csak akkor töltsük be az összes adatot, ha nincs más megoldás. Az érettségi feladatok szövegében így utalnak például arra, hogy ne töltsük be az összes adatot: "A program tetszőleges fájlméret esetén működjön." (Tehát akkor is, ha nem fér el a memóriában.)

Minden fájl-beolvasásos feladatot azzal kell kezdeni, hogy megnézzük a szöveges fájl szerkezetét, például egy Jegyzettömbbel. Így el tudjuk dönteni, milyen módon kell az adatokat beolvasni és tagolni.

Gondot okozhat, ha a fájl végén egynél több üres sor található. Ha alkalmazzuk az adatok lebontására a **split** utasítást, hibaüzenetet fogunk kapni, mert nem képes listára bontani az üres sort.

61. mintafeladat – ismeretlen bekezdésszámú fájl beolvasása

Olvassuk be az adat.txt fájlban található adatokat. Minden sort írjunk ki a képernyőre *24/32 + 8/3* =alakban.

A fájl szerkezete már ismert, az előző témakörnél található a fényképe.

a) Egy sor beolvasása

```
1 re = open("adat.txt",'r')
2 line = re.readline()
3 print(line)
4 re.close()
```

1. Először megnyitjuk a fájlt. Megadjuk a memóriaterület nevét *(re)*, amin keresztül a fájlt elérjük. Megadjuk a fájl nevét *(adat.txt)*, amit szeretnénk beolvasni, majd hogy olvasásra nyitjuk meg a fájlt *('r')*. A fájl neve előtt azért nincs elérési út, mert az adat.txt fájlt előzőleg ugyanoda másoltuk, ahol a programfájl van. (Ez nálam a C:\python könyvtár.) (1. sor)
2. Egy *line* nevű változóba beolvasunk egy bekezdést (ami most megegyezik egy sorral) a fájlból (2. sor).
3. Egyelőre kísérletképpen, a bekezdést közvetlenül kiírjuk a képernyőre (3. sor).
4. A fájlt a használat befejeztével bezárjuk (4. sor).
5. Futtassuk a programot. Meg fog jelenni a fájl első sora a parancsértelmezőben.

b) Összes sor beolvasása

```
1 re = open("adat.txt",'r')
2 line = re.readline()
3 while line !="":
4     line = line.strip()
5     print(line)
6     line = re.readline()
7 re.close()
```

6. Bővítsük az előző programot azzal, hogy meg kell ismételni a bekezdések beolvasását. Nem tudjuk, hányszor kell elvégezni, tehát **while** ciklust fogunk alkalmazni. Addig kell ismételni a beolvasást (readline), míg üres bekezdést nem kapunk. Ezt az előzőleg megírt program **readline** utasítása (2. sor) mögé kell írni, hiszen csak egy sor beolvasása után van értelme a vizsgálatnak. Viszont a kiíratás (print) elé (3. sor), mert csak akkor lehet kiíratni, ha van mit.
7. A kiíratás után meg kell ismételni a beolvasást, hogy a ciklusnak legyen mit újból vizsgálnia.

8. Futtassuk ismét a programot. Azt fogjuk látni, hogy megjelenik a fájl minden sora, de van közöttük egy üres sor. Ennek az az oka, hogy a **readline** beolvasta a line változóba a bekezdés vége (chr(13) chr(10)) jelet, amit aztán a print utasítás kiír, és hozzáteszi a saját bekezdés vége jelét is.
9. Ha a **print** sora elé beiktatunk egy **strip** utasítást, azzal leszedhetjük a beolvasott bekezdés vége jelet (új program, 4. sor).
10. Futtassuk ismét a programot. Eltűntek a felesleges üres sorok.

c) Formázott kiíratás

```
 1 re = open("adat.txt",'r')
 2 line = re.readline()
 3 while line !="":
 4     line = line.strip()
 5     datas = line.split()
 6     print("%s/%s %s %s/%s =" % \
 7            (datas[0], datas[1],
 8             datas[4], datas[2],
 9             datas[3]))
10     line = re.readline()
11 re.close()
```

11. Ahhoz, hogy az adatok a feladatnak megfelelő formában jelenjenek meg, a bekezdéseket szét kell vágni a szóközök mentén, és az adatokat külön változókba tenni. Ennek legegyszerűbb módja a **split** utasítás, ami listát készít az adatokból, amikre aztán a lista nevével és indexszel hivatkozhatunk. A **split** utasítást a **strip** és a **print** közé kell helyezni. Legyen a lista neve *datas* (legújabb program, 5. sor).
12. Már csak a **print** utasítást kell módosítani, hogy az adatok a megfelelő formában legyenek kiíratva. Azért kell minden változónál %s jelölőt használni, mert az adatok nincsenek számmá alakítva. Ezt persze meg kellene tenni, ha számolni akarnánk velük, de ez most nem volt feladat (6. sor).
13. Futtassuk ismét a programot. Teljessé vált a megoldás.

62. mintafeladat – ismert bekezdésszámú fájl beolvasása rekordlistába

Az igeny.txt állomány tartalmazza a lifthasználati igényeket. Első sorában a szintek száma, a második sorban a csapatok száma, a harmadik sorban pedig az igények száma olvasható. A negyedik sortól kezdve soronként egy-egy igény szerepel a jelzés sorrendjében. Egy igény hat számból áll: az első három szám az időt adja meg (óra, perc, másodperc számsorrendben), a negyedik a csapat sorszáma, az ötödik az induló, a hatodik a célszint sorszáma. Az egyes számokat pontosan egy szóköz választja el egymástól. Írjunk programot, ami beolvassa és megjeleníti

igeny - Jegyzettö...

Fájl Szerkesztés Formátum Nézet Súgó

```
65
25
83
9 3 14 3 10 17
9 8 19 12 5 9
9 9 29 7 10 19
```

a képernyőn az igényeket az alábbi módon: *4. csapat – indul: 2. szint, érkezik: 10. szint, 2 óra 36 perc 47 mp.* Az utolsó igényt írjuk ki elsőnek, és haladjunk visszafelé az első igényig.

1. Formázott kiíratást alkalmazunk, tehát a bekezdéseket a szóközök mentén bontani kell majd.
2. A bekezdéseket fordított sorrendben kell kiíratni, tehát az adatokat a memóriába kell tölteni. Mivel egy bekezdésben több adat van, célszerű mátrixot vagy rekordlistát használni. Most maradjunk egyelőre az elsőnél, az a rövidebb.

a) Megoldás mátrix alkalmazásával

```
 1 need = []
 2 re = open("igeny.txt",'r')
 3 line = re.readline()
 4 line = re.readline()
 5 rn = int(re.readline())
 6 for i in range(rn):
 7     line = re.readline()
 8     line = line.strip()
 9     need.append(line.split())
10 re.close()
```

3. Létrehozunk egy üres befoglaló listát *need* néven. Ennek az elemei fognak tartalmazni egy-egy bekezdést (1. sor).
4. Megnyitjuk olvasásra az igeny.txt fájlt, és hozzárendelünk egy változót, amin keresztül elérjük *(re)*. A szövegfájlt a program könyvtárába másoltuk, ezért nincs megadva elérési út (2. sor).
5. Beolvassuk az első két bekezdést. Most ezek az adatok nem kellenek semmire, de a harmadik bekezdéshez csak így lehet hozzáférni (3. és 4. sor).
6. Beolvassuk a fájl harmadik bekezdését. Itt található az igények száma, azaz hogy hány bekezdést kell beolvasni a továbbiakban. Konvertáljuk számmá, és berakjuk az *rn* változóba (5. sor).
7. Mivel tudjuk, hány bekezdést kell beolvasni, számláló ciklust alkalmazunk a további bekezdések beolvasására és feldolgozására (6. sor).
8. A ciklus minden egyes lefutásakor beolvassuk a következő bekezdést (7. sor, **readline**), leszedjük róla a bekezdés vége jelet (8. sor, **strip**), majd felbontjuk a szóközök mentén (**split**), és betöltjük a *need* lista következő elemébe az **append** utasítással (9. sor). Így egy mátrixot kapunk, aminek soraiban a bekezdések adatai, oszlopaiban rendre az óra, perc, másodperc, csapat, indulás, érkezés adatok találhatók, 0, 1, 2, 3, 4, 5 indexekkel.

```
sorindex
|      h      min     sec     team    start    stop  <—— mezőnevek
v      0       1       2       3       4        5    <—— oszlopindex
0 ['9', '9', '29', '7', '10', '19']
1 ['9', '10', '58', '10', '19', '17']
2 ['9', '12', '0', '19', '20', '8']
3 ['9', '16', '17', '3', '17', '51']
...      need[3].sec ↗  ↖ need[3][2]
```

9. A beolvasás után a fájlt bezárjuk (10. sor).
10. A kiíratáshoz indítunk egy másik számláló ciklust. Most az utolsó bekezdés adataitól kezdünk, aminek az indexe rn-1. (Nem rn, mert 0-tól kezdődik a számozás.) 0-ig számlálunk, ezért van az első -1, és visszafelé számlálunk egyesével, ezért van a második -1.
11. A kiíratást a **print** utasítással végezzük. Jelölőként a **%s**-t használjuk, mert nem konvertáltuk a bekezdések adatait számmá (12. sor). A megjelenítendő adatok mindig az i. sorból kerülnek ki. Az oszlopszámot pedig a bekezdésen belüli sorrend adja. 3 – csapat, 4 – indulás, 5 – érkezés, 0 – óra, 1 – perc, 2 – másodperc.

```
11 for i in range(rn-1,-1,-1):
12     print("%s. csapat - indul: %s. szint, érkezik: %s. szint, %s óra %s perc %s mp" \
13         % (need[i][3], need[i][4], need[i][5], need[i][0], need[i][1], need[i][2]))
```

12. A mátrix alkalmazásának hátránya, hogy fejben kell tartani, hányadik oszlopban milyen adat található, ettől egy hosszabb fejlesztésnél kevésbé áttekinthető lenne ez a megoldás. Előnye volt a rövidebb program. Ám ha az adatokat konvertálni kell számmá esetleges további feladatokhoz, már további sorokkal kell úgyis bővíteni, megérheti rekordokat alkalmazni, ahol a mezőnevek beszédesek.

b) Megoldás rekord alkalmazásával

13. Az előző programhoz képest szükség lesz egy rekordtípusra *(Needs)* (1–2. sor).
14. A ciklus minden egyes lefutásakor egy üres rekordot hozzá kell adni a listához (12. sor).
15. A bekezdéseket a szóközök mentén felbontjuk, és egy listába helyezzük (11. sor):
16. A *datas* lista minden egyes elemét konvertáljuk számmá, és a *need* rekordlista megfelelő indexű (i) és nevű (h, min, sec, stb.) elemébe töltjük (13–18. sor).

```
 1 class Needs():
 2     pass
 3 need = []
 4 re = open("igeny.txt",'r')
 5 line = re.readline()
 6 line = re.readline()
 7 rn = int(re.readline())
 8 for i in range(rn):
 9     line = re.readline()
10     line = line.strip()
11     datas = line.split()
12     need.append(Needs())
13     need[i].h = int(datas[0])
14     need[i].min = int(datas[1])
15     need[i].sec = int(datas[2])
16     need[i].team = int(datas[3])
17     need[i].start = int(datas[4])
18     need[i].stop = int(datas[5])
19 re.close()
```

17. Különbség lesz még a megjelenítésben, hiszen a **print** utasításnál mezőnevekkel hivatkozunk (22. sor).

```
20 for i in range(rn-1,-1,-1):
21     print("%s. csapat - indul: %s. szint, érkezik: %s. szint, %s óra %s perc %s mp" \
22         % (need[i].team, need[i].start, need[i].stop, need[i].h, need[i].min, need[i].sec))
```

11. KIÍRÁS FÁJLBA

Az írás művelete is a fájl megnyitásával kezdődik (1. sor). Ilyenkor az **open** utasítás második paramétere **"w"**. Ha a fájl nem létezik, a program létrehozza. Alapértelmezés szerint, ha a fájl már létezik, az újbóli létrehozása teljesen felülírja a régi tartalmat. Tehát nem bővül, hanem törlődik. A fájl elérési útjának megadására ugyanazok a szabályok vonatkoznak, mint az olvasásnál.

```
1 wr = open("test.txt","w")
2 wr.write("András\n")
3 wr.close()
```

A fájl megnyitása után kezdhetünk írni a fájlba. A folyamatot ugyanúgy képzelhetjük el, mintha a képernyőre írnánk. A fájlba írás utasítása a **write** (2. sor). A write utasítással is lehet formázott adatmegjelenítést végezni, mivel ugyanúgy működnek benne a jelölők, mint a **print** esetén. A bekezdés vége jelről (\n) nekünk kell gondoskodni, az utasítás nem tartalmazza. A megnyitott fájl a beleírások hatására folyamatosan bővül. A legutóbb kiírt adat mindig a fájl végére kerül.

Az írás végeztével a fájlt be kell zárni a beolvasáskor megszokott módon a **close** utasítással (3. sor).

Egy fájl utólagosan is *bővíthető*. Ekkor a fájlt az előzőtől eltérő módon, **"a"** paraméterrel kell megnyitni. Bővítés esetén az adatok a fájl végéhez fognak csatlakozni. A fájlok bővítése érettségin nem követelmény.

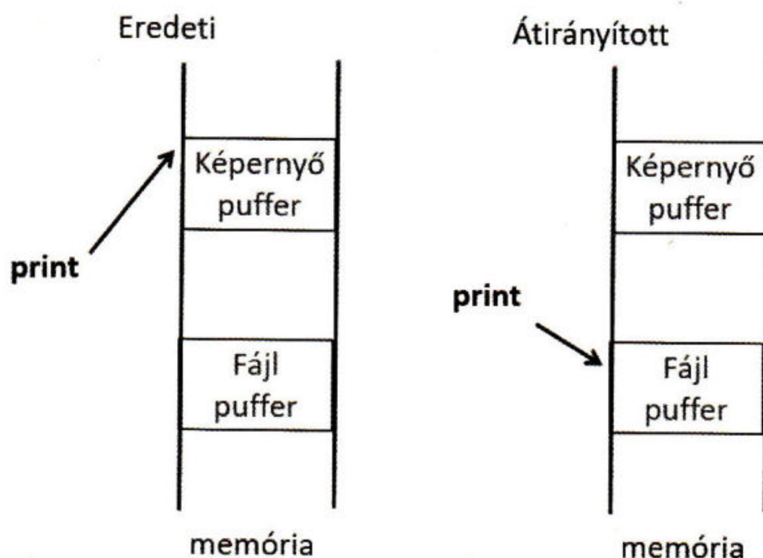
63. mintafeladat – ismerkedés a fájlba írással

1. Álljunk a parancsértelmező ablakba (**Python 3.8.0 Shell**).
2. A prompt jel >>> mögött villog a kurzor. Oda írjuk majd be a parancsokat, mindegyik után megnyomva az **Enter** billentyűt.
3. Nyissunk meg egy *test.txt* nevű fájlt írásra. Írjuk be, majd üssünk **Entert**:
   ```
   wr = open("test.txt",'w')
   ```
4. Írassuk ki a fájlba a nevünket. Írjuk be, majd üssünk **Entert**:
   ```
   wr.write("Magyary Gyula")
   ```
5. A parancsértelmező ablakában megjelenik: 13. Ez a kiírt karakterek száma. Keressük meg a *test.txt* fájlt a **Fájlkezelőben** (alapértelmezésben **C:\Python** könyvtárban találjuk), majd nyissuk meg a **Jegyzettömb** segítségével. Nem lesz benne semmi, mert még nem íródott ki ténylegesen a fájlba a név. Zárjuk be a Jegyzettömböt.

6. Zárjuk be a parancsértelmező segítségével a *test.txt* fájlt. Írjuk be, majd üssünk **Entert**:
   ```
   wr.close()
   ```
7. Nézzük meg ismét **Jegyzettömbbel** a *test.txt* fájlt. Ott lesz benne a nevünk. Zárjuk be a **Jegyzettömböt**. Tanulság: a fájlt be kell zárni a fájlművelet végén, mert ellenkező esetben adatvesztés léphet fel.
8. Nyissunk meg újra a *test.txt* nevű fájlt írásra. Írjuk be, majd üssünk **Entert**:
   ```
   wr = open("test.txt",'w')
   ```
9. Írassuk ki a fájlba a Python szót. Írjuk be, majd üssünk **Entert**:
   ```
   wr.write("Python")
   ```
10. Zárjuk be a parancsértelmező segítségével a *test.txt* fájlt. Írjuk be, majd üssünk **Entert**:
    ```
    wr.close()
    ```
11. Nézzük meg ismét **Jegyzettömbbel** a *test.txt* fájlt. Benne lesz a Python, de a nevünk nem. Zárjuk be a **Jegyzettömböt**. Tanulság: ha egy fájlt újra megnyitunk írásra, a régi fájl törlődik, adatai elvesznek, helyette ugyanolyan néven új fájl készül az új tartalommal.
12. Nyissunk meg újra a *test.txt* nevű fájlt, de most bővítésre. Írjuk be, majd üssünk **Entert**:
    ```
    wr = open("test.txt",'a')
    ```
13. Írassuk ki a fájlba egy új bekezdésben a dátumot. Az új bekezdés jele a *\n*. Írjuk be, majd üssünk **Entert**:
    ```
    wr.write("\n2020.04.01.")
    ```
14. Zárjuk be a parancsértelmező segítségével a *test.txt* fájlt. Írjuk be, majd üssünk **Entert**:
    ```
    wr.close()
    ```
15. Nézzük meg ismét **Jegyzettömbbel** a *test.txt* fájlt. Benne lesz a Python szó és a dátum. Zárjuk be a **Jegyzettömböt**. Tanulság: ha egy már bezárt fájlt szeretnénk bővíteni, akkor bővítésre *(a – append)* kell megnyitni, és nem írásra.

Átirányítás

A parancsértelmező ablakba és a fájlba hasonlóan írnak a programok. Mindkettőhöz puffereket használnak, ahogy a fájlkezelés elején láttuk. Ha a parancsértelmezőt kezelő utasítások számára a képernyőhöz használt puffer címét (ahol a parancsablak adatai tárolódnak a memóriában) átírjuk az adott fájl pufferének címére, a parancsablakba író utasítással (print) írhatunk a fájlba.



Tegyük fel, hogy az adatok képernyőre történő kiíratására van egy kész programrészletünk, és a feladat ezeknek az adatoknak egy szövegfájlba írása. Ekkor a legegyszerűbb megoldás a parancsablak fájlba irányítása, majd a programrészlet megismétlése. Ha ezt nem tesszük, két példányban kell létrehozni az adatok megjelenítését, egyszer képernyőre, egyszer a fájlba, s ez pluszmunka és helyfoglalás a kódismétlés miatt.

```
1 import sys
2 oldout = sys.stdout
3 print("Képernyőre ír.")
4 wr = open("test2.txt",'w')
5 sys.stdout = wr
6 print("Fájlba ír.")
7 wr.close()
8 sys.stdout = oldout
9 print("Képernyőre ír ismét.")
```

A kódoláshoz először ki kell bővíteni az utasításkészletet (1. sor).

Aztán eltároljuk a képernyő puffer címét egy *oldout* nevű változóban (2. sor). Erre azért van szükség, mert a fájlba írás után szeretnénk ismét a képernyőre írni, s ahhoz vissza kell állítani a parancsablakba író utasítások számára az eredeti helyzetet.

Elvégezzük a kiírást a képernyőre a szokásos módon (3. sor).

Megnyitjuk a fájlt írásra a szokásos módon (4. sor).

Átállítjuk a képernyő puffert a fájl pufferre (5. sor).

Kiírjuk a fájlba az adatokat, mintha a konzolablakba írnánk (6. sor).

Mivel a fájlt többet nem használjuk, bezárjuk a fájlt (7. sor).

Visszaállítjuk a parancsablakba író parancs kimenetét a parancsablak pufferre (8. sor), hogy ismét tudjunk írni a képernyőre. Az *oldout* változó tartalmazza a képernyő puffer címét.

Ismét írunk a képernyőre, hogy lássuk, sikerült-e a visszairányítás (9. sor).

64. mintafeladat – ismerkedés az átirányítással

1. Álljunk a parancsértelmező ablakba (**Python 3.8.0 Shell**).
2. A prompt jel >>> mögött villog a kurzor. Oda írjuk majd be a parancsokat, mindegyik után megnyomva az **Enter** billentyűt.
3. Az átirányítás használatához bővítsük a felhasználható utasításkészletet a rendszerutasításokkal (sys). Írjuk be, majd üssünk **Entert**:
   ```
   import sys
   ```
4. Jegyezzük fel egy változóban *(oldout)* a parancsértelmező pufferének címét, hogy majd vissza tudjuk állítani az eredeti értékre. Írjuk be, majd üssünk **Entert**:
   ```
   oldout = sys.stdout
   ```
5. Ellenőrzésként írjunk valamit a képernyőre. Írjuk be, majd üssünk **Entert**:
   ```
   print("Képernyőre ír.")
   ```
6. A parancsértelmező ki is írja a fenti szöveget a szokásos kék betűkkel.
7. Nyissunk meg egy *test2.txt* nevű fájlt írásra. Írjuk be, majd üssünk **Entert**:
   ```
   wr = open("test2.txt",'w')
   ```
8. Irányítsuk át a kiírást a fájl pufferébe. Írjuk be, majd üssünk **Entert**:
   ```
   sys.stdout = wr
   ```
9. Ellenőrzésként írjunk valamit a *test2.txt* fájlba a **print** utasítással, ami alapvetően a parancsértelmező ablakba írna. Írjuk be, majd üssünk **Entert**:
   ```
   print("Fájlba ír.")
   ```
10. Azt fogjuk látni, hogy a parancsablakban nem íródik ki kékkel a "Fájlba ír" szöveg. Valószínűleg sikerült az átirányítás. Azonban a fájlban még hiába keresnénk a szöveget. Ahhoz előbb be kell zárni a fájlt.
11. Zárjuk be a *test2.txt* nevű fájlt. Írjuk be, majd üssünk **Entert**:
    ```
    wr.close()
    ```
12. Keressük meg a *test2.txt* fájlt a **Fájlkezelőben** (alapértelmezésben C:\Python könyvtárban találjuk), majd nyissuk meg a **Jegyzettömb** segítségével. Benne lesz a "Fájlba ír." szöveg. Zárjuk be a **Jegyzettömböt**.
13. Próbáljunk meg írni valamit a *test2.txt* fájlba a **print** utasítással. Írjuk be, majd üssünk **Entert**:
    ```
    print("Hová ír?"))
    ```
14. Hibaüzenetet kapunk, hiszen a fájlt bezártuk, oda nem lehet írni. A parancsértelmezőbe azért nem tud írni, mert a kiíratások még mindig a fájl pufferre mutatnak.
15. Állítsuk vissza az átirányítást a képernyő pufferre. Írjuk be, majd üssünk **Entert**:
    ```
    sys.stdout = oldout
    ```

16. Ellenőrzésként írjunk valamit a képernyőre. Írjuk be, majd üssünk **Entert**:
```
print("Képernyőre ír ismét.")
```
17. Megjelenik a parancsértelmezőben kék színnel a "Képernyőre ír ismét." szöveg, tehát a visszairányítás sikeres volt.

Feladatok

1. Írjunk programot, ami beolvassa az adat.txt fájl tartalmát soronként, majd minden beolvasott sort rögtön ki is ír az *oper.txt* fájlba az alábbi formátumban: *24/32 + 8/3 =*.
2. Írjunk programot, ami egy *fracts.txt* nevű szöveges fájlba kiír 100 sor véletlenszerű adatot. Egy sorban négy darab, 100 alatti pozitív egész szám és egy műveleti jel legyen, egymástól szóközzel elválasztva. Például: 8 20 8 30 *. A lehetséges műveleti jelek: +, -, *, :.
3. Készítsük el a 15 x 15-ös szorzótáblát a képernyőre. A kódrészlet megismétlése nélkül írassuk ki egy *multtable.txt* nevű fájlba is. Végül a program végén a képernyőre írjuk ki: *Kész a fájl.*

VI. PROGRAMOZÁSI TÉTELEK

1. BEVEZETÉS

A programozási tételek olyan algoritmusok típusfeladatokra, amiknek helyessége *bizonyítható*.

Az egyszerűbb programozási tételek (pl. vektor elemeinek összegzése) annyira maguktól értetődőek, hogy akár ki is találhatnánk őket. A bonyolultabbakat már nem feltétlenül (pl. fraktál alapú képtömörítés), vagy hosszabb gondolkodással, esetleg kevésbé hatékony[50] algoritmust alkotva (pl. rendezés).

Rengeteg programozási tétel létezik, és sok-sok gyűjtemény, ahol többé-kevésbé hitelesen megtalálhatjuk őket. Például e könyv írásakor érvényes függvénytáblázat is tartalmazza a legismertebbeket. Azonban ne felejtsük el, hogy az emelt szintű informatika érettségin nem használható semmilyen segédeszköz, így a függvénytábla sem. Tehát a nem magától értetődő algoritmusokat meg kell tanulni fejből. Ez azonban nem kell, hogy magolás legyen. Az algoritmus működésének megértése alapján rekonstruálhatjuk annak pszeudokódját.

Mivel algoritmusokról van szó, nagyjából függetlenek a programozási nyelvtől, így algoritmus-leíró nyelven fogalmazzuk meg őket, mondatszerű leírással. Függetlenségük csak nagyjából igaz: még a magas szintű, Neumann-elvű programnyelveken belül is vannak eltérések. A gyűjteményekben található programozási tételekben a tömbök sorszámozása 1-től indul[51]. Ebben a könyvben az algoritmusokban a tömbök sorszámozását 0-tól indítom, megkönnyítve ezzel az olvasó dolgát. Ez a fejezet terjedelmi okokból nem tartalmaz Python-kódot. Mire az olvasó ideér, kell rendelkeznie annyi rutinnal, hogy ne jelentsen akadályt az algoritmusok kódolása.

Minden programozási tétel a következő részekből épül fel:

Az ***előfeltétel***[52] megmutatja, milyen kiindulási feltételek esetén alkalmazható a tétel. Azaz mivel kell rendelkeznünk a tétel végrehajtása előtt: milyen adatszer-

---

50 A hatékonyság mérőfoka lehet például a gyorsaság. Ezt időben mérni nehéz, mert a különböző számítógépek sebessége nagyon eltérő. Ezért gyakran az értékadások, összehasonlítások számában mérik. A hatékonyságot emelt szintű érettségin nem kérhetik számon, így itt sem foglalkozunk vele részletesen.

51 A Pascal és Basic nyelveknek megfelelően.

52 Felsőfokú tanulmányok során az elő- és utófeltételeket szokás a matematika jelrendszerével megfogalmazni. Ebben a könyvben a könnyebb érthetőség kedvéért megmaradunk a szöveges megfogalmazásnál.

kezet kell (pl. lista), szükséges-e egy felállítható sorrend az elemek között, netán rendezettnek kell lennie az adatoknak, stb.

Az ***utófeltétel*** megmutatja, mi lesz a tétel végrehajtásának eredménye. Az eredmény képződhet egy egyszerű változóban, vagy akár a kiindulási listában, stb.

A megadott elő- és utófeltételhez tartozhat egy- vagy többféle ***típusalgoritmus*** is, ami képes a feladatot helyesen megoldani. A többféle megoldás azonban nem feltétlenül egyenértékű. Bizonyos körülmények között az egyik, máskor pedig a másik lehet hatékonyabb.

A programozási tételeknek különböző *változatai* lehetnek. A változatok erősen hasonlítanak az eredeti tételre, csak kicsit módosulnak az elő-, illetve utófeltétel kismértékű változása miatt. Például vektor helyett mátrixra hajtjuk végre, vagy több tételt együtt alkalmazunk, stb. Gyakran ezeket a változatokat nem tartalmazzák a gyűjtemények, pedig az átgondolásuk esetenként nem is annyira könnyű.

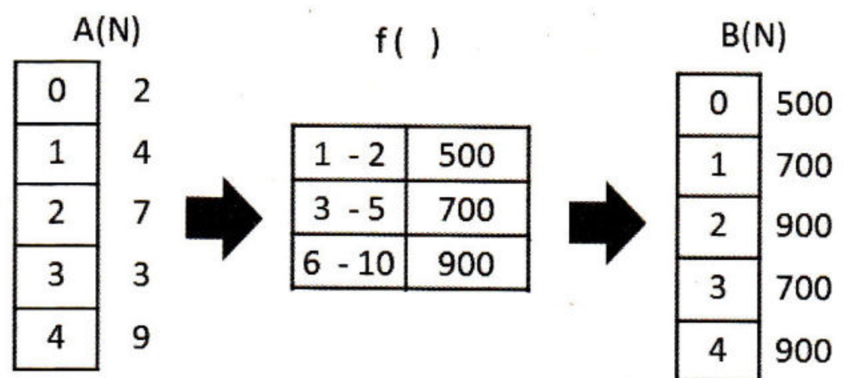
2. SOROZATSZÁMÍTÁS

Bevezető példa

Egy listában (A) tároljuk, hogy egy biciklis futár különböző kézbesítései alkalmával hány kilométert kerékpározik (2, 4, 7, 3, 9). Van egy már kész függvényünk (f), ami megmutatja, milyen távolságokért mennyi pénzt kap. Határozzuk meg egy másik listában (B), mennyi pénz jár az egyes kézbesítések után.

Elő- és utófeltétel

Legyen adott egy A(N) lista, aminek adatait átalakítjuk egy meghatározott módszer szerint, és az eredményt berakjuk egy B(N) lista ugyanolyan indexű elemébe. Lehetnek például az A(N) listában távolságok, amit megtesz egy futár. A B(N) listában pedig szeretnénk meg-




```
1) Ciklus i := 0-től N-1-ig
2)     B(i) := f(A(i))
3) Ciklus vége
```

határozni a fizetséget, amit az egyes utakért kap (utófeltétel). A B(N) lista egy eleme nyilván *függ* az A(N) lista azonos indexű elemétől. A számítás módját egy f függvény segítségével írjuk le. A példában a függvény megadása a képen látható táblázattal történik.

Működés

Az algoritmus igen egyszerű: végigmegyünk az A(N) lista minden egyes elemén egy számláló ciklus segítségével (1–3. sor). A lista aktuális értékére, tehát A(i)-re alkalmazzuk az f függvényt. Ezt jelöljük így: f(A(i)). A példában a táblázat alapján kikeressük, hogy a megtett kilométerért mennyi pénz jár. A keletkezett értéket berakjuk az eredmény tárolására szolgáló listába (2. sor).

Változatok

A kiindulási adatok a listán kívül jöhetnek fájlból is. Az eredményt gyakran csak képernyőre vagy fájlba írjuk, nem tároljuk listában.

Vegyük észre, hogy ezt az algoritmust már többször alkalmaztuk anélkül, hogy megneveztük volna. Például, mikor a hét napjainak sorszáma helyett a hét napjainak nevét írtuk ki, vagy az amőbatáblán a számok helyett figurákat jelenítettünk meg. Ezekben az esetekben a függvénykapcsolat egy listával egyszerűen leírható volt.

Feladatok

1. A 6p_tavok.txt fájlban egy futár adatai találhatók. A listákat másoljuk a vágólapon át a programunkba. Az azonos indexű listaelemek ugyanahhoz a fuvarhoz tartoznak. A futár az egyes utakra az út hosszától függően kap fizetést a mellékelt táblázatnak megfelelően. Határozzuk meg minden egyes út ellenértékét. Ezeket az értékeket írjuk ki a képernyőre a következő formátumban: *1. nap, 1. fuvar: 500 Ft.*

1–2 km	500 Ft
3–5 km	700 Ft
6–10 km	900 Ft
11–20 km	1 400 Ft
21–30 km	2 000 Ft

2. Az önkormányzat telekadót fog kivetni. Az adót Fabatkában számolják. A 700 négyzetméteres és annál kisebb telkek esetén ez 51 Fabatka négyzetméterenként, az ennél nagyobb telkeknél az első 700 négyzetméterre vonatkozóan szintén 51 Fabatka, 700 négyzetméter felett egészen 1000 négyzetméterig 39

Fabatka a négyzetméterenkénti adó. Az 1000 négyzetméter feletti részért négyzetméterárat nem, csak 200 Fabatka egyösszegű átalányt kell fizetni. A 15 m vagy annál keskenyebb, illetve a 25 m vagy annál rövidebb telkek tulajdonosai 20% adókedvezményben részesülnek. Az adó meghatározásánál 100 Fabatkás kerekítést kell használni (pl. 6238 esetén 6200, 6586 esetén 6600). Határozzuk meg, mekkora adóbevételre számíthat az önkormányzat a páratlan számú telkek után. Ezeket az értékeket írjuk ki a képernyőre a következő formátumban: *házszám: 1, szélesség: 20 m, hosszúság: 30 m, adó: 700 Fabatka*. A szükséges adatokat a 6p_telkek.txt fájlban találjuk. Az azonos indexű listaelemek ugyanahhoz a telekhez tartoznak. Másoljuk a listákat a vágólapon át a programunkba.

3. A szoveg.txt fájlban egy szógyűjtemény található. Minden egyes szó mellé írjuk ki a képernyőre, hány karakterből áll, és mi az utolsó betűje. Minden szó külön sorban kezdődjön. Ha nem tudunk fájlt kezelni, használjuk az alább mellékelt listát.

```
words = [”abbahagyjuk”, ”azon”, ”izomtalan”, ”mindenhonnan”, ”nem”, ”rideg”, ”tornyai”]
```

3. ÖSSZESÍTÉS

Bevezető példa

Egy listában (A) tárolódik, hogy egy biciklis futár különböző kézbesítései alkalmával hány kilométert kerékpározik (2, 4, 5, 1, 3). Határozzuk meg, mennyit kerékpározott összesen a futár.

Elő- és utófeltétel

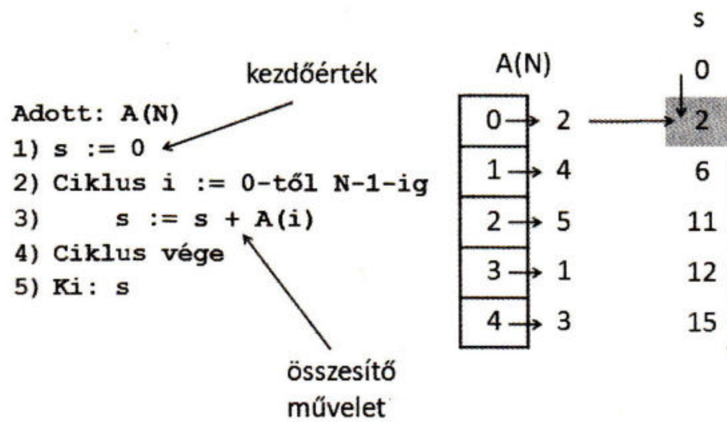
Legyen adott egy A(N) számokból álló lista, és egy s nevű, szám típusú változó. Összesítés tételéről beszélünk, amikor *s* változóban az A(N) listában található elemek összegét (utófeltétel) határozzuk meg.

Működés

Az *s* változó értéke kezdetben 0 (1. sor). Ehhez hozzáadjuk először a lista nulladik elemét (3. sor). A ciklus következő lefutásakor az *s* változóban tárolt átmeneti összeghez hozzáadjuk a lista következő elemét (3. sor). Mindezt megismételjük

a lista minden elemére (2–4. sor). Az összeget végül az s változóban találjuk, amit kiíratunk a képernyőre (5. sor).

Az egész folyamatot elképzelhetjük úgy, hogy számok érkeznek hozzánk (A(N) listából), amit egy papírdarabon (ami az s változónak felel meg) összeadunk. Minden egyes új szám érkezéskor azt hozzáadjuk a papíron éppen található számhoz. Az így keletkezett összeget leírjuk (az összeg az s változóba kerül), és az előző összeget töröljük.



Változatok

Legegyszerűbb változat, mikor a kiszámított összegen további műveleteket hajtunk végre. Például az összegből átlagot számolunk, elosztva azt az elemek számával (N).

A(N), s	művelet	kezdőérték
szám	+	0
szám	*	1
logikai	∧	igaz
logikai	∨	hamis
string	+	””

Feltételes összegzésről beszélhetünk, ha csak bizonyos tulajdonságú listaelemeket összesítünk. Például csak a páros számokat. Ilyenkor a 3. sor egy elágazással egészül ki. A példa szerint így:

Ha *A(i) páros* akkor s := s + A(i)

A tétel alkalmas az összeadáson kívül más műveletek, sőt más típusú változók használatára (lásd táblázat). Számok között maradva az A(N) lista értékei össze is szorozhatók a fenti algoritmus segítségével, ha a művelet az összeadás helyett szorzás, és kezdőértéknek az 1-et választjuk.

Az összesítés tétele segítségével összefűzhetjük egy szöveges lista elemeit. ÉS (∧), illetve VAGY (v) műveletet is végrehajthatunk logikai típusú adatokat tartalmazó listák elemein.

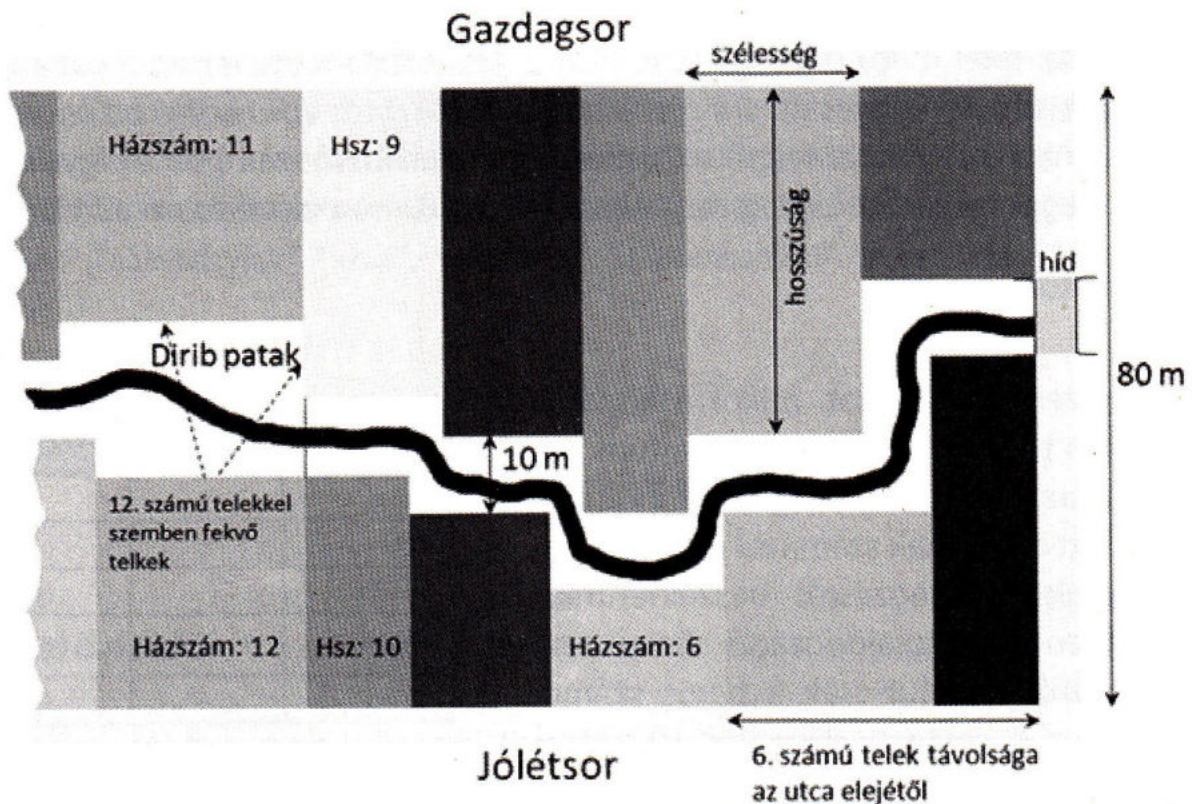
Feladatok

1. A 6p_tavok.txt fájlban egy futár adatai találhatók. Másoljuk a listákat a vágólapon át a programunkba. Az azonos indexű listaelemek ugyanahhoz a fuvarhoz tartoznak. A futár az egyes utakra az út hosszától füg-

1–2 km	500 Ft
3–5 km	700 Ft
6–10 km	900 Ft
11–20 km	1 400 Ft
21–30 km	2 000 Ft

gően kap fizetést a mellékelt táblázatnak megfelelően. Írjuk ki a képernyőre, hogy a futár mekkora összeget kap a heti munkájáért.

2. Hány métert kell annak gyalogolnia, aki *körbe* akarja járni a mellékelt ábrán látható két utcát? A kiszámított távolságot írassuk ki a képernyőre! A szükséges adatokat a 6p_telkek.txt fájlban találjuk. Az azonos indexű listaelemek ugyanahhoz a telekhez tartoznak. Másoljuk a listákat a vágólapon át a programunkba.



3. Az önkormányzat telekadót fog kivetni. Az adót Fabatkában számolják. A 700 négyzetméteres és annál kisebb telkek esetén ez 51 Fabatka négyzetméterenként, az ennél nagyobb telkeknél az első 700 négyzetméterre vonatkozóan szintén 51 Fabatka, 700 négyzetméter felett egészen 1000 négyzetméterig 39 Fabatka a négyzetméterenkénti adó. Az 1000 négyzetméter feletti részért négyzetméterárat nem, csak 200 Fabatka egyösszegű átalányt kell fizetni. A 15 m vagy annál keskenyebb, illetve a 25 m vagy annál rövidebb telkek tulajdonosai 20% adókedvezményben részesülnek. Az adó meghatározásánál 100 Fabatkás kerekítést kell használni (pl. 6238 esetén 6200, 6586 esetén 6600). Határozzuk meg, mekkora adóbevételre számíthat az önkormányzat a Gazdagsoron található telkek után. A szükséges adatokat a 6p_telkek.txt fájlban találjuk. Az azonos indexű listaelemek ugyanahhoz a telekhez tartoznak. Másoljuk a listákat a vágólapon át a programunkba.

4. MEGSZÁMLÁLÁS

Bevezető példa

Egy listában (A) 1 és 6 közötti véletlen egész számokat tárolunk (virtuális dobókocka dobásának eredményeit). Határozzuk meg, hány darab páros szám van közöttük.

Elő- és utófeltételek

Legyen adott egy A(N) lista, egy T tulajdonság, és egy db nevű, szám típusú változó. Ha a db változóban az A(N) listában található T tulajdonságú elemek számát (utófeltétel) határozzuk meg, megszámlálás tételéről beszélünk.

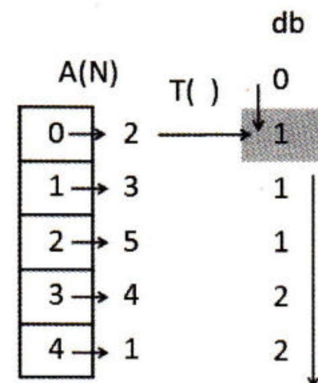
A T tulajdonságot egy logikai értéket visszaadó függvényként adhatjuk meg. Például jelentse T tulajdonság a *páros számokat.* Ebben az esetben T(A(i)) jelentése *A(i) páros*, és így fogjuk konkrétan a Pythonba átültetni: *A[i] % 2 == 0.*

Működés

A db változó értéke kezdetben 0 (1. sor). Először megvizsgáljuk a lista nulladik elemét, hogy T tulajdonságú-e (3. sor). Ha igen, db értékét eggyel megnöveljük (4. sor). Mindezt megismételjük a lista minden elemére (2–5. sorok). A darabszámot végül a db változóban találjuk, amit kiíratunk a képernyőre (6. sor).

```
Adott: A(N), db és T
1) db := 0
2) Ciklus i := 0-től N-1-ig
3)    Ha T(A(i)) akkor
4)          db := db + 1
5) Ciklus vége
6) Ki: db
```

A(i) T tulajdonságú,
Például: A(i) páros



Az egész folyamatot elképzelhetjük úgy, hogy elemek érkeznek hozzánk (A(N) listából), amiből a megfelelő tulajdonságúakat egy papírdarabon (ami a db változónak felel meg) megszámláljuk. Minden egyes új szám érkezésekor megvizsgáljuk, hogy T tulajdonságú-e (a példában páros-e.) Ha igen, eggyel nagyobb számot írunk a papírra, az előző számot töröljük.

Változatok

Ha sok értéket szeretnénk megszámlálni, érdemes segédlistát *(db(M))* alkalmazni. A segédlista indexei tartalmazzák a megszámlálandó értékeket, a segédlista értékei pedig az indexhez tartozó darabszámot.

Például legyen az A(N) listában a dobókockával történt véletlenszerű dobások eredménye, a feladat pedig összeszámolni, hány darab egyest, kettest... hatost dobtunk. Létrehozunk egy db nevű segédlistát. Ennek indexei lesznek 0, 1, 2, ..., 6 (a 0. elemet nem használjuk fel). A db lista értékeiben fog történni a megszámlálás, és a végén ott találjuk meg a darabszámokat is.

Az eredeti algoritmus kicsit módosul: az elején a segédlista minden értékét 0-ra állítjuk (1. sor). A kódban ez egy számláló ciklus segítségével valósul meg, de ezt mondatszerű leírásnál szabad egyszerűsíteni az ábrán látható módon. Hasonlóan alakul az algoritmus vége, ahol a segédlista minden elemét (azaz a darabszámokat) kiíratjuk a képernyőre (5. sor).

Például:
A(N)-ben dobókocka dobások

```
1) db() := 0
2) Ciklus i := 0-től N-1-ig
3)     db(A(i)) := db(A(i))+1
4) Ciklus vége
5) Ki: db()
```

A(N) —db(A(i))→ db(0..6)

i	A(N)
0	2
1	1
2	5
3	3
4	1
5	1
6	3
7	4

index	db
0	0
1	3
2	1
3	2
4	1
5	1
6	0

A legkomolyabb eltérés a 3. sorban van, ahol nem látszik a T tulajdonság. Pedig ott van, mégpedig az A és a db listák egymáshoz illeszkedésében, ahol az A(N) lista értéke *megegyezik* a db (1…6) lista indexével. Ez a "megegyezik" alkotja a T tulajdonságot.

Feladatok

1. Az önkormányzat előírásai szerint a 20 m széles vagy annál keskenyebb telkek esetén teljes utcafront-beépítést kell alkalmazni. Határozzuk meg, és a képernyőre írassuk ki, hogy ez hány telekre vonatkozik a Jólétsoron (páros házszámok). A szükséges adatokat a 6p_telkek.txt fájlban találjuk. Az azonos indexű listaelemek ugyanahhoz a telekhez tartoznak. Másoljuk a listákat a vágólapon át a programunkba.

2. Adott az alábbi robotvezérlő kódsor. A helyes sorozatokban csak a robot által értelmezhető négy karakter szerepelhet: "E" az előre, "B" a balra, "J" a jobbra, "H" a hátra haladást jelenti egy-egy egységgel. Hány irányváltás történt? Milyen távolságra jutott a kezdőponttól a robot, mikor a végén megállt?
   EEJJJEEBBBEEJEJEJEJEEBBB
3. Töltsünk fel egy listát n db véletlenszerű fej (F) vagy írás (I) értékkel. n értékét a felhasználó adhatja meg. Ellenőrzésképp írassuk ki a lista elemeit a képernyőre egymás mellé, szóköz nélkül.
   a. Milyen relatív gyakorisággal dobtunk a kísérlet során fejet? (A fej relatív gyakorisága a fejet eredményező dobások és az összes dobás hányadosa.) A relatív gyakoriságot két tizedesjegy pontossággal, százalék formátumban írassa ki a képernyőre.
   b. Hányszor fordult elő ebben a kísérletben, hogy egymás után pontosan két fejet dobtunk?
4. Vizsgázók feleletválasztós tesztet írtak. A helyes megoldást a "BAACAABBCCCBABB" karakterlánc tartalmazza. Írjunk függvényt, aminek bemeneti paramétere egy vizsgázó megoldása, visszaadott értéke pedig a jó megoldások száma.
5. Határozzuk meg, hogy a szoveg.txt fájl mely szavaiban van több magánhangzó, mint egyéb karakter. Ezeket a szavakat írjuk ki a képernyőre egy-egy szóközzel elválasztva. A szavak felsorolása után a mintának megfelelően az alábbi adatokat adjuk meg:
   a. hány szót találtunk, amiben több a magánhangzó
   b. hány szó van összesen az állományban
   c. A talált szavak hány százalékát teszik ki az összes szónak (A százalékot két tizedessel szerepeltessük. Például: *130/3000 : 4,33%)*
6. A *6p_kep.txt* fájlban egy 50 × 50 képpontos kép képpontjainak RGB kódjai vannak három mátrixban (r, g, b) tárolva. Az azonos indexű tömbelemek alkotnak egy képpontot. Másoljuk a tömböket a vágólapon át a programunkba.
   a. Határozzuk meg, hogy a kép 35. sor 8. képpontjának színe hányszor szerepel a 35. sorban, illetve a 8. oszlopban. (A sorok és oszlopok számozása a kérdésben 1-től kezdődik, de a tömbök indexelése 0-tól.) Az értékeket írja ki a képernyőre az alábbi formában: *Sorban: 5, Oszlopban: 10.*
   b. Állapítsuk meg, hogy a vörös RGB (255,0,0), kék RGB (0,0,255) és zöld RGB (0,255,0) színek közül melyik szín fordul elő legtöbbször a képen. Az (egyik) legtöbbször előforduló szín nevét írjuk ki a képernyőre.

5. MAXIMUMKIVÁLASZTÁS

Bevezető példa

Egy listában (A) 1...6 véletlen egész számokat tárolunk (virtuális dobókocka dobás eredményeit). Határozzuk meg, melyik a legnagyobb érték.

Elő- és utófeltételek

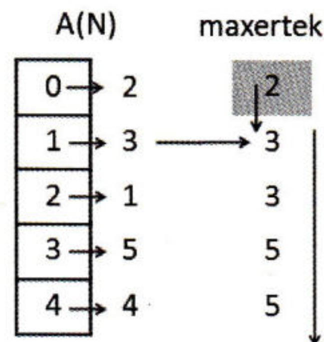
Legyen adott egy A(N) lista, két változó (maxhely, maxertek), valamint a lista lehetséges elemei között lehessen sorrendet felállítani[53]. E sorrend alapján tudjuk eldönteni, melyik a legnagyobb (vagy legkisebb) érték. Tipikus esetben ez a valós (egész, stb.) számok között lehet a hagyományos kisebb-nagyobb viszony, karakterek vagy karakterláncok között az ábécé szerinti sorrend. Amikor a maxertek változóban meghatározzuk az A(N) listában található legnagyobb értéket, a maxhely változóban pedig ennek helyét (utófeltétel), a maximumkiválasztás tételéről beszélünk.

Működés

A legnagyobb értéket a maxertek változóban fogjuk megkapni. Először a lista 0. elemét vesszük legnagyobb értéknek (1–2. sor), mivel a többit még nem ismerjük.

A következő lépésekben (4. sor) vesszük a következő listaelemet, és összehasonlítjuk az eddigi elemek közül a legnagyobbal, ami a maxertek változóban tárolódik. Ha a listaelem a nagyobb, a továbbiakban az lesz a lista eddigi legnagyobb értéke, tehát az kerül

```
Adott: A(N), és egy sorrend
1) maxertek := A(0)
2) maxhely := 0
3) Ciklus i := 1-től N-1-ig
4)      Ha maxertek<A(i) akkor
5)           maxertek := A(i)
6)           maxhely := i
7)      elágazás vége
8) Ciklus vége
9) Ki: maxhely, maxertek
```



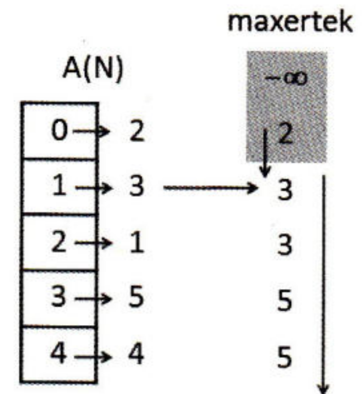
53 Kicsit precízebb, matematikai megfogalmazásban: legyen értelmezve a lehetséges elemeken egy rendezési reláció.

be a maxertek változóba (5–6. sor). Ezt a műveletsorozatot ismételjük a tömb összes elemére (3–8. sor). Végül a legnagyobb értéket kiírjuk a képernyőre (9. sor).

Az egész folyamatot elképzelhetjük úgy, hogy elemek érkeznek hozzánk (A(N) listából). A papírunkon mindig az eddigi legnagyobb érték van felírva. Ezt hasonlítjuk össze azzal az értékkel, ami éppen a listából érkezik. Ha a listából érkező nagyobb, az előző értéket a papírunkról töröljük, helyére az újonnan jött értéket írjuk. Ha a papíron lévő érték a nagyobb, nem teszünk semmit, csak várjuk a következő listalelemet.

Változatok

```
Adott: A(N), és egy sorrend
1) maxertek := -∞
2) maxhely := -1
3) Ciklus i := 0-tól N-1-ig
4)     Ha maxertek<A(i) akkor
5)         maxertek := A(i)
6)         maxhely := i
7)     elágazás vége
8) Ciklus vége
9) Ki: maxhely, maxertek
```



Nem mindig van szükség a legnagyobb érték *helyére*. Ilyenkor a maxhely változóval kapcsolatos bejegyzések kimaradhatnak az algoritmusból.

A relációs jel megfordításával az eredeti algoritmus 4. sorában, a **minimumkiválasztás tételét** kapjuk.[54]

Gyakran van szükség egyszerre a legnagyobb és legkisebb elemre. Ekkor egyetlen ciklus alkalmazása elegendő. A két feltételt a ciklusmagban egymás után, külön elágazásban helyezzük el.

```
Adott: A(N), egy T tulajdonság, és egy sorrend
1) maxertek := -∞
2) maxhely := -1
3) Ciklus i := 0-től N-1-ig
4)     Ha T(A(i)) ÉS maxertek < A(i) akkor
5)         maxertek := A(i)
6)         maxhely := i
7)     elágazás vége
8) Ciklus vége
9) Ha maxhely <> -1 akkor
10)     Ki: maxertek, maxhely
11) különben
12)     Ki: "Nem volt T tulajdonságú elem."
```

Kiküszöbölhetjük a lista 0. elemének kivételes kezelését, ami időnként könnyebbség, mint az a későbbiekben kiderül. Ehhez a maxertek változóba olyan kicsi elemet kell írnunk, ami egészen biztosan nem fordul elő a listában. Ezt jelöltem **-∞-nel.** A gyakorlatban általában találni fogunk ilyen értéket. Például, ha fizetések vannak a listában, bármilyen negatív érték megfelel a célnak.

A feltételes összegzés mintájára megalkothatjuk a feltételes maximumkiválasztás tételét, ahol csak a lista T tulajdonságú elemei közül keressük a legnagyobbat, például a legnagyobb páros számot. Ilyenkor például jól jön, hogy a 0. elemet nem kell külön vizsgálni, hogy T tulajdonságú-e, mert feleslegesen bonyolítaná az algoritmust.

---

54 Persze nem árt a maxertek, maxhely változókat is átnevezni minertek, minhelyre.

Feladatok

1. Hány háznyira van egymástól a legnagyobb és a legkisebb területű telek Gazdagsoron (páratlan házszámok)? A két telek között elhelyezkedő telkek számát, valamint a legnagyobb és legkisebb telek házszámát, illetve területét írassuk ki a képernyőre. A szükséges adatokat a 6p_telkek.txt fájlban találjuk. Az azonos indexű listaelemek ugyanahhoz a telekhez tartoznak. Másoljuk a listákat a vágólapon át a programunkba.
2. A 6p_tavok.txt fájlban egy futár adatai találhatók. Az azonos indexű listaelemek jelentenek egy-egy fuvart. Másoljuk a listákat a vágólapon át a programunkba.
   a. Írjuk ki a képernyőre, melyik nap dolgozott először a futár a héten. (Lehetnének olyan adatok, ahol hétfőn szünnapot tartott. Ezt vegyük figyelembe a program megírásakor.)
   b. Írjuk ki a képernyőre, hogy mekkora volt a hét legelső útja kilométerben. Figyeljünk arra, hogy olyan adatok esetén is helyes értéket adjon, amiben például a hét első napján a futár nem dolgozott.
   c. Írjuk ki a képernyőre, hogy mekkora volt a hét utolsó útja kilométerben.
   d. Írjuk ki a képernyőre, hogy a hét melyik napján volt a legtöbb fuvar. Amennyiben több nap is azonos, maximális számú fuvar volt, elegendő ezek egyikét kiírnia.
3. Írassuk ki a képernyőre, hogy melyik a leghosszabb szó a szoveg.txt állományban, és az hány karakterből áll. Ha több leghosszabb szó is van a szógyűjteményben, akkor azok közül elegendő egyet kiírnunk.

6. ELDÖNTÉS

Bevezető példa

Egy listában (A) 1...6 véletlen egész számokat tárolunk (virtuális dobókocka dobásának eredményeit). Van-e közöttük páros szám?

Elő- és utófeltétel

Legyen adott egy A(N) lista, egy T tulajdonság és egy VANELEM nevű, logikai típusú változó. Eldöntés tételéről beszélünk, ha a VANELEM változóban meghatároz-

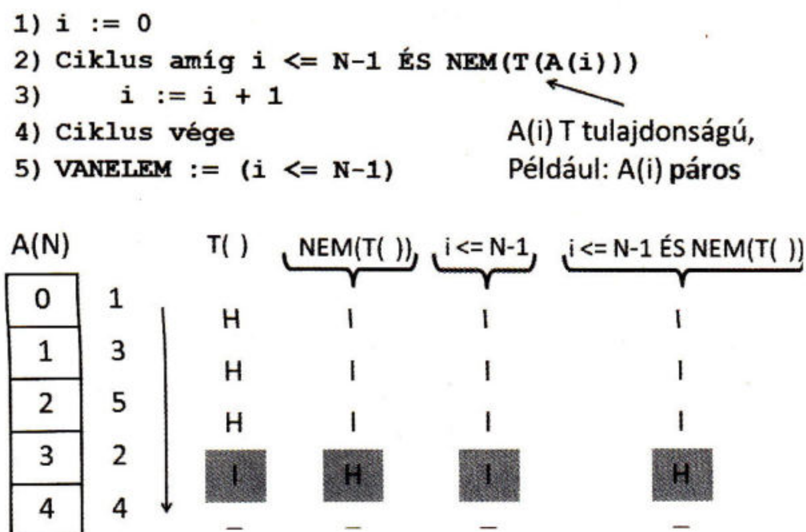
zuk, hogy az A(N) listában található-e T tulajdonságú elem (utófeltétel). A válasz igen-nem jellegű, tehát logikai változóban tárolhatjuk.

A T tulajdonságot egy logikai függvényként adjuk meg. Például jelentse T tulajdonság a *páros számokat.* Ekkor T(A(i)) jelentése: *A(i) páros,* és így fogjuk konkrétan Python nyelvbe átültetni: *A[i] % 2 == 0.*

Működés

Először megvizsgáljuk a lista 0. elemét (2. sorban: T(A(i))). Ha T tulajdonságú, vége a keresésnek, a VANELEM változót igazra állítjuk (5. sor). Ha nem T tulajdonságú, vesszük a következő listaelemet (3. sor), és megvizsgáljuk.

Ennyiből kiderül, hogy feltételes ciklust kell alkalmaznunk (3–5. sor), hiszen nem minden esetben kell végigmenni a listán.



A ciklusból két ok miatt kerülhetünk ki: találtunk megfelelő elemet (T(A(i)), *vagy* nincs több eleme a listának (i > N-1). Mivel a while ciklusban a bennmaradási feltételt kell megfogalmazni, most a kifejezés ellenkezőjét kell megadni (2. sor). Tehát addig kell az ismétlést folytatni, míg van még listaelem, tehát i <= N-1 ÉS nem találtunk T tulajdonságú elemet, azaz NEM(T(A(i))).

A ciklusból tehát két ok miatt léphettünk ki: találtunk megfelelő tulajdonságú elemet, vagy elfogytak a vizsgálandó elemek. A végeredmény megadásához el kell döntenünk, melyik eset történt meg. Legegyszerűbb azt megvizsgálni, hogy van-e még listaelemünk, azaz i <= N-1 igaz-e. Ha igen, akkor csak azért léphettünk ki a ciklusból, mert volt megfelelő tulajdonságú elem. Így a VANELEM értékét igazra állítjuk, egyébként pedig hamisra (5. sor).

Akinek a szöveg alapján nehéz végigkövetni a ciklusfeltétel alakulását, próbálja meg az algoritmus alatt található táblázatok alapján.

A felső táblázat mutatja azt az esetet, amikor van a listában megfelelő tulajdonságú elem, azaz a példában páros szám. Itt a ciklusból történő kilépés a 3-as indexű elemnél következik be. Mivel ott található az első páros szám, T(A(3)) igaz. Ha T(A(3)) igaz, NEM(T(A(3))) hamis. A két részfeltétel egymással ÉS kapcsolatban

van, így ha az egyik részfeltétel hamis, az egész feltétel is hamis, tehát kilépünk a ciklusból. A táblázat alsó sora azért lett kihúzva, mert odáig az algoritmus nem jut el ebben az esetben.

A(N)		T( )	NEM(T( ))	i <= N-1	i <= N-1 ÉS NEM(T( ))
0	1	H	I	I	I
1	3	H	I	I	I
2	5	H	I	I	I
3	1	H	I	I	H
		–	–	H	H

Az alsó táblázat mutatja be azt az esetet, amikor nincs a listában megfelelő elem (nincs páros szám), ezért a futóindex, azaz az i változó túlmegy a lista utolsó elemén. Tehát i <= N-1 hamis. A két részfeltétel egymással ÉS kapcsolatban van, így ha az egyik részfeltétel hamis, az egész feltétel is hamis, tehát kilépünk a ciklusból.

Változatok

Gyakran előfordul, hogy *kétdimenziós eldöntést* kell alkalmaznunk[55]. Például kétdimenziós a listánk, vagy egy lista adatait keressük egy másik listában.

Az algoritmus két egymásba ágyazott eldöntés tételből áll. A belső ciklus (4–7. sor) ismétli az egy soron belüli léptetést (6. sor) és a konkrét listaelem vizsgálatát (5. sorban: T(A(i))), míg a külső ciklus végzi el a következő sorra történő léptetést (9. sor).

Találat csak a belső ciklusban lehet. Ha viszont van találat, a külső ciklusból is ki kell lépni. A találat jelzését a külső ciklus felé a VANELEM logikai változó végzi el. Ha a belső eldöntés tételét megvalósító ciklus talál elemet, azaz j <= M-1 igaz, akkor a VANELEM értékét igazra állítjuk (8. sor). Mikor következő alkalommal a vezérlés a külső ciklusra kerül, NEM(VANELEM) értéke hamis lesz, azaz kilépünk a külső ciklusból is (3. sor). A korrekt működéshez kezdetben VANELEM értékének hamisnak kell lennie (1. sor), azaz kezdetben nincs találat.

```
Adott: A(N,M), egy T tulajdonság
1) VANELEM := hamis
2) i := 0
3) Ciklus amíg i <= N-1 és NEM(VANELEM)
4)      j := 0
5)      Ciklus amíg j <= M-1 és NEM(T(A(i,j)))
6)           j := j + 1
7)      Ciklus vége
8)      Ha j <= M-1 akkor VANELEM := igaz
9)                  különben i := i+1
10)     Elágazás vége
11)Ciklus vége
```

55 Ez az algoritmus a szakkönyvekben külön nem szerepel. A profik feltételezik, hogy az egydimenziós változat alapján ezt mindenki meg tudja írni. Tanári tapasztalatom az, hogy szokott nehézségeket okozni.

A belső ciklusnak két okból lehet vége, ezeket az eseteket szét kell választanunk (8–9. sor). Ha volt megfelelő tulajdonságú elem az i. sorban, azaz j <= M-1 igaz, akkor a VANELEM segítségével jelezzük a külső ciklusnak, hogy nem kell tovább működni. Máskülönben vége az i. sornak úgy, hogy nem volt találat. Ekkor a következő sorra kell továbblépni (9. sor).

Ha a listában nem volt sehol megfelelő tulajdonságú elem, az i futóindex túlmutat a lista utolsó során, és a külső ciklus az i <= N-1 feltétel nem teljesülése miatt véget ér anélkül, hogy a VANELEM értéke igazra állítódott volna.

Feladatok

1. Töltsünk fel egy 10 elemű, egész típusú tömböt véletlenszerűen 1 és 6 közötti egész számokkal (dobókocka). Írjuk ki a képernyőre, van-e a tömbben 6-os. Ellenőrzésként írassuk ki a tömb elemeit is a képernyőre.
2. Kérjünk be a felhasználótól egy szót, és döntsük el, hogy tartalmaz-e magánhangzót. Amennyiben tartalmaz, írjuk ki, hogy "Van benne magánhangzó.". Ha nincs, akkor írjuk ki, hogy "Nincs benne magánhangzó.". A begépelendő szóról feltételezhetjük, hogy csak az angol ábécé kisbetűit tartalmazza.
3. Robotot vezérelnek kódsorral. A helyes sorozatokban csak a robot által értelmezhető négy karakter szerepelhet: "E" az előre, "B" a balra, "J" a jobbra, "H" a hátra haladást jelenti egy-egy egységgel. Készítsünk programot, ami eldönti egy felhasználó által beadott kódsorról, hogy helyes-e.
4. A *6p_kep.txt* fájlban egy 50×50 képpontos kép képpontjainak RGB kódjai vannak három tömbben (r, g, b) tárolva. Az azonos indexű tömbelemek alkotnak egy képpontot. Másoljuk a tömböket vágólapon át a programunkba. Kérjünk be a felhasználótól egy RGB kódot. Állapítsuk meg a program segítségével, hogy a bekért szín megtalálható-e a képen!
5. Töltsünk fel egy listát n db véletlenszerű fej (F) vagy írás (I) értékkel. n értékét a felhasználó adhatja meg. Ellenőrzésképp írassuk ki a lista elemeit a képernyőre egymás mellé, szóköz nélkül. Mennyi volt a leghosszabb, megszakítás nélküli írás sorozat hossza?

7. LINEÁRIS KERESÉS

Bevezető példa

Egy listában (A) 1…6 véletlen egész számokat tárolunk (virtuális dobókocka dobás eredményeit). Van-e közöttük páros szám? Ha van, akkor hányadik dobással sikerült először páros számot dobni?

Elő- és utófeltétel

A lineáris keresés az eldöntés egy továbbfejlesztett változata, ahol megadjuk a T tulajdonságú elem első előfordulását (index) a listában, ha létezik ilyen elem.

```
1) i := 0
2) Ciklus amíg i <= N-1 ÉS NEM(T(A(i)))
3)       i := i + 1
4) Ciklus vége
5) Ha i <= N-1
6)     akkor Ki: i
7)     különben Ki: „Nincs megfelelő elem."
8) Elágazás vége
```

Legyen adott egy A(N) lista és egy T tulajdonság (előfeltétel). Adjuk meg a lista egy T tulajdonságú elemének indexét, ha létezik T tulajdonságú elem. Ha nem létezik, írjuk ezt ki üzenet formájában a felhasználónak (utófeltétel).

A T tulajdonságot egy logikai függvényként adjuk meg. Például jelentse T tulajdonság a *páros számokat.* Ebben az esetben T(A(i)) jelentése: *A(i) páros*, és így fogjuk konkrétan Python nyelvbe átültetni: *A[i] % 2 == 0.*

Működés

```
1) VANELEM := hamis
2) i := 0
3) Ciklus amíg i <= N-1 és NEM(VANELEM)
4)     j := 0
5)     Ciklus amíg j <= M-1 és NEM(T(A(i,j)))
6)         j := j + 1
7)     Ciklus vége
8)     Ha j <= M-1 akkor VANELEM := igaz
9)                 különben i := i+1
10)    Elágazás vége
11) Ciklus vége
12) Ha VANELEM
13)    akkor Ki: i,j
14)    különben Ki: „Nincs ilyen elem."
15) Elágazás vége
```

Az algoritmus működése nagyrészt megegyezik az eldöntésnél megismerttel. Különbség a végén található: ha az i változó értéke listán belül maradt kilépéskor (5. sor), akkor találtunk megfelelő tulajdonságú elemet, tehát

kiírjuk az elem helyét, azaz az indexét (6. sor). Ellenkező esetben (ha i túlfutott a listán), nincs T tulajdonságú elem, így ezt írjuk ki a képernyőre (7. sor).

A lineáris keresés csak az első T tulajdonságú elemet találja meg, nem az összes T tulajdonságút.

Változatok

Lineáris keresést is szokás mátrixon végrehajtani. Ha létezik T tulajdonságú elem, annak mindkét koordinátájára (mindkét indexére) szükség szokott lenni. Az algoritmus alapját a kétdimenziós eldöntés tétele képezi.

Feladatok

1. A *6p_igeny.txt* fájl listáiban az azonos indexű elemek egy-egy liftigénylést tárolnak a jelzés sorrendjében. Egy igény hat számból áll: az első három szám az időt adja meg (óra, perc, másodperc szám), a negyedik a csapat sorszáma, az ötödik az induló, a hatodik a célszint sorszáma. Előfordul, hogy egyik vagy másik szerelőcsapat áthágja a szabályokat, és egyik szintről gyalog megy a másikra. (Ezt onnan tudhatjuk, hogy más emeleten igényli a liftet, mint ahova korábban érkezett.) Generáljunk véletlenszerűen egy létező csapatsorszámot (24 csapat lehet). Határozzuk meg, hogy a vizsgált időszak igényei alapján lehet-e egyértelműen bizonyítani, hogy ez a csapat vétett a szabályok ellen. Ha igen, akkor adjuk meg, hogy melyik két szint közötti utat tették meg gyalog, ellenkező esetben írjuk ki a "Nem bizonyítható szabálytalanság" szöveget. Másoljuk a listákat vágólapon át a programunkba. (Teszthez: 3-as csapat csalt, 51. és 50. szint között, 13-as csapat nem csalt, 17-es csapat nem liftezett.)
2. A *6p_kep.txt* fájlban egy 50 × 50 képpontos kép képpontjainak RGB kódjai vannak három listában (r, g, b) tárolva. Az azonos indexű elemek alkotnak egy képpontot. Másoljuk a listákat vágólapon át a programunkba. Az 50 × 50-es képen egy sárga RGB (255, 255, 0) színű téglalap van. Határozzuk meg program segítségével a bal felső és a jobb alsó sárga képpontnak a helyét (sor, oszlop), majd határozzuk meg, hogy a sárga téglalap hány képpontból áll. (A képpontok számozása 1-től indul, de a tömbök 0-tól kezdődnek.)
3. Mindenkinek, aki az egyszámjáték egy fordulójában részt kíván venni, tippelnie kell egy számra 1 és 99 között. A játékot az nyeri, aki a legkisebb olyan számra tippelt, amelyre csak ő tippelt egyedül, ha nincs ilyen szám, akkor a fordulónak nincs nyertese. Egy forduló adatai vannak az alábbi listában. Határozzuk meg, és írjuk ki a nyertes tipp értékét, valamint azt, hogy hányadik tippelő nyert. Ha

nem volt nyertes tipp a vizsgált fordulóban, akkor a "Nem volt egyedi tipp a megadott fordulóban!" szöveget jelenítsük meg. Használjuk teszteléshez a következő listákat:
tipp = [8, 1, 2, 1, 4, 4, 2, 4, 9]
tipp = [3, 3, 1, 1, 4, 3, 1, 3, 4]
tipp = [6, 3, 4, 3, 3, 4, 2, 3, 2]

4. A *veetel.txt* fájl tartalmazza a rádióamatőrök által feljegyzett üzeneteket. Minden sorpár egy-egy feljegyzést tartalmaz. A sorpár első sorában két szám áll egymástól szóközzel elválasztva, az első a nap sorszáma, a második pedig a rádióamatőré. Ha a megfigyelés során láttak farkasokat, akkor az üzenet két egész számmal folytatódik, amelyeket / jel választ el egymástól, és a látott kifejlett és kölyök egyedek számát jelölik, amelyet szóköz követ. Más esetben nem szám az első karakter. A # a nem azonosítható, tehát hibás karaktereket mutatja. Olvassuk be egy nap és egy rádióamatőr sorszámát, majd írjuk a képernyőre a megfigyelt egyedek számát (a kifejlett és kölyök egyedek számának összegét). Ha nem volt ilyen feljegyzés, a "Nincs ilyen feljegyzés." szöveget jelenítsük meg. Ha nem volt megfigyelt egyed, vagy számuk nem állapítható meg, a "Nincs információ." szöveget jelenítsük meg. Amennyiben egy számot közvetlenül # jel követ vagy előz meg, akkor a számot tekintsük nem megállapíthatónak.

8. KIVÁLOGATÁS

Bevezető példa

Egy listában (A) 1...10 véletlen egész számokat tárolunk. Válogassuk ki egy B listába a páros számokat.

Elő- és utófeltétel

Legyen adott egy A(N) és egy B(M) lista, valamint egy T tulajdonság (előfeltétel). Amikor a B(M) listába átmásoljuk az A(N) listában található T tulajdonságú elemeket (utófeltétel), a kiválogatás tételéről beszélünk.

A T tulajdonságot egy logikai függvényként adjuk meg. Például jelentse T tulajdonság a *páros számokat*. Ekkor T(A(i)) jelentése: *A(i) páros,* és így fogjuk konkrétan a Pythonba átültetni: *A[i]* % 2 == 0.

Működés

A számláló ciklussal végighaladunk az A(N) lista minden egyes elemén (1–4. sor). Minden elemet megvizsgálunk, hogy T tulajdonságú-e (2. sor). Ha igen, betesszük a B(M) következő elemébe az A lista aktuális T tulajdonságú elemét (3. sor).

```
1) Ciklus i := 0-től N-1-ig
2)      Ha T(A(i)) akkor
3)         B() következő eleme A(i)
4) Ciklus vége
```

Változatok

A kiválogatott elemeket nem mindig listában helyezzük el. Gyakran közvetlenül a képernyőre, esetleg fájlba írjuk. Ekkor természetesen nincs szükség a B(M) listára.

Az is előfordulhat, hogy az A(N) lista helyett fájlból válogatunk ki T tulajdonságú adatokat.

Végül megemlítendő a szétválogatás tétele, ami egy továbbfejlesztése a kiválogatásnak. Szétválogatáskor az A(N) lista minden elemét valamilyen feltétel vagy feltételrendszer alapján külön listákba osztjuk szét. Ilyenkor persze módosul a megismert algoritmus: kétfelé válogatáskor az elágazás hamis ágát is használjuk. Többfelé válogatáskor többirányú elágazást használunk.

Feladatok

1. Írassuk ki a képernyőre minden nap első fuvarjának összes adatát a 6p_tavok.txt fájlban található listák alapján. Minden fuvar külön sorba kerüljön. A sorokon belül az adatok legyenek szóközzel elválasztva. Az azonos indexű elemekben vannak egy fuvar adatai. Másoljuk a listákat vágólapon át a programunkba.
2. Írassuk ki egy *joletsor.txt* fájlba a páros számú telkek összes adatát a 6p_telkek.txt fájlban található listák alapján. Ha nem tanultunk fájlkezelést, akkor a képernyőre. Minden telek külön sorba kerüljön. A sorokon belül az adatok legyenek szóközzel elválasztva. Az azonos indexű elemek tartoznak egy telekhez. Másoljuk a listákat vágólapon át a programunkba.

3. Írassuk ki a képernyőre a legszélesebb telkek házszámát és szélességét a 6p_telkek.txt fájlban található listák alapján. A szélességet egyszer az elején írjuk ki. Az azonos indexű elemek tartoznak egy telekhez. Másoljuk a listákat vágólapon át a programunkba.
4. Kérjünk be a felhasználótól egy 3 karakteres szórészletet. Írjuk ki a képernyőre a szólétra-építés szabályai szerint hozzá tartozó ötkarakteres szavakat a szoveg.txt fájlból. A kiírásnál a szavakat egy-egy szóközzel válasszuk el. Ha nincs ilyen szó, akkor írjuk ki: "Nincs megfelelő szó." A szólétra-építés egy olyan játék, amikor adott egy szó közepe, például *isz,* amit a létra fokának nevezünk. Ennek a szócsonknak az elejére és a végére kell egy-egy betűt illesztenünk úgy, hogy értelmes szót hozzunk létre, például *hiszi* vagy *liszt*. Ezt az értelmes szót a játékban létraszónak nevezzük.

9. RENDEZÉS

Bevezető példa

Egy listában (A) egy összejövetel helyszínére érkező résztvevők névsorát tároljuk az érkezésük sorrendjében. Rendezzük ábécésorrendbe a résztvevők névsorát.

Elő- és utófeltételek

elsod	masod	harmad
1	2	0
2	2	5
1	1	7
2	3	7
2	1	8

elsod	masod	harmad
1	2	0
1	1	7
2	2	5
2	3	7
2	1	8

Legyen adott egy A(N) lista, valamint a lista lehetséges elemei között lehessen sorrendet megállapítani[56]. Tipikus esetben ez a valós (egész, stb.) számok között lehet a hagyományos kisebb-nagyobb viszony, karakterek vagy karakterláncok között az ábécésorrend (előfeltétel). Cserélgessük a listaelemeket úgy, hogy bármely két egymást követő közül a nagyobb indexű ne legyen

56 Kicsit precízebb, matematikai megfogalmazásban: legyen értelmezve a lehetséges elemeken egy rendezési reláció.

kisebb (utófeltétel). Így egy növekvően rendezett listát kapunk eredményül. Ezt az eredményt előállító algoritmusokat nevezzük rendezéseknek.

Szokás összetett adatszerkezeteket, például rekord tartalmú listákat is rendezni. Ebben az esetben arra kell figyelni, hogy a teljes rekordot (sort) egyben kell mozgatni, ellenkező esetben összekeverednek az adatok. A mellékelt táblázatot (ami rekord tartalmú lista) *elsod* mező szerint rendeztük[57].

A rendezési algoritmusokról általában

Többféle olyan algoritmus is létezik, ami megfelel a rendezés elő- és utófeltételének, azaz többféleképpen lehet egy listát rendezni. Ezek az algoritmusok különböznek egymástól hatékonyságukban, bonyolultságukban. A hatékonyság gyakran azon is múlik, milyen rendezettségű a lista kezdetben. Ha a kiindulási elemek eleve majdnem rendezettek (például egy rendezett listához hozzáfűzünk a végén egy új elemet), a buborékos rendezés jó választás. Véletlenszerű rendezettség esetén a quick-sort és shell-sort rendezések nagyon hatékonyak. Viszont ezek az algoritmusok viszonylag bonyolultak.

Érettségin nem kértek eddig konkrét rendezési algoritmust, viszont valahogyan rendezni szinte minden feladatsorban kellett. Mivel érettségin nem használható segédeszköz, a hatékonyságot pedig nem értékelik, érdemes kiválasztani a legkönnyebben megtanulható rendezési módot. Javaslom a buborékos rendezést.

A Python listái rendezhetők algoritmus ismerete nélkül is a **sort** parancs segítségével. Azonban, ha több listában párhuzamosan kell mozgatni az adatokat, ez a parancs nem használható egyszerűen. Ugyanez vonatkozik a később említendő összetett rendezésekre. Persze adattranszformáció után használható a **sort** parancs, ha utána egy újabb transzformációval visszaállítjuk az eredeti adatokat. Ennél talán könnyebb az algoritmust alkalmazni.

Rendezés neve	Mikor hatékony?
Buborékos	Rendezetthez adatot fűzünk
Beillesztéses	Fordítottan rendezettre
Maximum-kiválasztásos	Véletlen rendezettre (kevéssé)
Quick-sort	Véletlen rendezettre
Shell-sort	Véletlen rendezettre

57 A listaindexek nincsenek feltüntetve, de nyilván soronként egymás után rendre: 0, 1, 2,...

Buborékos rendezés

Működését pontokba szedtem, hogy könnyebb legyen megtanulni és ez alapján felépíteni az algoritmust.

1. A buborékos rendezés mindig csak szomszédos elemekkel foglalkozik.
2. Elindul a lista végéről (1. sorban 10 és 5).
3. Összehasonlítja a két utolsó elemet, ha nem jó a sorrendjük, felcseréli őket (a 2. sorban az 5 és 10 sorrendje már megfelelő).
4. Egy elemmel előrébb lép (2. sor 2 és 5), és megismétli az előző műveletet (esetünkben a 2 és 5 sorrendje jó, tehát nem cseréli fel őket).
5. Ezt ismétli, míg a lista elejéig nem jut (6. sor). Bármilyen is legyen az elemek elrendezése, biztosak lehetünk benne, hogy az első elem (a 2) a helyére került. Ezért legközelebb elegendő a vizsgálatot a második elemig végezni.
6. Visszaugrik a lista végére, és kezdi ismét visszafelé haladva a vizsgálatot és cserét (7. sorban 5 és 10-zel indul, 2. oszlop 1. sorában van az utolsó vizsgálat).
7. A lista második eleméig végzi csak el a vizsgálatot. A lista második eleme is a helyére kerül (2. oszlop 2. sorban az 5).
8. Így folytatódik, míg el nem fogynak az összehasonlítandó elemek[58].

Ez a rendezés onnan kapta a nevét, hogy minden menetben a legkisebb elem a lista tetejére jut, mint ahogyan a buborék felszáll a vízben.

Az algoritmust két egymásba ágyazott ciklus és azon belül egy feltételes elágazás alkotja.

```
1. 12 6 14 2 10 5      10. 2 12 5 6 14 10
2. 12 6 14 2 5 10      11. 2 5 12 6 14 10
3. 12 6 14 2 5 10      12. 2 5 12 6 10 14
4. 12 6 2 14 5 10      13. 2 5 12 6 10 14
5. 12 2 6 14 5 10      14. 2 5 6 12 10 14
6. 2 12 6 14 5 10      15. 2 5 6 12 10 14
7. 2 12 6 14 5 10      16. 2 5 6 10 12 14
8. 2 12 6 14 5 10      17. 2 5 6 10 12 14
9. 2 12 6 5 14 10
```

Ha j-vel jelöljük a belső ciklusváltozót, akkor a szomszédos elemekkel való munka azt jelenti, hogy az indexük a cikluson belül mindenhol *j* és *j-1* lehet. Más index nem fog előfordulni.

A rossz sorrendet a 3. sorban található elágazás vizsgálja: *a(j-1) > a(j)*.

Ha a sorrend rossz, a két elemet megcseréljük (4. sor).

---

58 Egyszer kövessük végig teljesen a mellékelt számsorokon, hogyan történik a rendezés. Lehet találni a weben jó animációkat a buborékos rendezésre. Abból is érdemes egyet megnézni.

```
1) Ciklus i := 0-től N-2 -ig
2)      Ciklus j := N-1-től j>i-ig (-1)-vel
3)           Ha a(j-1) > a(j) akkor
4)                a(j-1), a(j) := a(j), a(j-1)
5)      Ciklus vége
6) Ciklus vége
```

A belső ciklus egyszer végighalad a listán (2–5. sor). Mivel visszafelé halad, a növekménye *-1*. A listánk N elemű, de a Pythonban 0-val kezdődik az indexelés, ezért az utolsó eleme N-1. Ezért indul *j:= N-1*-gyel a ciklus (2. sor).

A külső ciklus a listán történő végighaladásokat ismétli (1–6. sor). Kezdetben az értéke *i := 0,* hiszen az első menetben az első elem kerül a helyére, annak indexe pedig a Pythonban 0. Innen növekednie kell egyesével, mert minden menetben eggyel kevesebb elemet kell már csak vizsgálni az elejéről számítva. Mivel két szomszédos elemet hasonlítunk össze, és a lista eredetileg N elemű, ebből következően N-1 összehasonlítás lesz. (Aki nem hiszi, nézze meg a példában szereplő számokon: 6 szám között 5 összehasonlítás történik.) Mivel a ciklusváltozó 0-tól indul, az N-1 db lépésnek megfelelő végérték az *N-2* lesz.

A belső ciklus végértéke *j>i.* Ha megengednénk az egyenlőséget, j értéke lehetne 0, és mivel j-1 is szerepel az algoritmusban, az index negatív lenne, ami hibát eredményezne.

A 3. sorban a relációs jel megfordításával csökkenő sorrendben rendezhetjük a listánkat.

Összetett rendezés fogalma

Összetett a rendezés, ha egyszerre több szempont alapján történik.

A példaként mellékelt táblázatban az azonos elsődleges (elsod) mezővel rendelkező adatsorok között a sorrendet a másodlagos (masod) határozza meg. Másképpen megfogalmazva: *elsod* az elsődleges rendezési szempont, a *masod* a másodlagos rendezési szempont[59].

elsod	masod	harmad
1	2	0
2	2	5
1	1	7
2	3	7
2	1	8

elsod	masod	harmad
1	1	7
1	2	0
2	1	8
2	2	5
2	3	7

Az érettségin előszeretettel adnak összetett rendezési feladatot annak felmérésére, ki mennyire mélyen sajátította el a rendezések megértését.

59 Ezt a példát is egy érettségi feladatsor ihlette. Az elsődleges szempont a nap sorszáma, a másodlagos a napon belül a fuvar sorszáma, a harmadik oszlop a megtett kilométerek száma.

Összetett rendezés többszöri rendezéssel

A módszer lényege leolvasható az ábráról[60]:

1. Először is rendezzük végig az adattáblát az elsődleges szempont szerint (1. sor). A mellékelt algoritmusban nem részleteztem az elsődleges rendezést, csak utaltam rá egy mondattal.
2. Alkalmazzuk ismét a rendezést, de a rekordokat csak azonos elsődleges mezők között kell cserélnünk (5. sor)[61].

e	m	h
1	2	0
2	2	5
1	1	7
2	3	7
2	1	8

➡

e	m	h
1	2	0
1	1	7
2	2	5
2	3	7
2	1	8

➡

e	m	h
1	1	7
1	2	0
2	1	8
2	2	5
2	3	7

```
1) Egyszerű rendezés elsődleges szemponttal
2) Ciklus i := 0-től N-2 -ig
3)     Ciklus j := N-1 -től j>i -ig (-1) -vel
4)         Ha a(j-1).masod > a(j).masod És
5)            a(j-1).elsod = a(j).elsod akkor
6)            a(j-1), a(j) := a(j), a(j-1)
7)     Ciklus vége
8) Ciklus vége
```

Összetett rendezés adattranszformációval

Egyetlen rendezési algoritmusra egyszerűsíthetjük az összetett rendezést, ha átmenetileg egy alkalmas virtuális oszlopot tudunk képezni, és aszerint rendezni a rekordokat.

Ilyen megfelelő oszlopot kapunk a példatáblázatban, ha megszorozzuk az elsődleges szempont értékét 10-zel, majd hozzáadjuk a másodlagos

elsod	masod	barmi	10*e+m
1	2	0	12
2	2	5	22
1	1	7	11
2	3	7	23
2	1	8	21

➡

10*e+m	elsod	masod	barmi
11	1	1	7
12	1	2	0
21	2	1	8
22	2	2	5
23	2	3	7

```
1) Ciklus i := 0-től N-2-ig
2)     Ciklus j := N-1 -től j>i -ig (-1) -vel
3)         Ha a(j-1).elsod*szor + a(j-1).masod
           > a(j).elsod*szor + a(j).masod akkor
4)            a(j-1), a(j) := a(j), a(j-1)
5)     Ciklus vége
6) Ciklus vége
```

60 A mezőneveket rövidítettem helytakarékossági okokból.

61 Végezhetnénk harmadlagos szempont szerint is rendezést, hasonlóan a másodlagoshoz. Ekkor a harmadlagos szempont szerint cserélgetnénk azok között a rekordok között, ahol az elsődleges és másodlagos szempont szerinti adat megegyezik.

szempontot. A példában kicsit sötétebb háttérrel jelölt virtuális oszlop szerint elvégezve a rendezést, ugyanazt a helyes eredménytáblát kapjuk.

A példában megállapított képlet csak akkor működik, ha a másodlagos szempont értéke mindig 10-nél kisebb. Persze, ha előre tudjuk, hogy nem kisebb, mint 10, szorozhatunk 100-zal is. Ha 100-nál sem kisebb, 1000-rel, stb. Általánosítva, ismernünk kell a másodlagos szempont legnagyobb értékét. (Az ábrán látható algoritmusban a szor változóban tároljuk a szorzót – 3. sor). A példában ennek értéke 10.)

Ugyanez a módszer működik szöveges mezőkre is, ha az elsődleges szemponthoz hozzáfűzzük (hozzáadjuk) a másodlagos szempontot.

Összefoglalóan azt mondhatjuk, hogy az átalakításhoz előismeretekkel kell rendelkeznünk az adatokról. Ha ilyen ismeretekkel nem bírunk, marad a kétszeri rendezés.

Feladatok

1. Rendezzük a 6p_telkek.txt fájlban található házszám-listát növekvő sorrendbe. Másoljuk a házszám-listát vágólapon át a programunkba. (A többi listával nem kell törődni ebben a feladatban.) Jelenítsük meg a képernyőn az eredeti listát, majd az eredményt.
2. Rendezzük a 6p_telkek.txt fájlban található telkek minden adatát szélesség szerint csökkenő sorrendbe. Az azonos indexű elemek tartoznak ugyanahhoz a telekhez. Másoljuk a listákat vágólapon át a programunkba. Ne felejtsük el, hogy mindhárom listában mozgatni kell az adatokat. Jelenítsük meg a képernyőn az eredményt.
3. Rendezzük a 6p_tavok.txt fájlban található fuvarok minden adatát először nap szerint, majd azon belül fuvar szerint. Az azonos indexű elemek tartoznak ugyanahhoz a fuvarhoz. Másoljuk a listákat vágólapon át a programunkba. Jelenítsük meg a képernyőn az eredményt.
4. Írassuk ki a szoveg.txt fájlban található szavakat egy *szavak.txt* fájlba fordított ábécésorrendbe. A szavak között egy szóközt hagyjunk ki. (Idő kell a program lefutásához, türelem!)
5. A szoveg.txt fájlban található ötkarakteres szavakból csoportosítsuk azokat a szavakat, melyek ugyanannak a hárombetűs szórészletnek a létraszavai. Hozzunk létre egy *letra.txt* állományt, amelybe ezeket a szavakat írjuk az alábbiak szerint:
   - Minden szó külön sorba kerüljön.
   - Csak olyan szó szerepeljen az állományban, aminek van legalább egy párja, amivel egy létrát alkotnak (azaz első és utolsó karakter nélkül megegyeznek).
   - Az egy létrához tartozó szavak közvetlenül egymás után helyezkedjenek el.
   - Két létra szavai között egy üres elválasztó sor legyen.

```
5. feladat:

megye
vegye
hegyi
tegye

lehet
teher
mehet

tejes
fejet
fejen

...
```

10. FELADATSOROK MEGOLDÁSA

A vázlat szerepe

Első dolgunk legyen átolvasni az egész programozás feladatsort egyben. Ekkor felmérhetjük, melyik feladatot mennyi időráfordítással tudjuk megoldani. "Profi" vizsgázó azért sem esik neki az elejétől, mert a feladatok egymásra épülhetnek, és emiatt esetleg módosítani kell egy már megírt kódot, ami időveszteség.

Az előzetes felmérés során érdemes rövid jegyzetet készíteni, amiben az egyes feladatokhoz tartozó gondolatainkat odaírjuk. Megtehetjük ezt a leendő forráskódba megjegyzésként. Miközben a programon dolgozunk, eszünkbe juthatnak ötletek a többi feladathoz, így folyamatosan kiegészíthetjük a jegyzeteket. Ha jól dolgozunk, utána már nem is kell gondolkodni a feladatok megoldásán, csak kódolni a már megismert algoritmusok szerint.

Az alábbi három témakör szempontjából mindenképpen érdemes a feladatsorokat kijegyzetelni, de bátran fűzzük hozzá az egyéni észrevételeinket, illetve a feladat különlegességeit (pl. formázott kiíratás) is.

Adatszerkezet

A jegyzetben érdemes rögzíteni, milyen összetett adatszerkezetbe kell vagy érdemes beolvasni az adatokat a fájlból (feladatonként eltérő lehet):

- Semmilyenbe, ha soronként feldolgozható a feladat. Ekkor az eredményt közvetlenül a képernyőre vagy fájlba lehet írni. Lehet akár elvárás is, hogy ne legyen összetett adatszerkezet a memóriában. Ekkor azt szokták írni a feladathoz: "Működjön tetszőleges hosszúságú fájlra." Ilyen feladatsor volt például a *Törtek* (2012. május, idegen nyelvű).
- Listába, ha a fájlban egyetlen oszlopban (netán sorban) vannak az adatok. Például a *Fehérje* feladatsor (2006. május).
- Rekordok tartalmú listába, mátrixba vagy "párhuzamos" listákba. Onnan ismerhetjük fel általában ezt a helyzetet, hogy pár oszlopból áll (tipikusan 2–6) a fájl, de sokkal több sorból. Ilyen feladatsor volt például a *Futár* (2012. május), vagy a *Telek* (2010. május, idegen nyelvű).
- Rekord tartalmú mátrixba, vagy "párhuzamos" mátrixokba. Például: *Szín-kép* (2012. október) vagy *Rejtvény* (2011. május, idegen nyelvű).

Előfordulhat, hogy egy feladatsorban többféle adatszerkezet használatát várják el. Ha egyféle kell az egész feladatsorhoz, elég a fájlbeolvasásos feladatnál egyszer odaírni.

Fájlbeolvasás vagy kiíratás

Írjuk le a jegyzetbe, hová kell fájlbeolvasás vagy kiíratás. A fájlbeolvasásnál írjuk oda, melyik módszert fogjuk választani:

- Ismert a sorok száma, tehát for ciklust.
- Ismeretlen a sorok száma, tehát while ciklust.

Esetleg egyéb specialitás van-e? Például nem szóköz az elválasztójel, vagy többszörösen kell elválasztani.

Programozási tételek

Írjuk oda minden feladathoz, milyen programozási tételt vagy tételeket igényel a megoldásuk. Ha többet, írjuk oda az alkalmazásuk sorrendjét. Ha többször kell ugyanazt a tételt alkalmazni, tüntessük fel azt is.

A programozási tételek felismerését kezdetben az Összefoglalásban található táblázat segítheti.

Feladatsorok válogatása gyakorláshoz

Az érettségire készülve a feladatsorokat úgy érdemes összeválogatni, hogy mindenféle legyen benne. Legyen karakterláncokkal kapcsolatos, számokkal kapcsolatos, mátrix-adatszerkezetes, lista-adatszerkezetes, ismert és ismeretlen fájlhosszúságú, mindenféle programozási tételes, megadott algoritmust kódoló, stb.

65. mintafeladat - Szójáték (2011. május) vázlata[62]

A feladatok megoldásához rendelkezésére áll a szoveg.txt fájl, amelybe Gárdonyi Géza *Egri csillagok* című regényéből gyűjtöttünk ki szavakat. Az állományban csak olyan szavak szerepelnek, melyek az angol ábécé betűivel leírhatók, és minden szó csak egyszer fordul elő.

A könnyebb feldolgozhatóság érdekében valamennyi szót csupa kisbetűvel írtunk, szavanként külön sorban. Tudjuk, hogy ebben az állományban a szavak 20 karakternél nem hosszabbak.

---

62 A könyvben található vázlatok részletesebbek, mint amit az olvasó fog írni. Erre az érthetőség kedvéért van szükség.

A szólétra-építés egy olyan játék, amikor adott egy szó közepe, például isz, amit a létra fokának nevezünk. Ennek a szócsonknak az elejére és a végére kell egy-egy betűt illesztenünk úgy, hogy értelmes szót hozzunk létre, például hiszi vagy liszt. Ezt az értelmes szót a játékban létraszónak nevezzük. Az adott szórészlethez minél több létraszót tudunk kitalálni, annál magasabb lesz a szólétra. A cél az, hogy egy megadott szócsonkhoz a lehető legmagasabb szólétrát építsük.

Szórészlet: **isz**
A hozzá tartozó létraszavak:

```
hiszi
liszt
viszi
tiszt
```

Készítsen programot, amely az alábbi feladatokat megoldja. A program forráskódját szavak néven mentse.

Feladat szövege	Vázlat
1. Kérjen be a felhasználótól egy szót, és döntse el, hogy tartalmaz-e magánhangzót. Amennyiben tartalmaz, írja ki, hogy "Van benne magánhangzó." Ha nincs, akkor írja ki, hogy "Nincs benne magánhangzó." A begépelendő szóról feltételezheti, hogy csak az angol ábécé kisbetűit tartalmazza. (Az angol ábécé magánhangzói: a, e, i, o, u.)	eldöntés tétele
2. Írja ki a képernyőre, hogy melyik a leghosszabb szó a szoveg.txt állományban, és az hány karakterből áll. Ha több azonos leghosszabb hosszúságú szó is van a szógyűjteményben, akkor azok közül elegendő egyetlen szót kiírnia. A feladatot úgy oldja meg, hogy tetszőleges hosszúságú szövegállomány esetén működjön, azaz a teljes szöveget ne tárolja a memóriában.	nincs összetett adatszerkezet Fájlbeolvasás: fájl végéig (while) Maximumkiválasztás, csak érték kell, hely nem
3. A magyar nyelv szavaiban általában kevesebb a magánhangzó, mint a mássalhangzó. Határozza meg, hogy az állomány mely szavaiban van több magánhangzó, mint egyéb karakter. Ezeket a szavakat írja ki a képernyőre egy-egy szóközzel elválasztva. A szavak felsorolása után a mintának megfelelően az alábbi adatokat adja meg: • hány szót talált • hány szó van összesen az állományban • a talált szavak hány százalékát teszik ki az összes szónak A százalékot két tizedessel szerepeltesse. Például:130/3000 : 4,33%	nincs összetett adatszerkezet előző fájlbeolvasásba rakható 1. Megszámlálás tétele – szóban magánhangzóra 2. Feltételes megszámlálás – több magánhangzó 3. Megszámlálás – szavak 4. Formázott kiíratás
4. Hozzon létre egy tömb vagy lista adatszerkezetet, és ebbe gyűjtse ki a fájlban található ötkarakteres szavakat. A szoveg.txt állomány legfeljebb 1000 darab 5 karakteres szót tartalmaz. Kérjen be a felhasználótól egy 3 karakteres szórészletet. Írja ki a képernyőre a szólétra-építés szabályai szerint hozzá tartozó 5 karakteres szavakat a tárolt adathalmazból. A kiírásnál a szavakat egy-egy szóköz válassza el. (Teszteléshez használhatja például az "isz" vagy "obo" szórészleteket, mert ezekhez a megadott szövegállományban több létraszó is tartozik.)	Lista max. 1000 elemmel. 1. Fájlkezelés while Kiválogatás – fájlból tömbbe – 5 karakteresek (berakható 2. feladatba) 2. Kiválogatás – tömbből képernyőre létraszavak
5. Az eltárolt 5 karakteres szavakból csoportosítsa azokat a szavakat, melyek ugyanannak a hárombetűs szórészletnek a létraszavai. Hozzon létre egy letra.txt állományt, amelybe ezeket a szavakat írja az alábbiak szerint: • minden szó külön sorba kerüljön • csak olyan szó szerepeljen az állományban, aminek van legalább egy párja, amivel egy létrát alkotnak (azaz első és utolsó karakter nélkül megegyeznek) • az egy létrához tartozó szavak közvetlenül egymás után helyezkedjenek el • két létra szavai között egy üres elválasztó sor legyen *letra.txt* megye vegye hegyi tegye lehet teher mehet	1. Rendezés a belső három betű szerint 2. Kiválogatás – tömbből képernyőre, egyezik-e a megelőzővel, illetve a rá következővel

Feladatsorok

Készítsük el az alábbi érettségi feladatsor megoldási vázlatát, majd az alapján oldjuk meg a feladatsort.

1. FELADATSOR – FUTÁR (2012. MÁJUS)

A nagyvárosokon belül, ha csomagot gyorsan kell eljuttatni egyik helyről a másikra, akkor sokszor a legjobb választás egy kerékpáros futárszolgálat igénybevétele. A futárszolgálat a futárjainak a megtett utak alapján ad fizetést. Az egyik futár egy héten át feljegyezte fuvarjai legfontosabb adatait, és azokat eltárolta egy állományban. Az állományban az adatok rögzítése nem mindig követi az időrendi sorrendet. Azokra a napokra, amikor nem dolgozott, nincsenek adatok bejegyezve az állományba.

A fájlban legalább 10 sor van, és minden sor egy-egy út adatait tartalmazza egymástól szóközzel elválasztva. Az első adat a nap sorszáma, ami 1 és 7 közötti érték lehet. A második szám a napon belüli fuvarszám, ami 1 és 40 közötti érték lehet. Ez minden nap 1-től kezdődik, és az aznapi utolsó fuvarig egyesével növekszik. A harmadik szám az adott fuvar során megtett utat jelenti kilométerben, egészre kerekítve. Ez az érték nem lehet 30-nál nagyobb.

Készítsen programot, amely a tavok.txt állomány adatait felhasználva az alábbi kérdésekre válaszol. A program forráskódját mentse futar néven. (A program megírásakor a felhasználó által megadott adatok helyességét, érvényességét nem kell ellenőriznie, feltételezheti, hogy a rendelkezésre álló adatok a leírtaknak megfelelnek.) A képernyőre írást igénylő részfeladatok eredményének megjelenítése előtt írja a képernyőre a feladat sorszámát (például: 3. feladat:). Ha a felhasználótól kér be adatot, jelenítse meg a képernyőn, hogy milyen értéket vár. Az ékezetmentes kiírás is elfogadott.

1. Olvassa be a tavok.txt állományban talált adatokat, s annak felhasználásával oldja meg a következő feladatokat.
2. Írja ki a képernyőre, hogy mekkora volt a hét legelső útja kilométerben. Figyeljen arra, hogy olyan állomány esetén is helyes értéket adjon, amiben például a hét első napján a futár nem dolgozott.
3. Írja ki a képernyőre, hogy mekkora volt a hét utolsó útja kilométerben.
4. Tudjuk, hogy a futár minden héten tart legalább egy szabadnapot. Írja ki a képernyőre, hogy a hét hányadik napjain nem dolgozott a futár.
5. Írja ki a képernyőre, hogy a hét melyik napján volt a legtöbb fuvar. Amennyiben több nap is azonos, maximális számú fuvar volt, elegendő ezek egyikét kiírnia.

6. Számítsa ki és írja a képernyőre a mintának megfelelően, hogy az egyes napokon hány kilométert kellett tekerni.

```
1. nap: 124 km
2. nap: 0 km
3. nap: 75 km
...
```

7. A futár az egyes utakra az út hosszától függően kap fizetést az alábbi táblázatnak megfelelően. Kérjen be a felhasználótól egy tetszőleges távolságot, és határozza meg, hogy mekkora díjazás jár érte. Ezt írja a képernyőre.

1–2 km	500 Ft
3–5 km	700 Ft
6–10 km	900 Ft
11–20 km	1400 Ft
21–30 km	2000 Ft

8. Határozza meg az összes rögzített út ellenértékét. Ezeket az értékeket írja ki a dijazas.txt állományba nap szerint, azon belül pedig az út sorszáma szerinti növekvő sorrendben, az alábbi formátumban:
   1. nap 1. út: 700 Ft
   1. nap 2. út: 900 Ft
   1. nap 3. út: 2000 Ft
   ...
9. Határozza meg, és írja ki a képernyőre, hogy a futár mekkora összeget kap a heti munkájáért.

Az 1. feladatsor két lehetséges megoldása:

	1. változat	2. változat
1.	Rekord tartalmú tömb fájlbeolvasás: fájl végéig (while)	3 db tömb "párhuzamosan" fájlbeolvasás: fájl végéig (while)
2.	1. Minimum kiválasztás kétszer napra, majd fuvarra 2. Másodikra kell a minimumhely 3. Minimumhely alapján tömbelem kiíratás	1. Összetett rendezés: napra, majd fuvarra 2. A tömb első elemének kiíratása
3.	1. Maximumkiválasztás kétszer napra, majd fuvarra 2. Másodikra kell a maximumhely 3. Maximumhely alapján tömbelem kiíratás	A tömb utolsó elemének kiíratása
4.	Segédtömb: index – nap, érték – fuvarszám 1. Megszámlálás tétele csoportosan 2. Kiválogatás a segédtömbből indexek a képernyőre (s[i] == 0)	
5.	Maximumkiválasztás a 4. feladat segédtömbjén, tömbindex kiíratása	
6.	Segédtömb: index – nap, érték – kilométer 1. Összegzés tétele csoportosan 2. Segédtömb kiíratása	
7.	Függvény készítése (jó lesz a 8. és 9. feladathoz is)	
8.	1. Összetett rendezés (nap, majd fuvar) 2. Sorozatszámítás (függvény a 7. fel.-ból) 3. Fájlba írás (2. és 3. egy ciklusban)	1. Sorozatszámítás (függvény a 7. fel.-ból) 2. Fájlba írás (1. és 2. egy ciklusban)
9.	Sorozatszámítás + Összegzés tétele Sorozatszámítás függvénye a 7. feladatból alkalmazva	

2. FELADATSOR – REJTVÉNY (2011. MÁJUS, IDEGEN NYELVŰ)

Egy weboldalon érdekes rejtvényt tesznek közzé hétről hétre. A rejtvényekben egy N × M-es területre világítótornyokat helyeznek le. Ezeket a tornyokat számmal jelölik. Minden alkalommal az a feladat, hogy a területre el kell helyezni X darab hajót úgy, hogy minden toronyból (vízszintesen és függőlegesen összesen) annyi hajó legyen látható, ahányas szám a tornyot jelképező mezőben van.

			1
		2	
3			

●			1
●		2	
3		●	

A hajókra vonatkozó szabályok a következők:
- Minden hajó egy négyzet nagyságú.
- A hajók nem érintkezhetnek egymással, még átlós irányban sem.
- A hajók nem érinthetik a világítótornyokat, még átlós irányban sem.
- A hajók egymást nem takarják ki. Azaz a világítótoronyból az egy vonalban lévő hajók is látszanak.

A weboldalon ugyanúgy, mint az előző hetekben, egy 10 × 10-es négyzetbe kell elhelyezni 12 darab hajót. A versenyzők által beküldött megfejtéseket alkalmazás segítségével összefűzik egy txt állományba. Ennek a fájlnak az első sora a megfejtések számát tartalmazza, ami maximálisan 20 darab lehet. Minden megfejtés előtt pedig a megfejtő neve található. Az egyes megfejtésekben a vizet 0-val, a világítótornyot egy 1 és 9 közötti számmal, a hajókat pedig 11-es számmal jelölik. A fájlban a számokat egy-egy szóközzel választják el. Például: a megoldas.txt állomány egy részlete. (A példát szabályos táblázatban jelenítjük meg a jobb átláthatóság érdekében.)

```
10
Absolon
 0  0  0  0 11  0 11  0  0  0
11  0  2  0  0  0  0  0  0 11
 0  0  0  0  0  1  0 11  0  0
 0  0  0 11  0  0  0  0  0  0
 0  3  0  0  0  0  0  0 11  0
 0  0  0  0  2  0 11  0  0  0
 0 11  0  0  0  0  0  0  3  0
 0  0  0  0  0  0  3  0  0  0
 0 11  0  3  0  0  0  0 11  0
 0  0  0  0  0 11  0  0  0  0
...
```

A 2. sor 3. oszlopában tehát egy világítótorony van, amelynek sorában és oszlopában összesen 2 hajó lehet. A 2. sor 1. oszlopában és a 2. sor 10. oszlopában egy-egy hajó található.

Készítsen programot, amely a rejtvényre érkező megoldások helyességét ellenőrzi. A program forráskódját rejtveny néven mentse.

Minden feladat megoldása előtt írja a képernyőre a feladat sorszámát. Ha a felhasználótól kér be adatot, jelenítse meg a képernyőn, hogy milyen értéket vár (például az 1. feladat esetén: "Adja meg a torony adatait!"). Az ékezetmentes kiírás is elfogadott.

A feladatok megoldása során feltételezzük, hogy a beolvasott adatok helyesek, ezért azokat sehol nem kell ellenőrizni.

1. Kérje be a felhasználótól egy 10 × 10-es táblára vonatkozóan egy világítótorony pozícióját (a torony helyének sor- és oszlop-száma), és a toronyból látható hajók számát. A rejtvény megfejtését a nagy számmal rendelkező tornyoknál érdemes kezdeni. Ezért, ha a torony értéke nagyobb, mint három, akkor írja ki a képernyőre, hogy "Nehéz torony.", más esetben ne írjon ki semmit.
2. A megadott világítótorony helyzete alapján állapítsa meg, hogy a szabályok szerint a világítótorony körül mely helyekre biztosan nem kerülhet hajó. Az eredményt írassa ki a képernyőre úgy, hogy a tiltott helyek sor- és oszlopazonosítói vesszővel elválasztva külön sorokba kerüljenek. Például, ha a világítótorony a 2, 3 pozícióban van, akkor lásd a mellékelt ábrát.

```
1,2
1,3
1,4
2,2
2,4
3,2
3,3
3,4
```

3. A feladvany.txt állomány tartalmazza az erre a hétre kiadott rejtvényt a már ismert formában. Olvassa be a rejtvényt az állományból, és a megoldas.txt állományban beküldött megoldások közül szűrje ki azokat, amelyek nem az e heti feladványra érkeztek. Ezen megfejtő(k) nevét írja ki a képernyőre. Ha minden megfejtés az e heti feladványra érkezett, akkor írja ki a képernyőre, hogy "Mindegyik megoldás erre a heti feladványra érkezett.".
4. Azok közül a megoldások közül, amelyek erre a heti feladványra érkeztek, állapítsa meg, hogy melyikekben van kevesebb vagy több hajó megadva, mint 12. Írja ki a képernyőre, hogy e szempontból hány darab hibás "megoldás" volt.
5. Hány olyan szabálytalan megoldás született az e heti feladatra, amelyben a hajók száma megfelelő, és egy vagy több hajó elhelyezése a szomszédsági kapcsolatokra vonatkozó szabályoknak nem megfelelő? Az eredményt írja ki a képernyőre.
6. Határozza meg, hogy hány megoldás volt helyes a beküldöttek közül. Az ellenőrzésnél vegye figyelembe az előző pontokban leírtakat, valamint azt, hogy a világítótornyok az értéküknek megfelelő számú hajót látnak-e. A helyes beküldők nevét írja ki a képernyőre.

A 2. feladatsor egy lehetséges megoldása:

1.	Nincs összetett adatszerkezet
2.	Kiválogatás képernyőre – két egymásba ágyazott ciklussal – indexpár kiíratása – feltétel: i != j
3.	1. Rejtvény beolvasása Adatszerkezet: 10 x 10-es mátrix (R néven) Fájlbeolvasás: ismert elemszám, két egymásba ágyazott for ciklus 2. Megoldások beolvasása Adatszerkezet: 20 x 10 x 10-es mátrix (M néven) + karakterlánc tömb (neveknek) Fájlbeolvasás: ismert elemszám, 3 egymásba ágyazott for ciklus 3. Függvény írása kétdimenziós Eldöntés tételével (z. megoldás erre a hétre jött-e, R és M(z)-ben ugyanott vannak az 1–9 számok) 4. Kiválogatás képernyőre – Tulajdonságot az előző függvény adja
4.	1. Megszámlálás tétele – hajók száma a z. megoldásban. 2. Függvény készítése (megvan 12? igaz/hamis) 3. Megszámlálás tétele – hibás megoldások száma (előző és mostani feladat függvényével)
5.	1. Függvény írása – z. megoldásban x, y koordinátán megfelelőek-e a szomszédsági viszonyok (van-e 0-nál nagyobb szám a szomszéd cellákban) – kétdimenziós eldöntés tétele 2. Függvény írása – z. megoldásban jók-e a szomszédsági viszonyok – kétdimenziós eldöntés tétele 3. Megszámlálás tétele – feltétel az előző függvény és a 3., 4. feladat függvényei
6.	1. Megszámlálás tétele – z. megoldásban x, y pozícióból hány hajó látható 2. Függvénnyé alakítani – z. megoldásban x, y pozícióból annyi hajó látható-e, amennyi kell 3. Függvény készítése – kétdimenziós eldöntés tétele – z. megoldásban minden toronyból megfelelő számú hajó látható-e 4. Megszámlálás tétele – feltétel a 3., 4., 5. feladatban, és az ebben a feladatban megírt második függvény alapján 5. Kiválogatás tétele képernyőre (nevek) – feltétel a 3., 4., 5. feladatban, és az ebben a feladatban megírt második függvény alapján

3. FELADATSOR – TÖRTEK (2012. MÁJUS, IDEGEN NYELVŰ)

Készítsen programot, amely az alábbi – közönséges törtekkel kapcsolatos – feladatokat megoldja. A program forráskódját tort néven mentse. A feladatban csak pozitív számokkal kell dolgoznia, és ennek a tulajdonságnak a feldolgozandó fájlban található számadatok is megfelelnek. A felhasználótól bekérendő és a feldolgozandó fájlban található számokról feltételezheti, hogy legfeljebb kétjegyűek.

Minden – képernyőre írást igénylő – részfeladat megoldása előtt írja a képernyőre a feladat sorszámát. Ha a felhasználótól kér be adatot, jelenítse meg a képernyőn, hogy milyen értéket vár. Például az 1. feladat esetén:

```
1. feladat
Adja meg a szamlalot:
```

(Az ékezetmentes kiírás is elfogadott.)

1. Kérjen be a felhasználótól két számot, amely egy közönséges tört számlálója és nevezője. Döntse el, hogy az így bevitt tört felírható-e egész számként. Ha igen, írja ki értékét egész számként, ha nem, írja ki "Nem egész.".
2. A közönséges törteket úgy tudjuk a legegyszerűbb alakra hozni, ha a számlálóját és nevezőjét elosztjuk a két szám legnagyobb közös osztójával, és az így

A 2. feladatsor egy lehetséges megoldása:

1.	Nincs összetett adatszerkezet
2.	Kiválogatás képernyőre – két egymásba ágyazott ciklussal – indexpár kiíratása – feltétel: i != j
3.	1. Rejtvény beolvasása Adatszerkezet: 10 x 10-es mátrix (R néven) Fájlbeolvasás: ismert elemszám, két egymásba ágyazott for ciklus 2. Megoldások beolvasása Adatszerkezet: 20 x 10 x 10-es mátrix (M néven) + karakterlánc tömb (neveknek) Fájlbeolvasás: ismert elemszám, 3 egymásba ágyazott for ciklus 3. Függvény írása kétdimenziós Eldöntés tételével (z. megoldás erre a hétre jött-e, R és M(z)-ben ugyanott vannak az 1–9 számok) 4. Kiválogatás képernyőre – Tulajdonságot az előző függvény adja
4.	1. Megszámlálás tétele – hajók száma a z. megoldásban. 2. Függvény készítése (megvan 12? igaz/hamis) 3. Megszámlálás tétele – hibás megoldások száma (előző és mostani feladat függvényével)
5.	1. Függvény írása – z. megoldásban x, y koordinátán megfelelőek-e a szomszédsági viszonyok (van-e 0-nál nagyobb szám a szomszéd cellákban) – kétdimenziós eldöntés tétele 2. Függvény írása – z. megoldásban jók-e a szomszédsági viszonyok – kétdimenziós eldöntés tétele 3. Megszámlálás tétele – feltétel az előző függvény és a 3., 4. feladat függvényei
6.	1. Megszámlálás tétele – z. megoldásban x, y pozícióból hány hajó látható 2. Függvénnyé alakítani – z. megoldásban x, y pozícióból annyi hajó látható-e, amennyi kell 3. Függvény készítése – kétdimenziós eldöntés tétele – z. megoldásban minden toronyból megfelelő számú hajó látható-e 4. Megszámlálás tétele – feltétel a 3., 4., 5. feladatban, és az ebben a feladatban megírt második függvény alapján 5. Kiválogatás tétele képernyőre (nevek) – feltétel a 3., 4., 5. feladatban, és az ebben a feladatban megírt második függvény alapján

3. FELADATSOR – TÖRTEK (2012. MÁJUS, IDEGEN NYELVŰ)

Készítsen programot, amely az alábbi – közönséges törtekkel kapcsolatos – feladatokat megoldja. A program forráskódját tort néven mentse. A feladatban csak pozitív számokkal kell dolgoznia, és ennek a tulajdonságnak a feldolgozandó fájlban található számadatok is megfelelnek. A felhasználótól bekérendő és a feldolgozandó fájlban található számokról feltételezheti, hogy legfeljebb kétjegyűek.

Minden – képernyőre írást igénylő – részfeladat megoldása előtt írja a képernyőre a feladat sorszámát. Ha a felhasználótól kér be adatot, jelenítse meg a képernyőn, hogy milyen értéket vár. Például az 1. feladat esetén:

```
1. feladat
Adja meg a szamlalot:
```

(Az ékezetmentes kiírás is elfogadott.)

1. Kérjen be a felhasználótól két számot, amely egy közönséges tört számlálója és nevezője. Döntse el, hogy az így bevitt tört felírható-e egész számként. Ha igen, írja ki értékét egész számként, ha nem, írja ki "Nem egész.".
2. A közönséges törteket úgy tudjuk a legegyszerűbb alakra hozni, ha a számlálóját és nevezőjét elosztjuk a két szám legnagyobb közös osztójával, és az így

kapott érték lesz az új számláló, illetve nevező. Az egyszerűsítéshez készítsen egy rekurzív függvényt az alább leírt euklideszi algoritmusnak megfelelően.

```
Függvény lnko(a, b : egész számok) : egész szám
    ha a=b akkor lnko := a
    ha a<b akkor lnko := lnko(a, b-a)
    ha a>b akkor lnko := lnko(a-b, b)
Függvény vége
```

3. Az első feladatban bekért törtet hozza a legegyszerűbb alakra a létrehozott függvény segítségével. Amennyiben nem sikerül az előírt függvényt elkészítenie, alkalmazhat más megoldást, hogy a további feladatokat meg tudja oldani. Az eredményt írja ki a következő formában: 24/32 = 3/4. Amenynyiben a tört felírható egész számként, akkor ebben az alakban jelenjen meg: 24/6 = 4.
4. Két törtet úgy tudunk összeszorozni, hogy a két tört számlálóját összeszorozva kapjuk az eredmény számlálóját, és a két tört nevezőjét összeszorozva kapjuk az eredmény nevezőjét. Kérjen be a felhasználótól egy újabb közönséges törtet a számlálójával és a nevezőjével. Szorozza meg ezzel a törttel az első feladatban bekért törtet. Az eredményt hozza a legegyszerűbb alakra, és ezt írja ki a következő formában:
   24/32 * 12/15 = 288/480 = 3/5
   Amennyiben a tört felírható egész számként, akkor ebben az alakban jelenjen meg:
   24/32 * 8/3 = 192/96 = 2
5. Két közönséges tört összeadásához a következő lépésekre van szükség:
   - Mindkét számot bővíteni kell, azaz mind a számlálóját, mind a nevezőjét ugyanazzal a számmal kell megszorozni. Ezt a bővítést úgy célszerű elvégezni, hogy a közös nevező a két eredeti nevező legkisebb közös többszöröse legyen. Ez lesz az összeg nevezője.
   - A két bővített alakú tört számlálóját összeadjuk, ez lesz az eredmény számlálója. Ehhez készítsen függvényt az alábbiakban leírtak szerint – a korábban elkészített lnko függvény felhasználásával – a legkisebb közös többszörös meghatározására.

```
Függvény lkkt(a, b : egész számok) : egész szám
    lkkt := a * b / lnko(a, b)
Függvény vége
```

6. A függvény segítségével határozza meg a két bekért tört összegét, és ezt adja meg a következő formában. (Amennyiben nem sikerül az előírt függvényt elkészítenie, alkalmazhat más megoldást, hogy a további feladatokat meg tudja oldani.)
   24/32 + 8/3 = 72/96 + 256/96 = 328/96 = 41/12
   Amennyiben az eredmény felírható egész számként, akkor ebben az alakban jelenjen meg:
   22/4 + 27/6 = 66/12 + 54/12 = 120/12 = 10
7. Az adat.txt állományban található műveleteket végezze el, és az eredményeket a korábbi, képernyőre kiírt formátumnak megfelelően írja az eredmeny.txt állományba. Az adat.txt fájlnak legfeljebb 100 sora lehet; soronként 4 számot és egy műveleti jelet tartalmaz, melyeket mindenhol egy szóköz választ el egymástól. Műveleti jelként csak összeadás és szorzás szerepel.

   adat.txt:
   ```
   24 32 8 3 +
   24 32 8 3 *
   ```
   $\frac{24}{32}+\frac{8}{3}$ $\frac{24}{32}\cdot\frac{8}{3}$

   eredmeny.txt:
   24/32 + 8/3 = 72/96 + 256/96 = 328/96 = 41/12
   24/32 * 8/3 = 192/96 = 2

A 3. feladatsor egy lehetséges megoldása:

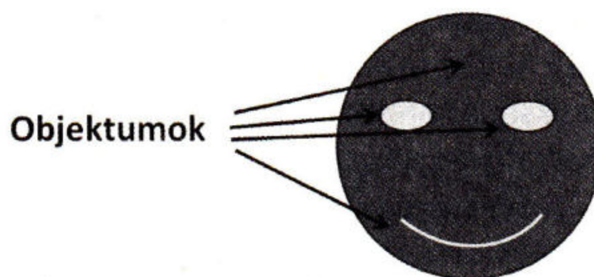
1.	-
2.	Megadott algoritmus kódolása
3.	Egyszerűsítő függvény készítése
4.	1. Szorzó függvény készítése 2. Bekért értékekre szorzó és egyszerűsítő függvény alkalmazása
5.	Megadott algoritmus kódolása
6.	1. Összegző függvény készítése 2. Bekért értékekre szorzó és egyszerűsítő függvény alkalmazása
7.	Egyszerűsítő, szorzó és összeadó függvényekről másolatot készíteni, és átírni fájlba írósra (Vagy konzolablak utasítások fájlba irányítása) Nincs összetett adatszerkezet Fájlbeolvasás fájlvége jelig (while) Sorozatszámítás függvénye a korábban átírt függvények elágazással a műveleti jel szerint Fájlba írás

VII. OBJEKTUMORIENTÁLT PROGRAMOZÁS

1. OBJEKTUM ÉS OSZTÁLY

Alapfogalmak

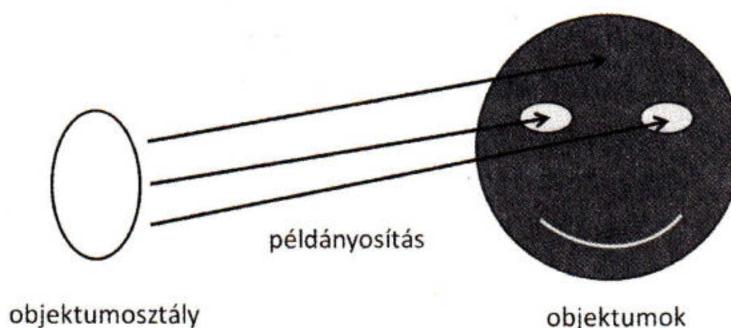
Az **objektum** az egész (pl. program, rajz) egy logikailag jól elkülöníthető része. A mellékelt rajz például összerakható az alábbi jól elkülönülő objektumokból: egy kék (sötétebb) kör, két sárga (világosabb) ellipszis, egy sárga ívdarab.



Az objektumokat minta (sablon) alapján hozzuk létre, amiket **objektumosztálynak** vagy röviden **osztálynak** is neveznek. Hasonlóan ahhoz, mint mikor egy szabásminta alapján készítjük el a ruhát akár több példányban is. Vagy bélyegzővel (sablon) alakítunk ki lenyomatokat papíron. Az előbbi képen látható kör és ellipszis objektumok mind származhatnak egyetlen ellipszis osztályból, hiszen a kör is egy ellipszis.

Azt a folyamatot, mikor az osztály alapján objektumot hozunk létre, szokás **példányosításnak** nevezni.

Az objektum létrehozásának és felhasználásának egy része erősen emlékeztethet minket a rekordokra. Ott is először létrehozunk egy rekordtípust (ez felel meg az osztálynak), majd a típus alapján konkrét rekordokat (ez felel meg az objektumnak).



Osztály létrehozása és példányosítás

Olvassuk át a Programozási tételek fejezet Feladatsorok témaköréből a Törtek című érettségi feladatsort. Ezt a feladatsort fogjuk elkészíteni objektumorientált változatban.

Keressünk egy logikailag jól elkülöníthető elemet a feladatban. Abból készíthetünk objektumot. Hamar rá fogunk jönni, hogy ez a *tört*. A feladat másról sem szól, mint törtekről.

Miután több is lesz belőle, létre lehet hozni egy osztályt (sablont) *Tortek* néven, amiből majd példányosítjuk az egyes *tort* objektumokat.

Az objektumorientált programozásban erőteljesen javasolt *szokás* az osztályt nagybetűvel kezdeni, és többes számban írni. Az objektumokat kisbetűvel és egyes számban írjuk.

A Pythonban az osztályokat a **class** kulcsszóval vezetjük be. Mögé írjuk az osztály nevét, majd a Pythonban megszokott kettőspontot (1. sor). A **pass** parancs szükséges ahhoz, hogy az osztály – egyelőre további beállítások nélkül – kipróbálható legyen.

A példányosítás úgy történik, mint a változók létrehozása: először megnevezzük az objektumot, majd egy egyenlőségjel után az osztályt, aminek alapján létrehozzuk (4. sor).

```
1 class Tortek:
2     pass
3
4 tort = Tortek()
```

2. AZ OBJEKTUM RÉSZEI

Tagváltozók

```
1 class Tortek:
2     pass
3
4 tort = Tortek()
5 tort.nevezo = 6
6 tort.szamlalo = 12
7 print(tort.szamlalo/tort.nevezo)
```

Az objektumoknak vannak jellemző adatai, amiket **tulajdonságoknak** nevezünk. A tulajdonságokat **tagváltozókban** tároljuk.

A bevezető mosolygó fej nagy körének adata például a mérete (pontosabban a sugara) és a színe. A tört két jellemzője a számlálója és a nevezője.

A tagváltozók számára a Python az első használatkor foglal le helyet a memóriában, így a változókhoz hasonlóan nem kell őket előzőleg deklarálni az osztályban.

A már példányosított objektumban a tulajdonságokra *objektumnév.tagváltozónév* formában hivatkozhatunk (5–7. sorok). Például: tort.nevezo = 6.

Ha most lefuttatjuk a programot, már kiír valami értelmeset. Ám az objektumot eddig csak úgy használtuk, mint egy rekordot.

Metódus, tageljárás

Az objektumokon végrehajtható műveleteket **metódusoknak** nevezzük, amik jellegüktől függően lehetnek **tageljárások**, illetve **tagfüggvények**.

A bevezető mosolygó arcos példában ilyen művelet lehetne a méret-változtatás, vagy a megjelenítés a képernyőn máshol (mozgatás).

A két legalapvetőbb dolog, amit meg kell oldani a tageljárásoknak, az adatmegjelenítés és az adatbevitel. Célszerű ezekkel a tageljárásokkal kezdeni, hogy utána könnyebb legyen a tesztelés.

Folytatva a tört feladatsort, készítsünk tageljárást, ami megjeleníti a törtet.

```
1 class Tortek:
2     def kiir(self):
3         print("%d/%d" % (self.szamlalo,\
4             self.nevezo),end=" ")
5
6 tort = Tortek()
7 tort.szamlalo = 180
8 tort.nevezo = 120
9 tort.kiir()
```

Az osztályban hasonlóan hozzuk létre a tageljárást, mint a strukturált programozásban az eljárást (2. sor). Különbség csak a zárójelek közötti **self**ben van[63], ami az éppen **aktuálisan működő példányt** (objektumot) jelenti. Ha a példányt *tort* néven hoztuk létre, akkor *self = tort,* ha *buff* néven, akkor *self = buff.*

Így amikor a tageljárás további soraiban (3–4. sor) felhasználjuk, akkor a self.szamlalo a tort.szamlalo-t jelenti, mivel a tort nevű példányt felhasználva hívjuk meg a tageljárást (9. sor).

A tageljárást meghívni a főprogramból az *objektumnév.tageljárásnév()* megadásával lehet. A zárójelek között fel kell tüntetni a tageljárás paramétereit (ha vannak).

Bővítsük a programot az adatbekérő eljárással. A neve legyen *beker*. Mivel semmi újdonságot nem tartalmaz, a megírását megpróbálhatjuk önállóan. Ellenőrizhetjük magunkat a mellékelt ábra segítségével (5–7. sor). Innentől kezdve nincs szükség rá, hogy a főprogramban közvetlenül a tagváltozóknak adjunk értéket. Használjuk helyette a beker() tageljárást.

```
 1 class Tortek:
 2     def kiir(self):
 3         print("%d/%d" % (self.szamlalo,\
 4             self.nevezo),end=" ")
 5     def beker(self):
 6         self.szamlalo = int(input("Kérem a számlálót: "))
 7         self.nevezo = int(input("Kérem a nevezőt: "))
 8
 9 tort = Tortek()
10 tort.beker()
11 tort.kiir()
```

63 A self helyett bármilyen más szót is alkalmazhatunk, ha következetesek vagyunk. A tageljárás neve mögötti zárójelben megadott szót kell használni az aktuális objektumra hivatkozásnál a tageljárás további részében. A nemzetközi szokás a self.

Egységbezárás

Az objektumorientált programozásnak is megvannak a maga *alapelvei,* amik sok-sok programozó tapasztalatai alapján álltak össze.

Az egyik ilyen alapelv az **egységbezárás**, ami már meg is valósult a példaprogramunkban, amikor az adatbevitelt a főprogramból áthelyeztük az objektumba, azaz a *beker* tagfüggvényt használjuk a tagváltozók közvetlen értékadása helyett.

Az elv a következőt mondja ki: **egységbezárt a program**, ha a tagváltozókat csak az objektumon belül érjük el közvetlenül. Az objektum tagváltozói és metódusai ily módon egy egységként viselkednek.

Tagfüggvények

Az osztályban ugyanúgy hozzuk létre a tagfüggvényt, mint a strukturált programozásban a függvényt. A különbséget ismét csak a **self** használata okozza, mint a tageljárásoknál (9. sor).

Hozzunk létre függvényt, ami visszaadja a tört számlálóját. Legyen a neve *szamlalotad.*

A tagfüggvény a **return** utasítás mögötti értéket adja vissza, ami nem más, mint az aktuális objektum (self) számláló tagváltozójának értéke (10. sor).

A 14. sorban láthatjuk, hogyan kell meghívni egy tagfüggvényt: *objektumnév.tagfüggvénynév()* alakban, amit aztán vagy egy másik utasítás dolgoz fel (a példában a print), vagy egy változónak adunk értéket vele.

A 9–10. sor mintájára írjuk meg önállóan azt a tagfüggvényt, ami visszaadja a tört nevezőjét. Legyen a neve *nevezotad.* (Mindkét függvényre szükségünk lesz, amikor a törtek között műveleteket fogunk végezni.)

```
 4                self.nevezo),end=" ")
 5        def beker(self):
 6            self.szamlalo = int(input("Kérem a számlálót: "))
 7            self.nevezo = int(input("Kérem a nevezőt: "))
 8
 9        def szamlalotad(self):
10            return self.szamlalo
11
12    tort = Tortek()
13    tort.beker()
14    print(tort.szamlalotad())
```

3. KÜLÖNLEGES METÓDUSOK ÉS VÁLTOZÓK

Konstruktor

A konstruktor az a különleges tageljárás, ami az objektum létrehozásakor automatikusan lefut.

A konstruktort elsősorban arra használjuk, hogy a **tagváltozóknak kezdőértéket adjunk**. A programozásban általában sem szeretjük az inicializálatlan változókat, mert azok később problémát okozhatnak (pl. 0-val való osztás).

Ugyanígy a konstruktorba helyezhetők azok a **metódusok, amik az objektum létrehozásakor lefuthatnak**.

Az objektumorientált programozás elveivel összhangban illik a konstruktort megírni. Javaslom, hogy a továbbiakban ezzel kezdjük az osztály kialakítását, és a kódban is sorrendben az első tageljárás legyen.

```
 1 class Tortek:
 2     def __init__(self):
 3         self.szamlalo = 0
 4         self.nevezo = 1

20 tort = Tortek()
21 tort.kiir()
```

A példánkban állítsuk be a számlálót 0-ra, a nevezőt 1-re[64] (2–4. sorok).

A Pythonban a konstruktornak kötelezően __init__ a neve. Két aláhúzás _ karakterrel kezdődik és végződik, nem tartalmaz szóközt (2. sor).

A teszteléshez nem kell meghívni a konstruktort, hiszen az objektum létrehozásakor automatikusan lefut. Amikor példányosítjuk a tort objektumot (20. sor), az objektum létrejön, a konstruktor lefut, beállítja a számlálót 0-ra (3. sor), a nevezőt 1-re (4. sor). Utána már csak kiíratjuk a törtet a *kiir* tageljárás meghívásával, és megjelenik a tört: *0/1*.

Statikus változók és metódusok

Általánosan elmondható, hogy statikusnak azokat az adattagokat, illetve metódusokat tekinthetjük, amiknek nincs közvetlenül közük az osztály példányaihoz.

64 Azért nem 0-ra a nevezőt, mert olyan törtnek nincs értelme, és a program a későbbiekben könnyen leállhatna hibaüzenettel.

Statikus változók

A statikus változó az osztály valamennyi objektuma számára összesen egyetlen példányban létezik.

Olyan, mint az osztály egy globális változója. Az objektumok megosztva használhatják. Bármelyik osztálybeli objektum hozzáférhet, és átírhatja. Lehetőséget biztosít az osztályon belül az objektumok közötti adatcserére. Elérhető akkor is, ha egyetlen példánya sincs az osztálynak.

A példa feladatsorunkban ilyen változót használhatnánk az objektumok tényleges számának vagy az eredmények összegének, minimumának, maximumának tárolására.

Osztályterület	
Példányterület	**Statikus terület**
Példány1 (példány1 változói és metódusai)	**Statikus változók és metódusok**
Példány2 (példány2 változói és metódusai)	
Példány3 (példány3 változói és metódusai)	
...	

Statikus metódusok

A statikus változókat statikus metódusokkal kezeljük. Statikus metódusból nem érhetünk el példányváltozókat, csak statikus változókat. Viszont példánymetódusból mind a statikus változók, mind az adott példány változói elérhetők. Ennek megfelelően a statikus metódusok akkor is lefutnak, ha nincs az osztálynak egyetlen példánya sem.

```
20      def lnko(a,b):
21          if a == b:
22              return a
23          if a < b:
24              return Tortek.lnko(a, b - a)
25          if a > b:
26              return Tortek.lnko(a - b, b)
27
28  print(Tortek.lnko(120,75))
```

A törttel kapcsolatos objektumunkban is találunk olyan metódust, aminek működése nem függ a tagváltozóktól. A legnagyobb közös osztót kiszámító tagfüggvény *(lnko)* lehet statikus. Ha statikusként írjuk meg, nemcsak a tört számlálójának és nevezőjének legnagyobb közös osztóját határozhatjuk meg az egyszerűsítéshez, hanem más számokét is.

A kódolásnál megfigyelhető, hogy a név megadása után a zárójelek között nem szerepelhet a self, hiszen nem példánymetódusról van szó, így nincs példány sem, amire hivatkozhatnánk[65].

A statikus metódus meghívása *osztálynév.metódusnév* alakban történik (28. sor). Vegyük észre, hogy a főprogramban nem hozunk létre objektumot (hiányzik a tort = Tortek() sor), a program mégis működik. Ez mutatja, hogy a statikus metódushoz nem szükséges példányosítani.

```
28     def egyszerusit(self):
29         if self.szamlalo % self.nevezo == 0:
30             print("= %d" % (self.szamlalo / self.nevezo))
31         else:
32             print("= %d/%d" % \
33             (self.szamlalo/Tortek.lnko(self.szamlalo,self.nevezo),
34             self.nevezo/Tortek.lnko(self.szamlalo,self.nevezo)))
35
36 tort = Tortek()
37 tort.beker()
38 tort.kiir()
39 tort.egyszerusit()
```

A legnagyobb közös osztó meghatározása után megírhatjuk az egyszerűsítést végző tageljárást: ha a tört egésszé alakítható, írjuk ki a képernyőre az egészet, egyébként jelenjen meg a tört a legegyszerűbb alakjában. Legyen a neve *egyszerusit*.

Példánymetódus lesz, mivel mindig az aktuális törtet egyszerűsítjük.

Az objektumunk már használható valamire: bekér egy törtet (37. sor), amit aztán meg is jelenít a képernyőn (38. sor), majd leegyszerűsítve kiírja a képernyőre (39. sor).

Feladatok

1. Írjunk a Tortek osztályba olyan metódust, ami eldönti, felírható-e a tört egészként. Ha igen, írja ki az egészet, ha nem, jelenítse meg a "Nem egész." üzenetet. A metódus neve legyen *egesz_e*.
2. Készítsünk a Tortek osztályba olyan metódust, ami feltölti a számlálót és a nevezőt is a paraméterként megkapott számokkal. A metódus neve legyen *megad*.
3. Írjuk meg úgy a főprogramot, hogy bekérjen egy törtet, eldöntse róla, egész-e, majd jelenítse meg a törtet és a leegyszerűsített változatot is a következő alakban: 16/12 = 4/3.

---

65 A legnagyobb közös osztó meghatározásához ugyanazt a rekurzív algoritmust használjuk itt, ami a Törtek feladatsorban szerepelt.

4. OBJEKTUMOSZTÁLY TERVEZÉSE

Az objektumorientált programokat általában előre megtervezik, és csak utána kódolják. Vannak olyan segédprogramok, amik a kódolás egy részét elvégzik[66].

A tervezéskor szokás az osztályokat téglalappal jelképezni (diagram), amit három részre osztanak: felül az osztály neve, középen a tagváltozók, alul a metódusok.

A mellékelt ábrán láthatjuk a *Tortek* osztályának diagramját

Tortek
- nevezo: int - szamlalo: int
+ __init__() : void + egesz_e() : void + lnko(a: int, b: int) : int + egyszerusit() : void + megjelenit() : void + beker() : void + szamlalotad() : int + megadt(sz: int, n: int) : void + nevezotad() : int

A – jelek azt jelentik, hogy csak az aktuális objektumból hozzáférhető, míg a + azt, hogy máshonnan is elérhető.

A visszatérési értékek típusai a metódus végén láthatók. A void jelentése: nincs visszatérési érték, tehát tageljárásról van szó.

Az aláhúzott tagváltozó vagy metódus statikus, míg az alá nem húzott példánymetódus.

Tulajdonképpen a tervezés nagy vonalakban ebből áll:

1. Felismerni a szükséges osztályokat.
2. Felismerni a szükséges tagváltozókat.
3. Felismerni a szükséges metódusokat.

Muveletek
- tort1: Tortek - tort2: Tortek - eredmeny: Tortek - muvelet: char
+ __init__() : void + kiir() : void + beker() : void + szoroz() : void + osszead() : void + lkkt(a: int, b: int) : int + valaszt() : void + megad(sz1: int, n1: int, sz2: int, n2: int, m: char) : void

Folytassuk a törtekkel kapcsolatos feladatokat. Most már, hogy vannak törtjeink, jó lenne közöttük műveleteket végrehajtani. Ehhez célszerű létrehozni egy *Muveletek* osztályt. Többes

66 A könyvben található ábrák az NClass nevű ingyenes programmal készültek, de a példa követéséhez elég egy Paintet megnyitni, netán egy darab papírt és tollat használni.

számban és nagybetűvel az elején, hogy szokjuk az objektumorientált programozás alapelveit. Rajzoljunk egy téglalapot, és a felső részébe írjuk bele az osztály nevét.

Tagváltozónak kelleni fog három Tortek típusú objektum: kettő, amivel a műveleteket végezzük *(tort1, tort2)*, és egy, amiben az egyszerűsítettlen eredményt fogjuk tárolni *(eredmeny)*. Kell még egy műveleti jel, ami karakterlánc lesz.

Válasszuk el vízszintes vonalakkal az osztálynevet a tagváltozóktól, majd a tagváltozókat a most következő metódusoktól.

Milyen metódusokra lesz szükség (a konstruktoron kívül)? Kétféle műveletet fogunk megvalósítani, az összeadást és a szorzást. Ezeknek mindenképpen lesz egy-egy tageljárása. Az objektumorientált programozás szokásaival összhangban igéket adunk a neveknek. Így lesz: *osszead* és *szoroz*. A szükséges változókat a *tort1*, *tort2*-ből veszik, így az eljárások paraméter nélküliek lesznek. Az eredményt az *eredmeny* objektumba töltik, így nem kell visszatérési érték sem.

Az összeadáshoz (közös nevező kialakításához) kelleni fog a legkisebb közös többszörös meghatározása. Legyen a tagfüggvény neve: *lkkt*. Függvény, mert a legkisebb közös többszörös lesz a visszatérési érték, és lesz két paraméter, ahol a két számot megadjuk. A függvény *statikus*, mivel nincs köze az aktuális példányhoz, bármilyen két szám legkisebb közös többszöröse kiszámítható vele.

Lesz egy tageljárásunk, ami eldönti, melyik műveletet kell elvégezni a kettő közül, és elindítja azokat. Legyen a neve: *valaszt*.

Az eredmények megjelenítését az *eredmeny* objektum tageljárásai elvégzik, de a műveletsor elején az eredeti művelet megjelenítéséhez szükség lesz egy paraméter nélküli tageljárásra (*kiir*).

Szükségünk lesz egy tageljárásra, ami feltölti konzolról adatokkal az objektumot. Legyen a neve: *beker*. Elsőre zavaró lehet, hogy ugyanaz a neve, mint a Tortek osztályban, de mivel a tageljárás neve elé mindig kiírjuk meghíváskor az objektum nevét is, nem fogjuk összekeverni. Sőt, a tervezés egységes lesz általa. Nem kell gondolkodnunk rajta, hogyan is hívtuk itt, és hogyan ott az adatbekérő metódust.

Kell még egy tageljárás, amivel feltöltjük a *Muveletek* típusú objektumokat adatokkal majd egy másik objektumból. Legyen a neve: *megad*. Öt paramétere lesz, a két tört számlálói *(sz1, sz2)* és nevezői *(n1, n2)*, valamint a köztük lévő műveleti jel *(muv)*.

A *Muveletek* osztály terve végül a mellékelt ábrán látható módon áll össze.

5. OBJEKTUMOSZTÁLY KÓDOLÁSA

A kódolás tervezése

A kódolást úgy kell megterveznünk, hogy a lehető leghamarabb tesztelhető legyen fejlesztési fázisban is a program.

Ezért a konstruktorral kezdünk, mert akkor lesznek alapértékek, amikkel a program a továbbiakban tesztelhető.

Az eredményt megjelenítő tageljárással folytatjuk *(kiir)*, hogy legyen mivel ellenőrizni az eredményeket.

Következhet az adatokat bekérő tageljárás, hogy ne csak az alapértelmezett adatokkal tudjunk tesztelni *(megad)*.

Utána jönnek a műveletet végző metódusok. Ha több is van, egymásra épülési, majd nehézségi sorrendben haladunk (szoroz, lkkt, osszead).

Osztály, tagváltozók és konstruktor

Az eddigi ismeretek elegendőek a Muveletek osztály létrehozásához, tesztcélú példányosításához.

```
45 class Muveletek:
46     def __init__(self):
47         self.tort1 = Tortek()
48         self.tort2 = Tortek()
49         self.eredmeny = Tortek()
50         self.muvjel = "*"
51
52 muvelet = Muveletek()
```

A konstruktorral kapcsolatban annyit megjegyeznék, hogy a *tort1, tort2, eredmeny* tagváltozókat itt nem kell kezdőértékkel ellátni, hiszen deklarálásukkor lefut a saját (Tortek-ben meghatározott) konstruktoruk. Egyedül a *muvelet* változónak kell kezdőértéket adni[67] (50. sor). Adjuk neki a szorzásjelet, mert azt a műveletet fogjuk előbb tesztelni.

A kód az eddig megírt törtekkel kapcsolatos kód folytatása, azaz a *Tortek* osztály már rendelkezésünkre áll.

Ha kiegészítettük a mellékelt kóddal a programunkat, ellenőrzésként futtassuk. Látszólag nem csinál semmit, hiszen nincs még megjelenítő metódusunk. Ha nincs hibaüzenet, akkor eddig valószínűleg jól dolgoztunk.

---

67 Szakkifejezéssel: inicializálni = kezdőértéket adni.

Megjelenítő tageljárás

A műveletek elejének megjelenítéséhez alapvetően a *Tortek* osztály megjelenítő és egyszerűsítő tageljárásait használjuk fel (53. sor és 56. sor). Közöttük a még hiányzó műveleti és egyenlőségjeleket, szóközöket ebben a tageljárásban kell megjeleníteni (54–55. és 57. sorok).

```
52      def kiir(self):
53          self.tort1.kiir()
54          print("%s " % \
55            (self.muvjel), end = '')
56          self.tort2.kiir()
57          print("= ", end = '')
58
59  muvelet = Muveletek()
60  muvelet.kiir()
```

A tageljárás megírása után tesztelés következik: meghívjuk a főprogramból (60. sor).

Futtassuk a programot. Ezt kell kapnunk a parancsértelmező ablakban: 0/1 * 0/1 =. A konstruktorban megadott értékeket látjuk viszont.

Adatot a konzolablakból feltöltő tageljárás

A teljesebb teszteléshez szükségünk lesz többféle adatra, tehát célszerű ebben a lépésben az adatok bevitelét megírni. A törtek számlálóját és nevezőjét bekérhetjük a *Tortek* osztály *beker* tageljárásának kétszeri alkalmazásával (61. és 63. sor). A köztes sorok segítségével különbözteti meg majd a felhasználó a két törtet egymástól (60. és 62. sor).

```
59      def beker(self):
60          print("1. tört")
61          self.tort1.beker()
62          print("2. tört")
63          self.tort2.beker()
64
65  muvelet = Muveletek()
66  muvelet.beker()
67  muvelet.kiir()
```

A teszteléshez meghívjuk először a most megírt adatbekérő tageljárást, majd a kiírót, hogy lássuk, mi történt.

Ha mindent a leírás szerint tettünk, futtatáskor megjelenik a két tört, közötte a szorzásjellel, például: 1/2 * 3/4 =.

Szorzást végző tageljárás

A *szoroz* tageljárást fogjuk most elkészíteni.

A *tort1* és *tort2* objektumokban vannak a kiindulási adatok, amikhez a *nevezotad, szamlalotad* tagfüggvényekkel tudunk hozzáférni (66–69. sorok). Ezért kellett őket megírni a Tortek osztályban, noha ott semmire nem használtuk őket.

```
65      def szoroz(self):
66          self.sz = self.tort1.szamlalotad() \
67                    * self.tort2.szamlalotad()
68          self.n = self.tort1.nevezotad()\
69                   * self.tort2.nevezotad()
70          self.eredmeny.megad(self.sz,self.n)
71          self.kiir()
72          self.eredmeny.kiir()
73          self.eredmeny.egyszerusit()
74
75  muvelet = Muveletek()
76  muvelet.beker()
77  muvelet.szoroz()
```

A jobb áttekinthetőség érdekében létrehoztunk két lokális változót *(sz,n)*, amibe kiszámoljuk az eredmény számlálóját és nevezőjét: számlálót a számlálóval (66–67. sorok), nevezőt a nevezővel (68–69. sorok) szorozzuk.

Az eredményeket az *eredmeny* objektumba helyezzük, a *megad* tageljárással (70. sor).

Már csak a műveletsor megjelenítése van hátra. Ehhez először kiírjuk a kijelölt műveletet, azaz az eredetileg megadott törteket (71. sor), majd az eredmény törtet (72. sor), végül az eredmény egyszerűsített változatát (73. sor).

Teszteljük a megírt tageljárást (77. sor). Ha jól dolgoztunk, ilyesmit kell látnunk: 22/6 * 7/12 = 154/72 = 77/36.

Legkisebb közös többszöröst kiszámító tagfüggvény

Két egész szám legkisebb közös többszöröse könnyen kiszámítható a legnagyobb közös osztó ismeretében: a két szám szorzata osztva a legnagyobb közös osztóval (76. sor).

```
75      def lkkt(a,b):
76          return int(a * b / Tortek.lnko(a,b))
77
78  print(Muveletek.lkkt(12,15))
```

A legnagyobb közös osztót kiszámító algoritmust már beírtuk a Tortek osztályba lnko néven. Mivel a függvény statikus, meg kell adni az osztály nevét is a függvény előtt: Tortek.lnko(a,b).

Mivel az *lkkt* függvény is statikus, a teszteléshez nem szükséges a *Muveletek* osztályból példányt létrehozni. (Nincs a főprogramban a muvelet = Muveletek() sor.)

A teszteléshez **print** utasításból hívjuk meg, mivel a függvény által visszaadott értéket valahogyan meg kell jeleníteni a képernyőn. A függvény neve előtt az osztály neve is szerepel, mivel a statikus metódusokat így kell meghívni (78. sor).

Összeadást végző tageljárás

```
78      def osszead(self):
79          self.kn = Muveletek.lkkt(self.tort1.nevezotad(),self.tort2.nevezotad())
80          self.sz1 = self.tort1.szamlalotad() * self.kn / self.tort1.nevezotad()
81          self.sz2 = self.tort2.szamlalotad() * self.kn / self.tort2.nevezotad()
82          self.eredmeny.megad(self.sz1 + self.sz2, self.kn)
83          self.kiir()
84          self.tort1.megad(self.sz1,self.kn)
85          self.tort2.megad(self.sz2,self.kn)
86          self.kiir()
87          self.eredmeny.kiir()
88          self.eredmeny.egyszerusit()
89
90  muvelet = Muveletek()
91  muvelet.beker()
92  muvelet.osszead()
```

A tageljárás elején meghatározzuk a közös nevezőt, ami a két tört nevezőjének legkisebb közös többszöröse (79. sor). Most használjuk fel az előző pontban megírt *lkkt* függvényt. Mivel statikus, a függvény neve előtt az osztály neve szerepel. A függvény paramétere a két tört nevezője, amit a *nevezotad* tagfüggvénnyel érünk el a tört objektumokból. A közös nevezőt átmenetileg egy *kn* változóban helyezzük el.

Az *sz1* és *sz2* lokális változókban kiszámítjuk a közös nevezőre hozott törtek számlálóját (80–81. sor). A számlálókat és nevezőket a tört objektumokból a *szamlalotad* és *nevezotad* függvényekkel érjük el.

Az eredmény törtbe feltöltjük az összeadás eredményét: a számlálóba a két számláló *(sz1, sz2)* összegét, a nevezőbe a közös nevezőt *(kn)*. A feltöltéshez az *eredmeny* tört objektum *megad* metódusát használjuk (82. sor).

Ezzel a számolással készen vagyunk, csak a műveletsor megjelenítése van hátra.

Először megjelenítjük az eredeti törtekkel a kijelölt műveletet a *Muveletek* osztály *kiir* tageljárás segítségével (83. sor). Majd kihasználjuk azt, hogy a következő részlet ugyanúgy néz ki, mint amit megjelenítettünk (tört1, műveleti jel, tört2, egyenlőségjel). Feltöltjük a közös nevezőre hozott törteket a *tort1, tort2* objektumokba (84–85. sor), majd ismét alkalmazzuk a *Muveletek* osztály *kiir* tageljárását (86. sor). Ezután megjelenítjük az eredmény törtet, felhasználva a *Tortek* osztály *kiir* metódusát (87. sor), majd a leegyszerűsített eredményt (88. sor).

Teszteljük a megírt tageljárást (92. sor). Előzőleg állítsuk át a *Muveletek* konstruktorában a műveleti jelet +-ra (50. sor). Ha jól dolgoztunk, ilyesmit kell látnunk: 22/6 + 22/4 = 44/12 + 66/12 = 110/12 = 55/6.

Műveletválasztó tageljárás

Szükségünk lesz egy tageljárásra, ami eldönti a műveleti jel alapján, melyik műveletet kell elvégezni.

Az elágazás műveleti *(muvjel)* jel alapján történik (91. sor). Ha szorzásjel, akkor a *szoroz* tageljárást hívjuk meg (92. sor). Különben összeadni kell, tehát az *osszead* tageljárást hívjuk meg.

A teszteléshez a főprogramból a *valaszt* tageljárást kell meghívni (98. sor).

```
90      def valaszt(self):
91          if self.muvjel == "*":
92              self.szoroz()
93          else:
94              self.osszead()
95
96  muvelet = Muveletek()
97  muvelet.beker()
98  muvelet.valaszt()
```

Feladatok

1. Az adatok gyakran nem a billentyűzetről érkeznek, hanem egy másik objektumból. Az adatok változókba töltéséhez szükségünk lesz egy tageljárásra. A tageljárás paraméterei a két tört számlálói, nevezői, valamint a műveleti jel, neve legyen *megad*. Úgy képzeljük el, hogy tetszőleges helyről ezzel a függvénnyel fogunk tudni feltölteni egy Muvelet típusú objektumot. A törtek változókba töltéséhez használjuk a *Tortek* osztályhoz megírt *megad* tageljárást.
2. Készítsük fel a Muveletek osztályt arra, hogy osztást is el tudjon végezni két tört között. A megjelenítendő sor a következő:
   22/6 : 22/4 = 22/6 * 4/22 = 88/132 = 2/3
3. Készítsük fel a Muveletek osztályt arra, hogy kivonást is el tudjon végezni két tört között. Elég, ha pozitív eredményekre működik. A megjelenítendő sor a következő:
   22/4 - 22/6 = 66/12 - 44/12 = 22/12 = 11/6
4. Készítsünk egy *muveletetmegad* nevű tageljárást, ami a paraméterében megadott műveleti jellel tölti fel az aktuális *Muveletek* típusú objektumot.
5. Írjuk meg úgy a főprogramot, hogy bekérjen egy törtet, és jelenítse meg a szorzatukat és az összegüket is.

6. OBJEKTUMLISTA ÉS A CONTROL OBJEKTUM

Objektumlista

Sok műveletünk lesz, ezért érdemes lenne egy listát létrehozni a Muveletek osztályból származó objektumoknak. Ehhez egy kicsit át kell alakítanunk a kódot.

Először is a létrehozás sorában a zárójelek közé írjuk be az **object** szót (45. sor).

Majd a konstruktor zárójelében a self után be kell írni a number szót, amiben az objektum listabeli sorszáma fog tárolódni (46. sor).

Végül ki kell egészíteni a konstruktor kódját a self.number = number értékadással (47. sor)[68].

A teszteléshez a főprogramban létrehozunk egy *muv* nevű listát (135. sor), majd egy számláló ciklus segítségével feltöltjük Muveletek típusú objektumokkal, amiket a sorszámuk különböztet majd meg egymástól (136–137. sorok).

```
45 class Muveletek(object):
46     def __init__(self, number):
47         self.number = number
48         self.tort1 = Tortek()
49         self.tort2 = Tortek()
50         self.eredmeny = Tortek()
51         self.muvjel = "*"
```

```
135 muv = []
136 for i in range(100):
137     muv.append(Muveletek(i))
```

A control objektum szerepe

Minden objektumorientált programnak van egy központi része, ami a többi objektumot irányítja. Ez a központi rész is egy objektum, ami szokásos esetben a **control** nevet viseli. A név utal a feladatra, de természetesen választhatunk másikat.

Tekinthetnénk magát a főprogramot a control objektumnak, de szokás inkább külön objektumot létrehozni neki.

68 A mi programunk most működne e lépés nélkül is, de általában jobb ezt így megírni, mert így hivatkozni tudunk a kódunkban az objektum sorszámára, ha ez szükséges valamiért.

Tervezés

Tekintsük át, mit kellene tudni a programunknak:

1. Bekérni egy törtet, és kiírni egész alakban, ha lehetséges. Ha nem, megjeleníteni a "Nem egész." feliratot.
2. Az előbb bekért törtet egésszé alakítani, ha lehet. Ha nem, akkor leegyszerűsítve megjeleníteni a képernyőn.
3. Bekérni két törtet, majd kiszámolni és megjeleníteni a szorzatukat.
4. Az előbb bekért két tört összegét kiszámolni és megjeleníteni.
5. Az *adat.txt* fájlban található tört műveleteket elvégezni, és kiírni az *eredmeny.txt* fájlba.

Control
- tort: Tortek - muvelet: Muveletek - muv: Muveletek
+ __init__() : void + fajlokatkezel() : void

Ennek tükrében már meg tudjuk tervezni a Control osztályunkat: Az első két feladathoz szükség lesz egy *Tortek* típusú objektumra. Legyen a neve *tort*.

A 3. és 4. feladatokhoz egy *Muveletek* típusú objektumra. Legyen a neve: *muvelet*.

Az 5. feladathoz egy *Muveletek* típusú objektumlistára. Legyen a neve: *muv*.

Az 5. feladat fájlkezelési részének kivételével minden szükséges metódus készen van a *Tortek* és *Muveletek* osztályokban. Az 5. feladat megoldásához készítsünk egy külön tageljárást. Legyen a neve: *fajlokatkezel*.

A metódusok hívását és a fájl beolvasását a konstruktorban fogjuk elvégezni, így a *Control* példányosításakor automatikusan minden lefut majd.

Kódolás

```
136 class Control():
137     def __init__(self):
138         tort = Tortek()
139         muvelet = Muveletek(1)
140         tort.beker()
141         tort.egesz_e()
142         tort.kiir()
143         tort.egyszerusit()
144         muvelet.beker()
145         muvelet.szoroz()
146         muvelet.muveletetmegad('+')
147         muvelet.osszead()
148         self.fajlokatkezel()
149
150     def fajlokatkezel(self):
151         pass
152
153 control = Control()
```

```
151     def fajlokatkezel(self):
152         self.muv = []
153         regi_kimenet=sys.stdout
154         be = open("adat.txt","r")
155         ki = open("eredmeny.txt","w")
156         sys.stdout = ki
157         line = be.readline()
158         i = 0
159         while line != "":
160             bontas = line.split()
161             self.muv.append(Muveletek(i))
162             self.muv[i].megad(int(bontas[0]), \
163                 int(bontas[1]), int(bontas[2]), \
164                 int(bontas[3]),bontas[4].strip())
165             self.muv[i].valaszt()
166             line = be.readline()
167             i+=1
168         be.close()
169         sys.stdout=regi_kimenet
170         ki.close()
```

Készítsük el a terv alapján a Control osztályt. Adjuk meg tagobjektumokat (138–139. sorok), és a főprogramban példányosítsuk, azaz készítsünk belőle egy objektumot (a kódban a neve *control*) (153. sor).

Készítsük el a konstruktort úgy, hogy végrehajtsa az első négy feladatot. Bekérjük a törtet (140. sor), eldöntjük, egész-e (141. sor), majd kiírjuk a törtet (142. sor), végül az egyszerűsített alakját (143. sor).

Bekérjük a műveletek elvégzéséhez a törteket (144. sor). Megjelenítjük a szorzatukat (145. sor), felhasználva, hogy a *Muveletek* konstruktorában az alapértelmezett műveleti jel a szorzás. Megváltoztatjuk a műveleti jelet összeadásra (146. sor), majd megjelenítjük az összegüket (147. sor).

Végül meghívjuk az 5. feladatot megoldó, egyelőre üres (150–151. sor) tageljárást (148. sor).

Írjuk meg a fájlkezelős tageljárást. A könnyebb fejlesztés érdekében érdemes először a képernyőre íratni a műveleteket, és utólag a fájlba irányítást hozzátenni. A minta már a kész változatot mutatja.

Ahhoz, hogy a fájlba irányításhoz meglegyen a szükséges utasításkészlet, szúrjuk be a program első soraként: import sys.

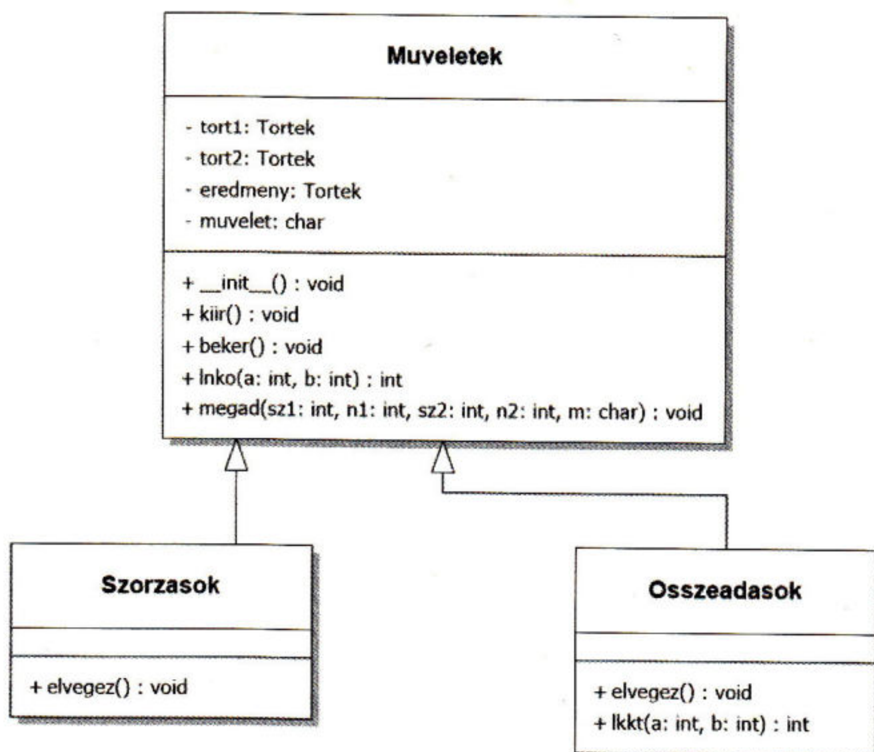
A tageljárás elején létrehozzuk a *muv* nevű listát (152. sor), amibe a ciklus minden egyes lefutásakor, azaz minden új sor beolvasásakor egy-egy üres, *Muveletek* típusú objektumot helyezünk el (161. sor), amit aztán feltöltünk az adatokkal (162–164. sor), majd elvégezzük a műveletet, és a műveletsort az elvárt alakban kiíratjuk (165. sor).

7. AZ OBJEKTUMORIENTÁLT PROGRAMOZÁS TOVÁBBI FOGALMAI

Öröklődés, hierarchia, többalakúság

Térjünk vissza a programunk olyan állapotára, amikor még csak a Tortek és Muveletek osztály létezett, nem volt kivonás és osztás, sem objektumlista[69].

Felépíthettük volna az osztályainkat más módon is. Például készíthetnénk egy külön osztályt az összeadásoknak. Ez célszerű lenne azért, mert egy szorzás példányosításakor nem jönne vele létre egy sor felhasználatlan metódus (például: lkkt, osszead). Jó lenne azért is, mert minél több objektumra bontjuk szét a programot, annál jobban áttekintető lesz, és jól szétbontható csoportmunkában.



Persze, akkor már érdemes lenne a szorzásoknak is saját osztályt készíteni.

Ugyanakkor vannak mindkettőben felhasználni kívánt metódusok, amiket felesleges lenne kétszer kódolni.

A megoldás a következő lesz:

1. Létrehozunk egy Muveletek osztályt, amiben a közös metódusok szerepelnek.
2. A Muveletek osztályból levezetjük az Osszeadasok és Szorzasok osztályokat, amikből elérhetők lesznek a Muveletek osztály tageljárásai, és kiegészíthetők sajátokkal.

69 A feladatVII_07_0.py nevű fájl a mellékletben.

Mikor egy osztály alapján új osztályt hozunk létre, előbbit **szülőosztálynak** vagy ősosztálynak, utóbbit **gyermekosztálynak** vagy **leszármazott osztálynak** nevezzük. A példában az Osszeadasok a Muveletek gyermekosztálya, tehát a Muveletek az Osszeadasok ősosztálya.

A gyermekosztály örökli a szülő osztály minden tagváltozóját és metódusát. Ezenfelül új tagváltozókat és metódusokat is deklarálhatunk bennük. Az öröklődés az objektumorientált programozás egy kulcsfogalma. Segítségével a *kódot újrahasznosítjuk,* hiszen nem kell többször megírni.

Egy osztályból újabb osztályok, azokból újabb osztályok származhatnak. Az osztályok öröklődés általi alá- illetve fölérendeltségi szerkezetét **hierarchiának** nevezzük.

A hierarchiában egy metódus neve lehet közös, miközben az általa végzett tevékenység osztályonként eltérő. Ezt nevezzük **többalakúságnak**, idegen szóval **polimorfizmusnak**. A példánkban az *elvegez* ilyen metódus, hiszen az összeadásnál több dolgot ír ki, mint a szorzásnál.

Öröklődés kódolása

Ha öröklést szeretnénk létrehozni, az újonnan deklarált gyermekosztály neve mögötti zárójelek közé kell beírni az ősosztályának nevét (83. és 99. sor).

Miután létrehoztuk a gyerekosztályokat a minta szerint, másoljuk át a *Muveletek* osztályból az *lkkt* és *osszead* tageljárásokat az *Osszeadasokba,* majd a *szoroz* tageljárást a *Szorzasokba*. Nevezzük át az *osszead* és *szoroz* tageljárásokat. Legyen mindkettőnek a neve *elvegez,* hogy lássuk, erre is van lehetőség.

Végül a főprogramból teszteljük az objektumokat (110–115. sor).

A 111. és 114. sorokban úgy hívjuk meg az objektumot, hogy nincs is *megad* nevű tageljárása. Ez azért lehetséges, mert örökli a *Muveletek* osztálytól.

```
 83 class Osszeadasok(Muveletek):
 84     def lkkt(a,b):
 85         return int(a * b / Tortek.lnko(a,b))
 86
 87     def elvegez(self):

 99 class Szorzasok(Muveletek):
100     def elvegez(self):
101         self.sz = self.tort1.szamlalotad() \
102                   * self.tort2.szamlalotad()
110 osszeadas = Osszeadasok()
111 osszeadas.megad(22, 4, 22, 6, '+')
112 osszeadas.elvegez()
113 szorzas = Szorzasok()
114 szorzas.megad(22, 4, 22, 6, '*')
115 szorzas.elvegez()
```

Virtuális metódus

Bonyolultabb a helyzet, ha

1. olyan metódust *(kiir)* hívunk meg a leszármazott osztály *(Emlosok)* objektumából *(elefant)*, ami csak az ősosztályban *(Allatok)* létezik *(jellemez)*,
2. és az meghív egy olyan metódust, amit mindkét osztályban különbözőre írtunk, de a nevük megegyezik *(labak)*.

Melyik osztály metódusát hívja meg? 3. vagy 4. nyíl?

A legtöbb programnyelvben, alapesetben a gyermekosztályból meghívott, csak a szülő osztályban létező metódus a vele azonos osztályban lévő metódust hívja meg, azaz a 4. nyíl szerint. Ha a 3. nyíl irányában kellene továbbmenni, akkor be kell avatkozni, és **virtuális metódust** kell használni.

A virtuális metódust az azonos nevű metódusok (a példában: *labak*) **közül a szülő objektumra** *(Allatok)* **alkalmazva elérhető, hogy a gyermekosztályból** *(Emlosok)* **meghívott, csak a szülő osztályban létező metódus** *(jellemez)* **a gyermekosztályban található metódust** *(labak)* **hívja meg.**

A Pythonban minden metódus virtuális.

```
 7 class Emlosok(Allatok):
 8     def labak(self):
 9         print("4")
10     def kiir(self):
11         self.jellemez()
```

```
1 class Allatok():
2     def jellemez(self):
3         self.labak()
4     def labak(self):
5         print("Nem tudni.")
```

```
13 allat = Allatok()
14 elefant = Emlosok()
15 elefant.kiir()
```

1. 2. 3. 4.

VIII. GRAFIKUS ALKALMAZÁS KÉSZÍTÉSE

1. EGYSZERŰ GRAFIKUS ALKALMAZÁS LÉTREHOZÁSA

Tkinter

Sokféle ablakkezelő rendszer létezik. Linuxon eleve többféle is megszokott. A Tkinter egy ilyen ablakkezelő rendszer függvénykönyvtárát nyújtja a fejlesztőknek. Természetesen programozható a Pythonból a Windows saját ablakkezelő rendszere is, de a Tkinternek vannak előnyei.

Egyrészt platformfüggetlen, éppúgy vannak ingyenes verziói Windowsra, mint Linuxra vagy Macre. Másrészt a Pythonnal együtt települ, így nem szükséges külön telepítés, nincs összeférhetetlenség. Harmadrészt nagyon jól dokumentált a Pythonnal történő együttműködése. Nem véletlenül ez a leggyakrabban használt ablakkezelő rendszer a Pythonhoz.

Kétségtelen hátránya, hogy kevesebb grafikus elemből áll, és azok sem annyira formatervezettek, mint a Windows ablakkezelője, valamint egyelőre nincs hozzá olyan alkalmazás, amiben igazán összeforrna az ablak grafikus szerkeszthetősége a kódolással.

A Tkinter alkalmazások létrehozása önmagában egy könyvre való témakör lenne. Itt csak a legfontosabb alapokat szedtem össze, amire építeni lehet az internetről összeszedhető részleteket.

Keretprogram

Minden **Tkinter** grafikus felületű programban szerepelni fog a függvénykönyvtárainak betöltése (1. sor), egy *Tk* típusú osztály példányosítása (3. sor), ami az ablakot adja, végül az eseménykezelés elindítása (5. sor). Utóbbi nélkül a programunk nem lenne érzékeny az egérkattintásokra, billentyű megnyomására, stb.

```
1 from tkinter import *
2
3 ablak = Tk()
4
5 ablak.mainloop()
```

Az ablak létrehozása és az eseménykezelő indítása közé fogjuk beszúrni azokat az utasításokat, amik a grafikus felület főprogram részét alkotják.

Futtassuk a programunkat. Meg fog jelenni egy kis, üres ablak.

Az ablak létrehozása után megadhatjuk a nevét, ami a címsorban fog megjelenni. Erre a célra szolgál a **title** utasítás a következő formában: ablak.title("Első program").

Megadható a legkisebb méret is, amire az ablak összehúzható a **minsize** utasítással. Például az ablak.minsize(width = 600, height = 400) utasítás legalább 600 képpont szélességű, 400 képpont magasságú ablakot enged használni.

Vezérlők

Vezérlőknek (gyermekablakoknak) nevezik az ablakban található objektumokat.

Vezérlő például a parancsgomb, a szövegmező, a címke, a gördítősáv, stb.

A vezérlőket a Tkinterben már definiált osztályukból (sablonjukból) hozzuk létre példányosítással, az ablakhoz hasonlóan. A vezérlő kötelező paramétere, hogy mely ablakhoz tartozik, azaz melyiknek a gyermekablaka. A többi paraméter között vannak általánosak, de olyanok is, amik csak az adott vezérlőfajtára jellemzők.

Egy vezérlőt két utasítással lehet életre kelteni: először létre kell hozni egy példányt belőle (5. sor), majd egy arra alkalmas paranccsal (metódussal) megjeleníteni (6. sor).

```
3 ablak = Tk()
4
5 cmdButton1 = Button(ablak,text="Új szó")
6 cmdButton1.pack()
7
8 ablak.mainloop()
```

Az objektum nevét szabadon megválaszthatjuk (5. sor). A parancsgomboknak szokás *cmdButton* nevet adni. *Button* a Tkinterben már létező parancsgomb osztály neve, így ezen nem módosíthatunk. Így az 5. sor tulajdonképpen ezt jelenti: legyen az ablakban a parancsgomb (Button) egy példánya (cmdButton1).

Az első paraméter a szülőablak neve *(ablak)*, amelyikre rakni óhajtjuk. A *text* paraméter után a vezérlőn megjelenő feliratot tudjuk beállítani.

A legegyszerűbb megjelenítési módot a **pack** metódus adja.

Futtassuk az eddigi programunkat. Az ablakon megjelenik egy parancsgomb Új szó felirattal.

Készítsük el egy szókitaláló játék (szerencsekerék, akasztófa, stb.) grafikus felületét. Kelleni fog hozzá három parancsgomb, két szövegmező, és két felirat, amik megmutatják, mi van a szövegmezőkben.

A szövegmezők osztálya az *Entry* (10–11. sorok). A szövegmező példányok szokásos, de nem kötelező elnevezése *textBox*.

A feliratok osztálya a Label (8–9. sorok). A felirat példányok szokásos, de nem kötelező elnevezése *label*.

A második mellékelt példa még a megjelenítésben rejt egy kis újdonságot:

A pack metódus a kiadott parancsok sorrendjében egymás alá helyezi a vezérlőket az ablakban, ha másképpen nem rendelkezünk.

Futtassuk ismét a programot. Megjelennek az ablakban a vezérlők a várt sorrendben.

```
 5 cmdButton1 = Button(ablak,text="Új szó")
 6 cmdButton2 = Button(ablak,text="Következő betű")
 7 cmdButton3 = Button(ablak,text="Kilépés")
 8 label1 = Label(ablak,text="Tippelt betű: ")
 9 label2 = Label(ablak,text="Kitalálandó szó: ")
10 textBox1 = Entry(ablak)
11 textBox2 = Entry(ablak)
12
13 cmdButton1.pack()
14 label1.pack()
15 textBox1.pack()
16 cmdButton2.pack()
17 label2.pack()
18 textBox2.pack()
19 cmdButton3.pack()
```

Sokat dob a látványon, ha picivel nagyobb betűméretet (11) és olvashatóbb betűtípust (Arial) alkalmazunk, valamint minden parancsgombot egyforma szélességűre definiálunk.

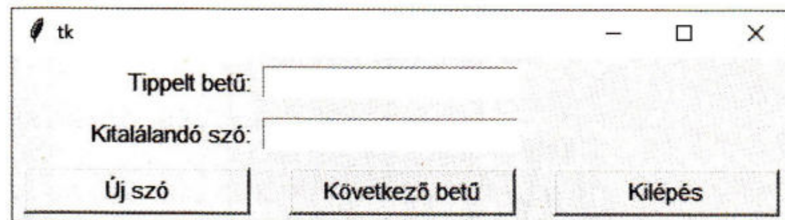
A parancsgombok szélességét a létrehozásukkor a *width* paraméterrel lehet meghatározni (6. sor).

Az ablak alapértelmezett betűtípusát és méretét még a *vezérlők létrehozása előtt* állítsuk be (5. sor).

```
3 ablak = Tk()
4
5 ablak.option_add("*font","Arial 11")
6 cmdButton1 = Button(ablak,text="Új szó", width = 15)
```

Rács

Általában egy ablakban nem csak egymás alá rakjuk a vezérlőket. Folytatva a szókitaláló játék példáját, rendezzük el a vezérlőket a képen látható módon.



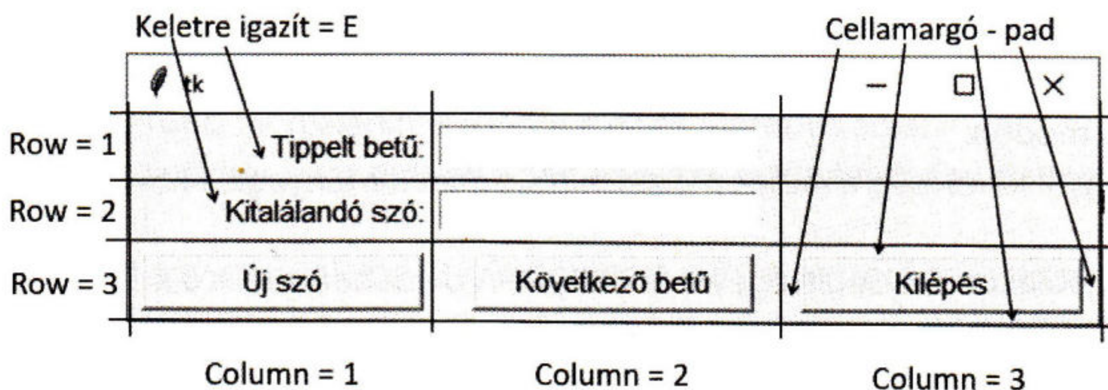
Az egyik lehetőség a rácsos *megjelenítés,* aminek a lényege a következő:

1. Az ablakot rács segítségével felosztjuk. A példában a rács 3 x 3 cellára osztja az ablakot.
2. Minden cellába legfeljebb egyetlen vezérlőt helyezhetünk el. Minden **grid** parancsban beállítjuk, hogy az adott vezérlő melyik sorba (row), illetve oszlopba (column) kerül. A sorok és oszlopok számozása 1-től indul (14–20. sor).
3. Alapértelmezés szerint a sor magassága akkora lesz, hogy a legmagasabb vezérlő is kiférjen. Ugyanez a helyzet az oszlopok szélességével is. A példában látható, hogy az első oszlop legszélesebb vezérlője a parancsgomb.

4. A cellán belül alapértelmezés szerint a vezérlők **középre igazodnak**. Az igazítás megváltoztatható a *sticky* paraméter segítségével, az égtájak angol neveinek első betűivel. A példában a címkéket igyekszünk közelebb vinni a hozzá tartozó szövegmezőhöz, ezért a címkéket jobbra, tehát keletre (E – east) (16–17. sor), a szövegmezőket balra, tehát nyugatra (W – west) igazítjuk (18–19. sor).
5. A cellán belül, a rácsvonal és vezérlő közötti távolságot **cellamargónak** nevezzük. Külön beállítható a vízszintes és függőleges cellamargó. Az alapértelmezés megváltoztatása nélkül a vezérlők gyakran túl szorosan helyezkednek el egymáshoz képest. Szükség esetén beállítható minden cella margója akár egyesével is, azonban általában sokkal egységesebb az ablak, ha egyformára állítjuk őket. A 22. sorban az i. oszlopban vízszintes cellamargóit állítjuk be 15 képpontra, a 23. sorban az i. sor függőleges cellamargóit állítjuk be 10 képpontra. A 21. sor ciklusa segítségével minden soron és oszlopon elvégezzük a beállítást.

Mivel megjelenítésről van szó, csak a **pack** utasításokat le kell cserélni **grid**ekre. A vezérlők létrehozásához nem kell hozzányúlni.

```
14 label1.grid(row = 1, column = 1, sticky = E)
15 label2.grid(row = 2, column = 1, sticky = E)
16 textBox1.grid(row = 1, column = 2, sticky = W)
17 textBox2.grid(row = 2, column = 2, sticky = W)
18 cmdButton1.grid(row = 3, column = 1)
19 cmdButton2.grid(row = 3, column = 2)
20 cmdButton3.grid(row = 3, column = 3)
21 for i in range(4):
22     ablak.columnconfigure(i,pad = 15)
23     ablak.rowconfigure(i,pad = 10)
```



Események

Az **esemény** az alkalmazás működését irányító felhasználói beavatkozás. Esemény például, ha egy parancsgombra kattintunk az egérrel, listából egy elemet kiválasztunk, megnyomunk egy billentyűt, stb.

Az eseményt azért váltjuk ki, hogy valami változás történjen. Hogy konkrétan mi, arról **utasítások** beírásával gondoskodhatunk.

A parancsgomb létrehozásakor megadható egy **command** paraméter, aminek az egyenlőségjele mögé beírhatjuk annak az eljárásnak a nevét, amit végre akarunk hajtani, ha rákattintanak a gombra. Például a 14. sorban a command mögé az van írva, hogy ujszo, tehát ha erre a gombra kattintunk majd, a 3. sorban található ujszo eljárás fog végrehajtódni. Jelenleg az eljárásokat még nem írtuk meg. A **pass** utasítás azért van ott, hogy ki tudjuk próbálni a programot így is.

Az ablakot bezárhatjuk, megszüntethetjük az **ablaknév.destroy** paranccsal. Ezt helyeztük el a "Kilépés" feliratú gomb **command** paraméterében (18. sor).

Ha most lefuttatjuk a programot, annyi különbség lesz az előző változathoz képest, hogy a "Kilépés" gombra kattintva bezáródik az ablak.

```
 3 def ujszo():
 4     pass
 5
 6 def betutkiertekel():
 7     pass
13 cmdButton1 = Button(ablak,text="Új szó",\
14                     width = 15, command=ujszo)
15 cmdButton2 = Button(ablak,text="Következő betű",\
16                     width = 15, command=betutkiertekel)
17 cmdButton3 = Button(ablak,text="Kilépés",\
18                     width = 15, command=ablak.destroy)
```

Szövegmezők kezelése

Építsük tovább a szókitaláló játékunkat. Készítsünk eljárást *(ujszo),* ami kiválaszt véletlenszerűen egy szót egy listából, és megjelenít helyette annyi X-et a 2. szövegmezőben, amilyen hosszú a szó[70].

```
 2 from random import randrange
 3
 4 def ujszo():
 5     global kitalalando, megjelenitendo, h, db
 6     szolista = ["DEBRECEN","BUDAPEST",\
 7                 "EGER","SZEGED","MISKOLC"]
 8     vel = randrange(5)
 9     kitalalando = szolista[vel]
10     megjelenitendo = ""
11     db = 0
12     h = len(kitalalando)
13     for i in range(h):
14         megjelenitendo += "X"
15     textBox2.delete(0,END)
16     textBox2.insert(0,megjelenitendo)
```

Az újdonság az alábbi kódokban a szövegmezők tartalmának felhasználása és beállítása.

A **delete** utasítás szolgál a szövegmező törlésére. Két paraméterével egy tartomány[71] adható meg, melyik részt törölje. A 0, END párosítás az egész mező tartalmát törli (15., 30., 32. sorok).

```
18 def betutkiertekel():
19     global kitalalando, megjelenitendo, h, db
20     tipp = textBox1.get()
21     tipp = tipp.upper()
22     elozo = megjelenitendo
23     megjelenitendo=""
24     for i in range(h):
25         if tipp[0] == kitalalando[i]:
26             megjelenitendo += tipp[0]
27             db += 1
28         else:
29             megjelenitendo += elozo[i]
30     textBox2.delete(0,END)
31     textBox2.insert(0,megjelenitendo)
32     textBox1.delete(0,END)
33     if db == h:
34         textBox1.insert(0,"Ön nyert!!!")
```

Az **insert** utasítással lehet elhelyezni egy új karakterláncot a szövegmezőben. Az első paraméterrel megadható, hányadik karaktertől történjen a beszúrás. Ha itt 0-t adunk meg, akkor az elejétől. A második paraméter a megjelenítendő karakterlánc (16., 31., 34. sorok).

A játékos a textBox1-be fog beírni mindig egy betűt találgatva. Ha befejezte a betű megadását, rákattint a *Következő betű* gombra (cmdButton2), ami elindít egy eljárást *(betutkiertekel).* Az eljárás megkeresi a tippelt betű összes előfordulását a keresett szóban, és ahol találat van, helyettesíti az X-et a kitalált betűvel.

---

70 Terjedelmi okokból el kell tekintenünk a program részletes elemzésétől. A változók neve beszédes, így remélem, hogy a könyv korábbi oldalai alapján nem lesz nehéz megérteni.

71 Tartomány alatt a Pythonban használt Range-et értem.

Ebben a kódrészletben újdonság, hogyan kérdezhető le egy szövegdoboz tartalma. Erre a célra a **get** utasítás szolgál, amit változónév = szövegmezőnév.get() formában kell használni (20. sor).

Ha a fenti két eljárással kiegészítettük a programunkat, már játszhatunk vele.

Üzenetablak

Az előzőekben a játék végét az egyik szövegmezőben jeleztük, ami nem egy elegáns megoldás. Ablakos alkalmazásoknál kiemelkedően fontos dolgok közlésére kisméretű üzenetablakokat használunk, amik az alkalmazás előterében jelennek meg. Oldjuk meg, hogy a játék végét egy ilyen üzenetablak jelezze, amit a *nyert* eljárásban hozunk létre.

```
36 def nyert():
37     uzenetablak = Toplevel(ablak)
38     uzenetablak.option_add("*font","Arial 11")
39     uzenet = Message(uzenetablak, \
40         text="Ön nyert!!!\nGratulálok!", width=300)
41     cmdButton4 = Button(uzenetablak, text="OK", \
42         command=uzenetablak.destroy)
43     uzenet.pack()
44     cmdButton4.pack()
45     uzenetablak.mainloop()
```

A főablak előtt megjelenő ablakokat a **Toplevel** tartalmazza. Ebből hozhatunk létre egy példányt, aminek most a neve *uzenetablak*. Paraméterként megadjuk, melyik ablak előtt jelenjen meg *(ablak)* (37. sor).

Magát az üzenetet egy **Message** típusú vezérlővel fogjuk megjeleníteni. Annyiban tér el a **Label**től, hogy ez több soros lehet. A sorokra tördelést a **\n** jellel oldhatjuk meg a **text** paraméteren belül (38–39. sor).

Az ablakot felvértezzük még egy *OK* feliratú parancsgombbal (cmdButton4), amire kattintva az üzenetablak bezáródik (uzenetablak.destroy) (40–41. sor).

A kipróbálásához csak át kell írni a *betutkiertekel* eljárás utolsó sorát, hogy hívja meg a most létrehozott eljárást. Tehát a 34. sorban ez legyen: nyert().

Feladatok

1. Egy futár az egyes utakra az út hosszától függően kap fizetést az alábbi táblázatnak megfelelően. Kérjünk be a felhasználótól egy 1–30 km közötti távolságot, és határozzuk meg, hogy mekkora díjazás jár érte. Legyen két szövegdobozunk, egyik az út hosszának megadásához, másik az eredmény megjelenítéséhez. A szövegdobozokat lássuk el címkékkel, hogy látható legyen, mire valók. Legyen egy parancsgomb, ami a számítást indítja.

1–2 km	500 Ft
3–5 km	700 Ft
6–10 km	900 Ft
11–20 km	1400 Ft
21–30 km	2000 Ft

2. Bővítsük az előző programot egy üzenetablakkal, ami kiírja, ha szabálytalan adatot adott meg a felhasználó. Szabálytalan az 1-nél kisebb, vagy 30-nál nagyobb távolság.
3. Sík padlófelületek burkolását Géza mester a felület nagyságánál 20%-kal több felületet lefedő burkolólap felhasználásával vállalja. Készítsünk programot, ami szövegmezőkből beolvassa a téglalap alakú helyiség két oldalának hosszúságát, valamint a kiválasztott téglalap alakú burkolólap két oldalának hosszát, majd megadja, hány darab burkolólapra van szüksége Géza mesternek a munka elvégzéséhez, szintén egy szövegdobozban. Az ablak a mellékelt módon nézzen ki. (Minden méret megadásakor méter legyen a mértékegység. Az egésznél pontosabb számokat használjunk.)

2. JELÖLŐNÉGYZET, RÁDIÓGOMB, SZÖVEGDOBOZ, MENÜ

Jelölőnégyzet

Jelölőnégyzetnek nevezik azt a kis négyzet alakú vezérlőt, ami úgy viselkedik, mint egy kapcsoló. A pipával ellátott a bekapcsolt, a pipa nélküli a kikapcsolt állapotot jelenti.

Például kérjük be egy telek hosszúságát és szélességét, és határozzuk meg az utána befizetendő adót, ha 1 területegység adója 50 Fabatka. Bizonyos körülmények teljesülésekor 25% kedvezmény jár. Az ügyintéző egy jelölőnégyzettel döntheti el, ad-e kedvezményt vagy sem.

```
 3 def szamol():
 4     a = int(textBox1.get())
 5     b = int(textBox2.get())
 6     t = a * b
 7     ado = t * 50
 8     if chk.get() == 1:
 9         ado = int(0.75 * ado)
10     textBox3.delete(0,END)
11     textBox3.insert(0,ado)
```

```
24 chk = IntVar()
25 chkButton = Checkbutton(ablak, \
26     text="Kedvezmény", variable = chk)
```

Az előző vezérlőkhöz hasonlóan létre kell hozni belőle egy példányt (25–26. sor). A **text** paraméterben adhatjuk meg a jelölőnégyzet mellett megjelenő szöveget. A **variable** paraméterben kell megadnunk egy Tkinter változót. Ezen a változón keresztül fogjuk tudni lekérdezni a jelölőnégyzet állapotát. A változó

nevét megválaszthatjuk, de előzőleg létre kell hozni a változónév = IntVar() utasítással (24. sor).

Az előző változónevet a **get** utasítással alkalmazva lekérdezhetjük a jelölőnégyzet állapotát: 1 a bekapcsolt állapot, 0 a kikapcsolt. A példában lekérdezzük a jelölőnégyzet állapotát, és ha bekapcsolt, azaz 1 (8. sor), akkor csökkentjük az adó összegét 25%-kal (9. sor).

A jelölőnégyzet használható úgy is, hogy a kattintása eseményt indít. Írjuk be a létrehozás sorába (26. sor) negyedik paraméterként: command = szamol, és futtassuk ismét a programot. A jelölőnégyzetre történő minden egyes kattintás újra számolja az eredményt.

Rádiógombok

A rádiógombok[72] kis kör alakú vezérlők, amikből több működik egymással összhangban. Az összetartozó gombok közül csak egy lehet aktív. Ha egy másikra kattintunk, az előző kikapcsolódik, hasonlóan a régi, nyomógombos rádiókhoz, amikről a nevüket kapták.

Például kérjük be egy telek hosszúságát és szélességét, és határozzuk meg az utána befizetendő adót, ha 1 területegység adója 50 Fabatka. Bizonyos körülmények teljesülésekor 10 %, más körülmények között 25% kedvezmény jár. Az ügyintéző három rádiógombbal döntheti el, ad-e kedvezményt, és ha igen, 10%-ot vagy 25%-ot.

Itt is felhasználunk egy Tkinter változót, amit először létrehozunk a változónév = StringVar() utasítással (24. sor). A közös változónév *(rb)* határozza meg, hogy melyik rádiógombok tartoznak egy csoportba. Az ábrán lévők mind, hiszen a **variable** paraméter értéke mindenütt *rb*. A **value** paraméter határozza meg, mi legyen az *rb* értéke, ha arra a rá-

```
 3 def szamol():
 4     a = int(textBox1.get())
 5     b = int(textBox2.get())
 6     t = a * b
 7     ado = t * 50
 8     szl = 1 - float(rb.get())
 9     ado = int(szl * ado)
10     textBox3.delete(0,END)
11     textBox3.insert(0,ado)
24 rb = StringVar()
25 rButton1 = Radiobutton(ablak, value='0', \
26     text="0%", variable = rb)
27 rButton2 = Radiobutton(ablak, value='0.1', \
28     text="10%", variable = rb)
29 rButton3 = Radiobutton(ablak, value='0.25', \
30     text="25%", variable = rb)
31 rButton1.invoke()
```

72 Választógombnak vagy opciógombnak is nevezi néhány szakirodalom.

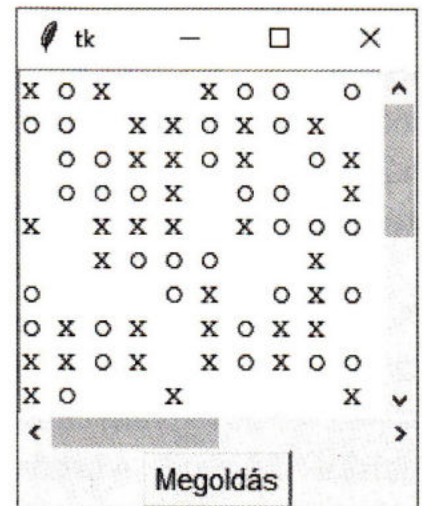
diógombra kattintunk. A value mindig karakterlánc (ha számot írunk oda idézőjelek nélkül, akkor is), tehát felhasználáskor szükség esetén majd konvertálni kell a megfelelő szám típussá. A **text** paraméterben adhatjuk meg a rádiógomb mellett megjelenő szöveget.

A megjelenítés előtt, de a létrehozások után feltétlenül adjuk meg, kezdetben melyik rádiógomb legyen bekapcsolt a csoportból. Ezt a rádiógombnév.invoke() paranccsal valósíthatjuk meg. Ha nem tesszük, tapasztalataim szerint induláskor mindhárom gomb bekapcsoltnak látszik, egészen az egyik gomb kiválasztásáig.

A rádiógomb-csoport értékéhez a változón *(rb)* keresztül juthatunk hozzá a **get** utasítás segítségével. Erre látunk példát a 8. sorban, ahol a változó értékét előbb valós számmá alakítjuk, majd kivonjuk 1-ből, hogy megkapjuk a százaléklábat (szl), amire csökkenteni kell az adó értékét (9. sor).

A rádiógomb használható úgy is, hogy a kattintása eseményt indít. Írjuk be a létrehozások sorába (26., 28., 30. sor) ötödik paraméterként: command = szamol, és futassuk ismét a programot. Bármelyik rádiógombra történő kattintás azonnal újra számolja az eredményt.

Szövegdoboz gördítősávval



A **szövegdobozok** többsoros adatmegjelenítést, adatbevitelt tesznek lehetővé[73].

A példaprogramban egy 20 x 20-as amőbapályát jelenítünk meg grafikus felületen, véletlenszerűen feltöltve a két játékos X és O jelével, valamint üres mezőkkel.

A pálya nem minden eleme látható, ezért **gördítősávokat** alkalmazunk a szövegdoboz kívánt részének megjelenítéséhez.

Az összetartozó vezérlőket szokás egy **keretbe** (frame) foglalni, mert így együtt pozicionálhatók az ablakban.

Keretet a **Frame** osztály alapján hozhatunk létre keretnév = Frame(ablaknév) formában (21. sor). Az *ablaknév* az az ablak, amiben létrehozzuk a keretet. A továbbiakban a vezérlőkben, amiket a kereten belül szeretnénk elhelyezni, első paraméterként a keret nevét kell megadni az ablak neve helyett (22., 26., 27. sorok).

A szövegdoboz létrehozásához a szövegdoboznév = Text(szülőablaknév,…) utasítást használhatjuk (22–23. sor). A szülőablaknév most a keret neve *(keret),* mivel itt szeretnénk elhelyezni. További paraméterek a szövegdoboz szélessége (**width**) és

73 Ellentétben a *szövegmezővel,* ami csak egyetlen sorban tartalmazhat adatot.

magassága (**height**) karakterekben kifejezve. A **font** meghatározza az alapértelmezett betűtípust és méretet. Azért érdemes a Courier-t választani, mert karakterei egyforma szélesek, így például minden sor harmadik betűi egymás alá kerülnek, egy oszlopot alkotva. Fontos paraméter a **wrap**, amivel a sortörést *tilthatjuk* szóhatáron. Alapvetően ez engedélyezve van, ami azzal jár, hogy a kiírt szó, ha már nem fér ki, automatikusan új sorban kezdődik, alkalmazkodva a szövegdoboz szélességéhez. Ekkor nincs szükség vízszintes gördítősávra, hiszen úgyis minden olvasható. Most mi nem ezt szeretnénk, tehát ezt a lehetőséget tiltani kell:

```
wrap = NONE
```

Gördítősávokat a gördítősávnév = Scrollbar(szülőobjektumnév) utasítással hozhatunk létre (26–28. sor). Alapértelmezésben függőleges gördítősávot kapunk. Ha vízszintest szeretnénk, alkalmazni kell az orient=HORIZONTAL paramétert (28. sor).

```
21 keret = Frame(ablak)
22 kijelzo = Text(keret, height=10, \
23     width=20, font="Courier 11", wrap=NONE)
24 cmdButton = Button(ablak, \
25     text="Megoldás", command=feladat)
26 scrkijelzof = Scrollbar(keret)
27 scrkijelzov = Scrollbar(keret, \
28     orient=HORIZONTAL)
29 scrkijelzof.config(command=kijelzo.yview)
30 kijelzo.config( \
31     yscrollcommand = scrkijelzof.set)
32 scrkijelzov.config(command=kijelzo.xview)
33 kijelzo.config( \
34     xscrollcommand = scrkijelzov.set)
35 keret.pack()
36 scrkijelzov.pack(side=BOTTOM, fill=X)
37 kijelzo.pack(side=LEFT)
38 scrkijelzof.pack(side=RIGHT, fill=Y)
39 cmdButton.pack()
```

Még hozzá kell rendelni a gördítősávokhoz a szövegdobozokat a gördítősávnév.config(command=szövegdoboznév.yview), illetve *...xview* paranccsal (29. és 32. sor). Ezek után a gördítősávot mozgatva, mozog a szövegdoboz tartalma is.

Ajánlatos a fordított kapcsolatot is létrehozni: Ha a szövegdobozban mozgunk, kövessék a gördítősávok. Erre a célra szolgál a szövegdoboznév.config(yscrollcommand=gördítősávnév.set) parancs, illetve ennek *xscrollcommand* párja (30–31. sor, 33–34. sor).

A vezérlőket a *kereten belül* kell megjeleníteni, például a **pack** utasítással. A **side** paraméterével megadjuk, hogy az adott vezérlő a keret melyik szélére kerüljön. A vízszintes gördítősáv az aljára (36. sor), a szövegdoboz a bal oldalra (37. sor), a függőleges gördítősáv a jobb oldalra (38. sor). A **fill** pa-

```
4  def feladat():
5      kijelzo.delete("1.0","end")
6      for i in range(20):
7          for j in range(20):
8              vsz = randrange(3)
9              if vsz == 0:
10                 jel = " "
11             elif vsz == 1:
12                 jel = "O"
13             else:
14                 jel = "X"
15             kijelzo.insert(INSERT, jel + " ")
16         kijelzo.insert(INSERT, "\n")
```

raméterrel adjuk meg, hogy a gördítősávok kitöltsék vízszintesen (36. sor), illetve függőlegesen (38. sor) a keretet.

A keretet is meg kell jeleníteni egy külön utasítással (35. sor).

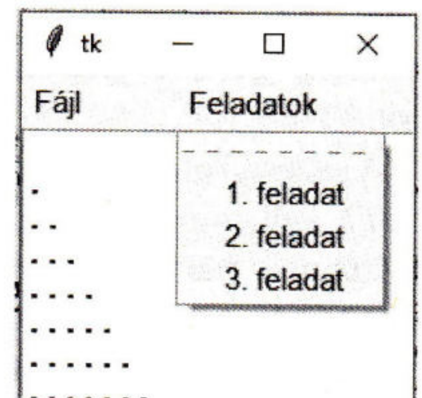
A szövegdoboz használatához elengedhetetlen utasítás a törlés, amit a **delete** utasítással végezhetünk el (5. sor). Az első paraméter megmutatja, honnan kezdődjön a törlés, a második, meddig tartson. A paramétereket sor.oszlop alakban adjuk meg általában. A sorok számozása 1-gyel, az oszlopoké 0-val kezdődik. Így a példában látható *1.0* a bal felső sarokban található karaktert jelenti. Hogy ne kelljen bajlódnunk a sorok és oszlopok megszámlálásával, az utolsó karakter helyzete helyett használhatjuk az **end** kulcsszót. Ha nem adjuk meg a második paramétert, az utasítás egyetlen karaktert töröl.

Karakterláncot az **insert** utasítással vihetünk a szövegdobozba. Első paramétere az a pozíció, ahonnan kezdve szeretnénk beszúrni a második paraméterben megadott karakterláncot. A pozíciót itt is megadhatjuk sor.oszlop alakban, vagy ha az aktuális pozíciótól akarunk beszúrni, használhatjuk helyette az **INSERT** kulcsszót (15. sor).

Kötelező sortörést a **\n** karakter elhelyezésével tudunk kiváltani (16. sor).

A szövegdoboz tartalma szerkeszthető, így szükségünk lehet tartalmának kiolvasására. Ezt a változónév = szövegdoboznév.get(kezdőpozíció, végpozíció) utasítással hajthatjuk végre. Például a print(kijelzo.get("1.0",END+"-1c")) kiírja a teljes szövegdoboz tartalmát a parancsértelmezőben[74].

Menü



Általában az ablak címsora és munkaterülete között helyezkedik el a **menüsor**, ami a **menü**ket tartalmazza. Egy menüt választva **menüpontok** válnak láthatóvá, amikre kattintva egy-egy eseményt válthatunk ki, ezzel irányítva programunkat.

Írjunk egy programot, ami egy szövegdobozban megold három feladatot. A három feladat közül a *Feladatok* menüből választhassunk. Legyen lehetőség a *Fájl* menüből a szövegdoboz törlésére, valamint az alkalmazásból kilépésre.

A menüsorokat célszerűen keretekben helyezzük el (28. sor). A keretet az ablak tetején (side = TOP) jelenítjük meg a **pack** utasítással (29. sor) úgy, hogy vízszintesen végig kitöltse az ablakot (fill = X).

---

74 A "-1c" egy karaktert levág a végéről, ami egy szövegdoboz végén biztosan egy sorvége jel. A példaprogram nem tartalmaz ilyen sort. Ez plusz információ.

A mindig olvasható példány menüket a **Menubutton** típusból lehet létrehozni (30–31., 39–40. sorok). Első paramétere annak a menüsornak neve, aminek a része lesz *(menusor)*, **text** paramétere a menü megjelenő neve *(Fájl)*. A menüt a menüsort képező kereten belül balra igazítva jelenítjük meg a **pack** utasítás segítségével (32., 41. sorok). Mivel a *menu1* (Fájl) megjelenítése van előbb, az lesz ténylegesen a bal szélen, majd mellette a *menu2* (Feladatok).

```
28 menusor = Frame(ablak)
29 menusor.pack(side = TOP, fill = X)
30 menu1 = Menubutton(menusor, \
31     text="Fájl")
32 menu1.pack(side = LEFT)
33 fajl = Menu(menu1)
34 fajl.add_command(label="Törlés", \
35     command=torles)
36 fajl.add_command(label="Kilépés",\
37     command=ablak.destroy)
38 menu1.config(menu = fajl)
39 menu2 = Menubutton(menusor, \
40     text="Feladatok")
41 menu2.pack()
42 feladat = Menu(menu2)
43 feladat.add_command(label="1. feladat",\
44     command=feladat1)
45 feladat.add_command(label="2. feladat",\
46     command=feladat2)
47 feladat.add_command(label="3. feladat",\
48     command=feladat3)
49 menu2.config(menu = feladat)
```

Minden menühöz létre kell hozni egy **Menu** típusú példányt (33., 42. sorok – *fajl, feladat*), amihez majd hozzá lehet adni a tényleges menüpontokat.

A menüpontok hozzáadása az **add_command** utasítással történik. Itt a **label** paraméter szolgál a menüpont megjelenő nevének megadására, a **command** után pedig azt adjuk meg, mi történjen, ha erre a menüpontra kattintottak. Ide kerülhet egy utasítás, például kilépésnél az ablak megszüntetése (37. sor), vagy egy eljárás neve, amit előzőleg elkészítünk (35., 44., 46., 48. sorok).

A menüpontok beállításának végeztével jelezni kell a menünek, hogy változás történt (38., 49. sorok).

Az ablak többi részének felépítését az olvasóra bízom, akárcsak az eljárások (feladat1, feladat2, feladat3, torles) megírását[75]. Ha mégsem menne, a fájlmelléklet segíteni fog.

Utólagos módosítások

Gyakran előfordul, hogy a létrehozott ablakon módosítani kell.

Leggyakrabban a vezérlők felirata szokott megváltozni. Más lesz egy parancsgomb feladata, vagy más szöveget kell megjelenítenie egy címkének. Ehhez meg

75 Nálam az 1. feladat 20 x 20 pont karakter megjelenítése volt, a 2. feladatban minden sorban eggyel több pont karakter jelent meg 20-szal bezárólag, a 3.-ban soronként eggyel kevesebb, 20-ról indulva. De a grafikus felület megismerése szempontjából mindegy, mit jelenítünk meg, csak három különböző dolog legyen.

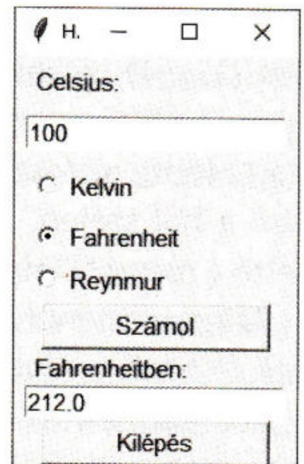
kell adni a vezérlő neve mögé írt **configure** utasítással az új feliratot, az alábbi példa szerint. A példában a *cmdButton* parancsgomb feliratát *(text)* változtatjuk meg "Vissza"-ra.

```
cmdButton.configure(text = "Vissza")
```

Másik gyakori eset, hogy egy feleslegessé vált vezérlőt le kell szedni. Mind a pack, mint a grid megjelenítő utasításnak van eltüntető párja: **pack_forget**, **grid_forget**. A vezérlő neve után kell írni őket. Például: chkButton.pack_forget().

Feladatok

1. Készítsünk egyablakos alkalmazást, ami átváltja a hőmérsékleti adatokat Celsius-fokból Kelvinbe, Fahreinheitbe, Réaumurba. Legyen két szövegmező, egy a bemeneti adatnak, egy az eredménynek, két címke, ami jelzi, hogy melyik szövegdobozban mi található, egy parancsgomb, ami az átváltást indítja, és három rádiógomb, amivel kiválasztható, mibe váltunk. A rádiógombok legyenek balra igazítottak. Az alsó címkén mindig az a mértékegység legyen olvasható, amibe váltunk. Az ablak a mellékelt ábrán látható módon nézzen ki. Tehát az opciógombok egymás alatt legyenek. Sem a feliratok, sem a gombok ne legyenek az ablak szélén. (A Kelvinben mért hőmérsékletet megkapjuk, ha a Celsiusban mérthez hozzáadunk 273,15-ot, a Reynmurban mért hőmérséklet a Celsiusban mértnek a 80%-a, a Fahrenheit értéket megkapjuk, ha a Celsiusban megadott értéket szorozzuk 1,8-del, és hozzáadunk 32-t.)



2. Jelenítsük meg az alábbi lista szavait egy szövegdobozban, amit függőleges gördítősávval látunk el. Minden szó külön sorban legyen, és a szövegdobozt állítsuk akkorára, hogy legyen értelme a függőleges gördítősávnak. Legyen egy parancsgombunk, amire kattintva a lista ábécésorrendben jelenik meg, majd még egyszer rákattintva visszaáll az eredeti állapot. Legyen még egy jelölőnégyzetünk is, amit bekapcsolva a rendezés fordított irányú lesz.

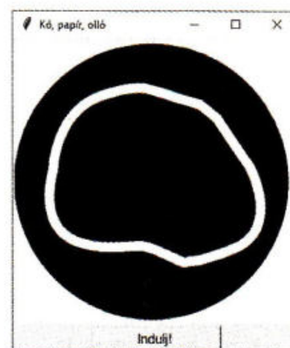
   words = ["izomtalan", "tornyai", "abbahagyjuk", "rideg", "azon", "mindenhonnan", "nem"]
3. Bővítsük az 1. feladat programját úgy, hogy térfogat egységeket is tudjon átszámítani (köbméterből köbdeciméterbe, köbcentiméterbe, literbe), majd nyomás egységeket is (Pascalból higanymilliméterbe, atmoszférába, barba). A három mennyiség között (hőmérséklet, térfogat, nyomás) a "Mennyiségek" menüből lehessen választani. Legyen még egy Fájl menü is, ahonnan ki lehet lépni az alkalmazásból, illetve megjelenít egy névjegyet a program adataival: készítő neve, elkészülés időpontja.

3. KÉPMEZŐ, LISTAMEZŐ, IDŐZÍTÉS

Képmező

Jelenítsük meg egy parancsgomb megnyomásával a 0.png képet (kő) grafikus felületen.

A Tkinterben nincs külön képmező-vezérlő. Helyette egy **vásznat** használunk, amire ráfeszítjük a képet. Ennek a menete a következő:



1. Létrehozzuk a vászonpéldányt *(vaszon)* a **Canvas** alapján a kódban a vezérlők területén (16–17. sor). Első paraméter az objektum neve, ahol elhelyezzük. Ez most az *ablak*. A **width** paraméterrel a szélességet, a **height**tal a magasságot állítjuk be, képpontokban kifejezve. Mivel a képeink 310 x 310-esek, ezért célszerű a vásznakat is ekkorára állítani, hogy ne maradjon szabadon vászon, de a kép teljesen kiférjen. A **bg**[76] a vászon háttérszínét állítja be, most fehérre *(white)*.
2. A vászon vezérlő, így gondoskodnunk kell a megjelenítéséről (18. sor).
3. Létrehozzuk a **Photoimage** típusú objektumot, amiben a képet tárolni lehet a memóriában (6. sor). A példány neve most *foto*. A **file** paraméterben adható meg a kép elérési útja és neve.
4. Elhelyezzük a képet a vásznon (7–8. sor). Az első két paraméter a kép közepének a helye a vásznon[77]. A 310 x 310-es kép középpontja a 155, 155-nél van, aminek a vászon 155, 155-ös pontjában kell elhelyezkednie. Ez utóbbi 155, 155 a két paraméter. Az **image** paraméterben adhatjuk meg a memóriában található kép nevét, ami most a *foto*.

```
 4 def indulj():
 5     global vaszon, foto
 6     foto = PhotoImage(file="0.png")
 7     item = vaszon.create_image(\
 8         155,155, image=foto)
14 cmdButton = Button(ablak, text="Indulj!",\
15     width=15, command=indulj)
16 vaszon = Canvas(ablak, width=310,\
17     height=310, bg="white")
18 vaszon.pack()
19 cmdButton.pack()
```

A *vaszon* és *foto* változóknak globálisnak kell lenniük, mert a főprogramnak hozzá kell tudni férni a megjelenítéskor.

76 A background rövidítése.

77 Egy vászonra akár több képet is tehetnénk, ezért ezek a látszólagos bonyodalmak.

Időzítés

Módosítsuk úgy az előző programot, hogy véletlenszerűen jelenítse meg a 0.png (kő), 1.png (papír), 2.png (olló) jelképét, rövid időközönként automatikusan váltakoztatva őket. A gomb megnyomása indítsa a folyamatot.

```
 4 def indulj():
 5     global vaszon, foto
 6     vel = randrange(3)
 7     fajlnev = str(vel) + ".png"
 8     foto = PhotoImage(file=fajlnev)
 9     item = vaszon.create_image(\
10         155,155, image=foto)
11     ablak.after(100,indulj)
```

Készítünk egy véletlenszámot (6. sor), majd karakterlánccá alakítás (**str**) után hozzáfűzzük a kiterjesztést (png) (7. sor). Ezek után a memóriába a fajlnev változóba megadott fájlt töltjük be, ami időnként 0.png, máskor 1.png, vagy 2.png lesz.

Az időzítés utasítása az **after**. Első paraméterében megadható, hány milliszekundum[78] múlva történjen meg a második paraméterben megadott utasítás. Az utasítás itt most egy eljárás, ami ráadásul önmagára hivatkozik. Így addig fog ismétlődni mindig újabb és újabb véletlen kép betöltése, míg le nem állítjuk a programot. Könnyen belátható, hogy ilyen módon mozgatni is tudnánk az ablakban objektumokat, így ez az utasítás az automatikus animációk kulcsa.

Listamező

A listamező elsősorban valaminek a kiválasztására való. Minden sor egy külön elem, ami kiválasztható, és a program ennek megfelelően folytatható.

Alakítsuk most úgy a programunkat, hogy a képmezőben megjelenjen a listamezőből kiválasztott kép.

A listamező példányát a **Listbox** alapján kell létrehozni (17. sor). Első paramétere az objektum, amiben megjelenítjük *(ablak)*. További paraméterként megadható a magassága (**height**) és a szélessége (**width**) karakterekben kifejezve. Ha a listának több eleme van, mint amekkora a magassága, praktikus mellé gördítősávot helyezni a szövegdoboznál megismert módon.

Az **insert** utasítással szúrhatunk be a listába elemeket (20–21. sor). Az első paraméter az a pozíció, ahonnan a beszúrást kezdjük. Az END annyit jelent, hogy az eddigiek után szúródnak be a további paraméterekben megadott elemek.

---

78 másodperc ezredrésze

```
 4 def valaszt(evt):
 5     global foto
 6     kiv = lista.curselection()
 7     k = kiv[0]
 8     fajlnev =  str(k) + ".png"
 9     foto = PhotoImage(file=fajlnev)
10     item = vaszon.create_image(\
11         155,155, image=foto)
17 lista = Listbox(ablak, height=3)
18 lista.bind("<<ListboxSelect>>", \
19     valaszt)
20 lista.insert(END, "kő", "papír",\
21     "olló")
22 lista.select_set(0)
23 vaszon = Canvas(ablak, width=310,\
24     height=310, bg="white")
25 vaszon.pack()
26 lista.pack()
```

A program indulásakor nincs kijelölve egyetlen listaelem sem. Ha szeretnénk, hogy az egyik aktív legyen (kék sáv legyen rajta), a **select_set** utasítással állíthatjuk be. Az utasítás mögött, a zárójelek közötti 0 a listamező legfelsőbb elemére utal (22. sor).

A **bind** utasítással beállíthatjuk, hogy milyen kiváltó ok (első paraméter) esetén milyen utasítás történjen (második paraméter). Esetünkben a lista valamelyik elemének kiválasztása a kiváltó ok, és a *valaszt* eljárásnak kell megtörténnie (18–19. sor).

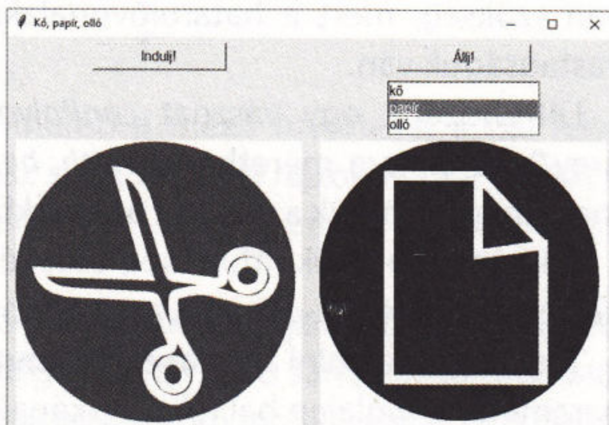
A listamezőt is meg kell jeleníteni az ablakban (26. sor).

A listamezőnek lehet több eleme is kiválasztva egyszerre. A kiválasztott elemek sorszámát (indexét) a **curselection** utasítással tölthetjük át egy listába *(kiv)* (6. sor). Ha csak egy elemet választottunk ki, az a 0-s indexű lesz. Így a 7. sor után a k változóba fog kerülni a kiválasztott listamezőelem indexe.

Ebből képezzük aztán a fájlnevet (8. sor), amit aztán betöltünk (9. sor), és a vásznon megjelenítjük (10–11. sor) az időzítésnél látott módon.

Feladat

Minden tudás adott, hogy elkészítsünk egy kő, papír, olló játékot, amiben a gép ellen lehet játszani. A bal oldalon a gép változtatja a képet gyors egymásutánban, a jobb oldalon mi választhatunk a listából. A gép tippjeinek változását az *Indulj!* gombbal lehet indítani. A játékot az Állj! gombbal lehet megállítani. Ez utóbbi egy kiértékelést is indít, és a program egy üzenetablakban megjeleníti, ki nyert.



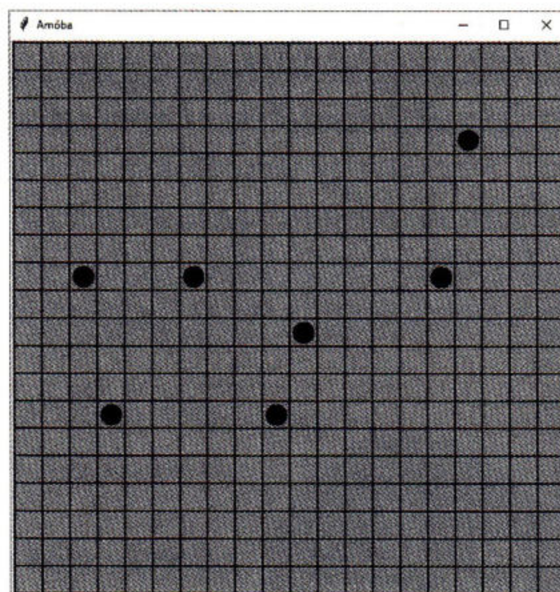
4. RAJZOLÁS

Áttekintés

A Tkinter lehetőséget biztosít egyszerű alakzatok (vonalak, téglalapok, ellipszisek, sokszögek, ívek) megrajzolására. Rajzolni csak már létező vászonra (Canvas) lehet.

A rajz helyének megadásához koordinátákat szoktunk használni. A koordinátarendszer kiindulási (0; 0) pontja a vászon bal felső sarka. Az x koordináták jobbra, az y koordináták lefelé növekednek, képpontonként eggyel.

Téglalap



Készítsük el egy amőbajáték tábláját. Téglalapokból fogjuk felépíteni a rácsozatot, aminek az oldalhosszait egyformának vesszük, így négyzeteket kapunk.

A többször felhasznált adatoknak (mező oldalának hossza – *mmeret,* mezők száma – *mezok,* pályaméret – **pmeret**, stb.) érdemes változókat használni, és csak egy helyen konkrét értéket adni neki. Így ha eszünkbe jut megváltoztatni, akkor egyetlen ponton kell csak hozzányúlnunk a programhoz (18–20. sor). A *korr* változóra azért van szükség, mert a határolóvonalaknak vastagságuk van.

Létrehozunk egy vásznat *canPalya* néven, sötétszürke háttérrel (bg="dark grey"), és akkora méretben *(width, height),* amekkora a pálya lesz (21–22. sor), majd megjelenítjük a vásznat az ablakban (23. sor).

A könnyebb áttekinthetőség érdekében a pálya megrajzolását egy külön eljárásban írjuk meg (3–12. sor), amit a főprogramból meghívunk (24. sor).

Téglalapot rajzolni a **create_rectangle** utasítással lehet (9–12. sor). Az első két paraméter a téglalap bal felső sarkának vízszintes, majd függőleges koordinátája (x1, y1). A harmadik és negyedik paraméter a jobb alsó sarok vízszintes, majd függőleges koordinátája (x2, y2). A **width** paraméterrel állíthatjuk be a határolóvonal vastagságát, az **outline**-nal a színét (most sötétkékre). A téglalap belseje is

kitölthető a **fill**ben megadott színnel. Ezt most ugyanolyanra állítottuk, mint a vászon hátterét, sötétszürkére.

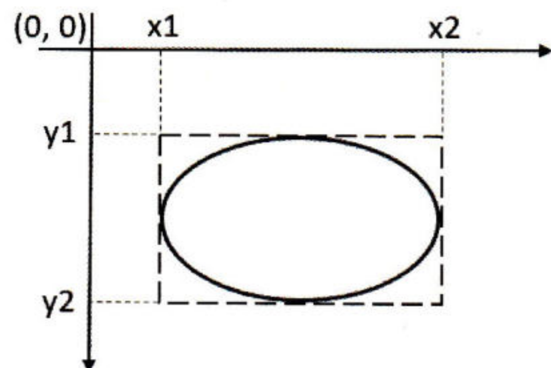
A megelőző sorokban (6–8. sor) meghatározzuk a téglalap bal felső és jobb alsó sarkának koordinátáit.

Az 5. sor ciklusa teszi lehetővé egy sornyi négyzet megjelenítését, a 4. sor ciklusa egy négyzetsor kirajzolását ismétli. Mindkét ciklus növekménye a mező mérete, így az x-ben és y-ban mindig a korrekció híján a téglalap bal felső sarkának koordinátái vannak.

```
 3 def palyarajz():
 4     for x in range(0, pmeret, mmeret):
 5         for y in range(0, pmeret, mmeret):
 6             x1, y1 = x+korr, y+korr,
 7             x2 = x+mmeret+korr
 8             y2 = y+mmeret+korr
 9             canPalya.create_rectangle(\
10                 x1,y1,x2,y2, width=2, \
11                 fill="dark grey", \
12                 outline="dark blue")
18 mezok, mmeret = 20, 30
19 pmeret = mezok * mmeret
20 korr = 4
21 canPalya = Canvas(ablak, bg="dark grey", \
22     height=pmeret+korr, width=pmeret+korr)
23 canPalya.pack()
24 palyarajz()
```

Kör, ellipszis

Készítsünk eljárást, ami kirajzol egy vörös, kör alakú figurát a megadott mezőbe. Az eljárás képes legyen a mező sorszámait fogadni (0–19, 0–19).



Az eljárásnak két paramétere lesz, amik a mező x és y irányú sorszámait tartalmazzák (13. sor).

A következő sorokban átszámítjuk a sorszámokat grafikus koordinátákra (14–16. sor). A sorszám szorozva (x*mmeret, y*mmeret) a mező méretével a bal felső sarok koordinátáit adja. Azonban akkor lesz szép, ha nem rajzoljuk rá a figurát a rácsozatra, ezért beljebb kezdjük valamennyivel (+2*korr). Ennek a mértékét kísérletezéssel állítottam be.

Az eljárás lelke a **create_oval** utasítás, ami az ellipszist rajzolja (17–18. sor). Az első két paraméter az ellipszist befoglaló téglalap bal felső sarkának koordinátái (x1, y1), a harmadik és negyedik paraméterek ugyanezen téglalap jobb alsó sarkának koordinátái (x2, y2). Mivel esetünkben a befoglaló téglalap oldalai egyformák, így négyzetről van szó. A négyzet pedig a kör befoglaló alakzata, tehát ténylegesen most kört rajzolunk. A **width** paraméterrel állíthatjuk be a határolóvonal

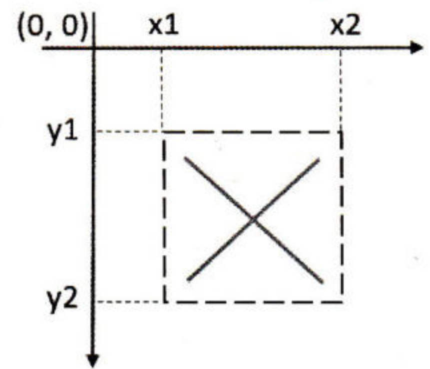
vastagságát, az **outline**-nal a színét (fekete). Az ellipszis belseje is kitölthető a **fill**ben megadott színnel (vörös).

Ha kész az eljárás, úgy tudjuk kipróbálni, ha a főprogramból meghívjuk (30–31. sor).

```
13 def kor(x, y):
14     x1 = x*mmeret+2*korr,
15     y1 = y*mmeret+2*korr
16     x2, y2 = (x+1)*mmeret, (y+1)*mmeret
17     canPalya.create_oval(x1,y1,x2,y2, \
18        width=2,fill="red",outline="black")
28 canPalya.pack()
29 palyarajz()
30 kor(5,10)
31 kor(19,19)
```

Szakasz

Készítsünk eljárást, ami kirajzol egy zöld, X alakú figurát a megadott mezőbe. Az eljárás képes legyen a mező sorszámait fogadni (0–19, 0–19).



Az eljárásnak két paramétere lesz, amik a mező x és y irányú sorszámait tartalmazzák (20. sor).

A következő sorokban átszámítjuk a sorszámokat grafikus koordinátákra (21–24. sor). A sorszám szorozva (x*mmeret, y*mmeret) a mező méretével a bal felső sarok koordinátáit adja. Azonban akkor lesz szép, ha nem rajzoljuk rá a figurát a rácsozatra, ezért beljebb kezdjük valamennyivel (+2.5*korr), és előbb is fejezzük be valamivel (-0.5*korr). Ennek a mértékét kísérletezéssel állítottam be.

Az eljárás központja a **create_line** utasítás, ami a szakaszokat[79] rajzolja (25–28. sor). Az első két paraméter a szakasz egyik végpontjának koordinátái (x1, y1), a harmadik és negyedik paraméterek a másik végpontjának koordinátái (x2, y2). A **width** paraméterrel állíthatjuk be a határolóvonal vastagságát, a **fill**-lel pedig a színét (zöld).

```
20 def iksz(x, y):
21     x1 = x*mmeret+2.5*korr
22     y1 = y*mmeret+2.5*korr
23     x2 = (x+1)*mmeret-0.5*korr
24     y2 = (y+1)*mmeret-0.5*korr
25     canPalya.create_line(x1,y1,x2,y2, \
26         width=4,fill="green")
27     canPalya.create_line(x2,y1,x1,y2, \
28         width=4,fill="green")
38 canPalya.pack()
39 palyarajz()
40 iksz(9,6)
41 iksz(19,0)
```

Az X-et két szakaszból rakjuk össze. Az első a mező bal felső sarkától a jobb alsó sarkáig tart (25–26. sor), a második a jobb felsőtől a bal alsóig (27–28. sor).

Ha kész az eljárás, úgy tudjuk kipróbálni, ha a főprogramból meghívjuk (40–41. sor).

79 szakasz = egyenes vonal, aminek végpontjai vannak

Kattintás koordinátái

Szerencsés lenne, ha a játékosok kattintással tudnák kiválasztani a mezőt, ahová a figurájukat lerakhatják.

A rajzvászonnak nincs command paramétere, így nekünk kell hozzárendelni, hogy az egér kattintásakor melyik utasítás fusson le. Ezt a hozzárendelést a **bind** paranccsal tehetjük meg (51. sor). Első paraméterként megadjuk a kiváltó okot *(<Button-1> =* kattintás az egér bal gombjával), majd a második paraméterben a végrehajtandó eljárás nevét *(palyankattint).*

Az eljárásnak (30. sor) lesz egy paramétere *(event),* ebben utaznak az esemény körülményei, esetünkben az egérkurzor koordinátái *(event.x, event.y)* a kattintás pillanatában.

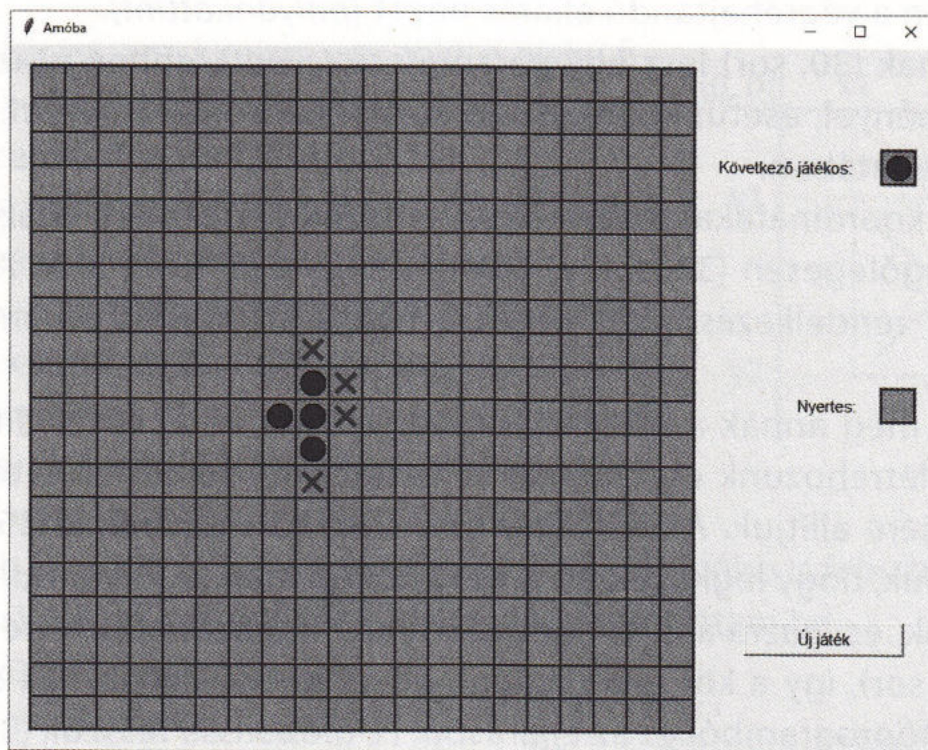
A grafikus koordinátákat átszámoljuk a mezők sorszámára vízszintesen (32. sor, x) és függőlegesen (33. sor, y) egyaránt. Így akár kört, akár X-et akarunk rajzolni, már rendelkezésünkre állnak a hozzá szükséges paraméterek (35. vagy 38. sor).

Hátra van még annak a megoldása, hogy a kör és az X felváltva jelenjen meg. Ehhez létrehozunk egy logikai változót *(em),* és ennek értékét mindig az ellenkezőjére állítjuk. Amikor *Igaz* (34. sor), kört rajzolunk (35. sor), majd *Hamis*ra állítjuk, hogy legközelebb X-et rajzoljon majd. Amikor *Hamis* (37. sor), X-et rajzolunk, és *Igaz*ra állítjuk. Kezdőértékként *Igaz*at adunk neki a főprogramban (52. sor), így a kör fog kezdeni. Mivel az *em* változónak elérhetőnek kell lennie a főprogramból és az eljárásból is, globálissá tesszük (31. sor).

```
30 def palyankattint(event):
31     global em
32     x = int((event.x-korr)/mmeret)
33     y = int((event.y-korr)/mmeret)
34     if em:
35         kor(x, y)
36         em = False
37     else:
38         iksz(x, y)
39         em = True
49 canPalya.pack()
50 palyarajz()
51 canPalya.bind("<Button-1>", palyankattint)
52 em = True
```

Feladat

Minden tudás adott, hogy elkészítsünk egy amőbajátékot, amiben két ember játszik egymás ellen, felváltva kattintva a táblán. A számítógép jelezze ki a következő játékost, és ellenőrizze minden lépés után, nyert-e valaki. Ha igen, jelenítse meg a győztest, és ne engedjen a pályán további módosítást, míg új játékot nem kezdünk. Egy parancsgombbal lehessen új játékot kezdeni.



IX. KIEGÉSZÍTÉSEK, ÖSSZEFOGLALÁS

1. ALAPOK

A parancsértelmező és a szerkesztő használata

Feladat	Menü	Billentyűk
Kilépés	File/Exit	Ctrl+Q
Új Python programfájl	File/New	Ctrl+N
Fájl megnyitása	File/Open	Ctrl+O
Fájl mentése más néven	File/Save As...	Ctrl+Shift+S
Fájl mentése az előző nevén	File/Save	Ctrl+S
Program futtatása (csak a szerkesztőben)	Run/Run Module	F5
Sorszámozás megjelenítése (csak a szerkesztőben)	Options/Show Line Numbers	-
Programfutás megszakítása (csak a parancsértelmezőben)	Shell/Interrupt Execution	Ctrl+C
Parancsértelmező újraindítása	Shell/Restart Shell	Ctrl+F6

Szintaktika általában

- Minden utasítás vagy parancs végén meg kell nyomni az **Enter** billentyűt. A programsor a bekezdés vége jelig (Enter) tart.
- Az utasítások vagy parancsok kulcsszavai után szabadon bánhatunk a szóközökkel.
- A karakterláncokat írhatjuk idézőjelek vagy aposztrófok közé is. Nincs különbség, de valamelyiket következetesen (egy konkrét karakterlánc esetén mindkettő egyforma) alkalmazni kell.
- A vezérlési szerkezetek **:**-ra végződnek.
- A vezérlési szerkezethez tartozó utasítások 4 karakterrel beljebb kezdődnek. A szerkesztő gyakran magától odarakja a kurzort. Ha ez mégsem történne meg, a **Tab** (tabulátor) gomb egyszeri megnyomásával megtehetjük. Ez strukturálja a programot.
- A programsor megtörhető a **** segítségével.
- A listák, tuple-k, szótárak sorai megtörhetők a zárójelen belül, valamelyik elem után. (\ használata nélkül is.)
- Ha a sorban bárhol **#** jelet alkalmazunk, a mögötte található rész *megjegyzésnek* számít.

Kiíratás

- A konzolra történő kiíratás a **print** utasítással történik.
- Több tartalom kiíratható, ha az utasításban azokat egymástól vesszővel elválasztjuk: Például: print("A téglalap kerülete:", k, "egység")
- Alapbeállítás szerint az egy utasításban kiírt tartalmak között egy-egy szóköz lesz, az utolsó után pedig új sor elején folytatódik a kijelzés.
- A kiírt tartalmak elválasztásában eltérhetünk az alapbeállítástól, ha beállítjuk a sep, illetve end paramétereket. A **sep** az utasításban szereplő kiírt tartalmak közötti elválasztást állítja be, az **end** a kiíratás utánit. Például: print(c, sep=";", end="")

Adattípus	Jelölő
egész	%d
valós	%f
karakterlánc	%s

- Formázott kiíratást érhetünk el a jelölők (markerek) segítségével. A jelölők (pl. **%d**) helyére helyettesítődik a sorban ugyanannyiadik változó. (Az első *%d* helyére az *sz1*, stb.)
  print ("%d/%d * %d/%d = %d/%d" % (sz1,n1,sz2,n2,sz1*sz2,n1*n2))
- Valós számok (tizedestörtek) esetén a tizedesjegyek számának beállítása a %f jelölő segítségével történik a következő minta szerint: **%.2f**, ha 2 a megjelenő tizedesjegyek száma.

Adatbekérés

Adatbekérésre konzolról az **input** utasítás szolgál, ilyen alakban:

n = input("Kérem a nevét: "), ahol az idézőjelek közé írt szöveg megjelenik a képernyőn, n pedig egy karakterlánc típusú változó.

Ha másmilyen típusú adatra van szükségünk, akkor a beérkező adatot konvertálni kell.

egészre, például így: b = int(input("Kérem a másik oldalt: "))
valósra, például így: b = float(input("Kérem a másik oldalt: "))

2. A TEKNŐCGRAFIKA FONTOSABB UTASÍTÁSAI

Általános utasítások	
Feladat	Utasítás
Teknőckezelő utasítások betöltése	from turtle import *
Képernyő törlése, teknőc alaphelyzetbe állítása	reset()
Teknőc elrejtése	hideturtle()
Teknőc megjelenítése	showturtle()

Rajzoló utasítások	
Feladat	Utasítás
Előremegy x képponttal	forward(x)
Hátramegy x képponttal	backward(x)
Ecset felemelése a "papírról", utána nem rajzol	up()
Ecset lerakása a "papírra", utána rajzol	down()
Elfordulás balra x fokot	left(x)
Elfordulás jobbra x fokot	right(x)
Mozgás az x, y koordinátájú pontra	goto(x,y)
Körív rajzolása. Középpontja balra a teknőctől pozitív sugár esetén. Ha a sugár negatív, a középpont a teknőctől jobbra lesz. Az ív kezdőpontja a teknőc kiindulási helyzete.	circle(<sugár>[,<szög>])

Színek kezelése	
Feladat	Utasítás
Háttérszín	bgcolor(<szín>)
Vonalszín	color(<szín>)
Alakzat kitöltő színe	fillcolor(<szín>)
Kitöltés kezdete zárt alakzat (alakzatok) rajzolása Kitöltés vége	begin_fill() <utasítások> end_fill()
<szín> megadása névvel vagy RGB kóddal	"black" vagy "#a0c8f0"

3. VÁLTOZÓK

Fogalma

Az írható-olvasható memória egy része, amire névvel hivatkozhatunk.

Fajtái

- **egyszerű változók**: egész szám, valós szám, karakterlánc, logikai
- **összetett változók:** lista, tuple, szótár, rekord

Értékadások

Feladat	Mondatszerű leírás	Python
b legyen egyenlő c-vel	b := c	b = c
b-t növeljük c-vel	b := b + c	b += c
b-t csökkentjük c-vel	b := b – c	b -= c
b-t szorozzuk c-vel	b := b * c	b *= c
b-t osztjuk c-vel	b := b / c	b /= c
inkrementálás (növelés 1-gyel)	inc(b)	b += 1
dekrementálás (csökkentés 1-gyel)	dec(b)	b -= 1

Típusátalakítás (konvertálás)

- karakterlánccá: karakterlanc = str(szam)
- valós számmá: valos = float(karakterlanc)
- egész számmá: egesz = int(karakterlanc)

Alapműveletek számokkal

Művelet	Mondatszerű leírás	Python	Eredmény
összeadás	5 + 3	5 + 3	8
kivonás	5 – 3	5 – 3	2
szorzás	5 * 3	5 * 3	15
osztás	5 / 3	5 / 3	1.666666667
osztási maradék	5 mod 3	5 % 3	2
maradékos osztás hányadosa	5 div 3	int(5 / 3)	1

Számok további műveletei

Előzőleg be kell tölteni a **math** függvénykönyvtárt: from math import *

Feladat	Python példa	Eredmény
négyzetgyökvonás	a = sqrt(2.25)	1.5
hatványozás (alap: 2, kitevő: 3)	a = pow(2,3)	8.0
logaritmus (alap: 2, szám: 8)	a = log(8,2)	3.0
tízes alapú logaritmus	a = log10(100)	2.0
kerekítés a kisebb egészre	a = floor(2.25)	2
kerekítés a nagyobb egészre	a = ceil(2.25)	3
kerekítés egészre	a = round(2.25)	2
abszolútérték	a = abs(-2.25)	2.25
szinusz[80] (szög radiánban: 3.14/2)	a = sin(3.14/2)	0.99999
inverz szinusz[81] (arkusz szinusz, $sin^{-1}x$)	a = asin(1)	1.57079 (rad)

Karakterek

karakterek	ASCII kódok
0–9	48–57
A–Z	65–90
a–z	97–122

- ASCII kód formájában tárolódnak a memóriában: 1 bájton, 1 db 0–255 egész szám megfelel 1 db betűnek.
- A karakterek sorba rendezhetők, mert a sorrend megegyezik a kódjaik sorrendjével. A karakterek rendezhetősége miatt a karakterláncok is rendezhetők.
- Az ékezetes betűk nincsenek a sorrend megfelelő helyén.
- Kód karakterré alakítása: karakter = **chr**(kód)
- Karakter kóddá alakítása: kód = **ord**(karakter)

80 Hasonlóan kell használni a koszinusz (cos) és tangens (tan) függvényeket.

81 Hasonlóan kell használni az inverz koszinusz (acos) és inverz tangens (atan) függvényeket.

Karakterláncok

A karakterlánc karakterek listája

A karakterláncokat idézőjelek vagy aposztrófok közé kell írni: "szemüveg" vagy 'szemüveg'. A kétféle megadás teljesen egyenértékű.

A példákban strin = "szemüveg", és sz = "␣␣Egy eresz␣␣"[82].

Feladat	Példa	Eredmény
Összefűzés (konkatenáció)	"nap" + strin + "es"	'napszemüveges'
Karakterlánc hossza	len(sz)	13
Hivatkozás egy karakterre	strin[4]	'ü'
Szeletelés balról	strin[4:]	'üveg'
Szeletelés jobbról	strin[:4]	'szem'
Szeletelés középről	strin[3:6]	'müv'
Karaktercsere	sz.replace('e','a')	'␣␣Egy arasz␣␣'
Szóközök, sorvége jelek eltávolítása	sz.strip()	'Egy eresz'
Karakter pozíciója adott hely után	sz.index("e,4)[83]	6
Karakterlánc bontása listaelemekre	sz.split()[84]	['Egy', 'eresz']
Csupa nagybetűssé alakít	sz.upper()	'␣␣EGY ERESZ␣␣'
Csupa kisbetűssé alakít	sz.lower()	'␣␣egy eres+'
Karakterlánc keresése karakterláncban (sz -ben keresi a mögé írt karakterláncot)	sz.find('Bubó') sz.find('re')	-1 7

Logikai változók

- Értékük csak **igaz** vagy **hamis** lehet.
- A Pythonban az angol megfelelőjüket használjuk, nagy kezdőbetűkkel: **False**, **True**.
- Logikai érték keletkezhet értékadással, például: b = False
- vagy összehasonlítás eredményeként, például: c = (x > y).

---

82 A ␣ karakter a szóközt jelöli.

83 Ha a második paramétert elhagyjuk, a karakterlánc elejétől keres.

84 Paraméterként megadható az elválasztó karakter. Például sz.strip(',') vesszők mentén bontja a karakterláncot.

4. VEZÉRLÉSI SZERKEZETEK

Vezérlési szerkezetek felhasználása általában

Az eljárások és függvények adják meg a program durva szerkezetét. A szekvenciák, elágazások és ciklusok a finomszerkezetet határozzák meg.

- Szekvencia: utasítások sorrendi végrehajtása.
- Elágazások (szelekciók): esetszétválasztás létrehozása. Feltételes végrehajtás.
- Ciklusok (iterációk): programrészlet ismétlése.
  - Számláló ciklust használunk, ha az ismétlések száma már a program írásakor tudható.
  - Feltételes ciklust használunk, ha az ismétlések száma csak futás közben dől el.
- Alprogramok: Program részekre bontása áttekinthetőség, gyorsabb módosítás és javítás, könnyebb tesztelhetőség, feladatok szétosztása miatt.
  - eljárások: nincs visszatérési érték
  - függvény: van visszatérési érték

Néhány alapvető vezérlési szerkezet kódolása

Vezérlési szerkezet	Mondatszerű leírás	Python
Elágazás egy irányban	**Ha** x = y **akkor**<utasítások>	**if** x == y: <utasítások>
Elágazás két irányban	**Ha** x = y **akkor** <utasítások1> **különben** <utasítások2> **Elágazás vége**	**if** x == y: <utasítások1> **else:** <utasítások2>
Elágazás több irányban	**Elágazás** x = y **esetén** <utasítások1> x > y **esetén** <utasítások2> ... **egyébként** <utasítások3> **Elágazás vége**	**if** x == y: <utasítások1> **elif** x > y: <utasítások2> ... **else:** <utasítások3>
Számláló ciklus kezdőérték = 0, növekmény = 1	**Ciklus** i:=0-**tól** 100-**ig** <Ciklusmag> **Ciklus vége**	**for** i in **range**(101): <Ciklusmag>
Számláló ciklus kezdőértékkel, növekmény = 1	**Ciklus** i:=1-**tól** 100-**ig** <Ciklusmag> **Ciklus vége**	**for** i in **range**(1,101): <Ciklusmag>
Számláló ciklus növekménnyel	**Ciklus** i:=0-**tól** 100-**ig** 10-**esével** <Ciklusmag> **Ciklus vége**	**for** i in **range**(0,110,10): <Ciklusmag>
Feltételes ciklus, elöltesztelő	**Ciklus amíg** x = y <Ciklusmag> **Ciklus vége**	**while** x == y: <Ciklusmag>

- *x = y, x == y, x > y*: az adott feladatnak megfelelő feltételek logikai kifejezések alakjában
- *<Ciklusmag>*: ismétlendő utasítások
- *<utasítások>, <utasítások1>, <utasítások2>, <utasítások3>*: adott feltételek esetén végrehajtandó utasítások
- *i*: ciklusváltozó, a ciklus minden lefutásakor változik az értéke
- *0, 1*: kezdőérték, innen kezdődik a számlálás
- *100*: végérték, eddig tart a számlálás
- *10*: növekmény, ennyivel növekszik a ciklusváltozó minden egyes lefutáskor
- *101, 110*: a végértéknek ennél kisebbnek kell lennie

Feltételek megadása

Művelet	Algoritmus-leíró	Python
egyenlő	a = b	a == b
nem egyenlő	a <> b	a != b
nagyobb	a > b	a > b
kisebb	a < b	a < b
nagyobb vagy egyenlő	a ≥ b	a >= b
kisebb vagy egyenlő	a ≤ b	a <= b

Összetett feltételek alkotása logikai műveletekkel

A	B	A ∧ B	A ∨ B	¬ A
igaz	igaz	igaz	igaz	hamis
igaz	hamis	hamis	igaz	hamis
hamis	igaz	hamis	igaz	igaz
hamis	hamis	hamis	hamis	igaz

Művelet	Algoritmus-leíró	Python
ÉS	x >= 1 ∧ x <= 8 1 < = x <= 8	x >= 1 **and** x <= 8 1 < = x <= 8
VAGY	x >= 1 ∨ x <= 8	x >=1 **or** x <= 8
NEM	¬(x >=1)	**not**(x >=1)

műveleti sorrend: zárójeles kifejezés > NEM > ÉS > VAGY

Alprogramok kódolása

Feladat	Mondatszerű leírás	Python
Eljárás paraméter nélkül	**Eljárás** tedd <utasítások> **Eljárás vége**	**def** tedd(): <utasítások>
Eljárás paraméterrel	**Eljárás** tedd2(a:egész) <utasítások> **Eljárás vége**	**def** tedd2(a): <utasítások>
Függvény paraméter nélkül	**Függvény** add : egész <utasítások> add := vissza **Függvény vége**	**def**<név>(): <utasítások> **return** vissza
Függvény paraméterrel	**Függvény** add2(a,b:egész): egész <utasítások> add2 := vissza **Függvény vége**	**def** add2(a,b): <utasítások> **return** vissza
Eljárás hívása paraméter nélkül	tedd	tedd()
Eljárás hívása paraméterrel	tedd2(15)	tedd2(15)
Függvény hívása, eredménye egy változóba kerül	e = add e = add2(15,10)	e = add() e = add2(15,10)
Függvény hívása, eredménye egy utasítás paraméterébe kerül	utasítás(add) utasítás(add2(15))	print(add()) print(add2(15,10))

- *tedd, tedd2, add, add2*: tetszőlegesen megválasztható nevek
- *egész*: típus megnevezése. Lehet még logikai, valós, karakter vagy karakterlánc (string).
- *a, b*: paraméterek neve, tetszőlegesen megválasztható. Lehet belőle egy vagy több. Utóbbi esetben vesszővel kell elválasztani őket egymástól.

Paraméterátadások

- **Érték szerinti:** Az alprogram lemásolja a főprogram által átadott változót, így az alprogram működése nem befolyásolja a főprogramban található változó értékét.
- **Cím szerinti:** A főprogram és alprogram érintett változója ugyanazon a memóriaterületen osztozik, így az alprogram módosítja a főprogram változójának tartalmát. Az alprogram paramétereként csak változónevet lehet megadni.

A Pythonban minden egyszerű változó érték szerinti, minden összetett változó cím szerinti paraméterátadással kerül át a főprogramból az alprogramba.

Változók hatóköre

- **Lokális**: Az alprogram aktuálisan futó példányában érvényes. Az alprogramban deklaráljuk.
- **Globális**: A teljes programban érvényes. A főprogramban deklaráljuk.

Ha a globális és az érvényes lokális változónak ugyanaz a neve, a lokális változó érhető el.

Ha alprogramban akarunk globális változót létrehozni, alkalmazzuk a változó neve előtt a **global** kulcsszót.

Teljes lefedés elve

A programunkat úgy kell tesztelni (azaz a teszteseteket úgy kell kiválasztani), hogy minden elágazási ág kipróbálására sor kerüljön.

Rekurzió

Olyan eljárások vagy függvények, amik önmagukra (önmaguk egy másik példányára) hivatkoznak.

5. ÖSSZETETT VÁLTOZÓK, ADATSZERKEZETEK

Összetett változók

- **lista**: Azonos nevű, sorszámozott (indexelt) változók. A sorszámozás 0-tól indul. Például: *nev[0], nev[1], nev[2],...*
- **tuple**: Olyan lista, aminek adatai nem módosíthatók a létrehozás után. Például: *honap[0], honap[1], honap[2],...*
- **szótár**: Azonos nevű, kulccsal megkülönböztetett, sorrend nélküli elemek. A kulcs típusa tetszőleges. Például: *megfelel['fa'], megfelel['az'], megfelel['vas']*
- **rekord**: Azonos nevű (és ha van, akkor indexű), összetartozó adatok, amiket egyben kezelhetünk, és mezőnevekkel különböztetünk meg egymástól. Például: *auto.rendszam, auto.marka, auto.szin*. Vagy *auto[1].rendszam, auto[1].marka, auto[1].szin.*
- **fájl**: Összetartozó adatok rendezett együttese a háttértárolón.
- **egyszerű szöveges fájl**: formázatlan szöveget tartalmaz.

Alapvető listaműveletek

Feladat	Példa
Lista létrehozása elemekkel	nev = ["Anna", "Bea", "Cecília"]
Tuple létrehozása	honap = ("január", "február", "március")
Üres lista létrehozása	nev = [ ]
Lista bővítése a végén egy elemmel	nev.append("Glázser Bozsó")
Lista létrehozása egyforma elemekkel	szuletes = [0]*40
Listaelem megváltoztatása	nev[0] = "Nyomasek Bobó"
Listaelem vagy tuple-elem kiíratása	print(nev[0])
Teljes lista vagy tuple kiíratása	print(honap)
Az index mint listaelem	honap[szuletes[29]]

Lista mátrix feltöltése adatokkal	table = [ ]
	for i in range(0,20):
	    row = [ ]
	    for j in range(0,20):
	        row.append(adat_amit_beviszünk)
	    table.append(row)

További listaműveletek

A példában n = [8, 16, 15, 4, 42, 23].

Feladat	Példa	Eredmény
rendezés	n.sort()[85]	[4, 8, 15, 16, 23, 42]
sorrend megfordítása	n.reverse()	[23, 42, 4, 15, 16, 8]
elem indexe	n.index(42)	4
elem eltávolítása	n.remove(42)	[8, 16, 15, 4, 23]
adott indexű elemek eltávolítása	del n[1:3]	[8, 4, 42, 23]
valódi másolat készítése a listáról[86]	d = n.copy()	[8, 16, 15, 4, 42, 23]
eleme-e a listának	8 in n 0 in n	True False

Szótárműveletek

A példában n = {'0': "0000",'1': "0001", '2': "0010"}.

Feladat	Példa	Eredmény
elem megváltoztatása, értékadás	table [4,4] = 'X'	
szótárban használt kulcsok listája	n.keys()	['0', '1', '2']
szótárban használt értékek listája	n.values()	['0000', '0001', '0010']
szótár kulcs-érték tuple-jeinek listája	n.items()	[('0', '0000'), ('1', '0001'), ('2', '0010')]
szótárról valódi másolat készítése	d = n.copy()	{'0': '0000', '1': '0001', '2': '0010'}
adott kulcsú elem eltávolítása	del n['1']	{'0': '0000', '2': '0010'}
tartalmazza-e a kulcsot a szótár	'A' in n '2' in n	False True
ha nincs a kulcs a szótárban, írja ki a 2. paraméterben megadott alternatívát	n.get('A','') n.get('2','')	'' '0010'

---

85 Vigyázat! Ékezetes karakterláncokra nem működik helyesen

86 A d = n értékadással az n lista csak elérhető lesz d néven is. Tehát a d-n keresztül ugyanazt a listát érjük el, mint n-en keresztül. Azaz, ha d-t módosítjuk, n is módosul. Ha a copy utasítást használjuk, a két lista egymástól független lesz. (Részletesebben lásd a Cím szerinti paraméterátadás témakört.)

Adatszerkezetek összes adatának bejárása

```
honap = ("január", "február", "március")
change = {'C': "1100",'D': "1101",'E': "1110",'F': "1111"}
amoba = [['.','X','O'],
         ['X','O','O'],
         ['X','.','.']]
table = {(5,5) : 'O', (4,5) : 'O', (5,6) : 'X', (4,4) : 'X'}
```

Feladat	Példa
Lista vagy tuple elemeinek bejárása	for i in honap: print(i, end=" ")
Lista vagy tuple indexeinek és elemeinek bejárása	for i in range(len(honap)): print(i+1, honap[i])
Szótár kulcsainak és elemeinek bejárása	for key in change: print (key, change[key])

Lista mátrix vagy tuple mátrix bejárása	for i in range(0,3):
	   for j in range(0,3):
	      print(amoba[i][j], sep="", end="")
	   print()

Szótár mátrix bejárása	for i in range(10):
	   for j in range(10):
	      print(table.get((i,j), '.'), end='')
	   print()

Fájl bejárása soronként	re = open("adat.txt", 'r')
	line = re.readline()
	while line != "":
	   line = line.strip()
	   <beolvasott adat feldolgozása>
	   line = re.readline()
	re.close()

Rekordok használata

Használatának lépései	Mondatszerű leírás	Python
1. Rekordtípus létrehozása	Frops = rekord sz1,n1,sz2,n2 : egész op : karakter rekord vége	class Frops(): pass
2. Rekord létrehozása	buff : Frops	buff = Frops()
2. Rekordtömb létrehozása	frop() : Frops	frop = [ ] for i in range(10): frop.append(Frops())
3. Értékadás azonos típusú rekordok közt	buff = frop(1)	buff = frop[1]
3. Értékadás mezőnként	buff.szaml1 = 8 frop(1).nev2 = 16	buff.szaml1 = 8 frop[1].nev2 = 16

Egyszerű szövegfájl kezelése

Feladat	Python példa
megnyitás olvasásra	re = open("igeny.txt","r')
sor beolvasása	line = re.readline()
fájl zárása	re.close()
megnyitás írásra	wr = open("fracts.txt","w')
írás fájlba	wr.write("András\n")
megnyitás bővítésre	wr = open("fracts.txt","a')
Konzol puffer címe	oldout = sys.stdout
Konzol puffer átirányítása	sys.stdout = wr

- Ha egy létező fájlt írásra megnyitunk, a régi fájl tartalma elvész. Ezért egy írásra nyitott fájlba addig tudunk írni, míg be nem zárjuk. Utána már csak bővíteni lehet.
- Ha egy nemlétező fájlt írásra megnyitunk, az utasítás először létrehozza a fájlt.

6. PROGRAMOZÁSI TÉTELEK

A programozási tételek olyan algoritmusok típusfeladatokra, amiknek helyessége *bizonyítható*.

Sorozathoz egyetlen elemet rendelő tételek

Tétel neve	Feladata
Összesítés	Sorozat (pl. lista) elemeinek összesítése
Feltételes összesítés	Sorozat adott tulajdonságú elemeinek összesítése
Megszámlálás	Sorozat adott tulajdonságú elemeinek száma (darab)
Maximumkiválasztás	Sorozat elemei közül a legnagyobb kiválasztása
Feltételes maximumkiválasztás	Sorozat adott tulajdonságú elemei közül a legnagyobb kiválasztása
Eldöntés	Van-e a sorozatban adott tulajdonságú elem? (igaz/hamis)
Lineáris keresés	Van-e a sorozatban adott tulajdonságú elem, és ha igen, akkor hol? (pl. Melyik indexű az?)

Sorozathoz sorozatot rendelő tételek

Tétel neve	Feladata
Sorozatszámítás	Egy sorozat (pl. lista) elemenkénti átalakításával egy másik sorozat előállítása. Például: adottak egy listában munkák. Előállítandó egy lista az értük járó bérekkel.
Kiválogatás	Egy sorozatból az adott tulajdonságú elemek átmásolása egy másik sorozatba. Például: egy autókat tartalmazó listából a 10 évesnél idősebb autókról készüljön egy lista.
Rendezés	Egy sorozat elemeinek sorrendbe rakása. Például: számok növekvő vagy csökkenő sorrendbe, karakterláncok ábécésorrendbe rakása.

Összesítés

```
Adott: A(N)
1) s := 0
2) Ciklus i := 0-től N-1-ig
3)      s := s + A(i)
4) Ciklus vége
5) Ki: s
6) Ki: s/n
```

Feltételes összesítés

```
1) s := 0
2) Ciklus i := 0-től N-1-ig
3)      Ha T(A(i)) akkor s := s + A(i)
4) Ciklus vége
5) Ki: s
```

Megszámlálás

```
Adott: A(N) és T
1) db := 0
2) Ciklus i := 0-től N-1-ig
3)    Ha T(A(i)) akkor
4)          db := db + 1
5) Ciklus vége
6) Ki: db
```

Sok érték megszámlálása

```
1) db() := 0
2) Ciklus i := 0-től N-1-ig
3)    db(A(i)):= db(A(i))+1
4) Ciklus vége
5) Ki: db()
```

Maximumkiválasztás

```
Adott: A(N), és egy sorrend
1) maxertek := A(0)
2) maxhely := 0
3) Ciklus i := 1-től N-1-ig
4)     Ha maxertek<A(i) akkor
5)         maxertek := A(i)
6)         maxhely := i
7)     elágazás vége
8) Ciklus vége
9) Ki: maxhely, maxertek
```

```
Adott: A(N), és egy sorrend
1) maxertek := -∞
2) maxhely := -1
3) Ciklus i := 0-től N-1-ig
4)     Ha maxertek<A(i) akkor
5)         maxertek := A(i)
6)         maxhely := i
7)     elágazás vége
8) Ciklus vége
9) Ki: maxhely, maxertek
```

Eldöntés

```
1) i := 0
2) Ciklus amíg i <= N-1 ÉS NEM(T(A(i)))
3)     i := i + 1
4) Ciklus vége
5) VANELEM := (i <= N-1)
```

Lineáris keresés

```
1) i := 0
2) Ciklus amíg i <= N-1 ÉS NEM(T(A(i)))
3)     i := i + 1
4) Ciklus vége
5) Ha i <= N-1
6)    akkor Ki: i
7)    különben Ki: „Nincs megfelelő elem.”
8) Elágazás vége
```

Eldöntés mátrixban

```
Adott: A(N,M), egy T tulajdonság
1) VANELEM := hamis
2) i := 0
3) Ciklus amíg i <= N-1 és NEM(VANELEM)
4)      j := 0
5)      Ciklus amíg j <= M-1 és NEM(T(A(i,j)))
6)          j := j + 1
7)      Ciklus vége
8)      Ha j <= M-1 akkor VANELEM := igaz
9)                  különben i := i+1
10)     Elágazás vége
11)Ciklus vége
```

Lineáris keresés mátrixban

```
1) VANELEM := hamis
2) i := 0
3) Ciklus amíg i <= N-1 és NEM(VANELEM)
4)      j := 0
5)      Ciklus amíg j <= M-1 és NEM(T(A(i,j)))
6)          j := j + 1
7)      Ciklus vége
8)      Ha j <= M-1 akkor VANELEM := igaz
9)                  különben i := i+1
10)     Elágazás vége
11)Ciklus vége
12)Ha VANELEM
13)    akkor Ki: i,j
14)    különben Ki: „Nincs ilyen elem.”
15)Elágazás vége
```

Sorozatszámítás

```
1) Ciklus i := 0-től N-1-ig
2)    B(i) := f(A(i))
3) Ciklus vége
```

Kiválogatás

```
1) Ciklus i := 0-től N-1-ig
2)      Ha T(A(i)) akkor
3)          B() következő eleme A(i)
4) Ciklus vége
```

Buborékos rendezés

```
1) Ciklus i := 0-től N-2 -ig
2)      Ciklus j := N-1-től j>i-ig (-1)-vel
3)          Ha a(j-1) > a(j) akkor
4)              a(j-1), a(j) := a(j), a(j-1)
5)      Ciklus vége
6) Ciklus vége
```

Összetett rendezés

```
1) Ciklus i := 0-től N-2-ig
2)      Ciklus j := N-1 -től j>i -ig (-1) -vel
3)          Ha a(j-1).elsod*szor + a(j-1).masod
            > a(j).elsod*szor + a(j).masod akkor
4)              a(j-1), a(j) := a(j), a(j-1)
5)      Ciklus vége
6) Ciklus vége
```

7. GRAFIKUS ALKALMAZÁSOK

Keretprogram

```
from tkinter import *
ablak = Tk()
<ide jön a főprogram>
ablak.mainloop()
```

Ablak beállítások

Szöveg a címmezőben (Amőba): ablak.title("Amőba")
Ablak legkisebb mérete (600 x 400 képpont): ablak.minsize(width = 600, height = 400)
Alapértelmezett betűtípus megváltoztatása (11 pont méretű, Arial betűtípusra):

```
ablak.option_add("*font", "Arial 11")
```

Vezérlők fajtái

Elnevezés	Osztály	Feladat
Parancsgomb	Button	Rákattintva esemény váltható ki.
Szövegmező	Entry	Egysoros, adatbevitelre és megjelenítésre is alkalmas.
Címke	Label	Magyarázó szövegek, feliratok elhelyezéséhez.
Jelölőnégyzet	Checkbutton	Kétállapotú vezérlő, aminek állapotai között kattintással kapcsolhatunk (kijelölt/nem kijelölt).
Rádiógomb	Radiobutton	Többet használunk együtt, amiből egyszerre csak egy aktív. Egy változó lehetséges állapotait mutatják.
Üzenet	Message	Üzenetek megjelenítésére használjuk. Többsoros címke.
Keret	Frame	Vezérlők összefogására használható. A keretben a vezérlők elrendezhetők, majd a keret az ablakban akár másféleképpen.
Listamező	Listbox	A felhasználónak felajánlott választéklista. Általában valamilyen dobozban, egymás alatt elhelyezkedő szövegek formájában.
Szövegdoboz	Text	Formázott, többsoros szöveg.
Gördítősáv	Scrollbar	Általában más vezérlőkhöz (szövegdoboz, listamező, vászon) kapcsolva használjuk a nem látható részletek megjelenítésére.
Üzenetablak	Toplevel	A program fontos üzeneteit szoktuk így megjeleníteni.
Menügomb	Menubutton	Legördülő menük.
Menü	Menu	Menük (pl. legördülő menük) egybefoglalása.
Vászon	Canvas	Képek megjelenítésére, rajzolásra szolgáló felület.
Skála	Scale	Egy változó értékét lehet csúszka segítségével beállítani.

Vezérlők általában

vezérlőnév = Típus(szülőablak [, további paraméterek])
textBox1 = Entry(ablak)
cmdButton1 = Button(ablak,text="Új szó", width = 15, command=ujszo)

Vezérlők fontosabb általános paraméterei	
Paraméter	Mire használjuk?
text="Felirat"	vezérlőn megjelenő felirat
command=utasítás	vezérlőre kattintáskor végrehajtott utasítás
width = 15	szélesség, ahol értelme van, ott karakterekben, egyébként képpontokban
height = 5	magasság, ahol értelme van, ott karakterekben, egyébként képpontokban

Vezérlő paramétereinek utólagos módosítása: vezérlőnév.configure(text = "Új szöveg")

Vezérlők elhelyezése, megjelenítése, megszüntetése

Megjelenítési módok		
Mód	Elhelyezés	Megszűntetés
Egyszerű elhelyezés	vezérlőnév.pack()	vezérlőnév.pack_forget()
Rácsra igazított	vezérlőnév.grid(row = 1, column = 2)	vezérlőnév.grid_forget()
Koordinátákkal megadott	vezérlőnév.place(x = 200, y = 100)	vezérlőnév.place_forget()

Megjelenítési paraméterek		
Feladat	Pack mód	Grid mód
igazítás balra	side = LEFT	sticky = W
igazítás jobbra	side = RIGHT	sticky = E
igazítás fel	side = TOP	sticky = N
igazítás le	side = BOTTOM	sticky = S
bal és jobb margó	padx = 10	
felső és alsó margó	pady = 10	

Rácsos megjelenítés esetén a sorok és oszlopok beállíthatók az alábbi módon:

```
ablak.columnconfigure(i, pad = 15)
ablak.rowconfigure(i, pad = 10)
```

Az első paraméter a módosítandó oszlop (column) vagy sor (row) sorszáma, a második paraméter a módosítandó tulajdonság (esetünkben a margó – pad) és annak értéke. A példa első sora az i. oszlop vízszintes (bal és jobb) margóit állítja 15 képpontra, a példa második sora az i. sor függőleges (alsó és felső) margóit állítja 10 képpontra.

Szövegmező

Törlés az első paramétertől a másodikig: szövegmezőnév.delete(0,END)

A második paraméterként megadott karakterlánc beszúrása az első paraméterben megadott helytől indulva: szövegmezőnév.insert(0,"ezt jeleníti meg")

Karakterlánc kiolvasása szövegmezőből: változó = szövegmezőnév.get()

Szövegdoboz

A fontosabb utasítások ugyanazok, mint a szövegmezőnél, de a paramétereket *sor.oszlop* alakban kell megadni. Például: szövegdoboznév.delete("1.0", "end"). A sorok számozása 1-től, az oszlopoké 0-tól indul.

Jelölőnégyzet

Létrehozás

```
chk = IntVar()
chkButton = Checkbutton(szülőablaknév, \
 text="Felirat", variable = chk, command=parancs)
```

Az állapota a változóból *(chk)* nyerhető ki a **get** utasítással: 1, ha bekapcsolt, 0, ha kikapcsolt.

```
if chk.get() == 1:
```

Rádiógomb-csoport

Létrehozás

```
rb = StringVar()
rButton1 = Radiobutton(szülőablak, value='0', \
 text="0%", variable = rb,command = parancs)
rButton2 = Radiobutton(szülőablak, value='0.1',\
 text="10%", variable = rb,command = parancs)
```

Az összetartozó rádiógombok változója (**variable**) ugyanaz *(rb)*. Az értékük (**value**) által lehet a választást tetten érni, ezért azok szükségképpen különbözőek.

A kiválasztott gombhoz tartozó érték a változóból olvasható ki a **get** utasítással.

```
szl = 1 - float(rb.get())
```

Listamező	
Feladat	**Példakód**
Létrehozás 3 sor magasságban	lista = Listbox(ablak, height=3)
Listamező kiválasztásakor végrehajtandó parancs *(fut)*	lista.bind("<<ListboxSelect>>", fut)
Listamező elemeinek beszúrása (END = utolsó elem után)	lista.insert(END, "kő", "papír")
Listamező aktív (kijelölt) sorának beállítása	lista.select_set(0)
Kiválasztott elemek sorszámai listába *(kiv)* kerülnek.	kiv = lista.curselection()

Rajzvászon

LÉTREHOZÁSA

A példában *vaszon* néven hozunk létre egy *420* képpont szélességű és *310* képpont magasságú, fehér háttérszínű (**bg**) vásznat a *szülőablak*ban.

```
vaszon = Canvas(szülőablak, width=420, height=310, bg="white")
```

A rajzvászon koordinátarendszerének kezdőpontja a bal felső sarok, tehát jobbra és lefelé növekednek a koordináták.

KÉP BETÖLTÉSE A MEMÓRIÁBA

```
fotónév = PhotoImage(file=fájlnév)
```

A **fotónév**vel tudunk hivatkozni a további felhasználáskor a képre. A **fájlnév**ben szerepelhet az elérési út is, ha a fájl másik mappában található, mint a programfájl. Például: "d:\\works\\kep.png".

KÉP VÁSZONRA HELYEZÉSE

```
név = vaszon.create_image(x, y, image=fotónév)
```

A fotónév egy, a memóriába betöltött kép neve, **x, y** a fotó középpontja idekerül a vásznon.

SZAKASZ RAJZOLÁSA

```
vaszon.create_line(x1,y1,x2,y2,width=4,fill="green")
```

x1, y1 – a szakasz egyik végpontja, **x2, y2** – a másik végpont, **width** a vonal vastagsága, **fill** a vonal színe.

TÉGLALAP RAJZOLÁSA

```
vaszon.create_rectangle(x1,y1,x2,y2, width=2, \
 fill="dark grey", outline="dark blue")
```

x1, y1 a téglalap bal felső sarka, **x2, y2** a téglalap jobb alsó sarka, **width** a vonal vastagsága, **fill** a kitöltés színe, **outline** a határolóvonal színe.

ELLIPSZIS RAJZOLÁSA

```
vaszon.create_oval(x1,y1,x2,y2, width=2,\
 fill="red",outline="black")
```

x1, y1 a befoglaló téglalap bal felső sarka, **x2, y2** a befoglaló téglalap jobb alsó sarka, **width** a vonal vastagsága, **fill** a kitöltés színe, **outline** a határolóvonal színe.

ELLIPSZIS-CIKK RAJZOLÁSA

```
vaszon.create_arc(x1,y1,x2,y2,outline="black",\
 width=2,fill="red",start=90,extent=180)
```

x1, y1 a befoglaló téglalap bal felső sarka, **x2, y2** a befoglaló téglalap jobb alsó sarka, **width** a vonal vastagsága, **fill** a kitöltés színe, **outline** a határolóvonal színe, **start** a cikk kezdetének szöge, **extent** a cikk középponti szöge az óramutató járásával ellentétes irányban. Mindkét szöget fokokban kell megadni. A kezdő szög 0 foka a keleti (jobbra) irány.

SOKSZÖG RAJZOLÁSA

```
vaszon.create_polygon(x1, y1, x2, y2, …, xn, yn, \
 width=2,fill="green", outline="black")
```

x1, y1 a sokszög első pontja, **x2, y2** a sokszög második pontja, **xn, yn** a sokszög utolsó pontja, **width** a vonal vastagsága, **fill** a kitöltés színe, **outline** a határolóvonal színe. Az utasítás az első és utolsó pontot automatikusan összeköti.

ALAKZAT KOORDINÁTÁINAK UTÓLAGOS VÁLTOZTATÁSA

```
vaszon.coords(alakzatnév, x1, y1, x2, y2)
```

Animációkészítéshez különösen hasznos utasítás a **coords**. Alakzatnév a megváltoztatandó alakzat neve. **x1, y1** a befoglaló téglalap új bal felső sarka, **x2, y2** a befoglaló téglalap új jobb alsó sarka.

Időzítés, ismételt művelet-végrehajtás

```
ablak.after(100,indulj)
```

Egyszer végrehajtja a második paraméterként megadott utasítást, az első paraméterben megadott ezredmásodperc múlva.

Rekurzív hívással megadott időközönként ismétlődő végrehajtás érhető el.

```
def indulj():
    <ismétlődően elvégzendő műveletek>
    ablak.after(100,indulj)
```

Eseményhez utasítás rendelése

```
vaszon.bind("<Button-1>", kattint)
```

Egy eseményhez végrehajtandó eljárást rendelhetünk a **bind** utasítással. Első paraméter az esemény, második a végrehajtandó eljárás. A lehetséges eseményeknek a Tkinter dokumentációjából lehet utánanézni. A **<Button-1>** az egér bal gombjával történő kattintást jelenti.

Az eseményhez tartozó eljárást úgy kell létrehozni, hogy legyen egy paramétere, amiben az esemény adatai tárolódnak. Ezt akkor is meg kell tenni, ha a továbbiakban nem használjuk ezt a paramétert semmire:

```
def kattint(event):
```

Esemény adatainak feldolgozása

Az eseményhez rendelt eljárás paraméterében találhatók az esemény körülményei, amik az objektum tulajdonságaiként érhetőek el. Például <Button-1> esetén a koordináták, ahol az egér volt a kattintás pillanatában **event.x** és **event.y**.

```
def kattint(event):
    x = int((event.x-korr)/mmeret)
    y = int((event.y-korr)/mmeret)
```

8. OBJEKTUMORIENTÁLT PROGRAMOZÁS

Alapfogalmak

Objektum, példány: a valóság modellezendő részének egy logikailag jól elkülöníthető része. Részei a tagváltozók és a metódusok.

Osztály, objektumosztály: sablon, ami alapján létrehozzuk az objektumokat.

Példányosítás: az a folyamat, amikor a sablon alapján objektumot (példányt) hozunk létre.

Tagváltozó: az objektum egy önálló adattárolásra alkalmas része, amire névvel hivatkozhatunk.

Metódus: az adott objektumon műveletet végző függvény (tagfüggvény) vagy eljárás (tageljárás) összefoglaló neve.

Konstruktor: az objektum létrehozásakor automatikusan lefutó metódus. Leggyakrabban a tagváltozók kezdőértékének beállítására szolgál.

Statikus tagváltozó: olyan adattag, ami az osztályon belül közös, tehát az osztály minden példánya használhatja. Alkalmas lehet például az objektumok közötti adatcserére. Példányosítás nélkül is működik.

Statikus metódus: olyan metódus, ami egy osztályon belül közös, így minden objektum ugyanazt használja. Alkalmas a statikus tagváltozók kezelésére.

Virtuális metódus: a többalakúsághoz kapcsolódó fogalom. Az ősosztályban hozzuk létre azzal a céllal, hogy a csak itt található metódus a leszármazott osztálybeli másik metódust hívja, ha a leszármazott osztályon keresztül történik a meghívása. (S az ősosztálybeli metódust, ha az ősosztályon keresztül történik a meghívása.)

A módszer főbb jellemzői

Egységbezárás: egy objektum adatait ugyanazon objektum metódusai kezelik, együtt egy egységet képeznek.

Öröklés: egy osztályból létrehozhatunk egy másikat, ami az ősosztály (szülő osztály) tagváltozóival és metódusaival rendelkezni fog. Az örökölt tagváltozókat és metódusokat a leszármazott osztályban (gyermekosztályban) bővíthetjük újakkal, vagy megváltoztathatjuk őket újra deklarálással.

Hierarchia: a leszármazott osztályok is lehetnek egyben ősosztályok, így kialakul az osztályok hierarchiája.

Többalakúság: az osztályok hierarchiájában az azonos nevű metódusok végezhetnek eltérő feladatokat, azaz lehet más a kódjuk.

Objektumorientált program tervezése

- A feladat szövege alapján az osztályok felismerése
- az esetleges hierarchiák felismerése
- majd osztályonként a tagváltozók és metódusok megadása, ha statikus, annak a feltüntetése

Kódolás tervezése

- **Szempontok**: folyamatosan tesztelhető legyen → egymásra épülési sorrend → könnyebbtől a nehezebb felé
- **Osztályok közötti sorrend**: ami nem tartalmaz másikat → szülő → gyermek → Control
- **Sorrend egy osztályon belül**: Osztály → metódusok
- **Metódusokon belül a sorrend**: konstruktor → alapvető adatmegjelenítő → alapvető adatbekérő → műveletet végző → egyéb (pl. fájlkezelés)

Szokások a nevek adásában

- **Osztályok neve** nagybetűvel kezdődjön. Legyen főnév, többes számban.
- **Objektumok neve** legyen főnév, egyes számban, kisbetűkkel.
- **Metódusok neve** legyen ige, kisbetűvel.

Kódolás

- Minden tagváltozó bárhonnan hozzáférhető.
- A tagváltozókat nem kell deklarálni, az első használatkor létrejönnek.
- Minden metódus virtuális.

Feladat	Megvalósítás

osztály
 	
konstruktor
 	
tagfüggvény
 	
tageljárás
	class Szulo():
	
	__init__(self):
	...
	
	def tagfuggveny(self):
	...
	return vissza
	
	def tageljaras(self, a, b):
	...

leszármazott osztály	class Gyermek(Szulo): ...
statikus metódus	def tagelj(a): ...
példányosítás	peldany = Szulo()
példányeljárás hívása	peldany.tageljaras(100,10)
példányfüggvény hívása	a = peldany.tagfuggveny()
statikus metódus hívása	Szulo.tagelj(10);

objektumlistához alkalmas osztály	class Listazhato(object):
	def __init__(self, number):
	self.number = number

objektumlista létrehozása	lista = [ ] lista.append(Listazhato(0))

FÜGGELÉK

„TÖRTEK" ÉRETTSÉGI FELADAT KÓDJA

```
import sys

def egesz_e(sz,n):
    if sz % n == 0:
        print(int(sz/n))
    else:
        print("Nem egész.")

def lnko(a,b):
    if a==b:
        return a
    elif a<b:
        return lnko(a,b-a)
    else:
        return lnko(a-b,a)

def lkkt(a,b):
    return a*b/lnko(a,b)

def szoroz(sz1,n1,sz2,n2):
    esz,en = sz1*sz2,n1*n2
    if esz % en == 0:
        print("%d/%d * %d/%d = %d/%d = %d" % \
              (sz1,n1,sz2,n2,esz,en,esz/en))
    else:
        print ("%d/%d * %d/%d = %d/%d = %d/%d" % \
               (sz1,n1,sz2,n2,esz,en,esz/lnko(esz,en),en/lnko(esz,en)))

def osszead(sz1,n1,sz2,n2):
    en = lkkt(n1,n2)
    bsz1,bsz2 = sz1*en/n1,sz2*en/n2
    esz=bsz1+bsz2
    if esz % en == 0:
        print("%d/%d + %d/%d = %d/%d + %d/%d = %d/%d = %d" % \
              (sz1,n1,sz2,n2,bsz1,en,bsz2,en,esz,en,esz/en))
```

```
    else:
        print ("%d/%d + %d/%d = %d/%d + %d/%d = %d/%d = %d/%d" % \
                (sz1,n1,sz2,n2,bsz1,en,bsz2,en,esz,en, \
                 esz/lnko(esz,en),en/lnko(esz,en)))

print("1. feladat")
print ("Számláló:")
szaml1 = int(input())
print ("Nevező:")
nev1 = int(input())
egesz_e(szaml1,nev1)
print("3. feladat")
print ("%d/%d = %d/%d" % \
      (szaml1,nev1,szaml1/lnko(szaml1,nev1),nev1/lnko(szaml1,nev1)))
print("4. feladat")
print ("2. tört számláló:")
szaml2 = int(input())
print ("2. tört nevező:")
nev2 = int(input())
szoroz(szaml1,nev1,szaml2,nev2)
print("6. feladat")
osszead(szaml1,nev1,szaml2,nev2)
print("7. feladat")
be = open("adat.txt",'r')
regi_kimenet=sys.stdout
ki=open("eredmeny.txt",'w')
sys.stdout=ki
line = be.readline()
while line!= "":
    valt = line.split()
    szaml1,nev1, szaml2, nev2 = \
                 int(valt[0]), int(valt[1]),int(valt[2]), int(valt[3])
    if valt[4]== "*":
        szoroz(szaml1,nev1,szaml2,nev2)
    else:
        osszead(szaml1,nev1,szaml2,nev2)
    line=be.readline()
be.close()
sys.stdout=regi_kimenet
ki.close()
```

„FUTÁR" ÉRETTSÉGI FELADAT KÓDJA

```
def kmtoft(km):
    if km < 3:
        return 500
    elif km < 6:
        return 700
    elif km < 11:
        return 900
    elif km < 21:
        return 1400
    else:
        return 2000

nap = []
ut = []
tav = []
print("1. feladat")
i = 0
be = open("tavok.txt","r")
line = be.readline()
while line != "":
    bont = line.split()
    nap.append( int(bont[0]))
    ut.append(int(bont[1]))
    tav.append(int(bont[2]))
    line = be.readline()
be.close()
print("Fájlbeolvasás kész.")
print("2. feladat")
napmin, napmax = 10, 0
for i in nap:
    if i < napmin:
        napmin = i
    if i > napmax:
        napmax = i
utmin, utmax = 50, 0
tavmin, tavmax = 0, 0
```

```
for i in range(len(nap)):
    if ut[i] < utmin and nap[i] == napmin:
        utmin = ut[i]
        tavmin = tav[i]
    if ut[i] > utmax and nap[i] == napmax:
        utmax = ut[i]
        tavmax = tav[i]
print("A hét első útja kilométerben: ", tavmin)
print("3. feladat")
print("A hét utolsó útja kilométerben: ", tavmax)
print("4. feladat")
napifuvar = []
napitav = []
for i in range(7):
    napifuvar.append(0)
    napitav.append(0)
for i in range(len(nap)):
    napifuvar[nap[i]-1] += 1
    napitav[nap[i]-1] += tav[i]
print("Ezeken a napokon nem dolgozott a futár: ", end=" ")
for i in range(7):
    if napifuvar[i] == 0:
        print(i + 1, end = " ")
print("\n6. feladat")
legtobbfuvar = -1
legtobbfuvarnap = -1
for i in range(7):
    if napifuvar[i] > legtobbfuvar:
        legtobbfuvar = napifuvar[i]
        legtobbfuvarnap = i
    print(i+1, ".nap: ", napitav[i], " km", sep="")
print("5. feladat")
print("A legtöbb fuvar ezen a napon volt: ", legtobbfuvarnap+1)
print("7. feladat")
t = int(input("Kérek egy távolságot: "))
print("Ennyi díjazás jár érte Ft-ban: ", kmtoft(t))
i = 0
while i < len(nap)-2:
    j = len(nap)-1
    while j > i:
        if 1000*nap[j]+ut[j] < 1000*nap[j-1]+ut[j-1]:
```

```
            nap[j], nap[j-1] = nap[j-1], nap[j]
            ut[j], ut[j-1] = ut[j-1], ut[j]
            tav[j], tav[j-1] = tav[j-1], tav[j]
        j-=1
    i+=1
osszeg = 0
ki=open("dijazas.txt","w")
for i in range(len(nap)):
    ki.write(str(nap[i])+ ".nap "+str(ut[i])+ \
        ".út: "+str(kmtoft(tav[i]))+ " Ft\n")
    osszeg+=kmtoft(tav[i])
ki.close()
print("A kiírás a dijazas.txt fájlba befejeződött")
print("9. feladat")
print("A kifizetett összeg: ",osszeg, " Ft", sep="")
```

„REJTVÉNY" ÉRETTSÉGI FELADAT KÓDJA

```
def egyezotabla(atvett):
    global megoldasok, feladvany
    egyezo = 1
    i = 0
    while i < 10 and egyezo == 1:
        j = 0
        while j < 10 and egyezo == 1:
            if 10 > megoldasok[(i,j,atvett)]:
                if megoldasok[(i,j,atvett)] != feladvany[(i,j)]:
                    egyezo = 0
            j+=1
        i+=1
    return egyezo

def hajodarabjo(atvett):
    global megoldasok
    dbhajo = 0
    for i in range(10):
        for j in range(10):
            if 11 == megoldasok[(i,j,atvett)]:
                dbhajo+=1
    if dbhajo == 12:
        return 1
    else:
        return 0

def szomszedsagjo(atvett):
    global megoldasok
    jo = 1
    x = 0
    while x < 10 and jo == 1:
        y = 0
        while y < 10 and jo == 1:
            i = x - 1
            while i < x + 2 and jo == 1:
                j = y - 1
                while j < y + 2 and jo == 1:
```

```
                    if i >= 0 and i < 10 and j >= 0 and j < 10 and \
                       not(x == i and y == j):
                        if megoldasok[(x,y,atvett)] != 0 and \
                           megoldasok[(i,j,atvett)] != 0:
                            jo = 0
                    j+=1
                i+=1
            y+=1
        x+=1
    return jo

def annyitlat(atvett):
    global megoldasok
    jo = 1
    x = 0
    while x < 10 and jo == 1:
        y = 0
        while y < 10 and jo == 1:
            if 0 < megoldasok[(x,y,atvett)] < 11:
                dbv, dbf = 0,0
                for i in range(10):
                    if megoldasok[(x,i,atvett)] == 11:
                        dbf+=1
                    if megoldasok[(i,y,atvett)] == 11:
                        dbv+=1
                if dbv+dbf != megoldasok[(x,y,atvett)]:
                    jo = 0
            y+=1
        x+=1
    return jo

feladvany = {}
megoldasok = {}
megfejtok = []
print("1. feladat")
x = int(input("Kérem a torony helyének oszlopszámát: "))
y = int(input("Kérem a torony helyének sorszámát: "))
h = int(input("Kérem a toronyból látható hajók számát: "))
if h > 3:
    print("Nehéz torony. ")
print("2. feladat")
```

```
i = x - 1
while i <= x + 1:
    j = y - 1
    while j <= y + 1:
        if i > 0 and i <11 and j > 0 and j < 11 and \
           not(x == i and y == j):
             print(i, j)
        j+=1
    i+=1
print("3. feladat")
be = open("feladvany.txt","r")
for i in range(10):
    line = be.readline()
    atm = line.split()
    for j in range(len(atm)):
        feladvany[(i,j)] = int(atm[j])
be.close()
be = open("megoldas.txt","r")
line = be.readline()
dbmegfejtok = int(line)
for k in range(dbmegfejtok):
    line = be.readline()
    megfejtok.append(line.strip())
    for i in range(10):
        line = be.readline()
        atm = line.split()
        for j in range(len(atm)):
            megoldasok[(i,j,k)] = int(atm[j])
be.close()
dbehet, dbhajorossz, dbszomszedrossz = 0,0,0
for k in range(dbmegfejtok):
    if hajodarabjo(k) == 0:
        dbhajorossz+=1
    if szomszedsagjo(k) == 0 and hajodarabjo(k) == 1 and \
       egyezotabla(k) == 1:
        dbszomszedrossz+=1
    if egyezotabla(k) == 0:
        dbehet+=1
        print(megfejtok[k], end=" ")
if dbehet == 0:
    print("Mindegyik megfejtés erre a hétre érkezett.")
```

```
print("\n4. feladat")
print("A hajók száma ennyi megoldáson hibás: ", dbhajorossz)
print("5. feladat")
print("A szomszédság nem megfelelő ennyi esetben: ", dbszomszedrossz)
print("6. feladat")
dbhelyes=0
for k in range(dbmegfejtok):
    if szomszedsagjo(k) == 1 and hajodarabjo(k) == 1 and \
       egyezotabla(k) == 1 and annyitlat(k)==1:
        dbhelyes+=1
        print(megfejtok[k], end=" ")
print("\nÖsszesen ennyi helyes megfejtés érkezett: ", dbhelyes)
```

„SZAVAK" ÉRETTSÉGI FELADAT KÓDJA

```
print("1. feladat")
szo = input("Kérek egy szót: ")
mgh = "aeiou"
h = len(szo)
i = 0
while i <= h-1 and not(szo[i] in mgh):
    i+=1
if i > h-1:
    print("Nincs benne magánhangzó.")
else:
    print("Van benne magánhangzó.")

print("3. feladat")
be = open("szoveg.txt","r")
szo = be.readline()
szo = szo.strip()
h = len(szo)
legh = h
leghszo = szo
dbmagan = 0
db = 0
otbetus = []
while szo!= "":
    db+=1
    magan,massal = 0,0
    for betu in szo:
        if betu in mgh:
            magan+=1
    massal = h -  magan
    if magan > massal:
        dbmagan+=1
        print (szo, end=" ")
    if legh < h:
        legh, leghszo = h, szo
    if h == 5:
        otbetus.append(szo)
    szo = be.readline()
```

```
        szo = szo.strip()
        h = len(szo)
be.close()
print("\n", dbmagan, "/", db, " : ", \
      "{:.2f}" . format(100*dbmagan/db), sep="")
print("2. feladat")
print("Leghosszabb szo hossza %d, például: %s. " %(legh, leghszo))
print("4. feladat")
keresettszo = input("Kérek egy 3 betűs szót: ")
print("Hozzá tartozó létraszavak: ")
for vizsgalt in otbetus:
    if vizsgalt[1:4] == keresettszo:
       print(vizsgalt, end=" ")
print("5. feladat")
i = 0
while i < len(otbetus)-2:
    j = len(otbetus)-1
    while j>i:
        if otbetus[j][1:4] < otbetus[j-1][1:4]:
            otbetus[j],otbetus[j-1] = otbetus[j-1],otbetus[j]
        j-=1
    i+=1
ki = open("letra.txt","w")
if otbetus[0][1:4] == otbetus[1][1:4]:
    ki.write(otbetus[0]+ "\n")
i = 1
while i < len(otbetus)-2:
    if otbetus[i][1:4] == otbetus[i-1][1:4] or \
       otbetus[i][1:4] == otbetus[i+1][1:4]:
        ki.write(otbetus[i]+ "\n")
    if otbetus[i][1:4] == otbetus[i-1][1:4] and \
       otbetus[i][1:4] != otbetus[i+1][1:4]:
        ki.write("\n")
    i+=1
veg = len(otbetus)-1
if otbetus[veg][1:4] == otbetus[veg-1][1:4]:
    ki.write(otbetus[veg]+"\n")
ki.close()
```

A „TÖRTEK" ÉRETTSÉGI FELADAT MEGOLDÁSA OBJEKTUMORIENTÁLT VÁLTOZATBAN

```
import sys

class Tortek:
    def __init__(self):
        self.szamlalo = 0
        self.nevezo = 1

    def kiir(self):
        print("%d/%d" % (self.szamlalo,\
            self.nevezo),end=" ")

    def beker(self):
        self.szamlalo = int(input("Kérem a számlálót: "))
        self.nevezo = int(input("Kérem a nevezőt: "))

    def szamlalotad(self):
        return self.szamlalo

    def nevezotad(self):
        return self.nevezo

    def lnko(a,b):
        if a == b:
            return a
        if a < b:
            return Tortek.lnko(a, b - a)
        if a > b:
            return Tortek.lnko(a - b, b)

    def egyszerusit(self):
        if self.szamlalo % self.nevezo == 0:
            print("= %d" % (self.szamlalo / self.nevezo))
        else:
            print("= %d/%d" %  \
            (self.szamlalo/Tortek.lnko(self.szamlalo,self.nevezo), \
             self.nevezo/Tortek.lnko(self.szamlalo,self.nevezo)))
```

```
    def egesz_e(self):
        if self.szamlalo % self.nevezo == 0:
            print(int(self.szamlalo / self.nevezo))
        else:
            print("Nem egész.")

    def megad(self, sz, n):
        self.szamlalo, self.nevezo = sz,n

class Muveletek(object):
    def __init__(self, number):
        self.number = number
        self.tort1 = Tortek()
        self.tort2 = Tortek()
        self.eredmeny = Tortek()
        self.muvjel = "*"

    def kiir(self):
        self.tort1.kiir()
        print("%s " % \
         (self.muvjel), end ='')
        self.tort2.kiir()
        print("= ", end = '')

    def beker(self):
        print("1. tört")
        self.tort1.beker()
        print("2. tört")
        self.tort2.beker()

    def szoroz(self):
        self.sz = self.tort1.szamlalotad() \
                  * self.tort2.szamlalotad()
        self.n = self.tort1.nevezotad()\
                 * self.tort2.nevezotad()
        self.eredmeny.megad(self.sz,self.n)
        self.kiir()
        self.eredmeny.kiir()
        self.eredmeny.egyszerusit()
```

```
def lkkt(a,b):
    return int(a * b / Tortek.lnko(a,b))

def osszead(self):
    self.kn = Muveletek.lkkt(self.tort1.nevezotad(),self.tort2.nevezotad())
    self.sz1 = self.tort1.szamlalotad() * self.kn / self.tort1.nevezotad()
    self.sz2 = self.tort2.szamlalotad() * self.kn / self.tort2.nevezotad()
    self.eredmeny.megad(self.sz1 + self.sz2, self.kn)
    self.kiir()
    self.tort1.megad(self.sz1,self.kn)
    self.tort2.megad(self.sz2,self.kn)
    self.kiir()
    self.eredmeny.kiir()
    self.eredmeny.egyszerusit()

def valaszt(self):
    if self.muvjel == "*":
        self.szoroz()
    elif self.muvjel == ":":
        self.oszt()
    elif self.muvjel == "-":
        self.kivon()
    else:
        self.osszead()

def megad(self, sz1, n1, sz2, n2, muv):
    self.tort1.megad(sz1,n1)
    self.tort2.megad(sz2,n2)
    self.muvjel = muv

def oszt(self):
    self.sz = self.tort1.szamlalotad() \
             * self.tort2.nevezotad()
    self.n = self.tort1.nevezotad()\
            * self.tort2.szamlalotad()
    self.eredmeny.megad(self.sz,self.n)
    self.kiir()
    self.muvjel = '*'
    self.sz = self.tort2.nevezotad()
    self.n = self.tort2.szamlalotad()
    self.tort2.megad(self.sz,self.n)
```

```
            self.kiir()
            self.eredmeny.kiir()
            self.eredmeny.egyszerusit()

        def kivon(self):
            self.kn = Muveletek.lkkt(self.tort1.nevezotad(),self.tort2.nevezotad())
            self.sz1 = self.tort1.szamlalotad() * self.kn / self.tort1.nevezotad()
            self.sz2 = self.tort2.szamlalotad() * self.kn / self.tort2.nevezotad()
            self.eredmeny.megad(self.sz1 - self.sz2, self.kn)
            self.kiir()
            self.tort1.megad(self.sz1,self.kn)
            self.tort2.megad(self.sz2,self.kn)
            self.kiir()
            self.eredmeny.kiir()
            self.eredmeny.egyszerusit()

        def muveletetmegad(self, jel):
            self.muvjel = jel

class Control():
        def __init__(self):
            tort = Tortek()
            muvelet = Muveletek(1)
            tort.beker()
            tort.egesz_e()
            tort.kiir()
            tort.egyszerusit()
            muvelet.beker()
            muvelet.szoroz()
            muvelet.muveletetmegad('+')
            muvelet.osszead()
            self.fajlokatkezel()

        def fajlokatkezel(self):
            self.muv = []
            regi_kimenet=sys.stdout
            be = open("adat.txt","r")
            ki = open("eredmeny.txt","w")
            sys.stdout = ki
            line = be.readline()
            i = 0
```

```
        while line != "":
            bontas = line.split()
            self.muv.append(Muveletek(i))
            self.muv[i].megad(int(bontas[0]), \
                int(bontas[1]), int(bontas[2]), \
                 int(bontas[3]),bontas[4].strip())
            self.muv[i].valaszt()
            line = be.readline()
            i+=1
        be.close()
        sys.stdout=regi_kimenet
        ki.close()

control = Control()
```

KŐ, PAPÍR, OLLÓ JÁTÉK GRAFIKUS FELÜLETEN

```
from tkinter import *
from random import randrange

def indulj():
    global foto1, item1, ablak, vege, vel
    vel = randrange(3)
    fajlnev =  str(vel) + ".png"
    foto1 = PhotoImage(file=fajlnev)
    item1 = vaszon1.create_image(155,155, image=foto1) #155 kép középponja
    if vege == 0:
        ablak.after(100,indulj) #100 ms szünet után hívja az indulj-t

def listabeallit():
    global lista, foto2, item2, kivalasztott
    kiv = lista.curselection() #kiválasztott listaelem indexei
    kivalasztott = kiv[0] #az első kiválasztot index
    fajlnev =  str(kivalasztott) + ".png"
    foto2 = PhotoImage(file=fajlnev)
    item2 = vaszon2.create_image(155,155, image=foto2)

def valaszt(evt):
    listabeallit()

def bezar():
    global uzenetablak, vege
    uzenetablak.destroy()
    vege = 0

def allj():
    global vege, vel, kivalasztott, uzenetablak
    vege = 1
    if vel < kivalasztott and vel >= 0:
        szoveg = "Ön nyert!!!"
    elif vel == 2 and kivalasztott == 0:
        szoveg = "Ön nyert!!!"
    elif vel == kivalasztott:
        szoveg = "Döntetlen!!!"
```

```
        else:
            szoveg = "A gép nyert!!!"
        uzenetablak = Toplevel(ablak) #üzenetablak létrehozása
        uzenetablak.option_add("*font","Arial 11")
        uzenet = Message(uzenetablak, text=szoveg, width=300)
        cmdButton3 = Button(uzenetablak, text="OK", command=bezar)
        uzenet.pack()
        cmdButton3.pack()
        uzenetablak.mainloop()

ablak = Tk()
ablak.option_add("*font","Arial 11") #ablakban alapértelmezett betűk
for i in range(3):
    ablak.columnconfigure(i,pad = 15) #oszlopok közötti hely
    ablak.rowconfigure(i,pad = 10) #sorok közötti hely
cmdButton1 = Button(ablak, text="Indulj!", width=15, command=indulj)
cmdButton2 = Button(ablak, text="Állj!", width=15, command=allj)
lista = Listbox(ablak, height=3) #3 karaktersor magasságú
lista.bind("<<ListboxSelect>>", valaszt) #listamezőre kattintva ez történik
lista.insert(END, "kő", "papír", "olló") #listamező elemei (az eddigiek után)
lista.select_set(0) #listamezőben alapértelmezetten kiválasztott a 0. sor
vaszon1 = Canvas(ablak, width=310, height=310, bg="white")
vaszon2 = Canvas(ablak, width=310, height=310, bg="white")
cmdButton1.grid(row = 1, column = 1)
cmdButton2.grid(row = 1, column = 2)
lista.grid(row = 2, column = 2)
vaszon1.grid(row = 3, column = 1)
vaszon2.grid(row = 3, column = 2)
listabeallit()
vege = 0
```

AMŐBA GRAFIKUS FELÜLETEN

```
from tkinter import *
import winsound

def kor(x, y, hely):
    global mezomeret, korr
    x1, y1 = x*mezomeret+2*korr, y*mezomeret+2*korr
    x2, y2 = (x+1)*mezomeret, (y+1)*mezomeret
    hely.create_rectangle(korr,korr, mezomeret+korr, \
        mezomeret+korr, width=2, fill="dark grey", outline="dark blue")
    hely.create_oval(x1,y1,x2,y2,width=2,fill="red",outline="black")

def iksz(x, y, hely):
    global mezomeret, korr
    x1, y1 = x*mezomeret+2.5*korr, y*mezomeret+2.5*korr
    x2, y2 = (x+1)*mezomeret-0.5*korr, (y+1)*mezomeret-0.5*korr
    hely.create_rectangle(korr,korr,mezomeret+korr, \
        mezomeret+korr, width=2, fill="dark grey", outline="dark blue")
    hely.create_line(x1,y1,x2,y2,width=4,fill="green")
    hely.create_line(x2,y1,x1,y2,width=4,fill="green")

def uj():
    global canPalya, mezomeret, palyameret, korr, jatekos, palyatar, vege
    palyatar = {}
    vege = 0
    for i in range(mezoszam):
        for j in range(mezoszam):
            palyatar[(i,j)] = 0
    jatekos = 1
    for x in range(0, palyameret, mezomeret):
        for y in range(0, palyameret, mezomeret):
            x1, y1, x2, y2 = x+korr, y+korr, \
                x+mezomeret+korr, y+mezomeret+korr
            canPalya.create_rectangle(x1,y1,x2,y2, \
                width=2, fill="dark grey", outline="dark blue")
    canNyert.create_rectangle(korr,korr,mezomeret+korr, \
            mezomeret+korr, width=2, fill="dark grey", outline="dark blue")
    kor(0,0,canKovetkezo)
```

```
def palyankattint(event): #bal egérgombbal kattintás eseménye
    global canKovetkezo, mezomeret, korr, jatekos, \
           palyatar, vege, canPalya, canNyert
    if vege == 0:
        x = int((event.x-korr)/mezomeret)
        y = int((event.y-korr)/mezomeret)
        if palyatar[(x,y)] == 0:
            palyatar[(x,y)] = jatekos
            if jatekos == 1:
                kor(x,y,canPalya)
            else:
                iksz(x,y,canPalya)
            if kiertekel(x,y) == 0:

                if jatekos == 1:
                    jatekos = 2
                    iksz(0, 0, canKovetkezo)
                else:
                    jatekos = 1
                    kor(0, 0, canKovetkezo)
            else:
      # rendszerhang lejátszás, ASYNC = hang miatt nem áll meg a program
                winsound.PlaySound("SystemAsterisk", winsound.SND_ASYNC)
                if jatekos == 1:
                    kor(0,0,canNyert)
                else:
                    iksz(0,0,canNyert)
                vege = 1

def kiertekel(x, y):
    global jatekos, palyatar, mezoszam
    db, xj = 0, x
    while db < 5 and xj < mezoszam and palyatar[(xj,y)]==jatekos:
        db, xj = db + 1, xj + 1
    xb = x - 1
    while db < 5 and xb >=0 and palyatar[(xb,y)]==jatekos:
        db, xb = db + 1, xb - 1
    if db != 5:
        db = 0
    yj = y
```

```
    while db < 5 and yj < mezoszam and palyatar[(x,yj)]==jatekos:
        db, yj = db + 1, yj + 1
    yb = y - 1
    while db < 5 and yb >=0 and palyatar[(x,yb)]==jatekos:
        db, yb = db + 1, yb - 1
    if db != 5:
        db = 0
    xj,yj = x,y
    while db < 5 and yj < mezoszam and xj < mezoszam and \
          palyatar[(xj,yj)]==jatekos:
        db, xj, yj = db + 1, xj + 1, yj + 1
    xb, yb = x - 1, y - 1
    while db < 5 and yb >=0 and xb >= 0 and \
          palyatar[(xb,yb)]==jatekos:
        db, xb, yb = db + 1, xb - 1, yb - 1
    if db != 5:
        db = 0
    xj,yj = x,y
    while db < 5 and yj < mezoszam and xj >=0 and \
          palyatar[(xj,yj)]==jatekos:
        db, xj, yj = db + 1, xj - 1, yj + 1
    xb, yb = x + 1, y - 1
    while db < 5 and yb <= mezoszam and xb >= 0 and \
          palyatar[(xb,yb)]==jatekos:
        db, xb, yb = db + 1, xb + 1, yb - 1
    if db == 5:
        return 1
    else:
        return 0

mezoszam, mezomeret = 20, 30
palyameret = mezoszam * mezomeret
korr = 4 # a mező korvonalának legyen helye
ablak = Tk()
ablak.option_add("*Font", "Arial 11")
lblNyert = Label(ablak, text="Nyertes: ")
lblKovetkezo = Label(ablak, text="Következő játékos: ")
canNyert = Canvas(ablak, bg="dark grey", \
    height=mezomeret+korr, width=mezomeret+korr)
canKovetkezo = Canvas(ablak, bg="dark grey", \
    height=mezomeret+korr, width=mezomeret+korr)
```

```
cmdButton = Button(ablak, text="Új játék", command=uj, width=15)
canPalya = Canvas(ablak, bg="dark grey", \
    height=palyameret+korr, width=palyameret+korr)
canPalya.grid(row=1, column=1,rowspan=3, padx=15,pady=15)
lblKovetkezo.grid(row=1, column=2)
canKovetkezo.grid(row=1, column=3, padx=15)
lblNyert.grid(row=2, column=2, sticky=E)
canNyert.grid(row=2, column=3, padx=15)
cmdButton.grid(row=3, column=2, padx=15, columnspan=2)
canPalya.bind("<Button-1>", palyankattint) #bal egérgombra érzékeny rajz-
terület
uj()
ablak.mainloop()
```