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.

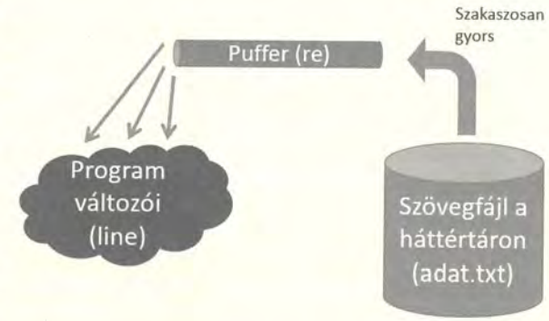
Természetesen a fájlt Jegyzettömbbel megnézve ezek a bájtok nem láthatók. Akkor honnan tudhatjuk, hogy tényleg léteznek? Bizonyos programok képesek bájtról bájtra megjeleníteni egy fájlt. Egy ilyen programmal megnézve a szöveges fájlt a sorvége jelek előtűnnek. Az alábbi ábrán az *adat.txt* fájl első 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.

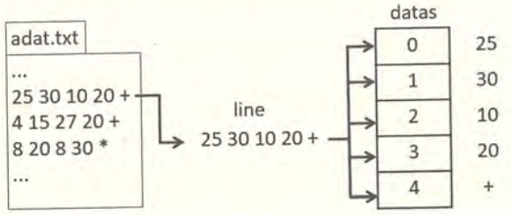
10. FÁJLOK BEOLVASÁSA

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. 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)

```
1 re = open("adat.txt",'r')
2 line = re.readline()
3 print(line)
4 re.close()
```

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

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.

```
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()
```

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.

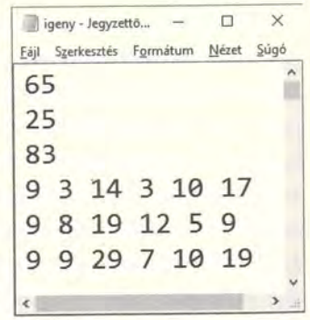
c) Formázott kiíratás

```
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



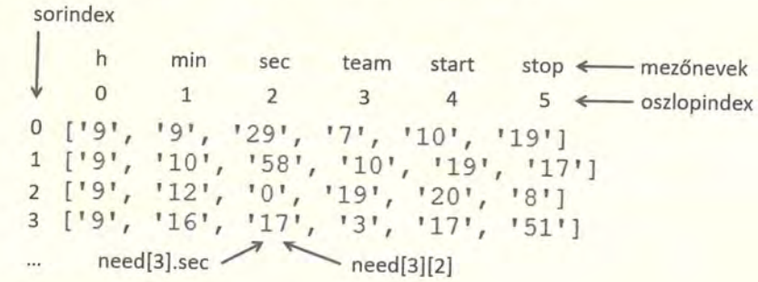
a képernyőn az igényeket az alábbi módon: *4. csapat – indul: 2. szint, érkezik: 10. szint, 2 óra 36 perc 47 mp*. Az utolsó igényt írjuk ki elsőnek, és haladjunk visszafelé az első igényig.

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.



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))
```


Feladatok

1. Írjunk programot, ami a tavok.txt fájl tartalmát jeleníti meg. Minden sor első adata a hét napjának sorszámával kezdődik (1 – hétfő, 2 – kedd...). Az adatok ilyen formában vannak a fájlban: 1 7 14, és ebben a formában kell megjelenniük a képernyőn: *hétfő 7. fuvar: 14 km.*
2. Adott a bsa.txt fájl, amiben egy DNS-szekvencia található (minden sorban egyetlen bázis). Írjuk ki a képernyőre a DNS-szekvenciát fordított sorrendben, sorfolytonosan. A DNS legfeljebb 1000 bázisból áll.
3. Írjunk programot, ami beolvassa a kep.txt fájl tartalmát a memóriába. A fájl egy sorában egy képpont adatai találhatók RGB kódban. A kép 50 x 50-es. Ellenőrzésképpen írassuk ki a képernyőre karakteres formában a színeket egy 50 x 50 karakteres táblázatban a következő módon: *P: vörös – RGB (255,0,0), Z: zöld –RGB (0,255,0), K: kék – RGB (0,0,255), S: sárga – RGB (255,255,0), egyéb.*

Minta részlet a 3. feladathoz

```
   0000000000111111111122222222223333333333444444444
   0123456789012345678901234567890123456789012345678
08 ..................................................
09 .........PPPPPPPPPP...........................
10 .........PPPPPPPPPP...........................
11 .........PPPPPPPPPP...........................
12 .........PPPPPPPPPP...........................
13 .........PPPPPPPPPP...........................
14 .........PPPPPZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ.........
15 .........PPPPPZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ.........
16 .........PPPPPZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ.........
17 .........PPPPPZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ.........
18 .........PPPPPZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ.........
19 .........PPPPPZZZZZZZZZZZZZZZZKKKKKKKKKKK.........
20 ..............ZZZZZZZZZZZZZZZZKKKKKKKKKKK.........
```