A Python nyelvű programozás alapjai

Tartalomjegyzék

Előszó

Ez a jegyzet középiskolás diákok számára készült. Elsősorban azoknak szántam, akik első programozási nyelvként tanulják a Pythont, de hasznos lehet azoknak is, akik már szereztek programozási tapasztalatokat egy másik nyelv tanulásával. A tartalom kiválasztásakor igyekeztem a Python nyelv gazdag lehetőségeiből azt a minimumot összegyűjteni, ami a közép- és emelt szintű érettségi vizsgán elengedhetetlenül szükséges.

A jegyzet nem tankönyv, azaz szövege tömör, kevés ábra és példaprogram található benne. Azoknak, akik olvasmányosabb leírásra, bővebb magyarázatokra vágynak, jó szívvel ajánlom Magyary Gyula könyvét:

Emelt szintű informatika érettségi 2. – Python lépésről lépésre
(2020 Metropolis Media Group Kft. ISBN 978 963 551 028 3)

Ebben a könyvben az érettségi követelményeken túl további érdekességeket is olvashatunk, például a grafikus felületű Windows applikációk készítéséről, illetve az objektum-orientált programozásról.

Szintén hasznos olvasmány a középiskolai digitális kultúra tankönyvsorozat, amelynek 9. 10. és 11. évfolyamra vonatkozó köteteiben találjuk ezt a témát. (Letölthető itt: https://www.tankonyvkatalogus.hu)

Azoknak, akik szeretnek oktatóvideókból tanulni, ajánlom ezt a remek portált: https://sulipy.hu/ Itt egy tapasztalt középiskolai tanár, Szabó-Bakos Gábor magyarázza el a Python nyelv alapjait. A portálon számos gyakorló feladatot is találunk, többek között a korábbi érettségi vizsgafeladatokat mintamegoldással és magyarázattal együtt. A középiskolai törzsanyagon túl itt is sok érdekességet találhatnak a programozás iránt érdeklődők.

A programozás kreatív tevékenység, ezért tanulásában elengedhetetlen a bőséges gyakorlás. Gyakorló feladatokat találhatunk például a sulipy.hu oldalon, illetve a Nemzeti Köznevelési Portálon (https://www.nkp.hu/) található Digitális kultúra 9–12, illetve 11–12 feladatgyűjteményekben.

Bevezető

Számítógépes programokkal felhasználóként különféle *applikációk* formájában találkozunk, amikor a számítógépünket, vagy okostelefonunkat használjuk. Ezeket az applikációkat erre szakosodott informatikusok, a programfejlesztők (programozók) készítik, de egyszerűbbeket – némi hozzáértéssel – akár mi is készíthetünk.

Minden applikáció mögött egy számítógépes program áll, pontosabban „fut”. A program határozza meg, hogy hogyan működjön az applikáció, azaz a számítógép. Fejlesztői szempontból a program nem más, mint számítógépnek adott *utasítások sorozata*, amelyek a számítógép működését vezérlik.

Számítógépes programokat különféle *programnyelveken* írhatunk. Számos ilyen programnyelv létezik, pl: Python, C++, JavaScript, stb. A Python nyelv kiválóan alkalmas a programozás alapjainak megtanulásához, de a profik körében is nagyon népszerű. (A nyelv egyébként nem a kígyóról kapta a nevét, hanem a Monty Python nevű angol humorista csoportról.)

Sajnos a számítógép ezeket a programnyelveket alapállapotban nem érti meg, ezért ún. *fordítóprogramokra* van szükségünk. A fordítóprogram az általunk valamilyen programnyelven megírt programot a számítógép saját „nyelvére” az ún. *gépi kódra* fordítja. (Lehetne gépi kódban is programozni, de az nagyon bonyolult, nem kezdőknek való. Ráadásul a gépi kódú programfejlesztés nagyon időigényes, ezért a profi gyakorlatban is csak ritkán, nagyon indokolt esetben alkalmazzák.)

A fordítás történhet a program készítésekor (compiler fordítók esetében) vagy a program futtatásakor, azaz működése közben (interpreter fordítóknál). Az előbbire példa a C++ nyelv, az utóbbira a Python és a JavaScript. (Mindkét módszernek vannak előnyei és hátrányai.) *A Python nyelv fordítóprogramját* letölthetjük a www.python.org webhelyről. (A sulipy.hu portálon részletes videót láthatunk a letöltésről és a telepítésről.)

Ahhoz, hogy elkészítsünk egy Python nyelven írt programot, egyszerű (formázást nem tartalmazó) szöveges fájlba kell írjunk bizonyos utasításokat, amik megfelelnek a Python nyelv szabályrendszerének (szakkifejezéssel: szintaxisának). Ezt megtehetjük akár a Jegyzettömb applikációval is, de érdemes inkább kifejezetten erre a célra készült alkalmazást, ún. *fejlesztő-környezetet* (IDE) használni. A fordítóprogrammal együtt települ az IDLE nevű fejlesztő alkalmazás, ami segít nekünk a programok írása közben. (Színes szövegkiemelés, hibajelzések, súgó, stb.) Az IDLE egy egyszerű fejlesztőeszköz, kezdőknek érdemes ezt, vagy a Thonny fejlesztőrendszert használni. Később ki lehet próbálni a nagyobb tudású, ámde bonyolultabb Visual Studio Code, vagy PyCharm alkalmazásokat is.

A Linux-alapú operációs rendszerekben általában már telepítve van a fordítóprogram, ezért a programozáshoz csak egy kódszerkesztő programra van szükségünk, amivel a programokat megírhatjuk és elmenthetjük. (A Python programfájlok kiterjesztése általában .py) Linux rendszer esetén az elmentett programot futtathatóvá is kell tennünk ahhoz, hogy használhassuk, illetve az is fontos, hogy a program első sorában ez álljon: `#!/usr/bin/env python3`

Az IDLE fejlesztőrendszert elindítva, a megnyíló ablak lesz a készülő programunk felhasználói ablaka, ezen keresztül kommunikál majd az applikációnk a felhasználóval. A program elkészítéséhez egy másik ablakot is kell nyitnunk: a *File* menü *New File* parancsával tehetjük ezt meg.

A megjelenő üres ablakba írhatjuk be a programunk utasításait, majd a *File* menü *Save As* parancsával menthetjük el a kész programot. Az IDLE fejlesztővel készített programot úgy tudjuk kipróbálni, hogy a *Run* menü *Run module* parancsát használjuk. (Futtatás előtt el kell menteni a programot.)

Miután a programot elkészítettük és elmentettük, már az IDLE alkalmazás nélkül is képes működni. Ha a fájlkezelővel megkeressük a .py kiterjesztésű programfájlt és duplán rákattintunk, akkor a program működésbe lép. A programot így indítva készüljünk fel arra, hogy amikor a program futása befejeződik, akkor az ablakát azonnal bezárja az operációs rendszer! Emiatt sokszor érdemes a programjaink végére az alábbi várakoztató utasítást tenni:

```
input("A program futása befejeződött, kilépéshez nyomjon Entert!")
```

Egy korábban készített Python programfájlt úgy tudunk újra szerkeszteni, hogy az IDLE fejlesztőrendszert elindítva, a *File* menü *Open...* parancsát választjuk.

Adatok kiírása, beolvasása és tárolása, feltételes utasítások

Az itt következő programrészlet bemutatja a Python programozási nyelv legalapvetőbb elemeit. A program utasításait felülről lefelé haladva, egymás után hajtja végre a számítógép.

```
print("Ez egy horoszkóp program. Melyik hónapban született?")
honap=input()
if honap=="január":
   print("Önt nagy szerencse fogja érni!")
if honap=="február":
   print("Ma ne menjen ki az utcára!")
```

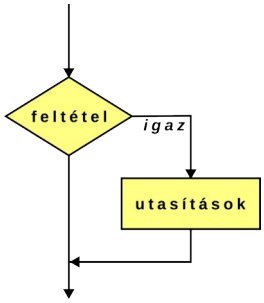
A `print` utasítás a képernyőre való írásra szolgál. Az utasítás után zárójelek között kell megadni a kiírandó adatot. Ha ez az adat egy szöveg, akkor idézőjelek (vagy aposztrófok) közé is kell zárni.

A számítógépes programok működésük folyamán különféle adatokat tárolnak. Az adatok tárolására szolgálnak az ún. *változók*. A változó nem más, mint egy névvel ellátott rekesz a számítógép memóriájában. (Olyan, mint egy fiók egy sok fiókos szekrényben.) A változók nevét a programozó adja. Egy változóban egyszerre csak egy adat lehet. Ha újabb adatot teszünk bele, a régi törlődik. A fenti programrészletben a `honap` nevű változót használjuk.

A változók nevének egy szóból kell állnia. Érdemes csak angol betűket, számjegyeket és aláhúzás karaktert használni a névben. A Python programozási nyelv *különbséget tesz a kis- és a nagybetűk között*, tehát a `honap` és a `Honap` nevű változók nem azonosak. Nagyobb programoknál célszerű úgy megválasztanunk a változók neveit, hogy utaljanak a funkciójukra.

A *`változónév`*`=input()` utasítás hatására a számítógép vár arra, hogy a felhasználó begépeljen valamit a billentyűzeten. Amikor a felhasználó Entert nyom, a gép tárolja az Enter előtt begépelt karaktereket a megadott nevű változóban.

Bizonyos utasításokat csak akkor hajt végre a számítógép, ha az utasítások előtt lévő feltétel teljesül. Erre való az `if… (ha…)` szerkezet:




```
if feltétel:
   utasítás1
   utasítás2
   stb.
```

Fontos, hogy a feltételhez tartozó utasításokat behúzással, mégpedig *azonos mértékű behúzással* kell írni! Célszerű ehhez a tabulátor gombot használni a billentyűzeten, ami 4 karakternyi behúzást eredményez.

A feltételben a szokásos összehasonlító jeleket (ún. *operátorokat*) használhatjuk:

`<` `<=` `>` `>=` `==` `!=`

Nagyon figyeljünk oda arra, hogy az egyenlőség-operátor *KÉT darab* `=` jelből áll! Ennek elvétése nagyon gyakori hiba. A `!=` operátor az egyenlőség tagadása (azaz a nem-egyenlő jel).

A feltételes programrészek további feltételeket is tartalmazhatnak, amint az a következő példában látható:

```
print("Köszönti Önt a jelszóellenőrző. Adja meg a jelszavát!")
jelszo=input()
if jelszo=="titok":
   print("A jelszó helyes, Ön beléphet.")
if jelszo!="titok":
   print("A jelszó hibás, próbálja meg újra!")
   jelszo=input()
   if jelszo=="titok":
      print("A jelszó most már helyes, Ön beléphet.")
   if jelszo!="titok":
      print("A jelszó ismét hibás!")
```

Az if szerkezetben *összetett feltételeket* is megfogalmazhatunk, az ún. *logikai operátorok* segítségével. A logikai operátorok a következők:

and : és **or** : vagy **not** : nem

```
print("Adja meg a felhasználónevét!")
nev=input()
print("Adja meg a jelszavát!")
jelszo=input()
if nev=="Főnök" and jelszo=="titok":
   print("Üdvözlöm, Főnök úr!")
if nev!="Főnök" or jelszo!="titok":
   print("A felhasználónév, vagy a jelszó hibás.")
```

Adattípusok átalakítása, az értékadó operátor

A változókban (azaz adattároló fiókokban) különféle *típusú* adatokat tárolhatunk: egész és törtszámokat, szöveget, stb. Az `input` utasítással a billentyűzetről bekért adatot szövegként kezeli a számítógép. Tehát például a felhasználó által beírt 123 adatot egy 3 karakterből álló szövegként és nem 3 jegyű számként kezeli.

Ha ez az adat szám és számolni is szeretnénk vele, akkor *át kell alakítani* szám típusú adattá. Erre láthatunk példát az alábbi program 2. és 4. sorában. A `float` utasítás valós számmá alakítja a zárójelei közé írt adatot. A 2. utasítássor értelme tehát: „A billentyűzetről beolvasott adatot alakítsd át tízes számrendszer-beli valós számmá és az eredményt tárold a tomeg nevű változóban!”

```
print("Hány kilogramm az Ön testtömege?")
tomeg=float(input())
print("Hány méter az Ön testmagassága? Használjon tizedespontot!")
magassag=float(input())
tti=tomeg/(magassag*magassag)
print("Az Ön testtömeg-indexe:",tti)
if tti<18 :
   print("Ön sovány!")
if 18<=tti<25 :
   print("Ön normális testalkatú.")
if 25<=tti :
   print("Ön túlsúlyos!")
```

A változókban nemcsak az `input` utasítással tárolhatunk adatokat, hanem az ún. *értékadó operátor* (`=`) segítségével is. A fenti program 5. sorában a `tti` nevű változóba tettük a számítás eredményét. Az értékadó operátor tehát a „legyen egyenlő” utasításnak felel meg. Az értékadó operátort sajnos könnyű összekeverni az egyenlőséget vizsgáló *összehasonlító operátorral.* (`==`)

A törtszámok megadásánál tizedesjelként pontot kell használni! Ha egy adat szöveg típusú, akkor értékét idézőjelek (vagy az angol nyelvben használatos aposztrófok) között kell megadnunk.

A program 6. sorában arra látunk példát, hogy miként lehet egy `print` utasítással egy sorba kiírni több adatot: egy szöveget és egy változó értékét. Figyeljük meg, hogy a `tti` változónév nincs idézőjelek közé téve, hiszen nem a „tti” szöveget szeretnénk kiíratni, hanem a `tti` nevű változó tartalmát!

Ha több adatot írunk ki egy `print` utasítással, akkor az adatok közé egy-egy szóközt ír a gép. Amennyiben ezt nem szeretnénk, akkor a következőt tehetjük:

`print(`*`egyik_adat`*`,`*`másik_adat`*`,sep="")`

Arra is rávehetjük a gépet, hogy a kiírás után ne kezdjen új sort:

`print(`*`kiírandó adatok`*`,end="")`

Véletlenszámok használata

A Python nyelvben vannak olyan utasítások, amelyeket csak akkor használhatunk, ha a megfelelő kiegészítő *modult importáljuk* a programunkba. Ezt a következőképpen tehetjük meg:

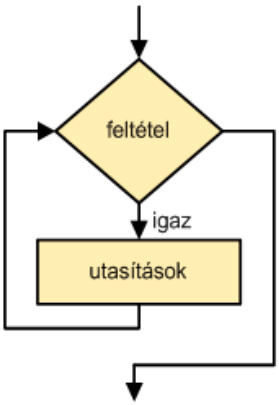
```
import modulnév
```

A véletlenszámokat sorsoló *randint* utasítás használatához importálnunk kell a *random* modult. A randint utasítás a zárójelben megadott két szám közötti tartományból sorsol ki egy egész számot. (A tartomány két szélét is beleértve.) Az alábbi példa egy szorzótábla-gyakorló applikáció programja:

```
import random
szam1=random.randint(1,10)
szam2=random.randint(1,10)
print("Szerinted mennyi",szam1,"x",szam2,"?")
szorzat=float(input())
if szorzat==szam1*szam2:
    print("Az eredmény helyes.")
if szorzat!=szam1*szam2:
    print("Nem jó! A helyes válasz:",szam1*szam2)
```

Ciklus, avagy ismétlés a programban

A programozásban gyakran szükség van arra, hogy a program egy részét megismételtessük a számítógéppel. Az ilyen szerkezetet **ciklusnak** hívják a programozók, az ismétlődő programrészt pedig **ciklusmagnak**. (A ciklus szó a köznyelvben is valamilyen ismétlődő dolgot jelent, gondolhatunk holdciklusra, választási ciklusra, árapály-ciklusra.)



A ciklusmag utasításait mindaddig ismétli a számítógép, amíg a ciklus-feltétel igaz. (Lásd a jobb oldali folyamatábrát.) Óvatlan programozók időnként ún. *végtelen ciklust* készítenek, amelynek feltétele mindig igaz marad (soha nem válik hamissá) és így a program soha nem ér véget, örök körforgásban marad.

A Python nyelvben a ciklus megvalósítására szolgál a következő szerkezet:

```
while feltétel:
   ismétlendő utasítások (ciklusmag)
```

Az ismétlendő utasításokat azonos mértékű behúzással kell írni a while sor után.

Az alábbi példa a fenti szorzótábla-gyakorló applikáció több menetes változata:

```
import random
print("Szeretnél szorzótáblát gyakorolni?")
gyakorlas=input()
while gyakorlas=="igen":
    szam1=random.randint(1,10)
    szam2=random.randint(1,10)
    print("Szerinted mennyi",szam1,"x",szam2,"?")
    szorzat=float(input())
    if szorzat==szam1*szam2:
        print("Az eredmény helyes.")
    if szorzat!=szam1*szam2:
        print("Nem jó! A helyes válasz:",szam1*szam2)
    print("Szeretnél még gyakorolni?")
    gyakorlas=input()
```

Tesztelés, hibakeresés

„Tévedni emberi dolog.” illetve „Csak az nem hibázik, aki nem dolgozik.” tartja a népi bölcsesség. Márpedig a programozók is keményen dolgozó emberek, ezért a programokba időnként hibák (bug) kerülnek. Minél nagyobb, bonyolultabb a program, annál nagyobb valószínűséggel csúszik bele hiba.

Alapvetően kétféle hiba kerülhet egy programba. A *szintaktikai* (formai) hibák olyanok, mint a beszélt nyelvekben vétett helyesírási hibák. A programnyelveknek is megvan a maguk „helyesírási”, azaz szintaktikai szabályrendszere, ami ugyan jóval egyszerűbb, mint a beszélt nyelveké, de a legapróbb hiba is működésképtelenné teszi a programot. A leggyakoribb ilyen hiba az, hogy hiányzik valamelyik zárójel, vagy idézőjel párja, esetleg rosszul írunk le egy kulcsszót. (A Python nyelvben a kis- és nagybetűk is számítanak!)

A szintaktikai hibák viszonylag könnyen megtalálhatók, ugyanis ilyenkor a program sokszor el sem indul, a fordítóprogram pedig jelzi a hibát. A hibaüzenetben azt is láthatjuk, hogy a program hányadik sorában van a hiba. (A sorok számozását az IDLE fejlesztőrendszer *Options* menüjének *Show Line Numbers* parancsával lehet bekapcsolni.)

A *szemantikai* (értelmi) hibákat már nehezebb felfedezni. Ilyenkor a program formailag helyes, ezért a fordítás sikeres lesz, a program működik, csak éppen nem úgy, ahogy azt a programozó elképzelte. (Ahogy a programozók tréfásan mondják: „A program nem kívánságaink, hanem utasításaink szerint működik.”) A tünetek változatosak lehetnek: a program működés közben váratlanul leáll, vagy éppen soha nem áll le, esetleg hibás eredményeket ír ki, vagy furcsa dolgokat cselekszik. Ezeket a hibákat futási időben megjelenő hibáknak is szokták nevezni. Ilyenkor vagy az a hiba oka, hogy a programozó rossz algoritmust tervezett a feladat megoldására, vagy az algoritmus ugyan jó, de rosszul kódolta azt az adott programnyelven.

Bonyolultabb programokban könnyen előfordulhatnak olyan apróbb hibák, amik nem jelennek meg az első próbafuttatások során, csak később. Az ilyen hibák kiszűrésére szolgál a program *tesztelése*, ami bizony időigényes feladat. A programok tesztelését általában úgy végzik, hogy különféle bemeneti adatokkal többször is lefuttatják a programot és figyelik, hogy az megfelelően működik-e. A működési hibák biztos kiszűréséhez elméletileg az összes lehetséges bemeneti adattal tesztelni kellene a programot, de ez általában nem lehetséges, illetve felesleges is. (Pl. ha egy összeadást végző program helyesen adja össze az 1 és 2 számokat, akkor nagy valószínűséggel helyesen fogja összeadni az 1 és 3 számokat is.)

Teszteléskor tehát igyekszünk olyan tesztadatokkal kipróbálni a programot, amelyek egymástól lényegesen különböznek. Az, hogy mit jelent a „lényeges különbség”, alapvetően a program algoritmusától, illetve a tárolásra használt adatszerkezetektől függ. A megfelelő tesztadatok kiválasztását sokszor az ún. *teljes lefedés elve* alapján szokták végezni. Az elv szerint a különböző tesztadatokat úgy kell megválasztani, hogy a program algoritmusában szereplő *összes elágazáson végigmenjünk* legalább egyszer.

A tesztelést érdemes már a program fejlesztése közben, a félkész programon is folyamatosan végezni. Kezdőknek azt szokták javasolni, hogy néhány programsor beírása után mindig próbálják ki a programot. Így elkerülhető az, hogy „rossz alapra építünk magas falat”.

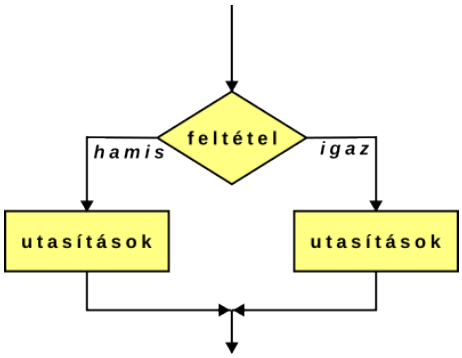
Ha a tesztelés során hibás működést tapasztalunk, akkor következik a *hibakeresés* művelete. A szemantikai hibák felderítésekor a programozónak végig kell gondolnia, hogy az általa írt utasítások hatására pontosan mit csinál a számítógép, és így megtalálni a hiba okát. A hiba sokszor nem ott van, ahol a hibajelenség fellép, hanem valahol korábban romlik el valami.

A hibakeresésben hasznos és gyakran alkalmazott módszer, hogy a program futása közben időnként *kiíratjuk bizonyos változók értékét*. Így fény derülhet arra, hogy mi az ami nem a terveink szerint történik. Fejlettebb fejlesztőrendszerek (pl. PyCharm, Visual Studio Code) különféle segédeszközökkel támogatják a hibakeresést, például lehetőséget adnak a program *lépésenként történő futtatására*, a változók értékeinek folyamatos megjelenítésére, illetve töréspontok elhelyezésére a programban.

Elágazások a programban

A korábban megismert if... (ha...) szerkezetnek létezik két bővebb változata is a Python nyelvben.

Az if… else… (ha… különben…) szerkezet
kétirányú elágazást tesz lehetővé a programban:

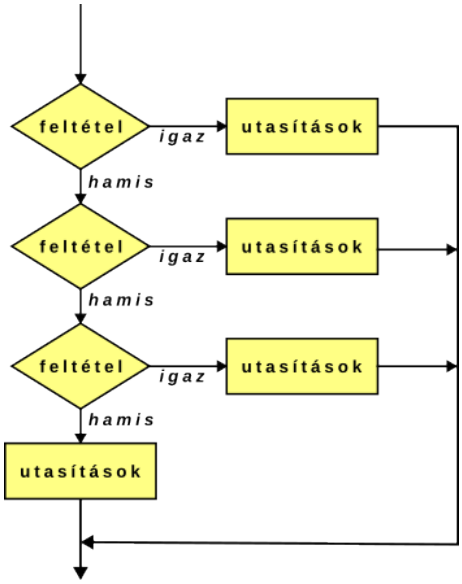



```
if feltétel:
   utasítások
else:
   utasítások
```

Nézzünk egy konkrét példát:

```
if szorzat==szam1*szam2:
    print("Az eredmény helyes.")
else:
    print("Nem jó, a helyes válasz:",szam1*szam2)
```

Az if… elif… else… (ha… különben ha… különben…) szerkezettel
pedig több irányú elágazások valósíthatók meg:




```
if feltétel:
   utasítások
elif feltétel:
   utasítások
elif feltétel:
   utasítások
else:
   utasítások
```

Nézzünk erre is egy konkrét példát:

```
print("Kérem az első számot!")
szam1=float(input())
print("Kérem a műveleti jelet!")
jel=input()
print("Kérem a második számot!")
szam2=float(input())
if jel=="+":
    print("A két szám összege:",szam1+szam2)
elif jel=="-":
    print("A két szám különbsége:",szam1-szam2)
elif jel=="*":
    print("A két szám szorzata:",szam1*szam2)
elif jel=="/":
    print("A két szám hányadosa:",szam1/szam2)
else:
    print("Csak a négy alapműveletet tudom elvégezni.")
```

A figyelmes olvasó észrevehette, hogy a programokban gyakran előfordul, hogy egy `print` utasítást egy `input` utasítás követ. Jellemzően azt szoktuk az input utasítás előtt kiíratni a képernyőre, hogy milyen adatot várunk a felhasználótól. A Python nyelvre jellemző a rövid, tömör fogalmazás, ezért lehetőségünk van ezt a két utasítást összevonni. Az input utasítás ugyanis nemcsak az adatok beolvasására képes, hanem egy rövid szöveg kiírására is. A kiírandó szöveget az input utasítás utáni zárójelek közé kell írni.

A fenti példaprogram első két sora helyett tehát ezt is írhatjuk:

```
szam1=float(input("Kérem az első számot!"))
```

Műveletek számokkal

A programozásban *operátoroknak* nevezzük a különféle műveleti jeleket. Az operátorok az adatokon „operálnak”, azaz különféle műveleteket végeznek az adatokkal. Ezeket az adatokat *operandusoknak* nevezzük. Például az 5 + 6 műveletben a + jel az operátor, az 5 és 6 számok az operandusok.

A négy alapműveletet a szokásos operátorokkal (műveleti jelekkel) végezhetjük el: `+ - * /`

Egy változó értékének növelése a legtöbb programnyelvben a következőképpen lehetséges:
`változó=változó+szám`
Például az `x=x+2` utasítás hatására az x változó értéke 2-vel nagyobb lesz.
A Python nyelvben erre a műveletre létezik egy rövidebb forma is: `változó+=szám` Például: `x+=2`
Hasonlóképp csökkenthető egy változó értéke: `x-=2`
továbbá szorozhatjuk, vagy oszthatjuk a változó értékét egy számmal: `x*=2` illetve `x/=2`

A hatványozás operátora a dupla csillag. Például: 2**3 értéke 8, hiszen $2^3 = 8$.

Négyzetgyököt vonni törtkitevőjű hatványozással lehet (lásd: matekórán).

Pl: $81 ** 0.5 = 81^{\frac{1}{2}} = \sqrt{81} = 9$ Hasonlóképpen lehet vonni köbgyököt, negyedik gyököt stb.

A maradékos osztás operátora a dupla osztásjel (//) az osztási maradékot a % operátorral számíthatjuk ki. Például 7//3 értéke 2, hiszen a 7/3 osztás hányadosának egész része 2. (A három 2-szer van meg a 7-ben.) 7%3 értéke pedig 1, hiszen a 7-et 3-mal osztva 1 maradékot kapunk.

További számításokat a megfelelő *függvények* alkalmazásával végezhetünk el:

Egy törtszám egész részét az `int()` függvénnyel kaphatjuk meg, a kerekített értékét pedig a `round()` függvénnyel. Például `int(1.76)` értéke 1, `round(1.76)` értéke pedig 2.

A round() függvénnyel nemcsak egész számra kerekíthetünk, hanem adott tizedesjegyre. A tizedesjegyek számát vesszővel elválasztva kell írni a kerekítendő szám után: `round(1.76,1)` értéke 1.8

A round() függvény a tizedesjeltől *balra* is tud kerekíteni: round(123,-1) = 120 és round(123,-2) = 100

Egy szám abszolút értékét az `abs()` függvény adja meg: abs(-5) = 5

Ha a programunkba importáljuk a *math* modult, akkor további függvényeket is használhatunk:

```
import math
```

A `ceil()` függvénnyel *felfelé* kerekíthetjük a törtszámokat. Pl: math.ceil(1.35) = 2

A `factorial()` függvénnyel faktoriális számítást végezhetünk. Pl: math.factorial(5) = 5! = 120

A `sin()`, `cos()`, `tan()` függvényekkel a szokásos trigonometriai műveleteket végezhetjük el.
Figyelem: ezek a függvények *radián* mértékegységben várják a szögek értékét!
Szerencsére léteznek átváltó függvények: a `radians()` függvénnyel fokot válthatunk át radiánba,
a `degrees()` függvénnyel pedig radiánt fokba.
Például a 30 fokos szög szinuszát a következőképpen számíthatjuk ki: math.sin(math.radians(30))

A `log()` függvénnyel logaritmust számolhatunk. Például: $math.\log(64,2) = \log_2 64 = 6$

Körrel kapcsolatos számításokhoz jól jöhet a math modulba épített `pi` változó. Ennek értékét „gyárilag” beállították, tehát a print(math.pi) utasítás hatására a jól ismert számot látjuk a képernyőn.
Figyelem: ne használjuk ezt a változónevet más célra a programjainkban mert az kavarodást okozhat!

Véletlen-számok előállításához a *random* modult kell importálnunk a programba:

```
import random
```

A `randint()` függvénnyel véletlen *egész* számot állíthatunk elő egy adott intervallumból.
Pl: random.randint(1,6) egy kockadobást szimulál.

A `random()` függvénnyel véletlen *valós* számot állíthatunk elő a [0 ; 1 [intervallumból.
Ennek a függvénynek nincs paramétere, tehát a zárójelei közé ne írjunk semmit: random.random()

A `choice()` függvénnyel egy listából választhatunk ki egy elemet véletlen-szerűen. (A listákat lásd később a jegyzetben.) Például a random.choice(["alma","körte","barack"]) függvényhívás eredménye az egyik gyümölcs lesz a felsoroltak közül.

Műveletek szövegekkel

A szövegeket – mint minden más adatot – számok formájában tárolja a számítógép. Minden egyes betűhöz (pontosabban karakterhez) tartozik egy szám, ezt nevezzük a betű (karakter) kódjának. Az egyes karakterek kódjait nemzetközi szabvány, az ún. Unicode határozza meg.

Egy karakter kódját az `ord(`*`karakter`*`)` függvénnyel tudhatjuk meg. Például ord("A") értéke 65.

Ennek „fordítottja” a `chr()` függvény, tehát egy adott kódú karaktert a `chr(`*`kód`*`)` függvénnyel kaphatunk meg. Például chr(66) eredménye a B karakter.

A szövegek között értelmezett az összeadás (pontosabban az összefűzés, szakszóval *konkatenáció*) művelete. Két szöveg összege a két szöveg egymás után fűzésével kapott szöveg lesz:

`"Aba"+"Sámuel"` értéke `"AbaSámuel"`

A szövegeket össze is lehet hasonlítani a szokásos relációs jelekkel. A < és > jelek esetében ez ABC sorrend szerinti összehasonlítást jelent, de sajnos ez csak az angol ABC karaktereire működik helyesen, a magyar ékezetes karakterekre nem. (Ugyanis az Unicode kódtáblában az ékezetes betűkhöz nem az ABC sorrend szerinti kódok tartoznak.) A kis- és nagybetűk összehasonlítása szintén problematikus: `"alma"<"narancs"`, viszont `"alma">"Narancs"`. (A nagybetűk előrébb vannak a kódtáblában.)

Egy szöveg karakterekben mért hosszát a `len()` függvénnyel tudhatjuk meg.
Például a `nev` változóban tárolt szöveg hosszát a következőképpen írhatjuk ki:

```
print(len(nev))
```

A szövegeket lehetőségünk van „ízekre szedni”, azaz egyes részeivel külön foglalkozni.
Például a `nev` változóban tárolt szöveg első karakterét a következőképpen írhatjuk ki:

```
print(nev[0])
```

Figyeljünk arra, hogy a szövegek karaktereinek sorszámozása, szaknyelven *indexelése* 0-tól indul, ezért egy 5 betű hosszúságú szöveg utolsó betűje a 4-es indexű!

Egy öt karaktert tartalmazó név utolsó karakterét kétféleképpen is kiírhatjuk:

`print(nev[4])` illetve `print(nev[-1])`

A fenti példából látszik, hogy amennyiben negatív indexet adunk meg, akkor a szöveg végéről számolja a karaktereket a Python. A név utolsó előtti karakterét tehát a nev[-2] kifejezés adja.

Egy ismeretlen hosszúságú szöveg utolsó karakterét kétféleképpen is kiírhatjuk:

`print(nev[len(nev)-1])` illetve `print(nev[-1])`

Lehetőségünk van egy szövegből egy részt kiemelni. A következő utasítás a név első két karakterét írja ki (tehát a 2-es indexű *előtti*, azaz a 0-ás és 1-es indexű karaktereket):

```
print(nev[:2])
```

A következő utasítás a név első két karaktere *utáni* szövegrészt írja ki (tehát a 2-es indexűtől *kezdve*):

```
print(nev[2:])
```

A következő pedig a név 2-es, 3-as és 4-es indexű karaktereit jeleníti meg:

```
print(nev[2:5])
```

A szövegek egy-egy karakterét *megváltoztatni* sajnos kissé nehezebb, mint más programnyelvekben. Ha például a név 5-ös indexű karakterét szeretnénk P betűre változtatni, akkor a következő utasítás nem működik, hibát okoz:

```
nev[5]="P"
```

Ehelyett a következőt tehetjük:

```
nev=nev[:5]+"P"+nev[6:]
```

A szöveg egy részét a következőképpen törölhetjük (a példában az 5-ös és 6-os indexű karaktereket):

```
nev=nev[:5]+nev[7:]
```

A szövegeket ideiglenesen, vagy akár véglegesen nagybetűssé alakíthatjuk.
A következő utasítás kiírja a nevet nagybetűs formában, de nem változtatja meg a `nev` változó tartalmát:

```
print(nev.upper())
```

A következő utasítás viszont megváltoztatja a `nev` változó tartalmát (véglegesen nagybetűssé alakítja):

```
nev=nev.upper()
```

Kisbetűssé a következőképp alakíthatunk:

```
nev=nev.lower()
```

A `find()` függvény arra szolgál, hogy egy szövegben megkeressük egy szövegrész első előfordulásának helyét. Például ha a `nev` változó értéke AbaSámuel, akkor `nev.find("Sám")` értéke 3 lesz.

Ha a keresett részszöveg nincs benne a szövegben, akkor a függvény a -1 számot adja eredményül.

Megadhatjuk azt is, hogy hányadik karaktertől (melyik indexű karaktertől) kezdődjön a keresés:

```
nev.find("keresett szöveg",kezdőindex)
```

Akár azt is megadhatjuk, hogy hányas indexű karakterig tartson a keresés:

```
nev.find("keresett szöveg",kezdőindex,végindex)
```

Jobbról balra haladva is kereshetünk a szövegben. A következő függvény az utolsó e betű indexét adja:

```
nev.rfind("e")
```

A következő függvény megadja a `nev` változóban tárolt szövegben előforduló e betűk számát:

```
nev.count("e")
```

A következő utasítás kicseréli a `nev` változóban tárolt szövegben az e betűket * karakterre:

```
nev=nev.replace("e","*")
```

A következő utasítás kitörli a `nev` változóban tárolt szöveg elejéről és végéről a szóközöket:

```
nev=nev.strip()
```

Más karaktereket is eltávolíthatunk a `strip()` függvénnyel. Az alábbi utasítás a + és * karaktereket törli:

```
nev=nev.strip("+*")
```

A következő utasítás szétdarabolja a nevet a szóköz karakterek mentén és a darabokat egy listában helyezi el. (A listákat lásd a következő fejezetben.)

```
nevdarabok=nev.split()
```

Nem csak a szóközök mentén darabolhatjuk a szövegeket. A következő utasítás a vesszők mentén bont:

```
nevdarabok=nev.split(",")
```

A strip() és a split() függvényeket a szöveges fájlokban lévő adatok beolvasásánál használjuk gyakran. (A fájlok kezelése emelt szintű tananyag, a módját lásd egy későbbi fejezetben.)

Listák használata

Amennyiben egy programban sok adatot kell tárolnunk, hosszadalmas (sokszor lehetetlen) volna mindegyik számára külön változót létrehozni. Ilyen esetekre szolgálnak a *változótömbök*, amiket a Python nyelvben *listának* neveznek. Ha a változókat úgy tekintjük, mint névvel ellátott fiókokat a memóriában, akkor a lista egy egész szekrénynek felel meg, amiben sorszámozott fiókok vannak. A sorszámot nevezzük a lista *indexének*.

Listákat például a következőképpen hozhatunk létre:

```
listanév=[]
```

Az így létrehozott lista egyelőre üres, azaz nem tartalmaz adatokat.

Listákat úgy is létrehozhatunk, hogy azonnal adatokat is teszünk beléjük:

```
szamok=[1, 23, -16, 0.25]
szavak=["alma", "körte", "barack"]
szemelyi_adatok=["Nagy Lajos", "király", 1342, 1382]
```

A fenti példákból látható, hogy egy listában különböző típusú adatok is lehetnek.

Új adatot a következőképpen tehetünk a listába:

```
listanév.append(ÚjAdat)
```

Ha a listában már vannak adatok, akkor a fenti utasítás hatására az új adat a lista végére kerül.

A listák indexelése szintén 0-tól kezdődik, azaz az első listaelemet a következőképp érhetjük el:

```
listanév[0]
```

Egy négy elemet tartalmazó lista utolsó elemét kétféleképpen is elérhetjük:

`listanév[3]` illetve `listanév[-1]`

A szövegektől eltérően, egy lista bármelyik elemét nyugodtan megváltoztathatjuk:

```
listanév[index]=ÚjTartalom
```

A lista egy elemét a következőképpen törölhetjük a listából:

```
listanév.remove(TörlendőAdat)
```

Ha a megadott adat többször is szerepel a listában, akkor a `remove()` függvény az *elsőt* fogja törölni. Ha viszont a megadott adat nem szerepel a listában, akkor a program hibaüzenettel le fog állni!

Azt, hogy egy adat szerepel -e egy listában, az `in` operátorral ellenőrizhetjük:

```
if adat in listanév:
    listanév.remove(adat)
```

Az `in` operátor (műveleti jel) a halmazelmélet *eleme* jelének felel meg. (Szövegekre is használható.)

Egy adott indexű listaelemet a következőképpen törölhetünk:

```
listanév.pop(index)
```

Amennyiben egy elemet törlünk a listából, úgy a mögötte lévő elemek „előrébb lépnek”.

A lista összes elemét a `listanév=[]` utasítással törölhetjük.

Ha egy új adatot nem a lista végére szeretnénk illeszteni, akkor használjuk az `insert()` függvényt:

```
listanév.insert(index,adat)
```

A függvény úgy illeszti be az adatot, hogy annak indexe a megadott legyen. Azok a listaelemek, amelyek indexe ennél nagyobb, vagy egyenlő volt, eggyel hátrébb tolódnak a listában.

A `len()` függvény listák esetében is működik, a lista elemeinek számát adja meg:

```
print(len(listanév))
```

A `listanév.count(adat)` függvény is használható listákra: egy adat előfordulási számát adja meg.

A `listanév.index(adat)` függvény megadja, hogy az adat hányas indexű eleme a listának.
Ha a megadott adat nem szerepel a listában, akkor a program hibaüzenettel le fog állni!
Amennyiben a keresett adat többször is szerepel a listában, akkor a legkisebb indexet adja meg.

A `listanév.sort()` utasítás *növekvő* sorrendbe rendezi a listát.
(Ékezetes karaktereket, illetve kis- és nagybetűket is tartalmazó szövegekre nem jól működik!)

A `listanév.reverse()` utasítás megfordítja a listaelemek sorrendjét.
A `sort()` és `reverse()` utasításokkal csökkenő sorrendbe is rendezhetjük a listát.

Két listát össze is adhatunk. Az eredmény egy új lista lesz, amely az első lista elemei után tartalmazza a második lista elemeit:

```
lista3=lista1+lista2
```

Egy listáról másolatot készíthetünk a következőképpen:

```
másolat=listanév.copy()
```

Figyelem! A következő utasítás *nem* készít másolatot a listáról, hanem csak egy „becenevet” ad a listának:

```
másolat=listanév
```

A „becenév” azt jelenti, hogy a „másolat” listában végrehajtott összes művelet (változtatás, hozzáfűzés, törlés) az eredeti listában is érvényre jut! (A listanév és a másolat valójában ugyanazt a listát jelöli.)

Egy lista összes elemét kiírhatjuk egyetlen `print` utasítással a következőképpen:

```
print(listanév)
```

Sajnos a fenti utasítás kiírási formátuma általában nem felel meg igényeinknek.
Más formátumban úgy írhatjuk ki a lista elemeit, ha egy ciklussal végiglépkedünk a listán:

```
i=0
while i<len(lista):
    print(lista[i])
    i=i+1
```

A listák bejárása for ciklussal

A listák kényelmesebb bejárására szolgál a ***for ciklus***, azaz a *bejárós ciklus*.
Az alábbi programrészlet ugyanazt csinálja, mint a fenti:

```
for elem in lista:
    print(elem)
```

A bejárós ciklus egyesével végiglépked a lista értékein, az elsőtől az utolsóig. Az épp aktuális értéket bemásolja a ciklusváltozóba (a fenti példában az „elem” nevűbe). A ciklus magjában felhasználhatjuk ezt a változót (a példában kiírjuk). Minden, ami `for` ciklussal megoldható, megoldható `while` ciklussal is, de sok esetben tömörebb, egyszerűbb a programkód, ha `for` ciklust használunk.

A for ciklus – ebben a formájában – azonban nem használható a listában tárolt elemek módosítására:

```
lista=[1,2,3]
for elem in lista:
    elem=elem+1
print(lista)
```

A print utasítás az eredeti számokat jeleníti meg. Ennek az az oka, hogy az „elem” nevű ciklusváltozóba a listaelemek *másolatai* kerülnek, tehát az elem megváltoztatása nincs hatással az eredeti listára.

A fenti feladatot (a listában tárolt számok 1-gyel való növelését) vagy while ciklussal lehet megoldani:

```
x=0
while x<3:
    lista[x]=lista[x]+1
    x=x+1
```

vagy pedig a for ciklus alábbi formájával:

```
for x in range(3):
    lista[x]=lista[x]+1
```

A megoldásban szereplő `range(3)` függvény egy számsorozatot készít, amelyben egész számok találhatók a 0-tól a 2-ig. A for ciklus x ciklusváltozója ezt a számsorozatot „járja be”. Valójában tehát ugyanaz történik, mint a while ciklus esetében, csak kissé tömörebb a kód, kényelmesebb megírni.

Amennyiben a lista hossza nem ismert (vagy változik), akkor használhatjuk a len() függvényt:

```
for x in range(len(lista)):
    lista[x]=lista[x]+1
```

A `range()` függvény arra is lehetőséget ad, hogy ne a 0 számtól indítsuk a számsorozatot, illetve ne egyesével „lépkedjen” a sorozat. Az alábbi példa a kétjegyű páros számokat írja ki:

```
for x in range(10,100,2):
    print(x)
```

A range() függvény első paramétere a számsorozat kezdőértéke, a második paraméter a sorozat vége (de ez a vég már nincs benne a sorozatban), a harmadik paraméter pedig a lépésköz. A lépésköz akár negatív szám is lehet, tehát a kétjegyű páros számokat csökkenő sorrendben a következőképpen írhatjuk ki:

```
for x in range(98,9,-2):
    print(x)
```

A for ciklus nemcsak listák és számsorozatok bejárására alkalmas, szövegeket is végig lehet vele járni, éspedig karakterenként. (A szövegek sok tekintetben úgy viselkednek, mint karakterekből álló listák.)
Az alábbi ciklus egy szöveg karaktereit írja ki, mindegyiket külön sorba:

```
for karakter in szoveg:
    print(karakter)
```

A mátrix adatszerkezet

A listákban nemcsak különböző típusú adatok tárolhatók, hanem akár újabb listák is.
Az ilyen – táblázathoz hasonló – adatszerkezetet *kétdimenziós listának*, vagy *mátrixnak* nevezik.

Az alábbi mátrix emberek adatait tartalmazza. A mátrix egy sorában egy ember adatai szerepelnek.

```
emberek = [["Lajos", "f", 15],
           ["Lujza", "l", 16],
           ["Ubul", "f", 19]]
```

A mátrixban lévő adatok eléréséhez két indexet (sorszámot) kell megadnunk. Az elsőt sorindexnek, a másodikat oszlopindexnek nevezzük. A fent szereplő „Lajos” életkorát a következőképp írathatjuk ki:

```
print(emberek[0][2])
```

Vegyük észre, hogy a mátrix adatszerkezet nagyon hasonlít az Excel táblázataira, illetve a relációs adatbázis-kezelő programok (Pl. MsAccess) adattábláira. (Emelt szintű érettségi feladatokban gyakran érdemes mátrix adatszerkezetet használni.)

A szöveges adatokat tartalmazó listák mátrixként is kezelhetők, amennyiben a listában tárolt szövegek egyes betűit szeretnénk elérni. Például a szavak lista 5-ös indexű elemének első betűjét így írhatjuk ki:

```
print(szavak[5][0])
```

Adatok tárolása szöveges fájlokban

Ha programjainkban hosszabb távon szeretnénk adatokat tárolni, akkor használnunk kell a háttértárakat is. Az adatok a háttértárakon fájlok formájában helyezkednek el. A fájlok nem mások, mint összetartozó adatok csomagjai. A programozásban többnyire ún. *szekvenciális* módon kezelhetjük a fájlokat, ami azt jelenti, hogy a fájlt úgy tekintjük, mint adatok *egymás után elhelyezkedő sorozatát*. A fájlba csak egymás után írhatunk adatokat, mindig a fájl végére, beolvasáskor pedig csak a fájl elejétől kezdve, egymás után olvashatjuk az adatokat. (Nem lehetséges tehát rögtön a fájl közepéről olvasni.)

Az érettségi vizsgán csak ún. *szöveges fájlokat* kell kezelni. Szöveges fájlok alatt most az általában (de nem feltétlenül) .txt kiterjesztésű, Jegyzettömbbel megnyitható és olvasható fájlokat értjük, amik tetszőleges karaktereket, akár számokat is tartalmazhatnak. A szöveges fájlok kezelése viszonylag egyszerű programozási feladat, de – főleg olvasáskor – azért problémákba is ütközhetünk.

Mielőtt nekiállnánk a programozásnak, érdemes megnyitni a beolvasandó fájlokat egy arra alkalmas programmal (pl. Jegyzettömb, Notepad++, Geany) és megnézni a szerkezetüket, különös tekintettel a karakter-kódolásra, az adatok elválasztására és a sortörésekre. Jó ha tudjuk, hogy míg a Linux-alapú rendszerekben egy speciális vezérlő-karakter (LF, a kódja 10) szolgál a sorvégek jelzésére, addig a Windows rendszerekben *két* vezérlő-karakter (CR és LF, kódjuk 13 és 10) használatos. Ezzel főként akkor gyűlhet meg a bajunk, ha a fájlt az egyik rendszerben hoztuk létre és a másikban szeretnénk olvasni. A másik gyakori probléma a karakterkódolás, ami akkor jelentkezik, ha a konzol (paranccsor) és a fájl között mozgatunk ékezetes szövegeket. A komolyabb text-editor programok (pl. Notepad++, Geany) kijelzik a karakter-kódolást és a sorvégeket, továbbá módosítani is lehet mindezeket a segítségükkel.

A fájlkezelés művelete három lépésből áll: Először megnyitjuk a fájlt, utána olvasunk belőle, vagy írunk bele, végül lezárjuk a fájlt. A fájl megnyitásakor adjuk meg a fájl nevét (és elérési útját), továbbá egy változót rendelünk a fájlhoz. A továbbiakban már a változó nevével hivatkozunk a fájlra a programban.

Fájlba *íráshoz* a következő utasítással kell megnyitni a fájlt:

```
fájlváltozó=open("fájlnév","w")
```

Ilyenkor az operációs rendszer létrehozza a háttértáron a megadott nevű fájlt. Ha nem adunk meg elérési utat a fájlhoz, akkor az aktuális mappában, azaz a programunkat tartalmazó mappában fog létrejönni. Amennyiben volt már a mappában ilyen nevű fájl, akkor a benne lévő adatokat törli a rendszer!

Amennyiben elérési utat is adunk meg a fájlnév előtt, akkor figyeljünk arra, hogy az egyes mappákat elválasztó \ jelet *duplán* kell használni! (Pl. C:\\munka\\programozas\\adatok.txt)

Amennyiben magyar ékezetes betűket, vagy egyéb speciális karaktereket tartalmazó fájlt szeretnénk írni, vagy olvasni, akkor megnyitáskor meg kell adnunk a karakter-kódolást:

```
fájlváltozó=open("fájlnév","w",encoding="utf-8")
```

Fájlba írni a képernyőre íráshoz hasonlóan, a print utasítással lehet:

```
print(kiírandó-adat,file=fájlváltozó)
```

A print utasítás szokásos paraméterei (sep, end) a fájlokba íráskor is használhatók.

A fájl használatának befejeztével le kell zárni a fájlt. Ha ezt elmulasztjuk, akkor gyakran előfordul, hogy az utolsó néhány adat nem kerül kiírásra! Ennek az az oka, hogy az operációs rendszerek blokkonként írják az adatokat a háttértárakra. A programból érkező adatokat az operációs rendszer visszatartja addig, amíg egy egész blokknyi össze nem gyűlik. A fájl lezárása arra utasítja az operációs rendszert, hogy írja ki az átmeneti tárolóban visszatartott adatokat.

```
fájlváltozó.close()
```

Létező fájlhoz *hozzáírni* (append) úgy lehet, hogy a kimenő adatfájlt a következőképp nyitjuk meg:

```
fájlváltozó=open("fájlnév","a")
```

Ilyenkor a fájlban lévő korábbi adatok megmaradnak, az újabban kiírt adatok pedig a fájl végére kerülnek.

Fájlból *olvasáshoz* a következő utasítással kell megnyitni a fájlt:

```
fájlváltozó=open("fájlnév","r")
```

Az "r" paraméter akár el is hagyható. (Alapértelmezetten olvasásra nyitja meg a fájlt az open parancs.)

Ha a fájl nem az aktuális könyvtárban (mappában) található, akkor a fájlnév előtt meg kell adni az elérési utat is. Amennyiben a fájl nem létezik (vagy nem ott van, ahol a rendszer keresi), az programhibához vezet!

Szöveges fájlból olvasni többnyire soronként szoktunk. A fájl megnyitásakor a rendszer létrehoz egy „olvasási mutatót”, amely kezdetben a fájl elejére mutat. A következő utasítás hatására a program beolvassa az olvasási mutató utáni karaktereket, egészen a következő sorvége jelig. (CR/LF) Az olvasási mutató ezután a következő sorra áll.

```
szövegváltozó=fájlváltozó.readline()
```

Ha a mutató után már nincs sorvége jel a fájlban, akkor a fájl végéig történik a beolvasás. A beolvasott karaktersorozat a szövegváltozóban kerül tárolásra. A változóba belekerül a sorvége jel is, ami a későbbiekben gondokat okozhat! A következő utasítással megszabadulhatunk a sorvége jeltől:

```
szövegváltozó=szövegváltozó.strip()
```

Sokszor előfordul, hogy a fájl egy sorában több adat is szerepel szóközökkel, vagy egyéb karakterrel elválasztva. A következő utasítás szétbontja az adatsort a szóközök mentén és az egyes adatokat egy listában helyezi el:

```
lista=szövegváltozó.split()
```

Amennyiben nem szóköz, hanem például pontosvessző választja el az adatokat, akkor így használjuk:

```
lista=szövegváltozó.split(";")
```

A fenti három utasítás össze is vonható a következőképpen:

```
lista=fájlváltozó.readline().strip().split()
```

Szöveges fájlból történő olvasáskor mindig *szöveges típusú adatok* kerülnek tárolásra. (Hasonlóképpen az input() utasítással a billentyűzetről történő beolvasáshoz.) Szám típusú adatokat az int(), vagy float() paranccsal át kell alakítani ahhoz, hogy számolni tudjunk velük.

A fájlokból általában nemcsak egy adatsort olvasunk be, hanem az összeset. Ha a program készítésekor (vagy legkésőbb az olvasás elkezdésekor) tudjuk, hogy hány sornyi adat szerepel a fájlban, akkor viszonylag egyszerű dolgunk van: ilyenkor az adatsorokat számlálós ciklus segítségével is beolvashatjuk:

```
for i in range(sorokszáma):
    szövegváltozó=fájlváltozó.readline().strip()
    #az adatsor feldolgozása…
```

Sokszor azonban nem tudjuk, hogy a fájl milyen hosszú. Ilyenkor figyelnünk kell arra, hogy mikor érjük el az utolsó adatsort. A fájl végének elérését például a következő módon lehet ellenőrizni:

```
szövegváltozó=fájlváltozó.readline().strip()       #az első sor beolvasása
while szövegváltozó!="":
    #az adatsor feldolgozása…
    szövegváltozó=fájlváltozó.readline().strip()  #a következő sor beolvasása
```

A fájl végének elérésekor üres szöveget fog beolvasni a program a szövegváltozóba, ezért a ciklus leáll.

Amennyiben a fájl összes sorát ugyanúgy akarjuk beolvasni és feldolgozni, akkor a következő szerkezetet (bejárós ciklus) is használhatjuk:

```
for szövegváltozó in fájlváltozó:
    szövegváltozó=szövegváltozó.strip()
    #az adatsor feldolgozása…
```

Ezzel a ciklussal hasonlóképpen járhatjuk végig a fájl sorait, mint egy lista elemeit.

Saját függvények készítése

A függvény nem más, mint a program egy önállóan működő része, amely a program egy részfeladatának elvégzéséért felelős. Egyszerűbb programokban nem szoktunk saját függvényeket készíteni, komolyabb programokban azonban fontossá válnak. Ennek két oka van:

Egyrészt a program áttekinthetőbbé válik, ha az egyes részfeladatokat végző programrészletek világosan elkülönülnek egymástól. Másrészt függvények használatával elérhető, hogy egy programrészletet többször is felhasználjunk anélkül, hogy újra bele kellene írnunk a programba.

Az alábbi *faktorialis* nevű függvény például az N! (N faktoriális) kiszámítását végzi el:

```
def faktorialis(N):
    x=1
    for i in range(2,N+1):
        x=x*i
    return x
```

A függvényt (a definiálása után) a következőképpen használhatjuk fel a (fő)programban:

```
print("20 faktoriális értéke:",faktorialis(20))
k=int(input("Írjon be egy pozitív egész számot!"))
print(k,"faktoriális értéke: ",faktorialis(k))
```

A fenti függvényben *N* a függvény *paramétere*. A paraméter szolgál arra, hogy a függvény adatokat kaphasson a főprogramtól. Az *x* és *i* a függvény saját (lokális) változói. A függvény a *return* utasítással kapja meg az ún. *visszatérési értékét*, azaz a végeredményt így tudja visszaadni a főprogramnak. A függvényeknek tetszőleges számú és típusú paraméterük lehet, amiket vesszővel elválasztva kell felsorolni, visszatérési értékük viszont csak egy. (Ami azonban lista is lehet!) Készíthetünk olyan függvényt is, amelynek nincs visszatérési értéke. Az ilyen függvényeket *eljárásnak* is szokták nevezni. Ebben az esetben a függvényben nem szerepel *return* utasítás.

A függvények kapcsán fontos kitérnünk a változók *hatókörének* kérdésére. A függvények belsejében létrehozott változók a függvény saját, ún. lokális változói. Ezek a változók csak a függvény belsejében működnek, a függvényen kívül, például a főprogramban, vagy más függvényekben nem érhetők el. Hasonlóképpen a paraméter(ek) sem használható(k) a függvényen kívül. A függvény tehát egy zárt egységet alkot a programon belül. (Többé-kevésbé…)

Felmerülhet bennünk a kérdés, hogy mi történik akkor, ha egy függvényben használt változó neve megegyezik egy a főprogramban használt változó nevével? Ez attól függ, hogy a változót módosítjuk-e a függvény belsejében. Ha nem, akkor a főprogram változójában tárolt értéket látjuk a függvény belsejében is. Amennyiben viszont módosítjuk, akkor egy új lokális változó jön létre, amiben a módosított érték tárolódik, de csak a függvény futásának idejére. Ilyenkor a függvényben használt változó „elfedi” a főprogram változóját. Ez azt jelenti, hogy a függvényen belül nem látszik a főprogram változójának értéke, de megmarad a függvény futásának idején is. A függvény használata után a főprogram változójában újra a függvényhívás előtti érték található.

Amennyiben a függvény egyik paraméterének neve egyezik meg egy a főprogramban használt változó nevével, akkor is ezt figyelhetjük meg. Érdemes kipróbálni az alábbi programrészletet:

```
x=N=99
print("20! értéke:",faktorialis(20))
print(x,N)
```

A faktorialis függvényben lokális változóként használtuk (és módosítottuk) az x változót, illetve N volt a függvény paraméterének neve, az utolsó print utasítás mégis az eredeti értéküket írja ki.

A listák azonban másképp viselkednek ebből a szempontból, mint az egyszerű változók. Ha egy, a főprogramban létrehozott lista elemeit változtatjuk meg egy függvény belsejében, akkor ez a változás maradandó. Egyébként van arra is lehetőség, hogy a főprogram egy egyszerű változóját a függvény maradandóan megváltoztassa, de ennek alkalmazása többnyire nem ajánlott. (A kulcsszó: global.)

Szintén fontos tisztában lennünk a függvények paraméter-átadásának működésével. A Python nyelvben többnyire ún. *érték szerinti* paraméter-átadás történik. Tekintsük az alábbi példát:

```
def novel(p):
    p=p+1
    return p

#főprogram
x=0
print(novel(x))
print(x)
```

A program először kiírja az x változó értékénél 1-gyel nagyobbat, majd kiírja x eredeti értékét. A függvény meghívásakor ugyanis az x változó értéke bemásolódik a p paraméterbe. A függvényen belül a p értéke megnő 1-gyel, de az x értékére ez nincs hatással. Az x értéke tehát marad 0.

Amennyiben viszont listát adunk át paraméterként, akkor ún. *referencia szerinti* paraméter-átadás történik. Ilyenkor a listában tárolt adatokról nem készül másolat, hanem az eredeti lista-adatokat használja a függvény. Ha a függvény módosítja a paraméterét, akkor ez a módosítás az eredeti listában történik!

```
def novel(p):
    for i in range(len(p)):
        p[i]=p[i]+1

#főprogram
x=[0,1,2]
novel(x)
print(x)
```

A fenti program *novel* eljárása a paraméterként kapott lista minden elemét 1-gyel növeli. Az eljárás hívása után lévő print utasítás már a növelt listaelemeket írja ki.

Egyéb hasznos adattípusok

Az alább leírt adattípusok ismerete nem feltétlenül szükséges az emelt szintű érettségi feladatok sikeres megoldásához, de bizonyos esetekben egyszerűbbé tehetik egy-egy részfeladat megoldását. Ezen túl számos helyen találkozhatunk velük, (pl. tankönyvekben) ezért érdemes megismernünk őket.

A tuple adattípus

Ennek az adattípusnak nincs elterjedt magyar neve. Olyan listát jelent, amelynek elemei nem módosíthatók, illetve az elemek sorrendje és száma sem változhat.

Tuple típusú változót így hozhatunk létre:

```
tantargyak = ("magyar", "matematika", "fizika", "kémia")
```

illetve így is:

```
tantargyak = "magyar", "matematika", "fizika", "kémia"
```

Kiíratni a következőképpen tudjuk a tuple első elemét:

```
print(tantargyak[0])
```

A halmaz (set) adattípus

A halmaz adattípus olyan listának tekinthető, amelyben minden elem csak egyszer szerepel (nincsenek benne azonos elemek), továbbá az elemek nem indexelhetők, azaz nincs sorrendjük. Az adattípus tehát megfelel a matematikában megismert halmaz fogalmának. A tuple-hez hasonlóan a halmaz elemeit sem lehet megváltoztatni, de a halmazba lehet betenni és belőle kivenni elemeket.

A halmaz adattípus legfőbb haszna az, hogy:

- egy listából könnyen képezhető halmaz,
- egy halmazhoz egy már benne lévő elemet hozzáadva nem lép fel hiba, de nem is kerül bele,
- könnyen megvalósíthatók vele a matematikából ismert halmazműveletek.

Halmaz típusú változót így hozhatunk létre:

```
tantargyak = {"magyar", "matematika", "fizika", "kémia"}
```

Listából így készíthetünk halmazt:

```
halmaz = set(lista)
```

Üres halmazt pedig így hozhatunk létre:

```
halmaz = set()
```

Figyelem! A következő utasítás nem üres halmazt, hanem szótárat hoz létre:

```
halmaz = {}
```

A halmazhoz elemet hozzáadni az `add` metódussal lehet, eltávolítani pedig a `remove` metódussal:

```
halmaz.add(elem)          halmaz.remove(elem)
```

A matematikából ismert halmazműveleteket a következő operátorokkal tudjuk elvégezni:

metszet:	&	pl: A & B
unió:	|	pl: A | B
különbség:	–	pl: A – B
szimmetrikus differencia:	^	pl: A ^ B

(Ez utóbbi azon elemek halmazát jelenti, amelyek vagy csak az egyik, vagy csak a másik halmazban szerepelnek, de a metszetükben nem.)

A szótár (dictionary) adattípus

Ez az adattípus olyan listának tekinthető, amelynek elemeit nem az indexük, hanem a „kulcsuk” alapján tudjuk elérni. A szótár elemeinek tehát nincs sorrendje, viszont minden elemnek saját kulcsa van.

Például egy ember adatait tartalmazó lista:

```
ember = ["Nagy", "Lajos", 1326]
```

így néz ki szótárként megvalósítva:

```
ember = {"vezetéknév":"Nagy", "keresztnév":"Lajos", "született":1326}
```

Az ember születési évszámát lista esetében így érhetjük el: `ember[2]`
szótár esetében pedig így: `ember["született"]`

A szótárak használata leginkább a többdimenziós adatszerkezetek alkalmazásakor segíti a programozó munkáját. Ha például több ember adatait akarjuk tárolni, akkor a listában lista (azaz mátrix) adatszerkezet használata esetén így érhetjük el a 10. ember születési évszámát:

```
emberek[9][2]
```

listában szótár használata esetén pedig így:

```
emberek[9]["született"]
```

Természetesen mindkét módszer jól használható a személyes adatok tárolására, de a listában szótár használata áttekinthetőbb programot eredményez, illetve csökkenti a „félreindexelésből” származó hibák lehetőségét.

A szótárak használatáról a 10. osztályos digitális kultúra tankönyvben olvashatunk, illetve a sulipy.hu portálon az „Adattípusok” fejezetben.