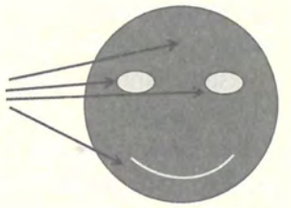
VII. OBJEKTUMORIENTÁLT PROGRAMOZÁS

1. OBJEKTUM ÉS OSZTÁLY

Alapfogalmak

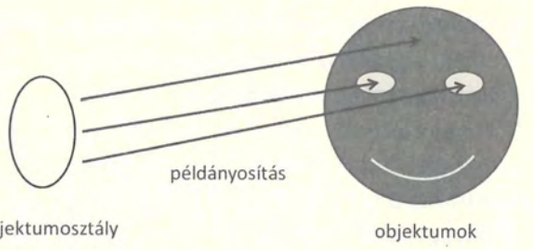
Az **objektum** az egész (pl. program, rajz) egy logikailag jól elkülöníthető része. A mellékelt rajz például összerakható az alábbi jól elkülönülő objektumokból: egy kék (sötétebb) kör, két sárga (világosabb) ellipszis, egy sárga ívdarab.



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

Az objektumoknak vannak jellemző adatai, amiket **tulajdonságoknak** nevezünk. A tulajdonságokat **tagváltozókban** tároljuk.

```
1 class Tortek:
2     pass
3
4 tort = Tortek()
5 tort.nevezo = 6
6 tort.szamlalo = 12
7 print(tort.szamlalo/tort.nevezo)
```

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.

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.*

```
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()
```

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.

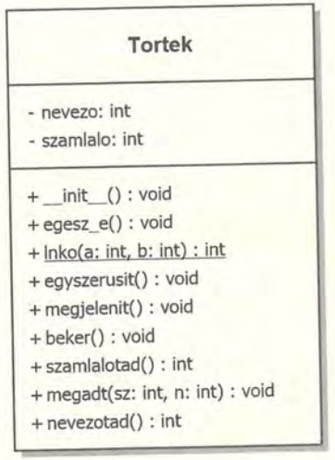
65 A legnagyobb közös osztó meghatározásához ugyanazt a rekurzív algoritmust használjuk itt, ami a Törtek feladatsorban szerepelt.

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



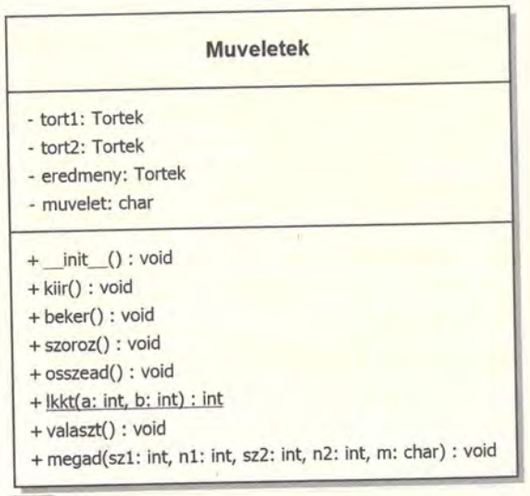
A – jelek azt jelentik, hogy csak az aktuális objektumból hozzáférhető, míg a + azt, hogy máshonnan is elérhető.

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.



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.

```
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 *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.

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.

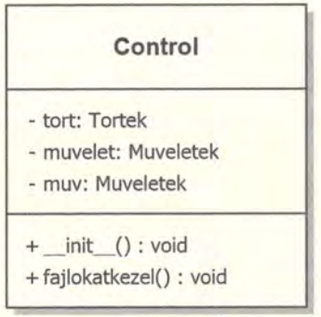
Tekinthetnénk magát a főprogramot a control objektumnak, de szokás inkább külön objektumot létrehozni neki.

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.



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.

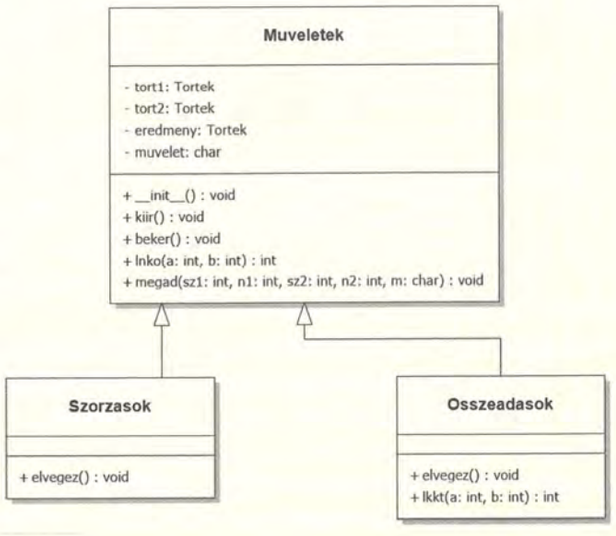
A tageljárás elején létrehozzuk a *muv* nevű listát (152. sor), amibe a ciklus minden egyes lefutásakor, azaz minden új sor beolvasásakor egy-egy üres, *Muveletek* típusú objektumot helyezünk el (161. sor), amit aztán feltöltünk az adatokkal (162–164. sor), majd elvégezzük a műveletet, és a műveletsort 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

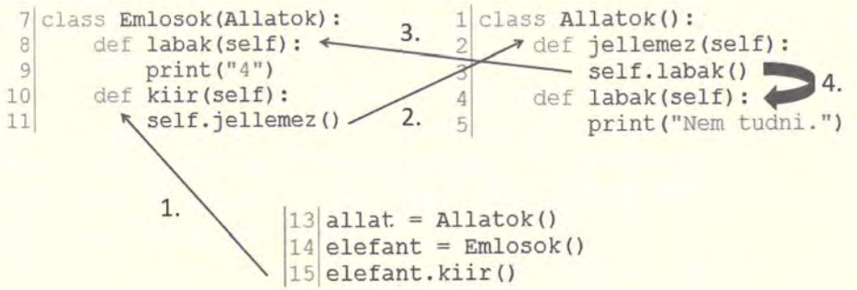
1. olyan metódust *(kiir)* hívunk meg a leszármazott osztály *(Emlosok)* objektumából *(elefant)*, ami csak az ősosztályban *(Allatok)* létezik *(jellemez)*,
2. és az meghív egy olyan metódust, amit mindkét osztályban különbözőre írtunk, de a nevük megegyezik *(labak)*.

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.



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.