Digitális kultúra

Digitális kultúra

11. osztály

Tartalomjegyzék

Géptermi szabályzat

1. A számítástechnika szaktantermekben tanulóknak tartózkodni csak szaktanári engedéllyel lehet.
2. A géptermek szünetekben, használaton kívül zárva tartandók, csak az arra felhatalmazást kapott személyek nyithatják ki.
3. Internethasználat, tanórán kívüli gyakorlás csak felügyelet mellett történhet.
4. Az internetezési, levelezési etikett (Netikett) betartása mindenkire kötelező.
5. Digitális adathordozót csak szaktanári engedéllyel szabad a termekbe behozni.
6. Élelmiszert, rágógumit behozni tilos.
7. Mobiltelefont csak a tanár által meghatározott feladatra szabad használni.
8. A hálózati főkapcsolókat, a központi számítógépeket bekapcsolni; a központi gépekhez nyúlni tanulóknak tilos.
9. A számítástechnikai eszközök belsejébe nyúlni, azokat rongálni anyagi felelősség terhe mellett szigorúan tilos.
10. A számítógépeket bekapcsolni csak a szaktanár engedélyével, az előírásoknak megfelelő módon szabad.
11. Csak a tanár által meghatározott szoftverek használhatók.
12. A szoftverek beállításainak megváltoztatása szigorúan tilos!
13. Ha a foglalkozás ideje alatt a tanuló rendellenességet, a beállítások megváltoztatását, rongálás észlel, ha a számítógépe meghibásodik, azt azonnal köteles jelenteni, ellenkező esetben magára vállalja a felelősséget.
14. A géptermek rendjéért és tisztaságáért, az asztalok, monitorok, gépházak, billentyűzetek, hangfalak (fülhallhatók), egerek állapotáért minden tanuló egyénileg is felelős.
15. Minden tanuló csak a saját helyén ülhet, melyet az ülésrendbe tanév elején be kell jegyezni.
16. A géptermi szabályzat mindenkire kötelező, aki bármelyik terembe belép!
17. Amennyiben valaki a géptermi munkarendet megsérti és ezzel anyagi kárt okoz – akár gondatlanul, akár szándékosan –, a szülő (törvényes képviselő) köteles a kárt a vonatkozó jogszabályok rendelkezése szerint megtéríteni.

Megjegyzések:

A géptermi szabályzat a 21. és 22. terem használatára egyaránt vonatkozik!

7. pont: Alapértelmezésben a termekben mobiltelefont használni tilos. Ez alól kivételt jelentenek azok a feladatok, amelyek során a mobiltelefon használata szükséges a feladat megoldásához. Ezen felül használható mobiltelefon vagy tablet az elektronikus segédanyagok (mint ez a dokumentum) eléréshez is.

12. pont: A szoftverbeállítások megváltoztatásának a tilalma a Windows rendszerre érvényes. A Linux Mint rendszer esetében minden felhasználó saját beállításokkal rendelkezik, így azok szabadon megváltoztathatóak. Azonban a hálózati beállításokat itt sem szabad módosítani, mert az a gép működőképességére lehet hatással!

8. pont: A hálózati főkapcsolókra vonatkozó tilalmat szükség esetén felülírhatja, ha balaset- vagy tűzvédelmi okokból szükséges a terem azonnali áramtalanítása.

Algoritmizálás és programozás

A 9. osztályban tanultak folytatásaként tovább ismerkedünk idén a Python programozási nyelvvel is és annak felhasználásával néhány egyszerűbb, de jól használható algoritmussal is.

Az idei tananyag erősen épít a két évvel ezelőtt tanultakra, így ha nem emlékszel ró, akkor mielőbb érdemes felelevenítened!

A jelenleg szerepelő első két lecke teljesen megegyezik a 9.osztaly.pdf azonos című leckéivel. Ennek az az oka, hogy eredetileg azok nem szerepeltek kilencedik évfolyamon, így van még olyan évfolyam, amelynél a függvények definiálása 11. évfolyamra maradt, de az újonnan érkező 9. évfolyamon ezek már szerepelni fognak. Amint nem lesz ezekre 11. évfolyamon szükség, ez a két lecke eltűnik majd ebből a dokumentumból. Addig azonban teljesen ugyanazzal a tartalommal szerepelnek, mint a kilencedikeseknek szóló dokumentumban – beleértve a feladatokat is. Ez azért lehetséges, mert mindkét dokumentumba ugyanaz az aldokumentum került beillesztésre, ugyanis a dokumentumot a LibreOffice Writer *Fődokumentum* funkciójával hoztam létre.

A témakör nem ér véget ebben a dokumentumban: az emelt szintű érettségire készülők számára készülő külön dokumentumban majd még folytatódni fog ott, ahol ebben a dokumentumban abbamarad. De az objektum-orientált programozás ott sem fog szerepelni, hiszen az még emelt szintű érettségin sem elvárt ismeret…

1. Függvények definiálása és használata

A következő vezérlési szerkezet, amellyel részletesebben megismerkedünk, a függvények, eljárások hívása. Ennek használatához előbb-utóbb hasznos ismerni azt is, hogyan lehet ilyen függvényeket létrehozni (a Python nyelvben nincs külön megkülönböztetve az eljárás). Mielőtt azonban ennek neki kezdenénk, a programfejlesztéssel kapcsolatban is néhány dolgot érdemes közelebbről megismerni.

1.1. Programfejlesztési módszerek

Az elmúlt évtizedek során különféle programfejlesztési módszerek fejlődtek ki. Ezek mind azért alakultak ki, mert egy összetettebb programnál nem fog működni az a megoldás, hogy leülök és begépelem a programot, kipróbálom, majd megkeresem benne a hibát. Általában sem szerencsés ötlet hosszabb programkódot írni annak kipróbálása nélkül.

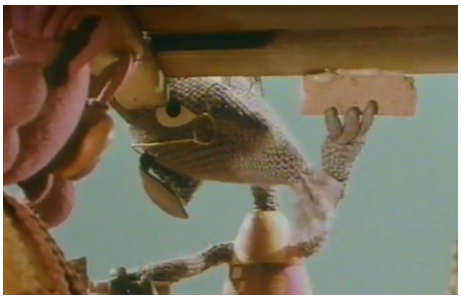
Egy összetettebb feladatot egyszerre át sem lehet már látni. Célszerű kisebb részekre bontani, és azok megoldásával foglalkozni, majd azokból összeépíteni az egész megalkotandó programot.

Ezen az úton többször végighaladva lehet a problémát egyre kisebb részproblémákra bontani, hogy az már átlátható, leprogramozható legyen. Az ilyen részproblémákra írt programkódot szokás függvényekbe helyezni.

Másik oka a feladat részfeladatokra bontásának, hogy a különböző típusalgoritmusokat alkalmazni lehessen. Egy-egy részfeladat megoldása mögött ugyanis sokszor lehet találni egy már pontosan leírt algoritmust. Ebben az esetben csak annyi a feladat, hogy az adott algoritmust alkalmazzuk a konkrét feladatra. A későbbiekben több olyan lecke is fog szerepelni, amely ezek közül az algoritmusok közül mutat be néhány egyszerűbbet – a többi ilyen algoritmust egyetemen tanulják meg a programozónak készülők.

1.1.1. Fentről lefelé történő fejlesztés

Ez a módszer elsőre olyannak tűnik, mint amikor Mekk mester házat épített és először a tetővel kezdte. De azért nem erről van szó!



1. ábra: Mekk mester házat épít felülről lefelé haladva (kép: a rajzfilmsorozat második részéből, az indavido videójából kifényképezve)

A felülről lefelé fejlesztés során először nagy vonalakban megtervezzük a főprogram működését. Ennek során lehet azt is csinálni, hogy a programkódba a későbbi műveleteket megjegyzés formájában leírjuk a saját szavainkkal. Ekkor még csak a nagyobb lépéseket kell leírni.

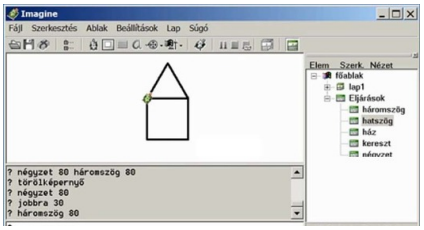
Ezután az egyes műveletek helyére – akár a megjegyzéseket is megtartva, akár azok helyére – függvényhívásokat teszünk. Ezek a függvények ekkor még nincsenek definiálva, tehát a fejlesztőkörnyezet mindre hibát fog jelezni. Ám a modern fejlesztőkörnyezetek erre a hibára javítási lehetőségként valamilyen módon felkínálják általában a hiányzó függvény definiálását.

Ezután az egyes függvényeken belül tovább bontjuk annak a műveletnek a megvalósítását, amelyet az adott függvénynek meg kell valósítania. Eközben ha egy olyan tevékenység szükséges, amelynek leprogramozását az adott helyen még nem akarjuk elvégezni (például mert azt még át kellene gondolni, de túl sok ideig tartana és megakasztaná az adott függvény egészének működését megvalósító kód megírását), akkor ott újra elhelyezhetünk egy még nem létező függvényt meghívó utasítást. Ennek a függvénynek a megvalósítását azután majd a következő körben végezzük el.

Ezzel a módszerrel akárhány körre bontható a program fejlesztése, minden körben egy szinttel részletesebben kidolgozva a program működését, míg végül el nem jutunk a végső programhoz.

A módszer előnye, hogy minden egyes körben csak az adott körnek megfelelő részletességgel kell végiggondolni a teendőket, csak az utolsó szinten kell ténylegesen a program utasításainál szükséges részletességgel megadni a lépéseket.

1.1.2. Alulról felfelé fejlesztés



2. ábra: Imagine Logo

Amikor alulról felfelé halad a fejlesztés, akkor először kis építőelemeket készítünk. Az egyszerű műveletekre írunk egy-egy függvényt, majd ezekből újabb függvényeket állítunk össze, ismételve ezt az összeépítést, amíg a végső programhoz nem jutunk.

Például az Imagine Logo használata esetén ezt a módszert szokás alkalmazni, ott a függvények helyett az eljárás elnevezés használatos. A 2. ábrán látható egy példa (a Mozaik kiadó honlapjáról származik a kép): a négyzet és a háromszög megrajzolását végző eljárásokból épül fel a ház megrajzolását végző függvény. Ezt lehetne folytatni tovább egy utca megrajzolását végző függvénnyel...

Ez a módszer nagyobb előre tervezést igényel, hiszen a kis alapelemek megírásakor tudni kell, hogy milyen műveletekre lesz szükség. Ám ilyenkor könnyebb szétosztani az elkészítendő függvényeket más-más programozókra.

Az alulról felfelé történő fejlesztést támogatja a **programkönyvtárak** fejlesztése. A programkönyvtár olyan *eljárásgyűjtemény*, amely valamely területen potenciálisan előforduló valamennyi műveletet megvalósító eljárást (függvényt) tartalmaz. Ezeket egy program írásakor készen felhasználhatjuk. Amikor valaki programkönyvtárat fejleszt, akkor nem feltétlenül tudhatja előre, milyen programokban fogják azt felhasználni.

Rengeteg példa hozható programkönyvtárakra. Íme ezek közül néhány:

- A Python moduljai tekinthetőek ilyen programkönyvtáraknak. Ezek egy-egy terület műveleteinek megvalósítására szolgáló függvényeket és ezekhez kapcsolódó változókat definiálnak. Pontosabban ennél is többet, de az már az objektum-orientált programozáshoz tartozik...
- A Qt egy C++ nyelven írt programkönyvtár. Elsősorban talán a grafikus felület elkészítéséhez szükséges eszközök miatt használatos, de más lehetőségeket is tartalmaz. Nagyon sok olyan szoftver van, amelynek megírásához használták a Qt valamelyik verzióját. Ilyen például a LibreOffice is. De egész grafikus ablakkezelő rendszereket is meg lehet példának említeni, amelyek a Qt programkönyvtárát használják, mint a KDE vagy a Deepin. A Qt-hez a Python is kínál hozzáférést specális modulokon keresztül.
- A Microsoft által a C#-hoz kifejlesztett (de már nem csak a C#-ban használható) .NET is gyakorlatilag tekinthető ilyen programkönyvtárnak, amellett, hogy ez egy futtató környezet is (azaz egy interpreter a .NET által használt közbülső nyelvre lefordított, például C#-ban megírt programoknak).

A két fent bemutatott programfejlesztési módszert a gyakorlatban nagyon ritkán használják kizárólagosan. Inkább a kettő keverékét – esetleg további módszerekkel együtt – szokás használni.

Például a fentebb említett programkönyvtárakat már mint létező eljárásokat beépítve egy egyebként alapvetően fentről lefelé haladva fejlesztett programba, máris keverve használjuk a két módszert. Ezen felül ha alapvetően felülről lefele halad is a program megírása, egy olyan függvényt, amelyet az egyik ponton kellett definiálni, már kész építőelemként lehet máshol is felhasználni, ahol az szükséges.

1.2. Függvények definiálása

A függvények arra is használhatóak, hogy ugyanazt a programkódot ne kelljen egy programban többször leírni. Amennyiben ugyanis ugyanaz a kód többször szerepel a programban és utólag módosítani kell valamiért, akkor mindenhol módosítani kell. **Tehát ha egy programban több helyen akarod ugyanazt a műveletsort szerepeltetni, akkor írd meg függvényként és csak a függvényt hívd meg a programban!** Ha az egyes helyeken nem ugyanazokkal az adatokkal kell ugyanazt a műveletsort elvégezni, akkor is: a függvények paraméterezhetőek.

Valójában a függvény úgy tekinthető, mint egy algoritmus megvalósítása. Az algoritmus bemeneti adatai a függvény *paraméterei*, kimenete pedig a függvény *visszatérési értéke*. Gyakorlatilag a **print()** függvény is egy algoritmust valósít meg, amelynek bemenete a függvény paramétereként megadott kifejezések sorozata, a fenti értelemben vett kimenete pedig nincs. Valójában van: a képernyőre kiírt szöveg. Az **input()** függvénynek kimenete is van: a felhasználó által begépelt szöveg.

A függvényt egyszer kell definiálni a programban, majd mindenhol, ahol „látszik”, meghívható. Python nyelv esetén a függvények definíciója mindig a program elején, a „főprogram” utasításai előtt szerepeljenek!

A definíciót egy `def` kulcsszóval kell kezdeni. A `def` kulcsszó után a definiálandó függvény neve, majd a *paraméterlistája* szerepel. Ezt a sort egy kettőspont zárja, amiből már következik, hogy a következő sortól indul a függvénydefiníció blokkja – bentebbkezdéssel.

```
5  def nagyobb(a: int, b: int) -> int:
6      """
7      Ez a függvény a két paraméter közül a nagyobb értékét adja vissza.
8      :param a: Az egyik egész szám.
9      :param b: A másik egész szám.
10     :return: A két egész szám közül a nagyobb érték.
11     """
12     if a < b:
13         return b
14     else:
15         return a
```

Ez a programrészlet egy függvénydefiníciót mutat. Az 5. sorban látható a függvénydefiníció-fejléce az előző bekezdésben leírtaknak megfelelően, a többi sor a hozzá tartozó blokkot tartalmazza. A fejlécben a `def` után szerepel a függvény neve, amellyel a későbbiekben meg lehet hívni a függvényt. Ezután kerek zárójelek között szerepel a paraméterlista – vagy más néven argumentumlista[1]. Ez a példában bővebb a ténylegesen kötelezőnél.

A paraméterlista lehet üres is, de mindenképpen szerepelnie kell. Üres paraméterlista esetén a függvény meghívásakor nem kap paramétert, de ott is szerepeltetni kell az üres paraméterlistát, mivel a kerek zárójelekről fogja tudni az értelmező, hogy függvény hívása történik, nem egy változó megadása – ugyanis a függvény nevével, mint változóval is lehetne bizonyos esetekben kifejezést megadni, de ezzel nem fogunk foglalkozni.

A fenti példában a függvénynek két paramétere lesz, mindkettő esetében meg van határozva, hogy az egész típusú legyen. A paraméterek értékeit a függvény definíciós blokkjában változóként lehet használni azzal a névvel, amellyel a definíció fejlécében szerepelnek. Az egyes paraméterek tí-

1 A függvénydefinícióban szereplő paraméterlista neve hivatalosan argumentumlista, míg a függvénymeghívásakor szereplő kifejezések a paraméterek, de egyszerűbb mindkét esetben paraméterekről beszélni.

pusát a Python nyelv esetében nem kötelező megadni, más nyelvek esetében azonban ez kötelező lehet. Annak ellenére érdemes azonban megadni a típust, hogy nem kötelező. Ekkor ugyanis a fejlesztő környezet figyelmeztethet, ha a függvény meghívásakor nem a megfelelő típusú értéket adjuk a paraméternek. Ha megadjuk a típust, akkor a paraméter nevét követően egy kettőspontot követően kell azt megadni, ahogy a példában is szerepel.

Hasonlóan elhagyható rész a paraméterlista utáni `->` és utána egy típusnév. Ez a függvény által visszaadott érték várható típusát adja meg. Vannak esetek, amikor a függvény nem minden esetben ugyanolyan típusú értéket ad vissza, amire ilyenkor a fejlesztőkörnyezet figyelmeztetni szokott. Kicsit lentebb lesz erre majd példa.

A függvény szokásos dokumentálásának a PyCharm által felkínált módját mutatja a 6–11. sor megjegyzése. A PyCharm használata esetén a kezdő tripla idézőjel (szimpla vagy dupla esetén is) begépelésére automatikusan létrejön. A VS Code nem feltétlenül hozza automatikusan létre és ha igen, akkor nem ilyen alakban, de mégis mindig ezt a formát érdemes követni, mivel ebből a megjegyzésből már a fejlesztőkörnyezetben is elérhető dokumentáció készül, amely például a VS Code esetén a függvényhívás beírásakor is megjelenik egy buborékban. Emellett vannak automatikus dokumentumkészítő szoftverek, amelyek ebből a megjegyzésből készítik el a dokumentációt.

1. A függvény rövid ismertetését adjuk meg először.
2. Ezután minden egyes paraméterre a `:param` *`név`*`:` kezdettel adjuk meg az adott paraméter funkcióját.
3. A végén a `:return:` kezdetű sorral lehet megadni, hogyan kell a függvény által visszaadott értéket felhasználni.

1.2.1. A függvény visszatérési értéke

Vannak olyan programozási nyelvek, amelyek elnevezésben különbséget tesznek eljárás és függvény között. Ilyen például a Pascal nyelv, ahol az eljárás nem ad vissza értéket, meghívása egy utasításként történik, míg a függvény értéket ad vissza az őt tartalmazó kifejezésbe, ahova a visszatérési értéket kell a függvényhívás helyére behelyettesíteni. A Python – ahogyan a C nyelvcsalád – nem tesz különséget, mivel bárhol ahol utasítás szerepelhet, ott kifejezés is megadható akár értékadás nélkül is, amely esetben a kifejezés értéke egyszerűen figyelmen kívül lesz hagyva. (Ám ha a kifejezés értékének meghatározása közben történik valami *mellékhatás*, akkor az megtörténik).

A függvény végrehajtása során utoljára mindig egy **return** utasítás hajtódik végre. Ez akkor is igaz, ha ez az utasítás nem szerepel a függvényt definiáló blokk végén. Ekkor az ott szereplő utasítás után automatikusan „odaképzeli” azt az értelmező. Ha viszont szerepel, akkor utána egy kifejezést lehet megadni. Ennek a kifejezésnek az értéke lesz a függvény által visszaadott érték.

Bárhol szerepelhet a **return** utasítás a függvény definíciójában, ahogy a példánkban is látható a 13. és a 15. sorban. Ahol az utasítás szerepel, ott véget ér a függvény és az utasításban szereplő kifejezés értékét adja vissza oda, ahol a függvény meghívása történt.

Amennyiben a **return** utasítás hiányzik vagy szerepel, de kifejezés nélkül, a függvény akkor is ad vissza értéket, de ez a speciális `None` érték lesz. Ez az érték a *NoneType* típus egyetlen lehetséges értéke. Ez eltárolható változóban is, vizsgálható az `is None` és az `is not None` logikai kifejezésekkel. Ilyen módon lehetőség van arra is, hogy ezt a „nem értéket” hibajelzésre használjuk: ha a függvény a végrehajtása során hibába ütközött, akkor a `None` értéket visszaadva ezt a tényt vizsgálva a függvényhívás után, a hibát a program kezelni tudja.

Nézzük erre példaként az osztás esetét! Amennyiben egy osztást tartalmazó kifejezésben az osztó nulla, akkor a program hibaüzenettel elszáll, hiszen a nullával való osztás nem értelmezett. Ezt elkerülendő készítsünk függvényt, amely úgy végzi el az osztást, hogy ha az osztó nulla, akkor a program nem ér véget hibaüzenettel, de a függvény hívásának helyén ez leellenőrizhető.

Ilyen egyszerű műveletekre azért nem érdemes a valóságban függvényt írni, mert a függvényhíváshoz szükséges extra tevékenység sokkal több időt igényel, mint magának a függvény által végrehajtott osztásnak a végrehajtása. A függvény csak a példa kedvéért szerepel, bonyolultabb esetekben a módszer tényleg hasznos lehet.

```
18  def osztas(osztando: float, oszto: float) -> float:
19      """
20      A függvény elvégzi a valós osztást, anélkül, hogy a nullával történő osztás hibát okozna.
21      :param osztando: Az elosztandó lebegőpontos szám.
22      :param oszto: Az osztó, lebegőpontos számként.
23      :return: Az osztás eredménye lebegőpontos számként vagy None, ha az osztó 0.
24      """
25      if oszto == 0:
26          return None
27      return osztando / oszto
```

A fenti kódrészlet 26. sorában a `None` szóra a PyCharm, amelyben a kép készült, azt jelzi, hogy a kifejezés típusa nem felel meg a fejlécben a visszatérési értéknek megadott típusnak. Ha ezt a figyelmeztetést (*Warning*) el akarjuk kerülni, akkor vagy a `None` szerepeltetését kell elhagyni – akkor nincs mit más színnel kiemelve jelezni a problémát –, vagy a visszatérési érték típusát nem kell megadni.

A fenti függény először ellenőrzi az osztó paraméter értékét. Ha ez nulla, akkor nem engedi az osztást végrehajtani, hanem a `None` értéket hibajelzésként használva véget ér a függvény.

Mivel nem garantált, hogy az osztás végrehajtódott, a függvény hívását nem szabad ott elhelyezni, ahol az osztás értékére szükség van, hanem előtte ellenőrizni kell azt. Viszont hogy ne kelljen fölöslegesen kétszer meghívni ugyanazokkal a paraméterekkel a függvényt, a függvény értékét el kell tárolni egy változóban. Ezt a változót megvizsgálva azután ellenőrizhető, hogy a függvénynek van-e felhasználható értéke. Az alábbi kódrészlet a fenti függvény helyes meghívására mutat példát:

```
hanyados = osztas(a, b)
if hanyados is None:
    print("Nullával nem lehet osztani!")
else:
    print(hanyados)
```

Nagyon fontos megjegyezni, hogy a **return** utasítás bárhol szerepel is a függvény kódjában, ott véget ér a függvény végrehajtása! Amennyiben egy ciklus magjában szerepel, akkor a ciklus is azonnal véget ér, ahogyan a **break** utasítás befejezné azt, majd rögtön a függvény is véget ér. Amennyiben a függvény kódjában a **return** után is írunk utasításokat, azok soha nem fognak végrehajtódni. Ezeket az utasításokat a fejlesztőkörnyezetek esetleg más megjelenéssel jelzik is.

1.2.2. Több érték kezelése egy értékként

Készítsünk függvényt, amely a paraméterében szereplő *a*, *b*, *c* lebegőpontos számokból kiszámolja az *a*, *b*, *c* oldalú háromszög kerületét, illetve területét! Amennyiben a megadott értékekből nem szerkeszthető háromszög, akkor `None` legyen a visszaadott érték!

Első lépésben készítsük el a két értéket kiszámoló függvényt külön-külön: a kerületet számoló függvény neve **kerulet()** legyen, a területet számolóé **terulet()**.

$$K = a + b + c$$

$$T = \sqrt{s(s-a)(s-b)(s-c)}$$

$$s = \frac{a+b+c}{2}$$

3. ábra: Háromszög kerülete és területe (Hérón-képlet)

A számoláshoz szükséges képletek a 3. ábrán láthatóak. Mint az ábrán látható, a Hérón-képlethez, amely a háromszög oldalaiból tudja kiszámolni annak területét, szükség lesz a négyzetgyökvonást tartalmazó *math* modulra, így annak beimportálását a függvények definíciója előtt el kell helyezni:

```
6  import math
```

Ezután következhet a kerület kiszámítását végző függvény, ami a 4. ábrán látható.

```
9  def kerulet(a: float, b: float, c:float) -> float:
10     """
11     A függvény a háromszög kerületét számolja ki.
12     :param a: A háromszög egyik oldalának hossza.
13     :param b: A háromszög másik oldalának hossza.
14     :param c:: A háromszög harmadik oldalának hossza.
15     :return: A háromszög kerülete, ha megszerkeszthető a háromszög, különben None.
16     """
17     if a <= 0 or b <= 0 or c <= 0:
18         return
19     if a <= b + c or b <= a + c or c <= a + b:
20         return
21     return a + b + c
22
```

4. ábra: A kerulet() függvény

A 9. sorban a már megismert módon kezdődik a függvénydefiníció fejléce. Ezután a 10–16. sorban a függvény dokumentációja látható – amit a VS Code valamiért nem hajlandó megjegyzésként színezni a képen.

A 17–20. sorok tesztelik a háromszög-egyenlőtlenség teljesülését. (A három oldalnak határozottan pozitív értékűnek kell lennie úgy, hogy bármely két oldal összegének nagyobbnak kell lennie a harmadik oldal hosszánál.) Amennyiben bármelyik feltétel nem teljesül, akkor a függvény érték visszaadása nélkül kilép.

Ezután egyetlen teendő marad: kiszámolni a kerületet és visszaadni azt. Ez egyetlen utasítással megtörténik a 21. sorban.

Ezután a terület kiszámítása következik. Azonban ne feledjük, hogy a Hérón-képletben szerepel a kerület fele. Vagyis ebben a függvényben is ki kellene számolni a kerületet. Ha ehelyett meghívjuk a **kerulet()** függvényt, akkor az elvégzi a háromszög-egyenlőtlenség teljesülésére vonatkozó vizsgálatot is, így annak a kódját sem kell újra leírni. Így ha esetleg a vizsgálatot elírjuk – mint én a mintaprogram első változatánál –, akkor csak egy helyen kell azt javítani.

A t**erulet()** függvény kódját mutatja a 5. ábra. Ennek első utasítása, a 32. sorban elintézi a **kerulet()** függvény meghívását. Amire itt figyelni kell, hogy nem lehet a kerület eredményét egy *kerulet* nevű változóban eltárolni! Ennek az az oka, hogy ezen a néven már létezik egy függvény – történetesen az, amit meg kell hívni. Ha a változó és a függvény neve megegyezne, abból később bemutatandó problémák adódhatnának.

```
23
24  def terulet(a: float, b: float, c:float) -> float:
25      """
26      A függvény a háromszög területét számolja ki.
27      :param a: A háromszög egyik oldalának hossza.
28      :param b: A háromszög másik oldalának hossza.
29      :param c:: A háromszög harmadik oldalának hossza.
30      :return: A háromszög területe, ha megszerkeszthető a háromszög, különben None.
31      """
32      K = kerulet(a, b, c)
33      if K is None:
34          return
35      s = K / 2
36      return math.sqrt(s * (s-a) * (s-b) * (s-c))
37
```

5. ábra: A terulet() függvény

A következő **if** utasítás megvizsgálja, hogy a **kerulet()** függvénynek sikerült-e a kerület kiszámítása, ha nem, akkor a függvény kilép.

Ezután a kerület felének kiszámítása, majd a terület értékének kiszámítása és visszaadása van már csak hátra.

A fenti feladatban szereplő függvények helyett egy függvényt is lehetne készíteni, amely kiszámolja egyszerre a háromszög területét és kerületét is, majd mindkét értéket visszaadja eredményül. Ehhez egyetlen akadály, hogy egy függvény csak egy értéket tud visszaadni. Ám ez az érték bármilyen típusú lehet.

Az ilyen esetekre, amikor több értéket kell ott szerepeltetni, ahol csak egy szerepel, a Python ismeri a **tuple** adattípust. Ez valójában különböző értékek felsorolása egyetlen elemként. Amennyiben egyértelműsíteni kell, akkor a *tuple* részét képező értékeket egy kerek zárójelben fogjuk össze:

```
>>> a = ("alma", 2)
>>> type(a)
<class 'tuple'>
>>>
```

6. ábra: A tuple típus

Az `(1, 42, False)` egy értékhármas, azaz három elemű *tuple* (*triple*).

A **return** utasítás kifejezése is lehet ilyen *tuple* – a mi esetünkben két érték, azaz értékpár. Ráadásul egy ilyen *tuple* segítségével egyszerre több változónak is értéket lehet adni:

elso, masodik = (42, True)

Ennél az értékadásnál valójában mindkét oldalon egy-egy kételemű *tuple* található. A baloldalon nem szerepel a zárójel, csak a két változó listája, a jobboldalon szerepel a zárójel. De a jobboldalon is elhagyható lenne. A fenti értékadásban az első érték az első változóba, a második érték a második változóba kerül.

Tehát ha a függvény két értéket ad vissza, akkor eleve két változónak is értékül adható a függvény értéke. Ebben az esetben a változók rendre megkapják az értékeket. De lehetséges az is, hogy egy változó kapja értékül a függvény értékeit, amely ekkor *tuple* típusú lesz. Ekkor a később a listáknál megismert indexeléssel lehet elérni az értékeket. (A következő oldalon levő példából megtudod, hogy hogyan is működik ez.)

A 7. ábra mutatja a **kerulet_terulet()** függvény kódját, amely egyben kiszámolja a paraméterében megadott adatokból az azokkal szerkeszthető háromszög kerületét és területét is.

```
38  def kerulet_terulet(a: float, b: float, c:float) -> float:
39      """
40      A függvény a háromszög kerületét és területét számolja ki.
41      :param a: A háromszög egyik oldalának hossza.
42      :param b: A háromszög másik oldalának hossza.
43      :param c:: A háromszög harmadik oldalának hossza.
44      :return: A háromszög kerületé és területét tartalmazó tuple,
45      ha megszerkeszthető a háromszög, különben None.
46      """
47      if a <= 0 or b <= 0 or c <= 0:
48          return
49      if a <= b + c or b <= a + c or c <= a + b:
50          return
51      K = a + b + c
52      s = K / 2
53      T = math.sqrt(s * (s-a) * (s-b) * (s-c))
54      return (K, T)
55
```

7. ábra: A kerulet_terulet() függvény

Ebben a függvényben ismét szerepel a háromszög-egyenlőtlenség ellenőrzése, mivel nem akarjuk a kerület kiszámítását a **kerulet()** függvénnyel végeztetni. Egyrészt lehet, hogy ebben az esetben nincsenek definiálva a külön-külön számításokat végző függvények. Másrészt az már túl sok függvényhívást eredményezne a kiszámolandó értékekhez képest, ha ez a függvény meghívná külön-külön a másik két függvényt.

Az újdonság az 54. sorban van, de ennek magyarázata már szerepelt az előző bekezdésekben.

A függvény által visszaadott értéket a fentebb írtak alapján lehetne a 8. ábra 57. sorában látható módon, eleve két változóban eltárolni. Azonban ez programhibát fog okozni minden olyen esetben, ha a függvény paramétereivel nem lehet háromszöget szerkeszteni!

```
57  k, t = kerulet_terulet(3, 4, 5)
58  print(t)
59  print(k)
60
61  eredmeny = kerulet_terulet(3, 5, 0)
62  if eredmeny is None:
63      print("Nem szerkeszthető háromszög!")
64  else:
65      print("kerület:", eredmeny[0])
66      print("terület:", eredmeny[1])
67
```

8. ábra: A tuple adattípusú érték felhasználási lehetőségei

Ebben az esetben ugyanis a függvény nem egy értékpárt, hanem a `None` értéket adja vissza, amit viszont nem lehet egyszerre két változóba elhelyezni.

Az 57. sorban látható megoldás csak akkor használható, ha pontosan annyi értéket fog adni a függvény, ahány változóba kell beletenni. **Sem több, sem kevesebb érték nem lehet!**

A mi esetünkben a lehetséges hibajelzés miatt a 61–66. sorban látható megoldást kell alkalmazni. Ekkor egy *eredmeny* nevű változóba kerül a *tuple* érték. Ennek első, illetve második elemét a 65. és a 66. sorban látható módon lehet elérni.

```
>>> type((42))
<class 'int'>
>>> type((42,))
<class 'tuple'>
```

9. ábra

A *tuple* típusra visszatérve érdemes még megjegyezni, hogy a típus lehet egyelemű is. Ahogyan az a 9. ábrán látható, az – ekkor kötelező – zárójelen belül az egy érték után egy vesszőt kell tenni. Enélkül a vessző nélkül a kifejezés egy zárójelbe tett szám lenne, ami a példa esetében *int* érték lenne.

1.1. Készíts függvény, amely paraméterként megkapja a derékszögű háromszög két befogójának hosszát (valós számként) és visszaadja az átfogó hosszát! Amennyiben a kapott két paraméter nem pozitív szám, akkor a függvény adja vissza a None értéket!

1.2. Készíts függvényt, ami az elágazásoknál szerepelt feladathoz hasonlóan a paraméterül kapott egész szám alapján könyvet ajánl. A függvény neve `konyvajanlo` legyen. Az alábbiak szerint alakuljon a függvény eredménye:

1. 0–3: „Totyogóknak a kettes számrendszerről”
2. 4–6: „Hackeljük meg az óvodát!”
3. 7–14: „Felhőtechnológia a menzán”
4. 15–18: „Big data a középiskolában”
5. 19–: „Donald Knuth: A számítógép-programozás művészete”

1.3. Készíts függvényt, amely egy egész számról eldönti, hogy egy másik egész számmal eloszthatóe!

A függvény első paramétere a vizsgált szám legyen, a második paramétere pedig az a szám, amellyel való oszthatóságot kell vizsgálni. Például ha azt akarjuk vizsgálni, hogy a 15 osztható-e néggyel, akkor az `oszthato(15, 4)` függvényhívásra legyen szükség.

A függvény False értéket adjon vissza, ha nem teljesül az oszthatóság, True értéket, ha teljesül.

1.4. Készíts függvényt, amely a bemenő paramétereiben óra, perc, másodperc alakban megadott időadatról megmondja, hogy az összesen hány másodperc!

1.5. Készíts függvényt, amely a paraméterében megadott másodperceket felbontja óra, perc, másodperc alakba és ezt egy számhármasként adja vissza!

1.3. Opcionális paraméterek

Előfordulhat, hogy úgy szeretnénk definiálni egy függvényt, hogy annak bizonyos paramétereit ne legyen kötelező megadni. Erre a legjobb példa a **print()** függvény, amelynek van két ilyen paramétere, az *end* és a *sep*. Az ilyen paramétereknek mindig van egy alapértelmezett értéke arra az esetre, ha a függvény meghívásakor nem szerepelnek. Megnézve a **print()** definícióját, látható, hogy ezeket hogyan kell megadni. A paraméter neve után egy egyenlőségjel után meg kell adni az alapértelmezett értékét is. Az ilyen paramétereknek a paraméterlista végén, az alapértelmezett értékkel nem rendelkező paraméterek után kell szerepelniük.

A függvény meghívása során ezek a paraméterek bármilyen sorrendben szerepelhetnek, ugyanis ahhoz, hogy az értelmező tudja, melyik kifejezés melyik paraméter értéke lesz, a hívásnál meg kell adni a paraméter nevét és egy egyenlőségjelet a kifejezés előtt.

Igazából az alapértelmezett értékkel nem rendelkező paraméterek neve is megadható ilyen módon, így nem kell azokat sem a definíciónál megadott sorrendben megadni ezzel a módszerrel élve.

A függvény definiálásánál az alapértelmezett értékkel nem rendelkező paramétereket kell először megadni. Ezután jöhetnek olyan paraméterek, amelyeknek alapértelmezett értéke is van. Ezt az alapértelmezett értéket úgy kell megadni, hogy a paraméter neve után egy egyenlőségjelet követően szerepel az alapértelmezett érték. Ez valójában olyan, mintha egy értékadás lenne, amely így a függvény fejlécében értéket ad a paraméternek, amelyet akkor kap, ha a függvény hívása nem adott a paraméternek más értéket.

1.4. Változók láthatósága

A függvények kapcsán felmerül a kérdés, hogy egy adott nevű változó (vagy általában valamilyen objektum) a program melyik részében érhető el. Ebből a szempontból megkülönböztetünk *globális* és *lokális* változókat.

A függvény paraméterei valamint azok a változók, amelyek a függvénydefiníció blokkjában kapnak értéket, **lokális változók**. Ezek csak az adott függvény kódjának futása során léteznek, csak ott érhetőek el.

Azok a változók, amelyek minden függvényen kívül jöttek létre, a **globális változók**. Ezek elvileg mindenhol, még a függvények kódján belül is láthatóak, értékük elérhető.

Amikor a program elindul, létrejön egy *globális adatterület*. Minden objektum – változó, függvény stb. – ami a függvényeken kívül van létrehozva, ebbe a globális adatterületbe kerül. Ez a globális adatterület a program befejezéséig létezik, majd ekkor megszűnik. Ha konkrétan nem törlünk a programban egy változót a **del** utasítással, akkor a globális adatterület létezéséig az létezni fog.

Nem érdemes nagyon elmélyedni abba, hogy hogyan történik egy függvény tényleges meghívása, legalábbis a rekurzióról szóló fejezet előtt. Azt azonban érdemes megjegyezni, hogy minden függvényhívás a memória egy *verem* nevű területén egy memóriaterület lefoglalását igényli. Ebben a memóriablokkban kerülnek a függvény paraméterei és lokális változói is elhelyezésre. Mivel ez a blokk a függvény befejezése után felszabadul, a lokális változók a függvény befejezésével fizikailag is megszűnnek létezni.

Ezzel ellentétben minden függvényhívás létrehoz egy *lokális adatterületet* – lásd a fenti megjegyzést –, amely a függvény befejezéséig létezik. A függvényhez tartozó változók és a függvény paraméterei abba lokális adatterületbe kerülnek, és az adatterület megszűnésével együtt megsemmisülnek.

Egyszerre több lokális adatterület is lehet: amennyiben az egyik függvény egy másikat hív, akkor a belső függvényhívás is új lokális adatterületet hoz létre.

1. A függvényeken kívüli kódban nem lehet használni egyetlen lokális változót sem, hiszen azok ott nem léteznek.
2. Azonban egy függvény kódjában a globális változókhoz is hozzá lehet férni.
3. Egy függvény kódja más lokális adatterületen levő változókhoz nem fér hozzá, csak a sajátjához (és a globális változókhoz).
4. Különböző adatterületeken lehet ugyanolyan nevű változót létrehozni.

A fenti pontok közül az utolsónak lehetnek nehezen felderíthető szemantikai hibát eredményező következményei!

Mivel a függvények a külvilággal csak a paraméterein és a visszatérési értékein keresztül tudnak kommunikálni – hiszen a lokális változói kívül nem elérhetőek – így kiszűrhetők az olyan rejté-

lyesnek tűnő problémák, hogy egy változó értéke minden látható ok miatt megváltozik a program futása során.

Amennyiben rászokunk arra, hogy egy függvényen belül nem használunk globális változót, csak a saját változókat, akkor még abból sem adódhat baj, ha valamelyik lokális változó neve megegyezik egy globális változóéval. Az ilyen esetekben ugyanis – bár alapból a globális változók láthatóak lennének a függvényen belül – az azonos nevű globális változót a lokális változó el fogja fedni.

Bár a globális változók látszanak a függvényen belül, azokat alapesetben nem lehet módosítani, mivel egy értékadás a globális változóval azonos nevű lokális változót hoz létre. Vannak azonban olyan esetek, amikor ez konkrétan szükséges. Például amikor egy globális változót arra használ a program, hogy egy függvény által elvégzett művelet kapcsán megszámolja, hogy az hányszor hajtódik végre (vagy lesznek még később egyéb példák is ilyen esetre). Ekkor ugyanis egy olyan értékről van szó, amit a függvény többszöri meghívásai között meg kellene őrizni.

Erre használatos a **global** utasítás, amely után annak a változónak a nevét kell megadni, amely ugyan globális, de a lokális változókhoz hasonlóan módosítania kell az értékét a függvénynek.

```
def spam():
    global eggs
    eggs = 'spam'

eggs = 'global'
spam()
print(eggs)
```

A fenti programkód a „spam” szöveget fogja kiírni, mert a függvény első utasítása engedélyezi az eggs globális változó értékének módosítását a függvény további része számára.

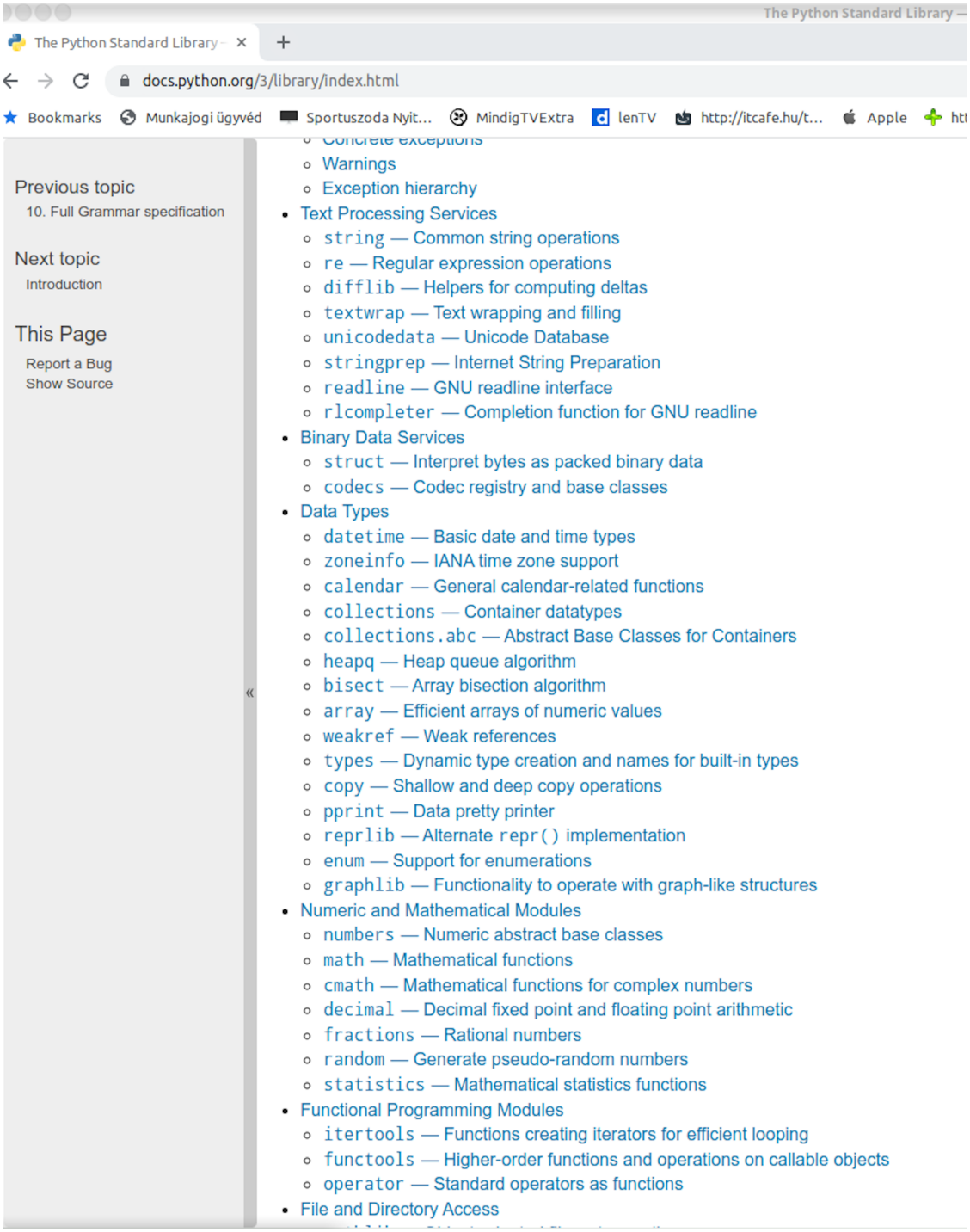
A **global** utasítás nélkül a következő történne:

1. A függvényen belül létrejönne az *eggs* lokális változó „spam” értékkel.
2. A függvény befejezésekor megszűnik a lokális *eggs* változó.
3. Innentől ismét a globális változó válik elérhetővé, amelynek értéke „global”, amit a **print()** ki is ír.

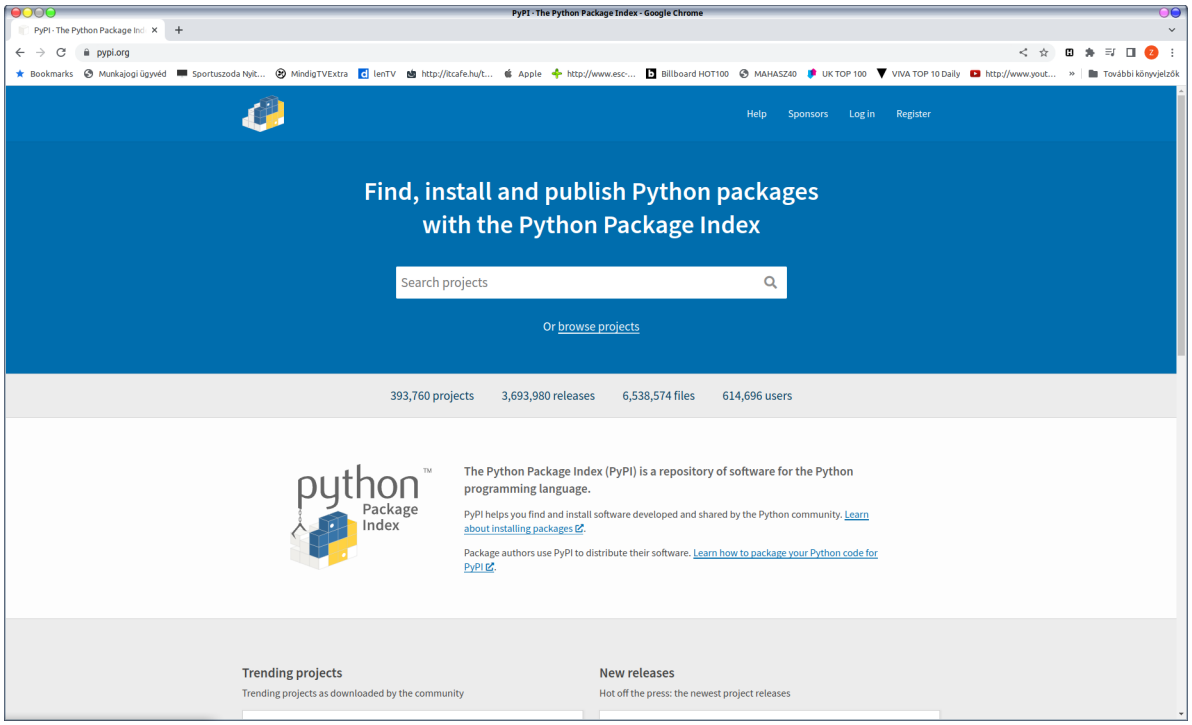
2. Modulok használata

A függvényekről szóló leckében már említett alulról felfelé fejlesztés során létrehozott függvénykönyvtárak létrehozását teszi lehetővé a Python nyelv esetében a modulok használata.

A 10. ábra egy részletet mutat a Python hivatalos dokumentációjából, ahol a külön telepítést nem igénylő modulok felhasználási terület szerint csoportosítva megtalálhatóak. Ezek az adott modul dokumentációját elég részletesen tartalmazzák angolul – az angol nyelv az informatikában kike-



10. ábra: A modulok téma szerinti listája a Python dokumentációjában



11. ábra: A pypi.org, a Python modulok hivatalos adatbázisának nyitóoldala

rülhetetlen. Néhány modullal ezek körül meg fogsz majd te is ismerkedni. Azonban nem minden modul szerepel ebben a dokumentációban sem a Python programokban használható modulok közül. Ezek csupán azok, amelyek egy jól feltelepített Python értelmező esetében biztosan rendelkezésre állnak.

Létezik azonban ezen felül egy hivatalos adatbázis (11. ábra), amelyből a Python által szabványos módon biztosított, további modulokat lehet feltelepíteni a számítógépünkre, amelyek onnantól kezdve szintén szabadon használhatóak – az adott számítógépen – egy programban.

Továbbá vannak olyan Python modulok is, amelyek nem a hivatalos adatbázison keresztül telepíthetőek, hanem más módon.

Végül van lehetőség arra is, hogy saját modulokat készítsünk, amelyeket a saját gépünkön található programok érnek csak el, vagy akár arra is lehetőség van, hogy csak egy adott programon belül legyenek elérhetőek.

Ez utóbbi lehetőséget nagyon egyszerű megvalósítani: a modul ugyanis nem más, mint egy speciális szerkezetű Python programállomány. Hogy miben speciális, arról hamarosan szó lesz.

2.1. Modulok alkalmazása

Ahhoz, hogy egy modult a programban használjunk, az **import** utasítással be kell azt importálni a programba. Az importálás tulajdonképpen úgy történik, hogy a modul tartalmát az értelmező végrehajtja az importálás helyén. Ez azt jelenti, hogy a modulban szereplő függvények az importálás helyétől tekinthetőek definiáltnak, ahogyan a modulban szereplő minden más objektum (pl. változók) is ettől kezdve léteznek. Ebből következően az importálásnak a program elején kell megtörténnie.

Bár elvi akadálya nincs, hogy ne így legyen, de alapvető szabály: **a modulok importálást rögtön a program bármely más utasítása elé kell elhelyezni**. Ez a gyakorlatban a program elején ál-

talában elhelyezett *shebang*[2] – esetleg az azt követő, az egész állomány tartalmára vonatkozó dokumentáló megjegyzés – után azonnal történik.

A 12. ábrán látható programrészlet egy példa arra, hogyan lehet a – később, a lecke vége felé bemutatásra kerülő – *math* modult beimportálni a programba. Az 1. sor a *shebang* néven is ismert sor, amely megadja, hol találja az operációs rendszer a Python értelmezőt, utána a jobb olvashatóság kedvéért kimarad egy sor, majd következik a 3. sorban a *math* modul beimportálása, amit illik legalább egy üres sorral megtoldani, hogy a program további részétől elkülönüljön. Az ábra az 5. sorban arra is példát mutat, hogyan lehet a modulban definiált **sqrt()** függvényt ezután használni.

```
1  #! /usr/bin/python3
2
3  import math
4
5  print(math.sqrt(4))
6
```

12. ábra

Mint az ábrán látható, az **import** utasítás legegyszerűbb formája, amikor egy `import` szó után megadjuk a beimportálni kívánt modul nevét (a példában ez a *math* modul). Ekkor az egész modul beimportálásra kerül olyan módon, hogy – a példában a 3. sorra érve – az értelmező a modult tartalmazó Python állományt végigolvassa és végrehajtja a benne található – modulok esetében jellemzően függvénydefiniáló – utasításokat. Az így definiált függvények és változók ettől a ponttól kezdve a program végéig úgy használhatók, mintha a program globális adatterében lettek volna definiálva. Azzal a kitétellel, hogy ahogyan a 12. ábra 5. sorában látható, a modulbeli függvény neve elé a modul nevét és egy pontot is ki kell tenni. Ugyanígy, ha a modulban definiált változót akarunk használni, az is így írandó. Ebből az írásmódból fogja tudni az értelmező, hogy az adott modul egy eleméről van szó.

Kicsit tudományosabban a fent leírtak így szólnának: az **import** utasítás hatására létrejön egy *névtér* (a példában *math* néven) és ebben kerül definiálásra a modul valamennyi publikus objektuma. A modul objektumainak elérésekor meg kell adni a modul névterét is a <modulnév>. előtag alkalmazásával, ahol a <modulnév> a modul importálásával létrejött névtér.

Van egy kényelmesebbnek tűnő megoldás is, amikor nem akarjuk a teljes modult beimportálni a programunkba, csupán a használni kívánt részét. Ez azonban a program saját globális objektumai és a modulból beimportált objektumok összekeveredését okozza, mivel a beimportált objektumok a program globális névterébe kerülnek, ezért nagyon körültekintően kell használni:

Az **import** utasítás hosszabb változata a `from` kulcsszóval kezdődik, amit a modul neve követ, majd jön az `import` kulcsszó után a modulból beimportálni kívánt objektum, vagy objektumok vesszővel elválasztott listája. Szélsőséges esetben az objektumok megnevezése helyett a `*` is használható.

A 12. és a 13. ábra végrehajtása ugyanazt eredményezi, de a 13. ábra esetén nem az egész *math* modul, hanem csak abból az **sqrt()** függvény kerül importálásra. Mint látható, ebben az utóbbi esetben nem kell alkalmazni a `math.` előtagot. Sőt, nem is szabad, mivel nem létezik a *math* névtér, az **sqrt()** függvény a program globális névterében található.

```
1  #/ /usr/bin/python3
2
3  from math import sqrt
4
5  print(sqrt(4))
6
```

13. ábra

Mitől veszélyesebb ez a megoldás, mint amikor az egész modult importáljuk? Azért, mert a program saját névterébe került a függvény, amit a program későbbi részében esetleg felüldefiniál egy ugyanilyen nevű függvény definíciója. Éppen ezért nem szabad – bár a nyelv lehetővé teszi – a `*` segítségével a modul minden objektumát a program névterébe beimportálni: nem lehet a program későbbi részében tudni, hogy egy modulból származó vagy egy a programban definiált függvény vagy változó szerepel-e a kódban (nem minden eleme van

2 A script nyelvekben az első sor adja meg, hogy pontosan melyik programnak kell az állomány tartalmát értelmezőként végrehajtania. Ezt az első sort nevezik *shebang*nak.

esetleg dokumentálva a modulnak). A helyzetet még inkább nehezíti, ha egy programban több modult is importálni kell. A különböző moduloknak lehetnek azonos nevű objektumai. Ha ezeket mind a `from` segítségével a program névterébe importáljuk, akkor azok egymással összeütközésbe fognak kerülni! Sokkal átláthatóbb megoldás, ha minden modul tartalma a saját névterébe kerül. Igaz, ilyenkor többet kell gépelni – főleg hosszú modulnevek esetében –, de a fejlesztőeszközök a névkiegészítés felkínálásával segítenek megspórolni az ilyen többletgépelések egy részét, cserébe viszont átláthatóbb programkódot kapunk.

Vannak esetek, amikor a modul importálása nem egyszerűen annak nevének megadásán keresztül lehetséges. Például a korábban már emlegetett Qt programkönyvtár annyira összetett, hogy a C++ könyvtár egyes elemeit elérhetővé tevő modulok egész hierarchiát alkotnak. Maga a Qt több változatban is elérhető Pythonban, ezeknek csupán az egyike – a dokumentum írásakor a legfrisebb változata – a *PySide6* modulcsomag. Csomagról van szó, azaz egy csomagon belül különböző modulok vannak, amelyeket szinte egyetlen programban sem kell együtt használni, hanem csak a csomag azon részét, amelyre éppen szükség van.

Az alábbi kép a PySide6 hivatalos dokumentációjából származik:

```
import PySide6.QtCore

# Prints PySide6 version
print(PySide6.__version__)

# Prints the Qt version used to compile PySide6
print(PySide6.QtCore.__version__)
```

Az első sorban látható a modul importálása, amely a *PySide6* csomag *QtCore* nevű modulját importálja be. Mint látható, a két név között egy pont helyezendő el ilyen esetben. A további sorok mutatják, hogy létrejött egy *PySide6* nevű névtér, azon belül pedig egy *QtCore* nevű névtér. A beimportált függvényeket és objektumokat a *PySide6.QtCore* névtérrel lehet elérni.

Ha külön nem akarjuk elérni a PySide6 névteret, akkor a fenti importálás helyett lehetett volna alkalmazni az alábbi utasítást is:

```
from PySide6 import QtCore
```

Ekkor csak egy *QtCore* névtér jön létre, amelyen keresztül ugyanúgy el lehet érni a modul valamennyi objektumát.

Valójában ez annak köszönhető, hogy a *PySide6* modulcsomag egy könyvtár, amelyben további könyvtárak, illetve az egyes modulokat tartalmazó állományok vannak. A könyvtárak és az azon belüli állományok elérésére szolgál a pont illetve a `from` megoldás használata. Amennyiben saját modulokat definiálunk egy program projektkönyvtárában, ott ugyanezt a megoldást lehet majd alkalmazni.

Végezetül jöjjön még egy példa a *PySide6* dokumentációjából arra az esetre, amikor több modult akarunk egyszerre beimportálni a csomagból, mindet önálló névtérbe:

```
import sys
import random
from PySide6 import QtCore, QtWidgets, QtGui
```

Itt az első két **import** utasítás egy-egy alapból elérhető Python modul (a *sys* és a *random*) importálását végzi. A harmadik utasítás a *PySide6* csomag három modulját, a *QtCore*, a *QtWidgets* és a *QtGui* modulokat importálja három különböző névtérbe. Így összesen öt névtér jön létre az öt importált modulra: *sys*, *random*, *QtCore*, *QtWidgets*, *QtGui*.

Az Qt programkönyvtárral ebben a dokumentumban nem foglalkozunk többet, lévén teljesen objektum-orientált megközelítést alkalmaz.

Amennyiben egy modul nevét túl hosszúnak találjuk, és szeretnénk elérni, hogy a beimportálásával létrejövő névtér neve rövidebb legyen, akkor az **import** utasítás kiegészíthető az **as** résszel, amely esetben a névtér az `as` szó utáni névvel jön létre. Például az előző oldali példában a *QtCore* név helyett *qt* névtérrel lehet importálni az alábbi utasítással a *QtCore* modult:

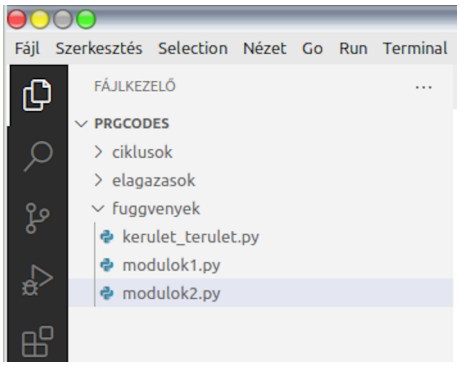
```
from PySide6 import QtCore as qt
```

2.2. Saját modulok készítése

Volt már szó arról, hogy összetettebb programokat nem ajánlatos egyetlen forrásállományban megírni, hanem a program különböző funkcióit megvalósító függvényeket külön-külön állományokba írják meg. Ezt a megközelítést a Python esetében a saját modulok írásával lehet alkalmazni.

Ekkor a program projekt könyvtárában egynél több állomány – kifejezetten összetett programok esetében több könyvtár, azokban számos állomány – jön létre a program fejlesztése során.

Ennek a dokumentumnak az elkészítéséhez tartozik egy könyvtár, amelyben levő állományok szolgálnak a dokumentumban szereplő kódok projektkönyvtáraként. Ezekből készített képernyőmentések jelennek meg a dokumentumban programkódot mutató ábraként. Az 14. ábra az ezen sorok írásakor ennek a könyvtárnak a tartalmát mutatja, ahogyan azt a VS Code megjeleníti az állománykezelő részben. Mint látható, jelenleg egy *ciklusok*, egy *elagazasok* és egy *fuggvenyek* nevű könyvtár van, utóbbi van kinyitott állapotban, azaz az ebben levő állományok listája jelenik meg. A VS Code egyébként ezt egy fa formájában mutatja meg, ahol az állományok felelnek meg a fa leveleinek. Amennyiben valamelyik állomány nevén duplán kattintunk, akkor azt az állományt megnyitja a program szerkesztésre. Nem látható a képen, de ha a területen van az egérmutató, akkor a jobb felső sarokban jelennek meg az új állományt, új könyvtárt létrehozni hivatott gombok is.



14. ábra: A dokumentum forráskódjait tartalmazó projekt könyvtára a VS Code-ban megjelenítve

Tehát egy program általában sok állományból áll, az egyes állományokban levő programrészlet pedig más állományokban definiált kódot is igénybe vehet. Ezt Python esetén szintén az **import** utasítással lehet megtenni, a projekt többi állományát modulként kezelve.

Vagyis ha például készítenék egy `haromszog.py` állományt a *fuggvenyek* könyvtárban, amelyben a `kerulet_terulet.py` állományban definiált, az előző leckében bemutatott függvényeket akarnám használni, akkor azt a `haromszog.py` állomány elején levő

```
import kerulet_terulet
```

utasítással tehetném meg. Amennyiben a `haromszog.py` állomány nem a *fuggvenyek* könyvtárban, hanem a projekt fő könyvtárában lenne, akkor azt is meg kellene adni az értelmezőnek, hogy az importáláshoz menjen be a *fuggvenyek* könyvtárba. Ez a következő két utasítás valamelyikével tehető meg:

```
import fuggvenyek.kerulet_terulet
from fuggvenyek import kerulet_terulet
```

Remélem észrevetted, hogy bár az állomány neve `.py` végződéssel rendelkezik, az import utasításban enélkül szerepel!

Amennyiben csak a területet akarjuk kiszámolni, akkor elég lenne csak a **terulet()** függvényt importálni? A válasz: igen. A

```
from fuggvenyek.kerulet_terulet import terulet
```

utasítás ezt megteszi úgy, hogy a függvény működni fog. Ebben az esetben a **kerulet()** függvény ugyan nem elérhető, de a **terulet()** függvény eléri. Visszautalva a függvényekről szóló lecke

végére, ebben az esetben a **kerulet()** a modul lokális adatterületén levő függvény, amelyet a **terulet()** függvény lát, ezért meghívhatja, de a programban a **terulet()** függvényen kívül nem látszik.

Hasonlóan, a *math* modult a `haromszog.py` állományba nem kell a **terulet()** függvény kedvéért importálni, mivel a **terulet()** függvény által elért lokális adatterületen az már elérhető. Azaz egy modul beimportálásakor nincs szükség a modulban importált további modulok importálására.

Hasonló módon lehet elrejteni bármilyen, a modulban definiált objektumot (függvényt, osztályt, változót) a modult importáló programtól, ha a nevét egy aláhúzásjellel (_) kezdjük. A modulban definiált, aláhúzásjellel kezdődő nevű objektumok csak a modulban levő függvények számára érhetőek el, a modult importáló kódban semmilyen módon nem lehet hozzájuk férni. Ezek az objektumok tehát a modul lokális objektumai.

2.3. Modul vagy főprogram?

Az előző szakaszban leírt módon létrehozott programrészek esetében fontos kérdés, hogy melyik állomány lesz a főprogram, amelyet elindítva kell a programot végrehajtani. A többi állomány vagy közvetlenül vagy közvetve ebbe a kódba lesz beimportálva modulként. Mivel egy összetettebb programot esetleg nem lehet egyben tesztelni, előfordulhat, hogy egy-egy később modulként használandó rész tesztelésére a programot önállóan kellene futtatni – ebben az esetben mintha az lenne a főprogram. Ekkor olyan kódot írnak a modul kódjába, amely a modulban definiált objektumokat használja valamilyen teszkódon keresztül.

Igen ám, de amikor a modult beimportáljuk, ez a kód is végrehajtódik! Ezt el kellene kerülni. Persze megtehető, hogy a modul tesztelése után ezt a kódot töröljük, de mi a helyzet, ha a modul fejlesztése közben már a program olyan részének a fejlesztése is zajlik párhuzamosan, amely ezt a modult használná? Ekkor nyilván még szükség van a tesztkódra, de már modulként is megtörténne az importálás, ahol viszont nem szabadna a kódnak végrehajtódnia.

Másrészt előfordulhat, hogy valaki olyan programot akar készíteni, amelyek egyes részei önállóan is végrehajthatóak, vagy egy nagyobb program részeként hajthatóak végre. Ekkor nem tesztkód

```
1  # Fibonacci számokat számító modul
2  def fib(n): # kiírja a számokat
3      a, b = 0, 1
4      while a < n:
5          print(a, end=' ')
6          a, b = b, a+b
7      print()
8
9  def fib2(n): # visszaadja a számokat egy listában
10     result = []
11     a, b = 0, 1
12     while a < n:
13         result.append(a)
14         a, b = b, a+b
15     return result
16
17 if __name__ == "__main__":
18     import sys
19     fib(int(sys.argv[1]))
20
```

15. Ábra: Fibonacci számokat számoló függvények modulja

kerül a kódba, hanem valóban végrehajtandó kód. Így ugyanaz az állomány önálló programként is futtatható és modulként más programba is ágyazhatóak a benne definiált függvények.

Hogyan lehet ilyen esetben elkerülni, hogy az önálló program esetén futtatandó kód a modulként importáláskor is végrehajtódjon?

A válaszhoz tudni kell, hogy a Python értelmező minden egyes modul feldolgozása során nyilvántartja, hogy éppen egy beimportált modult vagy a főprogramot dolgozza-e fel. Ezt a programkódban le is lehet kérdezni. Erre a *__name__* változó szolgál. A két aláhúzásjellel kezdődő és végződő változók az értelmező rendszerváltozói, amelyekkel az értelmező különböző állapotai kérdezhetőek le. Ez konkrétan megmondja, hogy az éppen végrehajtott kód melyik része a programnak. Amennyiben az értéke a „__main__” szöveg, akkor a főprogram.

Az 15. ábrán látható kód mutat példát a használatára. Ennek az elejével – a fibonacci számok számítási módjával és a listákkal – nem foglalkozunk. A 17–19. sor miatt szerepel a példa.

Itt található kód végzi el ugyanis a szükséges ellenőrzést: ha a „__main__” érték szerepel a *__name__* változóban, akkor a kód programként fut, tehát a 18–19. sorok végrehajtandóak. Modulként beimportálva a kódot, a 17. sor feltétele hamis lesz, így elkerülhető a nem kívánt kód végrehajtása.

3. További összetett adatszerkezetek

Korábban már megismerkedtünk néhány összetett adatszerkezettel. Ezek a szekvenciális adattípusok voltak: karakterlánc (*str*), listák (*list*) és a *tuple*, valamint az iterátorok, mint a **Range()** vagy az **Enumerate()**. Most újabb összetett adatszerkezeteket ismerhetsz meg, mint a halmazok vagy a szótárak.

3.1. Halmazok

A **set** adattípus a matematikából ismert halmazt képes megvalósítani. Ez különböző értékek *nem rendezett* gyűjteménye, amelyben minden érték legfeljebb egyszer fordulhat elő. Ebben a meghatározásban a *nem rendezett* azért fontos, mert ebből következik, hogy semmilyen sorrendet nem lehet feltételezni a halmaz elemeihez való hozzáférés során. Fontos továbbá, hogy *ugyanaz az érték még egyszer nem kerülhet bele a halmazba* – ellenben például a listával, amelyben ilyen korlátozás nincs. Tipikus használata lehet a *set* típusnak: tartalmazás vizsgálata, egy szekvenciából az ismétlődés eltávolítása, a matematikai halmazműveletek (unió, metszet, különbség) megvalósítása.

Mint a többi összetett adattípus, a *set* is támogatja az *x in set*, *len(set)* és *for x in set* műveleteket: ellenőrizni lehet, hogy egy adott érték eleme-e a halmaznak, lekérdezhető az elemek száma, végig lehet menni a halmaz elemein (bár előre nem lehet megjósolni ezek sorrendjét). Mivel a *set* elemeinek nincs definiálva a sorrendje, így nem lehet azokat indexelni, szeletelni vagy más sorrendet igénylő műveletet végezni.

A *set* mellett van még egy hasonlóan működő adattípus is, a **frozentset**. Míg a *set* típusú érték *módosítható* (*mutable*), azaz például az **add()** és a **remove()** metódusokkal bele lehet tenni, illetve ki lehet belőle venni elemet, a *frozenset nem módosítható* (*immutable*) típus. Mint a módosítható típusok mindegyike esetében, a *set* értéket nem lehet halmaz elemeként vagy a később szereplő szótár kulcsaként használni, míg a *frozenset* értéket lehet.

Nem üres *set* (nem *frozenset*) értékek létrehozhatók konstansmegadással, ha kapcsos zárójelek közé írva felsoroljuk az elemeit. Például `{'Jani', 'Béla'}`.

A típus neve mint konstruktorfüggvény használható mindkét típusú érték létrehozására:

```
class set([iterable])
class frozenset([iterable])
```

A konstruktorfüggvények visszadnak egy *set* illetve *frozenset* típusú halmazt, amelynek elemei lesznek a paraméterben szerepelő iterálható típusú kifejezés elemei. Értelemszerűen, ha nincs paraméter, akkor a létrejövő halmaz az üreshalmaz lesz.

A set típusú érték előállítható:

- Kapcsos zárójelek között, vesszővel elválasztva felsorolva az elemeket:
  `{"alma", "körte", "banán"}`
- halmaz-kifejezéssel: `{c for c in 'abracadabra' if c not in 'abc'}`
- a konstruktorfüggvényt használva:
  - `set()`[3]
  - `set('foobar')`
  - `set(['a', 'b', 'foo'])`

A *set* és a *frozenset* típusra a következő műveleteket lehet alkalmazni:

`len(s)`

Megadja az *s* halmazban levő elemek számát.

`x in s`

3 A `{}` kifejezés nem üres halmazt, hanem üres szótárat hoz létre! A szótárakról a következő szakaszban lesz szó.

Igaz, ha *x* eleme *s*-nek.
`x not in s`
Igaz, ha *x* nem eleme *s*-nek.
`set <= other`: Igaz, ha *set* részhalmaza az *other* halmaznak.
`set < other`: Igaz, ha *set* valódi részhalmaza az *other* halmaznak.
`set >= other`: igaz, ha `other <= set`.
`set > other`: igaz, ha `other < set`.
`set | other |` ...: A felsorolt halmazok uniója.
`set & other &` ...: A halmazok metszete.
`set - other -` ...: Azt a halmazt adja, amelyben *set* azon elemei szerepelnek, amelyek nem szerepelnek az *other* halmazban (illetve egyetlen további megadott halmazban sem).
`set ^ other`: A set és az other halmazok szimmetrikus különbsége (azok az elemek, amelyek a két halmaz közül pontosan az egyikben vannak benne).

A fenti halmazműveletekre léteznek metódusok is, amelyek megtalálhatóak a Függvénytárban. Ezek a metódusok paraméterként nem csak halmazt, hanem bármilyen iterálható – ezáltal halmazzá alakítható – típusú értéket elfogadnak, míg a fent felsorolt operátorok csak halmazzal tudnak dolgozni.

Mind a *set*, mind a *forzenset* típus támogatja a halmazműveleteket. Két halmaz azonosnak számít akkor és csak akkor, ha minden elem, amely az egyiknek eleme, eleme a másiknak is.

Egy *set* és egy *frozenset* összehasonlítása az elemeiken alapszik. Például a
`set('abc') == frozenset('abc')` igaz logikai értékű, ahogyan a
`set('abc') in set([frozenset('abc')])` is.

Mivel a halmaz értékeken semmiféle rendezettség nincs definiálva, a **list.sort()** függvény nincs definiálva olyan listákra, amelynek elemei halmazok.

Az alábbi műveletek módosítják a halmazt, így *frozenset* típus esetén nem használhatóak:
`set |= other | ...`
A *set* halmazhoz hozzáveszi a többi halmaz elemeit.
`set &= other & ....`
A set halmaznak csak a többi halmaz mindegyikében is szereplő elemeit tartja meg.
`set -= other | ...`
A *set* halmaznak csak azokat az elemeit tartja meg, amelyek nem szerepelnek egyetlen, a `-=` jobboldalán álló halmazban sem.
`set ^= other`
A *set* halmaznak azok az elemek lesznek elemei, amelyek vagy a *set*, vagy az *other* halmaznak elemei voltak, de nem mindkettőnek.

Egy szekvenciát a **set()** paraméterébe téve visszakapható az elemeinek halmaza az esetleges ismétlődések kiszűrésével. Például ha *l* egy lista, akkor a `list(set(l))` olyan listát adj, amelynek elemei *l* elemei, de ismétlődés nélkül. Azonban mivel a halmaz elemei nem rendezettek, így az új lista elemeinek sorrendje nem feltétlenül egyezik meg az eredeti lista elemsorrendjével. Amennyiben rendezetten akarjuk az elemeket megkapni, akkor a **list()** függvény helyett a **sorted()** függvényt célszerű használni.

3.2. Szótárak

A szótár tekinthető bizonyos értelemben a *set* típus kibővítésének. Más nyelvekben az *asszociatív tömb* nével létezik hasonló. Az adattárolók között létezik ún. *asszociatív memória* (például a CPU gyorsítótára ilyen), amely hasonlóan viselkedik, mint a Python nyelv szótár típusa. A szótárat ugyanakkor tekinthetjük olyan listának is, amelynek elemeit nem egész számokkal kell indexelni,

hanem bármilyen – nem változtatható típusú – értékkel lehet az indexelést végezni. Ugyanakkor míg a lista elemeinek sorrendjét az indexei meghatározzák, a szótár elemein nincs rendezettség definiálva. Ennél fogva a **.sort()** metódus sem alkalmazható szótárra. Viszont a **sorted()** függvényt egy szótárra alkalmazva, megkapható a szótár „indexeinek”, azaz a kulcsainak a rendezett listája, a **list()** függvénnyel ugyanez a lista a kulcsokat az elemek létrehozási sorrendjében adja.

A **dict** (*dictionary*, azaz szótár) adattípus elemei kulcs-érték párok. A kulcsokra ugyanazok a megkötések vannak, mint a halmaz elemeire, az értékek bármilyen típusúak lehetnek. Tehát ha nem lenne értéke az elemeknek, hanem csak a kulcsból állnának, akkor egy halmazt kapnánk. Ennek megfelelően, egy szótár konstanst is meg lehet adni kapcsos zárójelek között vesszővel elválasztva felsorolva az elemeket, azonban egy elem a *kulcs: érték* alakú megadást igényli:

```
>>> a = {1: "alma", "körte": 2}
>>> a
{1: 'alma', 'körte': 2}
```

A fenti példában az a szótárnak két eleme van:

- Az 1 kulcsú elem, amelynek értéke az „alma” szöveg
- A „körte” kulcsú elem, amelynek értéke a 2 egész szám.

A kulcsok a halmazok elemeihez hasonlóan csak nem módosítható típusból kerülhetnek ki. Az értékekre nincs típusra vonatkozó megkötés. Vigyázni kell, hogy azok az érték alapján összehasonlítandó kulcsok (például `1`, `1.1` és `True`), amelyek összehasonlítva ugyanazt az értéket adják, azonos elemet fognak azonosítani!

A típust a fenti példában látható mellett úgy is meg lehet adni, ha a **dict()** konstruktorfüggvényt használjuk. Amennyiben a függvény nem kap paramétert, úgy egy üres szótár jön létre. A kapcsos zárójeles módon azért nem lehet üres halmazt megadni, mivel a `{}` egy üres szótár típusú kifejezés.

Amennyiben a függvényt használva nem üres szótárat akarunk létrehozni, úgy például az alábbi módon tehetjük ezt meg:

- Tegyünk a paraméterbe két elemű *tuple*-k (vagy más kételemű szekvencia) listáját. Ekkor az egyes tuple-k első eleme lesz az elem kulcsa, a második az értéke: `dict([('foo', 100), ('bar', 200)])`
- Adjuk meg paraméternévként a kulcsot, utána egyenlőségjelet követően az elem értékét: `dict(foo=100, bar= 200)`

Ha a fenti esetekben egy kulcs egynél többször szerepel, akkor az utolsó előfordulásakori érték lesz az adott elem értéke.

A szótár adatszerkezeten az alábbi műveletek végezhetőek el:

list(d): Visszaadja a *d* szótárban található kulcsokat tartalmazó listát.

len(d): Visszaadja a *d* szótár elemeinek számát.

d[kulcs]: Visszaadja a *d* szótár *kulcs* kulcsú elemének értékét, ha az létezik. Amennyiben ilyen kulcsú elem nem létezik a szótárban, akkor a KeyError hiba keletkezik.

d[kulcs] = érték: A *d* szótár *kulcs* kulcsú eleme *érték* értéket kap. Amennyiben eddig nem létezett ilyen kulcsú elem, akkor mostantól a szótár új elemként kapja ezt az elemet.

del d[key]: Törli a *d* szótár *key* kulcsú elemét. Ha ilyen kulcsú elem nem létezik, akkor a KeyError hiba keletkezik.

key in d: Akkor igaz, ha a *d* szótárban van olyan elem, amelynek *key* a kulcsa.

key not in d: Az előző logikai művelet tagadása.

iter(d): Egy iterátort ad, amellyel *d* kulcsain végig lehet menni.

d.clear(): Törli a szótár valamennyi elemét.

d.copy(): A szótár egy teljes másolatát állítja elő.

d.keys(): A szótárban szereplő kulcsokat kapjuk egy lentebb bemutatott *nézet* formájában.
d.values(): A szótárban előforduló értékeket kapjuk *nézet* formájában.
d | other: Egy új szótárat hoz létre, amely a két szótár metszete, abban az értelemben, mintha a kulcsokat tekintenénk a halmaz elemeinek.
d |= other: A fenti művelet úgy módosítva, hogy az eredmény a d szótárban keletkezik.
Szótárak akkor és csak akkor egyenlőek, ha minden (kulcs, érték) pár megegyezik bennük.

3.2.1. Szótárak nézetei

A **dict.keys()**, **dict.values()** és a **dict.items()** függvények által visszaadott értékeket *nézet objektumnak* nevezi a Python. (A három függvény rendre a szótár kulcsait, értékeit, illetve a (kulcs, érték) pár formájú elemeit adja eredményül.) Ezek olyan objektumok, amelyek együtt élnek a szótárral, azaz ha a szótár tartalma változik, akkor a hozzájuk kapcsolódó nézetek is automatikusan követik a változást.

A szótár nézeteken iterálható értékekként végig lehet menni, illetve támogatják a tartalmazásvizsgálatokat:

len(dictview): Visszaadja az elemek számát.
iter(dictview): Egy iterátort ad, amellyel végig lehet menni az elemeken. Ehhez tudni kell, hogy a hivatalos dokumentáció szerint a szótár elemei a felvételük sorrendjében vannak tárolva, később felvett elemek mindig a végére kerülnek, hiába volt előtte elem törölve valahonnan. Ennek megfelelő sorrendben adja az elemeket az **iter()** függvény.
x in dictview: Igaz értéket ad, ha *x* benne van a *dictview* nézetben. Amennyiben *dictview* az elemek nézete (a **dict.items()** függvénnyel létrehozva), akkor *x* egy (kulcs, érték) tuple kell legyen!
reversed(dictview): Az **iter()** függvénnyel ellentétes sorrendben, azaz a végéről indulva adja a nézet elemeit.
dictview.mapping: „Return a types.MappingProxyType that wraps the original dictionary to wich refers.” (A Python 3.10-es verziójában jelent meg.)

A kulcsok nézete (**dict.keys()** függvény) gyakorlatilag tekinthető egy halmaznak, hiszen a szótár kulcsai halmazt alkotnak. Azonban a normál *set* halmazoktól eltérően ez a nézet követi az eredeti szótár tartalmának változását, azaz ha új elem kerül a halmazba, akkor annak a kulcsa is automatikusan megjelenik benne, ahogyan a törölt elem kulcsa is automatikusan eltűnik belőle. Az ilyen nézetekre alkalmazhatóak a halmazokra elérhető műveletek.

3.2.2. Ciklus a szótár vagy szekvencia elemeire[4]

A fenti nézeteket használva több módon is végig lehet menni a szótár elemein. Például az **.items()** nézettel, amely egyszerre adja a szótár kulcs/érték párjait, a következő képpen lehet az elemeken végigmenni:

```
knights = {'gallahad': 'the pure', 'robin': 'the brave'}
for k, v in knights.items():
    print(k, v)
```

A fenti kódrészlet az alábbi kimenetet eredményezi:

```
gallahad the pure
robin the brave
```

4 Ezt a szakaszt majd szét kell bontani később különböző részeken elhelyezendő elemekre az egyes iterátorok bemutatására...

A fenti kód hasonló ahhoz, mikor egy szekvencia elemein az enumerate() függvénnyel megyünk végig, amely az elem indexéből és értékéből álló párokat adja iterátorként:

```
for i, v in enumerate(['tic', 'tac', 'toe']):
    print(i, v)
```

Több szekvencia elemein egyszerre végig lehet haladni a **zip()** függvényt használva:

```
questions = ["name", "quest", "favorite color"]
answers = ['lancelot', 'the holy grail', 'blue']
for q, a in zip(questions, answers):
    print('What is your {0}? It is {1}.'.format(q, a))
```

Ebben az esetben is, mint a listáknál már szerepelt rá példa, vigyázni kell a szótár esetleges módosításánál! Mivel a szótár esetében az elemeknek nincs hivatalosan definiált sorrendjük, még nagyobb probléma lehet belőle, ha a cikluson belül módosul a szótár, amin a ciklus végighalad.

Az alábbi példa két megoldást mutat arra az esetre, ha egy szótár bizonyos értékű elemeit kell kigyűjteni:

```
29 users = {'Hans': 'active', 'Éléonore': 'inactive'}
30 for user, status in users.copy().items():
31     if status == 'inactive':
32         del users[user]
33
34 active_users = {}
35 for user, status in users.items():
36     if status == 'active':
37         active_users[user] = status
38
```

Az első **for** ciklus a szótár másolatának elemeit tartalmazó nézeten fut végig, ami nem fog megváltozni az eredeti szótár elemeinek törlésétől. Ezzel szemben a második ciklus ugyan az eredeti szótár elemeinek nézetén fut végig, de a ciklus nem módosítja azt a szótárat, hanem egy másik, kezdetben üres szótárba veszi fel az új elemeket. A ciklus futását egyik esetben sem fogja megzavarni a szótár módosítása.

4. Állománykezelés

Egy idő után elég macerássá válik, ha egy program bemenő adatait mindig a felhasználótól kell bekérni. Ennél sokkal praktikusabb megoldás a bemenő adatokat egy állományban előkészíteni és azután elindítani a programot, amely azután az állomány tartalmát beolvassa, majd esetleg az eredményt szintén egy állományban szolgáltatja. A következőkben az állománykezelés legegyszerűbb módjával ismerkedünk meg.

4.1. Állományok fajtái

Egy számítógép számára gyakorlatilag minden állománynak számít. Még a billentyűzeten begépelt tartalom is egy speciális állomány, ahogyan a program akkor is állományba ír, amikor a képernyőn jelenít meg valamit. UNIX rendszerek esetén maga az állomány is egy olyan állományban található, amely állományokat tud tárolni: ezt nevezzük *könyvtár*nak (manapság a grafikus megjelenése miatt inkább *mappá*nak). De még azt a háttértárat is, amelyen könyvtárakba rendezve az állományok tárolódnak, speciális állományként kezeli a rendszer. A továbbiákban csak azokról az állományokról lesz szó, amelyek a felhasználó számára is állománynak tekintendőek.

Ezeknek az állományoknak a bennük tárolt tartalom szempontjából is sok fajtája van. Az egyik ilyen típus maga a programállomány, amely a számítógép által végrehajtandó kódot tartalmazza. Ezeket az állományokat el lehet indítani, hogy a számítógép a benne található utasításokat végre tudja hajtani. Az állományok másik fajtája adatokat tartalmaz, a továbbiakban csak ezekkel foglalkozunk.

Az adatállományok lehetnek binárisak és szöveges állományok (a programállományok értelemszerűen mindig binárisak, mivel a végrehajtandó utsítások bináris kódját tartalmazzák. A bináris és a szöveges állományokat legegyszerűbben úgy lehet megkülönböztetni, hogy egy sima szövegszerkesztővel – például jegyzettömbbel – megnyitva el lehet e olvasni a tartalmukat (szöveges állományok) vagy speciális program képes csak értelmezni a tartalmukat (bináris állományok).

Fontos különséget tenni a kétféle állománytípus között, mert nem ugyanazokat a műveleteket lehet a tartalmukkal elvégezni! A bináris állományok is sokféle típusba tartozhatnak, ezek ismerete szükséges a bennük levő adatok értelmezéséhez.

Ellenben a szöveges állományok karaktereket tartalmaznak, amelyek

1. képernyőn megjeleníthetőek és billentyűzetről begépelhetőek
2. vezérlőkarakterek, amelyek speciális jelentéssel bírnak.

Utóbbi kategóriába tartoznak például azok a karkterek, amelyek a megjelenítőt új sor kezdésére utasítják vagy valamekkora hely kihagyására. Utóbbira példa a tabulátor karakter, amely meghatározott karakterpozícióra igazíthatja a következő kiírandó karaktert.

Előbbi, az új sor karakter operációs rendszerenként eltérő karakterkódokat kap. Például a UNIX rendszerekben a 13-as kódú byte-kód jelzi az új sort, míg a DOS és az abból eredő Windows operációs rendszer a 13, 10 kódú byte-ok sorozatát tekinti új sornak, ugyanakkor az Apple Macintosh operációs rendszere a 10 kódú byte-ot. Emiatt az egyik gépről a másikra átvitt szöveges állománynál a kódok konvertálására is oda kell figyelni. Ezt több programozási nyelv, így a Python is megoldja azzal, hogy az újsor jelet egy speciális karaktersorozattal kell jelölni: `\n`. Hasonlóan a többi vezérlőkarakternek is van ilyen jele, például a fentebb említett tabulátor karaktert a `\t` jelöli.

A továbbiakban elsősorban a szöveges állományok használatára fogunk koncentrálni, de érdemes tudni, hogy ez csak egy lehetőség az adatok állományban tárolására. A lecke végén röviden néhány további elterjedt megoldásról is találsz majd említést.

4.2. Állomány olvasása

Az **open()** függvény szolgál egy állomány megnyitására. Ez a függvény egy állomány-objektumot ad vissza, amelyen keresztül lehet az állományhoz hozzáférni.

Amennyiben egy szöveges állományt akarunk beolvasni, akkor nem kell mást tenni, mint az **open()** függvénnyel megnyitni, megadva paraméterként az állomány nevét. Ha az állomány a program munkakönyvtárában van, akkor elég az állomány neve, különben az útvonalát is meg kell adni.

```
1  f = open("102.files/peldak/pelda.txt")
2
```

*16. ábra: Állomány megnyitása az **open()** függvénnyel*

Az 16. ábrán látható utasítás a program munkakönyvtárához képest a `102.files` könyvtár `peldak` könyvtárában levő `pelda.txt` állmányt nyitja meg olvasásra.

A példában az elérési útvonal azért kell, mert a VS Code úgy lett beállítva, hogy a megnyitott könyvtárat tekintse munkakönyvtárnak (a 17. ábra mutatja a VS Code konfigurációs állományát, amelyben a 14. sor helyezendő el abból a célból, hogy az aktuális könyvtár legyen a munkakönyvtár) és ezen belül a `102.files` és abban a `peldak` könyvtár tartalmazza a megnyitandó állományt.

Az **open()** függvénynek van még két paramétere: a megnyitás módja és az állomány kódolása számára.

A megnyitás módja most az olvasás, ami az alapértelmezés, azért az ehhez tartozó ’r’ elhagyható. Ha megadnánk, akkor karakterláncként, idézőjelek közé írandó.

Ha írni szeretnénk az állományba, akkor a ’w’ adható meg, amely esetben az állomány tartalma felülíródik, vagy az ’a’ ha a meglévő tartalomhoz hozzá kell fűzni új tartalmat. Ha módként a ’r+’ szöveg szerepel, akkor az állományba írni is lehet és abból olvasni is.

Alapesetben az állomány szöveges módba kerül megnyitásra. Ekkor a harmadik paraméter határozza meg a nem ASCII karakterek kódolására használt kódtáblát. Ez alapértelmezésben az „utf-8”, így ha ezt használjuk, akkor nem kell megadni. Ellenkező esetben a harmadik paraméterrel határozható meg a kódtábla (pl. `encoding="iso-8859-2"` a latin 2 kódoláshoz). Amennyiben az állományt binárisan kell megnyitni, akkor a harmadik paraméter nem használható, viszont a második paraméter szövegébe a ’b’ betűt is bele kell írni.

A megnyitott állományt előbb-utóbb be is kell zárni. Ennek főleg akkor van jelentősége, ha az állomány tartalmát a program módosítja (vagyis ír bele): a tartalom fizikai kiírása addig nem feltétlenül történik meg, amíg az állomány nem kerül bezárásra. Ezért az utolsó állományművelet mindig a **.close()** legyen, amely bezárja az állományt, amelyre alkalmazzuk.

readfiles.py | launch.json | pelda.txt

.vscode > launch.json > ...

```
1  {
2      // IntelliSense használata a lehetséges attribútumok listázásához
3      // Húzza fölé az egeret a létező attribútumok leírásának megtekintéséhez!
4      // További információért látogassa meg a következőt: https://go.microsoft.com/fwlink/?linkid=830387
5      "version": "0.2.0",
6      "configurations": [
7          {
8              "name": "Python: Current File",
9              "type": "python",
10             "request": "launch",
11             "program": "${file}",
12             "console": "integratedTerminal",
13             "justMyCode": true,
14             "cwd": "${fileDirname}"
15         }
16     ]
17 }
```

17. ábra: A launch.json állomány a 14. sorba beírt paranccsal a munkakönyvtár beállításával

Az 18. ábra egy programrészt mutat, amelynek 1. sorában az állomány megnyitása, a 6. sorában annak az állománynak a bezárása látható.

```
1  f = open("102.files/peldak/pelda.txt")
2
3  szoveg = f.read()
4  print(szoveg)
5
6  f.close()
7
```

18. ábra: Állomány megnyitása, olvasása, majd bezárása

A kettő között az *f* változón keresztül lehet hozzáférni az állomány tartalmához.

Az állomány bezárása automatizálható, ha a **with** utasítást használjuk. Ez az utasítás a hozzá tartozó blokk befejeztével az állományt automatikusan bezárja, ahogyan az a 19. ábrán is látható.

```
8  with open("102.files/peldak/pelda.txt") as f:
9      szoveg = f.read()
10
11 print(szoveg)
12
```

19. ábra: A with utasítás használata

Mint az ábrák mutatják, az **open()** függvény által létrehozott állomány-objektumot el kell tárolni egy változóban, mert így lehet rajta különböző műveleteket elvégezni.

Olvasásra megnyitott állományoknál a leggyakoribb művelet az állomány tartalmának beolvasása a **.read()** metódussal. Ez az állomány teljes tartalmát egyetlen karakterláncként adja vissza, amely a fenti példákban a *szoveg* változóba kerül. Ezután valójában az állomány már be is zárható, hiszen a tartalma már elérhető a string típusú változóban.

A **.read()** metódusnak paramétert is lehet adni. A paraméter a beolvasandó karakterek (bináris módban byte-ok) számát határozza meg. Paraméter hiányában, vagy egynél kisebb egész szám megadása esetén az egész hátralevő tartalom beolvasásra kerül – még ha ez nem férne el a memóriában, akkor is. Amennyiben csak egy részét olvassa be a **.read()** az állománynak, akkor az állomány-objektum megjegyzi azt a pozíciót, ahol az olvasás befejeződött és a következő olvasás innen indul. Amennyiben ez a pozíció már az állomány végén van, akkor a beolvasás eredménye az üres szöveg (`''`) lesz.

További műveletek is elérhetőek az állomány tartalmának beolvasására, mint például a **.readlines()**, amely az állomány sorait (a sorvége jelek közötti részeket) egy-egy karakterláncba olvassa be és ezeket egy listába teszi, így egy olyan listát kapunk, amelynek elemei *str* típusúak. Helyette alkalmazhato a `list(f)` kifejezés is, amely az *f* állomány-objektum sorait tartalmazó listát eredményez.

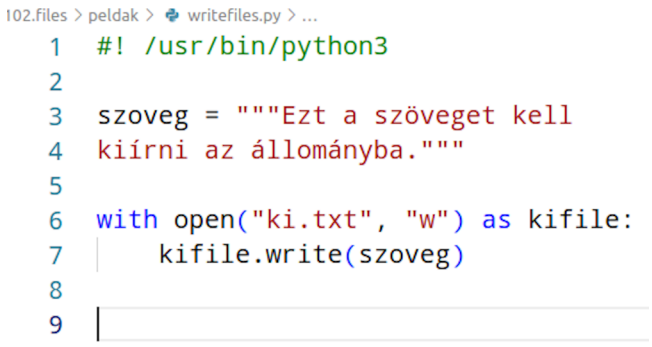
Kicsit talán kevésbé használható, de nagy méretű állományoknál biztonságosabb megoldás a **.readline()** (s nélkül!) használata, amely az állomány soron következő sorát olvassa be egy szöveges adatként. Azért kevésbé használható, mert ez a metódus a sorvége jelet is visszaadja a szövegben. Állomány végére érve ez is egy üres szöveget ad eredményül. Valójában onnan lehet tudni, hogy az állomány végére értünk és nem egy üres sort olvasott be, hogy az üres sor végén ott van a sorvége jel, tehát ebben az esetben a beolvasás eredménye a `"\n"` szöveg lesz.

Egy állomány sorait **for** ciklussal is be lehet olvasni, ahogyan az a 20. ábrán látható. Ekkor az olvasó metódusokra nincs is szükség. Az ábrán látható kód érdekessége a 15. sor, amelyben a **print()** automatikus sortörése el van nyomva, ami mutatja, hogy ennél a beolvasási módnál a sorvége jel is belekerül a beolvasott tartalomba a **.readline()** metódus működésének megfelelően. Annak ellenőrzését, hogy az állomány végére ért-e az olvasás, a **for** ciklus automatikusan elvégzi.

```
13 f = open("102.files/peldak/pelda.txt")
14 for line in f:
15     print(line, end='')
16
17 f.close()
```

20. ábra: Beolvasás for ciklussal

4.3. Írás állományba



21. ábra: Szöveg állományba írása

Amennyiben írni akarunk az állományba, akkor már az **open()** függvény második paraméterére is szükség van. Amennyiben ide a „w” szöveg kerül, akkor létező állomány esetén annak aktuális tartalma elveszik, nem létező állomány esetén az állomány létrejön. Ezt követően amíg az állomány nyitva van, akkor az abba történő írás hozza létre a tartalmát (*write*). Amennyiben a „a” szöveget használjuk, akkor léteznie kell az állománynak, amelynek a végére történik az új tartalom írása (*append*).

A 21. ábra mutat egy példát az állományba írásra – az állomány bezárását a **with** utasításra bízva.

Magát a kiírást leggyakrabban a **.write()** metódussal szokás végezni. Ennek használatánál figyelni kell arra, hogy a **print()** függvénnyel ellentétben, nem tesz a kiírás végére automatikusan sortörést, így arról a programban gondoskodni kell. A példában tehát a szöveg után nincs új sor, csak a *szoveg* változó szövegének közepén! A **.write()** metódus visszatérési értéke a kiírt karakterek száma. Ezt lehet a kiírás sikerességének ellenőrzésére is használni.

A **.write()** paramétere csak szöveg lehet (vagy *bytes* bináris mód esetén). Minden más típust át kell konvertálni, mielőtt a metódus paraméterébe tesszük!

4.4. Írás és olvasás egyszerre

Vannak esetek, amikor nem egyszerűen beolvasni kell egy állomány tartalmát vagy valamit egy állományba kiírni, hanem a kettőt felváltva kell végezni ugyanazon az állományon, folyamatosan változtatva annak tartalmát. Ekkor kell az **open()** függvény második paraméterébe a „+” jelet is beleírni. Ez azt jelenti, hogy a megnyitandó állományban „szabadon bolyongva” fogunk onnan beolvasni, oda beírni tartalmat. Ekkor szükség van az állományon belüli pozíció nyilvántartására illetve átpozícionálására.

A **.tell()** metódus segítségével lekérdezhető byte-ban (karakterben) mérve az állomány elejétől mérve az a pozíció, ahova a következő írás vagy ahonnan a következő olvasás fog történni. Ez a metódus akkor is elérhető, ha csak olvasásra vagy csak írásra van nyitva az állomány. Ám a pozíció megváltoztatásának nincs értelme, ha csak írásra van megnyitva.

A pozíció megváltoztatására a **.seek(offset, honnan)** metódus alkalmas. Az *offset* paraméter az az érték, amellyel el kell mozdítani valamerre a pozíciót, relatív értékként, míg a *honnan* paraméter adja meg a referenciapontot:

- 0 jelenti az állomány elejét (alapértelmezés);
- 1 jelenti az aktuális pozíciót;
- 2 az állomány végét (ekkor negatív értéket kell használni az *offset*-nél).

Szöveges állományok esetében a fentiek közül csak a 0 (állomány elejéről indulás) megengedett, valamint a `.seek(0, 2)` paraméterezés az állomány végére álláshoz. Mivel a szöveges állományoknál a pozíció nem feltétlenül byte-ban van meghatározva, csak olyan értékek adhatóak meg a nullán kívül az offset értékének, amelyeket egy **.tell()** hívás eredményezett.[5]

5 Az ilyen állományhasználati módszert általában csak bináris állományokon alkalmazzák, így ez nem jelent komoly megszorítást.

4.5. A json modul

Összetettebb adatstruktúrák állományba írására és onnan visszaolvasására szokás használni a *json* modult.

A JSON formátumot a JavaScript fejlesztői találták ki. A mozaikszó jelentése: JavaSrcipt Object Notation. A szintén szöveges formátumú HTML kódba ágyazott adattartalom szabványos megadására szolgált eredetileg, hogy azt a JavaScript nyelvű programkód fel tudja használni. Sok weboldal forráskódja tartalmaz ilyen formátumú adatokat beágyazva – például a Youtube oldalainak forráskódja is rengeteg metaadatot tartalmaz így a lejátszandó videóról.

A *json* modul valójában ezt a JSON formátumot teszi elérhetővé a Python programok számára.

Ahhoz, hogy egy JSON formátumú adatot tartalmazó szöveges állományt készítsünk, nem kell mást tenni, mint a tárolandó, összetett adatot tartalmazó változót a *json* modul megfelelő függvényével az adatnak megfelelő JSON formátumú szöveggé konvertálni, majd ezt egyszerűen a **.write()** metódussal ki lehet írni az állományba. Beolvasás esetén a **.read()** metódussal kapott szövegből a *json* egy függvénye tudja a megfelelő összetett adatot tartalmazó változót előállítani.

A két műveletből az elsőt az angol *serializing*, a másodikat *deserializing* kifejezéssel illetjük.

A **dumps()** függvény a paraméterében szereplő változóban levő adatstruktúrából képez egy JSON formátumú szöveget. Ha a függvény nevéből elhagyjuk az ’s’-t akkor a **dump()** függvény képes egyből egy írásra megnyitott szöveges állományba írni a JSON szöveget. Ekkor az első paraméter a JSON formátumra alakítandó adat, a második a célállomány.

Beolvasáshoz a **load()** függvényt lehet használni, amely a paraméterében szereplő, olvasásra megnyitott szöveges állomány tartalmát JSON szövegként értelmezi és a neki megfelelő adatstruktúrát adja eredményül. Ennek ’s’ végű változata, a **loads()** függvény paraméterként egy JSON formátumú szöveget vár.

```
1  #! /usr/bin/python3
2
3  import json
4
5  x = [1, 'simple', 'list']
6  s = json.dumps(x)
7  # s = '[1, "simple", "list"]'
8
9  f = open("file.json", "w")
10 json.dump(x, f)
11 f.close()
12
13 f = open("file.json")
14 x = json.load(f)
15 f.close()
16
```

22. ábra: A JSON formátum használata

A fenti ábra mutat például a JSON formátum gyakorlati használatára. Az *x* változó egy listát tartalmaz, ennek JSON formátumra alakítását végzi a 6. sor utasítása, a létrejövő szöveget mutatja a 7. sorban levő megjegyzés. A 9–11. sorok a **dump()** függvény közvetlen használatát mutaják az *x* lista állományba írására JSON formátumban. Ennek fordítottját, a JSON formátumú szöveg állományból betöltését a **load()** függvénnyel mutatja a 13–15. sor.

5. Elemi algoritmusok

Vannak olyan feladatok, amelyben bizonyos matematikai modellre épülő algoritmusok gyakran fordulnak elő. Ilyen feladatok valamilyen formában már a korábbiakban is előkerültek. Ebben a leckében ezek az **elemi algoritmusok** szerepelnek részletesebben. Természetesen az itt szereplő elemi algoritmusokon kívül is vannak még algoritmusok, amelyeket nyugodtan ide lehetne sorolni, de ezek talán a leggyakrabban előkerülők.

> Más tankönyvekben nem feltétlenül szerepel az elemi algoritmus elnevezés, hanem sokszor helyette a „*programozási tételek*” megnevezés alatt szereplnek ezek. Ez az egyébként hibás elnevezés valószínűleg onnan terjedt el, hogy azon tanárok, akik először elkezdtek Magyarországon általános és középiskolai szinten programozást (meg egyáltalán számítástechnikát, informatikát) tanítani, zömében más tantárgy tanításából tértek át és nagyon hozzá voltak szokva ahhoz, hogy tételekben fogalmazzák meg a megtanulandó ismereteket.
>
> A „*programozási tétel*” megnevezés amellett, hogy ezeknek az algoritmusoknak a hivatalos neve az **elemi algoritmus**, azért is hibás, mert az egyes tudományterületeken alkalmazott tételekkel ellentétben itt nem szó szerint bebiflázandó tudásról van szó, hanem a programozói gyakorlat során kialakult – egyébként matematikai úton bizonyíthatóan helyesen működő – eljárásokról, amelyek egy-egy feladat logikus megoldását adják.

A leckében bemutatásra kerülő elemi algoritmusok elsősorban listákon működnek, de több más olyan adattípusra is alkalmazhatóak, amelyek értékek valamilyen sorozatát tartalmazzák. Egyes algoritmusoknál még az sem szükséges, hogy az értékek valamilyen módon sorba legyenek állítva, azaz akár halmazon is alkalmazhatóak.

A következő leckében további algoritmusok szerepelnek, amelyek szintén elsősorban listákon, tömbökön, tehát értékek sorozatán végzendő műveletek során használatosak.

5.1. Elemek száma

Bár a Python nyelv használata során már találkoztunk a **len()** függvénnyel, amely megadja egy összetett adattípus esetén annak aktuális elemszámát, de felmerül a kérdés, hogyan számolhatjuk meg az elemeket abban az esetben, ha valamiért a **len()** függvény nem áll rendelkezésre. Másképpen fogalmazva: hogyan működik a **len()** függvény?

5.1. Adott egy lista, amelynek nem ismert az elemszáma és egyszerre csak egy elemhez lehet hozzáférni. Hogyan lehet a legegyszerűbben kideríteni, hogy eleme van a listának?
5.2. Hogyan lehet kideríteni, hogy egy adott feltételnek hány elem felel meg a lista elemei közül?
5.3. Mi a teendő akkor, ha nem egy elemre vonatkozó feltételről kell megmondani, hogy a listában hányszor fordul elő, hanem olyan feltétel előfordulási számát kell kideríteni, amely feltétel több egymást követő elemre vonatkozik (pl. hányszor követ egy elemet nála nagyobb elem, vagy hányszor szerepel két azonos érték egymás után)?

A fenti feladatok megoldására hasonló algoritmus vonatkozik. A legelső esetében nyilván végig kell menni a listán (tömbön) és minden elem esetében növelni egyel egy számlálót.

A második esetben csak akkor kell növelni a számlálót, ha az éppen vizsgált elemre teljesült a keresett feltétel.

A harmadik eset már egy kicsit bonyolultabb, hiszen ekkor nem elég egy elemet vizsgálni, hanem több egymás melletti elem viszonyát kell nézni.

Még bonyolultabb lehet az az eset, ha nem egy-egy, vagy néhány egymás utáni elem viszonyára vonatkozik a feltétel, hanem például azt akarjuk megtudni, hogy bizonyos tulajdonságú elemek sorozata hányszor fordul elő a listában.

Amennyiben azt kell megszámolni, hogy egy adott feltételnek hány elem felel meg, akkor a következő algoritmust alkalmazhatjuk:

számláló ← 0
Ciklus minden I értékre az L listából:
    Ha I megfelel a feltételnek:
        számláló növelése
Ciklus vége

Például ha a *szamok* listában a pozitív értékeket kell megszámolni, akkor arra az alábbi Python kód alkalmas:

```
5   szamlalo = 0
6   for szam in szamok:
7       if szam > 0:
8           szamlalo += 1   # számláló növelése
9
10  print(szamlalo)
```

Az utolsó sorban a print() függvény csak az eredmény kiírására szolgál, az algoritmus valódi alkalmazása során nem feltétlenül történik meg a használata. (Helyette a kapott érték másmilyen felhasználása is lehetséges.)

Amennyiben a feltétel több elem alapján állapítható meg, akkor már nem elég az értékeken végigmenni, hiszen az éppen vizsgált elem előtti vagy utáni értékére is szükség lesz a vizsgálathoz. Ilyen esetre példa a delfines feladatból a felszín áttörésének vizsgálata: ha két egymást követő érték ellentétes előjelű, akkor kell a számlálót növelni.

Ilyen esetben egyrészt az értékek helyett az elemek indexén kell végigmenni; másrészt vigyázni kell, hogy a vizsgálatot csak tényleg létező elemeken végezzük el.

```
12
13  szamlalo = 0
14  for index in range(len(szamok)):
15      if index == 0:
16          continue
17      if szamok[index] * szamok[index-1] > 0:
18          szamlalo += 1
19
20  print(szamlalo)
21
```

A fenti kód azt az esetet mutatja, ha a lista indexein megyünk végig. Ekkor a **range()** konstruktorfüggvény segítségével állítjuk elő az indexek sorozatát. A 15–16. sor gondoskodik róla, hogy a lista első elemre, amely előtt nincs elem, nem történik semmi. A 17. sor mutat példát arra, hogyan lehet az aktuális és az előző elemet összehasonlítani (itt éppen azok ellenkező előjelűségét vizsgálja a program).

Másik megoldás lehet Python esetén az enumerate() használata, amely egy olyan iterátort hoz létre, amelynek elemei értékpárok: az első érték a paraméterben megadott elem indexe, a második érték annak értéke. Ezzel a fenti kód az alábbi formában írható:

```
22  szamlalo = 0
23  for index, ertek in enumerate(szamok):
24      if index == 0:
25          continue
26      if ertek * szamok[index-1] > 0:
27          szamlalo += 1
28
```

Mint látható, az aktuális elemre így könnyebb hivatkozni, de az előző, vagy a következő elemet csak indexeléssel lehet továbbra is elérni.

Arra az esetre, ha valamilyen résszorozat előfordulási számára vagyunk kíváncsiak, szintén a delfines feladatot lehet példának említeni: ebben a feladatban szerepel egy olyan kérdés, hogy hány alkalommal merült mélyre a delfin, ahol mélymerülésnek a -3 alatti értékek sorozata számít.

Ilyen esetben nem egy elemre kell vizsgálni a feltételt, és még csak nem is egymást követő elemek összehasonlításából áll a vizsgálat, hanem egy állapotot változását kell vizsgálni. Ehhez tehát a szokásos számlálón felül szükségünk van még egy állapotjelzőre is, amely jelzi a példa esetén, hogy éppen a mélyben van-e a delfin. Amennyiben két állapot között kell különbséget tenni (mélyben van-e vagy sem), akkor erre a célra egy logikai változó is elég, amely egy kicsit egyszerűbb feltétel megadást tesz lehetővé, mint ha több állapotot kellene vizsgálni, hiszen a logikai érték vagy hamis, vagy igaz lehet.

```
29  szamlalo = 0
30  melyben = False
31  for ertek in szamok:
32      if ertek < -3 and not melyben:
33          szamlalo += 1
34          melyben = True
35      if ertek > -4 and melyben:
36          melyben = False
37
```

A 32–34. sorok arra az esetre vonatkoznak, ha az eddigi értékekkel ellentétben egy -3-nál kisebb érték érkezik. Hogy az előző érték nem volt -3 alatti, azt a *melyben* logikai változó mondja meg, amelyet a ciklus elején hamis értékre állítva feltételezzük, hogy a delfin nincs a mélyben. Ha mély érték kezdődik, akkor a változó igazra állításával jegyzi meg a program, hogy elindult egy mélymerülés. Itt meg is történik a számláló megnövelése, hiszen találtunk egy új mélymerülést.

A 35–36. sor arra az esetre vonatkozik, amikor a mély szakasz végén jön egy nem mély érték (-3 vagy nagyobb érték). Ekkor megint a *melyben* változó gondoskodik arról, hogy mélymerülés végére érve teljesüljön csak a feltétel. Mivel a mélymerülés kezdetekor a szakaszt egyszer már számbavette a program, így itt a számlálót már nem kell növelni.

Jobban végiggodolva, nem szükséges ez a része a feltételnek, hiszen ha -3 vagy nagyobb érték érkezik, és a jelzőt hamisra állítjuk, de mást nem csinálunk, akkor a hamisra állítás nem okozna hibát akkor sem, ha nem ért véget egy mély merülés, tehát a jelző eleve hamison áll. Viszont többlet feladatot végezne a program.

5.2. Összegzések

Most összegzés alatt nem csak a tényleges összegzést értjük, hanem más olyan számítást is, amelyhez szükséges a lista bizonyos értékeinek az összege is, mint például az átlaguk. Utóbbi esetben nyilván az összegen kívül ismerni kell a szóben forgó elemek számát is, amelyet az előző pontban megnéztünk, hogyan lehet kiszámolni.

5.4. Hogyan lehet egy számokat tartalmazó lista elemeinek összegét kiszámolni?
5.5. Hogyan lehet a fenti listában azokat az elemeket összeadni, amelyek pozitívak?
5.6. Hogyan lehet kiszámolni a lista elemeinek átlagát?
5.7. Hogyan lehet kiszámolni a lista pozitív elemeinek átlagát?

A fenti kérdésekre adják meg az összegző algoritmusok a választ. Lássuk ezeket!

Nyilván egy lista elemeinek az összegét úgy kapjuk meg, ha végighaladva a lista elemein egy munkaváltozóba gyűjtjük az eddig érintett elemek összegét, ehhez mindig az aktuális elemet hozzáadva. Amennyiben egy adott feltételnek megfelelő elemeket kell csak összegezni, akkor a részösszeget csak akkor kell változtatni, ha az éppen vizsgált elemre igaz a feltétel.

Átlagol számítása esetén az összegzendő elemek számát is meg kell menet közben határozni, majd az összeg kiszámítása után azt elosztani a darabszámmal.

Az alábbi programkódok mutatják a fentieket Python nyelven megvalósító kódot:

```
4   # Mindegyik elem összege:
5   osszeg = 0
6   for szam in szamok:
7       osszeg += szam
8
9   print(osszeg)
```

```
17  print(osszeg)
18
19  # Mindegyik elem átlaga:
20  szamlalo = 0
21  osszeg = 0
22  for szam in szamok:
23      osszeg += szam
24      szamlalo += 1
```

```
11  # Pozitív elemek összege:
12  osszeg = 0
13  for szam in szamok:
14      if szam > 0:
15          osszeg += szam
```

```
28  # Pozitív elemek átlaga:
29  szamlalo = 0
30  osszeg = 0
31  for szam in szamok:
32      if szam > 0:
33          osszeg += szam
34          szamlalo += 1
```

5.8. A táblázatkezelő programok **SumIf()**, magyarul **SzumHa()** függvénye egy kicsit több lehetőséget nyújt ezeknél: lehetséges egy tartományra alkalmazni a feltételt és egy másik tartomány megfelelő celláját figyelembe venni az összegzés során. Az **AverageIf()**, magyar **ÁtlagHa()** függvény ugyanerre ad lehetőséget átlagolással.

Határozd meg az algoritmust, amely ezen függvényeket megvalósítja *F* feltétel-lista és *E* értéklista esetén, mintha a képlet a `=sumif(F; „>0”; E)` lenne!

5.3. Adott érték megkeresése

5.9. Hogyan lehet egy meghatározott értékű (vagy megadott feltételnek megfelelő) elem indexét megkeresni egy listában? Hogyan lehet az összes adott feltételnek megfelelő elem indexének listáját meghatározni?

Amennyiben egy listában egy adott feltételnek megfelelő elemet kell megkeresni, annak legegyszerűbb – de nem feltétlenül leggyorsabb – módja, ha a lista elejéről indulva az összes elemet megvizsgáljuk, amíg olyat nem találunk, amely megfelel a feltételnek. Ha az összes ilyen elemet kell megtalálni, akkor folytatjuk az elemek vizsgálatát a lista végéig és a megtalált elemek indexét egy listába tesszük, míg ha csak egy ilyen elemet kell produkálni, akkor az elsőnek megtalált elemnél befejezhető a ciklus.

```
4   # Adott értékű elem megkeresése:
5   def keres(lista, ertek):
6       for index, elem in enumerate(lista):
7           if elem == ertek:
8               return index
9
10  # Adott értékű összes elem megkeresése:
11  def tobb_keres(lista, ertek):
12      talalatok = []
13      for index, elem in enumerate(lista):
14          if elem == ertek:
15              talalatok.append(index)
16      return talalatok
```

A fentiek közül az első függvény *None* értéket, a második pedig egy üres listát ad vissza, ha a keresett értékű eleme nincs a listának.

Bizonyos feltételek teljesülése esetén gyorsabban is meg lehet keresni egy elemet vagy megállapítani, hogy benne van-e a listában. 9. osztályban, a számkitaláló játék esetén már használtuk ezt a módszert, amelyet **intervallum-felezés**nek hívnak. A módszer lényege, hogy minden elem vizsgálata után az addigi elemszám felére csökken a még lehetséges, ezért megvizsgálandó elemek száma. Mivel ez egyben a lista valamelyik felére redukálja a további vizsgálatot, így a még megvizsgálandó elemek a lista egy összefüggő intervallumát alkotják. Erre utal a módszer neve.

A fentebb bemutatott kereséseket **lineáris keresés**nek szokás nevezni, utalva arra, hogy a keresés során a lista elemeit sorra kell venni, így az összehasonlítások száma a lista elemszámával egyenes arányban nő.

Ezzel ellentétben az intervallum-felezés esetén mivel minden elemvizsgálat megfelezi a lehetséges elemek számát, jóval kevesebb vizsgálatra van szükség. Ha m számú vizsgálatot végzünk, azzal 2^m elemre lehet elvégezni a vizsgálatot. Megfordítva: ha a listának n eleme van, akkor nagyságrendileg $\log(n)$ összehasonlításra van szükség, ahol a logaritmus alapszáma jelen esetben 2.

Van azonban egy korlátja a módszer alkalmazhatóságának: a felezéshez szükséges vizsgálat csak akkor alkalmazható, ha a lista elemei valamilyen módon *sorba vannak rendezve*, azaz egy meghatározott rendezési szempont szerint **monoton növekvő sorozatot alkotnak a lista elemeinek értékei**. Ez a rendezési szempont számok esetén egyértelműen megadott, de szövegekre is létezik például a lexikografikus rendezés, összetett adatokra pedig szintén definiálható valamilyen rendezettség.

```
18  # Elem keresése intervallumfelezéssel:
19  def intervallum(lista, ertek):
20      eleje = 0
21      vege = len(lista)-1
22      while eleje <= vege:
23          vizsgalt = (eleje + vege) // 2
24          if lista[vizsgalt] < ertek:
25              eleje = vizsgalt + 1
26          if lista[vizsgalt] > ertek:
27              vege = vizsgalt - 1
28          if lista[vizsgalt] == ertek:
29              return vizsgalt
30      return -1
```

5.4. Szélsőérték keresése

5.10. Hogyan lehet megkeresni egy nem-rendezett lista legkisebb értékű elemét?

A fenti kérdésben szándékosan van kiemelve, hogy *nem-rendezett* listában kell a legkisebb elemet megkeresni. Rendezett lista esetén ugyanis egyértelmű lenne a feladat: ha növekvően rendezett a lista, akkor az első, csökkenő rendezettség esetén a legutolsó elem a legkisebb.

Ebből adódóan a legegyszerűbbnek az tűnne, ha előbb rendeznénk a listát. Ezzel azonban ugyanaz a probléma merül fel, ami az előző szakaszban bemutatott intervallum-felezés esetén: a rendezés nem is olyan egyszerű feladat. A következő leckékben szó lesz néhány rendező algoritmusról, ahol látható lesz, hogy azok bizony nem olyan gyorsak, hogy minden esetben értelme legyen a listát előbb rendezni a feladat megoldásához.

Tehát érdemes megismerni legalább egy olyan megoldást is, amely során nem kell felhasználni a lista rendezettségét! Vajon mi lehet ez a megoldás a legkisebb (vagy a legnagyobb) elem megkeresésére?

Továbbra is fontos észben tartanunk, hogy a számítógép képességei odáig terjednek, hogy két értéket összehasonlíthat és az összehasonlítás eredményétől függően cselekedhet. Egyszerre kettőnél több értékkel nem tud dolgozni! Ennek a kritériumnak megfelelően kell végigmenni a lista elemein, mindegyiket legalább egyszer megnézni és kiválasztani közülük a legkisebbet (vagy a legnagyobbat, ha azt keressük).

A fentiekből következik, hogy célszerű menet közben mindig feljegyezni az eddig megvizsgált elemek közül az eddigi legkisebbet (legnagyobbat), hiszen ha annál kisebbet (nagyobbat) nem találunk a lista még meg nem vizsgált elemei között, akkor az lesz a keresett elem.

Tehát szükségünk lesz egy változóra, amely az eddig megvizsgált elemek minimumát (maximumát) tárolja. Emellett ha nem csak az érték kell, hanem annak helye is, akkor a „*lokális minimum*” elemének indexét is fel kell jegyezni egy másik változóban. Amennyiben az aktuális vizsgált elem értéke kisebb ennél a lokális minimumnál, akkor ez az elem lesz az új lokális minimum.

Így amikor a lista utolsó elemét is megvizsgáltuk, akkor a lokális minimum egyben az egész lista legkisebb eleme is lesz.

A fentiek alapján látható, hogy az algoritmus időigénye egyenes arányban áll a lista elemszámával, amit hivatalosan az $O(n)$ kifejezéssel jelölnek, ahol n a lista elemszámára utal.

Az alábbi függvény ezt az algoritmust valósítja meg:

```
3  def legkisebb(lista: list) -> tuple(int):
4      minimum = lista[0]
5      mindex = 0
6      for index, érték in enumerate(lista):
7          if index == 0:
8              continue
9          if érték < minimum:
10             minimum = érték
11             mindex = index
12
13     return (mindex, minimum)
14
```

Amennyiben a listának csak az első elemét vesszük figyelembe, azon a lokális minimum nyilvánvalóan az első elem lesz, azért a *minimum* változóban ennek értékét, a *mindex* változóban ennek indexét (0) kell induláskor eltárolni. Ezután a 6. sorban induló ciklusnak már nem kell az első elem-

mel foglalkoznia. De ha a 7–8. sort kihagynánk, a függvény akkor is helyesen működne – csak fölöslegesen megvizsgálná újra az első elemet.

A függvény végül a megtalált legkisebb elem indexéből és értékéből álló értékpárt adja vissza. Amennyiben csak az értékre van szükség, akkor a *mindex* változó a programkódból teljesen elhagyható lenne.

A **min()** és a **max()** függvények valójában ezt az algoritmust valósítják meg a paraméterében megadott listák értékeivel. Ezek a beépített függvények azonban csak olyan esetben működnek, ha az elemek között egyértelműen definiált sorrend létezik. Abban az esetben, ha az elemek összetett adatok, amelyek egyikén levő rendezettséget kell használni, akkor már valószínűleg a programozónak kell definiálnia olyan függvényt, amely a legkisebbnek vagy legnagyobbnak számító elemet megkeresi, mint a fenti függvény is.

Érdemes még megjegyezni, hogy vannak olyan esetek is, amikor nem az egész listán, hanem annak egy intervallumán kell a keresést elvégezni, mint a következő leckében bemutatott egyik rendező algoritmus esetén. Ebben az esetben az indexek intervallumát is bemenő paraméterként kell megkapnia a függvénynek és a ciklusnak a lista ezen részén kell dolgoznia. Erre példát majd a következő lecke fog mutatni.

5.5. Feladatok

5.11. Lottó A `/public/programozas/algoritmusok/` könyvtárban található egy `lotto.data` nevű szöveges állomány. Ez tartalmilag megegyezik a 2005. májusi emelt szintű informatika érettségi utolsó, programozás feladatának (Lottó) a forrásállományával: az Ötös Lottó játék során 2003. során, az első 51 hétben kihúzott számokat tartalmazza, soronként egy-egy hét számait, egymástól vesszővel elválasztva.

Írj programot, amely beolvassa az állomány tartalmát és kiírja azoknak a heteknek a sorszámát, amelyben kihúzták a *Végső Számot*, azaz a 42-t!

5.12. Sipályák A `/public/programozas/algoritmusok/` könyvtárban találsz egy `siadat.txt` állományt, amely a 2009. május 15-i emelt szintű informatika érettségi táblázatkezelési feladatának forrásállománya volt. Most az ebben található adatokkal dolgozó programot kell készítened.

Az állomány a magyarországi sípályák adatait tartalmazza CSV formátumban. Az első sorral a programnak nem kell foglalkoznia, de az itt található nevek segítenek a további sorokban az egymástól tabulátorokkal elválasztott értékeket beazonosítani.

Írj programot, amely beolvassa az állomány tartalmát szótárak listájaként, majd választ ad az alábbi kérdésekre! A szótár kulcsai az állomány első sora alapján meghatározhatóak.

1. Hány sípálya található a Bükkben?
2. Melyik a leghosszabb sípálya?
3. Milyen adatokkal rendelkezik a Lillafüreden található sípálya?
4. Átlagosan milyen hosszúak a kékestetői sípályák (ezen pályák megnevezése a ’Kékestető, ’ szöveggel kezdődik)?

6. Rendező algoritmusok

Már az elemi algoritmusok bemutatásában is előkerült példa arra, hogy valamilyen feladatot sorba rendezett értékeken könnyebb megoldani. Ezen felül is lehetnek még olyan esetek, amikor szükség lenne adatok rendezésére. Ugyan minden manapság használt programozási nyelv kínál eszközöket a rendezésre, de ezek az eszközök nem minden esetben alkalmazhatóak. A Python például a **sorted()** függvényt és a listák **.sort()** metódusát kínálja erre a célra, amelyek ráadásul olyan jól optimalizáltak, hogy a saját magunk által írt kódnák az esetek túlnyomó többségében gyorsabban végzik el a rendezést. Ha egyszerűen egy lista elemeit szeretnénk rendezni, akkor érdemes ezeket használni saját megoldások helyett.

Azonban abban az esetben, ha a lista elemei maguk is összetett adatok, akkor a beépített rendezések paraméterezése már túl bonyolulttá válhat, vagy nem is lehet megfelelően paraméterezni azokat. Ilyenkor hasznos, ha ismerünk legalább néhány rendező algoritmust, amellyel a szükséges rendezést megoldhatjuk.

Az interneten rákeresve – főleg angolul – a rendezési módszerekre, nagyon hosszú listát kaphatunk. Az alábbi 23. ábrán a geeksforgeeks.org portál rendező algoritmusokat bemutató oldaláról látható egy lista, a portál készítői által ismert algoritmusok listájáról.

- Selection Sort
- Bubble Sort
- Insertion Sort
- Merge Sort
- Quick Sort
- Heap Sort
- Counting Sort
- Radix Sort
- Bucket Sort
- Bingo Sort Algorithm
- ShellSort
- TimSort
- Comb Sort
- Pigeonhole Sort
- Cycle Sort
- Cocktail Sort
- Strand Sort
- Bitonic Sort
- Pancake sorting
- BogoSort or Permutation Sort
- Gnome Sort
- Sleep Sort – The King of Laziness
- Structure Sorting in C++
- Stooge Sort
- Tag Sort (To get both sorted and original)
- Tree Sort
- Odd-Even Sort / Brick Sort
- 3-way Merge Sort

23. ábra: Rendező algoritmusok listája a geeksforgeeks.org oldalról

Elég valószínűtlen, hogy egy programozónak ezek mindegyikét ismernie kellene. Főleg nem fejből, hiszen ezen algoritmusok leírása például az említett oldalról is elérhető szükség esetén. Ahhoz viszont, hogy egy feladatban ha szükséges valamilyen lista elemeit kézzel (tehát nem a beépített függvények használatával) rendezni, érdemes egyet-kettőt ismerni közülük.

Az alábbiakban a három legalapvetőbb rendező algoritmus bemutatása következik, majd néhány feladat, amelyek olyan rendezést igényelnek, amelyek a beépített eszközökkel nem, vagy csak nagyon körülményes paraméterezéssel oldhatóak meg.

6.1. Minimumelvű rendezés

Az előző leckében megnéztük, hogyan lehet egy lista legkisebb elemét megkeresni. Ezt az algoritmust felhasználva működik az egyik talán legegyszerűbb rendezés:

Bemenet: a rendezendő *L* lista
Kimenet: *L* rendezve
Ciklus *i* ← 0-tól len(L)-ig:

Keressük meg az *i*. legkisebb elemet
Cserével tegyük a legkisebb elemet a lista *i*-edik helyére!

Ciklus vége

Vagyis az aktuálisan vizsgált részlista legkisebb elemét helyezzük el a részlista elejére, majd vegyük az egyel magasabb indexen kezdődő részlistát és ismételjük ezt meg mindaddig, amíg a vizsgálandó részlistának van eleme. (Nyilván ha már csak az utolsó elemből áll a lista, akkor célszerű megállni, hiszen egy elemű listánál evidens, hogy az az egy a legkisebb elem is egyben.)

Ehhez a legkisebb elemet kereső függvényt egy kicsit módosítanunk kell, hogy egy adott elemtől kezdődjön a keresés, ne mindig az elejétől.

A 24. ábra mutatja kiemeléssel, hol kell az előző leckében bemutatott függvényt megváltoztatni. Egyrészt szükséges egy második paraméter a függvénynek, amellyel megadjuk annak az elemnek az indexét, ahonnan a keresést indítani kell. A *minimum* és a *mindex* változók induló értékét ennek a paraméternek az értéke alapján kell beállítani és a **for** ciklusnak is figyelmen kívül kell hagynia minden olyan elemet, amelynek az indexe nem nagyobb ennél a kezdőértéknél.

```
3   def legkisebb(lista: list, kezd: int) -> tuple(int):
4       minimum = lista[kezd]
5       mindex = kezd
6       for index, érték in enumerate(lista):
7           if index <= kezd:
8               continue
9           if érték < minimum:
10              minimum = érték
11              mindex = index
12
13      return (mindex, minimum)
14
```

24. ábra: Legkisebb elem keresése megadott indextől kezdve

A függvény egy egyszerűbb – és kicsit gyorsabb – megvalósítását mutatja a 25. ábra. Ebben az esetben nem **for** ciklus, hanem **while** ciklus szerepel, így nem kell a ciklusnak fölöslegesen végigmennie a kezdőértéknél nem nagyobb elemeken. A lista elemeit azonban ebben az esetben az indexein keresztül lehet csak elérni, hiszen ilyenkor az értékek közvetlenül nem állnak rendelkezésre, mint amikor az **enumerate()** függvényt használtuk az első változatnál.

```
3   def legkisebb(lista: list, kezd: int) -> tuple(int):
4       minimum = lista[kezd]
5       mindex = kezd
6       index = kezd + 1
7       while index < len(lista):
8           if lista[index] < minimum:
9               minimum = lista[index]
10              mindex = index
11
12      return (mindex, minimum)
13
```

25. ábra: Legkisebb elem keresése megadott indexű elemtől kezdve – while ciklussal

Immár csak az a kérdés, hogy hogyan lehet az aktuális rész-lista legkisebb elemét előre helyezni: ki kell cserélni azzal az elemmel, amelyik a helyén van.

Ennél a cserénél arra kell figyelni, hogy közben egyik érték se vesszen el. Azaz azt az értéket, amelynek a helyére akarjuk a másikat tenni, először egy ideiglenes változóba félre kell tenni:

```
temp = egyik
egyik = másik
másik = temp
```

Ahogyan a fenti kódrészlet is mutatja, három értékadással lehetséges két változó értékét megcserélni: első lépésben az egyik cserélendő értéket (*egyik* változóból) elmentjük egy ideiglenes változóba (*temp*), majd a változó értékét a másik cserélendő változóval (*másik*) felülírjuk, végül ebbe a másik változóba a félretett értéket bemásoljuk. (Mindezt úgy lehet a legkönyebben megérteni, ha egy kézzel megpróbálunk két egymás melletti tányérban levő süteményt felcserélni.)

Python esetében a fenti három értékadást azonban helyettesíthetjük egyetlen értékadó utasítással, kihasználva a *tuple* típus használatának lehetőségét:

```
egyik, másik = másik, egyik
```

Valójában a színfalak mögött itt is három értékadást fog végezni a számítógép, azonban ez egyetlen utasítással van megadva, amelyben egyszerre két változó kap két-két értéket, amely a nyelv definíciója szerint teljesen egy időben hajtódik végre.

A fentiek után végül lássuk a minimumelvű rendező algoritmus megvalósítását Python függvény formájában:

```
13
14  def minrendez(lista: list):
15      """
16      A függvény a paraméterben megkapott listát helyben rendezi
17      a minimumelvű rendezés algoritmusát használva.
18      """
19      for start in range(len(lista)-1):
20          minimum = legkisebb(lista, start)
21          if minimum[0] != start:
22              lista[start], lista[minimum[0]] = lista[minimum[0]], lista[start]
23
```

26. ábra: A minimumelvű rendezést megvalósító függvény

A függvény egyetlen ciklust tartalmaz, amely a lista első elemétől az utolsó előtti elemig veszi fel az elemek indexeit. Azért az utolsó előtti elemig halad, mert az utolsó elem esetében már triviális, hogy az maradt a legnagyobb értékű elem.

A ciklusban a 20. sor a *minimum* változóba egy értékpárt helyez, amelynek első eleme a megtalált legkisebb értékű elem indexe. Az algoritmus további része az elem értékével nem foglalkozik, csak annak helyével, hiszen a 22. sorban ezt az elemet és a részlista kezdő elemét cseréli ki. De csak akkor, ha ez két különböző eleme a listának! Amennyiben a kezdő index és a legkisebb elem indexe azonos, akkor a cserére nem kerül sor, hiszen nincs is rá szükség. Ha ezt a vizsgálatot kihagynánk, a függvény akkor is működne, ám ha a lista nem egyszerű elemeket tartalmaz, hanem minden elem több száz byte méretű – ekkor nyilván a legkisebb elemet kereső függvényt is módosítani kell –, akkor elég komoly méretű fölösleges adatmozgatást spórolhat meg ez a vizsgálat.

Ha már felmerült, akkor nézzük meg, vajon mit kell a 26. ábrán látható függvényen módosítani, ha nem egyszerű egész számok listáján, hanem valamilyen összetett adatszerkezetek listáján kell a rendezést végrehajtani! Valójában semmit nem kell módosítani, hiszen a függvény nem foglalkozik magukkal az értékekkel. Ellenben a **legkisebb()** függvényt át kell írni úgy, hogy a megfelelő vizsgálattal keresse meg a legkisebbnek számító elemet és annak indexét.

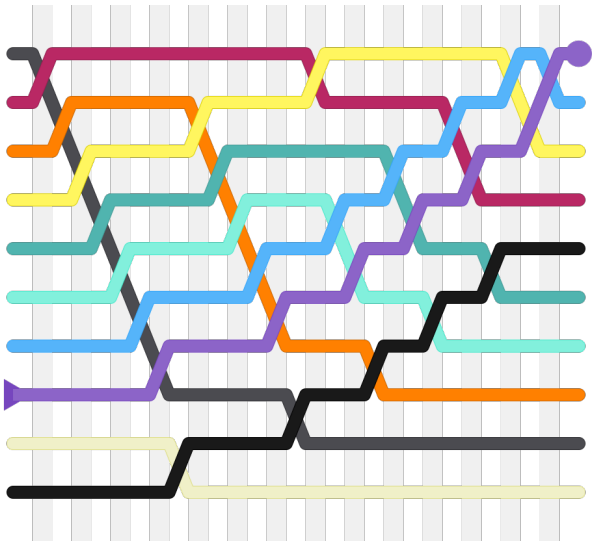
> Megjegyzés: A valóságban a minimumelvű rendezéshez használt **legkisebb()** függvénynek elegendő lenne csak a legkisebb elem indexét visszaadni, amely a **minrendez()** függvény kódját is kicsit egyszerűsíteni, hiszen csak egy egész értéket adna vissza a függvény, nem egy értékpárt. Sőt: az igazság az, hogy ha más célra keressük a legkisebb elemet, akkor is elég annak indexét visszaadni, hiszen az alapján egy egyszerű indexelő kifejezéssel megszerezhető az adott elem értéke.

6.2. Buborékrendezés

A buborékrendezés a nevét onnan kapta, hogy a kisebb elemek (amelyeknek a lista elején lenne a helyük) úgy jönnek lépésről lépésre a lista elejére, mint ahogyan a buborékok szállnak fel a mélyből a felszín felé. Eközben pedig a nagyobb (~nehezebb) elemek egymás után kerülnek a helyükre a lista végén.

A rendezés során a lista elejéről a vége felé haladva mindig a két szomszédos elemet hasonlítjuk össze és ha nem jó a sorrendjük, akkor megcseréljük őket. Amikor a lista végére érünk, az utolsó elem a helyére került, hiszen minden összehasonlításnál ha szükséges csere, akkor ez az elem egyel hátrább kerül a listában, így a következő összehasonlításban is részt vesz, tehát az összehasonlítások következtében a lista végére tolódik. Így következő menetben már nem kell a lista utolsó elemével foglalkozni.

Viszont a kisebb elemek még nem feltétlenül kerülnek a helyükre, így újabb és újabb menetben kell ismételni a szomszédos elemek összehasonlítását és cseréjét.



27. ábra: Az értékek vándorlása a buborékrendezés során (forrás: Wikipédia)

A 27. ábra mutatja egy példán ahogy a buborékrendezés során az egyes elemek változtatják a helyzetüket a listában. A https://en.wikipedia.org/wiki/File:Bubble-sort-example-300px.gif címen elérhető animált GIF folyamatában is bemutatja, hogyan zajlik le a rendezés.

A rendező algoritmusok közül talán ez a leglassabb, ezért a gyakorlatban nagyon ritkán használt. Kicsit lehet rajta gyorsítani ha figyelembe vesszük, hogy amennyiben egyszer a lista még rendezetlennek tekintett részén úgy végig lehet haladni, hogy egyetlen cserét sem kell végrehajtani, az azt jelenti, hogy a gyakorlatban az a része is rendezett már a listának, azaz több cserére már nem lesz szükség. Ekkor tehát a rendezést befejezettnek lehet tekinteni.

Ezért a gyakorlati megvalósítás során érdemes figyelni egy logikai változó segítségével, hogy volt-e csere a listán történő végighaladás során, hiszen ha nem volt, akkor befejezhetp a rendezés.

```
1
2
3  def bubblesort(L: list):
4      """
5      A függvény a buborékrendezést valósítja meg
6      :param L: a rendezendő lista
7      """
8      for j in range(len(L)-1, 1, -1):
9          voltcsere = False
10         for i in range(j):
11             if L[i] > L[i+1]:
12                 L[i], L[i+1] = L[i+1], L[i] # csere
13                 voltcsere = True
14         if not voltcsere:
15             break
16
```

28. ábra: A buborékrendezést megvalósító függvény kódja

A 28. ábra mutatja a rendezést megvalósító függvény kódját (az ábráról a mindig azonos tartalmú első sor maradt csak le). Az *L* lista rendezése egy dupla **for** ciklusban történik, amelyből a külső azt határozza meg, hogy melyik legyen az utolsó elem, amelyet az adott menetben a rendezésnek érintenie kell. Ennek a ciklusnak a ciklusváltozója, a *j* változó tehát az utolsó elemtől a második elemig halad, amit a `range(len(L)-1, 1, -1)` kifejezéssel érünk el: `len(L)-1` a lista utolsó eleme, innen indulunk; `1` a második elem indexe, eddig halad visszafelé a *j* változó értéke; mert a harmadik `-1` paraméter azt adja meg, hogy egyesével csökkennie kell az értékeknek.

A belső ciklus előtt először az állapotjelző *voltcsere* változó hamis értékre állítása történik. Ezzel a változóval ellenőrizhető a belső ciklus végrehajtása után, a 14. sorban, hogy kell-e még foly-

tatni a rendezést és ha nem, akkor a 15. sor **break** utasításának hatására a külső ciklus is befejeződik, mert *L* rendezve van.

A belső ciklus végigmegy a lista még nem rendezett első részén és ha talál két egymás melletti, nem jó sorrendben levő elemet, akkor nem csak kicseréli azokat a 12. sorban, hanem a 13. sor utasítása a *voltcsere* változót igaz értékre is állítja, jelezve, hogy még legalább egy további menetre szükség lesz.

Elemezve az algoritmus működését azt láthatjuk, hogy a nagyobb elemek, amelyeknek a helye a lista vége felé van, kifejezetten gyorsan a helyükre kerülnek. Ellenben egy olyan elem, amelynek a lista eleje felé kell vándorolnia, minden menetben csak egy hellyel tud előrébb kerülni. Így ha a legkisebb elem a lista végén van, akkor *n*-1 menet kell a helyére juttatáshoz, ahol *n* a lista elemszáma, míg a legnagyobb elem akár a lista elejéről is az első menetben eljut a helyére.

Ezt az aránytalanságot tudja kiegyenlíteni – és ezzel futási időt spórolni – az algoritmus olyan módosítása, ahol a menetek felváltva a lista elejéről és a végéről indulnak. Ezt a módosított változatot nevezzük *rázva rendezés*nek, vagy az angol neve alapján tükörfordítva *koktél-rendezés*nek.

Másik változata a fésű rendezés, ahol a buborékrendezés megkezdése előtt néhány előrendezési fázisban nem az egymás melletti elemek kerülnek összehasonlításra, hanem egy adott, egyenlő távolság van az összehasonlítandó elemek között. Ezzel a lista elejére való elemek gyorsabban a lista eleje közelébe tudnak ugrani.

6.3. Beszúró rendezés

Aki szokott kártyázni, annak talán ismerősen hangzik, hogy a kézben tartott lapokat is sokan szeretik sorba rendezni. Ez nem biztos, hogy jó ötlet, mert a többieknek adhat információt, hogy hogyan cserélgeti valaki a kezében levő lapokat. Mindenesetre állítólag innen származik ennek a rendezési módszernek a neve.

A módszer lényege, hogy egyesével véve az egyes elemeket, az eddig már rendezett részlistán belül megkeressük annak a helyét.

Az első elemnél ez nagyon egyszerű: ez lesz a rendezett lista első eleme.

A következő elemnél azt kell megnézni, hogy a rendezett lista aktuális utolsó eleménél kisebbe. Ha nem, akkor a lista végére kerül. Ha viszont kisebb, akkor az egyel előrébb levő elemmel kell folytatni az összehasonlítást mindaddig, amíg el nem érjük vagy a lista elejét vagy egy olyan elemet, amelynél a vizsgált elem már nagyobb. Ekkor ez után az elem után (vagy ha már a lista elején vagyunk, a lista elejére) kell beszúrni az elemet.

Amikor az utolsó elem is a helyére került, akkor a lista rendezve lesz.

Az algoritmusnak két megvalósítási módja létezik: tömbökön működő és listákon működő.

A tömbökön működő esetén minden olyan vizsgálat esetén, amelynél az elem, amelynek a helyét keressük kisebb a vizsgált, már a rendezett listában levő elemnél, a vizsgált elemet a lista egyel nagyobb indexű helyére kell átmozgatni, így csinálva helyet a beszúrandó elemnek. Ebben az esetben egy elem beszúrása *n* összehasonlítás mellett *n* adatmozgatást is igényel, ahol n szokás szerint a lista elemszámát jelenti. Az *n* elem beszúrása és ezáltal az egész rendezés így n^2 összehasonlítást és n^2 adatmozgatást igényel.

Amennyiben az adatok nem szigorúan egymás utáni memóriacímeken elhelyezkedő tömbökben vannak, hanem a sokkal rugalmasabban kezelhető lista adattípusban, akkor a beszúrandó elem helyének keresése közben nem kell adatmásolást végezni, csak a megfelelő helyre be kell szúrni az új elemet a listába. Ekkor is beszúrandó elemenként n összehasonlítás szükséges, de csak 1 adatmozgatás. Vagyis a teljes rendezésre n^2 összehasonlítás és *n* adatmozgatás szükséges.

Ha a rendezendő adatok mérete nagy, akkor a listás megoldás sokkal kevesebb idő alatt végrehajtható, mint a tömbös megoldás.

Mivel a Python a lista adattípuson biztosítja a tömböknél szokásos indexelés lehetőségét is, így ennél a programnyelvnél a lista adattípuson bemutatható mindkét változat megvalósítása:

```
1   #! /usr/bin/python3
2
3   def beszurtomb(L: list):
4       """
5       A beszúró rendezés megvalósítása tömbökön.
6       """
7       for i in range(len(L)):
8           if i == 0:
9               continue    # Induláskor az első elemből áll a rendezett lista
10          j = i - 1       # A rendezett lista utolsó eleme
11          beszurando = L[i]
12          while L[j] < beszurando and j > 0:
13              L[j+1] = L[j]   # egyel feljebb toljuk
14              j -= 1
15          L[j] = beszurando
16
17  def beszurlista(L: list):
18      """
19      A beszúró rendezés megvalósítása listákon.
20      """
21      for i in range(len(L)):
22          if i == 0:
23              continue    # Induláskor az első elemből áll a rendezett lista
24          j = i - 1
25          beszurando = L[i]
26          while L[j] < beszurando and j > 0:
27              j -= 1
28          # beszúrása az új helyre és a régi helyről eltávolítás:
29          L.insert(j, L.pop(i))
30
```

29. ábra: Beszúró rendezés tömbbel és listával megvalósítva

Összehasonlítva a két függvényt, két sorban térnek csak el, mégis az alsó jóval kevesebb adatmozgást eredményez. A tömbös megoldásnál a 13. sorban mindig egyel fentebb kell tolni a tömb elemét a következő indexű elembe történő átmásolással, majd ha megvan a beszúrandó elem helye, akkor azt oda be kell írni. A listás megoldásnál a feljebb tologatás a ciklusból eltűnt, helyette a 29. sorban a **.pop()** művelet kiveszi a beszúrandó elemet a listából, míg az **.insert()** művelet beszúrja azt az új helyére.

A listás megoldásnál azért nem lesz baj az elemek eltávolításából és beszúrásából a **for** utasítás szempontjából, mert pontosan ugyanannyi eleme marad a listának a 29. sor végrehajtása után, mint előtte volt és az *i* index alatt történik a helyváltoztatás, nem utána. Így nem kell attól tartani, hogy esetleg ugyanazt a listaelemet kétszer próbálja a függvény áthelyezni.

6.4. Feladatok

...

Adatbáziskezelés

Amikor már olyan nagy mennyiségű adattal kell dolgozni, amit táblázatkezelőben nem lehet megfelelően kezelni – mert túl nagy lenne a táblázat mérete, vagy csak átláthatatlan –, akkor valamilyen más, hatékonyabb eszközhöz érdemes fordulni.

Ilyen eszköz az adatbázis, amelyben hatalmas mennyiségű adatot lehet úgy használni, hogy az mégis átlátható maradjon. Ehhez természetesen nem elég az adatokat csak úgy, „ömlesztve” eltárolni, hanem azokban valamilyen logika szerint rendet kell tenni. Az ilyen rendszer lehetővé teszi, hogy az adatok közti kapcsolatok átláthatóak legyenek, egyben eszközöket is ad az adatokhoz való hozzáférésre, azokból olyan információk kinyerésére, amely azok rendszerezése nélkül nem lennének elérhetőek.

Ebben a témakörben az adatbázisokkal és azok kezelésére alkalmazott eszközökkel fogsz megismerkedni.

Az adatbázisokat rendkívül sok helyen használják az adatok nyilvántartására. Sok olyan példát is lehet hozni, amelyről elsőre nem is gondolná azt az ember, hogy adatbázis. Például egy vasúti menetrendet is tekinthetjük adatbázisnak. Ugyanígy adatbázis húzódik meg a hírhedt Kréta rendszer mögött is, amely az egyes iskolák digitális naplójaként is funkcionál. Gyakorlatilag minden olyan vállalkozás esetében, ahol vannak alkalmazottak, azok egyes adatait tárolni kell – funkcióját tekintva szintén adatbázisban –, például az állam felé fennálló kötelezettségek (pl. adóelszámolás) teljesítése céljából. De valójában minden olyan internetes oldal mögött, amelyen bejelentkezési lehetőség is van, áll egy adatbázis, amely – ha mást nem is – a felhasználók azonosítóját és jelszavát tárolja. Ma egyre több weboldalt egy CMS (Content Management System) működtet, amely az oldalon megjelenő tartalmat valójában egy adatbázisban tárolja. A Youtube, Facebook, Twitter és társai mögött éppúgy adatbázis lapul, mint a keresőrendszerek mögött, mint amilyen például a Google. A továbbiakban fogsz majd konkrét példákat is látni adatbázisokra – például a feladatok formájában.

Sokszor ezen alkalmazások esetében fel sem tűnik, hogy adatbázist használunk. Ennek az az alapvető oka, hogy nagyon ritka – gyakorlatilag nulla a valószínűsége – a valós életben annak, hogy valaki közvetlenül fér hozzá egy adatbázishoz. Inkább egy olyan programot használ, amely megfelelően definiált felületen keresztül ad hozzáférést az adatbázishoz. Az ilyen programokat szokás **információs rendszer**nek nevezni. Például egy utazási irodában ha lehet repülőjegyet venni, akkor az ügyintéző egy ilyen információs rendszert használ a helyfoglaláshoz, a program intézi a szükséges adatbázis-műveleteket. Szintén igaz, hogy egy banki alkalmazottnak nincs joga közvetlenül belenyúlni a bank ügyfeleinek számláit nyilvántartó adatbázishoz, csak azon a programon keresztül, amely egy jól megtervezett, megfelelő biztonsági protokollokat tartalmazó felületen keresztül fér hozzá a számlákhoz. Éppezért nem érdemes hinni annak – akár filmben, akár egy átveréssel próbálkozó, magát banki szakembernek kiadó személyről van szó –, aki azt állítja, hogy közvetlenül, nyom nélkül meg tudja változtatni valamely ügyfél személyes adatait, például a számla örökösének megjelölt személy kilétét (ahogy ezzel gyakran próbálkoznak internetes csalók).

Egy adatbázis sokkal bonyolultabb, mint egy Excel táblázat (még ha annak több munkalapja is van). Éppen ezért mielőtt létrehozunk egy adatbázist, azt alaposan meg kell tervezni. A feladatok

során a valódi adatbázisoknál sokkal egyszerűbbre lebutított adatbázisok fognak szerepelni, így ezt a tervezési lépést a gyakorlatban akár el is hagyhatnánk. Ám a feladatok megoldásának elengedhetetlen feltétele az adatbázis logikájának megértése, amelyet nagyban segíthet, ha azt – legalább fejben – vissza lehet alakítani egy szemléletes, grafikusan ábrázolható modellé. Ezért az adatmodellezésnek az alapjai fognak itt először szerepelni.

Ezután megismerkedsz az adatbázisok szabványossá vált nyelvével, az SQL nyelvvel, és ezen keresztül azzal, hogyan lehet egy adatbázist és a benne szereplő adatokat használni, alakítani. Aki azt tervezi, hogy valamilyen programot fog írni, amellyel egy adatbázishoz lehet hozzáférni (lásd a fentebb említett információs rendszereket), annak ezt a nyelvet kell ismernie.

Ezután megismerkedünk néhány olyan klienssel, amely még képes valamilyen formában adatbázis-hozzáférést biztosítani.

7. Az egyed-kapcsolat modell

Az adatbázis tervezéséhez használt adatmodellek kettővel fogunk megismerkedni – valamilyen szinten. Egyiket sem fogjuk olyan részletesen tárgyalni, mint a kifejezetten adatbázisokkal foglalkozó tanfolyamokon, egyetemi kurzusokon, csupán olyan alapszinten ismerkedünk meg velük, ami a későbbi leckék és az adatbázis-kezelési feladatok megértéséhez szükséges. Lesz azonban néhány olyan fogalomról szó ebben és a következő leckében, amelyekre a későbbiekben is szükség lesz!

Az adatmodellezés azért szükséges, mert egy adatbázis túl bonyolult ahhoz, hogy előzetes tervezés nélkül is esély legyen jól használható szerkezet felépítésére. Először is fontos, hogy az adatbázis a valóságnak mindig csak egy olyan részletét írja le, amely az adatbázis célja szempontjából fontos. Ezért szükséges egy matematikai modellt alkotni a valóság ábrázolandó részének logikus leírására. Ennek a modellnek az elkészítése a célja az adatmodellezésnek.

Fontos szempont, hogy a modell elkészítése során csak olyan részeit szabad a valóságnak figyelembe venni, amelyek az adatbázis célja szempontjából lényegesek. Vagyis például ha tanulók adatait kell egy iskolai nyilvántartásban tárolni, mint amilyen az elektronikus napló, akkor ott teljesen felesleges adat az, hogy az egyes tanulóknak milyen a szemük vagy a hajuk színe.

A mi szempontunkból inkább fordított célja lesz: az adatmodellek ismerete segíthet az egyes feladatokban létező adatbázisok felépítésének megértésében, amely nélkül a feladatot nem lehetne megoldani.

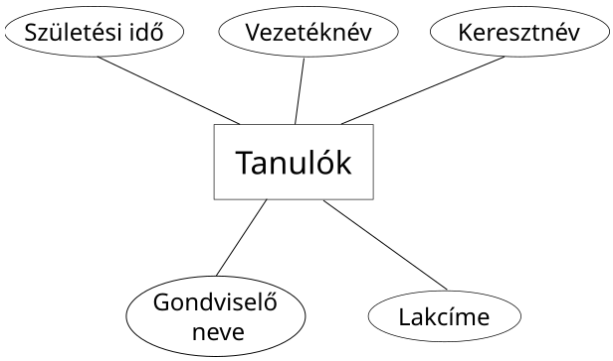
A két megismerendő adatmodell közül az első egy olyan, amely az adatbázisban tárolandó adatok közötti kapcsolatokat grafikusan szemlélteti. Ez az **Egyed-kapcsolat modell**, amelyről ez a lecke szól. A következő lecke mutatja be a másik adatmodellt, amely már a legelterjedtebb adatbázis-rendszerekben alkalmazott rendszer matematikai modelljének tekinthető, és amely a *relációs modell* nevet viseli.

7.1. Az Egyed-kapcsolat diagram

Az Egyed-kapcsolat modell valójában nem más, mint egy vagy több *Egyed-Kapcsolat diagram*, angolul *ER-diagram*. Valójában nem csak egy adatbázis megtervezésére alkalmas, hanem általában különböző dolgok egymáshoz való viszonyának ábrázolására. Éppen emiatt használható egy adatbázis megtervezésére is, hiszen az adatbázisban is különböző dolgok tulajdonságait kell úgy tárolni, hogy az azok közötti kapcsolatok is bekerüljenek az adatbázisba.

Mint általában az informatika területén – az angol Wikipedia ebben az esetben is elég kimerítően bemutatja az egyed-kapcsolat diagramot: https://en.wikipedia.org/wiki/Entity–relationship_model. A következőkben ennek a modellnek az alapjainak bemutatása következik.

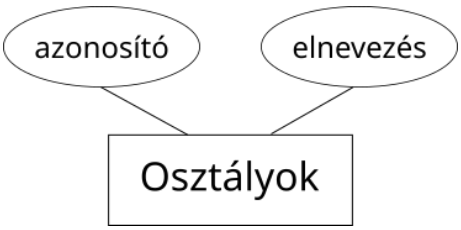
A modell a különböző dolgokat hasonló egyedek halmazaiba sorolja, az így kapott halmazok neve: **egyedhalmaz**. Az egyes egyedhalmazokat a diagramon egy-egy téglalap jelzi. Az adott egyedhalmazba tartozó valamennyi egyednek ugyanolyan tulajdonságai fontosak: ezek az **attribútumok**, amelyeket a diagramon az egyedhalmazt jelképező téglalaphoz vonallal kapcsolódó ovális jelzik.



30. ábra: A Tanulók egyedhalmaz

Az 30. ábrán a *Tanulók* egyedhalmaz látható azokkal az attribútumokkal, amelyeket például egy elektronikus naplóban nyilván kell az illetőről tar-

tani: mikor született, hogy hívják, ki a hivatalos gondviselője, hol lakik. Az ábráról lemaradt a születési hely, amelyet szintén célszerű az attribútumok közé felvenni.



31. ábra: Az Osztályok egyedhalmaz

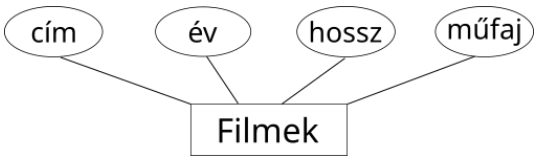
Nem szerepelnek olyan egyértelműen fontos adatok az attribútumok között, mint az, hogy melyik osztályba jár, melyik tantárgyból milyen jegyei vannak. Ennek oka, hogy ezek az adatok valahol máshol szerepelnek, mivel azok nem csak az egyes tanulók tulajdonságai, hanem az egyed más egyedhalmaz egyedeivel való viszonyát írják le.

Tovább folytatva ezt a példát, az osztályokhoz az egyedhalmaz a 31. ábrán látható. Az elnevezés az osztály használt elnevezése, például „11/A”, míg az azonosító valami olyan, ami a tanévek között nem változik, így az osztály létrejöttétől annak megszűnéséig alkalmas arra, hogy az osztályra az adatbázis más részeiről hivatkozni lehessen. Például: „2035A”, ha az osztály 2035-ben alakult. Nem szerepel más attribútum. Vajon miért? Az osztályfőnök neve nem szerepel, nem azért, mert változhat, hanem mert a tanárok között az adatai már szerepelnek nyilván, ezért ahogyan a tanuló osztályzatai esetében is, majd valamilyen más eszköz kapcsolja össze a tanárt az osztállyal, amelynek osztályfőnöke. Szintén nem szerepel az osztály létszáma, amely egy számolt adatként megkapható abból, hogy hány tanuló van az adott osztállyal összekötve. Ha az osztályoknak az iskolában van saját tantermük, azt sem célszerű attribútumként megadni, hiszen valószínűleg lesz a tantermeknek is egy saját egyedhalmaza, amely annak további tulajdonságait is megadja, így elég lesz összekötni a két egyedhalmazt egymással, hogy megadjuk az osztályok és a tantermek közti viszonyt.

A fentiekben több célzás is volt arra, hogy az egyedhalmazok egymáshoz való viszonyát is majd valahogyan meg kell adni. Erről majd a következő szakasz fog szólni.

Hasonló módon továbbgondolva a példát, még a tantárgyak egyedhalmaza is felmerül majd. Az órarend megadásához, illetve az egyes tanulók osztályzatainak megadásához viszont majd szintén szükség lesz a következő szakaszban szereplő ismeretekre, így a példát ott folytathatjuk.

Másik példa lehet a filmeket összegyűjtő egyedhalmaz (32. ábra):



32. ábra: A Filmek egyedhalmaz

Mik azok a tulajdonságai egy filmnek, amely az egész filmre vonatkozik és nem függ más típusú egyedektől? Először is a címe. Azzal a problémával, hogy több filmnek is lehet ugyanaz a címe, majd a lecke egy későbbi részén fogunk foglalkozni. Ugyanígy a film tulajdonsága lehet, hogy mikor készült – pontosabban, hogy hivatalosan mikor mutatták be. Ez azonban lehet az évszámnál pontosabban megadva: a bemutató dátuma is. Normális esetben egyedi tulajdonsága a filmnek, hogy hány perc a hossza. Itt most tekintsünk el attól, hogy lehet ugyanannak a filmnek több különböző hosszú változata is (pl. rendezői változat esetében), mivel azokat a változatokat külön filmnek kell tekinteni. Az ábrán jelölt negyedik attribútum, a műfaj már kényes kérdés: biztos, hogy egy adott műfajba sorolható a film? Ha nem, azaz ha egy adott film esetében az attribútum több értéket is kaphat egyszerre, akkor már érdemes elgondolkodni azon, vajon nem lenne-e jobb másképpen megadni azokat az értékeket az attribútum használata helyett.

Általában is igaz: ha egy tulajdonság esetleg egynél több értéket is kaphatna egyszerre, akkor nem a legjobb megoldás azt a tulajdonságot az egyedhalmaz tulajdonságának választani.[6]

6 A későbbiekben használt adatbázisok egyike, a PostgreSQL esetében lehet az attribútumhoz rendelt adattípus tömb is, így akár a több értékű attribútum sem mindig hiba.

A film rendezőjét érdemes attribútumként felvenni? A szereplőket nyilván az előbb bemutatott okból – vagyis hogy általában több szerepelője van egy filmnek – nyilván nem. A filmet készítő stúdiót nem csak azért nem attribútumként szerepeltetjük, mert több is lehet belőle egy film esetén, hanem azért is, mert további információkat is akarunk esetleg a stúdióhoz rendelni.

Mind a stúdió, mind a színészek, mind a stúdiók esetében valószínű, hogy azokról további információkat is fog az adatbázis tartalmazni, nem csak a nevüket. Ekkor viszont nyilván önálló egyedhalmazuk lesz ezeknek: *Rendezők*, *Színészek*, *Stúdiók*. Sőt, továbbgondolva lehet *Producerek*, *Operatőrök*, stb. egyedhalmazra is szükség. Hogy melyik szerepel, az már attól függ, mi az adatbázis célja, azaz melyik fajta egyedről mennyire részletes információkra van szükség. Érdemes lehet a valós életet ismerve azon is elgondolkodni, hogy mivel egy színész is lehet rendező vagy producer egy másik film esetében, érdemes-e ezeket tényleg önálló egyedhalmazként létrehozni. Ugyanakkor lehetnek olyan tulajdonságok is, amelyek csak egyik vagy másik szerepkörben számítanak. Itt már olyan területre tévedünk, amelynél az *alosztályok* használata merül fel, amelybe legfeljebb érintőleg fogunk belenézni a lecke vége felé.

Akárhogyan is hozzuk létre az egyedhalmazokat, azok egyedei kapcsolatban állnak az egyes filmekke, amely kapcsolatot még valamilyen módon meg kell határozni.

7.1.1. Kapcsolatok

Az egyes egyedhalmazok tehát azonos jellegű egyedeket – például autókat, embereket, vállalatokat – tartalmaznak. Azonban ezeknek az egyedeknek valamilyen kapcsolata is van egymással. A modell lehetővé teszi, hogy az egyes egyedhalmazok között **kapcsolatokat** adjunk meg, amelyekkel jelezhető, milyen lehetséges (vagy kötelező) kapcsolat van az egyes egyedhalmazokba tartozó egyedek között.

Például ha van egy autókat és egy embereket megadó egyedhalmazunk, akkor az *Autója* nevű kapcsolat (33. ábra) adhatja meg a két egyedhalmaz közötti kapcsolattal, hogy az egyes embereknek lehet az autók közül autója. Természetesen ez nem azt jelenti, hogy az összes embernek autója az összes autó! Csupán azt jelzi, hogy vannak olyan autók, amelyek valamely embernek a tulajdonában vannak.

Ráadásul van arra is lehetőség, hogy azt is jelölje valamilyen módon a diagram, hogy lehet-e ugyanannak az egyednek a kapcsolata a másik egyedhalmazból több egyeddel. Ahogyan arra is van lehetőség, hogy olyan kapcsolatot adjunk, amely egy adott egyedhalmaz két egyede között ír le kapcsolatot. Továbbá arra is van lehetőség, hogy egy kapcsolat kettőnél több egyedhalmaz közti kapcsolatot adjon meg.

Erre példa az az eset, amikor meg akarjuk adni, hogy egy adott tanuló melyik osztályba jár. Nyilván egy osztályba több diák is jár, de nincs olyan diák, aki egyszerre több osztályba is járna. Ekkor egy nyíllal jelezzük, ha egy egyedhalmazhoz csak egy egyed tartozhat a másik egyedhalmaz valamely eleméhez (33. ábra). Az ilyen kapcsolatot egyébként egy-sok kapcsolatnak nevezi a szakirodalom. Érdemes megjegyezni, hogy nem ugyanazt jelenti, ha legfelejebb vagy pontosan egy egyed tartozhat a másik egyedhalmaz valamely eleméhez, így ezt a két esetet elvileg különböző stílusú nyíllal kellene jelölni – de ez már a téma olyan részére vezetne, amellyel itt nem foglalkozunk.

33. ábra: Emberek és Autók közti kapcsolat: Autója

34. ábra: A Tanulója kapcsolat egy sok-egy kapcsolat

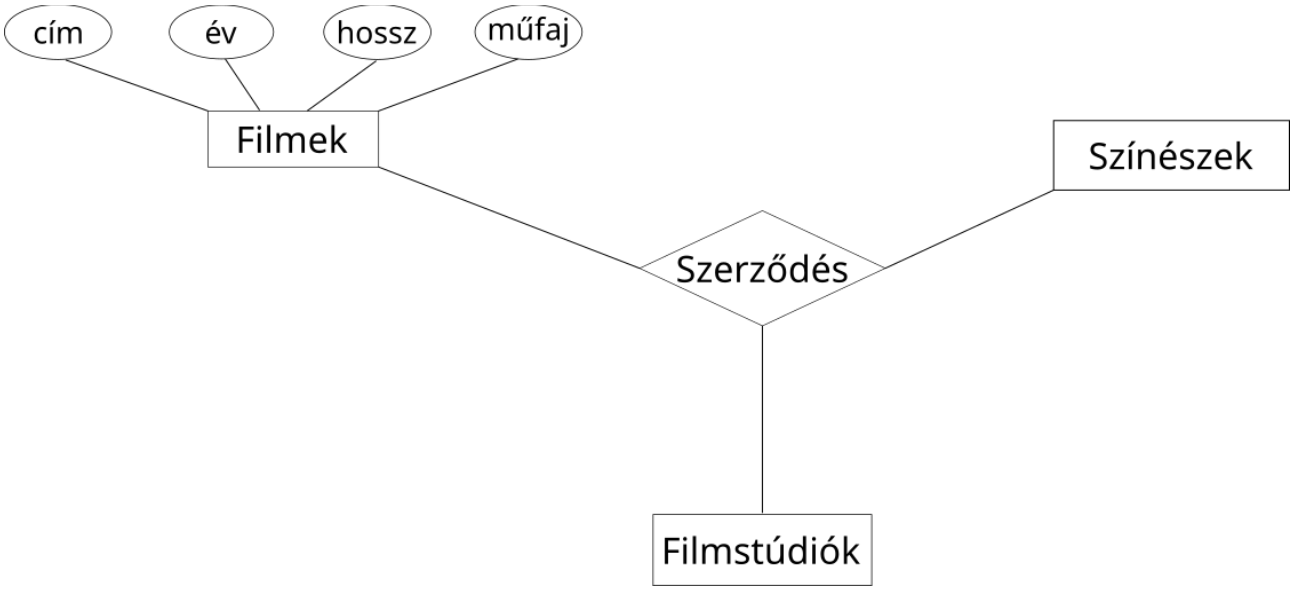
Tehát ha a kapcsolat egyik vonala végén egy nyilat látunk, az arra utal a kapcsolatban abból az egyedhalmazból egyszerre csak (pontosan vagy legfeljebb) egy egyed vehet részt. Tehát a 33. ábra példája esetében ha a *Tanulók* táblából kiválasztunk egy egyedet (tehát egy tanulót), akkor azt a Tanulója kapcsolat (pontosan) egy osztállyal kapcsolja össze az Osztályok egyedhalmazból.

Amennyiben a fenti példában az *Osztályok* egyedhalmazt a *Tanulócsoportok* egyedhalmazra cserélnénk, akkor a nyíl eltűnne, hiszen ugyanaz a tanuló a különböző tantárgyak esetében más-más csoportokba lehet besorolva. Ekkor azonban ha nem akarjuk elveszíteni az információt, hogy az egyed diákok melyik osztályba járnak, akkor is szükség lesz a fenti kapcsolata, azaz a két kapcsolat egy időben is létezik, létezhet.

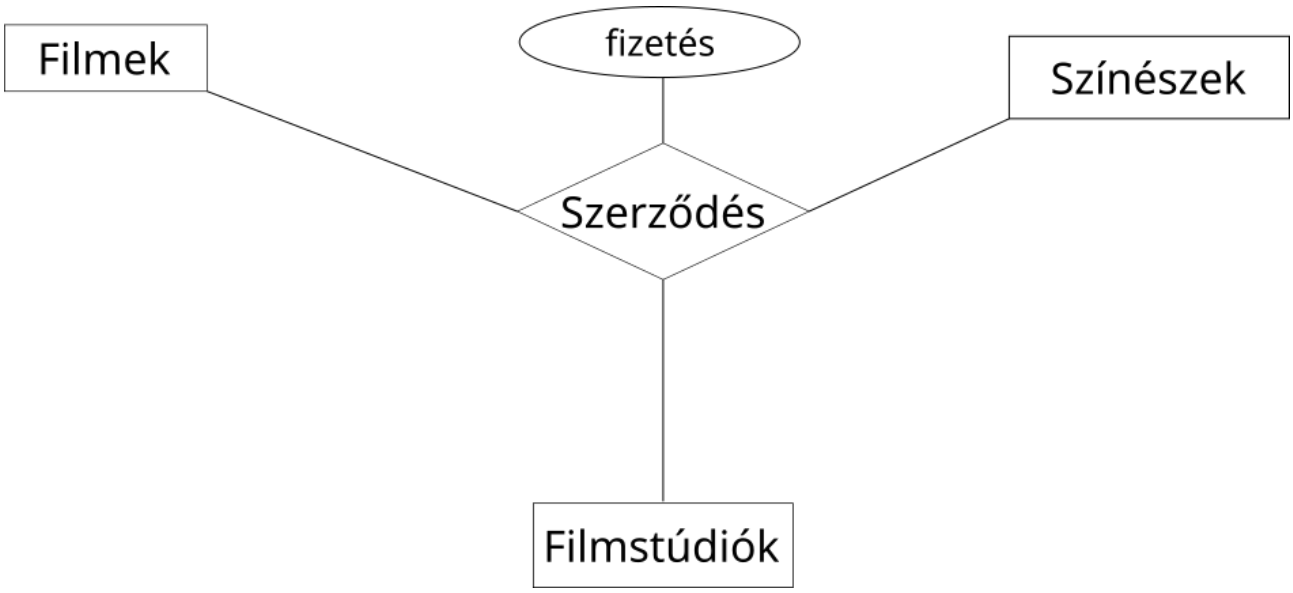
Térjünk vissza a filmes példára egy kicsit! A 35. ábra mutat egy példát arra az esetre, ha egy kapcsolat kettőnél több egyedhalmazt kapcsol össze. Ez az eset, amikor egy színész szerződést köt egy stúdióval annak valamelyik filmjében való szereplésre. Az ábrán csak a *Filmek* egyedhalmaznál szerepelnek attribútumok, de nyilván a többinél is vannak, csak azok nem szerepelnek, hogy ne legyen túl zsúfolt az ábra, illetve, hogy a lényeget lássuk rajta. A *Filmstúdiók* egyedhalmaz felé nyíl is mutathatna, hiszen a szóban forgó kapcsolat esetében minden színész-film páros kiválasztásánál pontosan egy stúdiót kapunk – amit konkrétan már a film is meghatároz a színésztől függetlenül.

Azonban itt felmerül egy kérdés: hogyan tarthatja nyilván az adatbázis az egyes szerződésekre meghatározott fizetések összegeit? A kérdés megfogalmazásában igazából már ott a válasz, de előbb gondoljuk végig a lehetőségeket, hogy mit jelentene a fizetést megadó attribútum elhelyezése az egyes egyedhalmazoknál!

Ha a *Filmek* egyedhalmaz attribútumává tesszük, az azt jelenti, hogy minden egy film valamennyi szerepéért, annak fontosságától függetlenül ugyanannyit fizet a stúdió. Ha a *Stúdiók* egyed-



35. ábra: Attribútum kettőnél több ággal

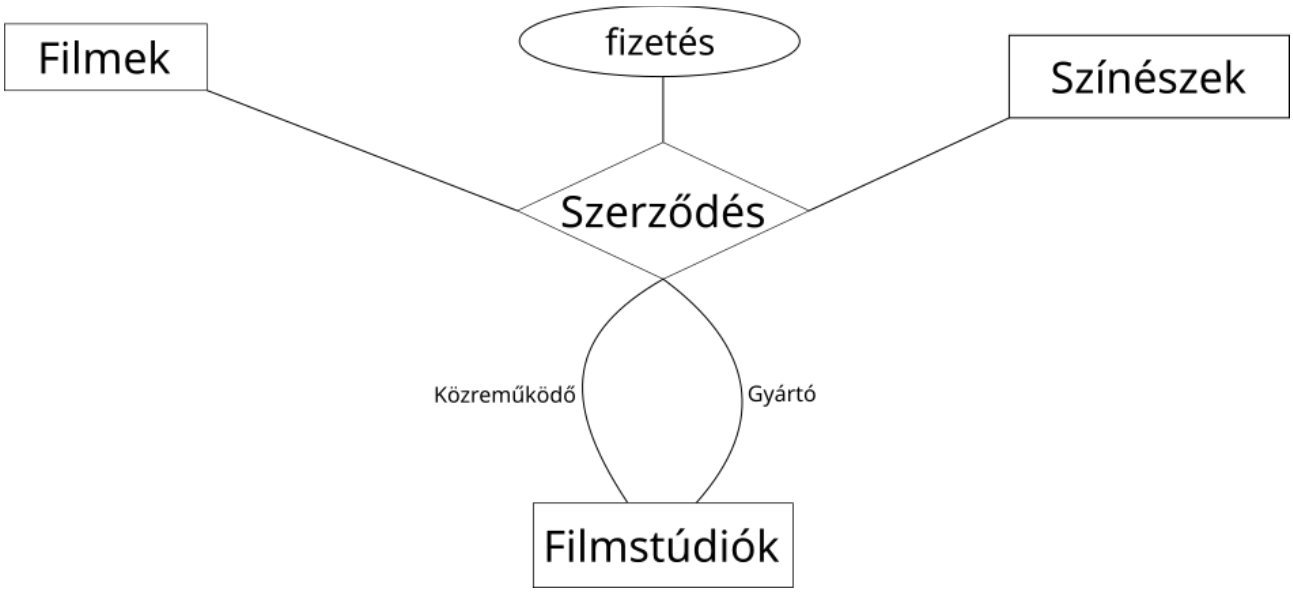


36. ábra: Kapcsolat attribútuma

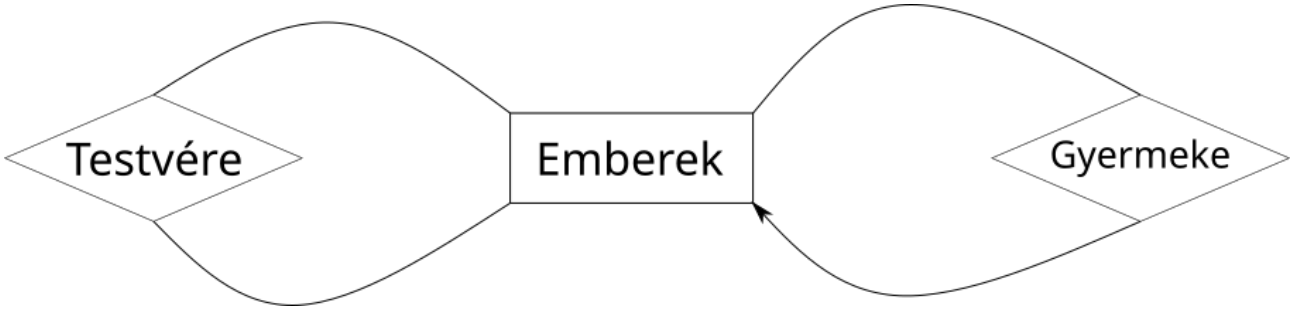
halmaz attribútuma, akkor még rosszabb a helyzet: a stúdió minden filmjében minden szerepért ugyanannyit fizet. A *Színészek* attribútumaként egy adott színész bárhol bármilyen szerepet játszik, ugyanannyit kap érte. A legnagyobb sztárokról még talán el is hinnénk.

A fentiekből következően az egyetlen helyes megoldás, ha a fizetés a kapcsolat attribútumaként szerepel, hiszen akkor minden egyes szerződéshez, amely egy stúdió és egy színész között kötetik egy filmre, megadható az arra a szerződésre érvényes fizetés összege.

Azonban a valós életben egyre sűrűbben fordul elő, hogy egy film elkészítésében egynél több stúdió is részt vesz. Az ilyen eseteket régen koprodukcióként emlegették, manapság viszont ezt már nem is hangsúlyozzák, annyira gyakori például a vizuális effektek más stúdióval megoldása. Ilyen eseteket a 37. ábra formájában lehetne megadni: van egy gyártó stúdió, vagy a hagyományos értelemben vett koprodukció esetén egy fő stúdió – itt lehetne nyíllal jelölni, hogy ebből biztosan egy van –, valamint számos közreműködő stúdió. Azonban kérdéses, hogy pont a Szerződés kapcsolat-



37. ábra: Kapcsolat szerepekkel

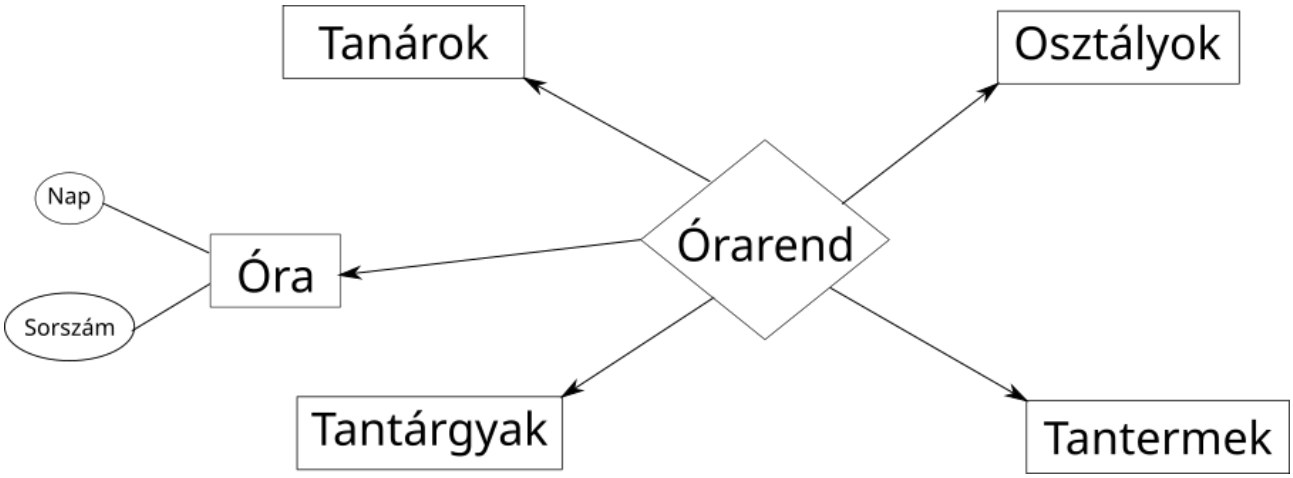


38. ábra: Kapcsolat szerepekkel

nál kell-e egyszerre egynél több stúdiót feltüntetni. Ha ugyanezzel a kapcsolattal akarjuk azt is megadni, hogy az együttműködő stúdiók között milyen szerződések köttetnek, akkor természetesen igen, de az ilyen szerződésekre a fizetés attribútum helyett másfajta attribútumokra lesz szükség.

A 38. ábra két olyan kapcsolatot mutat, amely ugyan két elemű, de mindkét eleme a kapcsolatoknak ugyanaz az egyedhalmaz. A fenti, stúdiók közti kapcsolat is valami hasonlóan értelmezhető. Ilyen eseteket még máshol is találhatunk, amikor ugyanannak az egyedhalmaznak két eleme – a fenti példában két ember – között fennálló kapcsolatot kell feltüntetni. A filmes esetre példaként lehetne még hozni a film és folytatása közötti kapcsolatot is, annak jelzésére, ha egy film valamely más film folytatásaként készült el. A 38. ábrával ellentétben ilyenkor mindig jelölni kell a kapcsolat vonalaival, hogy azon az oldalon elért elem milyen szerepben vesz részt a kapcsolatban – ahogyan az a 37. ábrán is látható.

Végezetül nézzünk egy példát, hogy hogyan lehetne az az órarendet adatbázisban tárolni! A 39. ábra mutat egy példát erre. Mint látható, az adatbázisban a tanárok, az osztályok (vagy csoportok), a tantárgyak és az órák megtartására szolgáló tantermek mind külön egyedhalmazt alkotnak, amelyek mind szükségesek egy órarendi óra meghatározására. Nyilván mindnek vannak saját attribútumai – sőt, a Tantermek esetében még lehetnek speciális esetek a szaktantermek olyan tulajdonságaival, amelyek egy átlagos tanterem esetében nem relevánsak –, amelyek ezen az ábrán nem szerepelnek, mert már nem lenne átlátható. Ezeket az Órarend kapcsolat tudja összekapcsolni, ám ahhoz, hogy ténylegesen egyedi órát adjon meg, kell még az is, hogy mikor kerül rá sor. Ezt az ábra önálló egyedhalmazként jelöli a hét napjának és az óra sorszámának attribútumával, de valójában a gyakorlatban ezt a két attribútumot nagy valószínűséggel az Órarend kapcsolathoz szokták inkább rendelni. Hogy miért, arra a következőkben fogunk kitérni és hogy a két megoldás a végső adatbázisban nem fog eltérést okozni, az a következő lecke végére tud kiderülni.



39. ábra: Az órarend kapcsolata

7.1.2. Megszorítások

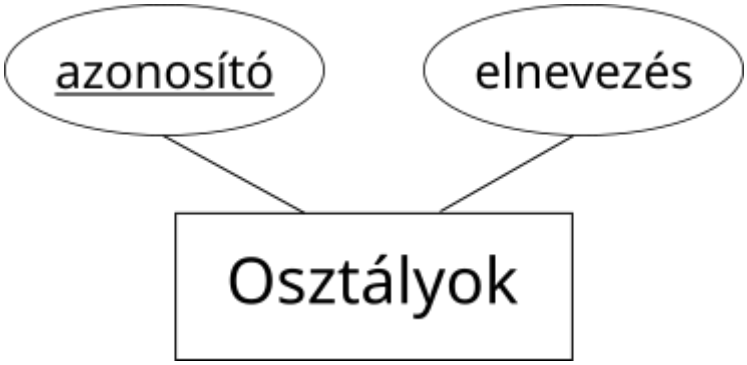
A következőkben néhány olyan fogalmat ismersz meg, amelyeket ismerni kell nem csak az adatbázis megtervezéséhez, de a már létező adatbázis használata során. Ezek közé a fogalmak közé tartozik a **redundancia** és a **kulcs** is.

Először nézzük meg, mit jelent a *redundancia*. A szóra a Google keresőjében rákeresve, ezt a meghatározást kapjuk: „*A redundancia (lat.) nyelvtudományi fogalma; a közlésben az egyértelmű megértéshez elegendő minimumon felüli, ezért fölösleges többlet. Terjengős kifejezések alkalmazása egyszerűbb szavak helyett, pl. javasol - javaslatot tesz.*” Ezt egyébként a Google a Wikipédia szócikkéből emelte ki. Az idegen szavak szótárának interenetes változatában több jelentést is találunk, ezek közül ide a következő illik leginkább: „*ugyanazon elemből az alapvetően szükségesnél többet tartalmazó*”. A redundancia tehát (felesleges) ismétlése valaminek. A „felesleges” szó azért szerepel zárójelben, mert vannak helyzetek, amikor a redundancia nem hogy nem felelsleges, hanem egyenesen kívánatos. Ilyen eset például amikor valamilyen eszközben biztonsági okokból valamit megtöbbszöröznek, tartalékot képezve arra az esetre, ha az eredetileg használt példány meghibásodik – például mert nincs lehetőség a javításra. Tipikus példa erre a világűrbe küldött eszközök esete, ahol ha például az adatrögzítő elromlik, nem lehet odamenni megjavítani, ezért tartalék adatrögzítővel lehet gondoskodni a szonda működéséről. Másik példa az interneten egy szerver, amelynek elérhetősége valamiért kritikus. Ilyenkor több, egymástól független hálózati kapcsolattal lehet ellátni, vagy ha szükséges, magát a szervert is többszörözni lehet. (Valójában a Google eredeti sikerét is pontosan az adta, hogy a keresőszolgáltatása a világ különböző részein elhelyezett, redundánsan működő adatközpontokat használ.)

Az adatbázisok esetében azonban a redundanciát veszélyesnek tekintjük. Ennek nagyon egyszerű oka van: ha egy adatot az adatbázis több helyen is tárol, akkor előfordulhat, hogy annak megváltoztatása nem mindenhol történik meg. Ekkor viszont attól függően, hogy hogyan tesszük fel az adott adatra vonatkozó kérdést, vagy a megváltozott, vagy az eredeti változatban levő adathoz juthat el az adatbázis-kezelő, így ellentmondás keletkezhet az adatbázisban, azaz megszűnik annak *konzisztenciája*.

Nagyon fontos fogalom az adatbázisok esetében a **kulcs**.

A **kulcs** attribútumok olyan halmaza, amely egyértelműen azonosít egy egyedet, azaz nincs két olyan egyede az egyedhalmaznak, amely a kulcsba tartozó attribútumok mindegyikén megegyező értékkel rendelkezik. Másképpen fogalmazva: ha az egyedhalmaz e_1 és e_2 egyede esetén a kulcsba tartozó valamennyi attribútum értéke azonos, akkor az e_1 és az e_2 valójában ugyanaz az egyed.



40. ábra: Az Osztályok egyedhalmaz kulcsa

Az E/K modellben a kulcsot úgy adjuk meg, hogy azon attribútumok nevét, amelyek a kulcs részét képezik, aláhúzzuk.

A korábbi példák közül az Osztályok egyedhalmaz esetén egyszerűen megválasztható a kulcs (40. ábra), hiszen az azonosító attribútum pont ezért szerepel.

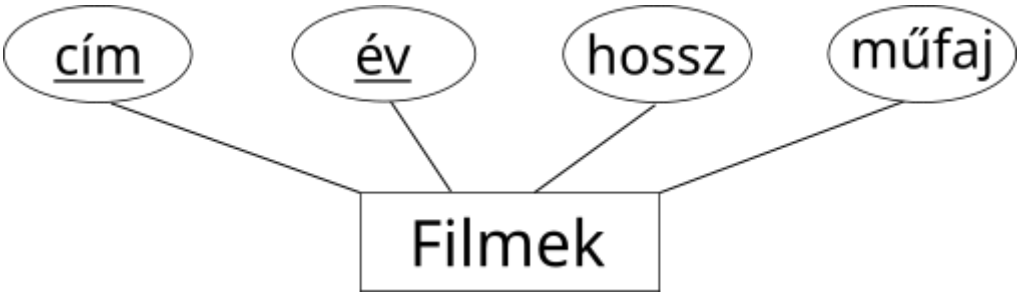
A Filmek esetében már nem ilyen egyszerű megtalálni a megfelelő kulcsot, hiszen az attribútumok egyike sem alkalmas önmagában egy film egyedi beazonosítására. Ám a kulcs nem csak egy attribútumból állhat, így ha nem kell attól tartani, hogy ugyanabban az évben két film is megjelenik ugyanazzal a névvel, akkor a cím és az év attribútumok együtt már alkalmasak lesznek kulcsnak (41. ábra). Itt azonban célszerű megjegyezni, hogy szöveges adatot nem jó ötlet kulcsnak választani, mivel egy esetleges elgépelés vagy egy fölösleges szóköz máris különbözővé teheti az értéket a szövegek esetében.

Mi a helyzet a Tanulók egyedhalmaz esetében? Semmi sem tiltja, hogy egy iskolában egyszerre több azonos nevű tanuló is legyen. Ha mellé vesszük a születés időpontját, akkor már talán egyediek lesznek a tanulók, így úgy tűnik, a születési idő, vezetéknév, keresztnév attribútumhármas már jó lesz kulcsnak – feltéve, hogy eltekintünk az előző bekezdés végi megjegyzéstől a szöveges adat kulcsként való használata kapcsán.

Ám bővítsük ki a példát általában az emberekre! Mi a garancia, hogy egy napon az ország két végében nem született két olyan ember, akinek a szülei ugyanazt a nevet adták? Ezek később azonos iskolába is járhatnak, így a Tanulók kulcsát is tönkreteheti ez a lehetőség!

A probléma megoldására az életben több megoldást is kitaláltak már. Ezek egyike az, hogy a születési helyet is felveszik, amivel ismét nem zárható ki az egyezés, de már elég valószínűtlenné tehető, hogy két ember azonos városban, azonos napon születve azonos nevet kapjon.

A fenti problémára létezik egy nagyon egyszerű megoldás, ami egy kicsit az Osztályok kulcsának általánosítása: lássuk el az egyedeket egyedi számokkal, azaz tegyük őket megszámolhatóvá (a megszámlálthatatlanul sok egyedeket úgysem lehet számítógépen tárolni a digitális adattárolás miatt). Ezt valósította meg az emberek esetében Magyarországon a személyi szám, amelyet valami-



41. ábra: A Filmek egyedhalmaz kulcsa

lyen jogi, politikai, de semmiképpen nem szakmai megfontolásból a rendszerváltás utáni első néhány év során az Alkotmánybíróság gyakorlatilag betiltott. Amellett, hogy az országnak ez a már létező nyilvántartások átalakítása formájában jelentős gazdasági károkat okozott, előállt az a helyzet, hogy az egy egyedi azonosító adat helyett a különböző nyilvántartások három azonosító adattal látták el az embereket: a személyi azonosító jellel (formailag a személyi számmal azonos volt), a TAJ-számmal és az adóazonosító jellel. Ezzel a három azonosító adattal ugyanakkor rendundancia jött létre, aminek veszélyéről fentebb már volt szó. Érdekesség még, hogy a személyi szám egyszerűen az illető születési időpontjából és egy azon napon belüli sorszámból áll egy ellenőrző számmal együtt, tehát a természetes azonosító adatokból képzett számként garantáltan egyedi azonosítója minden magyar állampolgárnak.

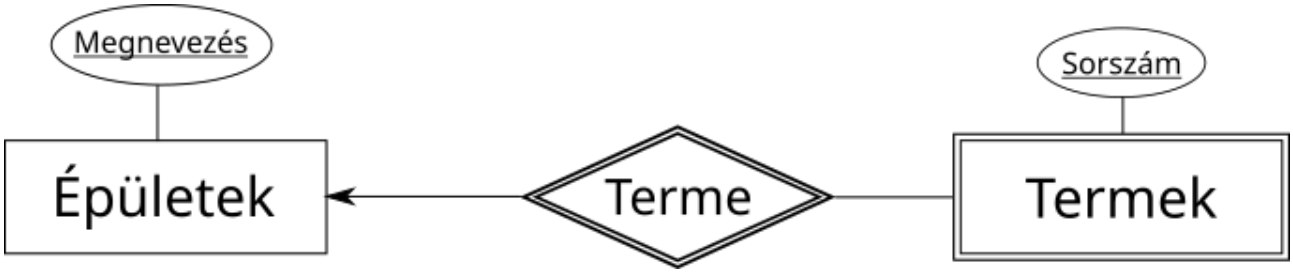
Ugyanilyen azonosító adat autók esetében az alvázszám, amely minden autó élete során annak egyedi azonosítására alkalmas, de mivel nem látható kívülről, ezért a gyakorlatban az autó nyilvántartására a rendszám szolgál, amely viszont bizonyos körülmények között megváltozhat.

7.1.3. Gyenge egyedhalmazok

A kulcs meghatározása egy egyedhalmaz esetében nem mindig egyszerű. Vannak olyan egyedhalmazok, amelynél több olyan attribútumhalmazt is lehet választani, amely értéke képes az egyed egyértelmű azonosítására. Ilyenkor ugyan mindegy melyiket választjuk *elsődleges kulcs*nak, ám felmerül a kérdés: vajon nem függ-e az egyik kulcs értéke a másik kulcstól. Ha igen, akkor egy olyan probléma vetődik fel, amellyel itt ugyan nem fogunk foglalkozni, de ha az ilyen problémát nem oldjuk meg, akkor az redundanciához vezethet az adatbázisban. Jó példa erre a problémára a fentebb bemutatott eset, amikor egy ember azonosítására egyaránt használható a személyi száma, a TAJ-száma és az adóazonosító jele – amelyet gyakran mindhármat kell egy munkáltatónak a munkavállalóiról tárolni a különböző kötelezettségei teljesítéséhez.

A másik eset, amikor az egyedhalmaz saját attribútumai nem alkalmasak az egyedek egyértelmű azonosítására, csak egy másik egyedhalmaz egyedein keresztül. Az ilyen egyedhalmazokat hívjuk **gyenge egyedhalmaz**nak. Példa ilyen esetekre, amikor egy cég vagy iskola több épülettel rendelkezik, és a termeket épületenként külön számozzák. Ebben az esetben a terem azonosításához nem elég a sorszáma, mivel az adott sorszám több épületnél is előfordul. A terem egyértelmű azonosításához meg kell adni azt is, hogy melyik épületben található. Ez az információ azonban nem található meg az egyedhalmaz attribútumai között, hanem egy másik egyedhalmaz valamelyik egyedét jelenti.

A gyenge egyedhalmazt dupla szegélyű téglalappal jelöljük. A gyenge egyedhalmazt egy szintén dupla keretes sok-egy kapcsolat köti össze azzal az egyedhalmazzal, amelynek kulcsa szükséges az egyedek meghatározásához (lásd 42. ábra). Ahogy az a példa ábráján látható, a sok-egy kapcsolat egy-ága a kulcsot kiegészítő egyedhalmaz felé mutat, mivel pontosan egy épületnek kell lennie, amit a kapcsolat az adott teremhez kapcsol. Valójában azt lehet mondani, hogy a terem kulcsa a sa-



42. ábra: Példa gyenge egyedhalmazra

ját *Sorszám* attribútumából és az *Épületek* egyedhalmaz *Megnevezés* kulcsából együtt áll össze, hiszen ez a két adat képes egy terem egyértelmű kiválasztására.

A példa egyébként azért érdekes, mert sok helyen valószínűleg ezt így tudatosan végig nem gondolva is érezték a felmerülő problémát, amikor a termek számozásának rendszerét egy-egy intézményben kigondolták: amikor a sorszám nem folyamatos, hanem az első egy vagy két szám az emelet sorszámából származik, akkor rejtve valójában ugyanez a gyenge egyedhalmaz bújik meg benne: az emelet sorszáma az Emeletek egyedhalmaz kulcsa, majd utána az emeleten belüli sorszám a Termek gyenge egyedhalmaz saját attribútuma. A probléma másik természetes megoldása, ha az egyes épületekben egyedi sorszámok vannak, amely esetben a Termek egyedhalmaz már nem gyenge egyedhalmaz, hiszen minden egyed kulcsa különbözik. Ekkor a kapcsolat sem dupla szegélyű és csak annak megadására szolgál, hogy a terem melyik épületben található. Tehát már eleve a terem sorszámába belerejthető az információ, amely a teremhez a gyenge egyedhalmazt kisegítő más egyedhalmazból származik. A következő leckében, amikor megnézzük, hogyan lehet a gyenge egyedhalmazból relációs modellt készíteni, látni fogjuk, hogy az előzetes megoldással és a gyenge egyedhalmaz megadásával gyakorlatilag ugyanarra az eredményre lehet jutni.

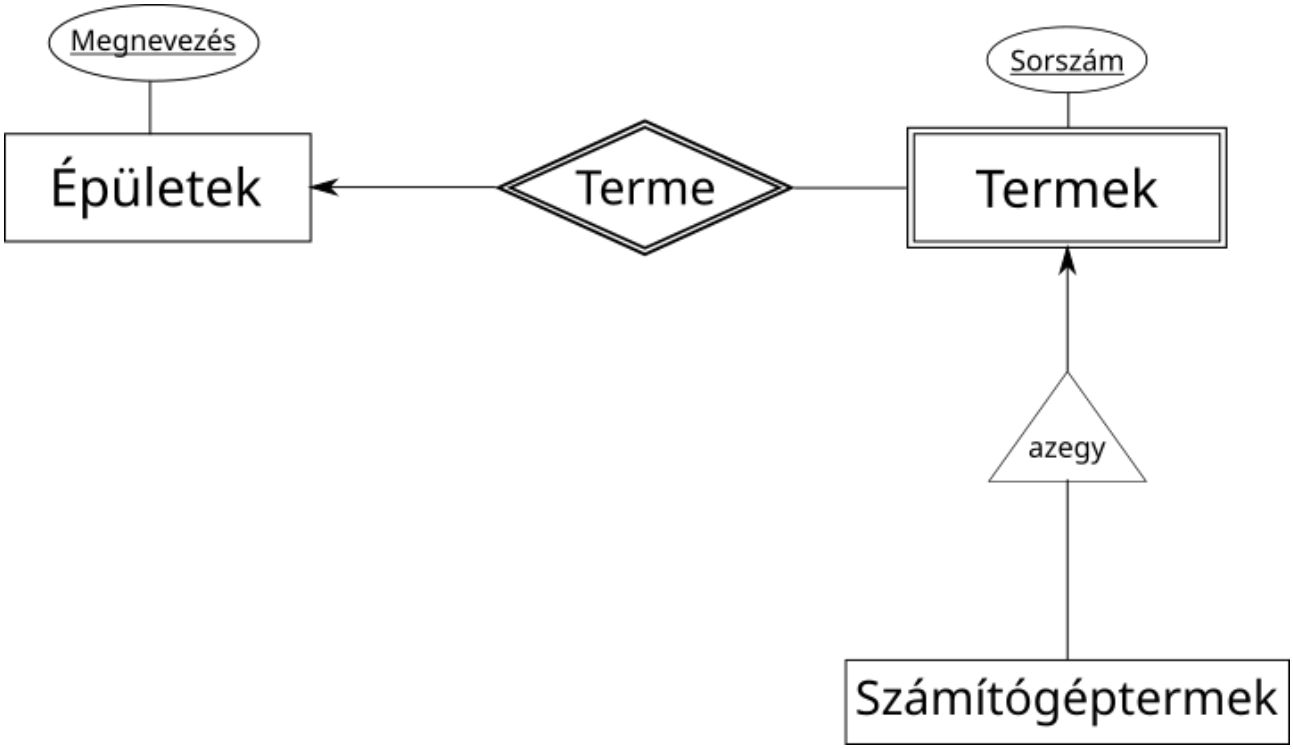
7.1.4. Alosztályok

Ha már a tantermek kerültek elő példaként, akkor vizsgáljuk meg egy kicsit közelebbről a tantermek esetét egy jól felszerelt iskolában! Azt láthatjuk, hogy vannak „átlagos” tantermek, amelyekben van egy tanári asztal székkel, valahány szék és asztal a diákok számára, egy hagyományos és jó esetben egy interaktív tábla a hozzá tartozó kivetítővel. Ez **a** tanterem. Ám vannak szaktantermek is, amelyekben valami más is van a normális tanterem felszerelésén felül. Például egy számítógépteremben – amely egyébként egy tanterem – van valahány számítógép is, jobb helyeken a tanári géphez kapcsolva egy nagyméretű monitor is, amit a diákok is látnak…

Ezeket a többlet információkat a Tantermek egyedhalmazban nem tudjuk elhelyezni, hiszen ott nincsenek ehhez megfelelő attribútumok. Vagy vannak, de akkor a legtöbb terem esetében azok értelmetlenek: üresek maradnak. Ezért felmerül az igény arra, hogy a különböző speciális funkcióval rendelkező tantermek számára külön egyedhalmazt hozzunk létre. Ám ekkor az adatbázisban bizonyos lekérdezések csak akkor találják meg ezeket a termeket, ha a Tantermek egyedhalmazban is szerepelnek az ott megadható attribútumaikkal. Így viszont redundancia lép fel!

Ennek a problémának az objektum-orientált programozásban nagyon egyszerű megoldása van: az egyes típusú objektumokat tartalmazó osztályok egymás leszármazottjaiként is definiálhatók, amely esetben a szülő-osztály minden tulajdonságát öröklik és újakkal egészítik ki. Ráadásul a programban az adott leszármazott osztálybeli objektumot lehet ott is szerepeltetni, ahol a szülő-osztály objektuma szerepelhet – igaz, akkor csak a szülő-osztályban is létező tulajdonságai érhetőek el. Pont, ahogyan a tantermek esetében erre nekünk szükségünk lenne!

Ezt az osztály-alosztály megoldást használva megvalósítható a speciális funkciójú tantermek esete: Azt mondhatjuk, hogy a Számítógéptermek egyedhalmaz eleme **az egy** Tantermek egyedhalmazbeli elem, amelynek a Számítógéptermek egyedhalmaz további tulajdonságokat ad. Amennyiben az egyedet a Tantermek egyedhalmaz felől vizsgáljuk az egyedet, akkor az ott megadott attribútumokat látjuk: férőhely, tábla típusa, kivetítő típusa stb. Amennyiben vizsont a Számítógéptermek egyedhalmaz felől érjük el az egyedet, akkor az ott megadott attribútumokat, kapcsolatokat is elérjük. Így olyankor, amikor – például az órarendnél – általában, mint tanteremként kell megadni az adott szaktermet, akkor a Tantermek egyedhalmaz elemeként létezik, míg ha az a kérdés, hogy egy számítógép pontosan hol található, akkor a Számítógéptermek egyedhalmaz elemeként.



43. ábra: Példa alosztályra: a tantermek és a szaktantermek esete

A 42. ábra kiegészítése a Számítógéptermek egyedhalmazzal látható a 43. ábrán. Mint látható, az ős- és a leszármazott egyedhalmaz között egy speciálisan, a rombusz helyett háromszöggel jelölt sok-egy kapcsolatot szokás megadni, amely háromszögbe magyarul az „azegy”, angolul az „isa” szöveget szokás írni (a 21. terem „azegy” tanterem; room 21 „isa” classroom). A fentebb leírtaknak megfelelően a Számítógéptermek egyedhalmaz elemeit egyúttal a Termek egyedhalmaz elemének is kell tekinteni, így az ott már megadott attribútumok és kapcsolatok megismétlésére a leszármazott egyedhalmaznál nem kell sort keríteni.

Hogyan lehet mindezt a valódi adatbázisokban megvalósítani? Erről szól a következő lecke!

8. A relációs adatmodell

Képzelj el egy Excel táblázatot, amelyben több munkalap is található, minden munkalapon egy-egy táblázattal amelynek speciális alakja van: az első sorban van megnevezve, hogy az adott oszlopban milyen adatok találhatóak, a további sorok pedig azonos tulajdonságú objektumokat (embereket, autókat stb.) tartalmaznak (lásd: 44. ábra).

Név	Születési idő	Születési hely	Anyja neve	Beosztás	Fizetés
Gábor Zsombor	2001-07-01	Nagykanizsa	Alma Rozália	Igazgató	1 000 000 Ft
Kovács Júlia	2000-09-01	Budapest	Juhász Éva	Portás	10 000 Ft

44. ábra: Példa relációra

A táblázatokat nevezzük *relációnak* a továbbiakban, a mukalap nevét tekinthetjük a reláció nevének. A táblázat első sorában levő neveket nevezzük *attribútumnak*, a további sorokat szokás egyszerűen *sor*nak, vagy a relációs adatbázisokban *rekord*nak nevezni. (Magát a relációt a relációs adatbázisban majd *adattáblának*, az attribútumokat *mezőnek* nevezik.)

Mint később látni fogjuk, az elnevezések nem véletlenül hasonlítanak az Egyed-kapcsolat modell neveire: a reláció gyakorlatilag megfelel az egyedhalmaznak, az attribútum az egyedhalmaz attribútumainak, míg a sorok egy-egy adatait tartalmazzák.

Amennyiben nem Excel táblázatban, hanem adatbázis-kezelő programban helyezzük el ezeket az adatokat, akkor már meg is kaptuk az adatbázisunkat. Na de azért ne szaladjunk ennyire előre! Hogyan lesz egy E-K diagramból relációs modell? A kérdés egyedhalmazra vonatkozó része már sejthető, de mi a helyzet a kapcsolatokkal, a gyenge egyedhalmazokkal és esetleg az alosztályokkal...? Ezekre a kérdésekre is hamarosan választ kapsz majd, de előbb még a relációs adatmodell néhány további tulajdonságáról is szót kell ejteni.

8.1. A relációs adatmodell felépítése

8.1.1. A séma

A fenti, 44. ábrán maga a reláció látható annak két sorával, mint *relációelőfordulással*. Maga a relációelőfordulás elnevezés azt akarja jelezni, hogy a reláció egy adott időpillanatbeli állapotát ábrázolja – a példánkban akkor, amikor Gábor Zsombor és Kovács Júlia adatai vannak benne. Ez az idők során rendszeresen változhat, ahogy új munkavállaló adatai kerülnek bele, valameny munkavállaló törlésre kerül, valakinek például a fizetése megváltozik.

Az adatbázis tervezése során ezek nem szükséges adatok: ezek majd az adatbázis használata során lesznek lényegesek. A tervezésnél viszont tudni kellene a reláció (vagy adattábla nevét), amely viszont az ábráról nem derül ki – mert a munkalap neve nincs rajta az ábrán.

Amennyiben a reláció általános felépítését akarjuk megadni, akkor a benne levő sorok helyett a reláció nevét és attribútumait kell megadni. Erre szolgál a *relációséma* vagy röviden *séma*, amely a fenti példa esetén – a Munkavállalók nevet feltételezve – így néz ki:

Munkavállalók(Név, Születési idő, Születési hely, Anyja neve, Beosztás, Fizetés)

A séma a relációelőfordulást nem mutatja, de azt igen, hogy mi a reláció neve és mi az attribútumok neve.

Az előző leckében megismert kulcs fogalma természetesen továbbra is fontos, így annak jelölésére itt is szükség van: ugyanúgy az attribútum nevének aláhúzása jelzi a sémában, hogy az adott attribútum része a kulcsnak. A fenti példán látható a kulcsot alkotó négy attrubútum. Amennyiben

nem ezeket az attribútumokat szeretnénk kulcsnak használni, akkor felvehetünk természetesen egy egyedi azonosító attribútumot, amely átvehetné a kulcs szerepét tőlük, addig azonban ez a négy attribútum együtt adja a kulcsot.

Fontos megjegyezni, hogy a relációt egy halmaznak kell tekinteni: a relációban ugyanaz a sor kétszer nem szerepelhet. A kulcs értékének éppen ezért minden sorra különbözőnek kell lennie, hiszen a kulcs meghatározza az adott egyeded, azaz a jelenlegi terminológia szerint a sort.

8.1.2. Az attribútumok típusa és értéktartománya

A relációs adatmodell esetében az attribútumokra szigorúbb szabályok vonatkoznak, mint az E-K modell esetében, így lehetséges, hogy az E-K modell egy attribútumának a relációban több attribútumot kell megfeleltetni.

A reláció minden attribútumának ugyanis soronként egy értéke lehet, amelynek egyszerű típusúnak kell lennie. Ez azt jelenti, hogy nem lehet olyan attribútumot használni, amely értékek felsorolása lenne, hanem helyette valamilyen átalakítással meg kell oldani, hogy mindin érték különkülön szerepelhessen. Ezt a legegyszerűbb egyébként már az E-K diagramon megoldani az értékekhez külön egyedhalmazt megadni és az attribútum helyett kapcsolattal megadni...

Az attribútumok értékeinek meghatározott típusúnak kell lennie. A lehetséges típusokat pontosan az egyes adatbázis-kezelők meghatározhatják, de alapvetően a következő kategóriákba sorolhatóak: egész szám, lebegőpontos szám, decimális szám, logikai érték, idő, dátum, szöveg. Ezekről a típusokról egy későbbi leckében még részletesebben is lesz szó. A lényeg, hogy a reláció minden egyes attribútumára meg kell határozni, hogy milyen típusú értéket kaphat.

Ha a típus ismert, akkor sem lehet azon belül akármilyen értéke. A felvehető értékekre szintén adható megszorítás, amely megadja azon értékek halmazát, amelyet az attribútum felvehet. Például a Fizetés attribútumra megadható, hogy határozottan pozitív értéket kell felvennie.

8.2. Egyed-kapcsolat diagram átírása relációs modellé

8.2.1. Egyedhalmazok átírása

Az egyedhalmaz és a reláció hasonlóságából logikusnak tűnik az egyedhalmazok relációvá alakítása: legyen a reláció neve az egyedhalmaz neve, a reláció attribútumai egyezzenek meg az egyedhalmaz attribútumaival, a kulcs úgyszintén legyen azonos – és már kész is van.

Valóban majdnem ennyi a teendő az egyedhalmazok esetén. Egy további lépés lehet még szükséges abban az esetben, ha az egyedhalmaznak van olyan attribúruma is, amelynek összetett értéke van, vagy amelynek több értéke is lehet. Első esetben szét kell bontani egyszerű típusú értékeket tartalmazó attribútumok halmazára. Utóbbi esetben jobban át kell gondolni a modellünket, hogyan lehetne a többértékűséget kiküszöbölni. Nagy valószínűséggel egy újabb egyedhalmazra lenne szükség, amely azokat az értékeket tartalmazza, és egy kapcsolatra, amely a megfelelő egyedekhez a megfelelő lehetséges értékeket tartalmazza.

Lássunk néhány példát!

Az előző leckében szereplő E-K diagramokban több egyedhalmaz is szerepelt az attribútumok megadásával együtt. Ezek közül a legelső a *Tanulók* egyedhalmazt mutatta. Ezt relációvá alakítva így néz ki:

Tanulók(Születési idő, Vezetéknév, Keresztnév, Gondviselő neve, Lakcím-város, Lakcím irányítószám, Lakcím-utca, Lakcím-házszám)

Mielőtt továbbmegyünk további példákra, nézzük meg alaposabban a kapott relációt! Az eredeti ábrán nem szerepelt még a kulcs megadása, míg itt a relációséma megadásánál azt feltételetük, hogy az aláhúzott három reláció alkotja együtt a kulcsot (a valóságban esetleg kevés lehet). A másik eltérés az ábrához képest az, hogy a *Lakcím* attribútum, amely egy összetett adatot tartalmazna, szét lett négy komponensre bontva: *város*, *irányítószám*, *utca* és *házszám*.

> Itt érdemes megjegyezni, hogy a város-irányítószám páros gyakran szokott az adatbázis tervezése során fejtörést okozni. Elsőre azt gondolnánk, hogy érdemes felvenni egy külön relációt a két adatnak, hiszen az irányítószám meghatározza a várost. Ez így is van – elméletben. Azonban gyakorlatban az irányítószám azt a postahivatalt határozza meg, amely az adott utca kiszolgálására hivatott. Ez tehát egy város esetében valóban egyedi kulcs lehet, nagyobb városban esetleg több irányítószámt rendelve ugyanannak a városnak különböző részeihez, de a kulcs definíciójába ez még beleférne. Ám vannak olyan falvak, amelyekhez egy közös postahivatal, ezáltal közös irányítószám tartozik, ami máris megbuktatja azt az elképzelést, hogy legyen a település kulcsa az irányítószáma.

Nézzünk további példákat!

Az *Osztályok* egyedhalmazból készült reláció így néz ki:

Osztályok(azonosító, elnevezés)

Egy sora (amely az eredeti egyedhalmaz egy egyedének felel meg) nyilván lehet valami ilyesmi: (2025k, 11K), ahol a 11K az osztály hétköznapokon használt neve, az azonosító pedig azt jelzi, hogy az osztály 2025. szeptemberében jött létre és a K betűjelet viseli.[7]

8.2.2. Kapcsolatok átírása

Egy kapcsolat több egyedhalmazt köt össze, hogy megadja, melyik egyed melyik elemmel áll kapcsolatban, esetleg erre a kapcsolatra további tulajdonságokat megadva a saját attribútumai által. Ennek megfelelője a relációs adatmodellben szintén reláció lesz.

Mégpedig olyan, amelyben az összekötött egyedhalmazok kulcsai és a kapcsolat saját attribútumai szerepelnek. Előbbiek azért, hogy beazonosítsák a kapcsolatban résztvevő egyedeket, így azok együtt fogják a létrejövő reláció kulcsát alkotni.

Például az *Emberek* és az *Autók* egyedhalmazt összekötő *Autója* kapcsolatból az alábbi reláció jönne létre, ha azt feltételezzük, hogy az Emberek kulcsa a *személyi szám*, azt *Autók* kulcsa pedig a *rendszám* attribútum:

Autója(*személyi szám*, *rendszám*)

Mivel a kapcsolatnak saját attribútuma nincs, ezért a reláció minden attribútuma együtt lesz a kulcs.

Itt érdemes megállni egy kicsi, hogy egy újabb fogalmat megismerjünk. Az Autója reláció személyi szám attribútuma amellett, hogy a reláció kulcsának a része, egy másik reláció – történetesen az Emberek reláció – egy sorának (egyedének) beazonosítására is szolgál. Az ilyen (nem feltétlenül a kulcsba tartozó) attribútum-halmazt **külső kulcs**nak nevezzük, amely azt jelzi, hogy az egy másik reláció kulcsával megegyező értékű. Ezt az információt az adatbázisban meg lehet adni, amellyel azt lehet kikényszeríteni, hogy az adatbázis-kezelő ellenőrizze, hogy az adott érték a hivatkozott reláció kulcsában szerepel-e valamelyik sor esetén, azaz a kapcsolat tényleg létezik és nem a levegőbe mutat.

Következő példánk a Szerződés kapcsolat a Filmek, a Színészek és a Stúdiók között:

Szerződések(filmcím, filmév, színészazonosító, stúdióazonosító, fizetés)

Az attribútumok az eredeti relációkban valószínűleg rövidebb néven szerepelnek, itt azonban a nevükbe belekerült egy utalás arra, hogy melyik reláció kulcsához tartoznak. Ha feltételezzük –

7 Azért K szerepel a példában, hogy nehogy valaki magára vegye...

most az egyszerűség kedvéért –, hogy a színészeket és a stúdiókat is egy-egy *azonosító* nevű kulcsattribútummal tervezzük, akkor ez az átnevezés nem is elkerülhető, hiszen nem lehet egy relációnak két azonos nevű attribútuma, hiszen akkor nem lehetne őket egymástól megkülönböztetni.

Mivel a *fizetés* a szerződés attribútuma, így értelemszerűen nem része a reláció kulcsának.

Gondoljuk egy kicsit tovább a példát! Amennyiben minden film esetében pontosan meghatározható, melyik az a pontosan egy stúdió, amely annak a készítését végzi, míg a többi stúdió csak közreműködő fél, akkor a fenti relációban az utóbbiakat kell csak felsorolni, míg a gyártó stúdiót magánál a filmnél is meg lehet adni, hiszen az egy sok-egy kapcsolattal kapcsolódik a filmhez.

Ekkor a *Filmek* reláció kap az egyedhalmazból származók mellé még egy attribútumot (amely nem része a kulcsnak):

Filmek(cím, év, …, gyártóStúdió)

Ez a *gyártóStúdió* attribútum egy külső kulcs a *Stúdiók azonosító* attribútumára, egyben példa arra az esetre, amikor a külső kulcs maga nem része a reláció kulcsának. Amennyiben sok-egy vagy egy-egy kapcsolattal van dolgunk, akkor sokszor nincs értelme a kapcsolatból külön relációt készíteni – kivéve, ha a kapcsolatnak van saját attribútuma –, hanem az egy-oldali egyedhalmaz kulcsát felvenni külső kulcsként a sok-oldali egyedhalmaz attribútumai közé. Ez azért működik, mert a sok egy kapcsolat sok-oldali egyedeinek mindegyikéhez pontosan/legfeljebb egy egyed fog megfelelni, tehát az attribútum nem lesz több értékű ennél a megoldásnál.

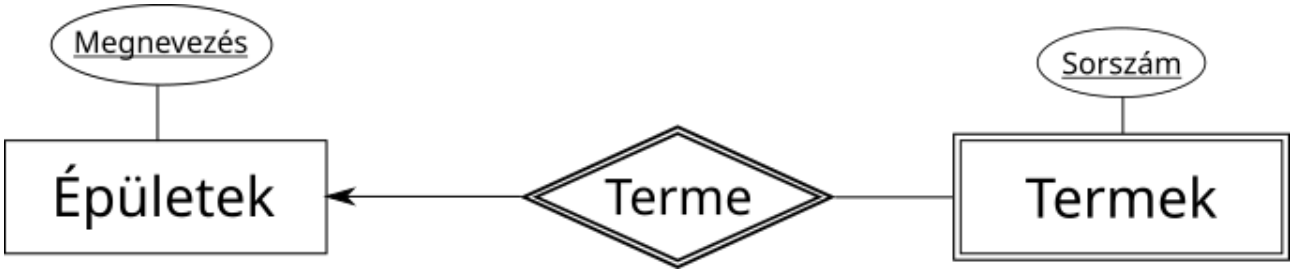
Még egy további példaként nézzük meg, mi a helyzet akkor, ha egy kapcsolat ugyanannak az egyedhalmaznak az egyedeit köti össze különböző szerepekben, mint a *Testvére* vagy a *Gyermeke* kapcsolat esetében! A válasz a fentebb leírtak alapján már sejthető: az egyes szerepeknek megfelelően az egyes kulcsokat át kell nevezni:

Gyermeke(szülőazonosító, gyermekazonosító)

8.2.3. Gyenge egyedhalmazok átírása

Vegyük az előző leckében szerepelt példát a gyenge egyedhalmazokra, amely a 45. ábrán megismételve látható! Amennyiben ezt át akarjuk alakítani relációkká, akkor a következő történik:

1. Létrejön az *Épületek* reláció a megnevezés kulccsal, további (az ábrán nem jelölt) attribútumaival:
   Épületek(megnevezés, ...)
2. A *Termek* reláció kulcsa kapcsán előkerül ugyanaz a dilemma, amely a gyenge egyedhalmaz létrehozásához vezetett: mi a kulcs? Mivel az egyes termek azonosításához szükség van az *Épületek* egyedhalmaz kulcsára is, így a készülő relációba azt is bele kell venni. Így a következő reláció keletkezik (szintén nem jelöli az ábra az összes attribútumot, csak a kulcsot):
   Termek(épületnév, sorszám, ...)
3. A *Terme* kapcsolat esetében a létrejövő relációnak a két összekapcsolt egyedhalmaz kulcsait kell együtt kulcsként használniuk:



45. ábra: Gyenge egyedhalmaz

Terme(épületnév, *épületnév*, *teremsorszám*)

Ebben az esetben a szükséges attribútum-átnevezés még nem történt meg, hogy látható legyen a keletkező probléma: ugyanaz az attribútum kétszer is szerepel, természetesen mindig ugyanazzal az értékkel, hiszen sok-egy kapcsolatról van szó. Másik megoldás: ha figyelembe vesszük, hogy sok-egy kapcsolatról van szó, akkor a kapcsolat egyszerűen beépíthető a *Termek* relációba. Meggondolva alaposabban, ez már meg is történt, hiszen a relációban szerepel az *Épületek* kulcsa, mint a kulcs szükséges része.

Utóbbiból egyértelműen következik, de az első megoldást átgondolva, az is odavezet, hogy a Terme reláció nem hoz semmilyen új információt, így felesleges.

A fenti három pont alapján a gyenge egyedhalmaz relációvá alakítása a következőképpen történik:

A dupla keretes egyedhalmazból készítünk egy relációt, amelynek saját kulcs attribútumai mellé a kulcs részeként fel kell venni az összes olyan attribútumot, amely valamely olyan egyedhalmaz kulcsába tartozik, amelyhez a gyenge egyedhalmaz dupla keretes sok-egy kapcsolaton keresztül kapcsolódik (nem csak egy ilyen lehet). A dupla keretes kapcsolatokból nem készül reláció.

8.2.4. Alosztályok kezelése

Az alosztályok bemutatásánál szerepelt egy utalás arra, hogy ezek gyakorlati megvalósítása kérdéseket vet fel. A gyakorlatban ugyanis nincs egyértelműen jó megoldás erre az esetre.

Az első megközelítés során az alosztályt reprezentáló relációkat külön el kell készíteni és az alosztályban és a főosztályban is szerepeltetni kell ugyanazt az egyedet, hiszen bizonyos tulajdonságai a főosztályban, más tulajdonságai az alosztályban szerepelnek.

A *Termek* egyedhalmaz és a szaktantermek példáját tekintve ez a következőt jelenti:

A Termek(épületnév, sorszám, …) relációban szerepel minden tanterem az alapvető tulajdonságokkal annak attribútumaival megadva. Emellett minden speciális tulajdonságokkal is rendelkező szaktanteremfajta számára van egy-egy külön reláció, így például a számítógéptermek számára a *Számítógéptermek* egyedhalmazból származó:

Számítógéptermek(épületnév, sorszám, gépszám, …)

Vegyük észre, hogy mind a főosztályban, mint az egyes alosztályokban szerepelnek a kulcsot képező attribútumok, hiszen azok azonosítanak egy adott tantermet, ezért mindenhol, ahol tanterem adatai találhatóak, ezekre az attribútumokra szükség van. A másik fontos jellemzője ennek a megoldásnak, hogy ha egy terem minden tulajdonságára szükség van, akkor kettő vagy több relációt is végig kell nézni, hogy azokat össze lehessen gyűjteni.

Mindkét jellemző okozhat problémát a gyakorlati megvalósítás során. Amennyiben összetett kulcsot kell használni, akkor annak megismétlése minden reláció esetén nagyon sok helyet igényelhet az adatbázis tárolása során. Erre persze azt lehet mondani, hogy ha csak néhány szaktanterem van, akkor a gyakorlatban ez nem jelent túl sok helyfoglalási többletet.

A másik jellemző viszont az adatok lekérdezését lassítja, hiszen több relációból kell keresést végezni. Ráadásul ahhoz, hogy ne történjen felesleges keresés az alosztályok relációiban olyan terem esetén, amelyek csak a *Termek* relációban vannak jelen, érdemes lehet olyan attribútumokat felvenni, amelyek jelzik, hogy az adott terem valamilyen szaktanterem, amelynek további tulajdonságai érhetőek el más reláción keresztül (pl. számítógépterem nevű logikai típusú attribúm annak jelzésére, hogy ez a terem egy számítógépterem). Ekkor pedig a felveendő extra attribútumok szintén megnövelik a reláció helyigényét.

Másik megoldás az alosztályok megvalósítására, ha csak az alap egyedhalmazhoz készítünk egy relációt, amelybe viszont összegyűjtjük az összes alosztály összes tulajdonságát. Ezt támogatja

az a lehetőség, hogy a relációs adatbázis-kezelő szoftverek támogatják a NULL érték használatát minden adattípusra, amellyel azt lehet jelezni, hogy az adott attribútum üres, azaz nincs értéke.

Ezzel a módszerrel elérhető a példánk esetén, hogy a terem minden tulajdonsága a *Termek* relációban legyen tárolva. Az ára ennek az, hogy esetleg rengeteg, a legtöbb terem esetén üresen hagyandó attribúm szerepel a relációban.

Hogy a két megoldás közül melyiket érdemes az adott adatbázis esetében használni, azt több minden is befolyásolja, így nem lehet előre megmondani, hogy a kettő közül melyiket célszerűbb használni.

Abban az esetben, ha kevés extra tulajdonsággal kell számolni – akár mert kevés extra tulajdonsága van az alosztályoknak, akár azért, mert kevés egyed fog alosztályba tartozni –, akkor kevesebb helyet igényelhet az egy relációba gyűjtés. Ám ha az adatbázis-kezelő az üres értékek tárolására is az adott típusú értékhez szükséges teljes helyet lefoglalja, akkor az egy relációba gyűjtés jelentősen megnövelheti a felhasznált adatterület méretét ahhoz képest, ha a kevés alosztályba is tartozó egyed extra tulajdonságait külön relációba tesszük. Szintén érdemes mérlegelni az adatok kinyeréséhez szükséges idő növekedésének várható mértékét ha külön relációk sokaságán kell egy-egy egyed adatai összegyűjteni.

Megjegyzendő az is, hogy a később megismerendő adatbázis-kezelő rendszerek közül a PostgreSQL nem tisztán relációs adatbázis-kezelő, hanem objektum-relációs rendszer, ami azt jelenti, hogy az alosztályok kezelésére a külön relációkra bontás módszerét natívan támogatja azáltal, hogy meg lehet adni egy adattábla definiálásánál, hogy az egy másik adattáblából származik. Ekkor csak a plusz attribútumokat kell megadni, mint az alosztálynál. Amennyiben az „ős” adattáblában keresünk, akkor minden sort figyelembe vesz, de csak az ott létező attribútumokat látja, míg az alosztályhoz tartozó tábla nevét megadva a keresésben az abban definiált és az örökölt attribútumok adatai is elérhetőek, de csak az alosztályban is szereplő sorok látszanak.

9. Az SQL nyelv

A legszélesebb körben használt relációs adatbázis-kezelő rendszerek egy **SQL**-nek nevezett nyelv segítségével kérdezik le és módosítják az adatbázist. Az SQL jelentése: *Structured Query Language*, magyarul *Struktúrált lekérdező nyelv*. Ez egy speciális programozási nyelv, amellyel adatbázis műveleteket lehet végezni, mint az adatbázisban tárolt információk lekérdezése (erre utal a nevében a „lekérdező”), adatok módosítása, de magának az adatbázisnak a definiálása is lehetséges benne.

A nyelvnek létezik szabványos változata, de szinte minden létező adatbázis-kezelő rendszer valamiben eltérő változatot használ. Általában kibővítik olyan szolgáltatásokkal, amiket a szabványos SQL nem ismer, de az adott adatbázis-kezelő igen. Vannak olyan esetek is, amikor szűkített az elérhető lehetőségek köre. Például a MySQL első verziói nem ismerték az alkérdéseket, ami pedig – mint látni fogjuk a későbbiekben – elég nagy szerepet játszik abban, hogy az SQL nyelv és így az adatbázis-kezelés olyan jól használható eszköz.

A következő leckékben ezt a nyelvet ismerheted meg – legalábbis alapszinten. A nyelv jóval többet tud az itt bemutatott lehetőségeknél, az interneten jelentős irodalma van a nyelvnek, amelyben inkább eltévedni könnyű, mint eligazodni. Ez utóbbi is abból adódik, hogy minden adatbázis-kezelő rendszer kicsit más dialektusát használja az SQL-nek. Éppen ezért a gyakorlatban mindig meg kell ismerni az éppen használt változat sajátosságait is.

Itt most természetesen elsősorban a szabványos lehetőségekre koncentrálunk – ezen belül is az érettségi követelményekhez igazodva. Azonban ha a bemutatott példákat a gyakorlatban ki akarod próbálni, akkor valamelyik adatbázis-kezelő rendszer használatára lesz szükséged. Ezért az alábbiakban néhány ilyen rendszer bemutatása szerepel, köztük többet is fogunk a gyakorlatban használni.

9.1. SQLite

Ahogy a neve is utal rá, az SQLite egy könnyített SQL-t használó adatbázis-kezelő. Nem is érdemes „rendszer” névvel illetni, ugyanis a valódi adatbázis-kezelő rendszerektől eltérően csupán egy állomány formájában létezik az adatbázisa. Ez viszont a könnyű kezelhetőségével együtt nagyon széles felhasználási kört hozott létre a számára. Sok olyan program létezik, amely által használt dokumentumokban valójában egy SQLite adatbázis található. Sokszor láthatatlanul. Egy példa lehet erre a Macintosh-on talán legprofibb projekt manager program, a Merlin, amely .merlin nevű állományokban tárolja a projekteket (mac-es programoknál gyakori, hogy az állomány kiterjesztése a program nevét viseli). Ezek az állományok azonban valójában SQLite adatbázisok.

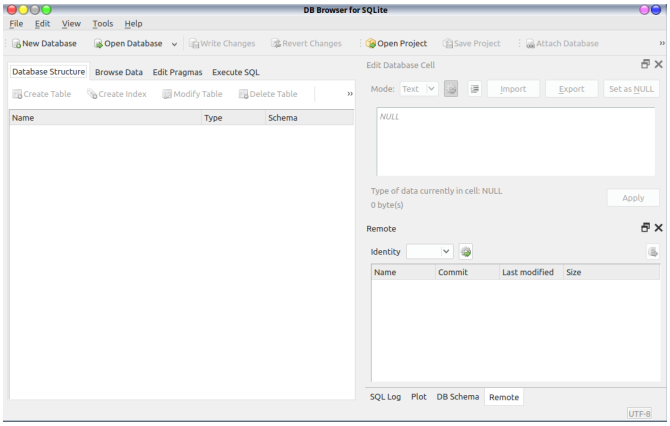
46. ábra: Innen tölthető le az SQLiteBrowser

Ilyen adatbázisokkal fogsz bőven találkozni, mivel egy nagyon egyszerű, de nagyon jól használható kliens-program létezik hozzá: az SQLiteBrowser, amely amellett, hogy mind Windows-ra, mind Linuxra, mind pedig Macintosh-ra elérhető, nyílt forráskódú lévén ingyen elérhető és könnyedén telepíthető. A példák és a feladatok a könnyebbség kedvéért ehhez a programhoz igazodnak, így az alábbiakban röviden szerepel a program használatának bemutatása. A példákban szereplő adatbázisok mindegyike elérhető lesz a `/public/adatbazis` könyvtáron belül egy-egy állomány formájában.

A program letölthető a 46. ábrán látható QR-kódról (a PDF-ben a QR-kódra kattintva is elérhetőnek kell lennie a linknek). A Windows telepítők mellett a Portable változat is hasznos lehet: azt elég egy pendrive-on elhelyezni és bármely Windows gépre csatlakoztatva az eszközt, arról elindítható a program.

A program egyébként egy nagyon egyszerű eszköz. Közvetlenül megnyitható benne bármely olyan állomány, amely egy SQLite adatbázist tartalmaz, SQL utasítások adhatóak ki azon, amelyek utána ki is menthetőek külső szöveges állományba.

A 47. ábra mutatja a programot közvetlenül a megnyitása után. Sajnos nincs egyelőre a magyar nyelv kiválasztására lehetőség – talán ezért nem ezt a programot választották az emelt szintű érettségihez.



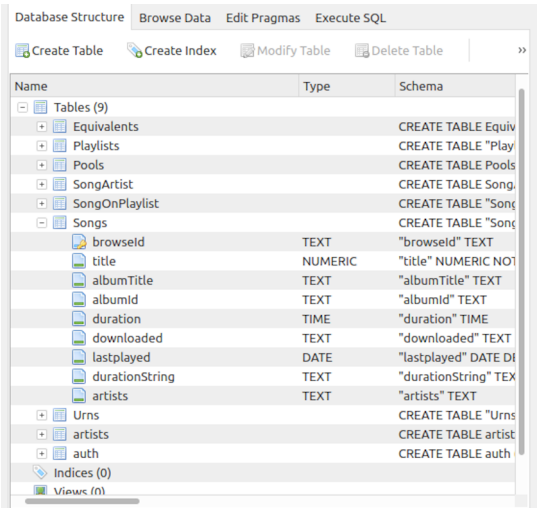
47. ábra: Az SQL Browser megnyitás után

A menü alatti sorban található gombok az adatbázis megnyitására („*Open Database*”), új adatbázis létrehozására („*New Database*”) alkalmasak. A következő két gomb mutatja, hogy ez a program az adatbázis-kezelőktől eltérően viselkedik, feltehetően azért, mert állományban található az adatbázis, amivel dolgozik és a valóságban a számítógép memóriájában tárolja az adatbázist, amin végzett változásokat kiírni a „*Write Changes*” gombbal lehet. Ennek a megoldásnak az az előnye viszont, hogy a „*Revert Changes*” gomb lehetővé teszi az elvégzett módosítások eldobását és az állományból a legutolsó mentett állapot visszatöltését. Ezt a lehetőséget érdemes is kihasználni a feladatok megoldása során: amennyiben egy feladatot sikerült megoldani, ha az módosítást végzett az adatbázison, akkor azt el lehet menteni, míg ha az adatbázist módosító műveletet rosszul sikerült megoldani, akkor az azt megelőző állapot visszatölthető az utolsó mentésből.

A következő két gombot („*Open Project*” és „*Save Project*”) nem fogjuk a feladataink során használni, ahogyan az „*Attach Database*” gombot sem. Utóbbival lehetővé teszi a program, hogy egyszerre több adatbázison is lehessen dolgozni, amelyek ekkor egymással interakcióba léphetnek – a projektekben azeket a helyzeteket lehet későbbi folytatáshoz elmenteni.

A gombsor alatt a baloldalon egy négy lapot tartalmazó rész látható, leginkább ezt fogjuk használni. A négy lap a fülein keresztül érhető el:

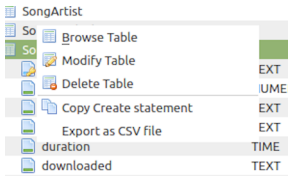
- „*Database Structure*”: Itt fogjuk látni többek között az adatbázisban található adattábláinkat is. A 48. ábra egy példát mutat egy privát projektbeli adatbázisból arra, hogyan néz ki ez, amennyiben van megnyitva adatbázis, amelyben adattáblák is vannak. A példán a „Tables” résznél levő szám mutatja, hogy a jelen esetben 9 adattábla van az adatbázisban. Ezek közül a példa kedvéért a Songs nevű adattábla tartalma van kinyitva, amelynek így a mezőit (attribútumait) is lehet látni azok típusával.

  48. ábra: A Database Structure lap tartalma

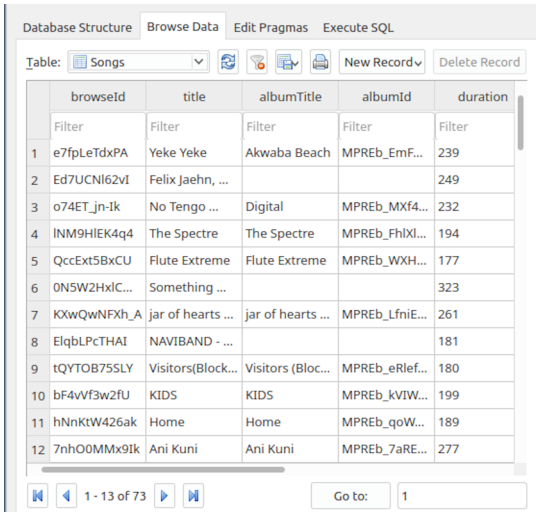
  A lapon található gombok közül az elsővel új táblát lehet létrehozni („*Create Table*”), a másodikkal indexet lehet létrehozni („*Create Index*”, később fogunk egy picikét majd foglalkozni az indexekkel), a harmadik a kiválasztott tábla szerkezetének módosítására alkalmas („*Modify Table*”), míg az utolsó gomb („*Delete Table*”) a kiválasztott tábla törlésére alkalmas.

A gombok helyett valamely tábla nevén jobb egér gombbal kattintva, az 49. ábrán látható helyi menüt kapjuk, amelyben a tábla módosítása a második, a tábla törlése a harmadik menüpontként jelenik meg. Az első menüpont („*Browse Table*”) a következő lapra vezet úgy, hogy az adott tábla tartalma legyen ott látható. Az elválasztó vonal alatti két menüpont akkor lesz hasznos, ha a tábla szerkezetét az adatbázison kívülre kell elmenteni. A „*Copy Create statement*” menüpont hatására a vágólapra másolódik az az SQL utasítás, amely a tábla létrehozását eremdényezi. Az „*Export as CSV file*” segítségével pedig a tábla tartalmát lehet egy CSV formátumú szöveges állományba kiexportálni, amit azután egy másik adattáblába lehet beexportálni, vagy akár táblázatkezelőben is megnyitható.



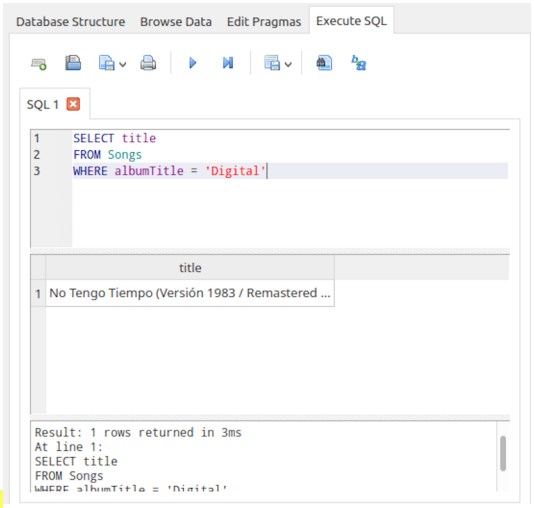
49. ábra: Adattábla helyi menüje

- „*Browse Data*”: Ezen a lapon az egyes adattáblák vagy itt nem tárgyalt nézettáblák tartalmát lehet megtekinteni (50. ábra). Az adattáblák tartalma másodítható is. A felső részen a „*Table*” legördülő listájában kiválasztott adattábla tartalma jelenik meg az első nagyobb területen. Ahogyan már a relációknál is szerepelt: a táblázat minden egyes sora egy-egy egyed, mostantól *rekord* adatait mutatja. Az oszlopok a megfelelő attribútumokat, vagy ahogy mostantól nevezni fogjuk őket: *mezők* adatait tartalmazzák, hogy melyiket, azt a táblázat fejléce jelzi. Alul a nyilakkal az egyes rekordok között lehet lapozni, vagy a „*Go To:*” gombbal a megadott sorra ugrani egyből.
Amennyiben a táblázat valamelyik cellájára kattintunk duplán, akkor az ablak jobboldali részén módosítani lehet az adott mező tartalmát. A sor elején levő sorszámra kattintva az adott rekord kiválasztható, ami után akár a DEL gomb is elegendő a sorhoz tartozó rekord törléséhez. Vigyázat: a törlés nem visszavonható – legfeljebb a fentebb említett „*Revert Changes*” gombbal minden nem mentett változtatás visszavonása árán!



50. ábra: A Browse Data lap

- „*Edit Pragmas*”: Ezzel a lappal nem fogunk foglalkozni. Olyan dolgokat rejt, amelyhez az adatbázis-kezelés sokkal mélyebb szintjére kellene ásnunk, mint ahova a középiskolai tanulmányok során érdemes.
- „*Execute SQL*”: Ezen a lapon lehet kiadni az SQL utasításokat (51. ábra). Az ábrán az „*SQL 1*” nevű fülön belül egy három sorba tördelt SQL utasítás látható. A felső gombsorban ehhez szükséges műveletek érhetőek el. Sorban: új SQL-lap létrehozása; SQL utasítást tartalmazó szöveges állomány megnyitása a tartalmának végrehajtása céljából; az aktuális SQL-lap tartalmának mentése szöveges állományba; az SQL utasítás kinyomtatása; az utasítás végrehajtása; az aktuális sor végrehajtása; az eredmény-nézet (lásd lentebb) mentése; keresés- illetve keresés és csere az SQL utasításban.



51. ábra: Az Execute SQL lap egy SELECT utasítással

A gombsor alatti lapokon tehát SQL utasítások adhatóak meg. Ezalatt az SQL utasítás végrehajtásának eredménye látható egy táblázat formájában – ha van az utasításnak visszaadott eredménye – ezalatt pedig az utasítás végrehajtásának eredménye: hány sort adott eredményül, mennyi ideig tartott az utasítást végrehajtani, stb. Ez a téglalap piros színre vált, ha az utasítás végrehajtása során hiba történt.

A továbbiakban az SQL megismerése során ezt a programot sokat fogjuk ugyan használni, de azt tudni kell, hogy az SQLite nem mindent tud, amit az SQL szabvány alapján tudnia kellene. A legfontosabb hiányossága – amely a használatát viszont megkönnyíti – a dátum és az idő adattípus, amelyek helyett az egész típust lehet használni. Azokban az esetekben, amikor a példa SQL a szabvány és az SQLite esetében valamely hiányosság miatt eltér, mindkét verzió szerepelni fog annak megadásával, hogy melyik melyik esetre vonatkozik.

9.2. PostgreSQL

A nyílt forráskódú szoftverek között elérhető adatbázis-kezelők közül alighanem a legfeljettebb szolgáltatásokat tudja nyújtani a PostgreSQL. Eredetije egy zárt kódú program volt, amelyet már régen nem fejlesztenek, helyette jött létre a PostgreSQL – valószínűleg ennek is köszönhető a sok szolgáltatás, amelyet képes nyújtani a felhasználók felé.

Nem teljesen relációs adatbázis-kezelő, hanem úgynevezett objektum-relációs adatbázis-kezelő. Ez abban nyilvánul meg, hogy képes egy relációnak különböző leszármazottait kezelni úgy, ahogyan azt egy objektum-orientált programnál szokás. Ez magyarra fordítva azt jelenti, hogy az egyed-kapcsolat modellnél megismert alosztályokat természetes módon lehet kezelni: A fő egyedhalmaznak megfelelő reláció adattáblájából kell származtatni az alosztályok relációit, így a fő adattáblát elérve a fő egyedhalmaz tulajdonságai érhetőek el, míg egy alosztály adattábláján keresztül az abba az alosztályba tartozó egyedek mind a főosztálybeli, mind az alosztálybeli tulajdonságai.

Előnye, hogy egy adatbázis valójában al-adatbázisokra osztható, az úgynevezett sémák használatával, így ha egy felhasználó egy adatbázis fölött rendelkezik, gyakorlatilag azon belül is képes egymástól független adatbázisokat használni.

A PostgreSQL-hez webes felületen keresztül is hozzá lehet férni a *phppgadmin* nevű program segítségével.

9.3. MySQL, MariaDB

Másik nagyon elterjedt nyílt forráskódú adatbázis-kezelő rendszer a MySQL, amely bár eleinte jóval fejletlenebb volt a PostgreSQL-nél, mégis sokkal népszerűbbé vált a webes alkalmazások háttéradatbázisaként. Mára nem nevezhető fejletlenebbnek a fentebb bemutatott társánál – immár tud mindent, amit az SQL szabvány előír. Azonban hasonló történt vele, mint az OpenOffice.org nevű irodai alkalmazással: Megszerezte a tulajdonosi jogait az Oracle, amelynek eredményeként az eredeti fejlesztők egy része úgy döntött, hogy az eredeti forráskódot felhasználva (hiszen az nyílt, ezért szabadon használható) egy új projektet indítanak, amely ma még gyakorlatilag teljesen egyenértékű az eredetivel, később nyilván egyre több eltérésre lehet számítani. Ahogy az OpenOffice.org a Sunhoz, majd később annak felvásárlásával az Oracle-hoz került és létrejött a LibreOffice, úgy a MySQL üzleti érdekektől mentes utódaként létrejött a MariaDB, amely tehát felhasználói szinten mindenképpen azonos értékűnek tekinthető az eredetijével.

Mindkettőre jellemző, hogy egy sokfelhasználós környezetbe nagyon bonyolult adminisztrációs beállításokat igénylő telepítése van a rendszernek, ennek ellenére meglepő, hogy elterjedtebb, mint a PostgreSQL.

Talán az is közrejátszik az elterjedségéhez, hogy létezik egy szoftvercsomag, a XAMPP, amely otthoni, egyfelhasználós környezetbe kínálja a webszerver kipróbálásának lehetőségét, olyan egyszerű telepítési lehetőséget kínálva, amelyen keresztül egy asztali számítógépet (vagy akár egy laptopot) alkalmassá lehet tenni egy teljes webes környezet működtetésére. Ezt arra lehet használni, hogy úgy lehessen egy weboldalt kialakítani, hogy ahhoz nem kell azt az interneten publikálni. Tartalmazza magát a webszervert, PHP értelmezőt és adatbázis-szervert. Az egésznek a tartalma pedig helyi kiszolgálással, a gépen elindított böngészőből elérhető és ugyanúgy viselkedik, mintha a neten lenne. A XAMPP adatbázisnak eredetileg a MySQL-t, jelenleg egy ideje pedig a MariaDB-t használja. A mozaikszó mögött egyébként az Apache + MySQL/MariaDB + PHP + Perl szavak húzódnak meg, jelölve a benne található komponenseket.

A program a https://www.apachefriends.org/download.html címről letölthető, telepítését a QR-kódon keresztül elérhető videó mutatja be, mint abban látható, nagyon egyszerű beüzemelni.

A programcsomag része a *phpmyadmin* nevű webes felület, amelyen keresztül a MariaDB adatbázis-kezelőt el lehet érni.

Ezt a programot azért érdemes megismerni, mert a jelenlegi érettségi követelményrendszer szerint az emelt szintű digitális kultúra érettségi adatbázis-kezelési feladatát a XAMPP környezetben kell megoldani. Órai használatát akadályozza, hogy a XAMPP adatbázisa egyfelhasználós környezet, amelyben dolgozva nem lehet semmilyen módon elkülöníteni egymástól a különböző diákok által létrehozott adatbázisokat. Ezzel szemben a PostgreSQL hasonló webes felületén, a *phppgadmin*on keresztül ez működne, hiszen minden felhasználónak külön adatbázist adva, abban a sémákat kihasználva al-adatbázisok léztehozásához, egymást nem zavarva tudnának ugyanazon a szerveren dolgozni többen, ám a XAMPP-hoz hasonló szoftvercsomag nem létezik a PostgreSQL felhasználásával.

52. ábra: A XAMPP telepítését bemutató Youtube videó

9.4. Oracle

A világ jelenleg legnagyobb adatbázis-kezelő szoftverét az Oracle hozta létre. A cégnek eredetileg ez a területe, de mostanra már sok egyéb informatikai területbe is bevásárolta magát. Fentebb szó volt róla, hogy egy ideje a nyílt forráskódú MySQL is az Oracle-hez tartozik, de ezen felül a Java nyelvet fejlesztő Sunt is felvásárolta, így az Oracle-hez került a Java, az OpenOffice.org (függetlenül fejlesztett változata a LibreOffice lett), a VirtualBox nevű virtualizációs szoftver is.

Mindezzel együtt a cég még mindig a leginkább az adatbázis-kezelésben szerzett nagy tapasztalatáról ismert. A legnagyobb adatbázisok az Oracle rendszerein léteznek. A cégnek nem csak általában van tapasztalata az adatbázisok terén, de különösen a banki adatbázisokban, ahol nagyon fontos a megbízhatóság és a napi huszonnégy órás rendelkezésre állás.

Mindebből persze az is következik, hogy az adatbázis szolgáltatásai a cégnek rendkívül sokba kerülnek, így nem véletlen, hogy létrejöttek olyan ingyen elérhető – de jóval alacsonyabb biztonsági követelményeknek megfelelő – rendszerek, mint a PostgreSQL vagy a MySQL, majd később ebből a MariaDB.

9.5. MS Access

A Microsoft amikor az Office nevű irodai programcsomagját létrehozta, valamiért egy „házi” adatbázis-klienst is beletett Access néven. Lehet azon vitatkozni, hogy az Access valójában tekinthető-e

tényleges adatbázis-kezelőnek vagy csak annak kis utánzatának. Lehet azon is vitatkozni, hogy vajon egy irodai programcsomagban szükség van-e adatbázis-kezelőre.

Egy adatbázis általában a háttérben húzódik meg. Nem közvetlenül az adatbázist használjuk, hanem egy olyan programot, amely által kezelt adatok adatbázisban találhatóak. Így elég kicsi az esélye annak, hogy egy egyszerű, irodai munkát végző személynek szüksége van arra, hogy közvetlenül egy adatbázison dolgozzon. A legtöbb esetben a szükséges adatnyilvántartásokat általában egy táblázatban helyezik csak el, vagy ha már olyan méretű adathalmazra van szükség, amely inkább adatbázist igényel, akkor az átlag irodai dolgozó informatikai tudása eltörpül ahhoz képest, ami annak a közvetlen kezeléséhez szükséges és ezért inkább készítenek neki egy programot, amely a megfelelő felhasználói felületen keresztül azokhoz a műveletekhez ad ellenőrzött hozzáférést, amelyre szüksége van. A program a háttérben pedig nyugodtan hozzáfér az adatbázishoz – a megfelelő SQL utasítások kiadásán keresztül. Az ilyen programokat szokás **információs rendszer**nek nevezni.

Lássunk egy konkrét példát: Adott egy bank, amely az ügyfelei számláit egy adatbázisban tárolja. A fent említett Oracle kínált talán először olyan adatbázist, amely a megfelelő jogosultságokkal, biztonsági megoldásokkal lehetővé tette, hogy az ügyfelek adatait – beleértve a számlákon levő összegeket is – csak az arra felhatalmazott személyek és azok is csak megfelelően dokumentált formában tudják módosítani.

Az átlagos banki alkalmazott, aki az ügyfelekkel kapcsolatot tart, nem jogosult arra, hogy közvetlenül hozzáférjen az adatbázishoz. De van egy szoftver, amelybe belépve, bizonyos űrlapokon keresztül hozzáfér a rendszerhez. Ez a megoldás nem csak azt biztosítja, hogy az alkalmazottnak ne kelljen az adatbázis-kezelési ismereteket is elsajátítania a remélhetőleg meglévő pénzügyi ismeretek mellett, hanem azt is, hogy minden általa végzett művelet jól dokumentált módon történjen: az ügyfél a megfelelő dokumentum aláírásával igazolja az adott művelethez adott hozzájárulását.

Amennyiben az alkalmazott közvetlenül az adatbázisba nyúlhatna bele, akkor nem lehetne garantálni, hogy csak az ügyfél által aláírt megbízási szerződéssel igazolt adatmódosítások történtek a rendszerben. Ezzel ellentétben az alkalmazott szabadon átírhatná például a számlán lévő pénzösszegeket.

Hasonló megfontolásokból látható, hogy igazából egy átlagos irodai környezetben semmi szükség közvetlen adatbázishoz való hozzáférésre, vagyis az Accesshez hasonló programokra sem.

Valószínűleg ez a megfontolás is közrejátszhat abban, hogy a MS Office legnagyobb nyílt forráskódú konkurrenciájának számító LibreOffice esetén ugyan van az Accessnek megfelelője LibreOffice Base néven, de annak fejlesztése láthatóan eléggé el van hanyagolva a többi komponenshez képest.

Kicsit talán rosszmájú, de úgy tűnik nem áll távol a valóságtól az az állítás, hogy a MS Accesst egyedül a magyarországi informatika érettségin használják az adatbázis-kezelési feladatok megoldására. Az új nevet kapott tantárgy, a Digitális kultúra érettségi követelményeiben már emelt szinten nem használható, mert emelt szinten a feladat mindenképpen az SQL nyelv használatát követeli meg, de sajnos a középszintű adatbázis-kezelési feladatok még mindig az Access használatára vannak tervezve. Ez azért is probléma, mert egészen másképpen közelíti meg a témát az Access grafikus felületet használó adatbázis-kezelése, mint amit az SQL nyelv használata igényel, így gyakorlatilag két nagyrészt különböző megoldási módot igényel a középszintű és az emelt szintű adatbázis-kezelés. Ez viszont a felkészítésben extra időt igényelne. A jó hír az, hogy az Access is lehetővé teszi az SQL utasítások használatát – bár néha elég sajátos dialektusát beszéli a nyelvnek –, így aki az SQL nyelvet megismerte, az nagyrészt az Accessben is elboldogul.

10. Az SQL adatdefiníciós utasításai

Az SQL nyelv utasításait több kategóriába lehet sorolni:

- Adatdefiníciós utasítások: ezekkel az utasításokkal lehet magát az adatbázis, annak adattábláit és egyéb olyan objektumokat létrehozni, amelyek az adatbázis szerkezetét alkotják. Ezen utasítások egy részéről lesz szó ebben a leckében.
- Lekérdezések: Az SQL nyelv legtöbbet használt utasítása, a SELECT utasítás az, amellyel az adatbázisban tárolt adatokból a kívánt információhoz hozzá lehet jutni. Ezzel az utasítással fogunk majd a legtöbbet foglalkozni, így több lecke is fog a következőkben foglalkozni vele.
- Adatmanipulációs utasítások: Új adat felvitele, meglévő adat módosítása, adat törlése végezhető ezekkel az utasításokkal.

Most tehát ezek közül az első kategóriával fogunk foglalkozni. Éppen csak a felszínt fogjuk megkapirgálni, hiszen a legtöbb feladat esetében az adatbázis már létezik, így nem lesz túl sok szükség az itt szereplő műveletek alkalmazására. Azonban bármikor előfordulhat olyan helyzet, hogy szükség van egy új adatbázis létrehozására, így nem árt legalább annyira ismerni ezeket az utasításokat, hogy a megfelelő dokumentációban a nevük alapján rájuk lehessen keresni. Ezen felül ebben a leckében fognak az adatok lehetséges típusai is szerepelni az adattáblát létrehozó utasítás kapcsán. Ez viszont fontos lehet már létező adatbázis használata esetén is.

Itt érdemes még a nyelv utasításainak felépítését is bemutatni:

Minden utasítástípus egy szóval kezdődik, amely utal arra, hogy milyen típusú műveletet végez az utasítás. Például a valamit létrehozó utasítások a CREATE szóval kezdődnek, a valamit módosító utasítások az ALTER szóval kezdődnek, a valamit törlő utasítások a DROP szóval. Ezután következik annak megnevezése, milyen objektumra vonatkozik a művelet: például DATABASE, TABLE, VIEW stb. Az SQL utasításai szabadon tördelhetőek több sorba, az utasítás végét mindig egy pontosvessző (;) jelzi. Mint a példákból majd látható lesz, a nyelvet úgy tervezték meg, hogy az utasítások gyakorlatilag angol nyelvű mondatokként olvashatóak legyenek.

De először lássuk, hogyan is lehet egy adatbázist egyáltalán létrehozni.

10.1. Adatbázis létrehozása

Az adatbázis létrehozásához az SQL nyelv a CREATE DATABASE utasítást biztosítja. Hogy ez pontosan hogyan használható, abban az egyes adatbázis-kezelők esetén lehet eltérés. Az ACCESS használata esetén például ezt az utasítást soha nem fogja senki kiadni, mivel az adatbázis létrejön az ACCESS elindításakor automatikusan egy állomány formájában.

Az utasítás szabványos alakja:

```
CREATE DATABASE <adatbázisnév>;
```

Az adatbázis nevét még követheti néhány további szó, amely meghatározza például hogy melyik felhasználó tulajdonába kerüljön az újonnan létrejövő adatbázis – amennyiben az utasítást kiadó felhasználónak jogában áll felhasználókat létrehozni, nekik adatbázist létrehozni. Ekkor például így nézhet ki az utasítás:

```
CREATE DATABASE könyvtár OWNER könyvtáros;
```

Ez az utasítás a ’könyvtár’ nevű adatbázist úgy hozza létre, hogy annak tulajdonosa a ’könyvtáros’ nevű felhasználó legyen.

Mint a példákból látható, a különböző nevekben akár ékezetes betűt is lehet használni. Feltéve, hogy az adatbázis-kezelő az ehhez megfelelő karakterkódolást használja. Hogy egy adatbázis melyik karakterkódolást használja, a nevek rendezésére milyen módszert használjon, ezek szintén

olyan információk, amelyek a CREATE DATABASE utasításban megadhatóak a tulajdonos felhasználó megadásához hasonló módon – viszont ezek már tényleg programfüggő módon történhetnek.

Ezeket az információkat utólag is meg lehet adni, vagy meg lehet változtatni – ha már van az adatbázisnak tartalma, akkor nem mindegyiket bölcs dolog megváltoztatni – az `ALTER DATABASE <adatbázisnév> …` utasítás használatával.

Amennyiben már nem kell egy adatbázis, akkor a `DROP DATABASE <adatbázisnév>;` utasítással törölhető. Ám itt érdemes felhívni a figyelmet arra, hogy a valódi adatbázis-kezelők (tehát az ACCESS ez alól mindenképpen kivételnek tekinthető) **megbíznak a felhasználóban**. Ez a DROP DATABASE utasítás esetében azt jelenti, hogy az utasítás kiadása után az adatbázis **annak minden tartalmával együtt azonnal és visszavonhatatlanul megsemmisül**, anélkül, hogy az adatbázis-kezelő előtte a művelet megerősítését kérné!

10.2. Adattábla létrehozása

Amennyiben már van adatbázisunk, akkor abban létre kell hozni a legfontosabb adatbázis-objektumokat: az adattáblákat. Az adattábla felel meg a relációs adatmodell relációjának – legalábbis a legelterjedtebb adatbázis-típus, a relációs adatbázis esetén, de más típusú adatbázissal nem foglalkozunk úgy sem.

Egy adattábla létrehozására szolgál a `CREATE TABLE` utasítás. Ez már összetettebb az adatbázist létrehozó utasításnál, így egy példán keresztül kezdjük a bemutatását. Amennyiben az adattáblákat az SQLite Browser segítségével hozzuk létre, akkor nem feltétlenül kell közvetlenül ezt az utasítást használni, de az elkészült adattábla nevén jobb egér gombbal kattintva elérhető menüben, a „*Copy Create Statement*” menüpontot választva a vágólapra másolódik az adott adattábla létrehozására alkalmas SQL utasítás. Egy ilyen SQL utasítást mutat a 53. ábra az oldal alján. Lássuk ezen a példán, hogyan is néz ki az utasítás.

Az első sorban a CREATE TABLE kulcsszavak után – amiről az utasítás a nevét kapta – a tábla neve szerepel. Itt érdemes megjegyezni, hogy a példán az azonosítók ugyan idézőjelek között szerepelnek, de ez nem előírás csak abban az esetben, ha egyébként SQL kulcsszó lenne a megadott név, vagy olyan karakter szerepel benne, amely egyébként nem lehetne azonosító neve (pl. szóköz vagy kötőjel stb.). Ha viszont idézőjelek között szerepel, akkor pontosan a megadott karakterekből áll a név, tehát a kis- és nagybetűk is megkülönböztetendőek, míg egyébként a kis- és a nagybetű azonosnak számítana az SQL nyelvben. A tábla neve (jelen esetben: Songs) után egy zárójelpáron belül történik az adattábla szerkezetének megadása. Ez a zárójel a példában az első sor végén nyílik és az utolsó sorban zárul be. A zárójelen belüli részhez tartozó sorokat az olvashatóság kedvéért szokás bentebb kezdeni, de mint már volt szó róla, az SQL nyelv utasításai tetszőlegesen tördelhetőek, így a bentebb kezdés sem befolyásolja az utasítás jelentését.

```
CREATE TABLE "Songs" (
    "browseId"  TEXT,
    "title" NUMERIC NOT NULL,
    "albumTitle"    TEXT,
    "albumId"   TEXT,
    "duration"  TIME,
    "downloaded"    TEXT DEFAULT NULL,
    "lastplayed"    DATE DEFAULT NULL,
    "durationString"    TEXT,
    "artists"   TEXT,
    PRIMARY KEY("browseId")
)
```

53. ábra: Példa a CREATE TABLE utasításra

A zárójelen belül minden sor egy egy mezőt – a relációs modell és az egyed-kapcsolat modell attribútumának megfelelőjét – határoz meg, amelyet a következő mező megadásától egy vesszővel kell elválasztani. Kivétel az utolsó sor (`PRIMARY KEY...`), amely már nem mezőt, hanem egy feltételt ír elő.

Egy mező definiálása során először annak nevét kell megadni (az idézőjelek használata itt sem kötelező), majd annak típusát és esetleg az adott mezőre vonatkozó valamilyen feltétel megadása is szerepelhet. Az utolsó mező megadása után szerepelhet még egy vagy több olyan feltétel megadása is (a példában egy szerepel), amely akár egynél több mezőre is előírhat valamilyen közös feltételt. Ezek a táblafeltételek. A jelen esetben megadott feltétel egyetlen mezőre, a *browseId* nevűre vonatkozik, így megadható lenne ott is, amely esetben a mező megadása a következőképpen nézne ki:

```
browseId TEXT PRIMARY KEY,
```

Ez a feltétel történetesen azt jelenti, hogy a *browseId* mező a tábla kulcsa. A kulcsot logikusabb a végén megadni, mert egyrészt akkor azonnal látszik, ha egynél több kulcsot akarnánk definiálni – holott csak egyet lehet, ezért „primary” –, valamint a valóságban használt adatbázisokban sokszor fordul elő olyan, hogy egy adatbázisnak több mező együtt alkotja a kulcsát. A lehetséges feltételekről és azok megadási módjáról egy kicsit később lesz szó részletesen.

A mező nevét követő szó (a példában: TEXT, NUMERIC, DATE, TIME) a mező típusát adja meg. Arról, hogy milyen típusokat lehet használni, a következő részben lesz szó.

A kulcsot megadó feltétel mellett még két feltételre van példa a vizsgált utasításban. Ezek közül az egyik a NOT NULL, a másik pedig a DEFAULT NULL. Mindkettő a speciális NULL értékkel függ össze: ha egy mező értéke NULL, akkor annak a mezőnek valójában nincs értéke. Más kifejezésekben a NULL jelentése *ismeretlen*, *meghatározatlan* is lehet. Itt a két feltétel közül az első előírja, hogy a title mezőnek mindig kell legyen értéke, azaz nem lehet NULL, míg a másik esetben a *downloaded* és a *lastplayed* mezőknek alapértelmezett értékként pont a NULL van megadva, azaz ha olyan rekordot (relációs modellben ezt még sornak hívtuk) adunk meg, amelynél ez a mező nem kap értéket, akkor az értéke alapértelmezésben NULL lesz.

Amennyiben utólag újabb mezőt szeretnénk felvenni a táblába, egy létező mezőt törölni szeretnénk, valamelyik mező típusát módosítani szeretnénk, mert eredetileg rosszul lett megadva, akkor azt az `ALTER TABLE …` utasítással lehet megtenni. Azonban ennek az utasításnak a használatával vigyázni kell ha már van a táblában adat, mivel csak akkor lehetséges a tábla szerkezetét módosítani, ha az új szerkezetbe a már létező adatok egyébként beilleszthetőek. Az `ALTER TABLE` utasítással utólag is fel lehet venni feltételeket is, de már csak táblafeltételként. De okos informatikus előre megtervezi az adatbázist, így az utólagos táblamódosításokat elkerüli… Kivéve, ha az adatbázis használata során derül ki, hogy újabb szempontok miatt módosítani kell azt!

Amennyiben egy táblára nincs már szükség, akkor az a `DROP TABLE <táblanév>` utasítással törölhető. Ám ekkor is igaz a fentebb, az adatbázis törlésénél már említett megjegyzés: az adatbázis-kezelő megbízik a felhasználóban, azaz megerősítő kérdés nélkül hajtja végre az utasításokat – jelen esetben a tábla végleges, visszavonhatatlan törlését annak teljes tartalmával együtt.

10.2.1. Adattípusok

Karakter típusok

Képernyőn megjeleníthető – hivatalosan nyomtatható karakterek – tárolására szolgáló típusból alapvetően kétfélét ismer az SQL: fix hosszúságú és változó hosszúságú karakterláncot. Mindkét esetben a típus neve részeként meg kell adni annak hosszát.

Fix hosszúságú karakterlánc adható meg a `char`(*n*) vagy `character`(*n*) típusmegadással. Akár a rövidebb, akár a hosszabb nevet használjuk, olyan mező jön létre, amely pontosan *n* karakter tárolására szolgál. Amennyiben ennél hosszabb szöveget akarunk benne eltárolni, akkor hiba keletkezik. Adatbázis-kezelőtől függ, hogyan kezeli le ezt a hibát: hibajelzést ad, vagy egyszerűen nem tárolja el a többlet karaktereket. Amennyiben a tárolandó szöveg rövidebb a megadott korlátnál, ak-

kor szintén a konkrét megvalósítástól függ a viselkedés, de általában automatikusan feltöltődik a szöveg baloldalról szóközökkel a kívánt hosszúságra.

A változó hosszúságú karakterlánc esetében is meg kell adni a maximum hosszt, de a rövidebb szöveg fizikailag kevesebb helyet foglal el. Ezt általában a `varchar`(*n*) vagy `character varying`(*n*) típusmegadással lehet megadni.

Sok adatbázis-kezelő egy ezeknél rugalmasabban kezelhető szöveges típust is alkalmaz, általában a `text` névvel. Emiatt a szabvány is bevezette újabban a CLOB nevet. Míg a char és a varchar esetében általában 255 karakter a felső határ, a `text` illetve a `clob` típus esetében ennél sokkal hosszabb, akár több ezer karakteres szövegek is tárolhatóak.

Lássuk részletezve, melyik adatbázis-kezelő pontosan milyen típusokat ismer!

- MySQL/MariaDB[8]:
  - `char`(*n*): fix szélességű karakterlánc, ahol *n* a hossz értéke. $n < 2^8$
  - `varchar`(*n*): változó szélességű karakterlánc, legfeljebb *n* karakter tárolására. $N < 2^{16}$
  - `tinytext`: Nagyon kis méretű karakterlánc, legfeljebb 2^8-1 karakter tárolására (de nem kell előre megadni a maximumot). Tehát megegyezik a `varchar(255)` típusmegadással.
  - `text`: Kis méretű karakterlánc, legfeljebb 2^{16}-1 karakter tárolására. Megegyezik a `varchar(65535)` típusmegadással.
  - `mediumtext`: Közepes méretű karakterlánc, legeljebb 2^{24}-1 karakter tárolására.
  - `longtext`: Nagy méretű karakterlánc, legfeljebb 2^{32}-1 karakter tárolására.
- SQLite:
  - `text`: Szöveg tárolására szolgáló adattípus (a dokumentáció nem tartalmazza a lehetséges maximális hosszt).
- PostgreSQL:
  - `character`(*n*), `char`(*n*): Pontosan *n* karakter tárolására alkalmas karakterlánc.
  - `character varying`(*n*), `varchar`(*n*): Legfeljebb *n* karakter tárolására alkalmas, a ténylegesen tárolt szöveghez igazodó tárolóhelyet használó karakterlánc.
  - `text`: Elvi korlát nélküli szövegtárolásra alkalmas típus, amelynél a tényleges helyfoglalás az aktuálisan tárolt szöveg hosszához igazodik.
- MS Access:
  - Rövid szöveg: Legfeljebb 255 karakter tárolására alkalmas szöveg.
  - Hosszú szöveg: Legfeljebb 63 999 karakter tárolására alkalmas szöveg.

Bináris adatok

Az SQL szabvány definiál adattípust bináris adatok tárolására is (ez akkor jó, ha nem nyomtatható karaktereket is tartalmazó, tehát hivatalosan nem karakterláncnak számító byte-sorozatot kell tárolni). Ezekkel a típusokkal mi nem fogunk foglalkozni, így csak említés szintjén a szabványos típusnevek: BINARY, VARBINARY és BLOB. A típusok a karakterláncnál szereplő CHAR típusok mintájára használhatóak.

Numerikus típusok

Az SQL ismeri az egész és a lebegőpontos adattípust is, de – mivel pénzügyi adatok tárolására egyik sem jó választás – ismer olyan típust is, amely tetszőleges számjegy pontossággal képest számadatok tárolására.

8 Itt csak egy része van felsorolva a lehetséges típusoknak!

A hagyományos egész típust az Int vagy Integer típus képviseli. Az SQL szabvány is több egész típust ismer, amelyek a tárolható számok nagyságában térnek el egymástól. Az egyes megvalósításokban nem feltétlenül azonos méretűek ezek a típusok. Az alábbiakban ezek a szabványos egész típusok szerepelnek, mindenhol felsorolva azokat a bennünket esetleg érdeklő adatbázis-kezelőket, amelyekben az adott típus definiálva van.

`integer`: Ez az alap egész típus, alapértelmezésben ezt használjuk! Csak akkor használunk másikat, ha ennél nagyobb tárolókapacitású kell, vagy a tárolóhellyel takarékoskodás céljából kisebb méretű adattípusra van szükség.

- MySQL/MariaDB esetén az `integer` és az `int` szó azonos jelentéssel rendelkezik. A szó után zárójelbe írható egy legfeljebb 255 értékű nemnegatív egész szám, amely az érték megjelenésének méretét szabályozza. További lehetséges kisebb vagy nagyobb méretű egész típusok nevei:
  - `BIT`(*size*): a paraméterben megadott számú (1 és 64 közti érték lehet) bitet tud tárolni.
  - `TINYINT`: 1 byte méretű egész számokat lehet benne tárolni. Előjeles számok esetén a legkisebb értéke -128, a megnagyobb 127. Előjel nélküli számok esetén 0 és 255 közti értékek férnek bele.
  - `BOOL` és `BOOLEAN`: a neve alapján logikai értéket tárol, de valójában 0 érték hamisnak, minden más érték igaznak számít.
  - `SMALLINT`: 2 byte méretű számokat tud tárolni.
  - `MEDIUMINT`: 3 byte méretű számokat tud tárolni.
  - `INT` vagy `INTEGER`: 4 byte méretű számokat tud tárolni.
  - `BIGINT`: 8 byte méretű számokat tud tárolni.
- SQLite: az egyetlen egész típus, amit ismer, a szabvány által előírt `INTEGER`, azonban a ténylegesen használt tárolási méret igazodva a tárolt érték igényeihez, lehet 1, 2, 4, 6 vagy 8 byte.
- PostgreSQL: szintén INTEGER a szabványos típusnév. Emellett az alábbi egész típusok léteznek még:
  - `smallint`: 2 byte méretű
  - `integer`: 4 byte méretű
  - `bigint`: 8 byte méretű
  - `smallserial`, `serial`, `bigserial`: 2, 4, illetve 8 byte méretű egész típus, amely automatikusan növekvő értékeket alkalmaz adat beszúrásánál.
- Az MS Access három egész típust ismer:
  - Byte: 1 byte méretű
  - Integer: 2 byte méretű
  - Long: 4 byte méretű

Az általunk használt keretek között bőven elég lesz az INTEGER típus, amely mint látható, mindegyik adatbázis-kezelőben azonos néven elérhető, bár a benne tárolható adatok mérete eltérő lehet.

`float`: A lebegőpontos adatok tárolására leginkább a DOUBLE nevet érdemes használni, mert ez az, amely mindegyik esetben elérhető. Egyes adatbázis-kezelők még ismernek néhány egyéb típust is:

- MySQL/MariaDB:
  - `FLOAT`(*size*, *d*): Az összes értékes számjegyet a *size* paraméter adja meg, míg a *d* a tizedes jegyek számát.

    - `DOUBLE`(size, d) vagy `DOUBLE PRECISION`(size, d): a `FLOAT`-hoz képest dupla pontosságú tárolást tesz lehetővé. A paraméterek jelentése ugyanaz, mint a `FLOAT` esetén.
- Az SQLite a `REAL` típusnevet használja a legebőpontos számok megadására.
- PostgreSQL:
    - `real`: az egyszeres pontosságú lebegőpontos szám neve (4 byte)
    - `double precision`: a dupla pontosságú lebegőpontos szám neve (8 byte)
- MS ACCESS:
    - Single: egyszeres pontosságú lebegőpontos szám
    - Double: dupla pontosságú lebegőpontos szám

`decimal`: A fentieken felül az SQL szabvány még definiál olyan decimális típust is, amely nem kettes, hanem tízes számrendszerben tárolja a számokat, így a számolás velük kicsivel hosszabb időt vesz igénybe (mert szoftveresen kell a legtöbb gépen megoldani), de mindig pontos tárolást tesz lehetővé – nyilván a decimális tárolás miatt több helyigénnyel.

A szabvány által definiált két pontos számtípus a NUMERIC(p,s) és a DECIMAL(p,s). A kettő valójában egymással azonos. Mindkét esetben a *p* a pontosságot határozza meg, míg az *s* ezen belül a tizedesjegyek számát. Azaz a *p* megadja, hogy összesen hány számjegy tárolandó, míg az *s* ezen belül a törtrész tárolására használt számjegyek számát adja meg.

- MySQL/MariaDB:
    - A `decimal` helyett azonos jelentéssel használható a `dec`, `numeric` és a `fixed` szó is. A maximális számjegyszám 65, tizedesjegyekre a 30.
- SQLite: Az SQLite ezt a típust nem ismeri.
- PostgreSQL:
    - `numeric` és `decimal`: mindkét néven ugyanazt a típust lehet megadni, ahogyan azt a szabvány is előírja.
- MS Access:

Dátum és idő

A DATE típus dátumok, a TIME időadatok, a TIMESTAMP dátum+időadatok, míg az INTERVAL egy időtartam megadására szolgál.

A DATE az éveket, hónapokat és napokat tárolja. A TIME órákat, perceket és másodperceket tárol. A TIMESTAMP az évektől a másodpercekig minden időadatot tárolni tud. A másodperceket lehet tizedesjegyekkel egésznél kisebb értékként is megadni. A TIME és a TIMESTAMP képes az időzóna tárolására is.

A szabvány hivatalosan kétfajta INTERVAL-t ismer: az egyikben a napok közti időtartamot, a másikban másodpercekben mért (ezáltal óra, perc, másodperc formában megadott) időtartamot lehet megadni.

A szabvány a dátum- és időértékek megadására speciális formátumú szövegkonstansokat definiál – amelytől csak a MS Access tér el –, amelyek így általában a szövegkezelő műveletekkel is használhatóak. A dátum alakja: `'YYYY-MM-DD'`, ahol `Y` az évet, `M` a hónapot, `D` a napot jelenti. Az idő alakja: `'hh:mm:ss.9999'`, ahol `h` az órát, `m` a percet, `s` a másodpercet, a `9` számjegy pedig azt jelöli, hogy meg lehet adni törtmásodpercet is. Amennyiben olyan típust használunk, amely mind dátumot, mind időadatot képes tárolni, akkor a két rész közé egy szóközt kell írni.

- MySQL/MariaDB:
    - `date`: Ez az adattípus `'1000-01-01'` és `'9999-12-31'` közötti dátumot tud tárolni.

    - `time`: `'-838:59:59.99999'` és `'838:59:59.99999'` közötti időadatokat tud a típus tárolni.
    - `datetime`, `timestamp`: Ez a típus `'1000-01-01 00:00:00.00000'` és `'9999-12-31 23:59:59.99999'` közti időpontokat tud tárolni.
    - `year`: 1901 és 2155 közötti évszámot képes négy jegyű értékként tárolni, valamint még a `0000` lehet legális értéke.
- SQLite: Az SQLite nem ismer dátum vagy idő típust. Amennyiben a TEXT típust használjuk, akkor a szabványos karakterkonstans formátummal lehet dátum- és időadatokkal dolgozni. Emellett a REAL típus a Julián naptár napjaival tud dolgozni, míg az INTEGER típussal lehet a UNIX-idő szerinti, 1970-01-01 nulla óra óta eltelt másodpercek formájában időadatokat tárolni.
- PostgreSQL:
    - `timestamp`: A típus mikromásodpercekben tartalmazza az 1970. 1. 1-től eltelt időt az adott időpillanatig. 8 byte a tárolási mérete. Mivel negatív értéket is tud tárolni, így a legkorábbi tárolható időpont bőven időszámításunk előttre nyúlik vissza (4713 i. e.), míg a legnagyobb érték is elépesztően távoli jövőben van: 294 276. A típusnak két változata van az alapján, hogy a `timestamp` szó után melyik szerepel:
        - `without time zone`: amennyiben ez a szöveg szerepel, vagy semmi nem szerepel, akkor az értékek az UTC idő szerint értendőek.
        - `with time zone`: ezt a szöveget megadva, az érték részeként meg kell adni az időzóna azonosítóját is, amelyben az értendő. Az értéket lekérdezve is szerepelni fog a megadott időzóna.
    - `date`: Dátum tárolására szolgáló típus. Tárolása 4 byte-on történik.
    - `time`: Időérték tárolására szolgál 8 byte méretben, mikromásodperc pontossággal. Lehet hozzá időzónát is megadni a `timestamp` esetében bemutatott módon, amely esetben 12 byte lesz a tárolási mérete.
    - `interval`: időtartam megadására szolgáló típus 16 byte méreten tárolva, mikromásodpercekben számolva az eltelt időt.
- MS Access:

Logikai értékek

Az SQL három-értékű logikát használ, amelynek értékeit a BOOLEAN típusban lehet tárolni.

A három lehetséges érték:

1. igaz: `true`
2. hamis: `false`
3. ismeretlen: `NULL`

A három-értékű logikai működéséről majd a lekérdezés feltételeinek megadásánál lesz szó.

- MySQL/MariaDB:
    - `boolean`: Hivatalosan egész típusnak számít, a 0 érték a hamis, 1 érték az igaz értéknek felel meg.
- SQLite: Az SQLite hivatalosan nem ismeri, de elfogadja a boolean típusnevet, amelyet az integer szinonímájának tekint. Elfogadja továbbá a TRUE és a FALSE értékeket.
- PostgreSQL: A `boolean` a típus neveként, mindkettő ugyanazt a típust jelenti. Az igaz érték megadható így: `true`, `yes`, `on`, `1`. A hamis érték megadható így: `false`, `no`, `off`, `0`. Az értéket `t` vagy `f` formában kapjuk kiíráskor.

- MS Access:

10.2.2. Feltételek

Blabla...

10.3. Indexek

Blabla...

10.4. Nézettáblák

Blabla...

Tábláblázatkezelés

11. Adatbázis-feladatok

Ezek a feladatok a későbbiekben elkészülő leckékbe lesznek integrálva. Addig azonban csak így egymás után szerepelnek…

11.1. Tanári kar

A feladat eredetije 2005-ben szerepelt a középszintű informatika érettségin.

A `tanarikar.sqlite` állományt az *SQLite Browser* programmal megnyitva lehet az adatbázist használni. A *phppgadmin* számára a forrásállomány, amelyből az adatbázis létrehozható, a `/public/adatbazis/tanarikar/` könyvtárban, `tanarikar.pg.sql` néven található. Ezt a *phppgadmin*-ba az SQL menüponton keresztül közvetlenül be lehet importálni, aminek hatására létrejön egy **tanarikar** nevű séma, abban a **tanarok** nevű adattáblával. Ugyanebben a könyvtárban, a `tanarikar.my.sql` állomány segítségével lehet az adatbázist a XAMPP adatbázisába beimportálni, az állomány a **tanarikar** nevű adatbázist létrehozó utasítást is tartalmazza.

Ebben az adatbázisban egy iskolában valaha tanító tanárok névsora, a szaktárgyaik a kezdés és a kilépés éve található meg.

Az adattábla a következő mezőket tartalmazza:

1. **azon**: a sorok egyedi azonosítója, kulcsként, integer típussal, hozzá automatikus sorszámozás rendelve.
2. **nev**: text típusú, tehát szöveges adat; az aktuális tanár nevét tartalmazza.
3. **szak**: text típusú, tehát szöveges adat; a tanár szakjainak megnevezése, amennyiben több szakja van a tanárnak, akkor kötőjel van azok között.
4. **mettol**: integer típusú adat; a belépés évét adja meg.
5. **meddig**: integer típusú adat; a kilépés évét adja meg

A következő pontokban felsorolt lekérdezéseket készítsd el az adatbázison annak importálása után! Mivel a phppgadmin nem teszi lehetővé a lekérdezések elmentését, így azok SQL parancsait egy-egy szöveges állományba kell elmentened. Az állomány neve a feladat sorszámát és a feladat nevét tartalmazza, .sql kiterjesztéssel ellátva. Például a második feladat esetében az állomány neve `2mat.sql` legyen! Az állomány neve a feladat végén, zárójelben szerepel.

1. Lekérdezésben gyűjts ki azon tanárok nevét, akik 1986-ban vagy 1987-ben kezdtek tanítani az iskolában! (`1kezd.sql`)
2. Gyűjtsd ki azon tanárok nevét és szaktárgyait, akik matematika tantárgyat tanítottak (a többszakos tanároknál minden szak fel van sorolva)! Anélkül, hogy megjelenítenéd, rendezd a kiírást a belépés éve szerinti növekvő sorrendbe! (`2mat.sql`)
3. Számítsd ki, hány évet tanított egy-egy tanár az iskolában! Add meg a tanár nevét, szakját és az iskolában eltöltött időt! (`3ido.sql`)
4. Add meg az iskolában leghosszabb ideig dolgozott tanár nevét és az iskolában eltöltött éveinek számát! (`4leg.sql`)
5. Hány tanár van az adatbázisban, aki 2000-ben az iskolában tanított? (`5ketezer.sql`)
6. Az előző évszázadban (2000. is a 20. század része) hány angol tanár volt összesen az iskolában az adatbázis szerint? (`6angol.sql`)

11.2. Tanulmányi versenyek

Ennek a feladatnak az alapja az azonos nevű feladat, amely 2005. tavaszán szerepelt a középszintű informatika érettségin.

A `/public/adatbazis/tanulmanyi_verseny/` könyvtárban található forrásállományt (`versenynaptar.slite` az SQLite Browser, `versenynaptar.pg.sql` a phpPGAdmin használatához `versenynaptar.my.sql` a XAMPP használatához) beimportálásával létrejövő **adatok** adattábla a 2004/2005-ös tanév – Oktatási Minisztérium által meghirdetett vagy támogatott – tanulmányi versenyeit tartalmazza. Egy-egy verseny több rekordban is szerepel, mert minden – a versenyhez kötődő – esemény külön szerepel benne. A tábla az alábbi módon épül fel:

- **azon**: a tábla soraihoz, azok sorszámozásával rendelt kulcs. Típusa: egész szám.
- **versenynév**: a verseny megnevezése. Típusa: text
- **típus**: a verseny típusa, pl. OKTV. Típusa: text
- **esemény**: az esemény megnevezése, pl. „Nevezés”, „I. forduló” stb. Típusa: text
- **dátum**: az adott esemény dátuma. Típusa: date

Az alábbi feladatok megoldását a végén, zárójelben megadott nevű szöveges állományba kell elmentened.

1. Lekérdezéssel gyűjtsd ki növekvő időrendi sorrendben a nevezési időket! Jelenítsd meg a verseny nevezési idejét, megnevezését és típusát! (`1nevezes.sql`)
2. Lekérdezésben add meg, hogy típusonként hány esemény szerepel az adatbázisban (pl. hány OKTV van)! (`2tipus.sql`)
3. Készíts lekérdezést, amely megadja a 2005. februári versenyjellegű (tehát nem nevezés) események összes adatát, növekvő időrendi sorrendben! (`3februar.sql`)
4. Készíts lekérdezést, amely időrendi sorrendben megjeleníti a Fizika OTKV versenyek valamennyi eseményének időpontját! Fizika OKTV-ből több kategória is van, csak annyit tudunk, hogy a verseny megnevezése a ’Fizika’ szöveggel kezdődik. Éppen ezért szeretnénk látni a verseny pontos megnevezését, az időpontját és az esemény típusát (**esemény** mező) is. (`4fizika.sql`)

11.3. Kosárlabda

Ez a feladat a 2006. tavaszi középszintű informatika érettségi azonos című feladatából származik.

Egy kosárlabda-mérkőzés egyik csapatának játékosai állnak rendelkezésre az adatbázisban, amely a `kosarlabda.sqlite` (SQLite Browser), `kosarlabda.pg.sql` (*phppgadmin*) és a `kosarlabda.my.sql` (XAMPP, *phpmyadmin*) állományokból állítható elő. Az adatbázis neve: `kosar`

Az adatbázis két adattáblát tartalmaz:

- **játékos**
  - *név*: A játékos neve, `text`
  - *mez*: A játékos mezszáma, `int(eger)`, a tábla kulcsa
  - *magasság*: A játékos magassága, `int(eger)`
  - *poszt*: A játékos feladata a mérkőzésen, `text`
- **jegyzőkönyv**
  - *azon*: A jegyzőkönyv egy bejegyzésének azonosítója, `int(eger)`, a tábla kulcsa
  - *mez*: A játékos mezszáma, `int(eger)`, külső kulcs a játékos tábla azonos nevű mezőjére (ez a MySQL és az SQLite adatbázis számára be van állítva)
  - *be*: A pályára lépés időpontja, `time (without time zone)`
  - *ki*: A lecserélés időpontja, `time (without time zone)`
  - *bkis*: Kosárra dobási kísérletek száma, `int(eger)`
  - *bjó*: A jó dobási kísérletek száma, `int(eger)`

A feladatok megoldásaként keletkező SQL utasítást a feladat végén zárójelben található nevű állományba kell elmentened.

1. Lekérdezés segítségével sorold fel a játékosok nevét, magasságát és mezszámát névsorban! (`1jatekosok.sql`)
2. Víg Péter a mérkőzés során mikor állt be és mikor cserélték le? (`2vig.sql`)
3. Lekérdezés segítségével határozd meg, hogy Magas Viktornak összesen hány jó dobási kísérlete volt a mérkőzésen! (`3magas.sql`)
4. Játékosonként listázd ki az összes jó dobási és összes kosárra dobási kísérletek számát! A lista a jó dobási kísérletek szerint csökkenő sorrendben jelenjen meg! (`4dobas.sql`)
5. Lekérdezéssel keresd meg annak a játékosnak a nevét, aki a 35 perc 0 másodperc – 40 perc 0 másodperc időintervallumban irányító posztra állt be cseréként! (`5iranyito.sql`)
6. A mérkőzésen összesen az egyes posztokon hány különböző játékos játszott? (`6posztok.sql`)

11.4. Thorma János Múzeum

A nagybányai festőiskola néhány szép képét őrzi a kiskunhalasi Thorma János Múzeum. A gyűjtemény képeit és azok festőinek adatait dolgozza fel az alábbi feladat. A galéria adatai megtalálhatóak a `kepek.txt` és `festok.txt` állományokban. A kész adatbázis SQLite változata a `thorma.sqlite` állományban található.

Készíts új adatbázist MS Access használata esetén thorma.accdb néven, és importáld be a két UTF-8 kódolású, tabulátorokkal tagolt CSV állományból az adattáblákat, beállítva az alábbiakban

megadott adattípusokat és kulcsokat! SQLite használata esetén elég a már kész adatbázist megnyiti az SQLite Browser alkalmazással. A feladat során elkészítendő lekérdezéseket előbbi esetben az adatbázisban kell létrehoznod, utóbbi esetben feladatonként egy-egy szöveges állományt kell létrehoznod a feladat végén megadott nevű, .sql kiterjesztésű állományban.

Íme a két adattábla relációs modellben megadva:

kepek(*leltar*, *fazon*, *cím*, *keszult*, *anyag*, *technika*, *szeles*, *magas*)

- *leltar*: A kép leltári azonosítója, szöveg, ez a kulcs
- *fazon*: A kép festőjének azonosítója, szám, külső kulcs
- *cim*: A kép címe, szöveg
- *keszult*: A kép alkotásának befejező éve, szám
- *anyag*: A kép alapjának anyaga, pl.: vászon, fa, szöveg
- *technika*: A kép festéstechnikája, pl. olaj, szöveg
- *szeles*: A kép szélessége centiméterben, valós szám, tizedesjegyek száma: 1
- *magas*: A kép magassága centiméterben, valós szám, tizedesjegyek száma: 1

festok(*azon*, *nev*, *szuletett*, *meghalt*)

- *azon*: A festő azonosítója, szám, ez a kulcs
- *nev*: A festő neve, szöveg
- *szuletett*: A festő születési éve, szám
- *meghalt*: A festő halálának éve, szám

A két tábla között az alábbi ábrán látható kapcsolat áll fenn:

A következő feladatok megoldásánál a lekérdezéseket a zárójelben megadott néven mentsd el! Ügyelj arra, hogy a megoldásban pontosan a kívánt mezők szerepeljenek!

1. Add meg azoknak a festőknek a nevét, akik éltek az I. világháború kitörésének évében (azaz 1914-ben vagy előtte születtek és 1914-ben vagy utána haltak meg)! A lekérdezés a festők nevét ABS-sorrendben jelentítse meg! (`1haboru`)
2. Készíts lekérdezést, amely megadja azokat az anyagokat, amelyekre festettek „*olaj*” technikával képet! Minden anyag neve csak egyszer jelenjen meg a listában! (`2anyag`)
3. A művészeket sok esetben megihlett a táj. Készíts lekérdezést, amely megadja azokat a képeket, amelyek címében szerepel a „*domb*” karaktersorozat! Jelenítsd meg a kép címét, szélességét és magasságát, valamint a festő nevét! (`3domb`)
4. Add meg lekérdezéssel, hogy melyik festő készített a legmagasabb képet! A kép címét és a festő nevét jelenítsd meg! (`4magas`)
5. Készíts lekérdezést, amely meghatározza, hogy melyik évben hány kép készült! A lista legyen az alkotások darabszáma szerint csökkenő sorrendben! (`5evek`)

11.5. Utónevek

A feladat a 2006. február 28-i középszintű informatika érettségi azonos nevű feladata alapján készült.

A feladat forrásállományai a `/public/adatbazis/utonevek/` könyvtárban találhatóak meg. Az `utonevek.pg.sql` a *phppgadmin*, az `utonevek.my.sql` a *phpmyadmin* (XAMPP) felületen keresztül teszi lehetővé az adatbázis létrehozását. Utóbbi esetben szükséges adatbázis-létrehozási joggal rendelkezni.

Az adatbázisban (PostgreSQL esetén a sémában) egy **lista** nevű tábla található. Ez a tábla a BM Központi Hivalának honlapjáról származó adatokat tartalmaz a 2005. január 1-jén elérhető statisztikai adatokkal az utónév választásról.

Az adattábla mezői a következők:

- **utónév**: a szóban forgó keresztnév, *ez a tábla kulcsa*, típusa: `text / varchar(50)`
- **első**: 2005-ben ennyi embernek volt az adott név első utóneve, típusa: `int(eger)`
- **második**: 2005-ben ennyi embernek volt az adott név a második utóneve, típusa: `int(eger)`
- **újsz1**: 2004-ben ennyi újszülött kapta első utónévként az adott nevet, típusa: `int(eger)`
- **újsz2**: 2004-ben ennyi újszülött kapta második utónévként az adott nevet, típusa: `int(eger)`
- **neme**: a név férfi (F) vagy női (N) név, típusa: `char(1)`

A tábla eredetileg négy külön listából származik, amelyek a 100 leggyakoribb férfi/női neveket valamint a 100 leggyakoribb leány/fiú újszülöttnek adott nevet tartalmazza. Az összemásolás eredményeként több olyan sor is szerepel az adattáblában, amelynél nem mind a négy számot tartalmazó mezőben van adat. Ott, ahol nincs adat, a mező értéke NULL (mySQL esetén 0). Ha egy név nél az első név előfordulását nem ismerjük, ott a második névként való előfordulás számát sem.

Az alábbi feladatok megoldására szolgáló SQL utasítást a feladat végén, zárójelben szereplő nevű szöveges állományba mentsd el!

1. Lekérdezéssel add meg nemenként a táblázatban szereplő nevek számát, valamint hogy első keresztneve alapján hány férfit illetve nőt vettek számba! (`1nemenkent.sql`)
2. Készíts lekérdezést, amely kigyűjti azokat a neveket, amelyekben az „anna” szórészlet szerepel! (`2anna.sql`)
3. Lekérdezéssel add meg a 2004-ben divatossá váló neveket: azokat a neveket, amelyekről első keresztnévként nincs adat, csak újszülöttként adott névként. Jelenítsd meg, hány újszülött kapta e neveket első, illetve második keresztnévként! Az adatok legyen rendezve a 2004-es újszülöttek névadási alapján első utónév, majd ezen belül második utónév száma szerint csökkenő sorrendbe! (`3divat.sql`)
4. Késztsd el azt a lekérdezést, amely kilistázza a 10 leggyakoribb első névként használt férfinevet, valamint azok előfordulási számát! (`4ferfinev.sql`)
5. Készíts lekérdezést, amely az újszülött névként adattal nem rendelkező női neveket és számszerű adataikat névsorban kilistázza! (`5ritkanoi.sql`)
6. Készíts lekérdezést, amely kilistázza az egyes nevek összes előfordulását azáltal, hogy minden névhez kiírja annak nemét, élő oszlopnévvel az első és második név előfordulási adatainak összegét, valamint újszülött oszlopnévvel az újszülöttek első és második névnek adott értékének összegét! (`6osszes.sql`)

11.6. Hajómenetrend

Az alábbi feladat[9] a balatoni hajómenetrend egy egyszerűsített kivonatát tartalmazza.

A feladathoz tartozó forráskönyvtár: `/public/adatbazis/hajomenetrend/`

A könyvtárban található forrásállományok: `hajomenetrend.pg.sql` a PostgreSQL (phppgadmin) számára és `hajomenetrend.my.sql` a MySQL (phpmyadmin, XAMPP) számára.

Az adatbázis a következő, **menetrend** nevű adattáblát tartalmazza:

- **azon**: Két állomás közötti út azonosítója, a tábla kulcsa, típusa: számláló
- **járat**: A hajóútvonal azonosítója, amelyen az adott út található, típusa: `text`
- **honnan**: A hajóút induló állomása, típusa: `text`
- **hova**: A hajóút érkező állomása, típusa: `text`
- **indul**: Indulási idő az állomásról, típusa: `time (without timestamp)`
- **érkezik**: Érkezési idő az állomásra, típusa: `time (without timestamp)`

Oldd meg az alábbi feladatokat az adatbázison! Minden megoldást a feladat végén, zárójelben megadott nevű szöveges állományba mentsd el!

1. Írja ki a lekérdezés a J1 hajójárat menetrendjét! Jelenjenek meg az indulási és érkezési állomások az időpontokkal együtt! (`1j1_menetrend.sql`)
2. Listázd ki, hogy Balatonfüredről milyen állomások felé indulnak hajók 11 óra 30 perc és 12 óra 30 perc között (beleértve a megadott időpontokat is)! (`2balatonfured.sql`)
3. Határozd meg lekérdezés segítségével, hogy naponta hány hajó érkezik a kikötőbe! A lista az állomás nevéből és a kikötő hajók számából álljon, az utóbbi értéke szerint csökkenő sorrendben! (`3hajok.sql`)
4. Mikor érkezik a legkésőbbi hajó Balatonföldvárra? (`4utolso.sql`)
5. Az E2-es hajójáratnak mi a végállomása és mikor érkezik oda? (`5vegallomas.sql`)

9 A feladat a 2007. májusi középszintű informatika érettségi adatbázis-kezelés feladat felhasználásával készült.

11.7. Mozdonyok

Ez a feladat a 2007. őszi középszintű informatika érettségi adatbázis feladata módosításával készült.

Forrásállományok helye: `/public/adatbazis/mozdonyok/`

`mozdonyok.pg.sql`: a phppgadmin számára

`mozdonyok.my.sql`: a XAMPP (phpmyadmin) számára

`mozdony.csv`: UTF-8 kódolású, vesszővel elválasztott CSV formátumú állomány az adatokkal esetlegesen a Microsoft Access-beli feladatmegoldáshoz.

Az adatbázis neve **mozdony** PostgreSQL esetén, **mozdonyok** MySQL esetén. A benne található adattábla neve: **mozdonyok**. A tábla mezőinek jelentése:

- **sorozat**: a mozdony „fajtájának” megnevezése (pl.: V43, 1047, stb.) típusa: `text` / `varchar(10)`
- **psz**: pályaszám, azaz az azonos sorozatú mozdonyok megkülönböztető száma. Típusa: `int(eger)`
- **gyártási_év**: a mozdony gyártási éve. Típusa: `int(eger)`
- **gyártó**: A mozdonyt gyártó cég neve. Típusa: `text`
- **típus**: A sorozaton belüli eltérések jelzésére használt megnevezés. Típusa: `text`
- **állagba**: A jármű állományba vételének dátuma, ekkor vette át a mozdony a tulajdonos. Típusa: `date`
- **tulaj**: A mozdony tulajdonosa. Típusa: `text`

A s**orozat** és ap**Psz** mezők együtt alkotják a tábla kulcsát – ezért lett a MySQL esetében a s**orozat** mező típusa `varchar(10)`, ugyanis a text típust a MySQL nem fogadta el kulcsnak. Ezt az Access használata esetén az adatok importálása után szintén be kell állítani!

Az alábbi lekérdezéseket old meg az adatbázison! Minden lekérdezést mentsd el a feladat végén zárójelben megadott nevű szöveges állományba (Access esetén ez legyen a lekérdezés neve `.sql` végződés nélkül)!

1. Lekérdezésben gyűjtsd ki a GySEV mozdonyainak adatait a sorozat, pályaszám, a gyártás éve, az állományba vétel és a típus megjelenítésével! Az eredményt az állományba vétel dátuma szerint csökkenően rendezve jelenítsd meg! (`1gysev.sql`)
2. A gyártó megnevezésében néhol helytelenül adták meg a nevet. Készíts módosító utasítást, amely kicseréli a „GANZ” bejegyzést „Ganz MÁVAG”-ra! (2ganz.sql)
3. Készíts lekérdezést arról, hogy egy-egy sorozatból, annak gyártóját is megjelenítve hány mozdony szerepel az adatbázisban! (`3fajta.sql`)
4. A típusmegnevezés sok esetben hiányzik. Listázd ki azokat a mozdonyokat (sorozat és pályaszám), amelyeknél nincs megadva ez az adat! (`4ures.sql`)
5. Lekérdezéssel add meg, hogy a MÁV tulajdonban levő mozdonyok között melyik a leggyakoribb évjáratú, és mennyi akkor gyártott mozdonnyal rendelkezik a MÁV! (`5legtobb.sql`)
6. Az állományba vétel időpontja több mozdonynál pontatlanul van feltüntetve. Ez onnan derül ki, hogy az **állagba** mezőben levő dátum az adott év szilvesztere. Lekérdezés segítségével, szükség esetén PostgreSQL esetén az EXTRACT()[10], MySQL esetén a Year(),

10 A függvény használata: év komponenshez: EXTRACT(YEAR FROM dátum), hónap komponenshez: EXTRACT(MONTH FROM dátum) és nap komponenshez: EXTRACT(DAY FROM dátum)

Month() és Day() függvényeket felhasználva gyűjts ki az érintett mozdonyok sorozat- és pályaszámát és az állományba vétel évét! (`6szilveszteri.sql`)

11.8. Iskolába járás

A Mivel Kutatóközpont az utóbbi években rendszeresen vizsgálta a diákok iskolába járási szokásait. 2011-ben évfolyamonként 25 gimnazista és 25 szakközépiskolás iskolába járási módját mérte fel. A diákokról csak a legszükségesebb adatokat rögzítették a vizsgálat indításakor és úgy tekintették, hogy azok a vizsgálat során nem változnak. A feladatban csak a januári és az áprilisi adatokat használtuk fel.

Az így elkészült adatbázist létrehozó állományok a következő könyvtárban találhatóak meg: `/public/adatbazis/iskolaba_jaras/`

Az állományok:

- phppgadmin (PostgreSQL) számára: `suliba.pg.sql`
- XAMPP, phpmyadmin számára: `suliba.my.sql`

Az adatbázis létrehozása során a **suliba** nevű sémában/adatbázisban a következő táblák jönnek létre:

diák tábla:

- **az**: diák azonosítója, típusa: `int(eger)`, ez a kulcs
- **iskolatipus**: character(1) típussal, a diák iskolájának típusa, értéke: g(imnázium), s(zakközépiskola)
- **nem**: character(1) típussal, a diák neme, értéke: f(iú), l(ány)
- **évfolyam**: a diák évfolyama, típusa: `int(eger)`, értéke 9 és 12 között lehet
- **távolság**: a diák lakásának az iskolától mért távolsága méterben, típusa: `int(eger)`

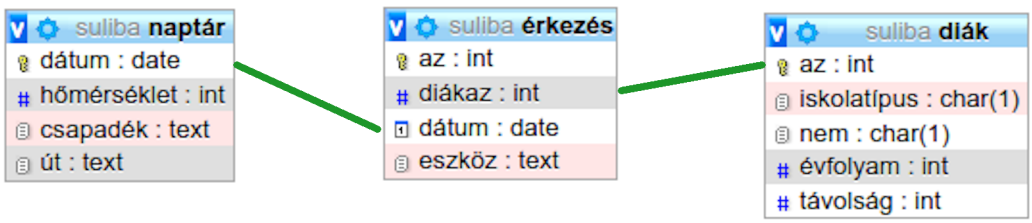
érkezés tábla:

- **az**:az iskolába érkezés azonosítója, típusa: `int(eger)`, ez a kulcs
- **diákaz**: a diák azonosítója, típusa: `int(eger)`
- **dátum**: az érkezés napja, típusa: `date`
- **eszköz**: az adott napon igénybe vett eszköz neve, típusa: `text`, értéke: *gyalog*, *kerékpár*, *motor*, *autó*, *tömegközlekedés*

naptár tábla:

- **dátum**: az adott nap dátuma, típusa: `date` ez a kulcs
- **hőmérséklet**: az adott napon mekkora volt az áltaghőmérséklet a városban, típusa: `int(eger)`
- **csapadék**: az adott nap reggelén milyen csapadék volt, csapadékmentes esetben *NULL* (üres), típusa: `text`
- **út**: az adott nap reggelén milyen volt az út, típusa: `text`, értéke például: *száraz*, *jeges* stb.

Az alábbi ábra mutatja a három tábla közötti kapcsolatot:



Készítsd el a következő feladatok megoldását! Ügyelj arra, hogy a megoldásban pontosan a kívánt mezők, kifejezések szerepeljenek, felesleges mezőt ne jeleníts meg! A megodásokat a zárójelben lévő néven mentse el egy-egy szöveges állományba!

1. Készíts lekérdezést, amely a csapadékmentes januári napokat listázza! (`1nincs.sql`)
2. Készíts lekérdezést, amely megadja a legmesszebb lakó diáknak az azonosítóját és azt a távolságot kilométerben, amennyire lakik az iskolától! (`2messze.sql`)
3. A vizsgált időszakban pontosan 38 tanítási nap volt. Add meg lekérdezés segítségével azon diákok azonosítóit, akik egyetlen napos sem mulasztottak! (`3mindig.sql`)
4. Készíts lekérdezést, amely megadja azon diákok azonosítóját, akik havas vagy jeges útviszonyok között is kerékpárral mentek iskolába! Mindegyik azonosító pontosan egyszer szerepeljen! (`4kerekpar.sql`)
5. A 119-es és 186-as azonosítójú diákok majdnem szomszédok, osztálytársak és ráadásul barátok. Készíts lekérdezést, amely megadja azokat a dátumokat, amikor mindketten voltak iskolában és ugyanolyan közlekedési eszközzel jutottak oda! (`5ugyanugy.sql`)

11.9. Csoportok

Ennek a feladatnak a 2006. őszi középszintű informatika érettségi azonos című feladat az alapja.

A feladat forráskönyvtára: `/public/adatbazis/csoportok/`

A phpPGAdmin forrásállománya: `csoportok.pg.sql`

A XAMPP forrásállománya: `csoportok.my.sql`

A forrásállomány beimportálása eredményeként egy osztály névsorát, különböző tanulócsoportokba történő beosztását és néhány egyéb adatát tartalmazó adattáblát kapunk **adatok** néven.

A tábla mezőjének jelentése a következő:

- **tanulókód**: a tanulók egyedi azonosítója, ez a tábla kulcsa.
- **név**: a diák neve (text)
- **mat**: matematika és informatika szerinti csoportbeosztás (text)
- **angol**: angol csoportok szerinti besorolás, a szint és a tanár megjelölésével (text)
- **mnyelv**: választott második idegen nyelv (text)
- **tesi**: a diák neve, ami alapján a testnevelés csoportbontás történik (char(1))
- **családszám**: a családban együttlakók száma (integer)
- **tesószám**: testvérek száma (smallint)

Oldd meg az alábbi feladatokat! A keletkező SQL utasításokat a feladat végén, zárójelben levő nevű szöveges állományba mentsd el!

1. Lekérdezésben gyűjtsd ki azon diákok nevét (csak a nevét), akiknek több, mint 1 testvérük van! (`1sok.sql`)
2. Viszonylag kevés azon családoknak a száma, ahol az együttlakók száma és a testvérek száma között nem három a különbség. Lekérdezéssel add meg a számukat! (`2spec.sql`)
3. Az angol nyelvet jövőre a 4. csoportban nem Kis tanár úr tanítja, hanem Kun tanárnő, ezért a „4. Kis” bejegyzést le kell cserélni a „4. Kun” szövegre. Készíts adatmódosító utasítást erre a célra és hajtsd is azt végre! (`3update.sql`)
4. Készíts lekérdezést, amely kiírja Hát Izsák minden adatát! (`4izsak.sql`)
5. Hát Izsák földkörüli út miatt hosszan fog hiányozni. Add meg lekérdezésben, de az előző lekérdezés alkérdésben való felhasználásával azon tanulók nevét, akik vele minden csoportban – még testnevelés órán is – együtt járnak! A listában ne jelenjen meg Izsák neve! (`5kapocs.sql`)
6. Bekre – a keresztneve nem derül ki a bejelentésből – hosszú időn át fog hiányozni. Készíts az előző lekérdezéshez hasonló lekérdezést `6hirvivo.sql` néven!

11.10. Feltalálók[11]

Magyar találmányok, technikai alkotások neveinek gyűjteménye és feltalálók adatai állnak rendelkezésre a `talalmany.txt`, `kutato.txt` és a `talalmanya.txt` állományokban. Ezekből készült az SQLite Browserben megnyitható `feltalalok.sqlite` állomány.

Amennyiben MS Accessben kívánod megoldani a feladatot, akkor az alábbiakat figyelembe véve készíts egy `feltalalok.accdb` nevű adatbázist és olvasd be az adatokat a három .txt állományból, beállítva a megfelelő adatformátumokat és kulcsokat! A táblákban ne vegyél fel új mezőt! SQLite használata esetén elegendő a már létező feltalalok.sqlite állományt megnyitni.

11 Ennek a feladatnak az eredetije 2007. őszén szerepelt az emelt szintű informatika érettségin.

(A feladat később PostgreSQL és MySQL forrásadatbázisokkal is el lesz látva.)
Az adatbázis adattáblái a következőképpen épülnek fel:
talalmany(*tkod*, *talnev*)

- *tkod*: A találmány azonosítója (szám), ez a kulcs
- *talnev*: A találmány neve (szöveg)

kutato(*fkod*, *nev*, *szul*, *meghal*)

- *fkod*: A kutató vagy feltaláló azonosítója (szám), a tábla kulcs
- *nev*: A kutató neve, vezeték- és utónév sorrendben (szöveg)
- *szul*: A kutató születési éve (szám)
- *meghal*: A kutató halálának éve – a 2007-ben még életben levők esetén NULL az értéke (szám)

talalmanya(*tkod*, *fkod*)

- *tkod*: A találmány azonosítója (szám) külső kulcs a talalmany(tkod)-ra
- *fkod*: A kutató azonosítója (szám) külső kulcs a kutato(fkod)-ra.
- Mivel ez a tábla a két másik tábla által reprezentált egyedhalmazt összekötő kapcsolatnak felel meg, így a két mezője együtt alkotja a tábla kulcsát.

Csak akkor szerepel egy találmány azonosítója a **talalmanya** táblában, ha a feltaláló neve ismert.

Készítsd el a következő feladatokat megoldó lekérdezéseket! MS Access esetén az adatbázison belül kell a feladat végén, zárójelben megadott néven menteni a lekérdezést. SQLite használata esetén minden lekérdezést egy-egy, a feladat végén megadott nevű, `sql` kiterjesztésű szöveges állományba kell elmenteni azt, beadni ezeket az állományokat kell (tehát például az első lekérdezés az `1motor.sql` nevű állományba kerüljön).

1. Listázd ki ábécérendben azoknak a találmányoknak a nevét, amelyek nevében szerepel a „*motor*” szó! (`1motor`)
2. Írasd ki Forgó László találmányainak nevét! (`2forgo`)
3. Add meg, hogy ki volt a golyóstoll feltalálója és hány évig élt! (`3golyostoll`)
4. Sorold fel azoknak a találmányoknak a nevét, amelyeknek a feltalálója nincs megadva az adatbázisban! (`4nevtelen`)
5. Milyen találmányaik voltak azoknak a kutatóknak, akik a XIX. század első felében (1801 és 1850) között – a határokat is beleszámítva – éltek? A kutatók és a találmányok nevét add meg a lekérdezés eredményeként! (`5felszazad`)[12]
6. Sorold fel azoknak a kutatóknak a nevét és a találmányaik számát, akik legalább 3 kutatási eredménnyel szerepelnek az adatbázisban! (`6kutszam`)
7. A „*transzformátor*” feltalálóinak – a „*transzformátor*”-on kívül – milyen más találmányaik vannak az adatbázisban! Minden találmány csak egyszer szerepeljen a listában! (`7transzformator`)
8. **Csak erős idegzetűeknek**: Listázd ki azokat a feltalálókat és találmányaik nevét, akiknek a vezetékneve szerepel a találmány nevében! (`8nevado`)

Segítségül néhány SQL szövegkezelő függvény:
LTRIM(*szöveg*, *hossz*): a *hossz* argumentumban megadott számú karaktert adja balról
RTRIM(*szöveg*, *hossz*): a *hossz* argumentumban megadott számú karaktert adja jobbról
LENGTH(*szöveg*): a *szöveg* karaktereinek számát adja
InSTR(*szöveg1*, *szöveg2*): *szöveg1*-ben a *szöveg2* hányadik karakternél kezdődik

12 A lekérdezés eredményéből annak is ki kell derülnie, hogy Ganz Ábrahám feltalálta a Ganz-gyárat ;-)

TRIM(*szöveg, kezdet, hossz*): a *szöveg*-ből *kezdet* pozíciótól *hossz* darab karatert ad eredményül.

MS Access esetén a fenti függvények nevei a következők:

- LTRIM = LEFT
- RTRIM = RIGHT
- LENGTH = LEN
- TRIM = MID

A feladat eredetileg az érettségin 30 pontot ért.

11.11. Csöpi-filmek[13]

A Bujtor István nevével fémjelzett Csöpi-filmek mind a mai napig népszerűek. Ez a feladat az elkészült hét Csöpi-film adatait dolgozza fel. A szükséges adatok megtalálhatóak a `film.txt`, a `stab.txt` és a `szereplo.txt` állományokban. Az ezekből készült SQLite adatbázis a `csopi.sqlite` állományban található meg.

Amennyiben a feladatot a MS Access használatával oldod meg, akkor készíts új adatbázist `csopi.accdb` néven! A mellékelt három – tabulátorokkal tagolt, UTF-8 kódolású – CSV állomány tartalmát importáld az adatbázisba az állománynévvel azonos nevű adattáblába! Az állományok első sora tartalmazza a mezőneveket. A létrehozás során állítsd be a megfelelő típusokat és az elsődleges kulcsokat! A `csopi.sqlite` állomány a már importált adatokat tartalmazó adatbázist tartalmazza, azt elég az SQLite Browser segítségével megnyiti a feladat megoldásához.

A táblák:

film(id, cím, ev, hossz)

- *id*: A film azonosítója (szám), ez a kulcs
- *cim*: A film címe (szöveg)
- *ev*: A film készítésének éve (szam)
- *hossz*: A film hossza percekben megadva (szam)

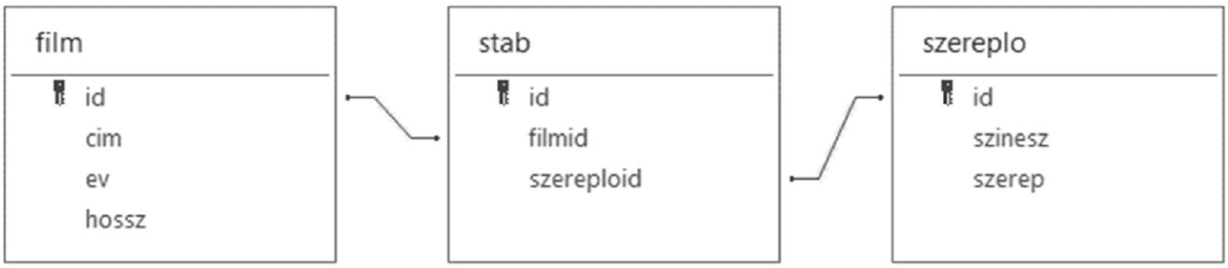
szereplo(id, szinesz, szerep)

- *id*: A szereplő azonosítója (szám), ez a kulcs
- *szinesz*: Az adott szerepben játszó színész neve (szöveg) Ha egy színész más filmekben eltérő szerepet játszott, akkor a neve többször előfordul. Az adattábla nem tartalmaz azonos nevű, különböző színészeket.
- *szerep*: Az adott színész által játszott szerep neve, leírása (szöveg) Egy filmnben egy színész nem játszott több szerepet.

stab(id, filmid, szereploid)

- *id*: Azonosító (szám), ez a kulcs
- *filmid*: A film azonosítója (szám) külső kulcs a film tábla id mezőjére.
- *szereploid*: Az adott film szereplőjének azonosítója (szám) külső kulcs a szerep tábla id mezőjére.

13 A feladat eredetije a 2023. őszi középszintű digitális kultúra érettségin szerepelt. A feladat úgy lett átalakítva, hogy SQLite Browserben is megoldható legyen.



A következő feladatok megoldásánál a lekérdezéseket a zárójelben olvasható néven mentsd el! MS Access esetén az adatbázist és a lekérdezéseket is tartalmazó állományt kell beadni, míg SQLite használata esetén az egyes lekérdezéseket tartalmazó szöveges állományokat (pl. az első feladat megoldását tartalmazó 1onmaga.sql állományt), ebben az esetben a lekérdezések mind külön szöveges állományba kerüljenek, amelynek kiterjesztése sql (lásd a zárójelbeli példát)!

Ügyelj arra, hogy a megoldásban pontosan a kívánt mezők szerepeljenek!

1. Néhány szereplő a filmekben önmagát játszotta, vagyis a színész neve és a szerep azonosak. Listázd ki lekérdezés segítségével ezeket a színészeket! Csak a színész neve jelenjen meg! (`1onmaga`)
2. Hány órára lenne szükségünk, ha az összes Csöpi-filmet egyhuzamban szeretnénk megnézni? A választ add meg lekérdezés segítségével? (`2ora`)
3. Add meg lekérdezés segítségével, hogy mely színészek szerepeltek és milyen szerepet játszottak a „*Hamis a baba*” című filmben! A lekérdezésben más mező ne jelenjen meg! (`3hamis`)
4. Kik azok a színészek, akik több szerepet is játszottak a Csöpi-filmekben? Lekérdezés segítségével jelenítsd meg a színészek nevét és azt, hogy hány szerepről van szó, a szerepek száma szerint csökkenő sorrendben! Minden színész neve csak egyszer jelenjen meg! (`4tobbszor.sql`)
5. Néhány színész mind a hét Csöpi-filmben szerepelt és mindig ugyanazt a szerepet játszotta. Add meg lekérdezés segítségével ezeknek a színészeknek a nevét és az általuk alakított szerepet! (`5mindig`)

A feladat eredetileg 15 pontot ért az érettségin.

11.12. Űrhajózás

A múlt század második felében az ember meghódította a világűrt. A legtöbb ember ismeri Gagarin, Armstrong és Farkas Bertalan nevét. Természetesen rajtuk kívül is sokan jártak az űrben. Az adatbázis az adatgyűjtéskor már befejezett küldetések és az űrhajósok adatait tartalmazza.

Amennyiben a feladatot Accessben oldod meg, akkor készíts új adatbázist `urhajozas` néven! A mellékelt három – tabulárotokkal tagolt, UTF-8 kódolású – szöveges állományt (`urhajos.txt`, `repules.txt`, `kuldetes.txt`) importáld az adatbázisba az állomány nevével azonos nevű adattáblát létrehozva! Az állományok első sora a mezőneveket tartalmazza. A létrehozás során állítsd be a megfelelő típusokat és az elsődleges kulcsot!

SQLite Browser használata esetén a fentiek helyett nyisd meg az `urhajozas.sqlite` állományt, amely már tartalmazza az adatbázist! Hasonlóan létezik[14] adatbázis a PostgreSQL adatbázisszerverbe beimportálhatóan az `urhajozas.pg.sql` néven, illetve az XAMPP-ba beimportálható `urhajozas.my.sql` néven.

14 Még nem létezik, de fog…

Az adatbázis táblái a következők:

urhajos(*id*, *nev*, *orszag*, *nem*, *szulev*, *urido*)

- *id*: Az *űrhajós* azonosítója (szám), ez a kulcs.
- *nev*: Az űrhajós neve (szöveg). Feltételezheted, hogy a nevek egyediek.
- *orszag*: Az űrhajós által képviselt ország az első kilövéskör. Három betűs szöveg.
- *nem*: Az űrhajós személy neme (szöveg). Értéke a férfiak esetén F, nőknél N.
- *szulev*: Az űrhajós születési éve (szám).
- *urido*: Az űrhajós által az űrben töltött összes idő (szöveg). Az első karaktere minden esetben a T betű, utána 3 karakter a napokat, 2 az órákat, 2 a perceket jelöli. A számokat kettőspont választja el egymástól.

repules(*urhajosid*, *kuldetesid*)

- *urhajosid*: Az űrhajós azonosítja, a kulcs része, egyben külső kulcs az urhajos tábla id mezőjére.
- *kuldetesid*: A küldetés azonosítója, a kulcs része, egyben külső kulcs a kuldetes tábla id mezőjére.

kuldetes(*id*, *megnevezes*, *kezdet*, *veg*)

- id*:* A küldetés azonosítója (szám), ez a kulcs.
- *megnevezes*: A küldetés hivatalos neve (szöveg).
- *kezdet*: A küldetés kezdetének dátuma (dátum).
- *veg*: A küldetés befejezésének dátuma (dátum).

A táblák között az alábbi ábrán látható kapcsolat áll fenn:

A következő feladatok megoldásánál a lekérdezéseket a zárójelben olvasható néven mentsd el – Access esetén az adatbázisba lekérdezésként, egyébként – `.sql` kiterjesztéssel egy-egy szöveges állományba! Ügyelj arra, hogy a lekérdezésben pontosan a kívánt mezők szerepeljenek, felesleges mezőt ne jeleníts meg!

1. Lekérdezés segítségével határozd meg, hogy a nők közül ki töltötte a legtöbb időt az űrben és mennyit! Jelenítsd meg az űrhajós nevét és az urido mező értékét! (`1legtobbido`)
2. Előfordult, hogy egy küldetés résztvevői az újévet az űrben köszötötték. Készíts lekérdezést, amely megadja ezen küldetések megnevezését és azt, hogy milyen hosszúak voltak, azaz hány napig tartottak! (`2szilveszter`)
3. Készíts lekérdezést, amelyik megadja, hogy a több űrrepülésen részt vevő űrhajósok hány éves korukban kezdték az első és hány éves korukban az utolsó küldetésüket! Jelenítsd meg az űrhajós nevét és a két életkort! (`3eletkor`)
4. Lekérdezés segítségével határozd meg, hogy az egyes országoknak hány űrhajósa szerepel az adatbázisban! Az országok azonosítóját és az adott ország űrhajósainak számát az űrhajósok száma szerint csökkenő sorrendben jelenítsd meg! (`4urhajosszam`)

5. Határozd meg, hogy ország képviseletében repültek űrhajósok! Ehhez adott az 5alap.sql állományban az a lekérdezés, amelynek a kiegészítésével a feladatot meg kell oldanod: az alábbi lekérdezésben a hiányzó helyeket kell megadnod. (`5orszagszam`)

```
SELECT Count(allekerdezes.orszag)
FROM (SELECT … FROM …) AS allekerdezes;
```

6. Készíts lekérdezést, amelyben felsorold azoknak a küldetéseknek a nevét, amelyben a legénység tagjai között férfi és nő is volt! (`6ferfino`)
7. Készíts lekérdezést, amellyel kigyűjtöd a 20. század utolsó évtizedében (1991–2000) megkezdett küldetéseken milyen legénység vett részt! A lekérdezésben szerepeljen küldetésenként, azonbelül időrendben és azon belül is névsorba rendezve a küldetés neve, a kezdete, az legénység tagja, országa és a nemét jelző F vagy N betű. Az oszlopok neve rendre a „*küldetés neve*”, „*kezdete*”, „*legénység*” „*ország*” „*nem*” szöveg legyen! (`7legenyseg`)

Tárgymutató