MINISTERUL EDUCAȚIEI ȘI CERCETĂRII

Cristina Elena Anton • Bartus Emőke • Andreea Floriana Bontea
Budai Edit • Budai István • Isabela-Patricia Coman
Melinda-Emilia Coriteac • Carmen-Diana Cosman
Fejér Magdalena • Violeta Marilena Grecea • Ignat Judit-Anna
Jakab Irma-Tünde • Sleghel Izabella • Amelia Stoian
Miruna-Andreea Zăvelcă • Zsigó Tamás

Informatika

9

magyar nyelven a magyar tannyelvű iskolák/ osztályok számára, 9. osztály, szakosított tanterv, elméleti tagozat, reál szakirány, matematika-informatika profil

Tankönyv a IX. osztály számára



SIGMA

Ez a tankönyv a „Ministerul Educației și Cercetării” tulajdonát képezi.

Ez a tankönyvtervezet a 6930/19.12.2025 számú minisztériumi rendelettel jóváhagyott iskolai tantervvel összhangban készült

A **119**-es telefonszám – országos egységes telefonszám a gyermekbántalmazási esetekhez.

A **116.111**-es telefonszám – gyermekek számára létrehozott segélyvonal, a gyermekek jogainak védelméről és előmozdításáról szóló, újraközölt 272/2004. sz. törvénynek, valamint az azt követő módosításoknak és kiegészítéseknek megfelelően.

Ministerul Educației și Cercetării

Cristina Elena Anton • Bartus Emőke • Andreea Floriana Bontea
Budai Edit • Budai István • Isabela-Patricia Coman
Melinda-Emilia Coriteac • Carmen-Diana Cosman
Fejér Magdalena • Violeta Marilena Grecea • Ignat Judit-Anna
Jakab Irma-Tünde • Sleghel Izabella • Amelia Stoian
Miruna-Andreea Zăvelcă • Zsigó Tamás

Informatika

magyar nyelven a magyar tannyelvű iskolák/ osztályok számára, 9. osztály, szakosított tanterv, elméleti tagozat, reál szakirány, matematika-informatika profil

Tankönyv a IX. osztály számára

SIGMA
București, 2026

A tankönyvet a(z) 5779/09.10.2026 számú miniszteri rendelet alapján hagyták jóvá
A tanulók a tankönyvet nyomtatott és digitális változatban, ingyenesen kapják meg, és a 2026–2027-es tanévtől kezdődően négy tanéven át használható.
..... Megyei Tanfelügyelőség
Iskola / Főgimnázium / Líceum .....

EZT A TANKÖNYVET HASZNÁLTÁK:

ÉV	Tanuló neve	Osztály	Tanév	A tankönyv állapota*			
				Nyomtatott példány		Digitális példány	
				átvételkor	átadáskor	átvételkor	átadáskor
1							
2							
3							
4							

* A tankönyv állapotának meghatározásához a következő kifejezések használhatóak: ***új, jó, rendes, elhanyagolt, sérült***.

A pedagógusok ellenőrzik a fenti táblázatban megjelenő adatok helyességét.
A tanulók nem írhatnak a tankönyvbe.

Tudományos referensek:
- ***Angela Blaga***: tanár, I. fokozatú oktatói minősítés, mérnöki és menedzsmenttudományok doktora, az „Alexandru Papiu Ilarian” Főgimnázium tanára, Marosvásárhely, Maros megye.
- ***Horea-Gavril Oros***: egyetemi adjunktus, PhD, a Nagyváradi Egyetem Informatikai és Természettudományi Karának Matematika és Informatika Tanszékén.
- ***Delia Tușe***: egyetemi adjunktus, informatika PhD, a Nagyváradi Egyetem Informatikai és Természettudományi Karának Matematika és Informatika Tanszékén.

Szerkesztő: Zsigó Tamás
Grafika és szövegszerkesztés: Zsigó Tamás
Borító: Adrian-Viorel Rașca
Illusztrációk: Canva, Magnific

ISBN: 978-606-727-699-2

© 2026 – SIGMA KIADÓ
A kiadás minden joga a SIGMA kiadót illeti.
A SIGMA kiadó írásos engedélye nélkül a nyomtatott és digitális anyag egyetlen része sem sokszorosítható.

SIGMA Kiadó

Székhely:
Str. Nicolae Cartojan, nr. 11-13-15, sector 2, București, cod poștal 023815
Tel.: 0758 044 829; 0723 541 020
E-mail: office@editurasigma.ro
Web: www.editurasigma.ro

Terjesztés:
Str. Nicolae Cartojan, nr. 11-13-15, sector 2, București, cod poștal 023815
Tel.: 0758 044 825; 0758 044 828
E-mail: comenzi@editurasigma.ro

Előszó

Ez a tankönyv azzal a céllal készült, hogy támogassa az informatika területén elengedhetetlen számítógépes, algoritmikus és kreatív gondolkodásmód kialakítását. Célja, hogy segítsen nektek helyes, hatékony és jól szervezett megoldások kidolgozásában, kihasználva a Python nyelv és a modern programozási eszközök adta lehetőségeket.

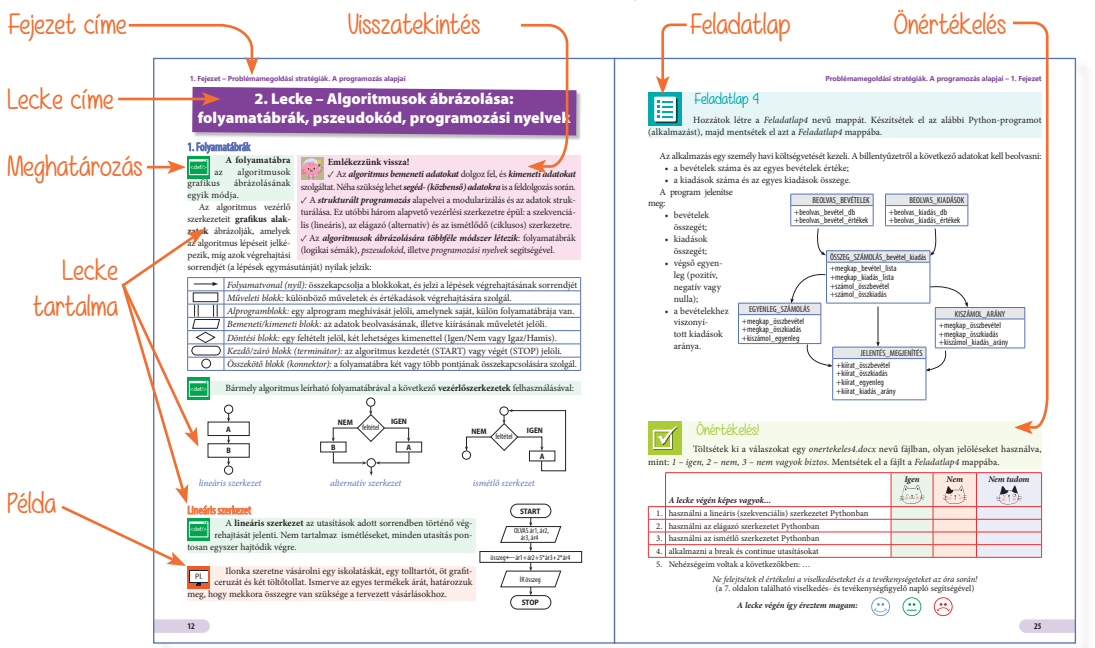
A kiadvány hat fejezetből áll: Problémamegoldási stratégiák - A programozás alapjai, Numerikus adatok feldolgozása, Modularitás és objektumorientált programozás, Grafikus interfészek és szöveges fájlok, Az adatok listákba szervezése, illetve Haladó rendezési stratégiák és algoritmusok hatékonysága.

A leckék úgy vannak felépítve, hogy elősegítsék a felfedezésen, példákon és gyakorlati feladatokon alapuló tanulást. Az egyes leckék végén található feladatlapok lehetőséget nyújtanak arra, hogy a tanultakat konkrét helyzetekben alkalmazzátok, az önértékelési feladatok pedig segítenek nyomon követni az előrehaladásotokat, és felelősségteljes hozzáállást kialakítani a saját tanulásotok iránt.

Arra bátorítunk benneteket, hogy ezt a tankönyvet a kíváncsiság, a felfedezés és a személyes fejlődés kiindulópontjaként használjátok. Írjatok kódot gondosan és kreatívan – minden megoldott feladat egy újabb alkalom arra, hogy tisztábban gondolkodjatok.

A szerzők

A tankönyv bemutatása



A tankönyben használt egyéb grafikai elemek

- Megjegyzés
- Tudtad-e?
- Jegyezd meg!
- Kutass!
- Ellenőrizd a tudásod
- Portfólió
- Projekt

TARTALOMJEGYZÉK

Előszó. A tankönyv bemutatása 3
Általános és specifikus kompetenciák 5
Útmutató a munka megszervezéséhez a feladatlapok/alkalmazások tekintetében 6
Viselkedés és tevékenységfigyelő napló 7
Év eleji tudásfelmérő 8

1. Fejezet – Problémamegoldási stratégiák. A programozás alapjai (1.3.; 1.4.; 2.1.; 2.3.; 2.4.; 3.3.; 3.4.; 4.1.; 4.3.; 4.4.; 5.3.; 6.3.)
1. Lecke *– Egy program elkészítésének szakaszai: az elemzéstől és tervezéstől a tesztelésig és hibakeresésig* ... 9
Feladatlap 1 11
2. Lecke *– Algoritmusok ábrázolása: folyamatábrák, pszeudokód, programozási nyelvek* 12
Feladatlap 2 17
3. Lecke *– Egy Python-program felépítése. Változók, adattípusok és kifejezések* 19
Feladatlap 3 22
4. Lecke *– Vezérlő szerkezetek: szekvenciális (lineáris), elágazó és ismétlő szerkezet* 23
Feladatlap 4 25
1. Fejezetvégi felmérés 26

2. Fejezet – Problémamegoldási stratégiák. Numerikus adatok feldolgozása (1.2.; 2.2.; 3.2.; 4.2.; 5.2.; 6.2.)
5. Lecke *– Egy szám számjegyeivel végezhető műveletek* .. 28
Feladatlap 5 29
6. Lecke *– Oszthatósági algoritmusok: Egy szám osztói; Prímszám-ellenőrzés* 32
Feladatlap 6 34
7. Lecke *– Prímtényezőkre bontás; LNKO, LKKT (Euklidész)* 36
Feladatlap 7 39
8. Lecke *– Számrendszerek* 42
Feladatlap 8 44
2. Fejezetvégi felmérés 46

3. Fejezet – Moduláris felépítés és objektum-orientált programozás (1.5.; 2.5.; 3.5.; 4.5.; 5.5.; 6.5.)
9. Lecke *– Alprogramok: fejléc, törzs, paraméterek* 48
Feladatlap 9 51
10. Lecke *– A végrehajtási mechanizmus: lokális és globális változók* 52
Feladatlap 10 55
11. Lecke *– Beépített függvények kollekciókhoz (Python)* 56
Feladatlap 11 59
12. Lecke *– Bevezetés az OOP-ba: osztályok, objektumok és példányosítás* 61
Feladatlap 12 65
3. Fejezetvégi felmérés 67

4. Fejezet – Grafikus interfészek és szöveges fájlok (1.4.; 2.3.; 2.4.; 3.3.; 3.4.; 4.4.; 5.4.; 6.4.)
13. Lecke *– A Tkinter könyvtár: ablakok létrehozása és az elrendezéskezelés* 69
Feladatlap 13 72
14. Lecke *– Interaktív widgetek: gombok, címkék és szövegmezők* 73
Feladatlap 14 79
15. Lecke *– Eseménykezelés grafikus felületeken (üzenetek megjelenítése kattintásra)* 80
Feladatlap 15 83
16. Lecke *– Szöveges fájlok (állományok): megnyitás, bezárás és adatátvitel* 85
Feladatlap 16 88
17. Lecke *– A TextIOWrapper osztály: a hatékony olvasás és írás módszerei* 90
Feladatlap 17 93
4. Fejezetvégi felmérés 96

5. Fejezet – Az adatok fogalmi szervezése (listák) (1.1.; 1.4., 2.1.; 2.4., 3.1.; 3.4., 4.1., 4.4.; 5.1.; 6.1.)
18. Lecke *– A lineáris fogalmi modell: jellemzők, közvetlen és szekvenciális hozzáférés* 98
Feladatlap 18 101
19. Lecke *– Gyakorisági/ előfordulási listák, a verem (LIFO) és a sor (FIFO)* 103
Feladatlap 19 105
20. Lecke *– A Python list osztálya: sajátos operátorok és alapvető metódusok* 107
Feladatlap 20 111
21. Lecke *– A list osztály speciális metódusai és az elemek bejárása* 112
Feladatlap 21 116
5. Fejezetvégi felmérés 118

6. Fejezet – Haladó megoldási stratégiák. (Egy lista elemeinek szisztematikus előállítási módszerei) (1.2.; 2.1.; 2.2.; 3.2.; 3.4.; 4.2.; 5.2.; 6.2.)
22. Lecke *– Az algoritmusok hatékonysága: végrehajtási idő, memória és az O(n) jelölés* 120
Feladatlap 22 123
23. Lecke *– Szisztematikus előállítás: előző tagok alapján felépülő sorozatok és adott tulajdonságú szekvenciák* ... 125
Feladatlap 23 128
24. Lecke *– Rendezési módszerek: a minimumkiválasztásos rendezés* 130
Feladatlap 24 132
25. Lecke *– Rendezési módszerek: a buborékrendezés* .. 134
Feladatlap 25 137
26. Lecke *– Rendezési módszerek: rendezés frekvenciatömb segítségével* 139
Feladatlap 26 142
6. Fejezetvégi felmérés 144
Évzáró felmérés 146
Útmutatók és megoldások 148

Általános és specifikus kompetenciák

ÁK1 – Ismeri a szoftverfejlesztés elméleti és működési modelljeinek főbb jellemzőit a programozás alapjainak megértése érdekében

- **SK 1.1.** Meghatározza az adatok szervezésének főbb jellemzőit alapvető fogalmi modellek – egyszerű adatok vagy listák – keretében, az adatok feldolgozás céljából történő strukturálása és elérése érdekében
- **SK 1.2.** Meghatározza a problématípusokra specializált algoritmusok sajátosságait, jellemzőit és lépéseit, hogy azokat számok algoritmikus feldolgozásához, rendezéshez vagy értékek sorozatának szisztematikus generálásához felhasználhassa
- **SK 1.3.** Meghatározza a problémamegoldási stratégiák sajátosságait, jellemzőit és lépéseit az algoritmusok moduláris tervezése révén.
- **SK 1.4.** Ismeri a programozási nyelv legfontosabb elemeit, képes azokat egyszerű adatok és listák hatékony feldolgozására alkalmazni.
- **SK 1.5.** Meghatározza a szintaxis elemeit és az alprogramok, valamint a szokásos műveletekhez előre definiált alprogramok definíciójának és hívásának összetevőit, hogy azokat a programozási nyelven történő algoritmusok megvalósításához felhasználhassa

ÁK2 – Megmagyarázza a szoftverfejlesztés fogalmi és operatív modelljeinek alapelveit, hogy logikusan alátámassza az informatikai megoldások tervezését és megvalósítását

- **SK 2.1.** Megmagyarázza az adatok szervezésének alapelveit az alapvető fogalmi modellek keretében – egyszerű adatok vagy listák –, hogy azokat hatékonyan kezelhesse
- **SK 2.2.** Ismerteti a kérdéskörökre szakosodott algoritmusok lépéseit, hogy azokat számok algoritmikus feldolgozásában, rendezésben vagy értékek sorozatának szisztematikus generálásában felhasználhassa
- **SK 2.3.** Megmagyarázza a problémamegoldási stratégiák lépéseit az algoritmusok moduláris tervezése alapján
- **SK 2.4.** Ismerteti az alapvető modellekbe szervezett adatok feldolgozásához – egyszerű adatok vagy listák – illetve az informatikai problémák hatékony megoldása érdekében a programok közötti adatátvitelhez használt programozási nyelv főbb elemeinek szerepét
- **SK 2.5.** Megmagyarázza az alprogramok és a szokásos műveletekre előre definiált alprogramok végrehajtási mechanizmusát, hogy azokat a programozási nyelven történő algoritmusok megvalósításában felhasználhassa

ÁK3 – A szoftverfejlesztés elméleti és operatív modelljeit alkalmazza a működőképes és hatékony informatikai megoldások kidolgozása érdekében

- **SK 3.1.** Az adatok szervezésének és feldolgozásának módszereit alapvető elméleti modellek – egyszerű adatok vagy listák – keretében alkalmazza az adatkezelési problémák megoldására
- **SK 3.2.** Problématípusokra specializált algoritmusokat alkalmaz a számok algoritmikus feldolgozásában, a rendezésben vagy értékek sorozatainak szisztematikus generálásában
- **SK 3.3.** Problémamegoldási stratégiákat alkalmaz az algoritmusok moduláris tervezésén keresztül
- **SK 3.4.** Használja az alapvető modellekbe szervezett adatok feldolgozásához – egyszerű adatok vagy listák – illetve az adatok programok közötti továbbításához használt programozási nyelv főbb elemeit az informatikai feladatok hatékony megoldása érdekében
- **SK 3.5.** Használja a szintaxis elemeit és az alprogramok és az általános műveletekre előre definiált alprogramok definiálásának és meghívásának összetevőit az algoritmusok programozási nyelven történő megvalósításában

ÁK4 – Elemzi a szoftverfejlesztés elméleti és működési modelljeinek jellemzőit és alkalmazhatóságát, hogy az adott informatikai kontextusnak megfelelően kiválaszthassa a legmegfelelőbb megoldásokat

- **SK 4.1.** Elemezi az alapvető fogalmi modellek – egyszerű adatok vagy listák – jellemzőit, feldolgozási módját, előnyeit és hátrányait, hogy konkrét problémák megoldásához a megfelelő struktúrát válassza
- **SK 4.2.** Elemezi a viselkedést, az alkalmazhatóságot, az előnyöket és hátrányokat bizonyos problématípusokra specializált algoritmusok használatával a számok feldolgozása, a rendezés vagy értékek sorozatainak szisztematikus generálása céljából
- **SK 4.3.** Elemezi a problémamegoldási stratégiák alkalmazhatóságát, előnyeit és hátrányait az algoritmusok moduláris tervezése révén
- **SK 4.4.** Elemezi a programozási nyelv egyes elemeinek alkalmazhatóságát, előnyeit és hátrányait annak érdekében, hogy meghatározza a megfelelő megoldást az alapvető modellekbe szervezett adatok – egyszerű adatok vagy listák – feldolgozásához, illetve az adatok programok közötti továbbításához, az informatikai problémák hatékony megoldása céljából
- **SK 4.5.** Elemezi az alprogramok használatának alkalmazhatóságát, előnyeit és hátrányait az algoritmusok programozási nyelven történő megvalósításában

ÁK5 – Értékeli az informatikai megoldások helyességét és hatékonyságát a működés optimalizálása és biztosítása érdekében különböző felhasználási esetekben

- **SK 5.1.** Megindokolja az alapvető fogalmi modellek – egyszerű adatok vagy listák – és azok feldolgozási módjainak kiválasztását konkrét informatikai problémák megoldása érdekében
- **SK 5.2.** Értékeli a számok feldolgozására, rendezésére vagy értékek sorozatainak szisztematikus generálására szolgáló, problématípusokra specializált algoritmusokkal működő megoldások helyességét és hatékonyságát
- **SK 5.3.** Elemzi az algoritmusok helyességét és hatékonyságát, amelyek az algoritmusok moduláris tervezésén alapuló problémamegoldási stratégiákat alkalmaznak
- **SK 5.4.** Elemzi azoknak az alkalmazásoknak a helyességét és hatékonyságát, amelyek a programozási nyelv specifikus elemeit használják az alapvető modellekben szervezett adatok – egyszerű adatok vagy listák – feldolgozására, illetve az adatok programok közötti továbbítására az informatikai kérdések hatékony megoldása érdekében
- **SK 5.5.** Megítéli a programok helyességét és hatékonyságát, amelyek alprogramokat használnak informatikai problémák megoldására

ÁK6 – Egyedi algoritmusokat és programokat dolgoz ki, hogy a követelményekhez igazodó, koherens informatikai megoldásokat hozzon létre

- **SK 6.1.** Egyedi algoritmusokat tervez, amelyek alapvető fogalmi modelleket – egyszerű adatokat vagy listákat – használnak az adatok hatékony szervezéséhez és feldolgozásához, alapvető algoritmusok alkalmazásával, problémák megoldása céljából
- **SK 6.2.** Olyan megoldásokat tervez, amelyek problématípusokra specializált algoritmusokat alkalmaznak a számok feldolgozására, a rendezésre vagy értékek sorozatainak szisztematikus generálására
- **SK 6.3.** Moduláris felépítésű, egyedi algoritmusokat tervez problémamegoldás céljából
- **SK 6.4.** Olyan alkalmazásokat valósít meg, amelyek testreszabott módon integrálnak programozási nyelvi elemeket az alapvető modellekbe szervezett adatok feldolgozásához – egyszerű adatok vagy listák –, illetve az adatok programok közötti továbbításához, funkcionális és hatékony megoldások megvalósítása céljából
- **SK 6.5.** Olyan programozási nyelven írt alkalmazásokat valósít meg, amelyek testreszabott alprogramokat integrálnak a kód hatékony modularizálása és szervezése érdekében az informatikai megoldásokban

Útmutató a munka megszervezéséhez a feladatlapok/alkalmazások tekintetében

A jelen tankönyv hat fejezetből és 26 leckéből áll, amelyek mindegyikének végén található egy feladatlap.

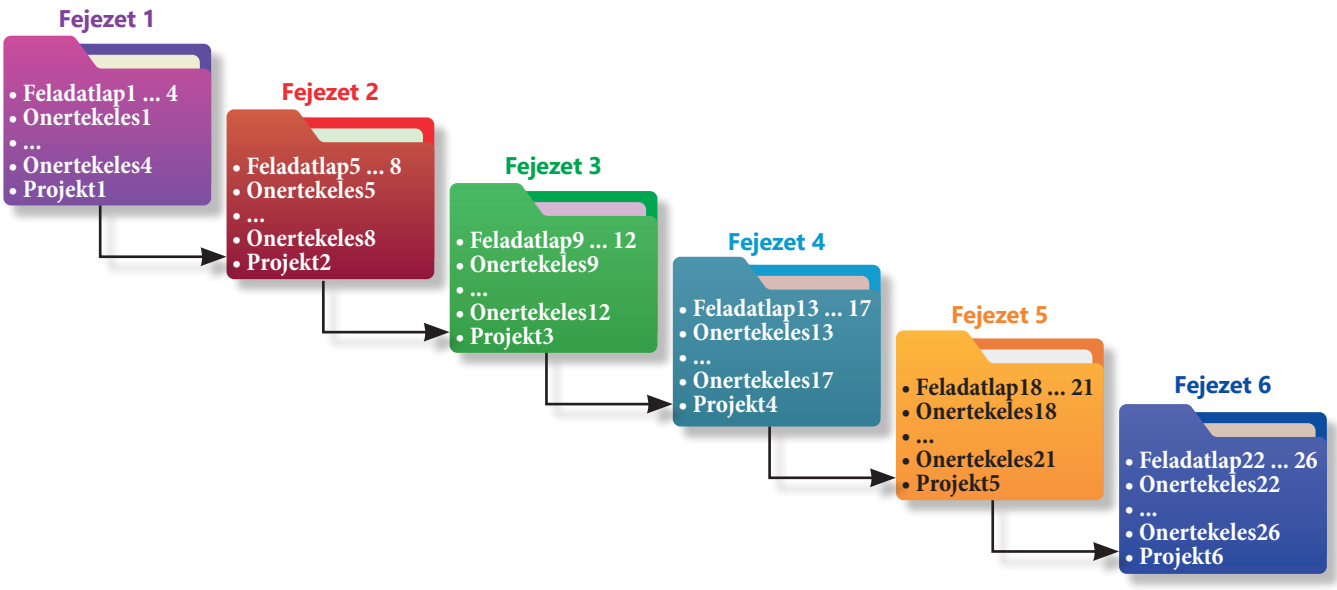
1. Hozzatok létre minden fejezethez egy ***FejezetSzám*** nevű mappát, ahol a *Szám* a fejezet sorszáma (1, 2, 3, 4, 5 vagy 6), egy olyan helyen a háttértárolón, ahol írási jogosultsággal rendelkeztek.

2. Minden feladatlaphoz hozzatok létre egy ***FeladatlapSzám*** nevű mappát, ahol a *Szám* a feladatlap sorszámát jelenti. Minden *FeladatlapSzám* nevű mappát a megfelelő *FejezetSzám* nevű mappába kell menteni. (A *Feladatlap1*, …, *Feladatlap4* mappákat a *Fejezet1* mappába, a *Feladatlap5*, …, *Feladatlap8* mappákat a *Fejezet2* mappába, a *Feladatlap9*, …, *Feladatlap12* mappákat a *Fejezet3* mappába, a *Feladatlap13*, …, *Feladatlap17* mappákat a *Fejezet4* mappába, a *Feladatlap18*, …, *Feladatlap21* mappákat a *Fejezet5* mappába, a *Feladatlap22*, …, *Feladatlap26* mappákat pedig a *Fejezet6* mappába menti. Ezekbe a mappákba mentsétek a feladatlapokban kért összes fájlt.

3. Minden óra végén önértékelési lap áll majd rendelkezésetekre. Az *1.*, …, *4.* önértékelési lapokat a *FeladatlapSzám* mappába kell menteni, ugyanazzal a számmal, a *Fejezet1* mappába; az *5.*, …, *8.* önértékelési lapokat a *Fejezet2* mappába; a *9.*, …, *12.* önértékelési lapokat a *Fejezet3* mappába; A *13.*, …, *17.* önértékelési lapokat a *Fejezet4* mappába; a *18.*, …, *21.* önértékelési lapokat a *Fejezet5* mappába; a *22.*, …, *26.* önértékelési lapokat a *Fejezet6* mappába.

4. Azokhoz a projektekhez (Fedezd fel/Vizsgáld meg), amelyeket a tanfolyam során el kell készítenetek, hozzatok létre ***Projekt1***, ***Projekt2***, ***Projekt3***, ***Projekt4***, ***Projekt5***, illetve ***Projekt6*** nevű mappákat, és ezekbe mentsétek el a kért fájlokat.

Végül a **portfóliótoknak** a következő felépítésűnek kell lennie:



Minden lecke végén javasoljuk, hogy a következő oldalon található értékelőlap segítségével értékeljék saját magatartásotokat.

Viselkedés és tevékenységfigyelő napló*

Család és keresztnév: _____ Dátum: _____

Lecke címe: _____

Osztály: _____

Értékelési mérce:

1 = egyáltalán	2 = alig	3 = közepesen	4 = jól	5 = nagyon jól

Önértékelés

Tevékenység / Kritérium	Diák	Tanár
Figyeltem az órán	☐1 ☐2 ☐3 ☐4 ☐5	☐1 ☐2 ☐3 ☐4 ☐5
Megértettem az elhangzott fogalmakat	☐1 ☐2 ☐3 ☐4 ☐5	☐1 ☐2 ☐3 ☐4 ☐5
Válaszoltam a kérdésekre	☐1 ☐2 ☐3 ☐4 ☐5	☐1 ☐2 ☐3 ☐4 ☐5
Aktívan részt vettem	☐1 ☐2 ☐3 ☐4 ☐5	☐1 ☐2 ☐3 ☐4 ☐5
Segítséget kértem, amikor nem értettem valamit	☐1 ☐2 ☐3 ☐4 ☐5	☐1 ☐2 ☐3 ☐4 ☐5
Megértettem a tanulási feladatokat	☐1 ☐2 ☐3 ☐4 ☐5	☐1 ☐2 ☐3 ☐4 ☐5
Megoldottam a gyakorlatokat / feladatokat	☐1 ☐2 ☐3 ☐4 ☐5	☐1 ☐2 ☐3 ☐4 ☐5
Hatékonyan dolgoztam csapatban	☐1 ☐2 ☐3 ☐4 ☐5	☐1 ☐2 ☐3 ☐4 ☐5
Segítettem a társaimnak	☐1 ☐2 ☐3 ☐4 ☐5	☐1 ☐2 ☐3 ☐4 ☐5
Építő jellegűen együttműködtem	☐1 ☐2 ☐3 ☐4 ☐5	☐1 ☐2 ☐3 ☐4 ☐5
Helyesen írtam / teszteltem a kódot	☐1 ☐2 ☐3 ☐4 ☐5	☐1 ☐2 ☐3 ☐4 ☐5
Önállóan elhárítottam a hibákat	☐1 ☐2 ☐3 ☐4 ☐5	☐1 ☐2 ☐3 ☐4 ☐5



Visszatekintés

✅ Mit csináltam jól: _____

_____

🔧 Mit tudok javítani: _____

_____

💡 Egy ötlet vagy érdekesség az órából: _____

_____

*Ez a kérdőív segít felmérni, mennyire értetted meg az órát, és mennyire vettél részt benne. Ez nem egy teszt, hanem egy önértékelés, amely segít a fejlődésben. A kérdőívet a tanuló és a tanár együtt tölti ki, és az esetleges véleménykülönbségeket megbeszélik, hogy meghatározzák a fejlődéshez szükséges irányokat.

Év eleji tudásfelmérő

Hozzatok létre egy *Eveleji_felmero* nevű mappát. Oldjátok meg az alábbi 1., 2., 3., 4., 5. és 6. feladatot, és írjátok le a válaszokat egy *valaszok.docx* nevű fájlba.

Az 1., 2., 3., 4. és 5. kérdésnél válasszátok ki a helyes választ (csak egy választ) **(20 pont)**

1. Az algoritmus:
a) egy játék **b)** egy kép **c)** egy véges lépéssorozat **d)** egy számítógép

2. Mi a műveletek végrehajtási sorrendje a következő kifejezésben: ***x+2*(z-y)***:
a) +,*, - **b)** -,*,+ **c)** -,+,* **d)** *,-,+

3. Az alábbi lehetőségek közül melyik adja meg a helyes lépéssorozatot, amelyben a két változó, a és b értékeit egy ***aux*** segédváltozó segítségével cseréljük fel?
a) *aux=a; a=b; b=aux*; **b)** *a=b; b=aux; aux=a*;
c) *a=aux; aux=b; b=a*; **d)** *b=a; aux=b; a=b*.

4. Mennyi lesz az **x** változó értéke a mellékelt értékadó utasítássorozat után?
a) 3 **b)** 5 **c)** 2 **d)** 1

```
y←2
x←y+1
x←x+y
```

5. A mellékelt szekvenciában szerepel:
a) 2 hozzárendelési utasítás **b)** 3 változó **c)** 2 változó **d)** 1 hozzárendelési utasítás

6. Vizsgáljátok meg az alábbi algoritmust, és válaszoljatok a kérdésekre. **(20 pont)**

```
beolvas a, b
s ← a + b
kiír s
```

a) A bemeneti adatok: ....., a kimeneti adat pedig ..... .
b) Az a = 7 és b = 5 esetén megjelenő érték ..... .
c) Írjátok le röviden, hogy mi lehetne a feladat szövege, amelyet ez az algoritmus old meg.

7. Írjatok egy programot, amely a billentyűzetről beolvassa ***a*** és ***b*** két egész számot, és kiszámítja:
a) a kettő közül a kisebbet **b)** a két szám szorzatának utolsó számjegyét **(20 pont)**

8. Írjatok egy programot, amely beolvas a billentyűzetről egy ***n*** nem nulla természetes számot, és csökkenő sorrendben megjeleníti az első ***n*** nem nulla természetes számot. **(15 pont)**

9. Írjatok egy programot, amely beolvassa a billentyűzetről egy ***n*** természetes számot, majd kiszámítja és megjeleníti annak páratlan számjegyeinek számát. **(15 pont)**

Javítókulcs: Az 1-5. feladatokra adott minden helyes válaszért **4 pont** jár, valamint hivatalból **10 pont**!
6. a) (3 pont), (2 pont), **b)** (5 pont), **c)** helyes megfogalmazásért – **10** pont
7. változók deklarálása (**2p**), adatbeolvasás (**3p**), **a)** meghatározás: minimum (**5p**), **b)** szorzat (**2p**), utolsó számjegy (**3p**), eredmény kiíratása (**3p**), szintaktikai helyesség (**2p**).
8. változók deklarálása (**2p**), adatbeolvasás (**3p**), az első **n** nem nulla természetes szám végigjárása (**5p**), csökkenő sorrend (**3p**), eredmény kiíratása (**2p**).
9. változók deklarálása (**2p**), adatbeolvasás (**3p**), egy számjegy meghatározása (**3p**), egy számjegy paritásának ellenőrzése (**2p**), a páratlan számjegyek száma (**3p**), eredmény kiíratása (**2p**).

1. Fejezet – Problémamegoldási stratégiák. A programozás alapjai

1. Lecke – Egy program elkészítésének szakaszai: az elemzéstől és tervezéstől a tesztelésig és hibakeresésig

*„Nem félek a számítógépektől. A hiányuktól félek” – **Isaac Asimov***

Informatika. Az adatok és az algoritmus

<def/> Az **informatika** olyan tudományterületek összessége, amelyek az információ számítógépes feldolgozásával foglalkoznak, és alapvető fontosságúak a *számítógépes gondolkodás* fejlesztésében.

Megj. Az *informatikai gondolkodás* az ember azon képessége, amelynek segítségével utasítássorozatokkal és ***algoritmusokkal*** megvalósított problémamegoldási módszereket dolgoz ki.

Az **informatikai gondolkodás fő jellemzői** a következők: a ***felbontás*** (az összetett problémákat egyszerűbb, könnyebben megoldható részproblémákra bontjuk), az ***általánosítás*** (azonosítjuk a hasonló mintákat vagy megoldásokat, hogy az új problémákat gyorsabban meg tudjuk oldani a már ismert megoldások felhasználásával), az ***absztrakció*** (csökkentjük a fölösleges részleteket), az **algoritmizálás** (kidolgozzuk a probléma megoldásának lépéseit) és az ***értékelés*** (ellenőrizzük, hogy a javasolt megoldás helyes és hatékony-e).

<def/> Az **algoritmus** véges számú, meghatározott sorrendben végrehajtott lépésből álló műveletsor, amely *adatokat* dolgoz fel. Az adatokat ***nevük***, ***típusuk*** és ***értékük*** azonosítja. Az algoritmusok problémák megoldására használhatók a matematika, a fizika, a kémia területén, de a mindennapi életben is.

Megj. Az **adatokat** osztályozhatjuk: **típusuk szerint** (*elemi adatok*: numerikus, logikai, karakterlánc típusú adatok, illetve strukturált adatok), **felhasználásuk időpontja szerint** (*bemeneti adatok*, *közbenső/segédadatok és kimeneti adatok*), **értékük szerint** (*változó adatok és konstans adatok*).

<def/> A számítógépek nem képesek közvetlenül értelmezni az emberi nyelvet, ezért az **adatok** digitális (bináris kód – **0** és **1**) formában vannak ábrázolva. A számítógép tehát gépi kódból álló nyelvvel működik.

Megj. A programozási nyelvek az *assembly* nyelvtől a *magas szintű és objektumorientált nyelvekig* fejlődtek, és ma már a mesterséges intelligenciában használt nyelvekig jutottak el. Segítségükkel az algoritmusok átírhatók programokra, amelyek egy adott problémacsoportot hatékonyan oldanak meg.



Egy program kidolgozásának szakaszai

Egy algoritmust megvalósító program elkészítéséhez a következő lépéseket kell végigjárni:

- **A probléma elemzése:** a bemeneti és kimeneti adatok, valamint a közöttük fennálló kapcsolatok azonosítása.
- **A megoldási mód kidolgozása vagy moduláris tervezés:** a problémát részproblémákra bontjuk, kiválasztjuk a megoldási módszert, és leírjuk a végrehajtandó lépések sorrendjét.
- **A megoldó algoritmus implementálása egy programozási nyelven:** a probléma megoldására szolgáló algoritmust egy programozási nyelven valósítjuk meg, annak szintaktikai szabályait betartva (például Pythonban).
- **A program tesztelése és hibakeresés (debug):** a program futtatásával azonosítjuk és kijavítjuk a szintaktikai vagy logikai hibákat.
- **A bonyolultság elemzése és a program optimalizálása**: az algoritmus hatékonyságának értékelését jelenti a felhasznált memória és a végrehajtási idő szempontjából.

• Támpontok ezen szakaszok alkalmazásához problémamegoldás során

1. Elemzés
- a bemeneti és kimeneti adatok azonosítása;
- a feladat követelményeinek megértése;
- a feltételek és korlátozások meghatározása.

2. Tervezés
- a megfelelő adatszerkezetek kiválasztása;
- az algoritmus vázlatának kidolgozása (pszeudokódban vagy folyamatábrában);
- a probléma részproblémákra bontása (moduláris tervezés).

3. Megvalósítás
- az algoritmus átírása programozási nyelvre;
- a szintaktikai szabályok betartása;
- a programkód világos és jól olvasható szerkesztése.

4. Tesztelés
- a program ellenőrzése különböző bemeneti adatokra (normál esetek, határesetek, hibás adatok);
- a kapott eredmények összehasonlítása a várt eredményekkel;
- a szintaktikai, logikai vagy futási hibák azonosítása.

5. Hibakeresés
- a tesztelés során azonosított hibák helyének meghatározása és kijavítása;
- a változó értékek nyomon követése;
- a program újraellenőrzése minden egyes javítás után.

Tesztek kidolgozásának szempontjai

– A **bemeneti adatok** érvényesítéséhez ellenőrizzük:
- hogy a megadott adatok megfelelnek-e az elvárt típusnak (egész szám, karakterlánc stb.);
- hogy az értékek a megengedett tartományba esnek-e (például az osztályzat 1 és 10 között van);
- mi történik határértékek esetén (minimumérték, maximumérték, nulla);
- mi történik érvénytelen adatok esetén (negatív értékek, üres karakterláncok, nem várt karakterek).

– Az **algoritmus működésének** ellenőrzéséhez:
- tesztelés szokásos adatokkal (általános eset);
- speciális esetek tesztelése (üres lista, egyetlen elem, azonos értékek);
- a kapott eredmények összehasonlítása a várt eredményekkel;
- vizsgáljuk meg, hogy a program ugyanazt az eredményt adja-e ismételt futtatások során ugyanazokra a bemeneti adatokra.

Támpontok a változó értékek nyomon követéséhez

- kiíratás utasítások használata a változók értékeinek megjelenítésére a program fontos pontjain;
- az algoritmus követése lépésről lépésre (kézzel papíron vagy debugger-rel);
- a hiba típusának azonosítása:
  - **szintaktikai** hiba – a kód nem felel meg a programozási nyelv szabályainak;
  - **logikai** hiba – a program lefut, de helytelen eredményt ad;
  - **futási** hiba – a program végrehajtás közben leáll (pl. nullával való osztás)
- a hiba kijavítása és a program újbóli tesztelése.

Pl.

A **programkészítés szakaszainak** szemléltetésére a következő feladatot vizsgáljuk meg: *Mihálynak informatikából **n** darab jegye van. Határozzuk meg a tanuló jegyeinek átlagát, majd állapítsuk meg, hogy átment-e vagy sem informatikából.*

ELEMZÉS

- BEMENETI ADATOK: AZ N TERMÉSZETES SZÁM, AMELY A JEGYEK SZÁMÁT JELENTI, VALAMINT AZ EGYES JEGYEK
- KIMENETI ADATOK: A JEGYEK ÁTLAGA, AMELY VALÓS TÍPUSÚ ADAT LESZ
- AZ ADATOK KÖZÖTTI KAPCSOLAT: EGY SEGÉDADATOT HASZNÁLUNK, AMELYBEN MEGHATÁROZZUK A JEGYEK ÖSSZEGÉT.

TERVEZÉS

- A FELADATOT MODULÁRIS MÓDON VALÓSÍTJUK MEG;
- BEOLVASSUK A JEGYEK SZÁMÁT, ELLENŐRIZZÜK, HOGY POZITÍV-E ÉS NEM HALAD MEG EGY ADOTT HATÁRÉRTÉKET;
- BEOLVASSUK A JEGYEKET, AMELYEKNEK 1 ÉS 10 KÖZÖTTI ÉRTÉKEKNEK KELL LENNIÜK;
- KISZÁMÍTJUK A JEGYEK ÖSSZEGÉT, MAJD AZ ÁTLAGOT, ÚGY, HOGY AZ ÖSSZEGET ELOSZTJUK A JEGYEK SZÁMÁVAL, ÜGYELVE ARRA, HOGY AZ ÖSSZEGET VALÓS TÍPUSÚ ADATTÁ ALAKÍTSUK;
- ELLENŐRIZZÜK, HOGY AZ ÁTLAG MEGFELEL-E AZ ÁTMENÉSI FELTÉTELNEK (≥ 5), ÉS KIÍRJUK AZ EREDMÉNYT: ÁTMENT / NEM MENT ÁT.

MEGVALÓSÍTÁS

- AZ ALGORITMUST ÁTÜLTETJÜK A TANULT PROGRAMOZÁSI NYELVRE (PYTHON, C/C++)

TESZTELÉS, HIBAKERESÉS

ESETLEGES SZINTAKTIKAI VAGY LOGIKAI HIBÁKAT.
(PÉLDÁUL: HA N = 3 ÉS A BEOLVASOTT JEGYEK: JEGY1 = 8, JEGY2 = 9, JEGY3 = 10, AKKOR AZ ÁTLAG = 9 LESZ, ÉS A PROGRAM AZ „ÁTMENT" MINŐSÍTÉST JELENÍTI MEG; HA N = 2 ÉS A BEOLVASOTT JEGYEK: JEGY1 = 4, JEGY2 = 3, AKKOR AZ ÁTLAG = 3,5 LESZ, ÉS A PROGRAM A „NEM MENT ÁT" MINŐSÍTÉST JELENÍTI MEG.)

OPTIMIZÁLÁS

- AZ ALGORITMUS MEMÓRIAHASZNÁLAT SZEMPONTJÁBÓL HATÉKONY, MIVEL KONSTANS MEMÓRIABONYOLULTSÁGGAL RENDELKEZIK (O(1)), ÉS VÉGREHAJTÁSI IDEJE LINEÁRIS A JEGYEK SZÁMÁNAK FÜGGVÉNYÉBEN (O(N)).

Feladatlap 1

Hozzátok létre a *Feladatlap1* nevű mappát. Oldjátok meg a következő feladatokat, a helyes válaszokat pedig jegyezzétek le egy *valaszok.docx* nevű fájlba.

1. *Állapítsátok meg a következő állítások igazságértékét:*

a. Az informatikai gondolkodás fő jellemzői lehetnek a következők: felbontás, általánosítás, absztrakció, algoritmizálás és értékelés;

b. Az algoritmus lépések végtelen sorozata, amelyek segítségével sajátos feladatok oldhatók meg;

c. A kimeneti adatok azok, amelyeket a program futása során használnak fel;

d. Értékük szerint az adatok lehetnek változók vagy konstansok.

2. *Rendezzétek sorrendbe az alábbi algoritmus lépéseit két pohár tartalmának felcserélésére: az egyikben gyümölcslé, a másikban víz van.*

Töltsd meg a kék poharat gyümölcslével. ☐
Vegyél egy üres, lila színű poharat. ☐
Öntsd a gyümölcslevet a kék pohárból a lila pohárba. ☐
Töltsd meg a sárga poharat vízzel. ☐
Öntsd a gyümölcslevet a lila pohárból a sárga pohárba. ☐
Öntsd a vizet a sárga pohárból a kék pohárba. ☐



Önértékelés!

Töltsétek ki a válaszokat egy *onertekeles1.docx* nevű fájlban, olyan jelöléseket használva, mint: 1 – *igen*, 2 – *nem*, 3 – *nem vagyok biztos*. Mentsétek el a fájlt a *Feladatlap1* mappába.

	A lecke végén tudom...	*Igen*	*Nem*	*Nem biztos*
1.	mi az algoritmus.			
2.	mik az adatok.			
3.	melyek egy program elkészítésének lépései.			
4.	mi az algoritmus lépéseinek helyes végrehajtási sorrendje.			

5. Nehézségeim adódtak a következőkben: …

Ne felejtsétek el értékelni a viselkedéseteket és a tevékenységeteket az óra során!
(a 7. oldalon található viselkedés- és tevékenységfigyelő napló segítségével)

A lecke végén így éreztem magam:

2. Lecke – Algoritmusok ábrázolása: folyamatábrák, pszeudokód, programozási nyelvek

1. Folyamatábrák

<def/> **A folyamatábra** az algoritmusok grafikus ábrázolásának egyik módja.

Az algoritmus vezérlő szerkezeteit **grafikus alakzatok** ábrázolják, amelyek az algoritmus lépéseit jelképezik, míg azok végrehajtási sorrendjét (a lépések egymásutánját) nyilak jelzik:

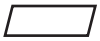


Emlékezzünk vissza!

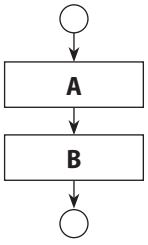
✓ Az ***algoritmus bemeneti adatokat*** dolgoz fel, és ***kimeneti adatokat*** szolgáltat. Néha szükség lehet ***segéd- (közbenső) adatokra*** is a feldolgozás során.

✓ A ***strukturált programozás*** alapelvei a modularizálás és az adatok strukturálása. Ez utóbbi három alapvető vezérlési szerkezetre épül: a szekvenciális (lineáris), az elágazó (alternatív) és az ismétlődő (ciklusos) szerkezetre.

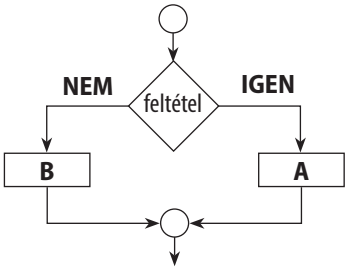
✓ Az ***algoritmusok ábrázolására többféle módszer létezik***: folyamatábrák (logikai sémák), *pszeudokód*, illetve *programozási nyelvek* segítségével.

	*Folyamatvonal (nyíl):* összekapcsolja a blokkokat, és jelzi a lépések végrehajtásának sorrendjét
	*Műveleti blokk:* különböző műveletek és értékadások végrehajtására szolgál.
	*Alprogramblokk:* egy alprogram meghívását jelöli, amelynek saját, külön folyamatábrája van.
	*Bemeneti/kimeneti blokk:* az adatok beolvasásának, illetve kiírásának műveletét jelöli.
	*Döntési blokk:* egy feltételt jelöl, két lehetséges kimenettel (Igen/Nem vagy Igaz/Hamis).
	*Kezdő/záró blokk (terminátor):* az algoritmus kezdetét (START) vagy végét (STOP) jelöli.
	*Összekötő blokk (konnektor):* a folyamatábra két vagy több pontjának összekapcsolására szolgál.

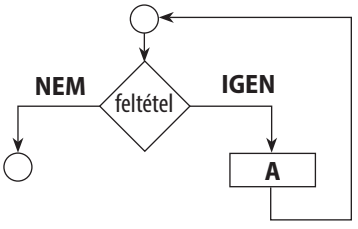
<def/> Bármely algoritmus leírható folyamatábrával a következő **vezérlőszerkezetek** felhasználásával:



lineáris szerkezet



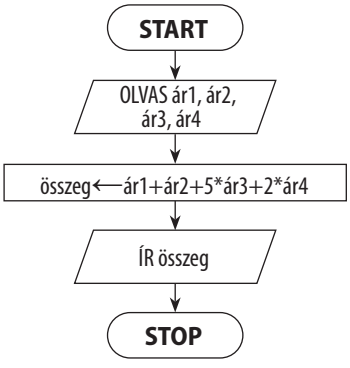
alternatív szerkezet



ismétlő szerkezet

Lineáris szerkezet

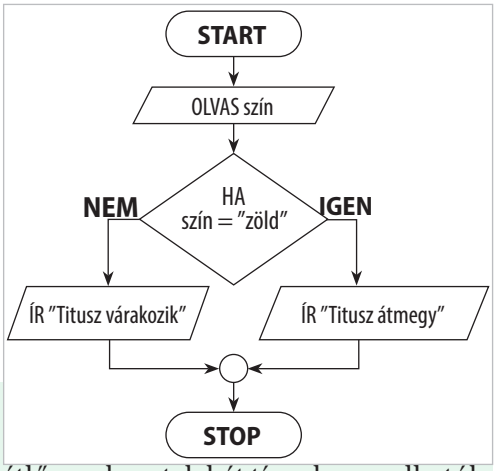
<def/> A **lineáris szerkezet** az utasítások adott sorrendben történő végrehajtását jelenti. Nem tartalmaz ismétléseket, minden utasítás pontosan egyszer hajtódik végre.



Pl. Ilonka szeretne vásárolni egy iskolatáskát, egy tolltartót, öt grafitceruzát és két töltőtollat. Ismerve az egyes termékek árát, határozzuk meg, hogy mekkora összegre van szüksége a tervezett vásárlásokhoz.

Alternatív szerkezet

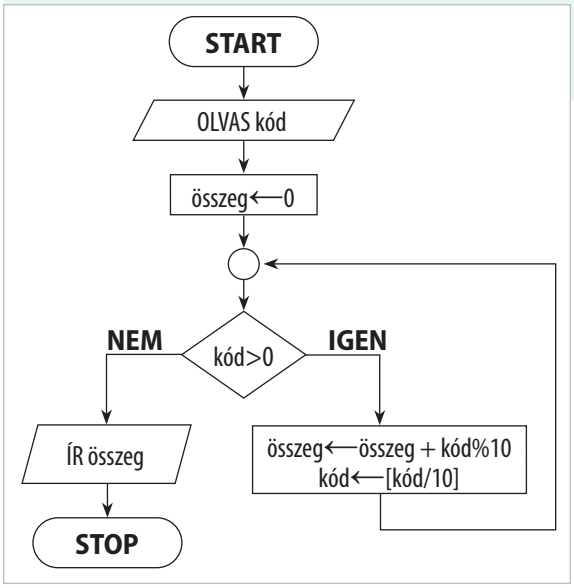
<def/> Az **alternatív szerkezet** lehetővé teszi, hogy a kifejezés értékétől függően kiválasszuk a végrehajtandó utasítást.



Pl. Titusz a közlekedési lámpánál áll, és szeretne átkelni a gyalogátkelőhelyen. A jelzőlámpa színétől függően eldönti, hogy mit tegyen. Ha a lámpa pirosat mutat, Titusz várakozik, ha pedig zöldet mutat, akkor átkelhet az úttesten. Segíts neki a helyes lépések követésében!

Ismétlő szerkezet

<def/> Az ismétlő (ciklusos) szerkezet egy utasítás vagy utasítássorozat többszöri végrehajtását végzi, egy feltételtől függően. Az ismétlő szerkezetek két típusba sorolhatók: *ismeretlen számú* ismétléssel működő ciklusok és előre *ismert számú* ismétléssel működő ciklusok.



Meaj. - Egy ismeretlen lépésszámú ismétlő szerkezet lehet **elöltesztelő** (a feltételvizsgálat a szerkezet elején van) vagy **hátultesztelő** (a feltételvizsgálat a szerkezet végén áll).

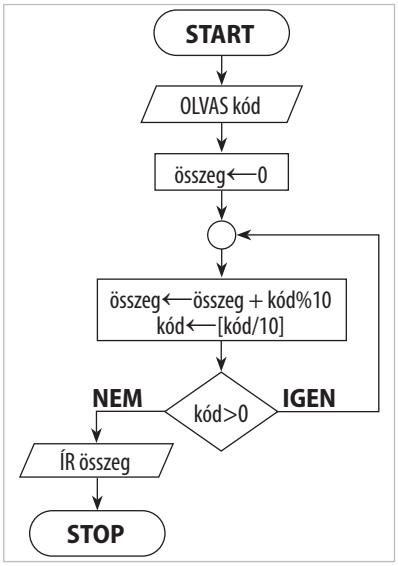
- Az **elöltesztelő** ismétlő szerkezetben a végrehajtás feltétele a ciklus elején, míg a **hátultesztelő** ismétlő szerkezetben a feltétel vizsgálata a ciklus végén történik.

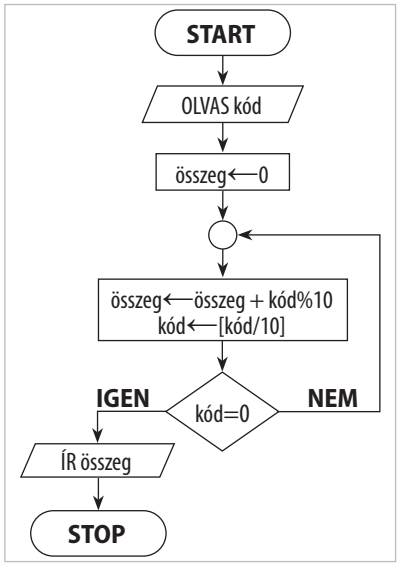
- Az előfeltételes és az utófeltételes ismétlő szerkezet közötti különbség az, hogy az utófeltételes szerkezetben az utasítások *legalább egyszer* végrehajtódnak, míg az előfeltételes szerkezetben csak akkor, ha a feltétel *igaz*.

Pl. Mirunának van egy doboza, amelyet egy 9 számjegyből álló kóddal ellátott lakat zár le. A doboz kinyitásához Mirunának meg kell határoznia a kód számjegyeinek összegét, majd ezt az összeget kell megadnia. Segíts Mirunának kinyitni a dobozt.

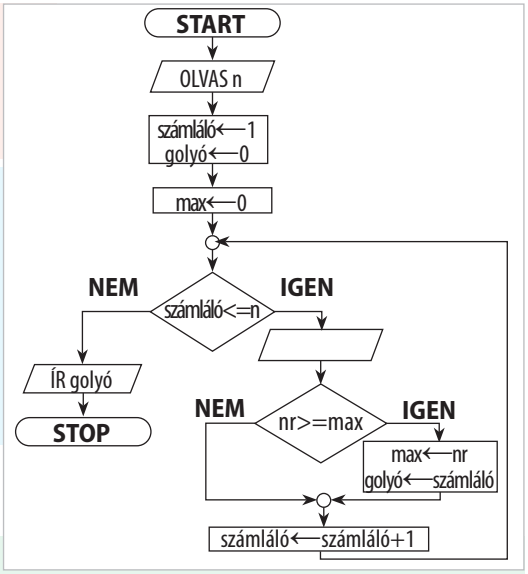
Meaj. A ***hátultesztelő ismétlő szerkezet*** kétféle formában fordulhat elő, attól függően, hogy a ciklus a feltétel teljesülése vagy a feltétel nem teljesülése esetén ismétlődik. Ezt a két változatot a mellékelt folyamatábrák mutatják be.

Meaj. Az ***ismert lépésszámú ismétlő szerkezetben*** a feltétel a szerkezet elején található, és akkor használjuk, amikor előre tudjuk, hogy az utasításokat hányszor kell megismételni.





Pl. Annának ***n*** darab doboza van. Minden dobozban el van rejtve egy-egy golyó, minden golyón egy természetes szám van. Írjunk algoritmust, amely segít Annának meghatározni, hogy melyik dobozban van az a golyó, amelyen a legnagyobb szám látható.



Megj. - Bármely ismert lépésszámú ismétlő szerkezet átalakítható ismeretlen lépésszámú ismétlő szerkezetté. ***Ennek a fordítottja nem mindig igaz!***

- Az algoritmusok folyamatábrákkal történő ábrázolása ***előnyös***, mert világosan szemlélteti a végrehajtandó lépéseket, és segíti a hibák azonosítását. Ez az ábrázolási mód azonban ***hátrányos*** lehet nagy méretű, összetett algoritmusok esetén.

2. Pszeudokód nyelv

<def/> ***A pszeudokód nyelv*** (a görög eredetű *pseudo* = hamis szóból) algoritmusok tervezésére szolgál. Meghatározott írási szabályokból és az azt alkotó utasítások sajátos szintaxisából áll. Az utasítások a leírásuk sorrendjében hajtódnak végre.

<def/> Bármely algoritmus leírható ***pszeudokód nyelven*** a következő vezérlőszerkezetek felhasználásával:

```
adat_tip változók
beolvas adat
utasítás 1
utasítás 2
.....
kiír adat
```

lineáris szerkezet

```
ha <feltétel> akkor
    <utasítások 1>
[különben <utasítások 2>]
ha vége
```

alternatív szerkezet

```
amíg <feltétel> végezd el
    <utasítások>
amíg vége
```

ismétlő szerkezet

Lineáris szerkezet

Pl. Mihály egy olyan utcán sétál, ahol 8 fa van. Minden fán egy tábla látható, amelyen fel van tüntetve a fa fajtája és magassága.

Mivel Mihály nagyon kíváncsi fiú, szeretné meghatározni a fák átlagos magasságát. Segíts neki úgy, hogy elkészíted a megoldás algoritmusát pszeudokód nyelven.

```
algoritmus átlag_magasság
valós h1, h2, h3, h4, h5, h6, h7, h8, atlag
beolvas h1, h2, h3, h4, h5, h6, h7, h8
átlag ← (h1+h2+h3+h4+h5+h6+h7+h8) / 8
kiír átlag
algoritmus vége
```

Alternatív szerkezet

Pl. Egy napon Anett az utcán sétált. Találkozott két kislánnyal, akik arra voltak kíváncsiak, hogy közülük melyikük idősebb. Anett megkérdezte a kislányokat, hogy hány évesek, majd megmondta nekik, melyikük az idősebb. Segíts Anettnek pszeudokód segítségével szemléltetni azokat a lépéseket, amelyek a rejtély megoldásához vezettek.

```
algoritmus ev
természetes x, y
beolvas x, y
ha x > y akkor kiír "Az első kislány nagyobb mint a második."
különben kiír "A második kislány nagyobb vagy egykorú mint az első."
ha vége
algoritmus vége
```

Ismétlő szerkezet

Pl. **Példa az ismeretlen lépésszámú elöltesztelő ismétlő szerkezet alkalmazására:** Anna egy zsákból természetes számokat tartalmazó cédulákat húz ki, addig, amíg a 0 számot ki nem húzza ki. Segíts Annának olyan algoritmust írni, amely meghatározza a kihúzott számsorozatban szereplő páros számok összegét.

```
algoritmus páros_számok
természetes x, s ← 0
beolvas x
amíg x > 0 végezd el
    ha x % 2 = 0 akkor s ← s + x
    vége ha
beolvas x
vége amíg
kiír s
algoritmus vége
```

Megj. A **hátultesztelő ismétlő szerkezet** esetén kétféle ábrázolási forma létezik. A két forma közötti különbség a következő: az ***első*** esetben az utasítások addig ismétlődnek, ***amíg a feltétel igaz***, a ***második*** esetben az utasítások addig ismétlődnek, ***amíg a feltétel igazzá nem válik***.

```
algoritmus páros_szám
természetes x, s ← 0
ismételd
beolvas x
    ha x % 2 = 0 akkor s ← s + x
    vége ha
ameddig x = 0
kiír s
algoritmus vége
```

```
algoritmus páros_számok
természetes x, s ← 0
végezd
beolvas x
    ha x % 2 = 0 akkor s ← s + x
    vége ha
amíg x > 0
kiír s
algoritmus vége
```

Pl. **Példa az előre ismert számú ismétléssel működő ismétléses szerkezet alkalmazására:** Freya egy nagyon okos, játékos kislány, aki szereti a számokat. Kisebb öccsével, Eliasszal játszik, és szeretné megtanítani őt kétjegyű számok összeadására. Ezért ***n*** darab kártyát mutat neki, amelyeken természetes számok szerepelnek, és arra kéri, hogy válassza ki azokat, amelyek pontosan kétjegyűek, majd határozza meg ezek összegét. A megoldás algoritmusát a mellékelt ábra szemlélteti.

```
algoritmus páros_számok
természetes n, i, s ← 0, nr
beolvas n
minden i ← 1, n végezd el
    beolvas nr
    ha nr >= 10 and nr <= 99 akkor
        s ← s + nr
    ha vége
minden vége
kiír s
algoritmus vége
```

Megj. Az algoritmusok pszeudokód nyelven történő ábrázolása ***előnyös*** lehet, mivel könnyen érthető, szemléletes, és független a programozási nyelvektől. ***Hátrány***a viszont, hogy az algoritmus közvetlenül nem futtatható és nem tesztelhető, ezért később át kell ültetni egy programozási nyelvre.

3. Programozási nyelvek

<def/> A **programozási nyelv** a programozó és a számítógép közötti kommunikáció eszköze. **Vokabulárium**ból (szókészletből) és az utasítások leírására szolgáló **szabályrendszer**ből áll. A programozási nyelvek segítségével az algoritmusok **programok** formájában valósíthatók meg.

Megj. A C/C++ nyelven írt algoritmusok írásához szoftverfejlesztői környezetet (pl. CodeBlocks/MinGW) használhatunk, amely lehetővé teszi a forráskód ***szerkesztését***, ***fordítását***, a ***hibakeresést*** és a ***tesztelést***.

<def/> A C/C++ nyelv **szókészlete** a következő elemekből épül fel: *karakterkészlet* (kis- és nagybetűk, számjegyek és speciális karakterek), *azonosítók*, *kulcsszavak*, *elválasztók* és *kommentek*.

<def/> Az algoritmusok által kezelt ***adatoknak*** három jellemzője van: *név*, *típus* és *érték*. Egy ***adattípus*** meghatározza, hogy az illető adatnak milyen értékei lehetnek, valamint azt is, hogy miként tárolódik a memóriában. Az adattípusok két nagy csoportba sorolhatók: *elemi* és *strukturált* adattípusok. Az elemi adattípusok a következők: *egész numerikus típusok* (int, short, long, unsigned int, unsigned long, long long), *numerikus valós* (float, double), *karakter* (char, unsigned char), *logikai* (bool).

Az adatok két csoportra oszthatók: ***változókra*** (amelyek értéke változhat) és ***konstansokra*** (értékük nem változik).

A **kifejezés** egy vagy több **operandusból** áll, amelyeket **operátorok** választanak el egymástól.

Az operátorok alapján a kifejezések **aritmetikai** vagy **logikai** kifejezések lehetnek.

Emlékezzünk! Operátor típusok!

✓ Aritmetikai: unáris (+, -), bináris (+,-, *, /, %)
✓ Reláció: egyenlőtlenség (<, <=, >, >=), egyenlőség (==, !=)
✓ Inkrementálás/Dekrementálás: ++, --
✓ Logikai: !, &&, ||
✓ Értékadó: egyszerű (=), összetett (+=, -=, *=, /=, %=)
✓ Explicit konverzió: (típus) operandus
✓ Feltételes: e ? kifejezés1: kifejezés2

A C/C++ programozási nyelven írt **vezérlési szerkezetek** a következők:

Lineáris szerkezet	Alternatív szerkezet	Ismétlő szerkezet
int nr1, nr2, s; cin>>nr1>>nr2; s=nr1+nr2; cout<<s;	**if** (feltétel) utasítás 1; **else** utasítás 2;	**while**(feltétel) utasítás ;
Példa: Anett biológiából három jegyet kapott. Segíts neki kiszámítani a jegyeinek átlagát.	*Példa: János felír egy számot egy papírlapra. Ennek értékétől függően barátja, András megállapítja, hogy a szám páros vagy páratlan. Segíts Jánosnak olyan programot készíteni, amely kiírja a kívánt eredményt.*	*Példa: Tánja kap egy cetlit, amelyen egy többjegyű természetes szám és a következő feladat szerepel: „Határozd meg, hány páratlan számjegye van a számnak!”. Írjunk programot, amely meghatározza ezt az értéket, és ha a telefonszám egyetlen páratlan számjegyet sem tartalmaz, akkor írja ki a „Nem létezik” üzenetet.*
#include <iostream> **using namespace** std; **int** main() { **unsigned int** nt1, nt2, nt3; **double** media; cin>>nt1>>nt2>>nt3; media=(nt1+nt2 +nt3)/3.0; cout<<media; **return** 0; }	*#include <iostream>* **using namespace** std; **int** main() { **unsigned int** nr; cin>>nr; **if**(nr%2==0) cout<<"A szám páros"; **else** cout<<"A szám páratlan"; **return** 0; }	*#include <iostream>* **using namespace** std; **int** main() { **unsigned int** nr, k=0; cin>>nr; **while**(nr!=0) { **if**(nr%2==1) k++; nr/=10; } **if**(k!=0) cout<<k; **else** cout<<"Nem létezik"; **return** 0; }

Hátultesztelő ismétlő struktúra

A C/C++ nyelvben a **hátul-tesztelő ismétlő** szerkezet egyetlen formában létezik: a ciklus végén történő feltételvizsgálattal megvalósított ciklusként.

```
#include <iostream>
using namespace std;
int main()
{
    unsigned int nr, k=0; cin>>nr;
    do
    {
        if(nr%2==1)
            k++;
        nr/=10;
    }while(nr!=0);
    if(k!=0)
        cout<<k;
    else
        cout<<"Nem létezik";
    return 0;
}
```

Ismert lépésszámú ismétlő struktúra

Példa: A testnevelő tanár szeretné megtudni, ki a legmagasabb tanuló az osztályban, valamint hogy hány tanuló van, aki vele azonos magasságú. Írjunk programot, amely elvégzi ezeket a műveleteket.

```
#include <iostream>
using namespace std;
int main()
{
    unsigned int n, h, hmax=0, counter=0; cin>>n;
    for(int i=1;i<=n;i++)
    {
        cin>>h;
        if(h>hmax)
        {
            hmax=h;
            counter=1;
        }
        else
            if(h==hmax)
                counter++;
    }
        cout<<hmax<<'    '<<counter;
        return 0;
}
```

Megj. Az algoritmusok programozási nyelven történő ábrázolása számos előnnyel jár. Például pontos, lehetővé teszi az algoritmus végrehajtását, hatékony és jól teljesít, támogatja a kód újrafelhasználását. Ugyanakkor hátrányt is jelenthet, mivel a programozási nyelvek szigorú szintaktikai szabályokat alkalmaznak, amelyeket pontosan be kell tartani.

Feladatlap 2

Hozzátok létre a *Feladatlap2* nevű mappát. Oldjátok meg a következő feladatokat, a helyes válaszokat pedig jegyezzétek le egy *valaszok.docx* nevű fájlba.

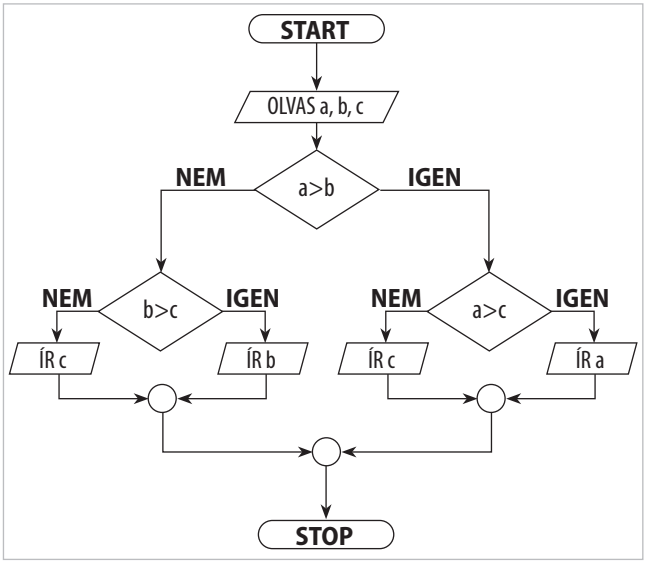
1. *Kövesd a mellékelt folyamatábrán látható algoritmust, majd oldd meg az alábbi feladatokat:*

a. Állapítsd meg, hogy mit ír ki az algoritmus a 175, 100 és 23 értékek bevitele esetén. Mi lesz a kimenet a 2, 2 és 0 értékek bevitelekor?

b. Írd le ide, hogy mit valósít meg ez az algoritmus: ..... .

c. Alakítsd át pszeudokóddá ezt a folyamatábrát.

d. Írd át az algoritmust C/C++ nyelvre.



2. *Határozd meg az alábbi kijelentések logikai értékét:* **a.** Az ismert lépésszámú ismétlő szerkezet átalakítható ismeretlen számú ismétléssel működő ismétlő szerkezetté. Ennek fordítottja is mindig igaz! **b.** A szekvenciális (lineáris) szerkezet lehetővé teszi az utasítások végrehajtását abban a sorrendben, amelyben le vannak írva. **c.** A hátultesztelő ismétlő szerkezetnek kétféle leírási módja van pszeudokódban is, és a C/C++ programozási nyelvben is. **d.** Egy vezérlési szerkezet lehet: szekvenciális, elágazó vagy ismétlő.

3. *Rendezd helyes sorrendbe az alábbi utasításokat úgy, hogy a kapott algoritmus két egész szám legnagyobb közös osztójának meghatározását végezze.*

algoritmus vége
r ← a % b
beolvas a, b
egész a, b, r
algoritmus lnko
a ← b
r ← a % b
amíg r!=0 végezd el
b ← r
kiír b
amíg vége

4. *Kövesd a mellékelt képen pszeudokód nyelven megadott algoritmust, majd oldd meg az alábbi feladatokat:* **a.** Mit ír ki, ha a következő értékeket visszük be? 5, 13, 2, 0, 28, 29. **b.** Tudva, hogy n értéke 6, add meg azt az egy egymástól különböző számokból álló sorozatot, amelynek beolvasása esetén a program által kiírt érték 0 lesz. **c.** Írj pszeudokódban a megadottal egyenértékű algoritmust, amelyben a „minden ... végezd el” ciklust egy „amíg ... végezd el” (előfeltételes) ismétlő szerkezettel helyettesítsd. **d.** Írd meg az algoritmusnak megfelelő C/C++ programot.

```
algoritmus feladat
egész n, nr, k, ok, i, d
beolvas n
k ← 0
minden i ← 1, n végezd el
    beolvas nr
    ok ← 1
    ha nr < 2 akkor ok ← 0
    ha vége
    ha nr % 2 = 0 && nr<> 2 akkor ok ← 0
    ha vége
    minden d ← 3, [nr/2], 2 végezd el
        ha nr % d = 0 akkor ok ← 0
        vége ha
    minden vége
    ha ok = 1 akkor k ← k + 1
    ha vége
minden vége
kiír k
algoritmus vége
```

5. *Írd meg pszeudokódban és C/C++ nyelven a következő feladatot megoldó algoritmust:* Beolvasunk egy n elemű egész számsorozatot. Minden beolvasott számra határozzuk meg és írassuk ki azt a prímtényezőt, amely a szám prímtényezős felbontásában a legnagyobb hatványon szerepel.



Önértékelés!

Töltsétek ki a válaszokat egy *onertekeles2.docx* nevű fájlban, olyan jelöléseket használva, mint: *1 – igen, 2 – nem, 3 – nem vagyok biztos benne*. Mentsétek el a fájlt a *Feladatlap2* mappába.

	A lecke végén tudom, hogyan lehet...	***Igen***	***Nem***	***Nem biztos***
1.	azonosítani a feladat megoldásához szükséges vezérlési szerkezeteket.			
2.	egy algoritmust folyamatábrával ábrázolni			
3.	egy algoritmust pszeudokóddal ábrázolni			
4.	egy algoritmust C/C++ nyelven ábrázolni			
5.	felismerni az algoritmusok különböző ábrázolási módjainak előnyeit és hátrányait.			

6. Nehézségeim voltak a következőkben: …

Ne felejtsétek el értékelni a viselkedéseteket és a tevékenységeteket az óra során!
(a 7. oldalon található viselkedés- és tevékenységfigyelő napló segítségével)

A lecke végén így éreztem magam:

3. Lecke – Egy Python-program felépítése. Változók, adattípusok és kifejezések

Egy Python-program felépítése



A **Python nyelv**ben írt program utasításait egy értelmező egymás után hajtja végre.



A fordítást igénylő nyelvekkel ellentétben a Python-programok futtatásához nincs szükség a program előzetes, teljes lefordítására.

- A **Python** használatához telepíteni kell a fejlesztői környezetet, a következő weboldalról: https://www.python.org/downloads/.
- Az IDLE mellett más fejlesztőkörnyezetek is használhatók:
  - *PyCharm* – korszerű fejlesztőkörnyezet Python-programok készítéséhez;
  - *Google Colab* – online környezet, amely lehetővé teszi programok írását és futtatását közvetlenül a böngészőben, telepítés nélkül.

Tudod-e?

✓ A Python nyelven írt programok hordozhatók

✓ A Python könnyen tanulható és használható, ingyenes, objektumorientált programozási nyelv

✓ A Python nyelv fejlesztését 1989-ben Guido van Rossum kezdte el

- A telepítés után az **IDLE** (**I**ntegrated **D**evelopment and **L**earning **E**nvironment) indításához kattints a Windows **Start** gombjára, majd keresd az „**IDLE**” alkalmazást. A konzolban (Interactive Python Shell) a parancsok a „>>>” karakterek után írhatók. Üzenet megjelenítésére a **print()** függvényt használjuk.

```
Python 3.14.4 (tags/v3.14.4:23116:
A11D64) J on win32
Enter "help" below or click "Help
>>> print ("Üdvözöllek benneteket!")
Üdvözöllek benneteket!
```



Új program létrehozásához válaszd a **File – New File** menüpontot (Ctrl+N). A megnyíló ablakban szerkeszthető a Python-kód.

```
print("Üdvözöllek benneteket!")
print("Ez az első programom!")
print("*****************")
```

A program mentéséhez a **File → Save As** (**Ctrl+Shift+S**) menüpontot kell választani, majd meg kell adni a mentés helyét és a fájl nevét. A Python-programok fájlkiterjesztése **.py**. A program futtatása a **Run → Run Module** (**F5**) paranccsal történik, a futás eredménye pedig a Python konzolban jelenik meg.



- A legegyszerűbb program a **print** függvény segítségével egy karaktersorozatot (szöveget) jelenít meg. Pythonban az utasítások pontosvesszővel (;) is elválaszthatók egymástól, de ennek használata nem kötelező. Az olvashatóság javítása érdekében ajánlott minden utasítást külön sorba írni. A Python megkülönbözteti a kis- és nagybetűket (case-sensitive).

A Python nyelv szókészlete



A Python nyelv **szókészlete** – más programozási nyelvekhez hasonlóan – a következő elemekből áll:

– A programok írásához használt *karakterkészlet*, amely magában foglalja az angol ábécé kis- és nagybetűit, a 0–9 számjegyeket, valamint a speciális karaktereket.

– *Azonosítók,* amelyek a változók, konstansok, függvények vagy modulok elnevezésére szolgálnak. Tartalmazhatnak betűket, számjegyeket és az „_” (aláhúzás) karaktert, de nem kezdődhetnek számjeggyel.

– *Kulcsszavak (reserved words)*, amelyeknek a nyelvben különleges jelentésük van (if, else, elif, for, True, while, False, continue, break, from, return, not, and, or).

– *Elválasztójelek,* amelyek az utasítások vagy a program elemeinek elkülönítésére szolgálnak, például: szóköz, behúzás (indentálás), zárójelek, „:”, „,” vagy „;”.

– *Megjegyzések (kommentek),* # karakterrel kezdve, a programkód részeinek magyarázatára szolgálnak.



Pythonban a behúzás (indentálás) különösen fontos, mivel az utasításblokkok határait jelöli, és a nyelv szintaxisának szerves részét képezi.

<def/> Pythonban egy változó **létrehozásához** elegendő nevet adni neki, majd értéket rendelni hozzá. A nyelv automatikusan meghatározza a változó adattípusát a hozzárendelt érték alapján, így nincs szükség az adattípus explicit megadására.

```
valtozo  = 23
print(valtozo )      #kiírja: 23
valtozo  = -55
print(valtozo )      #kiírja: -55
valtozo  = "egy sor"
print(valtozo )      #kiírja: egy sor
```

Az **input()** függvény üzenetet jeleníthet meg a konzolon, adatokat olvas be a billentyűzetről, és mindig egy karakterláncot (string) ad vissza. Az eredményt egy változóhoz rendelhetjük.

```
nev = input('Hogy hívnak? ')
kor = input('Hány éves vagy? ')
print(nev+','+kor+' éves vagy')
```

A **print()** függvény üzeneteket, változók értéit vagy ezek kombinációit jeleníti meg. A karakterláncok a + operátorral összefűzhetőek.

- Pythonban a változók értékei szöveges üzenetekkel együtt formázott karakterláncok segítségével írathatók ki. Erre használható a **format()** függvény, illetve az **f-string** szintaxis, amelynél az idézőjelek elé ***f*** vagy ***F*** kerül. Az *f-stringekben* a változókat és kifejezéseket *{ }* kapcsos zárójelek közé kell írni, a nyelv pedig automatikusan azok értékével helyettesíti őket.

```
nev = input('Hogy hívnak?')
kor = input('Hány éves vagy? ')
print(f'Szia {nev}. {kor} éves vagy.')
```

```
nev = input('Hogy hívnak? ')
kor = input('Hány éves vagy? ')
print('Szia {0}. {1} éves.'.format(nev,kor))
```

Adattípusok

<def/> A Python nyelvben a következő adattípusok léteznek: ***numerikus, logikai***, ***karakterláncok***, ***listák***, ***szótárak*** és ***tuple***-ök. A változók típusa a hozzárendelt érték alapján automatikusan meghatározódik, így az adattípust nem kell előre megadni.

```
nr1 = int(input("Első szám:"))
nr2 = int(input("Második szám:"))
print(nr1 + nr2)
print(nr1 - nr2)
print(nr1 * nr2)
```

```
a = int(input("a="))        # a=2
b = float(input("b="))      # b=3.5
print(a + b)                # 5.5
```

Megj. **Numerikus adattípusok:** az **int** (egész), a **float** (valós) és a **complex** (komplex szám). Az **int**(), **float**() és **complex**() függvények adattípus-konverzióra szolgálnak. Mivel az **input**() függvénnyel beolvasott adatok alapból szöveges típusúak, numerikus műveletek elvégzése előtt szükség esetén át kell alakítani őket a megfelelő numerikus adattípussá.

Ha egy kifejezésben egész és valós számok is szerepelnek, akkor az eredmény típusa valós szám lesz. A Python a valós számokat a szükséges pontossággal jeleníti meg.

```
lanc1 = str("Ez egy szöveg!")
print(lanc1)
lanc2 ="*********"
print(lanc2)
lanc3 = str('''Ez egy példa több sorba írt karakterláncra.''')
print(lanc3)
```

Megj. A változókhoz közvetlenül hozzárendelhető egy karakterlánc, amelyet aposztrófok vagy idézőjelek közé kell tenni. A ***str()*** függvény használata opcionális – ez más adattípusokat alakít át karakterláncokká.

Többsoros karakterláncok esetén három aposztrófot vagy három idézőjelet használunk a szöveg határolására.

```
L1 = list()
L2 = list(('Matek', 'Info', 'Fizika'))
print(L1)     # kiírja: []
print(L2)     # kiírja: ['Matek', 'Info', 'Fizika']
```

<def/> A lista egy Python-adattípus, amely vesszővel elválasztott, [ ] közé tett elemek gyűjteményét tárolja. Az elemek indexelése ***0***-tól ***len(lista)*** - ***1***-ig történik, ahol a ***len()*** függvény a lista hosszát adja vissza. A listát közvetlenül a [ ] szimbólumokkal vagy a ***list()*** függvénnyel lehet létrehozni.

Pl.

```
szinek = ['piros', 'zold', 'kek']
print(szinek[0]) #piros
```

<def/> A listák rendezettek, véges számú elemet tartalmaznak, és módosíthatók. Az elemeknek nem kell azonos típusúaknak lenniük. A hozzáférés index alapján történik: ***0***-tól ***len-1***-ig előre, vagy -1-től visszafelé.

Kifejezések

<def/> A kifejezés operandusok és operátorok meghatározott sorrendben álló együttese, például: 12 - 3 * 2. Kiértékelésük során figyelembe kell venni az operátorok *prioritását* (precedenciáját), valamint az operátorok *asszociativitását* (balról jobbra és jobbról balra). Az azonos prioritású operátorok azonos asszociativitással rendelkeznek.

• Aritmetikai operátorok

<def/> Az aritmetikai operátorok két csoportba sorolhatók: **unáris operátorok**, ezek egyetlen operandussal dolgoznak (+, -, ~), és **bináris operátorok**, amelyek két operandus között végeznek műveletet (+, -, *, **, /, //, %). Az operandusok típusa lehet egész, valós vagy komplex.

```
a = 5
print(+ a)          #5
print(- a)          #-5
print(~ a)          #-6
b = 3
print(a + b)        #8
print(a - b)        #2
print(a * b)        #15
print(a / b)        #1.6666666666666667
print(a % b)        #2
print(a // b)       #1
c=2.25
print(a + c)        #7.25
print(a - c)        #2.75
print(a * c)        #11.25
print(a / c)        #2.2222222222222223
print(a % c)        #0.5
print(a // c)       #2.0
```

- A ~ operátor a bitenkénti negáció operátora.
- A ** operátor a hatványozás műveletét valósítja meg. Az operátor asszociativitása jobbról balra érvényesül, ezért a 2**3**2 kifejezés értéke 512, míg a (2**3)**2 kifejezésé 64.
- Az / operátor két egész vagy valós szám osztását végzi el. Az eredmény minden esetben valós (float) szám lesz.
- Az // operátor egész osztást végez. Az eredmény az osztás hányadosának lefelé kerekített egész része.
- A % operátor két szám osztásakor a maradékot határozza meg.
- A % és az // operátorokat gyakran használjuk valamely szám számjegyeinek feldolgozása esetén.

1.1. Szám utolsó számjegye - a % operátor

A % (modulo) művelet az osztás maradékát adja vissza. Mivel minden természetes számot **n= q×10 + r** formában lehet felírni, a **10**-zel való osztás maradéka pontosan az utolsó számjegy ($0 \le r \le 9$).

```
n = 712345 utolso = n % 10
# Utolso szamjegy
print('Utolso szamjegy:', utolso)   #5
# 4732=473×10+2; 4732 % 10 = 2
# 8900=890×10 + 0; 8900 % 10 = 0
```

1.2. Szám utolsó számjegyének törlése - az // operátor

A // (egész osztás) művelet az osztás egész részét adja vissza. Ha **10**-zel osztunk, az utolsó számjegyet töröljük: 4732 // 10 = 473, mivel 4732 = 473 × 10 + 2.

```
n = 712345
utolso = n % 10   # Utolsó számjegy
print('Utolso szamjegy:', utolso)   #5
n = n // 10   # Utolsó számjegy elhagyása
print('n utolsó számjegy nélkül:', n)
# 71234
```

1.3. Egy szám k-adik számjegye

Ha **n** és egy **k** pozíció (jobbról számolva, 0-tól kezdve) adott, akkor a **k** pozícióban lévő számjegy a következő képlettel számítható ki: **k_számjegy = (n //10^k) % 10**.

```
n = int(input())
k = int(input())
# Ha n és k azonos sorban vannak
# n, k = map(int, input().split())
szamjegy_k = (n // (10 ** k)) % 10
print(f'{k}-ik jobbrol: {szamjegy_k}')
# n = 4732, k = 2, 4732 // 100 = 47; 47 % 10 = 7
```

• Relációs (összehasonlító) operátorok

<def/> A relációs operátorok két operandus összehasonlítására szolgáló bináris operátorok: >, >=, <, <=, == és !=. Az összehasonlítás eredménye mindig egy logikai érték, amely True (igaz) vagy False (hamis). Ezt Boole-értéknek is nevezzük.

```
a, b, c = 5, 3, 5.0
print(a < b)     #False
print(a > c)     #False
print(a == c)    #True
print(b !=c)     #True
print(a >=b)     #True
```

• Logikai operátorok

<def/> A logikai operátorok: **not**, **and** és **or**. Az **and** és az **or** bináris operátorok, míg a **not** unáris operátor. Az operandusok lehetnek egész, valós, logikai stb. típusúak, az eredmény mindig logikai érték lesz.

and	True	False
True	True	False
False	False	False

or	True	False
True	True	True
False	True	False

not	True	False
	False	True

• Értékadó (hozzárendelő) operátorok

Pythonban az értékadás lehet egyszerű (=) vagy összetett (+=, -=, *=, /=, %=, //=, **=). Az értékadó operátorok jobbról balra asszociatívak, ezért először a jobb oldali kifejezés értéke kerül meghatározásra, majd ez az érték a bal oldalon szereplő változóhoz rendelődik.

Feladatlap 3

Hozzátok létre a *Feladatlap3* nevű mappát. Oldjátok meg a következő feladatokat, a helyes válaszokat pedig jegyezzétek le egy *valaszok.docx* nevű fájlba.

1. *Kövesd az alábbi Python-program működését és határozd meg, mit ír ki, ha a beolvasott szám 209.*
Magyarázd meg, mit végez ez a program.

```
szam = int(input("Kérek egy számot:"))
szam = szam // 10
b = szam % 10
szam = szam // 10
c = szam
ujszam = a * 100 + b * 10 + c
print("A létrejött szám: ", ujszam)
```

2. *Írd át a pszeudokód nyelven megadott algoritmust Python nyelvre. Teszteld a programot.*

```
algoritmus átlag_magasság
valós h1, h2, h3, h4, h5, h6, h7, h8, átlag
beolvas h1, h2, h3, h4, h5, h6, h7, h8
átlag ← (h1+h2+h3+h4+h5+h6+h7+h8) / 8
kiír átlag
algoritmus vége
```

3. *Oldd meg Pythonban a következő feladatokat:*

a. Olvassunk be két egész számot. Végezzük el velük az összeadás, kivonás, szorzás, osztás, egész osztás és maradékos osztás műveleteit.

b. Olvassunk be egy 4-jegyű egész számot. Határozzuk meg a számjegyeinek összegét.

c. Olvassunk be egy pontosan 2 számjegyű egész számot. Határozzuk meg az 1-től a beolvasott számig terjedő számok összegét és írjuk ki a beolvasott szám számjegyeinek négyzeteiből képzett szorzatot.

Önértékelés!

Töltsétek ki a válaszokat egy *onertekeles3.docx* nevű fájlban, olyan jelöléseket használva, mint: *1 – igen, 2 – nem, 3 – nem vagyok biztos*. Mentsétek el a fájlt a *Feladatlap3* mappába.

	A lecke végén képes vagyok...	*Igen*	*Nem*	*Nem tudom*
1.	létrehozni egy programot Python nyelven			
2.	azonosítani a Python-beli adattípusokat			
3.	deklarálni változókat			
4.	felsorolni az operátorokat és megmondani, mik a kifejezések			
5.	követni egy Python program működését			

6. Nehézségeim voltak a következőkben: …

Ne felejtsétek el értékelni a viselkedéseteket és a tevékenységeteket az óra során!
(a 7. oldalon található viselkedés- és tevékenységfigyelő napló segítségével)

A lecke végén így éreztem magam:

4. Lecke – Vezérlő szerkezetek: szekvenciális (lineáris), elágazó és ismétlő szerkezet

Szekvenciális (lineáris) szerkezet

<def/> A Pythonban a szekvenciális (lineáris) szerkezet az utasítások egymás utáni végrehajtását jelenti. Az utasítások végrehajtási sorrendje megegyezik a leírásuk sorrendjével, és minden utasítás pontosan egyszer hajtódik végre.

Pl. Írjatok programot, amely meghatározza egy kocka teljes felszínét és térfogatát, ha az élhosszát a konzolról olvassuk be.

```
#szekvenciális szerkezet
l = int(input("Élhossz: "))
felszin = 6 * l * l
terf = l * l * l
print(f"A kocka felszíne {felszin} és térfogata {terf}")
```

Elágazó / döntéshozó szerkezet

<def/> Az ***elágazó szerkezet*** egy logikai feltétel kiértékelésén alapul. A feltétel igazságértékétől függően a program az egyik vagy a másik utasításblokkot hajtja végre. Ha a feltétel igaz (True), akkor az *utasítások_1* blokk fut le, ellenkező esetben az *utasítások_2*. A Python nyelvben az else ág nem kötelező, ezért el is hagyható.

Szintaxis:
if *logikai_feltétel:*
    utasítások_1
else:
    utasítások_2

Pl. Készítsünk programot, amely beolvassa két gyerek életkorát (különböző életkorúak). Határozzuk meg és írja ki, hogy melyik gyermek idősebb.

```
kor1 = int(input("Az első gyerek kora: "))
kor2 = int(input("A második gyerek kora: "))
if kor1 > kor2:
    print("Az első gyerek nagyobb, mint a második")
else:
    print("Az első gyerek nem nagyobb, mint a második")
```

Megj. Az elágazó szerkezet egy másik formája az **if ... elif ... else** szerkezet, amely több feltétel egymás utáni vizsgálatát teszi lehetővé. **Működése** a következő: először a ***logikai_feltétel1*** értékelődik ki, ha ez igaz, akkor az ***utasítások_1*** blokk hajtódik végre, ha hamis, akkor a program a következő (*elif*) ***logikai_feltétel2***-t vizsgálja. Amennyiben ez igaz, az ***utasítások_2*** blokk fut le, ellenkező esetben az else ágban szereplő ***utasítások_3*** blokk hajtódik végre.

Szintaxis:
if *logikai_feltétel1:*
    utasítások_1
elif *logikai_feltétel2:*
    utasítások_2
else:
    utasítások_3

Pl. Készítsünk programot, amely meghatározza a 2^n szám utolsó számjegyét, ahol n egész szám.

```
n = int(input("n="))
if n == 0:
    print(f"2^{n} utolsó számjegye 1")
elif n % 4 == 1:
    print(f"2^{n} utolsó számjegye 2")
elif n % 4 == 2:
    print(f"2^{n} utolsó számjegye 4")
elif n % 4 == 3:
    print(f"2^{n} utolsó számjegye 8")
else:
    print(f"2^{n} utolsó számjegye 6")
```

Megj. Amennyiben egy **if** szerkezetben több utasítás végrehajtására van szükség, az utasításokat egy közös blokkba kell foglalni. Pythonban a blokkhoz tartozó utasításokat behúzással (indentálással) jelöljük. A helyes behúzás a nyelv szintaxisának része, ezért annak hiánya értelmezési hibát eredményez. Az indentálás legegyszerűbben a Tab billentyű használatával végezhető el.

Az ismétlő szerkezet

<def/> Az ismeretlen lépésszámú ismétlő szerkezet esetén a program először kiértékel egy logikai feltételt. Ha a feltétel igaz, akkor végrehajtja a ciklus törzsében található utasításokat, majd ismét kiértékeli a feltételt. Ez a folyamat addig folytatódik, amíg a feltétel igaz. Amikor a feltétel hamissá válik, a ciklus végrehajtása befejeződik.

Szintaxis:
while *logikai_feltétel:*
    utasítások

Megj. - A Python nyelv csak az elöltesztelő ismétlő szerkezetet támogatja közvetlenül. Nincs egy kifejezett megvalósítása a hátultesztelő ismétlő szerkezetnek, ellentétben a C/C++ nyelvvel.

- Az elágazó szerkezethez hasonlóan az ismétléses szerkezet esetében is kiemelten fontos az utasítások helyes indentálása!

<def/> Az ***ismert lépésszámú ismétlő szerkezetet*** akkor használjuk, amikor egy lista, egy karakterlánc vagy más adatsor elemein szeretnénk végighaladni, és azokat a tárolás sorrendjében feldolgozni. Ennek a szerkezetnek a Python-beli szintaxisa a mellékelt ábrán látható.

Pl. Készítsünk programot, amely egy lista elemeinek bejárása során megszámolja a pozitív, a negatív és a nulla értékű elemek számát.

```
lista = list((0, -3, 12, 45, 3, 0, -5, 9))
poz = 0
neg = 0
nulla = 0
for nr in lista:
    if nr > 0:
        poz = poz + 1
    elif nr < 0:
        neg = neg + 1
    else:
        nulla = nulla + 1
print(f"{poz} pozitív, {neg} negatív és {nulla} nulla érték van.")
```

Megj. Az előző példában a lista elemeinek bejárására a for ciklust használtuk. Számsorozatok bejárására a Python a **range()** függvényt biztosítja. A **range(kezdő_érték, végérték, lépésköz)** függvény egy olyan sorozatot hoz létre, amely a kezdőértéktől indul, a végértéket nem tartalmazza, és a megadott lépésközzel halad. A kezdőérték és a lépésköz megadása nem kötelező. Alapértelmezés szerint a sorozat ***0***-tól indul, és a lépésköz 1.

Pl. Írjunk programot, amely meghatározza az ***1***-től ***n***-ig terjedő számok összegét, ahol ***n*** egy beolvasott érték.

```
n = int(input('n='))
s = 0
for nr in range(1, n+1):
    s = s + nr
print(f'A számok összege 1-től {n}-ig: {s}')
```

A break és continue utasítások

<def/> A **break** utasítást ismétlő szerkezetekben használjuk, ha azok végrehajtását kényszerített módon meg akarjuk szakítani.

Pl. Írjunk programot, amely számokat olvas be ***0*** végjelig, majd meghatározza a beolvasott számok összegét.

```
ossz = 0
while True:
    nr = int(input("nr="))
    ossz = ossz + nr
    if nr == 0:
        break
print(f"A számok összege  {ossz}")
```

<def/> A **continue** utasítás hatására a program kihagyja a ciklus törzsében hátralévő utasításokat és közvetlenül a következő iterációval folytatja a végrehajtást. A **continue** használata nem fejezi be a ciklust, csak az aktuális iterációt szakítja meg.

Pl. Írjunk programot, amely beolvassa az ***n*** értékét, majd kiírja az ***1*** és ***n*** közötti páratlan számokat.

```
n = int(input("n="))
for i in range(1, n + 1):
    if i % 2 == 1:
        print(i, end=" ")
    else:
        continue
```

[1] A print() függvény end paramétere határozza meg, hogy mi jelenik meg a megjelenített szám után – ahelyett, hogy új sorra lépne, a megadott karaktert (ebben az esetben egy szóközt) helyezi el.

Feladatlap 4

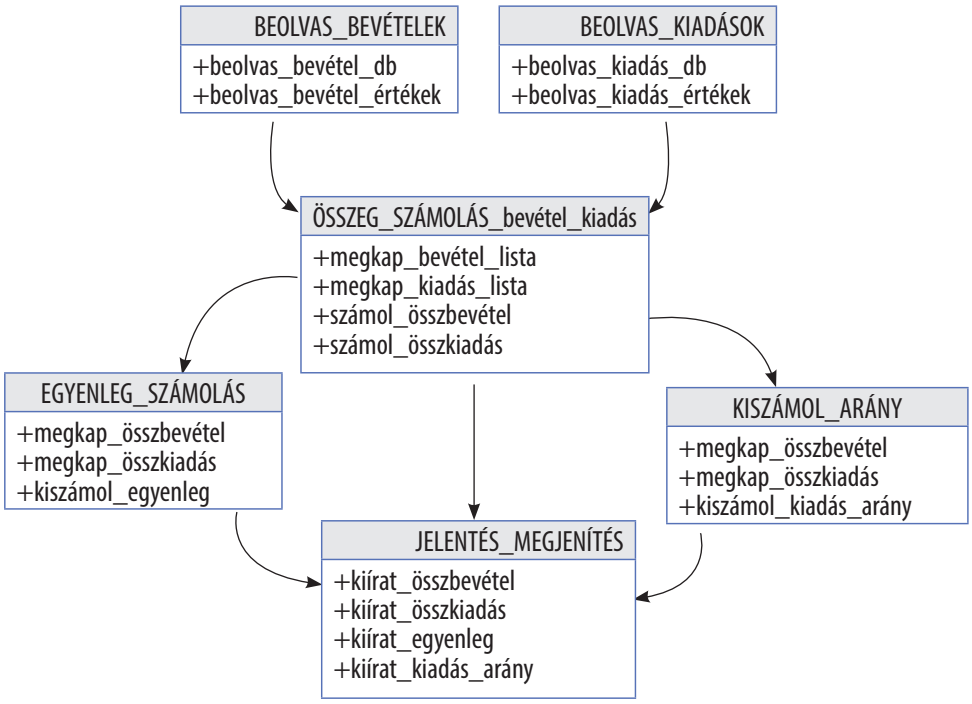
Hozzátok létre a *Feladatlap4* nevű mappát. Készítsétek el az alábbi Python-programot (alkalmazást), majd mentsétek el azt a *Feladatlap4* mappába.

Az alkalmazás egy személy havi költségvetését kezeli. A billentyűzetről a következő adatokat kell beolvasni:

- a bevételek száma és az egyes bevételek értéke;
- a kiadások száma és az egyes kiadások összege.

A program jelenítse meg:

- bevételek összegét;
- kiadások összegét;
- végső egyenleg (pozitív, negatív vagy nulla);
- a bevételekhez viszonyított kiadások arányát.



Önértékelés!

Töltsétek ki a válaszokat egy *onertekeles4.docx* nevű fájlban, olyan jelöléseket használva, mint: *1 – igen, 2 – nem, 3 – nem vagyok biztos*. Mentsétek el a fájlt a *Feladatlap4* mappába.

	A lecke végén képes vagyok...	*Igen*	*Nem*	*Nem tudom*
1.	használni a lineáris (szekvenciális) szerkezetet Pythonban			
2.	használni az elágazó szerkezetet Pythonban			
3.	használni az ismétlő szerkezetet Pythonban			
4.	alkalmazni a break és continue utasításokat			

5. Nehézségeim voltak a következőkben: …

Ne felejtsétek el értékelni a viselkedéseteket és a tevékenységeteket az óra során!
(a 7. oldalon található viselkedés- és tevékenységfigyelő napló segítségével)

A lecke végén így éreztem magam:

1. Fejezetvégi felmérés

Kutass!

(Maximális pontszám: 100 pont)

Vizsgáljátok meg a mellékelt programot, és válaszoljatok az alábbi kérdésekre. A válaszaitokat a ***kutass1.docx*** nevű fájlba írjátok:

```
kerdesek = {
   "Mi Bulgária fővárosa?": "Szófia",
   "Mi Magyarország fővárosa?": "Budapest",
   "Mi Görögország fővárosa?": "Athén",
   "Mi Horvátország fővárosa?": "Zágráb",
   "Zágráb?": "Ljubljána"
}
k = 0
for kerdes, felelet in kerdesek.items():
   aux = input(kerdes + " ").strip().capitalize()
   if aux == felelet:
       print("Helyes válasz!")
       k = k + 1
   else:
       print(f"Téves válasz!!! {felelet} a helyes válasz.")
print(f"Végeredmény: {k}/{len(kerdesek)}")
```

1. Azonosítsátok a program bemeneti adatait, kimeneti adatait és segédadatait.

2. a. Az elágazó szerkezet melyik ága fut le, ha a felhasználó helytelen választ ad meg? Írjátok le, mi történik ebben az ágban!

b. Hogyan módosítanátok a programot úgy, hogy a felhasználó válasza nagy- és kisbetűktől, valamint ékezetektől függetlenül elfogadott legyen? (például: az „Igen”, „igen” és „IGEN” kifejezéseket egyformán kezelje)

3. Adjatok meg két tesztesetet: egyet, amely az adatok helyességének ellenőrzését vizsgálja, és egyet, amely az algoritmus helyes működését ellenőrzi.

4. Javasoljatok moduláris felépítést a program számára! Nevezzetek meg legalább két modult és ismertessétek azok feladatát.

Javaslatok: Dolgozzatok 2–3 fős csapatokban. Írjátok át, módosítsátok a programot úgy, hogy legalább 10 kérdést tartalmazzon. Módosítsátok a végső eredmény kiszámításának módját is úgy, hogy a program a megszerzett pontszámot 100 pontos skálán jelenítse meg, ahol minden kérdés 10 pontot ér. Másoljátok be a programot az ***kutass1.docx*** fájlba.

Értékelési irányelvek:
- A ***Kutatás*** mappa tartalmazza a ***kutass1.docx*** állományt – **5p**
- Az **1**, **2**, **3**, **4** követelmények teljesítése – **60p** (minden követelményre **15p**)
- A program átírása és tesztelése – **25p**
- A pontszám meghatározására vonatkozó képlet módosítása – **10p**

Ellenőrizd a tudásodat!

(Pontszám: 90 pont + 10 pont hivatalból)

Követelmény: Oldjátok meg az alábbi feladatokat a ***tudas_ellenorzes1.docx*** fájlban. Az összes létrehozott fájlt mentsétek a ***Fejezet1*** mappába.

1. Állapítsd meg az alábbi állítások igazságértékét: (5p × 6 = 30p)

a) A **%** operátor két egész szám osztásának hányadosát adja vissza.

b) A ***break*** utasítás megszakítja egy ciklus végrehajtását.

c) A Python nyelv lehetővé teszi karakterláncok összefűzését a **+** operátor segítségével.

d) Az ***input()*** függvény alapértelmezés szerint ***int*** típusú értéket ad vissza.

e) Az ***if*** szerkezet nem létezhet else ág nélkül.

f) A ***3 + 5.0*** kifejezés adattípusa ***float***.

2. Határozd meg, hogy mit írnak ki az alábbi kódrészletek végrehajtásukkor: (5p × 4 = 20p)

a)
```
a = 12
a += 2 * 5
print(a)
```

b)
```
for i in range(1, 5):
    print(i, end=" ")
```

c)
```
x = 5
if x % 2 == 0:
    print("*")
elif x > 10:
    print("**")
else:
    print("***")
```

d)
```
print(len("A Python nyelv"))
```

3. Készíts Python-programot ***jatekos_szamok.py*** néven: (4 × 10p = 40p)

Egész számokat olvasunk be a 0 érték megjelenéséig. Meg kell határozni: **a)** a sorozat legnagyobb és legkisebb elemét, valamint ezek számtani átlagát. **b)** hány pozitív és hány negatív szám található a sorozatban a 0 érték megjelenéséig. **c)** a sorozat páros elemeinek összegét és a páratlan elemek szorzatát. **d)** hány olyan egymást követő számpár van, amelynek az utolsó számjegye megegyezik.



Projekt

(Pontszám: 100 pont)

Készíts egy projektet, amely egy hétköznapi problémára kínál megoldást. A projektet Python nyelven valósítsd meg. Lehetséges témák például: családi költségvetés tervezése, kérdőív vagy kvízjáték készítése, mértékegység-átváltó program fejlesztése stb. Mentsd a projektet a ***projekt_tanulo_neve*** nevű mappába, és a projekt fájlját is ezzel a névvel hozd létre.

1. **A projekt elemzése** – mutasd be a megoldandó problémát, majd csoportosítsd a projektben használt adatokat bemeneti adatokra, kimeneti adatokra és segédadatokra.

2. **Tervezés** – készíts egy legalább két modulból álló modultervet vagy folyamatábrát. A diagramon tüntesd fel a program főbb funkcióit, és röviden ismertesd az egyes modulok feladatát.

3. **Megvalósítás** – ezt a Python nyelven írt program képviseli, amely legalább egy elágazó szerkezetet és egy ismétléses szerkezetet tartalmaz.

4. **Tesztelés** – dokumentálj legalább 3 bemeneti teszt-esetet és azok várható vagy kapott eredményét. Az egyik teszt hibás adatokat tartalmazzon, a másik egy határesetet mutasson be, a harmadik pedig normál, érvényes adatokkal igazolja a program helyes működését.

5. **Következtetések** – ismertesd, milyen nehézségekkel találkoztál a program tervezése és megvalósítása során, milyen előnyei vannak az általad javasolt tervnek, valamint azt is, hogy mit fejlesztenél vagy javítanál tovább a programon.

(Értékelés: **10p**/minden követelményre, **20p**/a program elkészítése, **20p**/működőképesség, **10p**/kreativitás, **5p**/hasznosság, **5p**/tervezés)



Portfólió

(Pontszám: 100 pont)

A ***Fejezet1*** mappában található portfóliód elvárt tartalma:

- ***Feladatlap1*** mappa, állományok***: valaszok.docx, onertekeles1.docx***
- ***Feladatlap2*** mappa, állományok: ***valaszok.docx***, ***onertekeles2.docx***
- ***Feladatlap3*** mappa, állományok: ***valaszok.docx***, ***onertekeles3.docx***
- ***Feladatlap4*** mappa, állományok: ***alkalmazas.py***, ***onertekeles4.docx***
- ***Projekt1*** mappa, állományok: ***kutass1.docx***, ***projekt.docx***
- a ***tudas_ellenorzes1.docx***, , ***jatekos_szamok.py*** állományok.

Értékelés módja:

- Megvan a Fejezet1 mappa a megadott könyvtárszerkezettel – **20p**
- A mappákban megvan:
  - A fájlok több mint 95%-a – **80p**
  - A fájlok 75–94%-a– **60p**
  - A fájlok 50-74%-a – **40p**
  - A fájlok 25-49%-a – **20p**
  - 25% alatt– **0p**

2. Fejezet – Problémamegoldási stratégiák. Numerikus adatok feldolgozása

5. Lecke – Egy szám számjegyeivel végezhető műveletek

A természetes számok számjegyeinek feldolgozása az algoritmika egyik fontos problémakörét képezi. Az ebbe a kategóriába tartozó algoritmusok a helyiértékes számrendszer tulajdonságait használják ki: minden számjegy kinyerhető egész osztással és maradékos osztás (modulo) művelettel.

5.1. Természetes számok helyiértékes felírása

Bármely természetes számot 10-es számrendszerben írt számjegyek sorozataként lehet értelmezni. Például a 4732 szám jelentése: négyezer-hétszáz-harminckettő. Értéke a következőképpen bontható fel:

Számjegy	4	7	3	2
Helyiérték (balról)	ezresek (10^3)	százasok (10^2)	tízesek (10^1)	egyesek (10^0)
Helyiérték szerinti érték	$4 \times 1000 = 4000$	$7 \times 100 = 700$	$3 \times 10 = 30$	$2 \times 1 = 2$

A helyiértékek összege: 4000 + 700 + 30 + 2 = 4732. Ez a 10-es számrendszerbeli helyiértékes felbontás, amelyet minden algoritmus esetén használni fogunk, ebben a leckében.

5.2. Egy szám számjegyeinek elérése

Az előző fejezetben megtanultátok, hogyan lehet a % operátor segítségével megkapni egy szám utolsó számjegyét, valamint hogyan lehet a // operátor segítségével törölni egy szám utolsó számjegyét.

Lépések egy szám számjegyeinek bejárásához

A természetes számok számjegyeinek feldolgozásához két alapvető aritmetikai műveletet ismételünk:

- 10-zel való osztás maradéka (n % 10) → ***kinyeri a szám utolsó számjegyét***
- egész osztás 10-zel (n // 10) → ***eltávolítja az utolsó számjegyet***, lerövidítve a számot

⚠ **Figyelem!** Ha `n = 0`, a `while n > 0` ciklus egyáltalán nem fut le. Ezt az esetet mindig külön kell kezelni a ciklus előtt, egy `if n == 0` feltétellel (lásd a 4. lecke — A helyesség ellenőrzése).

Szám számjegyeinek bejárása jobbról balra

Ha ismételten alkalmazzuk a két műveletet – az utolsó számjegy kinyerését és annak törlését –, akkor sorban hozzáférhetünk az összes számjegyhez. A **while n > 0** ciklus akkor ér véget, amikor már nincs feldolgozandó számjegy. Kövesd végig lépésről lépésre, hogy mi történik a ciklus minden egyes iterációjában, az alábbi táblázatban:

```
n = int(input("n: ")) # beolvassuk n-t
if n == 0:
  print (0)
while n > 0:
  c = n % 10 #utolsó számjegy
  print (c)  #feldolgozzuk a számjegyet
  n //= 10   #töröljük az utolsót
```

Számjegyek kinyerési folyamatának szimulálása

Iteráció	n	c = n % 10	n //= 10 (után)	Kiírt számjegy
1	473	473 % 10 = 3	473 // 10 = 47	3
2	47	47 % 10 = 7	47 // 10 = 4	7
3	4	4 % 10 = 4	4 // 10 = 0	4
STOP	**0**	**a while n > 0 ciklus feltétele hamis**	**---**	**A ciklus véget ért**

Szám számjegyeinek bejárása balról jobbra

Ha a számjegyek sorrendje fontos, azokat balról jobbra kell bejárni. Ehhez addig nyerjük ki az első számjegyet és távolítjuk el a számból, amíg p > 0, hogy a végén lévő 0-ákat is megőrizzük.

```
p = 10**9     # n legtöbb 10 számjegyű
while p > n:
  p = p // 10
while p > 0:
  c = n // p # első számjegy
  print (c)  # számjegy feldolgozása
  n %= p   # töröljük az elsőt
  p //= 10 # lépésváltás
```

Alkalmazások — problématípusok

A számjegyfeldolgozó algoritmus használható egy szám számjegyei összegének, szorzatának, számjegyeinek számának kiszámítására, a legnagyobb számjegy megkeresésére, illetve adott feltételnek megfelelő számjegyeknek kiválasztására. Gyakorlatban a személyi azonosítószám (CNP) vagy a bankkártya kódjának ellenőrzésére is használhatjuk.

Pl. Az alábbi algoritmus kiírja az **n** szám számjegyeinek összegét. *Például: ha* **n = 245,** az eredmény ***11*** lesz.

```
n = int(input("n: "))
if n == 0: print("Szamjegy-db:",0)# speciális eset
else:
  osszeg = 0
  while n > 0:
    osszeg += n % 10 # hozzáadja az utolsó számjegyet
    n //= 10         # törli az utolsó számjegyet
  print("Szamjegyek osszege:", osszeg)
```

5.3. Szám alkotása számjegyek hozzáadásával

Egy meglévő számból új számot alkothatunk úgy, hogy egy számjegyet a végére (jobbra) vagy az elejére (balra) illesztünk.

Számjegy hozzáadása jobbról

Ha az n szám jobb oldalához hozzáadjuk a c számjegyet, akkor egy eggyel több számjegyből álló számot kapunk. Minden korábbi számjegy egy helyiértékkel balra kerül (a helyiértéke 10-szeresére nő), az új számjegy pedig az egyesek helyére kerül.

```
n = 456
c = 9
n_uj = n * 10 + c
print(n_uj)   # 4569
```

Képlet: n_uj = n × 10 + c

Egy szám tükörképe

Egy természetes szám tükörképe az a szám, amelyet úgy kapunk, hogy a számjegyek sorrendjét megfordítjuk. Például az 1234 tükörképe 4321, a 1200-é pedig 21 (a kezdő nullák elvesznek a számábrázolás során).

```
n = int(input("n = "))
inv = 0
while n > 0:
  inv = inv * 10 + n % 10
  n //= 10
print(inv)
```

Az algoritmus lépései: A számból kinyerjük az egyesek számjegyét az n % 10 művelettel, ezt hozzáadjuk az felépített tükörszám végéhez, majd a feldolgozott számjegyet eltávolítjuk az n // 10 művelettel. A folyamat addig ismétlődik, amíg vannak számjegyek.

Számjegy hozzáadása balról

Hozzáadva az **n** szám bal oldalához a **c** számjegyet, az új számjegy lesz a legnagyobb helyiértékű. Először meghatározzuk a 10-es számnak a helyérték-rendszerben megfelelő hatványát, majd kiszámítjuk az új számot.

Pl. Ha a 456-hoz balra hozzáadjuk a 7-es számot, először meghatározzuk a 10 megfelelő hatványát, p = 1000. Az új számot így számoljuk ki: 7 * 1000 + 456 = 7456.

```
n = int(input()) # 456
c = int(input()) # 7
p = 1
while p <= n:
  p *= 10
n_nou = c * p + n  # 7*1000+456
print(n_nou)    # 7456
```

Képlet: n_uj = c × 10x + n

5.4. A helyesség ellenőrzése –megfelelő tesztek, gyakori hibaforrások

Minden algoritmust a szélsőséges esetekben is ellenőrizni kell. Ha **n = 0**, a while ***n > 0*** ciklus egyáltalán nem fut le: A számjegyek összege 0 lenne (helyes), de a számjegyek száma helytelen (ennek 1-nek kellene lennie). A nullákkal végződő szám esetén a tükrözés elveszti a 0-kat (4300 → 34 → 43).

Feladatlap 5

Készítsetek egy *Feladatlap5* mappát. Oldjátok meg az alábbi feladatokat, a helyes válaszokat írjátok egy *valaszok.docx* dokumentumba.

1. Legnagyobb számjegy.

Módosítsd a Legnagyobb számjegy algoritmust úgy, hogy az a legkisebb páratlan számjegyet írja ki. Ha a számnak nincs páratlan számjegye, írja ki a *"Nem létezik" üzenetet. Példa:* n = 2301 esetén, az eredmény 1, míg n = 240 esetén a " Nem létezik" üzenet jelenik meg.

```
n = int (input ())
cmax = 0
while n > 0:
  c = n % 10  # megkapjuk az utolsó szj-t
  if c > cmax:
    cmax = c
  n //= 10    # törli az utolsó számjegyet
print (cmax)
```

```
n = int (input ())
nr_szamjegy = 1
while n > 9:
  c = n % 10
  nr_szamjegy += 1
  n //= 10
print( nr_szamjegy )
```

2. ***Számjegyek száma***

Módosítsd az algoritmust úgy, hogy írja ki a 3-mal osztható számjegyek számát.

3. Adott a mellékelt algoritmus, amely beolvassa az **n** nullától különböző természetes számot, és kiírja annak nem nulla számjegyeinek szorzatát (**P**), illetve azoknak a számjegyeknek az összegét (**S**), amelyek előtt 3-as számjegy áll.

Módosítsátok úgy az algoritmust, hogy azokat a számjegyeket számolja meg, amelyeket nullás számjegy követ**.**

Példa: n = **1035402** esetén az 1-es és 4-es számjegyeket követi 0, tehát a kiírt eredmény 2.

```
n = int (input ())
p = 1
s = 0
while n > 0:
  c = n % 10
  if c != 0:
    p = p * c
  if n > 9 and n // 10 % 10 == 3:
    s += c
  n = n // 10
print( p )
print ( s )
```

Javasolt feladatok - Számjegyek bejárása

1. Olvassunk be egy **n** természetes számot. Írassuk ki a szám számjegyeinek összegét és darabszámát. *Példa*: n = 12345 esetén kiírt eredmény s = 15, db = 5; n = 0 esetén s = 0, db = 1 (sajátos eset!).

2. Olvassunk be egy **n** természetes számot és egy **k** számjegyet. Határozzuk meg, hány számjegye egyenlő **k**-val.

3. Ellenőrizzük, hogy egy billentyűzetről beolvasott szám palindrom-e (balról és jobbról olvasva ugyanaz).

4. Olvassunk be egy ***n*** természetes számot. Írjuk ki az ***n*** páratlan számjegyeiből kapott számot, melynek számjegyei balról jobbra haladva növekvő sorrendben vannak. *Példa*: ***n*** = 429353 esetén a kiírt érték 3359.

5. Mária és Mihály jelszavakat küldenek egymásnak a következőképpen: minden számjegyet lecserélnek a saját és a következő számjegy összegének utolsó számjegyére, az utolsó számjegy változatlan marad.

Példa.: n = 1934 → 1+9=10, 9+3=12, 3+4=7, 4→4 → a jelszó 0274.

6. ***Mini projekt:*** Készítsetek projektet két, a való életben használt algoritmusról, amelyek adatfeldolgozást alkalmaznak (pl.: Luhn-algoritmus, Caesar-rejtjel stb.). A projekt kell tartalmazza az algoritmus felhasználási területének bemutatását, a működésének lépésről lépésre történő leírását, valamit a megvalósítását Python programozási nyelvben.

Önértékelés! Mit tanultam?

[ISMERET] [MEGÉRTÉS] [ALKALMAZÁS] [ELEMZÉS] [ÉRTÉKELÉS] [ALKOTÁS]

ISMERET

1. Társítsd össze:

`n // 10`	az n szám végére fűzi az 5-ös számjegyet
`n % 10`	kitörli n utolsó számjegyét
`n * 10`	az n szám végéhez ad egy 0-ás számjegyet
`n = n * 10 + 5`	megkapjuk az n szám utolsó számjegyét

MEGÉRTÉS

2. Adott a mellékelt algoritmus. Mit ír ki ***n = 256*** esetén?

a. 256; b. 13;

c. 62; d. 652.

```
n = 256
s = 0
while n != 0:
  s = s + n % 10
  n = n // 10
print(s)
```

ALKALMAZÁS

3. Egészítsd ki a hiányzó részt a mellékelt programrészben, amely megszámolja egy ***n*** szám páros számjegyeit:

a. c % 2 == 0; b. c % 2 == 1;

c. n % 2 == 0; d. c == 0.

```
n = 12045
nr = 0
while n != 0:
  c = n % 10
  if ____________:
    nr = nr + 1
  n = n // 10
print(nr)
```

ELEMZÉS

4. Elemezd az alábbi algoritmust és állapítsd meg, mit számol, feltételezve, hogy a számjegyek pozíciói 1-től vannak számozva:

a. a páros számjegyek szorzatát

b. a páros pozíciókon álló számjegyek szorzatát, jobbról balra számozva

c. a páros pozíciókon álló számjegyek szorzatát, balról jobbra számozva

d. a páratlan pozíciókon álló számjegyek szorzatát, jobbról balra számozva.

```
p = 1
d = 1
while n > 0:
  c = n % 10
  if d % 2 == 0:
    p = p * c
  d = d + 1
  n = n // 10
print (p)
```

ÉRTÉKELÉS

5. Egy tanuló azt állítja, hogy az alábbi algoritmus helyesen ellenőrzi, hogy az ***n*** szám palindrom-e. Igaza van vagy sem? Indokold meg, és azonosítsd azt az esetet, amikor hibázik. Az algoritmus mégis működik egy speciális esetben. Melyik az?

```
n = int (input())
ogl = 0
while n > 0:
  c = n % 10
  ogl = ogl * 10 + c
  n = n // 10
if ogl == n:
  print("IGEN")
else:
  print("NEM")
```

ALKOTÁS

6. Adott egy ***n*** természetes szám. Írj egy algoritmust, amely előállít egy „jelszót" úgy, hogy felváltva kiveszi az ***n*** **első** és **utolsó** számjegyét, ebben a sorrendben, amíg már nem maradnak számjegyek. A kivett számjegyeket összefűzzük és így kapjuk a jelszót. ***n*** = 12345 esetén, a jelszó 15243

Javítókulcs: 2p (hivatalból) + 0.5p (1) + 1p (2) + 1p (3) + 0.5p (4) + 2p (5) + 2.5p (6)

	A lecke végén ...	***Igen***	***Nem***	***Nem biztos***
1.	meg tudom határozni egy szám utolsó számjegyét, törölni a % és // operátorokkal.			
2.	végig tudom járni a szám összes számjegyét és kiszámolni azok összegét, szorzatát és darabszámát.			
3.	tudom, hogyan lehet egy számjegyet jobbról és balról is hozzáadni.			
4.	fel tudom építeni egy szám tükörképét.			
5.	ki tudok szűrni bizonyos feltételeknek (páros, páratlan, nem nulla) eleget tevő számjegyeket, és szelektíven feldolgozni őket.			
6.	be tudom azonosítani a szélsőséges eseteket, és kezelni azokat.			

7. Nehézségeim voltak a következőkben: …

Ne felejtsétek el értékelni a viselkedéseteket és a tevékenységeteket az óra során!
(a 7. oldalon található viselkedés- és tevékenységfigyelő napló segítségével)

A lecke végén így éreztem magam:

6. Lecke – Oszthatósági algoritmusok: Egy szám osztói; Prímszám-ellenőrzés

6.1. Egy szám osztói

Mi az osztó?

Egy ***d*** természetes szám osztója az ***n***-nek (***n≠0***) ha ***n % d == 0***, vagyis ***d*** maradék nélkül osztja ***n***-et. Az ***n*** osztóinak halmazát ***D(n)***-el jelöljük.

Pl. D(12) = {1, 2, 3, 4, 6, 12} | D(7) = {1, 7} | D(36) = {1, 2, 3, 4, 6, 9, 12, 18, 36}

Alapalgoritmus – bejárás 1-től n-ig

Legyen ***n*** egy természetes szám. Szeretnénk kiírni sorra az összes osztóját.

Megoldási ötlet: minden lehetséges ***d*** értéket megvizsgálunk ***1***-től ***n***-ig, és akkor írjuk ki, ha osztója ***n***-nek.

```
n = int(input())
d = 1
while d <= n:
    if n % d == 0:
        print(d)
    d = d + 1
```

Optimizált algoritmus – bejárás 1-től $\sqrt{n}$-ig

Legyen ***n*** egy nem nulla természetes szám. Határozzuk meg az ***n*** szám osztóinak összegét.

Megj. Ha ***d | n***, akkor igaz az, hogy ***(n // d) | n***. Az osztók párban vannak ***(d, n//d)***. Elég végigvizsgálni ***d***-t ***1***-től $\sqrt{n}$-ig, így a műveletek száma n-ről $\sqrt{n}$-re csökken. Van azonban egy speciális eset, a négyzetszámok esetén a gyököt csak egyszer kell hozzáadni.

```
#lassú variáns
n = int(input())
s = 0
d = 1
while d <= n:
  if n % d == 0:
    s = s + d
  d = d + 1
print (s)
```

```
#optimális variáns
n = int(input())
s = 0
d = 1
while d * d < n:
  if n % d == 0:
    s = s + d + n // d
  d = d + 1
if d * d == n:
  s = s + d
print (s)
```

Algoritmikai összehasonlítás

Vizsgáljuk meg a két variánst, az osztók összegének meghatározására:

Kritérium	1. Variáns: 1…n	2. Variáns: 1…$\sqrt{n}$	Előnyösebb variáns
Műveletek száma	n művelet	≈ $\sqrt{n}$ művelet	V2
Lépésszám n = 1 000 000-ra	1 000 000 lépés	1 000 lépés	V2
Implementáció	Egyszerű	Kissé összetettebb	V1
d^2 = n eset	Nem probléma	Ellenőrzést igényel	V1

Gyakorlati alkalmazások – A számok osztóinak valós életbeli használata:

- Egy ***L*** cm hosszú szalag feldarabolása, veszteség nélkül: a darabok lehetséges hosszai = ***L*** osztói;
- ***N*** füzet kiosztása: lehetséges tanulók száma = ***N*** bármely ***d*** osztója (minden tanuló kap ***N/d*** füzetet).
- ***N*** termék csomagolása minimális számú csomagokba, de legyen legalább 2 egyforma darabszámú csomag: keressük az ***N***-nek azt a ***d > 1*** osztóját, amely maximalizálja az ***N/d***-t (termék/csomag), tehát a legkisebb **d > 1**-et választjuk.

6.2. Prímszám-ellenőrzés

Mi az a prímszám

<def/> **Prímszám**: olyan ***n > 1*** szám, amelynek nincs más osztója csak az **1** és ***n***. *Példák:* 2, 3, 5, 7, 11, 13, 97, 101.
Összetett szám: olyan ***n > 1*** szám, amelynek van legalább egy ***d*** osztója, ahol ***1 < d < n***. *Példák*: $4=2^2$, $6=2\times3$, $15=3\times5$.

Speciális esetek: a ***0*** és ***1*** nem prímszámok és nem összetett számok – ezek nem tartoznak egyik kategóriába sem.

Prímszám megállapítása

Fentebb bemutattuk, hogy egy szám összes osztóját megkaphatjuk, ha csak 1-től $\sqrt{n}$, terjedő számokat vizsgáljuk, (d és n/d párokban). Most ezt az észrevételt alkalmazzuk: egy szám akkor és csakis akkor prímszám, ha ***nincs egyetlen d osztója sem, $2 \le d \le \sqrt{n}$***. Nincs értelme $\sqrt{n}$ -nél nagyobb osztót keresni, mert ha létezne ilyen d, akkor n/d egy másik osztója lenne, amely $\le \sqrt{n}$. Az elemzéshez bemutatjuk a lassú verziót is. Mindkét variáns d=2-vel indul és lépésenként halad. A megállási feltétel az, ami a hatékonyságot megkülönbözteti:

Python – V1 bejárás 1-től n-ig (lassú)

```
n = int(input('n = '))
d = 2
# Megállás: találtunk egy osztót VAGY d
# elért n-ig
while d < n and n % d != 0:
    d += 1
if d == n and n > 1: # ha d==n, n prim
# (egyetlen osztót sem találtunk)
  print('PRIM')
else:
  print('OSSZETETT')
# Legrosszabb eset: ha n prim => d végig
# megy n-1-ig
```

Python – V2 bejárás 1-től $\sqrt{n}$-ig (optimizált)

```
n = int(input('n = '))
d = 2
# Megállás: d*d > n (meghaladtuk sqrt-t)
# VAGY kaptunk egy osztót
while d * d <= n and n % d != 0:
  d += 1
if d * d > n and n > 1: # egyetlen osztót
# sem kaptunk
  print('PRIM')
else:
  print('OSSZETETT')
# Legrosszabb eset: ha n prim => d végig
# megy sqrt(n)-ig
```

Miért ekvivalens a d*d > n feltétel azzal, hogy «nincs egyetlen osztó sem $\sqrt{n}$-ig»?

A ciklus akkor áll le, ha:

(a) ***d*d > n*** – kimerítettünk minden jelöltet, ami $\le \sqrt{n}$ anélkül, hogy találtunk volna osztót, tehát ***n*** prím;

(b) ***n%d==0*** – találtunk egy osztót, tehát ***n*** összetett szám.

A ***d*d <= n*** feltétellel elkerüljük az ***sqrt()*** függvény használatát és így nagy értékekre is helyesen működik.

⚠ **Figyelem!** az ***n=1*** esetet külön le kell kezelni (1 nem prím). Az ***«n > 1»*** feltétel az ***if***-ben ezt biztosítja.

Algoritmikai összehasonlítás

n	Prím?	V1 lépései (1…n)	V2 lépései (1…$\sqrt{n}$)	V1/V2 arány
7	IGEN	5	2	2.5×
25	NEM	3	3	1× (egyenlő)
97	IGEN	95	8	~12×
1 009	IGEN	1 007	31	~32×
1 000 003	IGEN	~10^6	~1000	~1000×

Gyakorlati alkalmazások – Valós életbeli alkalmazások a prímszámokra

- Internetes biztonság: A prímszámokat adatvédelemre használják (ex. RSA). Nagyon könnyű összeszorozni két prímszámot, de az inverz művelet sokkal nehezebb. (Lásd a következő fejezetben a prímtényezős felbontást)
- Hashing: arra használják, hogy adatokat (szavakat, jelszavakat, fájlokat) gyorsan egy rövid „kóddá" alakítsanak.
- A prímszámok „alap építőkövei" minden számnak. A következő leckében látni fogjuk a számok prímtényezős felbontását.



Jegyezd meg!

✔ Az ***n*** osztói párban vannak ***(d, n//d)***, ezért elég $\sqrt{n}$-ig keresni.

✔ Egy szám akkor prím, ha egyetlen ***d*** osztója sincs, amelyre $2 \le d \le \sqrt{n}$.

✔ Speciális esetek: ***0*** és ***1*** nem prímek és nem is összetett számok.

✔ A ***d*d <= n*** feltételt előnyben részesítjük a ***d <= sqrt(n)*** feltétel használatával szemben a hatékonyság és pontosság miatt.



Feladatlap 6

Készítsetek egy *Feladatlap6* mappát. Oldjátok meg az alábbi feladatokat, a helyes válaszokat írjátok egy *valaszok.docx* dokumentumba.

1. Adott egy ***n*** természetes szám. Írjuk ki a szám osztóit növekvő sorrendben. Használjatok egy optimális algoritmust az osztók bejárására.

Megoldási ötlet: végigmegyünk az ***1*** -től a $\sqrt{n}$-ig terjedő intervallumon kétszer, először kiíratjuk $\sqrt{n}$-ig az osztókat, majd kiíratjuk ezek párjait.

Milyen sorrendben járjuk be az osztókat másodszor, és miért?

```
n = int(input())
# elöször kiíratjuk gyökéig az osztókat
# körülbelül sqrt(n) számot fogunk bejárni
d = 1
while d * d < n:
  if n % d == 0:
    print(d)
  d = d + 1
# kiíratjuk a gyök utáni osztókat
# és kiírjuk mindegyik párját
if d * d > n:    # ha meghaladjuk a gyököt
  d -= 1         # csökkentjük d-t
while d >= 1:
  if n % d == 0:
    print(n // d)
  d = d - 1
```

2. Egy **szám** (n>1) **tökéletes**, ha osztóinak összege (önmagán kívül) egyenlő magával a számmal. *Példa*: 6 = 1 + 2 + 3.

Megoldási ötlet: Kiszámítjuk az osztók összegét a leckében bemutatott algoritmus alapján, majd megvizsgáljuk, hogy a kapott összeg megegyezik-e ***n***-el.

Miért inicializáltuk az összeget 1-re? Írj egy ekvivalens algoritmust, amely az ***sqrt*** függvényt használja.

```
n = int(input())
s = 1
d = 2
# a d-vel gyök n-ig megyünk
while d * d < n:
  if n % d == 0:
    s = s + d + n // d
  d = d + 1
# itt a d vagy egyenlő lesz gyök(n)-el, ha n négyzetszám
# vagy nagyobb mint gyök(n), ellenkező esetben
if d * d == n:    # ha n teljes négyzet
  s = s + d       # hozzáadjuk a gyököt csak egyszer
if n == s:
  print('A szám tökéletes')
else:
  print('A szám nem tökéletes')
```

3. Határozzátok meg ***n*** legkisebb osztóját, amely egyben prímszám is.

4. Olvassunk be egy ***n*** számot. Döntsük el, az n számról, hogy prímszám négyzete-e, vagyis olyan szám, amely felírható egy prímszám négyzeteként. *Példa*: 9 prímszám négyzete, mert felírható 3^2-ként.

Javasolt feladatok

1. Írjatok egy programot, amely beolvassa ***n***-et és kiírja: ***n*** osztóinak számát, ***n*** legnagyobb osztóját, valamint ***n*** páratlan osztóinak összegét.

2. Egy ***L*** cm-es szalagot egyenlő darabokra kell vágni (természetes hosszúságú darabok, veszteség nélkül). Írjátok ki az összes lehetséges hosszúságot, ha a szalagot legalább ***3*** darabra vágjuk.

3. Olvassátok be ***N***-et. Határozzátok meg, hányféleképpen lehet ***N*** füzetet egyenlően elosztani legalább ***2*** és legfeljebb ***10*** tanuló között. Írjátok ki az összes lehetőséget (tanulók száma, füzetek száma/tanuló).

4. Egy ***n*** számot ***bővelkedőnek*** nevezünk, ha a valódi osztóinak összege nagyobb, mint ***n***. Írjátok ki az összes bővelkedő számot 500-ig.

5. Olvassunk be egy ***n*** természetes számot. Határozzátok meg a legkisebb prímszámot, amely szigorúan nagyobb, mint ***n***.

Önértékelés! Mit tanultam?

[ISMERET] [MEGÉRTÉS] [ALKALMAZÁS] [ELEMZÉS] [ÉRTÉKELÉS] [ALKOTÁS]

ISMERET

1. Az alábbi lehetőségek közül melyik adja meg helyesen az ***osztó*** definícióját?
a. d osztója n-nek, ha n % d > 0 b. d osztója n-nek, ha n % d == 0
c. d osztója n-nek, ha d % n == 0 d. d osztója n-nek, ha n // d == 0

MEGÉRTÉS

2. Mit ír ki a mellékelt algoritmus ha ***n = 12***?
a. 26 b. 28
c. 16 d. 12

```
n = int(input("n="))
s = 0
d = 1
while d * d < n:
    if n % d == 0:
        s = s + d + n // d
    d += 1
if d * d == n:
    s = s + d
print(s)
```

ALKALMAZÁS

3. Írj egy algoritmust, amely beolvas egy ***n*** természetes számot, majd kiírja a páratlan osztóinak összegét?

ELEMZÉS

4. Adott két algoritmus, amely egy ***n*** számról megállapítja, hogy prím-e. Elemezd őket és azonosíts egy különbséget és egy hasonlóságot az osztók vizsgálata szempontjából.

```
# A Algoritmus
d = 2
prim_e = True
while d <= n - 1:
  if n % d == 0:
    prim_e = False
  d += 1
```

```
# B Algoritmus
d = 2
prim_e = True
while d <= n // 2:
  if n % d == 0:
    prim_e = False
  d += 1
```

ÉRTÉKELÉS

5. Egy tanuló prímellenőrzést végez while ***d <= n // 2*** feltétellel. Ez jobb, vagy gyengébb, mint a while ***d * d <= n***? Indokold a lépések számával, ***n = 97*** esetén.

ALKOTÁS

6. Írj egy algoritmust, amely beolvas egy ***n*** természetes számot, és megállapítja, hogy az ***n*** „majdnem prím"-e(felírható két prímszám szorzataként, amelyek nem feltétlenül különbözőek).

Javítókulcs: 2p (hivatalból) + 0.5p (1) + 1p (2) + 1p (3) + 0.5p (4) + 2p (5) + 2.5p (6)

	A lecke végén ...	***Igen***	***Nem***	***Nem biztos***
1.	meg tudom magyarázni, az osztó fogalmát és tudok példákat adni rá.			
2.	meg tudom írni az algoritmust ***1…n*** Python-ban.			
3.	meg tudom írni az optimizált algoritmust ***1…$\sqrt{n}$*** rendezett kiírásban.			
4.	értem, hogy miért hatékonyabb a 2. variáns és össze tudom hasonlítani a lépések számát.			
5.	tudom alkalmazni az osztók meghatározásának algoritmusát, gyakorlati feladatok megoldására.			
6.	tudom alkalmazni és megmagyarázni a tökéletes szám algoritmust.			

7. Nehézségeim voltak a következőkben: …

Ne felejtsétek el értékelni a viselkedéseteket és a tevékenységeteket az óra során!
(a 7. oldalon található viselkedés- és tevékenységfigyelő napló segítségével)

A lecke végén így éreztem magam:

7. Lecke - Prímtényezőkre bontás; LNKO, LKKT (Euklidész)

7.1. Prímtényezőkre bontás

1. Esettanulmány: n prímtényezőinek kiíratása

Az előző leckében ***d***-vel bejártuk ***1***-től $\sqrt{n}$-ig az értékeket ahhoz, hogy megkapjuk ***n*** összes osztóját. Most azt vizsgáljuk: hogyan határozzuk meg csak a ***prímosztókat***, anélkül, hogy külön leellenőriznénk az osztókról, hogy prímek-e? ***A kulcsötlet***: miközben találunk egy osztót, el is ***távolítjuk*** a számból – addig osztjuk vele az ***n***-et, amíg már nem osztja. Így mire egy összetett ***d*** értékhez érünk, annak minden prímtényezőjét már korábban kivontuk, ezért ***d*** már nem fogja többet osztani a megmaradt n értéket – ***az összetett számok így automatikusan kiesnek.***

Pl.

Példa lépésről lépésre: n = 180

d=2: 180÷2=90, 90÷2=45 → 2^2 prímtényező, n értéke 45 lesz

d=3: 45÷3=15, 15÷3=5 → 3^2 prímtényező, n értéke 5 lesz

d=4: 4*4=16 > 5, a ciklus leáll. n=5 > 1 → 5^1 prímtényező (garantáltan prím)

Eredmény: $180 = 2^2 \times 3^2 \times 5^1$ | 3 különböző prímtényező | lépések száma: ~4

Ez az ötlet képezi a következő algoritmus alapját: prímtényezőkre bontás.

2. Prímtényezőkre bontás

Az aritmetika alaptétele garantálja, hogy minden $n > 1$ természetes szám **egyértelműen** felírható prímszámok szorzataként: $n = p_1^{a_1} \cdot p_2^{a_2} \cdot \ldots \cdot p_n^{a_n}$. Például: $360 = 2^3 \cdot 3^2 \cdot 5^1$.

Eltérően az osztók keresésének algoritmusától, ahol ***n változatlan maradt***, itt ***n*** minden lépésben ***változik***: amikor találunk egy prímtényezőt, ismételt osztásokkal teljesen eltávolítjuk, majd a keresést a megmaradt számon folytatjuk. A ***360*** esetén: a ***2***-t háromszor emeljük ki (360→180→90→45), majd a tényezőket már a ***45***-ben keressük, nem a ***360***-ban.

A lassú változat: Kezdjük ***d = 2***-vel és növeljük ***d***-t addig, amíg ***n*** el nem éri ***1***-et. Minden ***d***-nél, ha osztja ***n***-et, teljesen kiemeljük, és kiírjuk a tényezőt a hatványkitevővel együtt. Működik, de egy nagyon nagy prímszám esetén ***d***-nek egészen az ***n***-ig kell eljutnia.

Optimizált változat: Miután az összes tényezőt kiemeltük, az n-ben maradhat még egy fel nem fedezett prímtényező - de **legfeljebb egyetlen egy**. Ennek oka a következő: ha két tényező ***p*** és ***q*** maradna, akkor közülük a kisebbik, ***d***, teljesítené a $d \times d \le p \times q \le n$, feltételt, tehát $d \le \sqrt{n}$ – és a ciklus már megtalálta volna, ***ellentmondás***. Ezért megállunk, amikor $d \times d > n$: mert minden tényezőt, amely kétszer vagy többször szerepel, már korábban kiemeltünk. Így az ellenőrzések száma ***n***-ről körülbelül $\sqrt{n}$ -re csökken.

Python – V1 bejárás 1…n (lassú)

```
n = int(input())
d = 2

while n > 1:
  if n % d == 0:
    p = 0
    while n % d == 0:
      n //= d
      p += 1
    print(f'{d}^{p}', end=' ')
  d += 1
# A legrosszabb eset: ha n prím => d elmegy
# egészen n-1-ig
```

Python – V2 bejárás 1…√n (optimizált)

```
n = int(input('n = '))
d = 2
while d * d <= n:     # bejárjuk √n-ig
  p = 0               # a d hatványa
  while n % d == 0:   # amíg d osztja n-et
    n //= d           # osztjuk n-et d-vel
    p += 1            # növeljük a hatványt
  if p > 0:           # d osztó
    print(f'{d}^{p}', end='\n')
  d += 1
if n > 1:             # maradt egy prímosztó
  print(f'{n}^1', end='')
# Legrosszabb eset: ha n prím => d elmegy √n-ig
```

n	Prímtényezős felbontás	Lassú lépések száma	Optimális lépések száma
84	$2^2 \times 3 \times 7$	82	8
360	$2^3 \times 3^2 \times 5$	358	18
97	97^1 (prím)	96	9
1000003	1000003^1 (prím)	1000002	~1000

3. A felbontás gyakorlati alkalmazásai

A prímtényezős felbontás számos alapvető algoritmus alapját képezi:

lnko és lkkt lkkt(a,b) = a közös és nem közös prímtényezők, maximális hatványaival vett szorzata. **Példa**: lkkt(12,18) = $2^2 \times 3^2 = 36$	**RSA kriptográfia** Az RSA biztonsága azon a nehézségen alapul, hogy egy nagy számot (két nagy prímszám szorzataként előállított számot) nehéz prímtényezőkre bontani.
Osztók száma és összegük A prímtényezős felbontásból kiindulva kiszámítható egy szám osztóinak száma, illetve ezek összege is.	**Törtek egyszerűsítése** Az lnko(a,b) meghatározható prímtényezős felbontás segítségével, majd az a/lnko és b/lnko kifejezések lesznek a tört egyszerűsített tagjai.

7.2. LNKO, LKKT

Euklideszi algoritmus– lnko

Az ***Euklideszi algoritmus*** a leggyakoribb klasszikus módszer a legnagyobb közös osztó (LNKO) kiszámítására két nem nulla szám esetén, anélkül, hogy szükség lenne a prímtényezős felbontásukra. Az eljárás azon alapul, hogy a közös osztók megmaradnak akkor is, ha a problémát egy egyszerűbb alakra redukáljuk.

LNKO: Két pozitív egész szám a és b esetén, a legnagyobb közös osztójuk, jelöljük lnko(a,b)-val vagy gcd(a,b)-vel, az a legnagyobb $d>0$, amely osztja a-t és b-t is.

Pl. lnko(12,8)=4 (a közös osztói 1, 2, 4) lnko(17,5)=1 (relatív prímek) lnko(48,18)=6	Hasznos tulajdonságok: lnko(a,0) = a, ha $a \geq 0$. lnko(a,b) = lnko(b,a) (kommutativitás). Ha d = lnko(a,b) akkor léteznek az x,y egész $ax+by=d$ (Bézout-azonosság).

Az eredeti ötlet – Kivonásos algoritmus

Euklidész (i.e. kb. 300) megfigyelte a következő elvet: Ha egy d szám osztja a-t és b-t, akkor osztja a különbségüket is:

$a = d \times x$

$b = d \times y$

$a - b = d(x - y)$

Ebből következik az alapvető szabály: ***lnko(a, b) = lnko(a - b, b)***. Az eljárás ismételt kivonásokkal folytatódik, amíg a két szám egyenlő nem lesz. Ekkor ez a szám lesz az ***LNKO***.

Pl.

lnko(48, 18):

48 – 18 = 30 → lnko(30, 18)

30 – 18 = 12 → lnko(12, 18)

18 – 12 = 6 → lnko(6, 12)

12 – 6 = 6 → lnko(6, 6) = 6

Optimizálás – Osztásos algoritmus

Az alapvető megfigyelés az, hogy az ismételt kivonások valójában a maradékos osztás műveletét jelentik. Ezért: 48 = 2 × 18 + 12 → 48 mod 18 = 12. Ezért a sok kivonás helyett közvetlenül a maradékos osztást használjuk: ***lnko(a, b) = lnko(b, a mod b)***

A közös osztók azért maradnak változatlanok, mert minden közös osztója ***a***-nak és ***b***-nek osztja az ***a − q·b*** maradékot is. Az algoritmus addig folytatódik, amíg a maradék ***0*** lesz, ekkor az utolsó nem nulla szám az ***LNKO***.	lnko(48, 18): 48 mod 18 = 12 → lnko(18, 12) 18 mod 12 = 6 → lnko(12, 6) 12 mod 6 = 0 → lnko(6, 0) = 6

A két módszer összehasonlítása:

- Kivonásos: 4 lépés
- Osztásos: 3 lépés

A különbség nagy számoknál drámai. *Példa*: lnko(1.000.000, 1)

- Kivonásos: 999.999 művelet
- Osztásos: 1 művelet (1.000.000 mod 1 = 0)

Intuitív magyarázat

Képzelj el két zsinórt: az egyik ***48*** cm, a másik ***18*** cm hosszú. Szeretnéd a legnagyobb egyenlő darabokra vágni őket.

– A **48** cm-es szalagból ***18*** cm-es darabokat vágunk, → marad ***12*** cm. Most van ***18*** cm-es darabunk és ***12*** cm-es.

– A ***18*** cm-esből ***12*** cm-es darabokat vágunk, → marad ***6*** cm. Most van ***12*** cm-es és ***6*** cm-es darabunk.

– A ***12*** cm-esből ***6*** cm-est vágunk → maradék ***0***.

Az utolsó darab, ami mindkettőt „pontosan osztja", a ***6*** cm, vagyis az ***LNKO***.

Gyakorlati tevékenység: Két különböző hosszúságú (egész szám) darabokat vágunk fel, úgy hogy a végén azonos méretű (egész szám) darabokat kapjunk.

Alkalmazások – más problémák a valós életből***:***

– *Anyagok elosztása:* Egy laborban van két drótdarab, egyik ***72*** cm, a másik ***120*** cm hosszúságúak. Azonos, lehető legnagyobb azonos hosszúságú darabokra kell vágni őket, maradék nélkül. Mi lesz a maximális hossz?

– *Tanulók csoportosítása:* Két osztályban van ***24*** és ***30*** tanuló. A tanár vegyes csoportokat akar létrehozni, azonos létszámmal, úgy hogy senki se maradjon ki. Mi lesz a legnagyobb csoportlétszám?

– *Feladatok elosztása fejezetekre*: Egy tankönyv két fejezetében van ***42***, illetve ***63*** feladat. Egyforma feladatsorokat kell készíteni úgy, hogy minden sorban ugyanannyi feladat legyen, maradék nélkül. Mi lesz a maximális feladatszám egy lapon?

– *Egy téglalap alakú felület lefedése maximális oldalszámú négyzetekkel* (például: térkövezés).

Python – Ismételt kivonással

```
a, b = map(int, input().split()) # a,b pozitív
while a != b:
  if a > b:
    a = a - b
  else:
    b = b - a
print(a)
```

Python – optimális

```
a, b = map(int, input()).split())# a,b pozitív
while b > 0:
  r = a % b
  a = b
  b = r
print(a)
```

Legkisebb közös többszörös

<def/>

Két szám ***legkisebb közös többszöröse*** az a legkisebb pozitív szám, amely mindkettőnek többszöröse. Más szavakkal, ez az első szám, ahol a két többszörös sorozat „találkozik", az ***a*** szám többszörösei és a ***b*** szám többszörösei. Hatékonyan kiszámítható az LNKO és LKKT közötti összefüggés segítségével:

LKKT(a, b) = a * b / LNKO(a, b)

Ennek oka, hogy az a×b szorzat tartalmazza mindkét szám összes tényezőjét kétszer, de az ***LNKO***-val való osztás eltávolítja a közös osztókat.

Intuitív magyarázat

Képzelj el két fogaskereket: az egyik ***4*** másodpercenként, a másik ***6*** másodpercenként „kattan”. Az első alkalom, amikor egyszerre kattannak az ***LKKT(4, 6) = 12*** másodperc, mert ***12*** az első közös többszöröse ***4***-nek és ***6***-nak (***4*** többszörösei: ***4, 8, 12***…; ***6*** többszörösei: ***6, 12, 18***…). Ezért, az LKKT azt mutatja meg, mikor szinkronizálódik két ismétlő folyamat.

Megfigyelhető, hogy az Euklideszi algoritmus megváltoztatja a kezdetben megadott számokat, ezért a legkisebb közös többszörös meghatározásához szükség van az eredeti értékek másolatainak megőrzésére, hogy azokat a végén újra fel lehessen használni.

Python – LKKT

```
a, b = map(int, input().split())
ca = a
cb = b
while b > 0:
    r = a % b
    a = b
    b = r
print(ca * cb // a)
```

Feladatlap 7

Készítsetek egy *Feladatlap7* mappát. Oldjátok meg az alábbi feladatokat, a helyes válaszokat írjátok egy *valaszok.docx* dokumentumba.

1. *Felbontás – legnagyobb prímtényező.*

a. Kövesd nyomon, ha ***n = 84*** és töltsd ki a mellékelt táblázatot:

n	d	p	dmax
84	...		

b. Az ***if n > 1: dmax = n*** utasítás miért van a ***while d * d <= n*** cikluson kívül?

```
n = int(input())
d = 2;
dmax = 0
while d * d <= n:
    p = 0
    while n % d == 0:
      n = n // d; p += 1
    if p > 0: dmax = d
    d += 1
if n > 1: dmax = n
print(dmax)
```

2. *Felbontás – a legnagyobb hatványon szereplő prímtényező.*

a. Mit fog kiírni az algoritmus, ha ***n = 84***? És ***n = 18*** esetén?

b. Vezessétek le lapra n=2-re és magyarázzátok meg, miért szükséges az if ***n > 1*** and ***1 >= pmax*** feltétel.

```
n = int(input())
d = 2;
pmax = 0
while d * d <= n:
    p = 0
    while n % d == 0:
      n //= d; p += 1
    if p >= pmax:
      pmax = p; dmax = d
    d += 1
```

3. *LNKO – kivonásos Euklideszi módszer*

```
a, b = map(int, input().split())
while a != b:
    if a > b: a = a - b
    else:     b = b - a
print(a)
```

a. Töltsétek ki lépésről lépésre a táblázatot (oszlopok: a | b | a>b? | művelet).

a	b	a>b?	művelet
	...		

b. Miért áll le a ciklus, ha ***a == b*** és nem akkor, amikor ***a == 0***?

4. *LNKO – osztásos Euklideszi módszer*

```
a, b = map(int, input().split())
while b != 0:
    a, b = b, a % b
print(a)
```

a. Töltsétek ki lépésről lépésre a táblázatot (oszlopok: a | b | a%b → lesz).

a	b	a%b → lesz
	...	

b. Mi történne, ha az ***a, b = b, a % b*** sort ***a = a%b***; ***a, b = b, a*** -ra cserélnénk?

5. ***n legkisebb prímosztója.*** Írjatok egy Python programot, amely kiírja ***n*** legkisebb prímosztóját. *Példa:* n=20 → 2; n=17 → 17. *Útmutatás: álljatok meg azonnal, amint megtaláltátok az első osztót.*

6. ***LKKT kiszámítása.*** Írjatok egy Python programot, amely kiszámolja ***LKKT(a, b) = a * b / LNKO(a, b)***. *Példa:* a=12, b=18 → LNKO=6 → LKKT=36. *Útmutatás: Használjátok az a // lnko * b alakot a túlcsordulás (overflow) elkerülésére.*

7. ***Irreducibilis (egyszerűsített) tört.*** Olvassatok be egy a/b törtet és írassátok ki az irreducibilis alakját az ***LNKO*** segítségével. *Példa:* 18/24 → 3/4; 36/48 → 3/4.

8. ***Prímtényezős felbontás – A variáns (sqrt(n)-ig) ill. B (n-ig)***

a. Vezessétek le mindkettőt ***n = 12*** és ***n = 13*** esetén. Ugyanazt az eredményt adják?

b. Becsüljétek meg a lépések számát ***n = 1.000.003*** (prím) esetén. Melyik variáns hatékonyabb?

```
# A:
szamlalo = 1
d = 2
while d*d <= n:
  if n%d==0:
    szamlalo += 1
    while n%d==0: n//=d
if n>1:
szamlalo += 1
```

```
# B:
szamlalo = 1
d = 2
while d <= n:
  if n%d==0:
    szamlalo += 1
    while n%d==0: n//=d
  d += 1
```

9. ***LNKO –A variáns (kivonásos) ill. B (osztásos)***

a. Vezessétek le mindkettőt ***a = 100***, ***b = 75*** esetén. Ugyanazt az eredményt adják?

b. Becsüljétek meg a lépések számát ***a = 1.000.000***, ***b = 1***. Melyik variáns hatékonyabb, és miért?

```
# A:
a = int(input())
b = int(input())

while a != b:
  if a > b:
    a = a - b
  else:
    b = b - a
print(a)
```

```
# B:
a = int(input())
b = int(input())

while b != 0:
  a, b = b, a % b
print(a)
```

10. ***LNKO – if...if az if...else helyett***

a. Helyes ez a változat? Vezessétek le ***a = 6***, ***b = 4*** esetén és azonosítsátok a problémát.

```
while a != b:
  if a > b: a = a - b
  if b > a: b = b - a  # if, nem else!
```

b. Magyarázzátok meg, miért vezet logikai hibához az ***else*** lecserélése egy második ***if***-re.

11. *András 4 naponta jár edzésre, Boti pedig 6 naponta. Egy osztálytársuk azt állítja, hogy* **LNKO(4,6)=2** *nap múlva fognak újra találkozni. Határozzátok meg, milyen számítást kellett volna elvégezni, és ellenőrizzétek az eredményt.*

12. ***n legkisebb prímosztója (teljes program).***

Írjatok egy programot, amely kiírja ***n*** legkisebb prímosztóját. Ezután bővítsétek ki úgy, hogy az összes különböző prímosztót is kiírja növekvő sorrendben.

13. ***Szinkronizált jelzőlámpák.***

Az ***A*** jelzőlámpa ***45*** másodpercenként vált, a ***B*** pedig ***60*** másodpercenként. Mindkettő zöld ***t=0*** pillanatban. Írjatok egy programot, amely meghatározza, hány másodperc múlva lesznek újra egyszerre zöldek. Általánosítsátok a feladatot n darab, billentyűzetről beolvasott időközre.

Útmutatás: LKKT(a,b,c) = LKKT(LKKT(a,b), c).

Önértékelés! Mit tanultam?

[ISMERET] [MEGÉRTÉS] [ALKALMAZÁS] [ELEMZÉS] [ÉRTÉKELÉS] [ALKOTÁS]

ISMERET

1) Az alábbiak közül melyik a ***36*** helyes prímtényezős felbontása?
a) $36 = 2 \cdot 18$ b) $36 = 2^2 \cdot 3^2$ c) $36 = 4 \cdot 9$ d) $36 = 2^3 \cdot 3$
Az alábbi állítások közül melyik ***igaz*** az LNKO-ra és az LKKT-ra?
a) LNKO(a, b) mindig nagyobb, mint a és b; b) LNKO(a, b) × LKKT(a, b) = a × b; c) LKKT(a, b) mindig kisebb, mint a + b; d) LNKO(a, b) = a ha a osztja b-t, különben LNKO(a,b) = b.

MEGÉRTÉS

2) Mit ír ki az alábbi program ***n = 12*** esetén?
a) 2
b) 3
c) 4
d) 6

```
n = 12
d = 2
while d * d <= n:
    p = 0
    while n % d == 0:
        n = n // d
        p += 1
    if p > 0:
        szamlalo += 1
    d += 1
if n > 1:
    szamlalo += 1
print(szamlalo)
```

Mit ír ki a mellékelt Euklideszi algoritmus ***a = 48***, ***b = 36*** esetén?
a) 48 b) 12 c) 6 d) 3

```
a = int(input())
b = int(input())
while b > 0:
  a, b = b, a % b
print(a)
```

ALKALMAZÁS

3) Írjatok egy programot, amely beolvassa ***n*** értékét és kiírja, hogy a ***2***-es tényező milyen hatványon szerepel ***n*** prímtényezős felbontásában. Ha ***2*** nem osztja ***n***-et, írjátok ki a ***0***-t. Olvassatok be két számot ***a*** és ***b***, majd írjátok ki az ***a/b*** törtet egyszerűsített (irreducibilis) alakban.

ELEMZÉS

4) Adott két algoritmus a különböző prímosztók számának meghatározására (A: sqrt(n)-ig, B: ***n***-ig). Azonosítsatok egy különbséget és egy hasonlóságot a hatékonyságuk és helyességük szempontjából.

ÉRTÉKELÉS

5) Egy diák azt állítja, hogy a prímtényezős felbontó algoritmus ***n = 1*** esetén is helyesen működik, mivel a while ***d * d <= n*** ciklus egyáltalán nem fut le. Igaz ez a kijelentés? Indokold meg, mit ír ki a program ***n = 1*** esetén, és hogy ez az eredmény megfelel-e az elvártnak.

ALKOTÁS

6) Egy ***n*** természetes számot „***kiegyensúlyozottnak***" nevezünk, ha prímtényezős felbontásában minden kitevő azonos. (*Példa*: $36 = 2^2 \cdot 3^2$ — kiegyensúlyozott; $12 = 2^2 \cdot 3$ — nem kiegyensúlyozott). Írjatok egy programot, amely beolvassa az ***n*** számot, és kiírja azt a legkisebb kiegyensúlyozott ***m*** számot, amelyre igaz, hogy ***m > n***.

Javítókulcs: 2p (hivatalból) + 0.5p (1) + 1p (2) + 1p (3) + 1p (4) + 2p (5) + 2.5p (6)

	A lecke végén tudom...	***Igen***	***Nem***	***Nem biztos***
1.	alkalmazni a prímtényezős felbontó algoritmust			
2.	magyarázni az Euklideszi algoritmus elvét (kivonások → osztások)			
3.	megvalósítani az optimizált ***1…√n*** algoritmust rendezett kimenettel.			
4.	felismerni és indokolni a logikai hibákat az alternatív algoritmusokban			
5.	kiszámolni az LNKO-t Eukleidész-szel és az LKKT-t képlet segítségével.			

6. Nehézségeim voltak a következőkben: …

Ne felejtsétek el értékelni a viselkedéseteket és a tevékenységeteket az óra során!
(a 7. oldalon található viselkedés- és tevékenységfigyelő napló segítségével)

A lecke végén így éreztem magam:

8. Lecke – Számrendszerek

8.1. Helyiértékes számrendszer – alapvető analógia

Bármely szám a ***b*** alapú számrendszerben felírható ***b* hatványainak összegeként**, ahol az egyes hatványokat a számjegyek (***0*** és ***b–1*** közötti értékek) súlyozzák.

10-es számrendszer	**2-es számrendszer**
$347 = 3 \cdot 10^2 + 4 \cdot 10^1 + 7 \cdot 10^0$ számjegyek: 0 – 9	$1011_2 = 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 11$ számjegyek: 0 –1

⚠ **Figyelem!** A b-es számrendszerben a számjegyek mindig 0 és b-1 között vannak:
- 2-es számrendszer ⟶ számjegyek: 0, 1
- 8-as számrendszer ⟶ számjegyek: 0, 1, 2, 3, 4, 5, 6, 7
- 16-os számrendszer → számjegyek: 0, 1, ..., 9, A, B, C, D, E, F

8.2. Egy n szám utolsó számjegyének kinyerése – analógia az 10-es számrendszerrel

10-es számrendszer	**2-es számrendszer**
Utolsó számjegy = n % 10 A szám az utolsó számjegy nélkül = n // 10	Utolsó bit = n % 2 A szám az utolsó bit nélkül = n // 2
`n = 347` `szamjegy = n % 10  # 7` `n = n // 10  # 34`	`n = 11` `bit = n % 2  # 1` `n = n // 2  # 5`

8.3. Számjegyek bejárása jobbról balra – analógia az 10-es számrendszerrel

10-es számrendszer

Lépésenkénti végrehajtás - n = 13

n	n % 10	n // 10
13	3	1
1	1	0
0		

```
n = 13
while n > 0:
    c = n % 10
    print(c)
    n = n // 10

# A tízes számrendszer számjegyei az utolsó számjegytől az első számjegyig lesznek kiírva
```

2-es számrendszer

Lépésenkénti végrehajtás - n = 13 $n_2 = 1101$

n	n % 2	n // 2
13	1	6
6	0	3
3	1	1
1	1	0

```
n = 13
while n > 0:
    c = n % 2
    print(c)
    n = n // 2

# A bináris számjegyek az utolsó bináris számjegytől az elsőig lesznek kiírva
```

8.4. Számjegyek bejárása balról jobbra – Átalakítás 10-es számrendszerből 2-esbe

10-es számrendszer

A számjegyeket az első számjegytől kezdve járjuk be, és a számjegyeket sorrendben, balról jobbra jelenítjük meg.

Lépésenkénti végrehajtás - n = 13

n	p	n // p (c)	n % p
13	10	1	3
3	1	3	0
0	0		

```
n = 13
p = 100000000
while (p > n):
    p = p // 10
while p > 0:
    c = n // p
    print(c)
    n = n % p
    p = p // 10
```

2-es számrendszer

A bináris számjegyeket hasonló módon járjuk be, az első bináris számjegytől indulva, és a számjegyeket sorrendben, balról jobbra jelenítjük meg. A program valójában az n számot **kettes számrendszerben** írja ki, számjegyről számjegyre.

Lépésenkénti végrehajtás - n = 13
Kiíratjuk: $n^2 = 1101$ számjegyről számjegyre

n	p	n // p (c)	n % p
13	8	1	6
6	4	1	3
3	2	0	1
1	1	1	0
0	0		

```
n = 13
p = 2**31  # 2^31
while (p > n):
   p = p // 2
while p > 0:
      c = n // p
      print(c)
      n = n % p
      p = p // 2
```

Megj. Miért indulunk 2^{31}-től?
Szükségünk van egy ***2*** hatványára, amely nagyobb, mint bármely szám, amellyel dolgozni fogunk. $2^{31} = 1.073.741.824$ — ez elegendő a szokásos ***int*** típus minden értékéhez. A while ***p > n*** ciklus addig csökkenti ***p***-t, amíg el nem jutunk a legnagyobb olyan ***2***-hatványhoz, amely nem haladja meg ***n***-et.

Pl. n=13 → indulunk 2^{31}-tól, majd csökkentjük: ..., 16 > 13, 8 ≤ 13 → p=8. Az első kiírt bit: 13//8 = 1

8.5. Átalakítás 10-es számrendszerből k-ba

Az algoritmus megegyezik a ***2***-es számrendszerben használttal, csak a ***2***-est ***k***-ra cseréljük, függetlenül attól, hogy a számjegyeket jobbról balra, vagy balról jobbra járjuk be. Amennyiben ***k > 10,*** a ***0***-tól ***9***-ig terjedő maradékok, számjegyek maradnak, a ***10***-nél nagyobb maradékokat pedig betűkkel helyettesítjük. *Például*: ***k = 16*** esetén a ***10***-től ***15***-ig terjedő maradékok helyett az ***A***-tól ***F***-ig terjedő betűket használjuk (***10→A, 11→B, 12→C, 13→D, 14→E, 15→F***)

```
#Ex. Átalakítás 10-es számrendszerből 16-osba
n = 13
p = 1
while p <= n:
  p = p * 16
p = p // 16
while p > 0:
   c = n // p
   if ( c < 10):
     print(c)
   else:
     print(chr(ord('A') + c - 10))
   n = n % p
   p = p // 16
```

8.6. Átalakítás k alapú számrendszerből 10-esbe

A ***k*** alapú számrendszerből ***10***-es számrendszerbe történő átalakításhoz a számjegyeket balról jobbra járjuk be, és egy új számot állítunk össze, amely kezdetben 0, a következő képlet szerint: ***szam = szam * k + szamjegy***

Lépésenkénti végrehajtás $n_2 = 1101, k = 2$

nr	szamjegy	nr*k	Nr*k + szamjegy
0	1	0	1
1	1	2	3
3	0	6	6
6	1	12	13
13			

```
n = input().upper() # Beolvassuk n-et karakterláncként
# és nagybetűssé alakítjuk
k = int(input()) # Beolvassuk a számrendszer alapját
nr = 0           # Eredmény 10-es számrendszerben
for c in n:      # végigjárunk a szám minden számjegyén
  if ('0' <= c < '9'): # A c karakter számjegy
    szamjegy = int(c) # A karaktert számmá alakítjuk
  else:
    szamjegy = ord(c) - ord('A') + 10 # Átalakítja egy
                      # betűt a numerikus formátumjára
  nr = nr * k + szamjegy
print(nr)        # Kiíratjuk az eredményt
```

8.7. Alkalmazási terület – feladattípusok

A számrendszerek nem pusztán elméleti témát jelentenek –a gyakorlati informatikai rendszerek alapját képezik. Hardver szinten a processzorok kizárólag binárisan működnek, ezért a bitek reprezentációjának megértése elengedhetetlen a regiszterekkel, logikai kapukkal és áramkörökkel való munkához. Programo-

zásban a **bitműveletek** (***bitwise***), amelyeket a kriptográfiában, adatkompresszióban és grafikában használnak, közvetlenül a bináris és hexadecimális reprezentáció manipulációján alapulnak. A 16–os számrendszer szabványos a hibakeresőkben (debugger), hex szerkesztőkben és hálózati protokollokban (MAC-címek, IPv6, RGB színek HTML-ben). A 8-as számrendszer a Unix/Linux rendszerekben jelenik meg a fájlengedélyeknél (chmod 755). Algoritmikus szinten az olyan feladatok, mint „hány darab 1-es bit van n-ben?", „n hatványa-e 2-nek?", „mi a legmagasabb helyiértékű bit?", gyakoriak technikai interjúkban és versenyprogramozásban. A számrendszerek mély megértése lehetővé teszi a hibák felismerését és javítását redundáns kódokkal (Hamming-kód, paritás) – amelyek kritikus alkalmazások a kommunikációban és adattárolásban.

Feladatlap 8

Készítsetek egy *Feladatlap8* mappát. Oldjátok meg a következő feladatokat, és a helyes válaszokat írjátok be egy *valaszok.docx* nevű fájlba.

1. ***Tízes számrendszer→bináris.*** Alakítsátok át a ***173***-at bináris számmá ***2***-vel való sorozatos osztások segítségével. Mutassátok be az összes lépést egy táblázatban. *Példa:* $25_{10} \rightarrow 11001_2$

2. ***Bináris számrendszer→tízes.*** Számítsátok ki a $\mathbf{11010110_2}$ bináris szám tízes számrendszerbeli értékét a helyiértékes képlettel (***2*** hatványainak összege). *Példa*: $1011_2 = 8+0+2+1 = 11$

3. ***Bitek csoportosítása.*** Alakítsátok át a $\mathbf{10110111_2}$ bináris számot oktális és hexadecimális alakba a bitek csoportosításával. (***3***-as, illetve ***4***-es csoportosítások). *Példa:* $110\ 101_2 \rightarrow 65_8$

4. ***Átváltás 10-es számrendszerből 3-as számrendszerbe.*** Olvassunk be egy ***n*** természetes számot. Írassuk ki ennek a számnak a ***3***-as számrendszerbeli alakját. *Példa*: n = 230 esetén az kiírt eredmény 22112_3

```
n = int (input())
p = 3**20
while (p > n):
  p = p // 3
while p > 0:
    c = n // p
    print(c)
    n = n % p
    p = p // 3
```

a. Módosítsuk a "***10-esből 3-as számrendszerbe***" feladatot úgy, hogy az ***n*** számot egy ***k*** alapú számrendszerbe alakítsa át, ahol ***k <10***. Példa: n = 23 esetén a kiírt eredmény 212_3

b. Módosítsuk a "***10-esből 3-as számrendszerbe***" feladatot úgy, hogy az ***n*** számot egy ***16***-os számrendszerbe alakítsa át. *Példa*: n = 200 esetén a kiírt eredmény $C8_{16}$

5. ***Kutatás:*** Készíts kutatást a mesterséges intelligencia (ChatGPT, Claude, Gemini stb.) segítségével a hexadecimális rendszerről és annak szerepéről az informatikában. Azonosíts konkrét példákat a való életből (RGB/HEX színek, memóriacímek, MAC- címek, Unicode kódok stb.), és készíts egy rövid bemutatót arról, hogyan „látja" a számítógép a képeket és a színeket. Bele lehet foglalni egy egyszerű Python alkalmazást is, amely számrendszerek közti átváltást végez, vagy RGB színeket alakít HEX formátumba és vissza.

Javasolt feladatok

1. (**Python kód**) Írj egy Python programot (tömbök és függvények nélkül), amely beolvas egy ***n*** természetes számot, és kiírja a szám bináris alakban írt számjegyeinek összegét (vagyis az ***1***-es bitek számát). *Példa:* n=23 → $23=10111_2$ → bitek osszege= 4

2. (**Python kód**) Írj egy programot, amely beolvas egy természetes számot, és megállapítja, hogy 2-nek hatványa-e (egy szám akkor hatványa ***2***-nek, ha bináris alakjában pontosan egy darab ***1***-es bit van). *Példa:* 32 → IGEN (100000_2), 12 → NEM (1100_2)

3. (**Oktális→bináris→hexa átváltás**) Alakítsd át a $\mathbf{374_8}$ számot először binárissá (minden oktális számjegy → 3 bit), majd hexadecimálissá (4 bites csoportosítás). *Példa*: $52_8 \rightarrow 101\ 010_2 \rightarrow 0010\ 1010_2 \rightarrow 2A_{16}$

4. Számítsd ki két bináris szám bitenkénti összegét (bináris összeadás) csak egyszerű változók és while ciklus segítségével. Teszteld le a $1011_2 + 0110_2$ példára. *Példa*: $10001_2 = 17$

5. ***Mini projekt:*** **Készíts kutatást a szexageszimális (60-as) számrendszerről és annak valós alkalmazásairól.** A projektnek tartalmaznia kell: a rendszer eredete és az azt használó civilizáció történetét, a 60-as alapú számrendszer működésének magyarázatát és valós példákat (időmérés, szögek, földrajzi koordináták, csillagászat stb.). Készíts egy egyszerű Python programot is, amely átalakításokat végez ora/perc/másodperc vagy fok/perc/másodperc tizedes formába, ill. tizedes fokba.

Önértékelés! Mit tanultam?

[ISMERET] [MEGÉRTÉS] [ALKALMAZÁS] [ELEMZÉS] [ÉRTÉKELÉS] [ALKOTÁS]

ISMERET	1) Mi a képlete annak, hogy az ***N*** szám utolsó számjegyét meghatározzuk ***10***-es számrendszerben? És ***2***-es számrendszerben?
MEGÉRTÉS	2) Mennyi a ***25*** -nek ***2***-el való osztási maradéka, és mit jelent ez a bináris átalakítás szempontjából?
ALKALMAZÁS	3) Adott egy ***n*** természetes szám. Írd ki a szám bináris alakjában szereplő ***1***-es bitek számát. *Példa*: n = 13 esetén (= 1101_2), a kiírt eredmény 3 (a szám 3 darab 1-es bitet tartalmaz).
ELEMZÉS	4) A mellékelt programot teszteljük le ***n = 100***-ra. Kövessétek nyomon a változók, ***n*** és ***p*** alakulását minden lépésben, és töltsétek ki a táblázatot. Milyen matematikai tulajdonságot számol ki a program ***100***-ra? Ugyanez a program ***n = 36*** esetén is ugyanazt az eredményt adja? És ***n = 7*** esetén? Indokoljátok meg.

```
n = 100
p = 1
while n % 2 == 0:
    n = n // 2
    p = p * 2
print(p)
```

Lépés	n	p
1	100	1

ÉRTÉKELÉS	5) Egy tanuló azt állítja, hogy egy szám ***8***-as számrendszerből ***16***-osba való átalakításánál hatékonyabb először ***10***-es számrendszerbe konvertálni, majd onnan ***16***-osba. Egyetértesz? Indokold egy konkrét példával.
ALKOTÁS	6) Definiáljuk egy ***n*** szám komplementerét: ***n***-et átalakítjuk ***2***-es számrendszerbe, majd a felírásban minden bináris bitet tagadunk: ha ***1*** volt, akkor ***0*** lesz belőle, és fordítva. Majd újra visszaalakítjuk ***10***-esbe, ez az eredmény ***n*** komplementere lesz. Adott egy ***n*** természetes szám, írassuk a komplementerét. *Példa*: $45_{(10)}$ kettes számrendszerbeli alakja $101101_{(2)}$, tagadás után $010010_{(2)}$, újra 10-esbe alakítva $18_{(10)}$, tehát az eredmény 18.

Javítókulcs: 2p (hivatalból) + 0.5p (1) + 1p (2) + 1p (3) + 1p (4) + 2p (5) + 2.5p (6)

	A lecke végén tudom/tudok...	***Igen***	***Nem***	***Nem biztos***
1.	miért írható fel bármely szám az alap hatványainak összegeként.			
2.	10-es számrendszerből k alapú számrendszerbe alakítani.			
3.	átalakítani egy számot k alapú számrendszerből 10-esbe.			
4.	bejárni egy szám számjegyeit bármely számrendszerben.			
5.	olyan programot írni, amely tetszőleges k alapú számrendszerbe alakít.			

6. Nehézségeim voltak a következőkben: …

Ne felejtsétek el értékelni a viselkedéseteket és a tevékenységeteket az óra során!
(a 7. oldalon található viselkedés- és tevékenységfigyelő napló segítségével)

A lecke végén így éreztem magam:

2. Fejezetvégi felmérés

Kutass!

(Maximális pontszám: 90 pont + 10 pont hivatalból)

Számadatok feldolgozása

A vizsgálandó program:

```
n = int(input("Írjon be egy számot: "))
t=n; s=0; cmax=0; inv=0; nrszj=0
while t > 0:
    c = t % 10; s = s + c
    if c > cmax:
        cmax = c
    inv = inv * 10 + c
    nrszj = nrszj + 1
    t = t // 10
print("Osszeg:",s," Max:",cmax)
print("Forditott:",inv," NrSzamjegy:",nrszj)
if inv==n:
    print(n,"palindrom")
else:
    print(n,"nem palindrom")
p=2**31; t=n
print("2-es számrendszerben:", end=" ")
if (n == 0):
    print(0)
while p > n:
    p //= 2
while p > 0:
    print(t//p, end =""); t=t%p; p=p//2
# print('\n')
d=2; prim=1
if n < 2:
    prim=0
while d<n and prim==1:
    if n%d==0:
        prim=0
    d=d+1
if prim==1:
    print("\n",n,"prím")
else:
    print("\n",n,"nem prím")
```

1. Részletek azonosítása **(15p)**

Töltsétek ki a táblázatot:

Kategória	Változó	Szerepe
Bemeneti adatok		
Kimeneti adatok		
Közbenső adatok		

2. A futtatás követése **(20p)**

Kövessétek nyomon n = 37 esetén:

t	c=t%10	s	cmax	inv	nrszj	t//10
37						

Kimenetek n = 37 esetén: ...

3. Hibaelemzés és tesztelés **(20p)**

a) Elemezzétek a program viselkedését n = 0 esetén. Minden eredmény helyes? Indokoljátok.

b) Javasoljatok 3 tesztesetet:

Teszteset	n értéke	Várt kimenetek	Mit ellenőriz

c) Találjatok meg egy váratlan viselkedést (hibát), és javasoljatok megoldást.

4. A program bővítése **(20p)**

a) Módosítsátok a programot úgy, hogy a maximális számjegy helyett a minimálisat találja meg. Jelöljétek meg a módosított sorokat és a beillesztés helyét.

b) Adjátok hozzá az lnko(n, inv) számítását Euklidesz algoritmusa alapján.

5. Elemzés és értékelés **(15p)**

A program melyik részét lehetne módosítani annak érdekében, hogy az algoritmus hatékonyabbá váljon?

Magyarázzátok el.

Pontozás: 1 → 15p; 2 → 20p; 3 → 20p; 4 → 20p; 5 → 15p = 90p

Mentsd el a válaszokat az *kutass2.docx* fájlba, a módosított kódot pedig *vizsgalt.py* fájlként.

Ellenőrizd a tudásodat!

(Pontszám: 90 pont + 10 pont hivatalból)

Követelmény: Oldjátok meg az alábbi feladatokat a ***tudas_ellenorzes2.docx*** fájlban. Az összes létrehozott fájlt mentsétek a ***Fejezet2*** mappába.

ISMERET	1) Egy n természetes szám akkor palindrom, ha megegyezik a visszafelé olvasott változatával. Melyik lehetőség ellenőrzi helyesen ezt a tulajdonságot? A) `n % 10 == n // 10` B) felépítjük `inv` -et így: `inv*10 + n%10` majd ellenőrizzük `inv == n_eredeti` C) `n // 10 == 0` D) `n % 11 == 0`	(6p)
	2) Határozzátok meg a következő fogalmakat: prímszám, osztó, lnko és lkkt. Adjatok mindegyikre egy-egy számpéldát. *Példa:* 7 prím; div(12)={1,2,3,4,6,12}; lnko(12,18)=6; lkkt(4,6)=12.	(8p)
MEGÉRTÉS	3) Töltsétek ki az üres helyeket, hogy helyesen leírjuk a 10-es alapból 2-es alapba történő átalakítás algoritmusát: a) Minden lépésben kiszámítjuk az n %-os maradékot; ez a ..... -t jelenti. b) A számot felére csökkentjük az n = ... -val. c) A maradékokat a következő sorrendben olvassuk le	(10p)
	4) Magyarázzátok el, miért elegendő ***n*** prímjellegének ellenőrzéséhez a $\sqrt{n}$ -ig (nem pedig ***n-1***-ig) tesztelni az osztókat. Illusztráljátok a magyarázatot ***n=36*** és ***n=37*** esetén.	(12p)
ALKALMAZÁS	5) Írjatok Python kódot, amely beolvassa ***n***-t és ***k***-t, és kijelzi ***n*** számjegyösszegét ***k*** alapon, ahol k < 10. Példa: n=45 és k=5 esetén a kijelzett összeg=5 ($45_{10} \rightarrow 140_5$, összeg = 1 + 4 = 5).	(16p)
ÉRTÉKELÉS	6) Az egyik diák azt javasolja, hogy ***n*** prímjellegét úgy ellenőrizzük, hogy ***n-1***-ig minden osztójával elosztjuk. Egy másik diák azt javasolja, hogy csak a $\sqrt{n}$ -ig terjedő prímszámokkal osszuk el. Hasonlítsd össze a két módszert a lépések száma szempontjából ***n=97*** esetén. Hasonlítsd össze a két módszert a műveletek száma szempontjából.	(18p)
ALKOTÁS	7) Írjatok Python kódot, amely beolvassa az ***n***-t, megjeleníti annak prímtényezőkre való bontását, majd ellenőrzi, hogy a különböző prímtényezők száma páros vagy páratlan-e. Teszteljétek ***n = 360*** és ***n = 12*** esetén. *Példa*: $360 = 2^3 \cdot 3^2 \cdot 5^1 \rightarrow$ 3 különböző prímosztó → páratlan. $12 = 2^2 \cdot 3^1 \rightarrow$ 2 osztó → páros.	(20p)

Javítókulcs: 10p (hivatalból) + 6p (1) + 8p (2) + 10p (3) + 12p (4) + 16p (5) + 18p (6) + 20p (7)

Portfólió

(Pontszám: 100 pont)

E fejezet végén a tanuló személyes portfóliójának a ***Fejezet2*** mappában a következő anyagokat kell tartalmaznia:

- ***Feladatlap5*** mappa, a következő fájlokkal: ***valaszok.docx***, ***Onertekeles5.docx***
- ***Feladatlap6*** mappa, a következő fájlokkal: ***valaszok.docx***, ***Onertekeles6.docx***
- ***Feladatlap7*** mappa, a következő fájlokkal: ***valaszok.docx***, ***Onertekeles7.docx***
- ***Feladatlap8*** mappa, a következő fájlokkal: ***valaszok.docx***, ***Onertekeles8.docx***
- ***Projekt2*** mappa, a következő fájlokkal: ***kutass2.docx***, ***vizsgalt.py***
- A ***tudas_ellenorzes2.docx*** fájl.

Értékelés módja:

- Megvan a ***Fejezet2*** mappa a megadott könyvtárszerkezettel – **20p**
- A mappákban megvan:
  - A fájlok több mint 95%-a – **80p**
  - A fájlok 75–94%-a – **60p**
  - A fájlok 50-74%-a – **40p**
  - A fájlok 25-49%-a – **20p**
  - 25% alatt – **0p**

3. Fejezet – Moduláris felépítés és objektumorientált programozás

9. Lecke – Alprogramok: fejléc, törzs, paraméterek

9.1. Mi az alprogram? Jellemzői és szerepe



Egy **alprogram** (Pythonban **függvény**) egy ***név***vel azonosított utasássorozat, amely egy adott részfeladatot old meg, és a program több pontján is újra felhasználható. Az alprogram szükség szerint bármikor, a program bármely pontjáról meghívható.

Az **alprogramok szerepe** a kód **moduláris szervezésének** lehetővé tétele: egy összetett program kisebb alprogramokra bontható, amelyek mindegyike a feladat egy-egy részét oldja meg.

Az alprogramok használatának előnyei:

- ***A kód újrafelhasználása*** – egy alprogramot egyszer megírva tetszőleges számú alkalommal meghívható;
- ***Olvashatóság (átláthatóság)*** – a program könnyebben érthetővé és áttekinthetővé válik;
- ***Könnyebb tesztelés és hibakeresés*** – minden alprogram külön is tesztelhető;
- ***Együttműködés*** – a csapat tagjai egymástól függetlenül dolgozhatnak különböző alprogramokon.

9.2. Egy Python függvény felépítése

Egy Python függvénynek két fő összetevője van:

Összetevő	Leírás
Fejléc (header)	Az első sor, amely tartalmazza a def kulcsszót, a függvény nevét és a paramétereket. **Példa:** def min(a, b):
Törzs (body)	A fejléc alatt behúzással írt sorok; tartalmazza a függvény hívásakor végrehajtandó utasításokat.
return	Opcionális; visszaküldi az eredményt, *return* nélkül a függvény *None* értéket térít vissza.

A **függvényfej** (vagy header) az alprogram első sora. Ez a következőket tartalmazza:

- a ***def*** kulcsszót– ez jelzi az alprogram definíciójának kezdetét;
- az alprogram **nevét** – a programozó által választott egyedi azonosító;
- a **paraméterek listáját** – zárójelek között, vesszővel elválasztva, ez a lista üres is lehet;
- a **:** (kettőspont) karaktert – amely a függvényfej végét jelzi.

Az alprogram-definíció általános szintaxisa:

Egy alprogramot a ***def*** kulcsszóval definiálunk:

```
def fuggveny_neve(paraméter1, paraméter2, ...):
    utasítás1
    utasítás2
    return eredmény  # opcionális
```

Egy alprogram ***fejlécét*** a függvény első sora képezi, amelyben a függvény neve és paraméterei szerepelnek. Ez megadja a függvény nevét és azt, hogy milyen adatokat fogad. Minden, ami ez alatt behúzással szerepel, a függvény törzséhez tartozik.

9.3. Példa: két szám összege

```
def osszeg(a, b):
    ered = a + b
    return ered
```

Megj. Az alprogramot a nevével és a paraméterek listájával hívjuk meg. A fenti példára a függvény meghívása a következő lehet:

```
# Függvény meghívása:
s = osszeg(3, 5)        # meghívás
print(s)                # → 8
```

9.4. Adatátvitel a paraméterek segítségével

<def/> A ***paraméterek*** olyan változók, amelyeken keresztül a függvény külső adatokat fogad. A függvényhíváskor az átadott értékeket ***argumentumoknak*** nevezzük. Ezek lehetővé teszik ugyanazon függvény különböző értékekkel való használatát.

Vannak helyzetek, amikor elvárjuk, hogy egy paraméternek alapértelmezett értéke legyen, de különleges esetekben módosítani szeretnénk azt. Például egy teszt pontszáma normál esetben egy pontot érhet hivatalból, és maximum 10 lehet, de előfordulhat olyan helyzet, amikor a teszt pontszámát például egy projekthez szeretnénk hozzáadni, amely esetben maximum 5 pontot érhet, míg ekkor 0,5 pont járna hivatalból:

```
def jegy_szamitas(jegy, hivatalbol = 1, maximum = 10):
    return min(jegy + hivatalbol, maximum)
```

Az alapértelmezett értékeket (***hivatalbol = 1*** vagy ***maximum = 10***) közvetlenül a függvény fejlécében határozzuk meg. Ha az alprogramot az alapértelmezett értékekkel szeretnénk meghívni, akkor a híváskor figyelmen kívül hagyhatjuk ezeket a paramétereket; ellenkező esetben a szokásos módon megadhatjuk az értékeket:

```
jegy_szamitas(3) → min(3 + 1, 10) = min(4, 10) = 4 értéket térít vissza
jegy_szamitas(10) → min(10 + 1, 10) = min(11, 10) = 10 értéket térít vissza
jegy_szamitas(3, 5) → min(3 + 5, 10) = min(8, 10) = 8 értéket térít vissza
jegy_szamitas(3, 2, 5) → min(3 + 2, 5) = min(5, 5) = 5 értéket térít vissza
```

A ***jegy_szamitas*** alprogram egy, két vagy három paramétert fogadhat. Azok a paraméterek, amelyeket a híváskor nem adnak meg, automatikusan a függvény fejlécében meghatározott alapértelmezett értékeket kapják, míg a többi paraméter a megadott értéket kapja, a megadásuk sorrendjében, balról jobbra.

```
jegy_szamitas(jegy=3, hivatalbol=5) → min(3 + 5, 10) = min(8, 10) = 8
jegy_szamitas(3, maximum=5) → min(3 + 1, 5) = min(4, 5) = 4
```

Ha csak bizonyos értékeket szeretnénk módosítani anélkül, hogy mindet megadnánk, használhatjuk a név szerinti hívást, kifejezetten megjelölve azt a paramétert, amelynek értéket szeretnénk adni.

9.5. Az eredmények visszatérítése

A ***return*** utasítás a kiszámított eredményt visszaküldi a programnak arra a helyre, ahonnan az alprogramot meghívták. A return utasítás végrehajtása után az alprogram azonnal befejeződik.

```
import math
def tavolsag(x1, y1, x2, y2):
    d = math.sqrt((x2-x1)**2 + (y2-y1)**2)
    return d # ← eredmény visszatérítése
r = tavolsag(0, 0, 3, 4)    # r -be visszatérő érték: 5.0
print(r)                    # → 5.0
```

A paraméterek opcionálisak, lehetnek paraméter vagy visszatérített érték (*return)* nélküli függvényeink:

```
def koszon():
    return "Szia"
```

Egy függvény eredménye az az érték, amelyet a függvény létrehoz, és visszaküld a programnak arra a helyre, ahonnan meghívták. Ezt az értéket a *return* utasítással küldjük vissza. Az eredmény bármi lehet: szám, szöveg, tuple, egy másik függvény stb. Ha nem adunk vissza semmit, a Python automatikusan azt feltételezi, hogy *None*-t térítettünk vissza.

```
def koszon():
    print("Szia")
```

A fenti függvényt egyszerűen a *koszon()* utasítással lehet meghívni. Ha a függvény eredményét egy x = *koszon()* változóba menjük, a kód a „Szia” szöveget jeleníti meg a képernyőn, az x változó értéke pedig *None* lesz.

9.6. Egy alprogram meghívása

A **meghívás** az a művelet, amellyel egy alprogramot a főprogramból vagy egy másik alprogramból indítanak el. A meghívás szintaxisa a következő:

```
# Egyszerű meghívás (az eredmény tárolása nélkül):
koszon()
# Függvénymeghívás az eredmény változóba történő eltárolásával:
terulet = terulet_szamitas(5, 3)
# Közvetlen hívás egy kifejezésben:
print (terulet_szamitas(5, 3))
# Meghívás név szerinti argumentumokkal:
jegy_szamitas(jegy=8, hivatalbol=2)
```

9.7. Python alprogramok leírásának és meghívásának szintaxisa

Teljes szintaxis – definíció:

```
def alprogram_neve(param1, param2, param3=alapértelmezett_ertek):
    # alprogram törzse
    utasítás1
    utasítás2
    return eredmény   # nem kötelező
```

Teljes szintaxis – meghívás:

```
# 1. Variáns – argumentumok helyének megadásával
eredmeny = alprogram_neve(arg1, arg2)
# 2. Variáns – argumentumok nevének megadásával
eredmeny = alprogram_neve(param1=arg1, param2=arg2)
# 3. Variáns – meghívás eredmény eltárolása nélkül (nincs hasznos return)
alprogram_neve(arg1, arg2)
```

9.8. Gyakorlati tevékenység

Oldd meg az alábbi feladatokat:

1. Írd meg a ***terulet_szamitas(hosszusag, szelesseg)*** függvényt, ami visszatéríti egy téglalap területét.
2. Írj egy ***legnagyobb***(a, b, c) függvényt, ami visszatéríti három szám közül a legnagyobbat.
3. Alakítsd át a ***jegy_szamitas*** függvényt úgy, hogy egy bónuszpont (alapértelmezetten 0) opcionális paramétert is fogadjon ami a jegy értékét növeli.



Jegyezd meg!

- ✔ ***def*** → egy alprogram definiálásához szükséges kulcsszó
- ✔ A ***fejléc*** tartalmazza a nevet és a paramétereket; a ***törzs*** tartalmazza az utasításokat
- ✔ A ***return*** visszaküldi az eredményt a meghívás helyére
- ✔ Az alapértelmezett értékű ***paraméterek*** lehetővé teszik az opcionális argumentumok használatát
- ✔ A név szerinti ***hívás*** (paraméter = érték) lehetővé teszi az argumentumok szelektív megadását



Feladatlap 9

Hozzatok létre egy *Feladatlap9* nevű mappát. Oldjátok meg az alábbi feladatokat, és írjátok be a helyes válaszokat egy *valaszok.docx* nevű állományba.

1. Írj egy függvényt, amely 4 paramétert fogad (a, b, c és d), és visszatéríti közülük a legnagyobb értéket.
2. Írj egy függvényt, amely kiírja az „Üdvözlünk, ” üzenetet, majd a paraméterként megadott nevet.
3. Írj egy függvényt, amely két értéket kap és visszatéríti a különbségük abszolút értékét.
4. 4. Adott az alábbi alprogram. Mit ad vissza az ***f(5)*** meghívás?

```
def f(a):
    return a ** 2, a - # Egy függvény akár két értéket is visszaadhat – a Python
    ezeket automatikusan tupleként csomagolja.
```

5. ***Mini-projekt:*** Írj egy alkalmazást, amely kiszámítja a vásárlás teljes költségét (ár × mennyiség) egy, az ÁFÁ-ra vonatkozó alapértelmezett értékkel rendelkező függvény segítségével.



Önértékelés!

Az *onertekeles9.docx* állományba írjátok be a megfelelő válaszokat, a következő jelöléseket használva: *1 – igen, 2 – nem, 3 – nem vagyok biztos benne*, majd mentsétek el a *Feladatlap9* mappába.

	A lecke végén tudom...	***Igen***	***Nem***	***Nem biztos***
1.	mi az alprogram (függvény), és mi a szerepe egy program felépítésében.			
2.	melyek egy alprogram összetevői: a fejléc (def, név, paraméterek) és a törzs (behúzott utasítások).			
3.	hogyan definiálunk egy alprogramot Pythonban.			
4.	mik a paraméterek.			
5.	hogyan hívható meg egy alprogram.			
6.	milyen előnyei vannak az alprogramok használatának.			

7. Nehézségeim voltak a következőkben: …

Ne felejtsétek el értékelni a viselkedéseteket és a tevékenységeteket az óra során!
(a 7. oldalon található viselkedés- és tevékenységfigyelő napló segítségével)

A lecke végén így éreztem magam:

10. Lecke – A végrehajtási mechanizmus: lokális és globális változók

10.1. A végrehajtási mechanizmus

Egy függvény meghívásakor a program végrehajtása megszakad, az értékek átadódnak a paramétereknek, a függvény törzse végrehajtódik és visszatéríti az eredményt, majd a program onnan folytatódik, ahol abbahagyta. Ez a mechanizmus biztosítja a függvények egymás közötti és a főprogramtól való elszigeteltségét, lehetővé téve a kód hatékony újrafelhasználását.

Egy függvény ***végrehajtási mechanizmusa*** a következő lépéseket követi sorrendben:

1. A függvény meghívásakor a főprogram végrehajtása felfüggesztődik.
2. Egy új memóriaterület (végrehajtási kontextus) jön létre az alprogram számára.
3. Az argumentumok értékei a paraméterekhez rendelődnek.
4. A függvény törzse végrehajtásra kerül.
5. A visszatérési érték (return) visszakerül a hívás helyére.
6. Az alprogram végrehajtási kontextusa felszabadul a memóriából.
7. A főprogram végrehajtása onnan folytatódik, ahol abbamaradt.



Emlékezzünk vissza az előző leckében definiált jegyszámító függvényre:

```
def jegy_szamitas(jegy, hivatalbol = 1, maximum = 10):
        return min(jegy + hivatalbol, maximum)

print('Meghivas elott')      # → kiírja Meghivas elott
x = jegy_szamitas(8)         # → a függvényben: jegy=8,hivatalbol=1,maximum=10
                             #   return min(8+1, 10) = 9
print('x =', x)              # → kiírja 'x = 9'
```

A programok működésének megértéséhez alapvető fontosságú annak kezelése, hogy a változók hol jönnek létre, és mennyi az élettartamuk. Ezt a fogalmat a változók láthatóságának (hatókörének) nevezzük. Ennek megfelelően a változókat két fő kategóriába sorolhatjuk: lokális és globális változók.

10.2. Lokális változók



A **lokális** változók egy függvény belsejében vannak definiálva. ***Kizárólag a függvény végrehajtásának ideje alatt léteznek***, és kívülről nem érhetők el.

```
def teszt_lokalis():
    uzenet = 'Szia'          # lokális változó
    print(uzenet)            # OK - a függvény belsejében

teszt_lokalis()              # kiírja 'Szia'
print(uzenet)                # HIBA! 'uzenet' nem létezik a függvényen  kívül
```



A fenti példában a ***teszt_lokalis()*** függvényhívás a „Szia” szöveget jeleníti meg, de a hívást követő ***print(uzenet)*** utasítás hibát eredményez, mivel a uzenet változó a függvényen kívül nem ismert. A függvény paraméterei szintén helyi változók.

A lokális(helyi) változók jellemzői:

- Csak azon alprogram belsejében láthatók és elérhetők, amelyben definiálták őket
- Nem érhetők el a főprogramból vagy más alprogramokból
- Egy alprogram paraméterei szintén lokális változók
- Két különböző alprogramnak lehetnek azonos nevű lokális változói, anélkül, hogy ütközés lépne fel

⚠ **Figyelem!** A függvény paraméterei szintén lokális változók. A függvény végrehajtásának befejeztével eltűnnek a memóriából.

10.3. Globális változók

<def/> A **globális** változók bármely függvényen kívül vannak definiálva. A függvényekből kiolvashatók, de a **global** kulcsszó nélkül nem **módosíthatók**.

```
tva = 19          # globális változó
def ar_szamol(ar):
    return ar + (ar * tva / 100)  # globális kiolvasás – OK
pont = 0
def pont_hozzaad():
    global pont       # deklaráljuk, mert MÓDOSÍTANI szeretnénk a globális
    pont = pont + 1
pont_hozzaad()
print(pont)      # → 1
```

Megj. Ebben a példában az ***ar_szamol*** függvény a ***tva*** globális változót használja a számítás elvégzéséhez, még akkor is, ha azt nem kapta meg paraméterként. Ha egy függvényen belül meg akarjuk változtatni egy globális változó értékét, akkor a ***global*** kulcsszót kell használnunk. Ha nem használnánk a global ***pont*** utasítást, a Python megpróbálna egy új, azonos nevű lokális változót létrehozni, míg a globális változó értéke nem változna meg.

A globális változók jellemzői:

- A program bármely pontjáról láthatók és elérhetők, az alprogramokból is;
- Külön deklaráció nélkül közvetlenül olvashatók az alprogramokból;
- Ahhoz, hogy egy alprogramból módosítani lehessen őket, deklarálni kell ezeket a global kulcsszóval;
- A globális változók túlzott használata megnehezítheti a hibakeresést.

10.4. Összehasonlító táblázat: lokális és globális

Jellemző	Lokális és Globális
Hol van definiálva	**Lokális**: függvény belsejében | **Globális**: függvényeken kívül
Élettartam	**Lokális**: függvény után eltűnik | **Globális**: a teljes futás során fennmarad
Elérhetőség	**Lokális**: csak a függvényben | **Globális**: a kód bármely pontján
Módosíthatóság függvényen belül	**Lokális**: direkt | **Globális**: szükséges a 'global' változóra
Kockázat	**Lokális**: mellékhatások nélkül | **Globális**: véletlenül módosítható

A végrehajtás szigorú logikát követ: amikor meghívunk egy függvényt, egy új végrehajtási kontextus (egy memóriaterület) jön létre. Ide kerülnek be a paraméterek és a lokális változók. Amikor a program eléri a ***return*** (visszatérési) utasítást, végrehajtási kontextus felszabadul, a vezérlés pedig visszatér a hívási helyre. A globális változók a program futásának teljes időtartama alatt a memóriában maradnak.

10.5. Változók elfedése (árnyékolása)

<def/> Amikor egy lokális változó neve megegyezik egy globális változó nevével, az alprogram a lokális változót fogja használni. Ezt nevezzük ***elfedésnek/árnyékolásnak***:

```
x = 100          # globális változó
def teszt():
    x = 5        # lokális változó – nem befolyásolja a globális változót!
    print(x)     # kiír: 5
teszt()          # kiír: 5
print(x)         # kiír: 100 (globális nem változott)
```

10.6. A nonlocal kulcsszó (beágyazott függvények)

Vannak azonban olyan helyzetek, különösen összetettebb struktúrákban, amikor egy függvényt egy másik függvény belsejében definiálunk. Ebben az esetben szükségessé válik egy köztes változótípus használata, amely sem globális, sem az aktuális függvényhez tartozó helyi változó.

A ***nonlocal*** kulcsszó a beágyazott függvényekben (függvény a függvényben) használatos annak jelzésére, hogy egy változó a külső függvényhez tartozik, de nem globális változó.

A ***nonlocal*** lehetővé teszi, hogy a belső függvény hozzáférjen és módosítsa a külső függvény változóit:

```
def szamlalo():
    n = 0
    def novel():
        nonlocal n    # eléri a külső függvény 'n' változóját
        n = n + 1
    novel()
    novel()
    print(n)          # → 2
szamlalo()
```

```
def kulso_fuggveny():
    x = 10
    def belso_fuggveny():
        nonlocal x
        x = 20
        print("Belso x =", x)
    belso_fuggveny()
    print("Kulso x =", x)
kulso_fuggveny()
```

A mellékelt példában a nonlocal x utasítás nélkül a belső függvény egy új, 20-as értékű lokális ***x*** változót hozott volna létre, a külső x változót pedig változatlanul hagyta volna (az maradt volna 10). A ***nonlocal*** használatával arra utasítjuk a programot, hogy a közvetlenül felette lévő végrehajtási kontextusban keresse az x változót, és közvetlenül azt módosítsa.

Ez a mechanizmus nagyon hasznos, amikor egy függvény belső állapotát (például egy számlálót vagy egy beállítást) szeretnénk fenntartani anélkül, hogy globális változókat használnánk, így az adatokat védve tartjuk a főfüggvényen belül.

10.7. Gyakorlati tevékenység

1. Írj egy függvényt, amely kiszámítja egy nap legmagasabb hőmérsékletét. Határozd meg, mely változók legyenek lokálisak, és melyek lehetnek globálisak (pl. a riasztási küszöbérték).

2. Magyarázd el, miért okoz hibát a mellékelt kód, és hogyan javítanád ki:

```
veg = 0
def hozzaad(ert):
    veg = veg + ert
hozzaad(5)
```



Jegyezd meg!

- ✔ A függvényhíváskor új végrehajtási kontextus jön létre a memóriában
- ✔ A **lokális** változók csak a függvény végrehajtásának ideje alatt léteznek
- ✔ A **globális** változók a program teljes futási ideje alatt megmaradnak, és a függvényekből is elérhetők
  - A **global** kulcsszó az egész program általános kontextusához köti a változót
  - A **nonlocal** kulcsszó a szülőfüggvény kontextusához köti a változót, lehetővé téve annak módosítását
- ✔ **Árnyékolás** akkor lép fel, amikor egy lokális változó neve megegyezik egy globális változó nevével

Feladatlap 10

Hozzatok létre egy *Feladatlap10* nevű mappát. Oldjátok meg az alábbi feladatokat, és írjátok be a helyes válaszokat egy *valaszok.docx* nevű állományba.

1. Elemezd az alábbi kódot. Mit ír ki a végén?

```
x = 10
def modosit():
    x = 5
modosit()
print (x)
```

2. Írj egy függvényt, amely módosítja az „energia” nevű globális változót úgy, hogy levonja belőle a híváskor átadott „fogyasztás” paraméter értékét.

3. Mi történik, ha egy függvényből megpróbálsz hozzáférni egy másik függvény helyi változójához? Magyarázd el a végrehajtási mechanizmus alapján.

4. Vegyük az alábbi alprogramot. Mit fog kiíratni, ha egymás után kétszer hívják meg?

```
ossz = 0
def hozzaad(n):
    global ossz
    ossz = ossz + n
    return ossz
print (hozzaad(5))
print (hozzaad(3))
```

5. Mi a fő különbség egy globális változó és egy nonlocal-ként deklarált változó között?

6. Írj egy ***iskola()*** nevű alprogramot, amelyben szerepel egy diak = 20 változó, valamint egy ***beir()*** nevű függvény, amely a tanult kulcsszó használatával 1-gyel növeli a diákok számát.

7. Mit ír ki az alábbi kód?

```
def anya():
    nev = "Elena"
    def gyerek():
        nonlocal nev
        nev = "Andras"
        print("Belso nev =", nev)
    gyerek()
    print(nev)
anya()
```

8. ***Mini-projekt:*** Iskolai osztálynapló-alkalmazás globális változókkal (a diákok listája) és helyi függvényekkel az egyes diákok átlagának kiszámításához.

Önértékelés

Az *onertekeles10.docx* állományba írjátok be a megfelelő válaszokat, a következő jelöléseket használva: *1 – igen, 2 – nem, 3 – nem vagyok biztos benne*, majd mentsétek el a *Feladatlap10* mappába.

	A lecke végén tudom...	***Igen***	***Nem***	***Nem biztos***
1.	melyek egy alprogram végrehajtási mechanizmusának lépései.			
2.	mik a helyi változók.			
3.	mik a globális változók.			
4.	miért van szükség a global kulcsszóra ahhoz, hogy egy függvényen belül módosítsunk egy globális változót.			
5.	mit jelent a változók elrejtése/árnyékolása.			
6.	mi a nonlocal kulcsszó, és milyen helyzetekben használják.			
7.	melyek a legfontosabb különbségek a lokális és a globális változók között (élettartam, elérhetőség, kockázat)?			

8. Nehézségeim voltak a következőkben: …

Ne felejtsétek el értékelni a viselkedéseteket és a tevékenységeteket az óra során!
(a 7. oldalon található viselkedés- és tevékenységfigyelő napló segítségével)

A lecke végén így éreztem magam:

11. Lecke – Beépített függvények kollekciókhoz (Python)

A Python számos olyan beépített függvénnyel rendelkezik, amelyek közvetlenül meghívhatók gyors matematikai számítások elvégzésére, vagy az adattípusok egymás közötti átalakítására.

A Python beépített függvényeinek jellemzői és használati útmutatója a szokásos matematikai műveletekhez

11.1. Abszolút érték



Egy szám **abszolút értéke** a számtengelyen a nullától mért távolságát jelenti, előjel nélkül. Az ezt kiszámító beépített függvény az **abs(x)**, amely Pythonban importálás nélkül közvetlenül elérhető.

Jellemzők és használat: ***int*** vagy ***float*** típusú argumentumokat fogad el; mindig ***pozitív*** értéket vagy ***nullát*** ad vissza; Gyakori felhasználás: távolság számítása, hibák vizsgálata, szimmetrikus összehasonlítások.

```
abs(-7)          #  7
abs(3.5)         # 3.5
abs(0)           # 0
# Példa: tengely két pontja közötti távolság
def tavolsag_1d(a, b):
    return abs(a - b)
tavolsag_1d(3, 8)     #  5
tavolsag_1d(8, 3)     #  5  (ugyanaz az eredmény, függetlenül a sorrendtől)
```

11.2. Egészrész meghatározása

Egy valós szám **egészrésze** a tizedesjegyek nélküli egész számrészét jelenti. A Pythonban több előre definiált függvény is rendelkezésre áll ennek kiszámítására, amelyek eltérő módon viselkednek.



A bonyolultabb matematikai műveletekhez a Python a ***math*** modult használja. Ezt a program elején be kell importálni.

Függvény	Mit csinál?	Példa
int(x)	Levágja a tizedesjegyeket	`int(3.9)` → **3** | `int(-3.9)` → **-3**
math.floor(x)	Lefelé kerekítés (a -∞ irányába)	`floor(3.9)` → **3** | `floor(-3.1)` → **-4**
math.trunc(x)	Levágja a tizedesjegyeket a nulla irányába (megegyezik az int() függvénnyel)	`trunc(3.9)` → **3** | `trunc(-3.9)` → **-3**

⚠ **Figyelem a negatív számoknál**

int(–3.9) → –3 (Levágja a tizedesjegyeket a nulla irányába, az eredmény NAGYOBB)

math.floor(–3.9) → –4 (Lefelé kerekítés, az eredmény KISEBB)

A helyes választás a feladat kontextusától függ.

```
import math
# Iskolai osztályzat kiszámítása (tizedesjegyű osztályzat nem adható)
jegy  = 7.8
print(int(jegy ))                       #  7  (levágott tizedesek)
print(math.floor(jegy ))                #  7  (ugyanaz pozitív szám esetében)
# Hány teljes doboz fér el egy 47,3 m²-es raktárban?
terulet = 47.3
doboz_terulet = 5
print(int(terulet / doboz_terulet))     #  9
```

11.3. Kerekítés

A **kerekítés** egy valós számot adott számú tizedesjegyre alakít. A Python több beépített függvényt is kínál, amelyek eltérő módon működnek:

Függvény	Mit csinál?	Példa
round(x, n)	Visszaadja ***x*** értékét ***n*** tizedesjegyre kerekítve. Ha ***n*** nincs megadva, az eredmény egy egész szám lesz.	`round(3.567, 2)` → `3.57`
math.floor(x)	Mindig lefelé kerekítés	`floor(3.9)` → `3` | `floor(-3.1)` → `-4`
math.ceil(x)	Mindig felfelé kerekítés	`ceil(3.1)` → `4` | `ceil(-3.9)` → `-3`

Bankárkerekítés a Python 3-ban

```
A round() a bankárkerekítést használja (a legközelebbi páros számra kerekít):
  round(2.5) → 2   (nem 3!)
  round(3.5) → 4
  round(4.5) → 4   (nem 5!)
```

Ez a módszer megakadályozza a kerekítési hibák felhalmozódását a pénzügyi számításokban.

```
import math
szamla = 45.678
print(round(szamla, 2))                 #45.68   (standard)
print(math.floor(szamla*100)/100)       #45.67   (kerekítés 2-es tizedesjegyre)
print(math.ceil(szamla*100)/100)        #45.68   (felfelé kerekítés)
```

11.4. A négyzetgyök

A **négyzetgyököt** a math modulban található ***math.sqrt(x)*** beépített függvénnyel számítjuk ki. Mindig float (lebegőpontos) típusú számot ad vissza.

Jellemzők és használat:

- Használat előtt be kell importálni a math csomagot
- Az ***x*** argumentumnak **pozitívnak** vagy **nullának** kell lennie (ellenkező esetben ***ValueError*** hibaüzenet jelenik meg)
- ***float*** típusú értéket ad vissza akkor is, ha az eredmény ***egész*** szám: `math.sqrt(25)` → `5.0`
- Tipikus alkalmazások: két pont közötti távolság (Pitagorasz-tétel)

```
import math
math.sqrt(16)          #  4.0
math.sqrt(2)           #  1.4142135623730951
math.sqrt(0)           #  0.0
# Példa: két pont közötti távolság (Pitagorasz-tétel)
def tavolsag_2d(x1, y1, x2, y2):
    return math.sqrt((x2 - x1)**2 + (y2 - y1)**2)
tavolsag_2d(0, 0, 3, 4)     #  5.0
```

11.5. Konverziók (típusátalakítások)

A **típuskonverzió** (típusátalakítás) alapvető fontosságú, amikor a felhasználótól kapunk adatokat (amelyek mindig karakterlánc formájában érkeznek), és azokat matematikailag kell feldolgoznunk. A Python beépített függvényt kínál a leggyakoribb konverziókhoz, amelyek importálás nélkül, közvetlenül elérhetőek.

Függvény	Mit csinál?	Példa
int(x)	Egy karakterláncot egész számmá alakít, vagy egy valós szám egész részét adja vissza	`int('42')` → **42** | `int(3.9)` → **3**
float(x)	Egy értéket valós számmá alakít	`float('3.14')` → **3.14** | `float(5)` → **5.0**
str(x)	Egy értéket karakterlánccá alakít	`str(42)` → **'42'** | `str(3.14)` → **'3.14'**
bool(x)	Egy értéket logikai értékké alakít	`bool(0)` → **False** | `bool(5)` → **True**

Jellemzők és használat:

- ***int()*** egy karakterláncra alkalmazva csak akkor működik, ha a karakterlánc érvényes egész számot tartalmaz: `int('3.14')` → hiba
- ***float()*** egy karakterláncra alkalmazva mind egész, mind valós számokat elfogad: `float('3')` → **3.0**
- ***str()*** bármilyen adattípuson működik, hasznos karakterlánc-összevonáshoz és megjelenítéshez

```
# Felhasználói adatok beolvasása és konverziója (Típusátalakítása)
kor = int(input('Kor: '))   # '18' → 18
jegy   = float(input('Jegy: '))    # '9.5' → 9.5
# Konverzió a láncolt (összefűzött) megjelenítéshez
uzenet = 'A te jegyed: ' + str(jegy)    # float → str
print(uzenet)   # Számtípusok közötti átalakítás
x = 7 / 2       #  3.5  (float)
y = int(7 / 2)  #  3    (int, levágással)
```

11.6. Beépített függvények értékgyűjteményekhez (kollekciókhoz)

Amikor adathalmazokkal – például listákkal – dolgozunk, gyors mechanizmusokra van szükségünk ahhoz, hogy információkat nyerjünk ki a teljes elemcsoportról. A Python olyan beépített függvényeket kínál, amelyek megkímélik a programozót attól, hogy egyszerű számításokhoz ciklusokat kelljen írnia.

Az elemek számának meghatározása – len()

A **len(gyűjtemény)** függvény a gyűjtemény elemeinek számát adja vissza.

Jellemzők:

- listákon, tuple-okon, karakterláncokon és bármilyen Python-gyűjteményen működik;
- egészet térít vissza;
- az üres gyűjtemény 0-t ad vissza.

```
jegyek = [10, 8, 9, 7, 6]
len(jegyek)           #  5
len([])               #  0  (üres lista)
len('informatika')    #  11 (működik karakterláncokkal is)
```

A legkisebb érték meghatározása – min()

A **min(gyűjtemény)** függvény a gyűjtemény legkisebb értékét adja vissza.

Jellemzők:

- a gyűjtemény nem lehet üres (különben *ValueError* hiba lép fel);
- az elemeknek egymással összehasonlíthatóknak kell lenniük;
- különálló argumentumokat is kaphat: min(a, b, c).

```
jegyek = [10, 8, 9, 7, 6]
min(jegyek)           #  6
min(3, 1, 7, 2)       #  1  (lista nélkül, különálló argumentumok)
```

A legnagyobb érték meghatározása – max()

A **max(gyűjtemény)** függvény a gyűjtemény legnagyobb értékét adja vissza.

Jellemzők:

- azonosak a min()-el;
- különálló argumentumokat is kaphat: max(a, b, c).

```
jegyek = [10, 8, 9, 7, 6]
max(jegyek)           # 10
max(3, 1, 7, 2)       # 7
```

Az értékek összegének kiszámítása – sum()

A **sum(gyűjtemény)** függvény a gyűjtemény összes elemének összegét adja vissza.

Jellemzők:

- számgyűjteményeken (számok kollekcióin) működik;
- az üres lista 0-t ad vissza;
- elfogad egy második, opcionális argumentumot kezdőértéknek: sum(lista, start).

```
jegyek = [10, 8, 9, 7, 6]
sum(jegyek)       #  40
sum([])           #  0   (üres lista)
sum(jegyek, 100)  #  140  (100-tól indul)
```

11.7. Példa: egy naplóbeli jegylista elemzése

```
import math
jegyek = [10, 8, 9, 7, 6, 5, 9, 8]
def jegy_elemzes(jegyek):
    szam   = len(jegyek)                        # elemek száma
    minimum = min(jegyek)                       # minimum
    maximum = max(jegyek)                       # maximum
    ossz   = sum(jegyek)                        # összeg
    atlag  = round(ossz / szam, 2)              # kerekítés
    elteres = math.sqrt(maximum - minimum)      # négyzetgyök
    return minimum, maximum, atlag, elteres
mn, mx, med, ab = jegy_elemzes(jegyek)
```

Megj. Ezek a függvények arra vannak optimalizálva, hogy végighaladjanak a gyűjteményen és azonnal visszaadják az eredményt, így sokkal hatékonyabbak, mint a lista bejárása egy for ciklussal.

Feladatlap 11

Hozzatok létre egy *Feladatlap11* nevű mappát. Oldjátok meg az alábbi feladatokat, és írjátok be a helyes válaszokat egy *valaszok.docx* nevű állományba.

1. Elemezd az alábbi kódot. Mit ír ki és miért?

```
x = 15.7
print(int(x))
print(round(x))
```

2. Írj egy függvényt, amely paraméterként egy árlistát kap, és visszaadja a legdrágább termék árát.

3. Mi történik, ha a ***sum()*** függvényt olyan listára hívod meg, amely számokat és karakterláncokat is tartalmaz? Magyarázd el a kapott hibát.

4. Tekintsük az alábbi függvényt. Mit fog kiírni, ha a kapott lista [10, 20, 30, 40]?

```
def feldolgoz(lista):
    return (max(lista) + min(lista)) / 2
print(feldolgoz([10, 20, 30, 40]))
```

5. Mi a különbség az ***int()*** és a ***round()*** függvény használata között pozitív valós szám esetén?

6. Írj egy ***atlag_szamitas()*** nevű függvényt, amely egy jegylistát kap paraméterként, kiírja a bevitt jegyek számát (***len***), valamint azok átlagát (***sum / len***).

7. Mit ír ki az alábbi kód?

```
import math
x = -25
print(abs(x))
print(math.sqrt(abs(x)))
```

8. Számítsd ki az A(1, 2) és B(4, 6) pontok közötti távolságot a ***math.sqrt()*** és ***abs()*** függvényeket alkalmazva.

9. ***Mini-projekt:*** Készíts egy valutaváltó kalkulátort – egy függvényt amely egy összeget és egy alapértelmezett EUR/RON = 5.0 árfolyamot kap paraméterként, és a round() függvény segítségével 2 tizedesjegyre kerekítve jeleníti meg az eredményt.



Önértékelés!

Az *onertekeles11.docx* állományba írjátok be a megfelelő válaszokat, a következő jelöléseket használva: *1 – igen, 2 – nem, 3 – nem vagyok biztos benne*, majd mentsétek el a *Feladatlap11* mappába.

	A lecke végén tudom...	***Igen***	***Nem***	***Nem biztos***
1.	mi az abszolút érték, és hogyan számoljuk ki az abs() függvénnyel.			
2.	az int(), a math.floor() és a math.trunc() közötti különbséget az egészrész kiszámításakor.			
3.	hogyan működnek a round(), a math.floor() és a math.ceil() függvények a kerekítésnél.			
4.	hogyan számoljuk ki a gyököt a math.sqrt() függvénnyel.			
5.	milyen beépített függvények állnak rendelkezésre az átalakításokhoz/konverzióhoz.			
6.	milyen beépített függvények léteznek a gyűjteményekhez.			

7. Nehézségeim voltak a következőkben: …

Ne felejtsétek el értékelni a viselkedéseteket és a tevékenységeteket az óra során!
(a 7. oldalon található viselkedés- és tevékenységfigyelő napló segítségével)

A lecke végén így éreztem magam:

12. Lecke – Bevezetés az objektumorientált programozásba (OOP): osztályok, objektumok és példányosítás

Eddig procedurális kódot írtunk, vagyis egymás után végrehajtott utasítások sorozatát. Ahogy a programok bonyolultabbá válnak, az adatok és a hozzájuk kapcsolódó függvények kezelése nehezebbé válik. Az objektumorientált programozás (OOP) lehetővé teszi számunkra a valóság modellezését úgynevezett ***objektumok*** létrehozásával, amelyek ötvözik az ***adatokat*** (attribútumokat) és az ***alprogramokat*** (metódusokat).

12.1. Az osztály

Egy osztály egy sablon vagy tervrajz. Gondolj rá úgy, mint egy ház alaprajzára. Egy objektum az adott osztály konkrét példánya, vagyis maga a tervrajz alapján felépített ház. Ugyanabból az osztályból tetszőleges számú objektum hozható létre, amelyek mind saját adatokkal rendelkezhetnek.

Egy osztály definiálása



Az **osztály** egy sablon, amely meghatározza egy objektumtípus felépítését és viselkedését. Megadja, hogy egy objektum milyen adatokat tárolhat és milyen műveleteket hajthat végre, anélkül, hogy konkrét objektumokat hozna létre. Az osztályokat a ***class*** kulcsszóval definiáljuk.

Egy osztály jellemzői:

- Csak egyszer kell definiálni, de tetszőleges számú objektum létrehozására használható
- Két tagkategóriát tartalmaz: ***adatokat*** (attribútumokat) és ***alprogramokat*** (metódusokat)
- Pythonban a ***class*** kulcsszóval definiáljuk, amelyet az osztály neve követ (nagybetűvel)
- Hasonlat: az osztály egy sablon, amely alapján objektumokat hozunk létre. Ahogyan egy ház tervrajza vagy vázlata nem egy ház, úgy az osztály sem egy objektum, hanem leírja, hogy milyenek lesznek az alapján létrehozott objektumok

```
class Diak:             # osztály definíciója – SABLONJA
    pass                # (minimális törzs, tagok nélkül)
```

12.2. Az osztály tagjai – adatok és metódusok



Az **osztály tagjai** az osztályon belül definiált elemek. Két kategóriába sorolhatók:

Adatok (attribútumok):



Az **adatok** egy objektumhoz tartozó változók, amelyek az objektummal kapcsolatos információkat tárolnak. Ezeket az osztály konstruktorában a ***self.attribútum_név = érték*** kifejezéssel határozzuk meg.

- Minden objektumnak megvannak a saját attribútumértékei, függetlenül a többi objektumtól;
- Kívülről az ***objektum.attribútum*** szintaxissal érhetők el.

Metódusok:



A **metódusok** az osztályon belül definiált alprogramok (függvények), amelyek az objektumok viselkedését írják le. Első paraméterük mindig a ***self*** – az aktuális objektumra való hivatkozás.

- A metódusok a self segítségével érhetik el és módosíthatják az objektum attribútumait;
- Külső hívásuk az ***objektum.metódus(paraméterek)*** szintaxissal történik.

12.3. Osztálykonstruktor (__init__) és a self operátor



A konstruktor[1] az osztály egy speciális metódusa, amely **automatikusan** végrehajtódik egy objektum példányosításának pillanatában – vagyis amint leírjuk az *objektum = OsztalyNeve(argumentumok)* sort. Feladata az objektum belső állapotának inicializálása azáltal, hogy a létrehozás pillanatától kezdve konkrét értékeket rendel az attribútumaihoz.

[1] Az OOP-ban a konstruktor az a mechanizmus, amelyen keresztül egy újonnan létrehozott objektum megkapja adatainak kezdeti értékeit. Konstruktor nélkül az objektum létezne ugyan a memóriában, de „üres” lenne — bármiféle hozzárendelt adat nélkül.

A self operátor:



A **self** az a mód, amellyel egy objektum saját magára hivatkozik a metódusokon belül. Amikor egy objektum metódusát meghívjuk, a Python automatikusan átadja az adott objektumot első argumentumként (self), így a metódus pontosan tudja, melyik konkrét objektumról van szó.

```
class Allat:
    def __init__(self, nev, kor):
        #az egyes példányokhoz tartozó attribútumok inicializálása
        self.nev = nev      # attribútum- megmarad a memóriában
        self.kor = kor      # attribútum - megmarad a memóriában
    def leiras(self, x):
        # self.kor = objektum attribútuma
        # x = külső paraméter, csak ezen metódus futása alatt létezik
        return self.kor, x
# Objektumok létrehozása (példányosítás) a konstruktor meghívásával
kutya = Allat("Bobby", 2)
cica = Allat("Katy", 5)

print(kutya.nev)  # Elérjük az első objektum nev attribútumát - Bobby
print(cica.nev) # Elérjük a második objektum nev attribútumát - Katy
print(kutya.leiras(7))      # (2, 7)
```



- A ***self.atribut*** azt jelenti: „az aktuális objektum attribútuma”;
- Az attribútumok (self.nev, self.kor) olyan változók, amelyek külön-külön minden egyes objektumhoz „hozzá vannak kapcsolva”, és mindaddig megmaradnak a memóriában, amíg az objektum létezik;
- A metódusok paraméterei (például: x) függetlenek az objektum állapotától, és kizárólag a metódus futása alatt léteznek.

A Pythonban a konstruktort kötelezően az ***__init__*** névvel kell definiálni (az initialize – inicializálás szóból). Gyakorlatilag a konstruktor megkímél minket attól, hogy az objektum létrehozása után le kelljen írnunk a kutya.nev = 'Bobby' sort, mivel az adatok beállítását egyetlen lépésben elvégzi. Ebben a példában a ***self*** az éppen használatban lévő, aktuális objektumot képviseli. Figyeld meg, hogy az **x** és a ***self.kor*** teljesen különálló változók.

Az objektumok elszigetelése a memóriában

Minden objektum saját memóriaterülettel rendelkezik. A kutya objektum egyik attribútumának módosítása semmilyen formában nem befolyásolja a cica objektum megfelelő attribútumát. Ez az elszigeteltség lehetővé teszi több ezer különböző entitás kezelését ugyanazon sablon – az osztály – használatával.

```
class Diak:
    def __init__(self, nev, atlag):      # konstruktor
        self.nev = nev                   # adat (attribútum)
        self.atlag = atlag               # adat (attribútum)
    def koszon(self):                    # metodus
        print(f'Szia! {self.nev}. vagyok.')
    def atmeno(self):                    # metodus
        return self.atlag >= 5
```



Az ***__init__*** egy speciális konstruktor, amely automatikusan meghívódik egy objektum létrehozásakor, és az objektum attribútumainak inicializálására szolgál.

12.4. Speciális metódusok (Dunder Methods)

A szokásos metódusokon kívül a Python **speciális metódusokat** is kínál (ezeket **dunder metódusoknak** is nevezik a double underscore – dupla aláhúzás kifejezésből). Ezek __ jellel kezdődnek és végződnek, és előre meghatározott szerepük van a nyelvben — közvetlenül meghívhatók a megfelelő operátorral vagy függvénnyel, anélkül, hogy explicit módon nevükön neveznénk őket.

Speciális metódus	Szerep és kiváltó ok
__init__(self, ...)	A konstruktor – automatikusan meghívódik példányosításkor: Allat('Bobby', 2)
__str__(self)	Szöveges reprezentáció – automatikusan a print() hívja meg: print(objektum)
__eq__(self, alt)	Az == (egyenlőség) operátor definiálása: objektum1 == objektum2
__lt__(self, alt)	A < (less)operátor definiálása: objektum1 < objektum2
__le__(self, alt)	A <= (less than or equal to) operátor definiálása: objektum1<= objektum2
__gt__(self, alt)	A >(greater than) operátor definiálása: objektum1 > objektum2
__ge__(self, alt)	A >=(greater than or equal to)) operátor definiálása: objektum1 >= objektum2

```
class Allat:
    def __init__(self, nev, kor):
        self.nev   = nev
        self.kor = kor
    def __str__(self):                  # az str() hívja meg
        return f'{self.nev} {self.kor} éves.'
    def __eq__(self, mas_allat):        # == hívja meg
       # Két állatot akkor tekintünk azonosnak, ha megegyezik a nevük
        return self.nev == mas_allat.nev
    def __lt__(self, mas_allat):        # a < hívja meg
        # kor alapján összehasonlítjuk az állatokat
        return self.kor < mas_allat.kor
# Létrehozunk három objektumot (osztály példányát)
pufu = Allat('Pufu', 3)
pufi = Allat('Pufi', 5)
bobi = Allat('Bobi', 2)
# Az str() automatikus meghívásával kiiratjuk
print(pufu)           # Pufu 3 éves.   (__str__ meghívásával)
print(pufi)           # Pufi 5 éves.   (__str__ meghívásával)
print(bobi)           # Bobi 2 éves.   (__str__ meghívásával)
# Egyenlőséget ellenőrzünk  (__eq__)
print(pufu == pufi)   # False              (__eq__ meghívásával)
# sorrendet ellenőrzünk (__lt__)
print(bobi < pufu)    # True  (2 < 3)      (__lt__ meghívásával)
print(pufi < bobi)    # False (5 < 2)      (__lt__ meghívásával)
```

A függvényeket nevükkel, de közvetlenül is meg lehet hívni a megfelelő szimbólum használatával (*például* a *print(objektum)* automatikusan meghívja a *__str__ metódust*).

12.5. Az objektum



Az **objektum** egy osztály konkrét példánya. Ha az osztály a sablon, akkor az objektum a sablon alapján létrehozott termék – saját adatértékekkel, de ugyanazokkal a metódusokkal, mint az ugyanazon osztályba tartozó összes többi objektum.

Az objektum jellemzői:

- Saját attribútumértékekkel rendelkezik, amelyek függetlenek az azonos osztály más objektumaitól;
- Megosztja az osztály metódusait a többi objektummal;
- Tárolható változókban, listákban vagy átadható függvények argumentumaként;
- Analógia: ha a ***Diak*** osztály a minta, akkor a ***diak1*** és ***diak2*** konkrét diákok, mindegyik a saját nevével és jegyével.

```
diak1 = Diak('Anna', 8.5)   # objektum - konkrét példány
diak2 = Diak('Mihaly', 4.2) # másik objektum - másik példány
# a diak1 és diak2 ugyanazokkal a metódusokkal rendelkezik (koszon, atmeno)
# az attribútumaik értékei (nev,atlag) különbözőek
```

Esettanulmány: Állatorvosi rendelő kezelése

Képzeld el, hogy egy állatorvosi rendelő recepcióján dolgozol. Ahelyett, hogy minden állat adatait külön papírra írnád, van egy szabványos űrlapod. Ez az **osztály**. Amikor érkezik egy új állat, kitöltesz egy új űrlapot. A „Gréti" adataival kitöltött lap az **objektum**.

Ahelyett hogy számtalan változót használnánk, létrehozunk egy sablont:

```
class Allat:
    nev = 'Greti'
    faj = 'kutya'
    gazdi = 'Anna'
    kor = 4
# létrehozunk egy Bobi nevű halat
bobi = Allat()
bobi.nev = 'Bobi'
bobi.faj = 'Hal'
print(bobi.nev, bobi.faj, bobi.gazdi)
```

12.6. Könyvtárak



A **könyvtárak** (modulok és csomagok) előre elkészített osztályok, függvények és konstansok tematikus gyűjteményei, amelyek kiegészítik a Python nyelv alapvető funkcióit. Lehetővé teszik más programozók kódjának újrafelhasználását.

Könyvtárak jellemzői:

- **import** vagy **from** ... **import** ... utasítással importálhatók
- A Python gazdag szabványkönyvtárral rendelkezik (math, datetime, random, os stb.)
- Külső könyvtárak is telepíthetők. (numpy, pandas, tkinter stb.)

```
import math                          # importálja a teljes math könyvtárat
from datetime import date            # csak a date osztályt importálja a datetime-ból
# Már használhatjuk a date osztályt és a math függvényeit:
ma = date.today()                    # date osztály használata
r = math.sqrt(16)                    # a sqrt függvényt használjuk a math könyvtárból
```

12.7. Előre definiált osztály példányosítása

A **példányosítás** során konkrét objektumot hozunk létre egy osztályból. Ez a folyamat az osztály meghívásával történik, mintha egy alprogram lenne, átadva a konstruktornak a szükséges argumentumokat:

Ahhoz, hogy egy osztályból objektumot hozzunk létre, egy **példányosításnak** nevezett folyamatot hajtunk végre. Pythonban az osztály neve után zárójeleket használunk a példány létrehozásához.

Az osztályokban lévő függvények mindig a self operátorral kezdődnek – ez az adott példányra való hivatkozás. A hivatkozásnak bármilyen nevet adhatunk, a self csupán egy egyezményes elnevezés.

```
# Általános szintaxis:
# objektum = OsztalyNev(argumentum1, argumentum2, ...)
# az általunk definiált Diak osztaly példányosítása:
diak1 = Diak('Anna', 8.5)
# a datetime könyvtárból a date osztály példányosítása:
from datetime import date
ma = date.today()                    # osztálymetódus, ami objektumot hoz
szulinap = date(2008, 3, 15)         # példányosítás argumentumokkal: év,hónap,nap
# az előredefiniált list osztály példányosítása:
jegyek = list()                      # vagy: jegyek = []
karakterek = list('abc')             # → ['a', 'b', 'c']
```

12.8. Az objektum tagjainak elérése

Az **objektum tagjaihoz** a pont (.) operátor segítségével lehet hozzáférni, amely lehetővé teszi mind az attribútumok olvasását/módosítását, mind a metódusok meghívását:

Attribútumok elérése:

```
diak1 = Diak('Anna', 8.5)
# attribútum kiolvasása:
print(diak1.nev)              #  Anna
print(diak1.atlag)            #  8.5
# attribútumok módosítása:
diak1.atlag = 9.0
print(diak1.atlag)            #  9.0
```

Metódusok elérése:

```
# metódusok meghívása:
diak1.koszon()                      #  Szia! Anna vagyok.
eredmeny = diak1.atmeno()           #  True
# Hozzáférés a datetime.date osztály tagjaihoz:
from datetime import date
ma = date.today()
print(ma.year,ma.month)             # aktuális év,hónap attribútum
print(ma.strftime('%d/%m/%Y'))      # dátum alakító metódus
```

Szintaxis	Jelentés
objektum.attribútum	Az objektum egyik attribútumának értékének kiolvasása
objektum.attribútum = érték	Az objektum egyik attribútumának értékének módosítása
objektum.metódus()	Egy metódus meghívása további argumentumok nélkül
objektum.metódus(arg1, arg2)	Egy argumentummal rendelkező metódus meghívása



Jegyezd meg!

- ✔ **Osztály** – sablon, amely meghatározza egy objektumtípus szerkezetét és viselkedését
- ✔ **Osztálytagok** – attribútumok (adatok) és metódusok (függvények)
- ✔ **Objektum** – az osztály konkrét példánya
- ✔ **Könyvtárak** – előre elkészített osztályok és függvények gyűjteményei (import / from ... import)
- ✔ **Példányosítás** – objektum létrehozása: objektum = OsztalyNev(argumentumok)
- ✔ **Tagok elérése** – a pont operátorral: objektum.attribútum és objektum.metódus()



Feladatlap 12

Hozzatok létre egy *Feladatlap12* nevű mappát. Oldjátok meg az alábbi feladatokat, és írjátok be a helyes válaszokat egy *valaszok.docx* nevű állományba.

1. Mi a fő különbség egy osztály és egy objektum között? Adj egy valós életből vett példát.
2. Mi a szerepe a self paraméternek egy metóduson belül?
   a. Egy globális változót jelöl, amely bárhonnan elérhető.
   b. Az osztály aktuális példányára (objektumára) mutató hivatkozás.
   c. Egy kulcsszó, amely leállítja a program végrehajtását.

3. Ha definiálunk egy Kutya nevű osztályt egy név attribútummal, írd meg a szükséges kódot két különböző objektum (két különböző nevű kutya) létrehozásához.

4. Azonosítsd a hibát a következő kódban, és magyarázd el, miért nem fog lefutni:

```
class Auto:
    def __init__(nev):
        self.nev = nev
        auto = Auto('Dacia')
```

5. Írj egy Diák nevű osztályt, amely rendelkezik név és átlag attribútumokkal! Adj hozzá egy kituno_tanulo() nevű metódust, amely *True* értéket ad vissza, ha az átlag nagyobb vagy egyenlő 9-cel, ellenkező esetben *False*-t.

6. Definiálj egy Auto nevű osztályt a következő adatokkal: marka, modell, gyartasi_ev. Adj hozzá egy eletkor() és egy leiras() metódust, amely megjeleníti az összes adatot.

7. Hozz létre 3 Auto típusú objektumot! Érd el mindegyik objektum attribútumait és metódusait! Módosítsd az egyik objektum gyartasi_ev attribútumát, és ellenőrizd, hogy a többi objektum adatai nem változtak meg.

8. ***Mini-projekt:*** Bővítsd a Napló osztályt a következő metódusokkal: minim_jegy(), maximum_jegy() és atment(kuszob=5). Hozz létre két Napló objektumot különböző tantárgyak számára.



Önértékelés!

Az *onertekeles12.docx* állományba írjátok be a megfelelő válaszokat, a következő jelöléseket használva: *1 – igen, 2 – nem, 3 – nem vagyok biztos benne*, majd mentsétek el a *Feladatlap12* mappába.

	A lecke végén tudom...	***Igen***	***Nem***	***Nem biztos***
1.	mi az osztály.			
2.	mik az osztály tagjai.			
3.	mi az (__init__) konstruktor.			
4.	mi a self.			
5.	mik a speciális metódusok (dunder methods) és hogyan működnek.			
6.	mi az objektum.			
7.	mik a modulok és csomagok.			
8.	mi a példányosítás és hogyan történik.			
9.	hogyan lehet hozzáférni egy objektum tagjaihoz.			

10. Nehézségeim voltak a következőkben: …

Ne felejtsétek el értékelni a viselkedéseteket és a tevékenységeteket az óra során!
(a 7. oldalon található viselkedés- és tevékenységfigyelő napló segítségével)

A lecke végén így éreztem magam:

3. Fejezetvégi felmérés

Ellenőrizd a tudásodat!

(Pontszám: 90 pont + 10 hivatalból)

Követelmény: Oldjátok meg az alábbi feladatokat a ***tudas_ellenorzes3.docx*** fájlban. Az összes létrehozott fájlt mentsétek a ***Fejezet3*** mappába.

1. Válaszd ki azt a kijelentést, amely a legjobban leírja a lokális és a globális változó közötti különbséget Pythonban:

a. A lokális változó mindenhol működik, a globális csak függvényekben.

b. A globális változót a függvényeken kívül deklaráljuk, a lokális változót a függvényben deklaráljuk, és a függvényhívás után megsemmisül.

c. Csak az írásmódban különböznek: a lokális változókat kisbetűvel kell írni.

d. A lokális változó csak egész számokat tárolhat, a globális pedig bármilyen típusú értéket.

2. Elemezd a mellékelt kódot. Mi fog megjelenni a képernyőn és miért?

```
class Tort:
    def __init__(self, szamlalo, nevezo):
        self.szamlalo = szamlalo
        self.nevezo = nevezo
f1 = Tort(1, 2)
f2 = f1
f2.szamlalo = 3
f2.nevezo = 4
print(f1.szamlalo, f1.nevezo)
```

3. Párosítsd az **A** oszlopban található függvényeket/fogalmakat a **B** oszlop megfelelő leírásával.

Függvény/fogalom	Leírás/hatás
1. math.ceil(4.1)	a. Egy konkrét objektum létrehozásának folyamata egy osztály alapján.
2. int(9.8)	b. Meghatározza egy szám abszolút értékét (eltávolítja a mínusz jelet).
3. Példányosítás	c. Egész számmá alakítja a számot a tizedesjegyek levágásával, 9-et ad vissza.
4. abs(x)	d. Kiszámítja egy pozitív szám négyzetgyökét.
5. math.sqrt(x)	e. Felfelé kerekít a következő egész számra, 5-öt ad vissza.

4. a. Írd meg a **sub** nevű függvény teljes definícióját, amely két paramétert kap, ***a*** és ***b*** (természetes számok). A függvény az ***a*** egyesek helyiértékén lévő számjegyét hozzáfűzi ***b*** végéhez, majd visszaadja ***b*** módosított értékét.

b. Írj egy programot, amely a billentyűzetről beolvas egy ***n*** természetes számot, amely páros számú számjegyből áll. Kizárólag egyszerű változók és a sub függvény meghívásainak felhasználásával ellenőrizd, hogy a szám első fele és második fele azonos-e. A program az **IGEN** vagy a **NEM** üzenetet jelenítse meg.

- 1. Példa: Ha n = 12051205, az első fele: 1205, a második fele: 1205. A két rész azonos, ezért a program az **IGEN** üzenetet ír ki.
- 2. Példa: Ha n = 12344321, az első fele: 1234, a második fele: 4321. A két rész nem azonos, ezért a program a **NEM** üzenetet írja ki.

Értékelési útmutató (100 pont) hivatalból – **10p**, **1.feladat**-b (**10p**), **2.feladat** (**15p**): **kiírt értékek** - **8p**, **indoklás** - **7p**, **3.feladat** – (**15p** = **5x3p**) **1-e, 2-c, 3-a, 4-b, 5-d, 4.feladat** – (**50p**): **a)alpont (20p): (6p** helyes fejléc, **10p** a számjegyáthelyezés algoritmusa, **4p** helyes eredmény visszatérítése), **b)alpont (30p)**: (Adatbeolvasás, kizárólag egyszerű változók használata, a sub függvény helyes hívása a szám felépítéséhez, a két fél feldolgozásának helyessége, az IGEN/NEM üzenet megfelelő kiírása, helyes szintaxis, általános algoritmus-helyesség).



Projekt

(100 pont)

„Átváltó kalkulátor”

Készíts egy ***szamitasok.py*** nevű Python-programot, amely moduláris felépítésű, és egy interaktív menüt jelenít meg az alábbi átváltások elvégzéséhez. Írásban indokold meg a függvényekben használt lokális változók szerepét.

MENÜ	Technikai követelmények
1. **EURO → RON**: (napi árfolyamot a billentyűzetről olvasd be). Az eredményt 2 tizedesre kerekítsd. 2. **Celsius → Fahrenheit**. 3. **Centiméter → hüvelyk**: A távolságot abszolút értékként dolgozd fel. 4. **Kilépés** (leállítja a programot).	Minden átváltást **külön függvényben** kell kiszámítani. A függvények paramétereket fogadjanak és értéket adjanak vissza. A menünek egy while ciklusban kell futnia, amíg a felhasználó a 4-es opciót nem választja. Használd az adattípus-konverziós függvényeket (float(), int()).

Értékelési útmutató (100 pont): I. Technikai megvalósítás (30p): 3×10p minden egyes helyesen működő átváltásért. **II. Program (40p)**: 3 függvény × **5p** = **15p**, **20p** – menü létrehozása és helyes működése, **5p** – A kód általános minősége. **III. Elmélet és bemutatás (30p)**: **10p** – Indoklás, **10p** – szakszerű fogalomhasználat és előadásmód, **10p** – kreativitás, részletekre való figyelem (pl. menü személyre szabása, hibás menüpontok kezelése).

(**Ajánlások**: A munka 2-3 fős diákcsapatokban zajlik. Minden csapat egy értékelőlap alapján pontozza a többi csapatot, a végső pontszám pedig az elért pontszámok számtani közepe lesz.)



Kutass!

(100 pont)

Esettanulmány: Meteorológiai Állomás?

Az ***ido.docx*** fájlban dokumentáljátok az alábbi helyzetet: Az adott program egy meteorológiai állomás adatait dolgozza fel alprogramok és beépített függvények segítségével. Vizsgálandó feladatok:

- Indokold meg, miért különbözik a round(-5.71) és math.floor(-5.71) eredménye? **(30p)**
- Azonosítsd a programban: a függvény fejlécét, törzsét, egy paramétert. **(15p)**
- Elemezd, hogy a függvényen belül a homerseklet változón végzett módosítások miért nem befolyásolják a homerseklet_globalis változót? **(15p)**
- Azonosítsd a használt könyvtárat és a meghívott függvényt. Keress és használj még két függvényt ugyanebből a könyvtárból (például négyzetgyök-számításra vagy felfelé kerekítésre). **(40p)**

```
import math
homerseklet_globalis = 21.65
def adat_homerseklet(homerseklet):
    homerseklet_abs = abs(homerseklet)
    print(f"Abszolut ertek: {homerseklet_abs}")
    print(f"Kerekitve: round={round(homerseklet)},"
          f"floor={math.floor(homerseklet)}")
adat_homerseklet(-5.71)
print(f"A globalis homerseklet: {homerseklet_globalis}")
```



Portfólió

(100 pont)

A fejezet végén a diák személyes portfóliójának a ***Fejezet3*** mappában a következő anyagokat kell tartalmaznia: ***Feladatlap9***, ***Feladatlap10***, ***Feladatlap11***, ***Feladatlap12*** mappák (mindegyikben a ***valaszok.docx*** és ***onertekeles.docx*** fájlokkal) és a ***tudas_ellenorzes3.docx***, ***ido.docx***, ***szamitasok.py*** fájlok.

Értékelés: Létezik a ***Fejezet3*** mappa a megadott könyvtárszerkezettel – **20p**

Megtalálható:
- a fájlok több mint 95%-a – **80p**
- 75% és 94% között – **60p**
- 50% és 74% között – **40p**
- 25% és 49% között – **20p**
- 25% alatt – **0p**

4. Fejezet – Grafikus interfészek és szöveges fájlok

13. Lecke – A Tkinter könyvtár: ablakok létrehozása és az elrendezéskezelés

Bevezetés a Tkinter használatába és az első grafikus ablakok létrehozása

*„A jó programokat a kortársaknak írják, hogy olvassák, és csak mellesleg a számítógépeknek, hogy végrehajtsák.” – **Harold Abelson***

13.1. A grafikus interfész és a Tkinter könyvtár

<def/>

A programozásban az alkalmazások alapvetően kétféle módon működhetnek:

- *parancssoros felületen (CLI – Command Line Interface)*;
- *grafikus felhasználói felülettel (GUI – Graphical User Interface).*

A ***grafikus interfész*** lehetővé teszi a felhasználó számára, hogy ablakokon, gombokon, szövegbeviteli mezőkön, menükön és egyéb vizuális elemeken keresztül interakcióba lépjen a programmal.

A ***Python*** nyelvben a grafikus interfészek készítéséhez az egyik leggyakrabban használt könyvtár a ***Tkinter***.

A ***Tkinter*** standard könyvtár a GUI alkalmazások fejlesztéséhez Pythonban, és biztosítja a szükséges eszközöket az ablakok létrehozásához, a vizuális elemek elrendezéséhez és az eseménykezeléshez.

A Tkinter könyvtár importálása

<def/>

A Tkinter könyvtár használatához szükség van annak importálására.

A leggyakrabban használt szintaxis: `import tkinter as tk`

A ***tk*** alias (álnév) használatával a könyvtár komponensei sokkal egyszerűbben meghívhatók.

Pl.

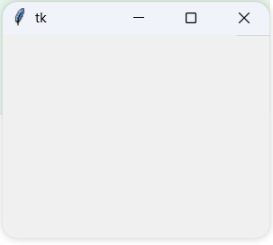
```
import tkinter as tk
```

Főablak létrehozása

<def/>

Minden Tkinter alkalmazás egy főablak létrehozásával kezdődik.

A főablak képviseli azt a teret, ahová az alkalmazás összes grafikus komponense elhelyezésre kerül.



Pl.

```
import tkinter as tk
foablak = tk.Tk()
foablak.mainloop()
```

Magyarázat:

- `tk.Tk()` létrehozza a főablakot;
- `mainloop()` elindítja az alkalmazás fő ciklusát (ciklus) és aktívan tartja az ablakot.

A `mainloop()` *metódus nélkül az ablak a program lefutása után azonnal bezáródna.*

Az ablak testreszabása

<def/>

A főablak a következők szerint állítható be: a cím beállítása; a méret módosítása; a háttérszín megváltoztatása; az átméretezés letiltása.

A cím beállítása

<def/>

Az ablak *címe a* `title()` metódussal állítható be.

Pl.

```
foablak.title("Első Tkinter alkalmazás")
```

Az ablak méretének beállítása

<def/> A *méret* a **`geometry()`** metódussal határozható meg.

Pl.
```
foablak.geometry("400x300")
```

Az *érték* képpontokban (pixel) van megadva: 400 = szélesség; 300 = magasság.

A háttérszín megváltoztatása

<def/> A *háttérszín* a **`config()`** metódussal módosítható.

Pl.
```
foablak.config(bg="lightblue")
```

Az átméretezés letiltása

<def/> Az *átméretezés* engedélyezhető vagy letiltható.

Pl.
```
foablak.resizable(False, False)
```

A *paraméterek* jelentése: vízszintes átméretezés; függőleges átméretezés.
A **`False`** *érték* megtiltja az átméretezést.

Pl.

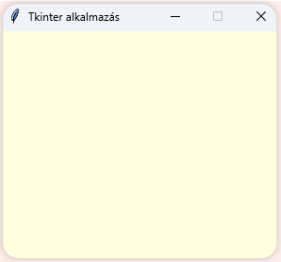
Teljes példa egy ablak létrehozására

```
import tkinter as tk

foablak = tk.Tk()

foablak.title("Tkinter alkalmazás")
foablak.geometry("300x250")
foablak.config(bg="lightyellow")
foablak.resizable(False, False)

foablak.mainloop()
```



13.2. Widgetek a Tkinterben

<def/> Az ablakhoz hozzáadott *grafikus elemeket* ***widgeteknek*** nevezzük.

A leggyakrabban használt widgetek a következők:

- `Label` – szöveget jelenít meg;
- `Button` – gombokat hoz létre;
- `Entry` – lehetővé teszi az adatbevitelt;
- `Text` – hosszabb szövegeket jelenít meg;
- `Frame` – más komponenseket rendszerez;
- `Checkbutton` – jelölőnégyzeteket hoz létre;
- `Radiobutton` – lehetővé teszi egy opció kiválasztását.

A `Label` widget– a szöveg megjelenítésére használatos.

Pl.
```
import tkinter as tk

foablak = tk.Tk()

szoveg = tk.Label(foablak, text="Üdv, Tkinter!")
szoveg.pack()

foablak.mainloop()
```

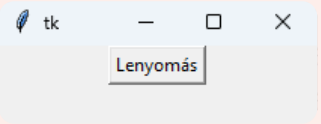


Magyarázat: • az első paraméter azt az ablakot jelöli, amelyben a widget elhelyezkedik;
• a text paraméter határozza meg a megjelenített üzenetet.

A `Button` widget – interaktív gombokat hoz létre.

Pl.
```
import tkinter as tk
foablak = tk.Tk()

gomb = tk.Button(foablak, text="Lenyomás")
gomb.pack()
foablak.mainloop()
```



13.3. Komponensek elrendezésének kezelése

<def/> A ***Tkinterben*** a grafikus komponenseket az ablakon belül kell rendszerezni.
A widgetek pozicionálásának folyamatát ***elrendezéskezelésnek*** nevezzük.

A ***Tkinter*** három fő metódust kínál: `pack()`; `grid()`; `place()`.

A `pack()` metódus – automatikusan rendszerezi a widgeteket.
Ez a legegyszerűbb elhelyezési módszer, amely az elemeket egymás alá vagy egymás mellé pozicionálja.

Pl.

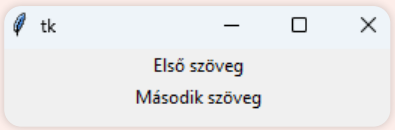
```
import tkinter as tk

foablak = tk.Tk()

label1 = tk.Label(foablak, text="Első szöveg")
label1.pack()

label2 = tk.Label(foablak, text="Második szöveg")
label2.pack()

foablak.mainloop()
```



Fontos opciók a pack() metódushoz

`side` – *Lehetővé teszi a widget pozicionálását.*

Pl.

```
gomb.pack(side="left")
```

Lehetséges poziciók: left; right; top; bottom.

`padx, pady` – *A widgetek körüli külső térközök beállítása.*

Pl.

```
gomb.pack(padx=10, pady=10)
```

A `grid()` metódus – sorokból és oszlopokból álló táblázatba rendezi a widgeteket.
Rendkívül hasznos űrlapok és rendezett interfészek kialakításához.

Pl.

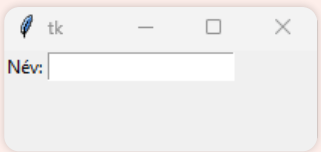
```
import tkinter as tk

foablak = tk.Tk()

szoveg_nev = tk.Label(foablak, text="Név:")
szoveg_nev.grid(row=0, column=0)

mezo_nev = tk.Entry(foablak)
mezo_nev.grid(row=0, column=1)

foablak.mainloop()
```



Magyarázat: **`row`** `a sort képviseli;` **`column`** `az oszlopot képviseli.`

Térközök beállítása a grid() metódussal

Pl.

```
szoveg.grid(row=0, column=0, padx=10, pady=10)
```

Cellák összevonása

A `columnspan` paraméter lehetővé teszi a widget kiterjesztését több oszlopra.

Pl.

```
szoveg.grid(row=0, column=0, columnspan=2)
```

A `place()` metódus – pontos koordináták használatával pozicionálja a widgeteket.

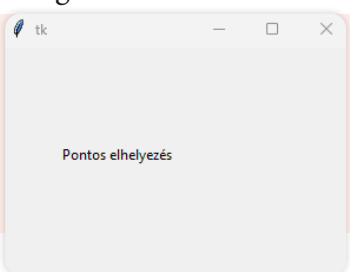
Pl.

```
import tkinter as tk

foablak = tk.Tk()
foablak.geometry("300x200")

szoveg = tk.Label(foablak, text="Pontos elhelyezés")
szoveg.place(x=50, y=80)

foablak.mainloop()
```



Magyarázat: ***`x`*** *a vízszintes pozíciót jelöli;* ***`y`*** *a függőleges pozíciót jelöli.*

Különbségek a pack(), grid() és place() között

Metódus	Fő jellemző	Felhasználás
`pack()`	Automatikus rendszerezés	Egyszerű interfészek
`grid()`	Sorok és oszlopok szerinti rendszerezés	Űrlapok és rendezett alkalmazások
`place()`	Pontos pozicionálás	Játékok és speciális interfészek

Fontos szabályok

Ugyanazon az ablakon belül nem ajánlott a `pack()` és `grid()` metódusok kombinálása. Ez hibákat vagy váratlan működést eredményezhet. Mégis a különböző metódusok használhatók különböző frame-eken belül.

A Tkinter használatának előnyei: A ***Tkinter*** számos előnyt kínál: ingyenes és a ***Python*** része; könnyen tanulható; lehetővé teszi a ***GUI*** alkalmazások gyors fejlesztését; működik Windows, Linux és macOS rendszereken; megfelelő oktatási projektekhez és egyszerű alkalmazásokhoz.

A Tkinter használatának hátrányai: Bár nagyon hasznos, a ***Tkinter***nek vannak korlátai is: a komponensek dizájnja elavultnak tűnhet; nem kínál összetett grafikai hatásokat; kevésbé használják modern alkalmazásokhoz.

Jegyezd meg!

- ✔ A ***Tkinter*** a standard ***Python*** könyvtár a grafikus interfészekhez.
- ✔ Minden alkalmazás egy főablak létrehozásával kezdődik a ***Tk()*** használatával.
- ✔ A ***widgetek*** az alkalmazás grafikus komponenseit képviselik. A ***widgetek*** pozicionálására használjuk: `pack()`; `grid()`; `place()`. A `grid()` metódus a leggyakrabban használt a rendezett interfészekhez.

Következtetések: A ***Tkinter*** a legkönnyebben használható ***Python***-könyvtár intuitív grafikus alkalmazások készítéséhez. A jól felépített felületek létrehozása közvetlenül függ a widgetek és az elrendezés kezelésének ismeretétől, mivel ezek a fogalmak képezik az alapját a komplex vizuális projektek fejlesztésének.

Feladatlap 13

Hozzatok létre egy *Feladatlap13* nevű mappát. Oldjátok meg a következő feladatokat Python nyelven, a Tkinter könyvtár használatával, majd mentsétek a fájlokat a munkamappába.

1. ***Készíts egy ablakot, betartva:*** 400×200-as méret; „Tkinter gyakorlat” cím; világoszöld háttér.
2. ***Adj hozzá két címkét és egy gombot a*** `pack()` ***metódus használatával.***
3. ***Készíts egy űrlapot a következő mezőkkel:*** vezetéknév; keresztnév; életkor. Használd a `grid()` metódust.
4. ***Helyezz három gombot különböző helyekre a*** `place()` ***metódus használatával.***

Önértékelés!

Az *onertekeles13.docx* állományba írjátok be a megfelelő válaszokat, a következő jelöléseket használva: *1 – igen, 2 – nem, 3 – nem vagyok biztos benne*, majd mentsétek el a *Feladatlap13* mappába.

	A lecke végén tudom...	***Igen***	***Nem***	***Nem biztos***
1.	elmagyarázni, mi az a Tkinter.			
2.	importálni a Tkinter könyvtárat és létrehozni egy főablakot.			
3.	személyre szabni az ablakot.			
4.	felsorolni a Tkinterben elérhető főbb widgeteket.			
5.	elrendezni a widgeteket az ablakban a pack() használatával.			
6.	táblázatba rendezni a widgeteket a grid() használatával.			

7. Nehézségeim voltak a következőkben: …

Ne felejtsétek el értékelni a viselkedéseteket és a tevékenységeteket az óra során!
(a 7. oldalon található viselkedés- és tevékenységfigyelő napló segítségével)

A lecke végén így éreztem magam:

14. Lecke – Interaktív widgetek: gombok, címkék és szövegmezők

14.1. Interaktív widgetek használata a Tkinterben

„Minden jó alkalmazás egy tiszta és könnyen használható felülettel kezdődik."

A felhasználó interakciója a grafikus alkalmazásokkal

A modern alkalmazások lehetővé teszik a felhasználó számára, hogy közvetlenül interakcióba lépjen a képernyőn megjelenő elemekkel. A grafikus felületeken ez az interakció a ***widgetek*** segítségével valósul meg.

<def/>

A ***widgetek*** olyan grafikus komponensek, amelyek lehetővé teszik:

- információk megjelenítését;
- adatok bevitelét;
- parancsok végrehajtását;
- a felhasználó-alkalmazás kommunikációt.

*A **Tkinter** könyvtárban a leggyakrabban használt interaktív **widgetek** a következők:*

- `Label` (címke);
- `Button` (gomb);
- `Entry` (szövegmező);
- `Text` (kiterjesztett szövegterület).

Ezek a komponensek alkotják minden grafikus alkalmazás alapját.

<def/>

A `Label` widget– szöveges vagy grafikus információk megjelenítésére szolgál.

A címkéket gyakran használják: címekre; utasításokra; információs üzenetekre; más komponensek leírására.

Címke létrehozása

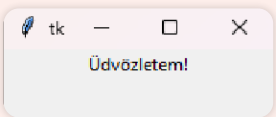
Pl.

```
import tkinter as tk

foablak = tk.Tk()

cimke = tk.Label(foablak, text="Üdvözletem!")
cimke.pack()

foablak.mainloop()
```



Magyarázat: a `Label()` létrehozza a *címkét*; a `text` paraméter határozza meg a *megjelenített üzenetet*; a `pack()` helyezi el a *widgetet* az ablakban.

Címkék testreszabása

A *címkéket* testre szabhatod a következők szerint: szín; betűtípus; méret; háttér; igazítás.

Pl.

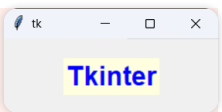
```
import tkinter as tk

foablak = tk.Tk()

cimke = tk.Label(foablak,text="Tkinter",
           font=("Arial", 20, "bold"), fg="blue", bg="lightyellow")

cimke.pack(pady=20)

foablak.mainloop()
```



Fontos paraméterek a Label widgethez

Paraméter	*Szerep*
text	A megjelenített szöveg
font	Betűtípus és méret
fg	A szöveg színe

Paraméter	*Szerep*
bg	A háttér színe
width	A widget szélessége
height	A widget magassága

<def/>

A `Button` widget – lehetővé teszi egy művelet végrehajtását, amikor a felhasználó megnyom egy gombot. A gombok alapvető fontosságú elemek az interaktív alkalmazásokban. Segítségükkel lehet: ablakokat megnyitni; üzeneteket megjeleníteni; adatokat feldolgozni; függvényeket végrehajtani.

Gomb létrehozása

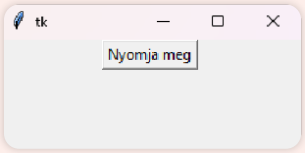
Pl.

```
import tkinter as tk

foablak = tk.Tk()

gomb = tk.Button(foablak, text="Nyomja meg")
gomb.pack()

foablak.mainloop()
```



Függvény hozzárendelése egy gombhoz

A gomb akkor válik interaktívvá, ha a `command` paraméteren keresztül egy függvényt rendelünk hozzá.

Pl.

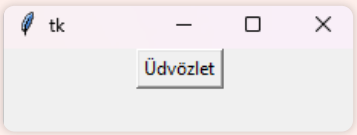
```
import tkinter as tk

foablak = tk.Tk()

def udvozlet():
    print("Szia!")

gomb = tk.Button(foablak, text="Üdvözlet", command=udvozlet)
gomb.pack()

foablak.mainloop()
```



Magyarázat: • Az `udvozlet()` függvény a gomb megnyomásakor hajtódik végre;
• A függvény nevét zárójelek nélkül kell átadni.

Címke módosítása gombbal

Pl.

```
import tkinter as tk

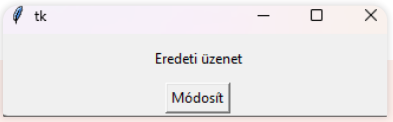
foablak = tk.Tk()

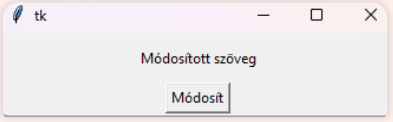
cimke = tk.Label(foablak, text= "Eredeti üzenet" )
cimke.pack(pady=10)

def modosit_szoveg():
     cimke.config(text= "Módosított szöveg" )

gomb = tk.Button(foablak, text="Módosít", command=modosit_szoveg)
gomb.pack()

foablak.mainloop()
```





Magyarázat: a `config()` metódus lehetővé teszi egy ***widget*** tulajdonságainak módosítását.

Gombok testreszabása

Pl.

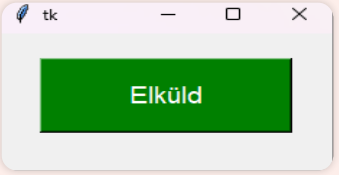
```
import tkinter as tk

foablak = tk.Tk()

gomb = tk.Button(
    foablak,
    text="Elküld",
    font=("Arial", 14),
    bg="green",
    fg="white",
    width=15,
    height=2
)

gomb.pack(pady= 20)

foablak.mainloop()
```



Fontos paraméterek a Button widgethez

Paraméter	*Szerep*
text	A gombon megjelenő szöveg
command	A végrehajtott függvény
bg	A háttér színe
fg	A szöveg színe

Paraméter	*Szerep*
font	A szöveg stílusa
width	A gomb szélessége karakterszámban mérve
height	A gomb magassága karakterszámban mérve

Az `Entry` widget – egysoros adatbevitelt tesz lehetővé.
Ezt a következőkre használják: név; jelszavak; címek; mértékek.

Szövegdoboz létrehozása

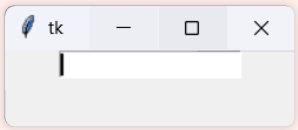
Pl.

```
import tkinter as tk

foablak = tk.Tk()

mezo = tk.Entry(foablak)
mezo.pack()

foablak.mainloop()
```



A beírt szöveg beolvasása

Egy szövegmező tartalmát a `get()` metódussal lehet beolvasni.

```
import tkinter as tk

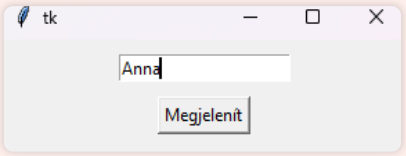
foablak = tk.Tk()

mezo = tk.Entry(foablak)
mezo.pack(pady=10)

def kiirat():
    text = mezo.get()
    print(text)

gomb = tk.Button(foablak, text="Megjelenít", command=kiirat)
gomb.pack()

foablak.mainloop()
```



Szöveg beszúrása és törlése

A ***Tkinter*** lehetővé teszi a következőket: szöveg beszúrása; tartalom törlése.

- ***Szöveg beszúrása***

Pl.

```
mezo.insert(0, "Szia")
```

Az első paraméter a pozíciót jelöli.

- ***Tartalom törlése***

Pl.

```
mezo.delete(0, tk.END)
```

Jelszómezők

A *jelszavakat* el lehet rejteni a `show` paraméter használatával.

Pl.

```
entry_kod = tk.Entry(foablak, show="*")
```

A `Text` Widget – több soros szövegbevitelt tesz lehetővé.
A következőkre használják: megjegyzések; leírások; jegyzetek; egyszerű szövegszerkesztők.

Text widget létrehozása

Pl.

```
import tkinter as tk

foablak = tk.Tk()

szoveg_zona = tk.Text(foablak, width=30, height=10)
szoveg_zona.pack()

foablak.mainloop()
```

Szöveg beolvasása a Text widgetből

Pl.

```
tartalom = szoveg_zona.get("1.0", tk.END)
```

Magyarázat: *1.0* jelenti a szöveg kezdetét; *tk.END* jelenti a szöveg végét.

Szöveg beszúrása a Text widgetbe

Pl.

```
szoveg_zona.insert(tk.END, "Ez egy szoveg.")
```

Szöveg törlése

Pl.

```
szoveg_zona.delete("1.0", tk.END)
```

Gyakorlati példa – Interaktív alkalmazás

Pl.

```
import tkinter as tk

foablak = tk.Tk()
foablak.title("Interaktív alkalmazás")
foablak.geometry("400x250")

label = tk.Label(foablak, text="Ird be a neved")
label.pack(pady=10)

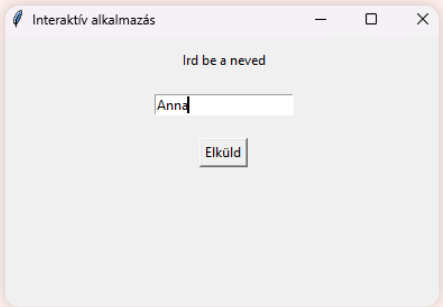
entry = tk.Entry(foablak)
entry.pack(pady=10)

def kiir_uzenet():
    nev = entry.get()
    eredmeny.config(text="Üdv., " + nev + "!")

gomb = tk.Button(foablak, text="Elküld", command=kiir_uzenet)
gomb.pack(pady=10)

eredmeny = tk.Label(foablak, text="")
eredmeny.pack(pady=10)

foablak.mainloop()
```



A get() metódus magyarázata a Text widgetnél

A `Text` **widget** tartalma a következő metódussal olvasható be:

```
zona_text.get("1.0", tk.END)
```

Magyarázat:

- **`"1.0"`** a kezdőpozíciót jelöli:
  - **`1`** = 1. sor;
  - **`0`** = 0. oszlop (a legelső karakter).
- **`tk.END`** a szöveg végét jelöli.

Az általános szintaxis:

```
widget_szoveg.get(sor.oszlop, vegso_pozicio)
```

Pl.

```
tartalom = szoveg_zona.get("1.0", tk.END)
print(tartalom)
```

A `Text` widget a következő típusú koordinátákat használja: sor; oszlop.
Így, "2.5" azt jelenti: 2. sor; 5. oszlop.

Az Entry és a Text közötti különbség

Widget	*Jellemző*
Entry	Egysoros adatbevitel
Text	Többsoros adatbevitel

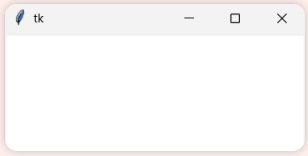
A *`Canvas`* widget – rajzolásra és grafikus ábrázolásokra szolgál.
A ***Canvas*** lehetővé teszi: geometriai alakzatok rajzolását; egyszerű játékok megvalósítását; képek meg-

jelenítését; animációk létrehozását; grafikus ábrázolásokat és szimulációkat. A ***Canvas*** egy rajzfelületté alakítja át az ablakot.

Canvas létrehozása

Pl.

```
import tkinter as tk
foablak = tk.Tk()
canvas = tk.Canvas(foablak, width=300, height=200, bg="white")
canvas.pack()
foablak.mainloop()
```



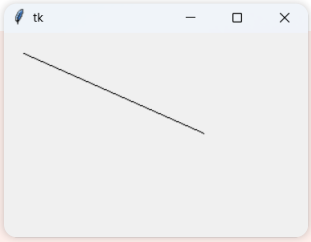
Magyarázat:

- A ***width*** határozza meg a rajzfelület szélességét;
- A ***height*** határozza meg a magasságát;
- A ***bg*** határozza meg a háttérszínt.

Vonal rajzolása

Pl.

```
import tkinter as tk
foablak = tk.Tk()
canvas = tk.Canvas(foablak, width=300, height=200)
canvas.pack()
canvas.create_line(20, 20, 200, 100)
foablak.mainloop()
```



A Canvas() létrehozza a rajzfelületet; a create_line() egy vonalat rajzol. A koordináták a vonal kezdő és végpontját jelölik.

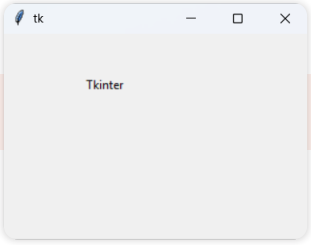
Megj.

Háttér is hozzáadható, például a `bg="white"` paraméterrel.

Szöveg hozzáadása a Canvasre

Pl.

```
canvas.create_text(100, 50, text="Tkinter")
```

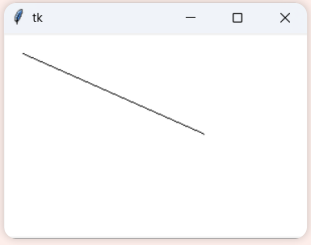


Szöveg hozzáadása a Canvasre

A **Canvas** widgetet a következőkre használják: 2D játékok; rajzalkalmazások; grafikus szimulációk; animációk; adatok vizuális ábrázolása.

Pl.

```
import tkinter as tk
foablak = tk.Tk()
canvas = tk.Canvas(foablak, width=300, height=200, bg="white")
canvas.pack()
canvas.create_line(20, 20, 200, 100)
foablak.mainloop()
```

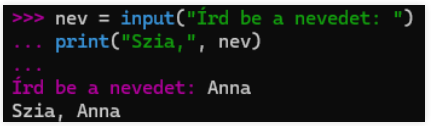


Gyakorlati tevékenység – CLI és GUI alkalmazások összehasonlítása

Ugyanez a funkcionalitás megvalósítható: parancssorban (**CLI**); grafikus felülettel (**GUI**).

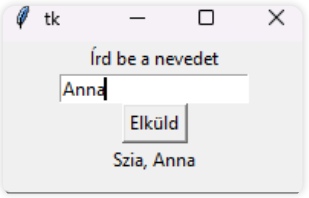
CLI változat – input() használatával

```
nev = input("Írd be a nevedet: ")
print("Szia,", nev)
```



Jellemzők: billentyűzettel történő vezérlés; futtatás konzolban; egyszerű felület.

GUI változat – Entry és Button használatával




```
import tkinter as tk

foablak = tk.Tk()

cimke = tk.Label(foablak, text="Írd be a nevedet")
cimke.pack()

entry = tk.Entry(foablak)

entry.pack()

def koszon():
    nev = entry.get()
    eredmeny.config(text="Szia, " + nev)

gomb = tk.Button(foablak, text="Elküld", command=koszon)
gomb.pack()

eredmeny = tk.Label(foablak, text="")
eredmeny.pack()

foablak.mainloop()
```

Jellemzők: vizuális felület; intuitívabb használat; interakció widgeteken keresztül.

A CLI és a GUI összehasonlítása

Jellemző	CLI	GUI
Interakció	Billentyűzet	Egér és billentyűzet
Vizuális megjelenés	Egyszerű	Grafikus és intuitív
Összetettség	Kisebb	Nagyobb
Könnyű kezelhetőség	Kisebb	Nagyobb



A **Frame** és **Canvas** widgetek kibővítik a **Tkinter** alkalmazások fejlesztési lehetőségeit:

- A **`Frame`** rendszerezi a komponenseket;
- A **`Canvas`** lehetővé teszi a rajzolást és a grafikus ábrázolásokat.

Ezek a widgetek elengedhetetlenek az összetett és jól strukturált grafikus alkalmazások elkészítéséhez.



Jegyezd meg!

✔ Widgetek: ***`Label`*** (megjelenítés), ***`Button`*** (műveletek), ***`Entry`*** (egysoros szöveg), ***`Text`*** (többsoros szöveg).

✔ Interakció: ***`command`*** (funkciók hozzárendelése a gombokhoz), ***`get()`*** (szöveg kiolvasása), ***`config()`*** (tulajdonságok módosítása).

Következtetések

Az interaktív **widgetek** (címkék, gombok, szövegdobozok) a **Tkinter** alapvető elemei, amelyek lehetővé teszik az információcserét a felhasználó és a program között, és elengedhetetlenek a modern grafikus alkalmazások fejlesztéséhez.

Feladatlap 14

Hozzátok létre a *Feladatlap14* nevű mappát. Oldjátok meg a következő feladatokat Python nyelven, a Tkinter könyvtár használatával, és mentsétek a fájlokat a munkamappába.

1. Hozz létre egy ablakot, amely a következőket tartalmazza:
- egy címke;
- egy gomb;
- egy szövegdoboz.

2. Készíts egy gombot, amely megváltoztatja egy címke szövegét.



3. Készíts egy bejelentkezési űrlapot a következőkkel:
- felhasználó;
- jelszó;
- bejelentkezés gomb.

4. Hozz létre egy alkalmazást, amely beolvassa a beírt szöveget, és megjeleníti azt egy címkében.

5. ***Szöveg beolvasása a Text widgetből***
Készíts egy alkalmazást, amely:
- tartalmaz egy `Text` widgetet;
- a `get()` használatával beolvassa a tartalmat;
- megjeleníti a szöveget egy `Label`-ben.

Önértékelés!

Az *onertekeles14.docx* állományba írjátok be a megfelelő válaszokat, a következő jelöléseket használva: *1 – igen, 2 – nem, 3 – nem vagyok biztos benne*, majd mentsétek el a *Feladatlap14* mappába.

	A lecke végén tudom...	***Igen***	***Nem***	***Nem biztos***
1.	hogyan kell címkét (Label) létrehozni és beállítani a tulajdonságait.			
2.	hogyan kell gombot (Button) létrehozni és paraméterekkel személyre szabni.			
3.	hogyan kell egy funkciót rendelni egy gombhoz a command paraméteren keresztül.			
4.	hogyan kell módosítani egy meglévő widget tulajdonságait.			
5.	hogyan kell szövegdobozt létrehozni/beilleszteni.			
6.	hogyan kell egy olyan egyszerű alkalmazást építeni, amely kombinálja a Label, Entry és Button widgeteket a felhasználói adatok átvételére és megjelenítésére.			

7. Nehézségeim voltak a következőkben: …

Ne felejtsétek el értékelni a viselkedéseteket és a tevékenységeteket az óra során!
(a 7. oldalon található viselkedés- és tevékenységfigyelő napló segítségével)

A lecke végén így éreztem magam:

15. Lecke – Eseménykezelés grafikus felületeken (üzenetek megjelenítése kattintásra)

Eseménykezelés Tkinter-ben és üzenetek megjelenítése kattintásra

„Egy interaktív program ugyanúgy válaszol a felhasználó lépéseire, mint egy ember és számítógép közötti beszélgetés."

15.1. Események a grafikus alkalmazásokban

A modern grafikus alkalmazások a felhasználó és a program közötti interakción alapulnak.

A felhasználó minden egyes lépése kivált egy eseményt.

Pl.

Példák *eseményekre*: *egy gomb megnyomása; az egér mozgatása; egy billentyű lenyomása; egy ablak bezárása; egy opció kiválasztása.*

<def/>

Az eseménykezelés az a folyamat, amellyel a program észleli ezeket a lépéseket, és válaszol rájuk.

A ***Tkinter***-ben az eseményeket a következőkkel kezeljük: függvények; a `command` paraméter; a `bind()` metódus.

15.2. Eseményvezérelt programozás

*A **Tkinter** az eseményvezérelt programozási modellt használja.*

Ebben a modellben:

- az alkalmazás a felhasználó lépéseire vár;
- egy esemény megjelenésekor lefut egy függvény;
- a hozzárendelt függvényt eseménykezelő függvénynek nevezzük.

Az általános folyamat:

1. a felhasználó végrehajt egy lépést;
2. létrejön egy esemény;
3. a program észleli az eseményt;
4. lefut a hozzárendelt függvény.

Eseménykezelésben használt függvények

A ***Pythonban*** a függvények lehetővé teszik a kód rendszerezését és az utasítások végrehajtását, amikor meghívjuk őket.

Pl.

```
def udvozlet():
    print("Üdvözlet!")
```

A függvényt így lehet meghívni: `udvozlet()`

A ***Tkinter***-ben a függvényeket az eseményekre való reagálásra használjuk.

Gombra kattintás esemény

A leggyakrabban használt esemény a gomb megnyomása. Ez a `command` paraméterrel kezelhető.

A command paraméter használata

Pl.

```
import tkinter as tk

foablak = tk.Tk()

def uzenet():
    print("Lenyomott gomb")

gomb = tk.Button(foablak, text="Klikk", command=uzenet)
gomb.pack()

foablak.mainloop()
```

Magyarázat:

- a uzenet() függvény a gomb megnyomásakor fut le;
- a függvény nevét zárójelek nélkül adjuk át;
- a command paraméter határozza meg a gomb műveletét.

Üzenetek megjelenítése egy címkében

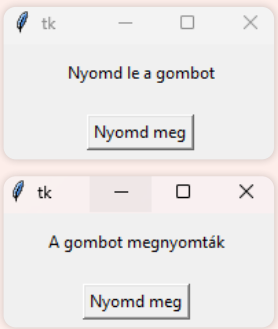
A grafikus alkalmazásokban az üzeneteket közvetlenül az ablakban is meg lehet jeleníteni a ***Label widget*** használatával.

Pl.

```
import tkinter as tk
foablak = tk.Tk()

szoveg = tk.Label(foablak, text="Nyomd le a gombot")
szoveg.pack(pady=10)

def uzenet():
     szoveg.config(text="A gombot megnyomták")
gomb = tk.Button(foablak, text="Nyomd meg",command=uzenet)
gomb.pack(pady=10)
foablak.mainloop()
```



A ***config()*** metódus módosítja egy már meglévő widget tulajdonságait.

Üzenetek megjelenítése speciális ablakokban

*A **Tkinter** a **messagebox** modult kínálja a párbeszédablakok megjelenítésére.*
Ezek az ablakok megjeleníthetnek: információkat; figyelmeztetéseket; hibákat; kérdéseket.

A messagebox modul importálása a következő módon történik:

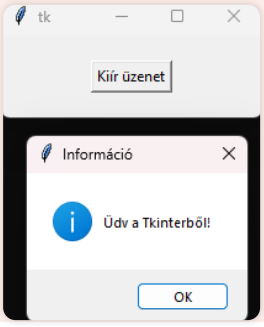
```
from tkinter import messagebox
```

Tájékoztató üzenet megjelenítése

Pl.

```
import tkinter as tk
from tkinter import messagebox
foablak = tk.Tk()

def kiir_uzenet():
     messagebox.showinfo("Információ","Üdv a Tkinterből!")
gomb = tk.Button(foablak, text="Kiír üzenet", command=kiir_
uzenet)
gomb.pack(pady=20)
foablak.mainloop()
```



Magyarázat: az első paraméter az ablak címét jelenti; a második paraméter a megjelenített üzenetet jelenti.

A messagebox ablaktípusai

Függvény	*Szerep*
showinfo()	Információkat jelenít meg
showwarning()	Figyelmeztetéseket jelenít meg
showerror()	Hibákat jelenít meg
askquestion()	Kérdéseket tesz fel
askyesno()	True vagy False értéket ad vissza

Pl.

Figyelmeztetés

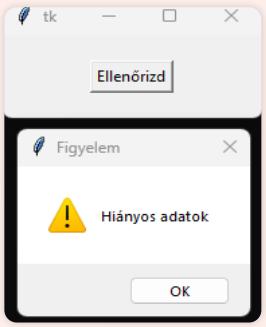
```
import tkinter as tk
from tkinter import messagebox

foablak = tk.Tk()

def figyelmeztetes():
     messagebox.showwarning("Figyelem", "Hiányos adatok")

gomb = tk.Button(foablak, text="Ellenőrizd")
gomb.config(command=figyelmeztetes)
gomb.pack(pady=20)

foablak.mainloop()
```



Pl.

Hibaüzenet

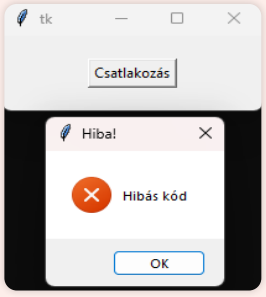
```
import tkinter as tk
from tkinter import messagebox

foablak = tk.Tk()

def hiba():
     messagebox.showerror("Hiba!","Hibás kód")

gomb = tk.Button(foablak, text="Csatlakozás", command=hiba)
gomb.pack(pady=20)

foablak.mainloop()
```



Színváltoztatás egérkattintással

Pl.

```
import tkinter as tk

foablak = tk.Tk()
foablak.geometry("300x200")

def szincsere():
     foablak.config(bg="lightblue")

gomb = tk.Button(foablak, text="Szín módosítása", command=szincsere)
gomb.pack(pady=50)

foablak.mainloop()
```

Pl.

Kattintásszámláló

```
import tkinter as tk

foablak = tk.Tk()
foablak.title("Számláló")

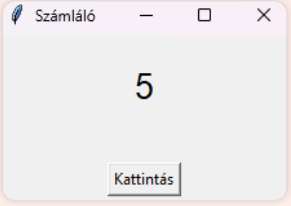
szamlalo = 0

szoveg = tk.Label(foablak, text="0", font=("Arial", 20))
szoveg.pack(pady=20)

def novekszik():
     global szamlalo
     szamlalo += 1
     szoveg.config(text=str(szamlalo))

gomb = tk.Button(foablak, text="Kattintás", command=novekszik)
gomb.pack(pady=20)

foablak.mainloop()
```



Események specifikus billentyűkhöz

A **Tkinter** lehetővé teszi a speciális billentyűk érzékelését a `bind()` metódus használatával.

Gyakran használt billentyűk:

- `<Return>` → Enter billentyű;
- `<Escape>` → Escape billentyű;
- `<Key>` → bármelyik billentyű.

```
# Az Enter, Esc és bármilyen billentyű azonosítása

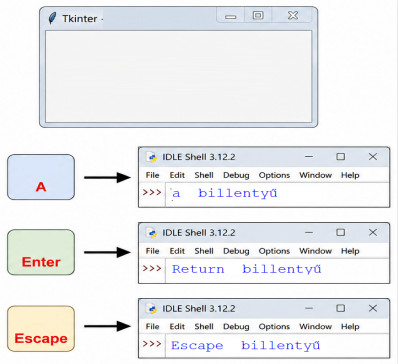
import tkinter as tk

def megjelenit(event):
    print(event.keysym," billentyű")

foablak = tk.Tk()

foablak.bind("<Key>", megjelenit)
foablak.bind("<Return>", megjelenit)
foablak.bind("<Escape>", megjelenit)

foablak.mainloop()
```



Kód rendszerezése az interaktív alkalmazásokban

Az összetettebb alkalmazásokban ajánlott: a függvények szétválasztása; egyértelmű változók használata; a kód kommentelése; a ***widgetek*** frame-ekbe való rendszerezése. Ezek a gyakorlatok javítják a program olvashatóságát és karbantarthatóságát.

Gyakori hibák

Zárójelek használata a command-nál

Hibás:

```
command=fuggveny()
```

Helyes:

```
command=fuggveny
```

A zárójelek azonnal végrehajtják a függvényt.

Az event paraméter hiánya a bind()-ben

Hibás:

```
def fuggveny():
    pass
```

Helyes:

```
def fuggveny(event):
    pass
```



Jegyezd meg!

- ✔ Események: Az alkalmazást irányító felhasználó műveletei (eseményvezérelt programozás alkalmazásával).
- ✔ Felület: A command paraméter (gombokhoz) és a messagebox modul (párbeszédablakokhoz).

Következtetések: Az eseménykezelés dinamikussá teszi a Tkinter-alkalmazásokat, összekapcsolva a felhasználói műveleteket meghatározott funkciókkal. A `command` és a `messagebox` elemek ismerete elengedhetetlen az interaktív grafikus felületek fejlesztéséhez.



Feladatlap 15

Hozzátok létre a *Feladatlap15* nevű mappát. Oldjátok meg a következő feladatokat Python nyelven, a Tkinter könyvtár használatával, és mentsétek el a fájlokat a munkamappába.

1. Hozz létre egy gombot, amely egy címkében egy üzenetet jelenít meg.
2. Készíts egy alkalmazást, amely egy gomb megnyomására megváltoztatja az ablak színét.
3. Készíts egy űrlapot, amely ha a felhasználó nem ad meg adatot, hibaüzenetet jelenít meg.
4. Elemezd a következő kódot, azonosítsd a hibát majd javítsd ki az utasítást.

```
gomb = tk.Button(foablak, text="Kattintas", command=feladatkor())
```

5. ***Mini-projekt***

Készítsen egy alkalmazást, amely a következőket tartalmazza: Entry widget; az adatok érvényesítése üzenetablakkal; az eredmény megjelenítése egy Label-ben; az adatbevitel megerősítése az Enter (<Return>) billentyűvel. Az alkalmazás:

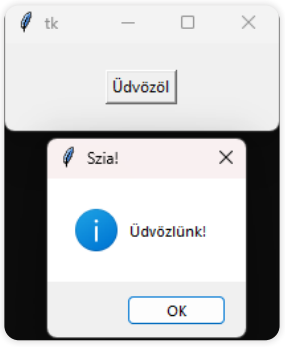
- figyelmeztetést jelenítsen meg, ha a felhasználó nem ad meg adatokat;
- üdvözlő üzenetet jelenítsen meg, ha az adatok helyesek.

Megoldott feladatok

Feladat 1 – Üdvözlő üzenet megjelenítése

Készíts egy alkalmazást, amely egy gomb megnyomására egy üdvözlő üzenetet jelenít meg.

Megoldás




```
import tkinter as tk
from tkinter import messagebox

foablak = tk.Tk()

def uzenet():
    messagebox.showinfo("Szia!", "Üdvözlünk!")

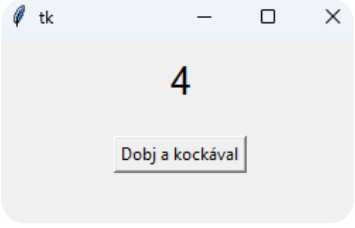
gomb = tk.Button(foablak, text="Üdvözöl", command=uzenet)
gomb.pack(pady=20)

foablak.mainloop()
```

Feladat 2 – Virtuális dobókocka

Készítsünk egy alkalmazást, amely egy gomb megnyomására 1 és 6 közötti véletlenszerű számot jelenít meg.

Megoldás




```
import tkinter as tk
import random

foablak = tk.Tk()

cimke = tk.Label(foablak, text="-", font=("Arial", 20))
cimke.pack(pady=10)

def kocka_dobas():
    cimke.config(text=str(random.randint(1, 6)))

gomb = tk.Button(foablak, text="Dobj a kockával", command=kocka_dobas)
gomb.pack(pady=10)

foablak.mainloop()
```



Önértékelés!

Töltsétek ki a válaszaitokkal egy *onertekeles15.docx* nevű fájlba, a következő jelöléseket használva: *1–igen, 2–nem, 3–nem vagyok biztos benne*. Mentsétek a fájlt a *Feladatlap15* nevű mappába.

	A lecke végén tudom...	***Igen***	***Nem***	***Nem biztos***
1.	mi az az esemény (event) egy grafikus alkalmazásban.			
2.	hogyan kell egy függvényt egy gombhoz rendelni a command paraméteren keresztül.			
3.	messagebox modul importálását és egy párbeszédablak megjelenítését.			
4.	egy teljes alkalmazás felépítését gombbal, kattintásra frissülő címkével és messagebox ablakkal.			

5. Nehézségeim voltak a következőkben: …

Ne felejtsétek el értékelni a viselkedéseteket és a tevékenységeteket az óra során!
(a 7. oldalon található viselkedés- és tevékenységfigyelő napló segítségével)

A lecke végén így éreztem magam:

16. lecke – Szöveges fájlok (állományok): megnyitás, bezárás és adatátvitel

Szöveges fájlok kezelése Pythonban

„Az adatok akkor válnak értékes információvá, ha hatékonyan tárolhatók, hozzáférhetők és feldolgozhatók."

A fájlok és szerepük az informatikában

Az informatikában a fájlokat az információk tartós tárolására használjuk.

Ellentétben a változókkal, amelyek csak a program futásának ideje alatt őrzik meg az adatokat, a fájlok lehetővé teszik az információk elmentését a későbbi használatra.

A fájlokat a következőkre használjuk: dokumentumok mentése; felhasználói adatok tárolása; eredmények megjegyzése; programok közötti információcsere.

Fájltípusok

A **fájlokat** két fő kategóriába sorolhatjuk: szöveges fájlok; bináris fájlok.

Szöveges fájlok

A ***szöveges fájlok*** karakterek formájában tárolják az információkat**.**

Pl. .txt; .csv; .py; .html.

Ezek a fájlok szövegszerkesztőkkel megnyithatók és olvashatók.

Bináris fájlok - olyan fájlok, amelyekben az információk kódolt formában, bináris adatokként vannak tárolva, és általában nem érthetőek közvetlenül az emberek számára.

Pl. .jpg, .png, .gif (képek), .mp3, .wav (audiofájlok), .mp4, .avi (videók), .exe, .com, .bat (futtatható fájlok) stb.

Fájlkezelés Pythonban

A **Pythonban** a fájlkezelés a következő lépésekből áll: a fájl megnyitása; adatok olvasása vagy írása; a fájl bezárása. A folyamat azért fontos, mert a számítógép erőforrásait megfelelően kell kezelni.

Fájl megnyitása

A fájlokat az *open()* függvénnyel nyitjuk meg.

Általános szintaxis: `fajl = open("utvonal\fajl_neve.txt", "mod")`

Paraméterek: `fajl_neve` – a fájl neve; `utvonal` – lehet relatív vagy abszolút;
`mod` – a használat módja.

Fájlmegnyitási módok

Mód	*Jelentés*
`r`	Olvasás (read)
`w`	Írás (write)
`a`	Hozzáadás a végéhez (append)

Mód	*Jelentés*
`x`	Új fájl létrehozása (create)
`rb`	Bináris olvasás (read binary)
`wb`	Bináris írás (write binary)

Az `r` mód – olvasás, lehetővé teszi egy fájl tartalmának olvasását.

Pl.
```
fajl = open("adatok.txt", "r") # Relatív elérési út
```

Ha a fájl nem létezik, a program hibát fog jelezni.

Adatok olvasása a fájlból

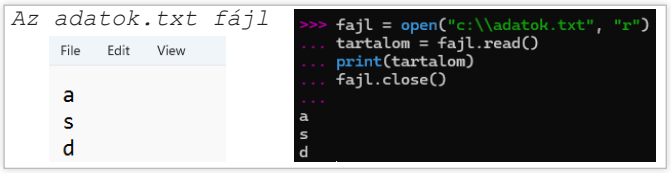
A Python többféle olvasási módszert kínál.

A `read()` metódus – a fájl teljes tartalmát beolvassa.

Pl.

```
fajl = open("c:\\adatok.txt", "r")
tartalom = fajl.read()
print(tartalom)
fajl.close()
```

Az adatok.txt fájl



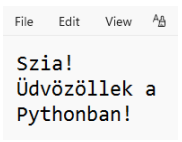

```
>>> fajl = open("c:\\adatok.txt", "r")
... tartalom = fajl.read()
... print(tartalom)
... fajl.close()
...
a
s
d
```

Adott számú karakter beolvasása

Pl.

```
fajl = open("c:\\adatok1.txt", "r")
szoveg = fajl.read(5)
print(szoveg)
fajl.close()
```

Az adatok1.txt fájl




```
>>> fajl = open("c:\\adatok1.txt", "r")
... szoveg = fajl.read(5)
... print(szoveg)
... fajl.close()
...
Szia!
```

A program beolvassa az első 5 karaktert.

A `readline()` metódus – egyetlen sort olvas be.

Pl.

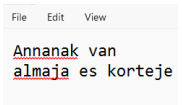
```
fajl = open("c:\\adatok1.txt", "r")
sor = fajl.readline()
print(sor)
fajl.close()
```

A `readlines()` metódus – beolvassa az összes sort, és egy listában tárolja.

Pl.

```
fajl = open("c:\\adatok1.txt", "r")
sorok = fajl.readlines()
print(sorok)
fajl.close()
```

Az adatok1.txt fájl




```
>>> fajl = open("c:\\adatok1.txt", "r")
... sorok = fajl.readlines()
... print(sorok)
... fajl.close()
...
```

Egy fájl sorainak bejárása

A fájlok bejárhatók a `for` utasítással.

Pl.

```
fajl = open("c:\\adatok1.txt", "r")
for sor in fajl:
    print(sor)
fajl.close()
```

```
>>> fajl = open("c:\\adatok1.txt", "r")
...
... for sor in fajl:
...     print(sor)
...
... fajl.close()
...
Annanak van

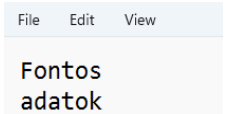
almaja es korteje
```

Megjegyzés: Amikor a read() metódus üres karakterláncot "" ad vissza, vagy a readline() ""-t ad vissza, az azt jelenti, hogy elértük a fájl végét (EOF = End Of File).

Pl.

```
fajl = open("c:\\adatok.txt", "r")
sor = fajl.readline()
while sor != "":
    print(sor)
    sor = fajl.readline()
fajl.close()
```

Az adatok.txt fájl




```
>>> fajl = open("c:\\adatok.txt", "r")
...
... sor = fajl.readline()
...
... while sor != "":
...     print(sor)
...     sor = fajl.readline()
...
... fajl.close()
...
Fontos

adatok
```

Fájlok bezárása

Használat után a fájlokat be kell zárni. Ez a `close()` metódussal történik .

Pl.

```
fajl.close()
```

A fájlok bezárása: memóriát szabadít fel; menti a módosításokat; elkerüli a hibákat.

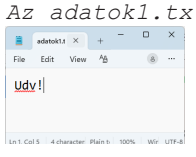
Írás a fájlokba

Az íráshoz a `w` módot használjuk.

Pl.

```
fajl = open("e:\\adatok1.txt", "w")
fajl.write("Udv!")
fajl.close()
```

Az adatok1.txt fájl




```
>>> fajl = open("e:\\adatok1.txt", "w")
... fajl.write("Udv!")
... fajl.close()
```

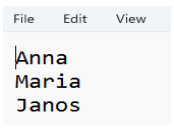
⚠ **Figyelem! A `w` mód törli a meglévő tartalmat az írás előtt.**

Írás több sorba

Pl.

```
fajl = open("e:\\adatok1.txt", "w")
fajl.write("Anna\n")
fajl.write("Maria\n")
fajl.write("Janos")
fajl.close()
```

```
>>> fajl = open("e:\\adatok1.txt", "w")
...
... fajl.write("Anna\n")
... fajl.write("Maria\n")
... fajl.write("Janos")
...
... fajl.close()
```



A `\n` karakter új sorba lépést eredményez.

Az `a` mód – hozzáadás, lehetővé teszi az adatok hozzáadását a fájl végéhez a meglévő tartalom törlése nélkül.

Pl.

```
fajl = open("e:\\adatok1.txt", "a")
fajl.write("\nUj szoveg")
fajl.close()
```

Az adatok1.txt fájl



Az `x` – mód – új fájl létrehozása, új fájlt hoz létre.

Pl.

```
fajl = open("e:\\uj.txt", "x")
fajl.close()
```

Ha a fájl már létezik, hiba lép fel.

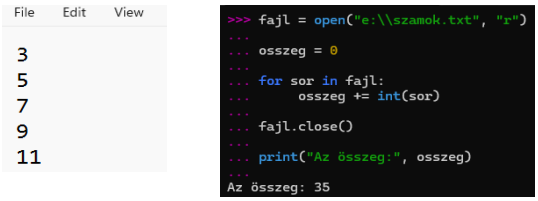
Adatátvitel a fájlok és a program között

Az adatátvitel jelentése: információbeolvasás fájlokból; azok feldolgozása; az eredmények elmentése.

Pl.

```
fajl = open("e:\\szamok.txt", "r")
osszeg = 0
for sor in fajl:
    osszeg += int(sor)
fajl.close()
print("Az összeg:", osszeg)
```

A szamok.txt fájl

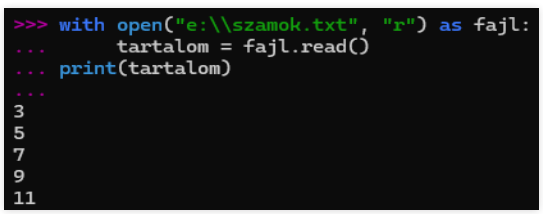


A with utasítás használata

A *Python* megtalálható a `with`, utasítást, amely automatikusan bezárja a fájlt.

Pl.

```
with open("e:\\szamok.txt", "r") as fajl:
    tartalom = fajl.read()
print(tartalom)
```



Előnyei: rövidebb kód; automatikus bezárás; biztonságos hibakezelés.

Egyidejű olvasás és írás

Pl.

```
with open("e:\\input.txt", "r") as forras:
    szoveg = forras.read()

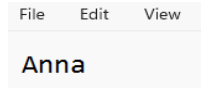
with open("e:\\output.txt", "w") as cel:
    cel.write(szoveg)
```

A program átmásolja az egyik fájl tartalmát a másikba.

Az input.txt fájl



Az output.txt fájl



Hibakezelés

A fájlokkal való munka során hibák léphetnek fel: nem létező fájl; jogosultságok hiánya; hibás név. Ezek a helyzetek a `try` és `except` utasításokkal kezelhetők.

A fájlok használatának előnyei

A fájlok lehetővé teszik: az adatok tartós tárolását; az információk átvitelét; az adatok rendszerezését; összetett alkalmazások megvalósítását.

Gyakori hibák

A fájl bezárásának elfelejtése

Hibás:

```
fajl = open("c:\\adatok.txt","r")
print(fajl.read())
```

Helyes:

```
fajl = open("adatok.txt","r")
print(fajl.read())
fajl.close()
```

A megnyitási mód helytelen használata

Pl.

```
open("adatok.txt","w")
```

Ez a mód törli a fájl meglévő tartalmát.



Jegyezd meg!

- ✔ A fájlokat az `open()` függvénnyel nyitjuk meg, a `close()` függvénnyel pedig bezárjuk. A `with` utasítás lehetővé teszi a fájlok bezárásának automatikus kezelését, hozzájárulva ezzel a biztonságosabb és hatékonyabb kód írásához.
- ✔ Olvasás/Írás: `read()` / `readline()` / `readlines()` olvassák az adatokat, a `write()` elmenti.
- ✔ Biztonság: A hiba kezelés megakadályozza a program leállását, ha problémák adódnak a fájlokkal.

Következtetések: A szöveges fájlok elengedhetetlenek az adatok tartós tárolásához és hatékony feldolgozásához. A program és a fájlok közötti adatátvitel (megnyitás, olvasás, írás, bezárás) megértése alapját képezi a nagy mennyiségű információ kezelésére képes szoftveralkalmazások fejlesztésének.



Feladatlap 16

Hozzátok létre a *Feladatlap16* nevű mappát. Oldjátok meg a következő feladatokat Python nyelven, a Tkinter könyvtár használatával, és mentsétek el a fájlokat a munkamappába.

Javasolt feladatok

1. Hozzatok létre egy szöveges fájlt, és írjatok bele három nevet.
2. Olvassátok be egy fájl tartalmát, és jelenítsétek meg a képernyőn.
3. Készítsetek egy programot, amely egy új mondatot ad hozzá egy fájl végéhez.
4. Számítsátok ki egy szöveges fájlban tárolt számok összegét.

Megoldott feladatok

1. feladat – Adatok beolvasása és megjelenítése

Követelmény: Olvasd be és jelenítsd meg a ***mediak.txt*** fájl tartalmát.

Megoldás:

```
with open("d:\\mediak.txt", "r") as fajl:
     tartalom = fajl.read()
print(tartalom)
```

2. feladat – Egy fájl tartalmának másolása

Követelmény: Másold át az ***adatok.txt*** fájl tartalmát a ***masolat.txt*** fájlba.

Megoldás:

```
with open("d:\\adatok.txt", "r") as forras:
     tartalom = forras.read()
with open("d:\\masolat.txt", "w") as celfajl:
     celfajl.write(tartalom)
```

3. feladat – Hibakezelés

Követelmény: Írj egy programot, amely megpróbál megnyitni egy ***adatok5.txt*** nevű fájlt. Ha a fájl nem létezik, a program a következő üzenetet jelenítse meg: `A fájl nem létezik!`

Megoldás:

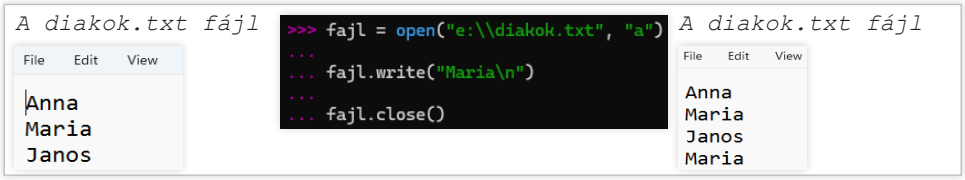
```
try:
     fajl = open("e:\\adatok5.txt", "r")
     print(fajl.read())
     fajl.close()
except FileNotFoundError:
     print("A fájl nem létezik!")
```

4. feladat – Adatok hozzáadása a fájlhoz

Követelmény: Adj hozzá egy új nevet egy fájlhoz a meglévő tartalom törlése nélkül.

Megoldás:

```
fajl = open("e:\\diakok.txt", "a")
fajl.write("Maria\n")
fajl.close()
```



5. feladat – A fájl közvetlen bejárása

Követelmény: Jelenítsd meg egy fájl összes sorát a *for* utasítás használatával.

Megoldás:

```
fajl = open("c:\\adatok.txt", "r")
for sor in fajl:
    print(sor)
fajl.close()
```



Megjegyzés - Fájl vs input() összehasonlítás

Az adatok fájlból történő beolvasása gyorsabb és praktikusabb, mint a billentyűzetről történő ismételt adatbevitel (`input()`), különösen akkor, ha nagy adatmennyiséggel dolgozunk.

Pl.

Adjatok meg 20 számot: • egyszer az `input()` használatával • egyszer pedig a fájlból beolvasva. Figyeljétek meg, melyik módszer gyorsabb és kényelmesebb.

Önértékelés!

Töltsétek ki a válaszaitokkal egy *onertekeles16.docx* nevű fájlba a következő jelöléseket használva: *1–igen, 2–nem, 3–nem vagyok biztos benne*. Mentsétek a fájlt a *Feladatlap16* nevű mappába.

	A lecke végén tudom...	***Igen***	***Nem***	***Nem biztos***
1.	elmagyarázni a különbséget a szöveges fájlok és a bináris fájlok között.			
2.	megnyitni egy fájlt Pythonban az open() függvénnyel, és kiválasztani a megfelelő módot (r, w, a, x) a kívánt művelet függvényében.			
3.	soronként bejárni egy fájlt.			
4.	adatokat írni egy fájlba.			
5.	adatokat hozzáadni egy fájl végéhez a meglévő tartalom törlése nélkül az 'a' mód használatával.			
6.	helyesen bezárni egy fájlt.			

7. Nehézségeim voltak a következőkben: …

Ne felejtsétek el értékelni a viselkedéseteket és a tevékenységeteket az óra során!
(a 7. oldalon található viselkedés- és tevékenységfigyelő napló segítségével)

A lecke végén így érzem magam:

17. Lecke – A TextIOWrapper osztály: a hatékony olvasás és írás módszerei

A TextIOWrapper osztály használata Pythonban

„Egy program teljesítménye nemcsak az algoritmusoktól függ, hanem attól is, hogy mennyire hatékonyan kezeljük az adatokat.”

Adatfolyamok és szöveges fájlok

A programozásban az adatok a következők között áramlanak: memória; fájlok; felhasználó; külső eszközök.

Ez az információátvitel adatfolyamokon keresztül valósul meg. Az adatfolyam egy mechanizmus, amelyen keresztül az információk beolvasása/kiírása szekvenciálisan történik.

Pythonban a szöveges fájlokkal való munkát az `open()` függvény által létrehozott speciális objektumok végzik. Ezek az objektumok a `TextIOWrapper` osztályhoz tartoznak.

A `TextIOWrapper` osztály – az `io` modul része, és a következőkre használatos: szöveges fájlok olvasása; szöveges fájlok írása; karakterfolyamok kezelése.

Amikor egy szöveges fájlt megnyitunk az `open()` függvénnyel, a Python automatikusan egy `TextIOWrapper` típusú objektumot hoz létre.

Pl.

```
fajl = open("adatok.txt", "r")
print(type(fajl))
```

```
<class '_io.TextIOWrapper'>
```

Eredmény: `<class '_io.TextIOWrapper'>`

A TextIOWrapper osztály jellemzői

A TextIOWrapper osztály: Unicode karakterekkel dolgozik; automatikusan kezeli a szöveg kódolását; hatékony olvasási és írási metódusokat biztosít; lehetővé teszi a nagy fájlokkal való munkát.

Karakterkódolás

A szöveges fájlokat különböző kódolásokkal mentik el. A leggyakrabban használtak: UTF-8; ASCII; UTF-16.

A **Python** gyakran az UTF-8 kódolást használja.

Pl.

```
fajl = open("adatok.txt", "r", encoding="utf-8")
```

Az `encoding` paraméter határozza meg a karakterek értelmezésének módját.

Fájl megnyitása

Az `open()` függvény egy `TextIOWrapper` objektumot ad vissza.

Általános szintaxis:

```
fajl = open(fajl_neve, mod, encoding="utf-8")
```

Pl.

```
fajl = open("adatok.txt", "r", encoding="utf-8")
```

A read() metódus

Nagyon nagy fájlok esetén a `read()` metódus használata jelentős memóriaterhelést jelenthet. Hatékonyabb módszer a soronkénti olvasás, amely nem tölti be a teljes fájlt a memóriába.

```
>>> with open("e:\\adatok1.txt", "r", encoding="utf-8") as fajl:
...     tartalom = fajl.read()
... print(tartalom)
...
Elektronikus
vallalkozas
```

Pl.

```
with open("e:\\adatok1.txt", "r") as fajl:
    for sor in fajl:
        print(sor)
```

```
>>> with open("e:\\adatok1.txt", "r") as fajl:
...     for sor in fajl:
...         print(sor)
...
Elektronikus

vallalkozas
```

Előnyök: alacsony memóriafogyasztás; alkalmas nagy fájlokhoz.

A fájlokból beolvasott adatok előkészítése

A *\n* karakter eltávolítása a strip() segítségével

Pl.

```
with open("e:\\adatok.txt", "r") as f:
    for sor in f:
        print(sor.strip())
```

```
>>> with open("e:\\adatok.txt", "r") as f:
...     for sor in f:
...         print(sor.strip())
...
Fontos
adatok
```

Magyarázat: a ***strip()*** metódus eltávolítja a szóközöket és a **\n** speciális karaktert a sor végéről.

Karakterláncok konvertálása számokká

Pl.

```
with open("e:\\jegyek.txt", "r") as fajl:
    for sor in fajl:
        jegy = float(sor.strip())
        print(jegy)
```

```
>>> with open("e:\\jegyek.txt", "r") as fajl:
...     for sor in fajl:
...         jegy = float(sor.strip())
...         print(jegy)
...
10.0
9.0
8.0
6.0
4.0
7.0
```

Magyarázat: a fájlból beolvasott adatok karakterláncok. Számításokhoz konvertálni kell őket az **int()** és **float()** függvényekkel.

Értékek szétválasztása a split() segítségével - nagyon fontos az egyszerűsített CSV típusú fájlok esetében.

Pl.

```
sor = "Anna,10"
adatok = sor.split(",")
print(adatok)
```

Eredmény:

```
['Anna', '10']
```

```
>>> sor = "Anna,10"
...
... adatok = sor.split(",")
...
... print(adatok)
...
['Anna', '10']
```

Pl.

```
with open("e:\\diakok9.txt", "r") as fajl:
    for sor in fajl:
        nev, jegy= sor.strip().split(",")
        jegy = int(jegy)
        print(nev, jegy)
```

```
>>> with open("e:\\diakok9.txt", "r") as fajl:
...     for sor in fajl:
...         nev, jegy= sor.strip().split(",")
...
...         jegy = int(jegy)
...
...         print(nev, jegy)
...
Janos 8
Anna 10
Tamas 9
```

A *write()* metódus – lehetővé teszi szöveg beírását egy fájlba.

Pl.

```
with open("e:\\adatok2.txt","w") as fajl:
    fajl.write("Udvozlunk Pythonbol")
```

```
>>> with open("e:\\adatok2.txt","w") as fajl:
...     fajl.write("Udvozlunk Pythonbol")
```

Az adatok2.txt fájl

File Edit View

Udvozlunk Pythonbol

```
>>> with open("e:\\diakok.txt","w") as fajl:
...     fajl.write("Anna")
...     fajl.write("Maria")
...     fajl.write("Janos")
```

A diakok.txt fájl

File Edit View

Anna
Maria
Janos

```
>>> sorok = ["Anna\n", "Maria\n", "Janos\n"]
... with open("e:\\diakok1.txt","w") as fajl:
...     fajl.writelines(sorok)
```

A diakok1.txt fájl

File Edit View

Anna
Maria
Janos

Több sor beírása

Pl.

```
with open("e:\\diakok.txt","w") as fajl:
    fajl.write("Ana\n")
    fajl.write("Maria\n")
    fajl.write("Ioan")
```

A *writelines()* metódus – egyszerre több sornyi információt ír a fájlba.

Pl.

```
sorok = ["Anna\n", "Maria\n", "Janos\n"]
with open("e:\\diakok1.txt","w") as fajl:
    fajl.writelines(sorok)
```

Különbség a *write()* és *writelines()* között

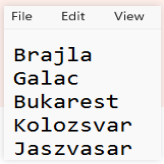
Metódus	*write()*	*writelines()*
Tulajdonság	Egy sort ír a fájlba	Több sort ír a fájlba

Pl.

Mentse el az 5 városból álló listát egy fájlba a ***writelines()*** metódussal.

```
varosok = ["Brajla\n","Galac\n", "Bukarest\n",
     "Kolozsvar\n", "Jaszvasar\n" ]
with open("d:\\varosok.txt", "w") as fajl:
    fajl.writelines(varosok)
```

A varosok.txt fájl



Megj.

`writelines()` **NEM** ad hozzá automatikusan '\n'-t.

A kurzor helyzete a fájlban

A ***TextIOWrapper*** osztály belső kurzort használ.

A kurzor jelzi az aktuális olvasási vagy írási pozíciót.

A fájlok bezárása

A fájlokat használat után be kell zárni.

Pl.

```
fajl.close()
```

A *close()* metódus: felszabadítja az erőforrásokat; elmenti a módosításokat; bezárja az adatfolyamot.

A *with* utasítás használata – ez a leginkább ajánlott munkamódszer.

Pl.

```
with open("e:\\adatok.txt","r") as fajl:
      tartalom = fajl.read()
```

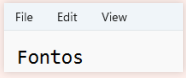
Előnyök: automatikus bezárás; biztonságosabb kód; kevesebb hibalehetőség.

Hatékony adatátvitel a fájlok között

Pl.

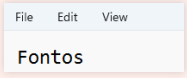
```
with open("e:\\adatok.txt","r") as forras:
     tartalom = forras.read()
with open("e:\\celfajl.txt", "w") as celfajl:
     celfajl.write(tartalom)
```

Az adatok.txt fájl




```
>>> with open("e:\\adatok.txt","r") as forras:
...     tartalom = forras.read()
... with open("e:\\celfajl.txt", "w") as celfajl:
...     celfajl.write(tartalom)
```

A celfajl.txt fájl



Hatékony soronkénti másolás

Nagy fájlok esetén a soronkénti másolás ajánlott.

Pl.

```
with open("e:\\adatok.txt","r") as forras:
     with open("e:\\celfajl.txt", "w") as celfajl:
          for sor in forras:
               celfajl.write(sor)
```

Ennek a módszernek kisebb a memóriaigénye.

Hibakezelés

```
try:
     with open("e:\\adatok3.txt", "r") as fajl:
          print(fajl.read())
except FileNotFoundError:
     print("A fájl nem létezik")
```

```
>>> try:
...     with open("e:\\adatok3.txt", "r") as fajl:
...         print(fajl.read())
... except FileNotFoundError:
...     print("A fájl nem létezik")
...
A fájl nem létezik
```

Tulajdonságok:

Tulajdonság	*Szerepe*
name	Fájlnév
mode	Megnyitási mód
closed	Ellenőrizze, hogy a fájl be van-e zárva
encoding	Használt kódolás

A TextIOWrapper osztály előnyei

A **TextIOWrapper** osztály a következőket kínálja: hatékony szövegfájl-kezelés; Unicode-támogatás; automatikus pufferelés; gyors olvasási és írási módszerek; egyszerű integrálás a programokba.

A raport.txt fájl



Gyakori hibák

A fájl bezárásának elmulasztása

Helytelen:

```
fajl = open("adatok.txt", "r")
print(fajl.read())
```

Helyes:

```
with open("adatok.txt", "r") as fajl:
    print(fajl.read())
```

Nagyon nagy fájlok teljes beolvasása

Helytelen:

```
tartalom = fajl.read()
```



Jó tudni!

- ✔ Fájltípus: az **open()** egy **TextIOWrapper** objektumot ad vissza, amely a szöveges fájlokat pufferéléssel kezeli (az adatátvitel optimalizálása érdekében).
- ✔ Olvasás/Írás: **read()**/**readline()**/**readlines()** az olvasáshoz; **write()**/**writelines()** az íráshoz.
- ✔ Vezérlés: a **tell()** azonosítja és a **seek()** módosítja a kurzor pozícióját a fájlban.
- ✔ Biztonság: az automatikus és biztonságos bezáráshoz a **with** utasítás használata ajánlott.

Összegzés

A **TextIOWrapper** osztály a Python nyelv fő eszköze a szöveges fájlok hatékony és biztonságos kezeléséhez. Az olvasási, írási és adatfolyam-vezérlési módszereinek megértése elengedhetetlen a memória és a fájlok közötti adatátvitel optimalizálásához, különösen nagy adatmennyiségek kezelése esetén.



Feladatlap 17

Hozzátok létre a *Feladatlap17* nevű mappát. Oldjátok meg a következő feladatokat Python nyelven, a Tkinter könyvtár használatával, és mentsétek el a fájlokat a munkamappába.

Javasolt feladatok

1. Hozz létre egy szöveges fájlt, és írj bele öt várost.
2. Olvass be egy fájlt sorról sorra, és jelenítsd meg a tartalmát.
3. Számold meg a karaktereket egy szöveges fájlban.

Megoldott feladatok

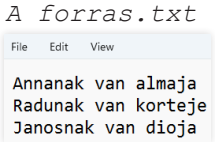
1. feladat – Fájl hatékony másolása

Követelmény: Másold át egy szöveges fájl tartalmát soronkénti olvasással.

Megoldás:

```
with open("e:\\forras.txt", "r") as forras:
    with open("e:\\cel.txt", "w") as masolat:
        for sor in forras:
            masolat.write(sor)
```

A forras.txt fájl




```
>>> with open("e:\\forras.txt", "r") as forras:
...     with open("e:\\cel.txt", "w") as masolat
...         for sor in forras:
...             masolat.write(sor)
```

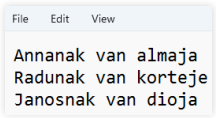
2. feladat – Karakterek megszámlálása

Követelmény: Határozd meg a karakterek teljes számát egy fájlban.

Megoldás:

```
with open("e:\\masolat.txt", "r") as f:
     tart = f.read()
print("Karakterek száma:", len(tart))
```

A masolat.txt fájl




```
>>> with open("e:\\masolat.txt", "r") as f:
...     tart = f.read()
... print("Karakterek száma:", len(tart))
...
Karakterek száma: 60
```

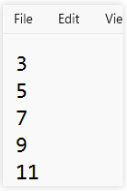
3. feladat – int() konverzió

Követelmény: A `szamok.txt` fájl soronként egy számot tartalmaz. Számítsd ki a számok összegét.

Megoldás:

```
osszeg = 0
with open("e:\\szamok.txt", "r") as f:
     for sor in f:
         osszeg += int(sor.strip())
print(osszeg)
```

A szamok.txt fájl




```
>>> osszeg = 0
... with open("e:\\szamok.txt", "r") as f:
...     for sor in f:
...         osszeg += int(sor.strip())
... print(osszeg)
...
35
```

4. feladat – A split() használata

Követelmény: A `diakok9.txt` fájl a következő adatokat tartalmazza:

```
Janos,8
Anna,10
Tamas,9
```

Jelenítsd meg külön sorokban minden egyes diák nevét és osztályzatát.

Megoldás:

```
with open("e:\\diakok9.txt", "r") as f:
     for sor in f:
         nev, jegy = sor.strip().split(",")
         print(nev)
         print(jegy)
```

```
>>> with open("e:\\diakok9.txt", "r") as f:
...     for sor in f:
...         nev, jegy = sor.strip().split(",")
...         print(nev)
...         print(jegy)
...
Janos
 8
Anna
 10
Tamas
 9
```

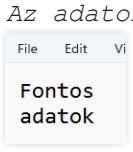
5. feladat – Üres fájl ellenőrzése

Követelmény: Ellenőrizd, hogy az `adat.txt` fájl üres-e.

Megoldás:

```
with open("e:\\adatok.txt", "r") as f:
     tartalom = f.read()
     if tartalom == "":
        print("Üres fájl")
     else:
        print("A fájl adatokat tartalmaz")
```

Az adatok.txt fájl




```
>>> with open("e:\\adatok.txt", "r") as f:
...     tartalom = f.read()
...     if tartalom == "":
...         print("Üres fájl")
...     else:
...         print("A fájl adatokat tartalmaz")
...
A fájl adatokat tartalmaz
```

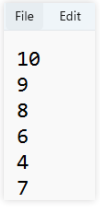
6. feladat – Osztályzatok átlagának kiszámítása

Követelmény: A `jegyek.txt` fájl soronként egy-egy osztályzatot tartalmaz. Számítsd ki az osztályzatok átlagát.

Megoldás:

```
osszeg = 0
db = 0
with open("e:\\jegyek.txt", "r") as f:
     for sor in f:
         osszeg += float(sor.strip())
         db += 1
print(osszeg / db)
```

A jegyek.txt fájl




```
>>> osszeg = 0
... db = 0
... with open("e:\\jegyek.txt", "r") as f:
...     for sor in f:
...         osszeg += float(sor.strip())
...         db += 1
... print(osszeg / db)
...
7.333333333333333
```

7. feladat – Statisztika számokról

Követelmény: Határozd meg egy fájlban található számok összegét, minimumát és maximumát.

Megoldás:

```
szamok = []
with open("e:\\szamok.txt", "r") as f:
    for sor in f:
        szamok.append(int(sor.strip()))
print(sum(szamok))
print(min(szamok))
print(max(szamok))
```

A szamok.txt fájl




```
>>> szamok = []
... with open("e:\\szamok.txt", "r") as f:
...     for sor in f:
...         szamok.append(int(sor.strip()))
... print(sum(szamok))
... print(min(szamok))
... print(max(szamok))
...
35
3
11
```

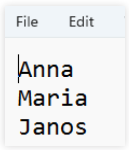
8. feladat – Keresés egy fájlban

Követelmény: Ellenőrizd, hogy a billentyűzetről bevitt név szerepel-e egy fájlban.

Megoldás:

```
keresett_nev = input("Név: ")
megvan = False
with open("e:\\nevek.txt", "r") as f:
    for sor in f:
        if sor.strip() == keresett_nev:
            megvan = True
if megvan:
    print("A keresett név megvan")
else:
    print("A keresett név nincs meg")
```

A nevek.txt fájl




```
>>> keresett_nev = input("Név: ")
... megvan = False
... with open("e:\\nevek.txt", "r") as f:
...     for sor in f:
...         if sor.strip() == keresett_nev:
...             megvan = True
... if megvan:
...     print("A keresett név megvan")
... else:
...     print("A keresett név nincs meg")
...
Név: Maria
A keresett név megvan
```



Önértékelés!

Töltsétek ki a válaszaitokkal egy *onertekeles17.docx* nevű fájlba a következő jelöléseket használva: *1–igen, 2–nem, 3–nem vagyok biztos benne*. Mentsétek a fájlt a *Feladatlap17* nevű mappába.

	A lecke végén tudom...	***Igen***	***Nem***	***Nem biztos***
1.	elmagyarázni, mi az a TextIOWrapper osztály.			
2.	fájlt megnyitni meghatározott kódolással az encoding='utf-8' paraméter használatával.			
3.	hatékonyan, sorról sorra beolvasni egy nagy fájlt, anélkül, hogy a teljes tartalmat betölteném a memóriába..			
4.	elérni egy TextIOWrapper objektum tulajdonságait.			
5.	azonosítani és kezelni a határeseteket: üres fájl, nem létező fájl, érvénytelen adatok..			

6. Nehézségeim voltak a következőkben: …

Ne felejtsétek el értékelni a viselkedéseteket és a tevékenységeteket az óra során!
(a 7. oldalon található viselkedés- és tevékenységfigyelő napló segítségével)

A lecke végén így éreztem magam:

4. Fejezetvégi értékelés

Ellenőrizd a tudásodat!

(Pontszám: 90 pont + 10 pont hivatalból)

Követelmény: Oldjátok meg az alábbi feladatokat a ***tudas_ellenorzes4.docx*** fájlban. Az összes létrehozott fájlt mentsétek a ***Fejezet4*** mappába.

1. **Válaszd ki az alábbi listából azt a helyzetet vagy leírást, amely a legjobban meghatározza a Tkinter könyvtárat a Python nyelvben: a.** Egy kizárólag komplex matematikai számítások elvégzésére használt könyvtár. **b.** Egy grafikus felhasználói felületek (GUI) létrehozására használt szabványos Python könyvtár, amely ablakokat, gombokat, beviteli mezőket és egyéb vizuális vezérlőket biztosít. **c.** Egy speciális funkció, amely csak a szöveg konzolon történő megjelenítését teszi lehetővé. **d.** Képek és multimédiás fájlok tárolására használt adatstruktúra.

2. **Alább látható egy diák által írt két kódrészlet. Mindegyik részlet tartalmaz egy-egy hibát, amely blokkolja a program futását (FileNotFoundError vagy ValueError).**

Követelmény: Minden esetben: **a.** Azonosítsd a hibát generáló kódsort. **b.** Nevezd meg a keletkezett hiba típusát. **c.** Írd le a kód helyes változatát.

1. eset

```
f = open("adatt.txt", "r")
tartalom = f.read()
print(tartalom)
f.close()
```

(A könyvtárban csak az „adat.txt” fájl létezik.)

2. eset

```
f = open("szamok.txt", "r")
sor = f.readline()
n = int(sor)
print(n)
f.close()
```

(A fájl első sorában lévő szöveg: „húsz”.)

3. **Párosítsd a bal oldali oszlopban leírt helyzetet a jobb oldali oszlop megfelelő elemével**.

Helyzet	Tkinter elem
1. Megjelenít egy szöveget egy ablakban	a. Entry
2. Lehetővé teszi az adatbevitelt billentyűzetről	b. Button
3. Megnyomásra végrehajt egy műveletet	c. Label
4. Létrehozza a főablakot	d. Tk()

4. **Hozz létre egy mini Tkinter alkalmazást, amely tartalmaz:** egy főablakot; egy „Egyszerű számológép” szövegű címkét (Label); két beviteli mezőt (Entry); egy „Összead” gombot; valamint a bevezetett számok összegének megjelenítését egy címkében. A programnak megfelelően kell működnie, és lehetővé kell tennie a művelet ismételt elvégzését.

5. **Adott egy felhasználói hitelesítésre szolgáló alkalmazás.** Melyik Tkinter komponensek a legalkalmasabbak egy olyan bejelentkezési ablak megvalósításához, amely tartalmazza a következőket: felhasználónév; jelszó; bejelentkezés gomb. Indokold a válaszodat, és készíts egy mini Python alkalmazást, amely szemlélteti annak működését.

6. **Egy grafikus alkalmazást szeretnénk készíteni a napi hőmérsékletek nyilvántartására.**
Az alkalmazásnak lehetővé kell tennie a hőmérsékletek bevitelét **n** napra egy Entry mező és egy „Hozzáad” gomb segítségével. Írj egy Tkinter programot, amely elvégzi a következő feladatokat: beolvassa és elmenti a bevezetett hőmérsékleteket egy listába; azonosítja és megjeleníti a hibás hőmérsékleteket (-40^0C-nál kisebb vagy 60^0C-nál nagyobb); ha nincsenek hibák, megjeleníti a „Helyes adatok” üzenetet; kiszámítja és megjeleníti az átlaghőmérsékletet, kizárólag a valós (érvényes) értékeket használva; megjeleníti azon napok számát, amelyeken a hőmérséklet meghaladta a kiszámított átlagot.

7. **Egy könyvesbolt egy grafikus felületű alkalmazással szeretné nyomon követni a könyvértékesítéseket.**
Írj egy Tkinter programot, amely beolvassa: az eladott könyvek **n** számát; az eladott könyvek **n** darab kódját (1 és 1000 közötti értékű természetes számok). A programnak gombok segítségével lehetővé kell tennie a következő műveletek elvégzését: a különböző kódok számának megjelenítése; a felhasználó által megadott **x** kód keresése; az x kód első előfordulási pozíciójának megjelenítése; az **x** kód előfordulási számának megjelenítése; az el nem adott kódok megjelenítése; a kódok megjelenítése az előfordulási gyakoriság csökkenő sorrendjében. A felületnek legalább a következőket kell tartalmaznia: Label; Entry; Button; Listbox vagy Text az eredmények megjelenítésére.

Pontozás: 1. feladat – **10 pont**, 2. feladat – **15 pont**, 3. feladat – **15 pont**, 4. feladat – **10 pont**, 5. feladat – **10 pont**, 6. feladat – **10 pont**, 7. feladat – **20 pont**.



Projekt

(100 pont)

„Grafikai alkalmazások a mindennapi életben”

Készíts Pythonban egy ***alkalmazas.py*** nevű grafikus alkalmazást a Tkinter könyvtár használatával. Az alkalmazásnak egy szabadon választott gyakorlati problémát kell megoldania (például: könyvtárkezelés, osztályzatok katalógusa, telefonkönyv, költségvetés-tervező, tevékenységszervező, terméknyilvántartás stb.).

Az alkalmazásnak legalább a következőket kell tartalmaznia: egy testreszabott főablak; címkék (Label); adatbeviteli mezők (Entry); legalább három gomb (Button); egy eredmény-megjelenítési terület (Label, Text vagy Listbox); a felhasználó által beírt adatok validálása; Tkinter függvények és események használata. A projektet az osztály előtt kell bemutatni.

Értékelési útmutató (100 pont): 10p – A grafikus felület tervezése és felépítése, **10p** – A Tkinter komponensek helyes használata, **10p** – A gombok és események helyes működése, **10p** – A felhasználó által megadott adatok kezelése, **10p** – Adatvalidálás és hibakezelés, **10p** – A forráskód minősége és szervezése **10p** – A használt komponensekre vonatkozó elméleti magyarázatok helyessége, **10p** – A prezentáció világossága és az informatikai terminológia használata, **10p** – A csapattagok egyenlő részvétele a prezentációban, **10p** – Kreativitás és vizuális megjelenés.

(**Ajánlások**: 2–3 diákból álló csapatokban dolgozzatok. Minden csapat egy értékelőlap alapján pontszámot ad a többi csapatnak, a végső pontszám pedig az elért pontszámok számtani közepe lesz.)



Kutass!

(100 pont)

Esettanulmány: Hogyan építsünk fel egy felhasználóbarát felületet?

Dokumentáld a tkinter.docx fájlban a következő helyzetet: Egy cég egy asztali alkalmazást szeretne fejleszteni az ügyfelek időpontfoglalásainak kezelésére. A felületnek intuitívnak és könnyen használhatónak kell lennie. Használj különféle forrásokat, és végezd el az alábbi feladatokat:

- Járj utána, mi az a grafikus felület, és milyen előnyei vannak a konzolos alkalmazásokkal szemben. **(20p)**
- Dokumentáld a főbb Tkinter komponenseket: Tk(), Label, Entry, Button, Text, Listbox és Frame. Mutasd be mindegyik szerepét egy táblázatban. **(20p)**
- Készíts és magyarázz el egy olyan ablakpéldát, amely legalább öt különböző komponenst használ. Mellékelj képernyőképeket és magyarázatokat. **(20p)**
- Írd le 4–6 sorban, milyen ajánlásokat tennél egy hatékony és könnyen használható grafikus felület elkészítéséhez. **(10p)**
- Keress a Tkinter dokumentációjában egy olyan komponenst vagy metódust, amelyet nem mutattak be az órán (például: Checkbutton, Radiobutton, Scale, Menu, Spinbox, Combobox). **(10p)**
- Írd le a működését, és készíts egy gyakorlati példát a használatára. **(10p)**
- Használd fel a tanulmányozott komponenst egy saját mini alkalmazásba, és magyarázd el a használatának előnyeit. **(10p)**



Portfólió

(100 pont)

E fejezet végén a diák személyes portfóliójának a ***Fejezet4*** mappában tartalmaznia kell a következő anyagokat a megfelelő megoldásokkal: ***Feladatlap13***, ***Feladatlap14***, ***Feladatlap15***, ***Feladatlap16*** és ***Feladatlap17*** mappa, a ***Projekt4*** mappa az ***alkalmazas.py***, ***tkinter.docx*** fájlokkal és a ***tudas_ellenorzes4.docx*** fájl**.**

Értékelés módja: Megvan a ***Fejezet4*** mappa a megadott könyvtárszerkezettel – **20p**

Megvan:
- A fájlok több mint 95%-a – **80p**
- A fájlok 75–94%-a – **60p**
- A fájlok 50-74%-a – **40p**
- A fájlok 25-49%-a – **20p**
- 25% alatt – **0p**

5. Fejezet – Az adatok fogalmi szervezése (listák)

18. Lecke – A lineáris fogalmi modell: jellemzők, közvetlen és szekvenciális hozzáférés

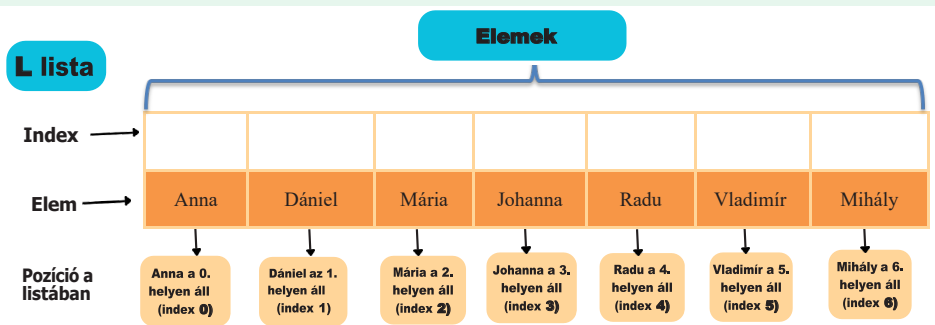
Egy probléma számítógéppel történő megoldásának folyamata több szakaszból áll. Az első szakasz az ***elemzés***, amely során a probléma adatainak azonosítása történik. Ezt követi a második szakasz, a ***tervezési*** fázis. Ez a szakasz egy megfelelő algoritmus kereséséből áll, amely magában foglalja az adatok rendszerezési módjának meghatározását is. A programozásban az ***adatok rendszerezésének folyamata*** azért hasznos, hogy az adatokat hatékonyan és logikusan lehessen tárolni, kezelni és feldolgozni. Gyakorlatilag az adatok rendszerezése segít megérteni, hogyan tárolódnak az adatok, és hogyan lehet azokat felhasználni. Minden program adatokkal dolgozik: nevekkel, számokkal, jegyekkel, üzenetekkel, termékekkel és egyebekkel. Az adatok rendszerezése segít abban, hogy azokat világos és rendezett formában tároljuk, így a program a lehető leggyorsabban megtalálhatja és használhatja azokat. Például az ***iskolai napló*** alkalmazásnak – többek között – a diákok neveit és jegyeit kell tárolnia.

Megj. Az informatikában az adatok egyszerű vagy összetett adatok formájában szervezhetők. Az ***egyszerű adatok*** az információ legkisebb egységei, és nem bonthatók tovább, míg az ***összetett adatok*** több egyszerű adat összekapcsolásával jönnek létre.

Példák egyszerű adatokra	Példák összetett adatokra
egy szám: 23	egy jegyekből álló lista: [8, 9, 10]
egy karakter: 'A'	egy teljes név, amely vezeték- és utónévből áll: "Pap Anna"
egy logikai érték: True	egy naptári dátum nap, hónap, év formájában: 12 január 2026

A **lineáris fogalmi modell**t gyakran **lista** típusú adatstruktúrák vagy **egydimenziós tömbök** (értékek sorozata) formájában ábrázolják. Ez az információk szekvenciális (egymást követő) rendszerezésének egy formája, amelyben az elemek egymás után helyezkednek el.

<def/> A **lista** egy olyan összetett adatstruktúra, amely meghatározott sorrendben tárol véges számú elemet. Az elemek sorrendjét az egyes elemek listában elfoglalt helye határozza meg, amely pozíció egy ***index*** segítségével azonosítható (***index*** = az elem pozíciója a listában, az első elemtől kezdve 0-tól számozva).

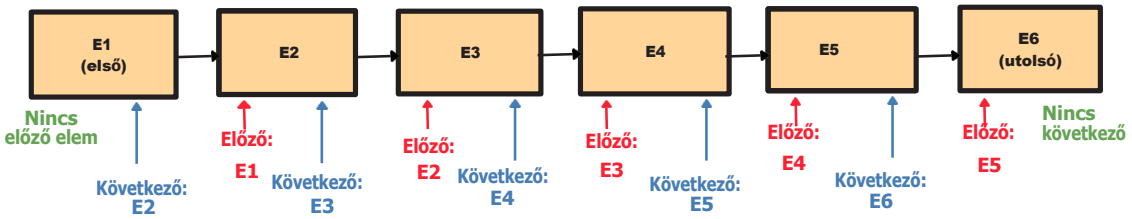


A C++ programozási nyelvben a listák/tömbök azonos típusú elemeket tartalmaznak. A Python programozási nyelvben a listák különböző típusú elemeket is tartalmazhatnak, ahogyan az alábbi példában láthatod.

Python programozási nyelv	C++ programozási nyelv
0 1 2 3	0 1 2 3 4 5
`lista=[1, 3.50, "a", "hétfő van"]`	`v=[12, 2, 3, 6, 9, 1]`

Egy lista a következő fogalmi jellemzőkkel rendelkezik:

- ***összetett adatstruktúra***: a lista több elemet tartalmaz, amelyek együtt rendszerezett értékek egy csoportját alkotják;
- ***szekvenciális (lineáris) rendszerezés***: a lista elemei úgy vannak elrendezve, hogy egyszerű egymásutánisági kapcsolatokat követnek: minden elemnek van egy megelőző eleme (az első kivételével) és egy rákövetkező eleme (az utolsó kivételével);



- ***jól meghatározott sorrend:*** minden elem egy egyértelmű (indexelt) pozíciót foglal el, amely lehetővé teszi az azonosítását és a közvetlen hozzáférést az első elemtől az utolsóig;
- ***szabályozott hozzáférés:*** a lista mind a közvetlen, mind a szekvenciális hozzáférést lehetővé teszi a szükségletek, valamint a fizikai tárolóeszköz függvényében;
- ***változó méret:*** egy listához hozzáadhatók vagy belőle törölhetők elemek.

Példa:

0 1 2 3 4 5

Legyen lista L=[4, 8, 15, 2, 32, 23]. Ebben a listában:

- a 4 az első elem
- a 23 az utolsó elem
- a 8 előtti elem a 4 és az utána levő a 15

A 15 indexe 2, vagyis L[2]=15.

Valódi életből vett példák a listákra:

- ✓ könyvtári könyvek listája
- ✓ egy esemény meghívottjainak listája
- ✓ egy diák informatika jegyeinek listája
- ✓ egy online áruház rendeléseinek listája
- ✓ egy osztály diákjainak listája
- ✓ egy előadó dalainak listája

A listákkal végezhető alapvető műveletek a következők:

Művelet	Leírás
Létrehozás	A memóriaterület lefoglalása és a struktúra inicializálása egy új (kezdetben üres) lista számára.
Végigjárás	A lista minden egyes elemének szekvenciális (egymás utáni) meglátogatása az elejétől a végéig, azok feldolgozása vagy megjelenítése céljából.
Beszúrás	Új elem hozzáadása a lista egy meghatározott pozíciójához (az elejére, a végére vagy egy belső pontjára).
Törlés	Egy létező elem eltávolítása a listából, valamint a megmaradt elemek közötti kapcsolatok (láncolatok) helyreállítása.
Keresés	Egy konkrét elem beazonosítása a listában egy megadott érték vagy feltétel alapján.
Hozzáférés	Egy adott pozíción lévő elem értékének kiolvasása vagy módosítása, a lista struktúrájának (szerkezetének) megváltoztatása nélkül.

Sajátos listatípusok a hozzáférés függvényében

A listák osztályozhatók a hozzáférés típusa alapján, de az alapműveletek végrehajtásának módja szerint is. A **lista elemeihez való hozzáférés** azt a módot jelenti, ahogyan az elemeket lokalizáljuk.

A hozzáférés típusa szerint a listák lehetnek: ***közvetlen hozzáférésűek*** és ***szekvenciális hozzáférésűek***. Bizonyos hozzáférési korlátozások által meghatározott különleges listák is léteznek: a ***vermek*** és a ***sorok***. Ezeket az elemek hozzáadási és törlési műveletek végrehajtásának módja teszi sajátossá.

A ***közvetlen hozzáférés*** lehetőséget biztosít egy elem azonnali lokalizálására annak indexe vagy címe alapján, anélkül, hogy végig kellene haladni a megelőző elemeken. Minden elem közvetlenül, a pozíciója alapján érhető el.

Pl. Legyen az L=[7, 14, 21, 28] számokat tartalmazó lista. Közvetlenül elérhetjük a 0 indexű elemet (L[0]=7) vagy a 2 indexű elemet (L[2]=21).

A ***szekvenciális hozzáférés*** az elemek sorrendben történő bejárását feltételezi, a lista elejétől indulva és egymás után haladva. Ily módon az elemeket egymás után érjük el. Ahhoz, hogy eljussunk egy elemhez, végig kell haladni az előtte lévő elemeken.

	Közvetlen hozzáférés	Szekvenciális hozzáférés
Előnyök	✓ gyors hozzáférés egy elemhez	✓ a beszúrások/törlések hatékonyak
Hátrányok	× a beszúrások/törlések nem hatékonyak	× lassú hozzáférés a lista egy tetszőleges eleméhez

Megj. Amikor egy listával dolgozunk, a következőket kell szem előtt tartanunk:

- A ***lista bejárásának módja***: az index az, ami megmutatja, hol tartózkodunk, a leállási feltétel, hogy tudjuk, elértük-e a lista végét (a lista felső határát), a lista bejárásának iránya, a megfelelő mutatók megválasztása.
- A ***felhasznált tárhely hatékonysága***: vannak helyzetek, amikor a lista elemeit el kell tárolnunk a memóriában a feldolgozásukhoz (például egy sorozat rendezése során), vagy egyszerre csak egyetlen elemet tárolunk, ami a felhasznált memória szempontjából hatékonyabb megoldás.

ALKALMAZÁSOK

1) ***Tárolás nélküli feldolgozás:*** végighaladunk a listán úgy, hogy az aktuális elemet egy változóban tároljuk. Ezáltal erőforrást (memóriát és végrehajtási időt) takarítunk meg. A lista elemeinek tárolás nélküli feldolgozását feltételező alapvető algoritmusok: az elemek összegének/szorzatának kiszámítása, a maximális/minimális érték megkeresése, egy adott feltételnek megfelelő elemek megszámlálása stb.

Pl. Adott egy n természetes számból álló sorozat (n nemnulla természetes szám). Hány olyan eleme van a sorozatnak, amely legalább kétjegyű?

Algoritmus	Python kód
beolvas n (természetes, nem nulla) ct=0 ┌minden i=1,n végezd el │ beolvas x │ ┌ha x>9 akkor │ │ ct=ct+1 │ └ha vége └mindenvége kiír ct	`n = int(input("Kérem n értékét: "))` `ct = 0` `for i in range(1, n + 1):` `  x = int(input(f"Kérem az elemet  {i}:"))` `  if x > 9:` `  ct = ct + 1` `print("Az eredmény: ", ct)`

2) ***Tárolással történő feldolgozás:*** az összes elemet elmentjük egy listába, amikor az algoritmusnak ismételt vagy nemlineáris hozzáférésre van szüksége az adatokhoz. Alapvető algoritmusok: az elemek rendezése, egy érték bináris keresése egy listában, az elemek sorrendjének megfordítása, bizonyos tulajdonságok ellenőrzése, a vektorban lévő – nem feltétlenül egymást követő – azon elempárok megszámlálása, amelyek teljesítenek egy bizonyos feltételt stb.

Pl. Adott egy n természetes számból álló sorozat (n nemnulla természetes szám). Hány olyan elempár van a sorozatban, amelyeknek az egyes helyiértékén ugyanaz a számjegy szerepel? Például, ha n=7, és L=[7, 12, 17, 51, 412, 67, 3], az egyesek helyén azonos számjeggyel rendelkező párok a sorozatban a következők: (7, 17), (7, 67), (12, 412), (17, 67), tehát a számuk 4.

```
beolvas n (nem nulla természetes szám)
  minden i=1,n végezd el
  beolvas v_i
  minden vége
ct=0
  minden i=1,n végezd el
    minden j=i+1,n végezd el
      ha vi %10==v_j %10  akkor
      ct=ct+1
      ha vége
    minden vége
  minden vége
kiír ct
```

Jegyezd meg!

- ✔ A **tárolás nélküli feldolgozás**i módot akkor választjuk, ha a lista elemeinek egyszeri bejárása is elegendő, mivel az elemeket egymás után, a hozzáférésük pillanatában dolgozzuk fel, anélkül, hogy visszatérnénk a korábbi elemekre.
- ✔ A **tárolással történő feldolgozás**i módot (memóriában lévő tömbökben/listákban) akkor választjuk, ha az algoritmus megköveteli az ugyanazon adatokhoz való többszörös hozzáférést vagy a nemlineáris hozzáférést.

Feladatlap 18

Hozzátok létre a *Feladatlap18* nevű mappát. Oldjátok meg az alábbi feladatokat, a válaszokat pedig jegyezzétek le egy *válaszok.rtf* nevű fájlba.

Az alábbi 1. és 2. feladatnál válasszátok ki a helyes választ (csak egy válasz jó).

1. A lineáris koncepcionális modell jellemzője:

a. a. összetett kapcsolatok az elemek között; **b.** hierarchikus felépítés;
c. szekvenciális felépítés; **d.** végtelen számú.

2. Az elemhez való közvetlen hozzáférés a következőre vonatkozik:

a. lassú véletlen elérés; **b.** index használata;
c. az összes elem bejárása; **d.** egy következő elem létezése.

3. Egészítsd ki az alábbi mondatot a megfelelő válaszokkal:

A lista elemeihez való hozzáférés lehet: …… és …… .
A lista változó mérete a következőre vonatkozik: …… .

4. Írj 3 érvet, amellyel igazolod a következő állítást: „A ***lista*** *egy összetett adatstruktúra, amely meghatározott sorrendben véges számú elemet tárol.*”, a lista koncepcionális jellemzőiből kiindulva.

5. Az alábbi esetek közül melyiknél kötelező listát használni az egyszerű változók helyett, és miért?

a. egy adott város napi hőmérsékleti adatainak rögzítése egy hónapon keresztül, hogy későbbi kérdésekre válaszolhassunk (*például*: melyik napon volt a legalacsonyabb a hőmérséklet);

b. annak ellenőrzése, hogy egy szám páros vagy páratlan;

c. a páratlan számok megszámlálása n darab megadott szám közül;

d. az egymást követő páratlan számpárok megszámlálása n darab megadott szám közül.

6. Írj egy algoritmust és egy Python-programot a következő feladat megoldására: *“Adott egy n elemből álló természetes sorozat (n nem nulla természetes szám). Hány egymást követő egyenlő elempár van a sorozatban?”*. Indokold meg a memóriában való tárolás nélküli vagy a memóriával történő feldolgozás választását.

7. Egy sportesemény résztvevőit a személyi számuk (CNP) alapján regisztrálják. Írj egy algoritmust és egy Python-programot, amely meghatározza, hány személy született novemberben.

8. Készíts egy érvelő esszé típusú ***projektet*** a ***Feladatlap18*** mappába, melynek címe: A listák jelentősége az adatszervezésben. A projekt mutasson rá az átmenetre a mindennapi listahasználat és a technikai tulajdonságok között, érvelve amellett, hogy miért alapvető ez a struktúra az informatikában és a mindennapi életben egyaránt.



Önértékelés!

Töltsétek ki a válaszaitokkal egy *onertekeles18.docx* nevű fájlba, a következő jelöléseket használva: *1–igen, 2–nem, 3–nem vagyok biztos benne*. Mentsétek a fájlt a *Feladatlap18* nevű mappába.

	A lecke végén tudom...	***Igen***	***Nem***	***Nem biztos***
1.	definiálni a listát és felsorolni a fogalmi jellemzőit.			
2.	a közvetlen és a szekvenciális elérés közötti különbséget.			
3.	mikor hasznos a tárolás nélküli és mikor a tárolással történő feldolgozás.			
4.	hogy írjak egy egyszerű algoritmust egy lista tárolás nélküli feldolgozására.			

5. Nehézségeim voltak a következőkben: …

Ne felejtsétek el értékelni a viselkedéseteket és a tevékenységeteket az óra során!
(a 7. oldalon található viselkedés- és tevékenységfigyelő napló segítségével)

A lecke végén így éreztem magam:

19. Lecke – Gyakorisági/ előfordulási listák, a verem (LIFO) és a sor (FIFO)

<def/> Egy ***gyakorisági lista*** egy olyan adatstruktúra, amely egy adathalmaz minden egyes különböző értékéhez hozzárendeli annak előfordulási számát (gyakoriságát) egy sorozatban. A lista indexét használja sorszámként, így a nyers adatokat egy kompakt formában ábrázolja, ami hasznos a statisztikában és az adatfeldolgozásban.

Pl. Az informatikatanár szeretné tudni, hogyan teljesített az osztály 30 diákja az első felmérőn. A diákok által kapott osztályzatok a következők: 7, 5, 4, 8, 9, 9, 10, 7, 8, 7, 4, 5, 7, 9, 10, 4, 6, 9, 8, 9, 3, 6, 7, 8, 9, 9, 7, 8, 10, 9. Bizonyos értékek kiderítéséhez a gyakorisági tömbben meg fogja számolni minden egyes lehetséges osztályzat előfordulásainak számát, a következőképpen:

Jegy	1	2	3	4	5	6	7	8	9	10
Gyakoriság	0	0	1	3	2	2	6	5	8	3

A fenti adatok alapján a tanár különféle következtetéseket vonhat le, mint például: melyik osztályzatot kapták a legtöbben, hány diáknak van szüksége sürgős felzárkóztatásra, mekkora az átmeneti arány, vagy mennyi a maximális osztályzatot elért tanulók száma.

A gyakorisági lista felépítésének algoritmusa:

Lépések	**1. lépés:** Létrehozunk egy fix méretű listát (vektort), amelynek mérete megegyezik azon értékek tartományával, amelyeknek a gyakoriságát ki szeretnénk számítani (*például* osztályzatok esetén 0-tól 10-ig) **2. lépés:** A lista minden egyes elemét nullával (0) inicializáljuk. **3. lépés:** Elemről elemre végighaladunk a bemeneti adathalmazon. **4. lépés:** Minden egyes elemnél 1-gyel növeljük a gyakorisági lista azon indexén lévő értéket, amely megegyezik az adott elem értékével.

Pszeudokód:

Létrehozunk egy L gyakorisági listát, ahol L[i] ← 0 a bemeneti adathalmaz minden egyes értékére
Minden egyes x elemre a bemeneti adathalmazból:
L[x] ← L[x] + 1
vége minden

Előnyök

- ✓ Azonos típusú adatok világos rendszerezése
- ✓ Memóriahatékonyság
- ✓ Az adatok gyakoriságának gyors kiszámítása (%)
- ✓ Bizonyos értékek könnyű azonosítása (legnagyobb gyakoriságú érték, legkisebb gyakoriságú érték, súlyozott átlag)
- ✓ Grafikus ábrázolások alapja (hisztogram, kördiagram)
- ✓ Nagy adatmennyiség gyors feldolgozása
- ✓ Hasznos a statisztikai elemzésekben és jelentésekben.

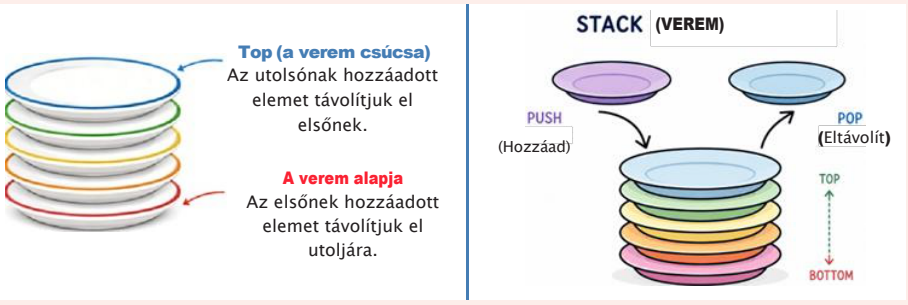
Hátrányok

- × Nem őrzi meg az adatok eredeti sorrendjét
- × Az egyedi adatok eltűnnek, így a kontextus elveszhet (már nem tudjuk, hogy egy bizonyos diák hányas osztályzatot kapott)
- × Nem hatékony egyedi értékek esetén

Verem (Stack) – A LIFO elv

<def/> A **verem** egy olyan lineáris adatstruktúra, amelyben az elemek beszúrása és törlése is csak egyetlen végponton, az úgynevezett **csúcson (top)** keresztül történhet. Működési elve a **LIFO – Last In, First Out** (utolsóként érkezett, elsőként kiszolgált).

Pl. Egy étterem konyhájában a tányérokat egymásra pakolják, így egy tornyot (vermet) alkotnak. Amikor szükség van egy tányérra, a legfelsőt (a tetejéről) veszik el. Egy új tányért a már meglévők tetejére (a verem tetejére) helyeznek rá.



Alapműveletek:

- **Push** – elem hozzáadása a verem tetejére;
- **Pop** – elem eltávolítása a verem tetejéről;
- **Top** – a verem tetején lévő elem megtekintése annak eltávolítása nélkül;
- **IsEmpty** – ellenőrzi, hogy a verem üres-e.

Előnyök

- ✓ Egyszerű megvalósítás
- ✓ Gyors hozzáférés az utolsó elemhez
- ✓ Végrehajtási sebesség: A Push és Pop műveletek rendkívül gyorsak (O(1) bonyolultságúak).

Hátrányok

- × Korlátozott hozzáférés (csak a tetejéhez férünk hozzá)
- × Nem teszi lehetővé a közvetlen hozzáférést

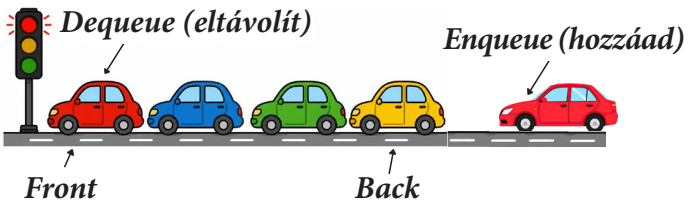
Pl. Egy **verem** az alábbiakból állhat:

- Egy halom dolgozat a tanári asztalon (a veremben elfoglalt sorrendjük szerint lesznek kijavítva);
- Több könyv, egymásra helyezve.

A várakozási sor (Queue) – FIFO elv

<def/> A **várakozási sor** egy olyan lineáris adatstruktúra, amelyben az elemek beszúrása az egyik végén, a **hátulján (rear – tail)** történik, míg az eltávolításuk a másik végén, az **elején (front – head)**. Működési elve a FIFO – First In, First Out (elsőként érkezett, elsőként kiszolgált).

Pl. A jelzőlámpánál az autók az érkezési sorrendjükben állnak meg, és ugyanebben a sorrendben fognak elindulni.



Alapműveletek:

- **Enqueue** – elem hozzáadása a sorhoz (mindig a sor végére);
- **Dequeue** – elem eltávolítása a sorból (mindig a sor elején lévő elemet);
- **Front/Peek** – lehetővé teszi a sor első elemének megtekintését annak eltávolítása nélkül;
- **IsEmpty** – ellenőrzi, hogy a sor üres-e.

Előnyök
- ✓ Egyszerű megvalósítás
- ✓ Gyors hozzáférés az első és az utolsó elemhez
- ✓ Végrehajtási sebesség: A hozzáadás (Enqueue) és az eltávolítás (Dequeue) műveletek rendkívül gyorsak (bonyolultságuk O(1)).

Hátrányok
- × Korlátozott hozzáférés az első és az utolsó elemhez.
- × Nem teszi lehetővé a közvetlen hozzáférést (ha az x-edik pozíción lévő elemet szeretnénk elérni, el kell távolítani a sorban előtte lévő elemeket)

Pl. A nyomtatóra elküldött dokumentumok kezelése **Enqueue** (amikor a „Nyomtatás” gombra kattintunk) és **Dequeue** (amikor a nyomtató befejezi egy oldal nyomtatását) típusú műveletekkel történik.

Megj. Egy algoritmus megvalósításakor, ha számít az elemek listához adásának sorrendje, akkor a sort választjuk, ha pedig azt szeretnénk, hogy az utolsóként hozzáadott elem élvezzen elsőbbséget, akkor a vermet választjuk.

Szempont	Lista	Verem (Stack)	Sor (Queue)
Működési elv	Szabad hozzáférés: Az elemek bármely pozícióból elérhetők, beszúrhatók vagy törölhetők.	LIFO (Last In, First Out): Az utolsóként hozzáadott elem az első, amit kiveszünk.	FIFO (First In, First Out): Az elsőként hozzáadott elem az első, amit kiveszünk.
Hozzáférési pontok	Bármely elem (a végétől/végeitől való bejárással)	Egyetlen végpont, amelyet csúcsnak nevezünk.	Két végpont: utolsó elem (Rear) – a beszúráshoz, és első elem(Front) – a törléshez.
Beszúrási művelet	Beszúrás az elejére, a végére vagy a belsejébe.	Push (elem hozzáadása a verem tetejéhez)	Enqueue / Add (hozzáadás a sor végéhez).
Törlési művelet	Törlés bármely pozícióról	Pop (elem eltávolítása a verem tetejéről).	Dequeue / Remove (eltávolítás a sor elejéről).
Megtekintési művelet	Hozzáférés/Olvasás bármely pozícióról.	Peek / Top (megtekinti a csúcson lévő elemet, anélkül hogy törölné azt).	Peek / Front (megtekinti az első elemet, anélkül hogy törölné azt)
Példák a mindennapi életből	Egy bevásárlólista vagy egy zenei lejátszási lista (playlist).	Egymásra rakott elmosogatott tányérok halmaza vagy a böngésző előzményei (Visszavonás / Undo).	Egy boltban várakozó emberekből álló sor vagy a nyomtató, amely egy sor dokumentumot nyomtat ki.

Feladatlap 19

Hozzátok létre a *Feladatlap19* nevű mappát. Oldjátok meg az alábbi feladatokat, a válaszokat pedig jegyezzétek le egy *valaszok.rtf* nevű fájlba.

Az alábbi 1. és 2. feladatnál válasszátok ki a helyes választ.

1. A verem egy olyan adatszerkezet, amely az alábbi elv szerint működik:

a. Elsőnek érkezett – elsőnek kiszolgált (F.I.F.O); **b.** Utolsónak érkezett – elsőnek kiszolgált (L.I.F.O);
c. Nincs kezelési elve ; **d.** Közvetlen hozzáférés bármelyik elemhez.

2. Rendeld hozzá az alábbi esetekhez a megfelelő adatszerkezetet (Verem/LIFO vagy Várakozási sor/FIFO), és indokold a választást.

a. Böngészési előzmények (a „Vissza” gomb): a Vissza gomb megnyomásával az utoljára meglátogatott oldalra térünk vissza.

b. Irodai nyomtató: 5 kolléga küld dokumentumokat nyomtatásra egy rövid időintervallumon belül.
c. Drive-thru egy gyorsétteremben: az autók behajtanak, és az érkezési sorrendben szolgálják ki őket.
d. Híváskezelés egy technikai ügyfélszolgálati központban.

3. Egy informatikai szolgáltató cég 40 ügyfél válaszát gyűjtötte össze a következő kérdésre: „Mennyire elégedett szolgáltatásainkkal?”. A válaszlehetőségek a következők voltak: *Nagyon elégedett (NE)*, *Elégedett (E)*, *Semleges (S)*, *Elégedetlen (EL)*, *Nagyon elégedetlen (NEL)*. A szolgáltatások javítása érdekében végezd el az összegyűjtött adatok elemzését! A kapott válaszok: *NE, NE, S, E, NE, EL, NEL, E, NE, E, S, NE, E, NE, E, NE, E, S, E, NE, EL, NE, E, NE, S, E, NE, NE, E, NE, S, E, EL, NE, E, NE, E, EL, NEL, E*. A szolgáltatások javítása érdekében az adatok elemzéséhez számoljátok ki a válaszok gyakoriságát, és töltsétek ki az alábbi táblázatot a *valaszok.docx* fájlban:

Minősítés	Gyakoriság	Százalék	Értelmezés
Nagyon elégedett	15	37.5%	Leggyakoribb válasz – értékelt szolgáltatás
Elégedett			
Semleges			
Elégedetlen			
Nagyon elégedetlen			
ÖSSZESEN			

4. ***Projekt:*** Hozz létre egy **fogalmi modellt** egy modern kézbesítési központ irányítási rendszeréhez. A kihívás egy csomag „útvonalának” megszervezése három különböző adatszerkezet intelligens használatával, oly módon, hogy a munkafolyamat optimalizált legyen, és ne alakuljanak ki torlódások: egy lista (általános raktár), egy várakozási sor / queue (rendelési lista) és egy verem / stack (ahogyan a futár elhelyezi a termékeket a kiszállításhoz).



Önértékelés!

Töltsétek ki a válaszaitokkal egy *onertekeles19.docx* nevű fájlba, a következő jelöléseket használva: *1 – igen, 2 – nem, 3 – nem vagyok biztos benne*. Mentsétek a fájlt a *Feladatlap19* nevű mappába.

	A lecke végén tudom...	***Igen***	***Nem***	***Nem biztos***
1.	definiálni a gyakorisági listát.			
2.	leírni a verem LIFO elvét, és példákat mondani rá a mindennapi életből.			
3.	leírni a sor FIFO elvét, és példákat mondani rá a mindennapi életből.			
4.	azonosítani a verem alapvető műveleteit.			
5.	azonosítani a várakozási sor alapvető műveleteit.			

6. Nehézségeim voltak a következőkben: …

Ne felejtsétek el értékelni a viselkedéseteket és a tevékenységeteket az óra során!
(a 7. oldalon található viselkedés- és tevékenységfigyelő napló segítségével)

A lecke végén így éreztem magam:

20. Lecke – A Python list osztálya: sajátos operátorok és alapvető metódusok

Pythonban a ***lista*** egy sokoldalú adattípus, amely elemek rendezett és mutábilis (módosítható) gyűjteményének tárolására szolgál. A lista ***rendezett***, mivel megőrzi az elemek hozzáadásának sorrendjét, és ***módosítható*** (***mutable***), mivel a tartalma megváltoztatható elemek hozzáadásával, törlésével vagy cseréjével. Bár a listák tartalmazhatnak különböző típusú elemeket is, a gyakorlatban leggyakrabban azonos típusú adatok tárolására használják őket. A listákat Pythonban úgy definiáljuk, hogy az elemeket szögletes zárójelek közé ***[ ]*** helyezzük és vesszővel választjuk el egymástól, vagy a ***list()*** konstruktor használatával.

```
Üres lista: L=[]
Homogén lista L=['sarga', 'piros', 'feher']
Heterogén lista L=['banan', 23, 130, True, [1,2,3]]
```

```
sajat_lista = list(('piros', 'zold', 'feher')) #['piros', 'zold', 'feher']
print(sajat_lista)
```

```
sajat_lista = ['Anna', 'Maria', 'Viktor']     #['Anna', 'Maria', 'Viktor']
print(sajat_lista)
```

Az elemek eléréséhez, az ***indexelő operátort*** vagy a ***slicing*** (***szeletelő***) operátort használjuk.

Az ***indexelő operátor***t szögletes zárójelek képviselik, **lista_ID[index]** formában.

Az ***indexelés*** lehet: ***pozitív*** (balról jobbra, az első elem indexével kezdve, ami a 0) és ***negatív*** (jobbról balra, -1-gyel kezdve, ami az utolsó elem indexe, -2 az utolsó előtti elem indexe stb.). **L** = [**'Anna'**, **'Maria'**, **'Mate'**, **'Viktor'**, **'Tamas'**]

-5	-4	-3	-2	-1
0	1	2	3	4
'Anna'	'Maria'	'Mate'	'Viktor'	'Tamas'

Negatív indexelés
Pozitív indexelés

L[2]
L[-3] → 'Mate'

A szeletelés **(slicing)** operátort a **:** jel képviseli, amely a lista egymást követő elemeinek elérésére szolgál a **[start:stop:pas]** (kezdet:vég:lépésköz) formátumban, ahol:

- **start** = az index, ahonnan a kinyerés kezdődik (az ezen az indexen lévő elem bele van számítva). Ha a start hiányzik, az alapértelmezetten 0-nak minősül.
- **stop** = az az index, ahol a kinyerés véget ér (az ezen az indexen lévő elem ki van zárva). Ha a stop hiányzik, a bejárás a lista végéig tart.
- **pas** = a kinyert elemek közötti lépésközt jelenti. Ha hiányzik, az alapértelmezett lépésköz 1. Ha a pas (lépésköz) értéke negatív, a lista bejárása jobbról balra történik.

```
L = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H']
print(L[2:4])   #['C','D'] megjeleníti a 2-es és 3-as indexen lévő elemeket
print(L[:4])    #['A','B','C','D']   megjeleníti az első 4 elemet
print(L[2:])    #['C','D','E','F','G','H'] megjeleníti az összest a 2-es indextől
print(L[0:8:2]) #['A','C','E','G']megjeleníti a páros indexű elemeket, a 0-tól kezdve
print(L[0:8:3]) #['A','D','G'] megjeleníti az elemeket a 0-s indextől kezdve,
                # 3-asával haladva
print(L[::-1])  #['H','G','F','E','D','C','B','A'] lista fordított sorrendben
print(L[5:2:-1])#['F','E','D'] az elemek megjelenítése jobbról balra történik
                # (az 5-ös, 4-es és 3-as indexű elemek)
print(L[:])     #['A','B','C','D','E','F','G','H'] megjeleníti a lista összes elemét
print(L[::])    #['A','B','C','D','E','F','G','H'] megjeleníti a lista összes elemét
```

Szintaxis	Elért elemek
L[0]	a lista első eleme (a bal szélső)
L[-1]	a lista utolsó eleme (a jobb szélső).
L[i:j]	az ***[i, j)*** index-tartományba eső elemek.
L[i:]	a lista összes eleme az ***i*** indextől kezdve a lista végéig.
L[::] L[:]	a lista összes eleme a ***0***-s indextől az utolsóig.

Szintaxis	Elért elemek
L[:i]	a lista összes eleme a ***[0, i)*** index-tartományban
L[:]	a lista összes eleme.
L[-x:]	a lista utolsó ***x*** eleme.
L[:-x]	a lista összes eleme, az utolsó ***x*** elem kivételével.
L[::-1]	a lista összes eleme, fordított sorrendben.



A lista végéhez viszonyított elemek eléréséhez negatív indexeket használunk.

```
L = ['Anna', 'Maria', 'Viktor', 'Mate', 'Tamas']
print(L[-4:-2])          # ['Maria', 'Victor']
print(L[:-2])            # ['Anna', 'Maria', 'Victor']
print(L[-2:])            # ['Matei', 'Tamas']
```



A Python viselkedése a listahatárok túllépésekor:

- Egyedi indexelés: Egy nem létező index elérése ***IndexError: list index out of range*** hibát eredményez (*például* L[10] hívása egy 5 elemű lista esetén).
- Szeletelés (slicing) határértéken kívüli **start**tal (a kezdőindex meghaladja a lista hosszát): egy üres listát ad vissza, hibaüzenet nélkül.
- Szeletelés (slicing) határértéken kívüli stop-pal (a záróindex meghaladja a lista hosszát): a listát a végéig adja vissza, hibaüzenet nélkül.

```
L=['Anna','Maria','Janos','Emma','Viktor']
print('Emma' in L)          #True
print('Andras' in L)        #False
print('Andras' not in L)    #True
if('Viktor' in L):
      print("Viktor a listaban van")
```

Annak ellenőrzésére, hogy egy elem ***benne van-e*** a listában, az ***in*** tagsági operátort használjuk. Ez True értéket ad vissza, ha az elem szerepel a listában, és False értéket, ha nem. Emellett lehetőség van annak ellenőrzésére is, hogy egy elem szerepel-e a listában, a ***not in*** operátor segítségével.

A ***két vagy több lista összefűzése (konkatenációja)*** az a folyamat, amely során egyesítjük őket egy új lista létrehozása érdekében. Az összefűzés operátora a **+**. Mindkét operandusnak lista típusúnak kell lennie.

```
L1=[1,2,3]
L2=[4,5,6]
eredmény=L1+L2
print(eredmény) #[1, 2, 3, 4, 5, 6]
```

Egy lista megszorzása egy n egész számmal egy olyan új listát eredményez, amely az eredeti lista elemeinek n-szeri megismétlésével jön létre. A szorzás operátora a *.



- A szorzás használható egy lista alapértelmezett értékkel történő inicializálására is.
- Ha egy n≤0számmal próbálunk szorozni, az eredmény egy üres lista lesz.

```
L1 = [1,2,3]
L2=L1*2    #L2=[1, 2, 3, 1, 2, 3]
L3=L1*0    #L3=[]
L4=[0]*5   #L4=[0, 0, 0, 0, 0]
```

A listák összehasonlítása a következő ***relációs operátorokkal*** történik: ==, !=, <, >, <=, >=. A Python a listákat elemről elemre, lexikografikus sorrendben hasonlítja össze, a 0-s indexű elemektől kezdve.

Egyenlőség: két lista akkor egyenlő, ha azonos a hosszuk, és az azonos indexű párokban található összes elemük megegyezik.

Egyenlőtlenség: az elemeket azonos indexenként hasonlítja össze, a 0. indextől kezdve. Ha az összehasonlított elemek egyenlőek, a folyamat a következő párral folytatódik. Ha különbözőek, az elemek közötti összehasonlítás eredménye lesz a végső eredmény. Ha minden elem megegyezik, de a listák hossza eltérő, a rövidebb lista számít kisebbnek.

```
print([3,2,1]==[3,2,1])          #True
print([1,2,3]>[3,2,1])           #False
print([3,2,1]!=['alma',2,1])     #True
```

Megj. A <, >, <=, >= operátorok listák közötti összehasonlításához az összehasonlítandó elemeknek azonos típusúaknak kell lenniük.

A ***listaelemek módosítása*** kétféleképpen történhet: egyetlen elem értékének megváltoztatásával az index megadásával (*például* L[3] = 7) vagy egymást követő elemek egy csoportjának cseréjével, szeletelés (slicing) segítségével. Szeleteléssel történő módosítás esetén a kijelölt szakasz bármilyen iterálható struktúrával helyettesíthető, leggyakrabban egy másik listával.

```
L = [1,2,3,4,5,6]
L[2:4]=[9,10]       #L=[1,2,9,10,5,6]
L[2:4]=[11,12,13]   #L=[1,2,11,12,13,5,6]
L[2:4]=[9]          #L=[1,2,9,10,5,6]
```

A ***helyettesítés*** lehet:

- **egyenlő** (egy elemcsoportot egy ugyanannyi elemet tartalmazó másik csoportra cserélünk)
- **bővítő.** Akkor fordul elő, ha a kicserélendő lista kevesebb elemet tartalmaz, mint az a lista, amellyel a helyettesítést végezzük. (Az L1[start:stop] = L2 parancs kicseréli az L1 listában megadott indexű elemeket az L2 lista értékeire).
- **szűkítő.** Akkor fordul elő, ha a kicserélendő lista több elemet tartalmaz, mint az a lista, amellyel a helyettesítést végezzük. (Az L1[start:stop] = L2 parancs kicseréli az L1 listában megadott indexű elemeket az L2 lista értékeire).

A lista típusú objektumok **alapvető, előre definiált metódusokkal rendelkeznek az elemzéshez és a manipulációhoz**: elemek hozzáadása/beszúrása, törlése/eltávolítása a listából, a lista bővítése (ezek módosítják a lista hosszát), valamint megszámlálás.

Egy elem hozzáadását a lista végéhez a **listaNeve.append(elem)** metódussal tudod megvalósítani.

Egy elem beillesztését a lista belsejébe a **listaNeve.insert(index, elem)** metódussal végezheted el.

Pl. Az L=[1,2,3,4,5] lista végéhez hozzáadjuk a 10-es értékű elemet. A 3-as indexű pozícióra pedig beillesztjük a 13-as értékű elemet.

```
L=[1,2,3,4,5,6]
L.append(10)     #L=[1,2,3,4,5,6,10]
L.insert(3,13)   #L=[1,2,3,13,4,5,6,10]
```

Egy lista bővítése során egy másik lista elemeit adjuk hozzá a lista végéhez. Ezt az extend() metódussal teheted meg: **listaNeve.extend(iterálható_objektum)**. Egy objektum akkor iterálható, ha elemenként bejárható.

```
L1=['Anna', 'Teo']
L2=['Janos', 'Daniel']
L1.extend(L2)
#L1=['Anna','Teo','Janos','Daniel']
```

A listaelemek törlése a következőképpen végezhető el:

- a **remove(elem)** metódussal, amely törli az elem első előfordulását: **listaNeve.remove(elem)**;
- a **clear()** metódussal, amely törli a lista összes elemét: **listaNeve.clear()**;
- a **pop([index])** metódussal, amely kiveszi a megadott indexű elemet a listából, és visszaadja azt későbbi felhasználásra. Ha nem adsz meg indexet, a metódus a lista utolsó elemét távolítja el és adja vissza.

Megj.

- Ha a ***pop()*** metódust egy üres listán használod, vagy egy olyan indexre hivatkozol, amely nem létezik, a Python IndexError típusú hibát fog generálni.
- Ha a ***remove()*** metódust egy üres listán használod, vagy ha az eltávolítani kívánt elem nem szerepel a listában, a Python ValueError típusú hibát fog generálni.

Egy adott értékű elem első előfordulási helyének (indexének) meghatározása az index(érték,[start, stop]) metódussal történik: **listaNeve.index(érték)**. Ha a keresett érték nem létezik, a Python ValueError típusú hibát generál, ezért használat előtt érdemes ellenőrizned az érték létezését a listában. ***Egy érték előfordulásainak megszámlálását*** a listában a **count(érték)** metódussal végezheted el.

A ***sorbarendezése*** az elemek egy adott szempont (növekvő sorrend, lexikografikus sorrend stb.) szerinti elrendezését jelenti. Ezt a ***sort()*** metódussal végezheted el, amely módosítja az eredeti listát, vagy a ***sorted()*** függvénnyel, amely egy rendezett másolatot készít és ad vissza a listáról.

```
L = [7,6,3,8,9]
L.sort()            #L=[3,6,7,8,9]
L.insert(4,5)       #L=[3,6,7,8,5,9]
L2=sorted(L)        #L2=[3,5,6,7,8,9]
```

A rendezéshez a lista összes elemének összehasonlíthatónak kell lennie egymással, különben a Python TypeError hibát generál.



A ***len()*** függvény visszaadja a listában található elemek számát: **len(lista)**.

```
L = [7,6,3,8,9]
print(len(L))    #5
```

Egy verem (stack) megvalósítása Pythonban

Egy verem megvalósításának egyik változata Pythonban a szabványos lista használata. Mivel a verem a LIFO (Last In, First Out) elvet követi, két fő műveletre van szükségünk:

- **Hozzáadás (Push)**: az .append() metódussal (a verem tetejére szúr be).
- **Törlés/Kivétel (Pop)**: a .pop() metódussal (törli és visszatérési értékként visszaadja az utolsó elemet).

Megoldott feladat: Verem (stack) adatszerkezet megvalósítása Python lista használatával.

```
verem = []          # Python lista mint verem (stack)
verem.append(5)     # Push – hozzáadás a verem tetejére
verem.append(3)
verem.append(7)
print(verem[-1])    # Top – a verem tetejének megtekintése: 7
verem.pop()         # kivétel a verem tetejéről: 7
print(verem)        # [5, 3]
```

Egy várakozási sor megvalósítása Pythonban

Egy várakozási sor megvalósításának egyik egyszerű változata Pythonban a szabványos lista használata. Mivel a várakozási sor a FIFO (First In, First Out) elvet követi, két fő műveletre van szükségünk:

- **Hozzáadás (Enqueue)**: az .append() metódussal (hozzáadja az elemet a lista végéhez).
- **Törlés/Kivétel (Dequeue)**: a .pop(0) metódussal, ahol a 0 az első elem indexe. Ez törli és visszatérési értékként visszaadja azt az elemet, amely a legrégebben van a várakozási sorban.



Várakozási sor adatszerkezet megvalósítása Python lista használatával.

```
vsor = []           # Python lista mint várakozási sor
vsor.append('Anna')  # Enqueue – hozzáadás a végéhez
vsor.append('Janos')
print(vsor[0])      # Front – az első elem: 'Anna'
vsor.pop(0)         # Dequeue – kivétel az elejéről: visszatérési érték 'Anna'
print(vsor)         # ['Janos']
```



Jegyezd meg!

Metódus	Mit csinál	Eredeti listát módosítja?	Visszatérít értéket?
append(x)	Hozzáadja az x elemet a lista végéhez.	IGEN	NEM
insert(i, x)	Beilleszti az x elemet az i. pozícióra/indexre.	IGEN	NEM
extend(L)	Hozzáadja az L lista összes elemét a végéhez.	IGEN	NEM
remove(x)	Törli az x elem első előfordulását.	IGEN	NEM
clear()	Törli az összes elemet (a lista üres lesz).	IGEN	NEM
pop(i)	Kiveszi (törli és visszatérési értékként visszaadja) az i. pozíción lévő elemet. Ha az i. hiányzik, az utolsó elemet veszi ki.	IGEN	IGEN (kivett elem)
index(x)	Visszaadja az x elem első előfordulásának helyét/indexét.	NEM	IGEN (indexet)
count(x)	Megszámolja, hányszor szerepel az x elem a listában.	NEM	IGEN (számot)
sort()	Rendezi az elemeket közvetlenül az eredeti listában.	IGEN	NEM

Feladatlap 20

Hozzátok létre a *Feladatlap20* nevű mappát. Ez a személyes portfóliód része lesz. Oldd meg az alábbi feladatokat, a válaszokat pedig jegyezzétek le egy *valaszok.rtf* fájlba.

Az alábbi 1. és 2. feladathoz válasszátok ki a helyes választ (csak egyet)

1. Adott a következő számlista: `L=[10, 20, 30, 40, 50]`. Mit fog megjeleníteni a: `print(L[1:4])`?

a. `[10,20,30,40];` **b.** `[20,30,40,50];` **c.** `[10,20,30];` **d.** `[20,30,40].`

2. Az `L=['ananasz', 'alma', 'korte' , 'szollo', 'dio']` lista esetén, mi az eredménye `L[::2]`-nek?

a. `L=['ananasz', 'korte' , 'dio'];` **b.** `L=['ananasz', 'alma', 'korte'];`

c. `L=['ananasz'];` **d.** `L=['ananasz', 'korte', 'dio' , 'korte', 'ananasz'].`

3. A Python-kód futtatása nélkül írd le, mi fog megjelenni a kijelzőn mindkét esetben:

```
szinek=['feher', 'zold']
print(szinek*2)
```

```
szamok=[10, 20, 30, 40]
szamok.pop(1)
print(szamok)
```

4. Adott az `L =['Teodora', 'Emma', 'Anna', 'Aladin', 'Mara']`. Melyek azok az utasítások, amelyekkel sorrendben elvégezhetők a következők: **a)** add hozzá a Sara nevet a lista végéhez; **b)** szúrd be a Viktor nevet a lista elejére; **c)** töröld az Emma nevet a listából; **d)** bővítsd a listát az eredeti lista első 3 szereplőjével; **e)** jelenítsd meg a végső listát.

Hozd létre a miniprojekt nevű Python-programot (a munkamappában) a következőképen:

5. Készíts egy ***miniprojektet*** egy bevásárlólista kezelésére (a lista a projekt elején jön létre) egy menü segítségével, amely a következő opciókat tartalmazza: ***a.*** lista megjelenítése; ***b.*** a felhasználót megkérdezi, mit szeretne vásárolni; ***c.*** ha a termék szerepel a listán, megjeleníti a „Termék megtalálva” üzenetet és törli a listáról, ha nem szerepel, megjeleníti a „A termék nem létezik” üzenetet; ***d.*** Megjeleníti a frissített listát; ***e.*** Kilépés. A menü ismétlődően fut a 0 billentyű megnyomásáig.

6. Tervezz egy ***mini-alkalmazást*** Pythonban, amely személyi asszisztensedként kezeli a megoldandó feladataidat. A feladatokat új feladat hozzáadásával, a teljesítettek eltávolításával, egyesek priorizálásával, a sürgős feladatok megjelenítésével, a még megoldandó feladatok listázásával stb. kell kezelned. Az alkalmazásnak addig kell futnia, amíg a felhasználó a bezárása mellett nem dönt.

Önértékelés!

Töltsétek ki a válaszokat egy *onertekeles20.docx* fájlba, az alábbi jelöléseket használva: *1 – igen, 2 – nem, 3 – nem vagyok* biztos benne. Mentsétek el a fájlt a *Feladatlap20* nevű mappába.

	A lecke végén tudom...	*Igen*	*Nem*	*Nem biztos*
1.	hogy pozitív és negatív indexeléssel is hozzá tudok férni egy lista elemeihez.			
2.	hogy a slicing (szeletelés) operátorral részlistákat emeljek ki.			
3.	hogy az in és not in operátorokkal ellenőrizzem, egy elem része-e a listának.			
4.	hogy a +, * operátorokat és a relációs operátorokat használjam listákon.			
5.	hogyan alkalmazzak alapvető metódusokat a listákon.			
6.	hogy megtaláljam egy elem pozícióját az index() metódussal, és megszámoljam az előfordulásait a count() metódussal.			
7.	hogyan rendezzek egy listát.			

8. Nehézségeim voltak a következőkben: …

Ne felejtsétek el értékelni a viselkedéseteket és a tevékenységeteket az óra során!
(a 7. oldalon található viselkedés- és tevékenységfigyelő napló segítségével)

A lecke végén így éreztem magam:

21. Lecke – A list osztály speciális metódusai és az elemek bejárása

Pythonban léteznek a list osztálynak speciális metódusai is. A ***sort() metódus*** akkor válik speciálissá, ha a key paramétert használjuk**: ID_lista.sort(key=függvény)**. Ez lehetővé teszi a személyre szabott rendezést, amely során megadhatjuk, milyen kritérium alapján rendeződjenek az elemek. *Például* egy szavakból álló listát nemcsak ábécérendben lehet rendezni, hanem a szavak hossza szerint is ***(Lista.sort(key=len))***, egy egész számokat tartalmazó listát pedig az abszolút értékük alapján ***(Lista.sort(key=abs))*** stb.

```
L = ["ananasz","dio","dinnye","alma","szolo"]
L.sort(key=len)  #L=['dio', 'alma', 'szolo', 'dinnye', 'ananasz']
L2=[3,9,-6,7,-8,5]
L2.sort(key=abs) #L2=[3,5,-6,7,-8,9]
```

A ***sort()*** metódus stabil: ha két elem a ***key*** kritérium szerint egyenlő, megőrzik eredeti egymáshoz viszonyított sorrendjüket. Hasonlóképpen, a ***sort()*** metódus használható a ***reverse*** paraméterrel is, amely **True** érték esetén a listát fordított (csökkenő) sorrendbe rendezi. A paraméter alapértelmezett értéke **False**. A paraméterek kombinálhatók is egy összetett rendezési kritérium elérése érdekében. Íme egy példa, amelyben az L lista a szavak hossza alapján, csökkenő sorrendben kerül rendezésre.

```
L1=[7,6,5,8,3,4]
L1.sort(reverse=True)          #L1=[8, 7, 6, 5, 4, 3]
L = ["ananasz","dio","dinnye","alma","szolo"]
L.sort(key=len, reverse=True) #L=[' ananasz', ' dinnye', 'szolo', ' alma', 'dio']
```

Egy lista másolása a következő módokon történhet:

- ***értékadás*** (lista1=lista2). Ebben a helyzetben nem jön létre új lista, hanem csak egy második változó (egy alias), amely ugyanarra a memóriacímre mutat[1] , mint az első lista. Így a ***lista1***-en végzett bármilyen módosítás tükröződni fog a ***lista2***-n is, és fordítva.
- A ***copy()*** metódus. Ebben a helyzetben az első listáról egy tőle független másolat jön létre.

Másolás értékadással	Másolás a copy() metódussal
`L1 = [1, 2, 3]` `L2 = L1  #Létrehozok egy aliast az L1-hez` `L2.append(4) #hozzáadok egy 4-est az egyetlen` `# objektum végéhez` `print(L1)  #L1=[1,2,3,4]` `print(L2)  #L2=[1,2,3,4]`	`L1 = [1, 2, 3]` `L2 = L1.copy()` `L2.append(4) #hozzáadok egy 4-est az L2` `# lista végéhez` `print(L1)  #L1=[1,2,3]` `print(L2)  #L2=[1,2,3,4]`

- ***másolás szeleteléssel (slicing).*** Egy új lista jön létre az első lista elemeivel**.**

Szintaxis: *uj_lista = regi_lista[start:stop].*

```
L1=[10,20,30]
L2=L1[:]
L2.append(40)
print(L1)  # [10, 20, 30]
print(L2)  #[10, 20, 30, 40]
```



Az ***identitás*** operátorral (***is***) ellenőrizhető, hogy két változó ugyanarra az objektumra mutat-e. Ez eltérően működik az ***egyenlőség operátortól*** (==). Az előbbi azt ellenőrzi, hogy a két változó ugyanarra a memóriaterületre mutat-e, míg az utóbbi azt, hogy ugyanaz-e a tartalmuk. Az is operátor gyorsabb, mivel két memóriacímet hasonlít össze, míg az egyenlőség operátor elemenként halad végig a listákon.

```
L1=[1,2,3]
L2=L1
L3=L1.copy()
print(L1 is L2) #True
print(L1 is L3) #False
```

[1] Egy memóriacímre mutatni azt jelenti, hogy a változó nem magukat az adatokat tárolja, hanem csak annak a memóriahelynek a címét, ahol azok tárolva vannak. A két változó így ugyanarra a memóriában lévő objektumra „néz”.

```
L1 = [1, 2, 3]
L2 = L1
       # 1. eset: Alias (hivatkozás ugyanarra az objektumra)
print(L1 == L2)  # True (ugyanaz a tartalmuk)
print(L1 is L2)  # True (pontosan ugyanaz az objektum a memóriában)
       # 2. eset: Másolat (különböző objektumok ugyanazzal a tartalommal)
L3 = [1, 2, 3]
print(L1 == L3)  # True (a tartalom megegyezik)
```

A lista elemeinek megfordítása a *reverse()* metódussal történik. Ezzel a módszerrel a lista nem lesz sorba rendezve, csak az elemei fordulnak meg.

```
L = [3, 1, 4, 2]
L.sort(reverse=True)
print(L)   # [4, 3, 2, 1]
```

```
L = [3, 1, 4, 2]
L.reverse()
print(L)      # [2, 4, 1, 3]
```

A listák elemeinek bejárása többféle módon történhet:

- ***elemenkénti bejárás***, akkor használjuk, amikor az elemek értéke érdekel minket, nem pedig az indexük. **Szintaxis:** *for elem in lista*:
- ***bejárás index alapján***: akkor használjuk, amikor szükségünk van az elem listában elfoglalt pozíciójára is. **Szintaxis**: *for contor in range(len(lista))*:
- haladó ***bejárás*** az ***enumerate()*** függvénnyel: a for ciklus minden lépésénél egy (index, elem) formátumú párt (tuple) ad vissza, így egyszerre biztosít hozzáférést az elem indexéhez és az értékéhez is. Íme egy összehasonlító példa egy rendezvény vendéglistájával:

```
szamok = [5,2,3,4,1]
s=0
for i in szamok:
   s=s+i
print(s)
```

Elemenkénti bejárás	Bejárás index alapján	Bejárás az enumerate() függvénnyel
`meghivottak = ["Anna", "Janos","Daniel"]` `for i in meghivottak:` `  print(i)`	`meghivottak = ["Anna","Janos", "Daniel"]` `for i in range(len(meghivottak)):` `  print(i, meghivottak[i])`	`meghivottak = ["Anna","Janos","Daniel"]` `for i,nev in enumerate(meghivottak):` `  print(i,nev)`
Kiír: **Anna** **Janos** **Daniel**	**Kiír:** **0 Anna** **1 Janos** **2 Daniel**	**Kiír:** **0 Anna** **1 Janos** **2 Daniel**

Megj. Az *enumerate()* függvény szintaxisa: **enumerate(lista, start=értek)**, ahol a start a számláló kezdőértékét jelöli; alapértelmezett értéke 0, így ebben az esetben elhagyható.

```
for i, elem in enumerate(["A", "B"]):
    print(i, elem)
# Kiír: 0 A
        1 B
```

```
for i, elem in enumerate(["A", "B"], start=1):
    print(i, elem)
# Kiír: 1 A
        2 B
```

Lista létrehozása **inicializálási sorozatokkal** (**list comprehensions**) is megvalósítható a következő szintaxissal: uj_lista = [kifejezés for elem in iterálható], ahol:

- *kifejezés az az érték, amely hozzáadásra kerül a listához*
- *elem az a változó, amely sorra veszi az iterálható objektum értékeit*
- *iterálható egy lista vagy bármilyen objektum, amelyen végig lehet haladni (iterálni lehet)*

```
# lista létrehozása - az első
# 5 nem nulla természetes szám
L = [x + 1 for x in range(5)]
print(L) #L=[1,2,3,4,5]
```

```
negyzetszamok = []  # klasszikus
for x in range(5):
   negyzetszamok.append(x**2)
#L=[0,1,4,9,16]
```

```
# List Comprehension változat
negyzetszamok = [x**2 for x in range(5)]
```

Pl. Egy lista létrehozása az első 5 négyzetszámmal.

Megj. **Egy feltétel (if) hozzáadásával** szűrheted a listába bekerülő elemeket:
[kifejezés for elem in iterálható if feltétel].

Pl. Páros számok kinyerése egy listából:

```
szamok = [1, 2, 3, 4, 5, 6, 7, 8]
paros = [n for n in szamok if n % 2 == 0]
print(paros) # Eredmény: [2, 4, 6, 8]
```

Az elem értékének egy feltétel szerinti módosítását az if-else szerkezet for ciklus elé helyezésével végezheted el.

Pl. Egy lista elemeinek módosítása „páros” vagy „páratlan” értékre, az értéküktől függően.

```
szamok = [1, 2, 3, 4, 5]
szamok = ["páros" if szamok[i] % 2 == 0 else "páratlan" for i in range(5)]
print(szamok) #szamok= ['páratlan', 'páros', 'páratlan', 'páros', 'páratlan']
```

Megj. A += operátor egyesíti az **összefűzést** a **hozzárendeléssel**, közvetlenül módosítva a meglévő listát. A + (összefűzés) operátorral ellentétben, amely egy teljesen új listát hoz létre a két lista kombinálásával, a += operátor nem foglal plusz memóriát egy köztes lista számára, így hatékonyabb.

```
szamok = [1, 2, 3]
szamok += [4, 5]
print(szamok) # Eredmény: [1, 2, 3, 4, 5]
```

```
# 1. példa: L = L + [1,2 ] (új lista)
L = [10, 20]
M = L
L = L + [1, 2]
print(L)  # [10, 20, 1, 2]
print(M)  # [10, 20]
```

```
# 2. példa: L += [1,2 ] (lista módosítása)
L = [10, 20]
M = L
L += [1, 2]
print(L)  # [10, 20, 1, 2]
print(M)  # [10, 20, 1, 2]
```

Más előre definiált függvények listákhoz:

- ***sum(lista):*** Kiszámítja az elemek összegét (csak ha numerikusak).
- ***max(lista) / min(lista):*** Visszaadja a legnagyobb, illetve a legkisebb elemet.

Gyakoriségi/előfordulási lista megvalósítása Pythonban

Egy gyakorisági lista megvalósítása Pythonban egy szabványos lista segítségével történik, amelyben az elemek listabeli pozíciói maguk a megszámolt elemek. A módszer tökéletesen működik, amikor a megszámolt elemek egész, pozitív és aránylag kicsi számok (mint például az osztályzatok 1-től 10-ig).

Pl. Beolvassuk egy osztály 30 tanulójának teszten szerzett osztályzatait. A tanár szeretné tudni, hogyan teljesítettek a diákok, és meg akarja tudni, melyik osztályzatot kapták a legtöbben.

Megj. A megoldáshoz az osztályzatokat egyenként olvassuk be, és közvetlenül egy gyakorisági listában számláljuk őket. Nincs szükség az összes osztályzat elmentésére egy listába; sokkal hatékonyabb minden egyes jegy számlálóját azonnal frissíteni, amint a tanár beírja azt a billentyűzetről. A végén végigmegyünk a gyakorisági listán, hogy megkeressük a legtöbbször előforduló osztályzatot (vagy osztályzatokat).

```
# Létrehozzuk a gyakorisági listát az osztályzatokhoz (amelyek 1-től 10-ig terjedhetnek)
# 11 elemet használunk, hogy közvetlenül tudjuk alkalmazni az indexeket 1-től 10-ig
gyakorisag = [0] * 11
print("Írjatok be a 30 jegyet egyenként:")
for i in range(30):
    jegy = int(input())
    gyakorisag[jegy] += 1            # Növeljük a beolvasott osztályzat gyakoriságát
# Megkeressük a legtöbbször előforduló jegyet (maximális gyakoriság)
max_gyakorisag = max(gyakorisag)
# Azonosítjuk, melyik jegynek felel meg ennek a maximális gyakoriságnak
print("Legnagyobb gyakorisággal megjelenő jegy: ")
for jegy in range(1, 11):
    if gyakorisag[jegy] == max_gyakorisag:
        print(jegy)
```

Megoldott feladatok:

1. Olvassunk be egy nem nulla természetes ***n*** számot, majd egy lista ***n*** darab természetes szám elemét

1. változat

```
n = int(input("Kérem az  n értékét: "))
# Az elemek beolvasása n sorból
L = []
for i in range(n):
    x= int(input())
    #minden elem után ENTER-t nyomunk
    L.append(x)
print(L)
```

2. változat

```
n = int(input("Kérem az  n értékét: "))
L = [int(x) for x in input("Kérem az elemeket:").split()]
if len(L) == n: print(L)
else: print("Nincs ", n, " eleme")
#Az elemeket egyetlen sorban, szóközökkel elválasztva adjuk meg; a bevezetett karakter-
láncot részekre bontjuk, majd ezeket az egységeket számokká alakítjuk
```

2. Jelenítsük meg a lista elemeit a beolvasás sorrendjében.

```
print(L)
```

3. Jelenítsük meg a lista elemeit a beolvasásukkal ellentétes (fordított) sorrendben.

1. változat

```
print(L[::-1])
# Létrehozza a lista teljes, de megfordított másolatát a memóriában, majd megjeleníti azt.
```

2. változat

```
for i in range(n - 1, -1, -1):
   print(L[i], end=" ")
# A lista bejárása elemenként történik, jobbról balra haladva.
```

3. változat

```
for elem in reversed(L):
   print(elem, end=" ")
# A lista bejárása elemenként történik, jobbról balra haladva, az indexek manuális keze-
lése nélkül.
```

4. Jelenítsük meg a lista páros pozícióin lévő elemeit.

```
print("A páros pozíción lévő elemek:", L[0::2])
# Szeletelést (slicing) használunk 2-es lépésközzel: L[start:stop:pas]
```

5. Jelenítsük meg a páratlan pozíciókon található páros elemeket.

1. változat

```
L2 = [L[i] for i in range(1, n, 2) if L[i] % 2 == 0]
print("A páratlan pozíciókon található páros elemek:", L2)
# List comprehension-t használunk
```

2. változat

```
L2=[]
for i in range(1, n, 2):
   if L[i] % 2 == 0:
     L2.append(L[i])
print("A páratlan pozíciókon található páros elemek:", L2)
# Az 1. indextől indulunk n-ig, 2-es lépésközzel.
```

6. A listában található páratlan elemek száma.

```
L3 = [x for x in L if x % 2 != 0]
print("A páratlan elemek száma:", len(L3))
# Létrehozunk egy új listát az első lista páratlan elemeiből.
```

7. Beolvasunk egy ***x*** természetes számot. Jelenítsük meg a listában szereplő, ***x***-szel egyenlő elemek számát.

```
# A count() metódust használjuk
x = int(input("x= "))
db_elofordul = L.count(x)
print(f"Az {x} elemmel egyenlő elemek száma {db_elofordul}.")
```

8. Rendezzük növekvő sorrendbe csak a páratlan számokat, a párosak pedig maradjanak az eredeti helyükön.

Pl.

Input: [8, 3, 2, 9, 7, 1, 6] Output: [8, 1, 2, 3, 7, 9, 6]

Megoldási ötlet: **1. lépés**: Létrehozunk egy új listát az összes páratlan számmal; **2. lépés**: Rendezzük a páratlan számok listáját; **3. lépés**: Végigmegyünk az eredeti listán, és valahányszor egy páratlan szám által elfoglalt helyet találunk, a helyére a rendezett lista következő számát tesszük.

```
L=[8,3,2,9,7,1,6]
paratlan = [x for x in L if x % 2 != 0]
paratlan.sort()
# Visszahelyezzük a rendezett elemeket az eredeti L listába
index_paratlan = 0
for i in range(len(L)):
    if L[i] % 2 != 0:
       L[i] = paratlan[index_paratlan]
       index_paratlan += 1
print(L)
```

9. Írjunk egy olyan függvényt, amely paraméterként egy listát kap, majd kiszámítja és visszatéríti a listában szereplő összes páratlan szám összegét. Ha nincsenek páratlan elemek, térítsen vissza ***-1*** értéket.

```
def paratlan_szamok_osszege (Lista):
    osszeg = 0
    talalt = False      # Kezdetben feltételezzük, hogy nincsenek páratlan számaink
    for szam in Lista:
        if szam % 2 != 0:
            osszeg += szam
            talalt = True      # Találtunk legalább egyet!

    if not talalt:
        return -1

    return osszeg

# A függvényhívás
L1 = [2, 4, 5, 6, 7]
print("A páratlan számok összege:", paratlan_szamok_osszege(L1))
```

Feladatlap 21

Készítsetek egy *Feladatlap21* nevű mappát. Oldjátok meg az alábbi feladatokat, a helyes válaszokat írjátok egy *valaszok.docx* dokumentumba.

Az alábbi 1-es és 2-es feladatnál válasszátok ki a helyes választ.

1. Melyik az a változó az alábbi szerkezetben, amelyik sorban felveszi a lista egyes elemeinek értékét?

a. `print`; **b.** `in`; **c.** `x`; **d.** `szamok`.

```
szamok = [10, 20, 30]
for x in szamok:
    print(x)
```

2. Mit ír ki a program az alábbi Python-kód futtatása után?

a. `['alma', 'körte', 'szőlő']` és `True`;

b. `['ananász', 'alma', 'körte']` és `False`;

c. `['alma', 'körte', 'szőlő']` és `False`;

d. `['alma', 'banán', 'szőlő']` és `True`.

```
L=['ananász', 'alma', 'körte', 'szőlő', 'dió']
L_masolat = L[1:4]
L[2] = 'banán'
print(L_masolat)
print(L[2] == L_masolat[1])
```

3. Az alábbi Python-kód futtatása nélkül írd le, mi fog megjelenni a képernyőn a két eset mindegyikében:

```
A = [1, 2, 3]
B = A[:]
print(A is B)
print(A == B)
```

```
L1 = [0]
L2 = L1 * 2
print(L2 == [0, 0])
```

4. Adott a `gyumolcsok = ["alma", "körte"]` lista és az `uj_lista = gyumolcsok`. Ezután módosítjuk az `uj_lista[0] = "banán"` elemet. Melyik gyümölcs lesz a *gyumolcsok* lista első pozícióján?

5. Adott egy bulin részt vevő személyek listája az érkezésük sorrendjében: `L=['Teo', 'Emma', 'Anna', 'Alfi', 'Mara']`. Írd meg a *buli.py* Python-programot (a Feladatlap21 nevű mappába), amely a fenti lista alapján megvalósítja a következőket:

a. *a meghívottak megjelenítése az érkezésük sorrendjében*;
b. *a meghívottak megjelenítése az érkezésükkel ellentétes sorrendben*;
c. *a meghívottak megjelenítése ábécésorrendben, az eredeti lista módosítása nélkül*;
d. *az első 2 és az utolsó 2 meghívott megjelenítése*.

6. Billentyűzetről beolvasunk egy ***n*** természetes számból álló listát (n nem nulla, természetes szám). Írj egy ***szamok.py*** nevű Python-programot, amely meghatározza a következőket: **a.** *a páros pozíciókon lévő elemek egyes helyiértékén álló számjegyeinek összegét*; **b.** *az x-szel osztható elemek számát, ahol x egy billentyűzetről beolvasott, nullától különböző természetes szám*; **c.** *a páratlan pozíciókon lévő páratlan elemek szorzatát*; **d.** *a listában lévő, az utolsó elemnél nagyobb elemek számát*; **e.** *az első páros elem pozícióját és az utolsó páratlan elem pozícióját*.

7. Billentyűzetről beolvasunk egy n természetes számból álló listát (n nullától különböző természetes szám), amely egy osztály tanulóinak egy teszten elért osztályzatait reprezentálja, a tanár által az osztályban lévő azonosítószámaik (1, 2...n) sorrendjében megadva. Melyek azoknak a tanulóknak az azonosítószámai, akiknek az osztályzata az osztályátlag felett van? Írjatok egy ***jegyek.py*** nevű Python-programot.



Önértékelés!

Töltsétek ki a válaszaitokkal egy *onertekeles21.docx* nevű fájlba, a következő jelöléseket használva: *1– igen, 2 – nem, 3 – nem vagyok biztos benne*. Mentsétek a fájlt a *Feladatlap21* nevű mappába.

	A lecke végén tudom...	***Igen***	***Nem***	***Nem biztos***
1.	hogyan kell bejárni egy listát Pythonban.			
2.	hogy hogyan használjak fejlett módszereket a listák rendezésére, másolására, megfordítására és létrehozására.			
3.	a különbséget a hozzárendeléssel történő másolás (alias) és a copy() függvénnyel vagy szeleteléssel (slicing) végzett független másolás között.			
4.	hogy hogyan fordítsam meg egy lista elemeit a reverse() metódussal.			
5.	hogy hogyan oldjak meg egyszerű/összetett feladatokat listákkal.			

6. Nehézségeim voltak a következőkben: …

Ne felejtsétek el értékelni a viselkedéseteket és a tevékenységeteket az óra során!
(a 7. oldalon található viselkedés- és tevékenységfigyelő napló segítségével)

A lecke végén így éreztem magam:

5. Fejezetvégi felmérés



Ellenőrizd a tudásodat!

(Pontszám: 90 pont + 10 pont hivatalból)

Követelmény: Oldjátok meg az alábbi feladatokat a ***tudas_ellenorzes5.docx*** fájlban. Az összes létrehozott fájlt mentsétek a ***Fejezet5*** mappába.

I. Válaszd ki azt a leírást, amely a legjobban meghatározza a Python nyelv standard listáját:

a. Azonos típusú elemek fix, később nem módosítható gyűjteménye.

b. Egy lineáris, rendezett, módosítható struktúra, amelynek elemei index alapján azonosíthatók.

c. Fix elemszámú verem.

d. Egy speciális függvény a fordított sorrendben bevitt számok hozzáadására és törlésére.

II. Az alábbiakban egy diák által írt két kódrészlet látható, amelyek mindegyike tartalmaz egy-egy, a program futását megakadályozó hibát. Mindkét esetre: **a.** Azonosítsd a hibát generáló kódsort; **b.** Nevezd meg a létrejött hiba típusát (**IndexError** vagy **ValueError**); **c.** Írd le a kód helyes változatát.

1. eset: Osztálynapló

A diák egy tanuló jegyeit tartalmazó lista utolsó elemét szeretné megjeleníteni a lista hosszának (n) felhasználásával.

```
jegyek = [9, 10, 8, 7, 10]
n = len(jegyek) # n 5 lesz
# A diák az utolsó jegyet szeretné kiíratni
print("Az utolsó jegy a naplóban:", jegyek[n])
```

2. eset: Egy vendég eltávolítása

A diák egy projektben részt vevő meghívottak listájával rendelkezik, és a .remove(). metódus használatével szeretné eltávolítani „Mihaly”-t a listából

```
meghivott_projekt=["Anna","Dani","Etelka"]
meghivott_projekt.remove("Mihaly")
print("A lista:", meghivott_projekt)
```

III. Párosítsd a bal oldali oszlopban leírt helyzetet a jobb oldali oszlopban található szintaktikai elemmel:

Helyzet / Leírás	Szintaktikai elem Pythonban
1. Andrei ellenőrzi, hogy két lista ugyanazt a RAM-területet foglalja-e el.	a. A .sort(key=...) metódust fogja használni.
2. Maria kiíratja a lista elemeit az indexükkel együtt.	b. A sorted() beépített függvényt fogja használni.
3. Róbert a listát közvetlenül módosítva, az abszolút érték szerint rendezi.	c. Az is operátort fogja használni.
4. Tibor egy rendezett listát kap meg anélkül, hogy az eredeti listát módosítaná.	d. A += operátort fogja használni.
5. Johanna egy lista elemeit egy másik lista végéhez adja hozzá.	e. Az enumerate() függvényt fogja használni.

IV. Olvassunk be egy ***n*** természetes számot és ***n*** darab egész számot (egy listát az n nap alatt regisztrált vízfogyasztással). A negatív vagy 100-nál nagyobb értékek hibás adatok. Írjál egy viz.py nevű programot, amely: ***a.*** megjeleníti a hibás napokat vagy a „ *Helyes adatok*” üzenetet; ***b.*** létrehoz egy listát a helyes adatokkal, és ez alapján kiszámítja az átlagfogyasztást vagy a „ *Nincsenek helyes adatok*” üzenetet, valamint azon napok számát, amelyeken a fogyasztás meghaladta az átlagot.

V. Egy üzletben a termékeket az eladásuk sorrendjében rögzítik egy listában. Beolvasunk egy n számot és egy listát ***n*** darab termékkóddal (1 és 1000 között), valamint egy x kódot. Írjál egy ***uzlet.py*** nevű programot, amely meghatározza a következőket: ***a.*** a listában szereplő különböző (egyedi) termékek számát, ***b.*** az x első előfordulási pozícióját (vagy a „Nincs vásárolva!” üzenetet), ***c.*** hányszor vásárolták meg az x terméket, ***d.*** a nem eladott termékek kódjait növekvő sorrendben, ***e.*** az egyedi termékek kódjait az eladott darabszám szerinti csökkenő sorrendben, indokolva az adatszerkezetek és a használt metódusok megválasztását.

Értékelési módszer (100 pont) hivatalból – **10p, I. b) feladat (5p), II. feladat – 30p (1. eset – 15p, 2. eset – 15p, mindegyik esetre: a) 5p, b) 5p, c) 5p), III. feladat – 5 x 3p = 15p (1-c, 2-e, 3-a, 4-b, 5-d), IV. feladat – 20p (a – 8p, b – 12p), V. feladat – 5 x 4p = 20p (a – 8p, b –12p), V. feladat – 5 × 4p = 20p**.



Projekt

(100 pont)

„Algoritmusok a hullámvasúton”

Készítsetek egy Python-projektet ***park.py*** néven egy vidámpark számára, a list osztály felhasználásával. Kezeljétek a látványosságokat közvetlen és szekvenciális hozzáféréssel, szervezzétek meg a beszállást a hullámvasútra egy várakozási sor (queue) segítségével, rendezzétek a szuvenírraktárat egy verem (stack) alkalmazásával, és valósítsatok meg egy gyakorisági listát a vélemények elemzéséhez. Mutassátok be a projektet az osztály előtt.

Értékelési módszer (**100 pont**) **10p**–A látványosságok listájának kezelése, **10p**–A várakozási sor működése, **10p**–A verem működése, **10p**–Gyakorisági lista, **10p**–A kód minősége, **10p**–A választott modellek elméleti helyessége, **10p**-A list osztály előnyeinek/hátrányainak elemzése, **10p**–Az előadás világossága és a pontos informatikai szakkifejezések használata. **10p**–A csapattagok egyenlő részvétele a bemutatóban, **10p**- kreativitás.

(**Ajánlások**: 2-3 fős csapatokban dolgozzatok. Minden csapat pontozza a többi csapatot egy értékelőlap alapján, a végső pontszám pedig a kapott pontszámok számtani közepe lesz).



Kutass!

(100 pont)

Esettanulmány: Hogyan állunk az idővel?

Dokumentáljátok a ***ido.docx*** fájlban a következő helyzetet: egy kiszállítási platformhoz naponta több tízezer rendelés érkezik. Minden új rendelés bekerül a listába, a teljesített rendeléseket pedig eltávolítják. Ezen műveletek sebessége létfontosságú. Használjatok különböző forrásokat, és végezzétek el az alábbi feladatokat:

- Fedezzétek fel, hogyan lehet mérni egy Python-kód futási idejét. **(20p)**
- A felfedezett modul használatával mérjétek le és hasonlítsátok össze az ***append()*** vs ***insert(0,x)***, valamint a ***pop()*** vs ***pop(0)*** műveletek futási idejét 100, 10 000 és 100 000 rendelést tartalmazó listák esetén. Az eredményeket mutassátok be egy táblázatban. **(20p)**
- Szimuláljatok két struktúrát ugyanazzal a 100 000 rendelési ID-val: egy listát és egy szótárt (dictionary). Mérjétek meg, mennyi ideig tart egy, a struktúra végén található ID megkeresése mindkét esetben (az in operátorral). Ismételjétek meg a mérést 100 és 10 000 elemre is, majd jegyezzétek fel a kapott adatokat a táblázatba. Magyarázzátok meg a megfigyelt különbséget. **(20p)**
- Írjátok le 4–6 sorban, hogy milyen struktúrát és műveleteket ajánlanátok a platformnak. **(10p)**
- Keressetek a Python dokumentációban egy olyan, az órán eddig nem használt metódust vagy függvényt, amely hasznos lehet a rendelések listájához (pl. filter(), zip()). **(10p)** Írjátok le, mit csinál, és mutassátok be egy konkrét példával. **(10p)** Oldjatok meg vele egy problémát a fenti rendelési listára vonatkozóan (pl. zóna szerinti szűrés). **(10p)**



Portfólió

(100 pont)

A fejezet végén a tanuló személyes portfóliójának a ***Fejezet5*** nevű mappában a következő anyagokat kellene tartalmaznia: a ***Feladatlap18*** mappát (a ***valaszok.rtf, p6.py, p7.py, projekt.docx, onertekeles18.docx*** fájlokkal), a ***Feladatlap19*** mappát (a ***valaszok.docx, onertekeles.docx*** fájlokkal), a ***Feladatlap20*** mappát (a ***valaszok.rtf, miniprojekt.py, miniaplikacio.py, onertekeles.docx*** fájlokkal), a ***Feladatlap21*** mappát (a ***valaszok.rtf, buli.py, szamok.py, jegyek.py*** fájlokkal), a ***Projekt5*** mappát a ***park.py*** és ***ido.py*** fájlokkal és az ***onertekeles.docx*** fájlt.

Értékelési mód:

- Megvan a ***Fejezet5*** mappa a megadott könyvtárszerkezettel – **20p**
- Megvan:
  - A fájlok több mint 95%-a – **80p**
  - A fájlok 75–94%-a – **60p**
  - A fájlok 50-74%-a – **40p**
  - A fájlok 25-49%-a – **20p**
  - 25% alatt – **0p**

6. Fejezet – Haladó megoldási stratégiák. (Egy lista elemeinek szisztematikus előállítási módszerei)

22. Lecke – Az algoritmusok hatékonysága: végrehajtási idő, memória és az O(n) jelölés

„Egy jó program nemcsak helyes, hanem hatékony is."

Megj. Az algoritmusok hatékonyságának fontossága. Az informatikában egy feladat helyes megoldása csupán az első lépést jelenti. Ugyanolyan fontos az is, hogy hogyan érjük el az eredményt a futási idő és a felhasznált erőforrások szempontjából.

<def/> Egy ***hatékony algoritmus*** lehetővé teszi nagy mennyiségű adat gyors feldolgozását, és hozzájárul a hatékony alkalmazások fejlesztéséhez. A gyakorlatban két algoritmus közötti különbségek akkor válhatnak jelentőssé, amikor az adatok mennyisége növekszik. Egy algoritmus teljesítményének elemzésekor két fő szempontot veszünk figyelembe:



- **Futási idő:** Az algoritmus teljes végrehajtásához szükséges időtartamot jelenti. Ezt befolyásolja az elvégzett műveletek száma és a bemeneti adatok mérete.
- **Felhasznált memória:** Az algoritmus végrehajtása során lefoglalt memóriaterületet jelenti. Ez magában foglalja a változóknak, a használt adatszerkezeteknek szükséges tárhelyet és az esetleges segédtárhelyet.

<def/> A ***hatékonyság mérése***. Egy algoritmus hatékonyságának értékelése nem a másodpercben mért futási idő alapján történik, mivel ez függ a programot futtató rendszer teljesítményétől. Az elemzés az elvégzett műveletek száma alapján történik, a bemeneti adatok méretéhez viszonyítva, amelyet általában n-nel jelölünk.

A nagy O jelölés (Big O)

<def/> Az algoritmus futási idejének növekedési módját **O (Big O)** jelölés segítségével írjuk le. Ez mutatja a műveletek számának növekedési ütemét a bemeneti adatok méretének függvényében. Segítségével összehasonlíthatunk különböző algoritmusokat, és kiválaszthatjuk a leghatékonyabb változatot.

Gyakori bonyolultság típusok:

Jelölés	Elnevezés	Jelentés
O(1)	konstans idő	A végrehajtási idő nem függ n-től
O(n)	lineáris idő	A végrehajtási idő n-nel arányosan növekszik
O(n^2)	négyzetes idő	nagy n értékek esetén gyorsan növekszik
O(n^3)	köbös idő	nagy n értékek esetén nagyon gyorsan növekszik
O(n^n)	szuper-exponenciális idő	már kis n értékek esetén is robbanásszerűen növekszik
O(log n)	logaritmikus idő	lassan növekszik, a hatékony algoritmusokra jellemző

Megjegyzés: *Az **O(log n)** bonyolultságú algoritmusok egyik sajátos példája a **bináris keresés**, amely minden lépésben megfelezi a keresési tartományt — ahelyett, hogy minden elemet megvizsgálna, minden egyes összehasonlításnál kizárja az elemek felét a további keresésből. Ezt az algoritmust a következő leckékben fogjuk tanulni.*

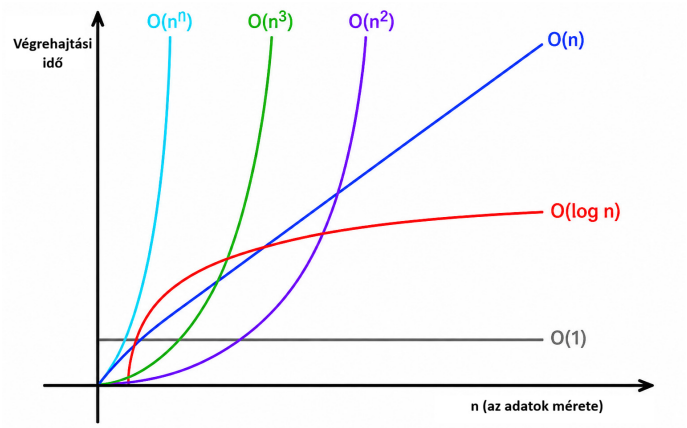
A bonyolultságok sorrendje. Hatékonyság szempontjából a bonyolultságok a következőképpen rendezhetők (a leghatékonyabbtól a leglassúbbig):

$$O(1) < O(\log n) < O(n) < O(n^2) < O(n^3) < O(n^n)$$

Megj. Megfigyelhetjük, hogy a bonyolultság növekedésével az algoritmus egyre lassúbbá válik és egyre nehezebben alkalmazható nagy adatmennyiségek esetén.

A ***bonyolultság grafikus ábrázolása:*** A függvények grafikus ábrázolása jól kiemeli a köztük lévő különbségeket: egyesek lassan növekednek (logaritmikusan), míg mások nagyon gyorsan (négyzetesen vagy exponenciálisan).

Megfigyelhetjük, hogy a bonyolultságok közötti különbségek nagy ***n*** értékek esetén válnak jelentőssé.



Python példák

Pl. A következőkben bemutatunk néhány Python nyelven írt algoritmust, azzal a céllal, hogy rávilágítsunk, hogy hogyan határozhatjátok meg a bonyolultságot a használt utasítások szerkezete alapján.

Egy lista bejárása: Feladat: Adott egy természetes számokat tartalmazó lista. Írasd ki a lista összes elemét!	Egymásba ágyazott ciklusok Feladat: Adott egy n természetes szám. Írasd ki az összes (i, j) alakú párt, ahol az i és j 0-tól n−1-ig kap értéket.	Hozzáférés az elemekhez Feladat: Adott egy számlista és egy k sorszám. Írasd ki a lista k. sorszámú elemét.
`v = [1, 2, 3, 4, 5]` `for x in v:` `  print(x)`	`n = 5` `for i in range(n):` `  for j in range(n):` `  print(i, j)`	`# beolvassuk a k pozíciót` `k = int(input())` `v = [10, 20, 30]` `print(v[k])`
Bonyolultság: **O(n)** Minden elem csak egyszer lesz feldolgozva.	Bonyolultság: $O(n^2)$ A műveletek száma nagyon gyorsan növekszik.	Bonyolultság: **O(1)** A hozzáférés azonnali.

<def/> *Elemzés.* Egy algoritmus a bemeneti adatoktól függően eltérően viselkedhet. Három esetet elemzünk:

- **kedvező eset** – a lehető leggyorsabb forgatókönyv
- **átlagos eset** – szokásos viselkedés
- **kedvezőtlen eset** – a lehető leglassabb forgatókönyv

Pl. **A lineáris keresés eseteinek elemzése:** *Adott egy egész számokból álló lista és egy x érték. Határozzuk meg, hogy az x érték szerepel-e a listában.*

Kedvező eset	Átlagos eset	Kedvezőtlen eset
• v = [7, 2, 9, 1, 4] • **x=7** • Az első elem → 1 összehasonlítás	• v = [7, 2, 1,9, 4] • **x=1** • A középső elem → ~ n/2 összehasonlítás	• v = [7, 2, 9, 1, 4] • **x=5** • Nem létező elem → n összehasonlítás

Kedvező eset. Az az eset, amikor a keresett elem az első helyen van. Példa: x = 7. Közvetlenül az első elemmel hasonlítja össze. A keresett elem azonnal megvan. Az összehasonlítások száma **1**. Az algoritmus nagyon gyorsan leáll.

Átlagos eset. Az az eset, amikor a keresett elem valahol a lista belsejében található. Példa: *x = 1*. Több elemet is ellenőrizni kell, amíg megtalálja. Átlagosan az elem a lista közepénél lesz meg. Az összehasonlítások száma megközelítőleg n / 2. A futási idő közepes.

Kedvezőtlen eset. Az az eset, amikor az elem nem szerepel a listában, vagy az utolsó helyen van. Példa: x = 5. A teljes listát leellenőrzi. A keresett elem nem található a listában. Az összehasonlítások száma n, és az algoritmus az összes lehetséges ellenőrzést elvégzi.

Következtetés: A lineáris keresés bonyolultsága **O(n)** — kedvezőtlen esetben az algoritmus a teljes listát bejárja. A hatékonyság elemzését általában a **kedvezőtlen esetre** végezzük el, mivel az nyújt megbízható becslést az algoritmus teljesítményéről.

Pl. Adott egy egész számokat tartalmazó lista. Ellenőrizzük, hogy vannak-e benne ismétlődő értékek, azaz két vagy több olyan elem, amelynek ugyanaz az értéke.

Nem hatékony változat – O(n²)

```
v = [1, 2, 3, 2]
# feltételezzük, hogy nincsenek is
# métlődő értékek
ok = False
for i in range(len(v)):
    # összehasonlítjuk a következő elemekkel
for j in range(i + 1, len(v)):
        if v[i] == v[j]:
        # találtunk két azonos értéket
        ok = True
# True-t ír ki ismétlődő értéknél
print(ok)
```

Minden elemet összehasonlítunk az összes többivel

Hatékony változat – O(n)

```
v = [1, 2, 3, 2]
# a már előfordult értékek halmaza
s = set()
# feltételezzük, hogy nincs ismétlődés
ok = False
for x in v:
    if x in s:
      # találtunk egy ismétlődő értéket
      ok = True
      break
    # hozzáadjuk az elemet a halmazhoz
    s.add(x)
# True-t ír ki ismétlődő értéknél
print(ok)
```

Használunk egy halmazt a már előfordult értékekhez

A módszerek összehasonlítása. Az első változatban minden elemet összehasonlítunk az ***összes többivel → sok művelet***. A második változatban minden elemet ***csak egyszer ellenőrzünk → kevesebb művelet***. Ezért a második módszer sokkal hatékonyabb, különösen nagy listák esetén.

Megj. Bár mindkét módszer helyesen oldja meg a feladatot, a teljesítménybeli különbség akkor válik jelentőssé, amikor az elemek száma nagy.

Megoldott feladatok:

1. Feladat: *Határozd meg a következő kódrészletek bonyolultságát:*

a)
```
v = [10, 20, 30, 40, 50]
print(v[len(v) - 1])
```

b)
```
v = [3, 1, 7, 2, 5]
maxim = v[0]
for x in v:
    if x > maxim:
        maxim = x
print(maxim)
```

c)
```
n = 5
for i in range(1, n+1):
    for j in range(1, n+1):
        print(i * j)
```

Várható válaszok:

a) kódrészlet: kiírja egy lista utolsó elemét.

Bonyolultság: **O(1)** — a hozzáférés azonnal megtörténik a lista hosszától függetlenül.

b) kódrészlet: kiírja egy lista legnagyobb elemét.

Bonyolultság: **O(n)** — minden elemet pontosan egyszer dolgoz fel.

c) kódrészlet: kiírja az összes i×j szorzatot, ahol **i** és **j** **1**-től **5**-ig kap értéket

Bonyolultság: **O(n²)** – minden **i** értékre sorra veszi a **j** összes értékét.

2. Feladat: *Adott egy egész számokat tartalmazó lista. Az elemek összegének kiszámítására két változatot javaslunk. Melyik változat hatékonyabb a bonyolultság szempontjából? Válaszodat indokold.*

A változat	**B változat**
egy **for** ciklus, amely bejárja a listát és összeadja az elemeket	a beépített **sum(lista)** függvény használata
`v = [1, 2, 3, 4, 5]` `osszeg = 0` `for x in v:` `    osszeg += x` `print(osszeg)`	`v = [1, 2, 3, 4, 5]` `osszeg = sum(v)` `print(osszeg)`
Bár a **B** változat kódja rövidebb, a **sum()** függvény belsőleg ugyanúgy sorra veszi az összes elemet, pontosan úgy, mint a for ciklus, így a bonyolultság mindkét esetben **O(n)**.	



Jegyezd meg!

- ✔ Egy hatékony algoritmus optimálisan használja az időt és a memóriát;
- ✔ A bonyolultság az algoritmus teljesítményét írja le;
- ✔ Az algoritmusok közötti különbségek nagy adatmennyiség esetén válnak jelentőssé;
- ✔ A legkedvezőtlenebb (leglassabb) eset érdekel minket leginkább;
- ✔ A megfelelő módszer kiválasztása alapvető fontosságú.



Feladatlap 22

Hozzatok létre egy *Feladatlap22* nevű mappát. Oldjátok meg az alábbi feladatokat, és írjátok be a helyes válaszokat egy *valaszok.docx* nevű fájlba.

I. *Keresse meg a helyes választ:*

1. Az O(n) bonyolultság azt jelenti, hogy:
- ☐ az idő nem függ n-től
- ☐ az idő arányosan növekszik n-nel
- ☐ az idő nagyon gyorsan növekszik

2. Melyik a hatékonyabb bonyolultság?
- ☐ $O(n^2)$
- ☐ O(log n)
- ☐ $O(n^3)$

3. A nagy O jelölés a következőket jelenti:
- ☐ a pontos futási időt
- ☐ az n függvényében kifejezett műveletek számát
- ☐ a felhasznált memóriát

4. A helyes összefüggés:
- ☐ $O(n^2) < O(n)$
- ☐ $O(n) < O(n^2)$
- ☐ $O(n^3) < O(\log n)$

II. *Töltsd ki:*

- Az O(1) bonyolultság ..... időtartamot jelent.
- Az $O(n^2)$ bonyolultság akkor jelentkezik, amikor ..... .
- Az elemzést általában a(z) ..... esetre végzik.

III. *Határozd meg a bonyolultságot:*

```
for i in range(n):
    print(i)
```

Válasz: .....

```
for i in range(n):
    for j in range(n):
        print(i, j)
```

Válasz: .....

IV. ***Válaszoljatok a következő kérdésekre***

1. Magyarázzátok el: Mit jelent egy algoritmus hatékonysága?

2. Indokoljátok meg: Miért válik az $O(n^2)$ hatékonyság nagy n esetén hatástalanná?

3. Elemzétek:

Mi a különbség a következők között:

- a két ciklusos módszer
- a halmazos módszer

V. ***Alkalmazás***

Adott a következő lista:

v = [1, 2, 3, 2]

Követelmény:

- ellenőrizzétek, hogy vannak-e duplikált elemek
- írjatok két függvényt
- adjátok meg a komplexitást

FELADAT – Esettanulmány – Algoritmusok hatékonyságának összehasonlítása

Leírás: Készítsetek egy esettanulmányt, amelyben összehasonlít két, ugyanazon probléma megoldására szolgáló módszert: egy $O(n^2)$ bonyolultságú és egy $O(n)$ bonyolultságú módszert.

Teendők:

Válasszatok egy egyszerű feladatot (például: ellenőrizze, hogy vannak-e duplikált értékek egy listában).

Írjatok két megoldási változatot:

- a nem hatékony változatot (két ciklussal)
- a hatékony változatot (halmazzal)

Tesztelje az algoritmusokat a következőkre:

- n=10
- n=100
- n=1000

Jegyezd fel: a műveletek hozzávetőleges számát vagy a végrehajtási időt (ha tudod, hogyan kell mérni).

Önértékelés!

Töltsétek ki a válaszaitokkal egy *onertekeles22.docx* nevű fájlba, a következő jelöléseket használva: *1 – igen, 2 – nem, 3 – nem vagyok biztos benne*. Mentsétek a fájlt a *Feladatlap22* nevű mappába.

	A lecke végén tudom ...	***Igen***	***Nem***	***Nem biztos***
1.	mi az algoritmus hatékonysága.			
2.	meghatározni egy Python kódrészlet bonyolultságát.			
3.	mi a Big O jelölés és mit ír le.			
4.	a bonyolultsági osztályokat a leghatékonyabbtól a leglassabbig sorba rendezni.			
5.	elmagyarázni a legjobb, átlagos és legrosszabb eset közötti különbséget.			
6.	két megoldási lehetőség közül a bonyolultságuk alapján választani.			

7. Nehézségeim voltak a következőkben: …

Ne felejtsétek el értékelni a viselkedéseteket és a tevékenységeteket az óra során!
(a 7. oldalon található viselkedés- és tevékenységfigyelő napló segítségével)

A lecke végén így éreztem magam:

23. Lecke – Szisztematikus előállítás: előző tagok alapján felépülő sorozatok és adott tulajdonságú szekvenciák

Megj. Az informatikában gyakran adódnak olyan helyzetek, amikor lépésről lépésre kell felépítenünk egy értéksorozatot, egy adott szabályt követve. Ezt a folyamatot ***szisztematikus előállításnak***. nevezzük. Az egyetlen elem elszigetelt kiszámításával ellentétben a generálás az első tagtól a kívánt tagig a **teljes sorozatot** előállítja.

<def/> Egy ***szekvencián*** (***sorozaton***) értékek rendezett listáját értjük: $a_1, a_2, a_3, \ldots, a_n$. A sorozat minden értékét **tagnak** nevezzük. *Példák* gyakori sorozatokra:

- 1, 2, 3, 4, 5 (egymást követő természetes számok)
- 2, 4, 6, 8, 10 (páros számok)
- 0, 1, 1, 2, 3, 5, 8 (Fibonacci-sorozat)
- 1, 4, 9, 16, 25 (négyzetszámok)



A ***szisztematikus generálás módszerei.*** Egy lista elemeinek előállítására három fő módszer létezik, amelyek mindegyike egy sajátos feladattípushoz igazodik:

Módszer	Leírás	Példa
1. Képlet alapján	Minden tagot közvetlenül egy olyan matematikai kifejezésből számítunk ki, amely függ a sorszámtól.	an = 2·n → 0, 2, 4, 6, …
2. Tulajdonságok alapján	Minden lehetséges jelöltet ellenőrzünk; csak az kerül be a sorozatba, amelyik megfelel egy adott feltételnek.	prímszámok, négyzetszámok
3. Egymásra épüléssel	Minden tag a sorozat előző tagjaitól függ. Nem számítható ki közvetlenül.	Fibonacci, kamatos kamat

<def/> ***1. módszer:*** Előállítás képlet alapján. Ha az ***n*** sorszám és a tag értéke között közvetlen matematikai kapcsolat van, akkor minden elem kiszámítható a többi tagtól ***függetlenül***, az előző értékek ismerete nélkül. A ciklus végigmegy az indexeken, és alkalmazza a képletet.

A 2 többszörösei $a_n = 2 \cdot n$	Négyzetszámok $a_n = n^2$	Az első n tag összege $S_n = n \cdot (n+1)/2$
`n = int(input())` `for i in range(n):` `  print(2 * i)`	`n = int(input())` `for i in range(1, n+1):` `  print(i * i)`	`n = int(input())` `for i in range(1, n+1):` `  print(i*(i+1)//2)`

<def/> ***2. módszer: Előállítás adott tulajdonságok alapján.*** A természetes számok halmazából néha csak azokat a számokat szeretnénk kiválasztani, amelyek megfelelnek egy adott feltételnek (tulajdonságnak). Az algoritmus megvizsgálja a jelölteket és csak azokat veszi be a sorozatba, amelyek megfelelnek a feltételnek.

Páros számok Feltétel: i % 2 == 0	Prímszámok Feltétel: ha nincs osztója	A 3 vagy 5 többszörösei Két feltétel vagy kapcsolattal
`n = int(input())` `for i in range(n):` `  if i % 2 == 0:` `  print(i)`	`n = int(input())` `for i in range(n):` `  if i >= 2:` `  prim_e = True` `  for d in range(2, i):` `  if i % d == 0:` `  prim_e = False` `  break` `  if prim_e:` `  print(i)`	`n = int(input())` `for i in range(n):` `  if i%3==0 or i%5==0:` `  print(i)`

<def/> ***3. módszer: Előző tagok alapján felépülő sorozatok.*** Az előző tagok alapján felépülő ***sorozat*** egy olyan sorozat, amelyben minden tag egy vagy több előző tagtól függ. A képlettel előállított sorozattal ellentétben, ebben az esetben a tag ***nem számítható ki*** az előző értékek ismerete nélkül. Az előző tagok alapján felépülő sorozat leírásához az alábbiak szükségesek:

- ***Kezdőtagok (inicializálási feltételek)*** – explicit módon megadott értékek (a_0, a_1, …).
- ***Egymásra épülő reláció*** – az aktuális tagot az előző tagokkal összekapcsoló képlet.

<def/> ***Elsőrendű egymásra épülés.*** Minden tag az őt közvetlenül megelőző tagtól függ. Az aktuális állapot tárolásához egyetlen változó szükséges.

Pl. Tekintsünk egy olyan sorozatot, amelynek első tagja 3, és minden további tagot úgy kapunk meg, hogy az őt megelőző tagot megszorozzuk 2-vel. Írassuk ki a sorozat első n tagját!

Jelölés: $\mathbf{a_1 = 3}$ a kezdőtag, $\mathbf{a_n = 2 \cdot a_{(n-1)}}$ az egymásra épülő reláció, $n \geq 2 \rightarrow$ Az előállított számsor: 3, 6, 12, 24, 48, …

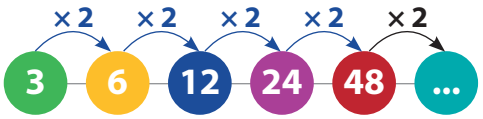
```
# beolvassuk a tagok számát
n = int(input("Tagok száma: "))
# a sorozat első tagja (a1 = 3)
a = 3  # kiírjuk a kezdőtagot
print (a, end=" ")
for i in range(1, n)
# alkalmazzuk az egymásra épülő relációt:
      a = 2 * a
        # kiírjuk a kapott tagot
       print (a, end=" ")
```

```
A tagok száma: 5
3 6 12 24 48
```

$\mathbf{a_1 = 3}$
$\mathbf{a_n = 2 \cdot a_{n-1}}$ $\mathbf{(n \geq 1)}$

⚠ **Figyelem!** Az ismétlődő sorozatokban a kezdőtagok a kiszámított tagok előtt jelennek meg.

3
$2 \cdot 3 = \mathbf{6}$
$2 \cdot 6 = \mathbf{12}$
$2 \cdot 12 = \mathbf{24}$
$2 \cdot 24 = \mathbf{48}$



<def/> ***Másodrendű egymásra épülés*** – A **Fibonacci sorozat**. Minden tag az őt közvetlenül megelőző két tagtól függ. **Két állapotváltozóra** van szükség.

Pl. Tekintsük a Fibonacci-sorozatot, amelynek minden tagja egyenlő az előző két tag összegével. Generáljuk a sorozat első ***n*** tagját, és figyeljük meg a kialakulását.

$\mathbf{F_0 = 0,\ F_1 = 1,}$
$\mathbf{F_n = F_{n-1} + F_{n-2}}$ $\mathbf{(n \geq 2)}$

$0 + 1 = \mathbf{1}$
$1 + 1 = \mathbf{2}$
$1 + 2 = \mathbf{3}$
$2 + 3 = \mathbf{5}$...



Jelentőség. A Fibonacci-sorozat megjelenik a természetben (csigaházak spirálja, szirmok elrendezése, levélállás), a művészetben (aranymetszés: $F(n+1)/F(n) = \varphi \approx 1{,}618$) valamint keresési és optimalizálási algoritmusokban.

Pszeudokód algoritmus	Python kód
beolvas n **p ← 0** **d ← 1** **kiír p** **ha n > 1 akkor** ** kiír d** ** minden i ← 3, n végezd el** ** c ← p + d** ** kiír c** ** p ← d** ** d ← c** ** minden vége** **ha vége**	`n = int(input("A tagok száma: "))` `# F0=0, F1=1` `a, b = 0, 1` `# kiírunk legalább egy tagot` `print(a, end=" ")` `# ellenőrizzük, hogy létezik-e második tag` `if n > 1:` `  # kiírjuk a második tagot` `  print(b, end=" ")` `  for i in range(2, n):` `  # alkalmazzuk az egymásra épülő relációt` `  a, b = b, a + b  # egyidejű` `  print(b, end=" ")`

```
A tagok száma: 10
0 1 1 2 3 5 8 13 21 34
```

⚠ **Figyelem!** Ügyeljünk a sajátos esetekre, amikor a tagok száma kisebb, mint 2 (n < 2). Ilyenkor ellenőrizzük, hogy létezik-e a második tag (**if n > 1**).

⚠ **Figyelem!** Az a, b = b, a+b utasítás az értékek ***egyidejű cseréjét*** végzi.

Egyes nyelvekben két változó értékének felcserélését általában egy segédváltozó segítségével valósítottuk meg. ***Pythonban*** erre már nincs szükség, mivel a nyelv lehetővé teszi az értékek közvetlen cseréjét egyetlen utasításban.

Pl. Egy bankszámla kezdeti egyenlege S_0. Minden hónap végén az egyenleg változik ***d%*** kamat hozzáadásával. Minden hónap után az egyenleg az előző havi egyenlegtől függ. Számítsuk ki és írassuk ki a számla egyenlegét ***n*** hónap után!

Gyakorlati alkalmazás – egy bankszámla egyenlege. Ha a kezdeti egyenleg S_0 és havonta ***d%*** kamattal nő, akkor: $S_n = S_{n-1} \cdot (1 + d/100)$. Ez egy elsőrendű egymásra épülés.

Python kód	Az egyenleg alakulása (S_0 = 1000 RON, d = 2%)

```
# Beolvasás: egyenleg, kamat,hónapok száma
S0 = float(input("Kezdeti egyenleg: "))
d = float(input("Kamat (%): "))
n = int(input("Hónapok száma: "))
# Inicializáljuk az aktuális egyenleget
egyenleg = S0 # Kiírjuk a 0. hónap egyenlegét
print("Hónap", 0, ":", round(egyenleg, 2), "RON")
# Végigjárjuk mindegyik hónapot 1-től n-ig
for i in range(1, n + 1):
    # Frissítjük az egyenleget,alkalmazzuk a kamatot
    # Képlet: Sn = Sn-1 * (1 + d/100)
    egyenleg = egyenleg * (1 + d / 100)
 # Kiírjuk az aktuális hónap egyenlegét
 # round(egyenleg,2) = az érték 2 tizedesjeggyel
  print("Hónap", i, ":", round(egyenleg, 2), "RON")
```

```
Kezdeti egyenleg: 1000
Kamat (%): 2
Hónapok száma: 12
0. Hónap: 1000.00 RON
1. Hónap: 1020.00 RON
2. Hónap: 1040.40 RON
3. Hónap: 1061.21 RON
4. Hónap: 1082.43 RON
5. Hónap: 1104.08 RON
6. Hónap: 1126.16 RON
7. Hónap: 1148.69 RON
8. Hónap: 1171.66 RON
9. Hónap: 1195.09 RON
10. Hónap: 1218.99 RON
11. Hónap: 1243.37 RON
12. Hónap: 1268.24 RON
```

A generáló algoritmusok hatékonysága

<def/>

A generáló algoritmusok hatékonysága a alkalmazott módszertől függően eltérő lehet.

- A képlettel történő generálás egyszer járja be a sorozatot → ***O(n)*** hatékonyság.
- A tulajdonságalapú való generálás az ***O(n)***-től (pl. páros számok) ***O(n^2)***–ig (pl: prímszámok) változhat.
- Az előző tagok alapján felépülő sorozatoknál minden tag az előző tagok alapján számítódik ki, és az első ***n*** tag generálása rendszerint ***O(n)*** hatékonyságú.

A hatékony módszer megválasztása jelentősen csökkentheti a program futási idejét.

Jegyezd meg!

✔ A szisztematikus generálás egy szabály szerint állít elő egy értéksorozatot.

✔ Az előállítás módszerei:

- képlet (közvetlen számítás)
- tulajdonságok (egy feltétel ellenőrzése)
- egymásra épüléssel (függőség az előző tagoktól)
  - ✔ Az előző tagok alapján felépülő sorozatot a kezdőérték és az egymásra épülő reláció határoz meg
- Elsőrendű egymásra épülés: minden tag az őt közvetlenül megelőző tagtól függ → egyetlen változót használunk.
- Másodrendű egymásra épülés: minden tag az őt megelőző két tagtól függ → két változót használunk (pl. Fibonacci).
  - ✔ Az: a, b = b, a utasítás a változók értékének egyidejű cseréjét végzi.
  - ✔ Az algoritmusok hatékonysága az alkalmazott módszertől függ.

Feladatlap 23

Készítsetek egy *Feladatlap23* nevű mappát. Oldjátok meg az alábbi feladatokat, a helyes válaszokat írjátok egy *valaszok.docx* dokumentumba.

I. *Egészítsétek ki a helyes válaszokkal:*

1. Egészítsétek ki a következő 3 taggal:

a. 2, 5, 8, ___, ___, ___

b. 3, 6, 12, ___, ___, ___

c. 1, 1, 2, 3, 5, ___, ___, ___

d. 4, 9, 16, 25, ___, ___, ___

2. Határozzátok meg a következő 3 tagot:

a. $a_0 = 2, a_n = a_{n-1} + 5 \rightarrow$

b. $a_0 = 1, a_n = 2 \cdot a_{n-1} \rightarrow$

c. $a_0 = 0, a_1 = 1, a_n = a_{n-1} + a_{n-2} \rightarrow$

II. *Válasszátok ki a helyes választ:*

3. Egy egymásra épülő számsor:

☐ függ az előző tagoktól

☐ közvetlen számítható ☐ nincs rá szabály

4. A Fibonacci-számsor:

☐ elsőrendű ☐ másodrendű

☐ nem egymásra épülő

III. *Figyeljétek meg és fogalmazzátok meg a szabályt:*

5. Alkalmazzátok az egymásra épülés elvét és építsétek fel az alábbi sorozatokat:

a. Az első tag 2, és minden további tagot úgy kapunk meg, hogy az előzőhöz hozzáadunk 3-at.

b. Az első tag 5, és minden további tagot úgy kapunk meg, hogy az előzőt megszorozzuk 2-vel.

c. Az első két tag 1 és 4, és minden további tag az előző két tag összege.

6. Adott a következő sorozat: 1, 3, 6, 10, 15, …

a. Számítsátok ki a következő 3 tagot

b. Határozzátok meg a képzési szabályt

c. Adjátok meg a típust (ismétlődő / képlet)

7. Magyarázzátok meg:

Mi a különbség a következők között:

- képlettel történő előállítás
- egymásra épüléssel történő generálás

8. Alkalmazás: Egy bankszámla kezdeti egyenlege 500 RON és havonta 10%-kal növekszik.

a. Számítsátok ki az egyenleget 3 hónap elteltével. **b.** Írjátok fel az egymásra épülés relációját. **c.** Határozzátok meg az egymásra épülés típusát

9. Elemzés: Legyen a következő sorozat: 2, 4, 7, 11, 16, …

a. Határozzátok meg a szabályt. **b.** Számítsátok ki a következő tagot. **c.** Indokoljátok meg a sorozat típusát

10. Feladat: Adott a következő sorozat:

$T_n = 1^2 + 2^2 + 3^2 \ldots + n^2$

a. Számítsátok ki az első 5 tagját **b.** Mit figyelhettek meg? **c.** Egymásra épülő vagy képlettel előállított sorozat?

11. *Elemezzétek a különbséget a sorozattípusok között:* egy képlettel előállított sorozat (pl. négyzetszámok) és egy egymásra épüléssel generált sorozat (pl. Fibonacci). Írjátok le mit figyeltetek meg! Hasonlítsátok össze a tagok kiszámításának módját, az előző értékektől való függőséget, valamint egy tetszőleges tag meghatározásának egyszerűségét.

IV. Készítsetek Python programokat az alábbi feladatokhoz :

12. Egymást követő páratlan számok. Adott egy ***n*** természetes szám. Írassuk ki az első n darab pozitív páratlan számot.

Bemenő adat:	Kimenő adatok:
Beolvas **n**.	Kiírja az első n páratlan számot.
Példa:	
5	**1 3 5 7 9**

13. Állandó különbségű sorozat. Adott egy sorozat, amelynek első tagja ***100***, és minden következő tagot úgy kapunk, hogy az előző tagból kivonunk 7-et. Írassuk ki az első ***n*** tagot.

Bemenő adat:	Kimenő adatok:
Beolvas **n**.	Kiírja a sorozat első n tagját.
Példa:	
5	**100 93 86 79 72**

14. A 3 hatványai. Adott egy ***n*** természetes szám. Írassuk ki a 3 első ***n*** hatványát a 3^0-tól kezdődően

Bemenő adat:	Kimenő adatok:
Beolvas: **n**	Kiírja az előállított értékeket.
Példa:	
5	**1 3 9 27 81**

15. Napi növekedés. Egy növény kezdeti magassága ***h*** centiméter. Minden nap ***p*** centiméterrel növekszik. Írassuk ki a növény magasságát a következő ***n*** nap mindegyikében.

Bemenő adat:	Kimenő adatok:
Beolvas: **h**, **p**, **n**.	Kiírja az ***n*** magasságok mindenikét.
Példa:	
10 3 5	**13 16 19 22 25**

16. Energiafogyasztás. Egy eszköz az első órában ***c*** egységnyi energiát fogyaszt. Minden következő órában a fogyasztás ***2*** egységgel nő. Határozzuk meg a teljes fogyasztást ***n*** óra elteltével!).

Bemenő adat:	Kimenő adatok:
Beolvas: **c** és **n**.	Kiírja a teljes fogyasztást.
Példa:	
5 4	**32**

Magyarázat: a fogyasztások 5, 7, 9, 11, az összeg pedig 32

17. Belépőkód. Egy számkód az ***a*** értékkel kezdődik. Minden lépésben az új értéket az ***a = a*2 +1*** szabály alapján kapjuk. Írassuk ki a kód első ***n*** értékét.

Bemenő adat:	Kimenő adatok:
Beolvas: **a** és **n**.	Kiírja az előállított értékeket.
Példa:	
2 5	**2 5 11 23 47**

18. Adott egy **n** természetes szám. Írassuk ki azokat a számokat az [**1**, **n**] intervallumból, amelyek páros négyzetszámok.

Bemenő adat:	Kimenő adatok:
Beolvas: **n**	Kiírja azokat a számokat, amelyek mindkét feltételt teljesítik.
Példa:	
30	**4 16**
Feltételek: i % 2 == 0 és i négyzetszám	

19. Elemezzétek a leckében szereplő Fibonacci-programot, és válaszoljatok az alábbi kérdésekre:

a. Mit ír ki a program, ha ***n = 0***?
b. Mit ír ki, ha ***n = 1***?
c. Mit ír ki, ha ***n = 2***?
d. Miért szükséges az ***if n > 1*** utasítás a programban?



Önértékelés!

Az *onertekeles23.docx* állományba írjátok be a megfelelő válaszokat, a következő jelöléseket használva: *1 – igen, 2 – nem, 3 – nem vagyok biztos benne*, majd mentsétek el a *Feladatlap23* mappába.

	A lecke végén ...	***Igen***	***Nem***	***Nem biztos***
1.	felismerek egy egymásra épülő sorozatot.			
2.	ki tudom számítani egy sorozat tagjait.			
3.	megkülönböztetem a képletet és az egymásra épülést.			
4.	írok egy egyszerű algoritmust.			
5.	azonosítom a generáló algoritmusok hatékonyságát (komplexitását).			

6. Nehézségeim voltak a következőkben: …

Ne felejtsétek el értékelni a viselkedéseteket és a tevékenységeteket az óra során!
(a 7. oldalon található viselkedés- és tevékenységfigyelő napló segítségével)

A lecke végén így éreztem magam:

24. Lecke – Rendezési módszerek: a minimumkiválasztásos rendezés

<def/> A **rendezés** az a folyamat, amelynek során egy lista elemeit egy megadott – általában növekvő vagy csökkenő sorrendbe rendezzük. Ez az informatika egyik alapvető művelete, mivel a rendezett adatok sokkal hatékonyabban kereshetők és dolgozhatók fel.



Rendezési módszerek. A következő leckékben egy lista elemeinek rendezésére három különböző módszert fogunk tanulmányozni. Mindegyiknek megvannak a sajátos elvei, előnyei és felhasználási esetei:

Módszer	Kulcsszó	Fő ötlet
1. Minimumkiválasztás	Minimum	Megkeresi a rendezetlen részsor legkisebb elemét, a helyére teszi, ismétli.
2. Buborékrendezés	Szomszédos	Összehasonlítja a szomszédos elemeket, felcseréli őket, ha rossz sorrendben vannak.
3. Frekvencia lista	Számlálás	Megszámolja az egyes értékek előfordulásait, a kapott értékek alapján sorrendbe újraépíti a listát. Kis értéktartományok esetén hatékony.

⚠ **Figyelem!** Nem létezik általánosan optimális módszer. A döntés függ a **lista méretétől**, **az értékek tartományától**, **a műveletek költségétől** (összehasonlítás vs. felcserélés) és attól, hogy a lista részben rendezett-e.

Rendezési módszerek: a minimumkiválasztásos rendezés

Pl. **A lecke forgatókönyve — Tehetségkutató verseny.** *Egy tehetségkutató verseny szervezőinek 7 döntőst kell a pontszámuk szerint, a legkisebbtől a legnagyobbig sorba rendezniük. Hogyan járunk el? Megkeressük a legkisebb pontszámot, az első helyre tesszük, majd ugyanezt megismételjük a lista további elemeivel. Pontosan ez a* ***minimumkiválasztásos rendezés*** *elve.*

Hogyan működik a minimumkiválasztás? Az algoritmus ***n−1 ismétlést*** végez. Az i. ismétlésnél (i tart 0-tól n−2-ig), megkeresi a ***legkisebb elemet*** a rendezetlen részsorozatból (az i. pozíciótól az n−1.-ig) és az i. pozícióra helyezi, egyetlen felcseréléssel.

Kezdeti	**5**	**2**	**8**	**1**	**9**	**3**	**7**	
1. lépés	**1**	**2**	**8**	**5**	**9**	**3**	**7**	*min=1 (poz.3) ↔ poz.0*
2. lépés	1	**2**	**8**	**5**	**9**	**3**	**7**	*min=2 (poz.1), helyes pozíció*
3. lépés	1	2	**3**	**5**	**9**	**8**	**7**	*min=3 (poz.5) ↔ poz.2*
4. lépés	1	2	3	**5**	**9**	**8**	**7**	*min=5 (poz.3), helyes pozíció*
5. lépés	1	2	3	5	**7**	**8**	**9**	*min=7 (poz.6) ↔ poz.4*
6. lépés	1	2	3	5	7	**8**	**9**	*min=8 (poz.5), helyes pozíció*
Végső ✓	**1**	**2**	**3**	**5**	**7**	**8**	**9**	*Rendezett lista*

Minden ismétlés végén egy újabb elem végleges helyére kerül. A **rendezett részsorozat** baloldalt egy-egy elemmel növekszik, míg jobboldalt a **rendezetlen rész** csökken.

<def/> **A minimumkiválasztásos rendezés** egy olyan algoritmus, amely minden ***i.*** lépésben megkeresi az ***a[i..n−1]*** részsorozat legkisebb elemét, és felcseréléssel az ***i.*** pozícióra helyezi. Az algoritmus pontosan ***n−1*** ismétlést és legtöbb ***n−1 felcserélést*** hajt végre, az elemek kezdeti sorrendjétől függetlenül.



Az algoritmus lépései

1. lépés: **Kezdet** – a teljes lista rendezetlen.
2. lépés: **Keresés** – megkeressük a rendezetlen részsorozat legkisebb elemének helyét.
3. lépés: **Elhelyezés** – felcseréljük a minimumot a rendezetlen rész első elemével.
4. lépés: **Előrelépés** – a határt egy pozícióval jobbra toljuk; az elem a végleges helyére került.
5. lépés: **Ismétlés** – addig ismételjük a 2–4 lépést, amíg a lista teljesen rendezett nem lesz.

Pszeudokód algoritmus

```
beolvas n
minden i ← 0, n-1 végezd el
    beolvas a[i]
minden vége
minden i ← 0, n-2 végezd el
    poz_min ← i
    minden j ← i+1, n-1 végezd el
        ha a[j] < a[poz_min] akkor
            poz_min ← j
        ha vége
    minden vége
    ha poz_min ≠ i akkor
        aux ← a[i]
        a[i] ← a[poz_min]
        a[poz_min] ← aux
    ha vége
minden vége
minden i ← 0, n-1 végezd el
    kiír a[i]
minden vége
```

Python kód

```
n = int(input("Az elemek száma: "))
# a lista elemeinek beolvasása
a = [] for i in range(n):
    x = int(input("a[" + str(i) + "] = "))
    a.append(x)
# minimumkiválasztásos rendezés
for i in range(n - 1):
    # feltét., hogy a min az i. pozíción van
    poz_min = i
    for j in range(i + 1, n):
        if a[j] < a[poz_min]:
            # elmentjük az új minimum helyét
            poz_min = j
    if poz_min != i:
            # felcserélés
        a[i], a[poz_min] = a[poz_min], a[i]
print("Rendezett lista:", a)
```

```
Az elemek száma: 7
a[0] = 5   a[1] = 2   a[2] = 8   a[3] = 1   a[4] = 9   a[5] = 3   a[6] = 7
A rendezett lista: [1, 2, 3, 5, 7, 8, 9]
```

⚠ **Figyelem!** A ***poz_min*** változó a ***legkisebb elem helyét menti el*** és nem az értékét. A felcserélés minden ismétlés végén, ***egyszer*** történik meg, nem minden összehasonlításkor.

Kritérium	Kedvező eset	Átlagos eset	Kedvezőtlen eset
Összehasonlítások	n(n−1)/2	n(n−1)/2	n(n−1)/2
Felcserélések	0 (már rendezett)	~n/2	n−1
Időbonyolultság	$O(n^2)$	$O(n^2)$	$O(n^2)$
További memóriaigény	O(1)	O(1)	O(1)



Jegyezzétek meg! Az összehasonlítások száma ***azonos*** minden esetben: $O(n^2)$. A felcserélések száma változik: legkevesebb ***0*** (rendezett lista esetén), legtöbb ***n−1*** (csökkenő sorrendbe rendezett lista esetén). Az algoritmus nem használja ki a már meglévő részleges rendezettséget.

Előnyök

- ✓ Legtöbb n−1 felcserélés — hatékony, ha az adatok mozgatása költséges
- ✓ Egyszerű, érthető algoritmus, könnyen kódolható
- ✓ Nem igényel további tárhelyet — helyben rendez, O(1)
- ✓ Kiszámítható viselkedés, az elemek kezdeti sorrendjétől független

Hátrányok

- × $O(n^2)$ összehasonlítás majdnem rendezett lista esetén is
- × Nem stabil algoritmus (az egyenlő értékek sorrendje megváltozhat)
- × Nagy listák esetén nem hatékony (n > 500–1000 elem)
- × Nem használja ki a részleges rendezettséget

GYAKORLATI ALKALMAZHATÓSÁG: A minimumkiválasztásos rendezés megfelelő kis elemszámú listák (n ≤ 50–100) esetén vagy akkor, ha az elemek cseréje költséges. Nagy adathalmazok esetén nem ajánlott – a hatékonyabb módszereket a későbbi leckékben fogjuk tanulmányozni.

Pl. **Megoldott feladat.** *Egy webáruház a vásárlói megrendeléseket az értékük szerinti csökkenő sorrendben szeretné megjeleníteni, hogy kiemelje a legfontosabb eladásokat. A 7 megrendelés értéke: [320, 85, 210, 55, 140, 390, 170]. Figyeljétek meg a lista csökkenő sorrendbe történő rendezését a kiválasztásos módszer használatával.*

Megoldás: Minden lépésben kiválasztjuk a maximumot, és a megfelelő pozícióba helyezzük.

```
a = [320, 85, 210, 55, 140, 390, 170]
n = len(a)
for i in range(n - 1):
     poz_max = i
     for j in range(i + 1, n):
         # csökkenő rendezés esetén csak a relációs jelet cseréljük
         if a[j] > a[poz_max]:
             poz_max = j
     a[i], a[poz_max] = a[poz_max], a[i]
print("Rendezett lista:", a)
```

```
Rendezett lista : [390, 320, 210, 170, 140, 85, 55]
```

Megj. Csökkenő sorrendbe rendezéskor az algoritmus ugyanaz. Csak az összehasonlítási feltétel változik: a minimum keresése helyett a maximumot keressük. a[j]>a[poz_max] lesz az a[j]<a[poz_min] helyett.

Jegyezzétek meg!

- ✔ A rendezés egy lista elemeit egy bizonyos sorrendbe rendezi; a minimumkiválasztás minden lépésben a legkisebb elemet a megfelelő pozícióra helyezi.
- ✔ Minden ismétlés után a lista rendezett része eggyel több elemet fog tartalmazni; az algoritmus pontosan n−1 lépést és $O(n^2)$ összehasonlítást végez és legtöbb n−1 cserét hajt végre.
- ✔ Egyszerű, megfelelő kevés adatot tartalmazó listák esetén; sok adat esetén nem hatékony.

Feladatlap 24

Készítsetek egy *Feladatlap24* nevű mappát. Oldjátok meg az alábbi feladatokat, a helyes válaszokat írjátok egy *valaszok.docx* dokumentumba.

Az alábbi 1. és 2. feladatoknál válasszátok ki a helyes választ.

1. Hány összehasonlítást végez a minimumkiválasztásos rendezés algoritmus egy 5 elemű listán?

a. 4; **b.** 10; **c.** 5; **d.** 20.

2. Alkalmazzuk a minimumkiválasztást a `[6, 2, 8, 1, 4]`. listán. Hogy néz ki a lista a 2. lépés után `(i=1)`?

a. `[1, 2, 8, 6, 4]`; **b.** `[1, 2, 6, 8, 4]`; **c.** `[2, 1, 8, 6, 4]`; **d.** `[1, 8, 2, 6, 4]`.

3. Anélkül, hogy gépen lefuttatnád az alábbi Python kódot, írd le, mit fog kiírni a program az alábbi két esetben:

a.

```
nre = [5, 3, 8, 1, 6]
n = len(nre)
for i in range(n - 1):
   poz_min = i
   for j in range(i + 1, n):
      if nre[j] < nre[poz_min]:
         poz_min = j
   if poz_min != i:
      nre[i], nre[poz_min] = nre[poz_min], nre[i]
print(nre)
```

Válasz: .....

b.

```
a = [4, 4, 4, 4]
n = len(a)
nr_csere = 0
for i in range(n - 1):
   poz_min = i
   for j in range(i + 1, n):
      if a[j] < a[poz_min]:
         poz_min = j
   if poz_min != i:
      a[i], a[poz_min] = a[poz_min], a[i]
      nr_csere += 1
print(a, nr_csere)
```

Válasz: .....

4. Miért nem hatékony a minimumkiválasztásos rendezés nagy listák esetén?

5. Legyen az *L = [320, 85, 210, 55, 140, 390, 170]* lista. Töltsétek ki a csökkenő irányú rendezéshez használt maximumkiválasztást ábrázoló futtatás táblázatot, minden lépéshez tüntessétek fel a kiválasztott elemet és a végrehajtott műveletet:

Lépés	0. poz.	1. poz.	2. poz.	3. poz.	4. poz.	5. poz.	6. poz.	Művelet
Start	320	85	210	55	140	390	170	
1. lépés								
2. lépés								
3. lépés								
4. lépés								
5. lépés								
6. lépés								
Stop ✓								

6. Elemezzétek a minimumkiválasztásos rendezés algoritmusának viselkedését a következő határesetekben. Mindegyiknél határozzátok meg, hány felcserélés történik, és miért:

a. [1, 2, 3, 4, 5] – már növekvő sorrendű lista; **b.** [5, 4, 3, 2, 1] – csökkenő sorrendbe rendezett lista;

c. [7] – egyetlen elemet tartalmazó lista; **d.** [3, 3, 3, 3] – azonos értékű elemeket tartalmazó lista.

7. Adott a következő (név, jegy) párokat tartalmazó lista: [("Anna", 8), ("Bobi", 9), ("Kara", 8), ("Dani", 9)]

A minimumkiválasztásos rendezés algoritmusa a jegyek szerint rendezi a listát. Válaszoljatok a kérdésekre:

a. Anna és Kara jegye azonos (8). Rendezés után melyikük jelenik meg hamarabb?

b. Bobi és Dani jegye azonos (9). Rendezés után melyikük jelenik meg hamarabb?

c. Mit értünk az alatt, hogy egy rendezési algoritmus stabil? A minimumkiválasztásos rendezés stabil?

Útmutatás: Egy algoritmus akkor stabil, ha az egyenlő értékű elemek megőrzik az eredeti listában lévő egymáshoz viszonyított sorrendjüket. A minimumkiválasztásos rendezés nem stabil — a felcserélés megváltoztathatja az egyenlő értékű elemek sorrendjét.



Önértékelés!

Az *onertekeles24.docx* állományba írjátok be a megfelelő válaszokat, a következő jelöléseket használva: *1 – igen, 2 – nem, 3 – nem vagyok biztos benne*, majd mentsétek el a *Feladatlap24* mappába.

	A lecke végén tudni fogom, hogy...	*Igen*	*Nem*	*Nem biztos*
1.	mi a rendezés és miért fontos az informatikában.			
2.	hogyan működik a minimumkiválasztásos rendezés algoritmusa.			
4.	hogyan számítható ki az összehasonlítások száma: n(n−1)/2 képlet.			
5.	mennyi az algoritmus időbonyolultsága: $O(n^2)$.			
6.	melyek a minimumkiválasztásos rendezés előnyei és hátrányai.			
8.	hogyan kódoljuk az algoritmust Pythonban, beépített rendezési függvények nélkül.			

9. Nehézségeim voltak a következőkben: …

Ne felejtsétek el értékelni a viselkedéseteket és a tevékenységeteket az óra során!
(a 7. oldalon található viselkedés- és tevékenységfigyelő napló segítségével)

A lecke végén így éreztem magam:



25. Lecke – Rendezési módszerek: a buborékrendezés

Megj. Megtudtuk, hogy a **minimumkiválasztás** minden i lépésben megkeresi az ***a[i..n–1]*** részsorozat legkisebb elemét, és egyetlen cserével a megfelelő pozícióba helyezi azt. Az algoritmus mindig ***n(n–1)/2* összehasonlítást** hajt végre, függetlenül a kezdeti sorrendtől – a sorozat részleges rendezettségét nem tudja kihasználni

Miért van szükségünk egy másik módszerre? A minimum kiválasztása egyszerű és kiszámítható, de nem használja ki a meglévő részleges rendezés előnyeit: még ha a sorozat már rendezett is, akkor is ***n(n–1)/2*** összehasonlítást hajt végre. A **Buborék módszer** csökkentheti ezt a korlátozást a ***csere*****,** változóval, amely lehetővé teszi egy már rendezett sorozat korai észlelését, kedvező esetben az összehasonlítások számát mindössze n–1-re csökkentve.

Rendezési módszerek: buborékrendezés módszere

<def/> A buborékrendezés (***Bubble Sort***) egy olyan rendező algoritmus, amely egy sorozat elemeit rendezi úgy, hogy ismételten összehasonlítja és felcseréli a ***szomszédos elemeket***. A név a víz felszínére fokozatosan emelkedő légbuborékok hasonlatból ered: a nagy elemek a sorozat vége felé "lebegnek", a kicsik pedig az eleje felé "süllyednek", a sorozat minden egyes bejárása során.

Pl. **A lecke forgatókönyve – Magasság szerinti rangsorolás**. *A tornatanár magasság szerint szeretné sorrendbe állítani a diákokat. A következőképpen jár el: összehasonlít minden egyes diákot a jobb oldalival – ha a bal oldali magasabb, helyet cserélnek. A folyamatot megismételi az egész osztállyal,* ***n–1 alkalommal****. A legmagasabb elem mindig a sorozat jobb végére kerül, mint egy felszínre kerülő buborék. Ez az ötlete a* ***buborékmódszernek****.*

Hogyan működik a „buborékmódszer”/ „Bubble Sort”? Az algoritmus legfeljebb n–1 bejárást végez. Minden i lépésben (0-tól n–2-ig) bejárja a rendezetlen részsorozatot, és összehasonlítja a ***szomszédos*** a[j] és a[j+1] elempárokat. Ha a[j] > a[j+1], akkor felcseréli őket. A csere változó lehetővé teszi a korai leállást: ha egy bejárás során nincs ***csere*** a sorozat már rendezett.

Kövessétek a magasságok sorozatának alakulását minden bejárás után, a mellékelt sorozatra [170, 155, 182, 163, 175, 168]:

Kezdet	170	155	182	163	175	168	
1. bejárás	155	170	163	175	168	182	*4 csere: (170↔155), (182↔163), (182↔175), (182↔168) →**182** a helyére került*
2. bejárás	155	163	170	168	175	182	*2 csere: (170↔163), (175↔168) → **175** a helyére került*
3. bejárás	155	163	168	170	175	182	*1 csere: (170↔168)→**170** a helyére került*
4. bejárás	155	163	168	170	175	182	*0 csere→ **Korai leállás!***
Utolsó bejárás ✓	155	163	168	170	175	182	***A sorozat rendezett***

Minden bejárás végén a rendezetlen részsorozat legnagyobb eleme eléri a végső pozíciót, vagyis a helyére kerül – **jobbra lebegve**. A ***csere*** változó lehetővé teszi az algoritmus leállását, ha egy bejárás során nincs csere.

<def/> A ***buborékrendezés*** módszere egy rendező algoritmus, amely minden j lépésben összehasonlítja a szomszédos ***a[j]*** és ***a[j+1]*** elemeket, és felcseréli őket, ha ***a[j] > a[j+1]***. Az algoritmus legfeljebb ***n–1 bejárást*** hajt végre. Az optimalizált változat esetén az algoritmus leáll, ha egy bejárás nem eredményez cserét (**csere = hamis**).

<def/> **Az algoritmus lépései**:

1. lépés: **Kezdet** — a sorozat rendezetlen.
2. lépés: **Összehasonlítás** — összehasonlítjuk az a[j] és a[j+1] szomszédos elempárokat

3. lépés: **Csere** — ha a[j] > a[j+1], akkor felcseréljük, és a csere ← igaz.
4. lépés: **Haladás** — a jelenlegi bejárás utolsó eleme a helyére kerül.
5. lépés: **Ismétlés** — ismételjük a 2–4-es lépéseket vagy kilépünk, ha csere = hamis

Pszeudokód algoritmus

```
beolvas n
minden i <- 0,n-1 végezd el
    beolvas a[i]
minden vége
// buborékrendezés (optimalizálva)
minden i <- 0,n-2 végezd el
    csere <- hamis
    minden j <- 0,n-2-i végezd el
        ha a[j] > a[j+1] akkor
            aux <- a[j]
            a[j] <- a[j+1]
            a[j+1] <- aux
            csere <- igaz
        ha vége
    ha csere = hamis akkor
        kilépés
    ha vége
    minden vége
minden vége
minden i <- 0,n-1 végezd el
    kiír a[i]
minden vége
```

Python kód

```
n = int(input("Elemek száma: "))
# sorozat beolvasása
a = []
for i in range(n):
    x = int(input("a["+str(i)+"] = "))
    a.append(x)
# buborékrendezés
for i in range(n - 1):
    csere = False
    for j in range(n - 1 - i):
        if a[j] > a[j + 1]:
            a[j], a[j+1] = a[j+1], a[j]
            csere = True
    # korai leállás
    if not csere:
        break
print("A rendezett sorozat:", a)
```

```
Elemek száma: 5
a[0] = 5   a[1] = 2   a[2] = 8   a[3] = 1   a[4] = 4
A rendezett sorozat: [1, 2, 4, 5, 8]
```

⚠ **Figyelem!** A ***csere*** változó az optimalizálás kulcsa: ha a sorozat elemeit bejártuk és nem történt csere, akkor a sorozat rendezett, és az algoritmus leáll. Így kedvező esetben (azaz, ha a sorozat már rendezett) csak ***n–1 összehasonlítás*** történik – ***O(n)***.

Kritérium	Kedvező eset	Kedvezőtlen eset	Megjegyzés
Összehasonlítások	n–1	n(n–1)/2	A kezdeti sorrendtől függ
Cserék	0 (már rendezett)	n(n–1)/2	Maximális teljesítmény a fordított sorrendű listánál
Bonyolultság	O(n)	$O(n^2)$	Rossz esetben az első átfutás után leáll
További memória	O(1)	O(1)	Nincs szükség további memóriára



Jegyezd meg! A buborékrendezés korán leáll, ha a számsorozat rendezett – ez az előnye a minimumkiválasztás módszerrel szemben, amely mindig n(n–1)/2 összehasonlítást végez. Kedvezőtlen esetben azonban, a buborékrendezés több ***cserét*** hajt végre (legfeljebb n(n–1)/2-t n–1-ből). A buborékrendezés módszer stabil: az egyenlő értékű elemek nem változtatják meg a relatív sorrendjüket a rendezés során.

Mikor használjuk a buborékrendezést?

Előnyök

✓ Stabil algoritmus — az egyenlő értékű elemek sorrendje megmarad
✓ Korai leállás — hatékony, ha a sorozat majdnem rendezett
✓ Egyszerűen érthető és megvalósítható
✓ Nem igényel további memóriát — rendezés egy helyen, O(1)

Hátrányok

- × $O(n^2)$ összehasonlítások és cserék kedvezőtlen esetben
- × Sok összehasonlítás a minimumkiválasztáshoz képest
- × Nem hatékony sok elem esetén (n > 500–1000 elem esetén)
- × Nem ajánlott sok csere esetén

GYAKORLATI ALKALMAZHATÓSÁG: A korai leállás miatt, a buborékrendezés inkább kevés elemű sorozatok vagy majdnem rendezett sorozatok esetén alkalmazható. Például: egy orvosi rendelő napi konzultációinak rendezése azok időtartama alapján. **Nem ajánlott** használni nagy adathalmazok esetén.

Pl.

Megoldott feladat. *Egy meteorológiai állomás 7 egymást követő napon méri a hőmérsékletet (°C): [3, -1, 5, 2, 0, 4, -2]. Rendezd a sorozatot növekvő sorrendbe a buborékrendezés módszerével, azonosítva a bejárásokat.*

Megoldás: Minden bejárás után, a még nem rendezett legnagyobb elem a helyére kerül.

Kezdet	**3**	**-1**	**5**	**2**	**0**	**4**	**-2**	
1. bejárás	-1	3	2	0	4	-2	**5**	*5 csere; 5 a helyére került*
2. bejárás	-1	2	0	3	-2	**4**	5	*3 csere; 4 a helyére került*
3. bejárás	-1	0	2	-2	**3**	4	5	*2 csere; 3 a helyére került*
4. bejárás	-1	0	-2	**2**	3	4	5	*1 csere; 2 a helyére került*
5. bejárás	-1	-2	**0**	2	3	4	5	*1 csere; 0 a helyére került*
6. bejárás	-2	**-1**	0	2	3	4	5	*1 csere; -1 a helyére került*
Utolsó ✓	**-2**	**-1**	**0**	**2**	**3**	**4**	**5**	*Teljesen rendezett sorozat*

Python kód

```
a = [3, -1, 5, 2, 0, 4, -2]
for i in range(len(a) - 1):
     csere = False
     for j in range(len(a) - 1 - i):
         if a[j] > a[j+1]:
             a[j],a[j+1]=a[j+1],a[j]
             csere=True
     if not csere: break
print("Rendezett sorozat:", a)
```

```
A rendezett sorozat:  [-2,  -1,  0,  2,  3,  4,  5]
```

Megjegyzés – Csökkenő sorrendbe rendezés esetén. Egy számsorozat **csökkenő** sorrendbe rendezéséhez egyetlen változtatás elegendő a növekvő változathoz képest: a cserefeltétel megfordul a következőhöz képest **a[j] > a[j+1]** helyett **a[j] < a[j+1]** lesz. Így, a legkisebb elem kerül minden bejárás után a jobb szélre, és a sorozat a legnagyobbtól a legkisebbig lesz rendezett.

Python kód

```
a = [3, -1, 5, 2, 0, 4, -2]
for i in range(len(a) - 1):
    csere = False
    for j in range(len(a) - 1 - i):
        if a[j] < a[j + 1]:          # relációs jel megfordítva
            a[j], a[j+1] = a[j+1], a[j]
            csere = True
    if not csere:
        break
print("Csökkenő sorozat:", a)
```

```
Csökkenő sorozat: [5, 4, 3, 2, 0, -1, -2]
```

Megj. A buborékrendezés hasonló a minimumkiválasztásos rendezéshez a bonyolultság szempontjából: **O(n²)**, de az elemek elhelyezésének mechanizmusában különbözik: a buborékos módszernél a cserék lokálisak (szomszédos elemek), és a **maximális** elem fokozatosan a sorozat vége felé halad. A minimumkiválasztás módszernél a **minimum** lesz közvetlenül kiválasztva, és a **megfelelő pozícióba** helyezve.

Jegyezd meg!

- ✔ A buborékrendezés összehasonlítja a szomszédos elemeket, és felcseréli őket, ha rossz sorrendben vannak.
- ✔ Minden bejárás végén a legnagyobb rendezetlen elem éri el a végső pozícióját.
- ✔ A csere változó lehetővé teszi a korai leállást — Kedvező esetben a bonyolultság O(n). Stabil: az azonos elemek sorrendje megmarad.
- ✔ Legfeljebb n−1 bejárás, a kedvezőtlen eset bonyolultsága O(n²). Egyszerű, alkalmas kevés elemű vagy majdnem rendezett sorozatok rendezéséhez (n ≤ 500).

Feladatlap 25

Készítsetek egy *Feladatlap25* nevű mappát. Oldjátok meg az alábbi feladatokat, a helyes válaszokat írjátok egy *valaszok.docx* dokumentumba.

I. *Válasszátok ki a helyes választ (csak egy válasz helyes).*

1. A buborékrendezés minden egyes bejárásának végén:

a. a legkisebb elem az első helyre kerül;
b. az elemek teljesen rendezettek lesznek;
c. a legnagyobb, még nem rendezett elem a végső helyére kerül;
d. a sorozat elemei megfordulnak.

2. Mi a szerepe a „csere” (csere történt) változónak?

a. eltárolja a legkisebb elemet;
b. megszámolja az összehasonlításokat;
c. lehetővé teszi az algoritmus korai leállását;
d. csökkenő sorrendbe rendezi az elemeket.

3. Mennyi a buborékrendezés időbeli bonyolultsága a legkedvezőtlenebb esetben?

a. O(1);
b. O(log n);
c. O(n);
d. O(n²).

4. Ha a sorozat elemei már növekvő sorrendbe vannak rendezve, az algoritmus:

a. végrehajtja az összes bejárást;
b. az első kör után leáll;
c. hibát okoz;
d. csökkenő sorrendbe rendezi a sorozatot.

5. Egészítsd ki az üres helyeket

a. A buborékrendezés ..... elemeket hasonlít össze.
b. Ha **a[j] > a[j+1]**, akkor az elemek ..... .
c. Kedvező esetben az időbeli bonyolultság ..... .

6. Párosítsd helyesen:

Helyzet	Az algoritmus viselkedése
A. Már rendezett sorozat	**a.** n(n−1)/2 csere
B. Csökkenő sorrendű sorozat	**b.** leáll az első kör után
C. Minden elem egyenlő	**c.** 0 csere

II. *Válaszolj teljes mondatokban!*

7. Magyarázd el, miért alkalmas a buborékrendezés kis méretű vagy majdnem rendezett sorozatok rendezésére.

8. Mi a fő különbség a buborékrendezés és a minimumkiválasztásos rendezés között?

9. Miért tekinthető a buborékrendezés stabil rendezési algoritmusnak?

10. Kövesd végig a rendezést buborékrendezéssel! Adott a következő sorozat: [4, -2, 7, 1, 0, 5]. Töltsd ki a táblázatot a növekvő sorrendű rendezéshez:

Lépés	Az eredmény	Cserék száma
Kezdet	**4 -2 7 1 0 5**	-
1. bejárás		
2. bejárás		
3. bejárás		
4. bejárás		
5. bejárás		
Utolsó bejárás ✓		

11. Futtasd le az alábbi programot, papíron. Mit ír ki:

```
a = [5, 1, 4, 2]
for i in range(len(a)-1):
    csere = False
    for j in range(len(a)-1-i):
        if a[j] > a[j+1]:
            a[j], a[j+1] = a[j+1],a[j]
            csere = True
    if not csere:
        break
print(a)
```

Válasz: .....

III. *Mini Python-projekt*

12. Készíts Python-alkalmazást egy online platformon közzétett videók megtekintéseinek elemzésére. A program beolvas a billentyűzetről egy n elemű sorozatot, amelynek elemei a videók megtekintéseinek számát jelentik, majd ezeket csökkenő sorrendbe rendezi buborékrendezés (bubble sort) segítségével, azért hogy a legnépszerűbb videók kerüljenek az első helyekre.

Az alkalmazásnak ki kell írnia:

- a megtekintések számát csökkenő sorrendben;
- az elvégzett összehasonlítások teljes számát;
- a végrehajtott cserék (felcserélések) teljes számát;
- egy üzenetet, amely jelzi, hogy a sorozat már rendezett volt-e az algoritmus alkalmazása előtt.

Mentsd a projektet ***megtekintesek.py*** néven a ***Feladatlap25*** mappába, majd készíts egy képernyőfelvételt a futtatott programról ***kepernyokep.jpg*** a megadott mappába.



Önértékelés!

Az *onertekeles25.docx* állományba írjátok be a megfelelő válaszokat, a következő jelöléseket használva: *1 – igen, 2 – nem, 3 – nem vagyok biztos benne*, majd mentsétek el a *Feladatlap25* mappába.

	A lecke végén ...	***Igen***	***Nem***	***Nem biztos***
1.	el tudom magyarázni, hogyan működik a buborékos rendezés.			
2.	tudom a csere változó szerepét.			
3.	le tudom írni az algoritmus lépéseit.			
4.	ki tudom számítani az algoritmus bonyolultságát.			
5.	le tudom kódolni a buborékos rendezést Python programozási nyelven.			
6.	meg tudom magyarázni az algoritmus előnyeit és hátrányait.			

7. Nehézségeim voltak a következőkben: …

Ne felejtsétek el értékelni a viselkedéseteket és a tevékenységeteket az óra során!
(a 7. oldalon található viselkedés- és tevékenységfigyelő napló segítségével)

A lecke végén így éreztem magam:

26. Lecke – Rendezési módszerek: rendezés frekvenciatömb segítségével

Megj. Megtanultuk, hogy a ***buborékrendezés*** a szomszédos elemeket hasonlítja össze, és szükség esetén felcseréli őket, időbonyolultsága pedig $O(n^2)$. A *csere* változó lehetővé teszi a korai leállást kedvező esetben, így a bonyolultság $O(n)$ lehet. Mindazonáltal mindkét tanult módszer (a minimumkiválasztásos rendezés és a buborékrendezés) *elemek összehasonlítását* igényli, ezért kedvezőtlen esetben $O(n^2)$ időbonyolultságra korlátozódnak.

Miért van szükségünk más módszerre? Az összehasonlításon alapuló rendezési algoritmusoknak elméleti alsó korlátjuk van, $\Omega(n \log n)$. Ha azonban az értékek ***egész számok, amelyek egy ismert és véges tartományból származnak***, akkor összehasonlítás nélkül is rendezhetünk, elérve az $O(n + k)$ időbonyolultságot, ahol **k** a lehetséges különböző értékek száma.

Rendezési módszerek: frekvenciatömbös rendezés módszere

<def/> A ***frekvenciatömbös rendezés*** (Counting Sort) egy olyan rendezési algoritmus, amely megszámolja a sorozat egyes értékeinek előfordulásait egy segédtömbben, majd összehasonlítás nélkül újraépíti a rendezett sorozatot. Kizárólag véges tartományba eső ***egész számok*** esetén alkalmazható, és különösen hatékony, amikor a ***k*** (a lehetséges különböző értékek száma) kicsi az ***n***-hez (az elemek számához) képest.

Pl. ***A lecke forgatókönyve — Osztálynapló***. A tanár 10 diák jegyeit szeretné növekvő sorrendbe rendezni (az osztályzatok egész számok 5 és 10 között). Ahelyett, hogy a jegyeket egymással hasonlítaná össze, másképp jár el: készít egy 6 cellából álló táblázatot (egyet minden lehetséges jegyhez, 5-től 10-ig), majd egyszer végighalad a naplón, és minden jegynél megjelöli a megfelelő cellát. Ezután újraírja a jegyzéket úgy, hogy minden jegyet annyiszor ír le, amennyit az illető jegynek megfelelő cella számlálója mutat. Pontosan ez az ötlet áll a ***frekvenciatömbös rendezés*** (Counting Sort) mögött.

Hogyan működik? A sorozatot egyszer bejárjuk, és létrehozzuk a **freq**, ahol a **freq[v – min]** a v érték előfordulásainak számát tárolja. Ezután a rendezett sorozatot úgy állítjuk elő, hogy balról jobbra végighaladunk a **freq** tömbön, és minden értéket annyiszor írunk ki, amennyit a hozzá tartozó előfordulás jelez. *Nincs szükség az elemek összehasonlítására!*

Kövesd nyomon a frekvenciatömb felépítését a következő jegyek esetén [7, 5, 8, 5, 9, 7, 6, 8, 5, 9]:

Lépések	jegy=5	jegy=6	jegy=7	jegy=8	jegy=9	Megjegyzés
Eredeti sorozat	3 előfordulás	1 előfordulás	2 előfordulás	2 előfordulás	2 előfordulás	[7,5,8,5,9,7,6,8,5,9]
freq[ ]	3	1	2	2	2	freq[0..4] = [3,1,2,2,2]
Végső sor ✓	5,5,5	6	7,7	8,8	9,9	[5,5,5,6,7,7,8,8,9,9]

Az 1. lépésnél megszámoljuk minden érték előfordulását a **freq**. tömbben. A 2. lépésben a rendezett sorozatot úgy építjük fel, hogy bejárjuk a **freq** tömböt $v = min$-től $v = max$-ig, és minden v értéket **freq[v – min]** alkalommal hozzáadunk. Nincs szükség az elemek összehasonlítására!

<def/> A **frekvenciatömbös rendezés** (Counting Sort) egy olyan rendezési algoritmus, amely egy ***n*** elemű egész számokból álló sorozatra (ahol az értékek a ***[min, max]*** intervallumba esnek) egy segédtömböt hoz létre: **freq[0..k–1]**, ahol k = max – min + 1, és **freq[v – min]** a v érték előfordulásainak számát jelöli. Ezután a rendezett sorozatot úgy állítja elő, hogy összehasonlítások nélkül újraépíti azt az elemek gyakorisága alapján. Időbonyolultság: **O(n + k)**, tárigény: **O(k)** extra memória.

<def/> Az algoritmus lépései:

1. Inicializálás – meghatározzuk a sorozatban a minimális és maximális értéket; majd létrehozzuk a freq segédtömböt, amelynek mérete k = max – min + 1, és minden elemét kezdetben 0-ra állítjuk.

2. Számolás – bejárjuk az a[0..n−1] sorozatot; minden a[i] elem esetén végrehajtjuk: freq[a[i] − min] ← freq[a[i] − min] + 1, azaz növeljük az adott értékhez tartozó gyakorisági számlálót.
3. Újraépítés – bejárjuk a freq tömböt 0-tól k−1-ig; minden indexnél az (index + min) értéket hozzáadjuk freq[index] alkalommal a rendezett eredményhez.
4. Befejezés – A sorozat elemei növekvő sorrendben vannak.

Pszeudokód algoritmus

```
beolvas n
minden i <- 0, n-1 végezd el
    beolvas a[i]
minden vége
// 1. lépés: freq inicializálása
min_v <- a[0];  max_v <- a[0]
minden i <- 1, n-1 végezd el
    ha a[i]<min_v: min_v<-a[i]
    ha vége
    ha a[i]>max_v: max_v<-a[i]
    ha vége
minden vége
k <- max_v - min_v + 1
minden i <- 0, k-1 végezd el
    freq[i] <- 0
minden vége
// 2. lépés: Számolás
minden i <- 0, n-1 végezd el
    freq[a[i]-min_v] <- freq[a[i]-min_v]+1
minden vége
// 3. lépés: Újraépítés
idx <- 0
minden v <- 0, k-1 végezd el
    minden j <- 1, freq[v] végezd el
        a[idx] <- v + min_v; idx<-idx+1
    minden vége
minden vége
minden i <- 0, n-1 végezd el
    kiír a[i]
minden vége
```

Python kód

```
# n>0
n = int(input("Elemek száma: "))
# sorozat beolvasása
a = []
for i in range(n):
    x = int(input("a["+str(i)+"] = "))
    a.append(x)
if len(a) == 0:
    print("A rendezett sorozat: []")
else:
    # 1. lépés – Az intervallum
    # megállapítása
    min_v = min(a)
    max_v = max(a)
    k = max_v - min_v + 1 freq = [0] * k
    # 2. lépés:  Számolás
    for x in a:
        freq[x - min_v] += 1
    # 3. lépés: Újraépítés
    idx = 0
    for v in range(k):
        for j in range(freq[v]):
            a[idx] = v + min_v
            idx += 1
    print("A rendezett sorozat:", a)
```

```
Elemek száma: 10
a[0]=7  a[1]=5  a[2]=8  a[3]=5  a[4]=9
a[5]=7  a[6]=6 a[7]=8  a[8]=5  a[9]=9
A rendezett sorozat: [5, 5, 5, 6, 7, 7, 8, 8, 9, 9]
```

⚠ **Figyelem!** A **min(a)** és **max(a)** Python függvények bejárják a sorozatot egyszer (O(n)) időben. Alternatív megoldásként a min/max értékeket is meghatározhatjuk ugyanebben a bejárásban, így a bonyolultság továbbra is O(n) marad. A **freq** tömb O(k) extra memóriát igényel. Az algoritmus stabil: az azonos elemek megőrzik a relatív sorrendjüket.

Kritérium	Kedvező eset	Kedvezőtlen eset	Megjegyzés
A számolási idő	O(n)	O(n)	mindig O(n) — egy bejárás
Eredeti idő/Újraépítési idő	O(k)	O(k)	az értéktartománytól függ, nem az adatoktól
Összes idő	O(n+k)	O(n+k)	nem függ a kezdeti sorrendtől!
Összehasonlítások	0	0	Nincs összehasonlítás az elemek között!
Memória	O(k)	O(k)	kiegészítő freq tömb



Jegyezd meg! Az előző módszerekkel ellentétben a bonyolultság nem függ a kezdeti sorrendtől – kedvező eset = kedvezőtlen eset = **O(n+k)**. Fontos, hogy a k érték kicsi legyen az n-hez képest.

Mikor használjuk a frekvenciatömbös rendezést?

Előnyök

- ✓ O(n+k) — leggyorsabb kis értéktartományban lévő egész számokra.
- ✓ Nincs összehasonlítás az elemek között — nem vonatkozik rá az összehasonlításon alapuló rendezési algoritmusokra jellemző alsó időkorlát.
- ✓ Stabil algoritmus — az azonos elemek megőrzik a relatív sorrendjüket.
- ✓ Egyszerű és intuitív implementáció.

Hátrányok

- × Csak egész számokra működik (nem működik sem valós számokra, sem karakterláncokra).
- × Nem hatékony, ha k >> n: az O(k) memória és inicializálás felesleges.
- × Összetett objektumokat közvetlenül nem rendez (átalakítás szükséges).
- × Nagy értéktartományok esetén magas a memóriaigény (például: $[0, 10^6]$)

Mi történik a határesetekben?
Egy helyes algoritmusnak nemcsak a gyakori példákon, hanem a **határesetekben** is jól kell működnie.

Határeset	Algoritmus viselkedése
Minden elem azonos	freq[0]=n, a többi=0; egy bejárás a freq tömbön; O(n+1)=O(n) ✓
Egy elem	k=1, freq[0]=1; a belső ciklus egyszer megy végbe — helyes ✓
Negatív értékek	min_v<0; offset=|min_v|; freq[a[i]+offset] helyes működés ✓
k >> n (pl. n=5, k=10^6)	O(k) fölösleges tárhelyhasználat; más algoritmust kell használni ×
Rendezett sorozat	Ugyanaz az időigény O(n+k) — Nem használja ki a kezdeti sorrendet ×

Pl. **Megoldott feladat.** *Egy forgalomfigyelő alkalmazás rögzíti egy óra alatt észlelt gépjárművek sebességét (km/h-ban, egész számokként): [72, 55, 88, 55, 100, 72, 61, 88, 55, 100, 72, 61]. Rendezd a sorozatot frekvenciatömb segítségével.*

Megoldás: min=55, max=100, k=46. Felépítjük a freq[0..45] (csak a jelenlévő értékekhez nullák):

Lépések	55 km/h	61 km/h	72 km/h	88 km/h	100 km/h	Eredmény
Kezdeti sorozat	3	2	3	2	2	**[72,55,88,55,100,72,61,88,55,100,72,61]**
freq[ ]	freq[0]=3	freq[6]=2	freq[17]=3	freq[33]=2	freq[45]=2	k=46, a többi=0
Végső sorozat ✓	55,55,55	61,61	72,72,72	88,88	100,100	**[55,55,55,61,61,72,72,72,88,88,100,100]**

Python megoldás

```
# Rövid, tömör megvalósítás
a = [72,55,88,55,100,72,61,88,55,100,72,61]
min_v, max_v = min(a), max(a)
freq = [0] * (max_v - min_v + 1)
for x in a:
  freq[x - min_v] += 1
idx = 0
for v, cnt in enumerate(freq):
     a[idx:idx+cnt] = [v + min_v] * cnt;  idx += cnt
print("A rendezett sorozat:", a)
```

```
A rendezett sorozat: [55, 55, 55, 61, 61, 72, 72, 72, 88, 88, 100, 100]
```

Megj. A frekvenciatömbös rendezés komplexitása ***O(n+k)***, míg a buborékos rendezésé ***O(n²)***.
Az előző algoritmus esetében a frekvenciás rendezés előnye csekély, de ez növekszik, ahogy az elemek száma növekszik.

Algoritmus	Idő (legrosszabb eset)	Összehasonlítások	Stabil	Adat típus
Minimumkiválasztásos	$O(n^2)$	$n(n-1)/2$	Nem	Minden
Buborékos	$O(n^2)$	$n(n-1)/2$	Igen	Minden
Frekvenciatömbös	**O(n+k)**	**0**	Igen	Egész

Jegyezd meg!

- ✔ A frekvenciatömbös rendezés minden érték előfordulását megszámolja, anélkül hogy az elemeket egymással összehasonlítaná.
- ✔ Bonyolultsága O(n+k) idő és O(k) extra memória, ahol k = max−min+1.
- ✔ Nem hasonlításon alapuló algoritmus lévén nem vonatkozik rá az Ω(n log n) alsó korlát.
- ✔ Az algoritmus stabil: az azonos elemek megőrzik a relatív sorrendjüket. Alkalmas jegyek, pontszámok, kódok stb. rendezésére.
- ✔ Valós alkalmazásokban a Python-beli **sort()** (Timsort, O(n log n)) beépített függvény ajánlott, amely bármilyen adattípusra működik.

Feladatlap 26

Készítsetek egy *Feladatlap26* nevű mappát. Oldjátok meg az alábbi feladatokat, a helyes válaszokat írjátok egy *valaszok.docx* dokumentumba.

I. *Válaszd ki a helyes választ (csak egy válasz helyes)*

1. Mi a frekvenciatömbös rendezés időbeli bonyolultsága ***n*** darab, ***[min, max]*** tartománybeli érték **és** *k=max−min+1 esetén*?

a. $O(n^2)$; **b.** $O(n \log n)$; **c.** $O(n + k)$; **d.** $O(k^2)$.

2. A [3, 1, 4, 1, 3, 2] listára alkalmazzuk a rendezést. Milyen értéke van a **freq[0]**-nak (min = 1)?

a. freq[0]=3; **b.** freq[0]=2; **c.** freq[0]=1; **d.** freq[0]=0.

II. *Határozd meg az alábbi kijelentések logikai értékét.*

1. A frekvenciatömbös rendezés nem hasonlítja össze a sorozat elemeit egymással. I / H

2. Ha ***k*** sokkal nagyobb, mint ***n*** (pl. n=5, $k=10^6$), akkor a frekvenciatömbös rendezés hatékonyabb a buborékrendezésnél. I / H

3. A bonyolultsága O(n+k) ugyanaz marad, függetlenül az elemek kezdeti sorrendjétől. I / H

III. *Kódkövetés – a program futtatása nélkül írja le, hogy mit jelenít meg az egyes print() utasítás*

```
a = [2, 5, 2, 1, 5, 1, 2]
min_v, max_v = min(a), max(a)
freq = [0] * (max_v - min_v + 1)
for x in a:
  freq[x - min_v] += 1
print("freq:", freq)
print("rendezve:", [v + min_v for v, c in enumerate(freq) for _ in range(c)])
```

freq: ..... . rendezve: ..... .

IV. ***Egészítsd ki a frekvenciatáblázatot a következő sorozat alapján [4, 2, 4, 1, 3, 2, 1] – min=1, max=4, k=4***

	val = 1	val = 2	val = 3	val = 4	A rendezett sorozat
freq[]	freq[0]=	freq[1]=	freq[2]=	freq[3]=	[..... , ... , ... , ... , ... , ... , ...]

V. ***Rövid válasz***

a. Miért nem alkalmazható közvetlenül a frekvenciatömbös rendezés a következő sorozatra [3.5, 1.2, 4.8]? Válasz: .....

b. Nevezz meg két olyan a valós életből vett területet, ahol a frekvenciatömbös rendezés különösen hatékonyan alkalmazható: 1. ..... . 2. ..... .

VI. ***Mini Python-projekt*** — Mentsd a projektet ***miniprojekt.py*** néven a Feladatlap26 mappába
Olvass be n darab, [0,100] intervallumba eső számot, rendezd őket frekvenciatömbös rendezés segítségével (a **sort()** használata nélkül), majd írasd ki:

a. növekvő sorrendbe rendezett listát;

b. a különböző értékek számát (freq[i] > 0);

c. egy üzenetet arról, hogy a lista eredetileg már rendezett volt-e.

VII. ***Mini Python-alkalmazás*** (***minialkalmazas.py*** a ***Feladatlap26 mappában***):
Készíts egy alkalmazást egy pontverseny kezelésére, ahol a pontszámok 0 és 50 közötti egész számok. Az alkalmazás ismétlődő menüt használjon (0 – Kilépés), és az alábbi feladatokat oldd meg:

a. pontszám hozzáadása;

b. a rangsor megjelenítése frekvenciatömbös rendezéssel;

c. A maximális pontszám és az azt elérő versenyzők számának kiírása;

d. A különböző pontszámok számának meghatározása.



Önértékelés!

Az *onertekeles26.docx* állományba írjátok be a megfelelő válaszokat, a következő jelöléseket használva: *1 – igen, 2 – nem, 3 – nem vagyok biztos benne*, majd mentsétek el a *Feladatlap26* mappába.

	A lecke végén tudom ...	***Igen***	***Nem***	***Nem biztos***
1.	Mi a frekvenciatömbös rendezés, és hogyan működik az algoritmus 3 lépése?			
2.	Hogyan épül fel a freq tömb, és mit jelent a freq[v – min]?			
3.	Mi az O(n + k) bonyolultság, és mikor hatékony ez az algoritmus az összehasonlításon alapuló módszerekkel szemben?			
4.	Mik a frekvenciatömbös rendezés előnyei és hátrányai (adattípus, tartomány, memória)?			
5.	Hogyan valósítom meg Pythonban a frekvenciatömbös rendezést beépített rendezőfüggvények nélkül?			

6. Nehézségeim voltak a következőkben: …

Ne felejtsétek el értékelni a viselkedéseteket és a tevékenységeteket az óra során!
(a 7. oldalon található viselkedés- és tevékenységfigyelő napló segítségével)

A lecke végén így éreztem magam:

6. Fejezetvégi felmérés

Kutass!

(Pontozás: 90 pont + 10 pont hivatalból)

Követelmény: Az alábbi feladatok megoldását írjátok a ***fejezetvegi_ertekeles6.docx*** fájlba. Mentsétek az összes, általatok lértehozott állományt a ***Fejezet6*** mappába.

I. Feleletválasztós kérdés – *Karikázd be a helyes választ* **[3 × 5p = 15p]**

1. Az alábbi időbonyolultságok közül melyik jelöli a leghatékonyabb algoritmust (a leglassubb növekedésűt)?

a. $O(n^2)$; **b.** $O(n)$; **c.** $O(\log n)$; **d.** $O(n^3)$.

2. Az alábbiak közül melyik az a rendezési módszer, amely egy segédváltozó segítségével leállíthatja a végrehajtást idő előtt, ha a lista már rendezett:

a. Minimumkiválasztásos; **b.** Buborékos; **c.** Frekvenciatömbös; **d** Mindhárom.

3. A frekvenciatömbös rendezés időbeli bonyolultsága:

a. $O(n^2)$; **b.** $O(n \log n)$; **c.** $O(n+k)$, ahol $k = max-min+1$; **d.** $O(k)$.

II. Igaz / Hamis – *[I] = Igaz • [H] = Hamis* **[3 × 3p = 9p]**

[I] [H] 1. A minimumkiválasztásos rendezési algoritmus mindig n(n−1)/2 összehasonlítást végez, függetlenül a sorozat elemeinek kezdeti sorrendjétől.

[I] [H] 2. A buborékrendezés stabil: az egyenlő elemek nem változtatják meg egymáshoz viszonyított sorrendjüket.

[I] [H] 3. Egy algoritmus időbeli bonyolultságát másodpercekben mérik, mivel ez független a hardvertől.

III. Python kód elemzése **[4p + 6p + 8p = 18p]**

```
n = int(input("Tagok száma: "))
a, b = 0, 1
print(a, end=" ")
if n > 1:
    print(b, end=" ")
    for i in range(2, n):
        a, b = b, a + b  # (X)
        print(b, end=" ")
```

a. Mit ír ki n = 8-ra? Írjátok le mind a 8 tagot. .....

b. Mi az (X) utasítás szerepe? Miért nincs szükség segédváltozóra? .....

c. Mi az időbonyolultság? Indokold válaszod, legfeljebb 2 mondatban. .....

IV. Töltsd ki az üres cellákat **[30p]**

1. Buborékrendezés. Rendezd növekvő sorrendbe [6, 1, 4, 8, 3]. Egészítsd ki a lenti táblázatot minden egyes bejárás után, valamint töltsd ki a cserék számát is. **[4 × 3p + 2p + 4p]**

Lépések	**[6, 1, 4, 8, 3]**					**Cserék száma**
Kezdet	**6**	**1**	**4**	**8**	**3**	.....
1. bejárás						.....
2. bejárás						.....
3. bejárás						.....
4. bejárás						.....
Utolsó bejárás ✓						**Összesen: .....**

Korai leállás? Indokold: .....

2. Frekvenciatömb. **[8,6,9,6,10,8,7,9,6,8]** – Egészítsd ki a **freq[]** tömböt és a rendezett sorozatot. **[6p + 4p + 2p]**

Jegy / freq[]	**6**	**7**	**8**	**9**	**10**
freq[ ]					

Rendezett sorozat: .....

min_v= .....

max_v= .....

k=.....

V. Írj egy Python alkalmazást: **[18p]**

Olvasd be az ***n*** –et, és generáld a következő sorozat első n tagját $\mathbf{a_0 = 5\ a_1 = 2,\ a_n = a_{n-1} + 2 \cdot a_{n-2}\ (n \geq 2)}$.

Példa: n=6 → 5 2 12 16 40 72. Vigyázz az n=1-re és n=2-re.

Pontozás: I - 15p + II - 9p + III - 18p + IV - 30p + V - 18p + hivatalból - 10p = 100p.



Projekt

(100 pont)

„A rendezési összehasonlító” – Python-alkalmazás a rendezési módszerek elemzéséhez

Követelmények:

- (1) Valósítsátok meg a 3 algoritmust
- (2) Interaktív menü legalább 6 opcióval
- (3) Minden rendezésnél jelenítse meg az összehasonlításokat/cseréket/bonyolultságot
- (4) Teszteljétek n=10,50,100
- (5) Hatékonysági összehasonlító táblázat
- (6) Megjegyzések a kódhoz
- (7) Értékelési ***jelentes.docx***

Fájlok: ***rendezesi_projekt.py*** és ***jelentes.docx***

Pontozás (100 pont): 30p – 3 algoritmus, **15p** – menü, **20p** – kiiratás, **15p** – összehasonlító táblázat, **10p** – Megjegyzések a kódhoz.



Kutass!

(100 pont)

Kérdés: Hogyan változik a három módszer hatékonysága az adatok típusától, méretétől és kezdeti sorrendjétől függően? Hipotézis: .....

Lépések:

- (1) Hozzatok létre 3 adatsort (véletlenszerű/↑/↓) n=10, 50, 100 értékekre
- (2) Írjátok meg az ***kutatas.py*** fájlt — a 3 algoritmus + műveletek rögzítése
- (3) töltsétek ki a táblázatot (9 sorozat)
- (4) Elemezzétek az eredményeket, ellenőrizzétek a hipotézist
- (5) Szerkesszétek meg az ***kutatas6.docx*** fájlt a következtetésekkel és ajánlásokkal

Fájlok: ***kutatas.py*** és ***kutatas6.docx***.

Pontozás (100 pont): 10p – hipotézis, **25p** – program, **25p** – táblázat adatokkal, **20p** – elemzés, **15p** – következtetések, **5p** – jelentes.



Portfólió

(100 pont)

E fejezet végén a tanuló személyes portfóliójának a ***Fejezet6*** mappában a következő anyagokat kell tartalmaznia:

- ***Feladatlap22*** mappa, és a ***valaszok.docx, alkalmazas.py, esettanulmany.py, onertekeles22.docx*** fájlok;
- ***Feladatlap23*** mappa, és a ***valaszok.docx, alkalmazas1.py … alkalmazas9.py, onertekeles23.docx*** fájlok;
- ***Feladatlap24*** mappa, és a ***valaszok.rtf, miniproiect.py, miniaplikacio.py, onertekeles24.docx*** fájlok;
- ***Feladatlap25*** mappa, és a ***valaszok.docx, vizualizari.py, captura.jpg, onertekeles25.docx*** fájlok;
- ***Feladatlap26*** mappa, és a ***valaszok.docx, miniproiect.py, miniaplikacio.py, onertekeles26.docx*** fájlok;
- ***Projekt6*** mappa, és a ***rendezesi_projekt.py, ertekeles.docx, kutass6.docx, kutass6.py*** fájlok;
- Fájlok: ***fejezetvegi_ertekeles6.docx***, ***rendezesi_projekt***, ***ertekeles.docx***, ***kutass.py***.

Értékelés módja:

- Megvan a ***Fejezet6*** mappa a megadott könyvtárszerkezettel – **20p**
- Megvan:
  - A fájlok több mint 95%-a – **80p**
  - A fájlok 75–94%-a – **60p**
  - A fájlok 50-74%-a – **40p**
  - A fájlok 25-49%-a – **20p**
  - 25% alatt – **0p**

Évzáró felmérés

Hozzátok létre az **evzaro_felmeres** mappát. Oldjátok meg az alábbi I. tételt, és írjátok le a válaszokat egy *valaszok.docx* nevű fájlba. Ezt a fájlt és az a kért Python programokat az ***evzaro_felmeres*** mappába kell menteni.

I. TÉTEL **(20 pont)**

A. Válasszátok ki a helyes választ: (6 pont)

1. A Pythonban az ismétlődő szerkezet:
a) while **b)** def **c)** if **d)** abs

2. A Python-függvény, amely visszaadja a lista elemeinek számát:
a) count() **b)** len() **c)** size() **d)** sum()

3. A FIFO működési elv jellemzi:
a) a vermet **b)** a gyakorisági listát **c)** a várakozási sort **d)** a változót

4. A grafikus felületekhez használt Python-könyvtár:
a) random **b)** math **c)** tkinter **d)** os

5. A buborékrendezés bonyolultsága a legrosszabb esetben:
a) O(1) **b)** O(log n) **c)** O(n) **d)** $O(n^2)$

6. Az alprogramon belül definiált változók:
a) globálisak **b)** lokálisak **c)** állandók **d)** előző tagok alapján felépülő

B. Töltsétek ki az üres helyeket **(8** pont)

1. Egy algoritmus grafikus ábrázolása a következővel történik ..... .

2. A szöveges fájlok megnyitásához Pythonban használt függvény/mechanizmus a következő ..... .

3. A lista elemeit helyben (in-place) rendező módszer a ..... .

4. Az algoritmusok bonyolultságának becsléséhez használt jelölés a ..... .

5. A Pythonban egy alprogram definiálásához használt utasítás a ..... .

6. A Tkinter komponens, amelyet egy soros szöveg beírására használnak, a ..... .

7. A **for i in range(5)** ciklus iterációinak száma ..... .

8. Az abszolút érték kiszámítására használt Python függvény a ..... .

C. Határozzátok meg az állítások igazságértékét (6 pont)

1. A Python-lista módosítható.

2. A pop() metódus eltávolítja a lista utolsó elemét, ha nem kap paramétereket.

3. A Tkinter lehetővé teszi grafikus felületek készítését Pythonban.

4. A helyi változók bármelyik alprogramban használhatók.

5. A buborékrendezés összehasonlítja a szomszédos pozíciókban lévő elemeket.

6. A round() függvény a valós számok kerekítését végzi el.

II. TÉTEL **(70 pont)**

A. Egy online áruházban az eladások nyomon követésére van szükség. Így a vezető ismeri az eladások értékét a nyomon követés első napján, majd a következő napokon a vezető feljegyzi azt az egész számot is, amely a jelenlegi nap és az előző nap eladási értéke közötti különbséget jelenti. Szeretné tudni, hány napon nőtt az eladás az előző naphoz képest, valamint hány napon volt az eladás nagyobb, mint az első napon. Írjatok egy ***uzlet.py*** nevű programot, amely segít a vezetőnek. Az ***uzlet.in*** bemeneti fájl egyetlen sorban tartalmazza az első nap forgalmát (természetes szám), majd a egymást követő napi különbségeket (szóközzel elválasztott egész számok). A kért számokat egy sorban, szóközzel elválasztva meg kell jeleníteni a képernyőn. Például, ha a menedzser a következő értékeket jegyzi fel: 100 -10 20 -5, akkor a képernyőn a 1 2 számok jelennek meg. Indokolja 2-3 sorban a használt adattípusokat és az algoritmus hatékonyságát.

B. Írjatok egy ***numerikus.py*** nevű Python programot, amely a következő függvényeket tartalmazza:

- LNKO(a, b) – kiszámítja és visszaadja a két természetes szám, a és b legnagyobb közös osztóját az euklideszi algoritmus segítségével.
- TUKOR(n) – kiszámítja és visszaadja a n természetes szám tükörképét.
- PRIM(n) – ellenőrzi, hogy a n természetes szám prímszám-e, és 1 vagy 0 értéket ad vissza.
- ALAKIT(n) – kiszámítja és visszaadja a n természetes szám 2-es alapú ábrázolását.

A program tartalmazni fog egy, a **tkinter** könyvtárral készített grafikus felületet (amely egy menüsort is tartalmaz majd). Ezen keresztül a felhasználó megadhat egy **n** darab természetes számból álló listát (n≥3). Egy gomb megnyomására a program a fent definiált függvények segítségével a következő műveleteket hajtja végre:

- kiszámítja a lista összes elemének legkisebb közös többszörösét (LKKT);
- meghatározza a lista elemeinek összegének tükörképét;
- megjeleníti a lista legnagyobb számát, amelynek minden számjegye különböző (ha ilyen nincs, akkor **-1**-et jelenít meg);
- ellenőrzi, hogy az összeg tükörképe prímszám-e, és megjeleníti a „prímszám" vagy „nem prím" üzenetet;
- megjeleníti ennek a tükörképnek a **2**-es számrendszerbeli alakját;
- növekvő sorrendbe rendezi a listát az egyes elemek számjegyeinek összege szerint (egyenlőség esetén az eredeti sorrendet tartja meg);
- eltávolítja a listából a **0**-val egyenlő elemeket, és megjeleníti az eredményül kapott listát.

Az eredmények az ablak szövegmezőjében (Text típusú widget) jelennek meg, és a **numerikus.out** fájlba kerülnek, minden eredmény egy-egy sorban.

***Értékelés módja*:** Összesen – **100 pont** (**10 pont hivatalból**)
I. Tétel – 20 pont (minden helyes válasz 1 pontot ér)
I. A: **1**-a, **2**-b, **3**-c, **4**-c, **5**-d, **6**-b
I. B: 1-folyamatábra/flowchart, 2-open(), 3-sort(), 4-O( ) jelölés vagy hasonló, 5-def, 6-Entry, 7-5, 8-abs()
I. C: **1**-A, **2**-A, **3**-A, **4**-F, **5**-A, **6**-A
II. Tétel – 70 pont (II.A – 20p, II.B – 50p)
II. A: A fájl helyes beolvasása – **4p**, a kizárólag pozitív különbségek helyes megszámlálása – **5p**, a napi eladások helyreállítása – **3p**, összehasonlítás az első nappal – **3p**, az eredmények megjelenítése – **3p**, indoklás – **2p**.
II. B: A 4 függvény – **16p** (**4p**/függvény), a main-ben végzett feldolgozások – **18p**, fájlba írás – **4p**, tkinter felület + menü – **12p**.

ÚTMUTATÓK ÉS MEGOLDÁSOK

1. Fejezet

Feladatlap 1:

1. ***a)*** igaz; ***b)*** hamis; ***c)*** hamis; ***d)*** igaz. **2.** 1 – (Töltsd meg a kék poharat gyümölcslével), 2 - (Töltsd meg a sárga poharat vízzel), 3 - (Vegyél egy üres lila poharat), 4 - (Öntsd a kék pohárból a lila pohárba a gyümölcslevet), 5 - (Öntsd a sárga pohárból a kék pohárba a vizet), 6 - (Öntsd a lila pohárból a sárga pohárba a gyümölcslevet)

Feladatlap 2:

1. ***a)*** 175 és 2; ***b)*** az algoritmus meghatározza a beolvasott 3 szám közül a legnagyobbat. **2.** ***a)*** hamis; ***b)*** igaz; ***c)*** hamis; ***d)*** igaz.

Feladatlap 3:

1. Vegyük ki egy háromjegyű szám számjegyeit (az utolsót, a másodikat és az elsőt), majd alkossunk belőlük egy új számot, amelyben a számjegyek fordított sorrendben szerepelnek. Gyakorlatilag egy pontosan háromjegyű számot fordítunk meg.

3. Fejezet

Feladatlap 12:

1. Az osztály egy sablon, az objektum pedig egy konkrét példány, amelyet az adott sablon alapján hoztak létre. Hasonlat: Az osztály olyan, mint egy süteményrecept – leírja az összetevőket és a lépéseket, de önmagában nem egy sütemény. Az objektum pedig az a konkrét sütemény, amelyet az adott recept alapján készítettek. Ugyanabból a receptből sokféle süteményt (különböző objektumot) készíthetsz. **2.** b)

5. Fejezet

Feladatlap 18:

1. c **2.** b **3.** közvetlen és szekvenciális; lehetőség elemek hozzáadására vagy eltávolítására a listából. **4.** A lista összetett adatstruktúra, mivel több azonos típusú elemet tartalmaz; a lista elemei egy meghatározott sorrendben vannak tárolva, minden elemnek van egy pozíciója (indexe); a lista véges számú elemet tartalmaz, mivel egy adott pillanatban meghatározott méretű, még akkor is, ha ez elemek hozzáadásával vagy eltávolításával módosítható. **5.** a. Lista szükséges, mivel kijelentik, hogy később további kérdések merülnek fel az adatokkal kapcsolatban, ezért minden értéket tárolni kell.

Feladatlap 19:

1. b **2.** a - verem, b - sor, c - sor, d - sor

Feladatlap 20:

1.d**2.**a**3.**1.eset:akimenet[‘fehér’,‘zöld’,‘fehér’,‘zöld’]2.eset: az 1-es indexű elem eltávolításra kerül, a fennmaradó és kimenetként megjelenített lista [10, 30, 40].

4. ***a)*** L.append(‘Sara’) ***b)*** L.insert(0, ‘Viktor’)
c) L.remove(‘Emma’) ***d)*** L.extend([‘Teo’, ‘Emma’, ‘Anna’])
e) print(L).

Feladatlap 21:

1. c **2.** c **3.** 1. eset: hamis**,** igaz; 2. eset: igaz **4.** Banan **5.** a. print(L) b. print(L[::-1]) c. print(sorted(L)) d. elsők 2: print(L[:2]) utolsók 2: print(L[-2:])

6. Fejezet

Feladatlap 22:

I: 1. b **2.** b **3.** b **4.** b **II:** O(1) – állandó idő; O(n^2) – egymásba ágyazott ciklusok; – legrosszabb eset. **III:** egy for range(n)→**O(n)**; kettő for egybeágyazva range(n)→**O(n^2)** **IV: 1.** műveletszám/adatméret. **2.** négyzetesen nő. **3.** ciklusok→O(n^2); set→O(n). **V:** dupla ciklusok→O(n^2); set→O(n).

Feladatlap 23:

I.1: a. 11,14,17 **b.** 24,48,96 **c.** 8,13,21 **d.** 36,49,64. **I.2: a.** 7,12,17 **b.** 2,4,8 **c.** 0,1,1,2,3,5. **II: 3.** függ az előző tagoktól. **4.** 2.rendű. **III: 5.** a. 2,5,8,11 **b.** 5,10,20,40 **c.** 1,4,5,9,14. **6: a.** 21,28,36 **b.** Δ nő 1-el **c.** rekurrens. **7:** képlet→direkt; rekurencia→az előzőkből. **8: a.** 665,50 RON **b.** $a_n=a_{n-1}\times 1.1$ **c.** rekurrens. **9: a.** Δ nő 1-el **b.** 22 **c.** rekurrens. **10: a.** 1,5,14,30,55 **b.** $T_n=T_{n-1}+n^2$ **c.** rekurrens (és n(n+1)(2n+1)/6).

Feladatlap 24:

I: 1. b(10) **2.** a [1,2,8,6,4] **3.** kód1→ [5, 3, 8, 1, 6]; kód2→ [1, 3, 5, 6, 8] 0 **4.** O(n^2)–nem hatékony. **5:** P1:55↔320; P2:85 fix; P3:140↔210; P4:170↔320; P5:210 fix; P6:320↔390 → [390,320,210,170,140,85,55].

Feladatlap 25:

I: 1. c **2.** c **3.** d **4.** b. **II.5:** elemek; felcseréli; O(n^2). **6:** A↔b, B↔a, C↔c. **7:** kicsi/majdnem rendezett→hatékony. **8:** Bubble–stabil; Kiválasztásos–instabil. **9:** stabil. **10:** T1→[-2,1,0,5,7]/3; T2→[-2,0,1,5,7]/1; T3–T5→0 → megállás. **11:** [1,2,4,5].

Feladatlap 26:

I: 1. c **2.** b. **II: 1.** A **2.** F **3.** A. **III:** a=[2,5,2,1,5,1,2]: **freq:[2,3,0,0,2]**; rendezve: **1 1 2 2 2 5 5**. **IV:** [4,2,4,1,3,2,1]: **freq[0..3]=[2,2,1,2]**; rendezve: **1 1 2 2 3 4 4**. **V: a.** A frekvenciás rendezés közvetlenül csak egész számokra alkalmazható, míg a 3,5, 1,2 és 4,8 értékek valós számok. **b.** [3.5…]→nem; [0..100]→igen.

Informatică în limba maghiară pentru școlile/ secțiile cu predare în limba maghiară, clasa a IX-a, curriculum de specialitate (CS), filiera teoretică, profilul real, specializarea matematică-informatică.

Editat în limba maghiară.



www.manuale.edu.ro