



ALGORITMUSOK ÉS ADATSZERKEZETEK

Python 2.

Oktatási hét	Témakör - Gyakorlat
1	Bevezetés a Python programozási nyelvbe, alapvető szintaxis és kifejezések, változók, típusok és alapvető műveletek, alapvető be- és kimenet, feltételes elágazások (if, elif, else)
2	Ciklusok (for, while), listák és egyszerű iterációk
3	Függvények és modulok, függvények definiálása és hívása, paraméterek és visszatérési értékek, beépített modulok használata, fájlkezelés, kivételkezelés
4	Algoritmusok és optimalizálási stratégiák rekurziós példán keresztül (brute force, dinamikus programozás, mohó algoritmusok, visszalépéses keresés)
5	Adatszerkezetek I. - Lista, tömb, sor, verem, listaműveletek, queue és stack implementációja statikusan (tömb) és dinamikusan (lista), gyakorlati feladatok
6	Adatszerkezetek II. – Halmaz és szótár, halmazműveletek és szótárműveletek, gyakorlati feladatok
7	Zárthelyi dolgozat 1.
8	Adatszerkezetek III. – Láncolt lista implementálása
9	Adatszerkezetek IV. – Bináris keresőfa implementálása
10	Gráfok alapjai, reprezentációik, gráfalgoritmusok implementálása
11	<i>Rektori szünet</i>
12	<i>Féléves beadandó feladat leadása és bemutatása</i>
13	Zárthelyi dolgozat 2.
14	Félév zárása, javító/pótló zárthelyi dolgozat



Függvények és modulok

*definiálás, paraméterek és visszatérési érték, beépített modulok
modularitás, újrahasználatosság, könnyebb olvashatóság*

- Egy függvény egy kódblokk, amelyet névvel látunk el.
- A **def** kulcsszóval definiáljuk.
 - ezt követi a függvény neve, zárójelben a paraméterei, kettőspont után a hozzá tartozó kódblokk

```
def függvény_neve(paraméter1, paraméter2, ...):  
    # Függvény törzse, kódblokk  
    return visszatérési_érték
```

def:

- Kulcsszó, a függvény definíciójának kezdete.

függvény_neve:

- A függvény egyedi neve, betűkkel és aláhúzással kezdődhet.

paraméter1, paraméter2, ...:

- A függvénynek átadott értékek. (opcionális)

return visszatérési_érték:

- A függvény által visszaadott érték.
- Ha nincs utasítás, a függvény None értéket ad vissza.

Előnyök:

- Kód újrahasználatosság: *Egy függvényt többször is meghívhatunk különböző bemeneti adatokkal.*
- Kód szervezés, olvashatóság: *A függvények kisebb, könnyebben kezelhető feladatokra, logikai egységekre bontják a programot.*
- Hibakeresés

Hátrányok:

- Túl bonyolult függvények nehezen érthetővé válhatnak.
- Túl sok függvény használatával a kód áttekinthetetlenné válhat.

FÜGGVÉNYEK FELHASZNÁLÁSA PÉLDÁK

- Ismételt műveletek elvégzése
- Kód szervezése
- Webalkalmazások
 - Bejelentkezés: `bejelentkezés(felhasznalonev, jelszo)`
 - Adatbázis lekérdezés: `kereses_adatbazisban(kerdoszo)`
 - E-mail küldés: `kuldes_email(cimzett, targy, uzenet)`
 - Fájl feltöltés: `feltoltes_fajl(fajlnev)`
- Adatfeldolgozás és elemzés:
 - Adatok betöltése: `betoltes_adatok(fajlnev)`
 - Adatok vizualizálása: `vizualizalas_adatok(adatok)`
 - Predikciók készítése: `predikcio(modell, uj_adatok)`
- Web scraping: `kinyer_adatok_weboldalrol(url)`
- Játékokban karakter mozgatása: `mozgat_karakter(irany)`



Feladat:

- 7. sorban hívjuk meg a saját_fuggveny alprogramot a függvény nevével és az üres paraméterlistával ()
- 12. sorban módosítjuk a második alprogram kódblokkját, illesszünk be egy kiíró eljárást, mely kiírja az első paramétert nagybetűsen (upper()), és a második paramétert eredeti formában
- 15. sorban hívjuk meg a második alprogramot tetszőleges szöveges értékű paraméterekkel („” használata!, paramétereket vesszővel választjuk el)

ALAPÉRTELMEZETT PARAMÉTEREK

- Ha egy függvénynek több paramétere van, akkor egyeseknek adhatunk alapértelmezett értéket.
 - *ha függvényhíváskor nem adunk meg értéket az adott paraméterhez, akkor az alapértelmezett értéket fogja használni a program*

Feladat:

- 10. sorban hívjuk meg a függvényt paraméterek megadása nélkül
- 12. sorban hívjuk meg a függvényt egy paraméterrel, melynek értéke legyen 2. Figyeljük meg miben változik a kiírás.
- 14. sorban hívjuk meg a függvényt (4,4) paraméterekkel, majd figyeljük meg ismét a változást.

KULCSSZAVAS ARGUMENTUMOK

- A kulcsszavas argumentumok segítségével a paramétereket név szerint adhatjuk át a függvénynek, így nem kell figyelniük a paraméterek sorrendjére.

Python_02_02_megoldás.py Python_02_02.py Python_02_kulcsszavasarg.py

```
1 # Kulcsszavas argumentumok
2 def személy(nev, életkor, szín):
3     print(f"A személy neve: {nev}, életkora: {életkor}, kedvenc színe: {szín}")
4 # A paraméterek sorrendje felcserélhető
5 személy(eletkor=30, szín="lila", nev="Péter")
6
```

A személy neve: Péter, életkora: 30, kedvenc színe: lila

Process finished with exit code 0

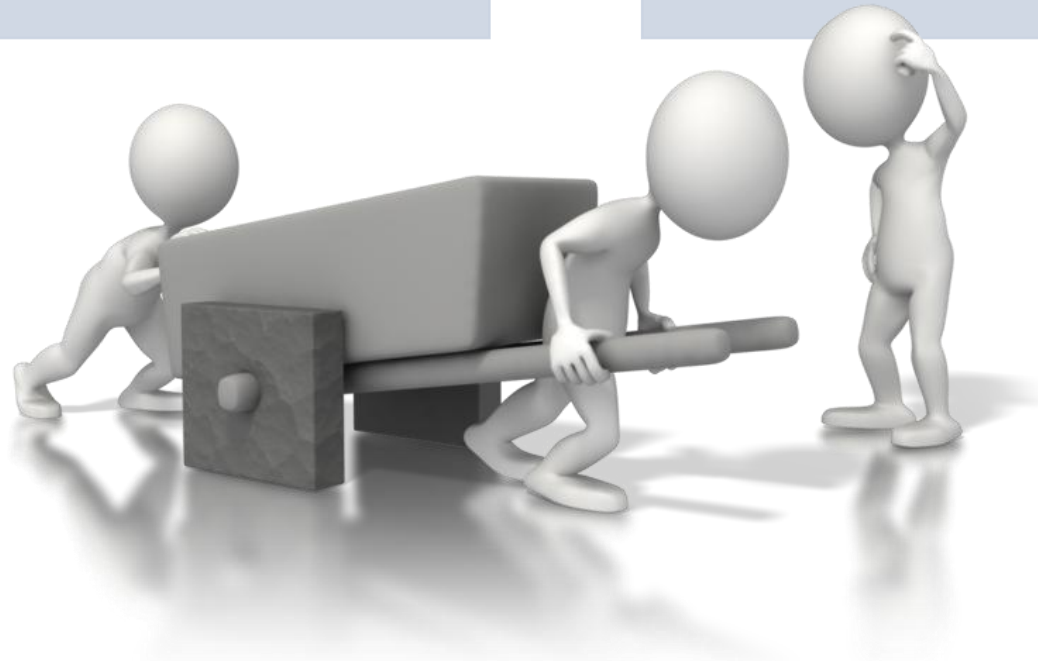
VÁLTOZÓ SZÁMÚ ARGUMENTUMOK

`*args`

Tetszőleges számú pozícionális argumentumot gyűjt össze egy tuple-be.

`**kwargs`

Tetszőleges számú kulcsszavas argumentumot gyűjt össze egy dictionary-be.



Feladat:

- 4. sorban összegezzük a számok tuple értékeit a megadott ossz változóba
- 7. sorban keressük meg a szintaktikai hibát, javítsuk a kódot
- 10. sorban ellenőrizzük a *szinek változó típusát a type() metódus segítségével, írjuk is ki az eredményt a képernyőre
- 17. sorba írjuk ki az **adatok változó típusát, ellenőrizzük az eredményt, milyen típus lett?
- 21. sorba töltsük fel a személy_adatok értékét a következő értékpárokkal:
 - név="Péter", életkor=30, varos="Budapest", szin="lila"

ÉRTÉK SZERINTI ÉS „CÍM SZERINTI” PARAMÉTEREK

#érték és címszerinti paraméterre példa

```
def DuplaOsszeg(szam1, szam2):
```

```
    szam1 = szam1 * 2
```

```
    szam2 = szam2 * 2
```

```
    return szam1 + szam2
```

```
def DuplaOsszeg2(c, d):
```

```
    c[0] = c[0] * 2
```

```
    d[0] = d[0] * 2
```

```
    return c[0] + d[0]
```

```
a = 6
```

```
b = 4
```

```
print("\nA függvény hívása előtt a változók értéke...")
```

```
print("\na értéke: {0}\nb értéke: {1}".format(a, b))
```

```
# átadja a és b ÉRTÉKÉT a függvényen belül szam1 és szam2 - nek
```

```
c_ertek = DuplaOsszeg(a, b)
```

```
print("\nA függvény hívása (érték sz.) után a változók értéke...")
```

```
print("\na értéke: {0}\nb értéke: {1}\n\neredmény: {2}".format(a, b, c_ertek))
```

```
c = [6]
```

```
d = [4]
```

```
print("\nA függvény hívása előtt a változók értéke...")
```

```
print("\nc értéke: {0}\nd értéke: {1}".format(c, d))
```

```
#a és b CÍMÉT adja át a függvénynek (listán keresztül - erőltetett megoldás)
```

```
c_ref = DuplaOsszeg2(c, d)
```

```
print("\nA függvény hívása (cím sz.) után a változók értéke...")
```

```
print("\nc értéke: {0}\nd értéke: {1}\n\neredmény: {2}".format(c, d, c_ref))
```

ÉRTÉK SZERINTI ÉS „CÍM SZERINTI” PARAMÉTEREK - EREDMÉNYE

A függvény hívása előtt a változók értéke...

a értéke: 6

b értéke: 4

A függvény hívása (érték sz.) után a változók értéke...

a értéke: 6

b értéke: 4

eredmény: 20

A függvény hívása előtt a változók értéke...

c értéke: [6]

d értéke: [4]

A függvény hívása (cím sz.) után a változók értéke...

c értéke: [12]

d értéke: [8]

eredmény: 20

REKURZÍV FÜGGVÉNY

emlékeztető

- A rekurzív függvények önmagukat hívják meg.

```
def faktorialis_rekurziv(n):  
    if n > 1:      # báziskritérium - leállási feltétel, amikor már nem hívja meg saját magát  
        return n * faktorialis_rekurziv(n - 1) # minden visszatéréskor meghívja saját magát az új értékkel  
    else:  
        return 1 # megállítja a rekurzív meghívást! Nélküle végtelen ciklus lenne...
```

- Amikor meghívjuk a függvényt n értékkel, rekurzívan fogja saját magát meghívni az n szám csökkentésével ($n-1$).
- Minden egyes függvényhívás megsokszorozza a számot az aktuális faktoriális számmal, amíg az n eléri az 1-et, mint határérték, báziskritérium, leállási feltétel.
 - *Minden rekurciónak rendelkeznie kell egy ilyen értékkel, különben végtelenszer hívja meg magát.*

Levezetve a függvényt...

$n = 4$ esetén

fakt(4)

$4 * \text{fakt}(3)$

$4 * 3 * \text{fakt}(2)$

$4 * 3 * 2 * \text{fakt}(1)$

n elérte a báziskritériumot, ezért elkezdi kiértékelni visszafelé haladva a fakt(n) értékeit

$4 * 3 * 2 * 1$

$4 * 3 * 2$

$4 * 6$

24

```
def fakt(n):  
    if n > 1:  
        return n * fakt(n-1)  
    else:  
        return 1
```

1. hívás $\rightarrow 4 - \text{el}$

2. hívás $\rightarrow 3 - \text{al}$

3. hívás $\rightarrow 2 - \text{el}$

4. hívás $\rightarrow 1 - \text{el}$

4. hívás kiértékelése $\leftarrow 1$

3. hívás kiértékelése $\leftarrow 2 * 1 = 2$

2. hívás kiértékelése $\leftarrow 3 * 2 = 6$

1. hívás kiértékelése $\leftarrow 4 * 6 = 24$

REKURZÍV FÜGGVÉNYEK GYAKORLATI ALKALMAZÁSAIRA PÉLDÁK

Adatszerkezetek bejárása

- Bináris fák bejárása: inorder, preorder, postorder bejárás
- Gráfok bejárása: mélységi első (DFS) és szélességi első (BFS) keresés

Algoritmusok

- Gyorsrendezés (Quicksort), összeolvasztásos rendezés (Merge sort)
- Hanoi tornyai, fibonacci-sorozat

Matematikai problémák

- Faktoriális számítás, hatványozás

Játékok és mesterséges intelligencia

- Játékfák bejárása: minimax algoritmus, alfa-béta metszés
- Rekurzív keresési algoritmusok: A-csillag algoritmus

MIÉRT HASZNOSAK A REKURZÍV FÜGGVÉNYEK?

- Egyszerű megoldást nyújtanak olyan problémákra, amelyek önmaguk kisebb példányaira bonthatók.
 - **Önmagába ágyazódó problémák** - Ha egy probléma kisebb, hasonló problémákra bontható.
 - Például:
 - *Faktoriális számítás: Minden faktoriális kiszámolható az egyel kisebb szám faktoriálisának és a szám szorzataként.*
 - *Fibonacci-sorozat: Minden szám az előző két szám összege.*
 - *Bináris fák bejárása: Egy fa minden részfája egy kisebb fa.*
- Elegánssá teszik a kódot és könnyebben érthetővé teszik a megoldást.
- Bizonyos problémáknál hatékonyabbak az iteratív megoldásoknál.

A REKURZIÓ HATÉKONYSÁGA

- **Memóriahasználat:**

- A rekurzív hívások minden egyes szintje memóriát foglal le a veremben.
- Nagyméretű problémák esetén ez memóriaproblémákhoz vezethet.

- **Futási idő:**

- A rekurzív hívások overheadje (plusz erőforrások – idő, memória) miatt a rekurzív megoldások gyakran lassabbak lehetnek, mint az iteratív megoldások.

- **Processzorigény:**

- A rekurzív hívások a processzort is jobban leterhelik.

- **Példák:**

- **Gyorsrendezés:** A **rekurzív** megoldás általában **hatékonyabb**, mint az iteratív.
- **Fibonacci-sorozat:** Az **iteratív** megoldás **hatékonyabb**, mert elkerüli a felesleges rekurzív hívásokat.

MIKOR ÉRDEMES ITERATÍV FÜGGVÉNYT HASZNÁLNI?

- **Egyszerű ciklusok:**
 - Ha egy feladat egyszerűen egy ciklussal megoldható, az iteratív megoldás általában hatékonyabb.
- **Memóriahatékony megoldások:**
 - Ha a memória kritikus tényező, az iteratív megoldás előnyösebb.
- **Nagy méretű problémák:**
 - Nagyméretű adatszerkezeteken a rekurzió könnyen stack overflow-hoz vezethet.
- *Amikor a rekurzió mögött rejlő logikát nehéz követni. 😊*

REKURZÍV VS ITERATÍV

Kritérium	Rekurzív függvény	Iteratív függvény
Önmagába ágyazódó problémák	Kiváló	Jó, de kevésbé elegáns
Adatszerkezetek bejárása	Kiváló (fák, gráfok)	Jó (lineáris adatszerkezetek)
Memóriahasználat	Gyakran magasabb	Általában alacsonyabb
Futási idő	Gyakran lassabb (<i>overhead miatt</i>)	Általában gyorsabb
Kód olvashatósága	Néha egyszerűbb	Néha bonyolultabb


Feladat:

- Egészítsük ki az első N természetes szám összegének kiszámításához az iteratív és rekurzív függvényeket.
- A 3. és 4. sorban egészítsük ki az iteratív függvényt, hogy elvégezze a sorozat elemeinek összegzését.
 - *3. sorban írjuk meg a for ciklust mely 1-től n -ig ismétli a 4. sor műveletét*
 - *a 4. sorban az eredmény változó értékét növeljük a ciklusváltozó értékével (i)*
- 10. sorban egészítsük ki a rekurzív függvényt a visszatérési értékkel
 - *mivel $n=0$ a báziskritérium, így a rekurzív hívásoknak gondoskodnia kell n folyamatos csökkentéséről*
 - *a visszatérési érték összegzi n értékeit (n -től 1-ig)*
- *Teszteljük az elkészült programot...*

SCOPE - LÁTHATÓSÁG

- Egy változó csak a létrehozott régió belsejéből érhető el. Ezt nevezik **hatókörnek**.
- Típusai:
 - **Lokális és globális** változók
 - Láthatóság – régió szerint:
 - Function – függvény
 - Module – modul
 - Class – osztály
- Az **IF szerkezet, FOR és WHILE** ciklusok **nem rendelkeznek saját scope-al**, hatókörrel *(eltérő a többi nyelvhez képest)*.
 - Ezekben a szerkezetekben **deklarált változók globális** hatókörrel rendelkeznek. *(kivéve ha bekerülnek modulba, függvénybe, osztályba)*

SCOPE – LÁTHATÓSÁG - PÉLDA

 Python_02_scopepld.py ×

```

1  # lokális változó függvényen belül
2  print("-----lokális változó függvényen belül-----")
3  def lokalis_valtozo(): # függvényen belül lokális hatókör 1 usage
4      x = 12 # függvényen belül látható, érhető el az x, y és eredmény változó
5      y = 25 # csak a függvény idejére ÉL utána törlődik a memóriából
6      eredmény = x + y
7      print("Az {0} + {1} eredménye: {2}".format(*args: x, y, eredmény))
8
9  for i in range(1,3):
10     print("i érték (for-on belül): ",i)
11 print("i érték (for-on kívül): ",i) # globális hatókör - i
12
13 lokalis_valtozo() # függvény hívása
14 print(x, y, eredmény) #hibát jelez, függvényen kívül nem érhetőek el a belső változók
15

```


- A Python kód fő részében, a fő modulban létrehozott változók globális változók, és a globális hatókörhöz tartoznak.
 - a **globális változók bárholnan elérhetők**, globális és helyi szinten.
- Függvényen belül **global kulcsszóval** változtathatjuk a belső változó hatókörét globálissá
- Ha **ugyanazzal a változó névvel** hozunk létre egy függvényen belül és kívül változót, a Python **két különálló változóként** kezeli őket,
 - *az egyik a globális hatókörben érhető el (a függvényen kívül),*
 - *a másik a helyi hatókörben érhető el (a függvényen belül)*

Feladat:

- 8. sorba írjuk ki x,y értékét (print()) – függvényen belüli kiírás. Figyeljük meg mi lett a kiírás eredménye.
- 11. sorban írjuk ki x,y értékét. Figyeljük meg mi lett a kiírás eredménye.
- 6. sorba illesszünk be egy új parancsot, tegyük y láthatóságát globálissá a global y parancssor megadásával
- Ismét figyeljük meg és értelmezzük a kiírás eredményét.

- A modulok Python fájlok, amelyek függvényeket, osztályokat és változókat tartalmaznak.
- Egy modul importálásához a következő szintaxist használjuk:

import modul_neve

- Egy adott modulból csak bizonyos elemeket is importálhatunk:

from modul_neve **import** elem1, elem2

- **Felhasználási terület**

- Kód szervezése és újrafelhasználása
- Standard függvények és osztályok biztosítása
(pl. matematikai függvények, fájlkezelés)

Előnyök

- Kód modularizálása:
 - Nagyobb programokat kisebb, önálló modulokra bonthatunk.
- Kód újrafelhasználása:
 - A modulokat más programokban is importálhatjuk.
- Névtér kezelése:
 - A modulok külön névteret biztosítanak, így elkerülhetjük a névütközéseket.

Hátrányok

- Túl sok modul használata bonyolulttá teheti a program szerkezetét.

BEÉPÍTETT MODULOKRA PÉLDÁK

Matematikai műveletek

- **math**: Alapvető matematikai függvények (pl. sin, cos, sqrt, pi).
- **numpy**: Numerikus számításokhoz, tömbökkel való műveletekhez, mátrixfeldolgozás. (+ Deep Learning – AI – neurális hálók)
- **scipy** (numpy alapú): Tudományos és műszaki számításokhoz, optimalizáláshoz, statisztikához. (+ képfeldolgozás, deep learning)

Szöveges adatok kezelése

- **string**: Szöveges adatok manipulálására.
- **nltk**: Természetes nyelv feldolgozásához (angol).

Fájlkezelés

- **os**: Operációs rendszer függvényekhez, fájl- és könyvtár műveletekhez.
- **shutil**: Magas szintű fájl műveletekhez (másolás, áthelyezés).
- **pathlib**: Objektum-orientált fájlútvonalkezelés.

BEÉPÍTETT MODULOKRA PÉLDÁK

Adatstruktúrák

- **collections**: Speciális adatszerkezetek (pl. deque, defaultdict).
- **heapq**: Heap adatszerkezethez.
- **bisect**: Rendezett listákba való beszúráshoz.

Adatbázisok

- **sqlite3**: Beágyazott SQL adatbázis.
- **MySQLdb**: MySQL adatbázishoz való csatlakozás.

Hálózati programozás

- **socket**: Alapszintű hálózati programozáshoz.
- **requests**: HTTP kérések küldéséhez.
- **urllib**: URL-ek kezeléséhez.

Webfejlesztés

- **flask**: Web alkalmazások fejlesztéséhez.
- **Django**: Nagyobb webalkalmazásokhoz.

BEÉPÍTETT MODULOKRA PÉLDÁK

- Grafikus felhasználói felület
 - Tkinter: Python alapú GUI könyvtár. (python része, kisebb projektekhez)
 - PyQt: Qt keretrendszer Pythonhoz (többplatformos keretrendszer).
 - wxPython: wxWidgets keretrendszer Pythonhoz. (többplatformos, számos widgetet tartalmaz)
 - PySimpleGUI: kezdőknek tervezve, Qt-Tkinter-wxPython-Remi alapú
 - <https://docs.pysimplegui.com/en/latest/>
- Egyéb hasznos modulok
 - **datetime**: Dátumok és időpontok kezeléséhez.
 - **random**: Véletlenszámok generálásához.
 - **json**: JSON formátumú adatok kezeléséhez.

Feladat:

- 1. sorban nevezzük át a statistics modult stat-ra (as kulcsszó segítségével) – 10. és 11. sorban megoldódik a szintaktikai hiba, elérhetővé válnak a modul metódusai
- 5. sort módosítsuk – használjuk a math modul pi értékét a számításban
- 7. sorba írjuk ki és számítsuk ki a kör területét – math.pow() és math.pi elemeket használva
- 14. sorban hozzunk létre egy min. 5 elemű, tetszőleges típusú elemekkel feltöltött listát[] lista néven, majd nézzük meg a kód futási eredményeit, értelmezzük a használt modulok metódusait.

SAJÁT MODULOK ÉS HASZNÁLATUK

- Létrehozhatunk saját modulokat .py kiterjesztéssel, amelyek függvényeket, osztályokat és változókat tartalmazhatnak.
- Importálása

import modul_neve **as** nev

- Egy modulból csak bizonyos elemek importálása

from modul_neve **import** elem1, elem2

Feladat:

- Egészítsd ki a saját modult két újabb függvénnyel:
 - 7. – 8. sorba készítsd el a szorzas függvényt, mely a és b értékét összeszorozza egymással, majd visszatér a szorzat eredményével
 - 10. – 11. sorba készítsd el az osztas függvényt, mely a értékét elosztja b értékével, visszatér a két szám hányadosának eredményével

Feladat:

- 5. sorban hívd meg a saját modul osztas metódusát 3, 5 értékekkel, majd írasd ki a képernyőre az eredményt
 - *sajatmodul.osztas() hívás alkalmazása*
- 11. sorban hívd meg a saját modul szorzas metódusát 3, 5 értékekkel, majd jelenítsük meg az eredményt a képernyőn
 - *szorzas() hívás alkalmazás, modul név nélkül*



Fájlkezelés és kivételkezelés

fájlok olvasása és írása

kivételek kezelése

- Az open() függvény fontosabb paraméterei:
 - fájlnev, mód és kódolás.
- Négy különböző alapszűr (mód) van a fájl megnyitására:
 - r – read - olvasásra *(létező fájl esetén)*
 - a – append - írás *(megnyitás, hozzáfűzés)*
 - w – write - írás *(létre is hozza a fájlt, ha nem létezik/megnyitás, hozzáfűzés)*
 - x – create - létrehozás *(hibát jelez ha létezett a fájl)*
- Elérési út megadás: relatív és abszolút hivatkozás
 - pl.: open('adatok.txt') vagy open('c:/doksik/adatok.txt') vagy with open()

Minden esetben ellenőrizze, hogy létezik-e a fájl, különben hibát fog kapni ha nem megfelelő módon történik a fájlkezelés!

FÁJL OLVASÁS

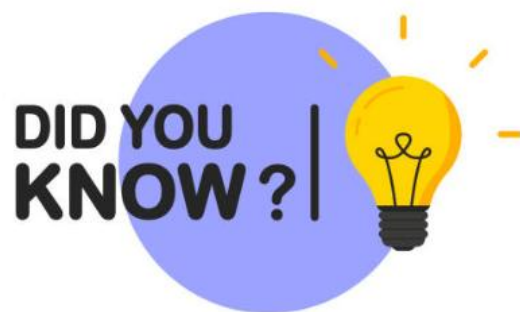
filekezeles.py × adatok.txt ×

```
1 A kisautó zöld.
2 Az alma piros.
3 A vonat kék.
4 A virág fehér.
```

filekezeles.py × adatok.txt ×

```
1 # fájlkezelés - fájl olvasása, open() függvény
2 # adatok.txt a filekezeles.py állománnyal megegyező mappában helyezkedik el
3
4 # r-read, ékezetes karakterek miatt utf-8 kódolás
5 file = open('adatok.txt', 'r', encoding="utf-8") # open() beépített függvény
6
7 print(file) # megadja az open függvény által tartalmazott objektumot
8
9 betu = file.read(3) # megadott számú karaktert olvas be
10 print("read művelet: ", betu)
11 sor = file.readline() # A soron következő sort olvassa be, \n karakterrel együtt
12 print("readline művelet + '\ n': ", sor)
13 sor2 = file.readline()
14 print("readline művelet + '\ n' nélkül: ", sor2.strip()) # strip() felesleges helyközt törli
15 sorok = file.readlines() # beolvassa a soron következő sorokat egy tömbbe
16 print("readlines művelet: ", sorok)
17
18 file.close() # minden fájlnyitást zárásnak kell követni
19
```

BEOLVASÁS EREDMÉNYE



```
filekezeles.py x adatok.txt x
1 A kisautó zöld.
2 Az alma piros.
3 A vonat kék.
4 A virág fehér.
```

```
filekezeles x
C:\Progs\Python\venv\Scripts\python.exe C:/Progs/Python/tananyag/python_03/filekezeles.py
<_io.TextIOWrapper name='adatok.txt' mode='r' encoding='utf-8'> print(file)
read művelet: A k file.read(3)
readline művelet + '\ n': isautó zöld. file.readline()

readline művelet + '\ n' nélkül: Az alma piros. file.readline() → sor2.strip()
readlines művelet: ['A vonat kék.\n', 'A virág fehér.'] file.readlines()

Process finished with exit code 0
```

FÁJL MEGNYITÁSA MÁSKÉPP...

- A **open()** függvény segítségével megnyithatjuk a fájlokat olvasásra vagy írásra,
- majd a **with utasítással** biztosíthatjuk, hogy a fájl automatikusan bezáruljon, ha már nincs rá szükség.
 - nem kell manuálisan hívni a **close()** metódust
- megadható itt is harmadik paraméterként a fájl kódolása az **encoding** kulcsszóval.

Feladat:

- Hozzunk létre egy adatok.txt fájlt abba a mappába ahol a .py állományaim mentésre kerültek.
 - Töltsük fel a .txt fájlt min. 4 sorba, 4 rövidebb mondattal.
- Python_02_08.py program módosítása:
 - 8. – 11. sorban valósítsuk a fájl tartalmának beolvasását és képernyőn való megjelenítését WHILE ciklus segítségével
 - Használjuk segítségképpen a programban lévő kommenteket
- Elemezzük a program futtatásának eredményét.

FÁJLBA ÍRÁS (W-ÍRÁS)

ha nem létezett létrehozza

filekezeles3.py × jatekok.txt ×

```

1 # fájlkezelés - fájlba írás (w-write, a-append)
2 with open('jatekok.txt','w',encoding='utf-8') as file_out:
3     jatek = 'baba' # ha szükséges sortörés --> a végére \n
4     file_out.write(jatek) # egy sort ír a fájlba
5
6     jatek2 = ['könyv\n','autó\n','kocka\n']
7     file_out.writelines(jatek2) # több sort ír a fájlba, vagy több adatot \n nélkül
8
9     jatek3 = ['színes könyv ','színes autó ','színes kocka ']
10    file_out.writelines(jatek3)
11
12 with open('jatekok.txt','r',encoding='utf-8') as file_in:
13     for sor in file_in:
14         print(sor.strip())
15

```

babakönyv

autó

kocka

színes könyv színes autó színes kocka

HA ÍRNI ÉS OLVASNI IS SZERETNÉNK EGYSZERRE

```
# fájlba írás, hozzáfűzés --> az utolsó sor után illeszti be \n-el a megadott stringet
with open('jatekok.txt','a',encoding='utf-8') as file_mod_ir, open('jatekok.txt','r',encoding='utf-8') as file_mod_olvas:
    uj_jatek = '\nkisvasút'
    file_mod_ir.write(uj_jatek)
    for sor in file_mod_olvas:
        print(sor.strip())
```

```
C:\Progs\Python\venv\Scripts\python.exe
babakönyv
autó
kocka
színes könyv színes autó színes kocka
kisvasút

Process finished with exit code 0
```

Egy blokkon belül történik a fájlba írás/módosítás és olvasás. → de! csak két külön logikai fájlal működik

Szintaxis

try:

megpróbálja az utasítást végrehajtani
teszteli a végrehajtást

except:


kezeli a hibát, bármennyi lehet belőle
végrehajtja, ha a try blokkban lévő utasítást
nem tudja végrehajtani

else:

ha try blokk lefut - else is, ha try nem fut le - else sem

finally:

minden esetben végrehajtódik, függetlenül a teszt eredményétől



az esetek
90%-ban
elegendő a try és
except blokk
használata..

KIVÉTELKEZELÉS - FÁJLKEZELÉS

kivételkez.py x

```

1  # kivételkezelés - példafeladat
2  # fájlkezelés
3  try:
4      with open('proba.txt','r') as file:
5          for sor in file:
6              print(sor.strip())
7  except FileNotFoundError as e:
8      print('A fájl nem található! -->', e)
9  else:
10     print('else ág kiírása')
11 finally:
12     print('Kilépés...')
13

```

```

C:\Progs\Python\venv\Scripts\python.exe C:/Progs/Python/tananyag/python_03/kivete
A fájl nem található! --> [Errno 2] No such file or directory: 'proba.txt'
Kilépés...

```

Process finished with exit code 0

Feladat:

- 5. sorba írjuk ki az aktuális elem (i) és 10 szorzatának eredményét. (i*10 a művelet, print() kiírás)
 - *Nézzük meg a kiírás eredményét, mit veszünk észre?*
- 5. sorban módosítsuk az i értékét int(i)-re
 - *Nézzük meg a kiírás eredménye miben változott, miért?*
- 8. sorba írjuk a lista 6. indexén lévő elemet a képernyőre, majd ellenőrizzük a program futásának eredményét. *Értelmezzük is!*

GYAKORLÓFELADAT – PYTHON_02_10.PY

Feladat *(kivételkezelés, beolvasás, adattípus konverzió, lista, feltételes utasítás, ciklus):*

- Írj egy Python programot, amely bekér több számot a felhasználótól, majd kiszámítja és kiírja ezek átlagát.
 - Beolvasás q karakter leütéséig történjen – végjeles beolvasás
- A programnak képesnek kell lennie arra, hogy kezelje azokat az eseteket, amikor a felhasználó nem számot ad meg, hanem más karaktert.
 - A program ezekre az esetekre kivételkezeléssel adjon hibaüzenetet („Érvénytelen bemenet, kérem számot adjon meg!”) – majd folytassa a beolvasást q leütéséig.

Feladat:

- Írj egy Python programot, amely egy véletlenszerű számot választ 1 és 100 között.
- A felhasználónak addig kell találgatnia, amíg el nem találja a számot.
- A program minden tipp után jelezze, hogy a tipp nagyobb vagy kisebb-e a keresett számnál.
- Használd a random modult.
- Kivételkezeléssel oldjuk meg az érvénytelen bemenetet.
- A program addig fut, amíg a felhasználó el nem találja a számot.



ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY



KÖSZÖNÖM

A MEGTISZTELŐ FIGYELMET!

Módné Takács Judit



modne.t.judit@amk.uni-obuda.hu