

Python Kurzus – Első előadás

Tárgyfelelős: Dr. Tejfel Máté

Előadó: Dr. Király Roland

ELTE IK



Tematika:

1. **Python bevezetés, telepítés, bevezetés a Pythonba és az AI etikus használata**
2. **Adattípusok és változók,**
3. **függvények és hibakezelés**
4. **Objektumorientált programozás**
5. **Objektumorientált programozás gyakorlati példákkal**
6. **Csomagkezelés, PIP, PIP3, grafikus felületek használata**
7. **Webfejlesztés Pythonban, Flask**
8. **REST API és api-kezelés -FastAPI**
9. **OpenCV, képek, objektumok azonosítása, computer vision**
10. **Mesterséges intelligencia és gépi tanulás alapjai, neurális hálók**
11. **Neurális hálók alkalmazása, TensorFlow**
12. **Adatkezelés, Big Data alapok**
13. **Adatvizualizáció, adatok térképek BigData alapokon**

A prezentációhoz tartozó források

A félévi munka értékelése:

- A gyakorlaton való aktív részvétel (WarmUp-ok) (**20%**)
- Legfeljebb 3 hiányzás lehetséges!
- Canvaskurzus: minden gyakorlaton egy WarmUpfeladatsor megírása
- A WarmUpkérdések az előadáson elhangzottak alapján.
- A ~7. gyakorlat elején midterm' teszt (**20%**)
- A ~13. gyakorlat elején final' teszt (**20%**)
- A félév során 2 beadandó program (**20-20%**)
- A beadandókat meg kell védeni!

Előadás:

1. A Python rövid története, előnyei és főbb jellemzői
2. Python telepítése, beállítások
3. Fejlesztői környezetek
4. Az első Python programok
5. Az AI-alapú kódgenerálás előnyei és korlátai
 - COCALC
 - GitHub Copilot
 - LLM-ek
6. Az AI etikus használata (ingyenes teljes verzió):
 - Plágium elkerülése és a saját kód értékének megőrzése
 - Kódminőség és felelősség a generált kód esetében
 - A Python programok írása a cél. (Nagyobb Python programok megírása és megértése.)

1. A python rövid története és előnyei



- Python rövid története, **Guido van Rossum**, '90-es évek
- Python népszerűsége és széles körű felhasználási területei
- Python előnyei:
 - Könnyű tanulhatóság
 - Egyszerű szintaxis
 - Hatékony és gazdagkönyvtárkészlet, komoly közösség
 - Több célra: Kutatáshoz kiváló, asztali, webes, beágyazottrendszerre írt alkalmazásokhoz jól használható
 - Webfejlesztés, API-k.

A nyelv főbb jellemzői

- Magas szintű programozási nyelv
- Imperatív, funkcionális és objektumorientált lehetőségek (nem tiszta OOP nyelv)
- Interpretert alkalmaz, de készíthető exe*
- Case Sensitive(a kulcsszavak kisbetűsek)
- A blokkokat indentálni kell (azonos darabszámú szóközökkel)
- Az utasításlezáró jel a sor vége, vezérlő szerkezeteknél a :
- A fejlesztői környezet több operációs rendszerre elérhető: Windows, Linux, macOS, Webes környezet, terminál

A python programozási nyelv felhasználási területei

1. **Adattudomány és gépi tanulás** (NumPy, Pandas, Matplotlib, TensorFlow) **tensorflowOK.ipynb**
2. **Webfejlesztés** (Web, API-kfejlesztése, e-kereskedelmi rendszerek, fastApi, Flask, Django)
3. **Automatizálás és szkriptelés**(folyamat automatizálás, fájlkezelés, vezérlés)
4. **Pénzügyi szektor**(pénzügyi modellek, kockázati előrejelzések, nagy adatbázisok elemzése, BigData)
5. **Tudományos kutatás**(SciPy(matek), NumPy, Matplotlib, egyebek) **maps.jpeg**
6. **Mesterséges intelligencia, computer vision és robotika** (TensorFlow, PyTorch)
objectdetection.py, objectdetectionpict.py
7. **Játékfejlesztés** (Játéklogika fejlesztése, prototípus készítése, 2D játékok fejlesztése, Pygame, Arcade, adventurelib, **Pyglet**) **test.py**
8. **Blockchaintechnológia** (gyors prototípuskészítés, könnyű adatkezelés)
9. **Data science**, adatbányászat **SanFranciscoCrime.ipynb**
10. **DevOps**(DevelopmentOperations, CI/CD pipeline-ok fejlesztése, infrastruktúra-automatizálás, Kubernetes)
11. **LLM-ek**tanítása (Ollama, Llama) FT, Webes API LLM használatához.

Python telepítése, beállítások

- Python telepítése különböző operációs rendszereken:
<https://www.python.org/downloads/>
 - Windows,
 - macOS,
 - Linux
- Pip (pip3) csomagkezelő használata
 - **pip install pygame, --version, update, install, egyebek**

3. Az első python programok

Classic

```
i = 0
for item in items:
    print(i, item)
    i += 1
```

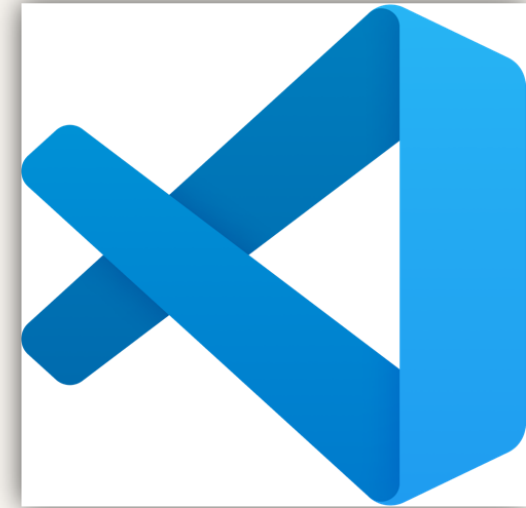
Modern (enumerate)

```
for i, item in enumerate(items):
    print(i, item)
```

- Az IDLE környezet
- Env! uv/env
- Az első program írása és futtatása:
 - A program futtatása és magyarázata
- A Python kód szerkezete: ([calendar.py](#), [textarts.py](#))
 - Kód: **x = 10**
y = 20
print(x + y)

4. Fejlesztési környezetek

- **Visual Studio Code** (VSCode):
 - Integrált fejlesztési környezet Pythonhoz
 - Hibakeresés, kódkiegészítés, integrált terminál
- **Jupyter Notebook** (ipynb):
 - Interaktív fejlesztési környezet
 - Adatfeldolgozás és vizualizáció támogatása böngésző alapú felületen



5. Az AI-alapú kódgenerálás előnyei és korlátai

- **Előnyök:**

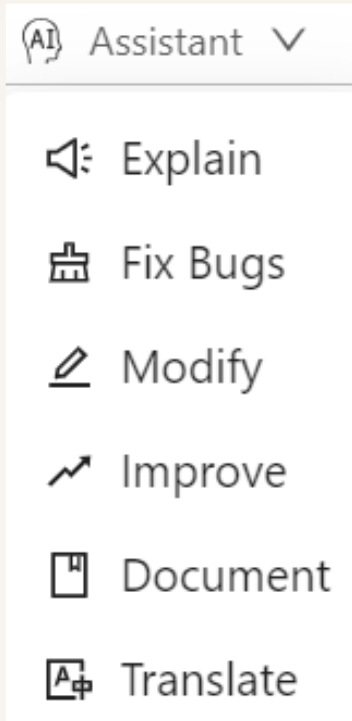
- Fejlesztési folyamat gyorsítása
- Kisebb problémák megoldásának segítése

- **Korlátok:**

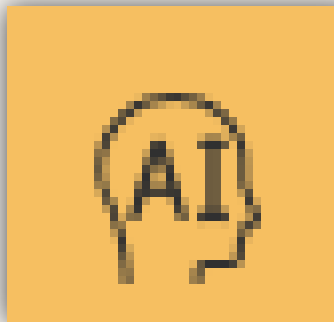
- Nem mindig pontos vagy optimális kód
- Ellenőrzés szükségessége



AI és COCALC (Collaborative Calculus)



- AI-alapú programozás támogatás (<https://cocalc.com>):
 - Természetes nyelvű inputból kód javaslatok
 - Kódgenerálás, magyarázat, hibajavítás,
 - Dokumentáció, fejlesztés, ...
 - Magyar nyelvű kommunikáció



GitHub Copilot

- Mi a GitHub Copilot?
 - AI-alapú kódgenerálás Codexmodell alapján
 - Természetes nyelvű inputból kód javaslatok
- Hogyan működik a Copilot?
- https://education.github.com/discount_requests/application
 - Integráció fejlesztési környezetekkel (pl. VSCode)
 - Kódgenerálás a fejlesztés során
- <https://docs.github.com/en/authentication/securing-your-account-with-two-factor-authentication-2fa/configuring-two-factor-authentication>



Kódgenerálás, példa

Írjunk egy Python kódot, amely egy lista elemeit összeadja és kiírja az eredményt:

```
# Define a list of numbers  
lista = [1, 3, 5, 7, 9, 11]  
# Calculate the sum of the numbers in  
the list  
osszeg = sum(lista)  
# Print the sum  
print("Az összeg:", osszeg)  
Esetleg valami komolyabbat!
```



Generate code cell using

A kód finomítása, a hatékonyság növelése, példa

Írjunk egy Python programot, amely megadja a Fibonacci-sorozatot rekurzív módszerrel

```
def fibonacci(n):  
    if n <= 1:  
        return n  
    else:  
        return fibonacci(n-1) +  
        fibonacci(n-2)  
# N-edik Fibonacci szám kiszámítása  
n = 10  
print(f"A(z) {n}. Fibonacci szám:  
{fibonacci(n)}")
```

A kód finomítása, a hatékonyság növelése, példa

Írjunk egy Python programot, amely megadja a Fibonacci-sorozatot iterációval

```
def fibonacci(n):  
    if n <= 1:  
        return n  
    a, b = 0, 1  
    for i in range(2, n+1):  
        a, b = b, a + b  
    return b  
  
# N-edik Fibonacci szám kiszámítása  
n = 10  
print(f"A(z) {n}. Fibonacci szám: {fibonacci(n)}")
```

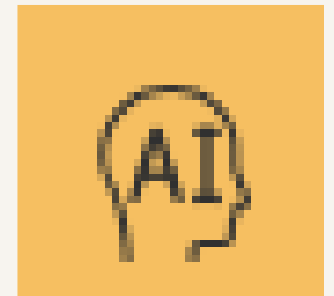
6. Az AI etikus használata

- **Plágium elkerülése:**

- Nyílt forráskódú adatokból tanuló AI használata
- A saját kód egyediségének

- **Kódminőség és felelősség:**

- A generált kód felülvizsgálatának fontossága
- A fejlesztők felelőssége a kód minőségéért és biztonságáért



Összegzés

- Bemutattuk a kurzus tematikáját, elvárásait és az értékelés szempontjait
- Ismertettük:
 - A Python nyelv és programozás bevezető gondolatait
 - Az AI használatát és etikus alkalmazásának elveit

Köszönöm a figyelmet!

ELTE IK

Python kurzus – 2. előadás

Tárgyfelelős: Dr. Tejfel Máté

Előadó: Dr. Király Roland

2. Előadás tematikája

Adattípusok és változók

1. Interpreter – compiler, byte code
2. Változók létrehozása és használata
3. Alapvető adattípusok
 - Elemi adattípusok
 - Összetett adattípusok
4. Műveletek és operátorok
5. Alapvető ki- és bemeneti műveletek

A prezentációhoz tartozó forráskódok

[bytecode.py](#) (pdf állomány megnyitása)

[called.py](#) (pdf állomány megnyitása)

[lesson1.py](#) (pdf állomány megnyitása)

[problem1.py](#) (pdf állomány megnyitása)

[deffun.py](#) (pdf állomány megnyitása)

[példák.py](#) (pdf állomány megnyitása)

1. Interpreter – compiler

A Python értelmező úgy futtatja a programot, hogy egyszerre csak egy utasítást hajt végre (szekvenciális végrehajtás).

Értelmezés és fordítás



Byte kód

- A lefordított .pyc fájlok. A `__pycache__`. Könyvtárba kerülnek. Ez a byte-kódos rendszer megakadályozza, hogy minden futtatáskor újra kellene fordítani a fájlokat.

- Miért fontos ez:

A lexikális, szintaktikai és szemantikai elemzés, az optimalizálás és a kód generálás időigényes.

- Mi a byte kód?

- A Pythont gyakran interpretált nyelvként írják le – amelyen a forráskódot a program futása közben natív CPU-utasításokká fordítják le.
- Ez csak részben igaz. A forráskódot egy virtuális gép utasításkészletére fordítja.
- A **Python interpreter** ez a virtuális gép.
- A formátum, amit használ a "bytecode".

CPython

- Nem forráskód, hanem a Python virtuális gép utasításai.
- A CPython értelmezőt használja.
- **Verem alapú virtuális gép (LIFO)**
- GitHub link: <https://github.com/python/cpython>
- (printbytecod.py)

Byte code:

```
def hello()
    print("Hello, World!") (code□ )

0 LOAD_GLOBAL                0 (print)
2 LOAD_CONST                  1 ('Hello, World!')
4 CALL_FUNCTION
```

CPython verem típusok

(bytecode.py, printbytecode.py)

- Három különböző verem típust használ
 - Hívási verem (call stack)
 - A függvényhívások, vagy a main program (frame) végrehajtását vezérli, az alja a belépési pont.
 - Kértékelő, vagy adat verem (evaluation, vagy data stack)
 - Minden híváshoz bekerül egy elem ebbe a verembe, és itt hajtódnak végre az utasítások.
 - Blokk verem (block stack)
 - A vezérlő struktúrák végrehajtása, pl.: with, try, break, continue, stb
- A byte kód végrehajtása többnyire a kiértékelő veremben zajlik.

Példa byte code

def fun1(variable1, 42, variable2)

LOAD_NAME 1 utasításra fordul, ami a függvény objektumot a kiértékelő verem tetejére rakja

LOAD_NAME 2 ez a paraméterét rakja a kiértékelési verem tetejére

LOAD_CONST 3 a kiértékelést követően a függvény konstans paraméterét (42) rakja a kiértékelési verem tetejére.

LOAD_NAME 4 ez a függvény második paraméterét rakja a kiértékelési verem tetejére

CALL_FUNCTION_W - kulcsszavas paramétereknél

CALL_FUNCTION_EX – kicsomagolandó paramétereknél, stb.

dis module - Python standard library,
disassembler -> human-readable version

```
import dis  
dis.dis(func)
```

Amennyiben van egy func() függvényed

Byte code megtekintése

```
import dis  
  
def func(param1):  
    return func(param1)  
  
dis.dis(func)
```

024 2/Python EA 2025/EA1/testByteCode.py"

```
3 0 Resume 0  
4 2 LOAD_GLOBAL 1(NULL + func)  
12 LOAD_FAST 0(param1)  
14 CALL 1  
22 RETURN_VALUE
```

<https://docs.python.org/3/library/dis.html>

Miért ismerjük meg a byte code-ot?

- Bizonyos konstrukciók miért gyorsabbak, mint mások?
- Miért gyorsabb a {}, mint dict()?

<https://opensource.com/article/18/4/introduction-python-bytecode>

```
dis.dis("{}")  
print("--- vs -----")  
dis.dis("dict()")
```

```
0      0 RESUME      0  
  
1      2 BUILD_MAP  
      4 RETURN_VALUE  
  
--- vs -----  
0      0 RESUME      0  
  
1      2 PUSH_NULL  
      4 LOAD_NAME      0 (dict)  
      6 CALL          0  
     14 RETURN_VALUE
```

Előnyök - hátrányok

Compiler/Interpreter

Compiler

- A programozónak rendelkeznie kell a fordítóval, de a felhasználó a fordítóprogram nélkül is tudja futtatni a kódot
- A lefordított kód végrehajtása általában gyorsabb
- Maga a fordítás időigényes folyamat Hibakeresés, javítás szükséges
- Különböző platformokon különböző fordítóprogram kell
- A fordítóprogram hibaüzeneteivel megakadályozza a későbbi futási hibákat
- Windows: <https://www.youtube.com/watch?v=bqNvkAfTvIc>

Interpreter

- A felhasználónak rendelkeznie kell az interpreterrel a kód futtatásához
- A kész kódot azonnal értelmezi és futtatja, nincsenek további fordítási fázisok
- A kódot a programozási nyelven tároljuk, nem gépi nyelven. Különböző gépi nyelveket használó számítógépeken futtatható, nem kell külön lefordítani a kódot minden egyes architektúrára
- Az interpretáció során a számítógép megosztja a teljesítményét az értelmezővel, ezért lassabb, mint a lefordított program
- MACOS: <https://www.youtube.com/watch?v=JZzu1mLXkog>

Csomag fordítás MacOS/ Linux

```
from setuptools import setup  
import distutils.core
```

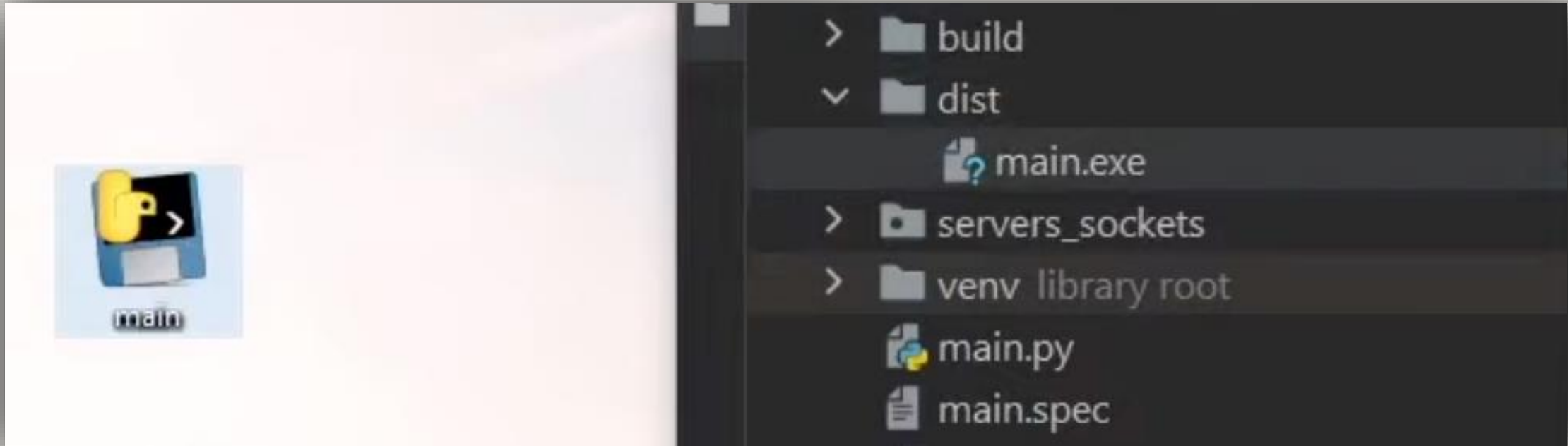
```
APP = ["test.py"]
```

```
OPTIONS = {  
    'argv_emulation': True  
}
```

```
setup(  
    app=APP,  
    options={'py2app': OPTIONS},  
    setup_requires=['py2app']  
)
```

EXE készítése Windows

```
pyinstaller main.py --onefile
```



A Python objektummodellje:

- Minden szám, karakterlánc, adatstruktúra, függvény, osztály, modul egy Python objektum
- Minden objektumhoz tartozik egy **típus** (pl. egész szám, karakterlánc vagy függvény) és **belső adat**
- Minden objektumnak megvannak a saját **műveletei** és **metódusai**

```
a = 1234
print(id(a))
#4310218384

a = 10
b = 10
c = 11
d = 12

print(id(a))
print(id(b))
print(id(c))
print(id(d))
```

Példa:

```
páros = [0,2,4,6,8]
páros.append(10)
print(páros)
```

Output:

```
páros = [0,2,4,6,8,10]
```

2. Változók létrehozása és használata

- **Adatok**, műveletek adatokon – **változók**
- Változó: egy memóriacímet jelölő **hivatkozás**
 - Ezen a címen egy **érték** bináris számok sorozata formájában van tárolva,
 - **változónév**
 - **érték típus**

```
1 | >>> a = 13234
2 | >>> id(a)
3 | 140712608160400
```



Variable name	Object id
a	140712608160400

A változók jellemzői Pythonban

- Nem definiálunk típust a változónak (megtehetjük)
- A változóban tárolt értéknek van típusa:
 - Nincs változódeklarálás
 - A változóban tárolt érték típusa a program futása során megváltozhat:

```
a = 10  
a = a = "alma"  
print(a)
```

- A változóban tárolt érték méretét csak a memória mérete korlátozza
- Konstans nem definiálható

Type check

*("We're all consenting adults
here")*

```
def pick(l: list, index: int = 0) -> int:  
    return l[index]
```

Ez nem valódi ellenőrzés!

Így lehet ellenőrizni:

```
def fun(l: list, index: int = 0) -> int:  
    if not isinstance(l, list):  
        raise TypeError  
    return l[index]
```

Változónevek

- „Beszédes” változónevek
- A névválasztás szabályai:
 - betűvel vagy aláhúzásjellel kell kezdődnie
 - nem kezdődhet számmal
 - csak alfanumerikus karaktereket és aláhúzásjeleket tartalmazhat
- A kis- és nagybetűk megkülönböztetése
- A változónév nem lehet kulcsszó:

and	assert	break	class	continue	def
del	elif	else	except	exec	finally
for	from	global	if	import	in
is	lambda	not	or	pass	print
raise	return	try	with	while	yield
False	None	True			

Értékadás

Egyszerű értékadás:

változó = érték

alma = 26

Többszörös (szimultán) értékadás:

változó₁, ..., változó_n = érték₁, ..., érték_n

alma, körte, dió = 26, 15, 4

alma = körte = dió = 15

3. Alapvető adattípusok (test.py)

- **Elemi adattípusok:**

- *Egész szám (integer)* 12 `int()`
- *Valós szám (float)* 12.34 `float()`
- *Komplex szám (complex)* 1.2 + 3.4j `complex()`
- *Logikai (boolean)* True v. False `bool()`

- **Összetett adattípusok:**

- *Szöveg (string)* "Almafa" v. 'Almafa' `str()`
- *Tuple* (10, 20, 30, 40) `tuple()`
- *Lista (list)* [1, 2, 2, 3, 3, 3, 4, 4, 4, 4] `list()`
- *Szótár (dictionary)* {a: 1, b: 2, c: 3, d: 4} `dict()`
- *Halmaz (set)* {1, 2, 3, 4} `set()`

Adattípusok csoportosítása

- **Kategóriák szerint:**
 - Numerikus típusok: int, float, complex
 - Sorozattípusok: list, tuple
 - *Halmaz* típusok: set, frozenset
 - Bináris típusok: bytes, bytearray, memoryview
- **NoneType** **a = None**
type(a) ➤ `<class 'NoneType'>`
- **Immutable** (nem megváltoztatható): szöveg (str), tuple
- **Mutable** (megváltoztatható): a többi típus

4. Műveletek és operátorok

Aritmetikai műveletek:

Symbol	Task Performed	Example	Result
+	Addition	4 + 3	7
-	Subtraction	4 - 3	1
/	Division	7 / 2	3.5
%	Mod	7 % 2	1
*	Multiplication	4 * 3	12
//	Floor division	7 // 2	3
**	Power of	7 ** 2	49



Fontos a nullával osztás elkerülése!

Összehasonlító operátorok: ==, !=, <, >, <=, >=

Stringműveletek

Összefűzés:

```
x = "Alma" + "fa" □ "Almafa"
```

Sokszorozás:

```
y = "Rin" + 2*" Tin" □ "Rin Tin Tin"
```

String hossza:

```
hossz = len(y) □ 11
```

```
i. karaktere: kar = x[2] □ "m"
```

```
i.-től a j-1.-ig tartó szelete: t = x[2:5] □ "maf"
```

Metódusok:

https://www.w3schools.com/python/python_ref_string.asp

Logikai műveletek: and, or, not

Listaműveletek

Összefűzés: $[1, 2] + [2, 1]$ ☐ $[1, 2, 2, 1]$

Sokszorozás: $3 * [2, 0]$ ☐ $[2, 0, 2, 0, 2, 0]$

Példa: jegyek = [1,2,3,4,5]

Tartalmazás-vizsgálat: **if 3 in jegyek: ...**

A lista hossza: **n = len(jegyek)**

A lista i. eleme: **elem = jegyek[i]; jegyek[i]=elem**

Lista törlése: **del jegyek**

A lista i.-től j-1.-ig tartó szelete (slices):

új=jegyek[i:j]

List comprehension (lista kifejezés):

számok = [i*2 for i in range(10)]

Példák listaműveletekre

Másolat (copies): **a = [1, 2, 3, 4, 5] b = a**

Szelet (slices): **f = a[:]**

Rendezés: **a = sorted(a, reverse = True)**

Komprehenzív lista (lista kifejezés):

j = [i2 for i in range(10) if i%3 == 0]**

k = [i2 if i%3 == 0 else i for i in range(10)]**

Zip: **m = [1,2,3,4,5]**
 n = ['a','b','c','d','e']
 o = list(zip(m, n))

[(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd'), (5, 'e')]

Lista metódusok

Példa: `lista = [1, 3, 5, 7]`

append()	A lista végéhez egy elemet hozzáfűz	lista.append(9)
clear()	Minden elemet eltávolít a listából	lista.clear()
copy()	A lista egy másolatát adja vissza	ptl = lista.copy()
count()	Visszaadja a feltételnek megfelelő listaelemek darabszámát	
extend()	Egy lista elemeit az aktuális lista végéhez hozzáfűzi	
index()	Visszaadja a feltételnek megfelelő első listaelem indexét	
insert()	Egy elemet beilleszt a megadott pozícióra	
pop()	Egy elemet eltávolít a megadott pozícióról	
remove()	Eltávolítja az elsőként előforduló megadott elemet	
reverse()	Megfordítja a lista elemeinek sorrendjét	
sort()	Rendezzi a listát	

Alapvető Python Adatstruktúrák

Tuple: immutable

Példa: `Gyümölcs = ("alma", "banán", "szőlő", "banán")`

Dictionary - Kulcs: érték pár

Példa:

```
car = {"brand": "Opel",  
      "modell": "Astra",  
      "year": 2003,  
      "mass": 1250}
```

```
print(car)  
  
{'brand': 'Opel', 'modell': 'Astra', 'year': 2003, 'mass': 1250}
```

Set: nem rendezett, nem megváltoztathatóak az elemei, és nem indexelt

Példák: `a = [1,1,1,2,2], b = {2,4,6}`

`a = set(a), b.add(8) □ {2,4,6,8}`

`print(a) □ {1,2}`

5. Alapvető ki- és bemeneti műveletek

```
print("Hi!", "Here I am.", sep = "***", end = "###")
```



```
Hi!***Here I am.###
```

vagy

```
print("\nHello", "World", sep = "***", end = "###")
```



```
Hello***World###
```

print(object(s), sep=separator, end=end, file=file, flush=flush)

print()

- `object(s)`: Azok az objektumok, amelyeket ki szeretnénk írni. Több objektumot is megadhatunk, vesszővel elválasztva.
- `sep`: Az elválasztó karakterek, amelyek az objektumok közé kerülnek. Alapértelmezett értéke egy szóköz (' ').
- `end`: Az a karakter vagy karakterlánc, amely a sor végére kerül. Alapértelmezett értéke egy új sor ('\n').
- `file`: Az a fájl vagy fájlszerű objektum, ahová a kimenet kerül. Alapértelmezett értéke a standard kimenet (`sys.stdout`).
- `flush`: Ha `True`, akkor a kimeneti puffer azonnal kiürül. Alapértelmezett értéke `False`.
Puffer! Mi a kimeneti puffer?

Input

Input(prompt)

```
a = input ("Kérek egy egész számot: ")
```

```
b = int(a)
```

```
b = int(input("Kérek egy egész számot: "))
```

Példa reverse...

Feladat:

Hozz létre egy listát [1,2,3,4,5], majd fordítsd meg a sorrendjét Python kóddal.

```
lista = [1,2,3,4,5]  
forditott_lista = lista[::-1]  
print(forditott_lista)
```



```
[5,4,3,2,1]
```

Köszönöm a figyelmet!

Python kurzus - 3. előadás

Tárgyfelelős: Dr. Tejfel Máté

Előadó: Dr. Király Roland

ELTE IK



Tematika

1. Logikai és bitenkénti műveletek
2. If - else és elif használata
3. While és for ciklusok
4. Függvények bevezetése, paraméterek, visszatérési értékek
5. Függvények és típusok
6. Lista kifejezések (advanced)

A prezentációhoz tartozó forráskódok

[ea.py](#) (pdf állomány megnyitása)

[ealistsvsvez.py](#) (pdf állomány megnyitása)

[errorp.py](#) (pdf állomány megnyitása)

[problem.py](#) (pdf állomány megnyitása)

Logikai és bitenkénti műveletek

- Logikai értékek: **True, False**
- Logikai műveletek: **not, and, or, ^ (xor)**
- Logikai kifejezések:

**Példák: $\text{not (A and B)} = (\text{not A}) \text{ or } (\text{not B})$
 $\text{not (A or B)} = (\text{not A}) \text{ and } (\text{not B})$**

- Bitenkénti műveletek ($\sim$, $\&$, $|$, $\wedge$) ☐

a	b	$\sim b$	$a \& b$	$a b$	$a \wedge b$
1	1	0	1	1	0
1	0	1	0	1	1
0	1	0	0	1	1
0	0	1	0	0	0

- Bináris balra és jobbra eltolás ($\ll$, $\gg$)

```
szám = 15  
  
sz_jobbra = szám >> 1      # Osztunk 2-vel ( // )  
  
sz_balra = szám << 2       # Szorzunk 4-gyel  
  
print(szám, sz_jobbra, sz_balra)  
  
# 15 7 60
```

- Műveleti azonosságok, ha b egy bit:

$$b \wedge 1 = \sim b \quad b \wedge 0 = b$$

- Példa:

egyes bitek
lekérdezése &-del:

```
If flag_register & the_mask:
```

```
    # A vizsgált bit 1 (set)
```

```
else:
```

```
    # A vizsgált bit 0 (reset)
```

```
flag_register = 45  #0b00101101
```

```
the_mask = 4       #0b00000100
```

```
if flag_register & the_mask:
```

```
    print('A vizsgált bit 1')
```

```
else:
```

```
    print('A vizsgált bit 0')
```

```
#A vizsgált bit 1
```

Az if - else és az elif használata

Feltételes végrehajtás:
a logikai kifejezés
kiértékelése szerint

- Az if - else
utasítás

if feltétel:

utasítások, ha igaz a feltétel

else:

utasítások, ha hamis a feltétel

```
if kánikula_van:
```

```
    menj_strandra()
```

```
else:
```

```
    menj_sétálni()
```


- Az elif utasítás

```
if jó_az_idő:  
    menj_sétálni()  
elif van_jegy:  
    menj_moziba()  
elif van_szabad_asztal:  
    menj_étterembe()  
else:  
    tévézz_otthon()
```

- Beágyazott if utasítás

```
if jó_az_idő:  
    if van_jó_étterem:  
        menj_ebédelni()  
    else:  
        menj_sétálni()  
else:  
    if van_jegy:  
        menj_moziba()  
    else:  
        tévézz_otthon()
```

A while és a for ciklus

- A while ciklus:

```
while feltétel:  
    utasítás_1  
    utasítás_2  
    .  
    .  
    utasítás_n
```

```
i = 0          #kezdőérték megadása  
while i < 5:    #ciklusfeltétel megadása  
    print(i)    #ciklusmag  
    i += 1      ##léptetés  
print("léptetés után", i)
```

Kimenet:

0

1

2

3

4

léptetés után 5

```
while True:  
    print(„Benn ragadtam egy ciklusban.”)
```

- Végtelen ciklus

Interrupt a billentyűzetről: Ctrl + c

- A for ciklus:

```
for i in range(10):  
    print(„Az i aktuális értéke", i)
```

```
# Az i aktuális értéke 0
```

```
# Az i aktuális értéke 1
```

```
# Az i aktuális értéke 2
```

```
# Az i aktuális értéke 3
```

```
# Az i aktuális értéke 4
```

```
# Az i aktuális értéke 5
```

```
# Az i aktuális értéke 6
```

```
# Az i aktuális értéke 7
```

```
# Az i aktuális értéke 8
```

```
# Az i aktuális értéke 9
```

- A for ciklus:

```
for i in range(5):           #ciklusfeltétel megadása
    print(i, end=")         #ciklusmag
    print(i*'#')
print("léptetés után", i)
```

Kimenet:

0

1#

2##

3###

4####

léptetés után 4

- A for ciklus:

```
for i in range(2,10,3):  
    print(i)
```

Kimenet:

```
#2 5 8
```

```
for i in range(10,2,-3):  
    print(i)
```

Kimenet:

```
#10 7 4
```

A break és continue utasítások

break:

azonnal kilép a ciklusból és a ciklus után a következő utasításra ugrik

continue:

a ciklusmag végére ugrik és megkezdődik a következő forduló, a feltétel kiértékelése

```
#break példa
print("break alkalmazása:")
for i in range(1, 6):
    if i == 3:
        break
    print("A ciklusban:", i)
print("A cikluson kívül.")
```

```
#continue példa
print("continue alkalmazása:")
for i in range(1, 6):
    if i == 3:
        continue
    print("A ciklusban:", i)
print("A cikluson kívül.")
```

break alkalmazása:

```
A ciklusban: 1
A ciklusban: 2
A cikluson kívül.
```

continue alkalmazása:

```
A ciklusban: 1
A ciklusban: 2
A ciklusban: 4
A ciklusban: 5
A cikluson kívül.
```

A while / for ciklusok és az else ág:

```
i = 1  
  
while i < 5:  
    print(i)  
    i += 1  
  
else:  
    print("else:", i)  
    print(i*'# ')
```

Kimenet:

```
1  
2  
3  
4  
  
else: 5  
  
#####
```


A while / for ciklusok és az else ág:

```
for i in range(5):
```

```
    print(i)
```

```
else:
```

```
    print("else:",i)
```

```
i = 100
```

```
for i in range(4, 0):
```

```
    print(i)
```

```
else:
```

```
    print("else:", i)
```

Kimenet:

0

1

2

3

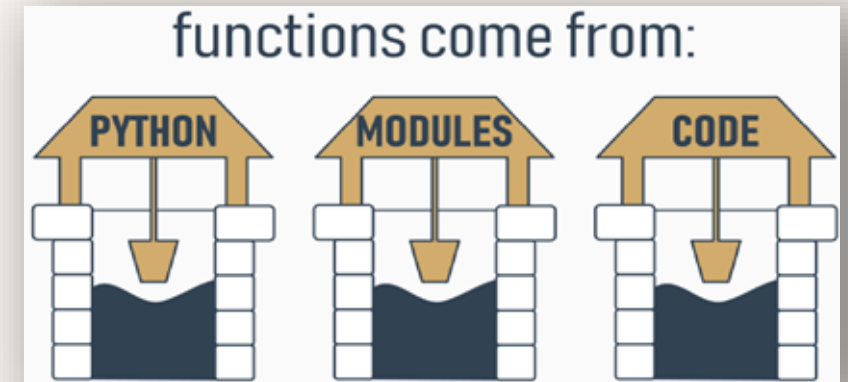
4

else: 4

else: 100

Függvények bevezetése, paraméterek, visszatérési értékek

- Miért van szükségünk függvényekre?
 - *Pl. ha egy adott kódrészlet sokszor ismétlődik a programban*
 - *Részfeladatokra bontás - decomposition*
- Honnan jönnek a függvények?



- Szintaxis:

```
def függvény_neve():  
    #függvény_utasításai
```

- Az első függvény:

```
def köszön():  
    print("Hello!")  
  
köszön()  
Kimenet:  
„Hello”
```

Type

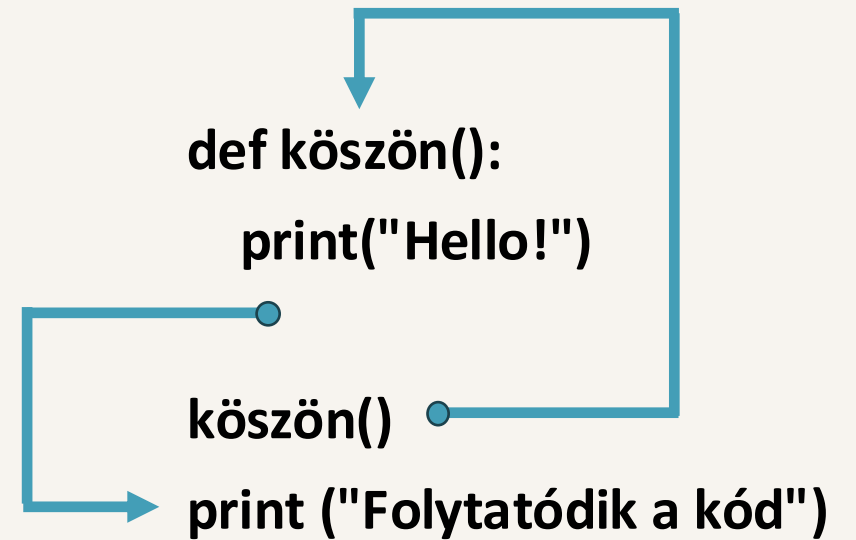
```
def pick(l: list, index: int = 0) -> int:  
    return l[index]
```

Ez nem valódi ellenőrzés!

Így lehet ellenőrizni:

```
def fun(l: list, index: int = 0) -> int:  
    if not isinstance(l, list):  
        raise TypeError  
    return l[index]
```

- Hogyan működnek a függvények:



- Paraméterezett függvények:

```
def függvény(paraméter):  
    ###  
def kiír(szám):  
    print("A kapott érték:", szám)  
kiír(12)
```

- Pozicionális és paraméter neve szerinti paraméterátadás:

```
def összead(a, b, c):  
    print(a + b + c)  
összead(1, 2, 3)  
összead(1, 2, c = 3)  
összead(c = 3, a = 1, b = 2)
```

```
összead(3, a = 1, b = 2)
```

TypeError: összead() got multiple values for argument 'a'

Példák paraméter átadá

```
def bemutatkozás(vezeték_név, keresztnév):  
    print ("Jó napot, a nevem: ", vezeték_név, keresztnév)  
  
bemutatkozás(vezeték_név = "Nagy", keresztnév = "Virág")  
bemutatkozás(keresztnév = "Matyi", vezeték_név = "Ludas")
```

```
def bemutatkozás(vezeték_név, keresztnév = "Lala"):  
    print("Jó napot, a nevem: ", vezeték_név, keresztnév)
```

```
bemutatkozás("Tündér", "Ilona")
```

```
bemutatkozás("Tündér")
```

```
bemutatkozás(keresztnév = "Ilona")
```

```
TypeError: bemutatkozás() missing 1 required positional argument: 'vezeték_név'
```

Hatások és eredmények : a return

Visszatérés érték nélkül **utasítás** Visszatérés értékkel

```
def viisszaszámlál(ez = True):
```

```
    print("Három...")
```

```
    print("Kettő..")
```

```
    print("Egy...")
```

```
    if not ez:
```

```
        return
```

```
    print("Indulás!")
```

```
viisszaszámlál()
```

```
viisszaszámlál(False)
```

```
Három...
```

```
Három...
```

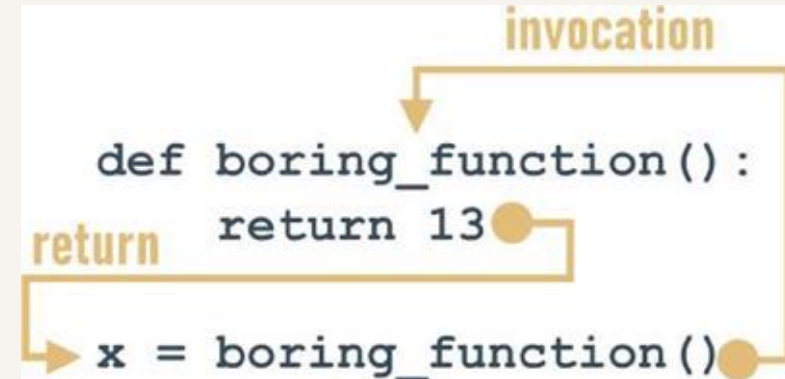
```
Kettő...
```

```
Kettő...
```

```
Egy...
```

```
Egy...
```

```
Indulás!
```



```
def boring_function():
```

```
    print("'Boredom Mode' ON.")
```

```
    return 13
```

```
print("This lesson is interesting!")
```

```
boring_function()
```

```
print("This lesson is boring ...")
```


• Láthatóság

```
def példa_függvény():  
    print("Ismered ezt a változót?", x)  
  
x = 5  
  
példa_függvény()  
print(x)  
  
def másik_függvény():  
    z = 10  
    másik_függvény()  
  
print("Ismered ezt a változót?", z)  
  
-> NameError: name 'z' is not defined  
  
Ismered ezt a változót? 5  
  
-> 5
```

```
def páros(n):  
    if n % 2 == 0:  
        return True  
  
print(páros(4))  
print(páros(3))
```

Kimenet:

True

None

- A None kulcsszó:

```
érték = None
```

```
if érték is None:
```

```
    print("Sajnálom, de nem adtál meg értéket.")
```

```
print(None+1)
```

Sajnálom, de nem adtál meg értéket.

TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'

- Listák ciklussal:

```
def lista(n):  
    li = []  
    for i in range(0, n):  
        li.append(i)  
    return li  
print(lista(6))  
  
[0, 1, 2, 3, 4, 5]
```

```
def lista_új(n):  
    li_új = []  
    for i in range(0, n):  
        li_új.insert(0,i)  
    return li_új  
print(lista_új(6))  
  
[5, 4, 3, 2, 1, 0]
```

- Listák ciklussal:

```
def fibonacci(n):  
    a, b = 1, 1  
  
    while a < n:  
        print(a, end=' ')  
  
        a, b = b, a + b  
  
fibonacci(10)
```

1 1 2 3 5 8

```
def fib2(n):  
    a, b = 1, 1  
  
    fibo = []  
  
    for i in range(n):  
        fibo.append(a)  
  
        a, b = b, a + b  
  
    return fibo  
  
print(fib2(10))  
  
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

Listakifejezések

```
n = len(a)
```

Ciklus listakifejezéssel

```
i = 0
```

```
n = len(a)
```

```
# create list
```

```
# Hagyományos megoldás: üres lista, majd append a ciklusban.
```

```
lv = []for i in range(100):
```

```
    lv.append(i*2)
```

```
# first advantage (quick and readable)
```

```
# Ugyanez tömörebben és általában olvashatóbban.
```

```
[x*2 for x in range(100)]
```

IF listakifejezés sel

```
# IF
```

```
evens = []for i in range(100):
```

```
    if i % 2 == 0:
```

```
        evens.append(i)
```

```
# A fenti páros szűrés egy sorban list comprehensionnel.
```

```
evens = [i for i in range(100) if i % 2 == 0]
```

```
words = ["alma", "körte", "dió", ]
```

IF listakifejezéssel

```
# Szökőév ellenőrzése
```

```
# Szabály: szökőév, ha osztható 400-zal, vagy osztható 4-gyel, de nem 100-zal.
```

```
evek = [2024, 2004, 1975, 1902]
```

```
szokoevek = []
```

```
for ev in evek:
```

```
    # Ha nem szökőév, a ciklus következő elemére ugrunk.
```

```
    if not (ev % 400 == 0 or (ev % 4 == 0 and ev % 100 != 0)):
```

```
        continue
```

```
    szokoevek.append(ev)
```

```
print(szokoevek)
```

```
leapyears = []
```

```
# Ugyanez list comprehensionnel, közvetlen szűrőfeltétellel.
```

```
leapyears = [ev for ev in evek
```

```
    if ev % 400 == 0 or (ev % 4 == 0 and ev % 100 != 0)]
```

```
print(szokoevek)
```


Jelszavak ellenőrzése

```
szavak = ["a_lma", "Okorte", "A_dio", "Aszilva"]
jelszavak = []

for szo in szavak:
    # Csak kisbetűvel kezdődő szavak maradjanak.
    if szo[0].upper() == szo[0]:
        continue

    # Kötelező legyen benne aláhúzásjel.
    if "_" not in szo:
        continue

    # Ne kezdődjön számmal (itt konkrétan '0'-val).
    if szo[0] == "0":
        continue

    jelszavak.append(szo)

    #s, *b, c = [1, 2, 3, 4]

print(jelszavak)
```

Több feltétel

```
# Több egymás utáni feltétel:  
mindnek teljesülnie kell.  
jelszavak = [szo for szo in szavak  
              if szo.upper() != szo  
              if "_" in szo  
              if szo[0] != "0"]
```

```
print(jelszavak)
```

```
tensor1 = [[1, 2, 4, 3, 4, 3], [4, 5, 5, 6], [7, 8, 8, 9]]  
flatlist = []
```

Több dimenziós listák

```
tensor1 = [[1, 2, 4, 3, 4, 3], [4, 5, 5, 6], [7, 8, 8, 9]]
flatlist = []

# Kétdimenziós lista „kilapítása” beágyazott ciklusokkal.
for row in tensor1:
    for element in row:
        flatlist.append(element)

print(flatlist)

print("[")

for row in tensor1:
    print(" ", row,)

print("]")

print(flatlist)

flatlist = []

# Kilapítás comprehensionnel:
# előbb soronként, azon belül elemenként.

flatlist = [element for row in tensor1 for element in row]

print(flatlist)
```

Kategorizálás listakifejezéssel

```
# Kategorizálás

# Feltételes kifejezés (ternary): ha/egyébként logika egy sorban.

szavak = ["Alma", "alma", "Körte", "KÖRTE"]

formatum = []

for szo in szavak:

    if szo[0].upper() != szo[0]:

        formatum.append("Kisbetűs")

    else:

        formatum.append("Nagybetűs")

print(formatum)

elemek = ["Kisbetűs" if szo[0].upper() != szo[0] else "Nagybetűs" for szo in szavak]

print(elemek)
```

Dict comprehension

```
# Dict comprehension#
```

```
Lista (kulcs, érték) párokból szótár készítése.
```

```
l = [("Alma", 560), ("Körte", 790), ("Szilva", 990)]
```

```
d = {k:v for k, v in l}
```

```
print(d)
```

```
# set comprehension
```

```
# Halmaznál az ismétlődő elemek automatikusan kiesnek.
```

```
szamok = [1, 2, 3, 3, 4, 4, 5, 6, 7, 9, 9, 8]
```

```
nums = {x for x in szamok}
```

```
print(nums)
```

```
# Generátoros összegzés: itt nem szükséges köztes lista.
```

```
numbers = sum(i*2 for i in szamok)
```

```
print(numbers)
```

```
# []!!!
```

Köszönöm a figyelmet!

ELTE IK

Python kurzus – 4. előadás

Tárgyfelelős: Dr. Tejfel Máté

Előadó: Dr. Király Roland

ELTE IK



Tematika

1. Ellenőrző teszt
2. Hibakezelés, try - except blokkok, kivételek
3. Függvények: paraméter átadás, pozicionális- és kulcsszó-argumentumok
4. Anonim és lambda függvények
5. Beépített függvények és modulok

A prezentációhoz tartozó forráskódok

[all_ea_indorder.py](#)(pdf)
[debug.py](#) (pdf)
[ea.py](#) (pdf)
[ea2026.py](#) (pdf)
[ex.py](#) (pdf)
[exceptionhandling.py](#) (pdf)
[n.py](#) (pdf)
[polinom.py](#) (pdf)
[test.py](#) (pdf)
[test2.py](#) (pdf)
[testz.py](#) (pdf)
[trac.py](#) (pdf)
[vol.py](#) (pdf)

1. Ellenőrző teszt - Canvas

- Listaműveletek
- String műveletek
- Ciklusok
- Függvény definíció, paraméterátadás

2. Hibakezelés

Hibák az adatokban és hibák a kódban – a tesztelés kötelező

- **Adat hibák:**

helyes kód, hibás adatbevitel

```
x = int(input('Kérek egy egész számot: ,))
```

```
Kérek egy egész számot: 120
```

```
ValueError: invalid literal for int(): '120'
```

2. Hibakezelés

- **Kódolási hibák:** hibakeresés kódelemzéssel, debugger használatával, AI eszközökkel

```
Sugár = 10
```

```
terület = sugár**2*3.14
```

```
NameError: name 'sugár'  
is not defined.
```

```
lista = []
```

```
a = lista[-1]
```

```
IndexError: list index out of range
```

További példák:

```
for i in range(5):  
    if i % 2 != 0:  
        print(i, 'páros')  
    else:  
        print(i, 'páratlan')
```

0 páratlan
1 páros
2 páratlan
3 páros
4 páratlan

```
import math  
x = float(input("Kérek egy számot: "))  
print(x, "négyzetgyöke", math.sqrt(x))
```

Kérek egy számot: -1
ValueError: math domain error

```
x = 2.  
y = '3'  
print(x + y)
```

TypeError: unsupported operand
type(s) for +: 'float' and 'str'

```
from math import pi #  
use the value of pi defined in the math library  
def main():  
    radius = input("Enter the radius: ")  
    height = input("Enter the height: ")  
    conevol = 1/3*pi*radius**2*height #  
    cone volume  
    cylvol = pi*r**2*height #  
    cylinder volume  
    spherevol = 4/3*Pi*radius**3 #  
    sphere volume  
  
    print ('The volume of a cone: ',conevol)  
    print ("The volume of a cylinder: ",cylvol)  
    print ("The volume of a sphere: ",spherevol)  
main()
```

Hibaüzenetek!

Kivételezés: try – except blokkok

- **Hiba esetén:** 1. leáll a program 2. egy kivétel generálódik
- **Hogyan kezeljük a kivételeket:** try – except utasítás

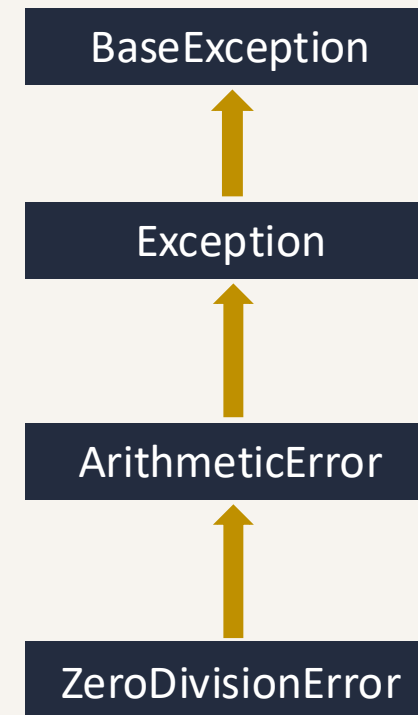
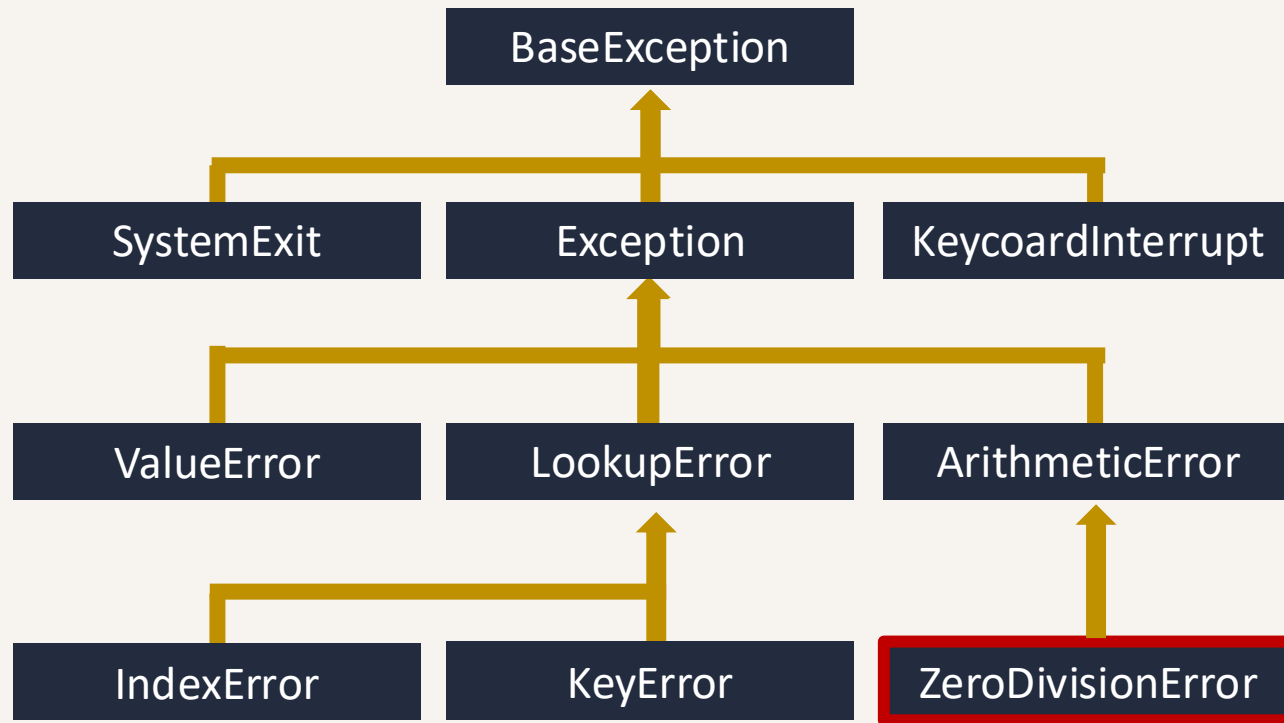
```
try:  
    x = int(input("Kérek egy számot: "))  
    y = 1 / x  
    print(y)  
  
except ZeroDivisionError:  
    print("Nullával nem lehet osztani!")  
  
except ValueError:  
    print("Egy számot adjon meg!")  
  
except:  
    print("Valami más hiba történt!")  
  
print("vége")
```



```
Kérek egy számot: 2  
0.5  
vége  
  
Kérek egy számot: 0  
Nullával nem lehet osztani!  
vége  
  
Kérek egy számot: a  
Egy számot adjon meg!  
vége  
  
Kérek egy számot: Valami más hiba történt!  
vége
```

Kivételek - Exceptions

A Python3 63 beépített kivételt definiál, amelyek egy **fa alakú hierarchiát** alkotnak:



Kivételek kiváltása: a raise utasítás

A **raise** utasítás az adott **exception** kivételt úgy fogja kiváltani, mintha az a szokásos (természetes) módon keletkezett volna:

Hiba eset felmerülésekor:

```
def próba(n):  
    try:  
        return 1 / n  
  
    except ArithmeticError:  
        print("Aritmetikaiprobléma!")  
  
    return None  
próba(0)          # a hiba kiváltása  
print("vége")
```

```
Aritmetikai probléma!  
vége
```

A **raise** alkalmazásával:

```
def próba(n):  
    raise ZeroDivisionError  
  
try:  
    próba(2) #nem okoz hibát  
  
except ArithmeticError:  
    print("Mi történt? Egy hiba?")  
print("vége")
```

```
Mi történt? Egy hiba?  
vége
```

A raise utasítás

```
def próba(n):  
    try:  
        return n / 0  
    except:  
        print("Megint egy hiba!")  
        raise  
  
try:  
    próba(2)  
except ZeroDivisionError:  
    print("Igen, ez egy aritmetikai hiba.")  
  
print("vége")
```



Megint egy hiba!
Igen, ez egy aritmetikai hiba.
vége

A raise nélkül:

Mi történt? Egy hiba?
vége

Beépített kivételek

- **ArithmeticError**

Aritmetikai műveletek által okozott kivételek, mint a nullával osztás vagy egy argumentum érvénytelen tartománya.

- **AssertionError**

Az assert utasítás sikertelen.

- **AttributeError**

Helytelen attribútumra való hivatkozás vagy értékadás.

- **BaseException**

A legáltalánosabb kivétel, az **except** és **except BaseException** kifejezések egyenértékűek.

- **IndexError**

Nem található egy index a sorozatban.

- **KeyError**

Nem létező dictionary kulcsra hivatkozunk.

Beépített kivételek

- **KeyboardInterrupt**
A felhasználó megszakítja a program végrehajtását
- **NameError**
Nem meghatározott lokális vagy globális név keresésekor.
- **SyntaxError**
Az elemző szintaktikai hibát talált.
- **TypeError**
Beépített operátornak vagy függvénynek helytelen típust adunk át.
- **ValueError**
Argumentum hiba esetén, melyet a TypeError nem tartalmaz, ha argumentumokként érvénytelen adatokat adtunk meg.
- **ZeroDivisionError**
Osztasnál vagy maradék (modulo) műveletnél 0 második argumentum esetében.

```
def próba(n):  
    try:  
        return n / 0  
    except Exception as error:  
        print("Megintegyhiba!")  
        print(error)
```

Megintegyhiba!
division by zero

```
import math  
try:  
    x = math.sqrt(-1)  
    print(x)  
except Exception as error:  
    print(error)
```

math domain error

```
lista = []  
try:  
    a = lista[-1]  
except Exception as error:  
    print(error)
```

list index out of range

```
try:
```

```
    print("1" + 1)
```

```
except Exception as error:
```

```
    print(error)
```

can only concatenate str (not "int") to str

```
def próba(a, b=2, c):
```

```
    print(a, b, c)
```

SyntaxError: non-default argument follows default argument

Az assert kifejezés

```
import math
x = float(input("Kérek egy számot: "))
assert x >= 0.0
x = math.sqrt(x)
print(x)
```



```
Kérek egy számot: -1
Traceback (most recent call last):
  File "4ea_proba.py", line 3, in <module>
    assert x >= 0.0
AssertionError
```

```
import math
while True:
    x = float(input("Kérek egy számot: "))

    try:
        assert x >= 0.0
        break

    except AssertionError:
        print("A szám negatív!")

x = math.sqrt(x)
print(x)
```



```
Kérek egy számot: 4
2.0
```



```
Kérek egy számot: -1
A szám negatív!

Kérek egy számot: -2
A szám negatív!

Kérek egy számot: 0
0.0
```

A **yield** speciális kulcsszó: generátor függvényekben használt

- A normál függvényeknél a **return** egyszeri értéket ad vissza és megszakítja a függvény futását
- A **yield** visszaad egy értéket és nem fejezi be a függvény végrehajtását

```
def my_generator():  
    print("Első lépés")  
    yield 1  
  
    print("Második lépés")  
    yield 2  
  
    print("Harmadik lépés")  
    yield 3  
  
gen = my_generator()  
print(next(gen))  
print(next(gen))  
print(next(gen))
```



```
Első lépés  
1  
  
Második lépés  
2  
  
Harmadik lépés  
3
```

```
#2 hatványok kiírása  
def kettő_hatvány(n):  
    hatv= 1  
    for iin range(n):  
        yield hatv  
        hatv*= 2  
  
for elemin kettő_hatvány(8):  
    print(elem)
```



```
1  
2  
4  
8  
16  
32  
64  
128
```


3. Függvények: paraméter átadás

Függvény meghívásakor az aktuális paramétert, ha az változó, értékadásban felelteti meg a formális paraméternek, mint lokális változónak.

```
def szorzás(érték, hányszor):
```

```
    érték *= hányszor  
    print(érték, hányszor)  
    print(szám, szorzó)
```

```
szám, szorzó = 5, 2  
print(szorzás(szám, szorzó))  
print(érték, hányszor)
```

```
10 2  
5 2  
None  
NameError: name 'érték' is not  
defined
```

```
def add_citrom(list):
```

```
    list.append("Citrom,,")  
    print("Belső lista:", list)
```

```
gyümölcs = ['Alma', 'Körte',  
            'Narancs']
```

```
add_citrom(gyümölcs)
```

```
print("Eredeti lista:", gyümölcs)
```

```
Belső lista: ['Alma', 'Körte', 'Narancs',  
             'Citrom']
```

```
Eredeti lista: ['Alma', 'Körte',  
               'Narancs', 'Citrom']
```

```
def add_citrom(list):
```

```
    list = ["Citrom"]  
    print("Belső lista :", list)
```

```
gyümölcs = ['Alma', 'Körte',  
            'Narancs']
```

```
add_citrom(gyümölcs)
```

```
print("Eredeti lista:", gyümölcs)
```

```
Belső lista : ['Citrom']
```

```
Eredeti lista: ['Alma', 'Körte',  
               'Narancs']
```

- ***args – pozicionális argumentumok**
 - A * jelöli azt, hogy a függvény **tetszőleges számú** pozicionális argumentumot fogadhat el.
 - Az argumentumok **tuple**-ként kerülnek tárolásra a függvényen belül.
- ****kwargs - kulcsszó-argumentumok (keyword arguments)**
 - **Változó számú** kulcsszó-argumentumot adhatunk át egy függvénynek.
 - A kulcsszó-argumentumok név-érték párokként kerülnek átadásra.
 - A **kwargs a kulcsszó-argumentumokat **szótárként (dict)** tárolja, ahol a kulcsok az argumentumok nevei, az értékek pedig az ezekhez tartozó értékek.

Pozicionális (*args) és kulcsszó-argumentumok (**kwargs)

```
def mindent_kezel(név, *args, **kwargs):  
    print(f"Név: {név}")  
    print("Pozicionális argumentumok:", args)  
    print("Kulcsszó-argumentumok:", kwargs)  
  
mindent_kezel("Anna", 10, 20, 30, város="Sopron", kor=20)
```

```
Név: Anna  
Pozicionális argumentumok: (10, 20, 30)  
Kulcsszó-argumentumok: {'város': 'Sopron', 'kor': 20}
```

```
def összeg(*args):  
    return sum(args)  
print(összeg(1, 2, 3))  
print(összeg(5, 10, 15, 20))
```

```
6  
50
```

```
def profil(**kwargs):  
    for kulcs, érték in kwargs.items():  
        print(f"{kulcs.capitalize()}: {érték}")  
profil(név="Anna", kor=20, város="Sopron")
```

```
Név: Anna  
Kor: 30  
Város: Sopron
```

4. Anonim és lambda függvények

lambda argument1, argument2, ... : kifejezés

- A **lambda** kulcsszóval egy **anonim** (névtelen) függvényt definiálunk
- Az argumentumok a függvény bemeneti paraméterei
- A kettőspont utáni kifejezés a függvény visszatérési értéke lesz
- A lambda függvényekben nincs return, hanem automatikusan visszatérnek a kifejezés kiértékelt eredményével

```
összeadás = lambda a, b: a + b  
print(összeadás(2, 5))
```

7

```
print((lambda a, b: a + b)(2, 5))
```

7

```
hatvány= lambda x, k: x ** k  
print(hatvány(2, 5))
```

32

lambda argument1, argument2, ... : kifejezés

```
print((lambda a: a + 6)(2)) #eltolás +6-tal
```

8

```
print((lambda a, b: a - b)(1,5)) #kivonás
```

-4

```
print((lambda a, b: a * b)(3,4)) #szorzás
```

12

```
print((lambda a, b: a ** b)(5,3)) #hatványozás
```

125

```
print((lambda a,b: (a**2 + b**2)**0.5)(5,12))
```

#Pitagorasz-tétel

13.0

További példák:

```
def fv_érték(args, polinom):  
    for x in args:  
        print('g(', x,')=', polinom(x), sep=',')
```

```
fv_érték([x for x in range(-2,4)],  
lambda x: 3 * x**2 - 6 * x + 3)
```

$g(-2)=27$

$g(-1)=12$

$g(0)=3$

$g(1)=0$

$g(2)=3$

$g(3)=12$

A lambda alkalmazása magasabb rendű függvényekben

A **map** függvény: egy függvényt alkalmaz minden elemre egy iterálható objektumban. Itt lambda függvényt használunk a számok négyzetre emeléséhez:

```
számok = [1, 2, 3, 4]
négyzetek = list(map(lambda x: x ** 2, számok))
print(négyzetek)
```

[1, 4, 9, 16]

A **filter** függvény: lambda függvényt használ arra, hogy kiválassza azokat az elemeket, amelyek egy adott feltételnek megfelelnek. Itt a páros számokat szűrjük ki:

```
számok = [1, 2, 3, 4, 5, 6]
párosok = list(filter(lambda x: x % 2 == 0, számok))
print(párosok)
```

[2, 4, 6]

5. Beépített függvények

- **A beépített függvények** a Python alapértelmezett részei. Segítségükkel különféle műveleteket hajthatunk végre extra könyvtárak importálása nélkül.
- Csoportosítva a legfontosabb beépített függvények:
 - **Matematikai:** `abs`, `round`, `min`, `max`, `sum`
 - **Típusokkal kapcsolatos:** `type`, `int`, `str`, `bool`, `float`, `list`, `dict`
 - **Iterációs:** `range`, `map`, `filter`, `zip`, `sorted`, `reversed`, `enumerate`
 - **Fájlkezelési:** `open`, `input`, `print`
 - **Funkcionális programozással kapcsolatos:** `lambda`, `eval`, `exec`
 - **Kivételkezelési:** `raise`, `assert`
 - **Objektumkezelési:** `dir`, `vars`, `id`, `classmethod`, `staticmethod`
- Pythonban a beépített függvények listáját a **`dir(__builtins__)`** paranccsal lehet megtekinteni. (pl.: `print(__builtins__)`)

Beépített modulok

- A beépített modulokat a Python-telepítés tartalmazza, használatukhoz nincs szükség külön telepítésre, csak importálni kell őket a programba.
- **Importálás:**
 - **import modul_név:** A teljes modult importálja.
 - **import modul_név as alias:** Alias nevet ad a modulnak.
 - **from modul_név import elem_név:** Csak egyes elemeket importál a modulból.
 - **from modul_név import *:** Minden elemet importál a modulból.

Néhány fontos beépített modul kategóriák szerint:

- **Matematikai és statisztikai:** math, cmath, random, statistics
- **Dátum és időkezelés:** datetime, time, calendar
- **Szövegfeldolgozás:** re, string, textwrap, unicodedata
- **Fájl- és mappakezelés:** os, os.path, pathlib, shutil, tempfile
- **Fájlformátumok kezelése:** json, csv, pickle, xml
- **Rendszerszintű :** sys, platform, subprocess, logging
- **Hálózatkezelési:** socket, http.client, urllib, ssl, ftplib
- **Adatstruktúrák és algoritmusok:** collections, array
- **Titkosítás és hash-elés:** hashlib, secrets
- **Többszálú és párhuzamos programozás:** threading, multiprocessing
- **Tesztelési és hibakezelési:** unittest, doctest
- **Adatbáziskezelés:** sqlite3, dbm, csv
- **GUI és felhasználói felületek:** tkinter

Package (csomag) – modul – függvény hierarchia

- **Függvény:** Egyedi, kisebb kódegység, amely egy műveletet hajt végre. Például egy matematikai művelet vagy egy szöveges művelet
- **Modul:** Egy fájl, amely Python kódot tartalmaz, és több függvényt, változót vagy osztályt foglalhat magába. Például a math modul tartalmazhat matematikai függvényeket.
- **Package:** Egy könyvtár, amely több modult (és alpackage-eket) tartalmaz, lehetővé téve a kód logikai szervezését és újra felhasználhatóságát. Ez a modulok strukturált gyűjteménye.

A Python package-ek installálása

- A Python package-eket leggyakrabban a **Python Package Index**-ből (**PyPI**) lehet installálni. A PyPI a legnagyobb nyilvános Python csomagtár, ahol több tízezer nyílt forráskódú csomag található, amelyeket fejlesztők hoztak létre különféle feladatokhoz, például webfejlesztéshez, adatfeldolgozáshoz, gépi tanuláshoz és sok más területhez.
- **Hogyan lehet installálni Python package-eket?**

A Pythonban a legelterjedtebb eszköz a package-ek telepítéséhez a **pip** nevű csomagkezelő. A **pip** a PyPI-ről tölti le és telepíti a csomagokat a Python környezetbe.

Példa 3.

```
import math #Testek térfogatai lambda
függvényekkel az [1..5] értékekre
kocka_térfogat = lambda oldal: oldal ** 3
gömb_térfogat = lambda átmérő: (4 / 3) *
math.pi * (átmérő / 2) ** 3
henger_térfogat = lambda átmérő, mag: math.pi
* (átmérő / 2) ** 2 * mag
kúp_térfogat = lambda átmérő, mag: (1 / 3) *
math.pi * (átmérő / 2) ** 2 * mag
elemek = range(1, 6) #
Intervallum [1..5]
kocka_térfogatok = [kocka_térfogat(x) for x in
elemek]
gömb_térfogatok = [gömb_térfogat(x) for x in
elemek]
henger_térfogatok = [henger_térfogat(x, x) for x
in elemek]
kúp_térfogatok = [kúp_térfogat(x, x) for x in
elemek]
print("Kocka térfogatok:", kocka_térfogatok)
print("Gömb térfogatok:", gömb_térfogatok)
print("Henger térfogatok:", henger_térfogatok)
print("Kúp térfogatok:", kúp_térfogatok)
```

```
import random
```

```
def függvény(args, szabály):
```

```
    for x in args:
```

```
        print(szabály(x), end=', ,')
```

```
    print()
```

```
lista = [random.randint(1, 100) for i in range(10)]
```

```
print(lista)
```

```
függvény(lista, lambda x: x % 2)
```

```
függvény(lista, lambda x: x % 5)
```

```
függvény(lista, lambda x: x - 1)
```

```
[5, 64, 70, 63, 99, 75, 8, 82, 80, 48]
```

```
1 0 0 1 1 1 0 0 0 0
```

```
0 4 0 3 4 0 3 2 0 3
```

```
4 63 69 62 98 74 7 81 79 47
```

```
def négyzetszámok(n):
```

```
    for i in range(n):  
        négyzet = (i+1)**2  
        yield négyzet
```

```
for elem in  
    négyzetszámok(10):  
    print(elem)
```

1 4 9 16 25 36 49 64 81 100

```
def háromszögszámok(n):
```

```
    s = 0  
    for i in range(1,n+1):  
        s += i  
        yield s
```

```
for elem in  
    háromszögszámok(10):  
    print(elem, end=' ,')  
print()
```

1 3 6 10 15 21 28 36 45 55


```
számok = [1, 2, 3, 4, 5]
köbszámok =
list(map(lambda x: x ** 3, számok))
print(köbszámok)
```

[1, 8, 27, 64, 125]

```
import random
lista = [random.randint(1, 100) for i in
range(10)]
print(lista)
print(list(filter(lambda x: x % 2 == 0,
lista)))
print(list(filter(lambda x: x % 2 != 0,
lista)))
```

[26, 99, 74, 53, 82, 53, 66, 93, 95,
79]
[26, 74, 82, 66]
[99, 53, 53, 93, 95, 79]

```
data = [random.randint(-  
10,10) for x in range(10)]  
filtered = list(filter(lambda x:  
x >= 0 and x % 2 == 1, data))  
print(data)  
print(filtered)
```

```
[7, 10, -8, 3, -4, -2, 0, -1, 5, 6]  
[7, 3, 5]
```

Példa 1.

```
def pizza_készítés(méret, *feltétek,
**extra_opciók):
    print(f"Készítünk egy {méret} pizzát
a következő feltétekkel:")
    for feltét in feltétek:
        print(f"- {feltét}")

    if extra_opciók.get("extra_sajt"):
        print("Extra sajtot teszünk rá!")
    if extra_opciók.get("glutén_mentes"):
        print("Gluténmentes tésztát
használunk.")

pizza_készítés("nagy", "szalámi",
"gomba", extra_sajt=True,
glutén_mentes=False)
```

Példa 2.

```
def rendelés_összegzés(**kwargs):
    print("Rendelés részletei:")
    for kulcs, érték in kwargs.items():
        print(f"{kulcs.capitalize()}: {érték}")
    if "fizetési_mód" not in kwargs:
        print("Figyelem! Nincs fizetési mód megadva.")
    if kwargs.get("ingyen_szállítás", False):
        #Ha ingyen_szállítás nem létezik,
        #False értéket ad
        vissza.
        print("Az ingyenes szállítás elérhető.")
    else:
        print("Nincs ingyenes szállítás.")

rendelés_összegzés(termék="Laptop",
                    ár=350000, darab=1,
                    fizetési_mód="bankkártya",
                    ingyen_szállítás=True)
```

Példa 4.

```
def pascal_háromszög(n):  
    #Generálja a Pascal-háromszög első n sorát.  
    háromszög = []  
    for sor in range(n):  
  
        # Minden sor egy új lista, amelyet 1-gyel kezdünk, és  
        új_sor = [1] * (sor + 1)  
  
        # a sor belső elemeit az előző sor elemeiből számoljuk  
        for i in range(1, sor):  
            új_sor[i] = háromszög[sor - 1][i - 1] + háromszög[sor - 1][i]  
        háromszög.append(új_sor)  
    return háromszög  
  
def háromszög_megjelenít(háromszög):  
  
    #Megjeleníti a P.-háromszöget formázottan  
    max_szélesség = len(" ".join(map(str, háromszög[-1])))  
    # A legutolsó sor szélessége  
    for sor in háromszög:  
        sor_str = " ".join(map(str, sor))  
        print(sor_str.center(max_szélesség))  
  
n = int(input("Add meg, hány sor Pascal-háromszöget szeretnél  
generálni: "))  
háromszög = pascal_háromszög(n)  
                                     # A Pascal-háromszög generálása,  
háromszög_megjelenít(háromszög)  
                                     # és megjelenítése
```

Összegzés

- Bemutattuk a hibakezelést, a try - except blokkok használatát és a kivételeket
- Elemeztük a függvény paraméter átadást, és használtunk pozicionális és kulcsszó függvényargumentumokat
- Készítettünk lambda függvényeket
- Áttekintettük a beépített függvények és modulok tárházát
- Bevezettük a package fogalmát
- Bemutattunk néhány példa programot

Köszönöm a figyelmet!

ELTE IK

Python kurzus – 5. előadás

Tárgyfelelős: Dr. Tejfel Máté

Előadó: Dr. Király Roland

ELTE IK



Tematika

1. Bevezetés az objektumorientált programozás világába
2. Osztályok és objektumok alapjai
3. Konstruktorkok és destruktorkok
4. Osztályattribútumok és metódusok
5. Öröklődés alapjai

A prezentációhoz tartozó forráskódok

[ea5.pdf\(pdf állomány megnyitása\)](#)

[ea5.py\(py állomány megnyitása\)](#)

[ea5.zip\(tartalom letöltése\)](#)

Bevezetés az Objektumorientált programozásba

- Mi az objektumorientált programozás?
- Előnyök: könnyebb karbantarthatóság, moduláris felépítés, kód újrahasznosítása.
- Az OOP négy alapelve:
 1. **Egységbezárás (Encapsulation):** Az adatok és a hozzájuk tartozó műveletek osztályon belüli védelme.
 2. **Absztrakció (Abstraction):** Csak a lényeges információk elérése és a bonyolult részletek elrejtése.
 3. **Öröklődés (Inheritance):** Az osztályok tulajdonságainak és metódusainak újrafelhasználása és kibővítése.
 4. **Polimorfizmus (Polymorphism):** Ugyanazon nevű metódus különböző viselkedése különböző osztályok objektumai esetében.

Bevezetés az Objektumorientált programozásba

- Az **objektumorientált programozás** (OOP) egy programozási paradigma, amely az adatokat és az ezekkel kapcsolatos műveleteket objektumokba szervezi. Az OOP fő célja, hogy a programok strukturáltabbak, könnyebben karban-tarthatók és újrahasznosíthatók legyenek. Ez a paradigma négy alapelve épül: **egységbezárás, absztrakció, öröklődés és polimorfizmus**.
- Az **objektumorientált programozás** egy programozási stílus, amelyben a szoftver rendszert objektumokból építjük fel. Az objektumok a valós világban létező dolgokat modellezik: tulajdonságokkal (**attribútumokkal**) rendelkeznek, és műveleteket hajtanak végre (**metódusokkal**). Az OOP célja, hogy a valós világ elemeit leképezze a programkódba, könnyebbé téve a nagy, bonyolult programok felépítésének megértését és kezelését.

Az OOP előnyei:

1. **Modularitás:** Az osztályokba és objektumokba szervezett kód könnyebben kezelhető és karbantartható. Az egyes modulok (osztályok) egymástól függetlenek, így könnyen változtathatók anélkül, hogy az egész programot át kellene írni.
2. **Újrahasznosíthatóság:** Az osztályok és objektumok többször felhasználhatók különböző programokban vagy a program különböző részein.
3. **Karbantarthatóság:** Az objektumok és osztályok hierarchiájának és szerkezetének köszönhetően a programok könnyebben bővíthetők, javíthatók.
4. **Rugalmas tervezés:** Az öröklődés és a polimorfizmus lehetőséget ad arra, hogy az alapvető programfunkciókat újra lehessen definiálni anélkül, hogy az alapvető logikát át kellene írni.

A class működése – type



Egységbezárás (Encapsulation)

```
class BankAccount:

    def __init__(self, balance):
        self.__balance = balance    # privát attribútum

    def deposit(self, amount):
        self.__balance += amount

    def withdraw(self, amount):
        if amount <= self.__balance:
            self.__balance -= amount
        else: print("Nincs fedezet ")
```

Az **egységbezárás** az a folyamat, amelyben az osztályokban az adatok (attribútumok) és a velük végzett műveletek (metódusok) együtt kerülnek kezelésre. Az osztály megvédi az adatokat a közvetlen külső hozzáféréstől, ezzel biztosítva az adatok integritását.

Példa egy **BankAccount** osztály, az osztály belsejében tárolhatjuk a bankszámla egyenlegét, és csak meghatározott metódusokon (pl. pénz befizetése, pénz kivétele) keresztül módosíthatjuk azt:

Az **__balance** attribútum el van rejtve, így közvetlenül nem férhetünk hozzá kívülről, ehelyett a **deposit()** és **withdraw()** metódusokon keresztül kezeljük.

Az absztrakció (*Abstraction*)

Az **absztrakció** (itt) a részletek elrejtését jelenti:

- csak azokat a tulajdonságokat és műveleteket tesszük elérhetővé, amelyek az objektum működéséhez szükségesek,
- csak a lényeges információkkal dolgozunk, így a komplexitás csökkenthető.

1. Példa: Egy autót tudunk vezetni anélkül, hogy ismernünk kellene a motor működésének pontos részleteit. Csak a pedálokat és a kormányt használjuk, hogy irányítsuk az autót.

2. Példa: Egy étteremben felszolgált étel esetén nem ismerjük annak elkészítési műveleteit, csak az eredményt kapjuk kézhez.

3. Példa: Egy verem esetén csak a műveleteket (push, pop) látjuk, de az adatok tárolását és a műveletek megvalósítását nem.

Az öröklődés (*Inheritance*)

```
class Vehicle:
    def __init__(self, speed):
        self.speed = speed
    def move(self):
        return f"Halad {self.speed} sebességgel "

class Car(Vehicle):
    def move(self):
        return f"Az autó halad {self.speed} sebességgel "
```

Az **öröklődés** mechanizmusa:

- Egy osztály (gyermekosztály) átveheti egy másik osztály (szülőosztály) tulajdonságait és viselkedését.
- Ez újra felhasználhatóságot biztosít, valamint az örökölt metódusokat és attribútumokat a gyerekosztályok új funkciókkal is bővíthetik.

Példa: Ha van egy **Vehicle** osztályunk, abból örökölhettünk egy **Car** és egy **Bicycle** osztályt, mindkettő ugyanazokat az alapvető jellemzőket (pl. speed) örökölheti, de egyedi viselkedést is kaphatnak.

A polimorfizmus (*Polymorphism*)

```
class Animal:
    def speak(self):
        pass

class Dog(Animal):
    def speak(self):
        return "Woof!"

class Cat(Animal):
    def speak(self):
        return "Meow!"
```

A polimorfizmus:

- Ugyanazt a metódust másképp implementálhatjuk különböző osztályokban.
- Így ugyanaz a művelet (metódus) különböző objektumok esetében eltérő viselkedést mutathat.

Példa: egy **Animal** osztály, amely egy **speak()** metódust tartalmaz. Különböző állatok különböző hangot adnak ki, amikor ezt a metódust meghívjuk.

Osztályok és Objektumok

```
class Dog:                # osztály definíció
    def __init__(self, name):
        self.name = name
my_dog = Dog("Buddy")     #példány definíció
print(my_dog.name)
```

Fogalmak tisztázása: Mi az osztály és mi az objektum?

Példák:

- Osztály definíciója Pythonban: **class** kulcsszó használata.
- Objektum (**példány**) létrehozása egy osztályból.

Osztályok és Objektumok

Az **osztályok** és az **objektumok**:

- A program szerkezetét határozzák meg, az adatok és a funkciók logikus, szervezett formában való kezelését.
- Az **osztály** az adatok (attribútumok) és az ezekkel kapcsolatos műveletek (metódusok) **mintája**.
- Az **objektum** egy konkrét **példány**, amely az osztály alapján jön létre.

Példa: egy autót reprezentáló osztály. Tulajdonságok lehetnek a márka, a model, és metódusok lehetnek az indítás és a gyorsítás.

```
class Car:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model
    def start(self):
        print(f"A(z) {self.brand} {self.model} elindult.")
```

Az osztályban definiáltunk két attribútumot, a **brand** és **model** változókat, amelyek az autó tulajdonságait tárolják.

Az **start** metódus az autó elindítását szimulálja.

Osztályok és Objektumok

Az **objektum** az osztály alapján létrehozott konkrét **példány**, amely az osztály összes attribútumát és metódusát tartalmazza, de ezek egyedi értékekkel rendelkeznek.

Példa folytatása: Létrehozhatunk több autót ugyanabból az osztályból, mindegyik saját márkával és típussal.

```
car1 = Car("Toyota", "Corolla")
car2 = Car("Ford", "Focus")
car1.start()      # Eredmény: A(z) Toyota Corolla elindul.
car2.start()      # Eredmény: A(z) Ford Focus elindul.
```

Az car1 és car2 objektumok az Autó osztály példányai, mindkettő saját értékekkel rendelkezik a brand és model attribútumokban.

Az osztály definíció szintaxisa:

```
class Person:
    def __init__(self, name, age):      # konstruktor metódus
        self.name = name               # attribútum
        self.age = age                 # attribútum
    def greet(self):                   # metódus
        return f"Szia, {self.name} vagyok és {self.age} éves"
```

- Az **__init__** egy speciális metódus, amit **konstruktor** metódusnak nevezünk. Egy új objektum létrejöttkor hívjuk meg. Itt inicializáljuk az objektum attribútumait.
- A **self paraméter** az aktuális példányra mutat, és minden metódusban az első paraméternek kell lennie. A self értékét a hívások során nem kell explicit módon átadni.

Objektum létrehozása:

```
person1 = Person("Anna", 25)  
person2 = Person("Béla", 30)
```

```
print(person1.greet())      # Eredmény: Szia, Anna vagyok és 25 éves.  
print(person2.greet())      # Eredmény: Szia, Béla vagyok és 30 éves.
```

- Az osztály **példányosításával** létrehozunk egy konkrét objektumot, amely saját attribútumokkal és metódusokkal rendelkezhet.
- Az **Person1** és **Person2** objektumok az **Person** osztály példányai, és mindegyik objektum saját névvel és korral rendelkezik.

Osztályok

Példa: egy Könyv osztály, amely tárolja a könyv címét és szerzőjét, és visszaadja a könyv információit

```
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author
        def book_info(self):
            return f"Title: {self.title}, Author:{self.author}"
book1 = Book("Egri csillagok", "Gárdonyi Géza")
book2 = Book("A Pál utcai fiúk", "Molnár Ferenc")
print(book1.book_info())
print(book2.book_info())
```

Kimenet:

Cím: Egri csillagok, Szerző: Gárdonyi Géza

Cím: A Pál utcai fiúk, Szerző: Molnár Ferenc

Összefoglalva az osztályok és objektumok előnyeit:

- **Modularitás:** Az osztályok és objektumok segítségével a program logikusan felépíthető, és a kódot modulokba szervezhetjük.
- **Kód újrahasznosíthatósága:** Egy egyszer megírt osztály több helyen is használható, így csökkentve az ismételt kódolást.
- **Egyszerűbb karbantartás:** Az osztályokon belül történő módosítások könnyen végrehajthatók anélkül, hogy az egész programot át kellene írni.

Konstruktorok és destruktorok

- **Konstruktorok:** Az `__init__()` metódus szerepe az objektum inicializálásában.
- **Destruktorok:** Az `__del__()` metódus használata és a Python memóriakezelése (destruktor ritka használata).

Kód példa:

```
class Dog:
    def __init__(self, name):
        self.name = name
    def __del__(self):
        print(f"{self.name} megszűnik")
```

Konstruktorok – Az objektumok inicializálása

- Speciális metódus.
- Automatikusan lefut egy új objektum létrehozásakor.
- Az objektum kezdőállapotának meghatározására szolgál: beállítja az objektumhoz tartozó attribútumokat (változókat).
- A konstruktor neve mindig `__init__`, amelyet az osztály definíciója során hozunk létre.
- Az `__init__` metódus paramétereit az objektum létrehozásakor adhatjuk át.
- A `self` paraméter mindig az aktuális objektumra utal.

Példa

```
class Student:
    def __init__(self, name, age):
        self.name = name    # A diák neve
        self.age = age      # A diák kora
    def introduce(self):
        return f„Szia, {self.name} vagyok és {self.age} éves.”
```

Példa: egy **Student** osztály, amely minden diáknak tárolja a nevét és a korát. A konstruktor beállítja ezeket az értékeket, amikor létrehozuk az objektumot.

Ebben az osztályban az **__init__** **konstruktor** elfogadja a diák nevét és korát, és ezeket az adatokat az objektum attribútumaiként tárolja. Amikor egy új diákot hozunk létre, a konstruktor automatikusan lefut.

Objektum létrehozása:

```
student1 = Student("Ádám", 20)
student2 = Student("Éva", 19)
print(student1.introduce())      # Eredmény: Szia, Ádám vagyok és 20 éves..
print(student2.introduce())      # Eredmény: Szia, Éva vagyok és 19 éves.
```

A konstruktor jellemzői:

1. **Self paraméter:** A self paraméter a konstruktorban és minden más osztálymetódusban mindig az aktuális objektumra utal, amit éppen létrehozunk. Ezen keresztül férünk hozzá az objektum attribútumaihoz és metódusaihoz.
2. **Paraméterek átadása:** Az `__init__` metódus további paramétereket is elfogadhat, amelyeket az objektum létrehozásakor adunk át, és ezekkel állítjuk be az objektum attribútumait.
3. **Attribútumok inicializálása:** A konstruktor fő feladata az, hogy az objektum attribútumait inicializálja, azaz beállítsa a kezdeti értékeket.

Destruktorok – Az objektumok megszüntetése

A **destruktor** metódus egy objektum megszüntetését és a memória felszabadítását eredményezi. A destruktor neve **`__del__`**.

A destruktor akkor hasznos, ha valamilyen erőforrást (fájlokat, hálózati kapcsolatokat) kell felszabadítani vagy bezárni, amikor egy objektum már nincs használatban.

Példa destruktorra: egy Fájl osztály fájlkezelésre

```
class File:
    def __init__(self, filename):
        self.filename = filename
        self.file = open(filename, 'w')
        print(f„A(z){self.filename} fájl megnyitva írásra”)
    def write(self, tex):
        self.file.write(text)
    def __del__(self):
        self.file.close()
        print(f„A(z) {self.filename} fájl lezárva.”)
```

Itt a `__del__` destruktorban beállítottuk, hogy amikor a fájl objektum megszűnik (például a program végeztével), a fájl automatikusan lezárásra kerüljön.

Objektum létrehozása és megszüntetése:

```
file = File("example.txt")  
file.write(„Ez egy példa szöveg.\n“)
```

Amikor a program befejeződik, a fájl objektum megszűnik, és a destruktor automatikusan bezárja a fájlt.

A destruktor jellemzői:

1. **Erőforráskezelés:** Általában az erőforrások felszabadítására vagy tisztítási feladatok elvégzésére szolgálnak.
2. **Automatikus lefutás:** akkor, amikor az objektumot a Python memóriakezelője megsemmisíti, általában akkor, amikor már nincs több hivatkozás az adott objektumra.
3. **Korlátozott használat:** ritkán kell explicit módon használni, mivel a Python automatikus szemétgyűjtési mechanizmusa (garbage collection) magától kezeli az objektumok törlését.

Példa konstruktorokra és destruktorokra - BankAccount osztály:

```
class BankAccount:
    def __init__(self, owner, initial_balance):
        self.owner = owner
        self.balance = initial_balance
        print(f"{self.owner} számlája megnyitva {self.balance} egyenleggel.")
    def deposit(self, amount):
        self.balance += amount
        print(f"{amount} befizetve. Új egyenleg: {self.balance}.")
    def withdraw(self, amount):
        if amount > self.balance:
            print(„Nincs elég fedezet.”)
        else: self.balance -= amount
            print(f"{amount} kivéve. Új egyenleg: {self.balance}.")
```

Példa konstruktorokra és destruktorokra - BankAccount osztály:

```
def __del__(self):  
    print(f"{self.owner} számlája lezárva.")  
account = BankAccount("Anna", 100000)  
# Eredmény: Anna számlája megnyitva 100000 egyenleggel.  
  
account.deposit(30000)      # Eredmény: 30000 befizetve. Új egyenleg: 130000.  
account.withdraw(5000)     # Eredmény: 5000 kivéve. Új egyenleg: 125000.  
  
# Amikor a program véget ér, a destruktor lefut: „Anna számlája lezárva.”
```

Összefoglalva a konstruktorok és destruktorok közötti különbségeket:

1. Konstruktor:

- Az objektum létrehozásakor fut le.
- Az objektum attribútumainak inicializálására szolgál.
- Pythonban a neve mindig `__init__`.

2. Destruktor:

- Az objektum megszűnésekor fut le.
- Az erőforrások felszabadítására vagy más tisztítási feladatok elvégzésére szolgál.
- Pythonban a neve mindig `__del__`.

Osztályattribútumok és metódusok

Attribútumok:

Objektum szintű attribútumok - osztályszintű attribútumok

- **Metódusok:**

Objektumspecifikus metódusok - osztályszintű metódusok (statikus metódusok)

- **Kód példa:**

```
class Circle:  
    pi = 3.1416  
    def __init__(self, radius):  
        self.radius = radius  
    def area(self):  
        return Circle.pi * (self.radius ** 2)
```

Osztályattribútumok és metódusok

Az osztályoknak és objektumoknak lehetnek saját attribútumaik és metódusaik.

Az attribútumok az objektumok állapotát (adattagjait), a metódusok pedig a viselkedésüket (funkcióikat) írják le.

Osztályattribútumok:

- Azok a változók, amelyek egy osztályban vannak definiálva, és az osztály minden példánya (objektuma) közösen osztozik rajtuk.
- Ezek globális értékek az adott osztályra nézve, tehát nem függnek az objektumoktól.
- Az osztályattribútumokat közvetlenül az osztályon belül definiáljuk, nem pedig az `__init__` konstruktorban.

Példa osztályattribútumra: egy **Circle** osztály, amely minden kör számára ugyanazt a **pi** értéket használja.

A **radius** viszont egy **objektum attribútum**, amely minden egyes kör objektumnál különböző lehet.

Objektum létrehozása és az attribútumok használata:

```
class Circle:  
    pi = 3.1416 # Osztályattribútum  
    def __init__(self, radius):  
        self.radius = radius # Objektum attribútum  
    def area(self):  
        return Circle.pi * (self.radius ** 2)
```

```
circle1 = Circle(5)  
circle2 = Circle(10)  
print(f" Az első kör területe: {circle1.area()}")  
# Eredmény: 78.54  
print(f" A második kör területe: {circle2.area()}")  
#Eredmény: 314.16
```

Mindkét kör esetében a **pi** értéke ugyanaz, de a **sugár** értéke eltérő az egyes objektumokban, ami különböző területeket eredményez.

Objektum attribútumok

- Azok a változók, amelyek minden egyes objektumhoz egyedileg tartoznak.
- Az objektum állapotát írják le.
- Minden objektumnak saját, független attribútuma van ezekből az értékekből.
- Példa: a sugár objektum attribútum, minden kör objektumnál egyedi értéke van.
- Példa: az Autó osztályban a márka és a típus objektum attribútumok:

```
class Car:
    def __init__(self, brand, model):
        self.brand = brand        # Objektum attribútum
        self.model = model        # Objektum attribútum
    def introduce(self):
        return f'Ez egy{self.brand} {self.model}.'

car1 = Car("Toyota", "Corolla")
car2 = Car("Opel", "Astra")
print(car1.introduce())          # Eredmény: Ez egy Toyota Corolla.
print(car2.introduce())          # Eredmény: Ez egy Opel Astra.
```

Osztálymetódusok és objektummetódusok

Az objektummetódusok az egyes objektumokra vonatkoznak, míg az osztálymetódusok magára az osztályra és annak összes példányára vonatkoznak.

Az objektummetódusok az osztályban definiált funkciók, amelyek egy adott objektum attribútumaihoz férnek hozzá, és az objektum attribútumainak kezelésére és műveletek végrehajtására használjuk.

```
class Rectangle:  
    def __init__(self, length, width):  
        self.length = length  
        self.width = width  
    def area(self):  
        return self.length * self.width  
    def perimeter(self):  
        return 2 * (self.length + self.width)
```


Osztálymetódusok és objektummetódusok

A terület és a kerület objektum-metódusok, a téglalap adott példányára vonatkozó számításokat végeznek.

Az objektummetódusok a self segítségével férnek hozzá az objektum attribútumaihoz.

```
rectangle1 = Rectangle(2, 3)
print(f" Az első téglalap területe: {rectangle1.area()}")
# Eredmény: 6
print(f" Az első téglalap kerülete{rectangle1.perimeter()}")
# Eredmény: 10
```

```
class Person:
    count = 0                # Osztályattribútum
    def __init__(self, name):
        self.name = name
        Person.count += 1    # Nő az osztályattribútum
    @classmethod
    def display_count(cls):
        return f"A személyek száma: {cls.count}"
person1 = Person("Anna")
person2 = Person("Béla")
print(Person.display_count())
# Eredmény: A személyek száma: 2
```

Az **osztálymetódusok** közvetlenül az osztályra vonatkoznak, nem az osztály példányaira. Ezeket a **@classmethod** dekorátorral jelöljük. Az első paraméterük a **cls**, amely magára az osztályra mutat, nem egy konkrét objektumra.

Példa osztálymetódusra: egy Person osztály, amelyben nyomon követjük, hogy hány személyt „hoztunk létre”.

Itt a létszám egy osztály-attribútum, amely minden személy létrehozásakor növekszik. A **display_count** osztálymetódus segítségével az osztály összes példányára vonatkozó információkat jeleníthetünk meg. Az osztálymetódusokat **közvetlenül az osztályon hívjuk meg**, nem egy konkrét objektumon.

Statikus metódusok

A **statikus metódusok** olyan metódusok, amelyek sem az osztályhoz, sem az objektumhoz nem kapcsolódnak közvetlenül. Ezeket a **@staticmethod** dekorátorral jelöljük, és nincs **szükségük sem self, sem cls paraméterre.**

Általában olyan funkciók végrehajtására használjuk őket, amelyek nem függenek sem az osztály állapotától, sem az objektum attribútumaitól.

Példa statikus metódusra:

```
class Math:
    @staticmethod
    def sum(a, b):
        return a + b

print(Math.sum(5, 10))    # Eredmény: 15
```

A statikus metódusokat közvetlenül az osztályon keresztül hívjuk meg, anélkül, hogy létrehoznánk egy objektumot.

Összefoglalás:

- **Osztályattribútumok:** Az osztályhoz tartozó globális változók, amelyek az osztály minden példányára nézve közősek.
- **Objektum attribútumok:** Az egyes objektumokhoz tartozó egyedi változók, amelyek az objektum állapotát írják le.
- **Objektummetódusok:** Az objektumokra vonatkozó funkciók, amelyek hozzáférnek az adott objektum attribútumaihoz.
- **Osztálymetódusok:** Az osztályra vonatkozó funkciók, amelyek az osztály összes példányára vonatkozó adatokat kezelik.
- **Statikus metódusok:** Független funkciók, amelyek nem függnnek sem az osztály, sem az objektum állapotától.

Az öröklődés alapjai

```
class Animal:
    def __init__(self, name):
        self.name = name
    def speak(self):
        pass

class Dog(Animal):
    def speak(self):
        return f"{self.name} mondja Woof-woof!"
```

- **Öröklődés fogalma:** Egy osztály hogyan örökölheti egy másik osztály attribútumait és metódusait.
- **Használat Pythonban:**
 - Szülőosztály és gyermekosztály kapcsolata.
- **Kód példa:**

Az öröklődés mechanizmusa:

- Az öröklődés lehetővé teszi egy új osztály létrehozását egy már meglévő osztály (szülőosztály) alapján.
- Az új osztály (gyermekosztály) automatikusan örökli a szülőosztály összes attribútumát és metódusát, és ezeket tovább bővítheti vagy módosíthatja.
- Az öröklődés elősegíti az újrahasznosítást, mivel a gyermekosztály nem igényli, hogy újra definiáljuk a szülőosztály már meglévő funkcióit.

Szülőosztály és gyermekosztály

- **Szülőosztály (Parent Class):** amelyből más osztályok örökölhetnek.
- **Gyermeosztály (Child Class):** amely öröklí a szülőosztály attribútumait és metódusait, és ezeket szükség esetén felüldefiniálhatja vagy kiegészítheti.
- **Példa öröklődésre:** egy Állat nevű szülőosztályunk, egy általános speak() metódussal.

```
class Animal:  
    def __init__(self, name):  
        self.name = name  
    def speak(self):  
        return f"{self.name} valamilyen hangot ad ki."
```

Szüülőosztály és gyermekosztály

```
class Dog(Animal):  
    def speak(self):  
        return f"{self.name} ugat!"  
  
class Cat(Animal):  
    def speak(self):  
        return f"{self.name} nyávog!"
```

Ebből **Kutya** és **Macska** osztályokat hozhatunk létre, amelyek felül-definiálják a `speak()` metódust a saját állathangjukkal.

```
dog1 = Dog("Buxsi")  
cat1 = Cat("Cirmi")  
print(dog1.speak())           # Result: Buxsi ugat!  
print(cat1.speak())          # Result: Cirmi nyávog!
```


Az öröklődés előnyei:

1. **Kód újrahasznosítása:** Az örökléssel lehetővé válik, hogy a gyermekosztály újra felhasználja a szülőosztály már meglévő kódját. Így a már megírt metódusokat nem kell újra és újra definiálni.
2. **Könnyű bővítés:** A gyermekosztályok könnyen hozzáadhatnak új tulajdonságokat vagy módosíthatják a szülőosztály viselkedését anélkül, hogy megváltoztatnánk a szülőosztály kódját.
3. **Moduláris tervezés:** Az osztályokat hierarchiában szervezhetjük, ahol az általános funkciókat a szülőosztály, míg a specifikus funkciókat a gyermekosztály tartalmazza.

A gyermekosztály metódusainak bővítése:

Ha szeretnénk, hogy a gyermekosztály egy metódusa először lefuttassa a szülőosztály metódusát is, a **super()** függvényt használhatjuk.

```
class Vehicle:
    def __init__(self, type):
        self.type = type
    def display(self):
        return f"A jármű típusa: {self.type}"

class Car(Vehicle):
    def __init__(self, type, brand):
        super().__init__(type)    # Szülőosztály konstruktorának meghívása
        self.brand = brand
    def display(self):
        return f"{super().display()}, Márka: {self.brand}"

car1 = Car("Személyautó", "Opel")
print(car1.display())  Eredmény: A jármű típusa: Személyautó, Márka: Opel
```

Példa a `super()` használatára: egy `Vehicle` osztály, egy `display()` metódussal, ami kiírja a jármű típusát. A gyermekosztályok tovább bővíthetik ezt a metódust.

Az **`Car`** gyermekosztály örökli a **`display()`** metódust, de szeretnénk, hogy a márkát is megjelenítse. Ehhez a **`super()`** segítségével először lefuttatjuk a szülőosztály metódusát, majd hozzáadjuk a saját kódunkat.

Több gyermekosztály egy szülőosztállyal

```
class Animal:
    def __init__(self, name):
        self.name = name
    def speak(self):
        return f"{self.name} valamilyen hangot ad ki."
class Dog(Animal):
    def speak(self):
        return f"{self.name} ugat!"
class Bird(Animal):
    def speak(self):
        return f"{self.name} csiripel!"
dog = Dog("Buxsi")
bird = Bird("Csőri")
print(dog.speak())    # Eredmény: Buxsi ugat!
print(bird.speak())   # Eredmény: Csőri csiripel!
```

Minden gyermekosztály megörökli a szülőosztály tulajdonságait, de egyedi metódusokat és attribútumokat is tartalmazhat.

A gyermekosztályok a szülőosztályból öröklik a név attribútumot, de a beszél() metódust felülírják, hogy a megfelelő állathangot adja vissza.

Felüldefiniálás (Override)

Az öröklés során a gyermekosztály felülírhatja a szülőosztály metódusait. A gyermekosztályban definiálhatjuk ugyanazt a metódust, mint ami a szülő-osztályban van, de más viselkedést biztosíthatunk számára.

A Diák osztályt örökli a Személy osztályt, de felüldefiniálja a bemutatkozás() metódust, hogy hozzá-adja a diák osztályát is.

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def introduce(self):
        return f"Szia, {self.name} vagyok, {self.age} éves."
class Student(Person):
    def __init__(self, name, age, grade):
        super().__init__(name, age)
        self.grade = grade
    def introduce(self):
        return f"Szia, {self.name} vagyok, {self.age} éves, és a(z) {self.grade}. osztályba járok."
student1 = Student("Anna", 15, 9)
print(student1.introduce()) # Eredmény: Szia, Éva vagyok, 15 éves, és a(z) 9. osztályba járok.
```

Összefoglalás:

- **Szülőosztály:** Az alap osztály, amelyből más osztályok örökölhetnek tulajdonságokat és metódusokat.
- **Gyermeosztály:** Az a specifikus osztály, amely örökli a szülőosztály tulajdonságait és metódusait, és tovább bővítheti azokat.
- **Felüldefiniálás:** A gyermeosztályban a szülőosztály metódusainak újradefiniálása.
- **super() függvény:** Lehetővé teszi, hogy a gyermeosztály hozzáférjen a szülőosztály metódusaihoz és attribútumaihoz.

1. Példa

Folytatás következő oldalon->

Python ELTE IK

```
class CreditCard:
```

```
    def __init__(self, name, number, limit = 8000, bank = "Python  
Bank"):
```

```
        self.name = name
```

```
        self.number = number
```

```
        self.bank = bank
```

```
        self.balance = 0
```

```
        self.limit = limit
```

```
    def charge(self, amount):
```

```
        #not int or not not float
```

```
        if not (isinstance(amount, int) or isinstance(amount, float)) or  
(amount <= 0):
```

```
            print("Charge Denied")
```

```
        else:
```

```
            self.balance += amount
```

1. Példa

```
def pay(self, amount):
    if not (isinstance(amount, int) or isinstance(amount,
float)) or (amount <= 0):
        print("Charge Denied")
    else:
        self.balance -= amount
def __str__(self):
    info = "Name: " + self.name + "\n"
    info += "Number: " + "XXXX" + self.number[4:] + "\n"
    info += "Bank: " + self.bank + "\n"
    info += "Balance: " + str(self.balance)
    return info
def __repr__(self):
    return str(self)
card = CreditCard("Kenny", "12345678")
card.charge(10000)
print(card)
```

2. Példa

Publikus lista

[]

0

```
# Verem osztály és az egységbezárás elve
print('Publikus lista')
class Stack:
    def __init__(self):
        self.stack_list = []
stack_object = Stack()
print(stack_object.stack_list)
print(len(stack_object.stack_list))
```


2. Példa

Privát hozzáférésű lista

A privát verem hossza: 3

```
print(len(stack_object.__stack_list))
```

#AttributeError

AttributeError: 'Stack' object has no attribute '__stack_list'

```
print('Privát hozzáférésű lista')
```

```
class Stack:
```

```
    def __init__(self):
```

```
        self.__stack_list = []
```

```
    def push(self, val):
```

```
        self.__stack_list.append(val)
```

```
    def pop(self):
```

```
        val = self.__stack_list[-1]
```

```
        del self.__stack_list[-1]
```

```
        return val
```

```
    def leng(self):          #a verem hosszának lekérdezése
```

```
        val = len(self.__stack_list)
```

```
        return(val)
```

```
stack_object = Stack()
```

```
stack_object.push(3)
```

```
print('A privát verem hossza:',stack_object.leng())
```

```
print(len(stack_object.__stack_list))    #AttributeError
```

3. Példa

Instance variables: dictionary-ben tárolva

{'egy': 1}

{'egy': 2, 'kettő': 3}

{'egy': 4, 'három': 5}

Folytatás következő oldalon->

Python ELTE IK

```
print('Instance variables: dictionary-ben tárolva')  
class PéldaOsztály:      #nem védett változókkal  
    def __init__(self, val = 1):  
        self.egy = val  
    def set_kettő(self, val):  
        self.kettő = val  
  
példa1 = PéldaOsztály()  
példa2 = PéldaOsztály(2)  
példa2.set_kettő(3)  
példa3 = PéldaOsztály(4)  
példa3.három = 5  
print(példa1.__dict__)  
print(példa2.__dict__)  
print(példa3.__dict__)
```

3. Példa

Instance variables: dictionary-ben tárolva

```
{'_PéldaOsztály__egy': 1}  
{'_PéldaOsztály__egy': 2,  
'_PéldaOsztály__kettő': 3}  
{'_PéldaOsztály__egy': 4,  
'__három': 5}
```

```
print('Instance variables: dictionary-ben tárolva')  
  
class PéldaOsztály:      #védett változókkal  
    def __init__(self, val = 1):  
        self.__egy = val  
    def set_kettő(self, val):  
        self.__kettő = val  
  
példa1 = PéldaOsztály()  
példa2 = PéldaOsztály(2)  
példa2.set_kettő(3)  
példa3 = PéldaOsztály(4)  
példa3.__három = 5  
print(példa1.__dict__)  
print(példa2.__dict__)  
print(példa3.__dict__)
```

Köszönöm a figyelmet!

ELTE IK

Python kurzus – 6. előadás

Tárgyfelelős: Dr. Tejfel Máté

Előadó: Dr. Király Roland

ELTE IK



6. Előadás tematikája OOP II.

1. Ismétlés
2. Polimorfizmus, absztrakt osztályok, interfészek
3. Többszörös öröklődés és MRO (Method Resolution Order)
4. Dekorátor függvények
5. Dekorátorok és speciális metódusok (pl. `__str__`, `__repr__`)
6. Indexelők
7. Tervezési minták (Design Patterns)
8. LLM lokális telepítése és használata
9. API elérése programból

A prezentációhoz tartozó forráskódok

- [PDFs\Python OOP alapok.pdf](#)
- [PDFs\abs.pdf](#)
- [PDFs\accesdecorator.pdf](#)
- [PDFs\accestester.pdf](#)
- [PDFs\adapter.pdf](#)
- [PDFs\builder.pdf](#)
- [PDFs\decocrpattern.pdf](#)
- [PDFs\decorator.pdf](#)
- [PDFs\factory.pdf](#)
- [PDFs\indexelo.pdf](#)
- [PDFs\interface.pdf](#)
- [PDFs\lencqlddecorators.pdf](#)
- [PDFs\MRO.pdf](#)
- [PDFs\prop.pdf](#)
- [PDFs\Singleton2.pdf](#)
- [oop alapok osszefoglalo.py](#)



```
class FnetLength:
    varians = 0
    def __init__(self):
        return self.varians:gcmlkes)

def oneion():
    enenomers = v1
    def selfl:
        """kenepond:cost:iskhere":
        self.conit>ngx\python> onces[x>""
        self.setf.sequenoess()

polimorfenifal = |
return fornton)
```

```
class Rotmetot6boagoh:
    def sov(f):
        valname = rootledvapexf(maan)
        return xwp

    else:
        def ident(s):
            moem = python.&int&scnpefdees~1)
            print sCemS.EORWUMCEGGWS1&
            return eop

Graohsust = "tanP"
def _get6asch():
    print(fae1&pbist1npexssalirew&#)

    return &ophsito)
```

Python OOP II.

Haladó koncepciók és szoftver-architektúrák.

POLIMORFIZMUS | TÖBBSZÖRÖS ÖRÖKLŐDÉS | DUNDER METÓDUSOK | TERVEZÉSI MINTÁK

2. *Polimorfizmus, absztrakt osztályok, interfészek*

Polimorfizmus: Az azonos metódusok különböző viselkedése különböző osztályokban.

Absztrakt osztályok: Az abc modul és az absztrakt metódusok használata.

Interfészek: Hogyan tervezhető elv alapján egy keret, amelyet az osztályok megvalósítanak.

A **polimorfizmus, az absztrakt osztályok és interfészek** az OOP fejlettebb funkciói, amelyek segítségével rugalmasabb és modulárisabb programokat készíthetünk.

Absztrakt osztályok

- **Absztrakt osztály** nem példányosítható közvetlenül, más osztályok származtatására szolgál
- Ha közös funkcionalitást szeretnénk megosztani a származtatott osztályok között
- Egyfajta sablon, amely meghatározza, hogy milyen metódusoknak kell létezniük a gyermekosztályokban, de nem biztosít ezekhez konkrét implementációt
- Arra kényszerítik a gyermekosztályokat, hogy bizonyos metódusokat implementáljanak
- Az absztrakt osztályok tartalmazhatnak normál metódusokat és argumentumokat is
- Pythonban az absztrakt osztályokhoz az **abc (Abstract Base Classes)** modult használjuk
- Az absztrakt metódusokat az **@abstractmethod dekorátorral** jelöljük

Polimorfizmu

S

A **polimorfizmus**: arra utal, hogy azonos nevű függvények különböző típusú objektumokon más-más viselkedést mutatnak.

Ez lehetővé teszi, hogy ugyanazt a függvényt vagy metódust több különböző objektumra alkalmazzuk, és az objektum típusa dönti el, hogyan reagáljon rá. Ugyanaz a művelet eltérő eredményt adhat különböző típusú objektumoknál.

```
class Állat:
    def beszél(self):
        pass
class Kutya(Állat):
    def beszél(self):
        return "A kutya ugat."
class Macska(Állat):
    def beszél(self):
        return "A macska nyávog."

állatok = [Kutya(), Macska()]
for állat in állatok:
    print(állat.beszél())

# Polimorfizmus révén más-más viselkedést
kapunk

A kutya ugat.

A macska nyávog.
```

Absztrakt osztályok

```
class B(ABC):  
    @abstractmethod  
    def add(self):  
        pass
```

```
b = B()
```

```
'''
```

TypeError: Can't instantiate abstract class B without
an implementation for abstract method 'add'

Interfészek

Az **interfész** hasonló az absztrakt osztályokhoz, de specifikusabb, mivel az interfészek **csak absztrakt metódusokat tartalmazhatnak** (attribútumokat, változókat nem).

Az interfészek segítségével meghatározhatjuk, hogy milyen funkciókat kell biztosítania egy osztálynak, de nem adunk hozzá konkrét megvalósítást.

Az interfészek lehetőséget adnak arra, hogy különböző osztályok közös funkciókat valósítsanak meg.

Az interfészeket olyan absztrakt osztályokként definiálhatjuk, amelyek csak absztrakt metódusokat tartalmaznak.

Interfész példája

Tegyük fel, hogy van egy **Mozgatható** interfészünk, amely biztosítja, hogy egy osztály képes legyen mozgatni egy objektumot, de nem definiáljuk, hogyan történik a mozgatás. Minden osztálynak, amely implementálja ezt az interfészt, meg kell valósítania a `mozog()` metódust.

Interfész létrehozása absztrakt osztállyal:

```
from abc import ABC, abstractmethod

class Mozgatható(ABC):
    @abstractmethod
    def mozog(self):
        pass

class Autó(Mozgatható):
    def mozog(self):
        return "Az autó gurul."

class Repülő(Mozgatható):
    def mozog(self):
        return "A repülő repül."

autó = Autó()
repülő = Repülő()
print(autó.mozog())
print(repülő.mozog())
```

Ezután bármely osztály, amely implementálja a Mozgatható interfészt, köteles lesz megvalósítani a mozog() metódust.

Az Autó és Repülő osztályok megvalósítják a Mozgatható interfészt, így biztosítják a mozog() metódust.

Az interfész garantálja, hogy bármelyik osztály, amely implementálja, biztosan tartalmazza ezt a metódust, de minden osztály másképp valósíthatja meg.

1. Polimorfizmus:

- A polimorfizmus azt jelenti, hogy ugyanazt a metódust különböző osztályokban eltérő módon valósíthatjuk meg. Ez rugalmasabbá teszi a kódot, mivel ugyanazt a függvényt többféle típusú objektumra is alkalmazhatjuk.

2. Absztrakt osztályok:

- Az absztrakt osztályok olyan osztályok, amelyek nem hozhatók létre közvetlenül. Csak sablont adnak a gyermekosztályok számára, és kötelezik azokat bizonyos metódusok implementálására.

3. Interfészek:

- Az interfészek csak absztrakt metódusokat tartalmaznak, és a különböző osztályoknak implementálniuk kell ezeket a metódusokat. Pythonban az interfészek gyakran absztrakt osztályokkal vannak reprezentálva.

3. *Többszörös öröklődés és Method Resolution Order (MRO)*

- **Többszörös öröklődés:** Pythonban lehetőség van több osztály öröklésére
- Az **öröklődés** az objektumorientált programozás egyik legfontosabb eszköze, amely lehetővé teszi, hogy egy osztály átvegye egy másik osztály tulajdonságait és metódusait. Az egyszerű öröklődés mellett Pythonban létezik egy fejlettebb mechanizmus, a **többszörös öröklődés** is, amely során egy osztály több szülőosztálytól is örökölhét.
- **MRO (Method Resolution Order):** Hogyan határozza meg Python az öröklés során a metódusok feloldási sorrendjét

Kód példa többszörös öröklődésre és MRO-ra:

[mro.py](#)

MRO alapján melyik metódus hívódik meg?

3. Többszörös öröklődés

- **A többszörös öröklődés** azt jelenti, hogy egy gyermekosztály több szülőosztályból örökölhet attribútumokat és metódusokat. Ez különbözik az egyszerű öröklődéstől, ahol egy gyermekosztálynak csak egy szülőosztálya van.
- **Miért hasznos a többszörös öröklődés?**
 - **Rugalmasság:** A többszörös öröklődés lehetővé teszi, hogy egy osztály különböző viselkedést örököljön több szülőosztályból, így rugalmasabban kombinálhatóak a különböző funkciók.
 - **Kód újrahasznosítása:** A kód újrahasznosítása és modulárisabb felépítése lehetővé válik, ha egy osztály több szerepet (több szülőosztálytól származó viselkedést) is betölt.
 - **Több funkció integrálása:** Egy osztály egyszerre több különböző funkciót is megörökölhet, például egy autó lehet egyszerre „Jármű” és „Mozgatható tárgy”.

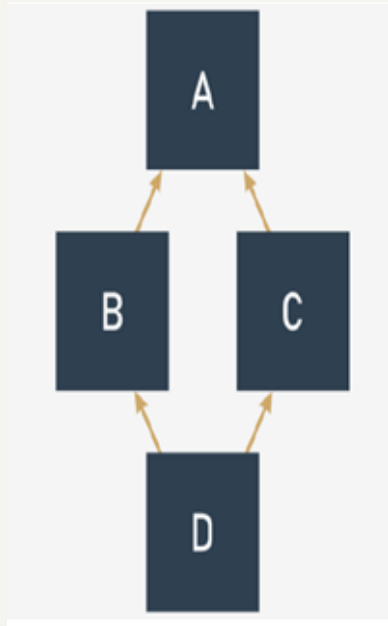
Konfliktusok a többszörös öröklődésnél

Többszörös öröklődés esetén előfordulhat, hogy ugyanaz a metódus több szülőosztályban is létezik. Ilyen esetben kérdésessé válik, hogy a gyermekosztály melyik szülőosztály metódusát örökölje. Amikor egy metódust hívunk egy gyermekosztályból, a Python az MRO alapján dönti el, hogy melyik szülőosztály metódusát kell használnia.

```
print(Osztálynév.__mro__)
```

Ez visszaadja, hogy milyen sorrendben keresi a Python a metódusokat az osztályokban.

Diamond probléma:



Konfliktusok a többszörös öröklődésnél

```
class A:
    def függvény(self):
        return "A-ból"
class B(A):
    def függvény(self):
        return "B-ből"
class C(A):
    def függvény(self):
        return "C-ből"
class D(B, C):
    pass
d = D()
```

`print(d.függvény())` # Melyik metódus fut le?

`print(D.__mro__)` # Method Resolution Order
lekérdezése

B-ből

```
(<class '__main__.D'>,
<class '__main__.B'>,
<class '__main__.C'>,
<class '__main__.A'>,
<class 'object'>)
```

Magyarázat:

- Az MRO szerint a Python először a D osztályban keresi a metódust, majd a B, aztán a C, végül pedig az A osztályban. Tehát a B osztály metódusa fut le.
- Ez a sorrend azért van így, mert a Python a balról jobbra történő keresést követi, és a B osztály van először a sorban.

A `super()` és az MRO

- A `super()` függvény segítségével a gyermekosztályban hozzáférhetünk a szülőosztály metódusaihoz anélkül, hogy közvetlenül meg kellene nevezni a szülőosztályt.
- Pythonban a `super()` is az MRO alapján működik, így a megfelelő szülőosztály metódusát hívja meg az MRO sorrend szerint.

```
class A:
    def függvény(self):
        return "A-ból"

class B(A):
    def függvény(self):
        return "B-ből " + super().függvény()

class C(A):
    def függvény(self):
        return "C-ből " + super().függvény()

class D(B, C):
    def függvény(self):
        return "D-ből " + super().függvény()

d = D()
print(d.függvény())
```

Magyarázat:

- A `super()` minden osztályban meghívja a szülőosztály metódusát az MRO sorrend szerint.
- A D osztályból indulva a B osztály metódusa fut le először, majd a C osztályé, végül az A osztályé.
- A `super()` az MRO alapján megy végig a szülőosztályok láncán.

Összefoglalás

- **Többszörös öröklődés:**
 - A többszörös öröklődés lehetővé teszi, hogy egy osztály több szülőosztályból örököljön metódusokat és attribútumokat. Ez rugalmasságot és újra felhasználhatóságot biztosít a programok számára, de metódus konfliktusokat is eredményezhet.
- **Method Resolution Order (MRO):**
 - Az MRO meghatározza, hogy Pythonban milyen sorrendben keres metódusokat, amikor több szülőosztályból öröklünk. A Python az MRO alapján oldja meg a több öröklődésből származó metódusok feloldási konfliktusait.
 - Az MRO sorrendje lekérdezhető az **Osztálynév.__mro__** kifejezéssel.
 - A `super()` függvény az MRO alapján működik, és az MRO sorrendjének megfelelően hívja meg a szülőosztály metódusait.

4. Dekorátorok és Speciális

metódusok

A dekorátorok és a speciális metódusok segítségével testreszabhatjuk és kibővíthetjük az osztályok, metódusok és függvények viselkedését. Bemutatjuk a dekorátorok működését, és megvizsgálunk speciális metódusokat (pl. `__str__`, `__repr__`, `__len__`), amelyek lehetővé teszik az objektumok egyedi viselkedésének meghatározását.

Dekorátorok: Hogyan működnek és mikor hasznosak: Funkcionális dekorátorok írása és használata Pythonban

Speciális metódusok: `__str__`, `__repr__` és egyéb dunder metódusok szerepe az osztályok viselkedésének testreszabásában

• [decorator.py](#)

```
class Személy:
    def __init__(self, név, kor):
        self.név = név
        self.kor = kor
    def __str__(self):
        return f"Person(név={self.név}, kor={self.kor})"
    def __repr__(self):
        return f"„Személy('{self.név}', {self.kor})"
```

Dekorátorok

A **dekorátorok** speciális funkciók, amelyek lehetővé teszik egy függvény vagy metódus viselkedésének módosítását anélkül, hogy meg kellene változtatnunk a függvény tényleges kódját. Pythonban a dekorátorok egyszerűen függvények, amelyek egy másik függvényt fogadnak paraméterként, és visszaadnak egy új függvényt, amelyik kiterjeszti vagy módosítja az eredeti függvény működését.

Dekorátorok működése: A dekorátorokat a **@ szimbólum** segítségével alkalmazzuk egy függvényre vagy metódusra.

```
def egyszerű_dekorátor(függvény):  
    def belső_függvény():  
        print("Dekorátor előtt.")  
        függvény()  
        print("Dekorátor után.")  
    return belső_függvény  
@egyszerű_dekorátor  
def köszönés():  
    print("Szia!")
```

köszönés()

Magyarázat:

- A köszönés() függvényt a @egyszerű_dekorátor dekorátorral díszítettük.
- A dekorátor hozzáadott egy plusz funkciót: kiír egy üzenetet a köszönés() függvény végrehajtása előtt és után.

Dekorátor előtt. Szia! Dekorátor után.

Metódusok dekorálása

```
def ellenőrzés_dekorátor(függvény):  
    def belső_függvény(self, *args, **kwargs):  
        if self.engedélyezett:  
            return függvény(self, *args, **kwargs)  
        else:  
            print("Hozzáférés megtagadva.")  
    return belső_függvény  
  
class Felhasználó:  
    def __init__(self, név, engedélyezett):  
        self.név = név  
        self.engedélyezett = engedélyezett  
    @ellenőrzés_dekorátor  
    def hozzáférés(self):  
        print(f'Hozzáférés engedélyezve: {self.név}')
```

```
felhasználó1 = Felhasználó("Anna", True)  
felhasználó2 = Felhasználó("Béla", False)  
  
felhasználó1.hozzáférés()  
felhasználó2.hozzáférés()
```

Pythonban a dekorátorokat metódusokra is alkalmazhatjuk. Ezáltal bővíthetjük egy osztály metódusainak működését anélkül, hogy azokat közvetlenül módosítanánk.

Magyarázat:

- A hozzáférés() metódust díszítettük az ellenőrzés_dekorátor dekorátorral.
- A dekorátor ellenőrzi, hogy a felhasználó engedélyezett-e a hozzáféréshez, és csak akkor engedi futni a hozzáférés() metódust, ha a felhasználónak van jogosultsága.

• `accesstester.py`

összefoglalóban!)

A **speciális metódusok** (más néven "**dunder**" metódusok, mert két (double) alulvonással kezdődnek és végződnek) lehetővé teszik, hogy a Python alapvető viselkedéseit személyre szabjuk az osztályainkban.

Ezek a metódusok határozzák meg, hogyan viselkednek az osztályaink bizonyos beépített funkciók (pl. `len()`, `str()`, összehasonlítási műveletek) során.

- **1. `__str__` és `__repr__`: Objektumok szöveges reprezentációja**

- A `__str__` és a `__repr__` speciális metódusok segítségével határozzuk meg, hogyan jelenjen meg az objektum szöveges formában. A `__str__` az objektum **felhasználói** reprezentációját adja vissza, míg a `__repr__` az objektum **fejlesztői célú**, hivatalosabb reprezentációját biztosítja.

- `__str__`: A felhasználónak szánt, könnyen olvasható szöveges reprezentáció. Akkor hívódik meg, ha egy objektumot a `print()` függvénnyel kiíratunk.
- `__repr__`: A fejlesztőnek szánt reprezentáció, amelyet általában a `repr()` függvénnyel vagy a konzolból való közvetlen kiírásakor használunk.

```
class Személy:
    def __init__(self, név, kor):
        self.név = név
        self.kor = kor

    def __str__(self):
        return f"Személy neve: {self.név}, kor: {self.kor}"

    def __repr__(self):
        return f"Személy(név='{self.név}', kor={self.kor})"

személy1 = Személy("Anna", 30)

print(személy1)
print(repr(személy1))
```

Személy neve: Anna, kor: 30

Személy(név='Anna', kor=30)

Magyarázat:

- a `__str__` metódus segítségével egy felhasználóbarát szöveges formát biztosítottunk
- a `__repr__` hivatalosabb, programozási környezetben használt formátumot ad vissza.

2. `__len__`: Hossz lekérdezése

A `__len__` speciális metódus segítségével meghatározhatjuk, hogy mi történjen, amikor egy objektumra alkalmazzuk a `len()` függvényt. Ez különösen hasznos lehet olyan osztályok esetén, amelyek belsőleg tárolnak olyan objektumot, amelynek a hosszát lekérdezhetjük.

```
class Csoport:
    def __init__(self, nevek):
        self.nevek = nevek

    def __len__(self):
        return len(self.nevek)

csoport = Csoport(["Anna", "Béla", "Cecil"])

print(len(csoport))
```

Magyarázat:

- A `__len__` metódust implementáltuk, hogy a csoportban lévő nevek számát adja vissza.
- A `len()` függvény a nevek lista hosszát adja meg.

3. `__eq__`, `__lt__`, `__le__`: Összehasonlító műveletek

A Pythonban az összehasonlító operátorok (pl. `==`, `<`, `<=`) testreszabhatók az `__eq__`, `__lt__`, `__le__` stb. speciális metódusokkal. Lehetővé teszik, hogy meghatározzuk, hogyan viselkedjenek az osztályaink összehasonlítás során.

```
class Személy:
```

```
    def __init__(self, név, kor):
```

```
        self.név = név
```

```
        self.kor = kor
```

```
    def __eq__(self, másik):
```

```
        return self.kor == másik.kor
```

```
    def __lt__(self, másik):
```

```
        return self.kor < másik.kor
```

```
személy1 = Személy("Anna", 30)
```

```
személy2 = Személy("Béla", 25)
```

```
print(személy1 == személy2)
```

```
print(személy1 < személy2)
```

Magyarázat:

- A `__eq__` metódus segítségével azt határoztuk meg, hogy két Személy objektum egyenlő-e a koruk alapján.
- A `__lt__` metódus meghatározza, hogy az egyik személy fiatalabb-e, mint a másik.

```
False
```

```
False
```

```
• lenegitdecorators.py
```

4. `__getitem__`, `__setitem__`: Indexelhető objektumok

A `__getitem__` és a `__setitem__` metódusok segítségével olyan osztályokat hozhatunk létre, amelyek hasonlóan viselkednek, mint a listák vagy szótárak. Ezek a metódusok lehetővé teszik, hogy indexek segítségével hozzáférjünk az osztályunk által tárolt adatokhoz, illetve módosítsuk azokat.

```
class Lista:
    def __init__(self, elemek):
        self.elemek = elemek

    def __getitem__(self, index):
        return self.elemek[index]

    def __setitem__(self, index, érték):
        self.elemek[index] = érték

lista = Lista([10, 20, 30])
print(lista[1])
lista[1] = 40
print(lista[1])
```

Magyarázat:

- A `__getitem__` metódus segítségével hozzáférhetünk az objektum elemeihez index alapján, hasonlóan a listákhoz.
- A `__setitem__` metódussal pedig módosíthatjuk az elemeket.

• [indexelo.py](#)

• [prop.py](#)

20
40

Osszetoglaló

S

Dekorátorok: Lehetővé teszik a függvények és metódusok működésének kibővítését anélkül, hogy módosítanánk azok belső kódját.

A dekorátorok egyszerűen alkalmazhatók a `@` szintaxis segítségével.

Speciális metódusok: Ezek a metódusok lehetővé teszik, hogy az osztályaink egyedi viselkedést biztosítsanak a beépített Python funkciók (pl. `len()`, `str()`, összehasonlító műveletek) számára.

Ezek használatával az osztályok testreszabottan viselkedhetnek a Python beépített funkcióival.

5. Tervezési minták (Design Patterns)

```
class Singleton:
    _instance = None

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super(Singleton,
cls).__new__(cls)
        return cls._instance
```

- **Singleton minta:** Egyetlen példány létrehozása egy osztályból.
- **Factory minta:** Objektumok dinamikus létrehozása.
- **Observer minta:** Egy esemény bekövetkezésekor értesítések küldése a regisztrált megfigyelőknek.

5. Tervezési minták (Design Patterns)

- A **tervezési minták** (design patterns) jól bevált, általános megoldások visszatérő szoftverfejlesztési problémákra. Ezek olyan sablonok, amelyek segítenek strukturálni a kódot, megoldani gyakori programozási feladatokat, és elősegítik a hatékonyabb, olvashatóbb és karbantarthatóbb szoftverek írását. A tervezési minták előnye, hogy elősegítik a kód újrafelhasználását, és következetes megközelítést biztosítanak a programozási problémák megoldására.
- A **tervezési minták három fő kategóriába sorolhatók:**
 - **Kreációs minták:** Olyan minták, amelyek az objektumok létrehozására fókuszálnak.
 - **Strukturális minták:** Olyan minták, amelyek az osztályok és objektumok közötti kapcsolatokra koncentrálnak.
 - **Viselkedési minták:** Olyan minták, amelyek az objektumok és osztályok közötti interakciók és kommunikáció kezelését segítik.

Kreációs tervezési minták

A **kreációs minták** célja, hogy az objektumok létrehozásának folyamatát szabályozzák és optimalizálják.

1. A Singleton minta: biztosítja, hogy egy osztályból csak egyetlen példány létezzen, és hogy ez a példány globálisan elérhető legyen. Ez akkor hasznos, amikor egyetlen objektumnak kell kezelnie valamilyen globális állapotot vagy erőforrást (például egy konfigurációs fájl, adatbázis-kapcsolat).

```
class Singleton:
    _instance = None
    def __new__(cls):
        if cls._instance is None:
            cls._instance =
super().__new__(cls)
        return cls._instance
singleton1 = Singleton()
singleton2 = Singleton()
print(singleton1 == singleton2)
```

Magyarázat:

- A `__new__()` metódus felelős a példány létrehozásáért. Itt ellenőrizzük, hogy már létezik-e az osztály példánya.
- Ha nem, akkor létrehozzuk, és elmentjük. Ha már létezik, ugyanazt az objektumot adjuk vissza.

2. *Factory minta*

A **Factory** minta egy általános megoldás arra, hogy egy osztály objektumokat hozzon létre, anélkül, hogy kifejezetten meghatároznánk az objektum pontos típusát.

Ezáltal az objektumok létrehozása elválík a konkrét implementációtól.

Magyarázat:

- A **Factory minta** lehetővé teszi, hogy anélkül hozzunk létre objektumokat, hogy konkrétan meg kellene adnunk az objektum típusát a kód minden részében.
- A Factory osztály dönt az objektum típusáról.

Az autó halad.

A motor száguld.

```
class Autó:
```

```
    def vezet(self):
```

```
        return "Az autó halad."
```

```
class Motor:
```

```
    def vezet(self):
```

```
        return "A motor száguld."
```

```
class JárműFactory:
```

```
    @staticmethod
```

```
    def jármű_létrehoz(típus):
```

```
        if típus == "autó":
```

```
            return Autó()
```

```
        elif típus == "motor":
```

```
            return Motor()
```

```
        else:
```

```
            raise ValueError("Ismeretlen járműtípus")
```

```
jármű1 = JárműFactory.jármű_létrehoz("autó")
```

```
jármű2 = JárműFactory.jármű_létrehoz("motor")
```

```
print(jármű1.vezet())
```

```
print(jármű2.vezet())
```

3. Builder minta

A **Builder** minta összetett objektumok lépésenkénti létrehozására szolgál. Olyan esetekben hasznos, amikor egy objektumot több lépésben kell felépíteni, és a konstrukció folyamata bonyolult.

Magyarázat:

A **Builder minta** lehetővé teszi, hogy egy összetett objektumot (ebben az esetben egy házat) lépésenként építsünk fel.

Alap: Betonalap,
Falak: Téglafalak,
Tető: Cseréptető

```
class Ház:
```

```
    def __init__(self):
```

```
        self.alap = None
```

```
        self.falak = None
```

```
        self.tető = None
```

```
    def __str__(self):
```

```
        return f"Alap: {self.alap}, Falak: {self.falak}, Tető: {self.tető}"
```

```
class HázÉpítő:
```

```
    def __init__(self):
```

```
        self.ház = Ház()
```

```
    def alap_épít(self):
```

```
        self.ház.alap = "Betonalap"
```

```
        return self
```

```
    def falak_épít(self):
```

```
        self.ház.falak = "Téglafalak"
```

```
        return self
```

```
    def tető_épít(self):
```

```
        self.ház.tető = "Cseréptető"
```

```
        return self
```

```
    def get_ház(self):
```

```
        return self.ház
```

```
házépítő = HázÉpítő()
```

```
ház = házépítő.alap_épít().falak_épít().tető_épít().get_ház()
```

```
print(ház)
```

[builder.py](#)

Strukturális tervezési minták

A **strukturális minták** az osztályok és objektumok közötti kapcsolatok kezelésére fókuszálnak, így megkönnyítik a nagyobb rendszerek felépítését és összekapcsolását.

1. Adapter minta

Az **Adapter** minta lehetővé teszi, hogy olyan osztályokat kapcsoljunk össze, amelyek különböző interfészekkel rendelkeznek. Az adapter átmenetet biztosít két nem kompatibilis osztály között.

Magyarázat:

Az **Adapter minta** segítségével egy amerikai dugasz európai rendszerben használható anélkül, hogy módosítanunk kellene az eredeti osztályt.

Amerikai dugasz csatlakoztatva.

```
class EurópaiDugasz:
    def csatlakoztat(self):
        return "Európai dugasz csatlakoztatva"

class AmerikaiDugasz:
    def plug_in(self):
        return "Amerikai dugasz csatlakoztatva."

class Adapter:
    def __init__(self, amerikai_dugasz):
        self.amerikai_dugasz = amerikai_dugasz
    def csatlakoztat(self):
        return self.amerikai_dugasz.plug_in()

amerikai_dugasz = AmerikaiDugasz()
adapter = Adapter(amerikai_dugasz)
print(adapter.csatlakoztat())
```

[adapter.py](#)

2. Decorator minta

A **Decorator minta** lehetővé teszi, hogy egy objektum funkcionalitását dinamikusan bővítsük anélkül, hogy módosítanánk az osztályt.

Ez a minta akkor hasznos, ha egy objektumhoz **új képességeket** szeretnénk hozzáadni.

Magyarázat:

A **Decorator minta** lehetővé teszi, hogy új funkciókat adjunk hozzá az eredeti Kávé osztályhoz anélkül, hogy azt módosítanánk.

Először tejet adunk a kávéhoz, majd cukrot.

650

```
class Kávé:
```

```
    def ár(self):
```

```
        return 500
```

```
class TejKávéDecorator:
```

```
    def __init__(self, kávé):
```

```
        self.kávé = kávé
```

```
    def ár(self):
```

```
        return self.kávé.ár() + 100
```

```
class CukorKávéDecorator:
```

```
    def __init__(self, kávé):
```

```
        self.kávé = kávé
```

```
    def ár(self):
```

```
        return self.kávé.ár() + 50
```

```
kávé = Kávé()
```

```
tej_kávé = TejKávéDecorator(kávé)
```

```
cukros_tej_kávé = CukorKávéDecorator(tej_kávé)
```

```
print(cukros_tej_kávé.ár())
```

[decocrpattern.py](#)

[accessdecorator.py](#)

Viselkedési Tervezési Minták

1. Observer minta: Az **Observer minta** lehetővé teszi, hogy egy objektum (az alany) több más objektumot (megfigyelők) értesítsen, amikor az állapota megváltozik. Ez hasznos olyan helyzetekben, ahol több objektumnak kell reagálnia egy adott eseményre vagy változásra.

A **viselkedési minták** az objektumok közötti kommunikációt és interakciót szabályozzák. Ezek a minták a rendszer különböző komponenseinek együttműködésére összpontosítanak.

Magyarázat:

Az **Observer minta** lehetővé teszi, hogy több megfigyelő értesítést kapjon, amikor az alany állapota megváltozik.

Observer minta

[observer.py](#)

```
class Alany:
    def __init__(self):
        self.megfigyelők = []
    def feliratkozás(self, megfigyelő):
        self.megfigyelők.append(megfigyelő)
    def értesítés(self):
        for megfigyelő in self.megfigyelők:
            megfigyelő.update()

class Megfigyelő:
    def update(self):
        print("Az alany állapota megváltozott.")

alany = Alany()
megfigyelő1 = Megfigyelő()
megfigyelő2 = Megfigyelő()
alany.feliratkozás(megfigyelő1)
alany.feliratkozás(megfigyelő2)
alany.értesítés()
```

OUTPUT: Az alany állapota megváltozott.

Az alany állapota megváltozott.

Összefoglalás

Kreációs minták:

Az objektumok létrehozásának módját szabályozzák.

Példák: Singleton, Factory, Builder.

Strukturális minták:

Az objektumok és osztályok közötti kapcsolatokat kezelik.

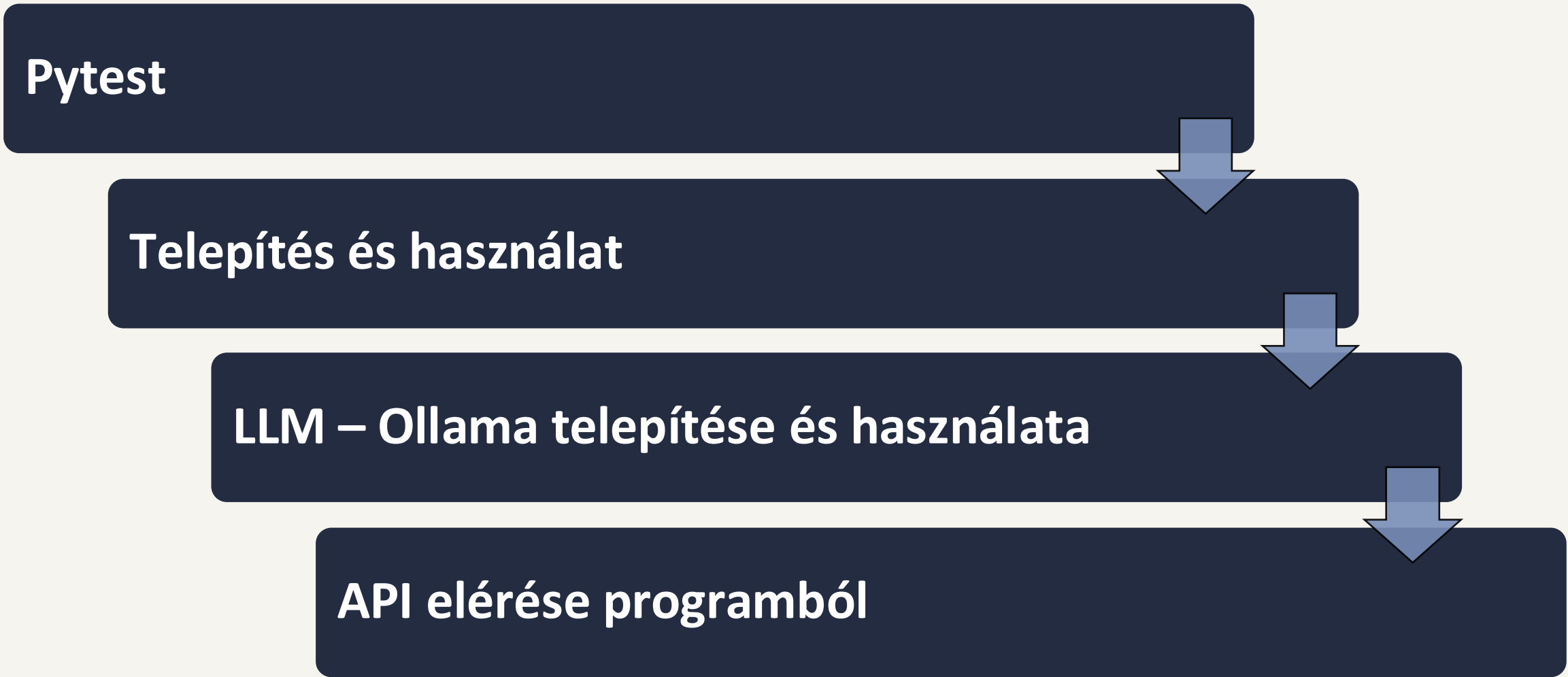
Példák: Adapter, Decorator.

Viselkedési minták:

Az objektumok közötti kommunikációra és interakcióra összpontosítanak.

Példák: Observer.

Pytest



Telepítés és használat

LLM – Ollama telepítése és használata

API elérése programból

Köszönöm a
figyelmet!

ELTE IK



Python kurzus – 7. előadás

Tárgyfelelős: Dr. Tejfel Máté

Előadó: Dr. Király Roland

ELTE IK



Tematika

1. Validátorok
2. JSON fájlok
3. RESTful API és HTTP protokoll
4. FastAPI: route-ok, request és response kezelés
5. JSON válaszok készítése
6. Egyszerű API létrehozása, telepítése
7. FastAPI és adatbázis integráció (SQLAlchemy bevezetése)

A prezentációhoz tartozó forráskódok

[data.py\(pdf állomány megnyitása\)](#)

[fastapi.py\(pdf állomány megnyitása\)](#)

[fastapitest.py\(pdf állomány megnyitása\)](#)

[flsjson.py\(pdf állomány megnyitása\)](#)

[main.py \(pdf állomány megnyitása\)](#)

[mainapi.py\(pdf állomány megnyitása\)](#)

[maintest.py\(pdf állomány megnyitása\)](#)

[test.py\(pdf állomány megnyitása\)](#)

[validation.py\(pdf állomány megnyitása\)](#)

[programok.zip\(tartalom letöltése\)](#)

[api1.py\(pdf állomány megnyitása\)](#)

[database.py\(pdf állomány megnyitása\)](#)

[fastapitest.py\(pdf állomány megnyitása\)](#)

[files.py\(pdf állomány megnyitása\)](#)

[filewriter.py\(pdf állomány megnyitása\)](#)

[flsjson.py\(pdf állomány megnyitása\)](#)

[main.py\(pdf állomány megnyitása\)](#)

[main2.py\(pdf állomány megnyitása\)](#)

[models.py\(pdf állomány megnyitása\)](#)

[dbtest.zip\(tartalom letöltése\)](#)

Validátorok

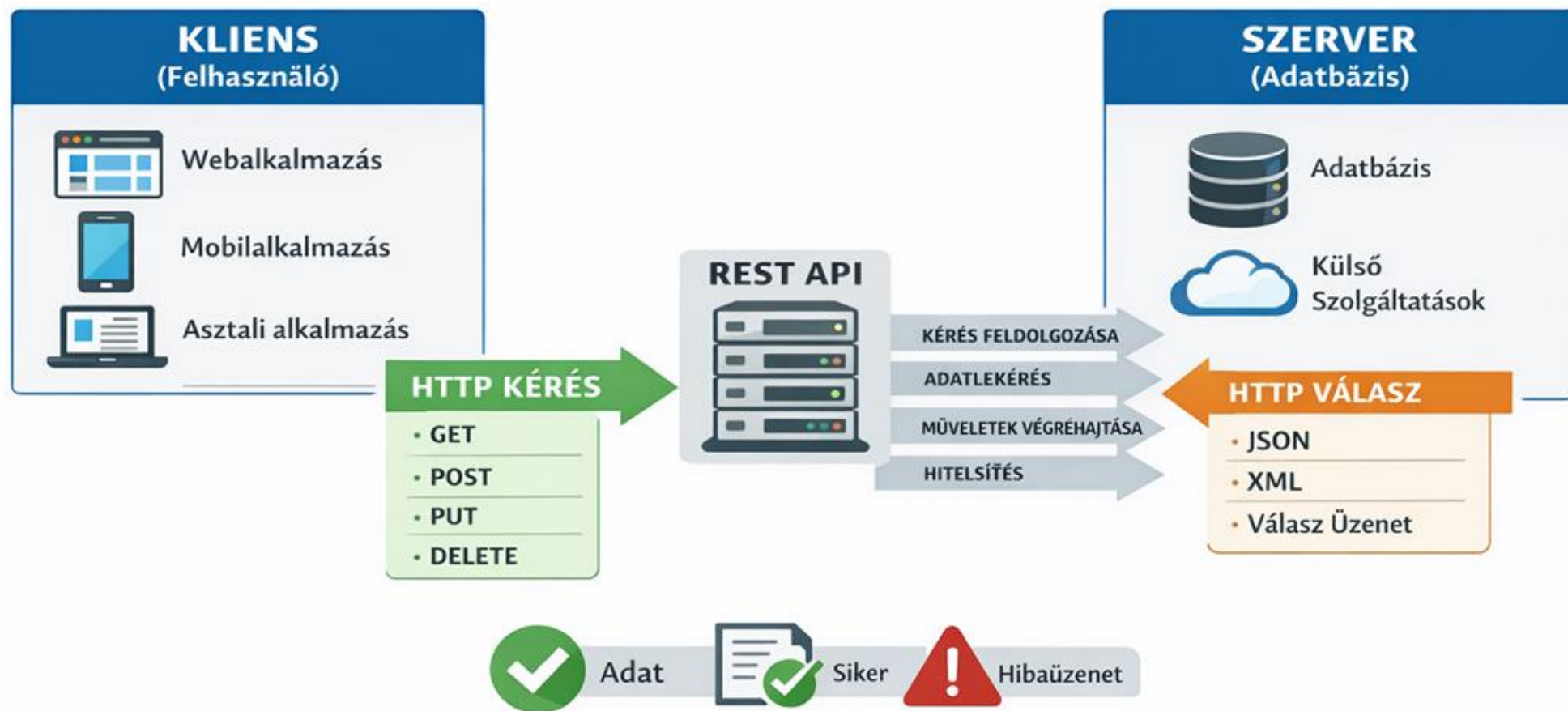
```
from pydantic import EmailStr, BaseModel  
from pydantic import field_validator  
from pydantic import model_validator
```

```
from typing import Optional
```

A validator deprecated, helyette különböző
validátorok jelennek meg. (+ 'pydantic[email']')

```
message = {  
    1: {  
        {"name": "Bob"},  
        {"level": 1},  
        {"profession": None}  
    }  
}  
  
#{"name": "Bob"}, vagy a {"level": 1} - Dict(str,Any)
```

REST API MŰKÖDÉSE



EmailStr, BaseModel, str

```
message = {  
    1: {  
        {"name": "Bob"},  
        {"level": 1},  
        {"profession": None}  
    }  
}
```

```
#{"name": "Bob"}, vagy a {"level": 1} - Dict(str,Any)
```

from typing import Optional

```
class User(BaseModel):  
    name: str  
    level: int = 0  
    email: EmailStr  
    profession: Optional[str] = None
```

```
from pydantic  
import  
field_validator
```

```
class User(BaseModel):  
    name: str  
    level: int = 0  
    email: EmailStr  
    profession: Optional[str] = None  
  
    @field_validator("level")  
    def checklevel(cls, level):  
        if (level > 5):  
            raise ValueError("Error: This field must be under five!")  
        return level
```

from pydantic import model_validator

```
@model_validator(mode="after")
def check_consistency(cls, model):
    #Például: ha profession nincs meg, level nem lehet 5
    if model.profession is None and model.level == 5:
        raise ValueError("If profession is missing, level cannot be 5")
    return model
```

Fájlok (json)

- `__file__`: a jelenlegi Python modul fájlneve
- `os.path.dirname(__file__)`: visszaadja a fájl könyvtárát
- `os.path.join(..., "data.json")`: a könyvtárhoz fűzi a "data.json"-t

Amire szükség lesz:

- `import json`
- `import os`
- `from typing import Dict, Any`

#A JSON fájl elérési útja

```
JSON_FILE_PATH = os.path.join(os.path.dirname(__file__),  
                                "data.json")
```

```
message = {  
    1: {  
        "name": "Bob",  
        "level": 1,  
        "profession": None  
    }  
}
```

#{"name": "Bob"}, vagy a {"level": 1} - Dict(str,Any)

Json fájl betöltése

```
def load_json() -> Dict[str, Any]:  
    with open(JSON_FILE_PATH, "r", encoding="utf-8") as file:  
        return json.load(file)
```

```
message = {  
    1: {  
        "name": "Bob",  
        "level": 1,  
        "profession": None  
    }  
}
```

```
class User(BaseModel):  
    name: str  
    level: int = 0  
    email: EmailStr  
    profession: Optional[str] = None
```


Json fájl mentése

`json.dump(data, file)`: soronként
JSON-t ír

`indent=2`: formázás (pretty-print) két
szóköznyi behúzással

`ensure_ascii=False`: megengedi a
nem-ASCII ékezetes betűket
eredeti formában - nem escape-eli
őket `\uXXXX` formára.

```
def save_json(data: Dict[str, Any]) -> None:  
    with open(JSON_FILE_PATH, "r", encoding="utf-8") as file:  
        json.dump(data, file, indent=2, ensure_ascii=False)
```

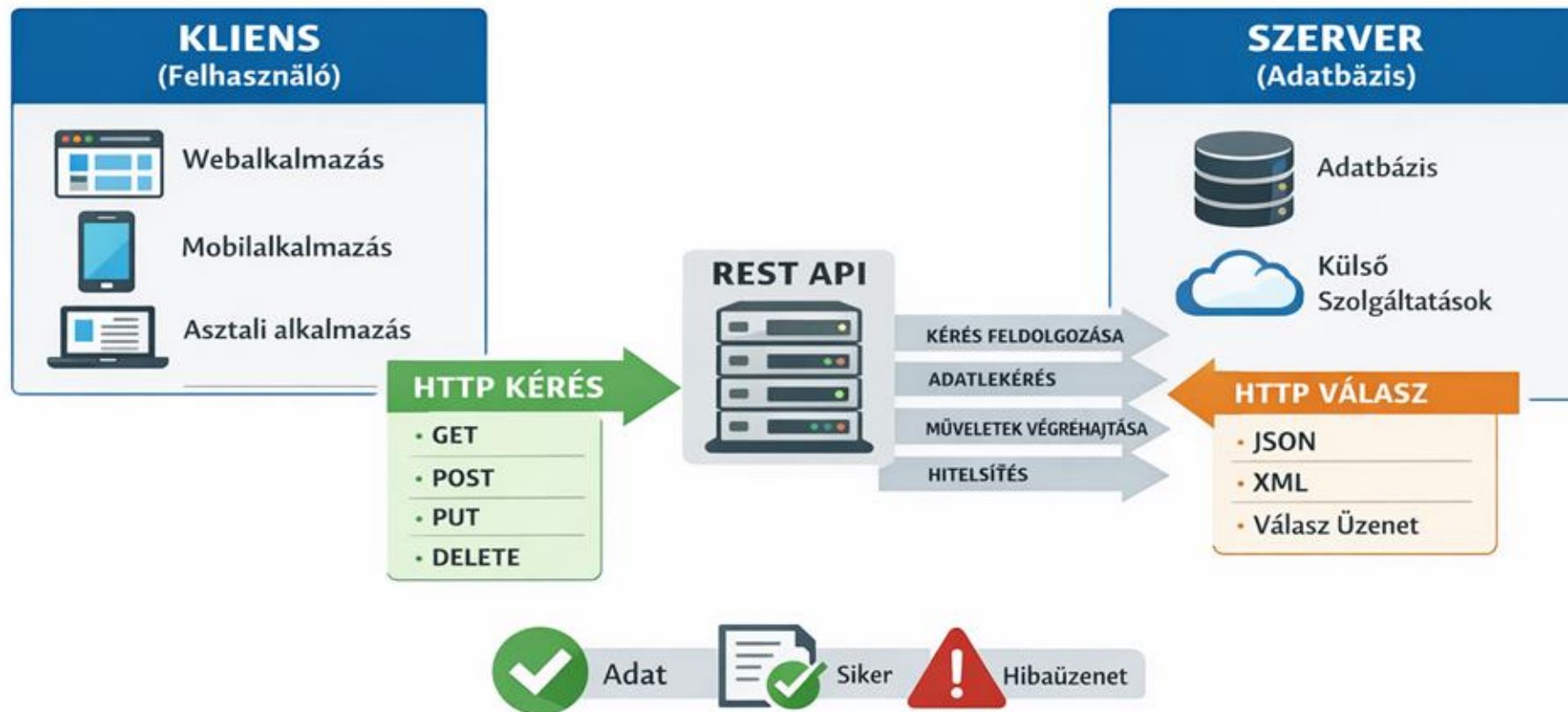
Json adatok kezelése

```
def add_user(user: Dict[str, Any]) -> None:
    data = load_json()
    data["Users"].append(user)
    save_json(data)

new_user = {
    "name": "Roli",
    "email": "roland@example.com"}

add_user(new_user)
```

REST API MŰKÖDÉSE



Fast API

- pip(3) install fastapi, uvicorn
- uv init fstapi
- uv add fastapi (env!)
- (postman)
- doc/redoc

- **Egységes interfész (uniform interface):** Az API-n keresztül elérhető erőforrásokat egységes módon lehet kezelni. Ezek a műveletek általában a **HTTP módszerek**hez kötődnek:
 - **GET:** Adatok lekérése
 - **POST:** Új adat létrehozása
 - **PUT:** Adat frissítése
 - **DELETE:** Adat törlése
- **Erőforrás alapú:** Az API erőforrásokat (resources) kezel, amelyek URL-ekkel azonosíthatók.

JSON alapú kommunikáció

- Az API JSON (JavaScript Object Notation) formátumú válaszokat ad vissza.

```
message = {  
    1: {  
        {"name": "Bob"},  
        {"level": 1},  
        {"profession": None}  
    }  
}
```

```
#{"name": "Bob"}, vagy a {"level": 1} - Dict(str,Any)
```

2. RESTful API alapok és HTTP protokoll

2.1. RESTful API fogalma: A REST egy **architekturális stílus**, amelyet gyakran használnak web API-k tervezésére és fejlesztésére.

A **RESTful API** (REST-kompatibilis API) **alapelvei**:

- **Kliens-szerver architektúra:** Az API külön választja a kliens (amely kérdez) és a szerver (amely válaszol) logikáját.
- **Állapotmentesség (statelessness):** Minden kérelem (request) független, a szerver nem tárolja az előző kérések állapotát.

HTTP(S) protokoll

A HTTP(s)! (Hypertext Transfer Protocol), amelyen keresztül az API-k is kommunikálnak. A HTTP kérés-válasz modellben működik, ahol a kliens küld egy kérést a szervernek, és a szerver választ ad rá. (stateless!)

A HTTP kérés részei:

- **URL:** Az erőforrás elérési útvonala.
- **HTTP metódus:** A művelet típusa (pl. GET, POST).
- **Fejléc (headers):** Metaadatok a kérésről (pl. az adatok formátuma).
- **Törzs (body):** A kérés tartalma (pl. POST és PUT kéréseknél).

A HTTP válasz részei:

- **Állapotkód (status code):** A válasz státusza (pl. 200 OK, 404 Not Found).
- **Fejléc (headers):** Metaadatok a válaszról.
- **Törzs (body):** A válasz tartalma, amely általában JSON formátumban van.

A Fast API használata

Telepítés: `pip(3) install fastapi, uvicorn`
vagy

- `uv init mappa`
- `uv add fastapi`
- `uv run uvicorn main:app --reload`

```
app = FastAPI()

@app.get('/')
def root():
    return {'root': 'Hello'}
```

2.4. HTTP státuskódok kezelése

```
from fastapi import FastAPI, HTTPException

app = FastAPI()

@app.get("/adat/{adat_id}")          # GET kérés státuskód
kezeléssel

def adat_lekerese(adat_id: int):

    if adat_id == 1:

        return {"adat_id": adat_id, "nev": "Példa adat"}

    raise HTTPException(status_code=404, detail="Adat nem
található")
```

A FastAPI lehetővé teszi azt is, hogy különböző HTTP válaszkódokat adjunk vissza.

HTTP státuskódok ismertetése:

[https://www.w3schools.com/tags/ref_httpmessages.a
sp](https://www.w3schools.com/tags/ref_httpmessages.asp)

Magyarázat:

A példában, ha az adat_id értéke 1, akkor visszatérünk egy adattal.

Ha nem találunk ilyen azonosítót, akkor egy **404-es hibát** adunk vissza.

2.5. HTTP kérés-válasz folyamat FastAPI-ban

A FastAPI automatikusan kezeli a HTTP kérés-válasz ciklust, és JSON válaszokat generál az API hívásokra. A kliens a HTTP metódusok használatával **kérdezi le**, **hozza létre** vagy **törli** az adatokat, és a szerver a megfelelő válaszkóddal és üzenettel válaszol.

2.6. FastAPI futtatása

Az alkalmazás futtatásához használjuk az **uvicorn** szerveret:

uvicorn main:app --reload/uv run uvicorn main:app --reload

Ez elindítja a FastAPI szerveret, amelyet elérhetünk a böngészőben vagy API tesztelő eszközökkel (pl. Postman).

3. FastAPI alapok: route-ok, request és response kezelés

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/udvozlet")

def udvozlet():

    return {"uzenet": "Üdvözöllek a FastAPI világában!"}
```

3.1. Route-ok (útvonalak) FastAPI-ban

A route-ok vagy útvonalak határozzák meg az alkalmazás különböző végpontjait (endpoints), amelyek elérhetők a HTTP metódusok segítségével. Minden útvonal egy adott URL-hez és HTTP metódushoz kötött. FastAPI-ban egyszerűen hozhatunk létre különböző útvonalakat.

Példák különböző HTTP metódusokra:

GET: adatok lekérésére,

POST: új adatok létrehozására,

PUT: már létező adatok módosítására,

DELETE: adatok törlésére használjuk.

Magyarázat:

A példa egy egyszerű **GET** kérés kezelésére szolgáló útvonalat definiál. Ha valaki a /udvozlet URL-t hívja meg, a szerver egy "Üdvözöllek a FastAPI világában!" üzenetet küld vissza JSON formátumban.

Példa POST kérésre FastAPI-ban:

```
from fastapi import FastAPI

from pydantic import BaseModel

app = FastAPI()

class Adat(BaseModel):          # Modell az adat validációhoz

    nev: str

    életkor: int

@app.post("/uj_felhasznalo")

def uj_felhasznalo(adat: Adat):

    return {"uzenet": f"Új felhasználó hozzáadva: {adat.nev}, életkor: {adat.életkor}"}
```

Magyarázat:

Ebben a példában a **POST** kérést használjuk arra, hogy új adatokat fogadjunk el a kliens oldaláról. A kliens egy JSON formátumú adatot küld, amely tartalmazza a felhasználó nevét és életkorát. A **pydantic modellt** használjuk az adat validálására, hogy biztosak legyünk abban, hogy a megfelelő formátumú adat érkezett.

3.2. Request (kérés) kezelés

```
from fastapi import FastAPI

from typing import Optional

app = FastAPI()

@app.get("/konyv/")

def konyv_lekerese(cim: Optional[str] = None, szerzo: Optional[str] =
None):

    if cim and szerzo:

        return {"uzenet": f"A könyv címe: {cim}, szerzője: {szerzo}"}

    return {"uzenet": "Nem adtál meg információt a könyvről."}
```

A **request** az a HTTP kérés, amelyet a kliens küld a szervernek. A kérés tartalmazhat különféle információkat, mint például URL paraméterek, lekérdezési paraméterek, kérés törzse (body), és fejléc (headers).

<- Példa a GET request paraméterek kezelésére

Magyarázat:

A kódban opcionális paramétereket (címet, szerzőt) várunk a kérésben. Ha mindkét paramétert megadjuk, akkor a visszaadott üzenet tartalmazza a könyv címét és szerzőjét. Ha nem adunk meg paramétert, egy alapértelmezett üzenetet kapunk.

Request objektum használata POST metódusnál:

```
from fastapi import FastAPI, Request

app = FastAPI()

@app.post("/adat_fogadasa")
async def adat_fogadasa(request: Request):
    # aszinkron adatfogadás beállítása

    json_adat = await request.json()
    # adat fogadása aszinkron módon

    return {"kapott_adat": json_adat}

curl -X POST "http://127.0.0.1:8000/data_request" -H
"Content-Type: application/json" -d '{"kulcs": "ertek"}'
```

Magyarázat:

Ebben a példában a **Request** objektumot használjuk, hogy közvetlenül hozzáférjünk a kérés tartalmához. Az **async** és **await** kulcsszavakat használjuk, mert a FastAPI **aszinkron** támogatást biztosít, ami gyorsabb válaszidőt eredményezhet nagy terhelés alatt.

3.3. Response (válasz) kezelés

```
from fastapi import FastAPI

from fastapi.responses import JSONResponse

app = FastAPI()

@app.get("/json_valasz")

def json_valasz():

    tartalom = {"uzenet": "Ez egy JSON válasz!"}

    return JSONResponse(content=tartalom)
```

A response a szerver által adott válasz, amelyet a kérés feldolgozása után küld vissza a kliensnek. A válasz tartalmazhat adatokat (pl. JSON formátumban), valamint HTTP státuszkódokat és fejléceket is.

<- Egyszerű válasz JSON formátumban manuális beállítással

Magyarázat:

Itt egy JSON választ adunk vissza manuálisan, a **JSONResponse** objektum segítségével. FastAPI alapértelmezésben automatikusan JSON válaszokat generál, de **manuálisan is megadhatjuk a válasz formátumát.**

HTTP státuszkód és fejléc beállítása a válaszban:

```
from fastapi import FastAPI

from fastapi.responses import JSONResponse

app = FastAPI()

@app.get("/egyedi_valasz")

def egyedi_valasz():

    tartalom = {"uzenet": "Ez egy egyedi válasz!"}

    fejlecek = {"X-Special-Header": "Egyedi fejléc érték"}

    return JSONResponse(content=tartalom, headers=fejlecek,
status_code=201)
```

Magyarázat:

A kódban beállítunk egy egyedi HTTP státuszkódot (201), és egy egyedi fejlécet (**X-Special-Header**). Ezzel **testreszabhatjuk** a válaszunkat.

Redirect válasz (átirányítás):

```
from fastapi import FastAPI

from fastapi.responses import RedirectResponse

app = FastAPI()

@app.get("/atiranyitas")

def atiranyitas():

    return RedirectResponse(url="/udvozlet")
```

Magyarázat:

A **RedirectResponse** által egy átirányítást hozunk létre, amely egy másik útvonalra irányítja át a kérést. Most a **/udvozlet végpont** felé történik az **átirányítás**.

3.4. Response modellek használata

```
from fastapi import FastAPI

from pydantic import BaseModel

app = FastAPI()

class ValaszModell(BaseModel):

    uzenet: str

    adat: dict

@app.get("/valasz_modell", response_model=ValaszModell)

def valasz_modell():

    return {"uzenet": "Sikeres válasz!", "adat": {"kulcs": "ertek"}}
```

FastAPI lehetőséget biztosít arra, hogy válaszainkat **modellekkel validáljuk**. Ez azt jelenti, hogy meghatározhatjuk, milyen formátumú adatokat adhatunk vissza.

<- Példa válasz modellre:

Magyarázat:

A **response_model** paraméter használatával megadhatjuk, hogy a visszaadott adat formátuma illeszkedjen egy **Pydantic modell**hez. Ezzel biztosítjuk, hogy a válaszok mindig a megfelelő szerkezetűek legyenek.

4. JSON válaszok készítése FastAPI-val

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/json")

def egyszeru_json_valasz():

    return {"uzenet": "Ez egy egyszerű JSON válasz."}
```

Kimenet:

```
{

  "uzenet": "Ez egy egyszerű JSON válasz."

}
```

Áttekintjük, hogy hogyan készíthetünk és kezelhetünk JSON formátumú válaszokat, amelyeket a FastAPI automatikusan támogat. A JSON (JavaScript Object Notation) egy könnyen olvasható, könnyen feldolgozható adatsere-formátum, amely a legtöbb modern webes alkalmazásban az alapvető adatformátum.

<- Egyszerű példa egy JSON válaszra

4.1. Alapértelmezett JSON válaszok FastAPI-ban

A FastAPI alapvetően automatikusan JSON formátumban adja vissza a válaszokat, ha a visszaadott adat egy **Python dict típusú** objektum. A kliens tehát egy JSON válaszban fogja megkapni az adatokat.

Magyarázat (serializálja):

Itt egy egyszerű útvonalat hozunk létre, amely visszaad egy **Python dict** objektumot. A

4.2. JSON válasz testreszabása

```
from fastapi import FastAPI

from fastapi.responses import JSONResponse

app = FastAPI()

@app.get("/egyedi_json")

def egyedi_json_valasz():

    tartalom = {"uzenet": "Ez egy egyedi JSON válasz."}

    return JSONResponse(content=tartalom, status_code=201)
```

Bár a FastAPI alapértelmezés szerint JSON formátumot használ, lehetőség van testreszabni a válaszokat. Ehhez használhatjuk a FastAPI beépített **response** osztályait, például a `JSONResponse`-t, amely lehetőséget ad a **válasz részleteinek kézi beállítására**.

<- Példa a `JSONResponse` használatára

Magyarázat:

Ebben a példában a **`JSONResponse`**-t használjuk arra, hogy egy egyedi JSON válasz formátumot és egyedi státuszkódot adjunk vissza (201 Created).

Ez a válasz rugalmasabb testreszabási lehetőségeket biztosít, például további HTTP fejlécek hozzáadását.

4.3. *Pydantic modellek használata a JSON válaszok validálására*

```
from fastapi import FastAPI

from pydantic import BaseModel

app = FastAPI()    # Pydantic modell a válasz adatainak szerkezetéhez

class FelhasználóValasz(BaseModel):

    nev: str

    életkor: int

@app.get("/felhasznalo", response_model=FelhasznaloValasz)

def felhasznalo_lekerese():

    return {"nev": "János", "életkor": 30}

Kimenet: { "nev": "János", "életkor": 30 }
```

A FastAPI egyik nagy előnye, hogy a **Pydantic** könyvtárral integrálódik, amely lehetővé teszi a JSON adatok validálását és szerializálását. A Pydantic modellek segítségével biztosíthatjuk, hogy a visszaadott adatok megfeleljenek az előre definiált szerkezetnek.

<- Példa válasz modell használatára

Magyarázat:

A **response_model** paraméter használatával meghatározzuk, hogy milyen formátumú adatokkal szeretnénk visszatérni.

A FastAPI automatikusan ellenőrzi, hogy a visszaadott adatok illeszkednek-e a Pydantic modellhez, és ha szükséges, validálja és konvertálja az adatokat.

4.4. Egyszerű és összetett válasz modellek

A Pydantic lehetővé teszi, hogy összetettebb válaszmodelleket is definiáljunk, például beágyazott struktúrákat vagy listákat.

Példa összetett válasz modellre ->

Magyarázat:

Ebben az esetben a válaszmodell összetettebb, hiszen egy **List** típusú mezőt tartalmaz, amely lakcímek listáját adja vissza.

A beágyazott Cim modell biztosítja, hogy minden lakcím helyes struktúrájú legyen.

```
from fastapi import FastAPI    from pydantic import BaseModel    from typing import List

app = FastAPI()                # Pydantic modellek

class Cim(BaseModel):

    utca: str

    varos: str

class FelhasznaloValasz(BaseModel):

    nev: str

    etekor: int

    cimek: List[Cim]

@app.get("/felhasznalo_osszetett", response_model=FelhasznaloValasz)

def felhasznalo_osszetett():

    return {

        "nev": "Anna",

        "etekor": 25,

        "cimek": [

            {"utca": "Kossuth utca", "varos": "Budapest"},

            {"utca": "Rákóczy utca", "varos": "Szeged"}

        ]

    }
```

4.5. HTTP válaszkódok és JSON válaszok kezelése

```
from fastapi import FastAPI, HTTPException

app = FastAPI()

@app.get("/ellenorzes/{felhasznalo_id}")

def felhasznalo_ellenorzes(felhasznalo_id: int):

    if felhasznalo_id == 1:

        return {"uzenet": "Felhasználó megtalálva."}

    else:

        raise HTTPException(status_code=404, detail="Felhasználó nem  
található.")
```

A FastAPI lehetőséget biztosít arra, hogy a válaszkódokat manuálisan beállítsuk, és közben JSON válaszokat adjunk vissza. Így a megfelelő státuszkóddal együtt a kliens pontosan megérti, hogy mi történt a kérése során.

<- Példa HTTP válaszkód beállítására

Magyarázat:

Ha a felhasználó azonosítója nem létezik (pl. nem 1), akkor egy 404-es státuszkóddal térünk vissza. Az ilyen típusú hibakezelés rendkívül fontos az API-k megbízhatóságának növeléséhez.

4.6. Fejlécek és sütik hozzáadása a válaszhhoz

```
from fastapi import FastAPI

from fastapi.responses import JSONResponse

app = FastAPI()

@app.get("/fejlécek")
def fejlécek_hozzaadasa():
    tartalom = {"uzenet": "Fejléc hozzáadva a válaszhhoz."}

    return JSONResponse(content=tartalom, headers={"X-
Custom-Header": "EgyediErtek"})
```

A JSON válaszhokon túl a FastAPI lehetőséget biztosít fejlécek (**headers**) és sütik (cookies) hozzáadására a válaszhokhoz.

<- **Példa válaszh fejlécek hozzáadására:**

Magyarázat:

Ebben a példában egy egyedi fejlécket adunk a válaszhhoz. Ez gyakran hasznos lehet, ha speciális metaadatokat szeretnénk küldeni a kliensnek, például biztonsági vagy követési információkat. (token)

4.7. Async JSON válaszok kezelése

```
from fastapi import FastAPI

import asyncio                # aszinkron
input/output

app = FastAPI()

@app.get("/async_json")

async def async_json_valasz():

    await asyncio.sleep(2)     # Szimulálunk egy késleltetést

    return {"uzenet": "Ez egy aszinkron JSON válasz, 2
másodperc késleltetéssel."}
```

FastAPI teljes mértékben támogatja az aszinkron műveleteket. Az aszinkron válaszok lehetővé teszik a jobb teljesítményt, különösen akkor, ha a szervernek más forrásokhoz (pl. adatbázisokhoz) kell hozzáférnie a válasz előtt.

<- Példa aszinkron JSON válaszra

Magyarázat:

Az `async` és `await` használata lehetővé teszi, hogy a válasz késleltetést szimuláljuk. Ez gyakran előfordulhat például adatbázis-műveletek során, ahol a válasz megvárja az adatbázisból érkező eredményt.

5. Egyszerű API létrehozása és telepítése

5.1. Egyszerű API létrehozása FastAPI-val

FastAPI segítségével nagyon egyszerű egy API-t létrehozni, amely különböző funkciókat kínálhat a felhasználóknak. Például készíthetünk egy API-t, amely kezeli a felhasználók adatait, termékeket listáz, vagy egyszerű matematikai műveleteket végez.

5.1.1. API alap létrehozása

Első lépésként hozzuk létre a projektet és telepítsük a szükséges csomagokat.

`pip install fastapi uvicorn`

A projekt fájljait egy **main.py** nevű fájlban fogjuk létrehozni. Ez a fájl lesz az API belépési pontja.

5.1.2. Egyszerű FastAPI API kód

Létrehozunk egy egyszerű FastAPI alkalmazást, amely két végpontot biztosít: az egyik üdvözlő minket, a másik pedig felhasználói adatokat ad vissza (l. a következő dián):

Forráskódok

Magyarázat:

1. Létrehoztuk az alkalmazást egy FastAPI() példányon keresztül.
2. Létrehoztunk egy Felhasznalo nevű Pydantic modellt, amely biztosítja, hogy a felhasználói adatok helyes szerkezetűek legyenek.
3. Egy üdvözlő végpontot definiáltunk a GET metódussal, amely visszaad egy egyszerű üzenetet.
4. A /felhasznalok/ végpont visszaad egy listát felhasználókról.

```
from fastapi import FastAPI

from pydantic import BaseModel

from typing import List    # Alkalmazás inicializálása:

app = FastAPI()    # Felhasználó modell:

class Felhasznalo(BaseModel):

    nev: str

    etekor: int    # Üdvözlő végpont:

@app.get("/")

def udvozles():

    return {"uzenet": "Üdvözöllek a FastAPI alkalmazásban!"}    # Felhasználói adatok lekérdezése

@app.get("/felhasznalok/", response_model=List[Felhasznalo])

def felhasznalok_listazasa():

    felhasznalok = [

        {"nev": "Anna", "etektor": 25},

        {"nev": "János", "etektor": 30}

    ]

    return felhasznalok
```

5.2. API futtatása lokálisan

A FastAPI alkalmazást lokálisan futtathatjuk az uvicorn szerver segítségével.

Az uvicorn egy gyors ASGI (Asynchronous Server Gateway Interface) szerver, amely képes futtatni FastAPI alkalmazásokat.

Parancs az API futtatásához: `uvicorn main:app --reload`

Magyarázat:

- Az `uvicorn main:app --reload` parancs elindítja a FastAPI alkalmazást az uvicorn szerveren, ahol a **main** a fájl neve, és az **app** az alkalmazás objektum neve.
- A **--reload** kapcsoló engedélyezi az automatikus újratöltést, amikor a fájlok módosulnak, ami hasznos a fejlesztés során.

Ezt követően az API elérhető lesz a `http://127.0.0.1:8000` címen. Itt megtekinthetjük az alkalmazás dokumentációját is a FastAPI által automatikusan generált interaktív felületen, amely a **Swagger UI**.

Swagger UI elérése:

Az API dokumentációját a következő URL-en érhetjük el: `http://127.0.0.1:8000/docs`

5.3. API telepítése webszerverre

```
git init
git add .
git commit -m "Első commit"
```

Miután az API-t lokálisan sikeresen teszteltük, a következő lépés az, hogy elérhetővé tegyük az interneten. Ehhez szükségünk lesz egy szerverre, ahol az alkalmazás futtatható. Ehhez sok szolgáltató közül választhatunk, például Heroku, DigitalOcean, AWS, Google Cloud, vagy Render.

5.3.1. Heroku használata FastAPI telepítéséhez

A Heroku egy felhő alapú platform, amely egyszerűen használható API-k telepítéséhez. **Példaként** bemutatjuk, hogyan telepíthetünk egy FastAPI alkalmazást a Heroku-ra:

1. Git repository létrehozása

Először hozzuk létre a projektet git repository-ként:

2. Procfile létrehozása

A Procfile egy speciális fájl, amely meghatározza, hogyan futtassa a Heroku az alkalmazásunkat. Ebben meg kell adni az uvicorn szerver indítási parancsát.

Procfile tartalma:

```
web: uvicorn main:app --host=0.0.0.0 --port=${PORT}
```

3. requirements.txt létrehozása

A Heroku Python környezetben futtatja az alkalmazásokat, ezért szükség van egy **requirements.txt** fájlra, amely tartalmazza a szükséges Python csomagokat.

pip freeze > requirements.txt

4. Heroku alkalmazás létrehozása és telepítése

1. Jelentkezz be a Heroku CLI-be: **heroku login**
2. Hozz létre egy új Heroku alkalmazást: **heroku create**
3. Telepítsd a kódot a Heroku-ra: **git push heroku master**
4. Az alkalmazás futtatása után a Heroku kijelöl egy URL-t, amelyen az API elérhető lesz. Például: **<https://sajat-api-app.herokuapp.com>**
5. A **Swagger UI** dokumentáció szintén elérhető lesz a /docs végponton.

5.4. API skálázása és optimalizálása

Amint az API már él, érdemes gondolkodni annak skálázhatóságáról és optimalizálásáról. Ez magában foglalhatja a következőket:

- **Adatbázis kapcsolatok kezelése** (pl. SQLAlchemy, MongoDB integráció).
- **Aszinkron feldolgozás használata**, ami növeli a teljesítményt.
- **Cache alkalmazása** az API válaszidők optimalizálására (pl. Redis).
- **Load balancer** használata nagyobb terhelés kezelésére.

6. FastAPI és adatbázis integráció (SQLAlchemy bevezetése)

6.1. SQLAlchemy áttekintése

Az **SQLAlchemy** az egyik legnépszerűbb **ORM** (Object Relational Mapping) könyvtár a Pythonban. Egy hatékony ORM keretrendszer, amely lehetővé teszi az **adatbázis táblák Python osztályokként** történő leképezését. Ezáltal egyszerűbbé válik az adatbázissal való munka, és kevésbé kell SQL kódot írni, mivel az SQLAlchemy elintézi a háttérben a lekérdezések generálását. Az adatbázis integrációs folyamat főbb lépései:

1. **Adatbázis kapcsolat létrehozása.**
2. **Modellek létrehozása**, amelyek megfelelnek az adatbázis tábláknak.
3. **CRUD műveletek** (Create, Read, Update, Delete) implementálása.
4. **Adatbázis tranzakciók kezelése.**

6.2. Adatbázis kapcsolat létrehozása FastAPI-ban

```
pip install sqlalchemy  
pip install databases  
pip install asyncpg      # Ha PostgreSQL-t használunk,  
szükséges
```

Az első lépés, hogy létrehozzuk az adatbázis kapcsolatot. SQLAlchemy többféle adatbázist támogat, beleértve a MySQL, PostgreSQL, SQLite és mások.

<- SQLAlchemy és más szükséges csomagok telepítése

Forráskódok

6.2.1. Adatbázis kapcsolat konfigurációja

Az adatbázis kapcsolatot konfigurálhatjuk egy aszinkron motorral (például PostgreSQL-lel).

Például egy **PostgreSQL adatbázis** használata esetén az alábbi kódot alkalmazzuk ->

Magyarázat:

- A **create_engine** funkció segítségével hozunk létre kapcsolatot az adatbázissal. A kapcsolat konfigurációját a **DATABASE_URL** tartalmazza.
- A **SessionLocal** biztosítja az adatbázis tranzakciók kezelését. Ezen keresztül kommunikálunk majd az adatbázissal.
- A **Base** az összes adatbázis modellünk alapjául szolgál, amely az adatbázis táblákat definiálja.

```
from sqlalchemy.ext.declarative import declarative_base
```

```
from sqlalchemy import create_engine
```

```
from sqlalchemy.orm import sessionmaker
```

```
# Adatbázis URL
```

```
DATABASE_URL =
```

```
"postgresql://felhasznalonev:jelszo@localhost/adatbazis_nev"
```

```
# SQLAlchemy motor inicializálása
```

```
engine = create_engine(DATABASE_URL)
```

```
# Adatbázis session
```

```
SessionLocal = sessionmaker(autocommit=False, autoflush=False,  
bind=engine)
```

```
# Alap osztály a modellek számára
```

```
Base = declarative_base()
```

6.3. Modellek létrehozása

```
from sqlalchemy import Column, Integer, String

# Modell osztály létrehozása, amely egy adatbázis táblát reprezentál

class Felhasznalo(Base):

    __tablename__ = "felhasznalok"

    id = Column(Integer, primary_key=True, index=True)

    nev = Column(String(60), index=True)

    email = Column(String, unique=True, index=True)

    életkor = Column(Integer)
```

A modellek az adatbázis táblák reprezentációi Python osztályokként.

Ezek segítségével definiáljuk az adatbázis táblák mezőit és azok típusait.

6.3.1. Felhasználó modell létrehozása

A példa bemutatja egy egyszerű Felhasználó modell létrehozását, amely egy adatbázis táblát képvisel:

Magyarázat:

A Felhasznalo osztályunk egy **adatbázis táblát** reprezentál. Az egyes **oszlopok** a tábla **mezőit** képviselik (pl. id, nev, email, életkor).

Az `__tablename__` attribútum határozza meg a **tábla nevét** az adatbázisban.

A **Column** segítségével adjuk meg az egyes **mezők típusát** és **tulajdonságait** (pl. `primary_key`, `unique`, `index`).

Példa adatbázis session kezelésére:

Magyarázat:

A **get_db()** függvény egy új adatbázis sessiont hoz létre minden kéréshez, majd bezárja azt, amikor a kérés befejeződik.

A **Depends(get_db)** segítségével adjuk át a session objektumot a végpontjainknak, így minden végpont közvetlenül elérheti az adatbázist.

```
from fastapi import Depends, FastAPI, HTTPException

from sqlalchemy.orm import Session

from . import crud, models, schemas

from . database import SessionLocal, engine

models.Base.metadata.create_all(bind=engine)

app = FastAPI()

def get_db():                                # Adatbázis session kezelés

    db = SessionLocal()

    try:

        yield db

    finally:

        db.close()

# Példa végpont: Felhasználó létrehozása:

@app.post("/felhasznalok/", response_model=schemas.Felhasznalo)

def felhasznalo_letrehozasa(felhasznalo: schemas.FelhasznaloCreate, db: Session =
Depends(get_db)):

    return crud.uj_felhasznalo_letrehozasa(db, felhasznal)
```

6.4. CRUD műveletek implementálása

```
from sqlalchemy.orm import Session

from . import models, schemas # Új felhasználó létrehozása

def uj_felhasznalo_letrehozasa(db: Session, felhasznalo: schemas.FelhasznaloCreate):

    db_felhasznalo = models.Felhasznalo(nev=felhasznalo.nev, email=felhasznalo.email,
    eletkor=felhasznalo.eletkor)

    db.add(db_felhasznalo)

    db.commit()

    db.refresh(db_felhasznalo)

    return db_felhasznalo
```

A következő lépés az alapvető CRUD (Create, Read, Update, Delete) műveletek implementálása, amelyekkel adatokat hozunk létre, olvasunk ki, módosítunk vagy törölünk az adatbázisból.

6.4.1. Adatok hozzáadása (Create)

<- A példa bemutatja, hogyan adhatunk hozzá új felhasználót az adatbázishoz

Magyarázat:

A Session objektum segítségével kapcsolódunk az adatbázishoz.

A **db.add()** metódus hozzáadja az új felhasználót az adatbázis tranzakcióhoz.

A **db.commit()** elmenti a módosításokat, míg a **db.refresh()** frissíti a felhasználói objektumot az adatbázisból.

6.4.2. Adatok lekérése (*Read*)

```
# Összes felhasználó lekérézése

def felhasznalok_listazasa(db: Session):

    return db.query(models.Felhasznalo).all()

# Egy felhasználó lekérézése az id alapján

def felhasznalo_lekerese(db: Session, felhasznalo_id: int):

    return db.query(models.Felhasznalo).filter(models.Felhasznalo.id ==
felhasznalo_id).first()
```

Lekérhetjük az **összes** felhasználót az adatbázisból, vagy egy adott felhasználót az **id** alapján.

6.4.3. Adatok módosítása (Update)

A kód bemutatja, hogyan
módosíthatunk egy létező
felhasználót

```
def felhasznalo_frissitese(db: Session, felhasznalo_id: int,
uj_adatok: schemas.FelhasznaloUpdate):

    db_felhasznalo =
db.query(models.Felhasznalo).filter(models.Felhasznalo.id ==
felhasznalo_id).first()

    if db_felhasznalo:

        db_felhasznalo.nev = uj_adatok.nev

        db_felhasznalo.eletkor = uj_adatok.eletkor

        db.commit()

        db.refresh(db_felhasznalo)

    return db_felhasznalo
```


6.4.4. Adatok törlése (*Delete*)

```
def felhasznalo_torlese(db: Session, felhasznalo_id: int):  
  
    db_felhasznalo =  
    db.query(models.Felhasznalo).filter(models.Felhasznalo.id ==  
    felhasznalo_id).first()  
  
    if db_felhasznalo:  
  
        db.delete(db_felhasznalo)  
  
        db.commit()  
  
    return {"uzenet": "Felhasználó törölve"}
```

<- Egy felhasználó törlése az alábbi módon történhet

6.5. Adatbázis tranzakciók kezelése FastAPI-ban

Az adatbázis tranzakciók FastAPI-ban általában egy middleware segítségével kezelhetők, amely biztosítja, hogy minden HTTP kéréshez egy új adatbázis session legyen nyitva, majd a kérés befejeztével ez bezáruljon.

A következő dián egy **adatbázis session** kezelését mutatjuk be:

6.6. Adatbázis migrációk kezelése – Alembic

Ha az adatbázis sémája megváltozik (pl. új mezők hozzáadása vagy módosítása), használhatjuk az **Alembic** nevű eszközt, amely SQLAlchemy alapú adatbázis-migrációkat kezel.

Alembic telepítése: `pip install alembic`

Alembic inicializálása: `alembic init alembic`

Az Alembic segítségével nyomon követhetjük a séma változásait, és automatikusan alkalmazhatjuk az adatbázis migrációkat.

Összefoglalás

- Az SQLAlchemy integrációja FastAPI-val lehetővé teszi, hogy hatékonyan kezeljük az adatbázisokat a webalkalmazásunkban.
- Az ORM-en keresztül az adatbázis tranzakciókat könnyen és biztonságosan tudjuk végrehajtani Python objektumok segítségével.
- Az adatbázis-migrációk kezelése, a CRUD műveletek megvalósítása, és a session kezelés mind alapvető építőkövei egy robusztus API-nak, amely biztosítja az adatok biztonságos és hatékony kezelését.

Köszönöm a figyelmet!

ELTE IK

Python kurzus – 8. előadás

Tárgyfelelős: Dr. Tejfel Máté

Előadó: Dr. Király Roland

ELTE IK



Tematika

1. Big Data alapfogalmak
2. Adatelemzés Python eszközökkel
3. Pandas adatstruktúrák: DataFrame, Series
4. Adatok beolvasása és előfeldolgozása
5. Adatmanipulációk (szűrés, rendezés, aggregálás)
6. Adatfeldolgozási feladatok és példák

A prezentációhoz tartozó forráskódok

[San Francisco Crime H.ipynb](#) (jupyter notebook fájl megnyitása)

[Map-Crime Incidents-Previous Three Months.csv](#)(excel file
megnyitása)

[rpi_data_compact.csv](#)(excel fájl megnyitása)

Big Data alapfogalmak

- a) **Az adatok jelentősége, értéke:** Mind az üzleti, mind a tudományos életben egyre fontosabbá válik. Az adatelemzés segít a vállalatoknak és szervezeteknek döntéshozataluk javításában, hatékonyságuk növelésében. Az adatvezérelt döntéshozatal ügyfél és versenytárs elemzésen és a piac változásainak gyorsabb, pontosabb követésén alapul.

- a) **Az adat típusai:**
 - a) **Strukturált adatok:** táblázatok, adatbázisok
 - b) **Strukturálatlan adatok:** szöveges dokumentumok, e-mailek, képek és videók

Big Data alapfogalmak

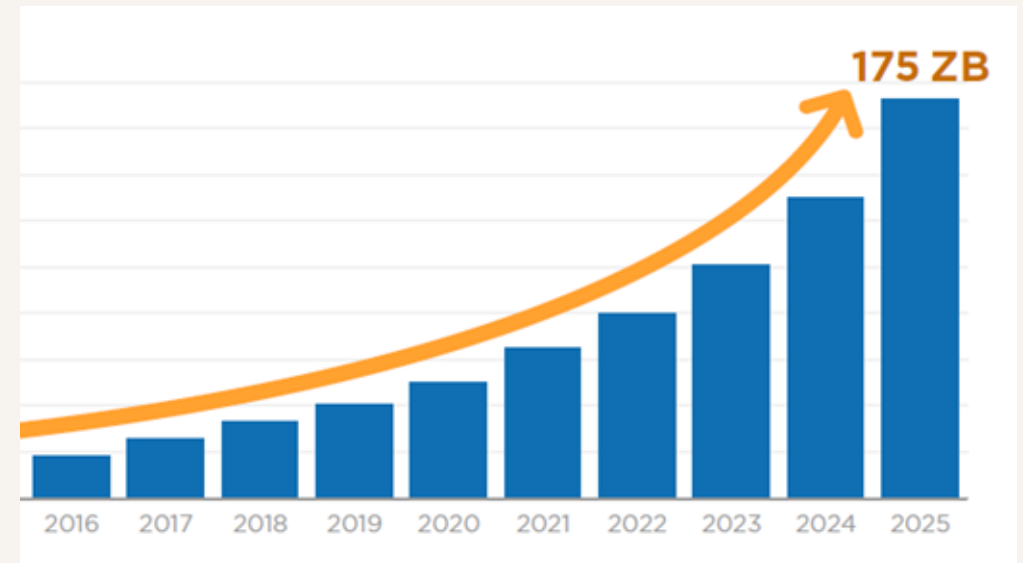
c) Adatok exponenciális növekedése:

Naponta trillió (exa) bájt adat keletkezik az internet, a szenzorok és a digitális eszközök révén. **IOT**

A globális adatállomány évente több, mint 50%-kal nő, és az előrejelzések szerint

2025-re elérte nagyjából a 175 zettabájtot..

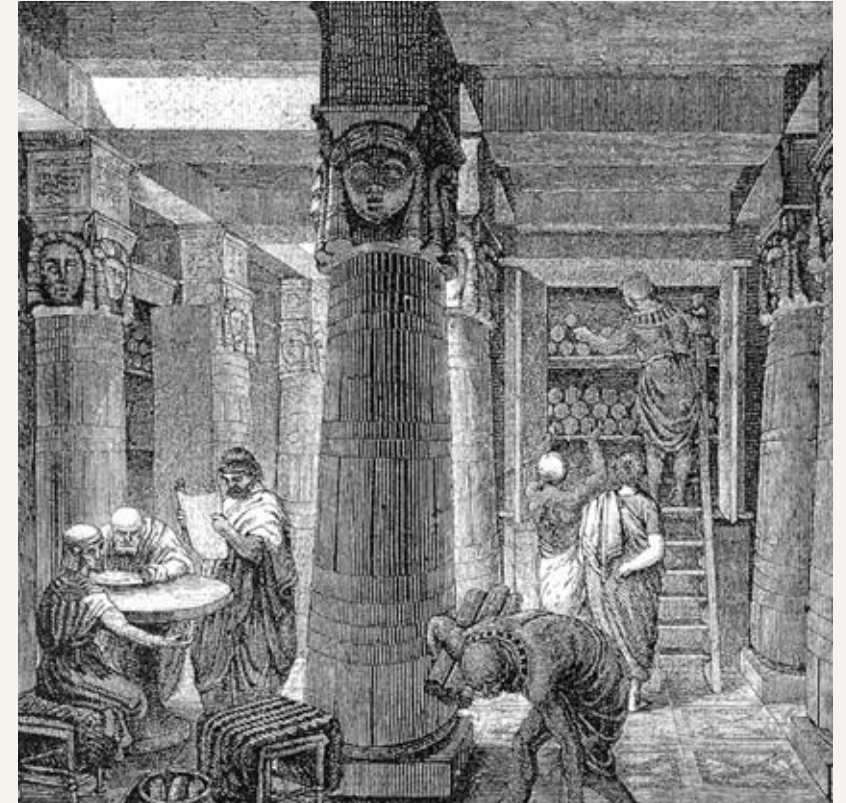
<https://www.idc.com/>



International Data Corporation

Információ

1. Beszéd, alapvető kommunikáció
2. Írás, generációk közötti tudásátadás (i.e. 8. század)
3. Könyvtárak, írott tudás megőrzése és rendszerezés (Alexandria i.e 3. század)
4. Nyomtatás (1450)
5. 2000-2020 – internet forradalma - Web1 (dot.com),
6. Web2
7. 2010+ Big Data, adatelemzés automatizálása
 1. IOT, (Horizont 2020, ...)
8. Adatbányászati módszerek fejlődése (Py)
9. HPC használata adatbányászatra
10. AI alapú adatbányászati módszerek

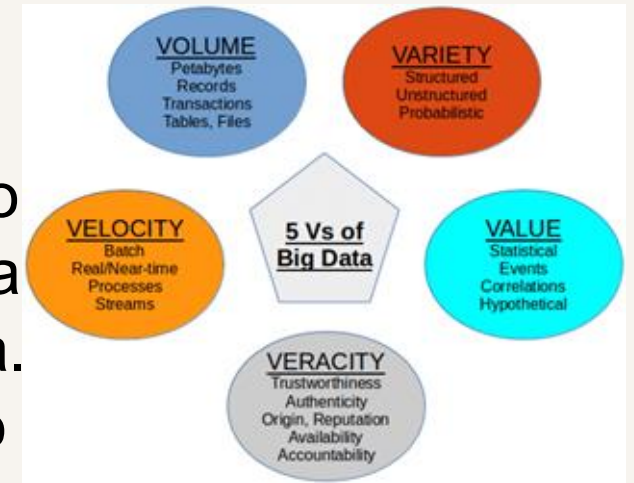


Big Data alapfogalmak

d) **A Big Data definíciója:** Olyan adathalmazok, amelyek túl nagyok vagy összetettek ahhoz, hogy a hagyományos adatfeldolgozási módszerekkel kezelhetőek legyenek. A Big Data rendszerek lehetővé teszik a nagy mennyiségű, gyorsan áramló és különböző forrásokból származó adatok valós idejű elemzését.

Big Data jellemzői: az 5V modell szerint:

- **Volume (Mennyiség):** óriási adatmennyiségek.
- **Velocity (Sebesség):** az adatok gyors beérkezése és feldolgozása.
- **Variety (Változatosság):** különböző típusú és formátumú adatok.
- **Veracity (Hitelesség):** az adatok minősége és pontossága.
- **Value (Érték):** az adatokból kinyerhető hasznos információ.
 - Információ tartalom.



Big Data alapfogalmak

Hozzáférés szerinti csoportosítás:

- **Open Data:** Nyílt adatok, amelyek mindenki számára hozzáférhetők és szabadon használhatók, pl. kormányzati statisztikák, közlekedési adatok.
- **Private Data:** Privát adatok, amelyek egy szervezet vagy egyén tulajdonában vannak, korlátozott hozzáférésűek, pl. ügyféladatbázisok. Üzleti elemzések.

Big Data alapfogalmak

e) Adatkezelési eszközök:

Hagyományos adatok esetén:

A relációs adatbázis-kezelő rendszerek (RDBMS), mint az Oracle, MySQL, SQL Server (kis- és közepes méretű adatok kezelése).

Big Data adatkezelési technológiára példák:

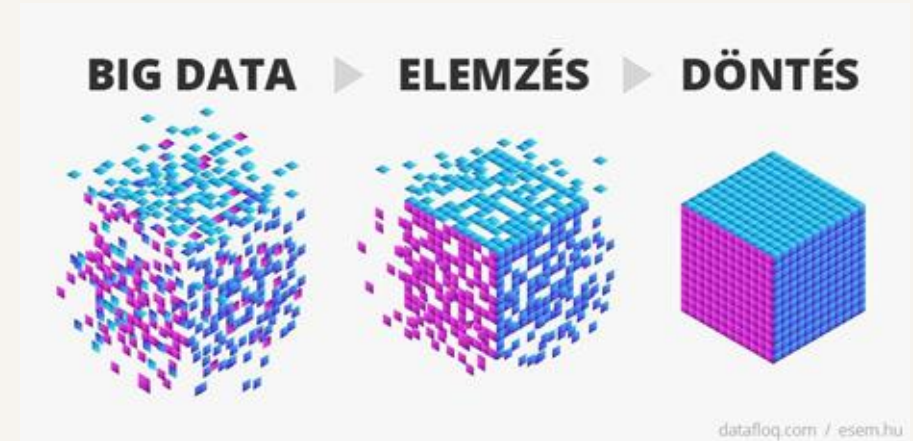
Strukturálatlan adatok: A Hadoop és a NoSQL adatbázisok (pl. MongoDB, Cassandra).

Strukturált adatok: SQLite, beágyazott adatbázis-kezelő rendszerek.

Big Data alapfogalmak

Az adatelemzés lépései:

1. Adatok gyűjtése.
2. Adatok előkészítése és tisztítása.
3. Adatok vizualizációja.
4. Adatok elemzése.
5. Értelmezés és következtetések levonása.
6. Döntéshozatal és intézkedések végrehajtása.

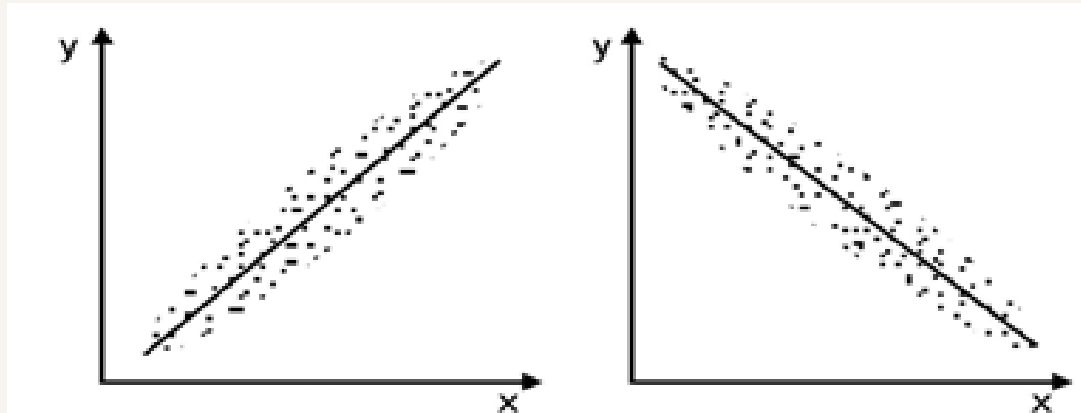


Big Data alapfogalmak

f) Adatelemzési módszerek

Descriptive (leíró): az adatok összegzése, a meglévő információk bemutatása:

- Statisztikai elemzések (átlag, medián, szórás).
- Adatvizualizációs technikák, pl. grafikonok, diagramok.
- Adatbányászati eszközök, az adatok közti összefüggések kimutatására.

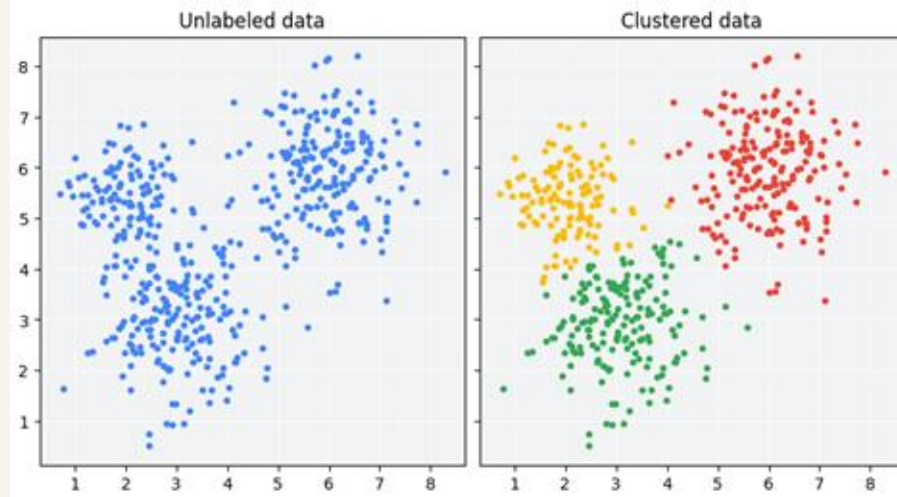


http://www.jgypk.hu/tamop15e/tananyag_html/spss/alapfogalmak3.html

Big Data alapfogalmak

Prediktív (előrejelző): a jövőbeli események előrejelzése:

- Regressziós elemzések, klasszifikáció.
- Gépi tanulási modellek, döntési fák, neurális hálózatok.



<https://developers.google.com/machine-learning/clustering/overview>

Adatelemzés Python eszközökkel

Series példa

```
series = pd.Series([10, 20, 30, 40], index=['a',  
'b', 'c', 'd'])
```

DataFrame példa

```
df = pd.DataFrame({  
    'Column1': [1, 2, 3],  
    'Column2': ['a', 'b', 'c']  
})
```

A Python nyelvű adatalemzés és adatfeldolgozás során a **NumPy** és **Pandas** könyvtárak alapvető eszközök. Ezek hatékony és gyors adatfeldolgozást tesznek lehetővé nagy mennyiségű adattal.

- **NumPy:** Egy hatékony, tömbökkel dolgozó könyvtár, amely nagy mennyiségű numerikus adat kezelésére és számítások végrehajtására alkalmas.
- **Pandas:** Az adatok hatékony feldolgozására és elemzésére szolgál, két fő adatstruktúrája van: **Series** (egydimenziós) és **DataFrame** (kétdimenziós).

Adatelemzés Python eszközökkel

NumPy példa:

```
import numpy as np
# Egydimenziós tömb létrehozása
arr = np.array([1, 2, 3, 4, 5])
print("Egydimenziós tömb:", arr)
# Többdimenziós (mátrix) tömb
matrix = np.array([[1, 2, 3], [4, 5, 6]])
print("Mátrix:\n", matrix)
```

- A NumPy könyvtár a numerikus számítások alapja
- Tömbök használata, amelyek különböző dimenziókat támogatnak (egy-, két- vagy többdimenziós).
- Hatékony matematikai műveletek tömbökkel.
- Beépített függvények a lineáris algebrára, statisztikai számításokra, stb.

```
# Alapvető műveletek
arr_sum = np.sum(arr)
arr_mean = np.mean(arr)
print("Összeg:", arr_sum)
print("Átlag:", arr_mean)
```

Adatelemzés Python eszközökkel

Pandas – Adatfeldolgozás és elemzés

- A Pandas könyvtár lehetővé teszi a komplex adatstruktúrák kezelését és elemzését, legfontosabb elemei a Series és DataFrame objektumok.
- Ezt a könyvtárat használjuk a táblázatos adatok kezelésére, amelyek sorokból és oszlopokból állnak.
- Az adatok előfeldolgozását, manipulálását és megjelenítését is lehetővé teszi.

Pandas főbb elemei:

- **Series:** Egy dimenziós adatszerkezet, amely egy indexelt adattömb, hasonló a Python listákhoz vagy szótárakhoz.
- **DataFrame:** Kétdimenziós, táblázatos adatszerkezet, amely több sorból és oszlopból áll.

Adatelemzés Python eszközökkel

Pandas – Series példák

Egydimenziós adatszerkezet
létrehozása és kezelése:

```
import pandas as pd

# Series létrehozása

series = pd.Series([10, 20, 30, 40], index=['a',
'b', 'c', 'd'])

print("Series példa:\n", series)
```

Pandas – DataFrame példák

```
# DataFrame létrehozása

data = {
    'Name': ['Anna', 'Béla', 'Cecília'],
    'Age': [29, 34, 22],
    'City': ['Budapest', 'Debrecen', 'Szeged']
}

df = pd.DataFrame(data)

print("\nDataFrame példa:\n", df)
```

Adatelemzés Python eszközökkel

Kétdimenziós adatszerkezetek
létrehozása és kezelése:

```
# Adatokhoz való hozzáférés  
print("\nElső sor adatai:\n", df.iloc[0])  
print("\nÉletkor oszlop adatai:\n",  
df['Age'])
```

Adatelemzés Python eszközökkel

Alapvető műveletek a DataFrame-ekkel:

Adatok importálása és exportálása:

- CSV beolvasása DataFrame-be: `df = pd.read_csv('fajl_neve.csv')`
- CSV exportálás: `df.to_csv('output.csv', index=False)`

Alapvető adatvizsgálat:

- Az első néhány sor megtekintése: `df.head()`
- Adatstatisztika: `df.describe()`
- Adattípusok ellenőrzése: `df.dtypes`

Adatok manipulálása:

- Sorok és oszlopok kiválasztása: `df[['Name', 'Age']]`
- Adatok szűrése: `df[df['Age'] > 30]`
- Oszlopok hozzáadása vagy módosítása: `df['NewColumn'] = df['Age'] * 2`

Pandas példa:

Az index=False eredménye, hogy az output.csv fájlban nem lesz benne az első index oszlop.

```
df = pd.read_csv('fajl_neve.csv')
```

```
df.head()
```

```
df.describe()
```

```
df.dtypes
```

```
df[['Name', 'Age']]
```

```
df[df['Age'] > 30]
```

```
# Új oszlop hozzáadása
```

```
df['Age x2'] = df['Age'] * 2
```

```
# Szűrés: csak a 'Debrecen'-ben élők
```

```
debrecen_df = df[df['City'] == 'Debrecen']
```

```
# Átlagéletkor kiszámítása
```

```
average_age = df['Age'].mean()
```

```
print("\nÁtlagéletkor:", average_age)
```

```
df.to_csv('output.csv', index=False)
```

Pandas adatstruktúrák

Series:

A **Series** egy egydimenziós adatszerkezet, amely hasonló a Python listához vagy szótárhoz, de tartalmaz egy kapcsolódó indexet, amely minden értéket azonosít.

```
import pandas as pd
# Series létrehozása
grades = pd.Series([5, 4, 3, 5, 2], index=['tanuló1', 'tanuló2',
'tanuló3', 'tanuló4', 'tanuló5'])
print("Series példa:\n", grades)
# Adathozzáférés index alapján
print("\n'Tanuló1' jegye:", grades['tanuló1'])
# Adatszűrés (csak azok a jegyek, amelyek nagyobbak 3-nál)
filtered_grades = grades[grades > 3]
print("\nSzűrt jegyek:\n", filtered_grades)
```

Series példa:

tanuló1 5

tanuló2 4

tanuló3 3

tanuló4 5

tanuló5 2

dtype: int64

'Tanuló1' jegye: 5

Szűrt jegyek:

tanuló1 5

tanuló2 4

tanuló4 5

Pandas *adatstruktúrák*

DataFrame:

A **DataFrame** egy kétdimenziós, táblázatos adat-struktúra, sorok és oszlopok. Hasonló egy táblázathoz vagy SQL adatbázishoz, ahol az **oszlopok különböző típusú adatokat** tartalmazhatnak.

```
# DataFrame létrehozása
datas = {
    ,name': ['Anna', 'Béla', 'Cecília', 'Dávid', 'Emma'],
    ,age: [29, 34, 22, 30, 25],
    ,city': ['Budapest', 'Debrecen', 'Szeged', 'Pécs', 'Miskolc']
}
students_df = pd.DataFrame(datas)
print("\nDataFrame példa:\n", students_df)
# Oszlop kiválasztása
print("\n'Kor' oszlop:\n", students_df[,age'])
# Sor kiválasztása index alapján
print("\nElső diák adatai:\n", students_df.iloc[0])
# Adatszűrés (csak 30 év feletti diákok)
elder_students = students_df[students_df[,age'] > 30]
print("\n30 év feletti diákok:\n", elder_students)
```

Pandas *adatstruktúrák*

DataFrame példa:

	name	age	city
0	Anna	29	Budapest
1	Béla	34	Debrecen
2	Cecília	22	Szeged
3	Dávid	30	Pécs
4	Emma	25	Miskolc

„Age” oszlop:

0	29
1	34
2	22
3	30
4	25

Name: Age, dtype: int64

Első diák adatai:

name	Anna
age	29
city	Budapest

Name: 0, dtype: object

30 év feletti diákok:

	név	kor	város
1	Béla	34	Debrecen
3	Dávid	30	Pécs

Pandas adatstruktúrák

DataFrame műveletek

Példa: Új oszlop hozzáadása:

```
# Új oszlop hozzáadása a diákok DataFrame-hez
students_df['grades average'] = [4.5, 3.8, 4.0, 4.7, 3.9]
print("\nÚj oszloppal bővített DataFrame:\n",
students_df)
```

Példa: Adatok rendezése:

```
# Adatok rendezése 'kor' szerint csökkenő
sorrendben

sorted_df = student_df.sort_values(by='age',
ascending=False)
print("\nRendezett DataFrame:\n", sorted_df)
```

Pandas adatstruktúrák

Alapvető statisztikai műveletek

**Példa: Átlag, maximum,
szórás és módusz**

```
# Átlagéletkor kiszámítása
avg_age = students_df[,age'].mean()
print("\nÁtlagéletkor:", avg_age)

# Legmagasabb életkor megkeresése
max_age = students_df[,age'].max()
print("\nLegidősebb diák életkora:", max_age)

# Életkorok szórásának kiszámítása
scatter = students_df[,age'].std()
print("\nÉletkorok szórása:", scatter)

# Életkorok móduszának kiszámítása
mode = students_df[,age'].mode()
print("\nÉletkorok módusza:", mode)
```

Adatok beolvasása és előfeldolgozása

Az adatok hatékony beolvasása és előfeldolgozása az adatelemzési folyamat egyik legfontosabb lépése. A **Pandas** könyvtár számos módszert biztosít különböző formátumú adatok importálására és azok előkészítésére az elemzéshez.

Adatok beolvasása

A leggyakoribb adatforrások közé tartoznak a **CSV**, **Excel** és **SQL** adatbázisok. Ezek beolvasására a Pandas különböző függvényeket biztosít:

CSV fájl beolvasása:

```
import pandas as pd
# CSV fájl beolvasása
data_df = pd.read_csv('pelda_adatok.csv')
print("Beolvasott adatok:\n", data_df.head())
```

Adatok beolvasása és előfeldolgozása

**Excel fájl
beolvasása:**

```
# Excel fájl beolvasása
data_df = pd.read_excel('pelda_adatok.xlsx',
sheet_name='Adatlap1')
print("Beolvasott Excel adatok:\n", data_df.head())
```

**Adatok betöltése
SQL adatbázisból:**

```
# SQL adatbázisból történő beolvasás
import sqlite3

connection = sqlite3.connect('adatbazis.db')

data_df = pd.read_sql_query('SELECT * FROM diakok', kapcsolat)
connection.close()
```

Adatok beolvasása és előfeldolgozása

Hiányzó adatok kezelése

Az adatok előfeldolgozásának egyik lépése a hiányzó adatok kezelése. A hiányzó adatok miatt **pontatlan elemzések és hibák léphetnek fel**, ezért fontos, hogy kezeljük őket.

**Hiányzó adatok
eltávolítása és hiányzó
adatok kitöltése:**

```
# Sorok eltávolítása, ahol hiányzó adat van
data_df = data_df.dropna()
print("Hiányzó adatok eltávolítása után:\n", data_df.head())

# Hiányzó értékek kitöltése egy adott értékkel
data_df['age'] = data_df['age'].fillna(30)

# Hiányzó értékek kitöltése az oszlop átlagával
data_df['grades'] = data_df['grades'].fillna(data_df['jegyek'].mean())
print("Hiányzó értékek kitöltése átlaggal:\n", data_df.head())
```

Adatok beolvasása és előfeldolgozása

Adatok típusának konverziója

Az adatelemzési folyamat során előfordulhat, hogy az adatok típusát konvertálni kell, például szöveget számmá vagy dátumformátummá.

Típuskonverzió egész számokká és dátumformátum konverzió:

```
# ,age' oszlop konvertálása egész számokká  
data_df[,age'] = data_df[,age'].astype(int)  
print("Adatok típusa konvertálás után:\n", data_df.dtypes)  
  
# ,date' oszlop konvertálása dátumformátummá  
data_df[,date'] = pd.to_datetime(data_df[,date'])
```


Adatok beolvasása és előfeldolgozása

Adatok előkészítése és adattisztítás

Az adatok előkészítéséhez gyakran szükséges az adatok szűrése, duplikált sorok eltávolítása, illetve az adatok rendezése.

Adatok beolvasása és előfeldolgozása

Duplikált sorok eltávolítása,
adatok rendezése, szűrése:

```
# Duplikált sorok eltávolítása
data_df = data_df.drop_duplicates()

# Adatok rendezése 'age' szerint
data_df = data_df.sort_values(by='age', ascending=True)

# Adatok szűrése, ahol a 'age' oszlop értéke 25 feletti
young_adults = data_df[data_df['age'] > 25]
print("Szűrt adatok:\n", young_adults.head())
```

Összegzés

Az adatok előfeldolgozása biztosítja, hogy az adatok tiszták, konzisztens formátumúak és előkészítettek legyenek a további feldolgozásra és elemzésre.

Adatmanipulációk

Az adatmanipuláció a Pandas egyik legerősebb funkciója, amely lehetővé teszi az adatok hatékony feldolgozását, szűrését, rendezését és aggregálását. Ezeket a funkciókat alkalmazzuk az adatok előkészítésében és az elemzési folyamatban.

Szűrés:

Az adatok szűrése lehetővé teszi, hogy csak a megadott feltételnek megfelelő sorokat tartsuk meg egy új DataFrame-ben.

30 év feletti diákok:

	name	age	city
1	Béla	34	Debrecen

```
import pandas as pd
# Minta DataFrame létrehozása
data = {
    ,name': ['Anna', 'Béla', 'Cecília', 'Dávid', 'Emma'],
    ,age': [29, 34, 22, 30, 25],
    ,city': ['Budapest', 'Debrecen', 'Szeged', 'Pécs', 'Miskolc']
}
data_df = pd.DataFrame(data)
# Szűrés, ahol a ,age' oszlop értéke 30 év felett van
elder_students = students_df[students_df[,age'] > 30]
print("30 év feletti diákok:\n", elder_students)
```

Adatmanipulációk

Rendezés:

A rendezés lehetővé teszi az adatok sorainak újra sorrendezését egy vagy több oszlop alapján.

Rendezett diákok (kor szerint növekvő sorrendben):

	name	age	city
2	Cecília	22	Szeged
4	Emma	25	Miskolc
0	Anna	29	Budapest
3	Dávid	30	Pécs
1	Béla	34	Debrecen

```
# Adatok rendezése ,age' szerint növekvő sorrendben
```

```
sorted_students = students_df.sort_values(by='age')
```

```
print("Rendezett diákok (kor szerint növekvő sorrendben):\n",  
sorted_students)
```

```
# Adatok rendezése ,name' szerint csökkenő sorrendben
```

```
sorted_students_name = students_df.sort_values(by='name', ascending=False)
```

```
print("Rendezett diákok (név szerint csökkenő sorrendben):\n",  
sorted_students_name)
```

Adatmanipulációk

Aggregálás:

Az aggregálás lehetővé teszi az **adatok összesítését** különböző metrikák segítségével, például **átlag**, **összeg**, **minimum**, **maximum**.

```
# Adatok aggregálása az 'age' átlagának kiszámításával
avg_age = students_df['age'].mean()
print("Átlagos életkor:", avg_age)

# Adatok csoportosítása város szerint és a kor átlagának
kiszámítása
avg_age_by_city = students_df.groupby('city')['age'].mean()
print("Városi átlagos életkor:\n", avg_age_by_city)
```

```
Átlagos életkor: 28.0
Városi átlagos életkor:
város
Budapest    29.0
Debrecen    34.0
Miskolc     25.0
Pécs        30.0
Szeged      22.0
Name: kor, dtype: float64
```

Adatmanipulációk

Összetett példa

A szűrés, rendezés és aggregálás kombinálható a mélyebb elemzés érdekében.

```
# Adatok szűrése, ahol a 'age' nagyobb 25-nél, majd rendezés 'name' szerint
filtered_sorted_students = students_df[students_df['age'] > 25].sort_values(by='name')

print("Szűrt és rendezett diákok:\n", filtered_sorted_students)

# Csoportosítás város szerint, majd a diákok számának megszámlálása
city_student_number = students_df.groupby('city').size()

print("Diákok száma városonként:\n", city_student_number)
```

Szűrt és rendezett diákok:

	name	age	city
0	Anna	29	Budapest
1	Béla	34	Debrecen
3	Dávid	30	Pécs

Diákok száma városonként:

city	
Budapest	1
Debrecen	1
Miskolc	1
Pécs	1
Szeged	1

Adatfeldolgozási feladatok és példák

Az adatfeldolgozási feladatok során különböző műveleteket végezhetünk a **Pandas** könyvtár segítségével, amelyek segítségével az adatokat **előkészíthetjük, módosíthatjuk**, és értékes **információkat nyerhetünk ki** belőlük.

Adattranszformációk

Az adatok transzformációja során például új oszlopok hozzáadását és a meglévő oszlopok értékeinek módosítását végezzük el.

Adafeldolgozási feladatok és példák

Példa: Új oszlop létrehozása

	név	kor	város	kor_kétszer
0	Anna	29	Budapest	58
1	Béla	34	Debrecen	68
2	Cecília	22	Szeged	44
3	Dávid	30	Pécs	60
4	Emma	25	Miskolc	50

```
import pandas as pd
# Minta DataFrame létrehozása
data = {
    ,name': ['Anna', 'Béla', 'Cecília', 'Dávid', 'Emma'],
    ,age': [29, 34, 22, 30, 25],
    ,city': ['Budapest', 'Debrecen', 'Szeged', 'Pécs',
'Miskolc']
}
students_df = pd.DataFrame(data)
# Új oszlop hozzáadása, amely a kor kétszerese
students_df[,age_multiplied'] = students_df[,age'] * 2
print("Új oszloppal bővített DataFrame:\n", students_df)
```


Adafeldolgozási feladatok és példák

Adatok normalizálása

Az adatok normalizálása során az értékeket egy közös skálára helyezzük, ami segíthet az adatok összehasonlíthatóságában.

Példa: Értékek skálázása 0 és 1 közé

```
students_df[,'age_normalized'] = (students_df[,'age'] - students_df[,'age'].min()) /  
(students_df[,'age'].max() - students_df[,'age'].min())  
print("Normalizált 'kor' oszlop:\n", students_df)
```

Eredmény:

	név	kor	város	kor_kétszer	kor_normalizált
0	Anna	29	Budapest	58	0.636364
1	Béla	34	Debrecen	68	1.000000
2	Cecília	22	Szeged	44	0.000000
3	Dávid	30	Pécs	60	0.727273
4	Emma	25	Miskolc	50	0.272727

Adafeldolgozási feladatok és példák

Feltételes adatmanipuláció

Az adatok manipulációja során gyakran szükséges egyes sorokat vagy oszlopokat feltételek alapján módosítani.

Példa: Új oszlop
létrehozása feltételes
értékekkel

```
# Új oszlop, amely megjelöli, ha a kor 30 év felett van
students_df[,'above_30'] = students_df[,'age'].apply(lambda x: 'Igen' if x > 30 else 'Nem')
print("Feltételes oszloppal bővített DataFrame:\n", students_df)
```

	név	kor	város	kor_kétszer	kor_normalizált	30_fölött
0	Anna	29	Budapest	58	0.636364	Nem
1	Béla	34	Debrecen	68	1.000000	Igen
2	Cecília	22	Szeged	44	0.000000	Nem
3	Dávid	30	Pécs	60	0.727273	Nem
4	Emma	25	Miskolc	50	0.272727	Nem

Adafeldolgozási feladatok és példák

Adatok összekapcsolása és egyesítése

Az adatok összekapcsolása több forrásból származó adatok egyesítését jelenti, például **merge**, **join** vagy **concat** használatával.

Adafeldolgozási feladatok és példák

Példa: DataFrame-ek egyesítése

```
# Második DataFrame létrehozása
subjects_df = pd.DataFrame({
    ,name': ['Anna', 'Béla', 'Cecília', 'Dávid', 'Emma'],
    ,subject': ['Matematika', 'Fizika', 'Kémia', 'Biológia', 'Irodalom']
})

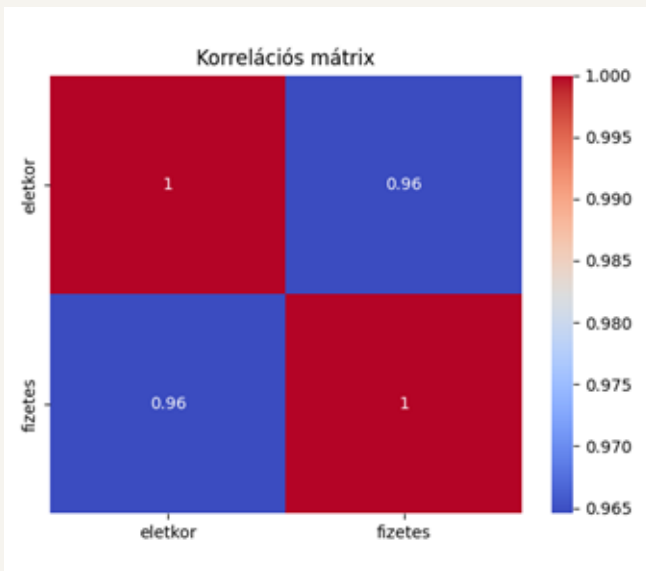
# DataFrame-ek egyesítése a ,name' oszlop alapján
connected_df = pd.merge(students_df, subjects_df, on='név')
print("Összekapcsolt DataFrame:\n", connected_df)
```

	név	kor	város	kor_kétszer	kor_normalizált	30_fölött	tantárgy
0	Anna	29	Budapest	58	0.636364	Nem	Matematika
1	Béla	34	Debrecen	68	1.000000	Igen	Fizika
2	Cecília	22	Szeged	44	0.000000	Nem	Kémia
3	Dávid	30	Pécs	60	0.727273	Nem	Biológia
4	Emma	25	Miskolc	50	0.272727	Nem	Irodalom

Példa: Adatok korrelációjának vizsgálata

Korrelációs mátrix:

	eletkor	fizetes
eletkor	1.000000	0.964549
fizetes	0.964549	1.000000



```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

data = {
    'age': [23, 45, 34, 25, 32, 40, 29, 48, 37, 22],
    'salary': [250, 500, 300, 260, 320, 480, 290, 520, 410, 240]
}

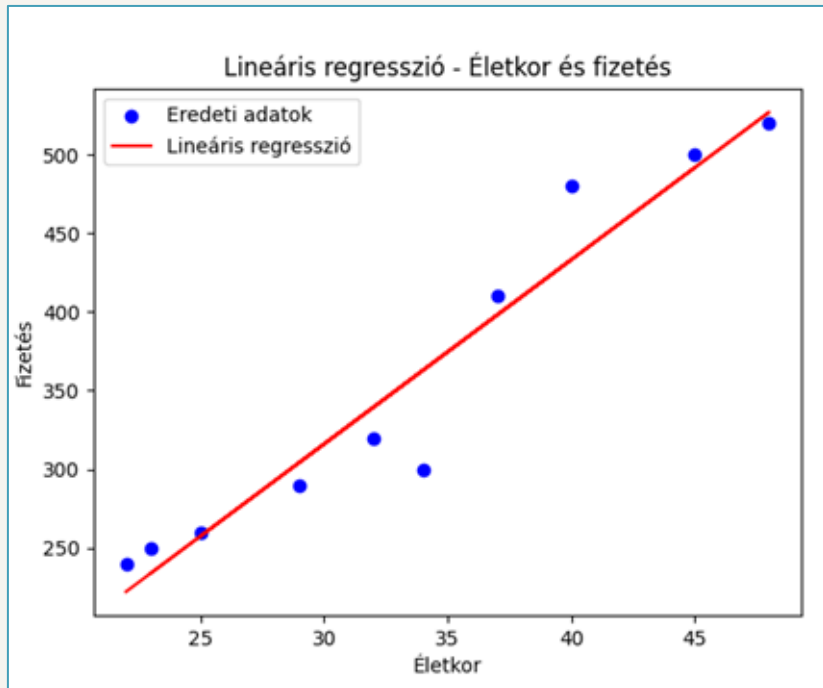
data_df = pd.DataFrame(data)

# Korrelációs mátrix kiszámítása
correlation = data_df.corr()

print("\nKorrelációs mátrix:")
print(correlation)

# Korrelációs mátrix megjelenítése
sns.heatmap(correlation, annot=True, cmap='coolwarm')
plt.title('Korrelációs mátrix')
plt.show()
```

Példa: Lineáris regresszió végrehajtása



$$Y = 11.73 * X + -35.92$$

```
from sklearn.linear_model import LinearRegression
```

```
# Független (x) és függő (y) változók kiválasztása
```

```
x = data_df[['age']]
```

```
y = data_df['age']
```

```
# Lineáris regressziós modell létrehozása
```

```
modell = LinearRegression()
```

```
modell.fit(x, y)
```

```
# Egyenes egyenletének kiírása
```

```
print(f'Y = {modell.coef_[0]:.2f} * X + {modell.intercept_:.2f}')
```

```
# Előrejelzés az adatokra
```

```
y_eforejelzes = modell.predict(x)
```

```
# Diagram rajzolása
```

```
plt.scatter(adat_df['age'], data_df['salary'], color='blue', label='Eredeti adatok')
```

```
plt.plot(data_df['age'], y_forecast, color='red', label='Lineáris regresszió')
```

```
plt.xlabel('age')
```

```
plt.ylabel('salary')
```

```
plt.title('Lineáris regresszió - Életkor és fizetés')
```

```
plt.legend()
```

```
plt.show()
```

Big Data példák különböző területekről

Keresőmotorok – Google

- Napi több milliárd keresés feldolgozása
- Big Data technológiák: Hadoop, Spark, saját rendszerek
- Alkalmazások: webindexelés, keresési relevancia, gépi tanulás

E-kereskedelem – Amazon

- Vásárlói szokások elemzése és ajánlások
- Big Data technológiák: DynamoDB, Amazon ML, Hadoop, Spark
- Alkalmazások: termékajánlás, ügyfélszegmentáció, árképzés

Big Data példák különböző területekről

Pénzügyi szektor – Hitelkártya-csalások

- Valós idejű tranzakciók elemzése csalások felderítésére
- Big Data technológiák: Apache Kafka, Apache Flink, gépi tanulás
- Alkalmazások: kockázatelemzés, csalásellenes rendszerek

Egészségügy – Járványok előrejelzése

- Adatok gyűjtése kórházi nyilvántartásokból és szenzorokból
- Big Data technológiák: Cloudera, Hadoop, gépi tanulás
- Alkalmazások: betegségmodellezés, állapotfigyelés, gyógyszerkutatás

Big Data példák különböző területekről

Közösségi média – Facebook, Twitter

- Felhasználói tevékenységek adatainak feldolgozása
- Big Data technológiák: Hadoop, Spark, Cassandra, Storm
- Alkalmazások: sentiment analízis, trendfigyelés, hirdetéscélzás

Közlekedési szektor – Forgalomfigyelő rendszerek

- GPS, mobil adatok és közlekedési kamerák elemzése
- Big Data technológiák: Hadoop, Spark, IoT-platformok
- Alkalmazások: valós idejű forgalom-előrejelzés, útvonal-optimalizálás

Big Data példák különböző területekről

Energiaipar – Intelligens mérési rendszerek

- Energiafogyasztási adatok gyűjtése és elemzése
- Big Data technológiák: Hadoop, Spark, IoT eszközök
- Alkalmazások: energiaeelosztás optimalizálása, anomáliaészlelés

Klasszikus példa – A Titanic-projekt

Ez egy adatfeldolgozási gyakorlat valós adatállományon a **Big Data** és **gépi tanulás** területén. A **Titanic katasztrófa adata** a Kaggle platformon is elérhető, és tartalmazza az utasok adatait, például:

A feladat során az adatok elemzésével válaszokat kaphatunk arra, hogy milyen tényezők növelték az utasok túlélési esélyeit.

- **Név** (name)
- **Osztály** (pclass)
- **Nem** (sex)
- **Életkor** (age)
- **Testvérek/házastársak száma** a fedélzeten (sibsp)
- **Szülők/gyermek száma** a fedélzeten (parch)
- **Jegy ára** (fare)
- **Beszállás kikötője** (embarked)
- **Túlélés** (survived, ahol 1 a túlélőket, 0 a nem túlélőket jelenti)

Klasszikus példa – A Titanic-projekt

Adatfeldolgozási lépések a Titanic-projekthez:

1. Adatok beolvasása és előfeldolgozása:

- Hiányzó adatok kezelése (pl. életkor kitöltése az átlaggal).
- Kategóriák numerikus értékekké alakítása (pl. a „férfi” és „nő” szöveges adatok 0-ra és 1-re cserélése).

2. Adatok elemzése:

- Alapvető statisztikai elemzések készítése.
- Különböző tényezők (pl. osztály, nem) hatásának vizsgálata a túlélési arányra.

Klasszikus példa – A Titanic-projekt

3. Vizualizáció:

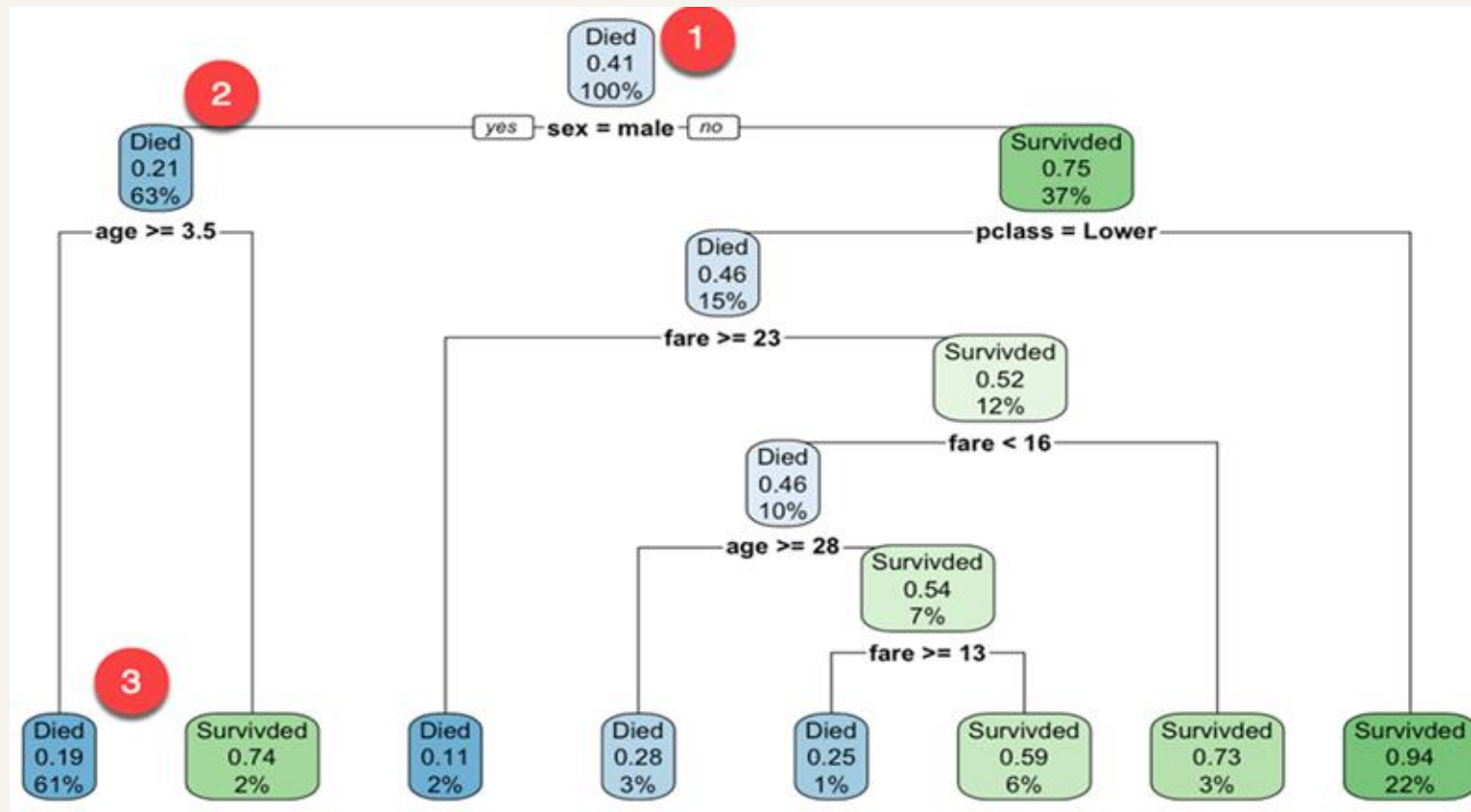
- Diagramok készítése a túlélési arányok bemutatására különböző csoportokban.
- Heatmap-ek és diagramok a jellemzők közötti korrelációk megjelenítésére.

4. Gépi tanulási modellek alkalmazása:

- Egyszerű algoritmusok, például döntési fa használata a túlélési esélyek meghatározására.
- A modell kiértékelése különböző metrikákkal.

Klasszikus példa – A Titanic-projekt

Titanic katasztrófa döntési fa



Köszönöm a
figyelmet!

ELTE IK



Python kurzus – 9. előadás

Tárgyfelelős: Dr. Tejfel Máté

Előadó: Dr. Király Roland

ELTE IK



Large Language Models

1. LLM-ek általában
2. LLama és Ollama keretrendszerek
3. LLM-ek használata parancssorból, saját gépen
4. Onlie LLM - Collab
5. REST API használta LLM, cahtbot
6. Webes keretrendszerek
7. Felhasználási területek, kutatás

A prezentációhoz tartozó forráskódok

[main.py](#) (pdf állomány megnyitása)

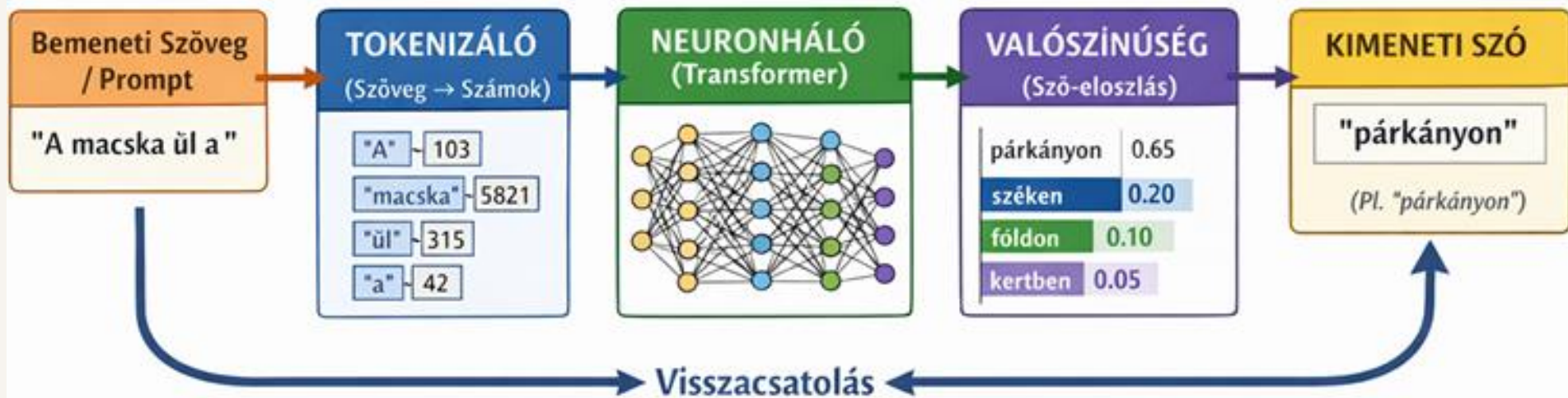
[modellsources.py](#) (pdf állomány megnyitása)

[ragmodell.py](#) (pdf állomány megnyitása)

[similarity_functions.py](#) (pdf állomány megnyitása)

[txtconverter.py](#) (pdf állomány megnyitása)

Nyelvi Modellek - LLM

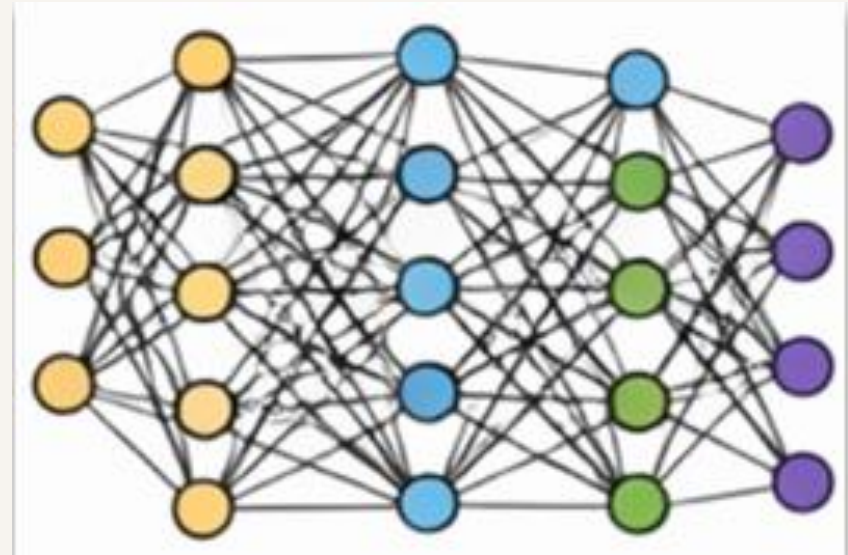


LLM

- Bemenet (Input/Prompt): A felhasználó által megadott szöveg.
- Tokenizáló (Tokenization): A szöveget a gép számára érthető kisebb egységekre, "tokenekre" (szavakra vagy szótagokra) bontja.
- Neurális háló (Transformer): Ez a modell "szíve", amely megtanulta a hatalmas adathalmazból a szavak közötti összefüggéseket.
- Valószínűségi eloszlás: A modell kiszámítja, hogy az adott kontextusban melyik szó következzen, és mindegyikhez valószínűségi értéket rendel.
- Kimeneti szó: A modell a legmagasabb valószínűségű szót választja, majd ezt a folyamatot ismétli (visszacsatolja), amíg a mondat el nem készül.

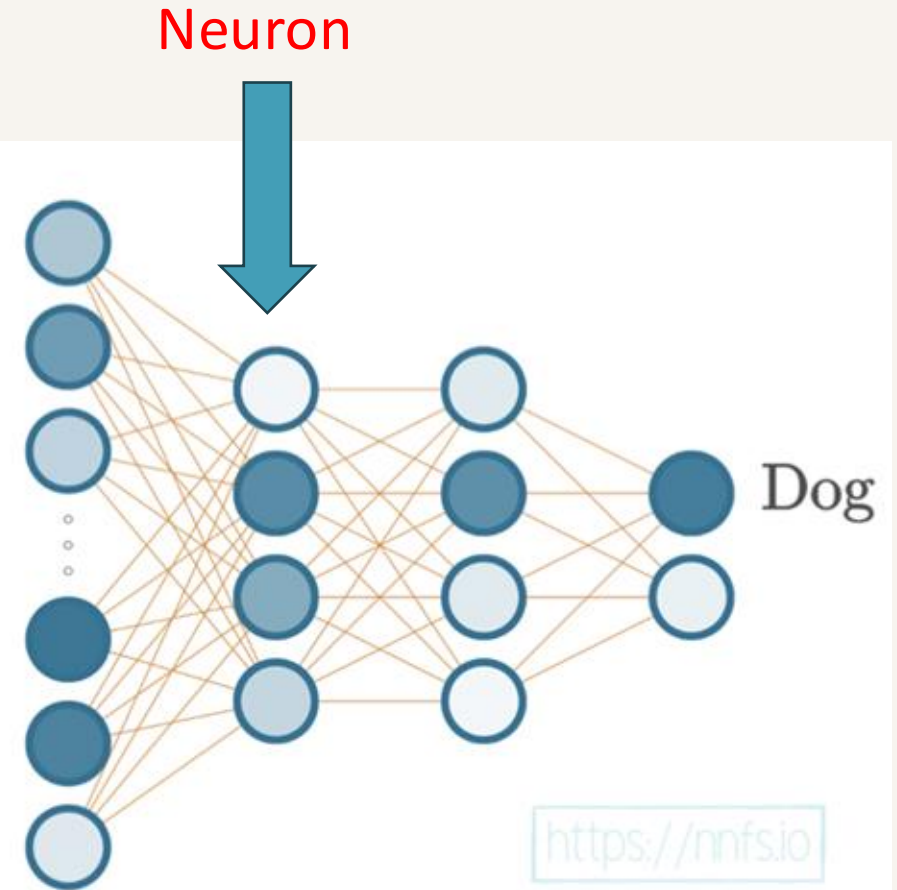
Neurális háló - Pythonban

Neurális háló Megtanítható az adathalmazokban található szavak közötti összefüggések használatára.



Neurális hálók - a cél:

A kiszámolt eredmény és az elvárt érték közötti különbség minimalizálása. Veszteség kiszámítása, majd a súlyok módosítása.



Egyszerű neuron (perceptron)

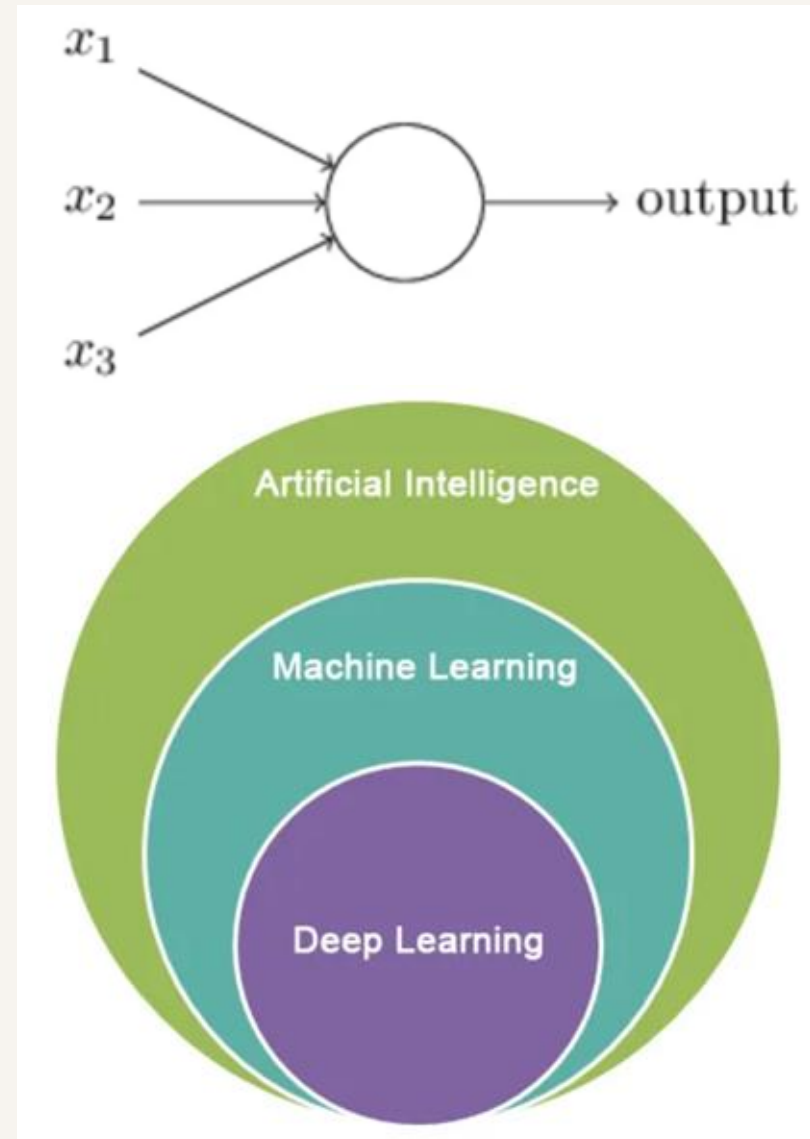
Pl.:

```
inputs = [1.0, 2.0, 3.0, 2.5]
```

```
weights = [0.2, 0.8, -0.5, 1.0]
```

```
bias = 2.0
```

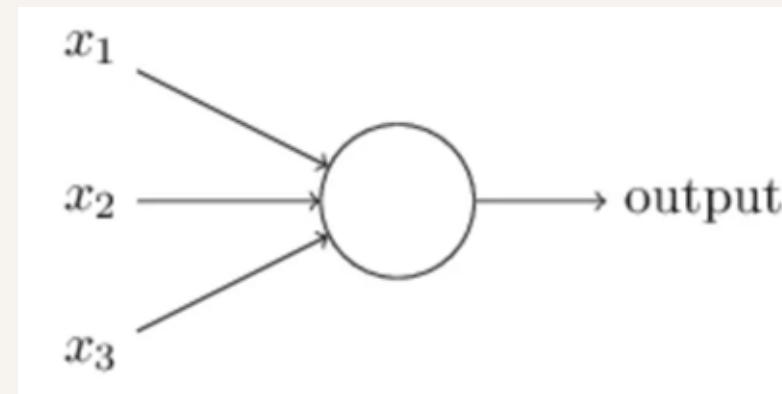
(VSCode – single.py)



Mesterséges neuron és neurális háló

- Weight: A neuron minden bemenetéhez (x) tartozik egy súly (w), amelyet a többi bemenethez viszonyított relatív fontossága alapján rendelünk hozzá.
- A neuron úgy működik, hogy ha a bemenetek súlyozott összege nagyobb, mint egy bizonyos küszöbérték, akkor 1 kimenetet ad, egyébként 0-t. Ez a neuron matematikai modellje (amit perceptronnak is neveznek).
- Bias: egy konstans, amelyet a bemenetek súlyozott összegével együtt a kimenet beállítására használunk, hogy a modell a lehető legjobban illeszkedjen az adott feladathoz.

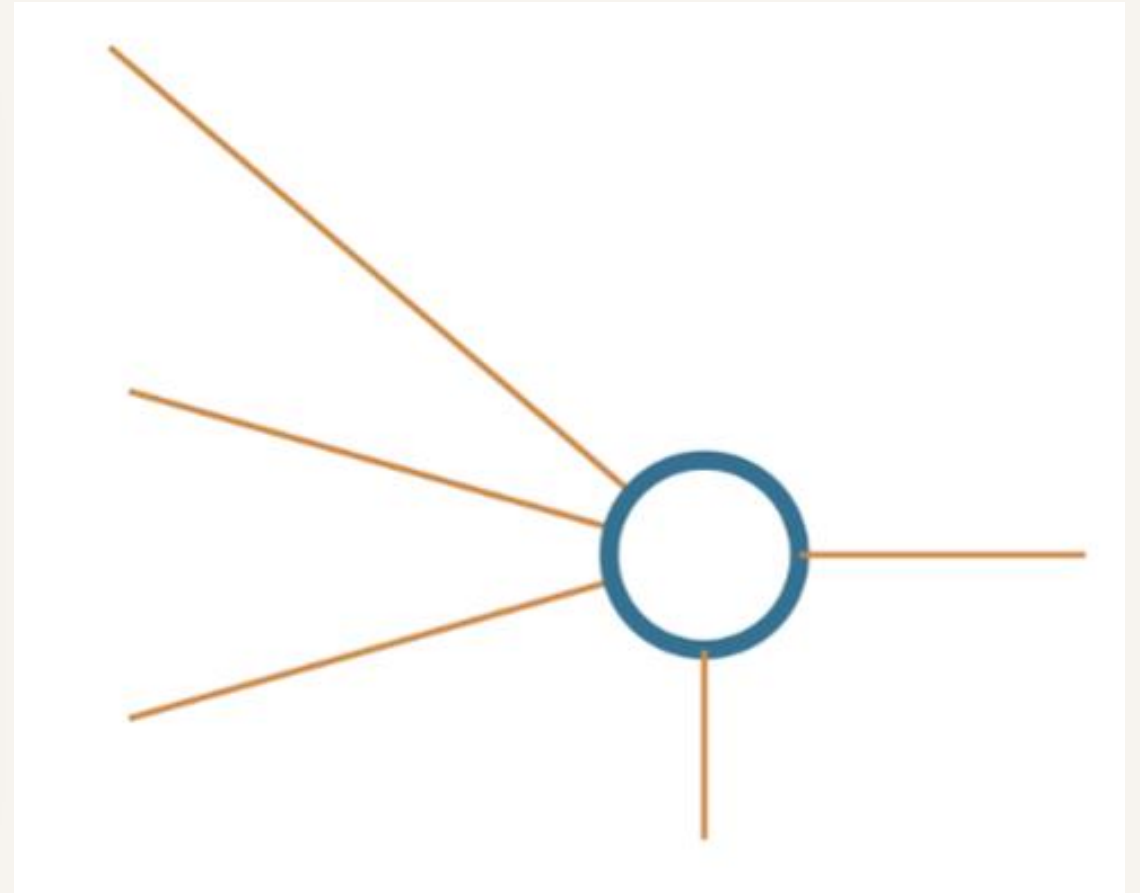
$$Z = W * X + B$$



Eredmény kiszámítása egyetlen neuronra

```
inputs = [1.0, 2.0, 3.0, 2.5]  
weights = [0.2, 0.8, -0.5, 1.0]  
bias = 2.0  
  
outputs = np.dot(weights, inputs) + bias  
  
print(outputs) # -> 4.8
```

VSCode-ban (single.py)



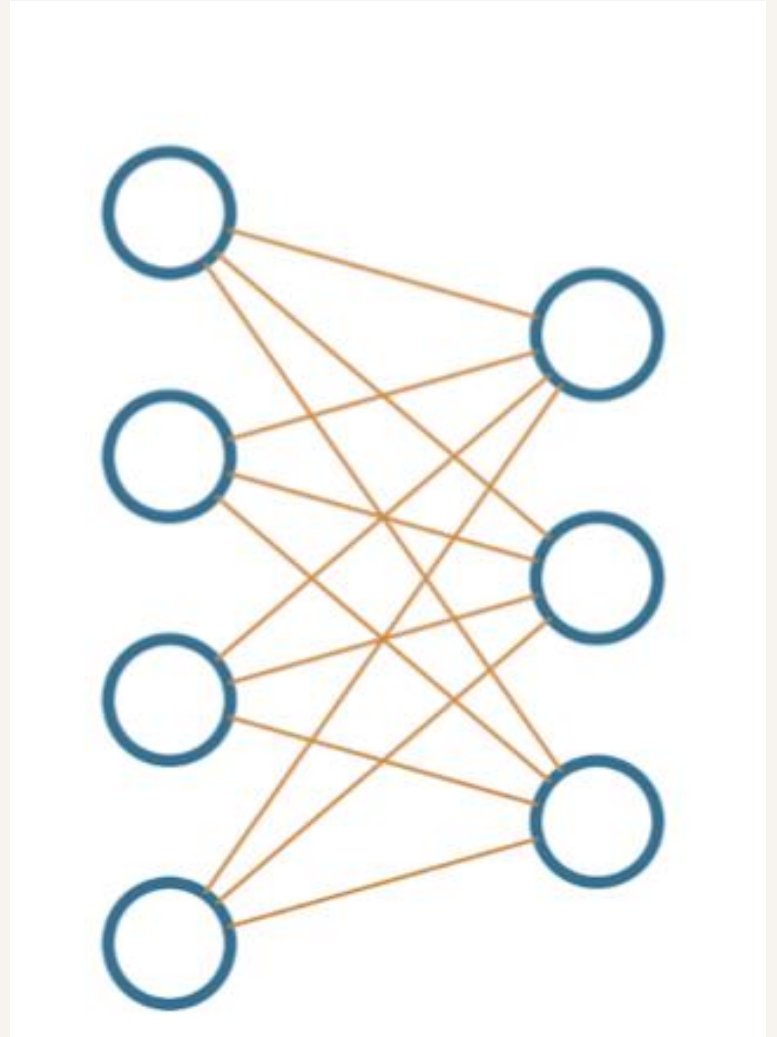
Ugyanez egy layer esetén

```
inputs = [1.0, 2.0, 3.0, 2.5]
weights = [[0.2, 0.8, -0.5, 1],
           [0.5, -0.91, 0.26, -0.5],
           [-0.26, -0.27, 0.17, 0.87]]
biases = [2.0, 3.0, 0.5]

layer_output = np.dot(weights, inputs) +
biases

print(layer_output) # [4.8  1.21  2.385]
```

VSCoDe-ban (layer.py)



Aktiválási függvény

- Az aktiválási függvény célja, hogy (nem)linearitást vezessen be a neuron kimenetébe. Egyetlen számot vesz (input), és valamilyen matematikai műveletet hajt végre rajta.

Pl.:

- Sigmoid: A kimenetet a $[0, 1]$ tartományba képezi le.
- ReLU: Ez a leggyakrabban használt aktiválási függvény (Rectified Linear Unit).
Értéke x , ha x pozitív, egyébként pedig 0 .
- $y = \max(x, 0)$, (pl: $-1 = 0$, $1 = 1$)

$$\frac{1}{1 + e^{-z}}$$

Összefoglalás - neurális háló

- A neurális háló rétegekből áll, minden réteg neuronokból.
- A különböző rétegek között kapcsolat van (jelen esetben teljes).
- Ezek a rétegek úgy alakítják át az adatokat, hogy először kiszámítják a bemenetek súlyozott összegét, majd a neuronokhoz rendelt aktiválási függvények segítségével normalizálják azt.

Tensorflow/keras – neurális hálók tanítása

VSCoDe-ban (tensorflowOK.ipynb) Következő óra

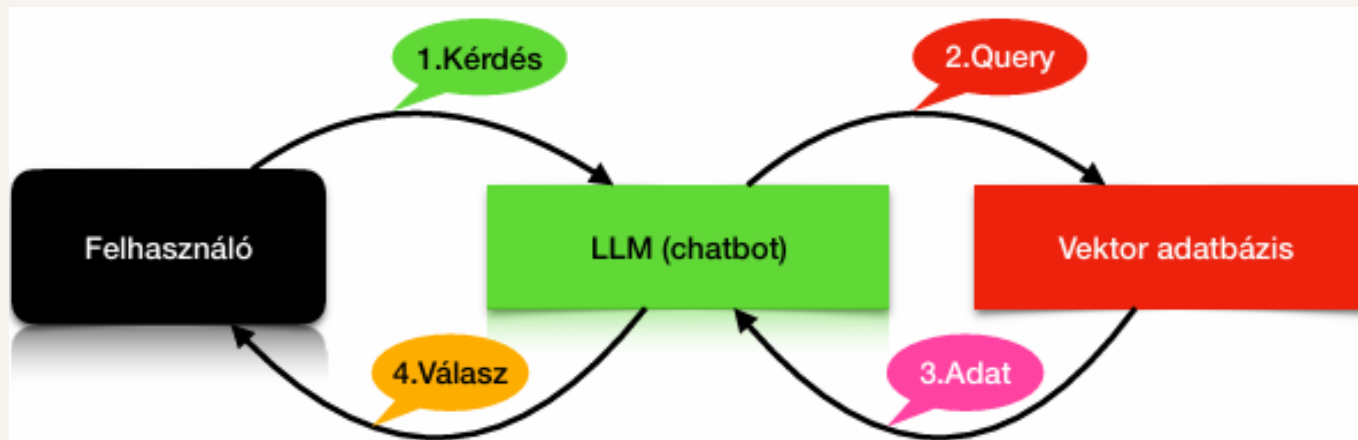
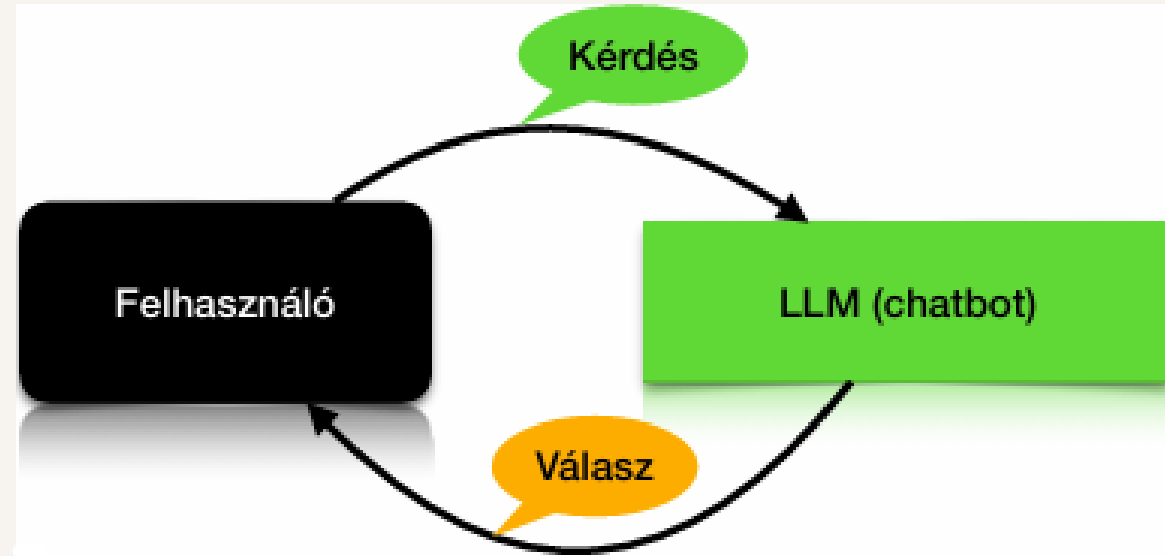
LLM – Llama, API based Ollama, WEB interface

- LLM
- RAG,
- API based Ollama,
- WEB interface - később

Retrieval-Augmented Generation (RAG)

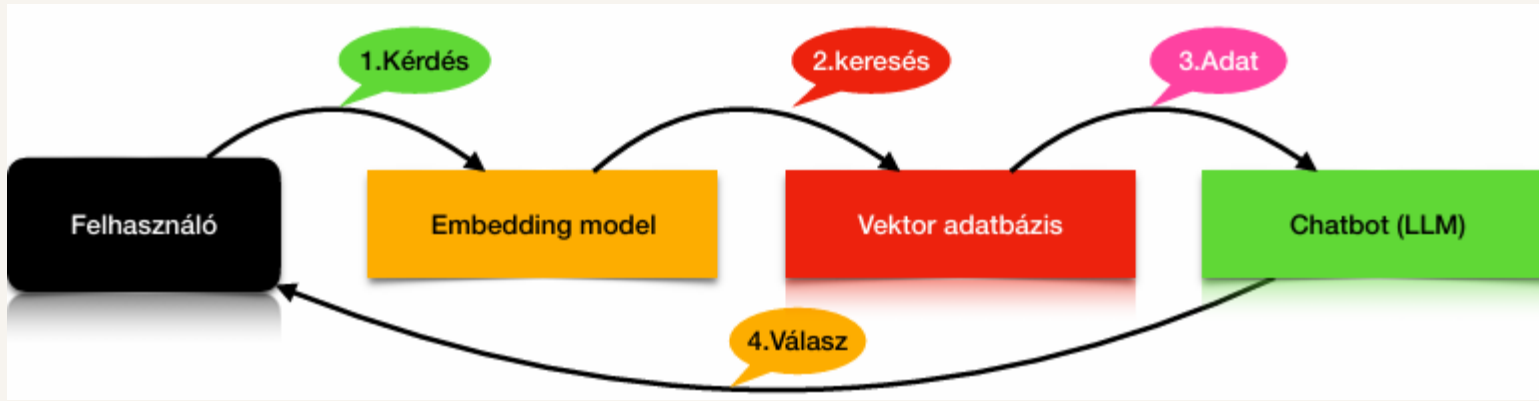
RAG

Retrieval-Augmented Generation



RAG modell

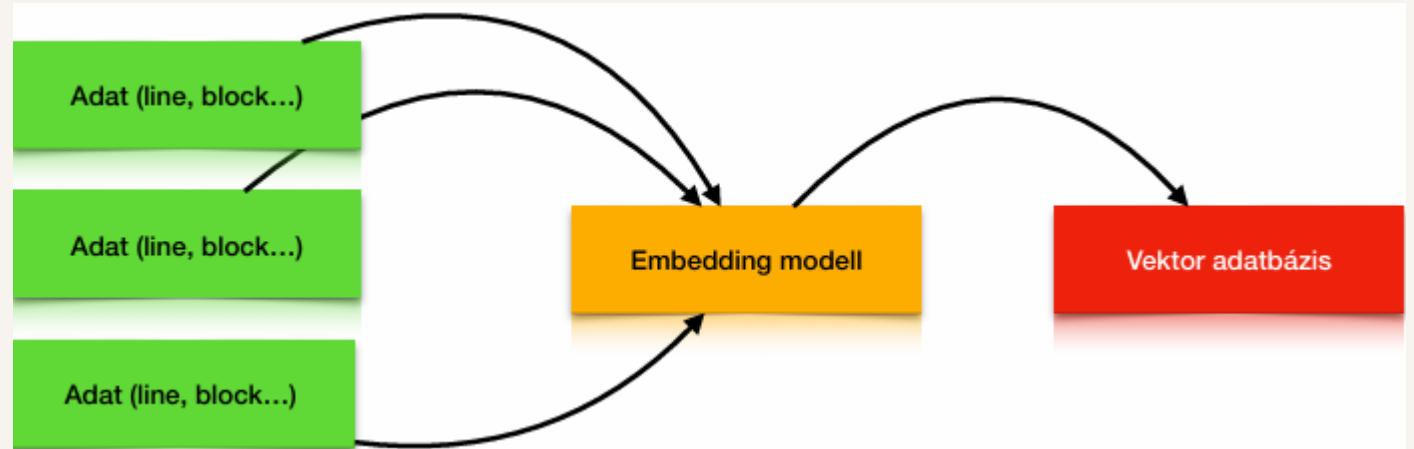
Vektor adatbázis - tudás és a hozzá tartozó beágyazási vektorok tárolására (pl.: pgvector, Qdrant)



Embedding modell - előre betanított nyelvi modell, ami a bemeneti szöveget szemantikai jelentést rögzítő vektorreprezentációkká - ezekben keressük az információt

Indexelés

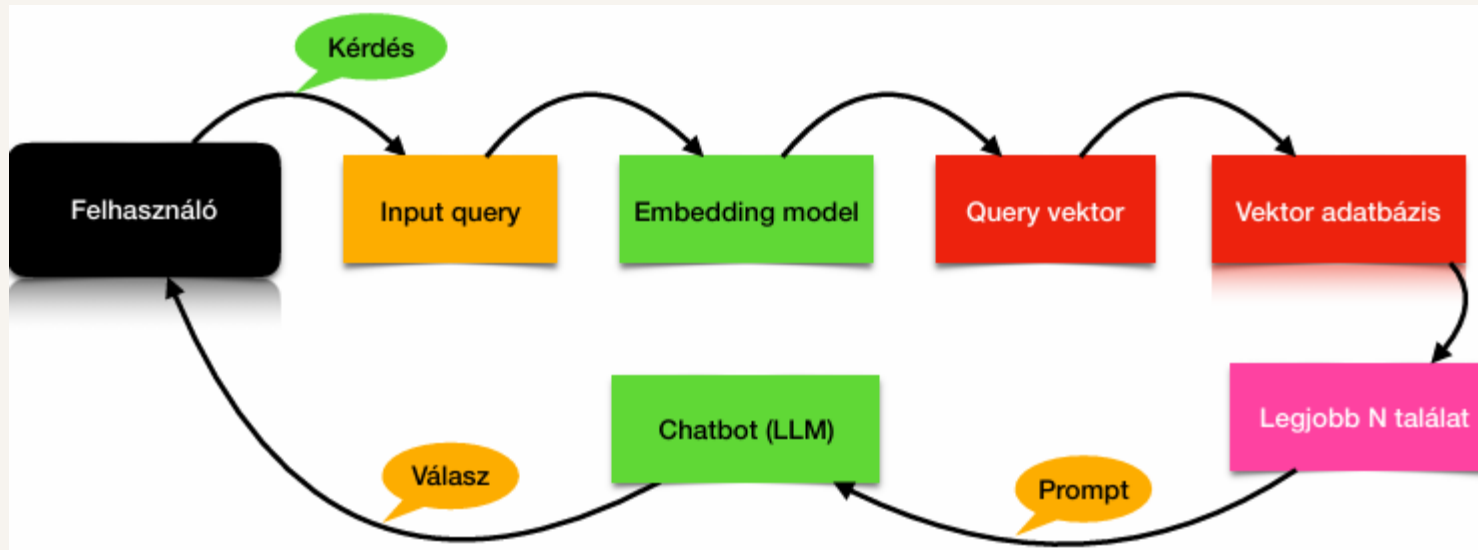
Az adatokat (pl.: sorok) kis darabokra bontjuk, majd minden egyes darabnak kiszámítjuk a vektoros reprezentációját, amely hatékonyan kereshető a válasz generálásához. Ez lehet, sor, bekezdés, nagyobb blokk



Az indexelés után minden egyes adatblokk a hozzá tartozó beágyazási vektorral együtt egy vektoradatbázisban tárolódik. A vektorok felhasználhatók releváns információk lekérésére (mint egy query)

Keresés

A kérdés vektoros reprezentációja alapján a vektor adatbázisban hasonló vektorokra keresünk és a hozzájuk tartozó mondatokat adjuk oda az LLM (chatbot)promptjához.



LLM-ek a RAG modellhez

ollama pull <https://huggingface.co/bartowski/Llama-3.2-1B-Instruct-GGUF>

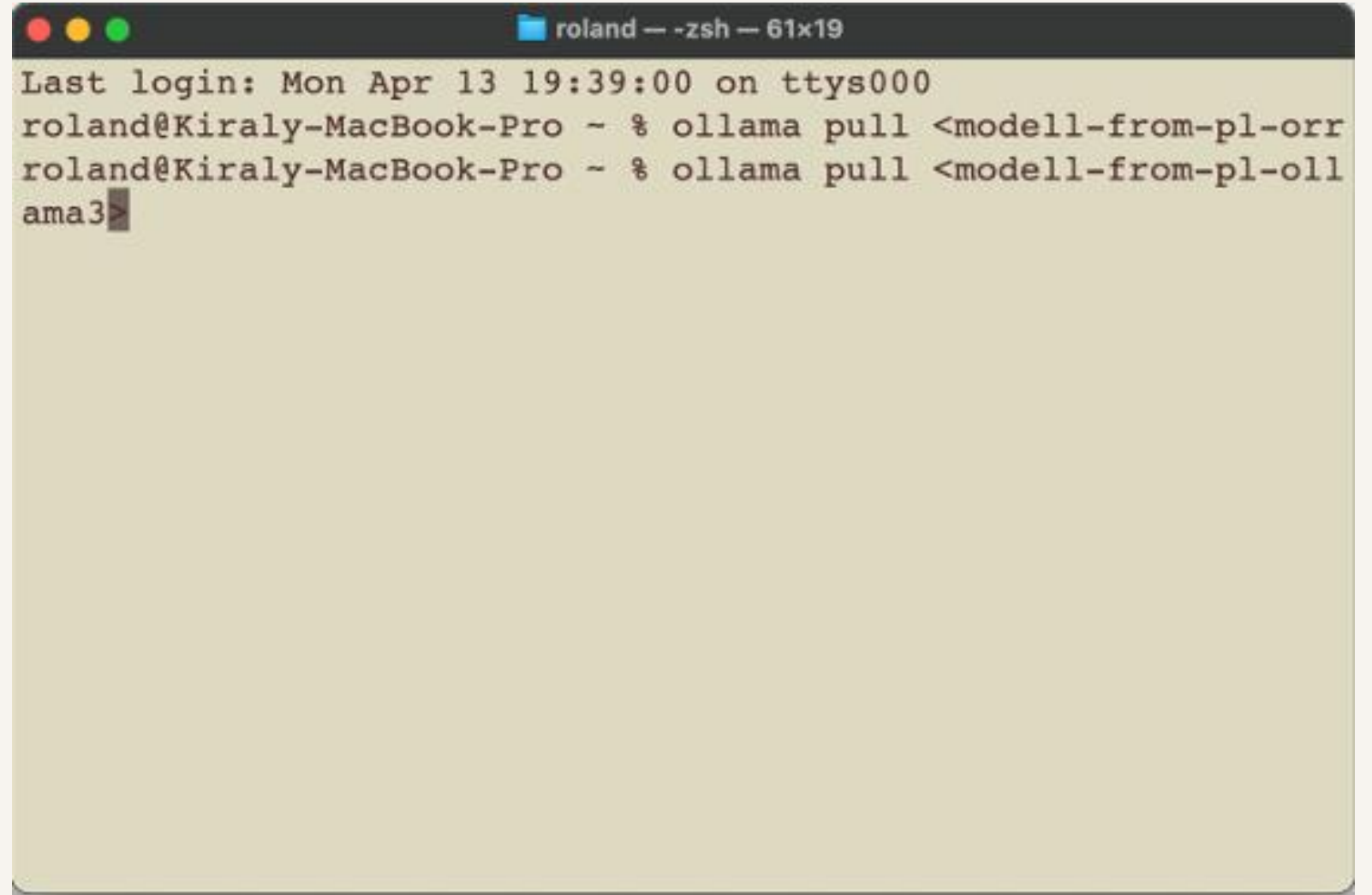
ollama pull <https://huggingface.co/CompendiumLabs/bge-base-en-v1.5-gguf>

A nyelvi modellhez a modell a Hugging Face-ről:

Kisebb modellek: 8GB ram körül, de jobb minőségben

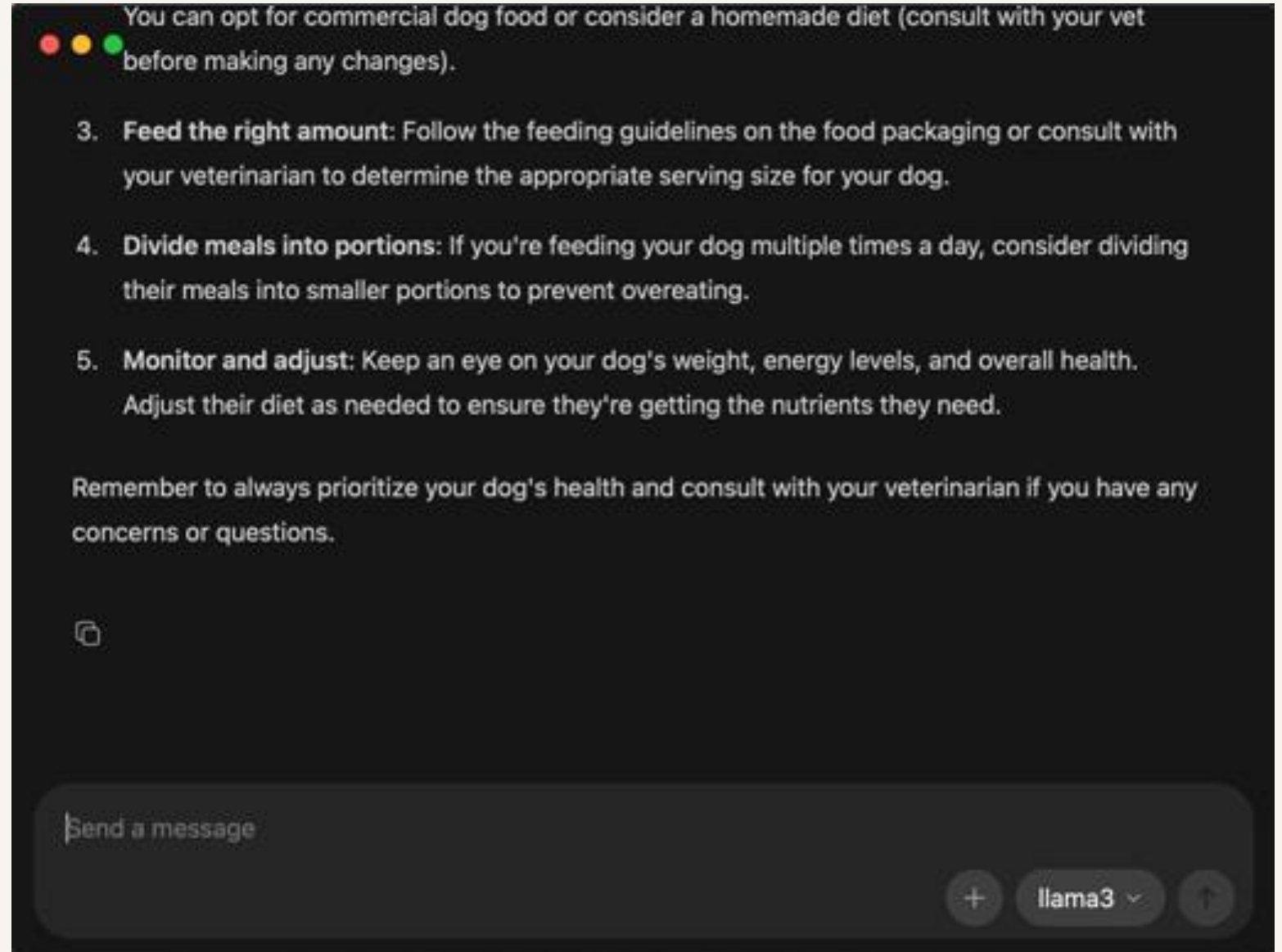
<https://huggingface.co/bartowski/Llama-3.2-3B-Instruct-GGUF>

Ollama

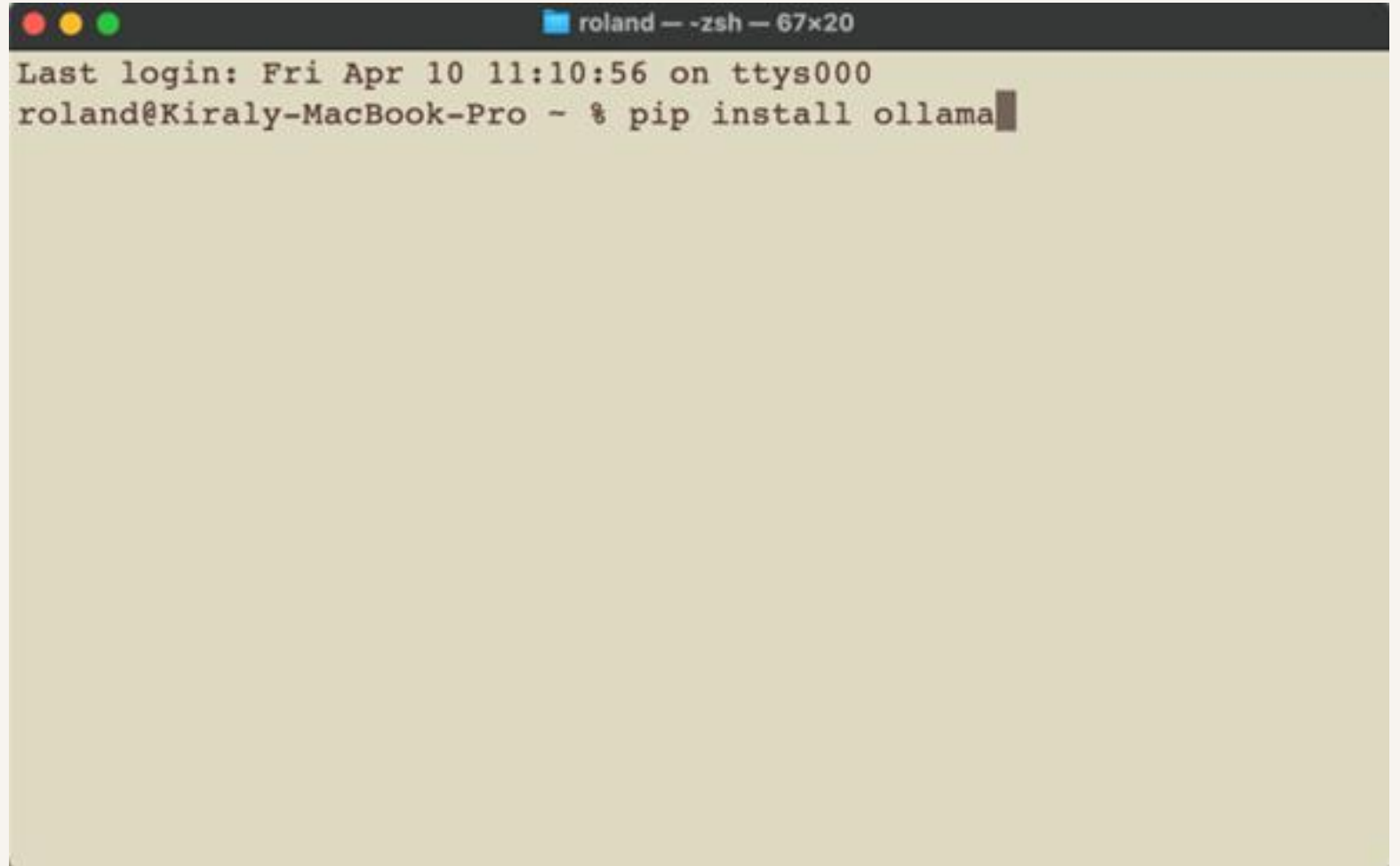
A terminal window with a dark title bar containing the text 'roland — -zsh — 61x19'. The terminal has a light beige background and shows the following text: 'Last login: Mon Apr 13 19:39:00 on ttys000', 'roland@Kiraly-MacBook-Pro ~ % ollama pull <modell-from-pl-orr', 'roland@Kiraly-MacBook-Pro ~ % ollama pull <modell-from-pl-oll', and 'ama3' followed by a cursor. The window has standard macOS window control buttons (red, yellow, green) in the top-left corner.

```
roland — -zsh — 61x19
Last login: Mon Apr 13 19:39:00 on ttys000
roland@Kiraly-MacBook-Pro ~ % ollama pull <modell-from-pl-orr
roland@Kiraly-MacBook-Pro ~ % ollama pull <modell-from-pl-oll
ama3
```

Ollama



Python

A terminal window with a dark title bar containing three colored window control buttons (red, yellow, green) and the text 'roland - zsh - 67x20'. The terminal has a light beige background. It displays the text 'Last login: Fri Apr 10 11:10:56 on ttys000' followed by the prompt 'roland@Kiraly-MacBook-Pro ~ %' and the command 'pip install ollama' which is followed by a black cursor block.

```
roland - zsh - 67x20
Last login: Fri Apr 10 11:10:56 on ttys000
roland@Kiraly-MacBook-Pro ~ % pip install ollama
```

Felhasznált irodalom

- Harrison Kinsley & Daniel Kukiela: Neural Networks from Scratch in Python (2020 Harrison Kinsley)
- Deep Learning for Computer Vision (Stanford - Spring 2025)

Köszönöm a
figyelmet!

ELTE IK



Python kurzus – 10. előadás

Tárgyfelelős: Dr. Tejfel Máté

Előadó: Dr. Király Roland

ELTE IK



Tematika

1. A venv használata (2)
2. Matplotlib alapjai: diagramok készítése
3. Adatok megjelenítése - Pandas
4. Komplex diagramok létrehozása
5. Seaborn bevezetése és használata
6. Diagramok készítése különböző példák alapján

A prezentációhoz tartozó forráskódok

- eacodes (pdf formátumban)
- eacodes_py (.py forráskódok)

Adatvizualizáció bevezetés

- Az adatvizualizáció segíti az adatok megértését és kommunikálását
- Kiemelt szerepe van a döntéshozatal támogatásában és a mintázatok felismerésében

Miért fontos az adatvizualizáció?

- Komplex adatok egyszerűsítése – átláthatóság és értelmezhetőség
- Mintázatok felismerése – trendek, anomáliák azonosítása
- Döntéshozatal támogatása – vizuális információk közvetlen hatása
- Hatékony kommunikáció – információk egyszerű és érthető közvetítése
- Adatok feltárása – összefüggések és ok-okozati viszonyok felfedezése



Adatvizualizáció bevezetés

Példa:

Az ingatlanárak trendvonallal történő becslése fix (korábbi vagy összehasonlító) árak alapján egy adatvezérelt módszer, amely a piaci változások irányát és ütemét vetíti előre.

Lépései:

- adatgyűjtés és fix árak rögzítése,
- adatok tisztítása,
- trendvonal illesztése: Egy grafikonon ábrázolva a pontokat, húzunk egy trendvonalat (lineáris, exponenciális stb.), amely a legjobban közelíti a pontokat.
- Értékbecslés (interpoláció/extrapoláció): A trendvonal egyenlete alapján kiszámítható a keresett ingatlan becsült értéke (grafikus/manuális módszerrel is).

Alapvető diagram típusok

- Trendek figyelése, időbeli változások bemutatása – vonaldiagram
- Adatok eloszlása – hisztogram, eloszlásgrafikon
- Kapcsolatok azonosítása – scatter plot
- Kiugró értékek felismerése – box plot, scatter plot
- Adatok csoportos különbségei – oszlopdiagram, sávdiaagram
- Összegzés és részek megoszlása – kördiagram, halmozott diagram



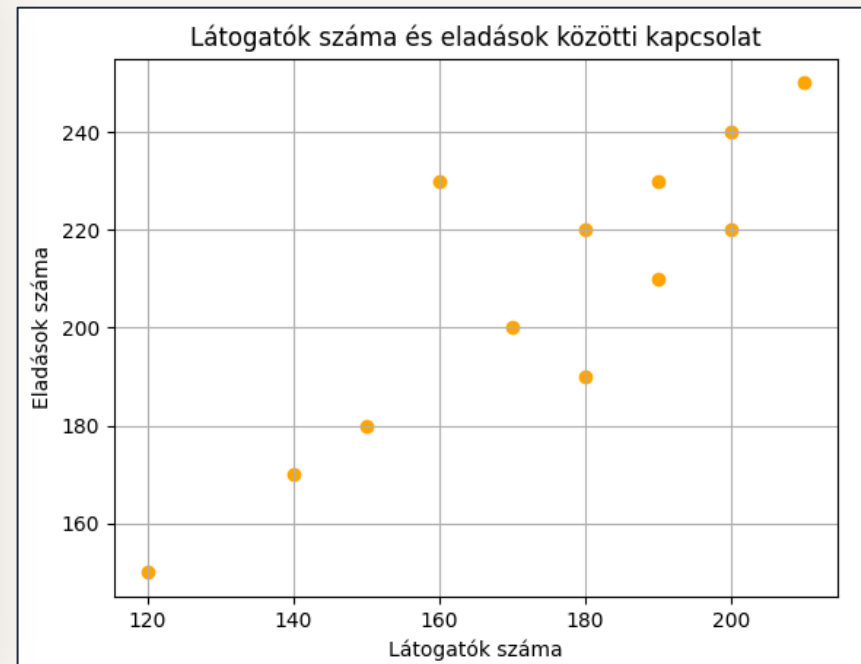
1. A Matplotlib alapjai: diagramok készítése

- A Matplotlib egy Python könyvtár, az adatvizualizáció egyik eszköztára
- Vonaldiagramok, hisztogramok, scatter plotok és más grafikonok készítésére használjuk
- Jól integrálható más adatfeldolgozó könyvtárakkal, mint a Pandas és a NumPy
- Telepítés: **`pip(3) install matplotlib`**

Scatter plotok - a korrelációk vizsgálatához

```
import matplotlib.pyplot as plt
latogatok = [120, 150, 180, 200, 170, 160, 140, 180, 190, 210, 190, 200]
eladasok = [150, 180, 220, 240, 200, 230, 170, 190, 210, 250, 230, 220]

plt.scatter(latogatok, eladasok, color='orange')
plt.title('Látogatók száma és eladások közötti kapcsolat')
plt.xlabel('Látogatók száma')
plt.ylabel('Eladások száma')
plt.grid(True)
plt.show()
```



A **plt.scatter()** hozza létre a diagramot.

A **plt.grid(True)** jeleníti meg a rajzterület vezetőrácsait.

Scatter plot diagram rajzolása a `plt.plot()` függvénnnyel

```
import matplotlib.pyplot as plt

latogatok = [120, 150, 180, 200, 170, 160, 140, 180, 190, 210, 190, 200]
eladasok = [150, 180, 220, 240, 200, 230, 170, 190, 210, 250, 230, 220]

plt.plot(latogatok, eladasok, color='orange', marker='o', linestyle='None')
plt.title('Látogatók száma és eladások közötti kapcsolat')
plt.xlabel('Látogatók száma')
plt.ylabel('Eladások száma')
plt.grid(True)
plt.show()
```

[scatter.py \(.pdf\)](#)

A `plt.plot()` függvény **`linestyle='None'`** beállítása hozza létre a scatter diagramot.
A **`marker`** argumentum a pontok jelölésére szolgál.

Hozzá lehet adni egy illeszkedő egyenest és kiszámolni a korrelációt.

```
import numpy as np
import matplotlib.pyplot as plt

visitors = [120, 150, 180, 200, 170, 160, 140, 180, 190, 210, 190, 200]
sales = [150, 180, 220, 240, 200, 230, 170, 190, 210, 250, 230, 220]

plt.scatter(visitors, sales, color='orange', alpha=0.8)
m, b = np.polyfit(visitors, sales, 1) # lineáris illesztés
xs = np.array(sorted(visitors))
plt.plot(xs, m*xs + b,
color='blue', linestyle='--', label=f'fit: y={m:.2f}x+{b:.1f}')

r = np.corrcoef(visitors, sales)[0,1] # korreláció
plt.title(f'Látogatók vs eladások (Pearson r = {r:.2f})')
plt.xlabel('Látogatók száma')
plt.ylabel('Eladások száma')
plt.grid(True)
plt.legend()
plt.show()
```

[koorelacio1.py \(.pdf\)](#)

Vonaldiagram készítése

```
import matplotlib.pyplot as plt

hónapok = ['Január', 'Február', 'Március', 'Április', 'Május',
           'Június']
bevételek = [1000, 1200, 1300, 900, 1100, 1500]

plt.plot(hónapok, bevételek, marker='o', linestyle='-', color='b')
plt.title('Cég havi bevételei')
plt.xlabel('Hónapok')
plt.ylabel('Bevétel (ezer Ft)')
plt.grid(True)
plt.show()
```

A vonaldiagram segít bemutatni egy változó időbeli alakulását.

Példa: Készítsünk egy vonaldiagramot, amely egy cég havi bevételeit ábrázolja.

Használat: **plt.plot()** függvény a grafikon létrehozásához.

Tengelyfeliratozás és cím hozzáadása: **plt.xlabel()**, **plt.ylabel()**, **plt.title()**.

A **linestyle** paraméter a vonal stílusát állítja be.

[linedig.py \(.pdf\)](#)

Tengelyek, címek és feliratok testreszabása

Cím és tengelyek formázása:
betűméret, szín, stílus beállítása.

Elforgatás alkalmazása az x tengely feliratain: **plt.xticks(rotation=45)**

Rácsvonalak vezérlése: **plt.grid()**

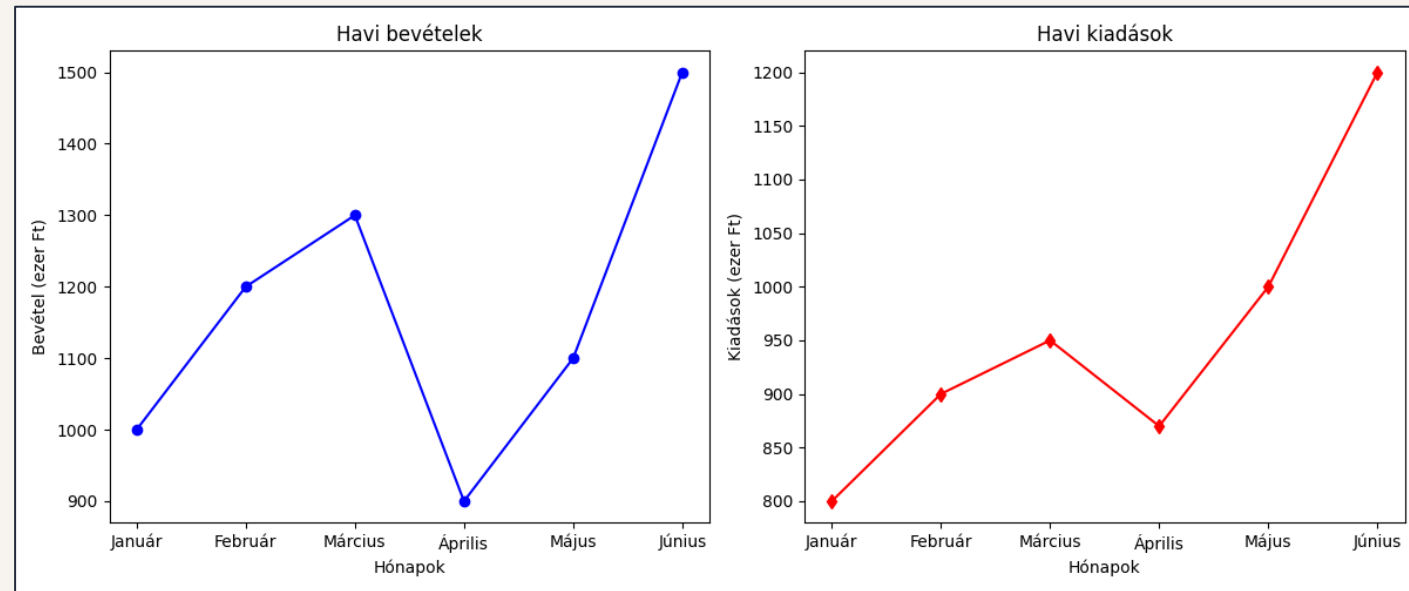
[differentplot.py \(.pdf\)](#)

```
plt.plot(hónapok, bevételek, marker='s', color='g')
plt.title('Cég havi bevételei', fontsize=16, fontweight='bold', color='navy')
plt.xlabel('Hónapok', fontsize=12, color='darkgreen')
plt.ylabel('Bevétel (ezer Ft)', fontsize=12, color='darkgreen')
plt.xticks(rotation=45) # x tengely értékek elforgatása
plt.grid(color='gray', linestyle='--', linewidth=0.5)
plt.show()
```

Többszörös diagramok egy ábrán belül (subplotok)

Példa:

Ábrázoljunk egyszerre két különböző diagramot egy ablakban – az egyik a bevételeket, a másik a kiadásokat mutatja.



Többszörös diagramok egy ábrán belül (subplotok)

A **plt.subplot()** függvény
használata:

Paraméterek: sorok, oszlopok
száma és a subplot pozíciója.

A **plt.tight_layout()** az ábrák
közötti térközöket automatikus
optimalizálja.

[subplot.py \(.pdf\)](#)

```
plt.figure(figsize=(12, 5))  
# Első subplot  
plt.subplot(1, 2, 1) # 1 sor, 2 oszlop, 1. subplot  
plt.plot(hónapok, bevételek, color='blue', marker='o')  
plt.title('Havi bevételek')  
plt.xlabel('Hónapok')  
plt.ylabel('Bevétel (ezer Ft)')  
  
# Második subplot  
plt.subplot(1, 2, 2) # 1 sor, 2 oszlop, 2. subplot  
plt.plot(hónapok, kiadások, color='red', marker='d')  
plt.title('Havi kiadások')  
plt.xlabel('Hónapok')  
plt.ylabel('Kiadások (ezer Ft)')  
plt.tight_layout() # Igazítás a helyes megjelenítéshez  
plt.show()
```

Példa

3 db subplot diagram egy ábrában

[moresubplot.py \(.pdf\)](#)

```
hónapok = ['I', 'II', 'III', 'IV', 'V', 'VI', 'VII', 'VIII', 'IX', 'X', 'XI', 'XII']
bevételek = [1000, 1200, 1300, 900, 1100, 1500, 1100, 1200, 1000, 900, 800, 1400]
kiadások = [800, 900, 950, 870, 1000, 1200, 500, 900, 650, 1050, 700, 1000]

plt.figure(figsize=(12, 12))
plt.subplot(2, 2, 1) # 1. diagram
plt.plot(hónapok, bevételek, marker='o', linestyle='-', color='b')
plt.title('Cég havi bevételei')
plt.xlabel('Hónapok')
plt.ylabel('Bevétel (ezer Ft)')
plt.grid(True)
plt.subplot(2, 2, 2) # 2. diagram
plt.plot(hónapok, kiadások, marker='o', linestyle='-', color='r')
plt.title('Cég havi kiadásai')
plt.xlabel('Hónapok')
plt.ylabel('Kiadás (ezer Ft)')
plt.grid(True)
plt.subplot(2, 2, 3) # 3. diagram
plt.scatter(kiadások, bevételek, marker='o', color='b')
plt.title('Kiadások és bevételek összefüggése')
plt.xlabel('Kiadások (ezer Ft)')
plt.ylabel('Bevétel (ezer Ft)')
plt.grid(True)
plt.tight_layout()
plt.show()
```


Hisztogramok és eloszlás-vizsgálatok

A hisztogramok az adatok gyakoriságának bemutatására szolgálnak. Ez segít az adatok szimmetriájának, eloszlásának és anomáliáinak azonosításában.

A **plt.hist()** függvény a hisztogram elkészítésére szolgál.

A **bins** paraméter határozza meg a sávok számát.

Az **edgecolor** a sávok körvonalának színét állítja be a jobb láthatóság érdekében.

[histogram.py \(.pdf\)](#)

```
import matplotlib.pyplot as plt
import pandas as pd

eladasok = [150, 180, 220, 240, 200, 230, 170, 190, 210, 250, 230, 220]

plt.hist(eladasok, bins=6, color='lightblue', edgecolor='black')
plt.title('Napi eladások eloszlása')
plt.xlabel('Eladások száma')
plt.ylabel('Gyakoriság')
plt.grid(axis='y', linestyle='--', linewidth=0.5)
plt.show()
```

Box plotok használata adatok szórásának bemutatására

A box plotok (dobozábrák) vizuálisan mutatják az adatok szórását, középértékét, valamint a kiugró értékeket.

Ezek segítenek gyorsan azonosítani az adatok eloszlását és az esetleges anomáliákat.

Box plotok használata adatok szórásának bemutatására

A **plt.boxplot()** függvény hozza létre a dobozábrát.

A **patch_artist** és **boxprops** paraméterek segítségével testreszabhatjuk a doboz megjelenését.

Az ábra megmutatja a **mediánt**, a **kvartiliseket**, az esetleges kiugró értékeket és a minta terjedelmét.

[boxplot.py \(.pdf\)](#)

```
# Heti eladások adatai
heti_eladasok = [150, 180, 220, 240, 200, 230,
                 170, 190, 210, 250, 230, 220, 180, 195, 205]

# Box plot készítése
plt.boxplot(heti_eladasok, patch_artist=True,
            boxprops=dict(facecolor='lightgreen'))
plt.title('Heti eladások dobozábrája')
plt.ylabel('Eladások száma')
plt.grid(axis='y', linestyle='--', linewidth=0.5)
plt.show()
```

Medián és quartilisek

Medián: egy sorba rendezett adathalmaz középső értéke. Ha páros elemszám van, a két középső érték átlaga.

Quartilisek (Q_1 , Q_2 , Q_3): az adatok három osztópontja, amelyek az adatokat négy egyenlő részre bontják. Q_2 a medián (50%-os percentilis), Q_1 a 25%-os, Q_3 a 75%-os percentilis.

IQR (interquartile range): $Q_3 - Q_1$, az eloszlás középső 50%-ának terjedelme. Az $1.5 \cdot \text{IQR}$ szabály segít kiugró értékeket találni (alsó határ = $Q_1 - 1.5 \cdot \text{IQR}$, felső határ = $Q_3 + 1.5 \cdot \text{IQR}$).

Medián és quartilisek

Példa:

Rendezett sorozat: 150, 170, 180, 190, 200, 210, 220, 220, 230, 230, 240, 250

$Q_1 = 185$,

$Q_2(\text{medián}) = 215$,

$Q_3 = 230$

$IQR = 45$

kiugró küszöbök: $[117.5, 297.5]$

nincs kiugró érték

Stílusok beállítása

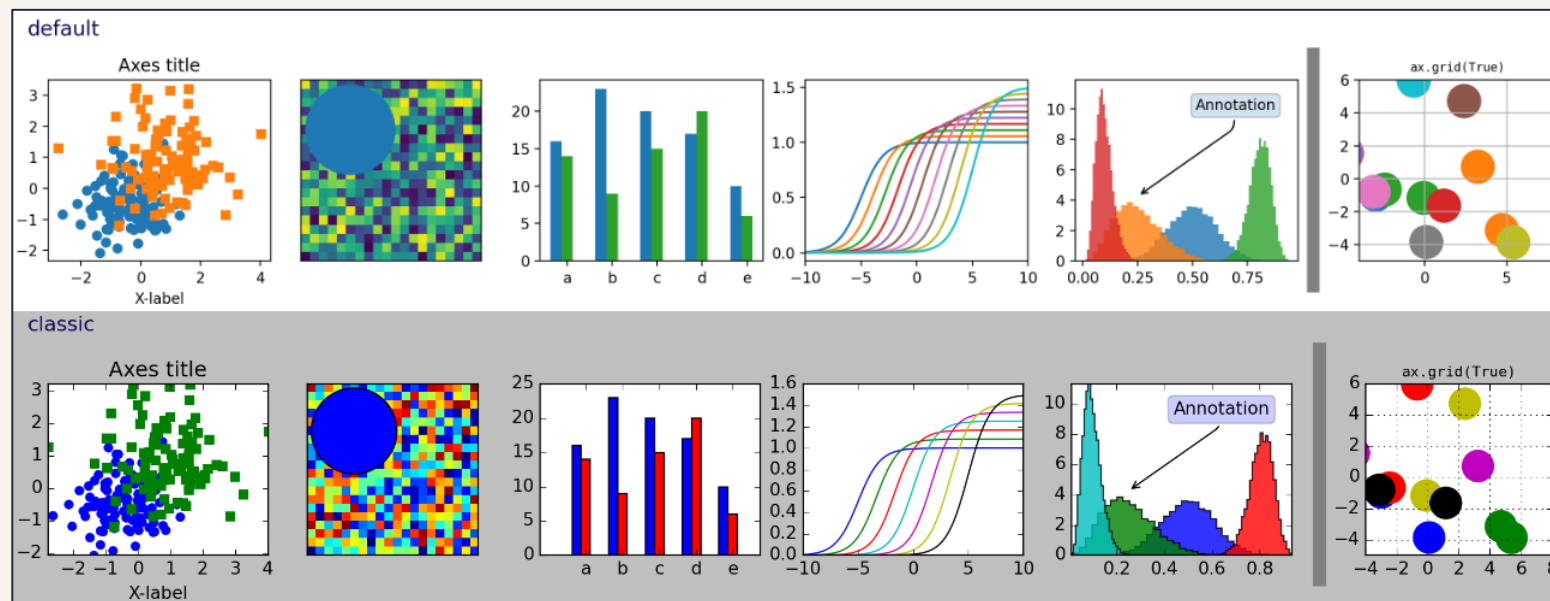
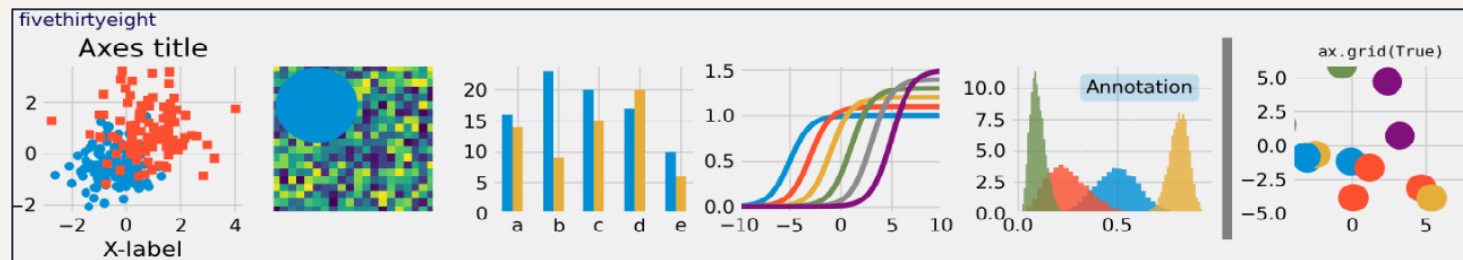
A **plt.style.use()** paraméterével stíluslapot választhatunk a megjelenítéshez az alapbeállítástól eltérően.

```
# Stílus beállítása  
plt.style.use('fivethirtyeight')
```

[styles.py \(.pdf\)](#)

Stílusok beállítása

[Style sheets reference — Matplotlib 3.9.2 documentation](#)



2. Adatok megjelenítése Pandas használatával

- A Pandas és a Matplotlib jól integrálhatók, lehetővé téve a DataFrame objektumok közvetlen vizualizálását.
- A Matplotlib alapértelmezés szerint megjelenik, ha a Pandas **plot()** metódusát hívjuk meg a DataFrame-re.
- Fontos csomagok importálása:

```
import pandas as pd  
import matplotlib.pyplot as plt
```


Háttérszínnek beállítása

```
import matplotlib.pyplot as plt
import pandas as pd

data = {'X': [1, 2, 3, 4, 5], 'Y': [2, 3, 5, 8, 13]}
#Jöhetne csv-ből is!
df = pd.DataFrame(data)

fig = plt.figure(facecolor='lightgreen') # ábra háttérszíne
ax = plt.gca() # Az aktuális tengelyek lekérése
ax.set_facecolor('lightblue') # A rajzterület háttérszínének beállítása
# Adatok ábrázolása
plt.plot(df['X'], df['Y'], 'ro')
# Diagram megjelenítése
plt.show()
```

[backgroundcolor.py \(.pdf\)](#)

Adatok betöltése Pandas-szal és megjelenítése Matplotlib-bel

Példa: Egy CSV fájlban egy bolt napi eladásait tartjuk nyilván. Az adatok beolvasása egy DataFrame-be történik a Pandas segítségével. Ábrázolás Matplotlib használatával.

A **plt.plot()** függvényel a beolvasott adatokat ábrázoljuk, ahol az x tengelyen a napok, az y tengelyen pedig az eladások szerepelnek.

[pandasplot1.py \(.pdf\)](#)

```
import pandas as pd
# CSV adatállomány betöltése
adatok = pd.read_csv('eladasok.csv')
napok = adatok['Nap']
eladasok = adatok['Eladás']
# Vonaldiagram készítése
plt.plot(napok, eladasok, marker='x', linestyle='-', color='purple')
plt.title('Napi eladások')
plt.xlabel('Napok')
plt.ylabel('Eladások (db)')
plt.show()
print(adatok.head())
```

	Nap	Eladás
0	2024-11-01	150
1	2024-11-02	200
2	2024-11-03	170
3	2024-11-04	220
4	2024-11-05	180

Adatmegjelenítés beépített Pandas plot metódussal

A Pandas DataFrame és Series objektumai beépített plot() metódussal rendelkeznek.

Példa: Vonal-diagram készítése beépített Pandas metódussal

A **kind** paraméter határozza a meg a diagram típusát (pl. line, bar), az **x** és **y** a tengelyek megadása, a **marker** és **color** paraméterek a megjelenést szabályozzák.

[pandplotMain.py \(.pdf\)](#)

```
adatok = pd.read_csv('eladasok_30.csv')
adatok.plot(x='Nap', y='Eladás', kind='line', marker='x',
            color='green', title='Napi eladások')
plt.xlabel('Napok')
plt.ylabel('Eladások (db)')
plt.grid(True)
plt.show()
# A beolvasott adatok megtekintése
print(adatok.head())
```

	Nap	Eladás	Látogatók
0	2024-11-01	288	78
1	2024-11-02	196	56
2	2024-11-03	267	75
3	2024-11-04	202	61
4	2024-11-05	228	70

Oszlopdigram készítése

```
adatok.plot(x='Nap', y='Eladás', kind='bar', color='coral', title='Napi eladások oszlopdigramon')  
plt.xlabel('Napok')  
plt.ylabel('Eladások (db)')  
plt.show()
```

A **kind='bar'** beállítással oszlop-diagramot készítünk.
Ez a diagram jól használható az adatok összehasonlítására különböző időszakokban.

[oszlopdia.py \(.pdf\)](#)

Hisztogram készítése az eladások eloszlásának bemutatására

```
# Hisztogram készítése  
adatok['Eladás'].plot(kind='hist', bins=6, color='skyblue', title='Eladások eloszlása'), edgecolor='black')  
plt.xlabel('Eladások száma')  
plt.ylabel('Gyakoriság')  
plt.show()
```

A **kind='hist'** beállítással hisztogramot készítünk, amely az adatok eloszlását mutatja.

A **bins** paraméter határozza meg a csoportok számát.

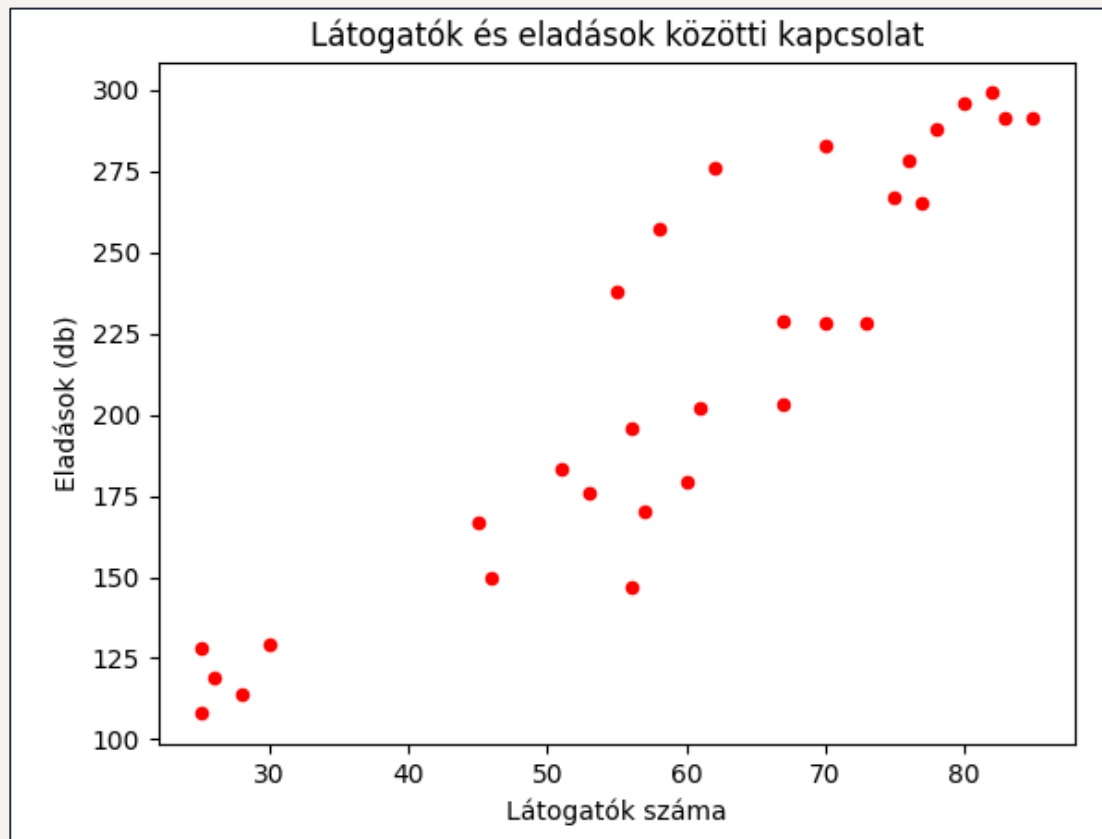
Scatter plot a korreláció vizsgálatához

Ehhez van van egy másik oszlopunk, amely a bolt látogatóinak számát tartalmazza.

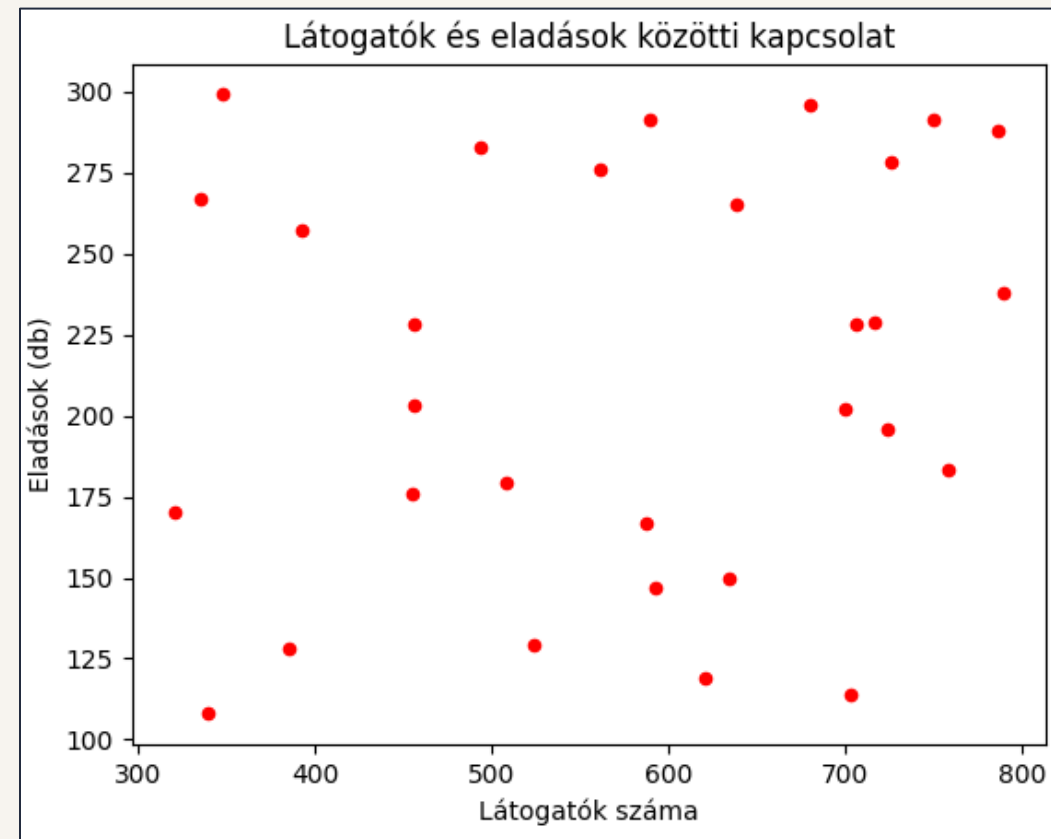
```
# Scatter plot készítése a látogatók száma és eladások közötti kapcsolat ábrázolására
adatok.plot(kind='scatter', x='Látogatók', y='Eladás', color='red', title='Látogatók és eladások közötti kapcsolat')
plt.xlabel('Látogatók száma')
plt.ylabel('Eladások (db)')
plt.show()
```

[plotandscatter.py \(.pdf\)](#)

Scatter plot a korreláció vizsgálatához



eladasok_30.csv



eladasok_30_uj.csv

3. *Komplex diagramok létrehozása*

Példa: Egy adatkészlet alapján két változó ábrázolását egy diagramon két tengellyel oldjuk meg. Egy bolt havi bevételeit és látogatóinak számát ábrázoljuk a hónapok függvényében.

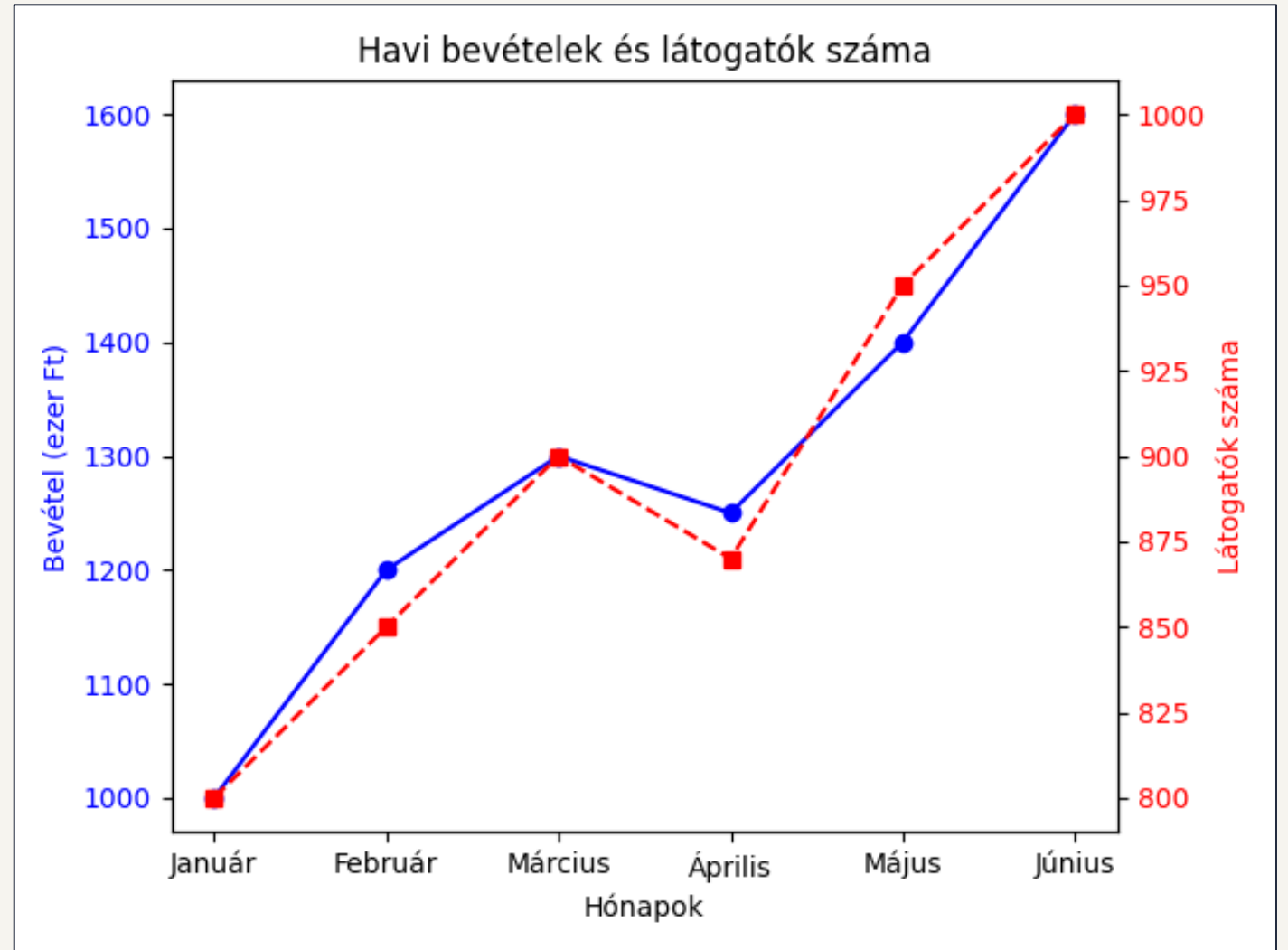
[complex.py \(.pdf\)](#)

```
# Adatok generálása
honapok = ['Január', 'Február', 'Március', 'Április', 'Május', 'Június']
bevetelek = [1000, 1200, 1300, 1250, 1400, 1600]
latogatok = [800, 850, 900, 870, 950, 1000]

# Kombinált diagram készítése
fig, ax1 = plt.subplots() # egy ábra és egy tengely létrehozása
# Első Y tengely: bevételek
ax1.plot(honapok, bevetelek, color='blue', marker='o', label='Bevétel')
ax1.set_xlabel('Hónapok')
ax1.set_ylabel('Bevétel (ezer Ft)', color='blue')
ax1.tick_params(axis='y', labelcolor='blue')
# Második Y tengely: látogatók száma
ax2 = ax1.twinx()
ax2.plot(honapok, latogatok, color='red', marker='s', linestyle='--',
         label='Látogatók száma')
ax2.set_ylabel('Látogatók száma', color='red')
ax2.tick_params(axis='y', labelcolor='red')
plt.title('Havi bevételek és látogatók száma')
fig.tight_layout()
plt.show()
```


A **plt.subplots()** és a **twinx()** metódusok segítségével két y tengelyt hozhatunk létre, így egy diagramon egyszerre két különböző adatot ábrázolhatunk.

[twx.py \(.pdf\)](#)



Példa: Overlay diagram készítése

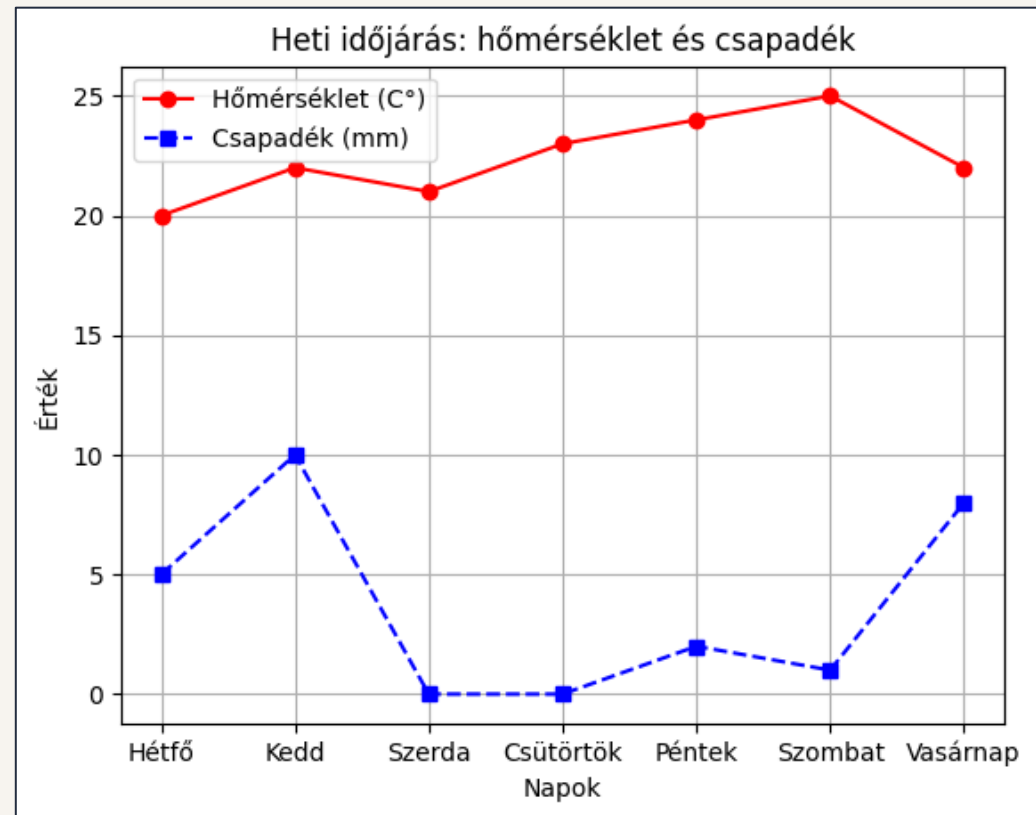
Több különböző adatsor ábrázolása egy diagramon: példánkban két grafikon, amelyek egy város napi legnagyobb hőmérsékletét és napi csapadékmennyiségét mutatják.

```
napok = ['Hétfő', 'Kedd', 'Szerda', 'Csütörtök', 'Péntek', 'Szombat', 'Vasárnap']
homerseklet = [20, 22, 21, 23, 24, 25, 22] # Celsius fok
csapadek = [5, 10, 0, 0, 2, 1, 8] # mm
plt.plot(napok, homerseklet, marker='o', color='red', label='Hőmérséklet (C°)')
plt.plot(napok, csapadek, marker='s', color='blue', linestyle='--', label='Csapadék (mm)')
plt.title('Heti időjárás: hőmérséklet és csapadék')
plt.xlabel('Napok')
plt.ylabel('Érték')
plt.legend()
plt.grid(True)
plt.show()
```

[complex.py \(.pdf\)](#)

Példa: Overlay diagram készítése

Az y tengelyre az Érték feliratot írjuk és a **label** címke, mint jelmagyarázat azonosítja az egyes grafikonokat.



4. *Seaborn bevezetése és használata*

A Seaborn könyvtár alkalmazásának előnyei:

- A Seaborn egy magasabb szintű Python-könyvtár Matplotlib alapokon
- Előnyei: egyszerű használat, különleges diagramok, Pandas integráció
- Jól alkalmazható statisztikai adatok vizualizálására és összetett grafikonok létrehozására
- Hőtérképek, eloszlásdiagramok és trenddiagramok
- Beépített esztétikai beállítások
- Könnyen integrálható Pandas DataFrame-ekkel
- Több beépített plot típust tartalmaz, amelyekkel különböző típusú adatvizualizációkat készíthetünk.
- Telepítés: **pip install seaborn**

Alapvető Seaborn plotok: relplot()

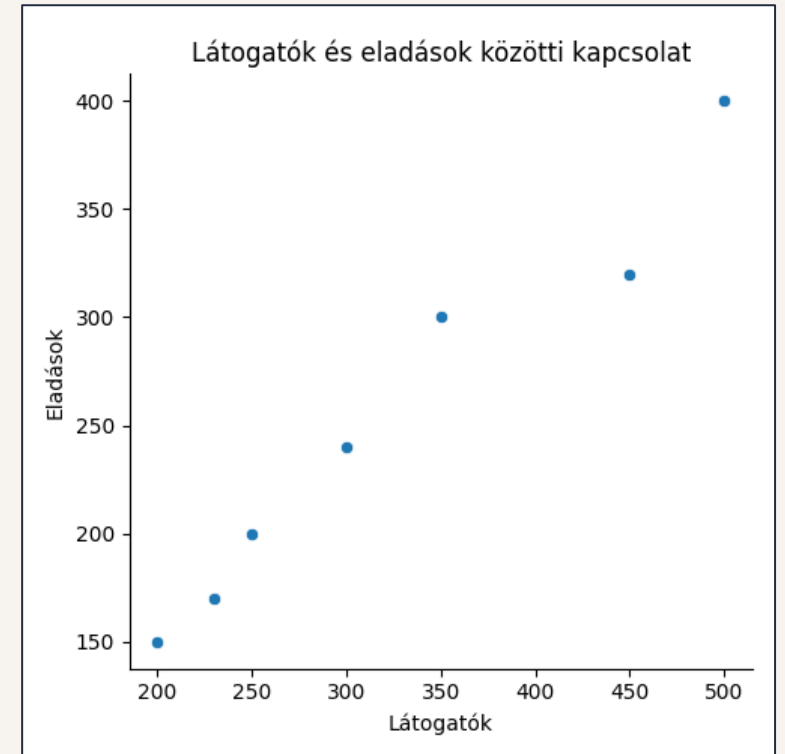
A **relplot()** egy sokoldalú függvény, amely kapcsolatokat jelenít meg az adatok között.

Példa: Egy cég napi eladásait és a látogatók számát ábrázoljuk scatter plot-tal:

```
import seaborn as sns
import pandas as pd
import matplotlib.pyplot as plt

# Adatok generálása DataFrame-ben
adatok = pd.DataFrame({
    'Nap': ['Hétfő', 'Kedd', 'Szerda', 'Csütörtök', 'Péntek', 'Szombat', 'Vasárnap'],
    'Eladások': [150, 200, 170, 240, 300, 400, 320],
    'Látogatók': [200, 250, 230, 300, 350, 500, 450]
})

# relplot használata
sns.relplot(x='Látogatók', y='Eladások', data=adatok, kind='scatter')
plt.title('Látogatók és eladások közötti kapcsolat')
plt.show()
```



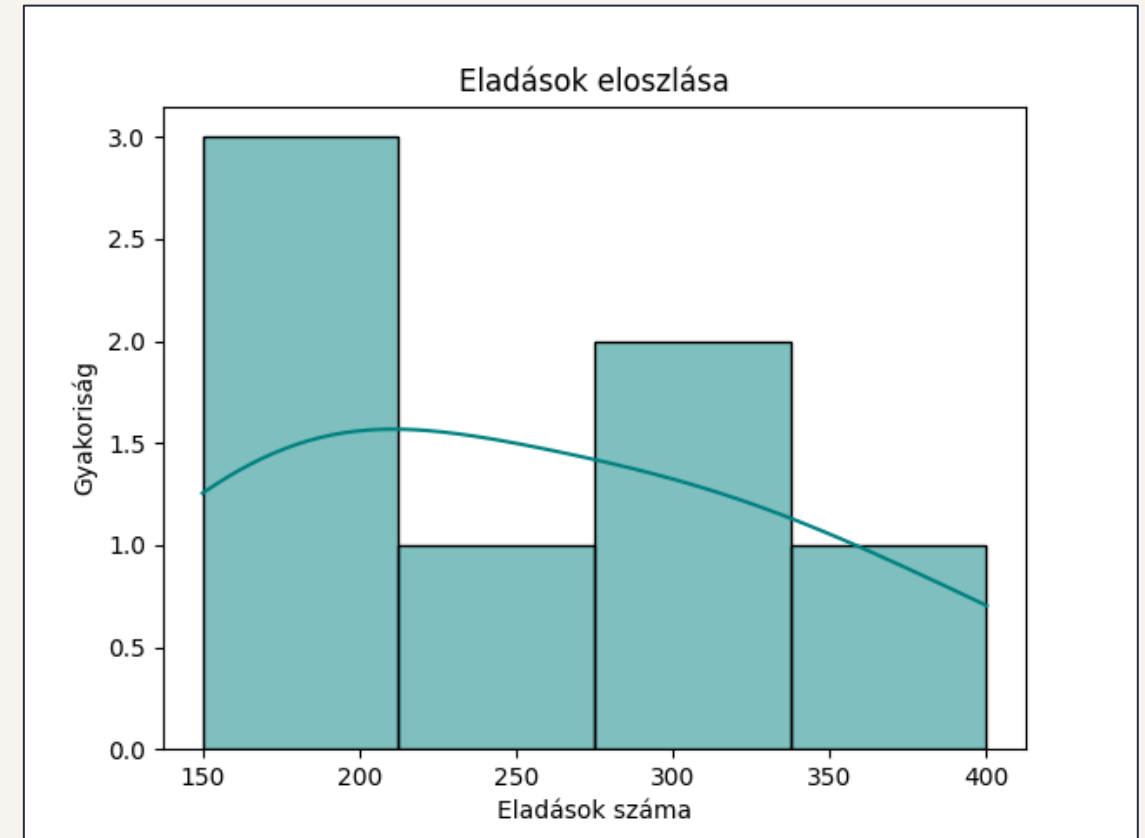
Alapvető Seaborn plotok: histplot()

Eladások eloszlásának megjelenítése **histplot()**-tal:

```
sns.histplot(adatok['Eladások'], kde=True, color='teal')  
plt.title('Eladások eloszlása')  
plt.xlabel('Eladások száma')  
plt.ylabel('Gyakoriság')  
plt.show()
```

A **hisztogram** mellett egy **Kernel Density Estimate (KDE)** görbét is megjelenít, amely az adatok **sűrűségét** mutatja.

[seabornhist.py\(.pdf\)](#)



Alapvető Seaborn plotok: boxplot()

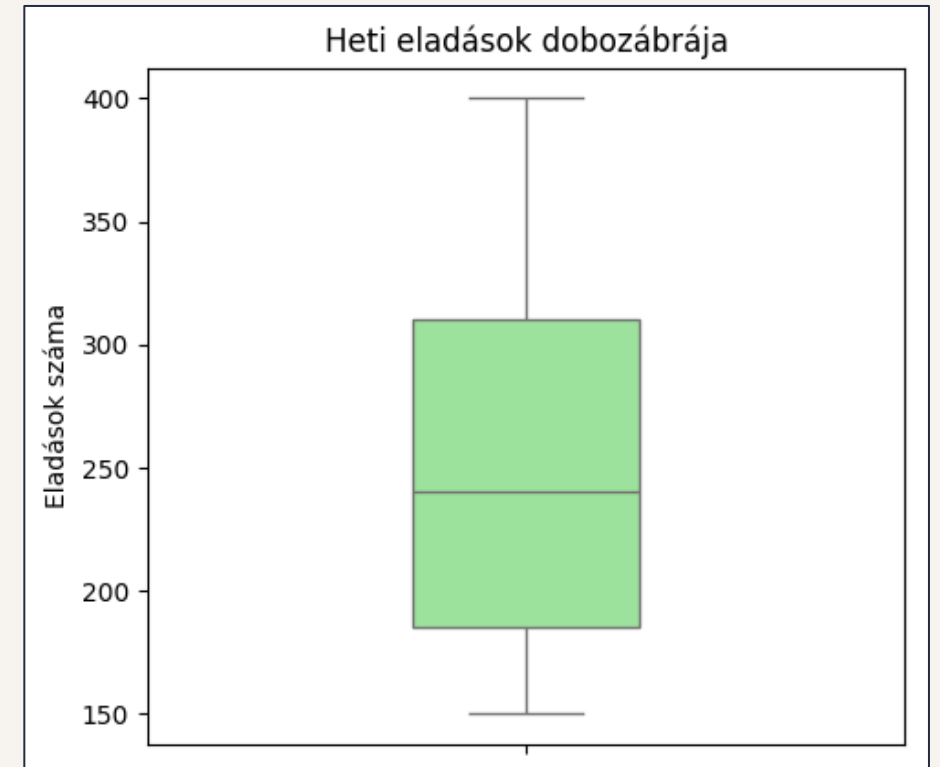
A **boxplot()** segítségével az adatok szórását, mediánját és kiugró értékeit vizualizálhatjuk.

Példa: Heti eladások ábrázolása dobozábrán.

```
sns.boxplot(y=adatok['Eladások'], color='lightgreen', width=0.3)
plt.title('Heti eladások dobozábrája')
plt.ylabel('Eladások száma')
plt.show()
```

A Seaborn boxplotok téglalapja az alapbeállítás miatt szélesebb a plt boxploténál. A width=0.3 az eredetileg kapott téglalap szélességét 30%-ra állítja be.

[seabornboxplot.py \(.pdf\)](#)



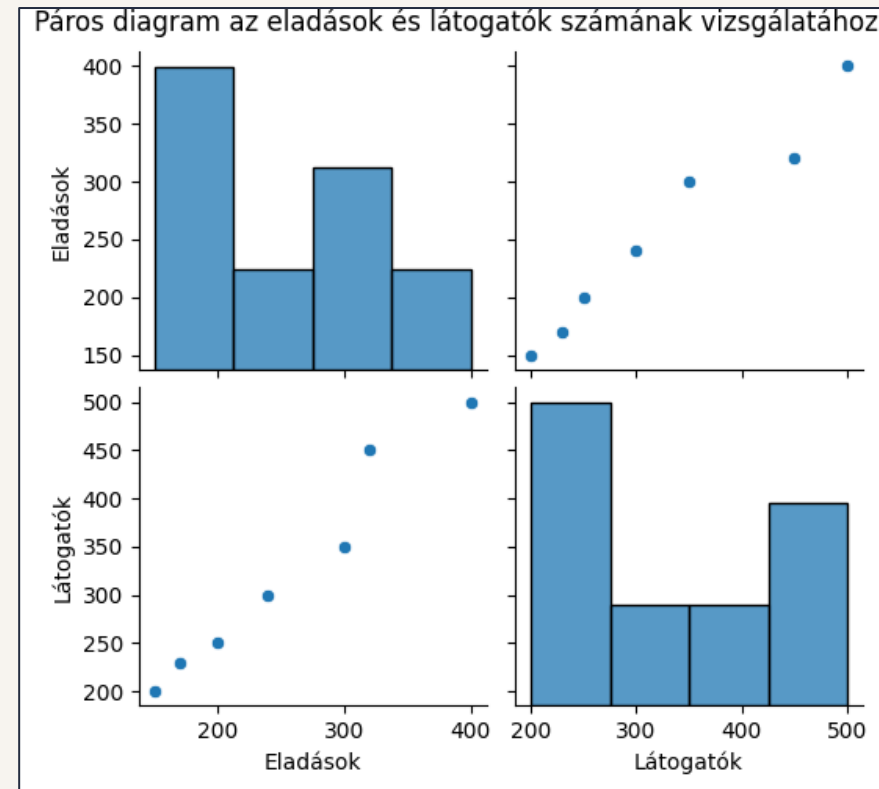
Alapvető Seaborn plotok: pairplot()

A pairplot() több változó közötti kapcsolatok és eloszlások bemutatására szolgál, és automatikusan elkészíti a scatter plotokat és hisztogramokat.

Példa: Többváltozós elemzés:

```
sns.pairplot(adatok)
plt.suptitle('Páros diagram az eladások
és látogatók számának vizsgálatához', y=1.0)
plt.show()
```

[seabornpairplot.py \(.pdf\)](#)



Alapvető Seaborn plotok: heatmap()

Adatok korrelációjának vizualizációja

Korrelációs mátrix:

	eletkor	fizetes
eletkor	1.000000	0.964549
fizetes	0.964549	1.000000

[seabornheatmap.py \(.pdf\)](#)

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

```
adatok = {
    'eletkor': [23, 45, 34, 25, 32, 40, 29, 48, 37, 22],
    'fizetes': [250, 500, 300, 260, 320, 480, 290, 520, 410, 240]
}
```

```
adat_df = pd.DataFrame(adatok)
```

```
# Korrelációs mátrix kiszámítása
```

```
korrelacio = adat_df.corr()
```

```
print("\nKorrelációs mátrix:")
```

```
print(korrelacio)
```

```
# Korrelációs mátrix megjelenítése
```

```
sns.heatmap(korrelacio, annot=True, cmap='coolwarm')
```

```
plt.title('Korrelációs mátrix')
```

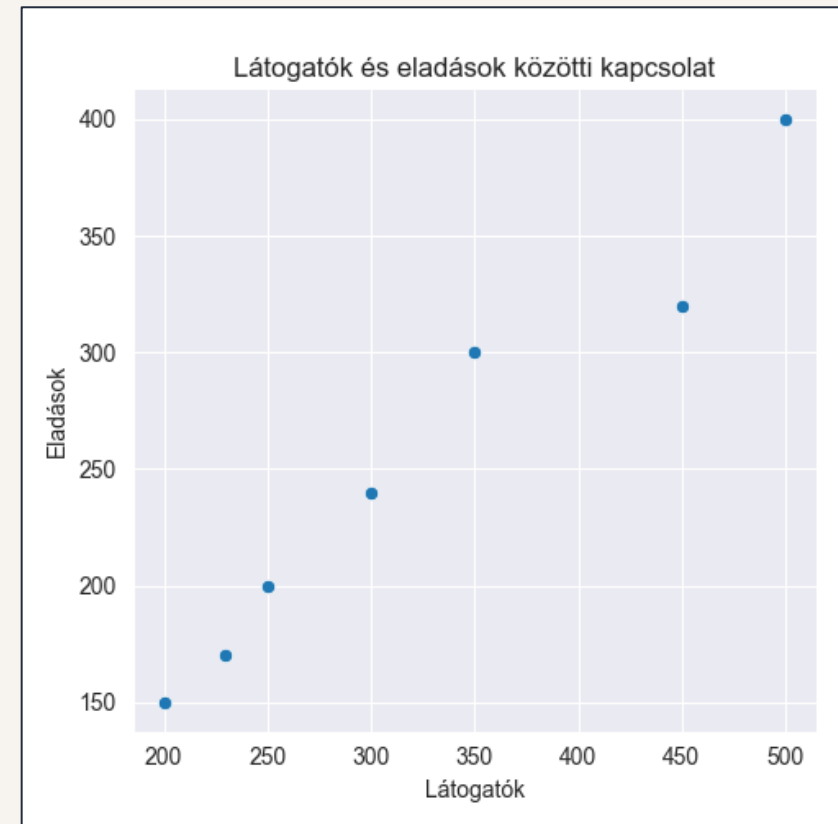
```
plt.show()
```

Stílusok és esztétikai testreszabások

A Seaborn beépített stílusai lehetővé teszik az ábrák gyors testreszabását. Például beállíthatjuk a sötét háttérrel, rácsosítást vagy egyéb esztétikai elemeket.

Stílusbeállítások használata:

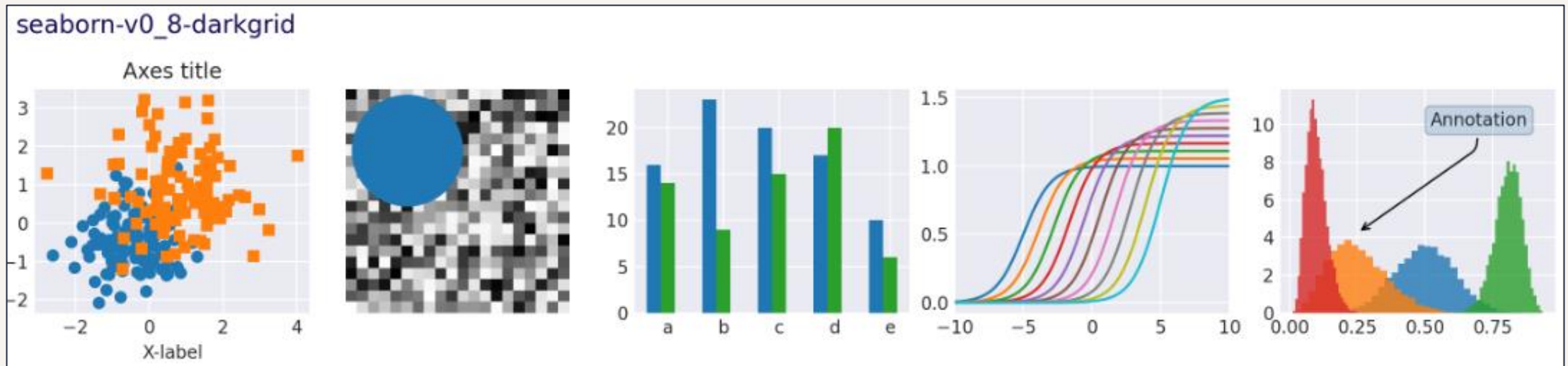
```
sns.set_style('darkgrid')
plt.figure(figsize=(10, 6))
# relplot használata stílusban
sns.relplot(x='Látogatók', y='Eladások', data=adatok,
            kind='scatter')
plt.title('Látogatók és eladások közötti kapcsolat')
plt.show()
```



Stílusok és esztétikai testreszabások

Beépített Seaborn stílusok változatai:

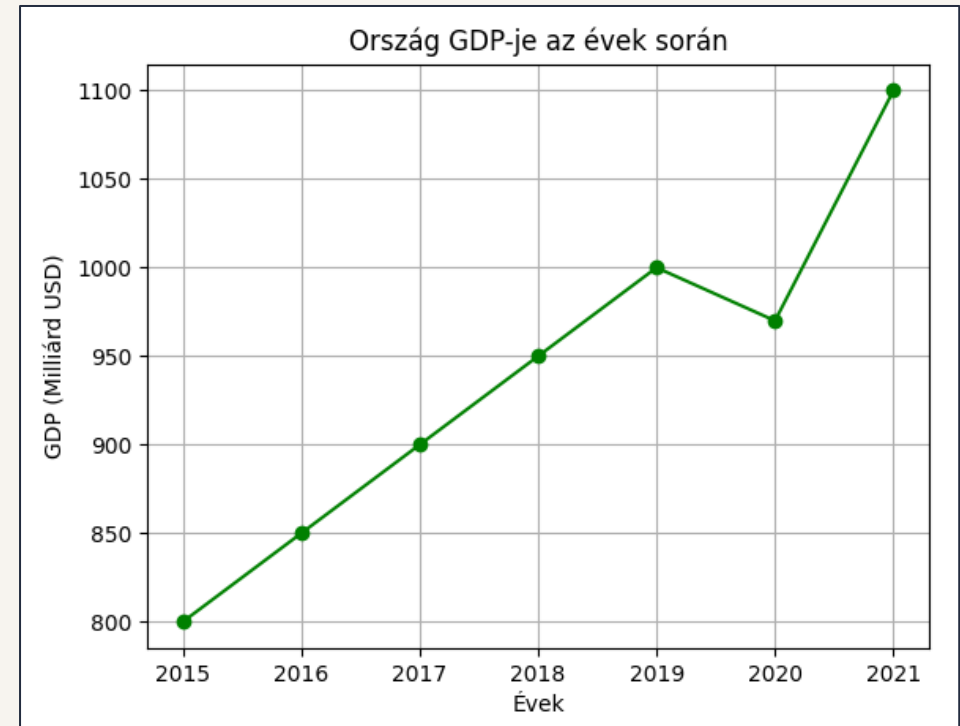
[Style sheets reference – Matplotlib 3.9.2 documentation](#)



5. Grafikonok készítése gyakorlati példák alapján

Példa: Gazdasági adatok vizualizálására készítsünk egy vonaldiagramot, amely egy ország GDP-jének alakulását ábrázolja az évek során.

```
import matplotlib.pyplot as plt
import pandas as pd
# Gazdasági adatok (például éves GDP)
evek = [2015, 2016, 2017, 2018, 2019, 2020, 2021]
gdp = [800, 850, 900, 950, 1000, 970, 1100] # Milliárd USD
plt.plot(evek, gdp, marker='o', linestyle='-', color='green')
plt.title('Ország GDP-je az évek során')
plt.xlabel('Évek')
plt.ylabel('GDP (Milliárd USD)')
plt.grid(True)
plt.show()
```



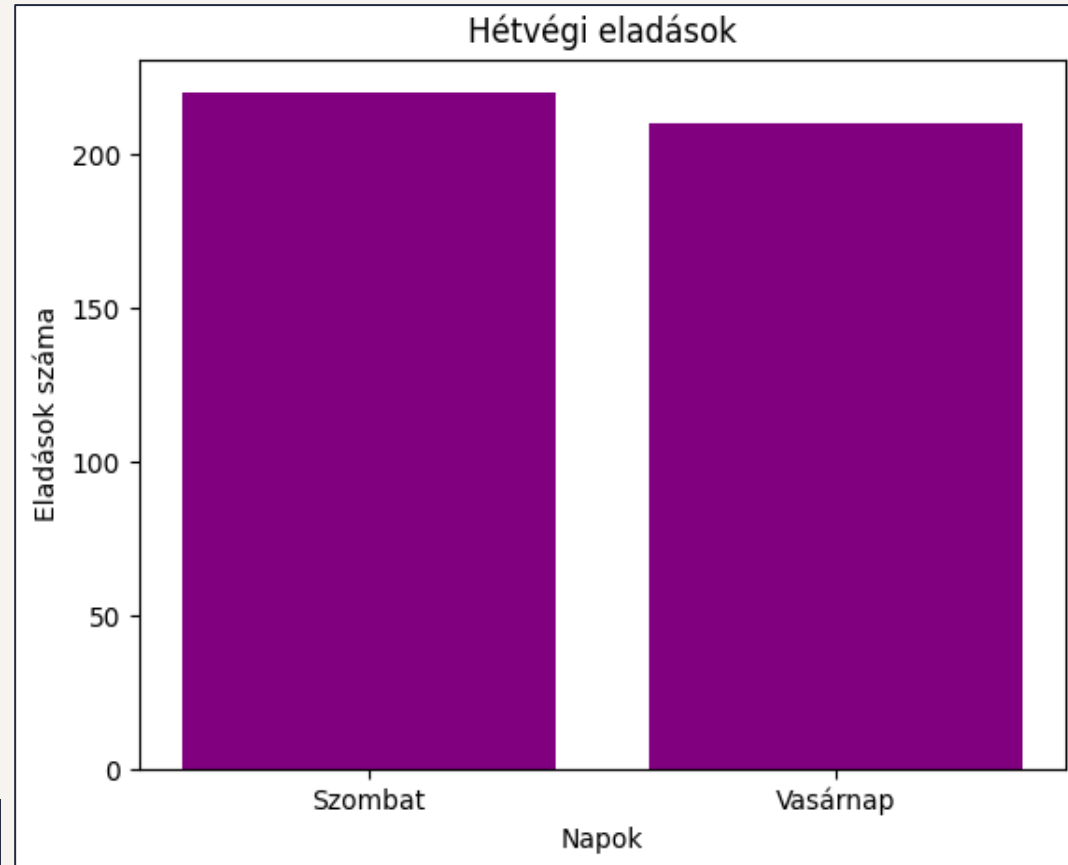
5. Példa: Adatok szűrése és különböző szempontok szerinti ábrázolás

Oszlopdiagramot készítünk, amely egy bolt eladásait mutatja a hét különböző napjain, csak a hétvégi adatokat kiemelve. Szűrés a DataFrame-ben:

```
adatok = pd.DataFrame({
    'Nap': ['Hétfő', 'Kedd', 'Szerda', 'Csütörtök', 'Péntek', 'Szombat', 'Vasárnap'],
    'Eladások': [120, 150, 130, 160, 170, 220, 210]
})
# Adatok szűrése csak hétvégi napokra
hetvegi_adatok = adatok[adatok['Nap'].isin(['Szombat', 'Vasárnap'])]
plt.bar(hetvegi_adatok['Nap'], hetvegi_adatok['Eladások'], color='purple')
plt.title('Hétvégi eladások')
plt.xlabel('Napok')
plt.ylabel('Eladások száma')
plt.show()
```

5. Példa: Adatok szűrése és különböző szempontok szerinti ábrázolás

Az **isin()** metódussal szűrjük a DataFrame-t, majd a kiválasztott adatokat ábrázoljuk.



Köszönöm a figyelmet!

ELTE IK

Python kurzus – 11. előadás

Tárgyfelelős: Dr. Tejfel Máté

Előadó: Dr. Király Roland

ELTE IK



Tematika

1. Flask telepítése és elindítása
2. Native code a template fájlokban, Template kezelés
3. Bootstrap a Flask template-ben
4. NAV Bar és Serach Bar – bootstrap
5. GET és POST
6. Oldalak betöltése menüből
7. Statefull vs Stateless, SESSION kezelés és Flash üzenetek

A prezentációhoz tartozó forráskódok

1. [eacodes\Flask\from flask import Flask.pdf](#)
2. [eacodes\Flask\indexblock.pdf](#)
3. [eacodes\Flask\indexbootstrap.pdf](#)
4. [eacodes\Flask\indexea.pdf](#)
5. [eacodes\Flask\indexlogin.pdf](#)
6. [eacodes\Flask\indexsession.pdf](#)
7. [eacodes\Flask\indextemplate.pdf](#)
8. [eacodes\Flask\indexvars.pdf](#)
9. [eacodes\Flask\forráskódok.zip](#)

Python webfejlesztés

Flask

ELTE IK



Flask alapok

- A Flask egy lightweight, Python alapú webes keretrendszer, amely minimalista megközelítést alkalmaz.
- Ideális kisebb alkalmazásokhoz vagy prototípusokhoz, de nagyobb projektekhez is.
- Flask támogatja az URL routingot, a sablonkezelést és a RESTful API-k készítését.
- Moduláris felépítésének köszönhetően könnyen integrálható bővítményekkel, például adatbázis-kezeléshez vagy hitelesítéshez
- A Flask beépített fejlesztői szerveret és hibakeresési eszközöket biztosít a gyors fejlesztés érdekében.

Flask telepítés és elindítása

ELTE IK



Flask alapok, telepítés és a webszerver elindítása

```
from flask import Flask, request url_for
from flask import render_template, redirect,
session, flash
from datetime import timedelta
```

Telepítés: `pip install flask`

`http://127.0.0.1:5000/`: Ez megjeleníti a kezdőoldalt

A `http://127.0.0.1:5000/users/<name>`

Fejlesztői mód bekapcsolása (opcionális):

`FLASK_ENV=development`

Automatikus újraindítás — a Flask szerver magától újraindul, nem kell manuálisan leállítani/elindítani.

Interaktív hibakereső — ha a kód kivételt dob, a böngészőben megjelenik egy részletes hibaoldal a stack trace-szel, ahol Python kódot is futtathatunk.

Részletes hibaüzenetek — a 500 Internal Server Error helyett látod a pontos hiba okát.

Flask alapok, telepítés és a webszerver elindítása

```
from flask import Flask, redirect, url_for
```

```
app = Flask(__name__)
```

```
@app.route('/')  
def home():
```

```
    return '<h1>Hello Flask</h1>'
```

[eacodes\Flask\indexea.pdf](#)

Debug mode

```
3  app = Flask(__name__)
4
   Tabnine | Edit | Test | Explain | Document
5  @app.route('/<user>')
6  def home(user: str):
7      #return render_template("index2.html", bodycont
8      return render_template("index3.html", namelist=
9
10
11  if __name__ == '__main__':
12      app.run(debug=True)
```


Flask alapok, telepítés és a webszerver elindítása

```
if __name__ == '__main__':  
    app.run(debug=True) # opcionális
```

Telepíteni kell a Flask csomagot
Redirect az átirányításokhoz
url_for a függvények kezeléséhez

? debug mód - nem kell újraindítani a
szerver

? Mint a uvicorn esetén a Fast API-ban.

HTML file template

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Flask App</title>
</head>
<body>
  <h1>Welcome to Flask!</h1>
  <p>This is a basic HTML template served by Flask.</p>
</body>
</html>
eacodes/templates/simple.html
```

Template kezelés

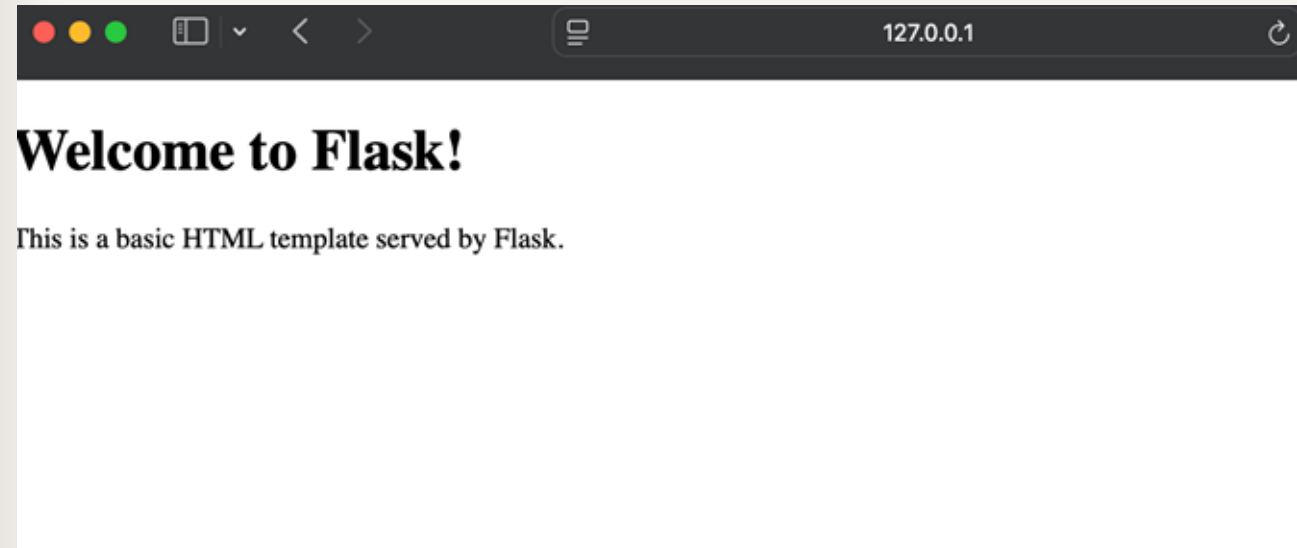
```
from flask import Flask, redirect, url_for,
render_template

app = Flask(__name__)

@app.route('/')
def home():
    return render_template("index.html")

if __name__ == '__main__':
    app.run()
```

Html kimenet



Információ átadás(Information passing, GET-readme-ben)

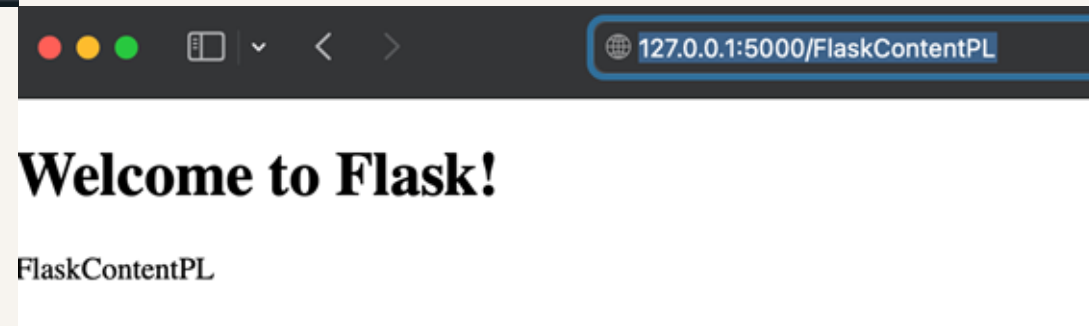
```
1 from flask import Flask, render_template
2
3 app = Flask(__name__)
4
5 @app.route('/<user>')
6 def home(user: str):
7     return render_template("index.html", bodycontent=user)
8
9 if __name__ == '__main__':
```

Információ átadás

```
6     <title>Flask App</title>
7 </head>
8 <body>
9     <h1>Welcome to Flask!</h1>
10    <p>{{bodycontent}}</p>
11 </body>
12 </html>
```

{{user}} formában hívunk a Flask tartalomra.
<https://127.0.0.1:5000/FlaskContentPL>
(simple2.html, indexea3.py)

{{user}} formában hívunk a Flask tartalomra.
<https://127.0.0.1:5000/FlaskContentPL>
(simple2.html, indexea3.py)



Flask alapok, telepítés és a webserverver elindítása

```
from Flask import Flask, render_template

app = Flask(__name__)

@app.route('/<user>')
def home(user: str): return
    render_template("index.html", bodycontent=user, message="Logged in")

if __name__ == '__main__':
    app.run()
```

Több adat/tartalom is átadható egy rendszerben.
([simple3.html](#), [indexea3.py](#))
[eacodes/templates/simple3.html](#)

Native code a template fájlokban

ELTE IK



Native Python kód a HTML template-ben

```
<body>
    <h1>Native python code</h1>
    {% for i in range(0, 10) %}
    <p>Line {{i}}</p>
    {% endfor %}
</body>
```


Listák kezelése

(indexvars.py, index3.html)

```
<title>Flask App</title>
</head>
<body>
  <h1>Native python code</h1>
  {% for name in namelist %}
  <p>{{name}}</p>
  {% endfor %}
</body>
</html>
```

```
app = Flask(__name__)

@app.route('/<user>')
def home(user: str):
    #return render_template("index2.html", bodycontent=user, message="Logged in")
    return render_template("index3.html", namelist=["Név1", "Név2", "Név3"])
```

Template kezelés

ELTE IK



*(template.html,
index4.html, indexea5.py)*

```
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>{% block title %}{% endblock %}</title>
</head>
<body>
  <div>Menubar</div>
  <h1>
    {% block content %}
    {% endblock %}
  </h1>
</body>
</html>
```

```
{% extends "template.html" %}
{% block title %} Home and Navbar later {% endblock %}
{% block content %}
Welcome to Flask!
{% endblock %}
```

Template

```
{% extends "template.html" %}  
{% block title %} Home and Navbar later {% endblock %}  
{% block content %}  
Welcome to Flask!  
{% endblock %}
```

```
indexea5.py > route  
1 from flask import Flask, request, url_for, render_template  
2  
3 app = Flask(__name__)  
4  
5 @app.route('/')  
6 def route():  
7     return render_template("index4.html")  
8  
9 if __name__ == '__main__':  
10     app.run()
```

Bootstrap a Flask template-ben

<https://getbootstrap.com>

#

<https://getbootstrap.com/docs/5.3/components/navbar/#nav>

ELTE IK

Bootstrap
(base.html, index5.html,
indexea5.py)

```
{% extends "base.html" %}  
{% block title %}  
    Home and Navbar later  
{% endblock %}  
{% block content %}  
    Flask!  
{% endblock %}
```

A bootstrap link a
head tag-be kerül

```
<link  
  href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.5/  
  dist/css/bootstrap.min.css" rel="stylesheet"  
  integrity="sha384-  
  SgOJa3Dml69IUzQ2PVdRZhwQ+dy64/BUtbMJw1  
  MZ8t5HZApcHrRKUc4W0kG879m7"  
  crossorigin="anonymous">
```

Java script BSTRP a
body tag végére
kerül!

```
<script  
src="https://cdn.jsdelivr.net/npm/@popperjs/core@2.  
11.8/dist/umd/popper.min.js" integrity="sha384-  
I7E8VVD/ismYTF4hNIPjVp/Zjvgyol6VFvRkX/vR+Vc4jQkC  
+hVqc2pM8ODewa9r"  
crossorigin="anonymous"></script>
```

```
<script  
src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.5/dis  
t/js/bootstrap.min.js" integrity="sha384-  
VQqxDN0EQCkWoxT/0vsQvZswzTHUVOImccYmSyhJTp7  
kGtPed0Qcx8rK9h9YEgx+"  
crossorigin="anonymous"></script>
```


NAV bar és Search Bar bootstrap-ból

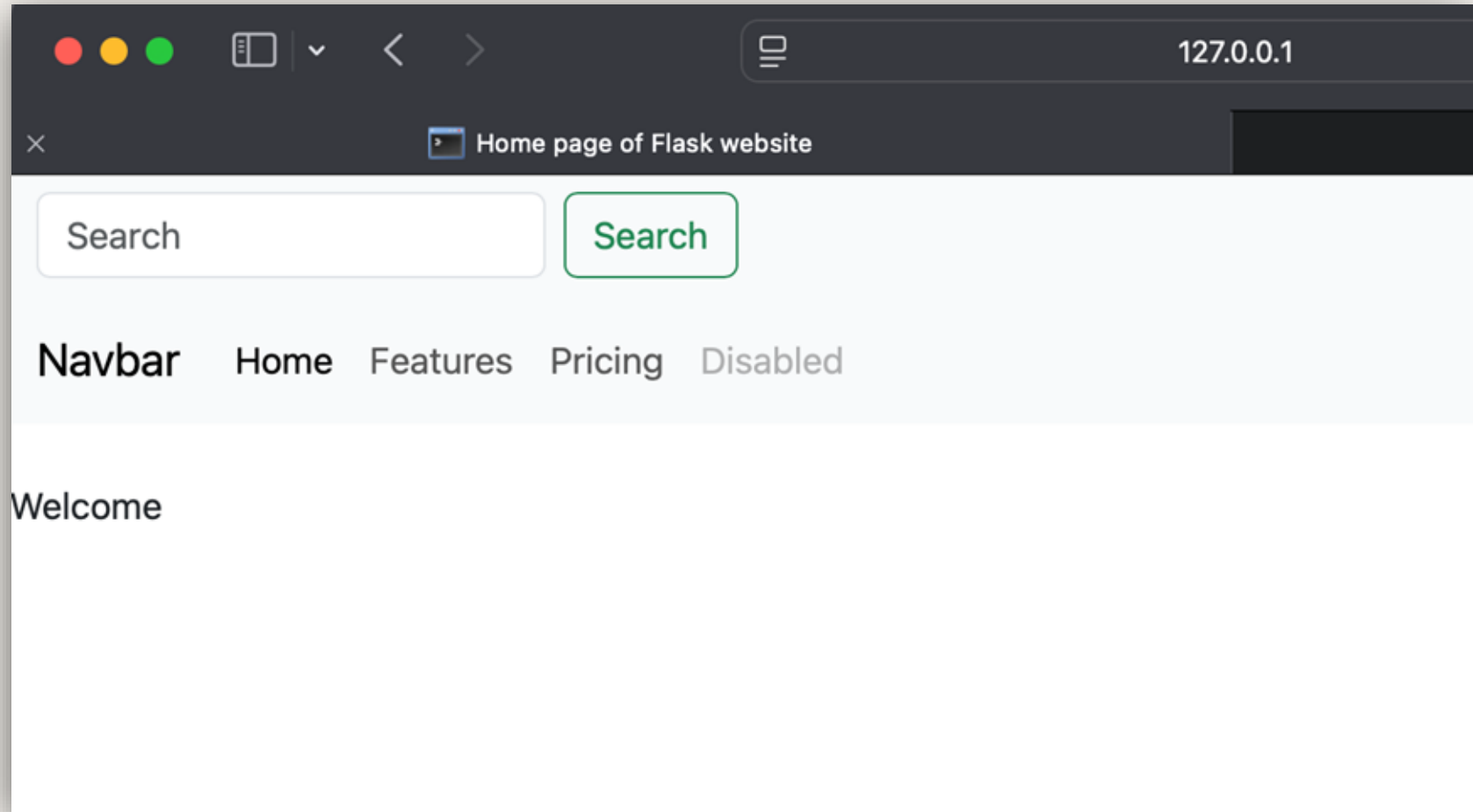
ELTE IK



Navbar

<https://getbootstrap.com/docs/5.3/components/navbar/#nav>

(templatenv.html, index6.html,
indexea6.py)
[eacodes\templates\index6.html](#)



GET és POST

ELTE IK

GET és POST

eacodes\templates\login.html

eacodes\Flask\indexlogin.pdf

eacodes\templates\templatonav.html

127.0.0.1 - - [01/Dec/2025 20:57:00] "GET / HTTP/1.1"

200 -

127.0.0.1 - - [01/Dec/2025 20:57:02] "GET /loginpage
HTTP/1.1" 200 -

127.0.0.1 - - [01/Dec/2025 20:57:10] "POST /loginpage

```
@app.route('/')
def home():
    return render_template("index5.html")

@app.route('/loginpage', methods=["POST", "GET"])
def loginpage():
    if request.method == "POST":
        username = request.form["username"]
        return redirect(url_for("user", usr=username))
    else:
        return render_template("login.html")

@app.route('/<usr>')
def user(usr):
    return f"<h1>{usr}</h1>"
```

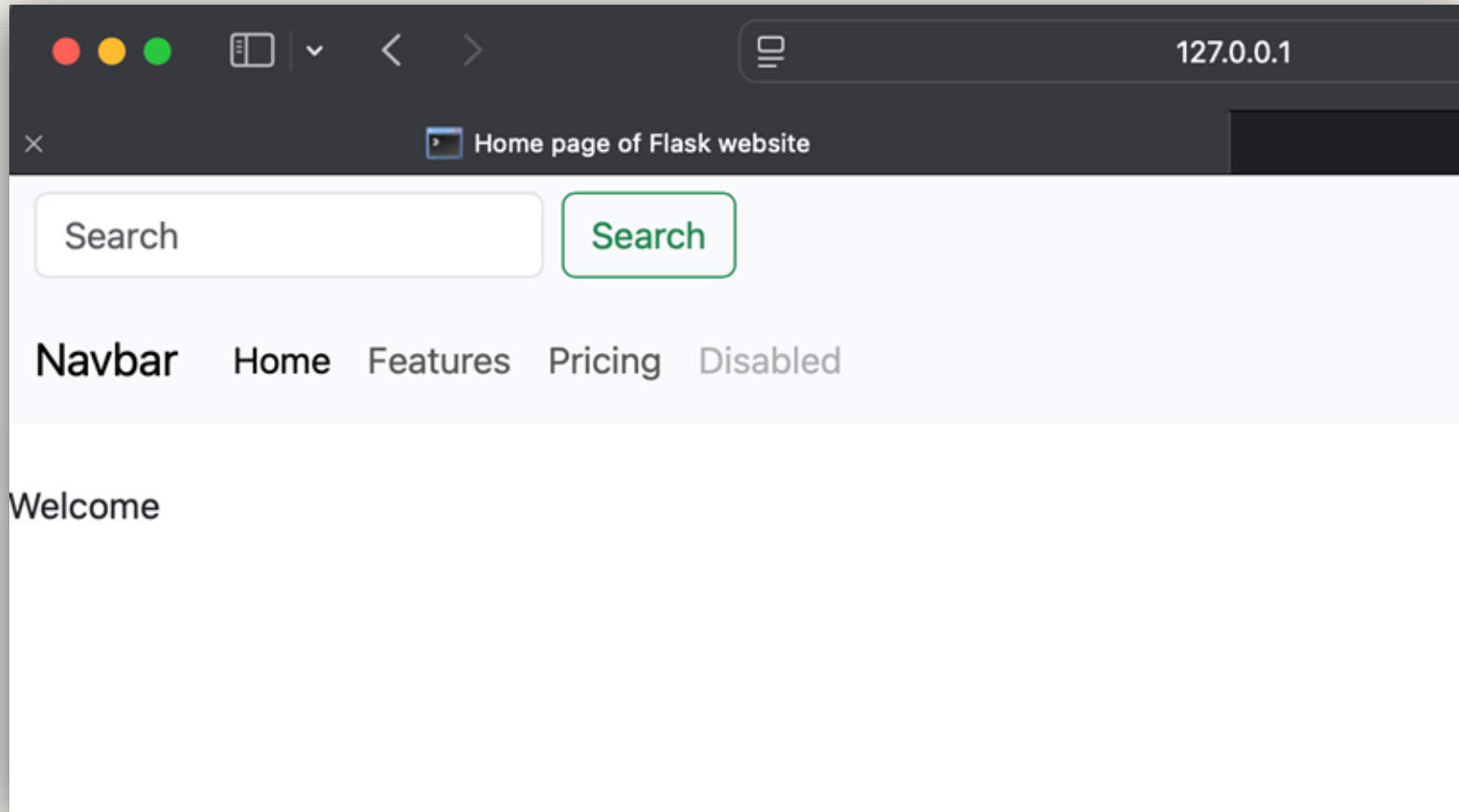
```
{% extends "templatonav.html" %}
{% block title %} Login {% endblock %}
{% block content %}
<form action="#" method="post">
    <p><input type="text" name="username"></p>
    <p><input type="submit" value="submit"></p>
</form>
{% endblock %}
```

Oldalak betöltése menüből

ELTE IK



Másik oldal betöltése HTML menüből



Statefull vs stateless

ELTE IK



Session timeout

```
from flask import Flask, request, url_for
from flask import render_template, redirect, session
from datetime import timedelta

app = Flask(__name__)
app.secret_key = 'alma'
app.permanent_session_lifetime = timedelta(minutes=20)
```

```
@app.route('/loginpage', methods=["POST", "GET"])
def loginpage():
    if request.method == "POST":
        session.permanent = True
        username = request.form["username"]
        session["user"] = username
        return redirect(url_for("user"))
    else:
        return render_template("login.html")
```


session – kezelés – állapotok/állapottér

[eacodes\Flask\indexsession.pdf](#)
[eacodes\templates\index6.html](#)
[eacodes\templates\login.html](#)

```
@app.route('/loginpage', methods=["POST", "GET"])
def loginpage():
    if request.method == "POST":
        username = request.form["username"]
        session["user"] = username
        return redirect(url_for("user"))
    else:
        return render_template("login.html")

@app.route('/user')
def user():
    if "user" in session:
        user = session["user"]
        return f"<h1>{user}</h1>"
    else:
        #return f"<h1>nouser in session</h1>"
        return redirect(url_for("loginpage"))
```

ELTE IK

Flash  zenetek

ELTE IK



Flash üzenetek

```
from flask import Flask, request, url_for
from flask import render_template, redirect, session, flash
from datetime import timedelta
```

```
@app.route('/logout')
def logout():
    session.pop("user", None)
    flash("Logged out!", "Message")
    #return f"<h1>Logged out</h1>"
    return redirect(url_for("loginpage"))
```

Flash üzenetek kiíratása

```
<div class="container-fluid">{% block content %}{%endblock%}</div>
{% with messages = get_flashed_messages() %}
{% if messages %}
<ul class=flashes>
{% for message in messages %}
  <li>{{ message }}</li>
{% endfor %}
</ul>
{% endif %}
{% endwith %}
```

```
@app.route('/logout')
def logout():
    if "username" in session:
        user = session["username"]
        session.pop("username", None)
        flash("Logout", "Message")
        return redirect(url_for("loginpage"))
    else:
        flash("No user!")
        return redirect(url_for("/"))
```

Flash megjelenítése

```
{% with messages =  
get_flashed_messages(with_categories=true) %}  
  {% if messages %}  
    <ul class="flashes">  
      {% for category, msg in messages %}  
        <li class="{{ category|lower|replace(' ',  
'-') }}">{{ msg }}</li>  
      {% endfor %}  
    </ul>  
  {% endif %}  
{% endwith %}
```

Flash megjelenítése

A `get_flashed_messages(with_categories=True)` azt jelenti, hogy az üzeneteket kategóriával együtt adja vissza.

`with_categories=False` (alapértelmezett) — csak az üzenet szövege

`get_flashed_messages()`

```
["You have been logged in!", "No user!"]
```

`with_categories=True` — (kategória, üzenet) tuple lista

`get_flashed_messages(with_categories=True)`

```
[("Message", "You have been logged in!"), ("error", "No user!")]
```

Flash megjelenítése

```
class="{{ category|lower|replace(' ', '-') }}">{{  
msg }}
```

Jelentése:

category — a flash("Logout", "User logged in!")
második argumentuma, tehát "User logged in!"
|lower — kisbetűssé alakítja: "user logged in!"
|replace(' ', '-') — a szóközöket lecseréli
ködőjelre: "user-logged-out!"

Static folder

ELTE IK

Static

```
<html>
<head>
  <title>{% block title %} {% endblock %}</title>
  <link rel="stylesheet" type="text/css" href="{{ url_for('static', filename='style.css')}}">
</head>
<body>
  {% block content %}
  {% endblock %}
</body>
</html>
```

```
{% extends 'base.html' %}
{% block title %}
  Home Page
{% endblock %}

{% block content %}
  <h1>Home Page</h1>
  
{% endblock %}
```

Köszönöm a
figyelmet!

ELTE IK



Python kurzus – 12. előadás

Tárgyfelelős: Dr. Tejfel Máté

Előadó: Dr. Király Roland

ELTE IK



8. Előadás tematikája

Adatkezelés, fájlműveletek, kép és „mozgóképfeldolgozás”

1. SQLAlchemy, SQLite, Database Client
2. Fájlok megnyitása, olvasása, írása (txt, csv, json)
3. Fájlműveletek hibakezelése, with enter/exit
4. JSON és egyéb fájl formátumok formátum bevezetése és feldolgozása Pythonban
5. Képek feldolgozása opencv csomaggal
6. Video feldolgozása, skeleton based handle detection

A prezentációhoz tartozó forráskódok

[ea13.pdf\(pdf állomány megnyitása\)](#)

[test_with_skeleton.py\(pdf állomány megnyitása\)](#)

[Flask LLM APP.zip \(tartalom letöltése\)](#)

2. Fájlok megnyitása, olvasása, írása (txt, csv, json)

Pythonban a fájlok megnyitása és kezelése a beépített `open()` függvénnyel
különböző formátumú fájlokat olvashatunk és írhatunk

2.1. Text (TXT) fájlok kezelése

A szöveges fájlok szöveges adatokat tárolnak, például naplófájlokat vagy beállítási adatokat. Pythonban a **with open()** szerkezetet használjuk a fájl megnyitására és kezelésére.

a) TXT fájl megnyitása és olvasása

```
# TXT fájl megnyitása olvasásra  
  
with open("data.txt", "r") as file:  
  
    content = file.read()  
  
    print(content)
```

Itt az data.txt fájlt **olvasási módban** ("r") nyitjuk meg. A **file.read()** utasítás a teljes fájl tartalmát beolvassa és eltárolja a **content** változóban, amelyet a program végül kiír a képernyőre.

b) TXT fájl soronkénti olvasása

```
# TXT fájl soronkénti olvasása  
  
with open("data.txt", "r") as file:  
  
    for line in file:  
  
        print(line.strip()) # A strip() eltávolítja a felesleges  
        sortöréseket
```

Nagyobb fájlok esetén hasznos lehet a fájlt soronként olvasni, különösen akkor, ha egyszerre nem akarjuk a teljes tartalmat betölteni.

c) TXT fájl írása

```
# TXT fájl megnyitása írásra  
with open("output.txt", "w") as file:  
    file.write("Ez az első sor.\n")  
    file.write("Ez pedig a második sor.\n")
```

Ebben a példában a kimenet.txt fájlt **írási módban** ("w") nyitjuk meg. A **file.write()** függvénnnyel sorokat írunk a fájlba. Ha a fájl már létezik, annak tartalma **felülíródik**.

d) Adatok hozzáfűzése meglévő TXT fájlhoz

```
# TXT fájl megnyitása hozzáfűzésre  
with open("output.txt", "a") as file:  
    file.write("Ez egy új sor, amit hozzáadunk.\n")  
    file.write("Ez egy újabb sor, amit hozzáadunk.\n")
```

A with open() szerkezet **append** módban ("a") történő használatával új sorokat fűzhetünk hozzá egy már meglévő fájlhoz anélkül, hogy felülírnánk a tartalmát.

2.2. CSV fájl kezelése

A CSV (**Comma-Separated Values**) fájlok olyan táblázatos adatokat tárolnak, amelyek mezőit vesszővel választjuk el egymástól. Pythonban a beépített csv modul segítségével könnyen olvashatunk és írhatunk CSV fájlokat.

A CSV fájlok kezeléséhez szükség van a **csv modul importálására**.

a) CSV fájl olvasása

```
import csv

with open("data.csv", "r") as file:
    reader = csv.reader(file)
    for line in reader:
        print(line)
```

Itt a **csv.reader(file)** objektum **minden sorát listaként** adja vissza. A **for line in reader** ciklus segítségével a fájl összes sorát feldolgozzuk és megjelenítjük.

b) CSV fájl írása

A **csv.writer()** objektum segítségével az adatok lista tartalmát a kimenet.csv fájlba írjuk.

Az **writerows()** metódus minden listaelemet (egy-egy sort) a fájlba ír.

A **newline=""**: a sorok közti üres sorok nem íródnak ki a fájlba

```
import csv

data = [

    ["Név", "Kor", "Város"],

    ["Ann", 25, "Budapest"],

    ["Bell", 30, "Szeged"],

    ["Yoda", 28, "Dagoba"]

]

# CSV fájl megnyitása írásra

with open("kimenet.csv", "w", newline="") as file:

    writer = csv.writer(file)

    writer.writerows(data)
```

2.3. JSON fájl kezelése

A JSON (**JavaScript Object Notation**) egy könnyen olvasható, **hierarchikus** adat formátum.

Pythonban a beépített **json modul** teszi lehetővé JSON fájlok beolvasását és írását.

a) JSON fájl beolvasása

```
import json

with open("data.json", "r") as file:
    data = json.load(file)
    print(data)
```

A **json.load()** függvény a `adatok.json` fájl tartalmát beolvassa és **Python szótárrá** alakítja. A beolvasott adatok így közvetlenül elérhetőek a programban.

b) JSON fájl írása

```
import json

data = {
    "nev": "Bob",
    "kor": 25,
    "varos": "Budapest",
    "devices": ["notebook", "charger", "tablet"]
}

with open("output.json", "w") as file:
    json.dump(data, file, indent=4)
```

A **json.dump()** függvénnnyel az adatok szótárat JSON formátumba írjuk ki a kimenet.json fájlba. Az **indent=4** paraméter biztosítja a JSON fájl olvashatóságát, tagolt formátumban.

Összefoglalás

Pythonban a `with open()` szerkezet használatával egyszerűen kezelhetjük a fájlokat, mivel automatikusan biztosítja a fájl megfelelő bezárását. Az egyes fájl típusok esetén külön modulokat használhatunk: a szöveges fájlokhoz nincs szükség külön modulra, a CSV fájlokhoz a `csv` modul, míg a JSON fájlokhoz a `json` modul biztosít könnyű kezelhetőséget.

3. Fájlműveletek hibakezelése

3.1. Hibakezelés Pythonban: Try-Except struktúra

- A fájlműveletek során Pythonban előfordulhatnak különféle hibák, például ha a fájl nem létezik, vagy nem rendelkezünk a szükséges jogosultságokkal a fájl megnyitásához.
- Az ilyen hibák kezelése elengedhetetlen ahhoz, hogy programjaink stabilan működjenek, és a felhasználó megfelelő tájékoztatást kapjon a problémáról.
- Pythonban a hibakezelést a **try-except blokk** segítségével valósítjuk meg, amely lehetővé teszi, hogy különféle hibákat (kivételkezeléseket) kezeljünk anélkül, hogy a program megszakadna.
- A **try blokkba** kerül a végrehajtandó kódrészlet, az **except blokk** pedig meghatározza, hogyan kezeljük az adott hibát.

Példa:

Fájl megnyitásának hibakezelése

- Ha a fájl nem található, a **FileNotFoundError** kivétel lép fel.
- Ha nincs jogosultságunk, a **PermissionError** kerül kiváltásra.

```
try:
    with open("nonexistent_file.txt", "r") as file:
        content = file.read()
except FileNotFoundError:
    print("Hiba: A fájl nem található!")
except PermissionError:
    print("Hiba: Nincs jogosultság a fájl megnyitásához!")
finally:
    if file:
        close(file)
```

3.2. Gyakori fájlműveleti hibák és azok kezelése:

```
try:
    with open("data.txt", "r") as file:
        content = file.read()
except FileNotFoundError:
    print("Hiba: Az adatok.txt fájl nem található. Ellenőrizze a fájl elérési útját!")
```

a) **FileNotFoundError**

Ez a hiba akkor fordul elő, ha a megnyitni kívánt **fájl nem létezik** az elérési útvonalon.

b) `PermissionError`

Ez a hiba akkor fordulhat elő, ha a fájlhoz szükséges jogosultságok hiányoznak.

```
try:
```

```
    with open("/system/protected_file.txt", "r") as file:
```

```
        content = file.read()
```

```
except PermissionError:
```

```
    print("Hiba: Nem rendelkezik jogosultsággal a fájl megnyitásához.")
```

```
try:
    with open("some_directory", "r") as file:
        content = file.read()
except IsADirectoryError:
    print("Hiba: Az adott útvonal egy könyvtár, nem fájl.")
```

c) IsADirectoryError és NotADirectoryError kezelése

Az IsADirectoryError akkor lép fel, ha egy könyvtárat próbálunk fájlként kezelni.

3.3. Általános kivételkezelés fájlműveleteknél

Előfordulhatnak váratlan hibák, amelyeket általánosan, az **Exception** kivételkezelővel kezelhetünk.

Ilyenkor a kiváltott hibát kiírhatjuk, ami segíthet a probléma **diagnosztizálásában**.

```
try:
    with open("data.txt", "r") as file:
        content = file.read()
except Exception as e:
    print(f"Hiba történt: {e}")
```

Ez a kód minden kivételt megfog, és **a hibát kiírja**.

A példában a fájl nem létezik hiba lépett fel.

Hiba történt: [Errno 2] No such file or directory:
'adatok.txt'

3.4. Fájlok bezárása hibakezelés során

A **finally** blokk garantálja a fájl bezárását.

A **with** kulcsszóval ezt egyszerűbben is elérhetjük.

```
file = None

try:

    file = open("data.txt", "r")

    content = file.read()

except FileNotFoundError:

    print("Hiba: A fájl nem található!")

finally:

    if file:

        file.close()
```

Összegzés

A fájlműveletek hibakezelése fontos része az adatkezelési folyamatoknak.

A specifikus hibakezelési blokkok segítenek a felhasználói hibák kezelésében anélkül, hogy a program lefagyna.

A **with** kulcsszó használata biztosítja, hogy a fájlok megfelelően bezáruljanak a műveletek végén, és így elkerüljük a potenciális adatvesztést.

4. A Python *with* kulcsszó használata

4.1. A *with* kulcsszó alapvető működése

- A Python **with** kulcsszó használata rendkívül hasznos, ha olyan erőforrásokkal dolgozunk, amelyeket a művelet végén automatikusan fel kell szabadítani, például fájlok kezelésekor, hálózati kapcsolatok nyitásakor vagy adatbázis-kapcsolatok létrehozásakor.
- A *with* kulcsszó a **kontextuskezelést** végzi, azaz biztosítja, hogy a megfelelő erőforrások mindig rendben lezáruljanak a művelet végén, még akkor is, ha hiba történik a művelet során.
- Alkalmazásával elkerülhetjük az elfelejtett close() hívásokat.

4.4. Több fájl kezelése egyszerre a with kulcsszóval

```
# Két fájl megnyitása egyszerre: egyik olvasásra, másik írásra  
with open("source.txt", "r") as source_file, open("target.txt", "w") as  
target_file:  
  
    for line in source_file:  
        target_file.write(line)
```

A with szerkezet egyszerre több fájl kezelését teszi lehetővé, és automatikusan gondoskodik azok bezárásáról.

4.5. A with kulcsszó működése a háttérben

A with kulcsszó a háttérben két metódust használ:

`__enter__()` - Felelős az erőforrás megnyitásáért.

`__exit__()` - Gondoskodik az erőforrás bezárásáról. Ez biztosítja, hogy a fájlok mindig lezáródjanak a with blokk végén.

5.4. *JSON szöveg kezelése Pythonban*

A **json.loads()** függvény JSON szöveget Python objektummá (dictionary) alakít.

```
import json
```

```
# JSON formátumú szöveg
```

```
jsondata = '{"name": "Sir Robin", "age": 25, "job": ["brave",  
"running", "fight"]}'
```

```
data = json.loads(jsondata)
```

```
print(data)
```

6.1. Bináris fájlok

- A bináris fájlok bináris formátumban tárolják az adatokat.
- Ilyen fájlok lehetnek például képek, hangfájlok, videók vagy egyéb adatformátumok, amelyek nem olvashatók egyszerűen szöveggént.
- A bináris fájlokat **'rb'** és **'wb'** módban kezelhetjük Pythonban.

Példa: Kép másolása bináris módban

```
# Kép másolása bináris módban

with open("source.jpg", "rb") as source:

    content = source.read()

with open("copy.jpg", "wb") as copy:

    copy.write(content)
```

6.2. XML fájlok

- Az XML (**Extensible Markup Language**) egy másik elterjedt fájlformátum, amelyet strukturált adatok tárolására használnak.
- Az XML hasonló a JSON-hoz, de inkább hierarchikus adatstruktúrákhoz használják, és gyakran találkozhatunk vele webes alkalmazásokban vagy konfigurációs fájlokban.
- Pythonban az **xml.etree.ElementTree** modul segítségével könnyen olvashatunk és írhatunk XML fájlokat.

Itt az adatok.xml fájlt beolvassuk, majd végigiterálunk a gyökérelem elemein, hogy kiírjuk azok címkéit, attribútumait és szöveges tartalmát.

```
import xml.etree.ElementTree as ET

tree = ET.parse("data.xml")      #beolvasás

root = tree.getroot()

for element in root:

    print(element.tag, element.attrib,
          element.text)
```

6.3. Microsoft Excel fájlok (XLSX)

Példa: Excel fájl beolvasása a pandas segítségével

```
import pandas as pd

# Excel fájl beolvasása

data = pd.read_excel("data.xlsx")    #beolvasás

print(data)
```

A Microsoft Excel fájlok (.xlsx) kezelésére a **pandas** és az **openpyxl** könyvtárak használatosak Pythonban, amelyek lehetővé teszik Excel táblázatok olvasását és írását.

A **pandas** könyvtár és a **pd.read_excel()** függvény segítségével beolvashatjuk az Excel fájlokat.

6.4. PDF fájlok

Példa: PDF szöveg kinyerése a PyPDF2 segítségével

```
import PyPDF2

with open("document.pdf", "rb") as pdf:    #bináris olvasási mód

    reader = PyPDF2.PdfReader(pdf)

    first_page = reader.pages[0]

    print(first_page.extract_text())
```

PDF fájlok kezelése Pythonban az **PyPDF2** vagy a **pdfplumber** könyvtárakkal lehetséges, amelyek lehetővé teszik a PDF dokumentumok olvasását és szövegük kinyerését.

A **PdfReader()** metódus segítségével a dokumentum.pdf első oldalának szövegét olvashatjuk ki.

6.5. Képfájlok (JPEG, PNG stb.)

Példa: Képfájl beolvasása és méretének módosítása

```
from PIL import Image  
# Képfájl beolvasása és átméretezése  
image = Image.open("image.jpg")  
new_image = image.resize((200, 200))  
new_image.save("new_image.jpg")
```

Pythonban a **Pillow** könyvtár (PIL, Python Imaging Library) segítségével dolgozhatunk különféle képfájlokkal, például JPEG, PNG, BMP stb. A Pillow lehetővé teszi képek beolvasását, manipulálását és mentését.

Ebben a példában az image.jpg képfájlt beolvassuk, átméretezzük 200x200 pixeles méretre, és elmentjük new_image.jpg néven.

6.6. Hangfájlok (MP3, WAV)

Példa: WAV fájl beolvasása és hossza meghatározása

```
import wave

# WAV fájl beolvasása

with wave.open("sound.wav", "rb") as sound:

    print("Csatornák száma:", sound.getnchannels())

    print("Mintavételezési frekvencia:", sound.getframerate())

    print("Fájl hossza másodpercben:", sound.getnframes() /
          sound.getframerate())
```

Hangfájlok (pl. MP3, WAV) kezelésére Pythonban az **pydub** és **wave** modulok használhatók. Ezek lehetővé teszik a hangfájlok betöltését, átalakítását és lejátszását.

A wave modul lehetővé teszi a WAV hangfájlok beolvasását és jellemzőik elérését. Ez a kód a sound.wav fájlt beolvassa, és kiírja a csatornák számát, mintavételezési frekvenciát, és a hangfájl hosszát másodpercekben.

6.7. ZIP és egyéb tömörített fájlok

Példa: ZIP fájl tartalmának kicsomagolása

```
import zipfile

# ZIP fájl kicsomagolása

with zipfile.ZipFile("files.zip", "r") as zip:

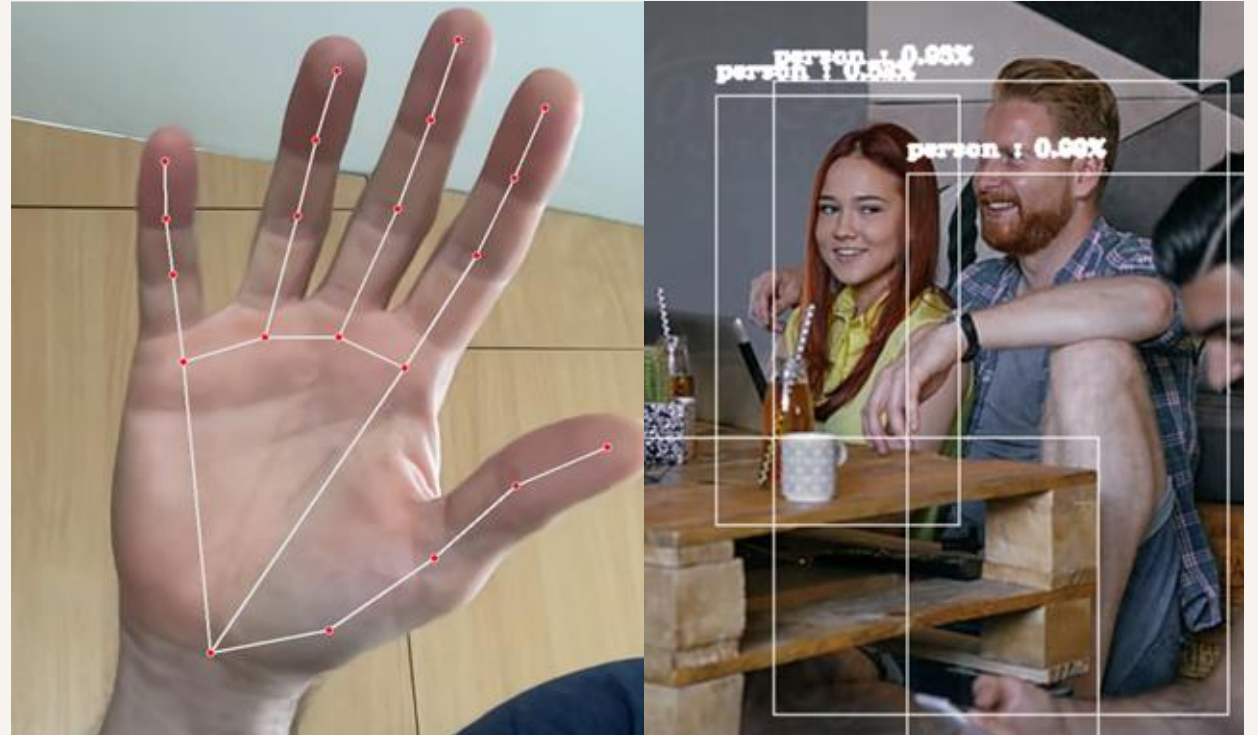
    zip.extractall("kicsomagolt")
```

A zipfile és tarfile modulokkal ZIP és TAR tömörített fájlokat kezelhetünk Pythonban.

Ez a kód a fajok.zip fájl tartalmát kicsomagolja egy kicsomagolt nevű könyvtárba.

Képek feldolgozása

- opencv
- `cv2.dnn.readNetFromCaffe(m1, m2)`
- Object detection, automtikus azonosítás
- Hand detection



Köszönöm a
figyelmet!

ELTE IK



Forráskódok és programok.