

Python Kurzus – 1. 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

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
 - 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



forrás:<https://gvanrossum.github.io/images/index.html>

- Python rövid története - Guido van Rossum fejlesztette ki az 1990-es évek elején.
- A Python népszerűsége és sokoldalú felhasználási lehetőségei.
- Python előnyei:
 - Könnyen elsajátítható programozási nyelv.
 - Átlátható, egyszerű és jól olvasható szintaxis.
 - Gazdag beépített és külső könyvtárkészlet, valamint aktív fejlesztői közösség.
 - Széles körben alkalmazható: kutatási feladatokra, asztali alkalmazásokhoz, webes fejlesztéshez és beágyazott rendszerekhez is.
- Webfejlesztés, API-k készítése és egyéb modern szoftverfejlesztési feladatok.

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

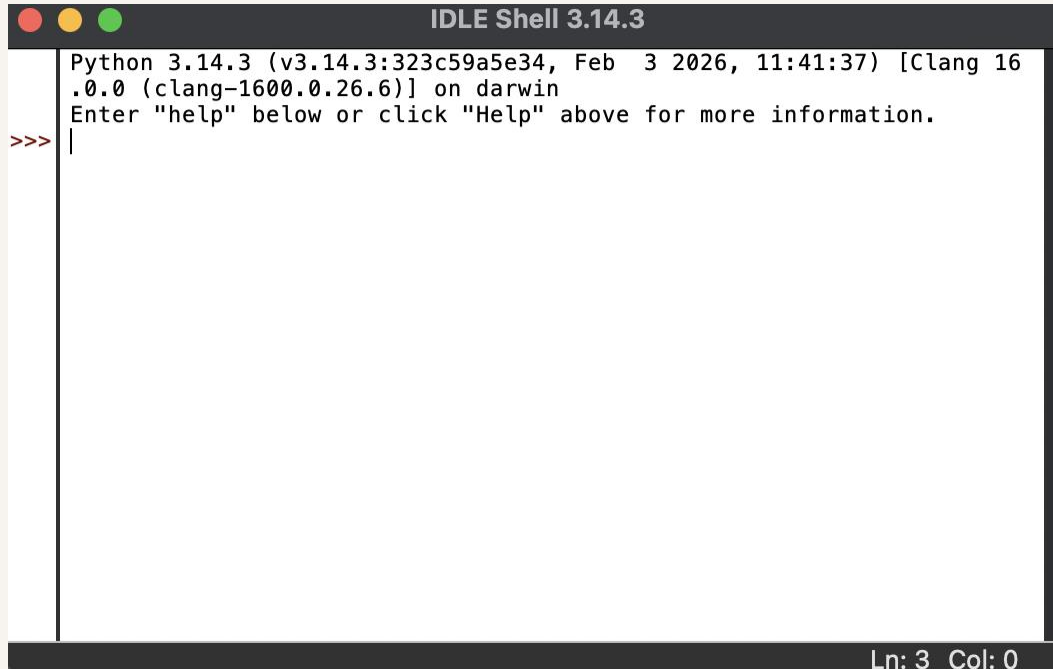
A python programozási nyelv felhasználási területei

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

- Telepítés menete: <https://www.python.org/downloads/>
- 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



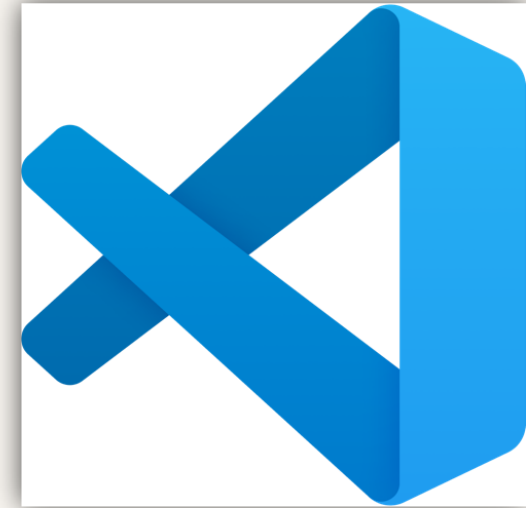
```
Python 3.14.3 (v3.14.3:323c59a5e34, Feb 3 2026, 11:41:37) [Clang 16
.0.0 (clang-1600.0.26.6)] on darwin
Enter "help" below or click "Help" above for more information.
>>> |
```

Ln: 3 Col: 0

- 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)`

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



<https://code.visualstudio.com>



<https://jupyter.org>

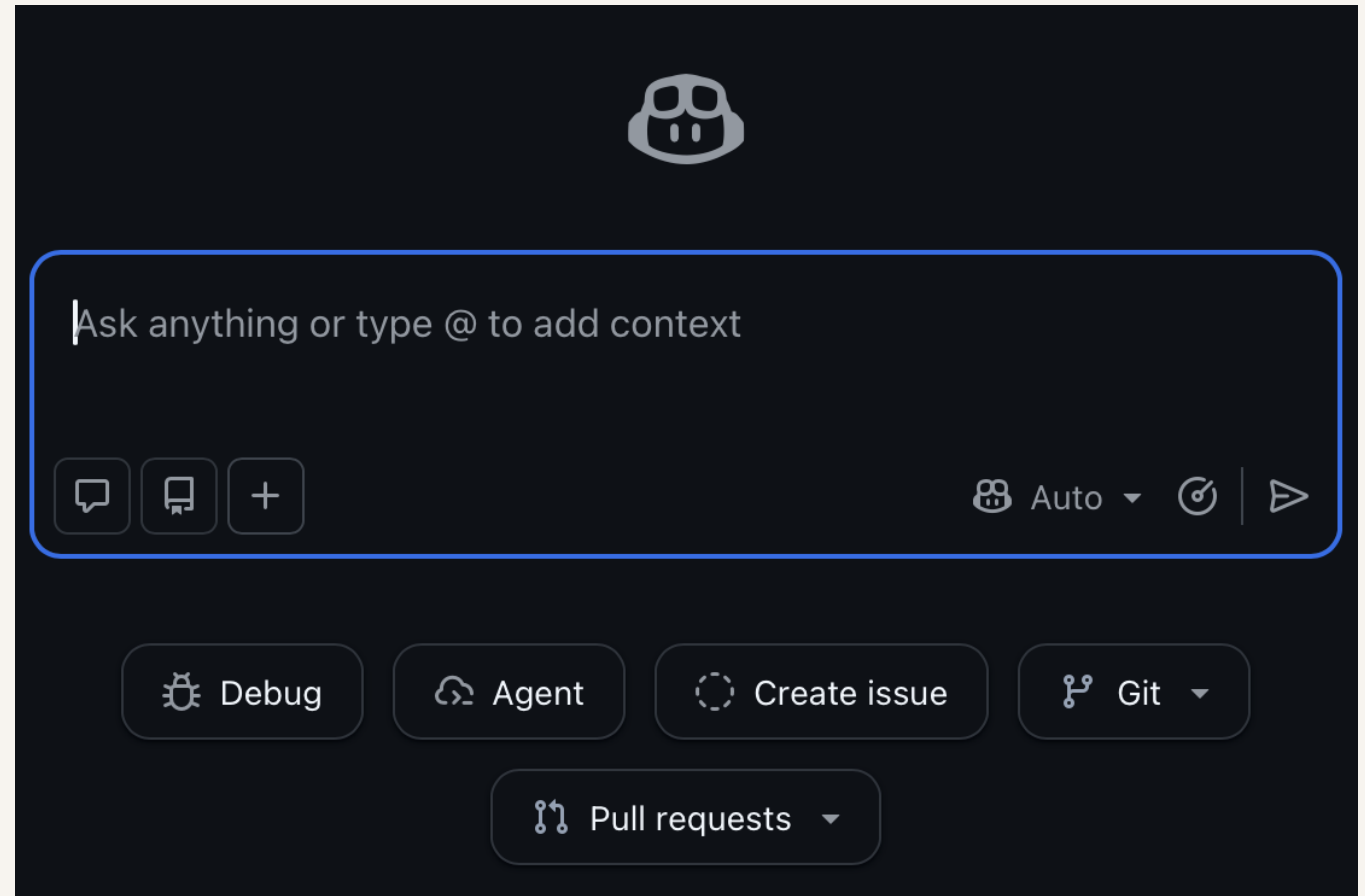
Az AI-alapú kódgenerálás előnyei és korlátai

- **Előnyök:**

- Fejlesztési folyamat gyorsítása
- Kisebb problémák megoldásának segítése

- **Korlátok:**

- Nem mindig pontos vagy optimális kód
- Ellenőrzés szükségessége



<https://github.com/copilot>

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>



<https://github.com/copilot>

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!
```



<https://github.com/copilot>

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



<https://github.com/copilot>

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

Előadás tematikája

Adattípusok és változók

1. Interpreter – compiler, byte code
2. Változók létrehozása és használata
3. Alapvető adattípusok
 - Elemi adattípusok
 - Összetett adatípusok
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)

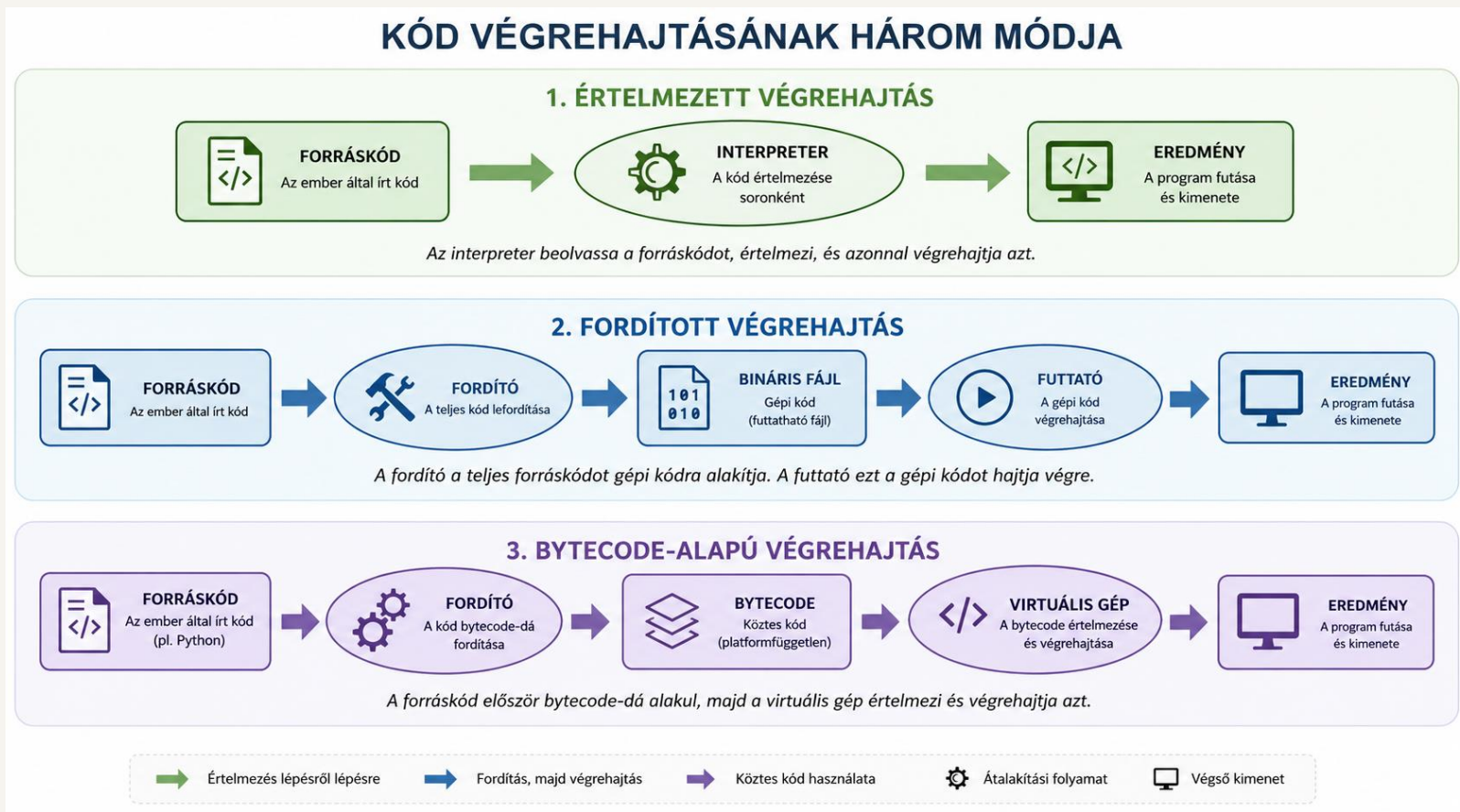
[problem1.py](#) (pdf állomány megnyitása)

[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). A program fordításának lépései:



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
```

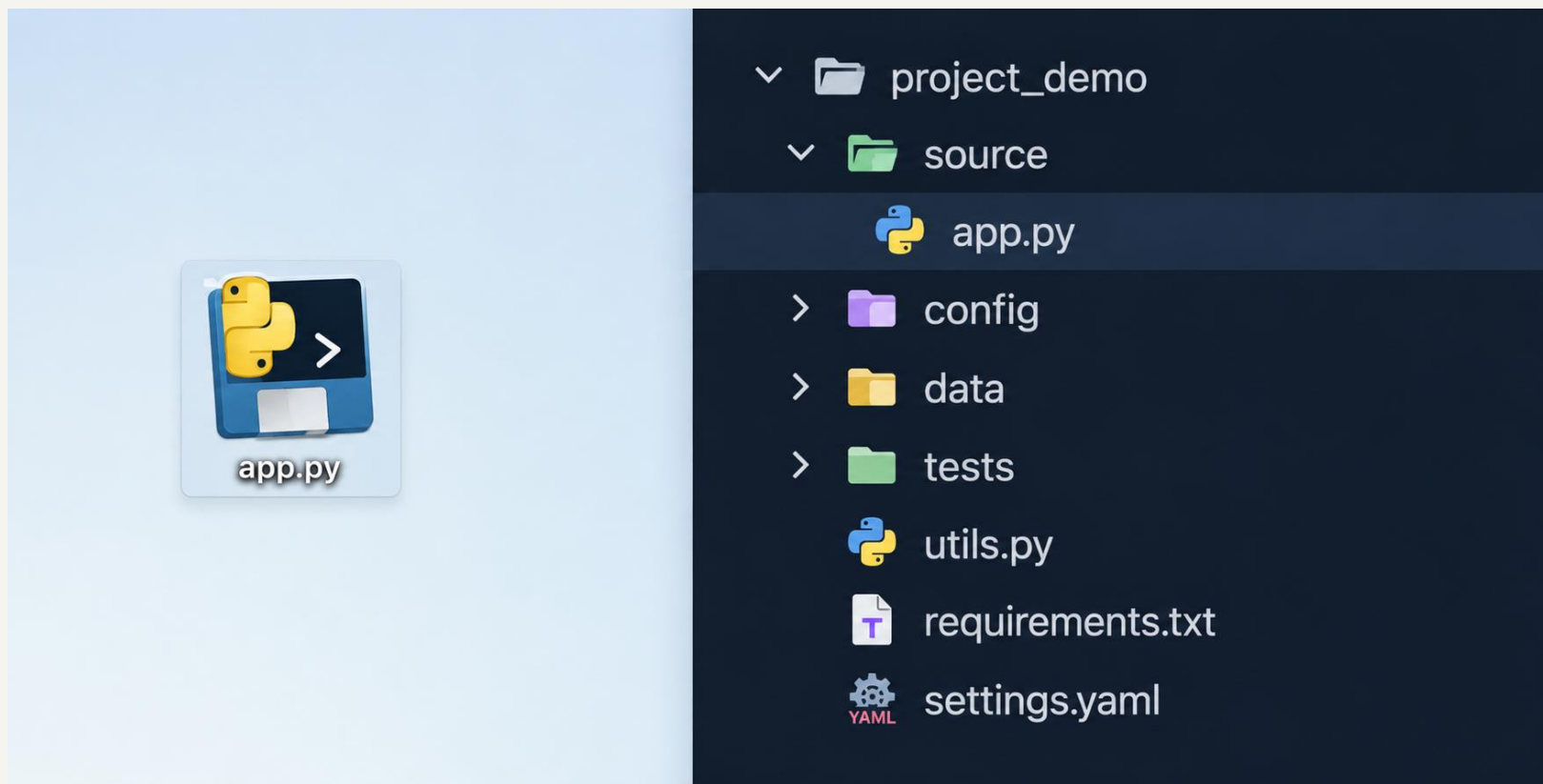
```
APP = ["test.py"]
```

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

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

EXE készítése Windows

pyinstaller /source/app.py - - onefile -> app.exe



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**

Példa:

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

```
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))
```

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**

```
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))
```

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:

$\text{változó}_1, \dots, \text{változó}_n = \text{érték}_1, \dots, \text{érték}_n$

alma, körte, dió = 26, 15, 4

alma = körte = dió = 15

3. Alapvető adattípusok (*test.py*)

- **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()*

- **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()*

Adattípusok csoportosítása

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

4. Műveletek és operátorok

Aritmetikai műveletek:

#	Operátor	Művelet	Leírás	Példa	Eredmény
	+	Összeadás	Két érték összeadása	8 + 5	13
	-	Kivonás	Az első érték csökkentése a másodikkal	8 - 5	3
	*	Szorzás	Két érték szorzata	8 * 5	40
	/	Osztás	Az első érték osztva a másodikkal	8 / 5	1.6
	%	Maradékos osztás	Az osztás maradéka	8 % 5	3
	//	Egész osztás	Az osztás egész része	8 // 5	1
	**	Hatványozás	Az első érték a második hatványa	8 ** 5	32768

Összehasonlító operátorok:

==,

!=,

<,

>,

<=,

>=

Fontos a nullával osztás elkerülése!

Stringműveletek

Összefűzés:

```
x = "Alma" + "fa" - "Almafa"
```

Sokszorozás:

```
y = "3" + 2*" 2" - "3 2 2"
```

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: `szamok = [1,2,3,4,5]`

Tartalmazás vizsgálat:

A lista hossza:

A lista index alapján kiválasztott eleme:

Lista törlése:

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

List comprehension (lista kifejezés):

`if 3 in szamok: ...`

`n = len(szamok)`

`elem = szamok[i]; szamok[i]=elem`

`del szamok`

`new = szamok[i:j]`

`szmk = [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 = [i**2 for i in range(10) if i%2 == 0]`

`k = [i**2 if i%2 == 0 else i for i in range(10)]`

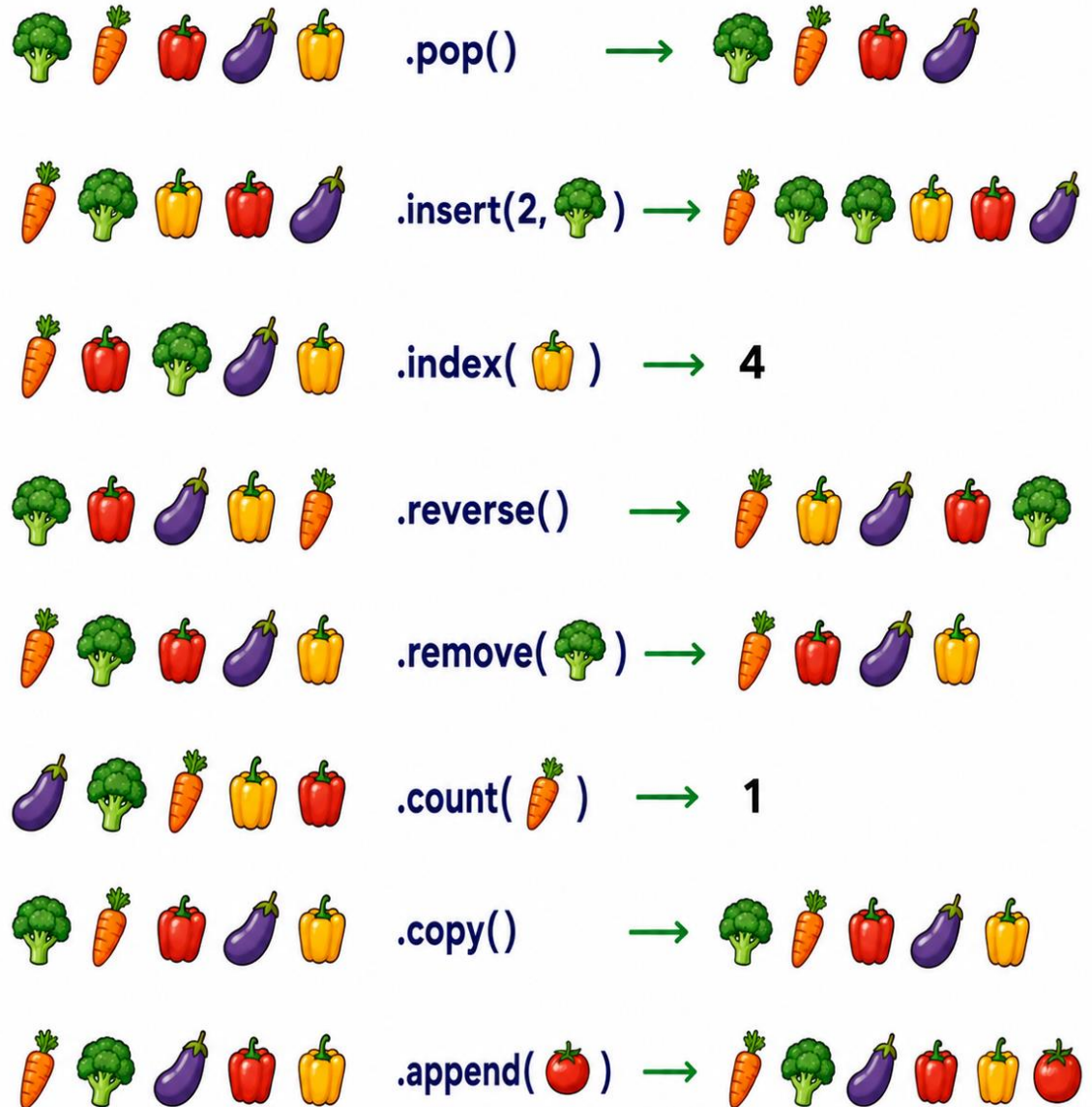
Zip: `m = [1, 2, 3, 4, 5]`
 `n = ['a', 'b', 'c', 'd', 'e']`
 `o = list(zip(m, n))`

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

Lista kezelése

```
l = [1,2,3,'4']  
print(l)  
  
l.append(10)  
  
print(l)  
  
l = l.append(10) # None  
  
print(l)
```

```
[1, 2, 3, '4']  
[1, 2, 3, '4', 10]  
None
```



Lista metódusok

Példa: `lista = [1, 3, 5, 7]`

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

Példák: `a = [1, 1, 1, 2, 2], b = {2, 4, 6}`
`a = set(a), b.add(10) {2, 4, 6, 10}`
`{1, 2}`

5. Alapvető ki- és bemeneti műveletek

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

```
Hi!***.###
```

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

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

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

print()

- **object(s)**: Azok az értékek vagy objektumok, amelyeket a `print()` függvény megjelenít. Egyszerre több elemet is kiírhatunk, ezeket vesszővel választjuk el.
- **sep**: Meghatározza, hogy milyen elválasztó kerüljön a kiírt objektumok közé. Ha nem adunk meg mást, a függvény szóközt (' ') használ.
- **end**: Meghatározza, milyen karakter vagy karakterlánc kerüljön a kiírás végére. Alapértelmezés szerint egy újsor karakter ('\n') zárja a kimenetet.
- **file**: Megadja azt a fájlt vagy fájlszerű objektumot, amelybe a `print()` a kimenetet írja. Ha ezt nem állítjuk be, a kimenet a szabványos kimenetre (`sys.stdout`), vagyis általában a konzolra kerül.
- **flush**: Ha értéke `True`, a kimenet nem várakozik a pufferben, hanem azonnal továbbításra kerül. Alapértelmezett értéke `False`.
- **Puffer** – A kimeneti puffer egy ideiglenes tárolóterület, amelyben a program a kiírandó adatokat összegyűjti, mielőtt azok megjelennének a képernyőn vagy egy fájlban. Ez hatékonyabb működést tesz lehetővé, mert nem minden kiírás történik azonnal. A `flush=True` használatával a puffer azonnal kiürül, így a kimenet késedelem nélkül megjelenik.

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

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 ($\sim$, $\&$, $|$, $\wedge$)
- Bináris balra és jobbra eltolás ($\ll$, $\gg$)

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

```
szám = 15
sz_jobb = szám >> 1          # Osztunk 2-vel ( // )
sz_bal = szám << 2           # Szorzunk 4-gyel
print(szám, sz_jobb, sz_bal)

# 15 7 60
```

- Műveleti azonosságok, ha b egy bit:

[illegible]
$$x = x \wedge y \qquad x \wedge = y$$
$$b \wedge 1 = \sim b \quad b \wedge 0 = b$$

- Példa:

egyes bitek
lekérdezése &-del:

```
if status_register & bit_mask:
    # A vizsgált bit értéke 1
else:
    # A vizsgált bit értéke 0

status_register = 26    # 0b00011010
bit_mask = 8            # 0b00001000

if status_register & bit_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

Example if statements:

1. Simple if

```
x = 10  
  
if x > 5:  
    print("x is greater than 5")
```

2. if - else

```
x = -3  
  
if x >= 0:  
    print("x is non-negative")  
else:  
    print("x is negative")
```

3. if - elif - else

```
score = 85  
if score >= 90:  
    print("Excellent")  
elif score >= 75:  
    print("Good")  
elif score >= 60:  
    print("Average")  
else:  
    print("Poor")
```

4. Multiple conditions (and)

```
age = 20  
has_ticket = True  
  
if age >= 18 and has_ticket:  
    print("Access granted")  
else:  
    print("Access denied")
```

5. Multiple conditions (or)

```
is_weekend = True  
is_holiday = False  
  
if is_weekend or is_holiday:  
    print("You can relax!")  
else:  
    print("Back to work!")
```

6. Nested if

```
password = "abc123"  
if len(password) >= 8:  
    if password.isalnum():  
        print("Strong password")  
    else:  
        print("Use only letters and digits")  
else:  
    print("Password too short")
```

A while és a for ciklusok

- 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

- Végtelen ciklus

```
while True:
    szoveg = input("Írj be valamit (kilépés: exit): ")

    if szoveg == "exit":
        print("Kilépés...")
        break

    print("Beírtad:", szoveg)
```

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(1, 6):      # ciklusfeltétel
    print(i, end=' -> ')  # ciklusmag
    print(i * '*')

print("A ciklus után:", i)
```

```
1 -> *
2 -> **
3 -> ***
4 -> ****
5 -> *****
A ciklus után: 5
```

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 használata:")
for szam in range(2, 8):
    if szam == 6:
        break
    print("Aktuális érték:", szam)

print("A ciklus véget ért.\n")

# continue példa
print("continue használata:")
for szam in range(2, 8):
    if szam == 6:
        continue
    print("Aktuális érték:", szam)

print("A ciklus véget ért.")
```

```
break használata:
Aktuális érték: 2
Aktuális érték: 3
Aktuális érték: 4
Aktuális érték: 5
A ciklus véget ért.
```

```
continue használata:
Aktuális érték: 2
Aktuális érték: 3
Aktuális érték: 4
Aktuális érték: 5
Aktuális érték: 7
A ciklus véget ért.
```

A *while*/*for* ciklusok és az *else* ág:

```
szamlalo = 3

while szamlalo <= 7:
    print(szamlalo)
    szamlalo += 1
else:
    print("A ciklus vége:", szamlalo)
    print(szamlalo * "* ")
```

```
3
4
5
6
7
A ciklus vége: 8
* * * * *
```


Függvények bevezetése, paraméterek, visszatérési értékek

Miért használunk függvényeket?

- Elkerülhetjük ugyanannak a kódrészletnek a többszöri leírását.
- A nagyobb programokat könnyebben kezelhető, kisebb részfeladatokra bonthatjuk (decomposition).
- A kód áttekinthetőbbé, újra felhasználhatóvá és könnyebben karbantarthatóvá válik.

Hol találunk a függvényeket?

- A Python beépített függvényei.
- Külső modulok és könyvtárak által biztosított függvények.
- Saját magunk által létrehozott függvények.

Függvények bevezetése, paraméterek, visszatérési értékek

Szintaxis:

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

Példa függvény:

```
def hi():  
    print("Hello!")
```

```
hi()  
Kimenet:  
„Hello”
```

Type – típus címkék

```
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 átadásra:

```
def osszeg(a, b):  
    print("Az összeg:", a + b)
```

```
osszeg(a=15, b=8)  
osszeg(b=20, a=5)
```

```
def szorzas(a, b=10):  
    print("A szorzat:", a * b)
```

```
szorzas(6, 4)  
szorzas(6)
```

```
szorzas(b=7)
```

```
# TypeError: szorzas() missing 1 required positional argument: 'a'
```

```
Az összeg: 23
```

```
Az összeg: 25
```

```
A szorzat: 24
```

```
A szorzat: 60
```

```
TypeError: szorzas() missing 1 required positional argument: 'a'
```

Mellékhatás és visszatérési érték : return

```
def atlag(a, b):  
    eredmeny = (a + b) / 2  
    print("Átlag:", eredmeny)  
  
atalag(18, 26)  
  
print("A program folytatódik...")
```

```
Átlag: 22.0  
A program folytatódik...
```

Láthatóság

```
def kiir_ertek():  
    print("A szám értéke:", szam)  
  
szam = 12  
  
kiir_ertek()  
print(szam)  
  
def szamitas():  
    eredmeny = 48  
  
szamitas()  
  
print("Az eredmény:", eredmeny)  
  
# NameError: name 'eredmeny' is not defined
```

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

Kimenet:

True

None

A szám értéke: 12

12

NameError: name 'eredmeny' is not defined

A None kulcsszó:

```
def keres_erteket(szamok, keresett):  
    for szam in szamok:  
        if szam == keresett:  
            return szam  
    return None  
  
adatok = [12, 25, 38, 47]  
  
eredmeny = keres_erteket(adatok, 30)  
  
if eredmeny is None:  
    print("A keresett érték nem található.")  
else:  
    print("Találat:", eredmeny)  
  
print(eredmeny + 10)
```

A keresett érték nem található.

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

A példa alapján:

- a függvény None értéket ad vissza, ha nincs találat;
- a None-t érdemes is None segítségével ellenőrizni;
- a None nem szám, ezért nem végezhetők vele számtani műveletek.

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]

Lista- kifejezések

```
# Hagyományos ciklus
for szam in range(5, 15):
    print(szam)

# Ugyanez list comprehension használatával
[print(szam) for szam in range(5, 15)]

# Páros számok listájának létrehozása
paros = [szam for szam in range(0, 1000000) if szam % 2 == 0]

# Lista létrehozása, majd összegzés
osszeg1 = sum([ertek**2 for ertek in range(100000)])

# Generátorkifejezés használata (nem készít teljes listát)
osszeg2 = sum(ertek**2 for ertek in range(100000))

print("Lista alapú összeg:", osszeg1)
print("Generátoros összeg:", osszeg2)

db = len(paros)
print("Páros számok darabszáma:", db)
```

Ciklus lista kifejezéssel

```
szavak = ["alma", "körte", "szilva", "barack"]

# Hagyományos megoldás: üres lista, majd append() használata.
nagybetus = []

for szo in szavak:
    nagybetus.append(szo.upper())

print(nagybetus)

# Ugyanez list comprehension segítségével.
nagybetus = [szo.upper() for szo in szavak]

print(nagybetus)
```

```
['ALMA', 'KÖRTE', 'SZILVA', 'BARACK']
['ALMA', 'KÖRTE', 'SZILVA', 'BARACK']
```

Az IF és a listakifejezés

```
# IF

negyzetek = []

for szam in range(1, 21):
    if szam % 3 == 0:
        negyzetek.append(szam ** 2)

# Ugyanez list comprehension segítségével.
negyzetek = [szam ** 2 for szam in range(1, 21) if szam % 3 == 0]

szinek = ["piros", "kék", "zöld", "sárga"]
```

```
# IF

szavak = ["alma", "körte", "szilva", "dió", "ananász"]

rovid_szavak = []

for szo in szavak:
    if len(szo) <= 5:
        rovid_szavak.append(szo)

# Ugyanez list comprehension segítségével.
rovid_szavak = [szo for szo in szavak if len(szo) <= 5]
```

Karakterek keresése - példaprogram

```
szoveg = "programozas Python nyelven"  
keresett = "o"  
  
darab = 0  
  
for karakter in szoveg:  
    if karakter != keresett:  
        continue  
    darab += 1  
  
print(f'A(z) "{keresett}" karakter {darab} alkalommal fordul elő.')
```

A(z) "o" karakter 3 alkalommal fordul elő.

Több feltétel alkalmazása

```
# Több egymás utáni feltétel:  
# minden feltételnek teljesülnie kell.
```

```
szamok = [5, 12, 18, 21, 24, 35, 42, 51, 60]
```

```
eredmeny = [szam for szam in szamok  
             if szam > 10  
             if szam % 2 == 0  
             if szam < 50]
```

```
print(eredmeny)
```

```
[12, 18, 24, 42]
```

```
# Több egymás utáni feltétel:  
# minden feltételnek teljesülnie kell.
```

```
szavak = ["alma", "Autó", "eper", "Ősz", "barack", "Eper", "citrom"]
```

```
eredmeny = [szo for szo in szavak  
             if len(szo) >= 5  
             if szo[0].islower()  
             if "a" in szo]
```

```
print(eredmeny)
```

```
['barack']
```


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.

szamok = [8, 15, 22, 31, 46, 57]

kategoriak = []

for szam in szamok:
    if szam % 2 == 0:
        kategoriak.append("Páros")
    else:
        kategoriak.append("Páratlan")

print(kategoriak)

kategoriak = ["Páros" if szam % 2 == 0 else "Páratlan"
              for szam in szamok]

print(kategoriak)
```

```
['Páros', 'Páratlan', 'Páros', 'Páratlan', 'Páros', 'Páratlan']
['Páros', 'Páratlan', 'Páros', 'Páratlan', 'Páros', 'Páratlan']
```



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!

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)

Kérdések

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

Hibakezelés

Hibák az adatokban és hibák a kódban – a tesztelés kötelező

- **Adat hibák:**

helyes kód, hibás adatbevitel

```
x = int(input(Input érték: ,))
```

Input érték: 120

ValueError: invalid literal for int(): , 11'

2. Hibakezelés

- Kódolási hibák:
- hibakeresés kódelemzéssel, debugger használatával, AI eszközökkel – példa!

```
numbers = [10, 20, 30]

value = numbers[5]

# IndexError: list index out of range
```

```
length = 25

area = width * length

# NameError: name 'width' is not defined
```

További példák:

```
weight = float(input("Enter the package weight (kg): "))

if weight <= 0:
    raise ValueError("Weight must be greater than zero.")

shipping_cost = weight * 2.5
print("Shipping cost:", shipping_cost)
```

```
Enter the package weight (kg): 0
ValueError: Weight must be greater than zero.
```

```
month = int(input("Enter a month (1-12): "))

if month < 1 or month > 12:
    raise ValueError("Invalid month.")

print("Selected month:", month)
```

```
Enter a month (1-12): 15
ValueError: Invalid month.
```

```
for temperature in [12, 18, 25, 31, 16]:
    if temperature >= 20:
        print(temperature, "Warm")
    else:
        print(temperature, "Cold")
```

```
12 Cold
18 Cold
25 Warm
31 Warm
16 Cold
```

```
price = 19.99
quantity = "4"

total = price * quantity

print(total)

# TypeError: can't multiply sequence by non-int of type 'float'
```

```
temperature = 22
message = " °C"

print(temperature + message)

# TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

```
def average_speed():
    distance = float(input("Enter the distance (km): "))
    time = float(input("Enter the travel time (hours): "))

    speed = distance / time

    print("Average speed:", speed, "km/h")

average_speed()
```

```
Enter the distance (km): 120
Enter the travel time (hours): 0

ZeroDivisionError: float division by zero
```

Kivételezés: try – except blokkok

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

```
try:  
    filename = input("Enter a file name: ")  
    file = open(filename, "r")  
  
    line = int(file.readline())  
    print("First number:", line)  
  
    file.close()  
  
except FileNotFoundError:  
    print("The file does not exist!")  
  
except ValueError:  
    print("The first line is not an integer!")  
  
except Exception:  
    print("An unexpected error occurred!")  
  
print("Program finished.")
```



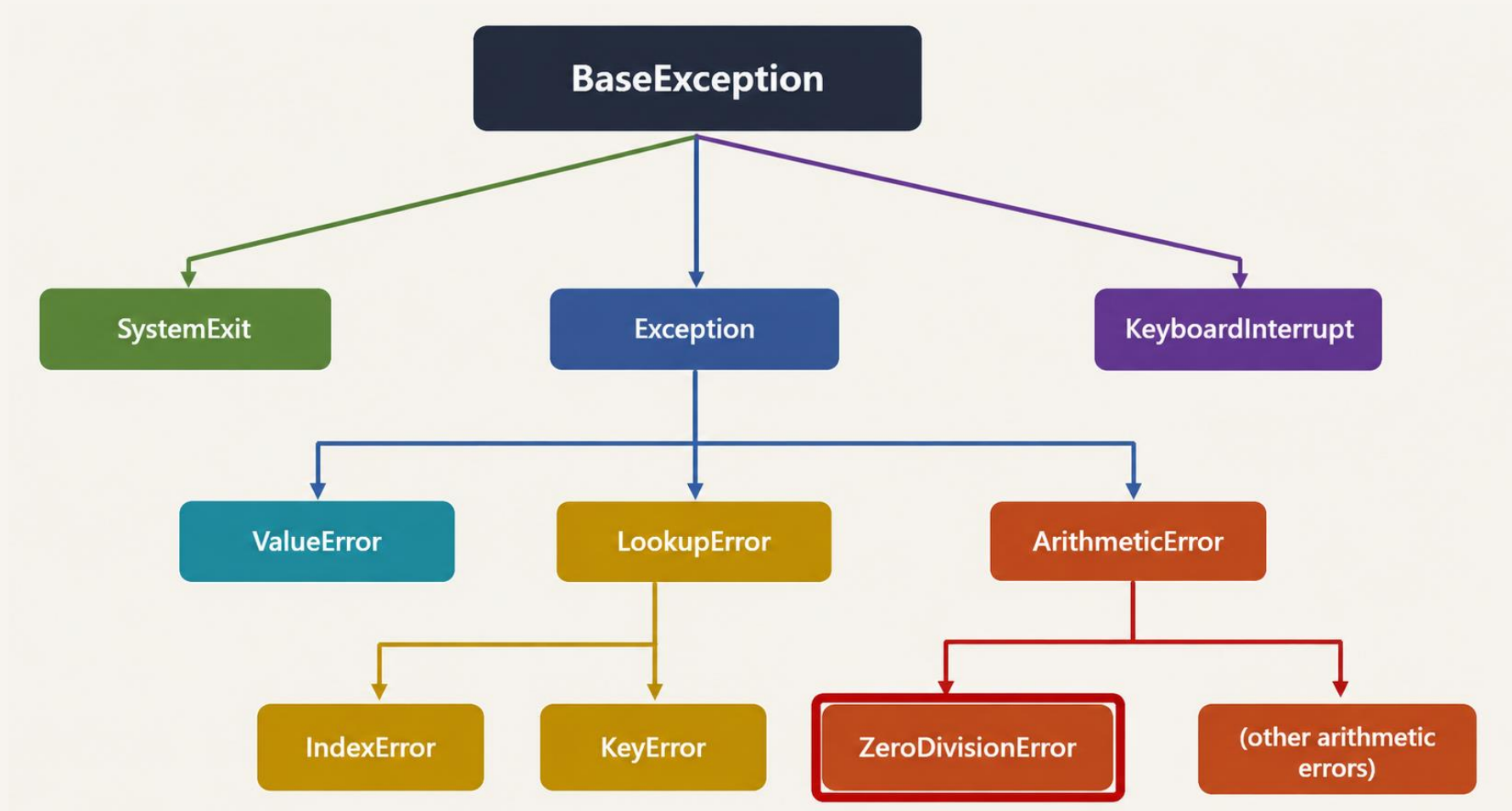
```
Enter a file name: numbers.txt  
First number: 125  
Program finished.
```

```
Enter a file name: numbers.txt  
First number: 125  
Program finished.
```

```
Enter a file name: text.txt  
The first line is not an integer!  
Program finished.
```

Kivételek - Exceptions

A Python3 63 beépített kivételt definiál, amelyek egy **fa hierarchiában** vannak:



Kivételek kiváltása: a raise utasítás

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

```
def check_temperature(temp):  
    if temp < -50 or temp > 60:  
        raise ValueError("Temperature is out of the valid range.")  
  
try:  
    check_temperature(120)  
  
except ValueError:  
    print("Invalid temperature value.")  
  
print("Program finished.")
```

```
Invalid temperature value.  
Program finished.
```

```
The number of items cannot be zero.  
None  
Program continues...
```

```
def atlag(osszeg, darab):  
    try:  
        return osszeg / darab  
  
    except ZeroDivisionError:  
        print("The number of items cannot be zero.")  
  
    return None  
  
print(atlag(150, 0))  
  
print("Program continues...")
```


A raise utasítás

```
def check_score(score):  
    if score < 0 or score > 100:  
        print("Invalid score!")  
        raise ValueError("Score must be between 0 and 100.")  
  
    return "Score accepted."  
  
try:  
    print(check_score(120))  
  
except ValueError:  
    print("The exception has been handled.")  
  
print("Program finished.")
```

```
Invalid score!  
The exception has been handled.  
Program finished.
```

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**
Osztásnál vagy maradék (modulo) műveletnél 0 második argumentum esetében.

```
student = {  
    "name": "Alice",  
    "age": 21  
}  
  
try:  
    print(student["grade"])  
  
except Exception as error:  
    print("An error occurred!")  
    print(error)
```

An error occurred!
'grade'

```
try:  
    file = open("report.txt", "r")  
    print(file.read())  
  
except Exception as error:  
    print("Unable to open the file.")  
    print(error)
```

Unable to open the file.
[Errno 2] No such file or directory: 'report.txt'

```
try:
```

```
    print("10" + 20)
```

```
except Exception as error:
```

```
    print(error)
```

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

```
def test(a, b=2, c):
```

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

SyntaxError: non-default argument follows default argument

Az assert kifejezés

```
age = int(input("Enter your age: "))

assert age >= 18, "You must be at least 18 years old."

print("Access granted.")
```

```
while True:
    age = int(input("Enter your age: "))

    try:
        assert age >= 18
        break

    except AssertionError:
        print("The age must be at least 18.")

print("Access granted.")
```

```
Enter your age: 21
Access granted.
```

```
Enter your age: 16
```

```
Traceback (most recent call last):
```

```
...
```

```
assert age >= 18, "You must be at least 18 years old."
AssertionError: You must be at least 18 years old.
```

```
Enter your age: 15
The age must be at least 18.
```

```
Enter your age: 17
The age must be at least 18.
```

```
Enter your age: 20
Access granted.
```

```
numbers = [8, 15, 21, 34]

index = int(input("Enter an index: "))

assert 0 <= index < len(numbers), "Index is out of range."

print(numbers[index])
```

A **yield** kulcsszót generátor függvényekben használjuk

- 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 read_logs():  
    logs = [  
        "INFO: Server started",  
        "WARNING: High memory usage",  
        "ERROR: Database connection failed",  
        "INFO: Server stopped"  
    ]  
  
    for entry in logs:  
        yield entry  
  
for log in read_logs():  
    print(log)
```

```
INFO: Server started  
WARNING: High memory usage  
ERROR: Database connection failed  
INFO: Server stopped
```

```
def sensor_values():  
    value = 20  
  
    while value <= 30:  
        yield value  
        value += 2  
  
for temperature in sensor_values():  
    print(f"{temperature} °C")
```

```
20 °C  
22 °C  
24 °C  
26 °C  
28 °C  
30 °C
```

Yield

```
def fibonacci(limit):  
    a, b = 0, 1  
  
    while a <= limit:  
        yield a  
        a, b = b, a + b  
  
for number in fibonacci(50):  
    print(number)
```

```
0  
1  
1  
2  
3  
5  
8  
13  
21  
34
```

```
def read_lines(filename):  
    with open(filename, "r") as file:  
        for line in file:  
            yield line.strip()  
  
for line in read_lines("data.txt"):  
    print(line)
```

```
Processing page 1  
Processing page 2  
Processing page 3  
Processing page 4  
Processing page 5
```


Függvények: paraméter átadás

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

```
def apply_discount(price, percent):  
    price = price * (1 - percent / 100)  
    print("Inside function:", price)  
  
product_price = 25000  
  
apply_discount(product_price, 20)  
  
print("Outside function:", product_price)
```

```
Inside function: 20000.0  
Outside function: 25000
```

```
def add_product(products):  
    products = products + ["Monitor"]  
    print("Inside function:", products)  
  
inventory = ["Keyboard", "Mouse", "Laptop"]  
  
add_product(inventory)  
  
print("Outside function:", inventory)
```

```
Inside function: ['Keyboard', 'Mouse', 'Laptop', 'Monitor']  
Outside function: ['Keyboard', 'Mouse', 'Laptop', 'Monitor']
```

***args – változó számú pozicionális argumentum**

- A * operátor azt jelzi, hogy a függvény tetszőleges számú pozicionális paramétert képes fogadni.
- Az átadott értékek a függvényen belül egy **tuple** (rendezett, nem módosítható adatszerkezet) formájában érhetők el.
- Ennek segítségével nem szükséges előre meghatározni a paraméterek pontos számát.

****kwargs – változó számú kulcsszó-argumentum**

- A ** operátorral olyan függvény írható, amely tetszőleges számú névvel ellátott (kulcsszavas) argumentumot fogad.
- Az argumentumok **kulcs-érték párokként** kerülnek átadásra.
- A függvényen belül a **kwargs egy **szótárként (dict)** tárolja ezeket az adatokat: a kulcsok az argumentumok nevei, az értékek pedig a hozzájuk tartozó értékek.
- Ez a megoldás különösen hasznos, ha a paraméterek köre előre nem ismert vagy rugalmasan szeretnénk kezelni azokat.

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

```
def register_product(product_name, *prices, **details):  
    print(f"Product: {product_name}")  
    print("Available prices:", prices)  
    print("Additional details:", details)
```

```
register_product(  
    "Laptop",  
    899, 949, 999,  
    brand="Lenovo",  
    ram="16 GB",  
    storage="512 GB SSD"  
)
```

```
Product: Laptop  
Available prices: (899, 949, 999)  
Additional details: {'brand': 'Lenovo', 'ram': '16 GB', 'storage': '512 GB SSD'}
```

```
def log_event(event, *messages, **metadata):  
    print(f"Event: {event}")  
    print("Messages:", messages)  
    print("Metadata:", metadata)
```

```
log_event(  
    "LOGIN",  
    "User authenticated",  
    "Session created",  
    user="alice",  
    ip="192.168.1.15",  
    status="success"  
)
```

```
Event: LOGIN  
Messages: ('User authenticated', 'Session created')  
Metadata: {'user': 'alice', 'ip': '192.168.1.15', 'status': 'success'}
```

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

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

```
6  
50
```

```
def profile(**kwargs):  
    for key, value in kwargs.items():  
        print(f"{key.capitalize()}: {value}")  
  
profile(name="Alice", age=30, city="London")
```

```
Name: Alice  
Age: 30  
City: London
```

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

```
2 ** 5      # 32  
pow(2, 5)   # 32
```

```
sum1 = lambda a, b: a + b  
print(sum1(2, 5))
```

7

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

7

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

32

lambda argument1, argument2, ... : kifejezés

```
# Celsius → Fahrenheit
print((lambda c: c * 9 / 5 + 32)(25))
# 77.0

# Téglalap kerülete
print((lambda width, height: 2 * (width + height))(8, 5))
# 26

# Átlag számítása
print((lambda x, y, z: (x + y + z) / 3)(12, 18, 24))
# 18.0

# Legnagyobb szám kiválasztása
print((lambda a, b: a if a > b else b)(17, 29))
# 29

# Kör területe
print((lambda r: 3.14159 * r ** 2)(4))
# 50.26544
```

```
# ÁFA hozzáadása (27%)
print((lambda price: price * 1.27)(10000))
# 12700.0

# Kedvezmény levonása (15%)
print((lambda price: price * 0.85)(8000))
# 6800.0

# Másodpercek → percek
print((lambda seconds: seconds / 60)(450))
# 7.5

# Pozitív vagy negatív?
print((lambda x: "Positive" if x >= 0 else "Negative")(-8))
# Negative

# Téglalap területe
print((lambda width, height: width * height)(9, 6))
# 54
```

További példák:

```
def evaluate_function(values, function):  
    for x in values:  
        print('f(', x, ') = ', function(x), sep='')  
  
evaluate_function(  
    [x for x in range(-3, 3)],  
    lambda x: 2 * x**3 - x + 1  
)
```

f(-3)= -50
f(-2)= -13
f(-1)= 0
f(0)= 1
f(1)= 2
f(2)= 15

```
def evaluate_function(values, function):  
    for x in values:  
        print('h(', x, ') = ', function(x), sep='')  
  
evaluate_function(  
    [x for x in range(-2, 5)],  
    lambda x: x**2 + 2 * x - 3  
)
```

h(-2)= -3
h(-1)= -4
h(0)= -3
h(1)= 0
h(2)= 5
h(3)= 12
h(4)= 21

A lambda alkalmazása magasabb rendű függvényekben

A **map** függvény: egy függvényt alkalmaz minden elemre egy iterálható objektumban:

```
temperatures = [0, 10, 20, 30, 40]

fahrenheit = list(map(lambda c: c * 9 / 5 + 32, temperatures))

print(fahrenheit)
```

```
[32.0, 50.0, 68.0, 86.0, 104.0]
```

A **filter** függvény: lambda függvényt használ arra, hogy kiválassza azokat az elemeket, amelyek egy adott feltételnek megfelelnek:

```
ages = [12, 17, 18, 21, 15, 30, 42]

adults = list(filter(lambda age: age >= 18, ages))

print(adults)
```

```
[18, 21, 30, 42]
```


A lambda alkalmazása magasabb rendű függvényekben

```
cities = ["london", "paris", "rome", "berlin"]  
  
capitalized = list(map(lambda city: city.capitalize(), cities))  
  
print(capitalized)
```

```
['London', 'Paris', 'Rome', 'Berlin']
```

```
words = ["apple", "kiwi", "banana", "fig", "orange"]  
  
short_words = list(filter(lambda word: len(word) <= 4, words))  
  
print(short_words)
```

```
['kiwi', 'fig']
```

Beépített függvények

A **beépített függvények (built-in functions)** a Python nyelv alapértelmezett eszközkészletének részét képezik. Ezek a függvények különböző feladatok elvégzésére használhatók anélkül, hogy további modulokat vagy könyvtárakat kellene importálni. A leggyakrabban használt beépített függvények az alábbi csoportokba sorolhatók:

- **Matematikai függvények:** `abs()`, `round()`, `min()`, `max()`, `sum()`
- **Típuskezelő függvények:** `type()`, `int()`, `str()`, `bool()`, `float()`, `list()`, `dict()`
- **Iterációt támogató függvények:** `range()`, `map()`, `filter()`, `zip()`, `sorted()`, `reversed()`, `enumerate()`
- **Bemenet–kimenet és fájlkezelés:** `input()`, `print()`, `open()`
- **Funkcionális programozáshoz kapcsolódó függvények:** `lambda`, `eval()`, `exec()`
- **Kivételkezeléshez használható utasítások:** `raise`, `assert`
- **Objektumok vizsgálatára és kezelésére szolgáló függvények:** `dir()`, `vars()`, `id()`, `classmethod()`, `staticmethod()`
- A Python környezetben elérhető összes beépített függvény és objektum megtekinthető a `dir(__builtins__)` függvényhívással. A beépített elemek tartalma közvetlenül is kiíratható például a `print(__builtins__)` utasítással.

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

A függvény egy önálló, újrafelhasználható kódrészlet, amely egy meghatározott feladatot hajt végre. Bemeneti adatokat (paramétereket) fogadhat, majd eredményt adhat vissza. A függvények segítségével a program könnyebben áttekinthetővé és karbantarthatóvá válik.

```
def circle_area(radius):  
    return 3.14159 * radius ** 2  
  
print(circle_area(5))
```

Modul (Module)

- A modul egy Python fájl (.py), amely függvényeket, változókat, osztályokat vagy egyéb Python-kódot tartalmaz. A modulok célja a kapcsolódó funkciók egy helyen történő összegyűjtése és más programokban való felhasználása.

```
import math  
  
print(math.sqrt(64))  
print(math.factorial(5))
```

Csomag (Package)

- A csomag olyan könyvtár, amely több modult – és akár további csomagokat (alcsomagokat) – foglal magába. A package-ek segítségével a nagyobb projektek logikusan szervezhetők, és a kód könnyebben újrafelhasználható.

```
from urllib import request  
  
response = request.urlopen("https://example.com")
```

A Python package-ek

- A Python Package Index (PyPI) a Python hivatalos online csomagtára, ahonnan a legtöbb külső könyvtár és csomag telepíthető. Több tízezer nyílt forráskódú csomagot tartalmaz, amelyek számos területen – például webfejlesztésben, adatelemzésben, mesterséges intelligenciában vagy automatizálási feladatokban – nyújtanak kész megoldásokat a fejlesztők számára.
- **Package install?**
A Pythonban a legelterjedtebb eszköz a package-ek telepítéséhez a **pip/pip3** nevű csomagkezelők.

Példák:

```
# Befektetések végösszegének számítása lambda függvényekkel

alaptoke = 100000

3_szazalek = lambda osszeg: osszeg * 1.03
5_szazalek = lambda osszeg: osszeg * 1.05
8_szazalek = lambda osszeg: osszeg * 1.08

egyszeru_kamat = lambda osszeg, ev, kamat: osszeg * (1 + kamat * ev)
kamos_kamat = lambda osszeg, ev, kamat: osszeg * (1 + kamat) ** ev

evok = range(1, 6)          # 1...5 év

ertek_3 = [round(3_szazalek(alaptoke), 2) for _ in evok]
ertek_5 = [round(5_szazalek(alaptoke), 2) for _ in evok]
ertek_8 = [round(8_szazalek(alaptoke), 2) for _ in evok]

egyszeru = [
    round(egyszeru_kamat(alaptoke, ev, 0.06), 2)
    for ev in evok
]

kamos = [
    round(kamos_kamat(alaptoke, ev, 0.06), 2)
    for ev in evok
]

print("3% annual growth:", ertek_3)
print("5% annual growth:", ertek_5)
print("8% annual growth:", ertek_8)
print("Simple interest (6%):", egyszeru)
print("Compound interest (6%):", kamos)
```

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!

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?

Az objektumorientált programozás (OOP) olyan programozási szemlélet, amelyben a program egymással együttműködő objektumokból épül fel. Célja az átláthatóbb, könnyebben bővíthető és újrahasznosítható kód készítése.

Az OOP négy alapelve:

- Egységbezárás (Encapsulation):** Az adatok és a hozzájuk tartozó műveletek egy osztályba szervezése.
- Absztrakció (Abstraction):** Csak a szükséges funkciók megjelenítése, a részletek elrejtése.
- Öröklődés (Inheritance):** Meglévő osztályok tulajdonságainak és metódusainak átvétele, bővítése.
- Polimorfizmus (Polymorphism):** Azonos metódusnév eltérő működése különböző objektumok esetén.

Bevezetés az Objektumorientált programozásba

- Az objektumorientált programozás (Object-Oriented Programming, OOP) olyan programozási szemlélet, amelyben a program egymással együttműködő objektumokból épül fel. Az objektumok adatokat (attribútumokat) és az azokhoz tartozó műveleteket (metódusokat) egyetlen egységben kezelik.
- Az OOP célja, hogy a programok áttekinthetőbbek, könnyebben karbantarthatók és egyszerűbben bővíthetők legyenek. Működésének alapját négy fő elv alkotja: **egységbezárás (encapsulation)**, **absztrakció (abstraction)**, **öröklődés (inheritance)** és **polimorfizmus (polymorphism)**.

Az OOP előnyei:

- 1. Modularitás:** A program kisebb, önálló osztályokra bontható, amelyek külön-külön is könnyen fejleszthetők és módosíthatók.
- 2. Újrahasznosíthatóság:** Az elkészített osztályok és objektumok más projektekben vagy a program több részében is felhasználhatók.
- Karbantarthatóság:** A jól felépített osztályszerkezet megkönnyíti a program bővítését, javítását és továbbfejlesztését.
- 4. Rugalmas tervezés:** Az öröklődés és a polimorfizmus segítségével új funkcionális adható a programhoz anélkül, hogy a meglévő kódot jelentősen módosítani kellene.

A class működése – type

Gyakorlati példák, forráskód elemzés!

Egységbezárás (Encapsulation)

```
class Warehouse:

    def __init__(self, quantity):
        self.__quantity = quantity    # privát attribútum

    def add_items(self, amount):
        self.__quantity += amount

    def remove_items(self, amount):
        if amount <= self.__quantity:
            self.__quantity -= amount
        else:
            print("Not enough items in stock.")

    def show_quantity(self):
        print("Current stock:", self.__quantity)
```

Az **egységbezárás (encapsulation)** során az objektum adatai és a rajtuk végzett műveletek egy osztályba kerülnek. A belső adatok közvetlen módosítása korlátozható, így az objektum állapota biztonságosan kezelhető.

Például a **Warehouse** osztályban a raktár aktuális készlete egy privát attribútumban (`__quantity`) tárolható.

A készletet csak az `add_items()` és `remove_items()` metódusokon keresztül lehet módosítani, míg a `show_quantity()` metódus megjeleníti az aktuális készletet.

A `__quantity` attribútum kívülről közvetlenül nem érhető el, ami megakadályozza a véletlen vagy szabálytalan módosításokat.

Az absztrakció (Abstraction)

Absztrakció (Abstraction): Az absztrakció lényege, hogy az objektum csak a használathoz szükséges funkciókat teszi elérhetővé, miközben a belső működés részletei rejtve maradnak. Ez egyszerűbbé és átláthatóbbá teszi a programok használatát.

Adatbázis-kezelés: Egy SQL-lekérdezést (SELECT) futtatunk anélkül, hogy ismernénk az adatbázismotor belső működését vagy az adatok fizikai tárolását.

Felhőalapú tárhely: Egy fájlt feltöltünk egy felhőszolgáltatásba, de nem kell tudnunk, hogy melyik szerveren vagy milyen módon tárolják azt.

Titkosítási könyvtár: Egy függvénnyel titkosítunk adatokat, miközben a használt kriptográfiai algoritmus részleteit nem kell ismernünk.

Az öröklődés (Inheritance)

```
class Employee:

    def __init__(self, name):
        self.name = name

    def calculate_salary(self):
        return "Salary calculation"

class FullTimeEmployee(Employee):

    def calculate_salary(self):
        return f"{self.name}: Monthly salary"

class HourlyEmployee(Employee):

    def calculate_salary(self):
        return f"{self.name}: Hourly wage"
```

Az **öröklődés (Inheritance)** lehetővé teszi, hogy egy új osztály átvegye egy meglévő osztály tulajdonságait és metódusait, majd szükség szerint kibővítse vagy módosítsa azokat.

A példában a **FullTimeEmployee** és az **HourlyEmployee** osztály az **Employee** osztályból származik, de mindkettő saját módon valósítja meg a `calculate_salary()` metódust.

Ez egyben a **polimorfizmus (Polymorphism)** példája is: ugyanaz a metódusnév különböző osztályokban eltérő működést valósít meg.

A polimorfizmus (Polymorphism)

```
class Notification:

    def send(self):
        pass

class EmailNotification(Notification):

    def send(self):
        return "Sending notification via e-mail."

class SmsNotification(Notification):

    def send(self):
        return "Sending notification via SMS."
```

A **polimorfizmus (Polymorphism)** azt jelenti, hogy azonos nevű metódus különböző osztályokban eltérő működést valósíthat meg. Ennek köszönhetően ugyanaz a metódushívás az objektum típusától függően más eredményt ad.

A példában az **EmailNotification** és az **SmsNotification** osztály a **Notification** alaposztályból származik, és mindkettő saját módon valósítja meg a `send()` metódust. Ez lehetővé teszi, hogy különböző értesítési típusokat egységes felületen kezeljünk.

Osztályok és Objektumok

```
class Sensor:                                # class definition

    def __init__(self, sensor_id):
        self.sensor_id = sensor_id

temperature_sensor = Sensor("T-101")         # object creation

print(temperature_sensor.sensor_id)
```

Az **osztály (class)** egy sablon, amely meghatározza, hogy egy objektum milyen adatokat (attribútumokat) és műveleteket (metódusokat) tartalmazhat. Az **objektum (instance)** az osztály egy konkrét példánya, amely saját adatokkal rendelkezik.

A példában a **Sensor** osztály egy érzékelőt ír le, míg a `temperature_sensor` ennek egy példánya, amelynek azonosítója **"T-101"**.

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 Product:

    def __init__(self, product_id, price):          # constructor
        self.product_id = product_id              # attribute
        self.price = price                        # attribute

    def get_price(self):                            # method
        return f"Price: ${self.price:.2f}"
```

- **osztály (class)** → **Product** **konstruktor (constructor)** → `__init__()` **attribútumok (attributes)** → `product_id`, `price` **metódus (method)** → `get_price()`

Osztályok

```
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")
print(book1.book_info())
print(book2.book_info())

# Cím: Egri csillagok, Szerző: Gárdonyi Géza
```

Az osztályok és objektumok előnyei

- **Moduláris felépítés:** Az osztályok segítségével a program jól elkülöníthető részekre bontható, ami átláthatóbb kódszerkezetet eredményez.
- **Kód újrafelhasználása:** Az egyszer elkészített osztályok több projektben vagy a program különböző részeiben is felhasználhatók.
- **Könnyebb karbantartás:** Az osztályok módosítása vagy bővítése általában csak egy helyen igényel változtatást, így a program fejlesztése egyszerűbb és biztonságosabb.

Konstruktorok és destruktorok

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

```
class Logger:  
  
    def __init__(self, log_file):  
        self.log_file = log_file  
  
    def __del__(self):  
        print(f"Closing log file: {self.log_file}")
```

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.

Objektum létrehozása:

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.

Példák

```
class Employee:

    def __init__(self, employee_id, department):
        self.employee_id = employee_id    # Employee identifier
        self.department = department      # Department

    def info(self):
        return f"Employee {self.employee_id} works in {self.department}."
```

```
server1 = Server("web01", "192.168.1.10")
server2 = Server("db01", "192.168.1.20")

print(server1.status())
# Output: web01 is available at 192.168.1.10.

print(server2.status())
# Output: db01 is available at 192.168.1.20.
```

```
class Server:

    def __init__(self, hostname, ip_address):
        self.hostname = hostname          # Server name
        self.ip_address = ip_address      # IP address

    def status(self):
        return f"{self.hostname} is available at {self.ip_address}."
```

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"Account for {self.owner} opened with a balance of ${self.balance}.")

    def deposit(self, amount):
        self.balance += amount
        print(f"${amount} deposited. New balance: ${self.balance}.")

    def withdraw(self, amount):
        if amount > self.balance:
            print("Insufficient funds.")
        else:
            self.balance -= amount
            print(f"${amount} withdrawn. New balance: ${self.balance}.")
```

```
class BankAccount:

    def __init__(self, owner, initial_balance):
        self.owner = owner
        self.balance = initial_balance
        print(f"Bank account created for {self.owner}. Initial balance: ${self.balance}.")

    def deposit(self, amount):
        self.balance += amount
        print(f"Deposit successful. Current balance: ${self.balance}.")

    def withdraw(self, amount):
```

Példa konstruktorokra és destruktorokra - BankAccount osztály:

```
def __del__(self):  
    print(f"Account belonging to {self.owner} has been closed.")  
  
account = BankAccount("Anna", 100000)  
# Output: Bank account created for Anna. Initial balance: $100000.  
  
account.deposit(30000)  
# Output: Deposit successful. Current balance: $130000.  
  
account.withdraw(5000)  
# Output: Withdrawal successful. Current balance: $125000.  
  
# When the program terminates (or the object is destroyed),  
# the destructor is called:  
# Output: Account belonging to Anna has been closed.
```

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 osztályok és objektumok **attribútumokkal** és **metódusokkal** rendelkeznek. Az attribútumok az objektum állapotát, a metódusok pedig annak működését írják le.

Az **osztályattribútumok** az osztályhoz tartoznak, ezért az osztály összes példánya közösen használja ugyanazt az értéket. Ezeket az osztály törzsében definiáljuk, nem az `__init__()` konstruktorban.

Példa: egy Circle osztály PI attribútuma minden kör esetén azonos, míg a radius objektumattribútum, ezért minden példány saját sugárértékkel rendelkezhet.

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: a Computer osztályban a márka és a típus objektum attribútumok:

```
class Computer:

    def __init__(self, manufacturer, model):
        self.manufacturer = manufacturer    # Object attribute
        self.model = model                  # Object attribute

    def description(self):
        return f"This is a {self.manufacturer} {self.model}."

computer1 = Computer("Dell", "XPS 15")
computer2 = Computer("Lenovo", "ThinkPad X1")

print(computer1.description())
# Output: This is a Dell XPS 15.

print(computer2.description())
# Output: This is a Lenovo ThinkPad X1.
```

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
```

Osztálymetódusok és objektummetódusok

```
class Person:

    count = 0                # Class attribute

    def __init__(self, name):
        self.name = name

        Person.count += 1    # Increment the class attribute

    @classmethod
    def display_count(cls):
        return f"Number of people: {cls.count}"

person1 = Person("Anna")
person2 = Person("John")

print(Person.display_count())

# Output: Number of people: 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”.

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:

- **Osztályattribútum:** Az osztályhoz tartozó változó, amelyet az osztály összes példánya közösen használ.
- **Objektumattribútum:** Az egyes objektumok saját adatait tároló változó, amely minden példánynál eltérő lehet.
- **Objektummetódus:** Olyan metódus, amely az adott objektum attribútumaival dolgozik, és azok állapotát is módosíthatja.
- **Osztálymetódus:** Az osztályhoz tartozó metódus, amely az osztályszintű adatokat kezeli, és nem egy konkrét objektumhoz kapcsolódik.
- **Statikus metódus:** Az osztályhoz tartozó segédfüggvény, amely nem használ sem objektum-, sem osztályszintű adatokat.

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 (inheritance)** lehetővé teszi, hogy egy új osztály egy meglévő osztályra épüljön.
- A gyermekosztály automatikusan átveszi a szülőosztály attribútumait és metódusait, amelyeket szükség szerint kibővíthet vagy felülírhat.

Szülőosztály és gyermekosztály

- **Szülőosztály (Parent Class):** amelyből más osztályok örökölhetnek.
- **Gyermekeosztály (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} kommunikál"
```

Szülőosztály és gyermekosztály

```
class UFO(Spacecraft):  
  
    def action(self):  
        return f"{self.name} is scanning the planet."  
  
class AlienShip(Spacecraft):  
  
    def action(self):  
        return f"{self.name} is launching a drone."
```

```
ufo1 = UFO("Explorer-X")  
ufo2 = AlienShip("Predator-Z")  
  
print(ufo1.action())  
# Output: Explorer-X is scanning the planet.  
  
print(ufo2.action())  
# Output: Predator-Z is launching a drone.
```

Az öröklődés előnyei:

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.

Egyszerű bővítés: A gyermekosztályok könnyen hozzáadhatnak új tulajdonságokat vagy módosíthatják a szülőosztály viselkedését anélkül, hogy megváltoztatnánk a szülőosztály kódját.

Moduláris tervezés: Az osztályokat hierarchiában szervezhetjük, ahol az általános funkciókat a szülőosztály, míg a specifikus funkciókat a gyermekosztály tartalmazza.

A leszármazott metódusai

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

Az öröklődés előnyei:

```
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.

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.

```
class NetworkDevice:

    def __init__(self, hostname):
        self.hostname = hostname

    def connect(self):
        return f"{self.hostname} connected."

class Router(NetworkDevice):

    def connect(self):
        return f"{self.hostname} connected as a router."

router = Router("Core-R1")
print(router.connect())

# Output: Core-R1 connected as a router.
```

```
class Document:

    def __init__(self, title):
        self.title = title

    def export(self):
        return f"Exporting '{self.title}'."

class PDFDocument(Document):

    def export(self):
        return f"Exporting '{self.title}' as PDF."

document = PDFDocument("Annual Report")
print(document.export())

# Output: Exporting 'Annual Report' as PDF.
```

Példa

```
class CreditCard:

    def __init__(self, name, number, limit = 3000, bank = "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
```

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)
```

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
```


Köszönöm a figyelmet!

Python kurzus – 6. előadás

Tárgyfelelős: Dr. Tejfel Máté

Előadó: Dr. Király Roland

ELTE IK



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.tet.aos/ons:qmikes)

def oneion():
    enenomers = u1
    def selfl:
        """kenepond:cosi: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 s.cens.60RWUMCE0GWS1&
            return eop

Graohsust = "tanP"
def _get6asch():
    print(fae1&6ist1npexssalirew&#)

    return &ophsito)
```

Python OOP II.

Haladó koncepciók és szoftver-architektúrák.

POLIMORFIZMUS | TÖBBSZÖRÖS ÖRÖKLŐDÉS | DUNDER METÓDUSOK | TERVEZÉSI MINTÁK

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

Polimorfizmus

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 Printer:

    def print_document(self):
        pass

class InkjetPrinter(Printer):

    def print_document(self):
        return "Printing with an inkjet printer."

class LaserPrinter(Printer):

    def print_document(self):
        return "Printing with a laser printer."

class PDFPrinter(Printer):

    def print_document(self):
        return "Saving document as PDF."

printers = [InkjetPrinter(), LaserPrinter(), PDFPrinter()]

for printer in printers:
    print(printer.print_document())

# Output:
# Printing with an inkjet printer.
# Printing with a laser printer.
# Saving document as PDF.
```

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 létrehozása absztrakt osztállyal:

Ez a példa jól szemlélteti, hogy a PaymentMethod absztrakt osztály előírja a pay() metódus meglétét, míg az egyes fizetési módok saját módon valósítják meg azt. Ez egy valós alkalmazásokban is gyakran előforduló tervezési minta.

```
Paid $150 with credit card.  
Paid $75 via PayPal.
```

inreface.py

Python ELTE IK

```
from abc import ABC, abstractmethod  
  
class PaymentMethod(ABC):  
  
    @abstractmethod  
    def pay(self, amount):  
        pass  
  
class CreditCard(PaymentMethod):  
  
    def pay(self, amount):  
        return f"Paid ${amount} with credit card."  
  
class PayPal(PaymentMethod):  
  
    def pay(self, amount):  
        return f"Paid ${amount} via PayPal."  
  
credit = CreditCard()  
paypal = PayPal()  
  
print(credit.pay(150))  
print(paypal.pay(75))
```

Összegzés

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.

Többszörös öröklődés és Method Resolution Order (MRO)

- Többszörös öröklődés: Egy osztály egyszerre több szülőosztályból is örökölhet attribútumokat és metódusokat.
- Öröklődés: Az objektumorientált programozás egyik alapelve, amely lehetővé teszi, hogy egy gyermekosztály átvegye a szülőosztály tulajdonságait és metódusait, majd azokat szükség szerint kibővítse vagy módosítsa.
- MRO (Method Resolution Order): Meghatározza, hogy többszörös öröklődés esetén a Python milyen sorrendben keresi meg és hívja meg a metódusokat az öröklési hierarchiában.

Kód példa többszörös öröklődésre és MRO-ra:

[mro.py](#)

MRO alapján melyik metódus hívódik meg?

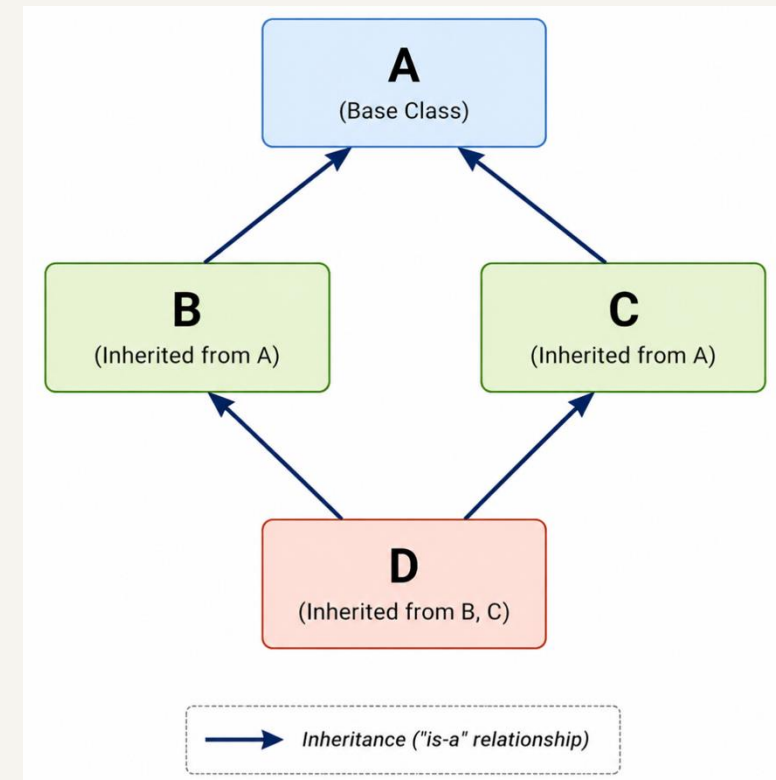
Többszörös öröklődés

- **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 módszer több szülőosztályban is létezik. Ilyen esetben kérdésessé válik, hogy a gyermekosztály melyik szülőosztály módszerát örökölje. Amikor egy módszert hívunk egy gyermekosztályból, a Python az MRO alapján dönti el, hogy melyik szülőosztály módszerát kell használnia.

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



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'>)
```

Konfliktusok a többszörös öröklődésnél

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.

Konfliktusok a többszörös öröklődésnél

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.

D-ből B-ből C-ből A-ból

```
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())
```

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.

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](#)

Python ELTE IK

```
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 fn():  
    print("Most")fn
```

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

Dekorátor előtt. Most! Dekorátor után.

Metódusok dekorálása

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.

Hozzáférés engedélyezve:
Anna

Hozzáférés megtagadva.

Python ELTE IK

```
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()
```


Speciális metódusok (oop 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.

`__str__` és `__repr__`: szöveges reprezentáció

```
class Book:

    def __init__(self, title, author):
        self.title = title
        self.author = author

    def __str__(self):
        return f'"{self.title}" by {self.author}'

    def __repr__(self):
        return f'Book(title='{self.title}', author='{self.author}')"

book = Book("Clean Code", "Robert C. Martin")

print(book)
print(repr(book))
```

```
"Clean Code" by Robert C. Martin
Book(title='Clean Code', author='Robert C. Martin')
```

`__str__()` – felhasználóbarát, olvasható szöveges megjelenítés.

`__repr__()` – fejlesztőknek szánt, részletes objektumreprezentáció, amely megmutatja az objektum állapotát.

`__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(["Nev1", "Nev2", "Nev3"])

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.

__eq__, __lt__, __le__: Összehasonlító műveletek

```
class Product:

    def __init__(self, name, price):
        self.name = name
        self.price = price

    def __eq__(self, other):
        return self.price == other.price

    def __lt__(self, other):
        return self.price < other.price

product1 = Product("Laptop", 1200)
product2 = Product("Monitor", 350)
product3 = Product("Tablet", 1200)

print(product1 == product2)    # False
print(product1 == product3)    # True
print(product2 < product1)     # True
print(product1 < product2)     # False
```

A Pythonban az összehasonlító operátorok (pl. ==, <, <=) testreszabhatók az __eq__, __lt__, __le__ stb. speciális metódusokkal.

Lehetővé teszik, hogy meghatározzuk, hogyan viselkedjenek az osztályaink összehasonlítás során.

__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])
```

20
40

Magyarázat:

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

Összefoglalás

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.

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.

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.

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.

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 elválik 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.

`factoryp.py`

```
class WordDocument:

    def open(self):
        return "Opening Word document."

class TextDocument:

    def open(self):
        return "Opening text document."

class DocumentFactory:

    @staticmethod
    def create_document(doc_type):

        if doc_type == "pdf":
            return PDFDocument()
        elif doc_type == "word":
            return WordDocument()
        elif doc_type == "text":
            return TextDocument()
        else:
            raise ValueError("Unknown document type.")

doc1 = DocumentFactory.create_document("pdf")
doc2 = DocumentFactory.create_document("word")
doc3 = DocumentFactory.create_document("text")

print(doc1.open())
print(doc2.open())
```

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 lépésenként építsünk fel.

`builder.py`

```
class Ship:

    def __init__(self):
        self.hull = None
        self.engine = None
        self.deck = None

    def __str__(self):
        return f"Hull: {self.hull}, Engine: {self.engine}, Deck: {self.deck}"

class ShipBuilder:

    def __init__(self):
        self.ship = Ship()

    def build_hull(self):
        self.ship.hull = "Steel hull"
        return self

    def build_engine(self):
        self.ship.engine = "Diesel engine"
        return self

    def build_deck(self):
        self.ship.deck = "Wooden deck"
        return self

    def get_ship(self):
        return self.ship
```

```
builder = ShipBuilder()
```

```
ship = (builder
        .build_hull()
        .build_engine()
        .build_deck()
        .get_ship())
```

```
print(ship)
```

Hull: Steel hull, Engine: Diesel engine, Deck: Wooden deck

Strukturális tervezési minták

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

A **PrinterAdapter** feladata, hogy a `print_document()` hívást átirányítsa a régi objektum `print_page()` metódusára. Így a régi osztály módosítása nélkül használható olyan programban, amely a modern interfészt várja.

Ez jól szemlélteti az **Adapter tervezési minta** lényegét: két eltérő interfésszel rendelkező osztály együttműködését biztosítja anélkül, hogy az eredeti osztályokat módosítani kellene.

```
class ModernPrinter:

    def print_document(self):
        return "Document printed."

class OldPrinter:

    def print_page(self):
        return "Document printed."

class PrinterAdapter:

    def __init__(self, old_printer):
        self.old_printer = old_printer

    def print_document(self):
        return self.old_printer.print_page()

old_printer = OldPrinter()
adapter = PrinterAdapter(old_printer)

print(adapter.print_document())
```

Document printed.

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.

3200

[decocrpattern.py](#)

[accessdecorator.py](#)

```
class Pizza:

    def price(self):
        return 2000

class CheeseDecorator:

    def __init__(self, pizza):
        self.pizza = pizza

    def price(self):
        return self.pizza.price() + 500

class HamDecorator:

    def __init__(self, pizza):
        self.pizza = pizza

    def price(self):
        return self.pizza.price() + 700

pizza = Pizza()
cheese_pizza = CheeseDecorator(pizza)
ham_cheese_pizza = HamDecorator(cheese_pizza)

print(ham_cheese_pizza.price())
```

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

The subject's state has changed.
The subject's state has changed.

Python ELTE IK

```
class Subject:

    def __init__(self):
        self.observers = []

    def subscribe(self, observer):
        self.observers.append(observer)

    def notify(self):
        for observer in self.observers:
            observer.update()

class Observer:

    def update(self):
        print("The subject's state has changed.")

subject = Subject()

observer1 = Observer()
observer2 = Observer()

subject.subscribe(observer1)
subject.subscribe(observer2)

subject.notify()
```

Összefoglalás

Kreációs minták:

Az objektumok létrehozásának módját szabályozzák.

Példák: Singleton, Factory, Builder.

Strukturális minták:

Az objektumok és osztályok közötti kapcsolatokat kezelik.

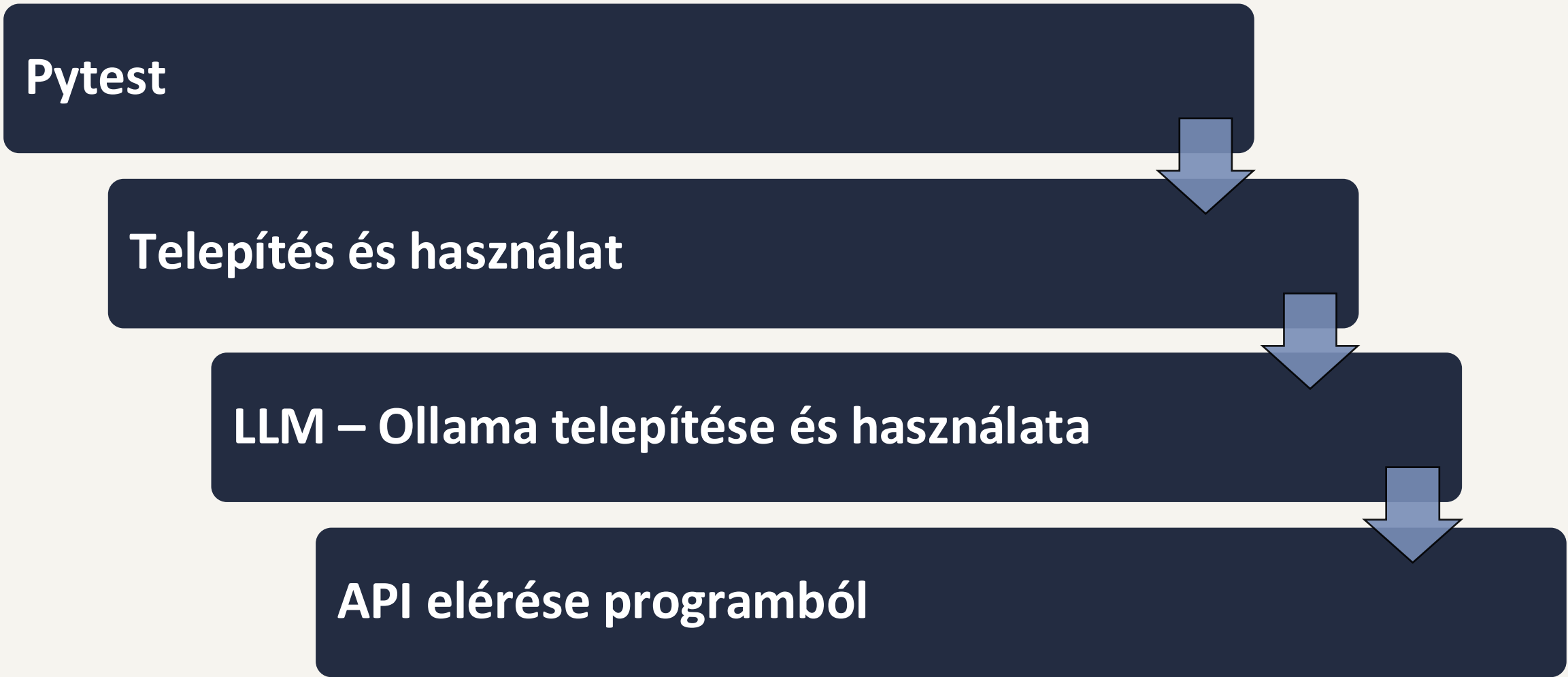
Példák: Adapter, Decorator.

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!

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

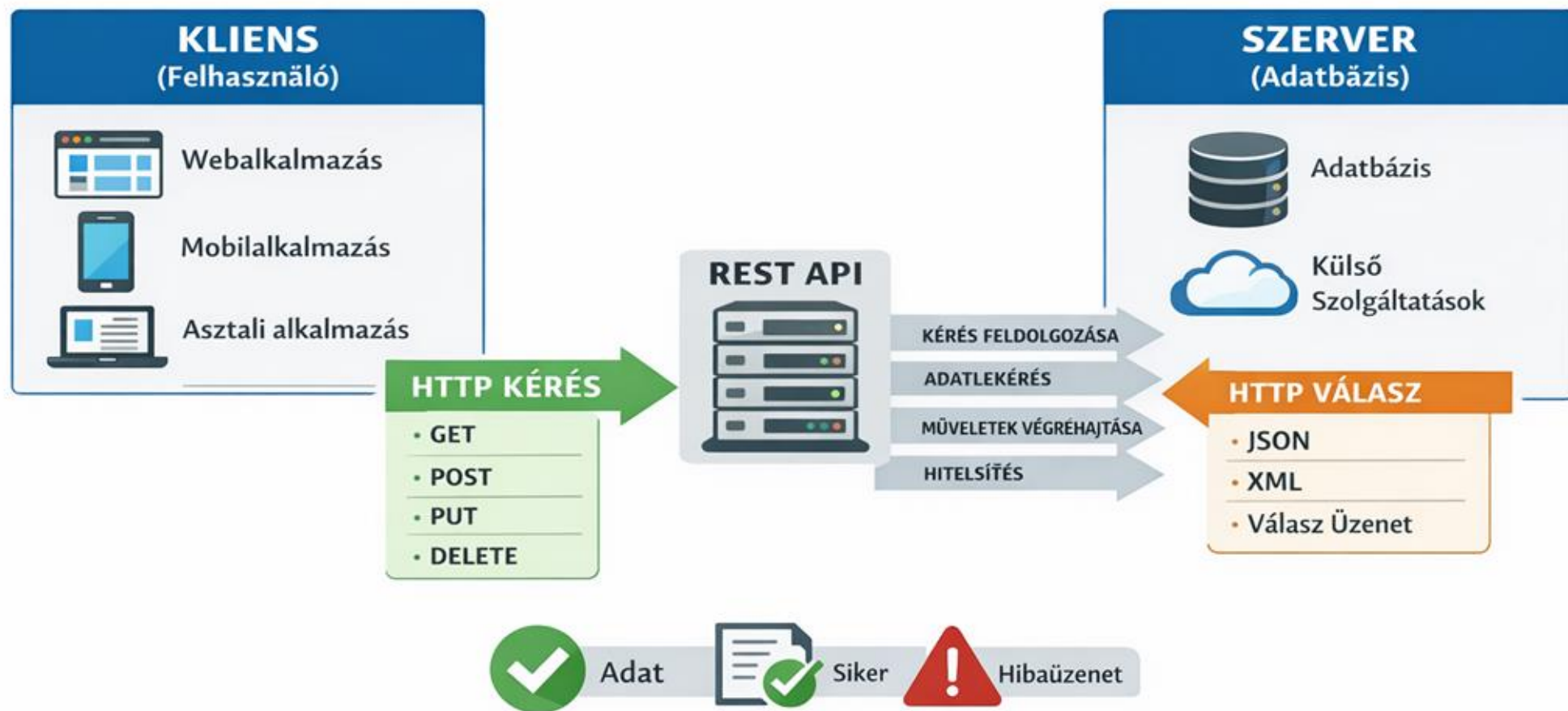
```
from pydantic import EmailStr, BaseModel  
from pydantic import field_validator  
from pydantic import model_validator
```

```
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

Import:

`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)
```

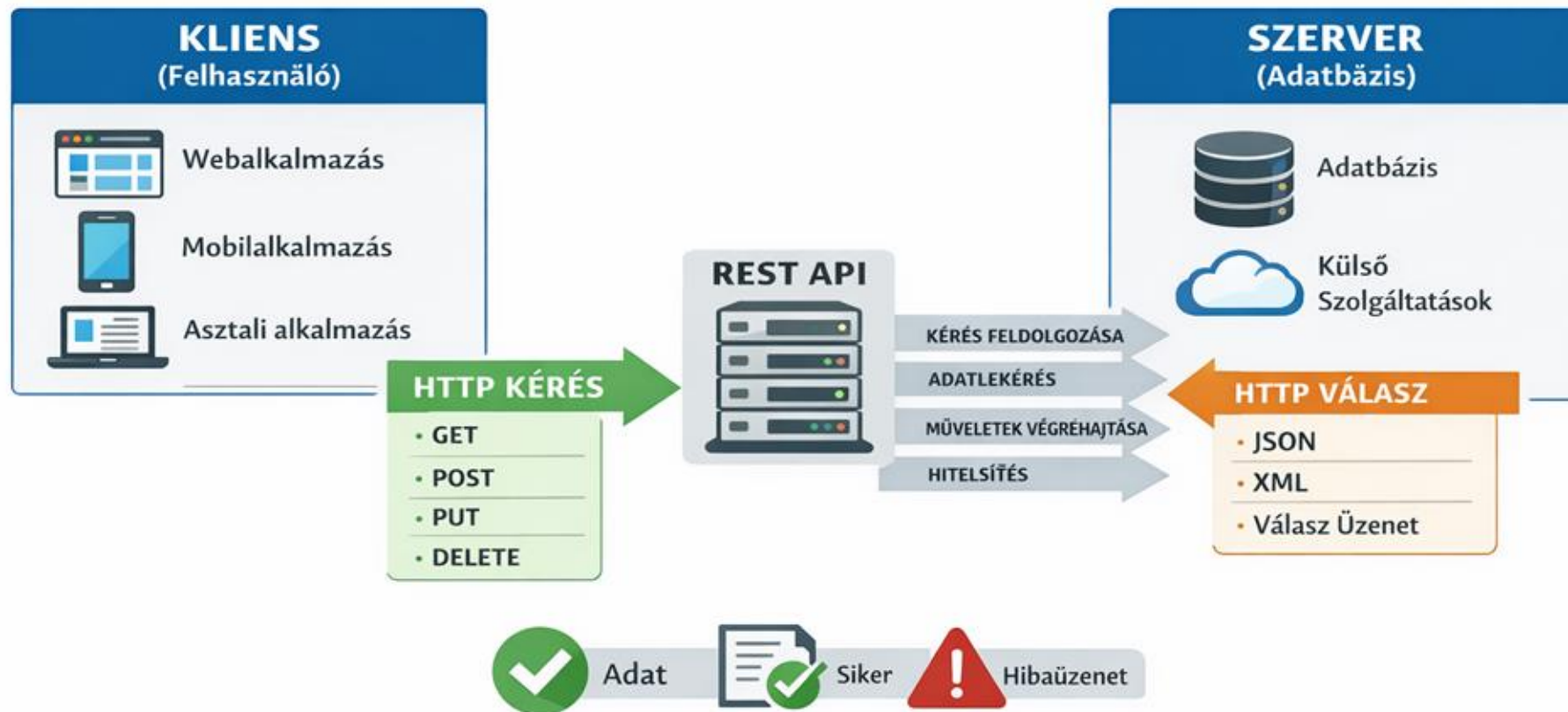
Json adatok kezelése

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

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

add_user(new_user)
```


REST API MŰKÖDÉSE



Fast API

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

Fast API

- **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)

RESTful API alapok és HTTP protokoll

A **REST (Representational State Transfer)** egy webes szolgáltatások fejlesztésére használt architektúrális stílus. A RESTful API olyan API, amely a REST alapelveit követi, és HTTP protokollon keresztül biztosítja az erőforrások elérését és kezelését.

A RESTful API főbb jellemzői:

- Kliens–szerver architektúra:** A kliens küldi a kéréseket, a szerver feldolgozza azokat és válaszol, így a két fél egymástól függetlenül fejleszthető.
- Állapotmentesség (Stateless):** Minden kérés önálló, a szerver nem tárolja a korábbi kérések állapotát. A szükséges információkat minden kérésnek tartalmaznia kell.

HTTP(S) protokoll

A **HTTP (Hypertext Transfer Protocol)** és annak biztonságos változata, a **HTTPS**, a webes kommunikáció alapját képezi. A REST API-k is ezen a protokollon keresztül cserélnek adatokat. A kommunikáció **kérés–válasz (request–response)** modellen alapul, és **állapotmentes (stateless)**, vagyis minden kérés önállóan kerül feldolgozásra.

A HTTP kérés fő részei:

- **URL:** Az elérni kívánt erőforrás címe.
- **HTTP metódus:** A végrehajtandó művelet (pl. GET, POST, PUT, DELETE).
- **Fejlécek (Headers):** A kéréshez tartozó metaadatok (pl. hitelesítés, adattípus).
- **Törzs (Body):** Az elküldött adatok, jellemzően POST és PUT kérések esetén.

A HTTP válasz fő részei:

- **Állapotkód (Status Code):** A kérés eredménye (pl. 200 OK, 201 Created, 404 Not Found).
- **Fejlécek (Headers):** A válasz metaadatai.
- **Törzs (Body):** A visszaküldött adatok, amelyek legtöbbször **JSON** formátumúak.

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'}
```

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.

HTTP státuskódok kezelése

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.

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).

FastAPI alapok: route-ok, request és response kezelés

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/Hi")
def udvozlet():
    return {"uzenet": "Page1, Hi!"}
```

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 Product(BaseModel):          # Model for data validation
    name: str
    price: float

@app.post("/products")
def create_product(product: Product):
    return {
        "message": f"Product '{product.name}' has been added.",
        "price": product.price
    }
```

```
{
  "message": "Product 'Laptop' has been added.",
  "price": 899.99
}
```

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.

```
{
  "name": "Laptop",
  "price": 899.99
}
```

Request (kérés) kezelés

```
from fastapi import FastAPI
from typing import Optional

app = FastAPI()

@app.get("/movies/")
def search_movie(title: Optional[str] = None,
                 year: Optional[int] = None):

    if title and year:
        return {
            "message": f"Movie: {title}, Release year: {year}"
        }

    return {
        "message": "No movie information was provided."
    }
```

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

```
GET /movies/?title=Inception&year=2010
```

```
{
    "message": "Movie: Inception, Release year: 2010"
}
```

```
GET /movies/
```

```
{
    "message": "No movie information was provided."
}
```

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"}'
```

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.

Response (válasz) kezelés

```
from fastapi import FastAPI

from fastapi.responses import JSONResponse

app = FastAPI()

@app.get("/json_valasz")

def json_valasz():

    tartalom = {"uzenet": "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 státuskód 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": "Egyedi válasz!"}

    fejlecek = {"X-Special-Header": "Egyedi header"}

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

A kódban beállítunk egy egyedi HTTP státuskódot (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="/otherpage")
```

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

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:

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.

JSON válaszok készítése FastAPI-val

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/json")

def egyszeru_json_valasz():

    return {"uzenet": "Egyszerű JSON válasz."}
```

Kimenet:

```
{

  "uzenet": "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

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.

Itt egy egyszerű útvonalat hozunk létre, amely visszaad egy **Python dict** objektumot. A FastAPI automatikusan **JSON formátumú válaszra konvertálja** ezt az objektumot.

JSON válasz testreszabása

```
from fastapi import FastAPI

from fastapi.responses import JSONResponse

app = FastAPI()

@app.get("/egyedi_json")

def egyedi_json_valasz():

    tartalom = {"uzenet": "Egyedi JSON válasz."}

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

AFastAPI 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 státuskódot adjunk vissza (201 Created).

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

Pydantic modellek használata a JSON válaszok validálására

```
from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()      # Pydantic model for the response structure

class UserResponse(BaseModel):

    name: str
    age: int

@app.get("/user", response_model=UserResponse)
def get_user():

    return {
        "name": "John",
        "age": 30
    }
```

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

Példa válasz modell használatára

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.

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 userresponse(BaseModel):

    nev: str

    életkor: int

    cimek: List[Cim]

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

def felhasznalo_osszetett():

    return {

        "nev": "Nev1",

        "életkor": 40,

        "cimek": [

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

            {"utca": "Második utca", "varos": "Eger"}

        ]

    }
```

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": "User:."}

    else:

        raise HTTPException(status_code=404, detail="No user found!")
```

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.

Fejlécek és sütik hozzáadása a válaszhoz

```
from fastapi import FastAPI
from fastapi.responses import JSONResponse

app = FastAPI()

@app.get("/status")
def get_status():

    content = {
        "message": "Service is running."
    }

    return JSONResponse(
        content=content,
        headers={"X-API-Version": "1.0"}
    )
```

- A JSON válaszokon túl a FastAPI lehetőséget biztosít fejlécek (**headers**) és sütik (cookies) hozzáadására a válaszokhoz.
- A JSONResponse segítségével testreszabható a HTTP válasz.
- A content paraméter tartalmazza a kliensnek visszaküldött JSON adatokat.
- A headers paraméterrel egyedi HTTP fejlécek adhatók a válaszhoz.
- Ebben a példában az X-API-Version fejléc jelzi, hogy a választ az API 1.0-s verziója szolgáltatta.

X-API-Version: 1.0

```
{
    "message": "Service is running."
}
```

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": "Aszinkron JSON válasz, 2 másodperc
késleltetéssel."}
```

A **FastAPI** natív módon támogatja az **aszinkron programozást**, amely hatékonyabb erőforrás-kezelést és jobb teljesítményt biztosít. Ez különösen előnyös olyan műveleteknél, amelyek külső erőforrásokra – például adatbázisokra vagy webszolgáltatásokra – várnak.

Az `async` és az `await` kulcsszavak lehetővé teszik, hogy a program várakozás közben más kéréseket is kiszolgáljon. Ez tipikusan adatbázis-lekérdezések vagy külső API-hívások esetén hasznos, amikor a szervernek meg kell várnia a válasz megérkezését, de közben nem blokkolja a többi kérés feldolgozását.

Egyszerű API létrehozása és telepítése

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.

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.

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):

Forráskód

Magyarázat:

A Product osztály definiálja a termékek adatszerkezetét.

A / végpont egy egyszerű üdvözlő üzenetet küld vissza.

A /products/ végpont egy terméklistát ad vissza.

A response_model=List[Product] biztosítja, hogy a válasz a Product modelnek megfelelő objektumok listája legyen.

```
[
  {
    "name": "Laptop",
    "price": 999.99
  },
  {
    "name": "Mouse",
    "price": 24.99
  }
]
```

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

app = FastAPI()      # Initialize the application

class Product(BaseModel):      # Product model
    name: str
    price: float

@app.get("/")
def home():
    return {"message": "Welcome to the Product API!"}

@app.get("/products/", response_model=List[Product])
def get_products():

    products = [
        {"name": "Laptop", "price": 999.99},
        {"name": "Mouse", "price": 24.99}
    ]

    return products
```

API futtatása lokálisan

A FastAPI alkalmazások futtatásához általában az Uvicorn ASGI (Asynchronous Server Gateway Interface) szerveret használjuk. Az Uvicorn egy gyors és hatékony webszerver, amely támogatja az aszinkron Python alkalmazások futtatását.

```
uvicorn main:app --reload
```

main – a Python fájl neve (main.py).

app – a FastAPI alkalmazás objektuma (app = FastAPI()).

--reload – fejlesztői mód, amely a fájlok módosításakor automatikusan újraindítja a szerveret.

Az alkalmazás indítása után az API a <http://127.0.0.1:8000> címen érhető el.

A FastAPI automatikusan elkészíti az API interaktív dokumentációját, amely a **Swagger UI** felületen tekinthető meg:

Swagger UI: <http://127.0.0.1:8000/docs>

Emellett az **OpenAPI** dokumentáció JSON formátumban is elérhető a következő címen:

OpenAPI JSON: <http://127.0.0.1:8000/openapi.json>

API telepítése webszerverre

```
git init  
git add .  
git commit -m "Main first 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.

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:

Git repository létrehozása

Először hozzuk létre a projektet git repository-ként:

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}
```

API skálázása és optimalizálása

Miután az API elkészült és működik, fontos a teljesítmény és a skálázhatóság biztosítása. Ennek érdekében több technika is alkalmazható:

Adatbázis-integráció: Hatékony adatkezelés ORM-ek (pl. SQLAlchemy) vagy NoSQL adatbázisok (pl. MongoDB) használatával.

Aszinkron feldolgozás: Az `async/await` alkalmazásával több kérés kezelhető párhuzamosan, ami javítja a válaszidőt.

Gyorsítótárazás (Cache): Gyakran lekért adatok tárolása (pl. Redis segítségével), így csökkenthető az adatbázis terhelése és gyorsabbá válik az API.

Terheléelosztás (Load Balancing): A beérkező kérések több szerver között oszthatók szét, ami nagyobb felhasználói terhelés mellett is stabil működést biztosít.

FastAPI és adatbázis integráció (SQLAlchemy bevezetése)

SQLAlchemy áttekintése

Az SQLAlchemy a Python egyik legelterjedtebb ORM (Object-Relational Mapping) könyvtára, amely megkönnyíti az alkalmazások és a relációs adatbázisok közötti kommunikációt. Segítségével az adatbázistáblák Python osztályokként, a rekordok pedig objektumokként kezelhetők, így jelentősen csökkenthető a kézzel írt SQL lekérdezések mennyisége.

Az SQLAlchemy használatának lépései:

Adatbázis-kapcsolat létrehozása az alkalmazás és az adatbázis között.

Adatmodellek definiálása, amelyek az adatbázis tábláit reprezentálják.

CRUD műveletek megvalósítása (Create, Read, Update, Delete) az adatok kezelésére.

Tranzakciók kezelése, amelyek biztosítják az adatműveletek megbízható és konzisztens végrehajtását.

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

Forráskód

Adatbázis kapcsolat konfigurációja

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

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

- 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 import create_engine
from sqlalchemy.orm import declarative_base, sessionmaker

# SQLite adatbázis létrehozása
DATABASE_URL = "sqlite:///library.db"

# Adatbázis kapcsolat
engine = create_engine(DATABASE_URL, echo=True)

# Session létrehozása
SessionLocal = sessionmaker(bind=engine)

# Alaposztály az ORM modellekhez
Base = declarative_base()
```


Modellek létrehozása

```
from sqlalchemy import Column, Integer, String, Float

# Model representing a database table

class Product(Base):

    __tablename__ = "products"

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

    name = Column(String(100), index=True)

    category = Column(String(50))

    price = Column(Float)
```

A modellek az adatbázis táblák reprezentációi Python osztályokként.

Ezek segítségével definiáljuk az adatbázis táblák mezőit és azok típusait.

Felhasználó modell létrehozása

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

`__tablename__` – Specifies the name of the database table.

`id` – Primary key that uniquely identifies each product.

`name` – Name of the product.

`category` – Category to which the product belongs.

`price` – Price of the product stored as a floating-point value.

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 FastAPI, Depends
from sqlalchemy.orm import Session

from database import SessionLocal
from models import Product

app = FastAPI()

# Database session
def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()

# Get all products
@app.get("/products")
def get_products(db: Session = Depends(get_db)):
    products = db.query(Product).all()
    return products
```

CRUD műveletek implementálása

```
from sqlalchemy.orm import Session

from . import models, schemas

# Create a new product
def create_product(db: Session, product: schemas.ProductCreate):

    db_product = models.Product(
        name=product.name,
        category=product.category,
        price=product.price
    )

    db.add(db_product)

    db.commit()

    db.refresh(db_product)

    return db_product
```

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

Adatok hozzáadása (Create)

A példa bemutatja, hogyan adhatunk hozzá új felhasználót az adatbázishoz

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.

Adatok lekérése (Read)

```
from sqlalchemy.orm import Session

from . import models

# Get all products
def get_products(db: Session):

    return db.query(models.Product).all()

# Get a product by ID
def get_product(db: Session, product_id: int):

    return (
        db.query(models.Product)
        .filter(models.Product.id == product_id)
        .first()
    )
```

Lekérhetjük az **összes** felhasználót az adatbázisból, vagy egy adott felhasználót az **id** alapján.

Adatok módosítása (Update)

A kód bemutatja, hogyan
módosíthatunk egy létező product-ot

```
from sqlalchemy.orm import Session

from . import models, schemas

# Update a product
def update_product(
    db: Session,
    product_id: int,
    updated_product: schemas.ProductUpdate
):
    db_product = (
        db.query(models.Product)
        .filter(models.Product.id == product_id)
        .first()
    )

    if db_product:
        db_product.name = updated_product.name
        db_product.category = updated_product.category
        db_product.price = updated_product.price

        db.commit()
        db.refresh(db_product)

    return db_product
```

Adatok törlése (Delete)

```
from sqlalchemy.orm import Session

from . import models

# Delete a product
def delete_product(db: Session, product_id: int):

    db_product = (
        db.query(models.Product)
        .filter(models.Product.id == product_id)
        .first()
    )

    if db_product:
        db.delete(db_product)
        db.commit()

        return {"message": "Product deleted successfully."}

    return {"message": "Product not found."}
```

Egy product törlése az alábbi módon történhet

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.

Összefoglalás

- Az SQLAlchemy és a FastAPI együttes használata egyszerűvé és hatékonyá teszi az adatbázisok kezelését Python webalkalmazásokban.
- Az ORM megközelítésnek köszönhetően az adatbázis rekordjai Python objektumokként kezelhetők, így kevesebb SQL kód írására van szükség.
- A megfelelő session-kezelés, a CRUD műveletek megvalósítása és az adatbázis-migrációk alkalmazása együttesen biztosítják, hogy az API megbízhatóan, biztonságosan és hatékonyan kezelje az adatokat.

Köszönöm a figyelmet!

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)

Forrás: CISCO.NETACAD.NET

A projekt elérhetősége:

<https://contenthub.netacad.com/legacy/IoTFBDA/2.01/es/course/files/2.2.4.5%20Lab%20-%20San%20Francisco%20Crime.html>

Big Data alapfogalmak

Az adatok napjainkban az üzleti és a tudományos élet egyik legfontosabb erőforrásai közé tartoznak. Az adatelemzés segítségével a szervezetek megalapozottabb döntéseket hozhatnak, növelhetik működésük hatékonyságát, és gyorsabban reagálhatnak a változó környezetre. Az adatvezérelt döntéshozatal lehetővé teszi az ügyfelek, a versenytársak és a piaci trendek pontosabb elemzését, ezáltal versenyelőnyt biztosít.

Az adatok típusai

Strukturált adatok: Meghatározott szerkezetű adatok, amelyek táblázatokban vagy relációs adatbázisokban tárolhatók (pl. SQL adatbázisok, Excel-táblák).

Strukturálatlan adatok: Nem előre definiált szerkezetű adatok, például szöveges dokumentumok, e-mailek, képek, hangfelvételek vagy videók.

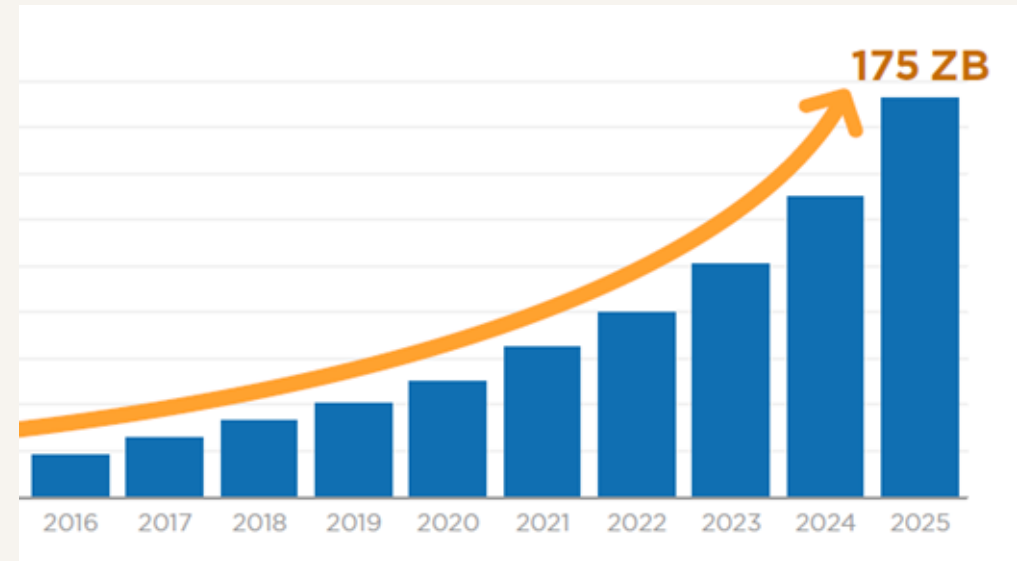
Big Data alapfogalmak

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..

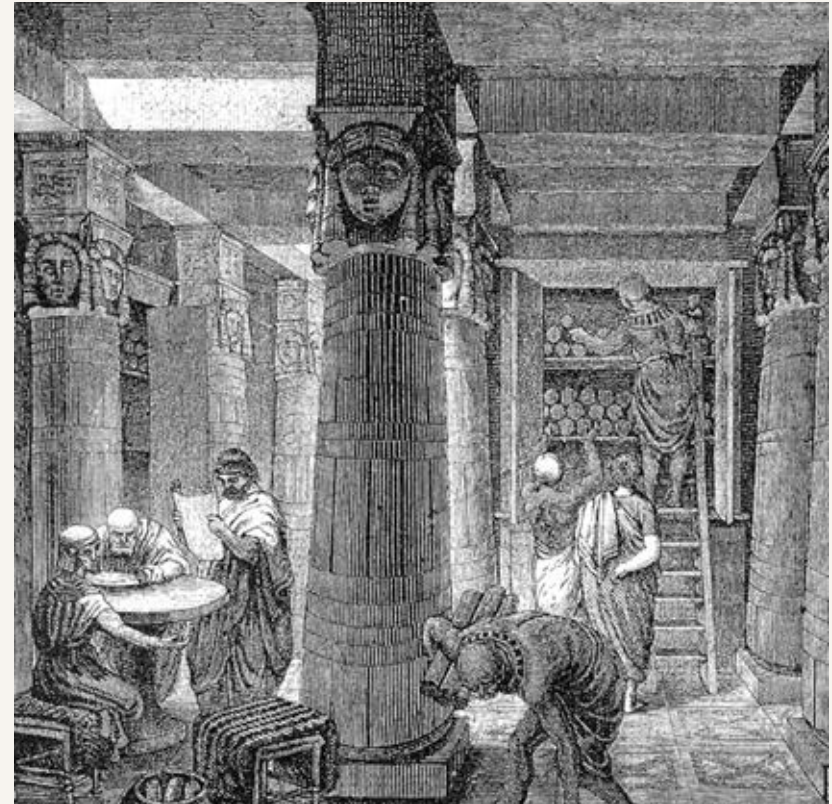
<https://www.idc.com/>



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



Kép forrása: wikipedia.org

Big Data alapfogalmak

A Big Data olyan nagy méretű és összetett adathalmazokat jelent, amelyek feldolgozása már nem oldható meg hatékonyan hagyományos eszközökkel. A Big Data technológiák lehetővé teszik a különböző forrásokból származó, folyamatosan keletkező adatok tárolását, feldolgozását és elemzését, akár valós időben is.

A Big Data fő jellemzői – az 5V modell

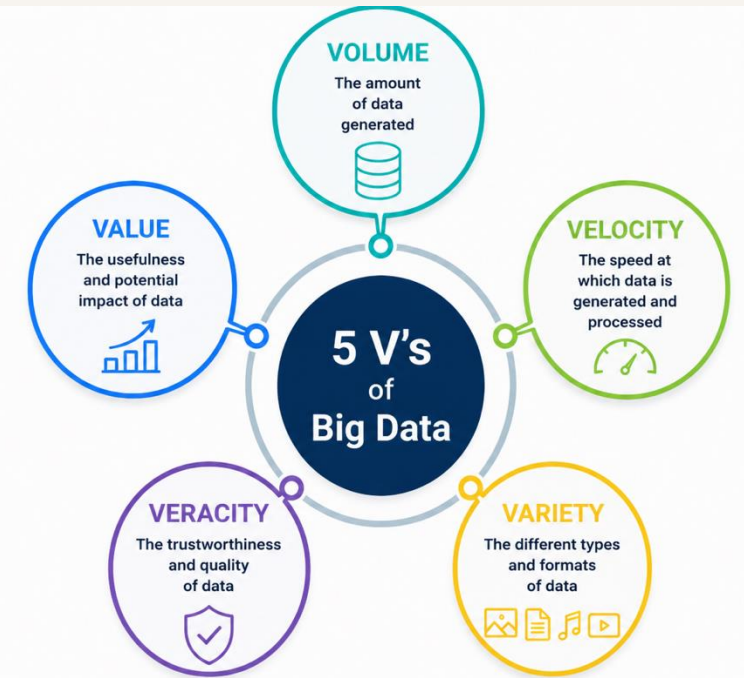
Volume (Mennyiség): Nagy mennyiségű adat kezelése.

Velocity (Sebesség): Az adatok gyors keletkezése, továbbítása és feldolgozása.

Variety (Változatosság): Különböző típusú és formátumú adatok (pl. szöve, kép, videó, szenzoradatok).

Veracity (Hitelesség): Az adatok megbízhatósága, minősége és pontossága

Value (Érték): Az adatokból kinyerhető hasznos információ és üzleti vagy tudományos érték.



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

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:

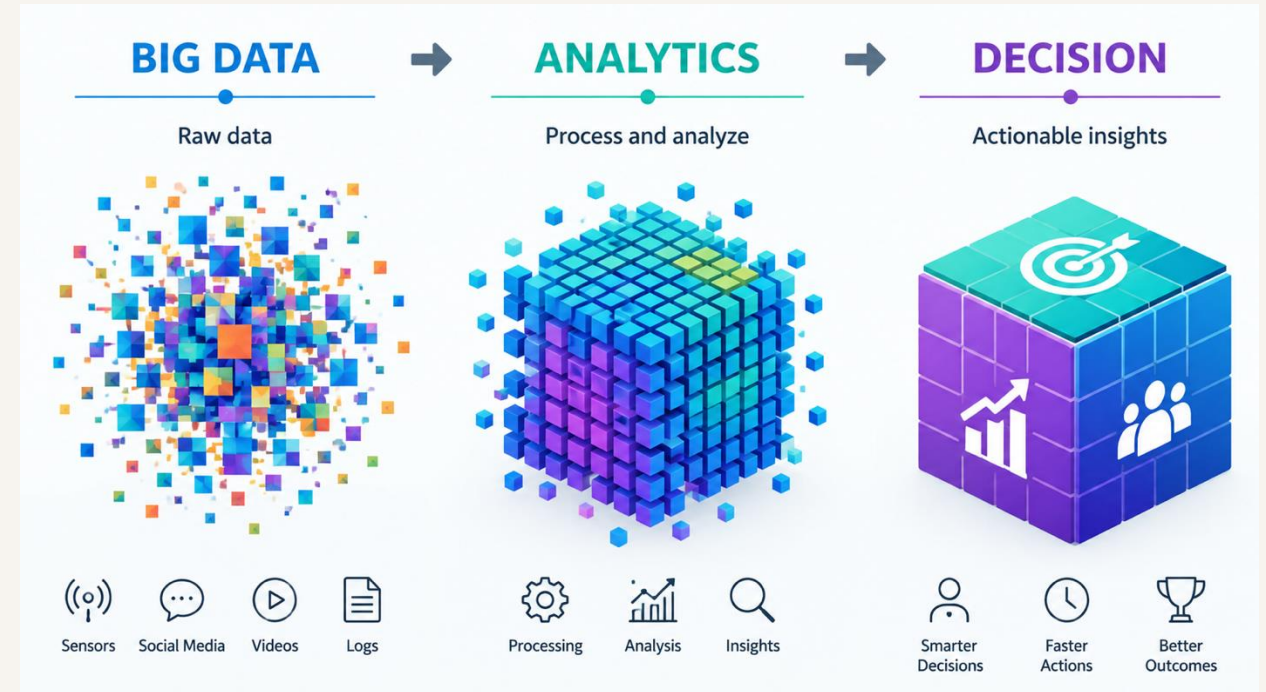
- Adatok gyűjtése.
- Adatok előkészítése és tisztítása.
- Adatok vizualizációja.
- Adatok elemzése.
- Értelmezés és következtetések levonása.
- Döntéshozatal és intézkedések végrehajtása.

Big Data alapfogalmak

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.jgy.pk.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 adatelemzés 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

```
import numpy as np

# Havi eladások
sales = np.array([120, 150, 180, 200, 170])

print("Monthly sales:", sales)

# Összes eladás
total_sales = np.sum(sales)

# Átlagos havi eladás
average_sales = np.mean(sales)

# Legnagyobb havi eladás
highest_sales = np.max(sales)

print("Total sales:", total_sales)
print("Average sales:", average_sales)
print("Highest monthly sales:", highest_sales)
```

- 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.
- `np.array()` – NumPy tömb létrehozása.
- `temperatures + 2` – elemről elemre végrehajtott művelet (vektorizáció).
- `np.max()` – a legnagyobb elem meghatározása.
- `np.mean()` – az átlag kiszámítása.

```
Monthly sales: [120 150 180 200 170]
Total sales: 820
Average sales: 164.0
Highest monthly sales: 200
```

Adatelemzés Python eszközökkel

Pandas – Adatfeldolgozás és adatelemzés

A Pandas a Python egyik legnépszerűbb könyvtára, amely táblázatos adatok kezelésére és elemzésére szolgál. Segítségével egyszerűen végezhetünk adatbetöltést, szűrést, módosítást, összesítést és egyéb adatfeldolgozási műveleteket.

A Pandas hatékony eszközöket biztosít a strukturált adatok kezeléséhez, ezért széles körben alkalmazzák az adatelemzés, az üzleti intelligencia és a gépi tanulás területén.

A Pandas főbb adatszerkezetei

Series: Egydimenziós, indexelt adatszerkezet, amely egy értéksorozatot és a hozzá tartozó indexeket tárolja. Funkcióiban hasonlít egy Python listára, de annál jóval több adatkezelési lehetőséget biztosít.

DataFrame: Kétdimenziós, sorokból és oszlopokból álló táblázatos adatszerkezet. A Pandas legfontosabb objektuma, amely lehetővé teszi az adatok tárolását, szűrését, módosítását, elemzését és megjelenítését.

Adatelemzés Python eszközökkel

Pandas – Series példák

Egydimenziós adatszerkezet
létrehozása és kezelése:

```
import pandas as pd

# Termékek árai
prices = pd.Series(
    [350, 420, 280, 510],
    index=["Apple", "Banana", "Orange", "Grapes"]
)

print("Product prices:")
print(prices)
```

```
Product prices:
Apple      350
Banana     420
Orange     280
Grapes     510
dtype: int64
```

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

A Pandas számos beépített funkciót biztosít a táblázatos adatok kezelésére és elemzésére. Segítségével egyszerűen beolvashatunk adatokat különböző fájlformátumokból, megvizsgálhatjuk azok szerkezetét, majd szűrhetjük, módosíthatjuk vagy bővíthetjük az adatokat.

Gyakori DataFrame műveletek

•Adatok beolvasása és mentése

- CSV fájl betöltése DataFrame-be
- DataFrame exportálása CSV fájlba

•Az adatok gyors áttekintése

- Az első néhány sor megjelenítése (head())
- Statisztikai összefoglaló készítése (describe())
- Az oszlopok adattípusainak megjelenítése (dtypes)

•Adatok kiválasztása és módosítása

- Meghatározott oszlopok kiválasztása
- Sorok szűrése feltétel alapján
- Új oszlop létrehozása vagy meglévő módosítása

```
import pandas as pd

# DataFrame létrehozása
students = {
    "Name": ["Alice", "Bob", "Charlie", "Diana"],
    "Grade": [85, 92, 78, 95],
    "Course": ["Math", "Physics", "Math", "Computer Science"]
}

df = pd.DataFrame(students)

# Első sorok megjelenítése
print(df.head())

# Alapvető statisztika
print(df.describe())

# Adattípusok
print(df.dtypes)

# Két oszlop kiválasztása
print(df[["Name", "Grade"]])

# 90 pont feletti hallgatók
print(df[df["Grade"] > 90])

# Új oszlop létrehozása
df["Passed"] = df["Grade"] >= 80

print(df)

# DataFrame mentése CSV fájlba
df.to_csv("students.csv", index=False)
```

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

A Series egy egydimenziós adatszerkezet, amely hasonló a Python listához vagy szótárhoz, de tartalmaz egy kapcsolódó indexet, amely minden értéket azonosít.

```
Book sales:
Python      120
SQL          85
Machine Learning  210
Java         95
Data Science 160
dtype: int64
```

```
import pandas as pd

# Eladott könyvek példányszáma
sales = pd.Series(
    [120, 85, 210, 95, 160],
    index=["Python", "SQL", "Machine Learning", "Java", "Data Science"]
)

print("Book sales:\n", sales)

# Egy adott könyv eladásainak lekérdezése
print("\nPython copies sold:", sales["Python"])

# Csak a 100 példánynál többet eladott könyvek
bestsellers = sales[sales > 100]

print("\nBest-selling books:\n", bestsellers)
```

Pandas adatstruktúrák

A **DataFrame** egy kétdimenziós, táblázatos adat-struktúra, sorok és oszlopok. Hasonló egy táblázathoz vagy SQL adatbázishoz, ahol az **oszlopok különböző típusú adatokat** tartalmazhatnak.

```
import pandas as pd

# DataFrame létrehozása
products = {
    "Product": ["Laptop", "Mouse", "Keyboard", "Monitor", "Headphones"],
    "Price": [950, 25, 60, 320, 80],
    "Stock": [15, 120, 75, 30, 45]
}

products_df = pd.DataFrame(products)

print("Product inventory:\n")
print(products_df)

# Egy oszlop kiválasztása
print("\nPrices:\n")
print(products_df["Price"])

# Egy sor kiválasztása index alapján
print("\nFirst product:\n")
print(products_df.iloc[0])

# Szűrés: csak a 100 dollárnál drágább termékek
expensive_products = products_df[products_df["Price"] > 100]

print("\nProducts costing more than $100:\n")
print(expensive_products)
```

Pandas adatstruktúrák

DataFrame példa

Product inventory:

	Product	Price	Stock
0	Laptop	950	15
1	Mouse	25	120
2	Keyboard	60	75
3	Monitor	320	30
4	Headphones	80	45

Prices:

0	950
1	25
2	60
3	320
4	80

Name: Price, dtype: int64

First product:

Product	Laptop
Price	950
Stock	15

Name: 0, dtype: object

Products costing more than \$100:

	Product	Price	Stock
0	Laptop	950	15
3	Monitor	320	30

Pandas adatstruktúrák

```
# Termékek rendezése ár szerint csökkenő sorrendben

sorted_df = products_df.sort_values(
    by="Price",
    ascending=False
)

print("\nProducts sorted by price:")
print(sorted_df)
```

	Product	Price	Stock	Category
0	Laptop	950	15	Computer
3	Monitor	320	30	Display
4	Headphones	80	45	Audio
2	Keyboard	60	75	Accessory
1	Mouse	25	120	Accessory

```
# Új oszlop hozzáadása: kategória
```

```
products_df["Category"] = [
    "Computer",
    "Accessory",
    "Accessory",
    "Display",
    "Audio"
]

print("\nDataFrame with a new column:")
print(products_df)
```

	Product	Price	Stock	Category
0	Laptop	950	15	Computer
1	Mouse	25	120	Accessory
2	Keyboard	60	75	Accessory
3	Monitor	320	30	Display
4	Headphones	80	45	Audio

Pandas adatstruktúrák

Alapvető statisztikai műveletek

Átlag, maximum, szórás és módusz

```
Average price: 287.0  
  
Highest price: 950  
  
Price standard deviation: 390.36  
  
Most common price:  
0    80  
dtype: int64
```

```
# Átlagár kiszámítása  
avg_price = products_df["Price"].mean()  
print("\nAverage price:", avg_price)  
  
# Legdrágább termék árának meghatározása  
max_price = products_df["Price"].max()  
print("\nHighest price:", max_price)  
  
# Árak szórásának kiszámítása  
price_std = products_df["Price"].std()  
print("\nPrice standard deviation:", price_std)  
  
# Leggyakoribb ár meghatározása  
most_common_price = products_df["Price"].mode()  
print("\nMost common price:")  
print(most_common_price)
```

Product	Price	Stock	Category
Laptop	950	15	Computer
Mouse	25	120	Accessory
Keyboard	80	75	Accessory
Monitor	320	30	Display
Headphones	80	45	Audio

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('adatok.xlsx', sheet_name='Data1')  
print(Excel adatok:\n", data_df.head())
```

Adatok betöltése SQL
adatbázisból:

```
# SQL adatbázisból történő beolvasás  
import sqlite3  
connection = sqlite3.connect('adatb.db')  
data_df = pd.read_sql_query('SELECT * FROM users', connection)  
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.

```
import pandas as pd
import numpy as np

# Termékek DataFrame-je hiányzó adatokkal
products_df = pd.DataFrame({
    "Product": ["Laptop", "Mouse", "Keyboard", "Monitor", "Printer"],
    "Price": [950, 25, np.nan, 320, 180],
    "Stock": [15, np.nan, 40, 20, np.nan]
})

print("Original DataFrame:")
print(products_df)

# Hiányzó adatot tartalmazó sorok eltávolítása
clean_df = products_df.dropna()

print("\nRows without missing values:")
print(clean_df)

# Hiányzó készlet kitöltése 0-val
products_df["Stock"] = products_df["Stock"].fillna(0)

# Hiányzó ár kitöltése az átlaggal
products_df["Price"] = products_df["Price"].fillna(products_df["Price"].mean())

print("\nCompleted DataFrame:")
print(products_df)
```

Original DataFrame:

	Product	Price	Stock
0	Laptop	950.0	15.0
1	Mouse	25.0	NaN
2	Keyboard	NaN	40.0
3	Monitor	320.0	20.0
4	Printer	180.0	NaN

Rows without missing values:

	Product	Price	Stock
0	Laptop	950.0	15.0
3	Monitor	320.0	20.0

Completed DataFrame:

	Product	Price	Stock
0	Laptop	950.0	15.0
1	Mouse	25.0	0.0
2	Keyboard	368.75	40.0
3	Monitor	320.0	20.0
4	Printer	180.0	0.0

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:

```
# A 'Quantity' oszlop átalakítása egész számmá
orders_df["Quantity"] = orders_df["Quantity"].astype(int)

print("Data types:\n", orders_df.dtypes)

# Az 'OrderDate' oszlop átalakítása dátummá
orders_df["OrderDate"] = pd.to_datetime(orders_df["OrderDate"])

print("\nUpdated data types:\n", orders_df.dtypes)
```

```
# A 'Price' oszlop átalakítása lebegőpontos számmá
products_df["Price"] = products_df["Price"].astype(float)

print("Data types after conversion:\n", products_df.dtypes)

# A 'Category' oszlop átalakítása kategória típusúvá
products_df["Category"] = products_df["Category"].astype("category")

print("\nUpdated data types:\n", products_df.dtypes)
```

Adatok beolvasása és előfeldolgozása

Duplikált sorok eltávolítása, adatok rendezése, szűrése:

`drop_duplicates()` – ismétlődő sorok eltávolítása,
`sort_values()` – adatok rendezése,
feltételes szűrés (`df[df["Oszlop"] > érték]`) – adott feltételnek megfelelő sorok kiválasztása.

```
# Duplikált termékek eltávolítása
products_df = products_df.drop_duplicates()

# Termékek rendezése ár szerint (növekvő sorrend)
products_df = products_df.sort_values(by="Price")

# Csak a 100 egységnél drágább termékek kiválasztása
expensive_products = products_df[products_df["Price"] > 100]

print("Expensive products:\n", expensive_products)
```

```
# Duplikált könyvek eltávolítása
books_df = books_df.drop_duplicates()

# Könyvek rendezése megjelenési év szerint
books_df = books_df.sort_values(by="Year")

# A 2020 után megjelent könyvek kiválasztása
new_books = books_df[books_df["Year"] > 2020]

print("Recent books:\n", new_books)
```

Adatmanipulációk

Az adatmanipuláció a Pandas egyik legfontosabb képessége, amely számos eszközt biztosít az adatok kezelésére és átalakítására. Segítségével egyszerűen végezhetünk szűrést, rendezést, módosítást és összesítést, amelyek elengedhetetlenek az adatok előkészítéséhez és az elemzések elvégzéséhez.

Szűrés:

Az adatok szűrése lehetővé teszi, hogy csak a megadott feltételnek megfelelő sorokat tartsuk meg egy új DataFrame-ben.

```
import pandas as pd

# Minta DataFrame létrehozása
data = {
    "Title": ["Python Basics", "AI Guide", "Data Science", "Web Development"],
    "Pages": [180, 320, 450, 210],
    "Year": [2021, 2023, 2020, 2024]
}

books_df = pd.DataFrame(data)

# Szűrés: csak a 250 oldalnál hosszabb könyvek
long_books = books_df[books_df["Pages"] > 250]

print("Books with more than 250 pages:")
print(long_books)
```

Books with more than 250 pages:

	Title	Pages	Year
1	AI Guide	320	2023
2	Data Science	450	2020

Adatmanipulációk

Rendezés

A rendezés lehetővé teszi az adatok sorainak újra sorrendezését egy vagy több oszlop alapján.

```
import pandas as pd

# Minta DataFrame létrehozása
data = {
    "Product": ["Laptop", "Mouse", "Monitor", "Keyboard", "Printer"],
    "Price": [950, 25, 320, 80, 180],
    "Stock": [15, 120, 20, 40, 8]
}

products_df = pd.DataFrame(data)

# Termékek rendezése ár szerint növekvő sorrendben
sorted_by_price = products_df.sort_values(by="Price")

print("Products sorted by price:")
print(sorted_by_price)
```

```
# Termékek rendezése készlet szerint csökkenő sorrendben
sorted_by_stock = products_df.sort_values(by="Stock", ascending=False)

print("\nProducts sorted by stock (descending):")
print(sorted_by_stock)
```

```
Products sorted by price:
   Product  Price  Stock
1   Mouse     25    120
3  Keyboard     80     40
4   Printer    180      8
2   Monitor    320     20
0   Laptop    950     15
```

```
Products sorted by stock (descending):
   Product  Price  Stock
1   Mouse     25    120
3  Keyboard     80     40
2   Monitor    320     20
0   Laptop    950     15
4   Printer    180      8
```


Adatmanipulációk

Aggregálás

Az aggregálás lehetővé teszi az adatok összesítését különböző metrikák segítségével, például átlag, összeg, minimum, maximum.

```
Average price: 311.0
```

```
Average price by category:
```

```
Category
```

```
Accessory      52.5
```

```
Computer      635.0
```

```
Office        180.0
```

```
Name: Price, dtype: float64
```

```
import pandas as pd

# Minta DataFrame létrehozása
data = {
    "Product": ["Laptop", "Mouse", "Keyboard", "Monitor", "Printer"],
    "Category": ["Computer", "Accessory", "Accessory", "Computer", "Office"],
    "Price": [950, 25, 80, 320, 180]
}

products_df = pd.DataFrame(data)

# Az összes termék átlagárának kiszámítása
avg_price = products_df["Price"].mean()
print("Average price:", avg_price)

# Átlagár kiszámítása kategóriánként
avg_price_by_category = products_df.groupby("Category")["Price"].mean()

print("\nAverage price by category:")
print(avg_price_by_category)
```

Adatmanipulációk

Összetett példa

A szűrés, rendezés és aggregálás kombinálható a mélyebb elemzés érdekében.

Products costing more than 100:

	Product	Category	Price
4	Printer	Office	180
2	Monitor	Computer	320
0	Laptop	Computer	950

Number of products by category:

Category	
Accessory	2
Computer	2
Office	1

dtype: int64

```
import pandas as pd

# Minta DataFrame létrehozása
data = {
    "Product": ["Laptop", "Mouse", "Keyboard", "Monitor", "Printer"],
    "Category": ["Computer", "Accessory", "Accessory", "Computer", "Office"],
    "Price": [950, 25, 80, 320, 180]
}

products_df = pd.DataFrame(data)

# 100 egységnél drágább termékek kiválasztása, majd rendezés ár szerint
filtered_products = products_df[products_df["Price"] > 100].sort_values(by="Price")

print("Products costing more than 100:")
print(filtered_products)

# Termékek számának meghatározása kategóriánként
products_per_category = products_df.groupby("Category").size()

print("\nNumber of products by category:")
print(products_per_category)
```

Adatfeldolgozási feladatok és példák

Az adatfeldolgozási feladatok során különböző műveleteket végezhetünk a **Pandas** könyvtár segítségével, amelyek segítségével az adatokat **előkészít-hetjük, 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

Új oszlop létrehozása

```
import pandas as pd

# Minta DataFrame létrehozása
data = {
    "employee": ["John", "Emma", "David", "Sophia"],
    "salary": [3200, 4100, 2900, 5000],
    "department": ["IT", "HR", "Sales", "IT"]
}

employees_df = pd.DataFrame(data)

# Új oszlop hozzáadása: 10%-os bónusszal növelt fizetés
employees_df["salary_with_bonus"] = employees_df["salary"] * 1.10

print("DataFrame with bonus salary:")
print(employees_df)
```

	employee	salary	department	salary_with_bonus
0	John	3200	IT	3520.0
1	Emma	4100	HR	4510.0
2	David	2900	Sales	3190.0
3	Sophia	5000	IT	5500.0

Adafeldolgozási feladatok és példák

Adatok normalizálása

Az adatok normalizálása során az értékeket egy közös skálára helyezzük, ami segíthet az adatok összehasonlíthatóságában.

Products with category:

	product	price	category
0	Laptop	1200	Expensive
1	Mouse	25	Affordable
2	Monitor	300	Affordable
3	Keyboard	80	Affordable

```
import pandas as pd

# Minta DataFrame létrehozása
data = {
    "product": ["Laptop", "Mouse", "Monitor", "Keyboard"],
    "price": [1200, 25, 300, 80]
}

products_df = pd.DataFrame(data)

# Új oszlop létrehozása az ár alapján
products_df["category"] = products_df["price"].apply(
    lambda x: "Expensive" if x >= 500 else "Affordable"
)

print("Products with category:")
print(products_df)
```

Adafeldolgozási feladatok és példák

Feltételes adatmanipuláció

Az adatok manipulációja során gyakran szükséges egyes sorokat vagy oszlopokat feltételek alapján módosítani.

```
# Új oszlop, amely megjelöli, hogy a fizetés 4000 felett van-e

employees_df["high_salary"] = employees_df["salary"].apply(
    lambda x: "Yes" if x > 4000 else "No"
)

print("DataFrame with salary category:\n", employees_df)
```

	employee	salary	department	high_salary
0	John	3200	IT	No
1	Emma	4100	HR	Yes
2	David	2900	Sales	No
3	Sophia	5000	IT	Yes

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

DataFrame-ek egyesítése

Merged DataFrame:

	employee_id	name	department
0	101	John	IT
1	102	Emma	HR
2	103	David	Sales
3	104	Sophia	Finance

```
import pandas as pd

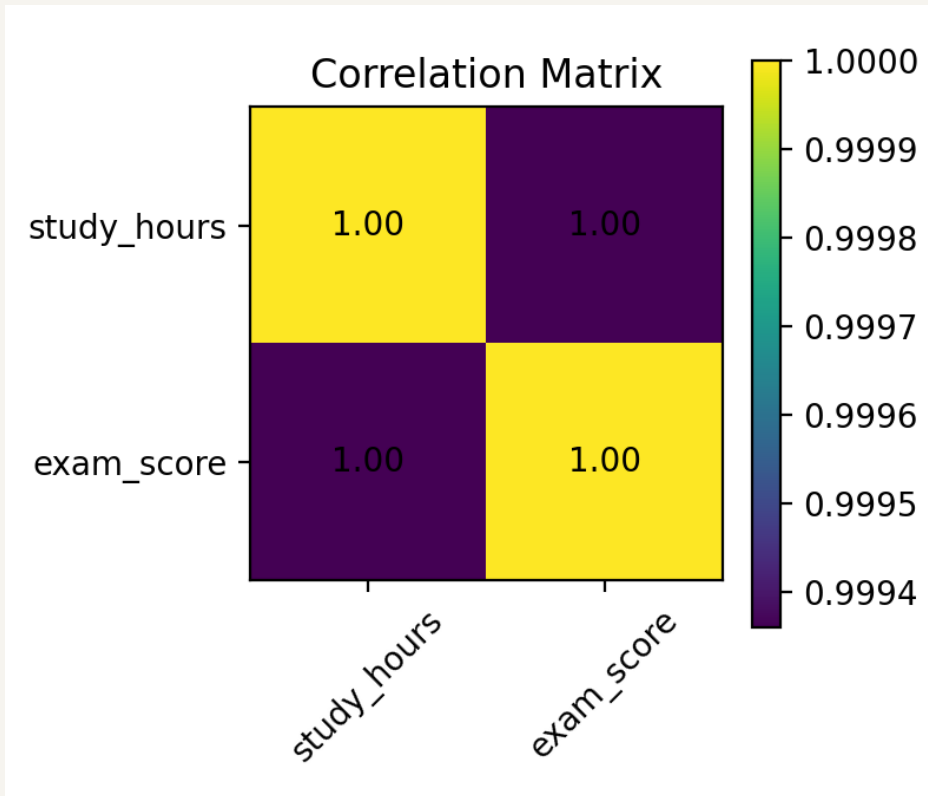
# Alkalmazottak DataFrame
employees_df = pd.DataFrame({
    "employee_id": [101, 102, 103, 104],
    "name": ["John", "Emma", "David", "Sophia"]
})

# Részlegek DataFrame
departments_df = pd.DataFrame({
    "employee_id": [101, 102, 103, 104],
    "department": ["IT", "HR", "Sales", "Finance"]
})

# DataFrame-ek egyesítése az employee_id oszlop alapján
merged_df = pd.merge(employees_df, departments_df, on="employee_id")

print("Merged DataFrame:\n", merged_df)
```


Adatok korrelációjának vizsgálata



	study_hours	exam_score
study_hours	1.000000	0.999361
exam_score	0.999361	1.000000

Python ELTE IK

```
import pandas as pd
import matplotlib.pyplot as plt

# Minta adatok
data = {
    "study_hours": [2, 3, 4, 5, 6, 7, 8, 9],
    "exam_score": [55, 60, 65, 72, 78, 84, 90, 96]
}

data_df = pd.DataFrame(data)

# Korrelációs mátrix kiszámítása
correlation = data_df.corr()

print("Correlation Matrix:")
print(correlation)

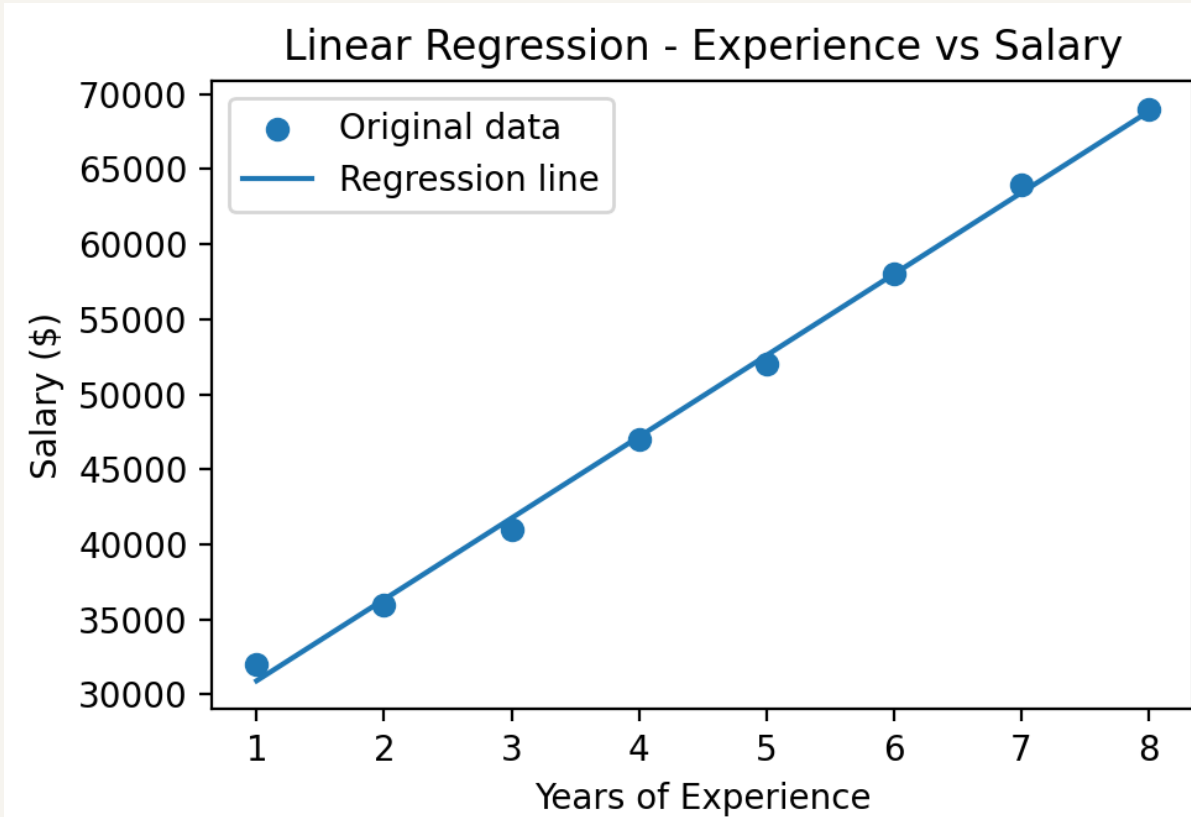
# Korrelációs mátrix megjelenítése
plt.imshow(correlation, interpolation="nearest")
plt.xticks(range(len(correlation.columns)), correlation.columns, rotation=45)
plt.yticks(range(len(correlation.columns)), correlation.columns)

for i in range(len(correlation)):
    for j in range(len(correlation.columns)):
        plt.text(j, i, f"{correlation.iloc[i, j]:.2f}",
                 ha="center", va="center")

plt.colorbar()
plt.title("Correlation Matrix")
plt.tight_layout()
plt.show()
```

Lineáris regresszió végrehajtása

$$\text{Salary} = 5416.67 \times \text{Experience} + 25500.00$$



```
plt.show()
```

```
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression

# Minta adatok
data = {
    "experience": [1, 2, 3, 4, 5, 6, 7, 8],
    "salary": [32000, 36000, 41000, 47000, 52000, 58000, 64000, 69000]
}

data_df = pd.DataFrame(data)

# Független (X) és függő (y) változó
X = data_df[["experience"]]
y = data_df[["salary"]]

# Lineáris regressziós modell létrehozása
model = LinearRegression()
model.fit(X, y)

# Egyenes egyenlete
print(f"Salary = {model.coef_[0]:.2f} × Experience + {model.intercept_:.2f}")

# Előrejelzés
y_pred = model.predict(X)

# Diagram megjelenítése
plt.scatter(data_df["experience"], data_df["salary"],
            color="blue", label="Original data")
plt.plot(data_df["experience"], y_pred,
         color="red", label="Regression line")

plt.xlabel("Years of Experience")
plt.ylabel("Salary ($)")
plt.title("Linear Regression - Experience vs Salary")
plt.legend()
```

Köszönöm a figyelmet!

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)

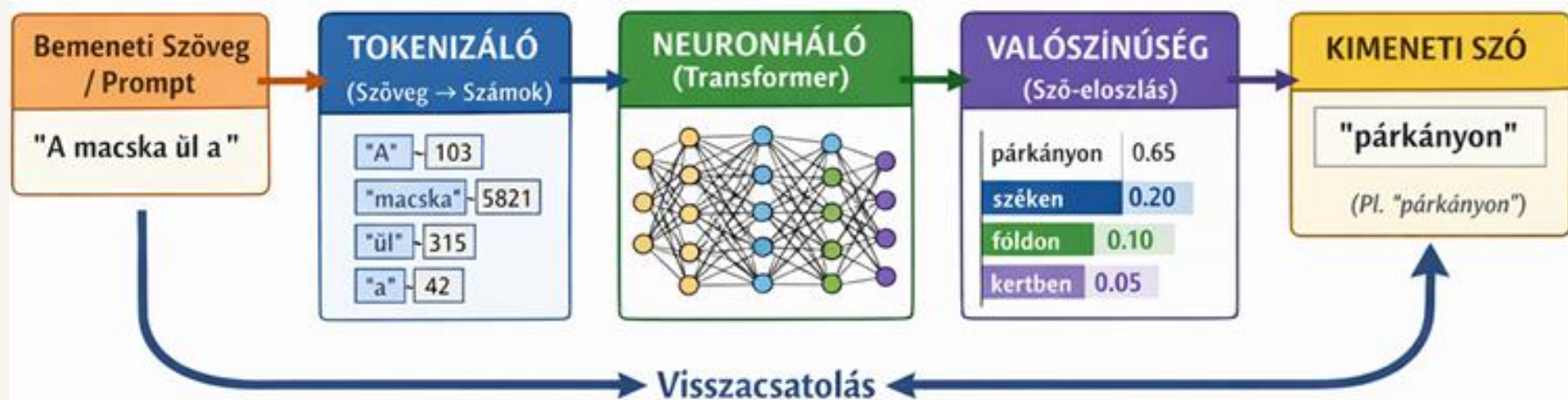
[modellsources.py](#) (pdf állomány megnyitása)

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

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

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

Nyelvi Modellek - LLM

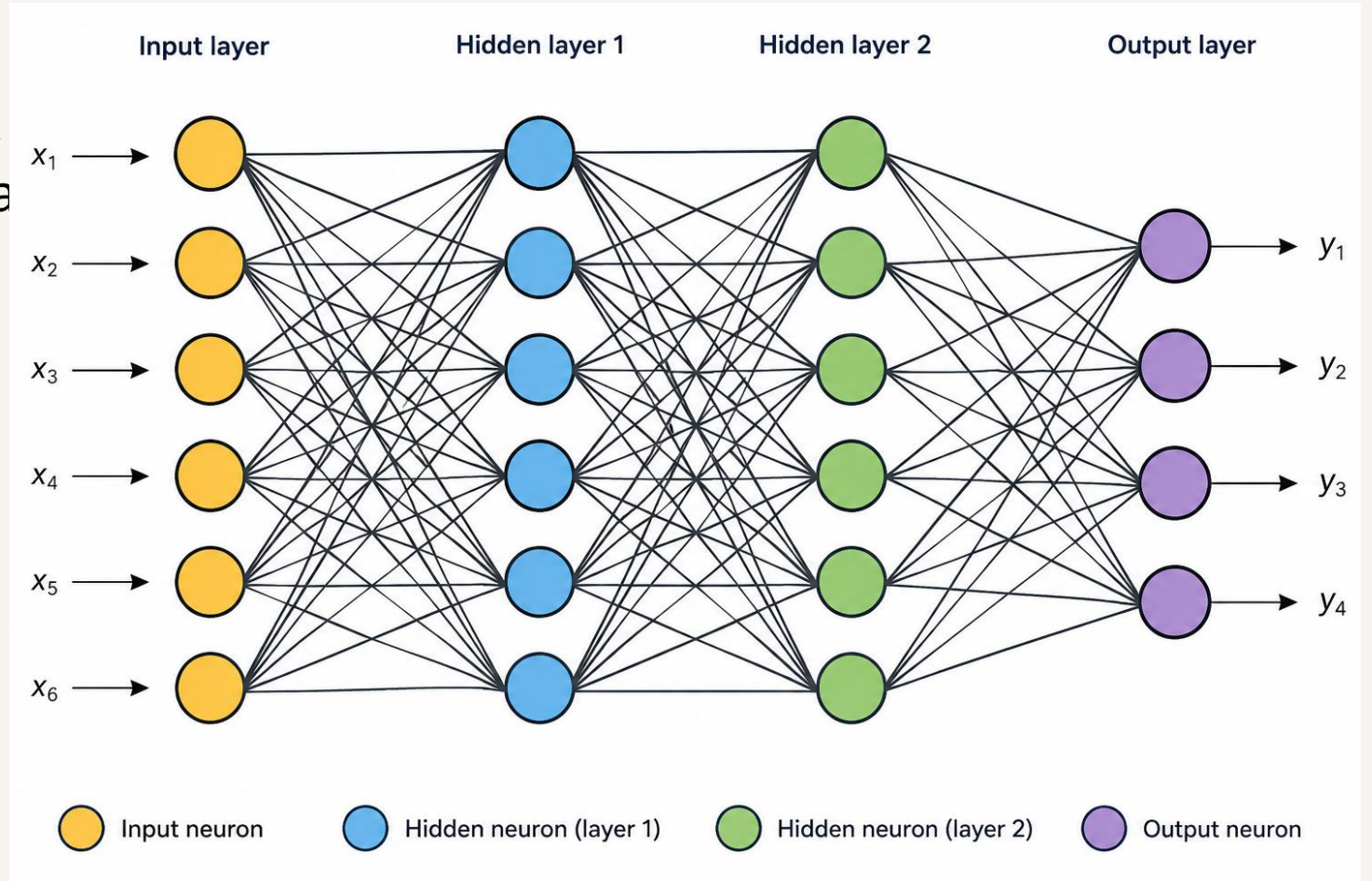


LLM

- 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.

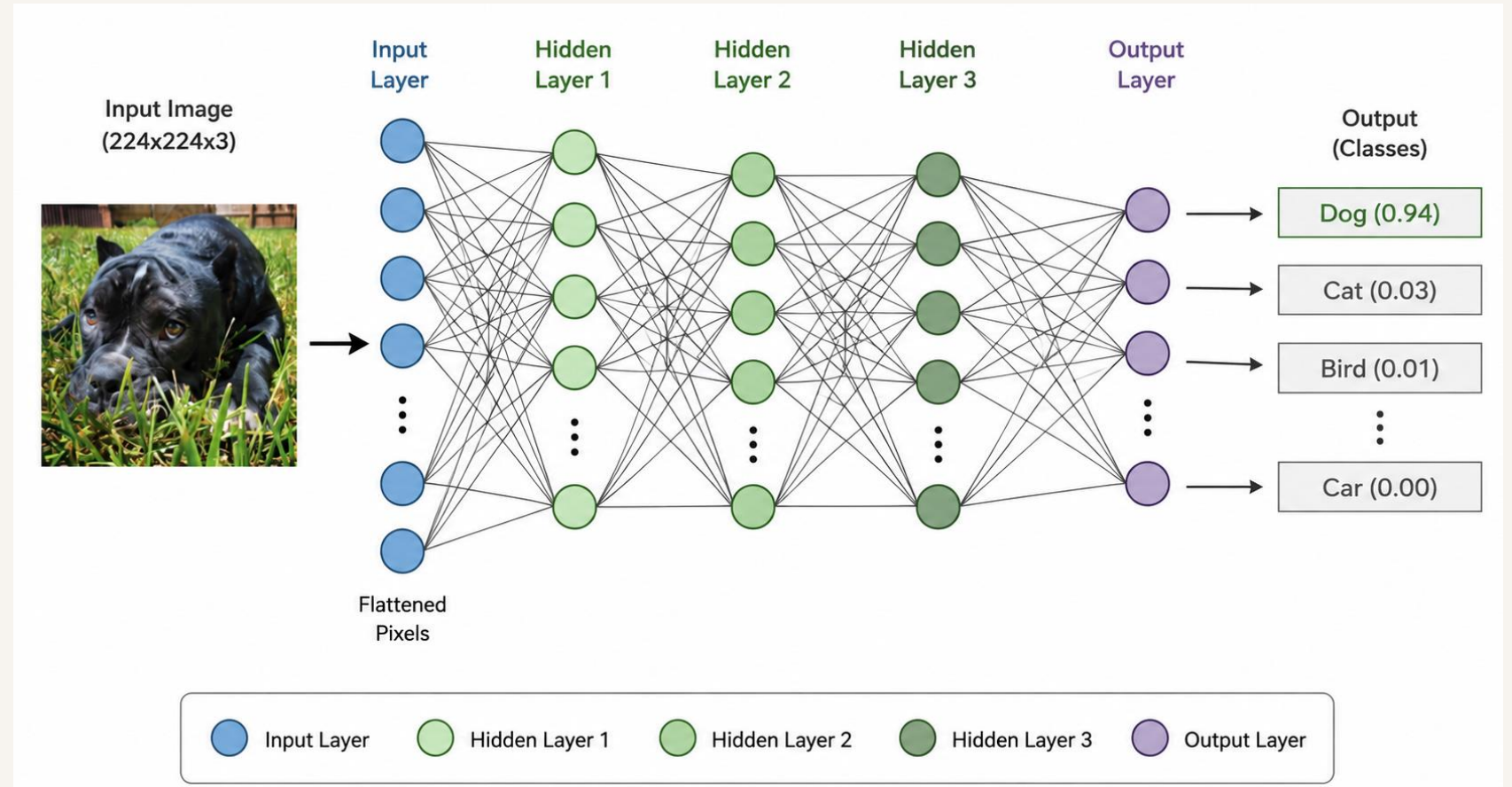
Neurális háló - Pythonban

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)

```
# Bemeneti értékek
inputs = [4.5, 1.2, 0.8, 3.1]

# Súlyok
weights = [0.6, -0.4, 0.9, 0.3]

# Bias
bias = 1.5

# Neuron kimenete
output = (
    inputs[0] * weights[0] +
    inputs[1] * weights[1] +
    inputs[2] * weights[2] +
    inputs[3] * weights[3] +
    bias
)

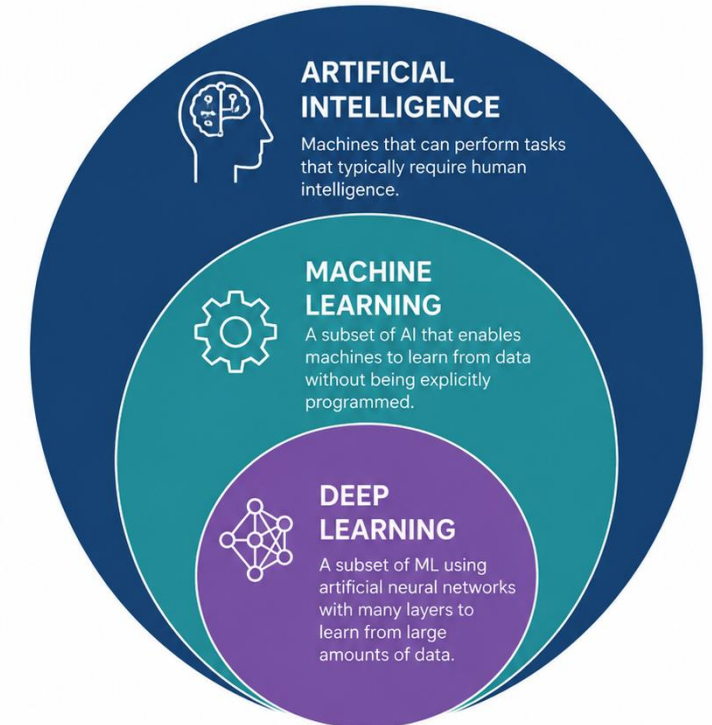
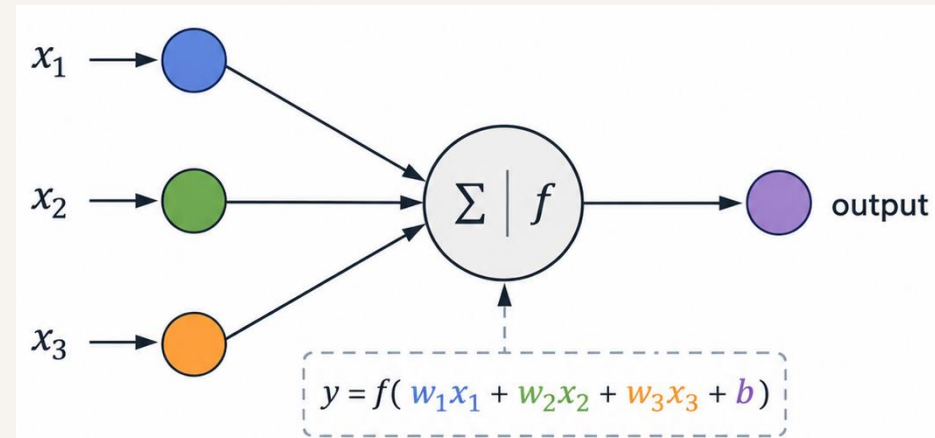
print("Neuron output:", output)
```

```
# Bemeneti értékek
inputs = [4.5, 1.2, 0.8, 3.1]

# Súlyok
weights = [0.6, -0.4, 0.9, 0.3]

# Bias
bias = 1.5
```

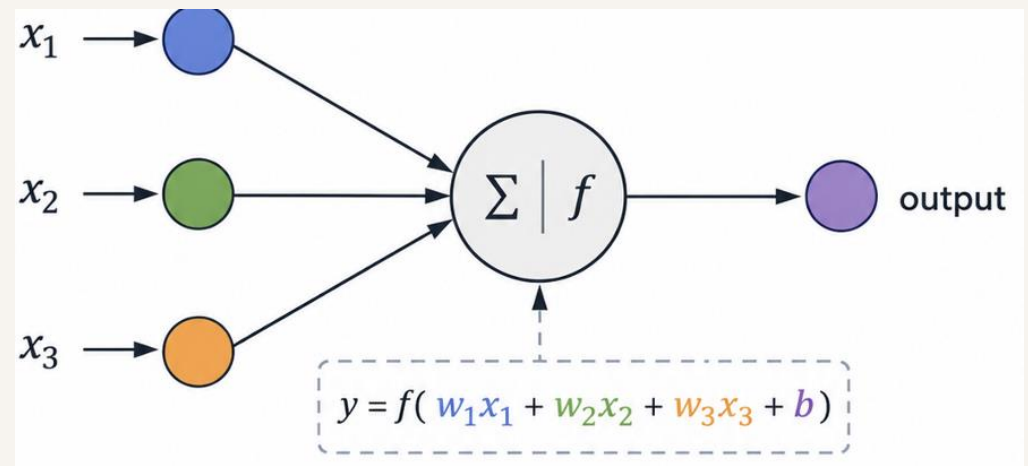
Neuron output: 5.45



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.

$$Z = W * X + B$$

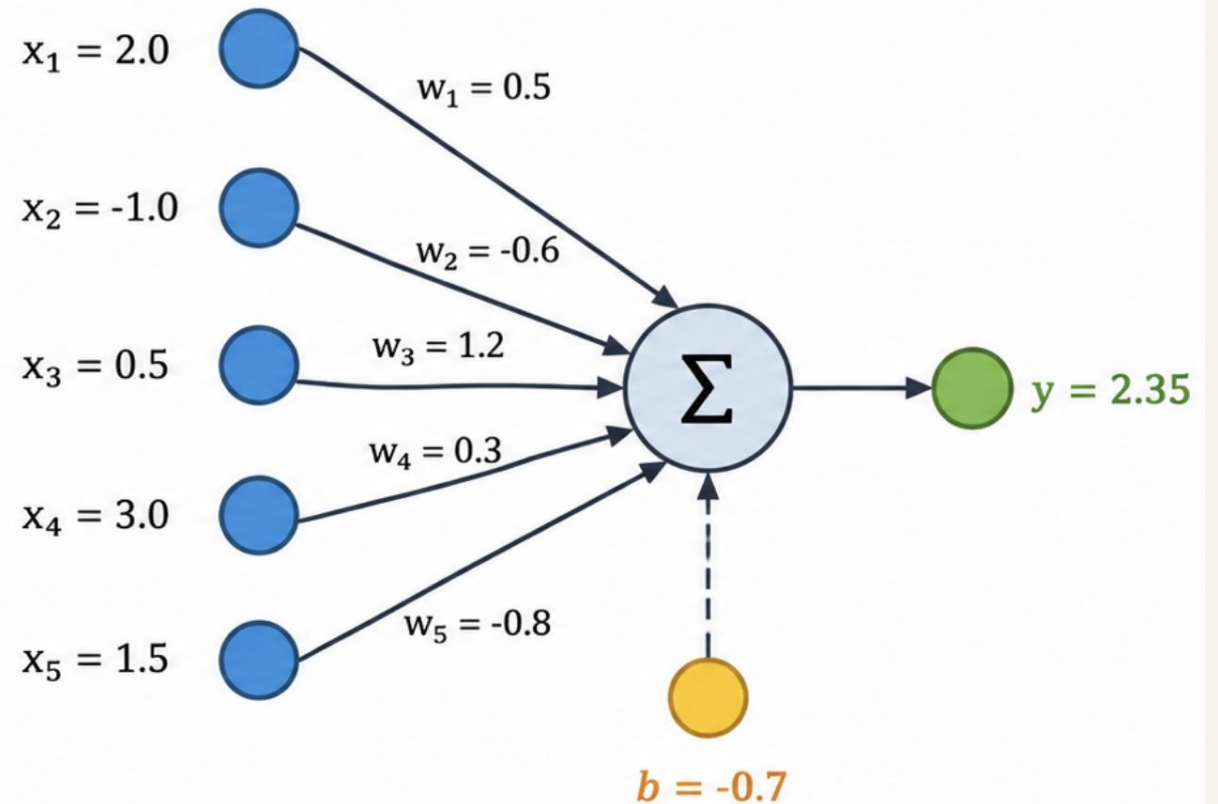


Eredmény kiszámítása egyetlen neuronra

```
inputs  = [2.0, -1.0, 0.5, 3.0, 1.5]
weights = [0.5, -0.6, 1.2, 0.3, -0.8]
bias    = -0.7

# számítás: súlyozott összeg + bias
outputs = np.dot(weights, inputs) + bias

print(outputs)  # -> 2.35
```

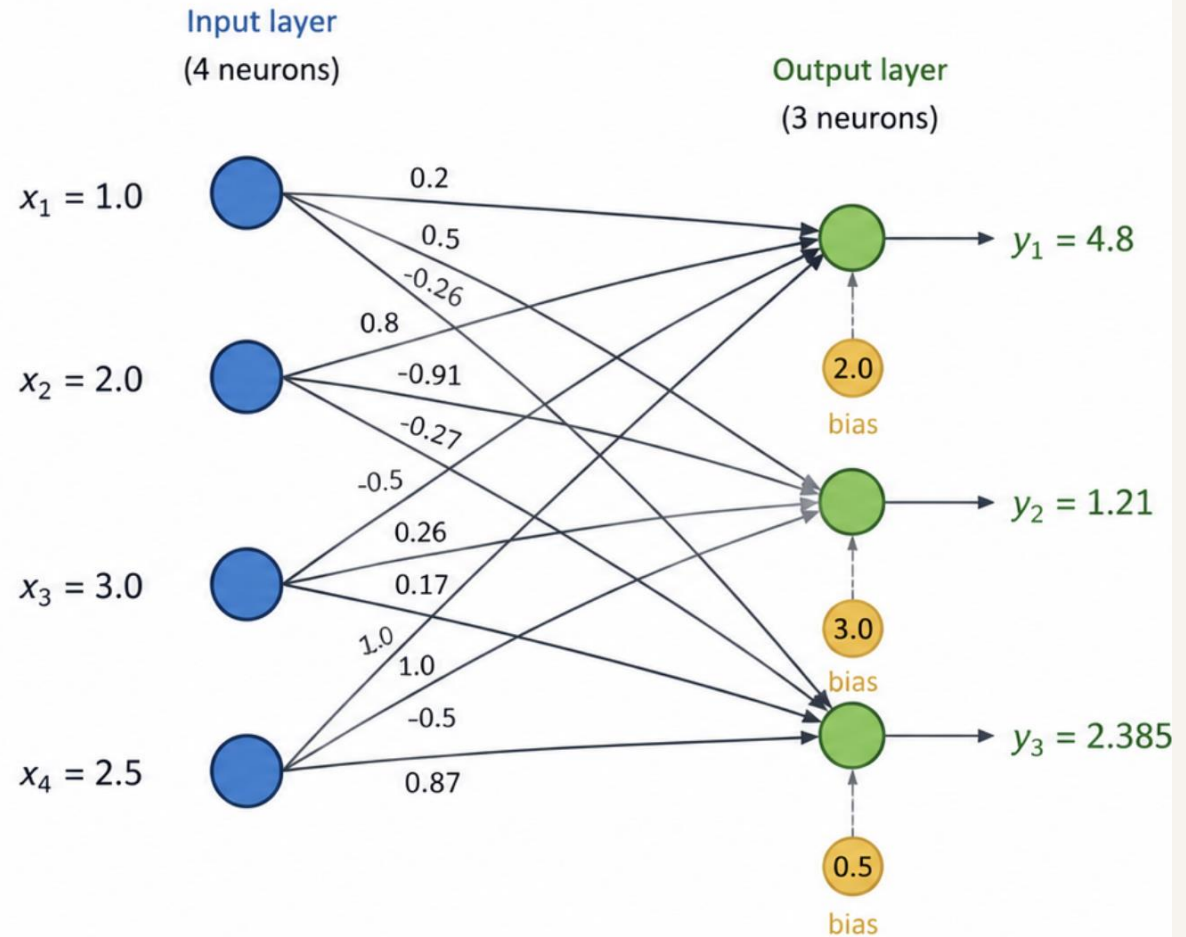


Ugyanez egy teljes layer esetén

```
inputs = [1.0, 2.0, 3.0, 2.5]
weights = [[0.2, 0.8, -0.5, 1.0],
           [0.5, -0.91, 0.26, -0.5],
           [-0.26, -0.27, 0.17, 0.87]]
biases = [2.0, 3.0, 0.5]

# számítás: bemenet · súlyok + biasok
layer_output = np.dot(weights, inputs) + biases

print(layer_output) # -> [4.8  1.21  2.385]
```



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: $-1 = 0$, $1 = 1$)

$$\frac{1}{1 + e^{-z}}$$

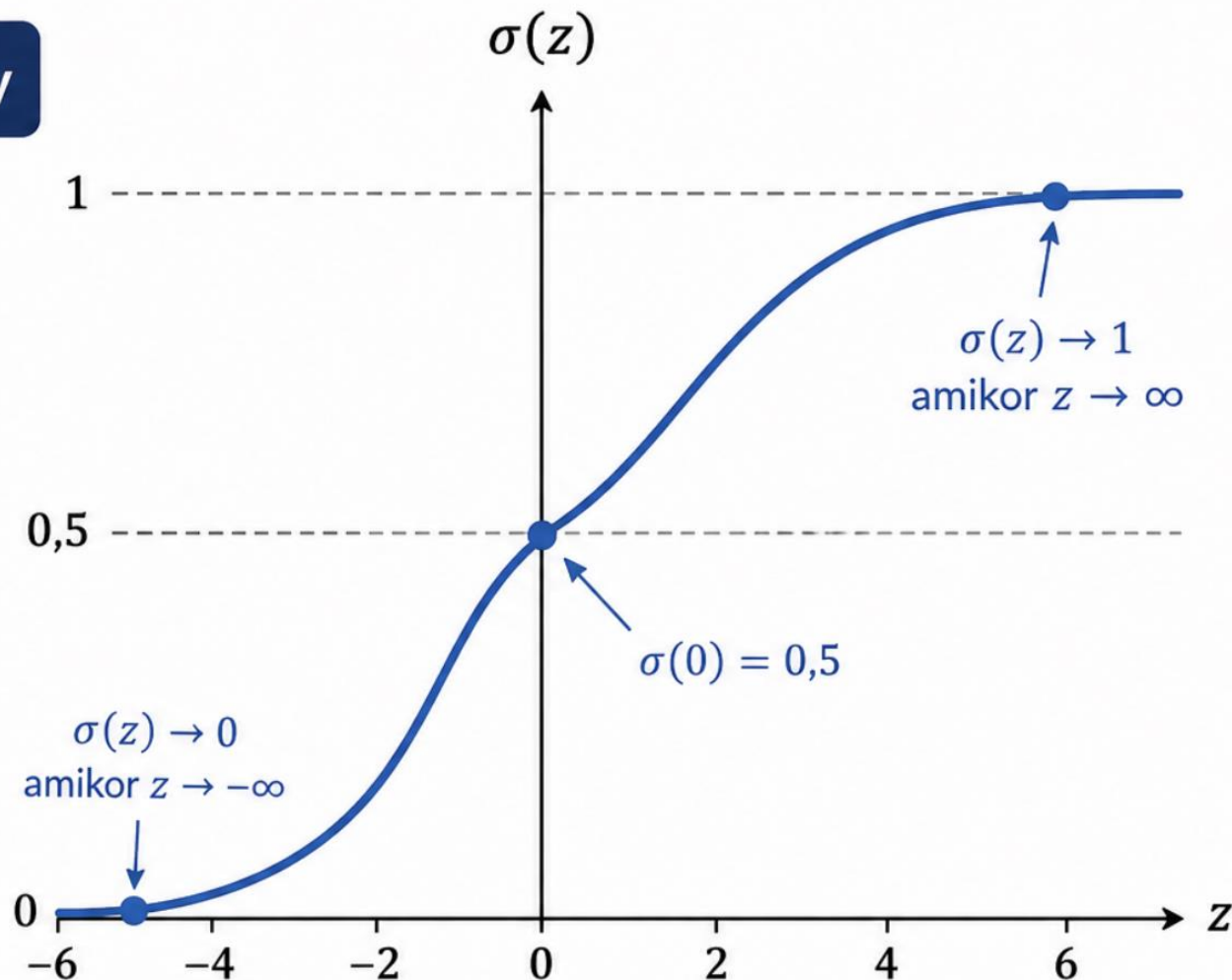
Sigmoid (logisztikus) függvény

Képlet:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Mit jelent?

- A szigmoid függvény bármely valós számhoz (z) egy 0 és 1 közötti értéket rendel.
- Kimeneti tartomány: $(0, 1)$
- Gyakran használják bináris osztályozási feladatokban és neurális hálóknban.
- Az S-alakú görbe simán közelít 0-hoz és 1-hez.



z	-6	-4	-2	0	2	4	6
$\sigma(z)$	0,0025	0,0180	0,1192	0,5000	0,8808	0,9820	0,9975

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

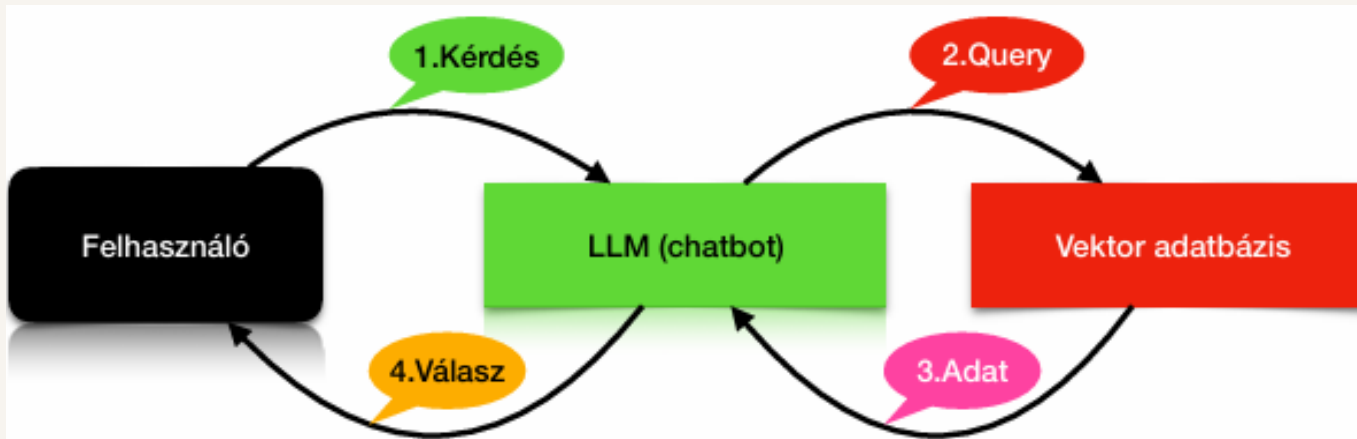
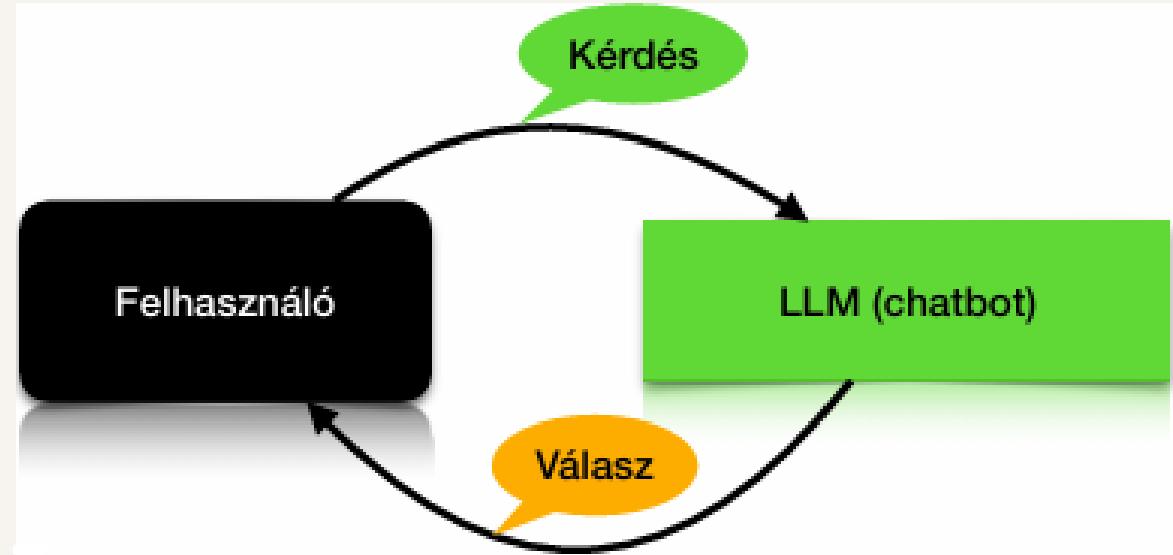
LLM – Llama, API based Ollama, WEB interface

- LLM
- RAG,
- API based Ollama,
- WEB interface - később

Retrieval-Augmented Generation (RAG)

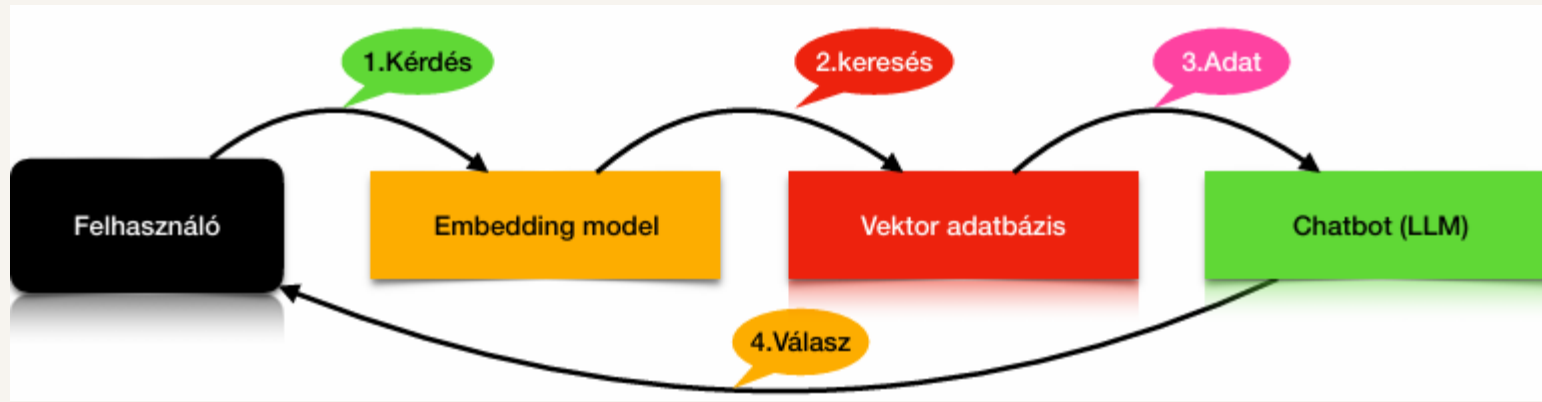
RAG

Retrieval-Augmented Generation



RAG modell

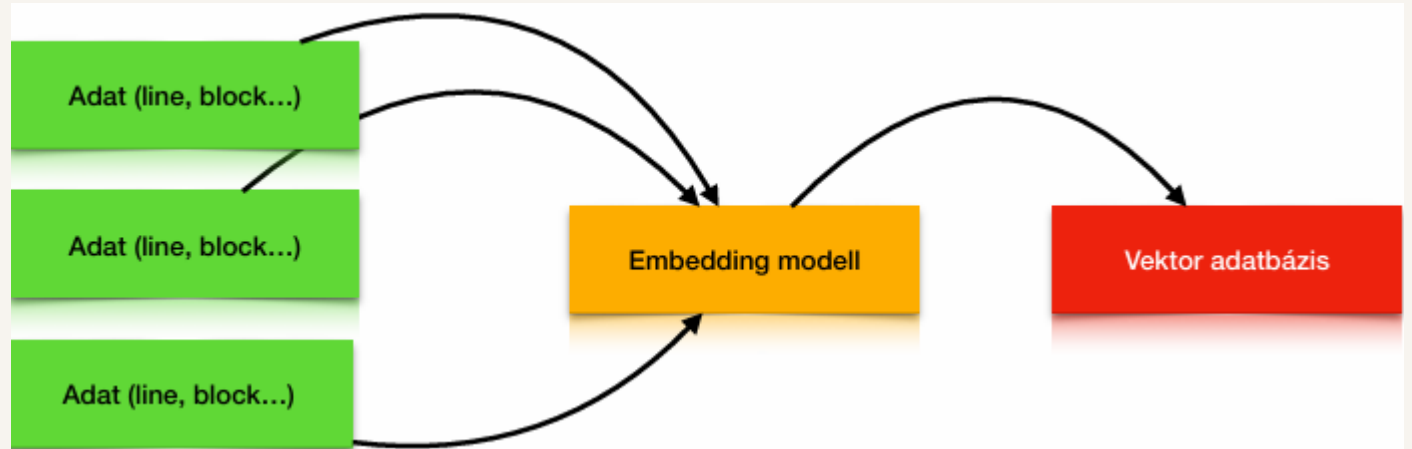
Vektor adatbázis - tudás és a hozzá tartozó beágyazási vektorok tárolására (pl.: pgvector, Qdrant)



Embedding modell - előre betanított nyelvi modell, ami a bemeneti szöveget szemantikai jelentést rögzítő vektorreprezentációkká - ezekben keressük az információt

Indexelés

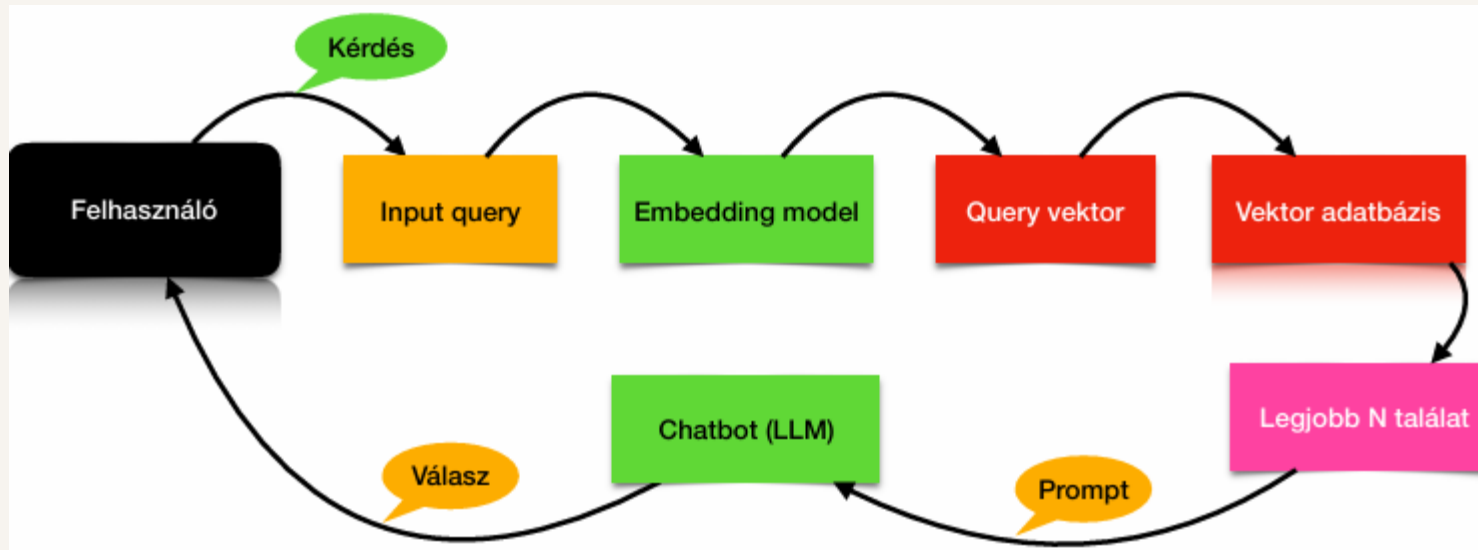
Az adatokat (pl.: sorok) kis darabokra bontjuk, majd minden egyes darabnak kiszámítjuk a vektoros reprezentációját, amely hatékonyan kereshető a válasz generálásához. Ez lehet, sor, bekezdés, nagyobb blokk



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>

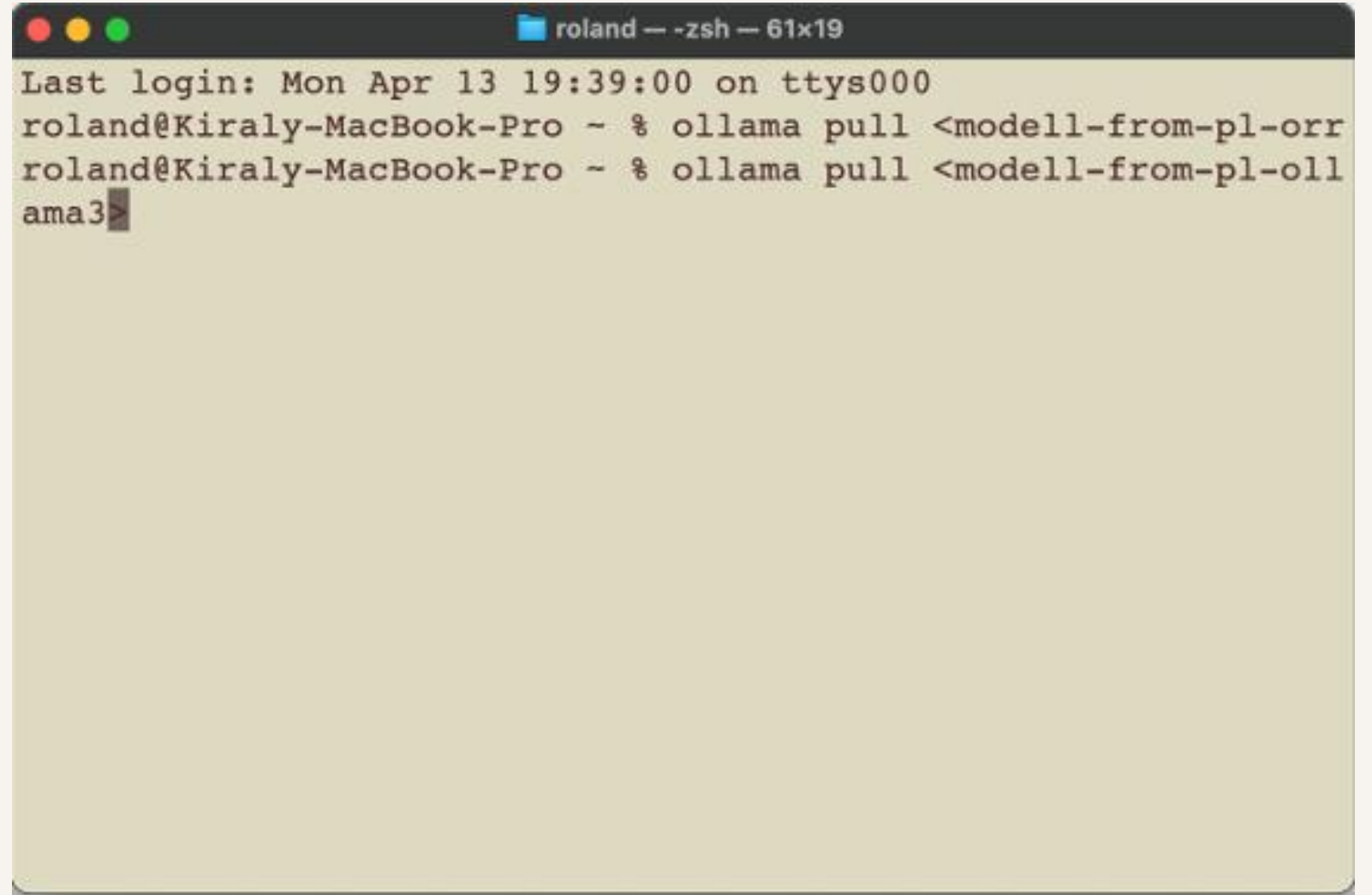
ollama pull <https://huggingface.co/CompendiumLabs/bge-base-en-v1.5-gguf>

A nyelvi modellhez a modell a Hugging Face-ről:

Kisebb modellek: 8GB ram körül, de jobb minőségben

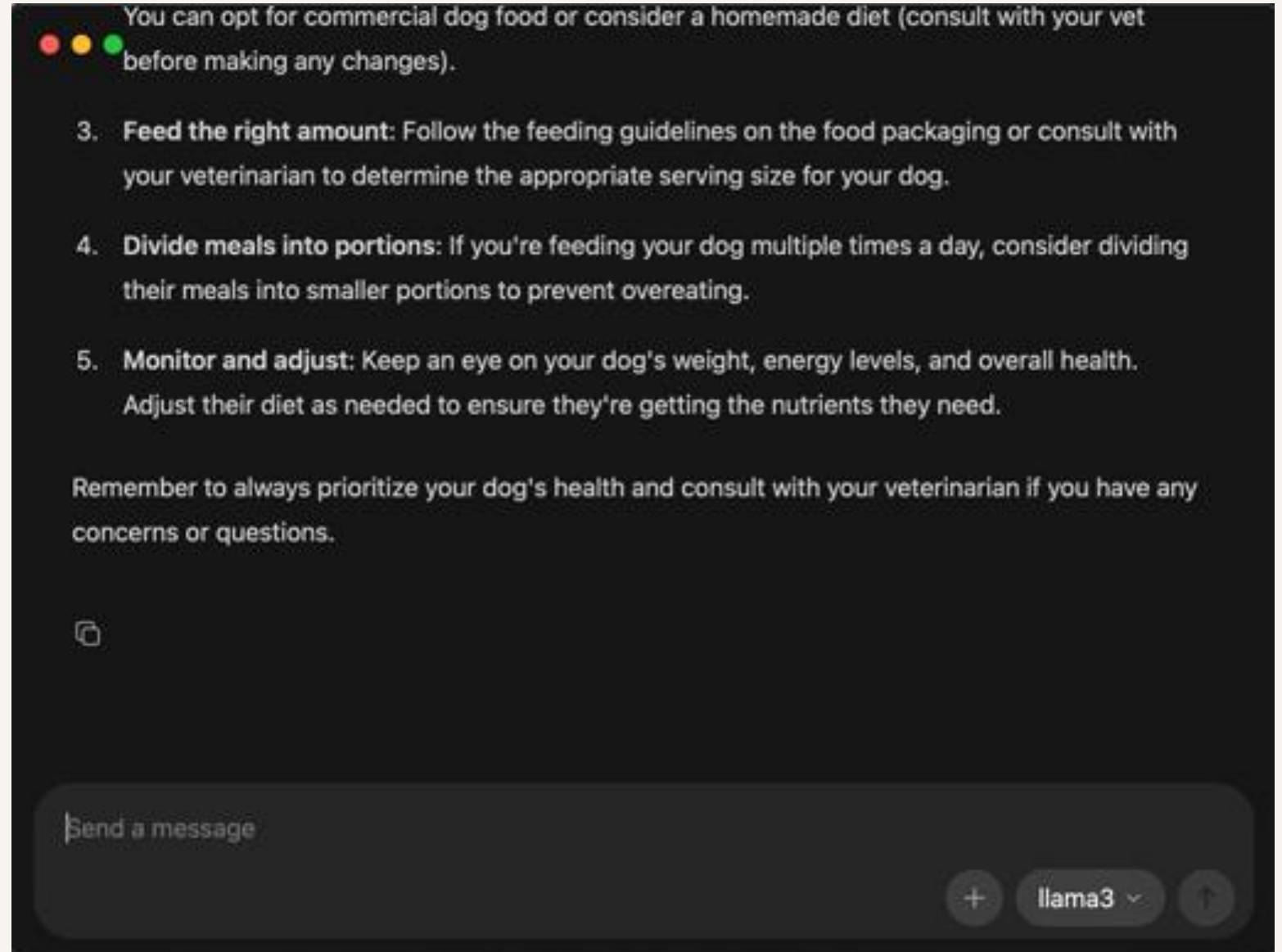
<https://huggingface.co/bartowski/Llama-3.2-3B-Instruct-GGUF>

Ollama

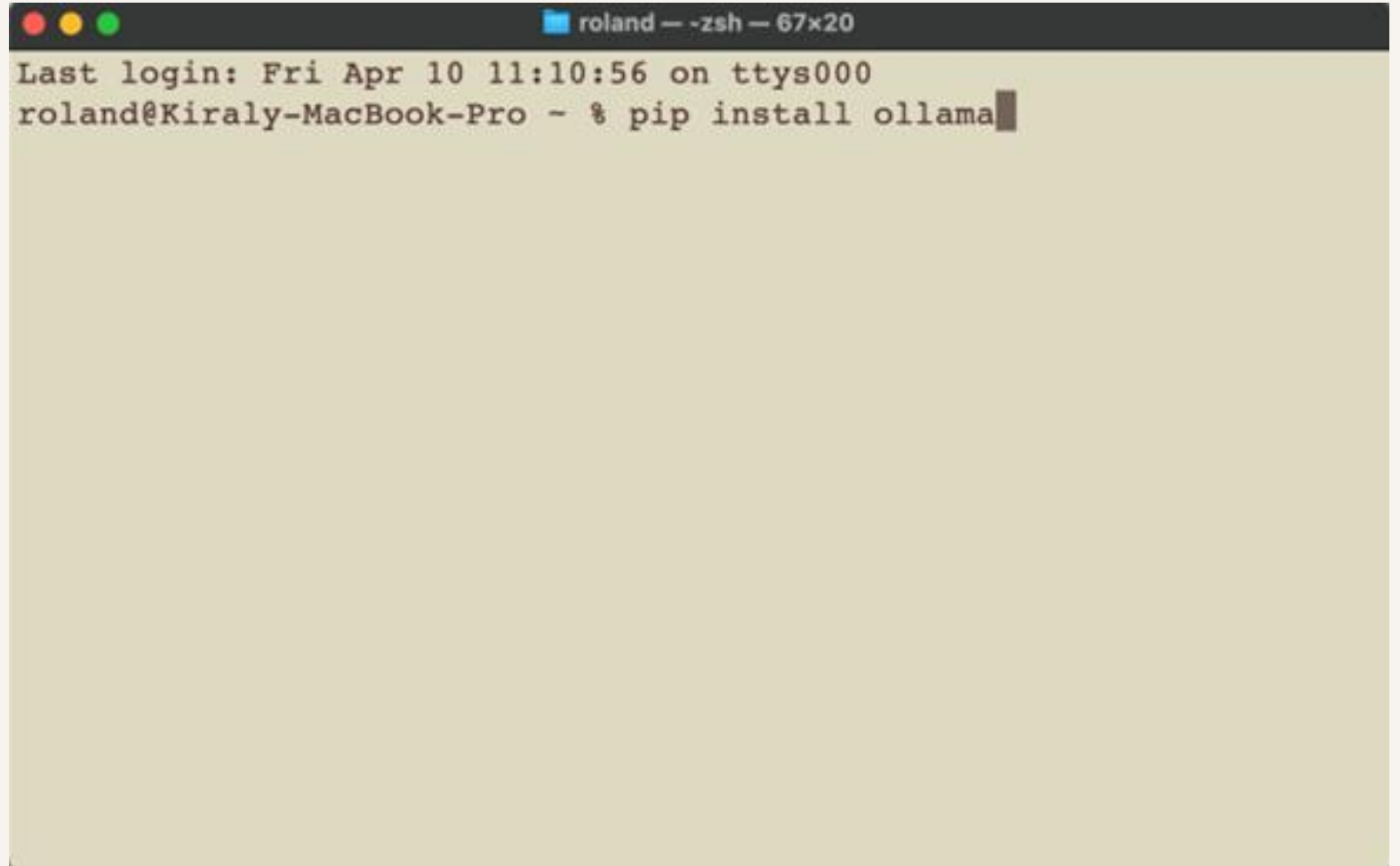
A terminal window titled 'roland - zsh - 61x19' with standard macOS window controls (red, yellow, green buttons). The terminal text shows a login message and two lines of the 'ollama pull' command. The second line is partially cut off at the end.

```
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



```
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!

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)

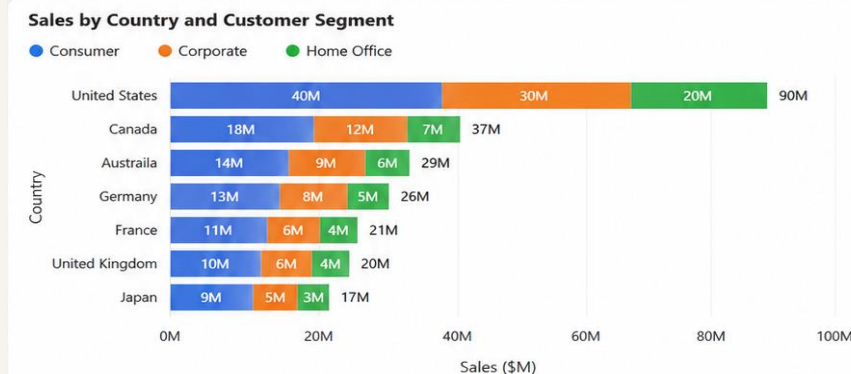
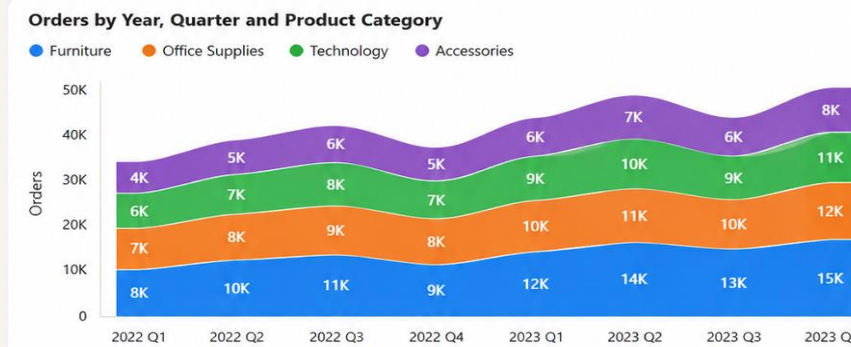
Adatvizualizáció bevezetés

- Az adatvizualizáció segíti az adatok megértését és kommunikálását
- Kiemelt szerepe van a döntéshozatal támogatásában és a mintázatok felismerésében

Miért fontos az adatvizualizáció?

Az összetett adatok áttekinthető megjelenítése – a nagy mennyiségű információ könnyebb értelmezése. Mintázatok és eltérések feltárása – trendek, összefüggések és rendellenességek gyors felismerése. A döntéshozatal támogatása – a vizuális megjelenítés segíti a gyorsabb és megalapozottabb döntéseket. Hatékony információközlés – az eredmények egyszerű, szemléletes és könnyen érthető bemutatása. Összefüggések feltérképezése – kapcsolatok, ok-okozati viszonyok és rejtett mintázatok azonosítása.

Python ELTE IK



Adatvizualizáció bevezetés

Példa:

Az ingatlanárak trendvonalal 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



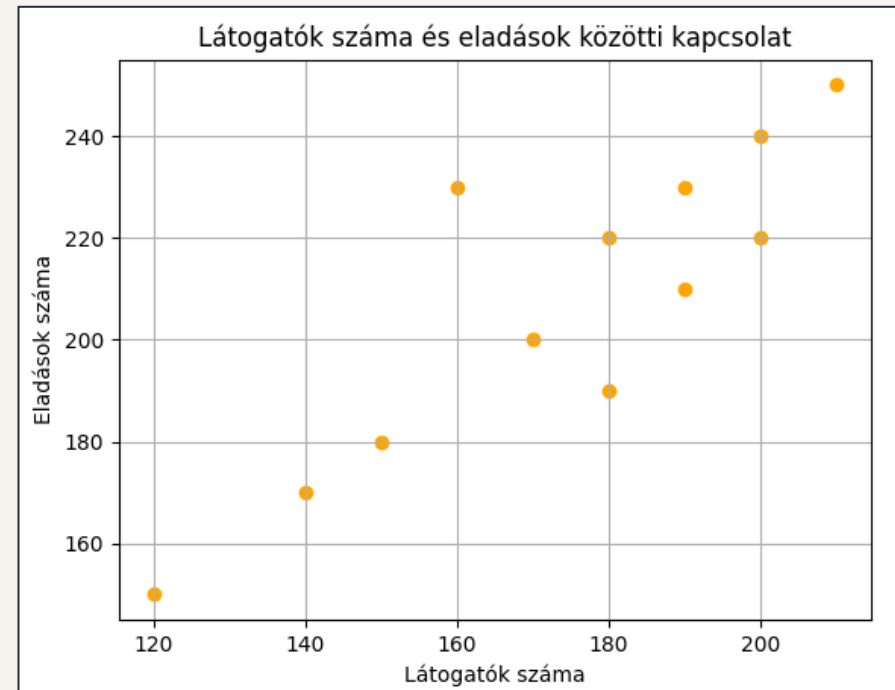
A Matplotlib alapjai: diagramok készítése

- 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 vezetérvonalait.

Scatter plot diagram rajzolása a plt.plot() függvénnnyel

[scatter.py \(.pdf\)](#)

```
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()
```

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

napok = ['Hétfő', 'Kedd', 'Szerda', 'Csütörtök', 'Péntek', 'Szombat', 'Vasárnap']
rendelések = [45, 52, 48, 60, 75, 90, 65]

plt.plot(napok, rendelések,
         marker='s',
         linestyle='--',
         color='green')

plt.title('Webáruház napi rendelései')
plt.xlabel('Napok')
plt.ylabel('Rendelések száma')
plt.grid(True)

plt.show()
```

A vonaldiagram segít bemutatni egy változó időbeli alakulását.

[linedig.py \(.pdf\)](#)

Tengelyek, címek és feliratok testreszabása

más jelölő (marker='D'),

eltérő szín (purple), vastagabb vonal (linewidth=2), egyedi cím- és tengelyszínek,

30°-kal elforgatott tengelyfeliratok,

pontosított rácsvonalak (linestyle=':')

[differentplot.py \(.pdf\)](#)

```
import matplotlib.pyplot as plt

hónapok = ['Január', 'Február', 'Március', 'Április', 'Május', 'Június']
kölcsonzések = [320, 410, 380, 450, 520, 490]

plt.plot(hónapok, kölcsonzések,
         marker='D',
         color='purple',
         linewidth=2)

plt.title('Könyvtári kölcsönzések alakulása',
         fontsize=16,
         fontweight='bold',
         color='darkred')

plt.xlabel('Hónapok',
         fontsize=12,
         color='darkblue')

plt.ylabel('Kölcsönzések száma',
         fontsize=12,
         color='darkblue')

plt.xticks(rotation=30)

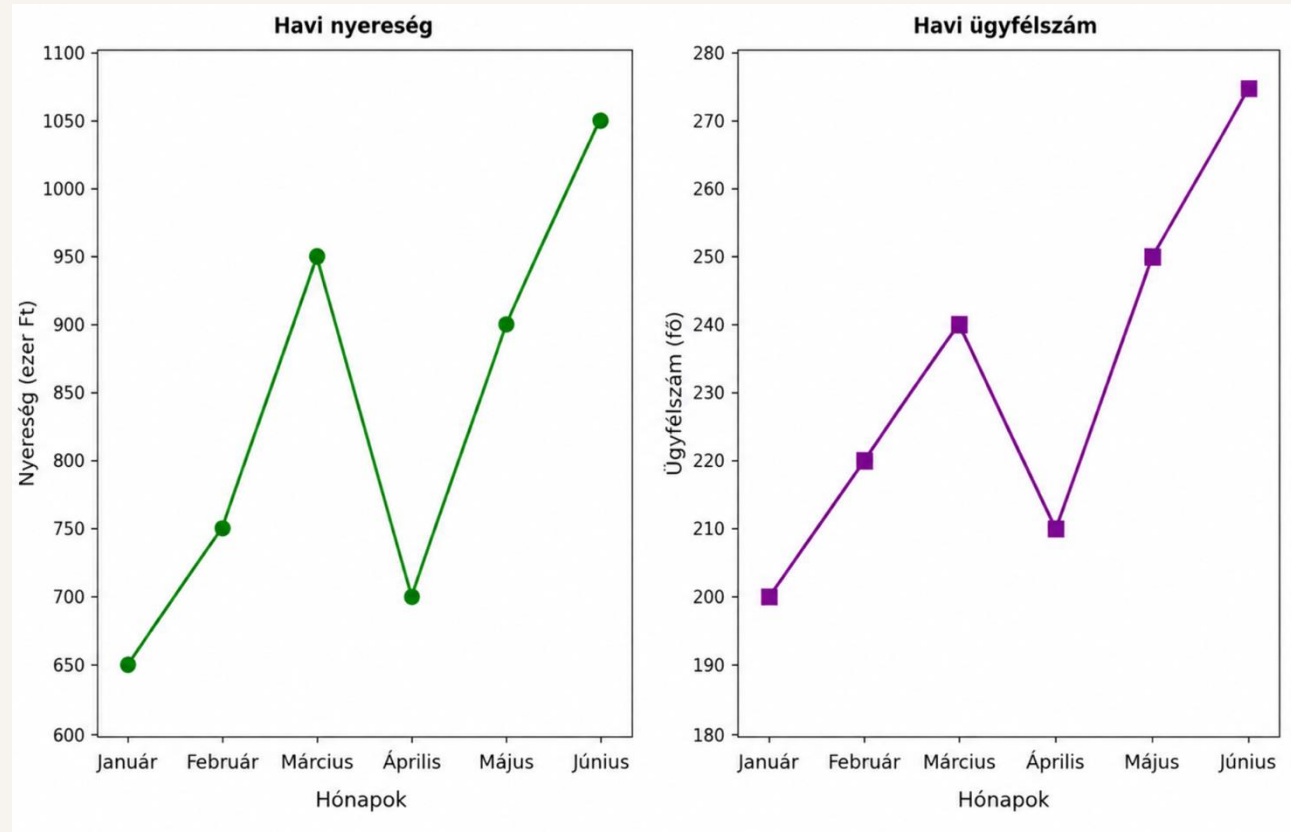
plt.grid(color='lightgray',
        linestyle=':',
        linewidth=1)

plt.show()
```

Többszörös diagramok egy ábrán belül (subplotok)

Példa:

Különböző diagram egy ablakban – az egyik a nyereséget, a másik a az ügyfélszámot 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.

```
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()
```

[subplot.py \(.pdf\)](#)

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

# Vizsgaeredmények (%)
eredmenyek = [52, 68, 75, 81, 90, 73, 66, 84,
               91, 77, 69, 58, 95, 87, 72, 80]

# Hisztogram készítése
plt.hist(eredmenyek,
         bins=5,
         color='lightgreen',
         edgecolor='black')

plt.title('Vizsgaeredmények eloszlása')
plt.xlabel('Eredmény (%)')
plt.ylabel('Tanulók száma')

plt.grid(axis='y',
        linestyle=':',
        linewidth=0.8)

plt.show()
```

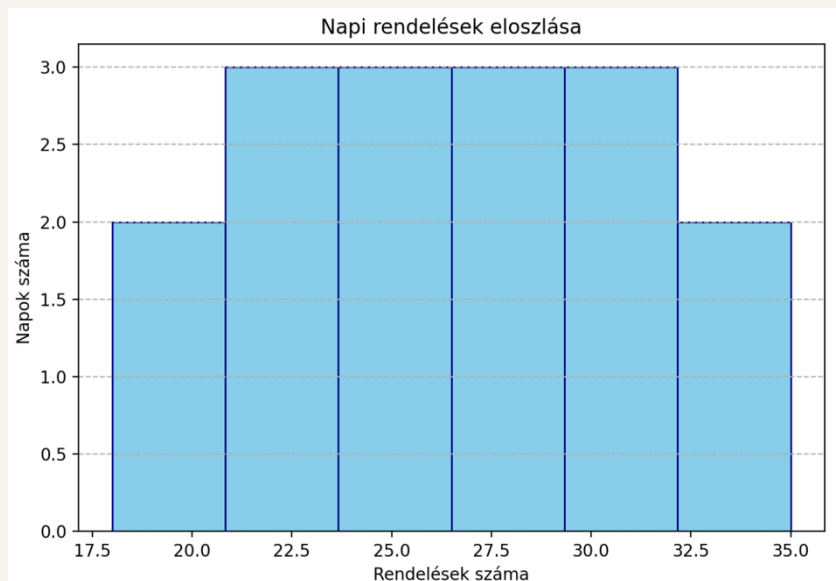

Box plotok használata adatok szórásának bemutatására

A box plotok (dobozábrák) vizuálisan mutatják az adatok szórását, középvértékét, 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.



```
import matplotlib.pyplot as plt

# Napi rendelések száma
rendelesek = [18, 22, 25, 30, 27, 24, 35, 29,
              21, 26, 32, 28, 31, 23, 20, 34]

# Hisztogram készítése
plt.hist(rendelesek,
         bins=6,
         color='skyblue',
         edgecolor='navy')

plt.title('Napi rendelések eloszlása')
plt.xlabel('Rendelések száma')
plt.ylabel('Napok száma')

plt.grid(axis='y',
        linestyle='--',
        linewidth=0.8)

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

Medián és quartilisek

```
import matplotlib.pyplot as plt
import numpy as np

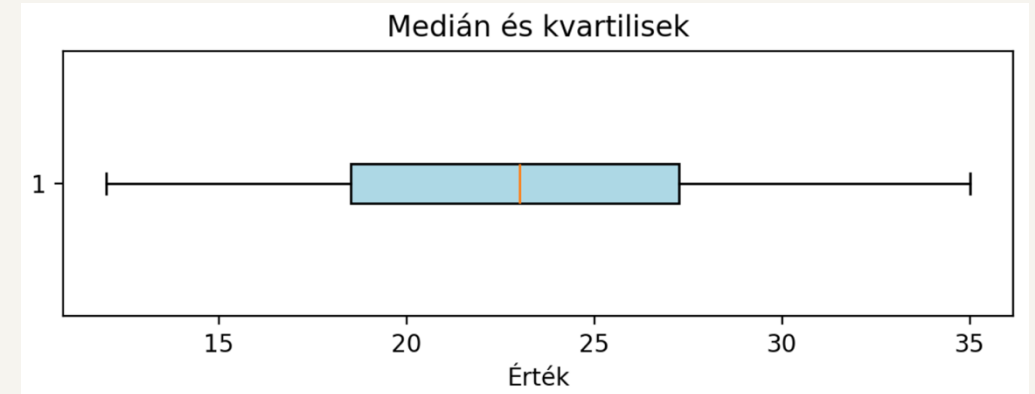
# Minta adatok
adatok = [12, 15, 18, 20, 22, 24, 25, 28, 30, 35]

# Statisztikák kiírása
print("Medián:", np.median(adatok))
print("Q1:", np.percentile(adatok, 25))
print("Q3:", np.percentile(adatok, 75))
print("IQR:", np.percentile(adatok, 75) - np.percentile(adatok, 25))

# Boxplot
plt.boxplot(adatok, vert=False, patch_artist=True,
            boxprops=dict(facecolor="lightblue"))

plt.title("Medián és kvartilisek")
plt.xlabel("Érték")

plt.show()
```



Stílusok beállítása

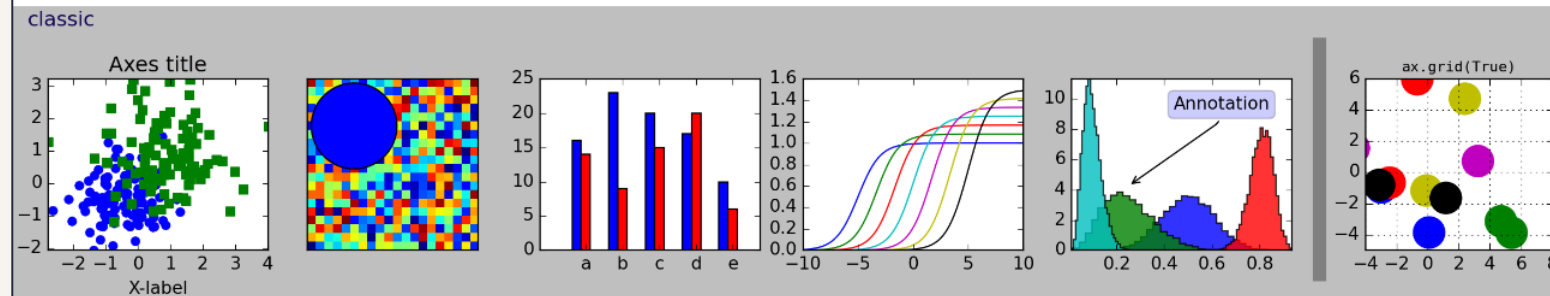
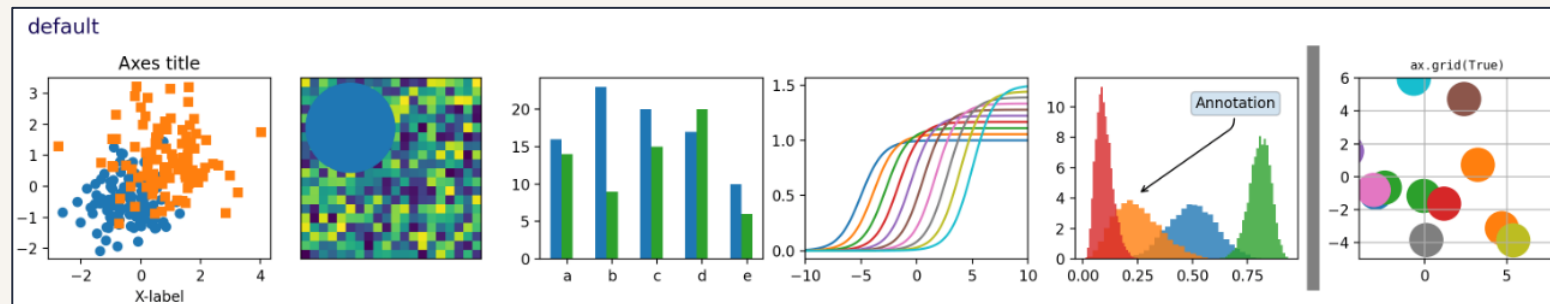
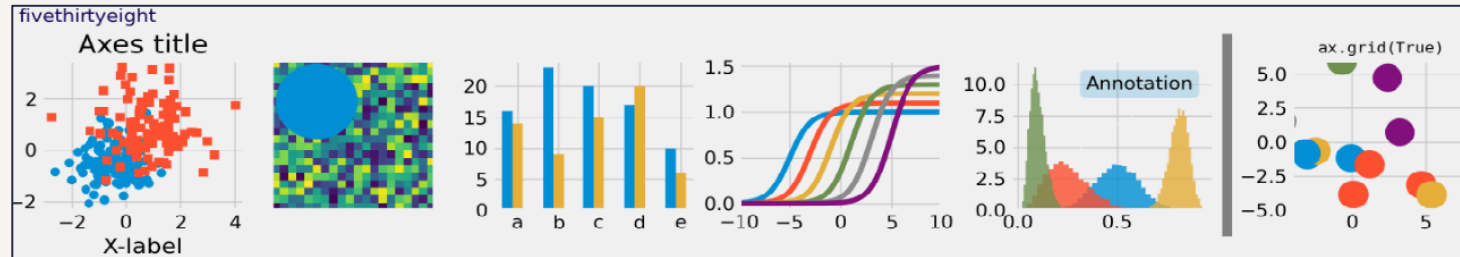
A **plt.style.use()** paraméterével stíluslapot választhatunk a megjelenítéshez az alapbeállítástól eltérően.

[styles.py \(.pdf\)](#)

```
# Stílus beállítása  
plt.style.use('fivethirtyeight')
```

Stílusok beállítása

[Style sheets reference — Matplotlib 3.9.2 documentation](#)

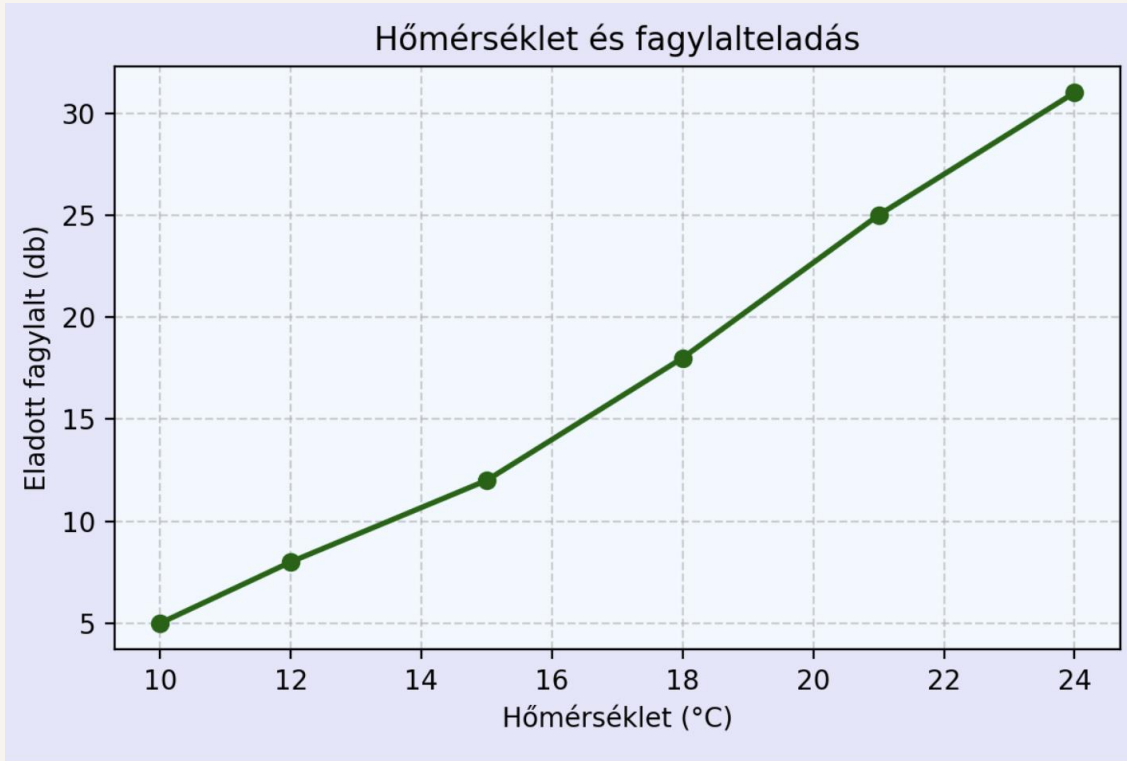


Adatok megjelenítése Pandas használatával

- 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.
- Csomagok importálása:

```
import pandas as pd  
import matplotlib.pyplot as plt
```


Háttérszínek beállítása



```
import matplotlib.pyplot as plt
import pandas as pd

# Minta adatok
data = {
    "Hőmérséklet": [10, 12, 15, 18, 21, 24],
    "Fagyalt": [5, 8, 12, 18, 25, 31]
}

df = pd.DataFrame(data)

# Ábra létrehozása
fig = plt.figure(facecolor="lavender")
ax = plt.gca()
ax.set_facecolor("aliceblue")

# Adatok ábrázolása
plt.plot(df["Hőmérséklet"],
         df["Fagyalt"],
         marker="o",
         color="darkgreen",
         linewidth=2)

plt.title("Hőmérséklet és fagyalteladás")
plt.xlabel("Hőmérséklet (°C)")
plt.ylabel("Eladott fagyalt (db)")
plt.grid(True, linestyle="--", alpha=0.6)

plt.show()
```

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.

	Hónap	Átlaghőmérséklet
0	Január	1
1	Február	3
2	Március	8
3	Április	14
4	Május	19

```
import pandas as pd
import matplotlib.pyplot as plt

# CSV fájl beolvasása
adatok = pd.read_csv("homerseklet.csv")

honapok = adatok["Hónap"]
homerseklet = adatok["Átlaghőmérséklet"]

# Vonaldiagram készítése
plt.plot(honapok,
         homerseklet,
         marker='o',
         linestyle='-',
         color='darkorange')

plt.title("Havi átlaghőmérséklet")
plt.xlabel("Hónap")
plt.ylabel("Hőmérséklet (°C)")

plt.show()

# Az első néhány sor megjelenítése
print(adatok.head())
```

Adatmegjelenítés beépített Pandas plot metódussal

A Pandas DataFrame és Series objektumai beépített plot() metódussal rendelkeznek.

Vonal-diagram készítése beépített Pandas metódussal

A **kind** paraméter határozza meg a diagram típusát (pl. line, bar), az **x és y** a tengelyek megadása, a **marker** és **color** paraméterek a megjelenést szabályozzák.

	Nap	Rendelések	Bevétel
0	2025-03-01	42	315000
1	2025-03-02	55	402000
2	2025-03-03	48	356000
3	2025-03-04	61	458000
4	2025-03-05	57	425000

[pandplotMain.py \(.pdf\)](#)

```
import pandas as pd
import matplotlib.pyplot as plt

# CSV fájl beolvasása
adatok = pd.read_csv("rendelesek.csv")

# Vonaldiagram készítése
adatok.plot(
    x="Nap",
    y="Rendelések",
    kind="line",
    marker="o",
    color="royalblue",
    title="Napi rendelések"
)

plt.xlabel("Napok")
plt.ylabel("Rendelések száma")
plt.grid(True)

plt.show()

# A beolvasott adatok megjelenítése
print(adatok.head())
```

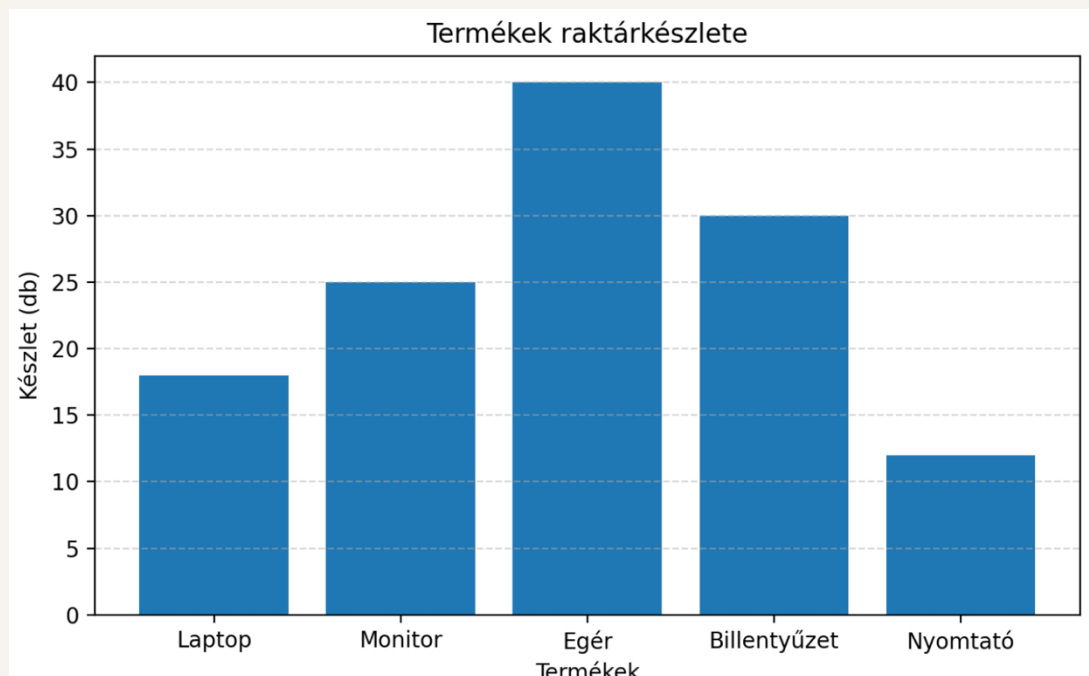
Oszlopdiagram készítése

```
# Oszlopdiagram készítése
adatok.plot(x='Hónap',
            y='Kiadás',
            kind='bar',
            color='skyblue',
            title='Havi kiadások')

plt.xlabel('Hónapok')
plt.ylabel('Kiadás (ezer Ft)')
plt.show()
```

A `kind='bar'` paraméterrel oszlopdiagramot hozhatunk létre. Ez a diagramtípus kiválóan alkalmas különböző kategóriák vagy időszakok adatainak összehasonlítására, így könnyen áttekinthetővé teszi az értékek közötti különbségeket.

Hisztogram készítése az eladások eloszlásának bemutatására



```
import pandas as pd
import matplotlib.pyplot as plt

# Minta adatok
adatok = pd.DataFrame({
    'Termék': ['Laptop', 'Monitor', 'Egér', 'Billentyűzet', 'Nyomtató'],
    'Készlet': [18, 25, 40, 30, 12]
})

# Oszlopdiagram készítése
adatok.plot(x='Termék',
            y='Készlet',
            kind='bar',
            color='royalblue',
            title='Termékek raktárkészlete')

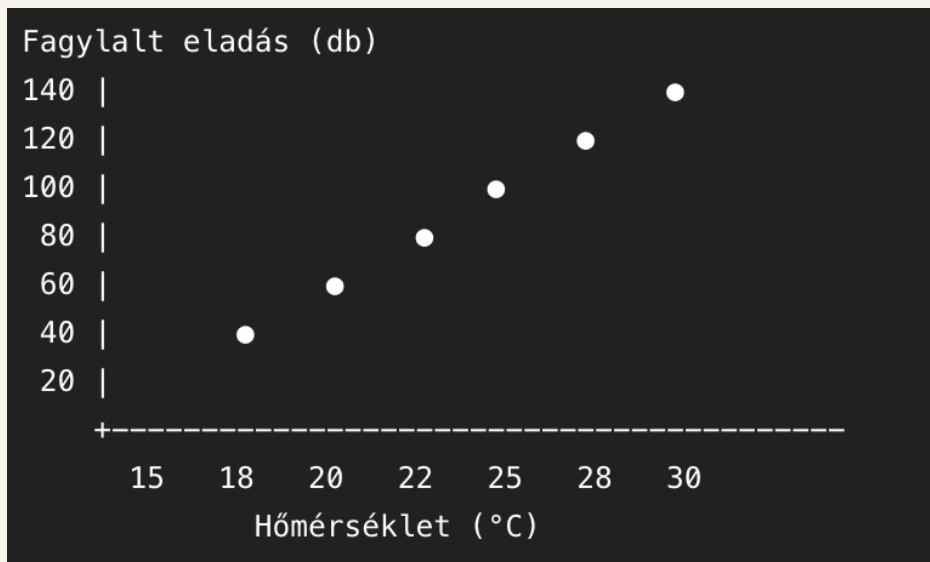
plt.xlabel('Termékek')
plt.ylabel('Készlet (db)')
plt.grid(axis='y', linestyle='--', alpha=0.5)

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



[plotandscatter.py \(.pdf\)](#)

```
import pandas as pd
import matplotlib.pyplot as plt

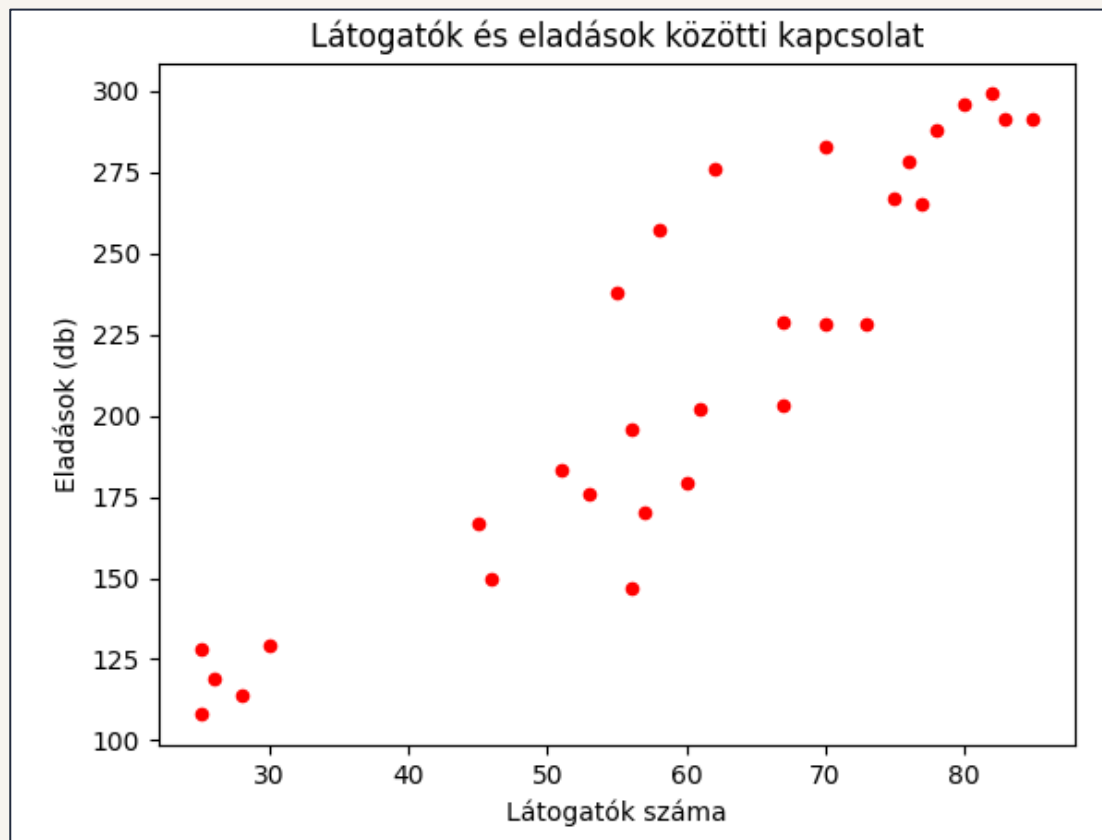
# Minta adatok
adatok = pd.DataFrame({
    'Hőmérséklet': [15, 18, 20, 22, 25, 28, 30],
    'Fagyalt_eladás': [40, 55, 68, 80, 95, 120, 140]
})

# Scatter diagram készítése
adatok.plot(kind='scatter',
            x='Hőmérséklet',
            y='Fagyalt_eladás',
            color='blue',
            title='Hőmérséklet és fagyalteladás')

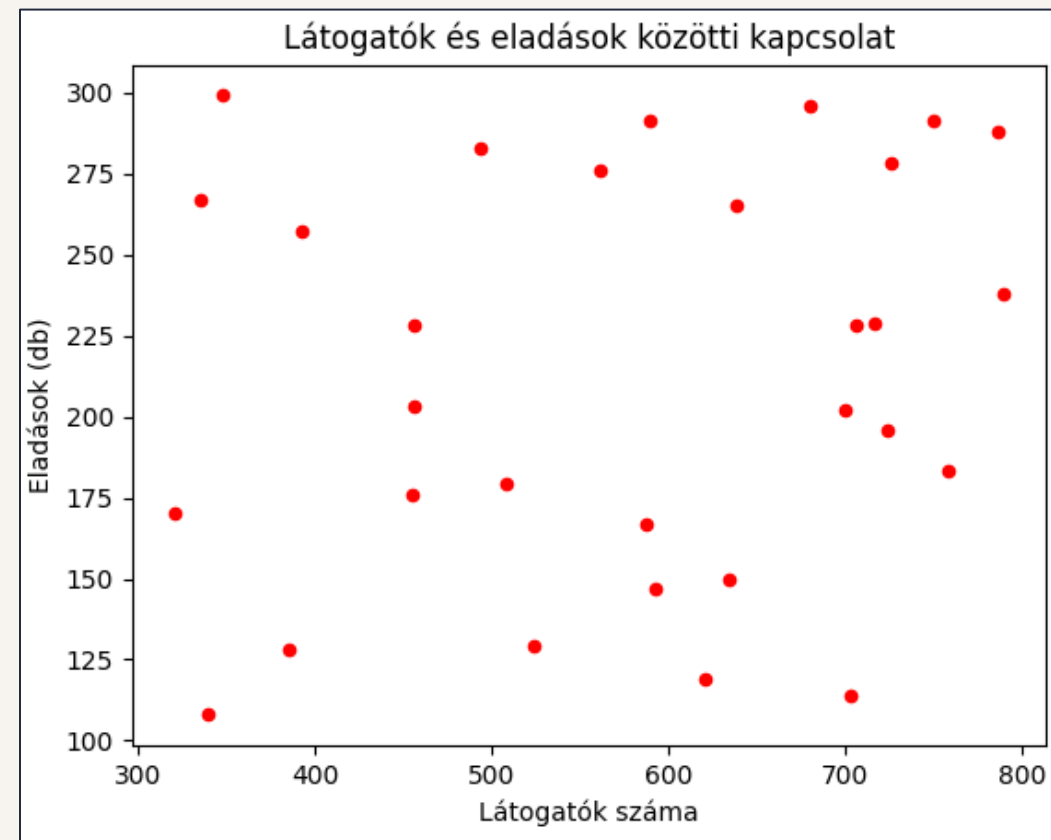
plt.xlabel('Hőmérséklet (°C)')
plt.ylabel('Eladott fagyaltok (db)')
plt.grid(True)

plt.show()
```

Scatter plot a korreláció vizsgálatához



sales1.csv



sales.csv

```
import matplotlib.pyplot as plt
```

```
# 1. példa – pozitív kapcsolat
```

```
adatok = pd.DataFrame({  
    'Órák': [1,2,3,4,5,6,7,8,9,10],  
    'Pontszám': [45,50,58,63,68,74,81,87,92,96]  
})
```

```
adatok.plot(kind='scatter',  
            x='Órák',  
            y='Pontszám',  
            color='blue',  
            title='Tanulási idő és vizsgaeredmény')
```

```
plt.xlabel('Tanulással töltött órák')  
plt.ylabel('Vizsgaeredmény (%)')  
plt.grid(True)  
plt.show()
```

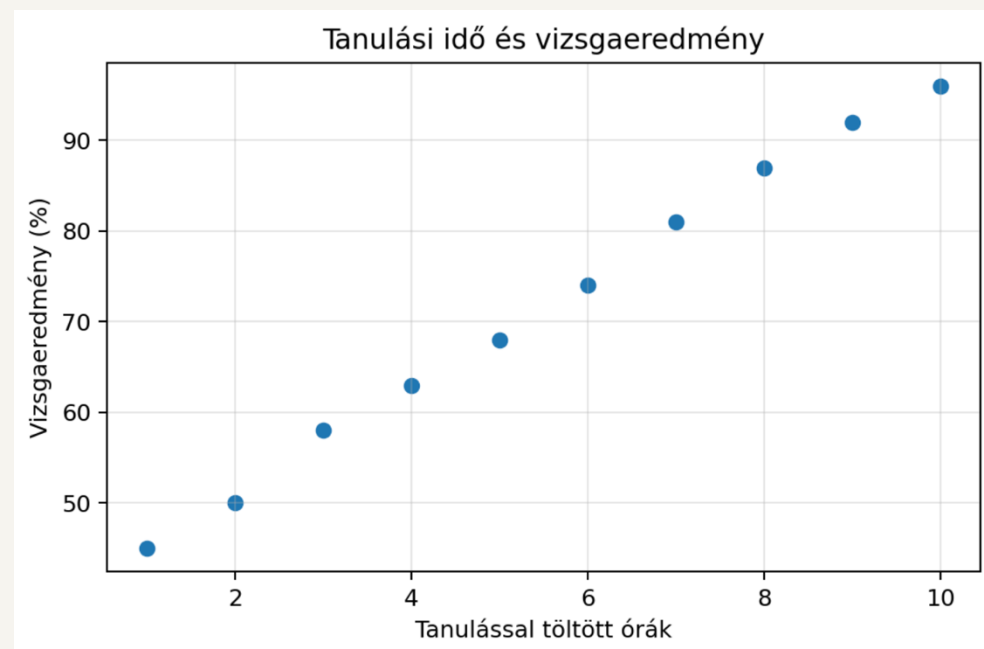
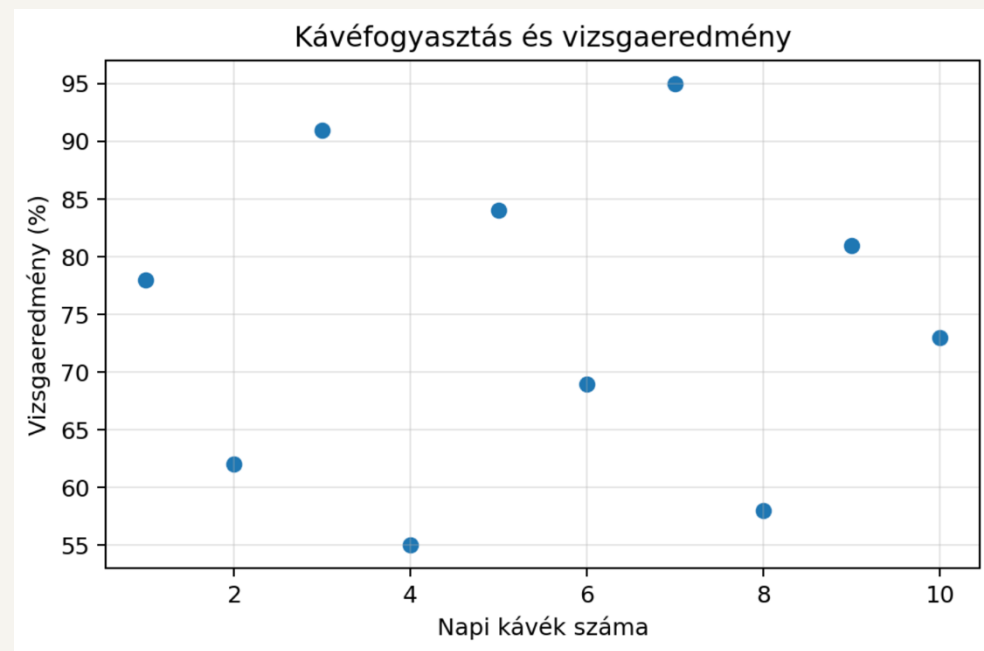
```
# 2. példa – nincs egyértelmű kapcsolat
```

```
adatok = pd.DataFrame({  
    'Kávé': [1,2,3,4,5,6,7,8,9,10],  
    'Pontszám': [78,62,91,55,84,69,95,58,81,73]  
})
```

```
adatok.plot(kind='scatter',  
            x='Kávé',  
            y='Pontszám',  
            color='red',  
            title='Kávéfogyasztás és vizsgaeredmény')
```

```
plt.xlabel('Napi kávék száma')  
plt.ylabel('Vizsgaeredmény (%)')  
plt.grid(True)  
plt.show()
```

Scatter plot a korreláció vizsgálatához



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 \(.pdf\)](#)

```
import matplotlib.pyplot as plt

# Adatok
honapok = ['Jan', 'Feb', 'Már', 'Ápr', 'Máj', 'Jún']
kiadas = [820, 900, 950, 920, 980, 1050]
megrendeles = [120, 145, 170, 160, 190, 215]

# Első tengely
fig, ax1 = plt.subplots(figsize=(8, 5))

ax1.plot(honapok, kiadas,
         marker='o',
         color='green',
         linewidth=2)

ax1.set_xlabel('Hónapok')
ax1.set_ylabel('Kiadások (ezer Ft)', color='green')
ax1.tick_params(axis='y', labelcolor='green')

# Második tengely
ax2 = ax1.twinx()

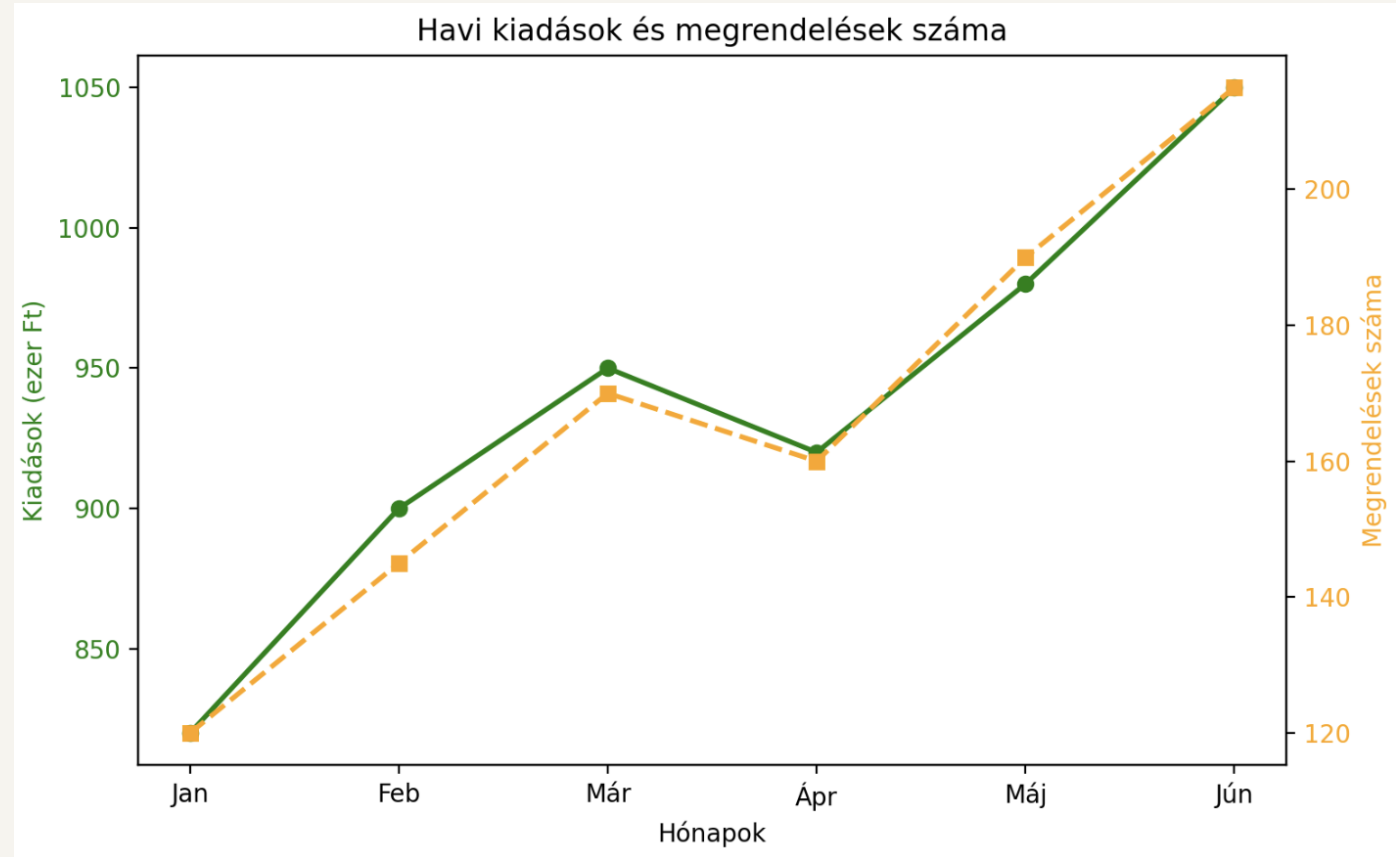
ax2.plot(honapok, megrendeles,
         marker='s',
         linestyle='--',
         color='orange',
         linewidth=2)

ax2.set_ylabel('Megrendelések száma', color='orange')
ax2.tick_params(axis='y', labelcolor='orange')

plt.title('Havi kiadások és megrendelések száma')
plt.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 \(.pdf\)](#)



Overlay diagram készítése

Több adatsor együttes megjelenítése: a példában egy város napi legmagasabb hőmérsékletét és csapadékmennyiségét ábrázoljuk ugyanazon a diagramon. A két adatsor összevetésével könnyebben megfigyelhető, hogyan változnak az időjárási értékek a hét során.

```
import matplotlib.pyplot as plt

napok = [
    'Hétfő', 'Kedd', 'Szerda',
    'Csütörtök', 'Péntek',
    'Szombat', 'Vasárnap'
]

latogatok = [120, 145, 138, 160, 175, 210, 190]
rendelesek = [18, 22, 20, 25, 28, 35, 31]

plt.plot(
    napok,
    latogatok,
    marker='o',
    linewidth=2,
    label='Látogatók száma'
)

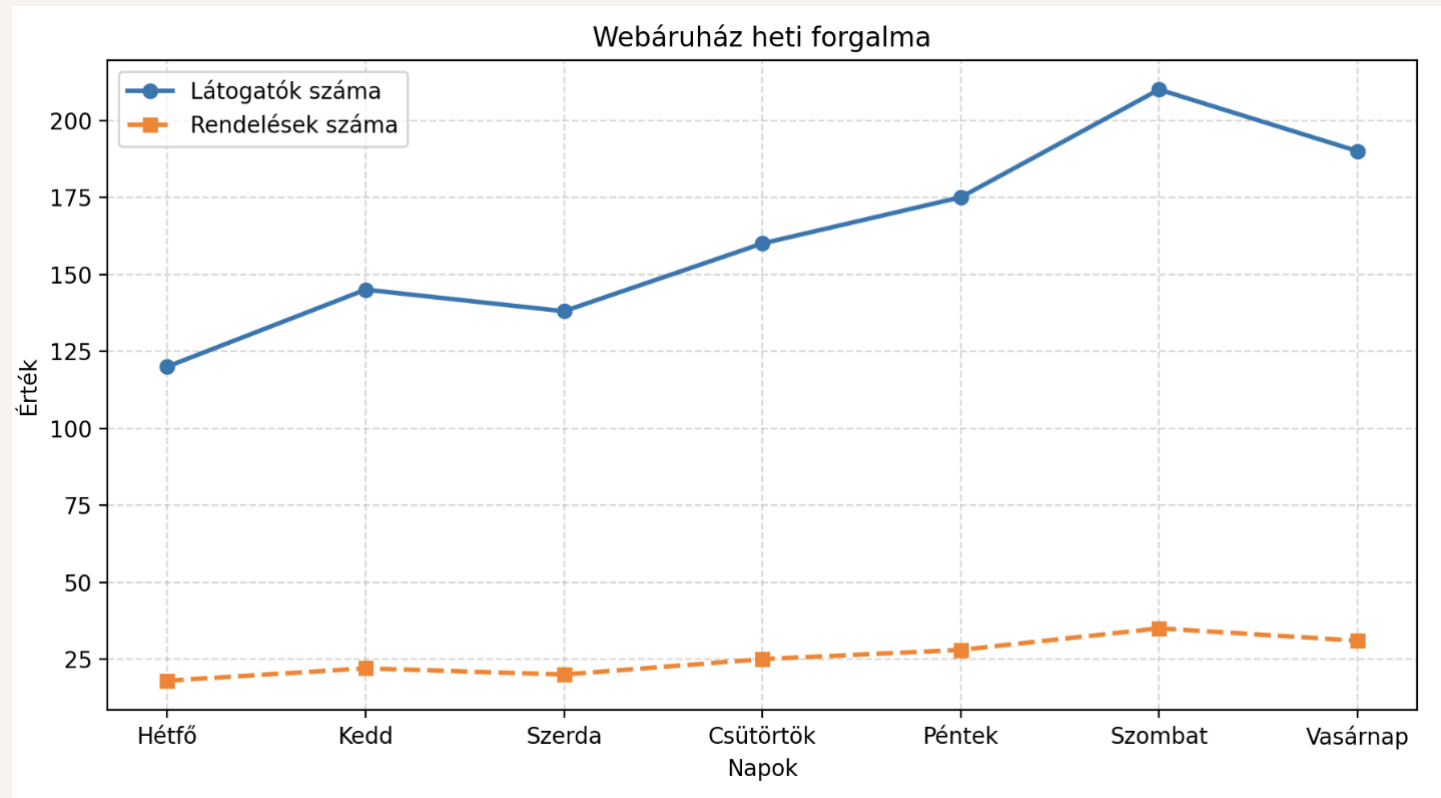
plt.plot(
    napok,
    rendelesek,
    marker='s',
    linestyle='--',
    linewidth=2,
    label='Rendelések száma'
)

plt.title('Webáruház heti forgalma')
plt.xlabel('Napok')
plt.ylabel('Érték')
plt.legend()
plt.grid(True)

plt.show()
```

Overlay diagram készítése

Az y tengelyre az Érték feliratot írjuk és a **label** címke, mint jelmagyarázat azonosítja az egyes grafikonokat.



Seaborn bevezetése és használata

A Seaborn könyvtár alkalmazásának előnyei:

- A **Seaborn** egy magasabb szintű Python-könyvtár, amely a Matplotlibre épül, és egyszerűbbé teszi az igényes, statisztikai jellegű diagramok elkészítését. Jól használható például hőterképek, eloszlásdiagramok, trendvonalak és más összetettebb grafikonok létrehozására.
- könnyen használható, beépített, esztétikus megjelenést biztosít, jól együttműködik a Pandas DataFrame objektumaival, számos előre elkészített diagramtípust tartalmaz, alkalmas adatok közötti kapcsolatok, eloszlások és trendek szemléltetésére.

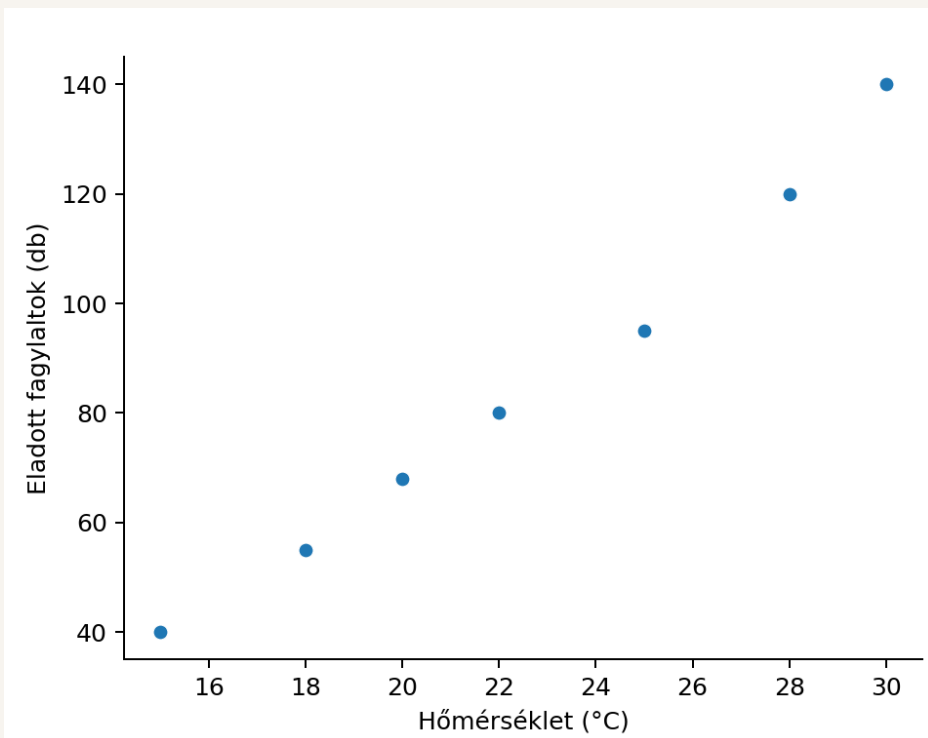
Telepítése:

```
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 létrehozása DataFrame-ben
adatok = pd.DataFrame({
    'Hőmérséklet': [15, 18, 20, 22, 25, 28, 30],
    'Fagylalt': [40, 55, 68, 80, 95, 120, 140]
})

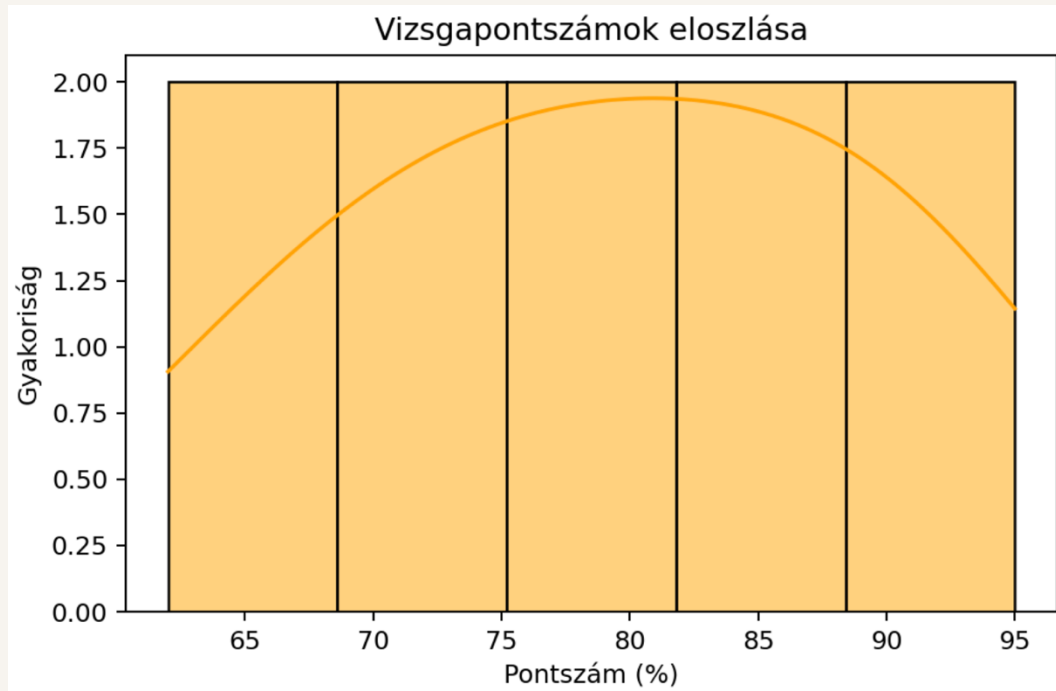
# Scatter diagram készítése
sns.relplot(
    data=adatok,
    x='Hőmérséklet',
    y='Fagylalt',
    kind='scatter'
)

plt.title('Hőmérséklet és fagylalteladás kapcsolata')
plt.xlabel('Hőmérséklet (°C)')
plt.ylabel('Eladott fagylaltok (db)')

plt.show()
```

Alapvető Seaborn plotok: histplot()

Eladások eloszlásának megjelenítése **histplot()**-tal:



A **hisztogram** mellett egy **Kernel Density Estimate (KDE)** görbét is megjelenít, amely az adatok **sűrűségét** mutatja.

```
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd

# Adatok
adatok = pd.DataFrame({
    'Pontszám': [62, 68, 71, 75, 78, 81, 85, 88, 91, 95]
})

# Hisztogram készítése
sns.histplot(
    adatok['Pontszám'],
    bins=5,
    kde=True,
    color='orange'
)

plt.title('Vizsgapontszámok eloszlása')
plt.xlabel('Pontszám (%)')
plt.ylabel('Gyakoriság')

plt.show()
```

Alapvető Seaborn plotok: boxplot()

[seabornboxplot.py \(.pdf\)](#)

A **boxplot()** segítségével az adatok szórását, mediánját és kiugró értékeit mutathatjuk meg.

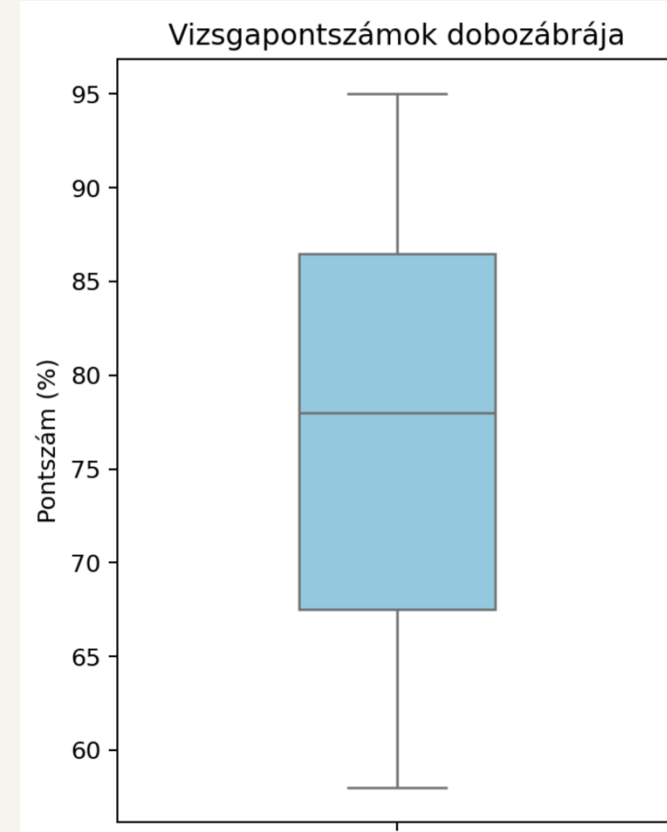
```
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd

# Mintaadatok
adatok = pd.DataFrame({
    'Pontszám': [58, 62, 65, 70, 74, 78, 82, 85, 88, 91, 95]
})

# Dobozábra készítése
sns.boxplot(
    y=adatok['Pontszám'],
    color='skyblue',
    width=0.35
)

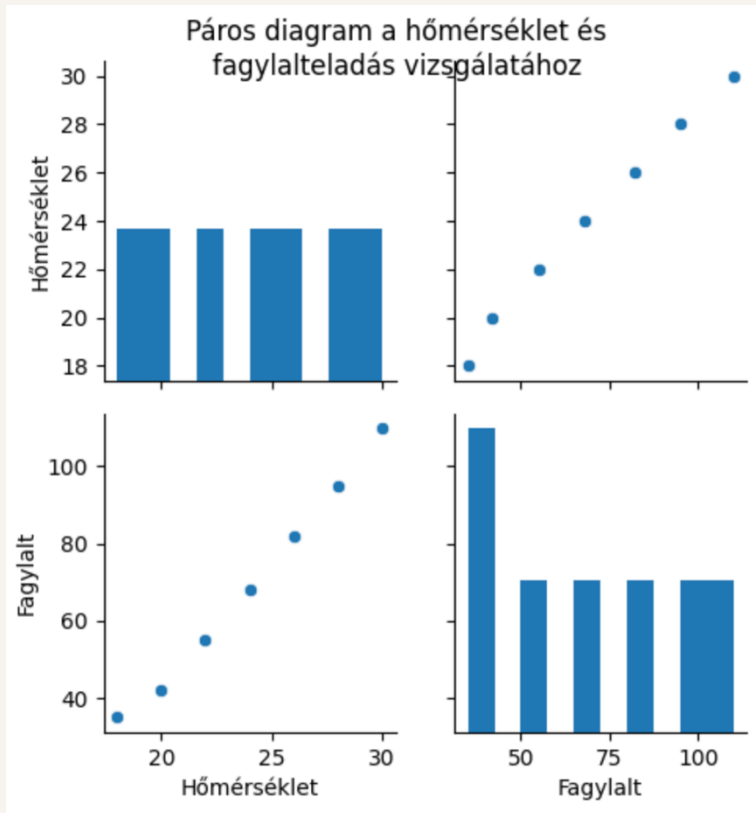
plt.title('Vizsgapontszámok dobozábrája')
plt.ylabel('Pontszám (%)')

plt.show()
```



Alapvető Seaborn plotok: pairplot()

A pairplot() több változó közötti kapcsolatok és eloszlások bemutatására szolgál, és automatikusan elkészíti a scatter plotokat és hisztogramokat.



```
import seaborn as sns
import pandas as pd
import matplotlib.pyplot as plt

# Adatok
adatok = pd.DataFrame({
    'Hőmérséklet': [18, 20, 22, 24, 26, 28, 30],
    'Fagyalt': [35, 42, 55, 68, 82, 95, 110]
})

# Páros diagram
g = sns.PairGrid(adatok)

g.map_upper(sns.scatterplot)
g.map_lower(sns.scatterplot)
g.map_diag(plt.hist)

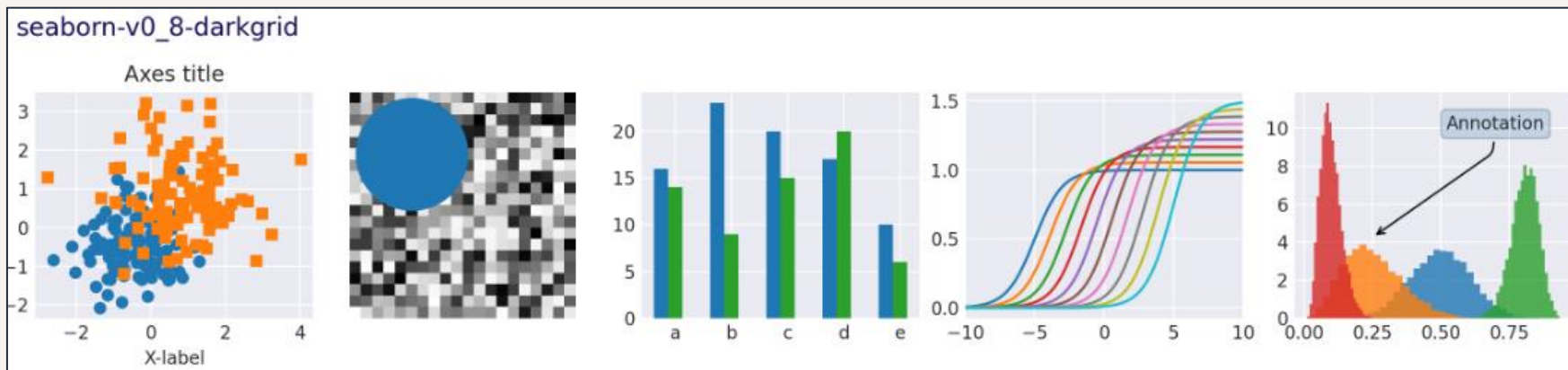
g.fig.suptitle(
    'Páros diagram a hőmérséklet és fagyalteladás vizsgálatához',
    y=1.02
)

plt.show()
```

Stílusok és esztétikai testreszabások

Beépített Seaborn stílusok változatai:

[Style sheets reference – Matplotlib 3.9.2 documentation](#)



Köszönöm a figyelmet!

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

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

Flask alapok, telepítés és a webservert 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 webserverver 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 szerveret
- ? 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

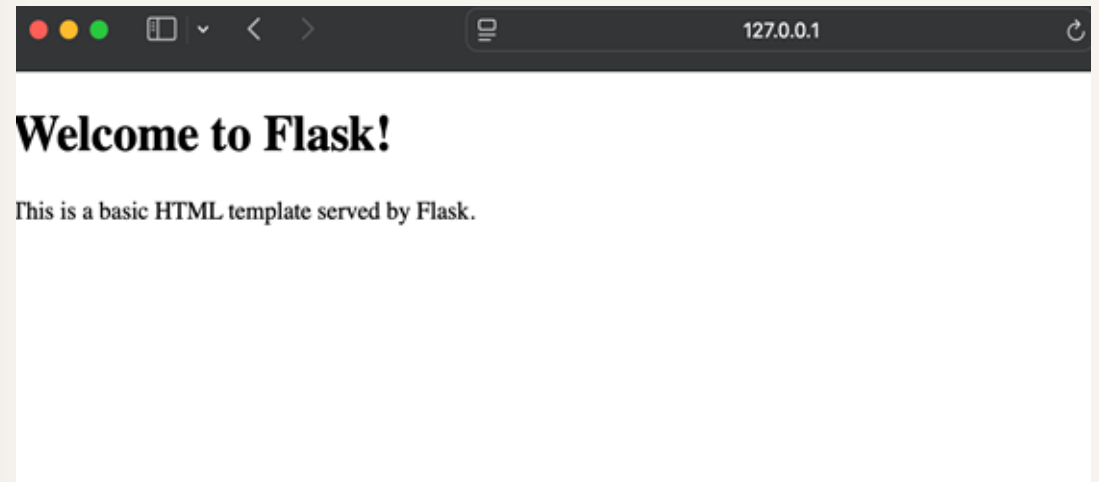
```
from flask import Flask, redirect, url_for,
render_template

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)

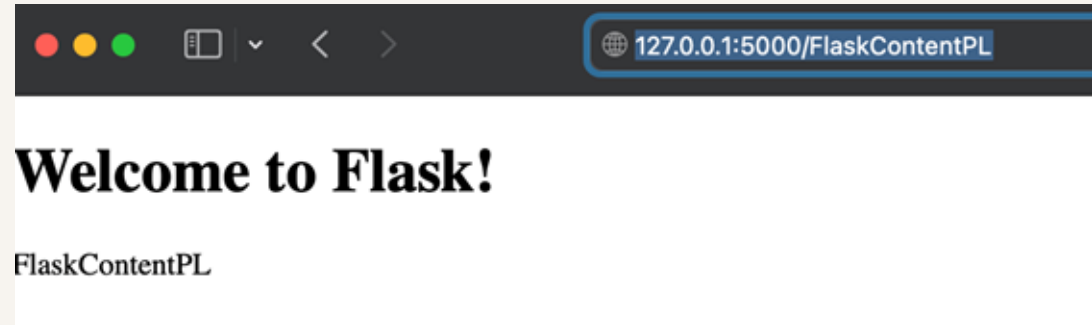
```
1 from flask import Flask, render_template
2
3 app = Flask(__name__)
4
5 @app.route('/<user>')
6 def home(user: str):
7     return render_template("index.html", bodycontent=user)
8
9 if __name__ == '__main__':
```

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 hivatkozunk a Flask tartalomra.
<https://127.0.0.1:5000/FlaskContentPL>
(simple2.html, indexea3.py)

{{user}} formában hivatkozunk a Flask tartalomra.
<https://127.0.0.1:5000/FlaskContentPL>
(simple2.html, indexea3.py)



Flask alapok, telepítés és a webszerver 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

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 i
    return render_template("index3.html", namelist=["Név1", "Név2", "Név3"])
```

Template kezelés

Template kezelés

(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>

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/BUtbMJw1MZ8t5HZApCHrRKUc4W0kG879m7"
crossorigin="anonymous">
```

Java script BSTRP a body tag végére kerül!

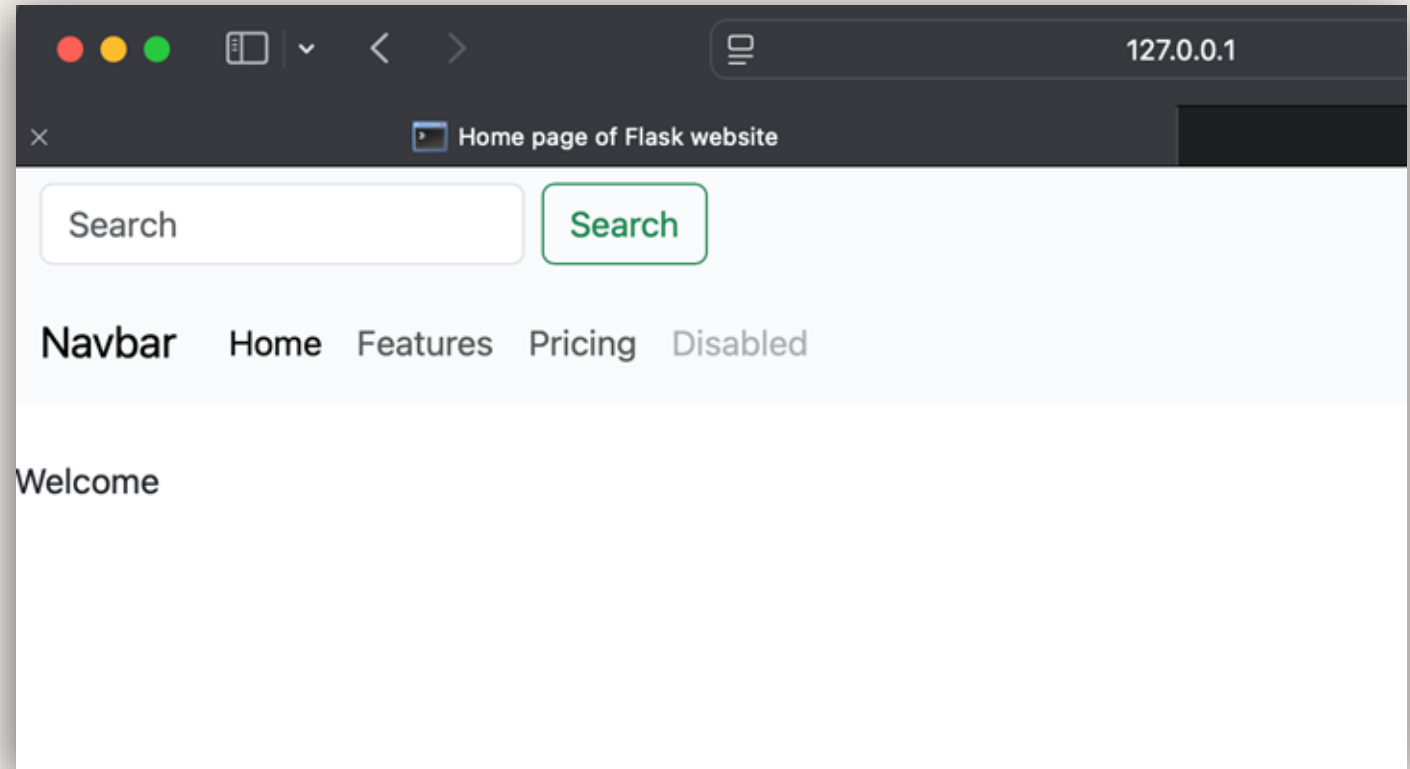
```
<script src="https://cdn.jsdelivr.net/npm/@popperjs/core@2.11.8/dist/umd/popper.min.js" integrity="sha384-I7E8VVD/ismYTF4hNIPjVp/Zjvgyol6VFvRkX/vR+Vc4jQkC+hVqc2pM8ODewa9r" crossorigin="anonymous"></script>  
<script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.5/dist/js/bootstrap.min.js" integrity="sha384-VQqxDN0EQCkWoxT/0vsQvZswzTHUVOImccYmSyhJTp7kGtPed0Qcx8rK9h9YEgx+" crossorigin="anonymous"></script>
```

NAV bar és Search Bar bootstrap-ból

NAVbar

<https://getbootstrap.com/docs/5.3/components/navbar/#nav>

(templatenav.html, index6.html,
indexea6.py)
[eacodes\templates\index6.html](#)



GET és POST

GET és POST

eacodes\templates\login.html

eacodes\Flask\indexlogin.pdf

eacodes\templates\templatena.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")

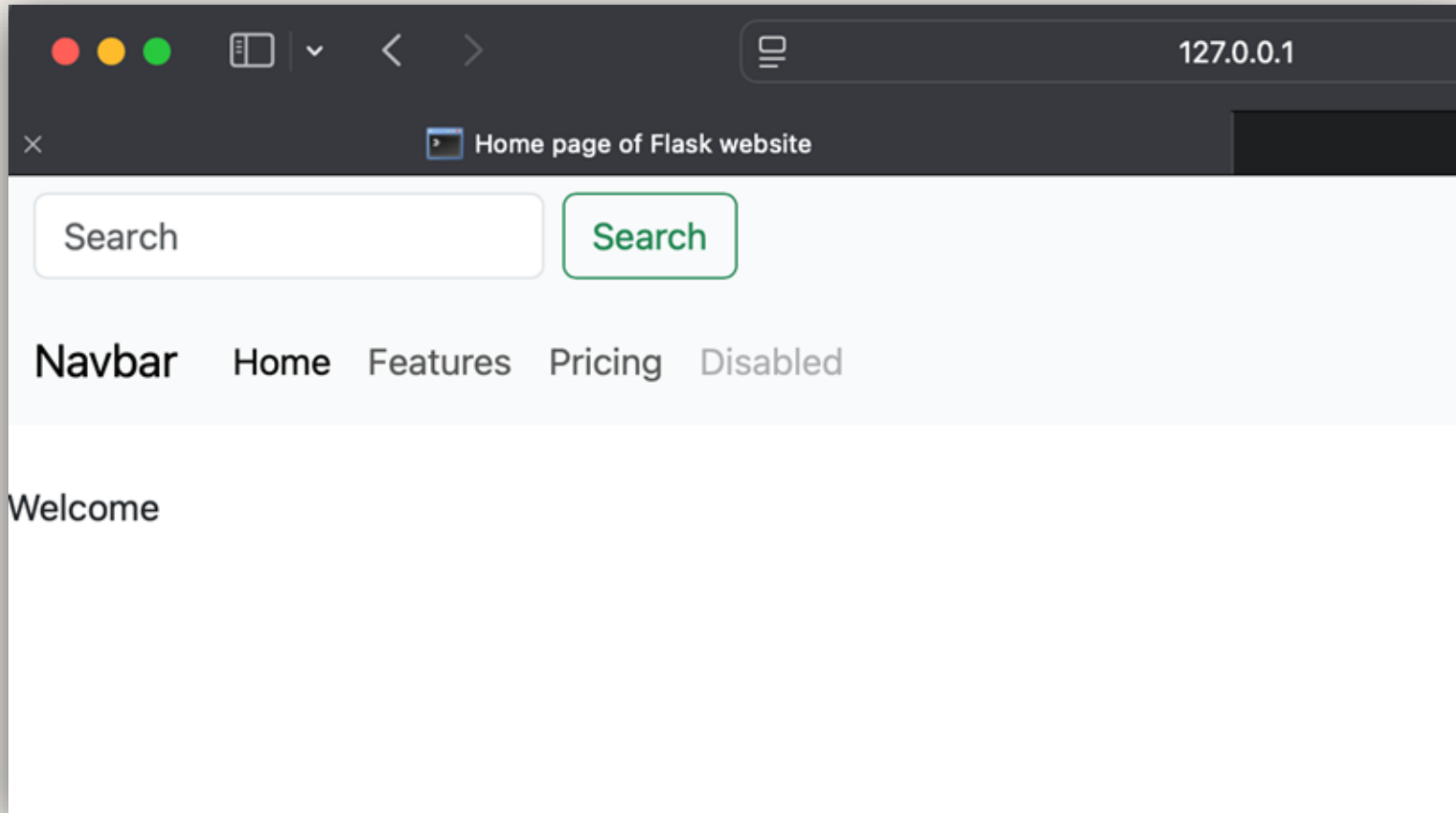
@app.route('/loginpage', methods=["POST", "GET"])
def loginpage():
    if request.method == "POST":
        username = request.form["username"]
        return redirect(url_for("user", usr=username))
    else:
        return render_template("login.html")

@app.route('/<usr>')
def user(usr):
    return f"<h1>{usr}</h1>"
```

```
{% extends "templatena.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

Másik oldal betöltése HTML menüből



Statefull vs stateless

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"))
```

Flash üzenetek

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 kiírása

```
@app.route("/profile")
def profile():

    if "email" in session:

        flash("Sikeres bejelentkezés!", "success")
        return render_template("profile.html")

    else:

        flash("Először jelentkezz be!", "warning")
        return redirect(url_for("login"))
```

```
<div class="container">

    {% with messages = get_flashed_messages() %}
        {% if messages %}
            <div class="alert alert-info">
                {% for message in messages %}
                    <p>{{ message }}</p>
                {% endfor %}
            </div>
        {% endif %}
    {% endwith %}

    {% block content %}
    {% endblock %}

</div>
```

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 kötőjelre: "user-logged-out!"

Static folder

Static folder

```
<html>
<head>
  <title>{% block title %}{% endblock %}</title>

  <link rel="stylesheet"
        href="{{ url_for('static', filename='css/main.css') }}">
</head>

<body>

<header>
  <h2>Webáruház</h2>
</header>

<hr>

{% block content %}
{% endblock %}

</body>
</html>
```

```
{% extends "base.html" %}

{% block title %}
Termékek
{% endblock %}

{% block content %}

<h1>Terméklista</h1>



<p>Ismerd meg legújabb termékeinket!</p>

<a href="{{ url_for('index') }}">
  Vissza a főoldalra
</a>

{% endblock %}
```

Köszönöm a figyelmet!

Python kurzus – 12. előadás

Tárgyfelelős: Dr. Tejfel Máté

Előadó: Dr. Király Roland

ELTE IK



Adatkezelés, fájlműveletek, kép és „mozgóképek” kezelés

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 \(tartalom letöltése\)](#)

Fájlok megnyitása, olvasása, írása (txt, csv, json)

Pythonban a fájlok megnyitása és kezelése a beépített `open()` függvénnyel különböző formátumú fájlokat olvashatunk és írhatunk

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.

TXT fájl megnyitása és olvasása

```
# TXT fájl megnyitása olvasásra  
  
with open("data.txt", "r") as file:  
  
    content = file.read()  
  
    print(content)
```

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
```

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.

Nagyobb fájlok esetén hasznos lehet a fájlt soronként olvasni, különösen akkor, ha egyszerre nem akarjuk a teljes tartalmat betölteni.

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()** függvénnyel sorokat írunk a fájlba. Ha a fájl már létezik, annak tartalma **felülíródik**.

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.

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**.

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.

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)
```


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.

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.

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()** függvénnyel az adatok szótárát 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.

Fájlműveletek hibakezelése

Hibakezelés Pythonban: Try-Except struktúra

Fájlok kezelése során különféle hibák fordulhatnak elő, például ha a megadott fájl nem található, vagy a program nem rendelkezik megfelelő jogosultsággal annak eléréséhez.

Az ilyen helyzetek megfelelő kezelése hozzájárul a program megbízható működéséhez, valamint segíti a felhasználót a hiba okának megértésében.

A Python nyelv a kivételkezelést a **try-except** szerkezettel valósítja meg. A **try** blokk tartalmazza a végrehajtandó műveleteket, míg az **except** blokk gondoskodik a felmerülő hibák kezeléséről.

Ennek köszönhetően a program nem áll le váratlanul, hanem szabályozott módon reagál a hibás helyzetekre, és szükség esetén megfelelő hibaüzenetet jelenít meg.

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) ))
```

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!")
```

FileNotFoundError

Ez a hiba akkor fordul elő, ha a megnyitni kívánt **fájl nem létezik** az elérési útvonalon.

PermissionError

Ez a hiba akkor fordulhat elő, ha a fájlhoz szükséges jogosultságok hiányoznak.

A program megpróbálja megnyitni az adatok.txt fájlt. Ha a fájl nem létezik, FileNotFoundError kivétel keletkezik. Az except ág elfogja a hibát, és a program leállás helyett egy érthető hibaüzenetet jelenít meg.

```
try:

    with open("adatok.txt", "r") as file:

        tartalom = file.read()

        print(tartalom)

except FileNotFoundError:

    print("Hiba: A megadott fájl nem található.")
```

IsADirectoryError és NotADirectoryError kezelése

Az IsADirectoryError akkor lép fel, ha egy könyvtárat próbálunk fájlként kezelni.

A program UTF-8 kódolással próbálja meg beolvasni a fájlt. Ha a fájl más karakterkódolású (például ANSI vagy Latin-1), UnicodeDecodeError kivétel keletkezhet.

```
try:
    with open("szoveg.txt", "r", encoding="utf-8") as file:
        tartalom = file.read()
        print(tartalom)
except UnicodeDecodeError:
    print("Hiba: A fájl karakterkódolása nem megfelelő.")
```


Általános kivételkezelés fájlműveleteknél

A program futása során olyan váratlan hibák is előfordulhatnak, amelyekre nem készítettünk külön kivételkezelőt. Ezek az **Exception** kivétel segítségével általánosan kezelhetők. A keletkezett hibaüzenet megjelenítésével könnyebben azonosítható a probléma oka, ami megkönnyíti a hibakeresést és a program javítását.

```
try:

    szam = int(input("Adj meg egy egész számot: "))

    eredmeny = 100 / szam

    print("Eredmény:", eredmeny)

except Exception as hiba:

    print("Váratlan hiba történt!")

    print("Hiba részlete:", hiba)
```

Akód minden kivételt megfog, és a hibát kiírja.

```
Adj meg egy egész számot: abc
Váratlan hiba történt!
Hiba részlete: invalid literal for int() with base 10: 'abc'
```

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 során alkalmazott hibakezelés hozzájárul a programok megbízható és biztonságos működéséhez. A célzott kivételkezelők lehetővé teszik, hogy a különböző hibák kezelése a program futásának megszakítása nélkül történjen. A **with** utasítás használata automatikusan gondoskodik a fájlok megfelelő lezárásáról a műveletek befejezése után, ezáltal csökkenti az erőforrás-szivárgás és az adatvesztés kockázatát.

A Python with kulcsszó használata

A with kulcsszó alapvető működése

- A **with** utasítás használata megkönnyíti az olyan erőforrások kezelését, amelyeket a művelet befejezése után megfelelően le kell zárni. Ilyen erőforrás lehet például egy fájl, egy adatbázis-kapcsolat vagy egy hálózati kapcsolat.
- A **with** egy kontextuskezelő, amely automatikusan gondoskodik az erőforrások felszabadításáról, függetlenül attól, hogy a művelet sikeresen befejeződik vagy hiba történik.
- Ennek köszönhetően nincs szükség a lezáró metódusok (például **close()**) manuális meghívására, így a kód áttekinthetőbbé, biztonságosabbá és kevésbé hibalehetővé válik.

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.

A with kulcsszó működése a háttérben

A with kulcsszó két metódust használ:

`__enter__()` - Felelős az erőforrás megnyitásáért.

`__exit__()` - Gondoskodik az erőforrás bezárásáról.

Ez segít, hogy a fájlok mindig lezáródjanak a with blokk végén és ne kelljen finally blokkot használni!.

JSON szöveg kezelése Pythonban

A **json.loads()** függvény JSON szöveget Python objektummá (dictionary) alakít.

```
{
    'title': 'Python Programming',
    'author': 'John Smith',
    'pages': 420,
    'available': True
}
```

```
import json

# JSON formátumú szöveg
json_data = '''
{
    "title": "Python Programming",
    "author": "John Smith",
    "pages": 420,
    "available": true
}
'''

# JSON szöveg átalakítása Python szótárrá
book = json.loads(json_data)

print(book)
```

Bináris fájlok

- A **bináris fájlok** az adatokat nem szöveges, hanem bináris (bájtsorozat) formában tárolják. Ide tartoznak például a képek, hangfelvételek, videók, PDF dokumentumok és más olyan fájlok, amelyek tartalma közvetlenül nem olvasható szöveggént.
- Pythonban a bináris fájlok kezelése általában **rb** (read binary) és **wb** (write binary) megnyitási módok használatával történik, amelyek lehetővé teszik a bájtok biztonságos beolvasását és kiírását.

```
# PDF fájl másolása bináris módban

with open("dokumentum.pdf", "rb") as source:
    data = source.read()

with open("dokumentum_masolat.pdf", "wb") as destination:
    destination.write(data)

print("A PDF fájl másolása sikeresen befejeződött.")
```

```
# ZIP fájl másolása bináris módban

with open("backup.zip", "rb") as source:
    binary_data = source.read()

with open("backup_copy.zip", "wb") as destination:
    destination.write(binary_data)

print("A ZIP fájl másolata elkészült.")
```


XML fájlok

Az **XML (Extensible Markup Language)** egy széles körben alkalmazott fájlformátum, amely strukturált adatok tárolására és cseréjére szolgál. Hierarchikus felépítésének köszönhetően jól használható összetett adatszerkezetek ábrázolására, ezért gyakran találkozhatunk vele webszolgáltatásokban, konfigurációs állományokban és különböző adatcsere-folyamatokban.

Pythonban az XML-fájlok feldolgozását az **xml.etree.ElementTree** modul teszi egyszerűvé, amely támogatja az adatok beolvasását, módosítását és új XML-fájlok létrehozását.

```
<library>
  <book>
    <title>Python alapjai</title>
    <author>Kiss Péter</author>
  </book>
  <book>
    <title>Adatkezelés Pythonban</title>
    <author>Nagy Anna</author>
  </book>
</library>
```

```
Cím: Python alapjai
Szerző: Kiss Péter
-----
Cím: Adatkezelés Pythonban
Szerző: Nagy Anna
-----
```

```
import xml.etree.ElementTree as ET

# XML fájl beolvasása
tree = ET.parse("books.xml")
root = tree.getroot()

# Könyvek adatainak kiírása
for book in root.findall("book"):
    title = book.find("title").text
    author = book.find("author").text

    print("Cím:", title)
    print("Szerző:", author)
    print("-" * 25)
```

Microsoft Excel fájlok (XLSX)

Excel fájl beolvasása a pandas segítségével

```
import pandas as pd

data = pd.read_excel("data.xlsx")

print(data)
```

A **Microsoft Excel (.xlsx)** fájlok kezelésére Pythonban leggyakrabban a **pandas** és az **openpyxl** könyvtárakat használjuk. Ezek segítségével egyszerűen beolvashatók, módosíthatók és menthetők az Excel-munkafüzetek, így hatékony eszközök táblázatos adatok feldolgozásához és elemzéséhez.

A **pandas** könyvtár és a **pd.read_excel()** függvény segítségével beolvashatjuk az Excel fájlokat.

PDF fájlok

A **PdfReader()** metódus segítségével a dokumentum.pdf első oldalának szövegét olvashatjuk ki.

```
import PyPDF2

# PDF fájl megnyitása
with open("konyv.pdf", "rb") as pdf_file:

    reader = PyPDF2.PdfReader(pdf_file)

    # Oldalak számának meghatározása
    page_count = len(reader.pages)

    print("A dokumentum oldalszáma:", page_count)
```

A **PDF dokumentumok** kezelésére Pythonban több könyvtár is rendelkezésre áll, amelyek közül a leggyakrabban használtak a **PyPDF2** és a **pdfplumber**. Ezek segítségével a PDF-fájlok tartalma beolvasható, a bennük található szöveg kinyerhető, valamint bizonyos esetekben az oldalak kezelése és a dokumentumok módosítása is megvalósítható.

```
import PyPDF2

# PDF fájl megnyitása
with open("jegyzet.pdf", "rb") as pdf_file:

    reader = PyPDF2.PdfReader(pdf_file)

    # Utolsó oldal kiválasztása
    last_page = reader.pages[-1]

    print(last_page.extract_text())
```

Képfájlok (JPEG, PNG stb.)

Képfájl beolvasása és méretének módosítása

```
import PyPDF2

# PDF fájl megnyitása
with open("beszamolo.pdf", "rb") as pdf_file:

    reader = PyPDF2.PdfReader(pdf_file)

    # Második oldal kiválasztása
    second_page = reader.pages[1]

    print(second_page.extract_text())
```

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.

```
import PyPDF2

# PDF fájl megnyitása
with open("jegyzet.pdf", "rb") as pdf_file:

    reader = PyPDF2.PdfReader(pdf_file)

    for i in range(2):
        print(f"{i + 1}. oldal:")
        print(reader.pages[i].extract_text())
        print("-" * 30)
```

Hangfájlok (MP3, WAV)

Példa: WAV fájl beolvasása és hossza meghatározása

```
import wave

# WAV fájl megnyitása
with wave.open("zene.wav", "rb") as audio:

    print("Minták száma:", audio.getnframes())
```

```
import wave

# WAV fájl megnyitása
with wave.open("felvetel.wav", "rb") as audio:

    print("Minta mérete (bájt):", audio.getsampwidth())
```

Hangfájlok (pl. MP3, WAV) kezelésére Pythonban az **pydub** és **wave** modulok használhatók. Ezek lehetővé teszik a hangfájlok betöltését, átalakítását és lejátszását.

```
import wave

# WAV fájl megnyitása
with wave.open("hang.wav", "rb") as audio:

    print("Paraméterek:")
    print(audio.getparams())
```

```
import wave

# WAV fájl megnyitása
with wave.open("audio.wav", "rb") as audio:

    if audio.getnchannels() == 1:
        print("Mono hangfájl")
    else:
        print("Sztereoó hangfájl")
```

ZIP és egyéb tömörített fájlok

ZIP fájl tartalmának kicsomagolása

```
import zipfile

# ZIP fájl tartalmának megjelenítése
with zipfile.ZipFile("projekt.zip", "r") as archive:

    print("A ZIP fájl tartalma:")

    for file in archive.namelist():
        print(file)
```

```
import zipfile

# Fájl tömörítése ZIP archívumba
with zipfile.ZipFile("mentes.zip", "w") as archive:

    archive.write("jegyzet.txt")

print("A ZIP fájl elkészült.")
```

A zipfile és tarfile modulokkal ZIP és TAR tömörített fájlokat kezelhetünk Pythonban.

```
import zipfile

# Szöveges fájl beolvasása ZIP-ből
with zipfile.ZipFile("adatok.zip", "r") as archive:

    with archive.open("info.txt") as file:
        print(file.read().decode("utf-8"))
```

```
import zipfile

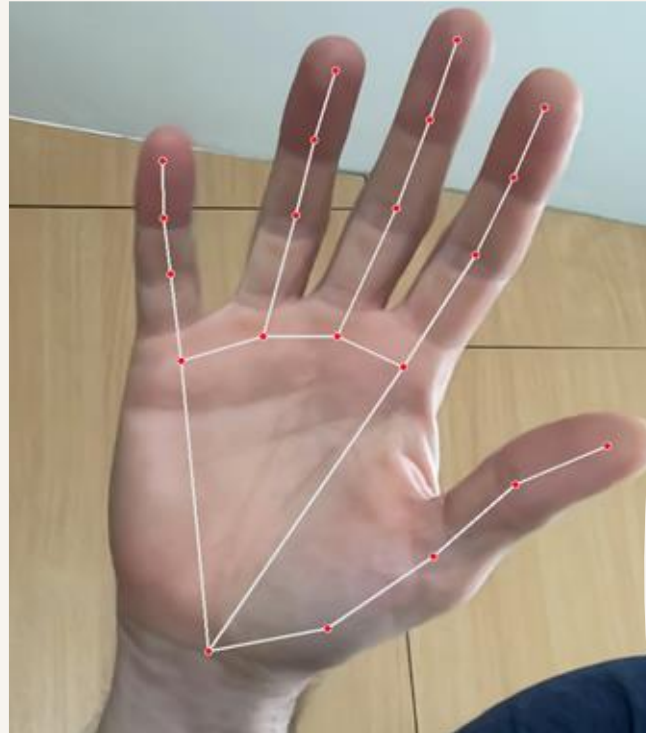
# Új fájl hozzáadása egy meglévő ZIP fájlhoz
with zipfile.ZipFile("projekt.zip", "a") as archive:

    archive.write("eredmeny.csv")

print("A fájl hozzáadása sikeres.")
```

Képek feldolgozása

- opencv
- cv2.dnn.readNetFromCaffe(m1, m2)
- Object detection, automtikus azonosítás
- Hand detection



Köszönöm a figyelmet!

