



# Szkriptyelvek

Szathmáry László  
Debreceni Egyetem  
Informatikai Kar

## 7. Gyakorlat

- osztályok, objektumok

(utolsó módosítás: 2025. aug. 14.)

2025-2026, 1. félév



# OO programozás Pythonban

Pythonban lehet procedurális, illetve OO módon is programozni.

Választhatunk, hogy melyiket használjuk: vagy az egyiket, vagy a másikat, vagy akár mindkettőt.

Python osztályokat már használtunk, pl.: `str` (sztring osztály).

```
name = "john"  
print(name.capitalize())
```

Most megnézzük, hogy hogyan tudunk saját osztályokat definiálni, illetve hogyan tudunk ezután objektumokat példányosítani.

# OO programozás Pythonban (folyt.)

A Python programozási nyelvben az összes standard OO tulajdonság megtalálható.

Van benne például:

- többszörös öröklődés
- a leszármazott osztály felülírhatja a szülőosztály bármely metódusát

Dinamikus nyelvről lévén szó, az osztályok futásidőben jönnek létre, s létrehozás után tovább módosíthatók!

Minden példányváltozó és példánymetódus *publikus*.

Minden példánymetódus *virtuális*.

A legtöbb beépített operátor túlterhelhető s használható az osztály objektumaira.

Az objektumok átadása paraméterként olcsó, ui. az objektumok címe lesz átadva (referencia). Vagyis ha egy paraméterként átadott objektumot módosítunk, akkor a hívó fél is látni fogja a változásokat.

# osztályok

```

3 class EmptyClass:
4     pass
5
6
7 class MyClass:
8     def hello(self):
9         return "hello world"
10
11
12 def main():
13     obj = MyClass()
14     print(obj.hello())

```

OsztályNeve

minden osztály az „**object**” osztály leszármazottja (Python 3-ban ezt már nem kell kiírni)

**példánymetódus**

az első paraméter kötelezően a „self”, de ezt a hívás helyén nem írjuk ki

példányosítás

Python 3: az „object” ősosztályt nem muszáj feltüntetni, ui. ez az alapértelmezés.  
Ki lehet írni, de nem muszáj.

Python 2: az „object” ősosztályt fel KELL tüntetni, különben egy régi stílusú osztály jön létre (old-style class).

# osztályok (példányváltozó, példánymetódus)

```
12 class Hello:  
13     """  
14     A class for greeting the user.  
15     """  
16     def create_name(self, name):  
17         self.name = name  
18  
19     def display_name(self):  
20         return(self.name)  
21  
22     def greet(self):  
23         print("Hello {0}!".format(self.name))  
24  
25  
26     def main():  
27         h = Hello()  
28         h.create_name('Alice')  
29         print(h.display_name())  
30         h.greet()  
  
print(h.name)
```

docstring

példánymetódus

példányváltozó

minden publikus

Alice  
Hello Alice!

# self



Minden metódus első paramétere ez kell hogy legyen.

Ez a Java „this” változójának felel meg, vagyis ez egy olyan referencia, mely az adott objektumra mutat. Megegyezés alapján „self” a neve, ezen ne változtassunk!

Minden (nem-statikus) függvény első paramétere a „self”, viszont a függvény meghívásakor ezt nem kell kiírni.

A Python dinamikus természetéből adódóan bármelyik függvényben létrehozhatunk egy példányváltozót, s onnantól kezdve az létezik.

## osztályok (init)

```
3 class Greetings:  
4     ...  
5     def __init__(self, name):  
6         self.name = name  
7     ...  
8     def say_hi(self):  
9         print("Hi {0}!".format(self.name))  
10    ...  
11  
12    def main():  
13        g = Greetings("Alice")  
14        g.say_hi()
```



A konstruktor automatikusan meghívja az `__init__()` metódust.  
Vagyis az `__init__()` nem a konstruktor, de nagyon közel áll hozzá.  
Ez fogja inicializálni az objektumot.

# osztályok (példánymetódus meghívása)

```

3  class Bag:
4
5      def __init__(self):
6          self.data = []
7
8      def add(self, value):
9          self.data.append(value)
10
11     def add_twice(self, value):
12         self.add(value)
13         self.add(value)
14
15     def __str__(self):
16         return str(self.data)
17
18
19 def main():
20     b = Bag()
21     b.add(5)
22     print(b)
23     b.add(3)
24     print(b)
25     b.add_twice(9)
26     print(b)

```

konténer osztály  
(a példányai adatokat / objektumokat tárolnak)



speciális metódus  
(az adott objektumot olvasható  
formában jeleníti meg)

lásd: Java `toString()`

*Próbáljuk ki enélkül is!*

# osztályok (rekord)

Néha jól jönne a C nyelv struct-jához hasonló **rekord** típus. Megoldható:

```
3 class Employee:  
4     pass  
5  
6 def main():  
7     john = Employee()  
8     john.name = "John Doe"  
9     john.dept = "IT"  
10    john.salary = 1000  
11  
12    print(john.dept)
```

Másik módszer: szótár használata

```
john = {}  
john['name'] = "John Doe"  
...  
...
```

## privát változók és metódusok

Privát változók/metódusok, melyek nem érhetők el kívülről csak az objektumon belülről: *nincs ilyen* Pythonban. Minden publikus.

Viszont van egy megegyezés: ha egy változó/metódus neve \_ (aláhúzás) jellel kezdődik, akkor azt nem-publikusként kell kezelni. Pl.: \_spam.

---

## accessors (getters / setters)

Nincs rá szükség, ui. minden publikus.

Egyszer megkérdezték Guidot, hogy miért nincsenek privát változók/metódusok.  
A válasza: „We are all adults.” :)

# accessors (getters / setters)

Java stílus

```
class Rectangle:
    def __init__(self, width, height):
        self._width = width
        self._height = height

    def get_width(self):
        return self._width

    def set_width(self, new_width):
        self._width = new_width

    def get_height(self):
        return self._height

    def set_height(self, new_height):
        self._height = new_height

    def area(self):
        return self._width * self._height

def main():
    rect = Rectangle(50, 10)
    rect.set_width(60)
    print(rect.area())
```

Python stílus

```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

def main():
    rect = Rectangle(50, 10)
    rect.width = 60
    print(rect.area())
```

**Feladat:** forrás kibővítése

```
print(rect) # produkálja a következőt:  
-> "Rectangle(60, 10)"
```

# speciális metódusok

Ezeknek a neve   -sal (dupla aláhúzás) kezdődik és ugyanígy végződik.

Már láttunk néhány ilyet:

- \_\_init\_\_
- \_\_str\_\_

Vannak további speciális metódusok is, lásd

<https://rszalski.github.io/magicmethods/> .

A speciális metódusokkal lehet megvalósítani az operátorok túlterhelését is (lásd [példa](#)).

---

## destruktor

Nincs, a garbage collector fogja majd megsemmisíteni az objektumot.

Ennek a pontos idejét viszont nem tudjuk befolyásolni.

# osztályváltozók

osztályváltozó  
(az osztály metódusain  
kívül lett definiálva)

```
3 class Proba:  
4     i = 12345 ←  
5  
6     def hello(self):  
7         print("hello")  
8  
9  
10    def main():  
11        print(Proba.i) ← hivatkozás  
12  
13    p = Proba()  
14    p.hello()  
15    print(p.i) ←
```

## Feladat:

Írunk egy olyan osztályt, amely számolja,  
hogyan szor példányosítottuk.

# osztálymetódusok (1. módszer)

Írunk egy Balloon osztályt, mely egy színes labdát reprezentál. Tartsuk számon azt is, hogy *hány különböző színű labdánk van.* (Pl. ha van 2 piros, 1 fehér és 5 zöld labdánk, akkor három különböző színű labdánk van.)

```

3  class Balloon:
4      unique_colors = set() ← osztályváltozó
5
6      def __init__(self, color):
7          self.color = color
8          Balloon.unique_colors.add(color)
9
10     @staticmethod ← dekorátor
11     def unique_color_count(): ← osztálymetódus
12         return len(Balloon.unique_colors)
13
14
15     def main():
16         a = Balloon("red")
17         b = Balloon("green")
18         c = Balloon("green")
19         d = Balloon("white")
20         print(Balloon.unique_color_count()) # 3
  
```

Vegyük észre, hogy a függvénynek **NINCS** extra paramétere!

Ez a statikus függvény tulajdonképpen az osztályon kívül is lehetne.  
 Azért tettük az osztályba, mert logikailag oda tartozik.

## osztálymetódusok (2. módszer)

```

3  class Balloon:
4      unique_colors = set() ← osztályváltozó
5
6      def __init__(self, color):
7          self.color = color
8          Balloon.unique_colors.add(color)
9
10     @classmethod ← dekorátor
11     def unique_color_count(cls): ← osztálymetódus
12         return len(Balloon.unique_colors)

```

A „cls” paraméter magát az osztályt jelenti.  
 Híváskor ezt sem kell kiírni.

Vegyük észre, hogy a függvénynek **VAN** extra paramétere (`cls`)!

Akkor használjuk ezt a módszert, amikor a függvényben hivatkozni akarunk az aktuális osztályra (pl. öröklődés esetén).

# öröklődés

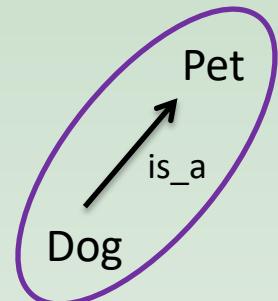
```

3  class Pet:
4      def __init__(self, name, species):
5          self.name = name
6          self.species = species
7
8      def __str__(self):
9          return f"{self.name} is a {self.species}"
10
11 class Dog(Pet):
12     def __init__(self, name, hates_cats=True):
13         super().__init__(name, "dog") ←
14         self.hates_cats = hates_cats
15
16     def __str__(self):
17         original = super().__str__() ←
18         extra = " (hates cats)" if self.hates_cats else ""
19         return original + extra
20
21 def main():
22     donci = Pet("Donci", "cat")
23     print(donci)
24
25     dugo = Pet("Dugo", "dog")
26     print(dugo)
27
28     frakk = Dog("Frakk")
29     print(frakk)
30
31     csibesz = Dog("Csibesz", hates_cats=False)
32     print(csibesz)

```

szülő osztály

leszármazott  
osztály



```

Donci is a cat
Dugo is a dog
Frakk is a dog (hates cats)
Csibesz is a dog

```

## öröklődés (folyt.)

```
class DerivedClass(BaseClass):  
    ...
```

A leszármazott osztályok felülírhatják a szülők metódusait. Pythonban az összes függvény virtuális.

A felülírt metódusban lehet, hogy a szülő osztályban lévő metódus által visszaadott értéket fel szeretnénk használni. Ekkor a szülő azonos nevű metódusát a következőképpen tudjuk meghívni:

```
super().method_name(arguments)
```

Hasznos beépített függvény:

```
print(isinstance(frakk, Dog)) → True  
print(isinstance(frakk, Pet)) → True
```

# többszörös öröklődés



Erre is van lehetőség, de inkább ne használjuk. A Java-ból sem véletlenül vették ki...

# Enum

Felsorolásos típus.

```
3 from enum import Enum
4
5 class Direction(Enum):
6     UP = 1
7     RIGHT = 2
8     DOWN = 3
9     LEFT = 4
10
11 def main():
12     print(Direction.UP)           # Direction.UP
13     print(type(Direction.UP))    # <enum 'Direction'>
14     print(Direction.UP.name)     # "UP" (str)
15     print(Direction.UP.value)    # 1 (int)
```

osztályváltozók

# Enum (folyt.)

Felsorolásos típus.

```
3 from enum import Enum, auto
4
5 class Direction(Enum):
6     UP = auto()
7     RIGHT = auto()
8     DOWN = auto()
9     LEFT = auto()
10
11 def main():
12     print(Direction.UP)          # Direction.UP
13     print(type(Direction.UP))    # <enum 'Direction'>
14     print(Direction.UP.name)     # "UP" (str)
15     print(Direction.UP.value)    # 1 (int)
```

**Feladat:** hangrendes feladat megoldása  
*enum* használatával ([link](#)).



házi feladat

# Feladatok



1. [20130325a] osztályok (sor két veremmel)
2. [20170511c] operátorok túlterhelése (átnézni, kipróbálni)