5. Előadás

Python kurzus

Tárgyfelelős:
Dr. Tejfel Máté

Előadó:
Dr. Király Roland

5. Előadás tematikája

Objektumorientált programozás I.

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

1. Bevezetés az Objektumorientált programozásba

- Mi az objektumorientált programozás?
- Előnyök: könnyebb karbantarthatóság, moduláris felépítés, kód újrahasznosítása.
- Az OOP négy alapelve:

  1. Egységbezárás (Encapsulation): Az adatok és a hozzájuk tartozó műveletek osztályon belüli védelme.
  2. Absztrakció (Abstraction): Csak a lényeges információk elérése és a bonyolult részletek elrejtése.
  3. Öröklődés (Inheritance): Az osztályok tulajdonságainak és metódusainak újrafelhasználása és kibővítése.
  4. Polimorfizmus (Polymorphism): Ugyanazon nevű metódus különböző viselkedése különböző osztályok objektumai esetében.

1. Bevezetés az Objektumorientált Programozásba

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

Az OOP előnyei:

1. **Modularitás:** Az osztályokba és objektumokba szervezett kód könnyebben kezelhető és karbantartható. Az egyes modulok (osztályok) egymástól függetlenek, így könnyen változtathatók anélkül, hogy az egész programot át kellene írni.

2. **Újrahasznosíthatóság:** Az osztályok és objektumok többször felhasználhatók különböző programokban vagy a program különböző részein.

3. **Karbantarthatóság:** Az objektumok és osztályok hierarchiájának és szerkezetének köszönhetően a programok könnyebben bővíthetők, javíthatók.

4. **Rugalmas tervezés:** Az öröklődés és a polimorfizmus lehetőséget ad arra, hogy az alapvető programfunkciókat újra lehessen definiálni anélkül, hogy az alapvető logikát át kellene írni.

A class működése – type

1. Egységbezárás (Encapsulation)

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

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

```
class BankSzámla:
    def __init__(self, egyenleg):
        self.__ egyenleg = egyenleg        # privát attribútum
    def betét(self, mennyi):
        self.__ egyenleg += mennyi
    def kivét(self, mennyi):
        if mennyi <= self.__ egyenleg :
            self.__ egyenleg -= mennyi
        else:
            print(„Nincs fedezet")
```

Az **__egyenleg** attribútum el van rejtve, így közvetlenül nem férhetünk hozzá kívülről, ehelyett a **betét()** és **kivét()** metódusokon keresztül kezeljük.

2. Az absztrakció (Abstraction)

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

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

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

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

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

3. Az öröklődés (Inheritance)

Az **öröklődés** mechanizmusa:

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

Példa: Ha van egy **Jármű** osztályunk, abból örökölhetünk egy **Autó** és egy **Bicikli** osztályt, mindkettő ugyanazokat az alapvető jellemzőket (pl. sebesség) örökölheti, de egyedi viselkedést is kaphatnak.

```
class Jármű:
   def __init__(self, sebesség):
      self. sebesség = sebesség
   def halad(self):
      return f"Halad {self.sebesség} sebességgel "
class Autó(Jármű):
   def halad(self):
      return f"Az autó halad {self.sebesség} sebességgel "
```

4. A polimorfizmus (Polymorphism)

A **polimorfizmus:**

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

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

```
class Állat:
   def beszél(self):
      pass
class Kutya(Állat):
   def beszél(self):
      return „Vau-vau!"
class Macska(Állat):
   def beszél(self):
      return "Miaú!"
```

2. Osztályok és Objektumok

- Fogalmak tisztázása: Mi az osztály és mi az objektum?
- Példák:
  - Osztály definíciója Pythonban: **class** kulcsszó használata.
  - Objektum **(példány)** létrehozása egy osztályból.
- Kód példa:

```
class Kutya:                          # osztály definíció
   def __init__(self, név):
       self.név = név

my_dog = Kutya("Buksi")               # példány definíció
print(my_dog.név)
```

2. 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) **mintá**ja.
- Az **objektum** egy konkrét **példány**, amely az osztály alapján jön létre.

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

```
class Autó:
  def __init__(self, márka, típus):
    self.márka = márka
    self.típus = típus
  def indít(self):
    print(f"A(z) {self.márka} {self.típus} elindul.")
```

Az osztályban definiáltunk két attribútumot, a **márka** és **típus** változókat, amelyek az autó tulajdonságait tárolják.
Az **indít** metódus az autó elindítását szimulálja.

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.

```
autó1 = Autó("Toyota", "Corolla")
autó2 = Autó("Ford", "Focus")

autó1.indít()        # Eredmény: A(z) Toyota Corolla elindul.
autó2.indít()        # Eredmény: A(z) Ford Focus elindul.
```

Az **autó1** és **autó2** objektumok az Autó osztály példányai, mindkettő saját értékekkel rendelkezik a márka és típus attribútumokban.

Az osztály definíció szintaxisa:

```
class Ember:
   def __init__(self, név, kor):              # konstruktor metódus
      self.név = név          # attribútum
      self.kor = kor          # attribútum

   def köszönés(self):                        # metódus
      return f"Szia, {self.név} vagyok és {self.kor} éves."
```

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

Objektum létrehozása:

```
ember1 = Ember("Anna", 25)
ember2 = Ember("Béla", 30)

print(ember1.köszönés())        # Eredmény: Szia, Anna vagyok és 25 éves.
print(ember2.köszönés())        # Eredmény: Szia, Béla vagyok és 30 éves.
```

Az osztály **példányosításával** létrehozunk egy konkrét objektumot, amely saját attribútumokkal és metódusokkal rendelkezhet.

Az **ember1** és **ember2** objektumok az Ember osztály példányai, és mindegyik objektum saját névvel és korral rendelkezik.

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

```
class Könyv:
   def __init__(self, cím, szerző):
      self.cím = cím
      self.szerző = szerző
   def könyv_információ(self):
      return f"Cím: {self.cím}, Szerző: {self.szerző}"

könyv1 = Könyv("Egri csillagok", "Gárdonyi Géza")
könyv2 = Könyv("A Pál utcai fiúk", "Molnár Ferenc")
print(könyv1.könyv_információ())
print(könyv2.könyv_információ())
```

```
Cím: Egri csillagok, Szerző: Gárdonyi Géza
Cím: A Pál utcai fiúk, Szerző: Molnár Ferenc
```

Összefoglalva az osztályok és objektumok előnyeit:

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

3. Konstruktorok és destruktorok

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

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

    def __del__(self):
        print(f"{self.név} megszűnik")
```

Konstruktorok – Az objektumok inicializálása

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

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

```
class Diák:
    def __init__(self, név, kor):
        self.név = név              # A diák neve
        self.kor = kor              # A diák kora

    def bemutatkozás(self):
        return f"Szia, {self.név} vagyok és {self.kor} éves."
```

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

Objektum létrehozása:

```
diák1 = Diák("Ádám", 20)
diák2 = Diák("Éva", 19)

print(diák1.bemutatkozás())        # Eredmény: Szia, Ádám vagyok és 20 éves.
print(diák2.bemutatkozás())        # Eredmény: Szia, Éva vagyok és 19 éves.
```

A konstruktor jellemzői:

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

Destruktorok – Az objektumok megszüntetése

A **destruktor** metódus egy objektum megszüntetését és a memória felszabadítását eredményezi. A destruktor neve **__del__**.
A destruktor akkor hasznos, ha valamilyen erőforrást (fájlokat, hálózati kapcsolatokat) kell felszabadítani vagy bezárni, amikor egy objektum már nincs használatban.

Példa destruktorra: egy **Fájl** osztály fájlkezelésre

```
class Fájl:
    def __init__(self, fájlnév):
        self.fájlnév = fájlnév
        self.fájl = open(fájlnév, 'w')
        print(f"A(z) {self.fájlnév} fájl megnyitva írásra.")
    def írás(self, szöveg):
        self.fájl.write(szöveg)
    def __del__(self):
        self.fájl.close()
        print(f"A(z) {self.fájlnév} 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:

```
fájl = Fájl("példa.txt")
fájl.írás("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 - **BankSzámla** osztály:

```
class BankSzámla:
  def __init__(self, tulajdonos, kezdő_egyenleg):
    self.tulajdonos = tulajdonos
    self.egyenleg = kezdő_egyenleg
    print(f"{self.tulajdonos} számlája megnyitva {self.egyenleg} egyenleggel.")
  def befizet(self, összeg):
    self.egyenleg += összeg
    print(f"{összeg} befizetve. Új egyenleg: {self.egyenleg}.")
  def kivesz(self, összeg):
    if összeg > self.egyenleg:
      print("Nincs elég fedezet.")
    else:
      self.egyenleg -= összeg
      print(f"{összeg} kivéve. Új egyenleg: {self.egyenleg}.")
  def __del__(self):
    print(f"{self.tulajdonos} számlája lezárva.")

számla = BankSzámla("Anna", 100000) #Eredmény: Anna számlája megnyitva 100000 egyenleggel.
számla.befizet(30000)          # Eredmény: 30000 befizetve. Új egyenleg: 130000.
számla.kivesz(5000)            # Eredmény: 5000 kivéve. Új egyenleg: 125000.
                               # Amikor a program véget ér, a destruktor lefut: „Anna számlája lezárva."
```

Összefoglalva a konstruktorok és destruktorok közötti különbségeket:

1. **Konstruktor:**
   - Az objektum létrehozásakor fut le.
   - Az objektum attribútumainak inicializálására szolgál.
   - Pythonban a neve mindig **__init__**.

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

4. Osztályattribútumok és metódusok

- **Attribútumok:**
  - Objektum szintű attribútumok - osztályszintű attribútumok
- **Metódusok:**
  - Objektumspecifikus metódusok - osztályszintű metódusok (statikus metódusok)
- **Kód példa:**

```
class Kör:
    pi = 3.1416
    def __init__(self, sugár):
        self.sugár = sugár
    def terület(self):
        return Kör.pi * (self.sugár ** 2)
```

4. Osztályattribútumok és metódusok

Az osztályoknak és objektumoknak lehetnek saját attribútumaik és metódusaik. Az attribútumok az objektumok állapotát (adattagjait), a metódusok pedig a viselkedésüket (funkcióikat) írják le.

Osztályattribútumok:

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

Példa osztályattribútumra: egy **Kör** osztály, amely minden kör számára ugyanazt a **pi** értéket használja.
A **sugár** viszont egy **objektum attribútum**, amely minden egyes kör objektumnál különböző lehet.

```
class Kör:
    pi = 3.1416                          # Osztályattribútum
    def __init__(self, sugár):
        self.sugár = sugár               # Objektum attribútum
    def terület(self):
        return Kör.pi * (self.sugár ** 2)
```

Objektum létrehozása és az attribútumok használata:

```
kör1 = Kör(5)
kör2 = Kör(10)
print(f"Az első kör területe: {kör1.terület()}")          # Eredmény: 78.54
print(f"A második kör területe: {kör2.terület()}")        # Eredmény: 314.16
```

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

Objektum attribútumok

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

```
class Autó:
    def __init__(self, márka, típus):
        self.márka = márka                  # Objektum attribútum
        self.típus = típus                  # Objektum attribútum
    def bemutatkozik(self):
        return f"Ez egy {self.márka} {self.típus}."

autó1 = Autó("Toyota", "Corolla")
autó2 = Autó("Opel", "Astra")
print(autó1.bemutatkozik())       # Eredmény: Ez egy Toyota Corolla.
print(autó2.bemutatkozik())       # Eredmény: Ez egy Opel Astra.
```

Osztálymetódusok és objektummetódusok

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

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

```
class Téglalap:
    def __init__(self, hossz, szélesség):
        self.hossz = hossz
        self.szélesség = szélesség
    def terület(self):
        return self.hossz * self.szélesség
    def kerület(self):
        return 2 * (self.hossz + self.szélesség)
```

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

Az objektummetódusok a **self** segítségével férnek hozzá az objektum attribútumaihoz.

```
téglalap1 = Téglalap(2, 3)
print(f"Az első téglalap területe: {téglalap1.terület()}")      # Eredmény: 6
print(f"Az első téglalap kerülete: {téglalap1.kerület()}")      # Eredmény: 10
```

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

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

```
class Személy:
    létszám = 0                          # Osztályattribútum
    def __init__(self, név):
        self.név = név
        Személy.létszám += 1             # Nő az osztályattribútum

    @classmethod
    def létszám_megjelenít(cls):
        return f"A személyek száma: {cls.létszám}"

személy1 = Személy("Anna")
személy2 = Személy("Béla")
print(Személy.létszám_megjelenít())

# Eredmény: A személyek száma: 2
```

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

Statikus metódusok

A **statikus metódusok** olyan metódusok, amelyek sem az osztályhoz, sem az objektumhoz nem kapcsolódnak közvetlenül. Ezeket a **@staticmethod** dekorátorral jelöljük, és **nincs szükségük sem self, sem cls paraméterre**. Általában olyan funkciók végrehajtására használjuk őket, amelyek nem függenek sem az osztály állapotától, sem az objektum attribútumaitól.

Példa statikus metódusra:

```
class Matek:
    @staticmethod
    def összead(a, b):
        return a + b

print(Matek.összead(5, 10))      # Eredmény: 15
```

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

Összefoglalás:

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

5. Az öröklődés alapjai

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

```
class Állat:
   def __init__(self, név):
      self.név = név
   def beszél(self):
      pass

class Kutya(Állat):
 def beszél(self):
      return f"{self.név} mondja Vau-vau!"
```

Az öröklődés mechanizmusa:

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

Szülőosztály és gyermekosztály

- **Szülőosztály (Parent Class):** amelyből más osztályok örökölhetnek.
- **Gyermekosztály (Child Class):** amely örökli a szülőosztály attribútumait és metódusait, és ezeket szükség esetén felüldefiniálhatja vagy kiegészítheti.

Példa öröklődésre: egy **Állat** nevű szülőosztályunk, egy általános **beszél**() metódussal.

Ebből **Kutya** és **Macska** osztályokat hozhatunk létre, amelyek felül-definiálják a beszél() metódust a saját állathangukkal.

```
class Állat:
    def __init__(self, név):
        self.név = név
    def beszél(self):
        return f"{self.név} valamilyen hangot ad ki."
```

```
class Kutya(Állat):
    def beszél(self):
        return f"{self.név} ugat!"
class Macska(Állat):
    def beszél(self):
        return f"{self.név} nyávog!"
```

```
kutya1 = Kutya("Buksi")
macska1 = Macska("Cirmi")
print(kutya1.beszél())          # Eredmény: Buksi ugat!
print(macska1.beszél())         # Eredmény: Cirmi nyávog!
```

Az öröklődés előnyei:

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

A gyermekosztály metódusainak bővítése:

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

Példa a super() használatára: egy **Jármű** osztály, egy **megjelenít()** metódussal, ami kiírja a jármű típusát. A gyermekosztályok tovább bővíthetik ezt a metódust.

```
class Jármű:
    def __init__(self, típus):
        self.típus = típus
    def megjelenít(self):
        return f"A jármű típusa: {self.típus}"
class Autó(Jármű):
    def __init__(self, típus, márka):
        super().__init__(típus)                # Szülőosztály konstruktorának meghívása
        self.márka = márka
    def megjelenít(self):
        return f"{super().megjelenít()}, Márka: {self.márka}"
autó1 = Autó("Személyautó", "Opel")
print(autó1.megjelenít())          # Eredmény: A jármű típusa: Személyautó, Márka: Opel
```

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

Több gyermekosztály egy szülőosztállyal

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

Példa:

```
class Állat:
    def __init__(self, név):
        self.név = név
    def beszél(self):
        return f"{self.név} valamilyen hangot ad ki."
class Kutya(Állat):
    def beszél(self):
        return f"{self.név} ugat!"
class Madár(állat):
    def beszél(self):
        return f"{self.név} csiripel!"
kutya = Kutya("Buksi")
madár = Madár(„Csőri")
print(kutya.beszél())  # Eredmény: Buksi ugat!
print(madár.beszél())  # Eredmény: Csőri csiripel!
```

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

Felüldefiniálás (Override)

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

Példa:

```
class Személy:
    def __init__(self, név, kor):
        self.név = név
        self.kor = kor
    def bemutatkozás(self):
        return f"Szia, {self.név} vagyok, {self.kor} éves."
class Diák(Személy):
    def __init__(self, név, kor, osztály):
        super().__init__(név, kor)
        self.osztály = osztály
    def bemutatkozás(self):
        return f"Szia, {self.név} vagyok, {self.kor} éves, és a(z) {self.osztály}. osztályba járok."
diák1 = Diák(„Anna", 15, 9)
print(diák1.bemutatkozás())
                        # Eredmény: Szia, Éva vagyok, 15 éves, és a(z) 9. osztályba járok.
```

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

Összefoglalás:

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

1. Példa:

```
class CreditCard:
    def __init__(self, name, number, limit = 8000, bank = "Python Bank"):
        self.name = name
        self.number = number
        self.bank = bank
        self.balance = 0
        self.limit = limit

    def charge(self, amount):
        #not int or not not float
        if not (isinstance(amount, int) or  isinstance(amount, float)) or (amount <= 0):
            print("Charge Denied")
        else:
            self.balance += amount

    def pay(self, amount):
        if not (isinstance(amount, int) or  isinstance(amount, float)) or (amount <= 0):
            print("Charge Denied")
        else:
            self.balance -= amount

    def __str__(self):
        info = "Name: " + self.name + "\n"
        info += "Number: " + "XXXX" + self.number[4:] + "\n"
        info += "Bank: " + self.bank + "\n"
        info += "Balance: " + str(self.balance)
        return info

    def __repr__(self):
        return str(self)

card = CreditCard("Kenny", "12345678")
card.charge(10000)
print(card)
```

2. Példa

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

```
Publikus lista
[]
0
```

```
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
```

```
Privát hozzáférésű lista
A privát verem hossza: 3
print(len(stack_object.__stack_list))    #AttributeError
AttributeError: 'Stack' object has no attribute '__stack_list'
```

3. Példa

```
print('Instance variables: dictionary-ben tárolva')
class PéldaOsztály:       #nem védett változókkal
   def __init__(self, val = 1):
      self.egy = val
   def set_kettő(self, val):
      self.kettő = val
példa1 = PéldaOsztály()
példa2 = PéldaOsztály(2)
példa2.set_kettő(3)
példa3 = PéldaOsztály(4)
Példa3.három = 5
print(példa1.__dict__)
print(példa2.__dict__)
print(példa3.__dict__)
```

```
Instance variables: dictionary-ben tárolva
{'egy': 1}
{'egy': 2, 'kettő': 3}
{'egy': 4, ,három': 5}
```

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

```
Instance variables: dictionary-ben tárolva
{'_PéldaOsztály__egy': 1}
{'_PéldaOsztály__egy': 2, '_PéldaOsztály__kettő': 3}
{'_PéldaOsztály__egy': 4, '__három': 5}
```

Összegzés

1. Bemutattuk az OOP előnyeit és alapelveit
2. Megbeszéltük az osztály és objektum fogalmakat
3. Megvizsgáltuk a konstruktorok és destruktorok szerepét
4. Megbeszéltük az osztályattribútumok és metódusok lehetőségeit
5. Megvizsgáltuk az öröklődés fogalmát
6. Bemutattunk néhány példát

Konzultáció

Minden héten csütörtökön 18:00 – 20:00

Köszönöm a figyelmet!