Python

6. gyakorlat

Strohmayer Ádám

Agenda

- Property
- Enumok
- Dataclassok
- Absztrakt modul
- Fontosabb tervezési minták

Property

Getter-settereket valósít meg alapértelmezetten

```
class Person:
    def __init__(self, age):
        self.age = age  #meghívja a settert

    @property #dekorátor
    def age(self):
        return self.__age #"privát" változóban tároljuk!

    @age.setter #értékek vizsgálatához – párban a @propertyvel
    def age(self, value):
        if value < 0:
            raise ValueError("A kor nem lehet negatív!")
        self.__age = value
```

```
person = Person(25)
print(person.age) #25

person.age = 5
print(person.age) #5
person.age = -10 #ValueError
```

Property – setter nélkül

Ha nincs setter – az adott adattag nem módosítható!

```
class Person:
    def __init__(self, age):
        self._age = age  #inicializálás konstruktorban

    @property #dekorátor
    def age(self):
        return self._age #"privát" változót visszaadjuk!
```

get-set metódusokhoz különböző viselkedést adhat!

```
person = Person(25)
print(person.age) #25

person.age = 5 #AttributeError
```

person._age formában ugyanúgy módosítható…

Enum

Egyedi konstansokat definiál értékkel!

```
from enum import Enum

class Lamp(Enum):
    RED = 1
    YELLOW = 2
    GREEN = 3

print(Lamp.RED) # Lamp.RED
print(Lamp(2).name) # érték szerint: "YELLOW"
print(Lamp["GREEN"].value) # név szerint: 3
Lamp["GREEN"].value = "RED" # AttributeError
```

Háttérben az adattagok immutable singletonok!

(azaz egyedi név-érték párok vannak benne)

Dataclass

Adattároló osztály – boilerplate nélkül

```python
from dataclasses import dataclass

@dataclass
class Pizza:
    name: str
    size: str
    gluten_free: bool = False

    def __str__(self):
        gluten_info = ", gluten free." if self.gluten_free else "."
        return f"{self.name}, {self.size} sized pizza {gluten_info}"
```

```python
pizza1 = Pizza(name="Margherita", size="Medium", gluten_free=True)
pizza2 = Pizza(name="Pepperoni", size="Large")

print(pizza1) # Margherita, Medium sized pizza, gluten free.
print(pizza2.gluten_free) #False
```

Immutable osztályok

Dataclass-szal immutablevé tehetőek a példányok!

```python
from dataclasses import dataclass

@dataclass(frozen=True)
class Food:
    name: str
    like: bool #nem fog típushelyességet ellenőrizni!

fruit = Food(name="Appel", like="igen")
print(fruit) #Fruit(name='Appel', like='igen')
fruit.name="Apple" #FrozenInstanceError!
```

Pydantic dataclass

Pydantic külső könyvtár lehetővé teszi a típushelyességet!

```python
from pydantic.dataclasses import dataclass

@dataclass(frozen=True)
class Meal:
    dish: str
    servings: int

meal = Meal("Salad", "uff") #ValidationError!
```

pip install pydantic

Nem csak dataclassokra!

Staticmethod

Egységbezárást segíti – valójában nincs kötve az osztályhoz!

```python
class TempConverter:
    @staticmethod
    def celsius_to_fahrenheit(celsius):
        return (celsius * 9/5) + 32

    @staticmethod
    def fahrenheit_to_celsius(fahrenheit):
        return (fahrenheit - 32) / (9/5)

c = 25
print(TempConverter.celsius_to_fahrenheit(c)) #77
```

Nem használ sem cls-t, sem self-et!

Logikai összetartozás miatt van osztályon belül.

Absztrakt modul

Lehetővé teszi absztrakt osztályok és metódusok definiálását!

Absztrakt osztályokat nem lehet példányosítani absztrakt metódusok megvalósítása nélkül!

```
from abc import ABC, abstractmethod

class Food(ABC): #ABC - ABstract Class
    @abstractmethod #implementálandó
    def yum(self):
        pass

moms_food = Food() #TypeError (yum miatt)
```

Interfész itt: csakis metódusokkal rendelkező absztrakt osztály

Absztrakt példa

```python
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def make_sound(self):
        pass

    @abstractmethod
    def move(self):
        pass
```

```python
class Dog(Animal):
    def make_sound(self):
        return "Moo!"

    def move(self):
        print("no")

texas = Dog()
texas.move() #no
print(texas.make_sound()) #Moo!
```

Ekkor magában az Animal osztály nem példányosítható!

A Dog osztály valósítja meg az Animal absztrakt osztályt!

Tervezési minták

OOP világban bevett jó gyakorlatok

Kreációs minták – létrehozáshoz

pl. Singleton, Factory, Builder

Strukturális minták – objektumok közötti kapcsolat

pl. Adapter, Decorator

Viselkedési minták – objektumok közötti interakció, adatcsere

pl. Observer

```python
class EUConnector:
    def csatlakoztat(self):
        return "Európai dugasz csatlakoztatva."

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

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

adapterEU = Adapter(USAConnector())
print(adapterEU.csatlakoztat())
```

```python
class Ház:
    def __init__(self):
        self.alap = None
        self.falak = None
        self.tető = None
    def __str__(self):
        return f"Alap: {self.alap}, Falak:
{self.falak}, Tető: {self.tető}"
```

```python
class HázÉpítő:
    def __init__(self):
        self.ház = Ház()
    def alap_épít(self):
        self.ház.alap = "Betonalap"
        return self
    def falak_épít(self):
        self.ház.falak = "Tégla falak"
        return self
    def tető_épít(self):
        self.ház.tető = "Cseréptető"
        return self
    def get_ház(self):
        return self.ház

házépítő = HázÉpítő()
ház =
házépítő.alap_épít().falak_épít().tető_épít().g
et_ház()
print(ház)
```

```python
class Kávé:
    def ár(self):
        return 500

class TejKávéDecorator:
    def __init__(self, kávé):
        self.kávé = kávé
    def ár(self):
        return self.kávé.ár() + 100

class CukorKávéDecorator:
    def __init__(self, kávé):
        self.kávé = kávé
    def ár(self):
        return self.kávé.ár() + 50

kávé = Kávé()
tej_kávé = TejKávéDecorator(kávé)
cukros_tej_kávé = CukorKávéDecorator(tej_kávé)
print(cukros_tej_kávé.ár())
```

```python
class Autó:
    def vezet(self):
        return "Az autó halad."

class Motor:
    def vezet(self):
        return "A motor száguld."

class JárműFactory:
    @staticmethod
    def jármű_létrehoz(típus):
        if típus == "autó":
            return Autó()
        elif típus == "motor":
            return Motor()
        else:
            raise ValueError("Ismeretlen
járműtípus")
```

```python
jármű1 = JárműFactory.jármű_létrehoz("autó")
jármű2 = JárműFactory.jármű_létrehoz("motor")
print(jármű1.vezet())
print(jármű2.vezet())
```

```python
class Singleton:
    _instance = None

    def __new__(cls): #memóriafoglaláskor lefutó dunder
        if cls._instance is None: #ha osztályszinten nincs ilyen még
            cls._instance = super(Singleton, cls).__new__(cls) #legyen
        return cls._instance #mindig a meglévőt adjuk vissza

singleton1 = Singleton()
singleton2 = Singleton()

print(singleton1 is singleton2)
# true - memóriában ugyanarra a példányra fognak mutatni
```

Feladatok Canvasben!

Köszönöm a figyelmet!