Objektumorientált Szoftverfejlesztés

Hipszki János
2024. ősz
hipszki.janos@gde.eu

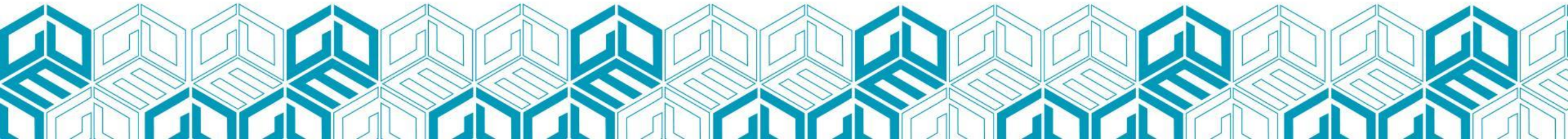
GÁBOR DÉNES EGYETEM

GÁBOR DÉNES EGYETEM

Összefoglaló

A követelmény

- Beadandó programozási feladat feltöltése github-ra
- Pontos feladat leírás később kerül publikálásra
- Határidő: 2024. 11. 10.



A tananyag

- év közben töltöm fel a frissített, átdolgozott anyagokat
- távoktatásos előadások videóra rögzítve
- Python telepítése
- PyCharm környezetet érdemes használni
- Academic licensing
- https://github.com/hipszkij/gde_oop_2024_2

Github használata

- legyen egy github account
- legyen git a gépen
- commitoljuk be a módosításokat

GÁBOR DÉNES EGYETEM

Miért a Python?

Miért a Python?

Ranglista:

- A Python a TIOBE Index szerint 2023-ban az első 3 programozási nyelv között szerepel.
- 2024 szeptemberében az első helyen.
- https://www.tiobe.com/tiobe-index/

Gyors Növekedés:

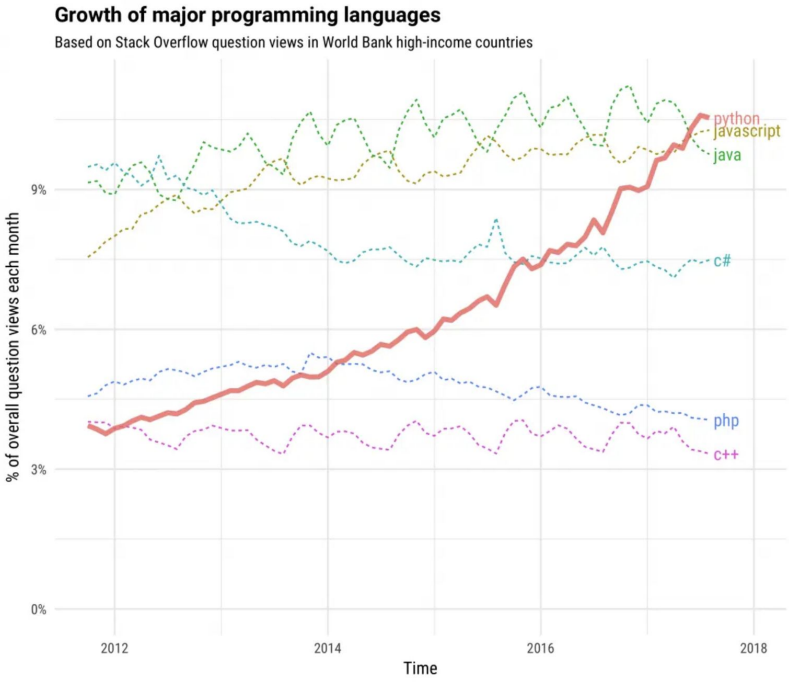
- Az elmúlt évtizedben a Python az egyik leggyorsabban növekvő programozási nyelvvé vált.

Első Választás:

- A kezdő programozók körében a Python gyakran az első választott nyelv.

Közösség:

- Több millió fejlesztő választja és támogatja aktívan a Python-t világszerte.



Python története

Kezdetek:

1991: Guido van Rossum bemutatja a Python 0.9.0 verzióját.
Zen of Python

Főbb Verziók:

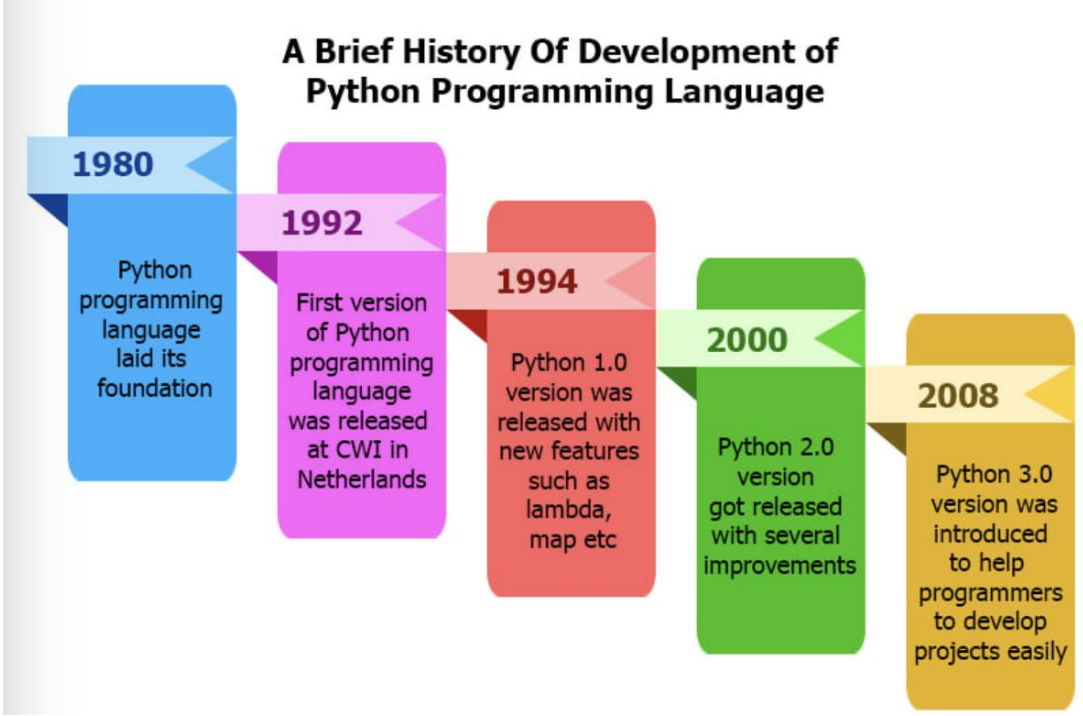
- 2000: Python 2.0 – List comprehensions, garbage collection
- 2008: Python 3.0 – Print függvénnyé vált, új szintaxis

Növekedés:

2010-től: A Python népszerűsége rohamosan nő az AI és a Data Science területén való alkalmazása miatt. (pl.: TensorFlow)

Közösség és Fejlődés:

2020: Python továbbra is aktívan fejlődik, támogatva az iparágat és a tudományt egyaránt.



Python sokoldalúsága - területek

Webfejlesztés:

- Django, Flask: Könnyen használható keretrendszerek a webalkalmazások létrehozásához.

Adattudomány és Mesterséges Intelligencia:

- Pandas, NumPy, SciPy: Adatelemzés és statisztika.
- TensorFlow, PyTorch, Scikit-learn: Gépi tanulás és mély tanulás.

Automatizálás:

- Automate, PyAutoGUI: Ismétlődő feladatok automatizálása a rendszeren.

Asztali alkalmazások:

- PyQt, Tkinter: Grafikus felhasználói felületek készítése asztali alkalmazásokhoz.

Python sokoldalúsága - egyéb alkalmazások

Játékfejlesztés:

- Pygame: Játékok és interaktív alkalmazások készítése.

Hálózatok és biztonság:

- Scapy, Requests: Hálózati kommunikáció, biztonsági tesztelés.

Pénzügyek és üzlet:

- QuantLib, Zipline: Pénzügyi analitika és kereskedelmi rendszerek fejlesztése.

Oktatás és kutatás:

- Jupyter Notebook: Interaktív jegyzetfüzetek az oktatás és a kutatás elősegítésére.

Python olvasmányossága I.

A Python Filozófiája (Zen of Python):

- "Simple is better than complex."
- "Readability counts."
- import this

Szintaxis:

- Szigorú behúzás: Strukturált és rendezett kód.
- Kód szegmensek: Következetes blokkok és kódszervezés.

Egyszerű nyelvi elemek:

- A kevesebb néha több: Az egyszerűsített szintaxis csökkenti a felesleges kódot és a bonyolultságot.

Python olvasmányossága II.

Változók deklarálása és inicializálása

Java:

```
String name = "John";
int age = 30;
```

Python:

```
name = "John"
age = 30
```

For ciklusok

***Java*:**

```
for (int i = 0; i < 5; i++) {
    System.out.println(i);
}
```

***Python*:**

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

Python olvasmányossága III.

Függvények deklarálása

Java:

```
public static int add(int a, int b) {
    return a + b;
}

int result = add(5, 3);
```

Python:

```
def add(a, b):
    return a + b

result = add(5, 3)
```

Lista létrehozása és hozzáférés

***Java*:**

```
ArrayList<String> fruits = new
ArrayList<String>();
fruits.add("apple");
fruits.add("banana");
String fruit = fruits.get(0);
```

***Python*:**

```
fruits = ["apple", "banana"]
fruit = fruits[0]
```

Python, pycharm telepítése

Python telepítése:

- https://www.python.org/downloads/

PyCharm telepítése

- Pro verziót lehet telepíteni ingyenes Educational license-szel, amit a gde.hu-s email címmel tudtok igényelni: https://www.jetbrains.com/shop/eform/students
- https://www.jetbrains.com/pycharm/download/
-

GÁBOR DÉNES EGYETEM

Python alapok

Python kód elindítása

- Nincs szükség külön fordítási lépésre.
- A Python kód közvetlenül futtatható.

```
print("Hello, Python!")
```

- Parancssorban futtatás:

```
python your_script_name.py
```

Változók

- Típus deklarálás nélkül is működik (automatikusan felismeri), de python 3.6-tól kezdődően használható.

```
name = "John"
age = 30
```

```
name: string = "John"
age: int = 30
```

- int, float, string, bool, list, tuple, set, dict (kulcs-érték)
-

Vezérlési Szerkezetek

- if, elif, else
- for

```
test_int = 10

if test_int > 10:
    print("Nagyobb, mint 10")
elif test_int > 9:
    print(10)
else: print("kevesebb, mint 10")
```

```
fruits = ["lemon","apple", "banana" ]

for fruit in fruits:
    print(fruit)
```

Feladat

- Hozz létre egy listát, amelyben 5 különböző gyümölcs neve található.
- Írj egy for ciklust, ami kiírja a gyümölcsök neveit betűrendben.
- Adj hozzá egy feltételvizsgálatot, ami csak az "a" betűvel kezdődő gyümölcsöket írja ki.

- Használd a sorted() függvényt a lista rendezéséhez.
- Használd a string startswith() metódusát az "a" betűvel kezdődő gyümölcsök kiválasztásához.
- https://docs.python.org/3/library/index.html

GÁBOR DÉNES EGYETEM

Python függvények és modulok

Függvények használata

- Funkció: Egy újrafelhasználható kód részlet, ami egy feladatot lát el.
- Modul: Egy fájl, ami Python kódokat tartalmaz.
- Példa Funkcióra:

```
def udvozles():
    print("Szia!")
```

- Példa modulra:

```
import math
```

Paraméterek és Visszatérési Értékek

- Paraméterek: Az értékek, amiket a funkció kaphat.
- return: Visszaad egy értéket a funkció meghívójának.
- Példa Paraméterre és return-re:

```
def osszead(a, b):
    return a + b
```

Paraméterek és Visszatérési Értékek

- import: Egy modul funkcióit teszi elérhetővé.
- Python Standard Library: Alapvető modulok gyűjteménye.
- Példa import használatra:

```
import math
math.sqrt(9)  # kimenet: 3.0
```

Feladat

- Készíts egy funkciót, ami kiszámolja egy kör területét és kerületét!
- Készíts egy függvényt, ami a paraméterben kapott tömbből visszaadja a legnagyobb értéket. (ne beépített függvényt használj)

Feladat - myMath modul

- irj egy négyzetre emelő funkciót és szervezd ki egy modulba
- használd a modult egy négyzet területének kiszámítására

GÁBOR DÉNES EGYETEM

Objektumorientált programozás alapok

Programozási paradigmák

- **Procedurális programozás:** A programot utasítások és eljárások sorozataként szervezi, ahol minden eljárás a program egy-egy feladatát hajtja végre. A C nyelv egy tipikus példája a procedurális programozásnak.

- **Objektumorientált programozás (OOP):** A programot objektumokként szervezi, amelyek adatokat és az adatokon végezhető műveleteket (metódusokat) egyaránt tartalmaznak. A Java, C++ és Python nyelvek objektumorientált paradigmákat is támogatnak.

- **Funkcionális programozás:** A programozást matematikai funkciókon alapuló megközelítésben szervezi, az oldalmellékhatások minimalizálására törekedve. Haskell és Erlang nyelvek jelentősek a funkcionális programozás területén.

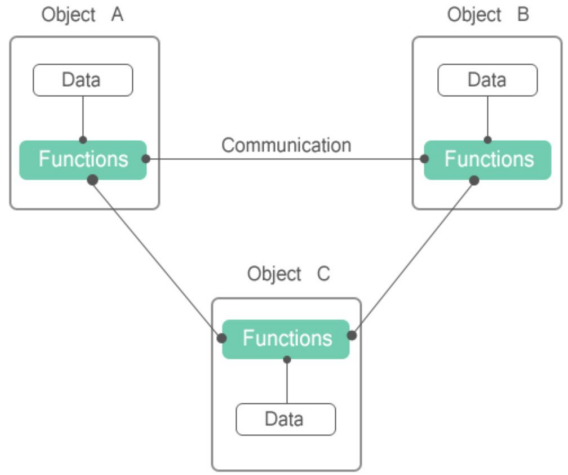
- **Deklaratív programozás:** Ahelyett, hogy a "hogyan"-ra fókuszálna (azaz hogyan oldjuk meg a feladatot), a "mit"-re koncentrál: mit szeretnénk elérni. A SQL (adatbázislekérdezések) egy jó példa a deklaratív paradigma alkalmazására.

OOP paradigma

- Az OOP egy programozási paradigma, mely az objektumok köré szerveződik.
- Az objektumok osztályokból példányosíthatók.
- Az osztályok tulajdonságokat és metódusokat definiálnak.
- Az OOP lehetővé teszi, hogy a kód szervezettebb és könnyebben kezelhető legyen, mivel az objektumok különálló egységek, amelyekkel könnyű dolgozni.

OOP paradigma II.

- Az objektumorientált megközelítés kialakulásának fő motiváló tényezője az procedurális megközelítésben tapasztalt hibák némelyikének kiküszöbölése.
- Az OOP az adatokat a programfejlesztés kritikus elemeként kezeli, és nem engedi, hogy azok szabadon áramoljanak a rendszerekben.
- Az adatokat szorosabban köti a rajtuk működő függvényekhez, és megvédi őket a külső függvények véletlen módosításától.
- Az OOP lehetővé teszi a probléma objektumoknak nevezett entitásokból álló egységekre való felbontását, majd az adatok és a funkciók ezen objektumok köré épülnek.
- Egy objektum bizonyos adataihoz csak az adott objektumhoz kapcsolódó függvény férhet hozzá.
- Egy objektum függvényei azonban hozzáférhetnek más objektumok függvényeihez.

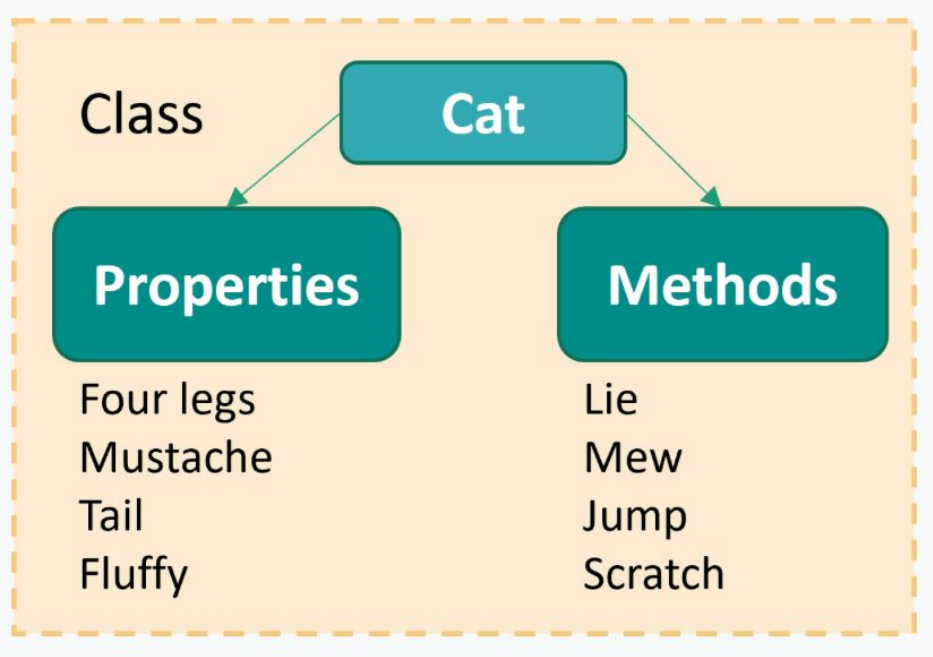


Miért OOP?

- Modularitás: Könnyebb a kód szervezése és kezelése.
- Újrafelhasználhatóság: Az osztályok többször felhasználhatók.
- Kiterjeszthetőség, öröklődés: Könnyű új funkciókkal bővíteni a meglévő kódot.
- Absztrakció és egységbezárás támogatása.

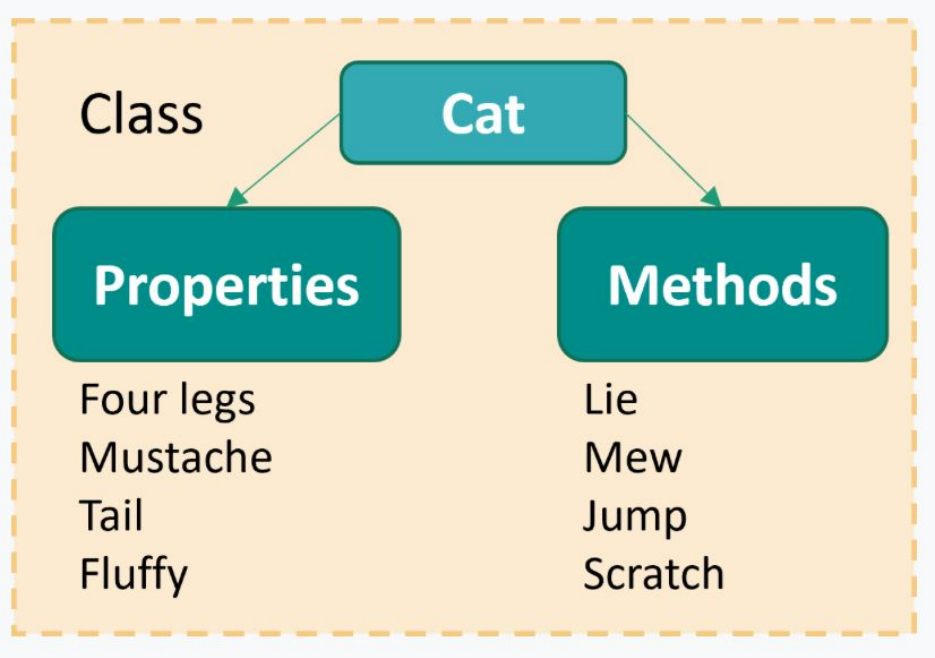
Mi az osztály (Class)?

- Egy összetett adatszerkezet, ami az alábbiakból áll
  - adatok (attributumok)
  - azokat kezelő metódusok
- Lényegében egy tervrajz
- Alapvető építőelemei az objektumorientált programozásnak
- pl: vásárló, eladó, termék, bolt stb.



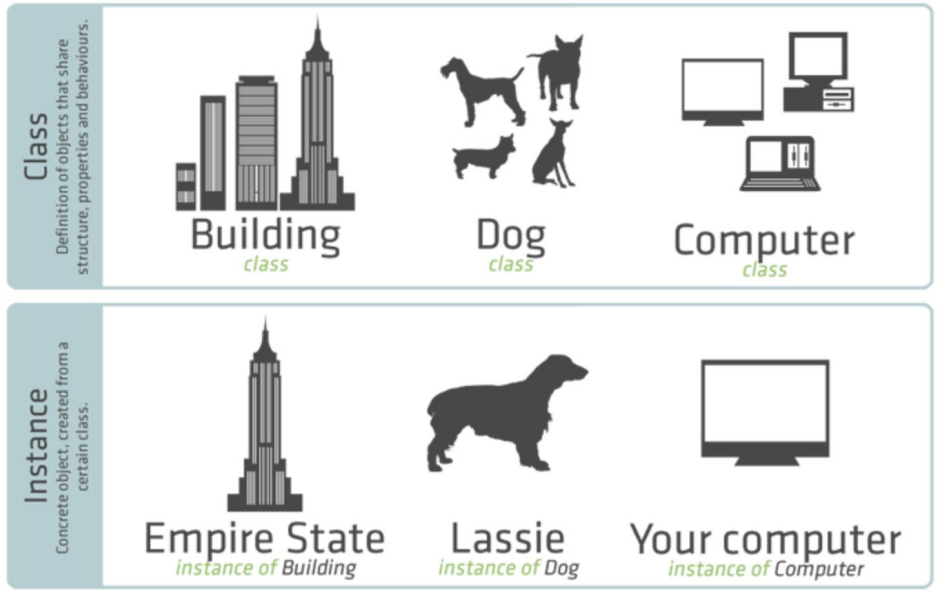
Mi az osztály (Class)?

- Attribútum:
  - Az osztály egy mezője, konkrét adattárolási képességű adattagja.
- Metódus:
  - Olyan eljárás, mely az adott osztály attribútumaival végez valamilyen műveletet.



Mi az osztálypéldány (Object)?

- A Osztály (class) tervrajz alapján létrehozhatunk példányokat a tervben szereplő osztályból
- Ezek lesznek az osztálypéldányok vagy object-ek
- Lényegében egy változó, aminek a típusa az adott osztály



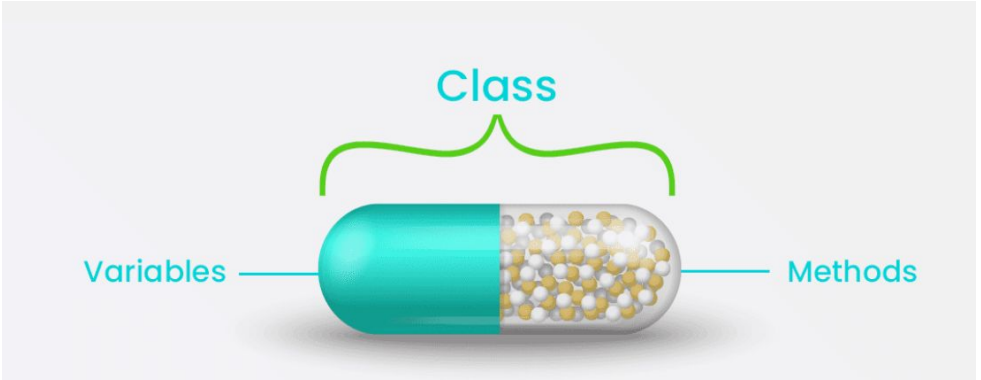
A pythonban minden osztály

- próbáljuk ki, hogy mit mond a python a type függvénnyel a különböző változótípusokról
- minden objektumnak van egy id-ja id()
- is operator az össszehasonlításra
- isinstance
- shallow copy, deep copy

```
a = 5
print(type(a))
x = {1:"one", 2: "two", 3: "three"}
print(type(x)) # <class 'dict'>
tesztnev = "alma"
print(type(tesztnev))
```

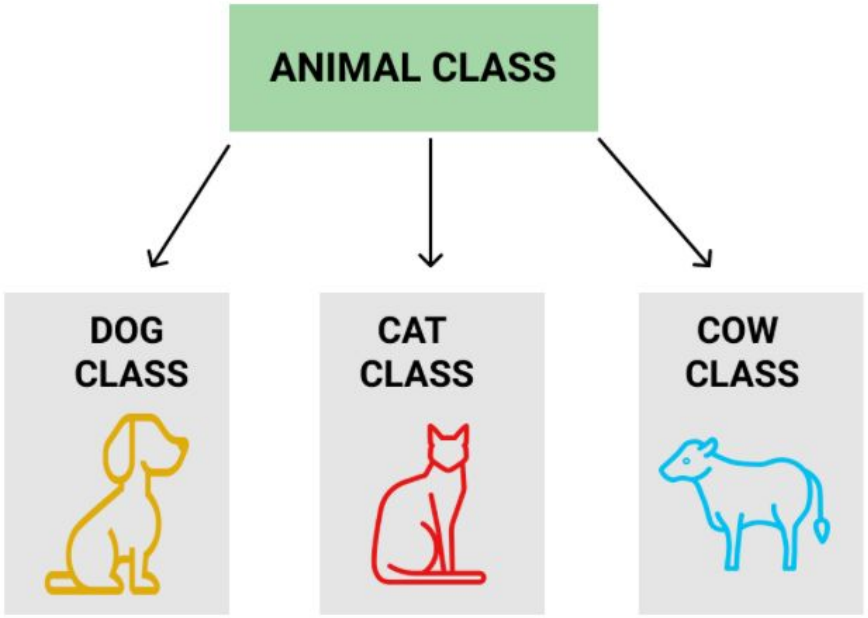
Egységbezárás (encapsulation)

- Az objektum állapotának és viselkedésének egyesítése.
- A belső állapot elrejtése a külvilág elől.
- Az encapsulation az adatok és a műveletek egy osztályba történő csomagolását jelenti, védi az objektum belső állapotát a külvilág elől.



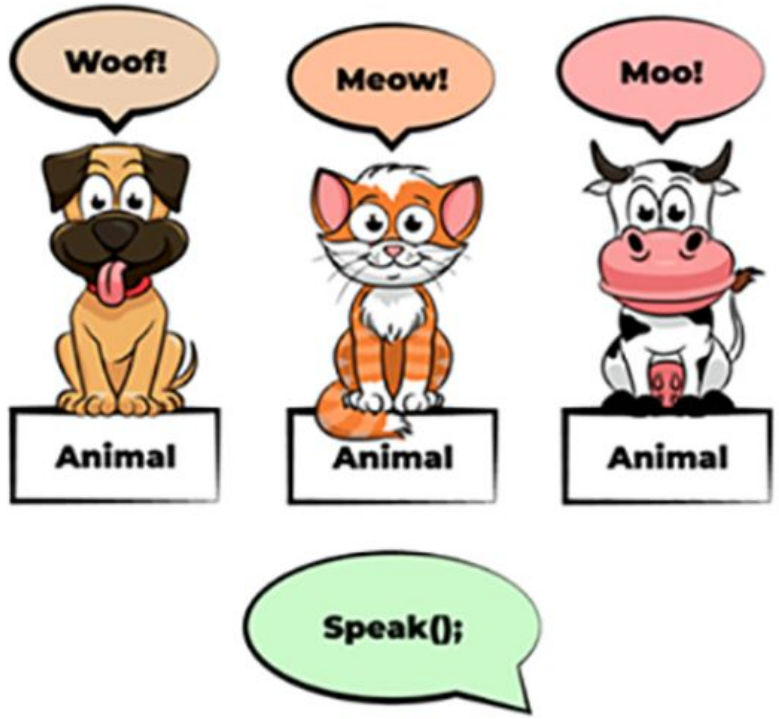
Öröklődés (inheritance)

- Az osztályok újrafelhasználása és bővítése.
- "is a" kapcsolat megteremtése osztályok között.
- triangle is a polygon
- háromszög egy poligon
- Az öröklődés lehetővé teszi egy osztály tulajdonságainak és metódusainak újrafelhasználását és bővítését.



Polimorfizmus

- Ugyanazon metódus különböző implementációi.
- Ugyanaz a kód különböző objektumokon.
- A polimorfizmus segít a kód újrafelhasználhatóságában, lehetővé téve ugyanazon metódus különböző implementációját.



Absztrakció

- Csak a szükséges információ megjelenítése.
- A komplexitás elrejtése a fejlesztő elől.
- Az absztrakció az összetett rendszerek egyszerűsítését jelenti, a részletek elrejtésével és csak a szükséges információk megjelenítésével.
- Adat absztrakció
- Metódus absztrakció

Osztályok problémafelvetésekben I.

- A tervezés első lépése a probléma megértése
- Meg kell találni és megfogalmazni a problémafelvetésben az objektumokat
  Problémafelvetés: Szeretnénk egy rendszert kialakítani egy könyvtár számára, ahol nyilvántarthatjuk a könyveket, a kölcsönzéseket és a tagokat.

- **Objektumok keresése:**
  - Könyv: Szerző, Cím, Kiadás éve, ISBN szám
  - Tag: Név, Tagazonosító, Email
  - Kölcsönzés: Tag, Könyv, Kölcsönzés dátuma, Visszavétel dátuma

Osztályok problémafelvetésekben II.

Problémafelvetés: Szeretnénk egy rendszert fejleszteni egy egyetem számára, amely kezeli a tanárokat, diákokat, kurzusokat és az osztályzatokat.

Objektumok keresése:

- Diák: Neve, Diákazonosító, Tanulmányi átlag
- Tanár: Neve, Munkatársi azonosító, Szak
- Kurzus: Kurzuskód, Név, Tanár
- Osztályzat: Diák, Kurzus, Érdemjegy

Problémafelvetés - Feladat I.

Problémafelvetés: Tervezz egy városi tömegközlekedési rendszert, amely integrálja a buszokat, vonatokat, villamosokat és metrókat. A rendszernek képesnek kell lennie kezelni az útvonalakat, megállókat, menetrendeket és az utasokat. Továbbá, a rendszernek nyomon kell követnie az utasok utazását, a jegyvásárlásokat és a bérleteket, valamint a járművek karbantartási állapotát és üzemidejét is.

GÁBOR DÉNES EGYETEM

Python osztályok, attribútumok és metódusok

Elnevezési szabályok

A Pythonban a Pascal Case-t (más néven Upper Camel Case-t) használjuk az osztálynevek elnevezési konvenciójaként.

- A Pascal Case egy elnevezési konvenció, amelyben minden szó első betűjét nagybetűvel írjuk.

- Például:
  - House
  - BankAccount
  - RunningShoe
  - LinkedList

Az osztály nevének első betűjét nagy kezdőbetűvel írjuk.

Osztálydefiníció

Header
- első sor
- megadja az osztály nevét, illetve öröklési szabályait
    - class Polygon:

Body:
- az osztály elemeit tartalmazza
- behúzással kell kezdeni
    - osztály attribútumok
    - metódusok
    - __init__() - konstruktor

```
class Polygon:
    pass
```

Konstruktor - __init__()

- Osztály példányosítás:

```
triangle = Polygon()
```

- Osztály példányosításakor meghívódik a konstruktor
- __init__(self)
  - a self, mint az osztály önmaga, mindig az argumentumlista első eleme
  - utána jöhet a többi argumentum
  - default értékűek a végén

Példány attribútum

- self kulcsszóval az adott osztálypéldányra hivatkozva
- A példány attribútumok az objektumok egyedi tulajdonságai, melyek az objektum létrehozásakor kerülnek inicializálásra.
- Egy példány attribútumok értékei az egyes objektumok között eltérhetnek.
- Az attribútumok a példány állapotát reprezentálják és meghatározzák annak viselkedését.

```
class Animal:

    new *
    def __init__(self, animal_type):
        self.animal_type = animal_type
        self.animal_name = ''
        self.animal_legs = 4
```

Példány attribútum módosítása

- A példány attribútumokhoz közvetlenül hozzáférhetünk és módosíthatjuk őket az objektumon keresztül.
- Az attribútumok módosítása befolyásolhatja az objektum viselkedését és állapotát.

```
my_animal = Animal('Parrot')
my_animal.animal_legs=2

print(my_animal.animal_legs)
```

Osztály Attribútumok - Alapok

- Az osztály attribútumok az osztályhoz tartoznak, és azonosak minden példány számára.
- Az osztály attribútumok hasznosak globális változók vagy konstansok tárolására, melyek az osztály összes példányára érvényesek.

```
class Kutya:
    fajta = "emlős"  # osztály attribútum

    def __init__(self, nev, kor):
        self.nev = nev  # példány attribútum
        self.kor = kor  # példány attribútum
```

Osztály Attribútumok - Hozzáférés és Módosítás

- Az osztály attribútumokhoz hozzáférhetünk a példányokon vagy az osztályon keresztül is.
- Az osztály attribútumok módosítása az osztályon keresztül befolyásolja az összes példány értékét.

```
class Animal:

    _id_counter = 1

    Aron Boga
    def __init__(self,breed,type="Kutya"):
        self.type=type
        self._breed=breed
        self._id=Animal._id_counter

        Animal._id_counter+=1
```

Osztály Attribútumok - Gyakorlati Alkalmazás

- Osztály attribútumok a közös tulajdonságok tárolására használhatók, mint például konstansok vagy beállítások.
- Jó gyakorlat az osztály attribútumokat csak az osztály szintű tulajdonságok tárolására használni.

```
class Auto:
    alap_ar = 20000  # osztály attribútum

    def __init__(self, tipus):
        self.tipus = tipus  # példány attribútum

    def ar_kiszamitas(self, extra):
        return self.alap_ar + extra
```

Osztályok Metódusai - Bevezetés

- A metódusok az osztályok által definiált függvények.
- Kapcsolatban vannak az osztály példányaival, és műveleteket végeznek azok attribútumaival.
- def kulcsszóval
- A self paraméter automatikusan átadódik, és referenciaként hivatkozik az osztály aktuális példányára.

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

    def ugat(self):
        print(f"{self.nev} mondja: Vau!")
```

Hogyan Használjuk az Osztály Metódusait?

- Az osztály metódusait az osztály példányain keresztül hívjuk meg.
- a hivásban átadhatjuk az attribútumokat

```
dog = Kutya("Bodri")
dog.bemutatkozik()  # Kimenet: Szia, a nevem Bodri!
```

GÁBOR DÉNES EGYETEM

input, hibakezelés és dunder metódusok

Felhasználói adatbekérés Pythonban

- Az input() függvény lehetővé teszi, hogy a felhasználótól szöveges bemenetet kérjünk.
- A kapott adatot stringként kezeli a Python.
- isinstance() függvénnyel tudjuk ellenőrizni a típust

```
szam1 = input("írj be egy számot ")

if isinstance(szam1,int):
    print(type(szam1))
else:
    print("Nem szám")
```

Az input() függvény típuskonverzióval

- Az input() függvény mindig stringként adja vissza a bemenetet.
- Több adatot is be lehet kérni a felhasználótól egymás után.

```
nev = input("Kérlek, add meg a neved: ")
kor = int(input("Kérlek, add meg a korod: "))
print(f"Szia {nev}, {kor} éves vagy.")
```

Hiba kiváltása (raise) Pythonban

- A raise utasítással manuálisan válthatunk ki hibákat a kódban.
- A raise segítségével hatékonyabban kezelhetjük a váratlan problémákat és jobban ellenőrizhetjük a kód futását.
- https://docs.python.org/3/tutorial/errors.html

```
szam1 = input("írj be egy számot ")

if not isinstance(szam1, int):
    raise TypeError("Nem szám")
```

Hogyan alkalmazzuk a raise-t a kód jobbá tételére?

- **Pontos hibaüzenet:** Adjon a raise utasítás pontos és érthető hibaüzenetet, hogy könnyebb legyen a hibaelhárítás.
- **Specifikus hibatípusok:** Használjunk specifikus kivételtípusokat a hiba természetének jobb tükrözésére.
- **Dokumentáció:** Dokumentáljuk, milyen hibákat válthat ki a függvény vagy metódus.
- **Szükségesség:** Csak akkor használjuk a raise-t, ha valóban szükséges, mert megállítja a kód futását.
- Megfelelő típus használata: TypeError, ValueError, Exception stb.

Try-Except a hibakezelésben

- A try-except blokk a Pythonban a hibakezelés egyik kulcseleme. Egy olyan mechanizmus, ami lehetővé teszi számunkra, hogy a program futása közben előforduló hibákat, kivételeket ("exceptions") kezelni tudjuk. A try blokkban helyezzük el azt a kódrészletet, amely hibát válthat ki, míg az except blokkban definiáljuk, hogy a program hogyan reagáljon a kiváltott hibára.
- Használata javítja a program robosztusságát, mivel lehetővé teszi, hogy a program tovább fusson egy hiba bekövetkezte után is, és kezelje azt, ahelyett, hogy összeomlana. Emellett segít abban is, hogy informatív hibaüzeneteket jeleníthessünk meg a felhasználó számára, ami megkönnyíti a hiba diagnosztizálást és a hibaelhárítást.

```
while True:
    try:
        x = int(input("Please enter a number: "))
        break
    except ValueError:
        print("Oops!  That was no valid number.  Try again...")
```

Magic Osztálymetódusok Pythonban

- A speciális metódusok, más néven "mágikus" vagy "dunder" (double underscore) metódusok, lehetővé teszik, hogy a saját osztályainkban testreszabott viselkedést definiáljunk bizonyos beépített Python műveletekhez vagy szintaktikai elemekhez. Ezek a metódusok dupla aláhúzással kezdődnek és végződnek (pl.: **__init__, __str__, __add__**).

- **__init__:** Az osztály példányosításakor automatikusan meghívódik.
- **__str__:** Meghatározza, hogy az osztály példánya hogyan jelenjen meg sztringként, amikor a print() függvényben használjuk.
- **__add__:** Lehetővé teszi, hogy az osztály példányaihoz adás műveletet definiáljunk.

```python
class Szam:
    def __init__(self, ertek):
        self.ertek = ertek

    def __str__(self):
        return f"Szám: {self.ertek}"

    def __add__(self, masik):
        return Szam(self.ertek + masik.ertek)

# Példányosítás
szam1 = Szam(5)
szam2 = Szam(3)

# __str__ metódus használata
print(szam1)  # Output: Szám: 5

# __add__ metódus használata
szam3 = szam1 + szam2
print(szam3)  # Output: Szám: 8
```

Osztálydefiniciók - Feladat I.

- Leírás: Hozz létre egy jegyszámoló osztályt, ami a megadott teszteredmény és fejlesztési feladat eredmény alapján kiszámolja az érdemjegyet
- GradeCalculator osztály
  - testScore és egy devScore attribútumok
  - készíts egy osztály metódust ami a fenti két attribútum átlagát kiszámolja
  - készítsünk egy metódust ami az átlag alapján kiírja az osztályzatunkat (>90 = 5, 80>=4, 70 >= 3, 60 >=2, egyébként: 1)

Osztálydefiniciók - Feladat II.

- Hozz létre egy olyan Étterem osztályt, aminek van neve és vannak a menüelemei, amelyek Étel osztály objektumai
- Az Ételnek legyen neve és ára
- Legyen kiíró metódusa (__str__), ami kiírja az étterem nevét
- Legyen egy getmenuItems metódus ami kiírja menü elemeit NÉV……..ÁR Ft felosztásban
- Lehessen userinputban ételt és árat hozzáadni, ellenőrizd ezen értékeket, hogy a szám ne lehessen több, mint 100000 Ft és nagyobb legyen mint 0
- Legyen hozzáadó (__add__) metódusa, ami egy új menüelemet ad hozzá

Osztálydefiniciók - Feladat II.

- Hozz létre egy User osztályt aki később majd rendelés tud leadni (id, név, e-mail cím, cím)
- Hozz létre egy Order osztályt ami a user által a kiválasztott étteremből, kiválasztott termékre rendelést tud leadni.
- A menü listázása után kérj be a felhasználótól egy id-t amire rendelést rögzít a rendszer.
- Írj egy függvényt amivel lekérhetőek a már leadott rendelések

GÁBOR DÉNES EGYETEM

Python egységbezárás és absztrakció

Non-public attribútumok

- Az egységbezárás (Encapsulation) megvalósítása a Pythonban a non-public attriútúmokon keresztül valósul meg
- non-public attribútumok olyan objektum attribútumok, melyek nem szándékoznak közvetlenül hozzáférhetőek lenni az osztályon kívülről. Pythonban a non-public attribútumok neve egy aláhúzással kezdődik (pl.: _color).
- A non-public attribútumok segítenek az adatok elrejtésében és az implementáció részleteinek elrejtésében.
- A non-public attribútumok használata nem akadályozza meg a hozzáférést, de egy "ne nyúlj hozzá" jelzésként szolgálnak.

```
class Titkos:
    new *
    def __init__(self):
        self._titkos_attributum = "Titkos üzenet"

    2 usages  new *
    def kiir(self):
        print(self._titkos_attributum)



obj = Titkos()
obj.kiir()  # Output: Titkos üzenet

obj._titkos_attributum = "Nem annyira titkos"

obj.kiir()
```

Name mangling

- Az attribútumok neveinek átalakítása
- A name mangling (név torzítás) egy technika, melynek során az interpreter megváltoztatja a két aláhúzással kezdődő attribútumok nevét, hogy elkerülje a névütközéseket az öröklődés során.

- Az attribútum neve ilyenre változik: _OsztalyNév__attributumNév
- Ez segít megőrizni az attribútum non-public, még akkor is, ha alosztályokban használják.
- Hozzáférhetünk, de soha ne tegyük

```
class Titkos:
    new *
    def __init__(self):
        self.__szuper_titkos_attributum = "Titkos üzenet"

    2 usages  new *
    def kiir(self):
        print(self.__szuper_titkos_attributum)


obj = Titkos()
obj.kiir()  # Output: Titkos üzenet

obj._Titkos__szuper_titkos_attributum = "Nem annyira titkos"

obj.kiir()
```

Non-public metódusok

- A non-public metódusokat arra tervezték, hogy csak az adott osztályon belül legyenek elérhetőek.
- A non-public metódusok neve előtt egy aláhúzás karakter (_) található.
- Például:_nem_publikus_logolas(self)
- Célja a belső logika vagy segédfüggvények elrejtése.
- A non-public metódusok lehetővé teszik az osztályok zártabbá tételét a külvilág felé.
- Bár technikailag elérhetők kívülről is, az a konvenció, hogy csak az osztályon belül használjuk őket.
- A non-public metódusok nem változtatják meg a Python futását, inkább egy megállapodás a fejlesztők között a kód olvashatóságáért és karbantarthatóságáért.

```
class myMath:
    new *
    def __init__(self):
        pass

    1 usage  new *
    def osszead(self,a,b):
        c = self._resz_osszead(a, b)
        return c

    1 usage  new *
    def _resz_osszead(self, a, b):
        return a + b
```

Getterek és Setterek - Miért?

- Segítenek az osztály attribútumainak elrejtésében és csak korlátozott hozzáférés biztosításában.
- Kontroll alatt tartják, hogyan érik el és módosítják az objektum attribútumait.
- konvencio:
  - getter: get + ‘_’ + attribútum név get_menu_items
  - setter: set + ‘_’ + attribútum név set_menu_items

Getterek

- A getter metódusok az attribútum értékének lekérdezésére szolgálnak.

- Segítenek lekérdezni az attribútum értékét anélkül, hogy közvetlenül hozzáférnénk.

```
class Auto:
    def __init__(self):
        self._sebesseg = 0

    def get_sebesseg(self):
        return self._sebesseg
```

Setterek

- A setter metódusok lehetővé teszik az attribútum értékének módosítását.
- Segítenek módosítani az attribútum értékét bizonyos szabályok vagy korlátozások mellett.

```
class Auto:
    def __init__(self):
        self._sebesseg = 0

    def set_sebesseg(self, ertek):
        if ertek >= 0:
            self._sebesseg = ertek
        else:
            print("A sebesség nem lehet negatív!")
```

‘property’ függvény

- A property függvény egy beépített függvény, amely gettereket és settereket rendel hozzá egy attribútumhoz, így annak direkt elérést korlátozza
- Így csak a gettereken és settereken keresztül érhető elő

```
class Auto:
    new *
    def __init__(self):
        self._sebesseg = 0

    1 usage  new *
    def get_sebesseg(self):
        print("Ez a getter")
        return self._sebesseg

    1 usage  new *
    def set_sebesseg(self, ertek):
        if ertek >= 0:
            self._sebesseg = ertek
        else:
            print("A sebesség nem lehet negatív!")

    sebesseg = property(get_sebesseg, set_sebesseg)
```

```python
class Auto:
    new *
    def __init__(self):
        self._sebesseg = 0

    3 usages  new *
    @property
    def sebesseg(self):
        return self._sebesseg

    1 usage  new *
    @sebesseg.setter
    def sebesseg(self, ertek):
        if ertek >= 0:
            self._sebesseg = ertek
        else:
            print("A sebesség nem lehet negatív!")
```

```python
class Auto:
    new *
    def __init__(self):
        self._sebesseg = 0

    1 usage  new *
    def get_sebesseg(self):
        print("Ez a getter")
        return self._sebesseg

    1 usage  new *
    def set_sebesseg(self, ertek):
        if ertek >= 0:
            self._sebesseg = ertek
        else:
            print("A sebesség nem lehet negatív!")

    sebesseg = property(get_sebesseg, set_sebesseg)
```

‘property’ decorator

- A property dekorátort attribútumok könnyebb kezelésére használjuk, anélkül, hogy közvetlenül változtatnánk az attribútum értékén.
- @property dekorátor elhelyzése a getter elé
- @propertyneve.setter elhelyzése a setter elé

```
class Auto:
    new *
    def __init__(self):
        self._sebesseg = 0

    3 usages  new *
    @property
    def sebesseg(self):
        return self._sebesseg

    1 usage  new *
    @sebesseg.setter
    def sebesseg(self, ertek):
        if ertek >= 0:
            self._sebesseg = ertek
        else:
            print("A sebesség nem lehet negatív!")
```

Miért használjuk a property-t?

- Az attribútumok közvetlen elérésének korlátozása és ellenőrzése
- A kód olvashatóbbá és karbantarthatóbbá tétele.
- Az implementációs részletek elrejtése a felhasználó elől, miközben az interface változatlan marad.

Feladat

- Készíts egy Student osztályt aminek 2 publikus attribútuma név, azonosító és egy non-public score néven.
- írjuk getter settert a score attribútumra, a property-k segítségével
- írjuk egy metódust ami megadja hogy az adott tanulónk teljesített-e a félévet vagy sem (score > 60, akkor igen)
- egy main fájlban próbáljuk ki a fent elkészített osztályt

Type Hinting és Visszatérési Érték Jelölés Pythonban

- A Type Hinting segít meghatározni az adattípusokat a függvények és metódusok paramétereinél és visszatérési értékeinél.
- Javítja a kód olvashatóságát és segíti a fejlesztői eszközöket a jobb kódanalízisben.
- A Python 3.5-től kezdődően használhatók a type hintek.
- A -> szimbólum használatával jelezhetjük a visszatérési érték típusát.
- A type hinting nem befolyásolja a kód futását, csak tájékoztatásul szolgál.

```
def osszead(a: int, b: int) -> int:
    return a + b
```

Feladat

Csináljunk egy rendszert, ami a online képalbumokat kezel.

- Legyenek felhasználóink, akiknek van többek között jelszavuk is van, és ezt lehet ellenőrizni (non-public)
- A képek legyenek albumokba rendezve. Az albumokba a képeket az album(!) tulajdonosa tudja berakni vagy kivenni
- A képeknek legyen méretük, formátumuk, nevük, de ezekből a méretet és formátumot csak a kép(!) tulajdonosa tudja módosítani.

GÁBOR DÉNES EGYETEM

Python Öröklődés és Polimorfizmus

Öröklődés - Bevezetés

- Az öröklődés lehetővé teszi az osztályok attributumainak és metódusainak újrahasználását.
- Az öröklődéssel a származtatott osztályok bővíthetik vagy módosíthatják a szülő osztályok funkcióit. (Polimorfizmus)
- Egyszerű Öröklődés esetén a származtatott osztály örökli a szülő osztály összes attribútumát és metódusát.

```
class Allat:
    new *
    def __init__(self, nev):
        self.nev = nev

    new *
    def hangot_ad(self):
        pass

    1 usage  new *
    def mi_a_neved(self):
        print(self.nev)

1 usage  new *
class Kutya(Allat):
    1 usage  new *
    def hangot_ad(self):
        print("Vau")

bodri = Kutya('Bodri')

bodri.hangot_ad()
bodri.mi_a_neved()
```

Konstruktorok az Öröklődésben

- A származtatott osztályban felüldefiniálhatók az szülő osztály metódusai ez Metódus Felülírás (Override)
- __init__ metódus is öröklődik, de felülírható a származtatott osztályban
- A super() függvény lehetővé teszi a szülő osztály metódusainak meghívását a származtatott osztályból
- Meghívhatjuk az osztály nevével is a szülő osztályt

```
class Macska(Allat,Negylabu):
    new *
    def __init__(self, nev, eletkor):
        super().__init__(nev)
        self.eletkor = eletkor

tom = Macska( nev: 'Tom', eletkor: 7)
```

Többszörös öröklődés

- Pythonban egy osztály több szülő osztálytól is örökölhet.
- Mindkét osztály metódusait és attribútumait megörökli
- Mindkettő szülőosztály metódusait meghívhatja

```
class Negylabu:
    LABAK_SZAMA = 4

    new *
    def __init__(self):
        print('negylabu')
1 usage   new *
class Macska(Allat,Negylabu):
    new *
    def __init__(self, nev, eletkor):
        Allat.__init__(self,nev)
        Negylabu.__init__(self)
        self.eletkor = eletkor
```

Az Absztrakt Bázisosztályok (ABC) Szerepe

- Az ABC-ket olyan osztályok definiálására használjuk, amelyekből nem hozhatók létre közvetlenül példányok.
- A céljuk, hogy közös interfészt biztosítsanak a származtatott osztályok számára.
- Az abc modul ABC osztálya szolgál bázisként az absztrakt osztályokhoz.

```
from abc import ABC

class Jarmu(ABC):
    pass
```

Az ‘abstractmethod’ Dekorátor Használata

- Az abstractmethod dekorátor azt jelzi, hogy egy metódus implementálása kötelező a leszármaztatott osztályokban.
- Egy absztrakt metódussal rendelkező osztály nem példányosítható.

```
from abc import ABC, abstractmethod

class Jarmu(ABC):
    @abstractmethod
    def mozog(self):
        pass
```

Absztrakt Osztályok Öröklése és Implementálása

- Az absztrakt osztályokból származtatott osztályoknak implementálniuk kell minden absztrakt metódust.
- A leszármaztatott osztályok ezt követően példányosíthatóak és használhatóak.

```
class Auto(Jarmu):
    def mozog(self):
        print("Az autó gurul az úton.")

# Most már példányosíthatjuk az Auto osztályt, mert implementálta a mozog metódust.
auto = Auto()
auto.mozog()  # Kimenet: Az autó gurul az úton.
```

Feladat

Tervezz és implementálj egy szálloda szobafoglalási rendszert, amely kezeli a különböző típusú szobákat, mint például egyágyas, kétágyas és lakosztály. A szobáknak legyenek közös tulajdonságai, mint például szobaszám és foglaltsági státusz, valamint egyedi attribútumai is, mint például ár és extrák listája. A rendszer lehetővé teszi a vendégek számára, hogy foglaljanak szobákat, ellenőrizhessék azok elérhetőségét, és megtekinthessék az árakat.

1. Legyen egy adatfeltöltő metódus, ami valamennyi foglalással feltölti az adatokat.
2. A felhasználónak legyen lehetősége kiválasztani, hogy foglalásokat, árakat szeretnének lekérdezni, vagy új foglalást leadni.

GÁBOR DÉNES EGYETEM

Tervezési minták

Miért Fontosak a Design Pattern-ök?

A design pattern-ök, más néven tervezési minták, bevált tervezési megoldások, amelyeket szoftverfejlesztés során alkalmazhatunk. Ezek a minták nem konkrét kódok, hanem problémamegoldási sablonok, amelyek segítségével strukturált és karbantartható kódot hozhatunk létre. A design pattern-ök előnyei közé tartozik, hogy elősegítik a fejlesztői kommunikációt, a kód újrafelhasználhatóságát és segítenek elkerülni a gyakori hibákat. Az ismétlődő problémákra standard megoldásokat kínálnak, megkönnyítve ezzel a fejlesztési folyamatot és növelve a szoftver minőségét.

A Design Pattern-ök Származása és Evolúciója

A tervezési minták fogalma az építészetből ered, ahol Alexander Christopher "A Pattern Language" című művében mutatta be először. A szoftverfejlesztésben a tervezési minták népszerűsítői a "Gang of Four" (GoF), akik a "Design Patterns: Elements of Reusable Object-Oriented Software" című könyvükben 23 alapvető mintát ismertettek. Ezek a minták azóta is a szoftvertervezés alappillérei, amelyeket a fejlesztők széles körben alkalmaznak. A design pattern-ök rendszerezésüknek köszönhetően könnyen tanulhatóak és alkalmazhatóak különböző programozási nyelveken és platformokon.

Hol és Mikor Használjunk Design Pattern-öket?

A design pattern-ök akkor a leghatékonyabbak, amikor olyan gyakori problémákkal szembesülünk, amelyeknek ismert és jól definiált a megoldása. A minták használata jelentősen javíthatja a kód minőségét, lehetővé téve annak könnyebb karbantartását és bővítését. Alkalmazásuk különösen nagy projektekben válik kritikussá, ahol a kódbázis mérete és a fejlesztői csapatok száma is nagyobb. Fontos azonban megjegyezni, hogy nem létezik univerzális megoldás, és a tervezési minták nem alkalmazandóak minden helyzetben; a minták alkalmazását mindig a projekt specifikus körülményei és igényei határozzák meg.

A Tervezési Minták Főbb Kategóriái

A tervezési minták három fő kategóriába sorolhatók, melyek a szoftvertervezés különböző aspektusaira fókuszálnak. Ezek a kategóriák segítenek a fejlesztőknek abban, hogy a megfelelő mintát válasszák ki a felmerülő kihívások és problémák megoldásához.

Létrehozási Minták (Creational Patterns):
Ezek a minták az objektumok létrehozásával foglalkoznak, optimalizálva és kontrollálva azok inicializálást.
Céljuk, hogy rugalmasságot biztosítsanak az objektumok létrehozásában, támogatva a kód bővíthetőségét és karbantarthatóságát.

Szerkezeti Minták (Structural Patterns):
A szerkezeti minták arra koncentrálnak, hogyan alkotnak egységet az objektumok és osztályok.
Segítségükkel hatékonyabban kezelhetővé válik az összetett struktúrák, és javítható az egyes elemek közötti kommunikáció és függőségek.

Viselkedési Minták (Behavioral Patterns):
Ezek a minták az objektumok közötti kommunikációs mintákat és felelősségeket határozzák meg.
Az objektumok közötti együttműködés finomhangolásával a viselkedési minták segítenek a komplex viselkedési folyamatok kezelésében.
Ezek a kategóriák nem csak a tervezési problémák megértését és megoldását segítik, hanem egyúttal a szoftverfejlesztők közös nyelvévé is váltak, lehetővé téve a tapasztalatok és tudás hatékony átadását.

Az Objektumok Világának Megalkotása

A létrehozási minták a szoftvertervezés alapkövei. Ezek a minták arra összpontosítanak, hogy hogyan hozzunk létre objektumokat egy rendszerben. Az egyszerű new operátortól kezdve a bonyolultabb létrehozási folyamatokig, a létrehozási minták célja a létrehozási logika absztrakciója és a felelősségek szeparálása.

Singleton: Biztosítja, hogy egy osztályból csak egy példány létezzen, és egy globális hozzáférési pontot kínáljon ehhez a példányhoz.
Factory Method: Egy interfész definiálásával engedi meg az alosztályoknak, hogy meghatározzák, mely osztályok példányosítandók.
Abstract Factory: Csoportosítja az objektumok létrehozását anélkül, hogy konkretizálná azok osztályait.
Builder: Leegyszerűsíti a komplex objektumok létrehozását, lehetővé téve az objektum részleteinek lépésről lépésre történő konfigurálását.
Ezek a minták segítik a SOLID elvek megtartását, növelik a kód moduláris voltát és elősegítik a komponensek közötti függetlenséget.

Az Objektumok Egységének megteremtése

Amikor az objektumok és osztályok kapcsolatrendszerét kell megtervezni, a szerkezeti minták biztosítják a szükséges kereteket. Ezek a minták az objektumok és osztályok összetett struktúráinak megteremtésére szolgálnak, segítve ezzel a rendszerek skálázhatóságát és rugalmasságát.

Adapter: Lehetővé teszi inkompatibilis interfészek integrációját, összekötve azokat úgy, hogy együtt tudjanak működni.
Decorator: Dinamikusan ad hozzá felelősségeket az objektumokhoz anélkül, hogy megváltoztatná azok szerkezetét.
Facade: Egyszerűsített interfészt biztosít egy összetett rendszerhez, csökkentve ezzel a komplexitást és javítva a használhatóságot.
Composite: Egyetlen objektumként kezeli az objektumok csoportjait, lehetővé téve a hierarchikus struktúrák egyszerű kezelését.

Objektumok Közötti Kommunikáció és Együttműködés

A viselkedési minták az objektumok közötti interakciókat és kommunikációt szabályozzák. Ezek a minták definiálják, hogy az objektumok hogyan és mikor kommunikálhatnak egymással, hogyan oszthatják meg a felelősségeket, és hogyan viselkedhetnek változó körülmények között.

Observer: Egy-egy kapcsolatot hoz létre az objektumok között úgy, hogy egy objektum állapotváltozása esetén az összes függő objektum automatikusan értesítést kap.
Command: Egységbe foglalja a műveleteket objektumokba, lehetővé téve a műveletek paraméterezését, sorba rendezését vagy tárolását.
Strategy: Egy család algoritmust definiál, és azokat cserélhetővé teszi, lehetővé téve az algoritmusok futási időben történő változtatását.
State: Lehetővé teszi egy objektum viselkedésének megváltoztatását, amikor annak belső állapota megváltozik.

A Singleton Minta Alapjai

A Singleton pattern az egyik leggyakrabban használt tervezési minta a létrehozási minták kategóriájában. A Singleton célja, hogy biztosítsa egy osztályból csak egy példány létezzen a program futásának teljes ideje alatt. Ezt úgy éri el, hogy korlátozza az új példányok létrehozását, és egy globálisan elérhető hozzáférési pontot biztosít az egyetlen példányhoz.

A Singleton minta előnyei között szerepel a memóriaoptimalizáció, mivel az osztályból csak egy példány van memóriában, valamint a kontrollált hozzáférés az osztály egyetlen példányához. Gyakori alkalmazási területei közé tartozik az adatbázis kapcsolatok kezelése, a konfigurációs beállítások, vagy a logger funkciók, ahol fontos, hogy az osztály állapota és funkciói egységesen elérhetőek legyenek a rendszer minden részében.

GOF Singleton

- A Singleton tervezési minta az egyik alkalmazás tervezési minta a "Gang of Four" (GOF) könyvből.
- Célja, hogy egyetlen egy példányt hozzon létre egy osztályból és biztosítsa, hogy csak ez az egy példány létezzen az alkalmazásban.
- Alapelvei:
  - Privát konstruktor.
  - Privát statikus változó az egyetlen példány tárolására.
  - Statikus metódus az egyetlen példány visszaadására vagy létrehozására.

Simple Singleton

- Az egyszerű (Simple) Singleton egy egyszerű implementációja a Singleton tervezési mintának Pythonban.
- Az objektum létrehozását szabályozza __new__
- Nem igényel különösebb osztályt vagy metaclass-t.
- Alapelvek:
  - A objektum első importálásakor inicializálódik.

Metaclass Singleton

- A Metaclass Singleton egy továbbfejlesztett megközelítése a Singleton tervezési mintának Pythonban.
- A metaclass a Pythonban egy olyan osztály, amely irányítja az osztályok létrehozását és viselkedését.
- A Singleton metaclass a Singleton osztályokat irányítja, és biztosítja, hogy csak egy példány jöjjön létre belőlük.
- Alapelvek:
  - A metaclass örökli a type metaclass tulajdonságait és viselkedését.
  - A metaclass definiálhatja a példányosítás szabályait, a _call_ metódus felülírásával.

Feladat

- Hozz létre egy Logger singleton osztályt, amely egy egyszerű naplózási rendszert modellez. Az osztálynak csak egy példánya lehet.
- Az osztálynak legyen egy log metódusa, amely egy szöveget kap argumentumként, és hozzáadja azt a naplózási rendszerhez. A naplóüzeneteket tárolja egy listában vagy más adatszerkezetben.
- Hozz létre két különböző függvényt vagy osztályt, amelyek mindegyike használja a Logger osztályt. Példányosítsd a Logger osztályt ezeken a helyeken, és használd a log metódust néhány üzenet rögzítésére.
- Ellenőrizd, hogy mindkét modul ugyanarra az egyetlen példányra hivatkozik-e a Logger osztályból.
- Ellenőrizd, hogy a naplóüzenetek megfelelően rögzítésre kerülnek-e a naplózási rendszerben.

Factory Tervezési Minta

Cél: Az objektum létrehozását szétválasztja annak használatától és típusától. A Factory tervezési minta az objektumokat gyártó osztályokat összpontosítja, és lehetővé teszi az objektumok különböző típusainak kezelését egy közös felületen keresztül.

Miért fontos?

- **Rugalmas kód:** A Factory segít a kód rugalmasságának növelésében, mivel a klienseknek nem kell ismerniük az objektumok konkrét típusait, csak azok általános interfészét.
- **Kód ismétlésének csökkentése:** Az objektumok létrehozásának kódja központosítva van a Factory-ban

Factory Működése

Általános Működés: A Factory tervezési minta során a kliens nem hozza létre az objektumokat közvetlenül, hanem egy "gyár" (factory) osztály felelős a példányosításért. A gyárnak lehet több altípusa, és azokat kliens igényeihez lehet igazítani.

Használati Szituációk:

- Összetett objektumok létrehozása: Amikor az objektumok létrehozása bonyolult vagy összetett konfigurációt igényel, a Factory tervezési minta segít az egységes és szabványos létrehozási folyamat biztosításában.
- A konkrét osztályoktól való függőség csökkentése: A Factory segít a klienseknek az objektumok közötti szoros kapcsolat csökkentésében, mivel csak az objektumok interfészét ismerik, nem pedig a konkrét implementációkat.

Factory Használata

Kliens kéri egy példányt a gyártól, gyár létrehozza és konfigurálja az objektumot, majd visszaadja azt a kliensnek.

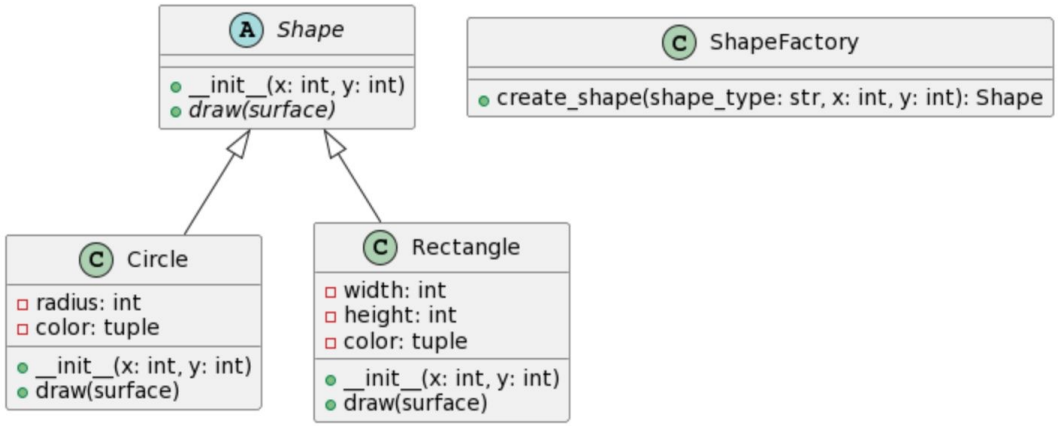
-

Előnyök:

- Kliens és objektumok közötti szoros kapcsolat csökkentése: A kliens csak az objektumok interfészére támaszkodik, és nem kell ismernie a konkrét típusokat.
- Objektumok létrehozásának központosítása: A Factory osztály felelős az objektumok létrehozásáért, így azok kezelése egyszerűsödik.

Példák:

- A Python **'datetime'** modul, amely segítségével dátumokat és időket lehet létrehozni.
- Az **'open'** függvény a fájlok kezeléséhez, amely különböző típusú fájlokat hoz létre a kliens igényeinek megfelelően.



Feladat

- A feladat egy egyszerű autógyártási rendszer létrehozása. Különböző autótípusokat gyártanak, és minden autó egyedi tulajdonságokkal rendelkezik. Használjuk a Factory tervezési mintát az autók létrehozásához.

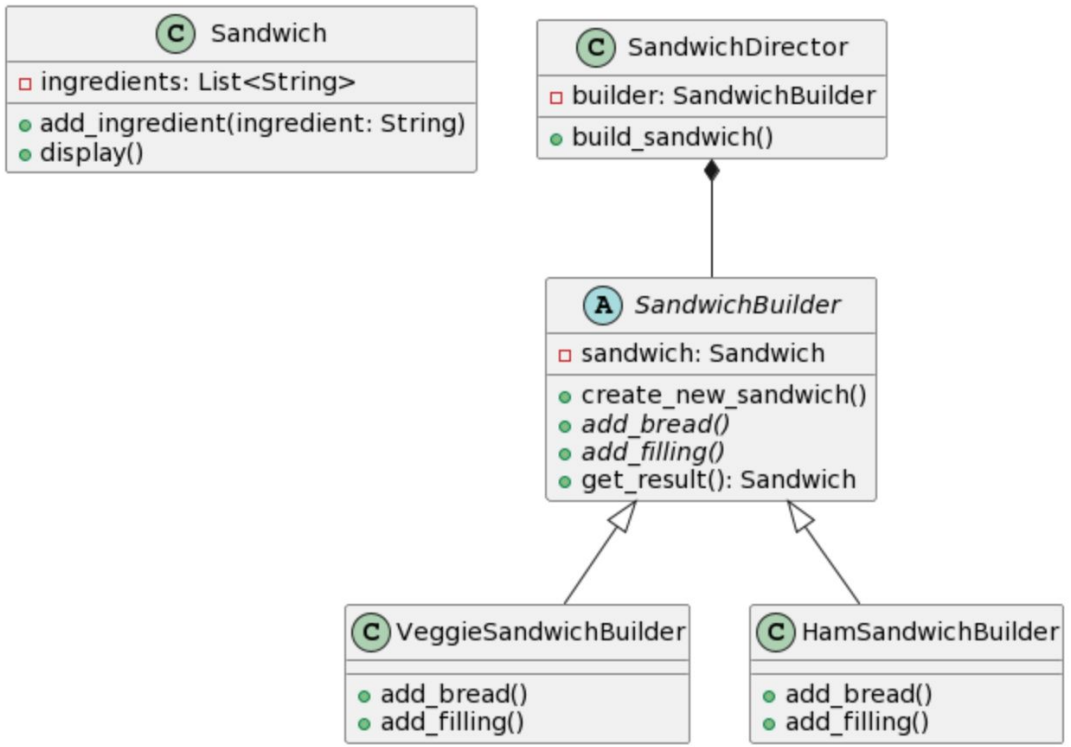
- Készíts egy Car absztrakt osztályt, amelynek van egy assemble metódusa.
- Hozz létre több különböző autótípust, például Sedan, SUV, Truck, amelyek mindegyike az Car osztályból öröklődik, és az assemble metódust megvalósítja.
- Készíts egy CarFactory osztályt, amelynek van egy create_car metódusa, amely egy adott típusú autót hoz létre és ad vissza.
- Készíts egy CarOrder osztályt, amely autórendeléseket képvisel. A CarOrder rendelésnél meg kell adnia, hogy milyen típusú autót szeretne, és opcionálisan további tulajdonságokat.
- Készíts egy CarManufacturer osztályt, amely kezeli az autórendeléseket. A CarManufacturer osztálynak van egy process_order metódusa, amely létrehozza a megfelelő típusú autót a CarFactory segítségével, majd összeszereli azt.
- Készítsen egy példányt a CarManufacturer osztályból, és adj hozzá néhány példányosított autórendelést különböző autótípusokkal és tulajdonságokkal.

Builder Pattern Áttekintése

- A Builder tervezési minta egy tervezési minta, amely lehetővé teszi komplex objektumok létrehozását lépésről lépésre.
- A célja az, hogy elkerülje a túlzott paraméterezést az objektumok létrehozásánál, és hozzon létre egy olyan interfészt, amely lehetővé teszi az objektumok részleteinek konfigurálását.
- A Builder minta fő komponensei a Director, a Builder interfész, a ConcreteBuilder osztályok és a Product osztály.
- A Director felelős a létrehozási folyamat irányításáért, míg a Builder interfész definiálja a lépéseket.
- A ConcreteBuilder osztályok valósítják meg a lépéseket, és az elkészült objektumot adják vissza.
- A Product osztály tartalmazza az elkészült objektumot.

Builder Pattern Használata

- A Builder mintát használva lehetőségünk van létrehozni összetett objektumokat lépésről lépésre anélkül, hogy túlzottan bonyolult konstruktorokat kellene használnunk.
- A kliens először létrehoz egy Builder példányt, majd egy Directorral irányítja a létrehozási folyamatot.
- A Builder objektumok felelősek a részletekkel kapcsolatos munkáért, például az alkatrészek hozzáadásáért és az objektum összeszereléséért.
- A kliens hozzáfér az elkészült objektumhoz a Builderen keresztül.



Builder Pattern Előnyei és Hátrányai

- A Builder minta előnyei közé tartozik, hogy lehetővé teszi a komplex objektumok könnyű és átlátható létrehozását.
- Segít elkerülni a hosszú paraméterlistákat és a túlterhelést a konstruktorokban.
- Növeli a kódban a rugalmasságot és a karbantarthatóságot, mivel könnyen módosíthatók a lépések és az objektumok struktúrája.
- Azonban hátrányai közé tartozik, hogy több osztályra van szükség a minta alkalmazásához, ami növelheti a kódbázis komplexitását.
- Kisebb objektumok létrehozása esetén túlzottan bonyolult lehet a Builder használata.

Feladat

- Készítsünk egy alkalmazást, amely lehetővé teszi a felhasználók számára, hogy egy kávégépet konfiguráljanak saját igényeiknek megfelelően. A kávégépnek két fő összetevője van: víztartály és kávészemtartó. A felhasználók kiválaszthatják a kívánt víztartály méretét és a kávészemtartó méretét.
- A Builder tervezési mintát használjuk a kávégép objektumok létrehozásához. Készítsünk egy CoffeeMachine osztályt, amely a kávégépet reprezentálja. Az CoffeeMachineBuilder absztrakt osztály tartalmazza a kávégép építőjét, amelynek metódusai lehetővé teszik a víztartály és a kávészemtartó méretének beállítását. Készítsünk konkrét építő osztályokat a kávégép konfigurálásához, például SmallCoffeeMachineBuilder és LargeCoffeeMachineBuilder.
- A kávégép konfigurálásához a felhasználó választhatja ki a kívánt méretű víztartályt és kávészemtartót, majd hozzon létre egy kávégépet a kiválasztott konfigurációval. A végén jelenítsük meg a létrehozott kávégép tulajdonságait.

Adapter Pattern

- Az Adapter pattern egy strukturális tervezési minta, amely lehetővé teszi inkompatibilis interfészek együttműködését. Az Adapter minta úgy működik, mint egy híd két különböző interfész vagy osztály között, lehetővé téve, hogy azok zökkenőmentesen kommunikáljanak egymással anélkül, hogy megváltoztatnák azok kódját. Ez a minta hasznos, amikor új komponenseket vagy rendszereket kell integrálnunk meglévő kódunkba, amelyek nem rendelkeznek kompatibilis interfészekkel. Az Adapter minta két formában létezik:
  - **Osztály Adapter:** Öröklődés útján valósítja meg az adaptációt.
  - **Objektum Adapter:** Összetétel (kompozíció) segítségével hoz létre kapcsolatot az inkompatibilis interfészek között.
- Az Adapter minta segít a meglévő rendszerek kiterjesztésében, a felesleges újraírás elkerülésében és a rendszer rugalmasságának növelésében.

Adapter Pattern Alkalmazása

- Az Adapter minta használata során létrehozunk egy új osztályt (az Adaptert), amely implementálja a célosztály interfészét, és összekapcsol egy meglévő osztályt (az Adaptee-t) az új interfésszel. Az Adapter osztály 'fordítja le' a célosztály metódushívásait az Adaptee számára érthetővé, lehetővé téve ezzel a két osztály közötti interakciót. A lépések a következők:
  - Meghatározzuk a célosztály interfészét.
  - Létrehozzuk az Adapter osztályt, amely implementálja ezt az interfészt.
  - Az Adapter osztályban delegáljuk a hívásokat az Adaptee objektum felé.
  - Az ügyfél (client) kód az Adapter objektumon keresztül kommunikál az Adaptee objektummal.

Adapter Pattern Előnyei és Hátrányai

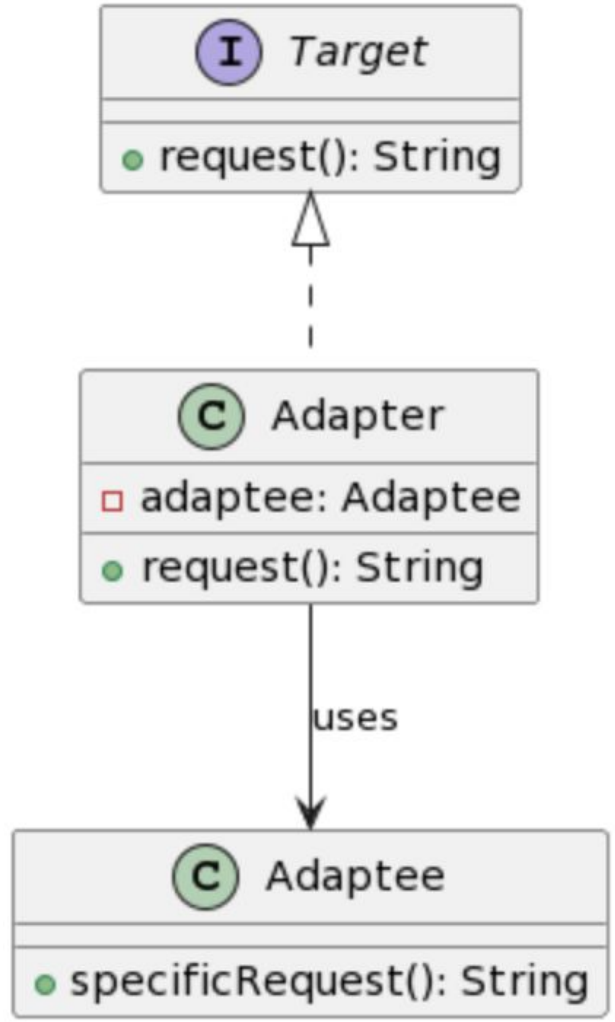
Target: Az interfész, amelyet az ügyfél (client) használni kíván.
Adapter: Az osztály, amely implementálja a Target interfészt, és "adapterként" szolgál az Adaptee és a Target között.
Adaptee: Az osztály, amelynek meglévő funkcionalitását az Adapter osztályon keresztül tesszük elérhetővé a Target interfész számára.

Előnyök:

- Növeli a Kód Újrafelhasználhatóságát: Lehetővé teszi a már létező osztályok újrafelhasználását anélkül, hogy módosítani kellene azokat.
- Csökkenti a Rendszer kötöttségét: Segít elválasztani a meglévő kód és az új interfészek közötti függőségeket.
- Rugalmasság: Az Adapter minta rugalmas megoldást nyújt inkompatibilis interfészek összekapcsolására.

Hátrányok:

- Bonyolultság: Növelheti a rendszer összetettségét, különösen, ha túl sok adaptert használunk.



Példa (charger.py)

Tegyük fel, hogy van egy meglévő LightningConnector interfészünk, amelyet az Apple eszközök használnak, és szeretnénk egy Android készüléket (amely MicroUSB csatlakozót használ) csatlakoztatni egy Apple töltőhöz.

- Először definiáljuk a két interfészt (Lightning és MicroUSB), majd létrehozunk egy Adapter osztályt, amely lehetővé teszi, hogy a MicroUSB-s eszköz Lightning csatlakozóként viselkedjen.

- Target (Cél) interfész: Ez a LightningConnector interfész. Ez az a cél interfész, amit az ügyfél (a példában az AppleCharger) elvár használni.

- Adapter (Átalakító) osztály: Az MicroUSBToLightningAdapter az adapter osztály. Ez az osztály felelős azért, hogy összekapcsolja a Target interfészt (LightningConnector) az Adaptee-vel (AndroidDevice). Ezáltal az adapter teszi lehetővé, hogy az Apple töltő (ami a LightningConnector interfészt használja) tölthessen egy Android eszközt.

- Adaptee (Adaptálandó) osztály: Az AndroidDevice osztály az adaptee. Ez az osztály rendelkezik a meglévő, de inkompatibilis interfésszel (MicroUSBConnector), amit az Adapter osztály hoz összhangba a Target interfésszel.

Feladat

Készíts egy zenelejátszó alkalmazást, amely képes különböző audio formátumok lejátszására. A rendszer alapvetően csak WAV formátumú fájlokat képes lejátszani, de szeretnéd bővíteni a funkcionalitását MP3 és FLAC formátumok támogatására is. Azonban a meglévő zenelejátszó osztályod nem támogatja közvetlenül ezeket a formátumokat. Az Adapter pattern segítségével oldd meg ezt a problémát.

- AudioPlayer Interfész Létrehozása: Hozz létre egy AudioPlayer interfészt, ami deklarál egy play metódust, ami elfogad egy fájlnevet és egy formátumot.
- WAVPlayer Implementálása: Implementáld az AudioPlayer interfészt egy WAVPlayer osztályban, ami kezeli a WAV formátumú fájlokat.
- MP3 és FLAC Lejátszók Implementálása: Hozz létre két osztályt, MP3Player és FLACPlayer, amelyek speciálisan kezelik az adott audio formátumokat.
- AudioAdapter Osztály Implementálása: Készíts egy AudioAdapter osztályt, ami implementálja az AudioPlayer interfészt és képes integrálni a MP3Player és FLACPlayer funkcióit.

Az Observer Pattern Alapjai

Az Observer pattern egy viselkedési tervezési minta, amelyet eseménykezelésre és állapotváltozások figyelésére használnak. Ez a minta lehetővé teszi, hogy egy objektum (az "Subject") automatikusan értesítse az összes regisztrált megfigyelőjét (az "Observers"), amikor változás következik be az állapotában. Az Observer pattern kulcsfontosságú szerepet játszik olyan rendszerekben, ahol az objektumok közötti laza kapcsolat és a dinamikus interakciók fontosak.

A minta két fő részből áll:

- Subject: Az objektum, amelynek állapotát figyelni szeretnénk. Ez tartalmazza a megfigyelők listáját és értesíti őket, amikor változás következik be.
- Observer: Az interfész, amelyet minden olyan objektumnak meg kell valósítania, amely értesítéseket kíván fogadni a Subject változásairól.
- Az Observer pattern segítségével csökkenthetők a komponensek közötti függőségek, növelve ezzel a rendszer moduláris voltát és rugalmasságát.

Hogyan Működik az Observer Pattern?

Az Observer pattern implementálásának lépései:

Subject Osztály:

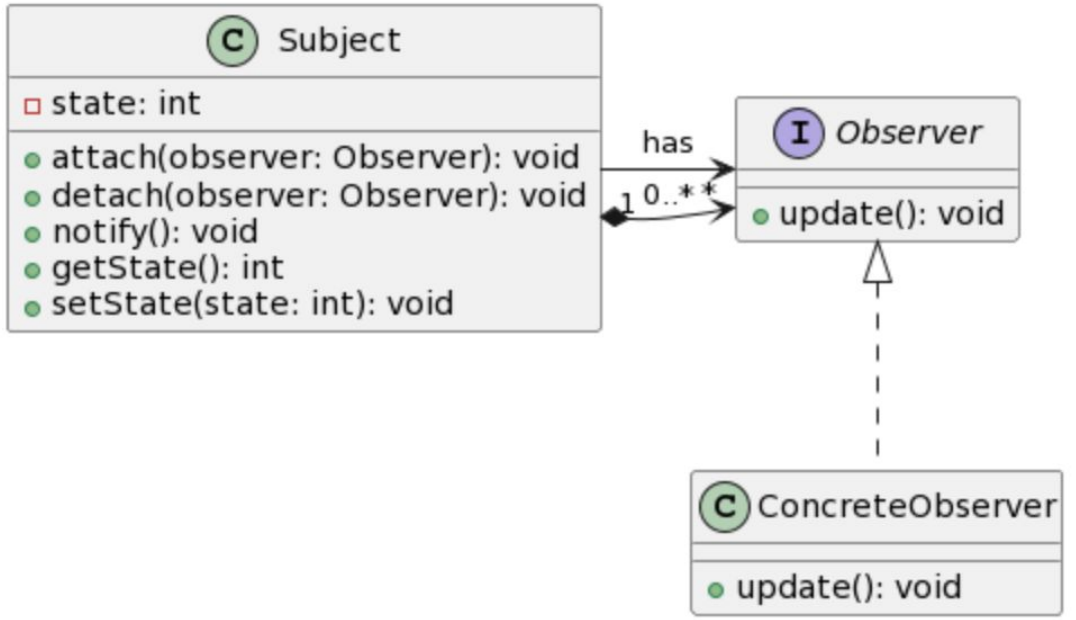
- Hozz létre egy Subject osztályt, amely kezeli a megfigyelők listáját és biztosítja azok regisztrálásának, eltávolításának és értesítésének lehetőségét.
- Amikor változás következik be a Subject állapotában, automatikusan értesíti az összes megfigyelőjét.

Observer Interfész:

- Definiálj egy Observer interfészt vagy absztrakt osztályt, amely meghatározza az értesítésre reagáló metódust (pl. update).
- Minden megfigyelő osztálynak meg kell valósítania ezt az interfészt.

Concrete Observers:

- Hozz létre konkrét megfigyelő osztályokat, amelyek megvalósítják az Observer interfész metódusait.
- Ezek az osztályok meghatározzák, hogyan reagálnak a Subject állapotának változásaira.



Az Observer Pattern a Gyakorlatban

Előnyök:

- Csökkenti az összefüggőséget: A Subject és Observers közötti laza kapcsolat javítja a kód karbantarthatóságát és bővíthetőségét.
- Dinamikus Interakció: Lehetővé teszi az objektumok közötti kommunikációt anélkül, hogy pontosan ismerniük kellene egymást.
- Rugalmasság: Könnyedén hozzáadhatunk vagy eltávolíthatunk megfigyelőket a rendszerből változások esetén.

Alkalmazási Területek:

- Eseménykezelő Rendszerek: GUI eseményfigyelők, felhasználói interakciók kezelése.
- Állapotfigyelés: Olyan rendszerek, ahol fontos az objektumok állapotának követése és az azokra adott válaszok.
- Publikáló-Feliratkozó Modellek: Hírlevél rendszerek, esemény értesítések.

Az Observer pattern egy univerzális tervezési eszköz, amely számos modern szoftverarchitektúrában megtalálható, kihasználva annak rugalmasságát és dinamikus kommunikációs képességeit.

Feladat

Készíts egy egyszerű hőmérséklet figyelő rendszert, amely az Observer pattern-t használja. A rendszerben legyen egy HőmérsékletMérő (Thermometer) osztály, amely a tárgy (subject), és több Figyelő (Observer) osztály, amelyek különböző módokon reagálnak a hőmérséklet változására. Lehetnek az observerek hírügynökségek vagy átlaghőmérsékeletet számoló időjárásjelentések.

GÁBOR DÉNES EGYETEM

KÖSZÖNJÜK A FIGYELMET!

www.gde.hu