2. előadás

Python és az OO 2

Programozás (2) előadás

2022. Szeptember 19.

Halász Gábor

Debreceni Egyetem

Általános tudnivalók

Ajánlott irodalom:

- Nyékyné G. Judit (szerk): Programozási nyelvek, Kiskapu, 2003.
- Juhász, István: Magas szintű programozási nyelvek 2, elektronikus egyetemi jegyzet, 2009
- Tarczali, Tünde: UML diagramok a gyakorlatban, Typotex Kiadó, 2011.
- Angster, Erzsébet: Objektumorientált tervezés és programozás: JAVA, 4KÖR Bt., 2002, ISBN: 9632165136
- Bird, S., Klein, E., Loper, E.: Natural Language Processing with Python, O'Reilly Media, 2009

Félév teljesítésének feltételei: jelenlét + 2 gyakorlati + 1 elméleti ZH

Érdemjegy: $1 < 60\% \leq 2 < 70\% \leq 3 < 80\% \leq 4 < 90\% \leq 5$

További részletek: `https://elearning.unideb.hu/`

Objektumorientáltság és a Pyton

OO és a Python

- Python-ban MINDEN objektum
  - egészek
  - stringek
  - szótárak
  - ...
- Ezek az objeltumok
  - beépített vagy
  - felhasználó által definiált

  osztályokból jönnek létre
  - példányosítással.

Python osztályok

- A programban bárhol definiálhatóak
- Alapértelmezés szerint
  - minden attribútum és
  - minden metódus

  publikus

(Igazából a python csak kevés támogatást ad a privát azonosítók használatához.)

Egy példa

```
class MyClass:
    def set(self, value):
        self.value = value
    def display(self):
        print(self.value)
```

- `MyClass` osztálynak van két metódusa:
  - `set`
  - `display`
- és egy attribútuma
  - `value`

Egy példa

- Minden metódus első paramétere a (majdan példányosítással létrejövő) objektum maga.
- Ennek tradícionális neve `self`, de bármi más is lehetne. Ennek megfelelően az alábbi két megoldás teljesen ekvivalens egymással:
  - ```
    def set(self, value):
        self.value = value
    ```
  - ```
    def set(en, value):
        en.value = value
    ```
- Amikor meghívjuk a metódust, akkor ezt az első paramétert nem kell megadnunk.
  - (Igaz ez?)

```
>>> class MyClass:
...     def set(self, value):
...         self.value = value
...     def display(self):
...         print(self.value)
...
>>> y = MyClass()
>>> y.set(4)
>>> y.display()
4
>>> y.value
4
```


Konstruktor

Python-ban az `__init__` nevű metódus lesz az osztály konstruktora:

```
def set(self, value):
```

helyett/mellett használhatjuk a

```
def __init__(self, value):
```

metódust

Konstruktor

```
def __inidef __init__(self):
    self.value = 0
```

vagy

```
def __inidef __init__(self,
value=0):
    self.value = value
```

Használatuk (második változat esetén):

- `x = MyClass()` $\Rightarrow$ `x.value = 0`
- `y = MyClass(5)` $\Rightarrow$ `y.value = 5`

Példa konstruktorra

```
>>> class Complex:
...     def __init__(self,
                     realp, imagp):
...         self.r = realp
...         self.i = imagp
...
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

Attribútumok

- Attribútmokat ugyanúgy értékadásokkal hozunk létre, mint változókat.
  - Nem kell deklarálnunk őket

```
def set(self, value):
    self.value = value
```

- Létrehozhatjuk őket az osztályban,
  - ezeket ismerni fogja minden példánya és minden alosztálya (és azok példányai) az adott osztálynak,
- vagy annak egy példányában
  - ilyenkor csak az adott objektumban hivatkozható az adott attribútum

```
>>> y = MyClass(5)
>>> y.uj_attr = 9
```


Példányosítás

- A példányosításhoz nincs szükségünk új operátorra
- Egyszerűen az osztály nevét használjuk függvényként hivatkozva rá

```
y = MyClass(5)
```

- Valahányszor egy osztályt „meghívunk”, létrejön egy új példánya
  - Ha van az osztálynak konstruktora, akkor azt is használhatjuk
- A példányok rendelkeznek az osztály összes attribútumával és metódusával

Példányok attribútumai

- Pythonban értékadással hozunk létre új változókat.
- Objektumokban ugyanígy tudunk új attribútumokat létrehozni

  `y.uj_attr = 9`

  új attribútumot hoz létre a `y` példányban, de az `x` példányban továbbra sem lesz `uj_attr` nevű változó

Az információrejtés pythonban

- Alapvetően: nincs ilyen
- Alapértelmezettem minden publikus

Öröklődés

```
class DerivedClassName(BaseClassName):
    <statement-1>
    .  .  .
    <statement-N>
```

- Ha hivatkozáskor egy attribútumot nem sikerül megtalálni az alosztályban, akkor a keresés folytatódik a szuperosztályban
- és folytatódik rekurzívan, ha az is alosztálya valaminek.

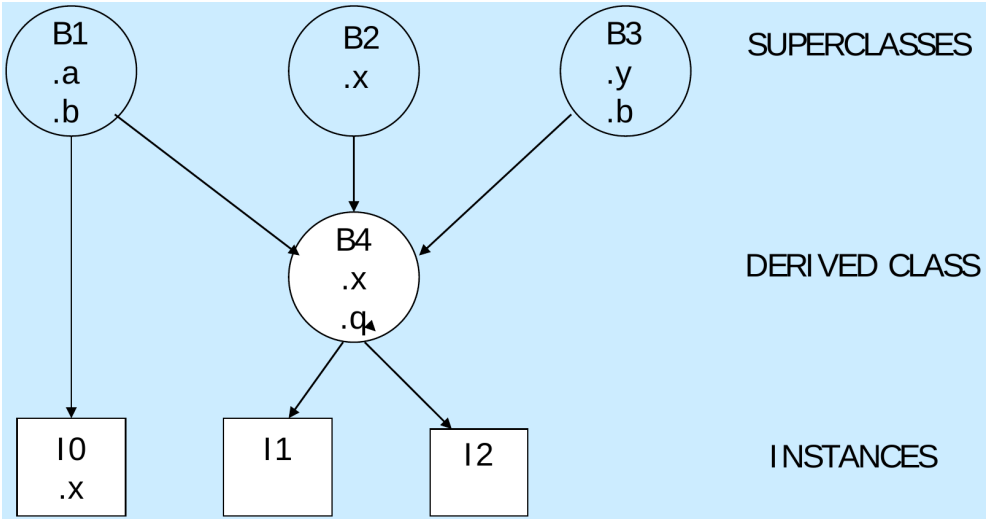
Többszörös öröklődés

A python támogatja a többszörös öröklődést:

```
class DerivedClassName(B1, B2, B3):
        <statement-1>
        .  .  .
        <statement-N>
```

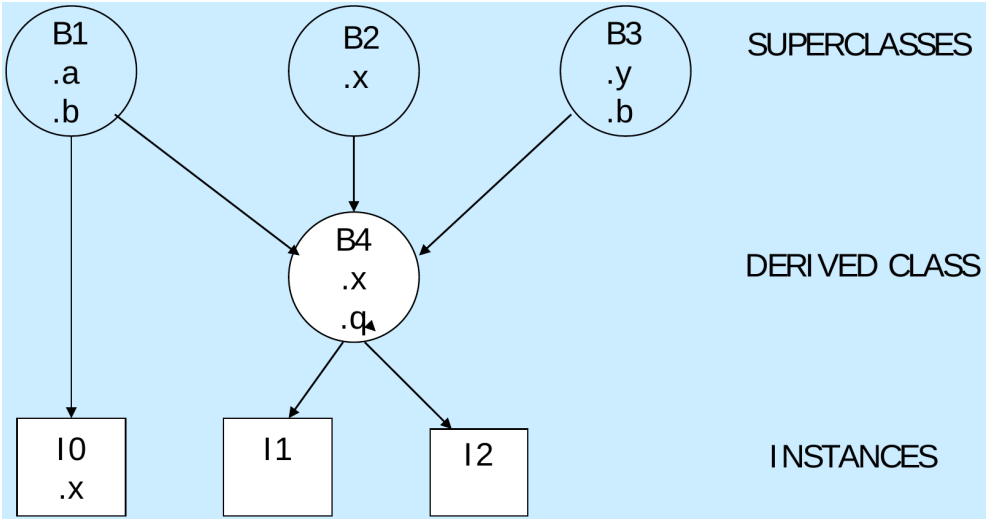

Többszörös öröklődés

- Konfliktusok feloldása:
- Depth-first, left-to-right search
- Mélységben kezdünk
  - példány, osztály, szuperosztály, ...
- Balról jobbra haladunk
  - ha egy osztályt többszörös öröklődéssel hoztunk létre, akkor a deklaráció sorrendjében (balról jobbra haladva) keressük a hivatkozott attribútumot



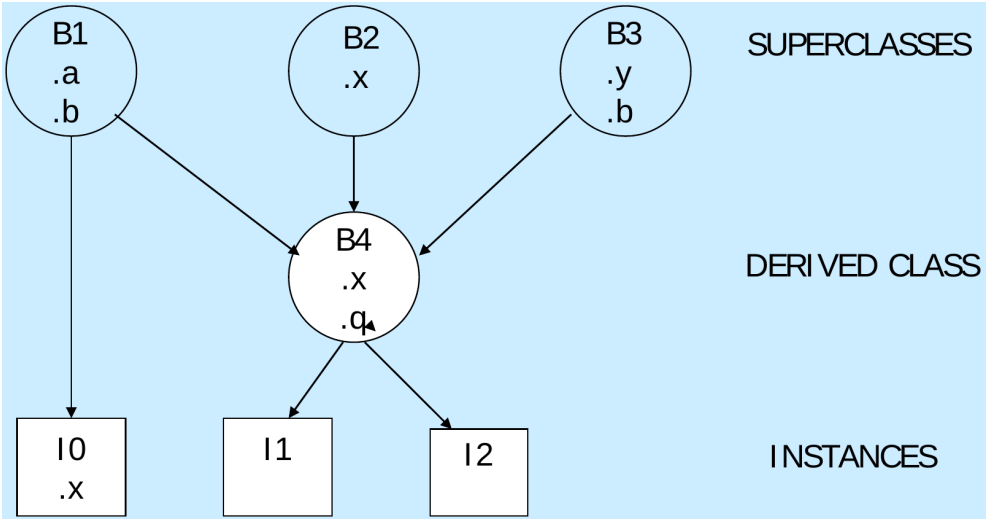
`I0.a` és `I0.b` a `B1`-ben vannak definiálva;

`I0.x` viszont `I0`-ban



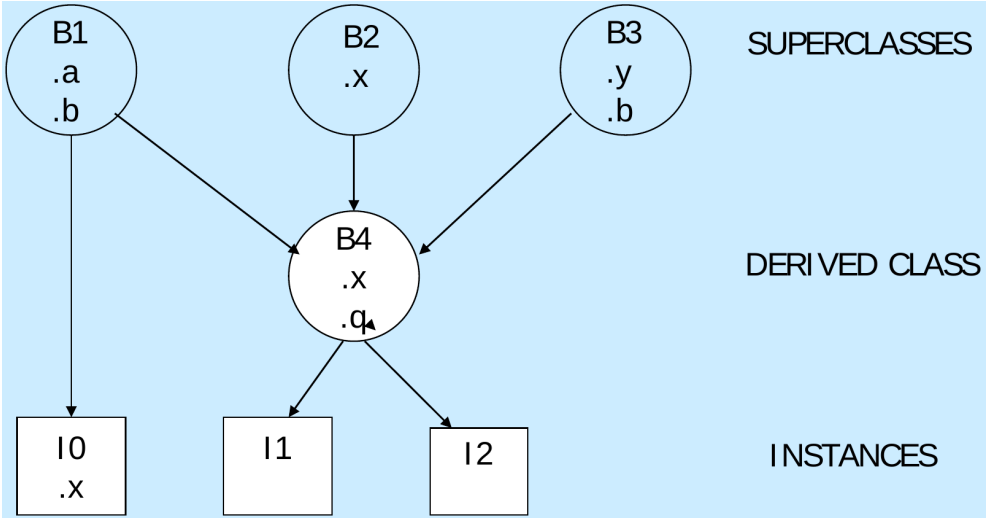
`I1.a`, `I1.b`, `I2.a` és `I2.b`
a `B1`-ben vannak definiálva;

`I1.x` és `I2.x` viszont `B4`-ben



`I0.y`, `I1.y` és `I2.y` a `B3`-ban vannak definiálva;

és így tovább



Hozzáférhetünk-e valahogyan (írás vagy olvasás céljából) az `I0` objektum `B2` vagy `B4` osztályokban definiált `x` ill. a `B3`-ban definiált `b` attribútumához?

Típus – osztály

Nagy hasonlóság van imperatív nyelvek típus fogalma és a python osztály között

- Már láttunk néhány típust: lista, szótár, sztring, ...
- ezek különféle adatok reprezentálására alkalmasak
- az adatok manipulálására vonatkozó különböző kéréseket tudják kezelni

Objektum orientáltság

- Mi is ez?
- És miért törődjünk vele?

Objektum orientáltság

- Röviden:
  - Az objektum orientált programozás egy „módja” annak, hogyan gondolkodunk a programban előforduló „dolgokról” – azaz objektumokról
    - változók
    - függvények
    - ...
  - A program kevésbés lesz
    - utasítások sorozata

    inkább szól
    - az objektumokról, amik köcsonhatnak egymással

Objektum orientáltság

- A programunk kölcsönható objektumok összessége, amik reagálnak a kapott „üzenetekre”
- Az objektumok üzenetekkel való együttműködése egyfajta magas szintű leírása annak, amit a „program csinál”

Objektum orientáltság

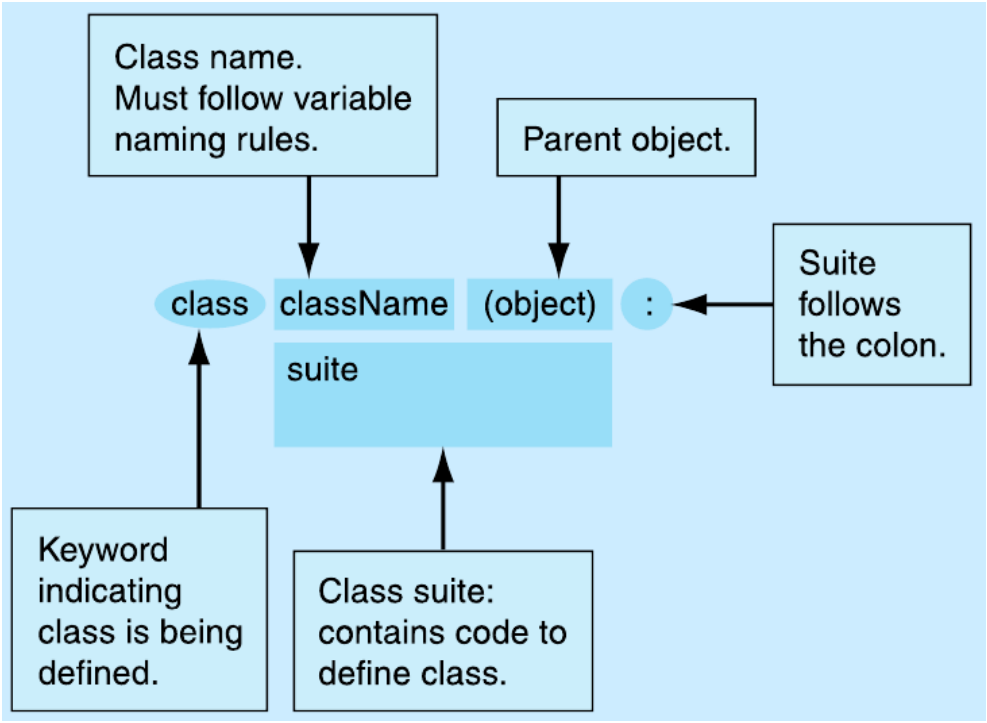
- Az objektum orientáltság segíti a szoftver fejlesztést,
- azaz a kódok hosszútávú használatát
  - egyszerűsíti a kódot, könnyíti a megértését
  - mindez nem érinti a funkcionalitást, csak a „formákat”

Típus – osztály

- Azért használunk osztályokat, hogy összetettebb, feladat-specifikus új adattípusokat – és azok példányait – tudjuk létrehozni
- Minden osztálynak alapvető két aspektusa:
  - adatok (attribútumok)
  - és az üzenetek, amiket kezelni tud (metódusok)


```
class Student(object):
    """Simple Student class."""
    def __init__(self,first='', last='', id=0): # initializer
        self.first_name_str = first
        self.last_name_str = last
        self.id_int = id

    def __str__(self):  # string representation, e.g. for printing
        return "{} {}, ID:{}".format\
               (self.first_name_str, self.last_name_str, self.id_int)
```


Class name.
Must follow variable naming rules.
Parent object.
class
className
(object)
:
Suite follows the colon.
suite
Keyword indicating class is being defined.
Class suite: contains code to define class.

a `dir()` metódus

A `dir()` metódus kilistázza az adott osztály/objektum attribútumait.

- Úgy gondolhatunk rá, mintha ezek egy, az osztályban tárolt szótár kulcsai lennének

```
>>> dir(MyClass)
['__class__', '__delattr__', '__dict__', '__dir__',
'__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__gt__', '__hash__',
'__init__', '__init_subclass__', '__le__', '__lt__',
'__module__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__',
'__weakref__', 'display', 'set']
```

a `pass` kulcsszó

A `pass` kulcsszót arra használjuk, hogy jelezzük, ha valamit szándékosan „hagyunk ki”

- Ha egy osztály definíciójában használjuk a pass kulcsszót, csak azoka dolgok kerülnek bele az osztályba, amiket a python alapból létrehoz

```
>>> class MyClass (object):
        pass

>>> dir(MyClass)
'__class__', '__delattr__', '__dict__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__gt__', '__hash__',
'__init__', '__le__', '__lt__', '__module__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', '__weakref__']

>>> my_instance = MyClass()
>>> dir(my_instance)
['__class__', '__delattr__', '__dict__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__gt__', '__hash__',
'__init__', '__le__', '__lt__', '__module__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', '__weakref__']

>>> type(my_instance)
<class '__main__.MyClass'>
```

Terminológia

- **Objektum alapú nyelvek:** (object-based) ha a nyelvben van objektum fogalom és bezárás, de nincs osztály és öröklés. (Pl. Ada)
- **Osztály alapú nyelvek:** (class-based) van osztály, bezárás, objektum fogalom, de nincs öröklődés. (Pl.: CLU)
- **Objektum-orientált nyelvek:** (object-orinted) minden létezik: bezárás, osztály, öröklődés fogalom. Ezek a nyelvek (imperatív nyelvként) fordítóprogramosak.
- És végül létezik az OO-nak egy olyan speciális nyelve, amelyben nincs osztály fogalom, de minden más OO eszköz megvan benne.