Programovanie v jazyku Python

Princípy OOP, metametódy, vzťahy medzi triedami
prednáška 7

Katedra kybernetiky a umelej inteligencie
Technická univerzita v Košiciach
Ing. Ján Magyar, PhD.

Princípy OOP

1. abstrakcia
2. enkapsulácia
3. dedenie
4. polymorfizmus

Abstrakcia

- pomocou abstrakcie skryjeme implementačné detaily o funkcionalite
- každá trieda by mala poskytovať iba API - súbor metód pre prácu s vnútornými premennými
- cieľom je, aby ďalšie objekty a programátor sa nezaoberali tým, ako presne funguje daná metóda, ale iba tým, čo robí
- pre vysokú mieru abstrakcie je nevyhnutná správna forma dokumentácie, najmä ak metóda má vedľajšie účinky

Abstrakcia v Pythone

1. ak používame triedy, tak neriešime, ako implementujú funkcionalitu
   - spoliehame sa na dokumentáciu a na autora daného kódu
2. keďže trieda definuje abstraktný dátový typ, nezaoberáme sa ani vnútornou reprezentáciou údajov
   - napr. hašovacia tabuľka môže byť reprezentovaná ako slovník alebo ako zoznam zoznamov
   - vnímame iba vonkajší kontext triedy a nie vnútorné detaily

Abstrakcia - ukážka

Pre reprezentáciu hašovacej tabuľky používame dictionary. Vzhľadom na to, že hašovacia funkcia vráti zvyšok po delení 6, možné kľúče sú 0, 1, 2, 3, 4, 5.

Upravíme triedu tak, že reprezentáciu vymeníme na zoznam zoznamov:
- pomocou prvého indexu vyberieme zoznam pod daným kľúčom
- jednotlivé hodnoty budú uložené pod druhým indexom vo vybranom zozname

Enkapsulácia

- kým abstrakcia skryje implementačné detaily, cieľom enkapsulácie je skryť vnútorný stav objektu
- vnútorný stav je definovaný ako súbor hodnôt vo vnútorných premenných
- enkapsulácia definuje spôsob, ako môžeme narábať s objektom z danej triedy

Enkapsulácia v Jave/C# vs. v Pythone

- objektovo orientované jazyky založené na C podporujú enkapsuláciu implicitne
- pre každý atribút vieme definovať viditeľnosť cez kľúčové slová: `public, private, protected, package-private`
- Python tieto kľúčové slová nemá, nemá ani balíky, programátor musí implementovať enkapsuláciu explicitne
- best practice: v rámci triedy pristupovať k vnútorným premenným priamo, mimo triedy cez pomocné metódy

Privátne atribúty v Pythone

- v triede môžeme zadefinovať atribúty ako privátne pomocou znakov __

```
class Product:
    def __init__(self):
        self.__price = 1000

t = Product()
t.__price = 500
```

Stav objektu

- stav objektu je súhrn hodnôt všetkých premenných objektu
- dva základné typy premenných:
  - členské premenné – jedinečné pre objekty
  - premenné triedy – jedinečné pre triedu
- stav objektu sa počas behu programu mení

Premenné tried

- premenné typu `self.name` sú špecifické pre každú inštanciu
- Python umožňuje používanie premenných, ktoré zdieľajú všetky inštancie danej triedy
- definujeme ich mimo konštruktora
- najčastejšie aplikácie
  - jedinečný index (práca s databázami)
  - počítadlo
  - pomocná premenná pre niektoré návrhové vzory

Dedenie

- dedenie znamená, že trieda “dedí” časť funkcionality od inej triedy
- trieda, ktorá dedí je **podtrieda**; trieda, od ktorej dedí je **nadtrieda**
- pomocou dedenia vieme vytvoriť hierarchiu, kde vytvoríme stále bližšiu špecifikáciu tried - podtriedy budú presnejšie definované verzie nadtriedy
- v Python 2 bolo možné definovať triedy bez nadtried, v Python 3 defaultná nadtrieda je `object`
- Python podporuje viacnásobné dedenie

Definícia dedenia v Pythone

```
class Person:
    def __init__(self, name):
        self.name = name


class Student(Person):
    def __init__(self, name, year):
        self.name = name
        self.year = year
```

Volanie metód nadtriedy

- pre podporu polymorfizmu a efektívnu prácu s triedami by sme mali použiť čo najviac už definovanú funkcionalitu v nadtriedach
- kľúčové slovo `super()`

```
class Person:
    def __init__(self, name):
        print('setting name in Person')
        self.name = name

class Student(Person):
    def __init__(self, name, year):
        super().__init__(name)
        self.year = year

janko_hrasko = Student("Janko Hrasko", 1)
```

Viacnásobné dedenie v Pythone

```
class Person: pass
class Worker(Person): pass
class Boss(Person): pass

class Manager(Worker, Boss): pass
class Secretary(Worker): pass
class BoardMember(Boss): pass

class CEO(Manager, BoardMember): pass
```

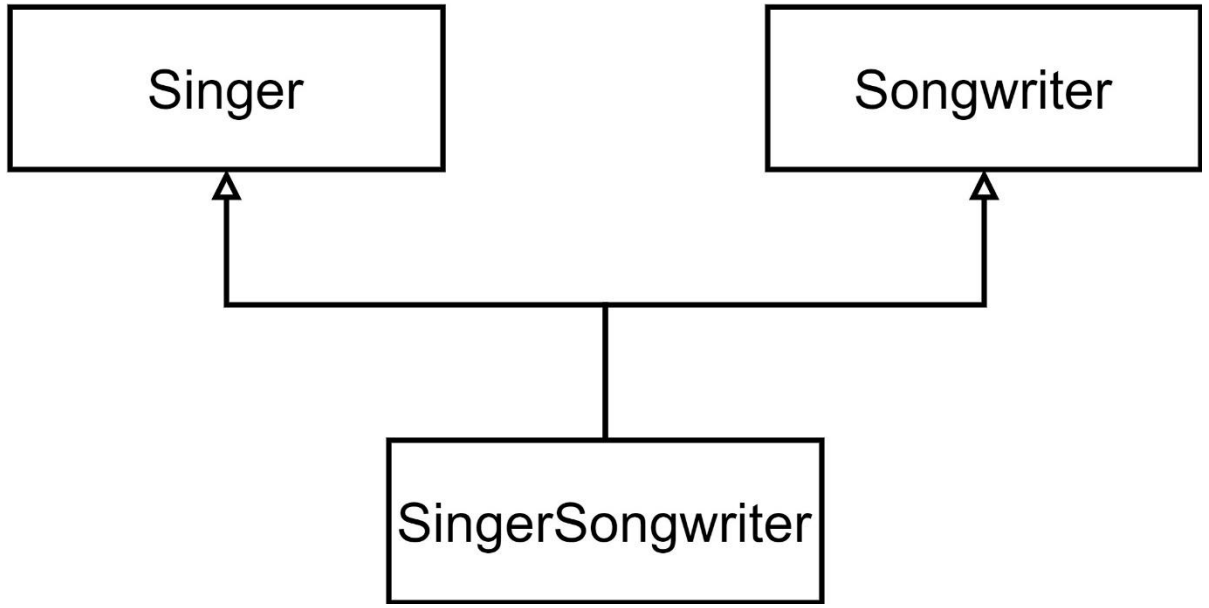
Problém viacnásobného dedenia

v triede `Singer`:

```
sings = True
writes = False
```

v triede `Songwriter`:

```
sings = False
writes = True
```

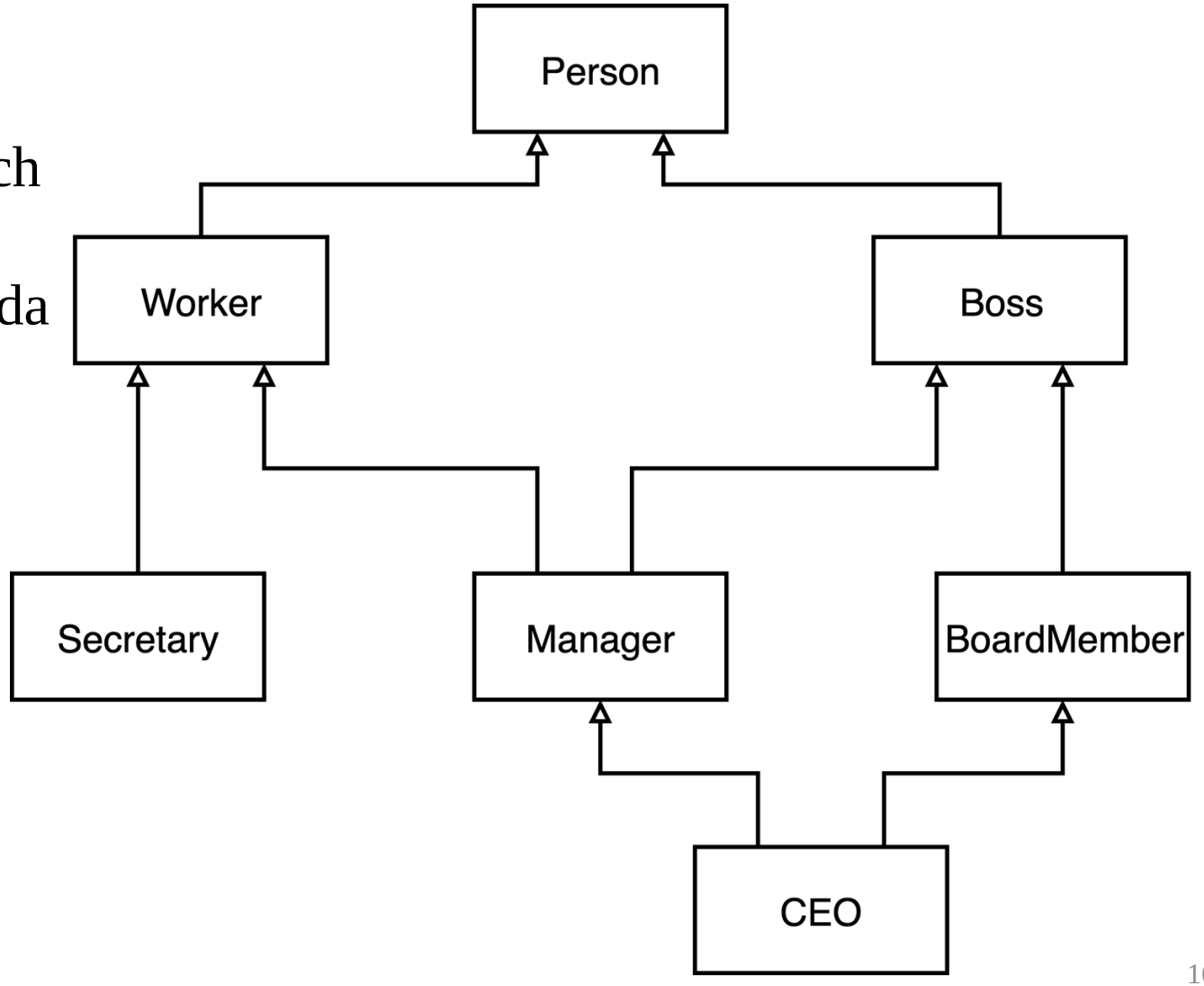


Aké hodnoty zdedí trieda `SingerSongwriter`?

Name resolution pri viacnásobnom dedení

1. vyhľadávanie v aktuálnej triede
2. vyhľadávanie do hĺbky v nadtriedach a zľava doprava
3. ako posledná sa berie do úvahy trieda `object`

- poradie vieme zistiť pomocou premennej `__mro__` alebo metódou `mro()`



Polymorfizmus

- polymorfizmus nám umožňuje použiť rovnaké rozhranie pre rôzne dátové typy, medzi ktorými existuje dedenie
- reálne to znamená, že s objektmi z podtriedy vieme pracovať rovnakým spôsobom, ako s objektmi z nadtriedy
- keďže Python je dynamicky typovaný jazyk, nie je až taký výrazný

Hlavné metametódy tried v Pythone

- `__init__`
- `__str__`
- `__eq__`, `__ne__`, `__lt__`, `__le__`, `__gt__`, `__ge__`
- `__hash__`
- `__del__`
- `dir()`

Konštruktor

- metóda `__init__(self[, …])`
- definuje spôsob vytvorenia inštancie triedy
- zavolá sa po spustení metódy `__new__` a vráti smerník na objekt
- konštruktor každej podtriedy musí zavolať konštruktor nadtriedy

```
SuperClassName.__init__(args)
```

Stringová reprezentácia objektu

- definuje sa v metóde `__str__(self)`
- použije sa pri volaniach
  - `str(my_object)`
  - `.format(my_object)`
  - `print(object)`
- má iba jednu návratovú hodnotu typu `string`

Metódy rovnosti/nerovnosti

- slúžia na porovnávanie hodnôt premenných
- funkcionalita v rôznych metódach

`__eq__`	`__ne__`	`__lt__`	`__le__`	`__gt__`	`__ge__`
`==`	`!=`	`<`	`<=`	`>`	`>=`

- návratová hodnota je zvyčajne `True` alebo `False`, ale môže byť ľubovoľná hodnota

Metóda `__hash__(self)`

- používa sa pri volaní funkcie `hash()`, alebo pri operáciách s hašovanými skupinami hodnôt (`set`, `frozenset`, `dictionary` - kľúč musí byť hašovateľný)
- môže vracať ľubovoľnú hodnotu, jediná podmienka je že ak `object1 == object2`, tak `hash(object1) == hash(object2)`
- implementácia úzko súvisí s metódou `__eq__`: odporúča sa spojiť hodnoty, ktoré sa kontrolujú pri zisťovaní rovnosti dvoch objektov do jednej n-tice, a zavolať funkciu `hash()` nad touto n-ticou
- ak nemáte definovanú metódu `__eq__`, nemali by ste definovať ani `__hash__`; ak máte definovanú `__eq__` ale nie `__hash__`, trieda bude reprezentovať nehašovateľný typ

Finalizer

- definovaný v metóde `__del__(self)`
- zavolá sa po volaní `del(my_object)`
- ak podtrieda definuje `__del__`, musí byť zavolaná metóda `__del__` nadtriedy
- odporúča sa použiť, ak objekt má aktívnu komunikáciu so súborom alebo s databázou - je potrebné uzavrieť tento komunikačný kanál
- v rámci `__del__` je možné vytvoriť nový smerník na aktuálny objekt - resurrection
- chyby, ktoré sa vyskytnú počas vykonávania metódy `__del__` sú ignorované, vypíše sa iba hláška na `sys.stderr`

Funkcia dir()

- slúži na získanie všetkých atribútov daného objektu, resp. triedy
- pre objekt vráti zoznam vnútorných premenných, zoznam vnútorných metód, zoznam atribútov triedy do ktorej objekt patrí, a rekurzívne zoznam atribútov všetkých nadtried
- pre triedu vráti zoznam premenných triedy, zoznam metód triedy, a rekurzívne zoznam atribútov všetkých nadtried
- je možné definovať vlastný spôsob získania zoznamu atribútov v metóde `__dir__` ale zvyčajne to nie je potrebné

Práca s objektmi ako s numerickými typmi

- je možné definovať spôsob, ako narábať s objektmi, ak sú argumentmi primitívnych algebraických operácií

```
+               object.__add__(self, other)
-               object.__sub__(self, other)
*               object.__mul__(self, other)
/               object.__truediv__(self, other)
//              object.__floordiv__(self, other)
%               object.__mod__(self, other)
divmod()        object.__divmod__(self, other)
**              object.__pow__(self, other)
```

Práca s objektmi ako s numerickými typmi

`+=`	`object.__iadd__(self, other)`
`-=`	`object.__isub__(self, other)`
`*=`	`object.__imul__(self, other)`
`/=`	`object.__itruediv__(self, other)`
`//=`	`object.__ifloordiv__(self, other)`
`%=`	`object.__imod__(self, other)`
`**=`	`object.__ipow__(self, other)`
`abs()`	`object.__abs__(self)`
`complex(object)`	`object.__complex__(self)`
`int(object)`	`object.__int__(self)`
`float(object)`	`object.__float__(self)`

Zaokrúhľovanie hodnoty objektu na celé čísla

```
round()         object.__round__(self[, ndigits])
trunc()         object.__trunc__(self)
floor()         object.__floor__(self)
ceil()          object.__ceil__(self)
```

Ukážka - hierarchia tried

- vytvoríme hierarchiu nadtried a podtried
- dedenie a overriding metód
- práca s podtriedami
- iterátory

Overriding metód v Pythone

- nazýva sa to aj shadowing - prekonávanie
- špecifikujeme inú funkcionalitu pre podtriedy, ale použijeme rovnaký názov metódy aj rovnaké parametre
- ak funkcionality nie sú úplne iné (nemali by byť), tak by sme mali využiť implementáciu z nadtriedy

Method overloading v Pythone

- preťaženie metód
- máme metódy s rovnakým názvom, ale s inou návratovou hodnotou/s inými parametrami
- typický príklad - preťaženie základných operácií
- v Pythone preťaženie vlastných metód nie je možné, používajú sa na rovnaký účel defaultné hodnoty

Čo by sme chceli:

```
class Number:
    def __init__(self, value):
      self.num = value

    def multiply(self):
      return self.num * 2

    def multiply(self, number):
      return self.num * number
```

Ako to urobíme:

```
class Number:
    def __init__(self, value):
      self.num = value

    def multiply(self, number=2):
      return self.num * number
```

Čo by sme chceli:

```
class Number:
    def __init__(self, value):
      self.num = value

    def multiply(self):
      return self.num * self.num

    def multiply(self, number):
      return self.num * number
```

Ako to urobíme:

```
class Number:
    def __init__(self, value):
      self.num = value

    def multiply(self, number=None):
      if number is None:
          return self.num * self.num
      else:
          return self.num * number
```

alebo použijeme iný názov metódy

Práca s podtriedami

- keďže podtriedy by mali byť bližšie špecifikácie nadtriedy, funkcionalita preťažených tried by mala byť podobná
- práve preto je dobrým zvykom použiť implementáciu z nadtriedy (volanie `super`)
- v Pythone máme dve možnosti:
  - ak poznáme názov nadtriedy (zvyčajne):

```
SuperClass.method_name(self, parameters)
```

  - ak nepoznáme názov nadtriedy (chceme vytvoriť podtriedu z knižnice):

```
super(SubClass, self).method_name(parameters)
```

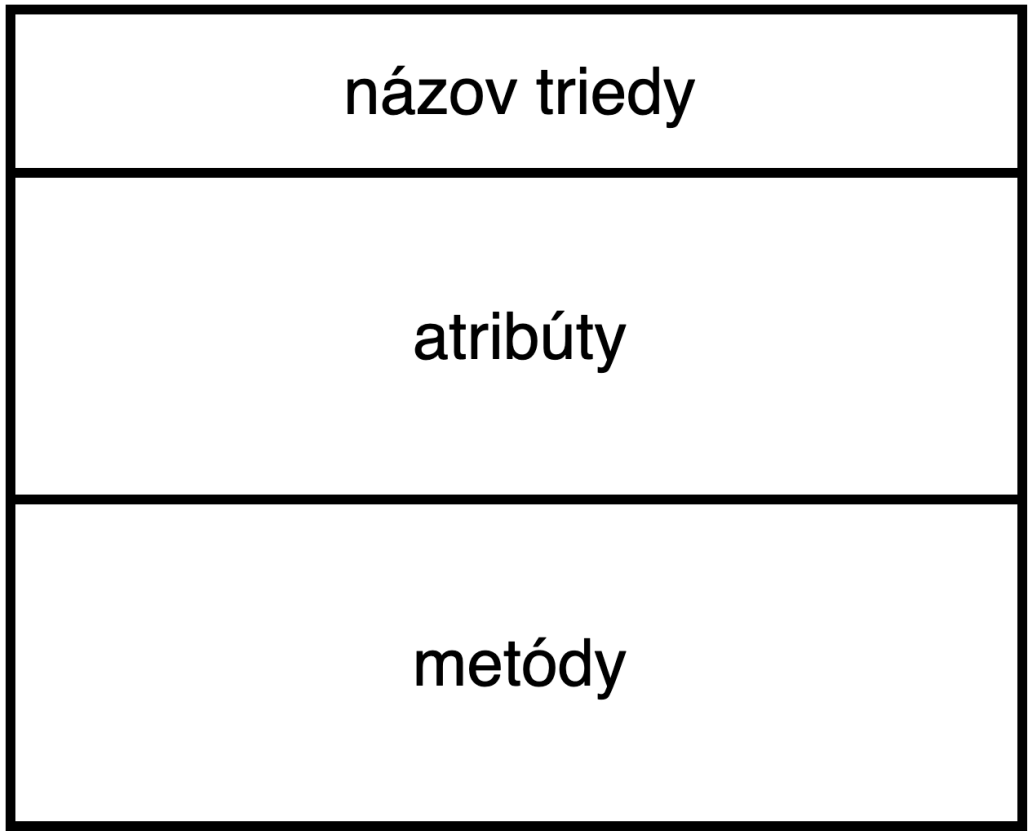
Iterátory

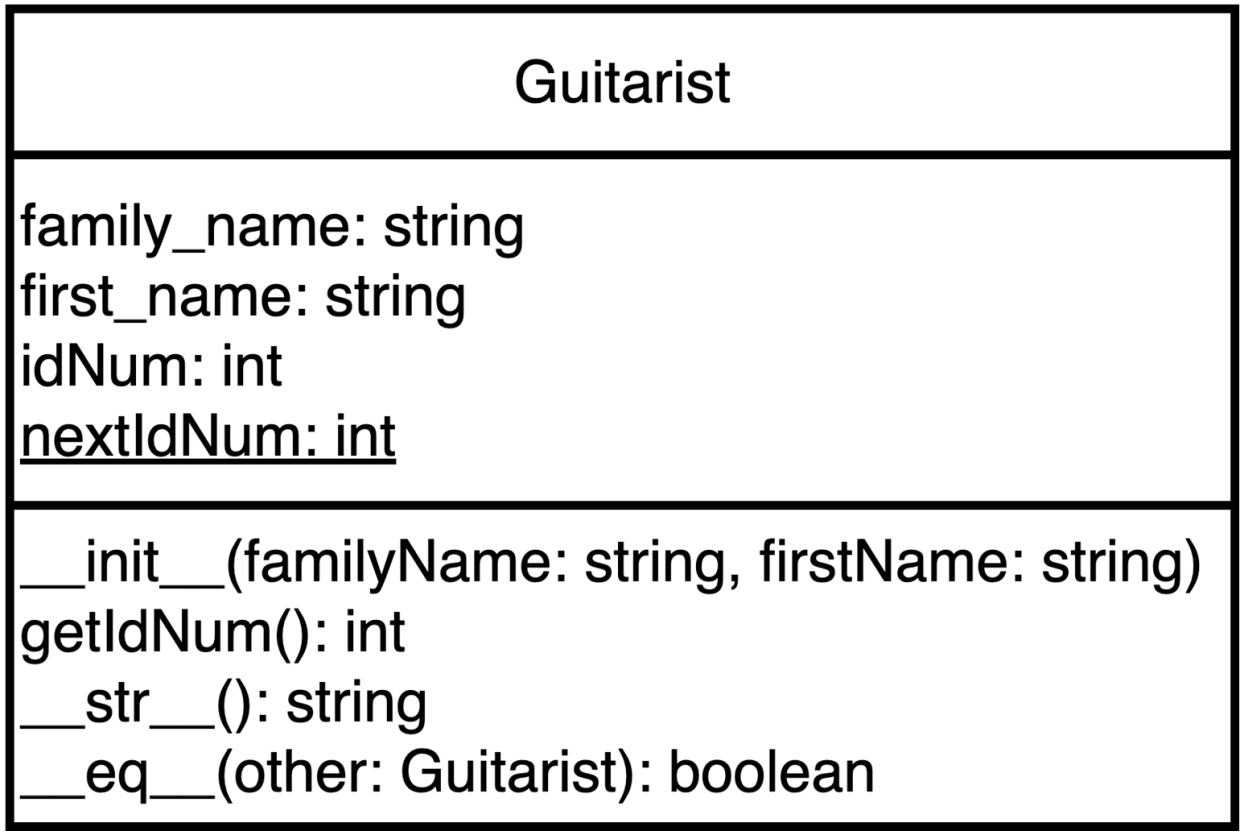
- pre kolekcie
- definujú podporu pre `for` cykly
- definícia pomocou dvoch metód
  - `__iter__(self)`
    - vytvorí iterátor
    - inicializuje pomocné premenné
    - vracia `self`
  - `__next__(self)`
    - vracia nasledujúci prvok v kolekcii
    - ak sme sa dostali na koniec kolekcie, vyhodí výnimku `StopIteration`

UML diagramy

- UML - Unified Modeling Language
- slúži na vizualizáciu architektúry a funkcionality softvérového riešenia
- základom objektovo orientovaného modelovania je diagram tried (class diagram)
- jeden blok reprezentuje jednu triedu
- môže byť použitý aj pre modelovanie dát

Class diagram



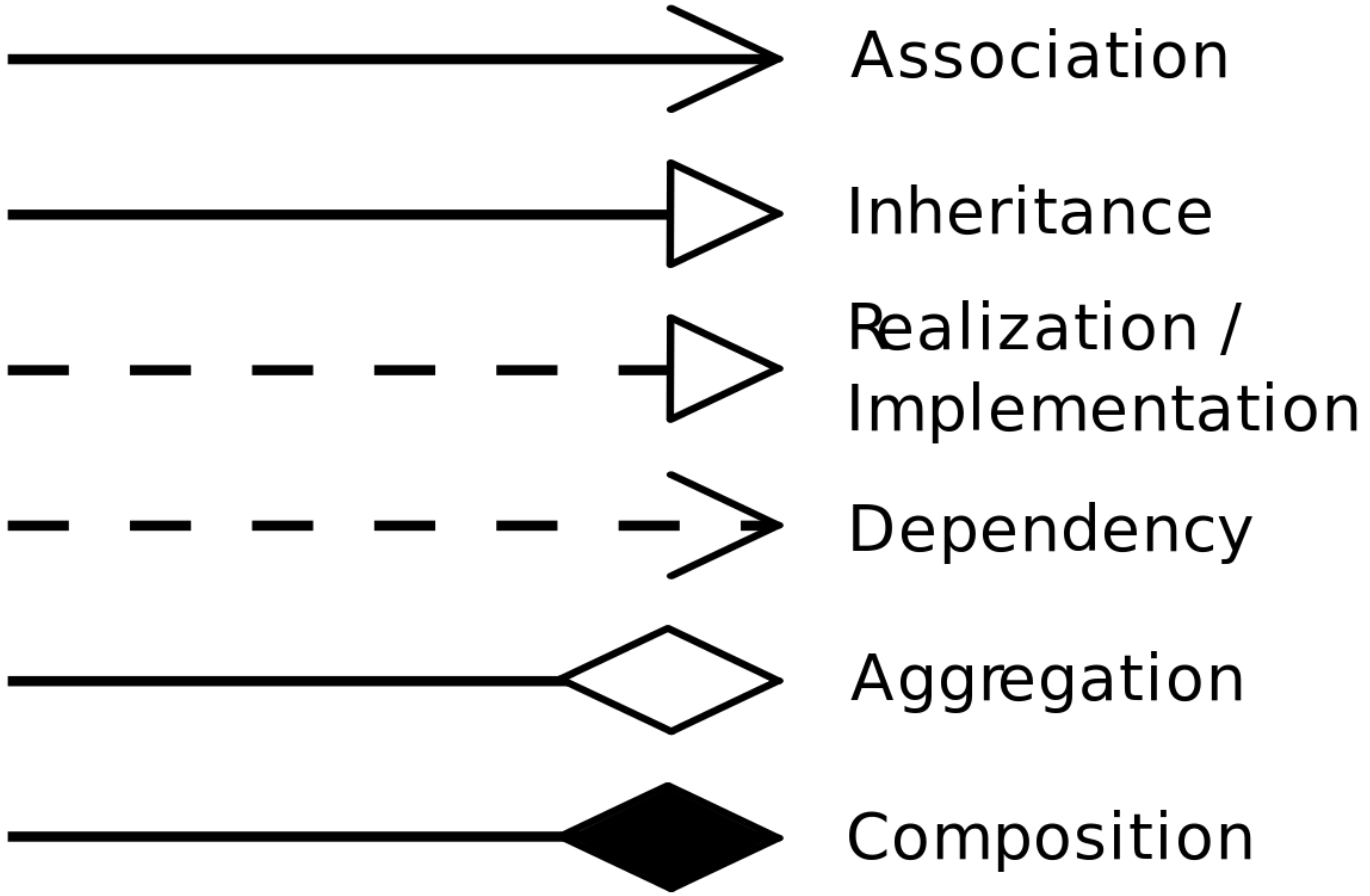


Enkapsulácia

- v Pythone nemáme príznaky, definujeme to priamo v názve premennej
  - `idNum: int` - public premenná
  - `__idNum: int` - private premenná
- premenné triedy sú podčiarknuté
  - `nextIdNum: int`

Vzťahy medzi triedami

- asociácia
- dedenie
- implementácia
- závislosť
- agregácia
- kompozícia

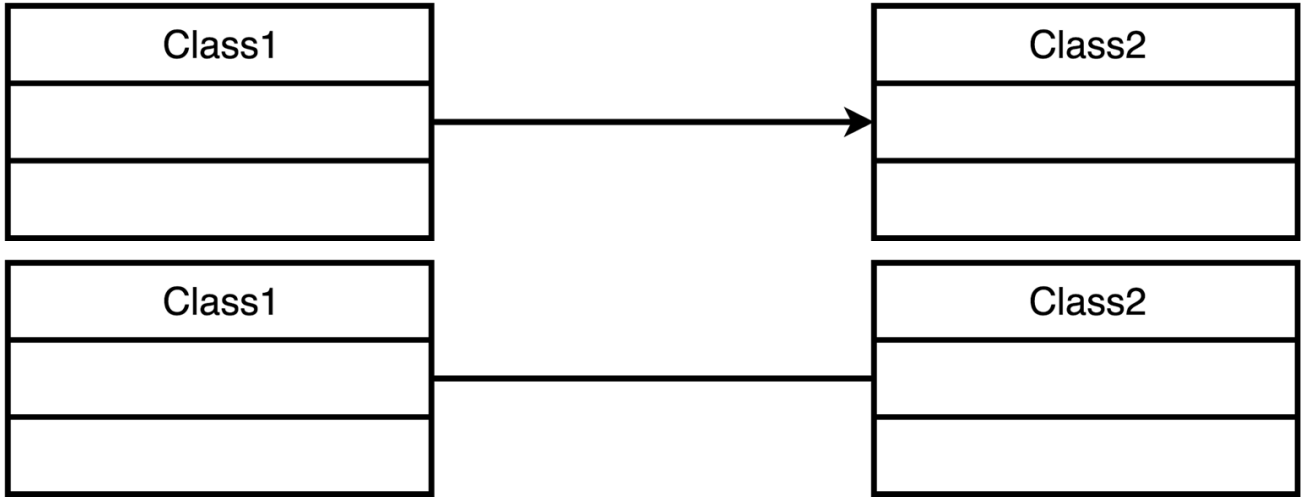


Kardinalita

- pre každý vzťah môžeme definovať multiplicitu (koľkonásobný je vzťah)
  - **0** - 0
  - **0..1** - 0 alebo 1
  - **0..*/*** - 0 až n
  - **1/1..1** - 1
  - **1..*** - 1 až n
- uvádza sa na konci čiary reprezentujúcej vzťah (pri triede)

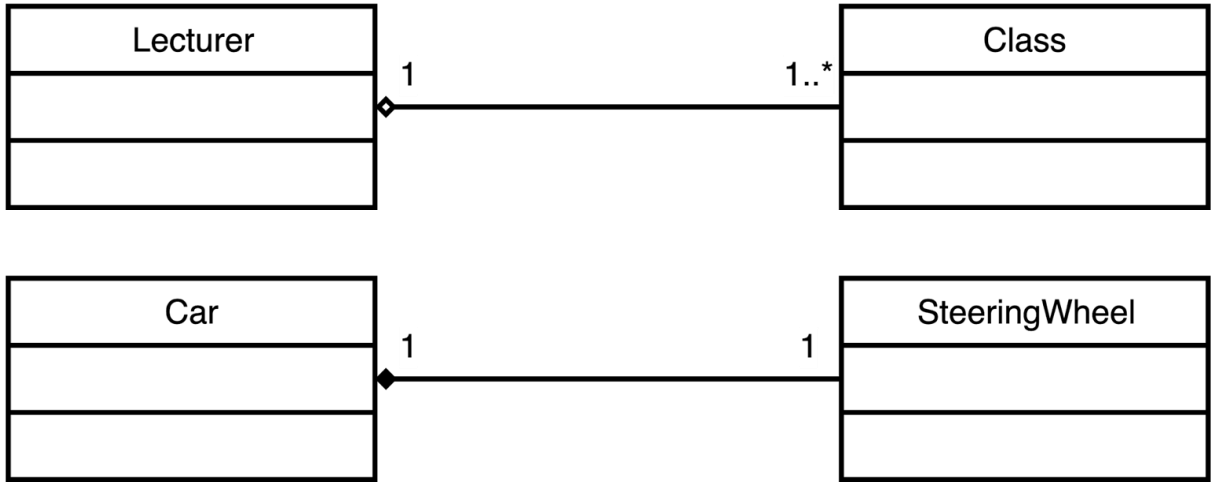
Asociácia

- na úrovni inštancií
- reprezentovaná šípkou (jednosmerná asociácia) alebo čiarou (obojsmerná)
- objekt jednej triedy sa spolieha na metódu druhého objektu
- jeden objekt používa druhý objekt
- jeden objekt je atribútom druhého objektu



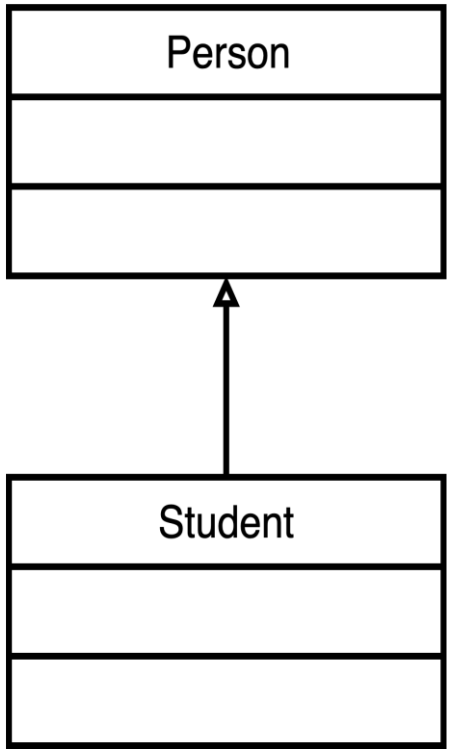
Agregácia a kompozícia

- na úrovni inštancií
- vyjadrujú vzťah, kde objekt jednej triedy sa skladá/obsahuje objekt druhej triedy
- agregácia - ak vymažeme kontajner objekt, jednotlivé časti môžu ďalej existovať
- kompozícia - jednotlivé časti nemajú funkcionalitu mimo kontajnera



Dedenie

- na úrovni tried
- dedenie vyjadruje, že podtrieda je bližšou špecifikáciou nadtriedy



Závislosť

- všeobecný vzťah
- vyjadruje prípad, kde jeden objekt použije druhý objekt, ale vzťah je omnoho slabší ako asociácia
- trieda, ktorá je závislá od druhej obsahuje metódu, kde objekt z nezávislej triedy je parameter metódy

Zhrnutie

- abstrakcia
- enkapsulácia
- stav objektu
- členské premenné a premenné tried
- dedenie
- polymorfizmus
- konštruktor
- metametódy a ich význam
- porovnávanie objektov
- overriding a overloading
- UML diagramy, vzťahy medzi triedami