Základy jazyka Python

objektovo orientované programovanie

prednáška 9

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

Ukážka - reprezentácia bodov v priestore

v dvojrozmernom priestore vieme každý bod reprezentovať dvoma spôsobmi:

1. zápis pomocou karteziánskych súradníc - hodnoty x a y
2. zápis pomocou polárnych súradníc - vzdialenosť od bodu [0, 0] a uhol spojnice bodu a osi

Môžeme vytvoriť reprezentáciu bodu, ktorá je nezávislá od zápisu?

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
- v Python 2 - jedna metóda `__cmp__(self, other)`, ktorá vracia
  - \+ ak `self > other`
  - 0 ak `self == other`
  - \- ak `self < other`
- v Python 3 funkcionalita v rôznych metódach

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

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

Metóda pre vlastnú hash hodnotu

- definuje sa v metóde `__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 a pokročilé OOP v Pythone

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

Premenné tried

- premenné typu `self.name` sú špecifické pre každú inštanciu
- Python umožňuje používanie triednych 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 (najmä singleton)

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
```

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(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`

Generátory

- nástroje pre definíciu iterátorov
- definujeme ich ako funkcie, použijeme ale `yield` namiesto `return`
- medzivýsledky a pomocné premenné sú zapamätané (napr. počítadlo)
- vracia špeciálny typ objektu: generátor iterátor

```
def reverse(data):
    for index in range(len(data)-1, -1, -1):
        yield data[index]
```

Generator expressions

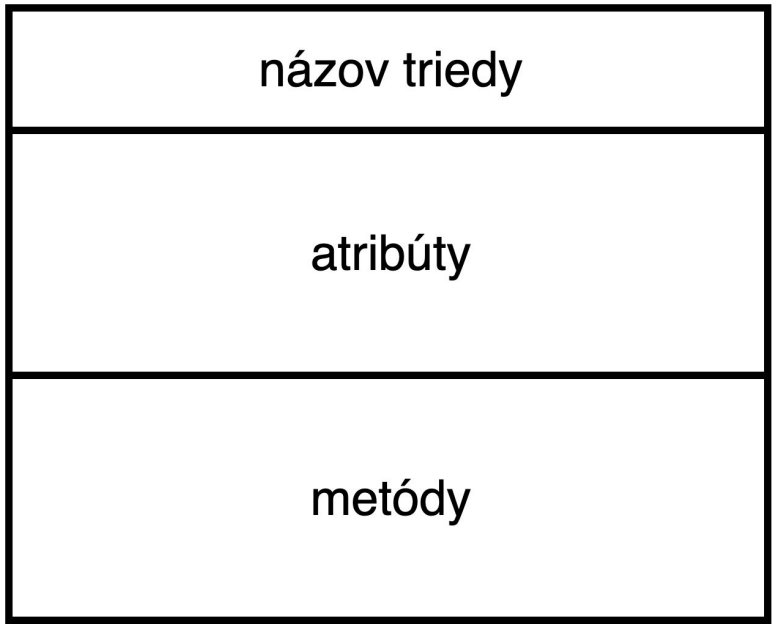
- podobné list comprehensionom, ale zaberajú menej pamäte
- používajú sa, ak chceme priamo použiť výsledok z generátora
- kompaktnejšie ako generátory, ale neponúkajú toľko možností
- syntax rovnaká, ako pri list comprehension, ale nepoužijeme hranaté zátvorky

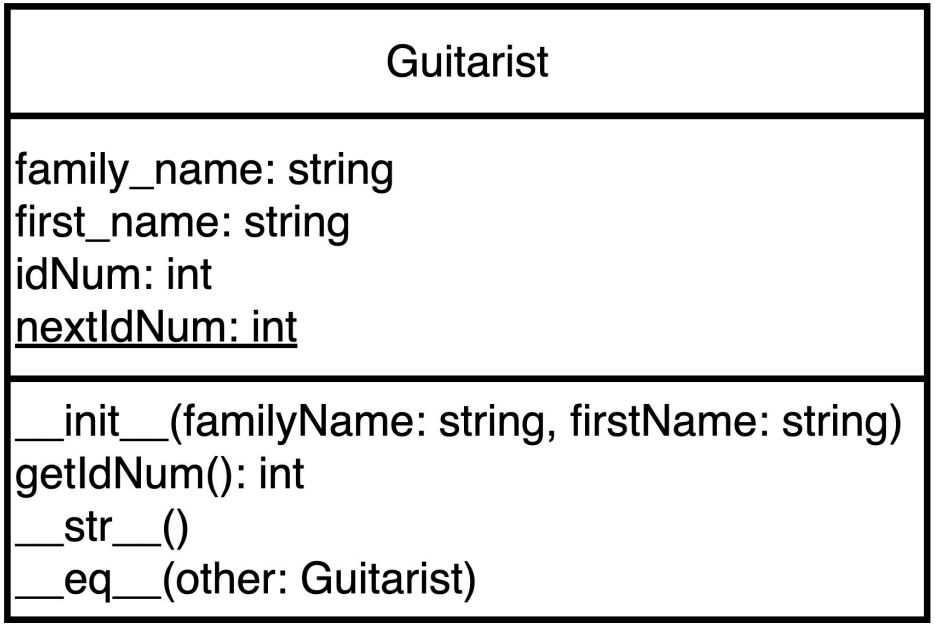
```
sum(i*i for i in range(10))

sum([i*i for i in range(10)])
```

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



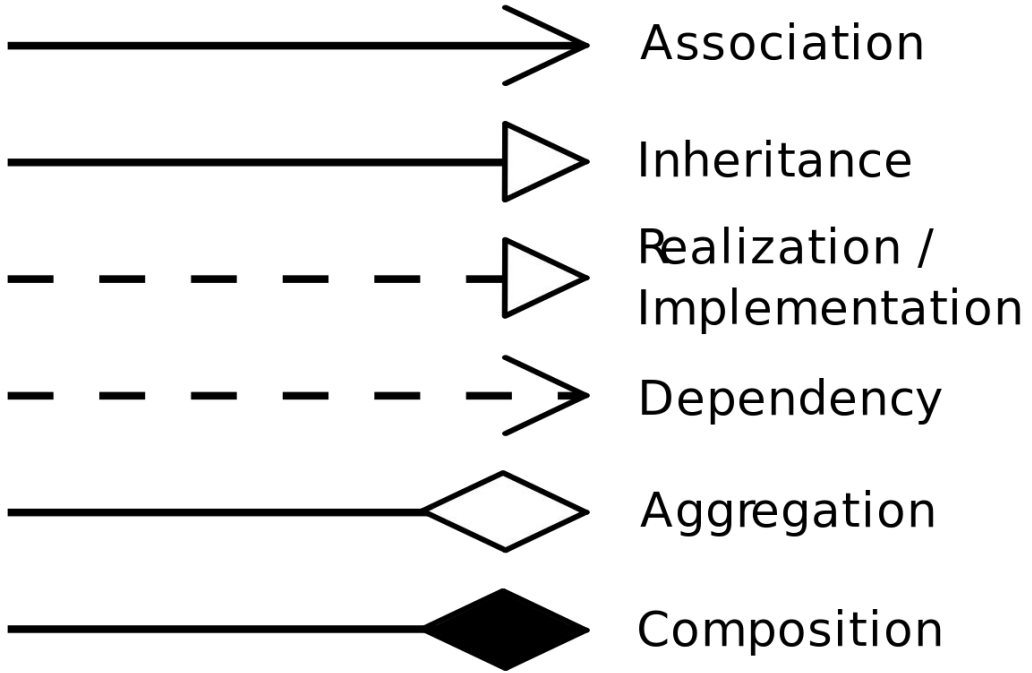


Enkapsulácia

- UML definuje nasledujúce znaky pre enkapsuláciu:
  - + public
  - − private
  - # protected
  - ~ package
- v Pythone tieto príznaky nemáme, 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

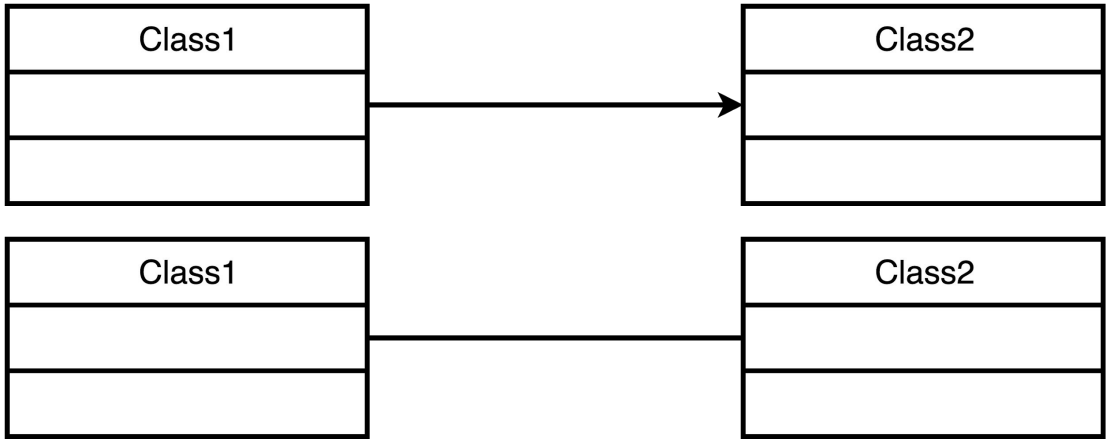


Kardinalita

- 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)

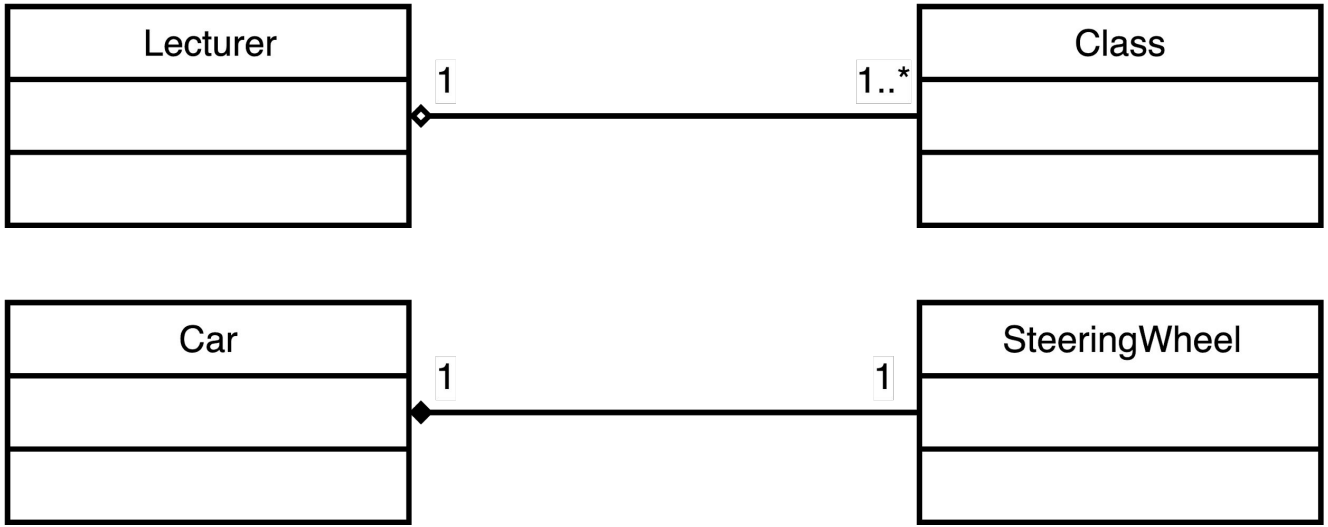
Asociácia

- 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



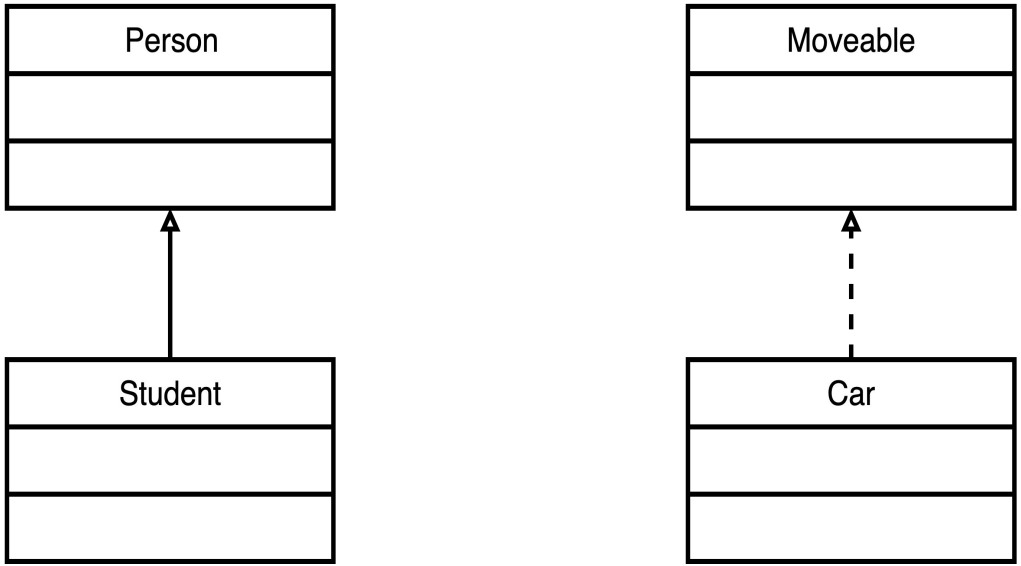
Agregácia a kompozícia

- 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



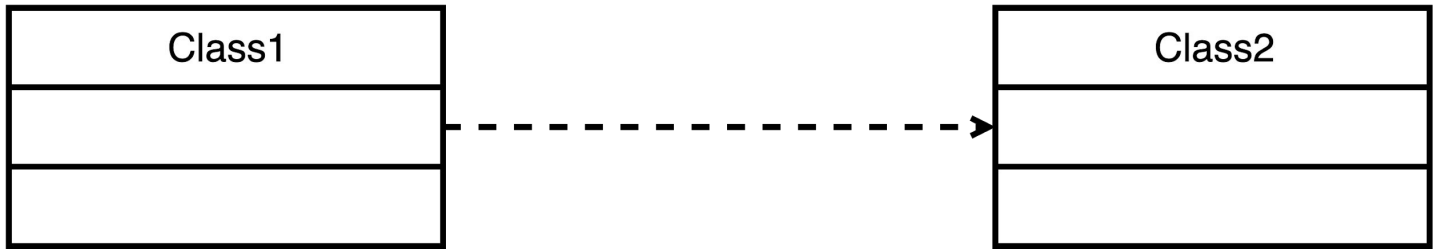
Dedenie a implementácia

- na úrovni tried
- dedenie vyjadruje, že podtrieda je bližšou špecifikáciou nadtriedy
- implementácia znamená, že trieda implementuje (realizuje) rozhranie



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



otázky?