Základy jazyka Python

objektovo orientované programovanie

prednáška 8

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

Prečo objektovo orientované programovanie?

- podporuje modularitu
- umožňuje znovupoužitie kódu (code reuse)
- umožňuje rozšíriť jazyk o vlastné údajové typy

Modularita

- modul je celok súvisiacich funkcií a hodnôt
- najčastejší príklad použitia - knižnice
- programátorské riešenie je modulárne ak časti kódu sú rozdelené do rôznych súborov

Moduly v Pythone

- ak chceme pracovať s modulmi, musíme ich naimportovať

```
import modul_name
```

- možnosť prideliť vlastný názov modulom

```
import modul_name as name
```

- možnosť cieleného importu

```
from modul_name import this, that [as name]
```

Riešenie menných konfliktov

- menný konflikt (name conflict) nastane ak v kontexte vykonávaného kódu existujú dva atribúty (zvyčajne funkcie) s rovnakým názvom
- riešenie - dot notation

```
import numpy as np
import random

np.random.randint(5, 10)
random.randint(5, 10)
```

Základné konštrukty OOP - trieda

- **trieda** je špeciálny typ modulov
- slúži na abstrakciu údajov, teda je to **abstraktný dátový typ**
- vzťah ku objektom
  - je to šablóna pre vytvorenie objektu
  - trieda je kolekcia objektov s rovnakými vlastnosťami
- z pohľadu interpretera je trieda definíciou vlastného dátového typu
- v Pythone je každá trieda zároveň aj objektom

Definícia tried v Pythone

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

    def set_price(self, new):
        self.price = new

    def sell(self):
        print("Was sold for {}".format(self.price))
```

Základné konštrukty OOP - objekt

- **objekt** je inštancia triedy - konkrétny príklad/údaj/premenná
- objekt sa skladá z **údajov** a z **funkcií (metód)** - v Pythone sa obe považujú za atribúty
- údaje sú uložené vo vnútorných premenných - existujú iba v rámci objektu

Vytvorenie objektu v Pythone

```
t = Test()
t.sell()
```

```
l = list()
l.append(5)
```

```
num = 5
```

Logický tok programu v OOP

- pri sekvenčnom programovaní môžeme vnímať program ako postupnosť operácií
- v prípade OOP sa program skladá z posielania správ medzi objektmi

```
lst = list()
lst.append(5)
```

Princípy OOP v Pythone

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

Enkapsulácia

- 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
```

Abstrakcia

- abstrakcia môže byť vnímaná ako rozšírenie enkapsulácie
- kým pomocou enkapsulácie skryjeme vnútorný stav objektu, 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 majú implementovanú funkcionalitu
    - spoliehame sa na dokumentáciu a na autora daného kódu
2. keďže trieda definuje abstraktný dátový typ, nezaoberáme sa ani vnútornou reprezentáciou údajov
    - napr. strom môže byť reprezentovaný ako zoznam alebo ako množina prepojených uzlov
    - vnímame iba vonkajší kontext triedy a nie vnútorné detaily

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
```

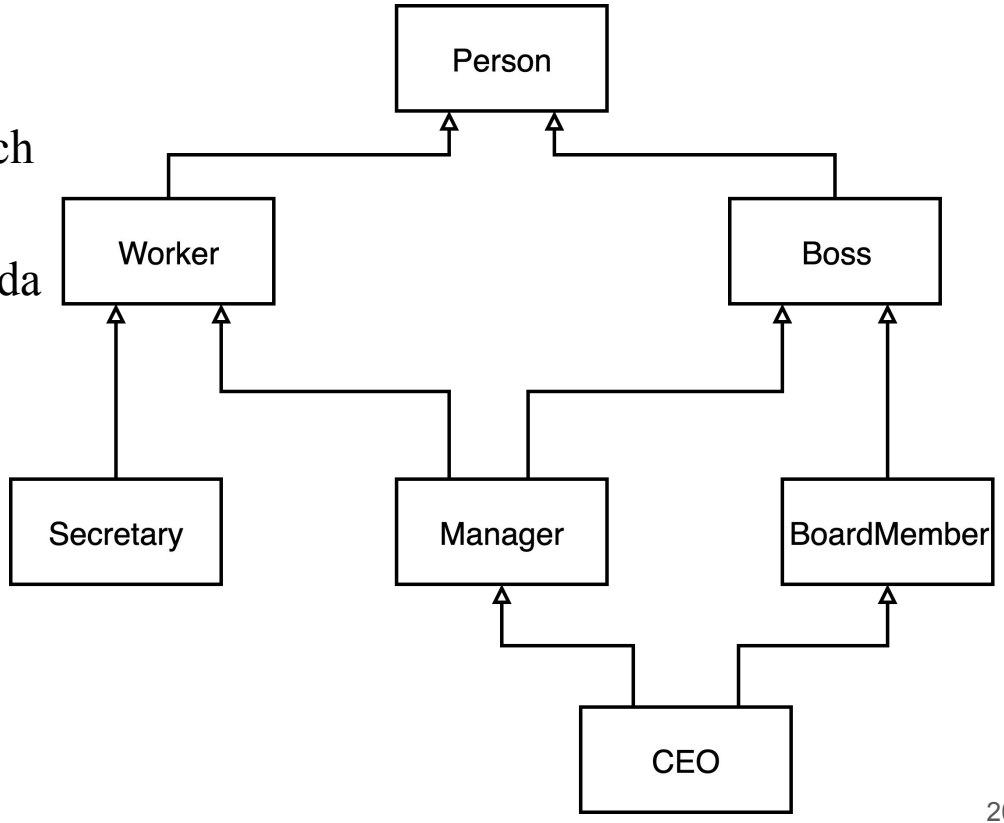
Viacnásobné dedenie v Pythone

```
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
```

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

otázky?