



# **Programovanie v jazyku Python**

Úvod do objektovo orientovaného programovania  
prednáška 8

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

# Objektovo orientované programovanie

- myšlienka zo 70-tych rokov
- náhly rozvoj až po príchode jazyka Java
- program je vnímaný nie ako postupnosť volaní, ale ako spolupráca nezávislých blokov

# Prečo objektovo orientované programovanie?

- podpora modularity
- znovupoužitie kódu (code reuse)
- rozšírenie jazyka o vlastné údajové štruktúry a typy

# Štruktúra objektového riešenia

- kód rozdelíme do modulov
- modul je celok súvisiacich funkcií a hodnôt
- najčastejší prípad použitia modulov sú knižnice
- riešenie je modulárne, ak časti kódu sú rozdelené do rôznych súborov

# Moduly v Pythone

- ak chceme pracovať s modulmi, musíme ich importovať

```
import modul_name
```

- možnosť pridelit' 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 a objekt
- trieda je šablóna pre vytvorenie objektov
- objekt je inštanciou triedy – konkrétny príklad

# Základné konštrukty OOP - trieda

- **trieda** je špeciálny typ modulov
- slúži na abstrakciu údajov, teda je to **abstraktný dátový typ**
- vzťah k 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



# Základné konštrukty OOP - objekt

- **objekt** je inštanciou 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

# Práce s objekty v Pythoně

```
x = 5
```

```
y = 6
```

```
x += 1
```

```
y -= 2
```

```
my_lst = list()
```

```
my_lst.append(6)
```

```
my_lst.append(4)
```

# 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))
```

# Vytvorenie objektu v Pythone

```
t = Product()  
t.sell()
```

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

# Diagram triedy

názov triedy
atribúty
metódy

Guitarist
family_name: string first_name: string idNum: int <u>nextIdNum: int</u>
__init__(familyName: string, firstName: string) getIdNum(): int __str__(): string __eq__(other: Guitarist): boolean

# 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`

# 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 funkcionálnosť
  - 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

# 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 Java/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

# Enkapsulácia v Pythone

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

# 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

# Definição de classes em Python

```
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 efektívnu prácu s triedami by sme mali použiť čo najviac už definovanú funkcionálnosť 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)
```



# 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

# Zhrnutie

- paradigmy programovania
- objektovo orientované programovanie
- moduly
- trieda a objekt
- abstrakcia
- enkapsulácia
- dedenie
- polymorfizmus