



# **Programovanie v jazyku Python**

Funkcie, rekurzia, generátory, lambda výrazy  
prednáška 2

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

# Definícia funkcie

- samostatná postupnosť inštrukcií v rámci programu
- zvyčajne reprezentuje vykonávanie úlohy
- je volaná ak je potrebné vykonať danú úlohu
- má jasne definované vstupy a výstupy

# Výhody funkcií

- dekompozícia problému
  - prístup “rozdeľuj a panuj”
  - najprv napíšeme riešenie pre jednoduchšie úlohy, následne ich spojíme
- abstrakcia
  - funkcie zakryjú implementačné detaily
  - stačí ak programátor vie aký vstup funkcia očakáva a aký je jej výstup
- znížené náklady pre vývoj
- eliminácia opakujúceho sa kódu
- jednoduchá údržba kódu

# Štruktúra funkcie

- názov
- parametre
- premenné
- návratová hodnota
- špecifikácia

# Syntax definície funkcií v Pythone

```
def nazov(parametre) :  
    telo  
    return navratova_hodnota
```

# Parametre funkcií v Pythone

- nepotrebujeme určiť dátový typ

```
def square(number) :
```

- parametrom vieme priradiť predvolenú hodnotu

```
def square(number, print_result=False) :
```

- parametre s predvolenou hodnotou sú nepovinné pri volaniach

- vieme definovať funkciu s neznámym počtom parametrov

```
def get_last(*keys) :
```

- s parametrami vieme narábať ako s n-ticou

- pri volaní funkcie môžeme uviesť názov parametrov, v tomto prípade nemusíme dodržiavať ich poradie

# Typové aliasy

- Python je dynamický typovaný jazyk, v kóde ale vieme naznačiť očakávané dátové typy

```
def square(number:int) -> int:  
    return number ** 2
```

- pre zložitejšie typy vieme použiť typové aliasy:

```
ListResult = list[int]  
DictionaryResult = dict[str, ListResult]
```

- aliasy sa používajú len pri statickej kontrole, nie počas behu

# Defenzívne programovanie

- pri implementácii funkcie treba vychádzať z predpokladu, že funkcia bude musieť spracovať aj chybný vstup
- pred vykonaním samotnej úlohy funkcie si skontrolujeme správnosť a platnosť vstupných hodnôt
  - typová kontrola
  - kontrola platnosti hodnôt (ak existujú obmedzenia)
- odporúča sa aj kontrola výstupu pred ukončením behu funkcie



# Premenné vo funkciách

- každá funkcia má prístup ku globálnym premenným
- každá funkcia má lokálny menný priestor (namespace), ktorý je neviditeľný pre ostatné funkcie
- pri volaní funkcie sa vytvorí vlastný menný priestor (namespace)

# Návratová hodnota funkcií

- určená pomocou kľúčového slova **return**
- v Pythone každá funkcia musí mať návratovú hodnotu - defaultne `None`
- funkcia môže mať niekoľko návratových hodnôt  
**return value1, value2, ...**
- návratovú hodnotu funkcií vieme uložiť pomocou priradenia pri volaní  
**value1 = funkcia1()**  
**value1, value2 = funkcia2()**

# Špecifikácia funkcií

- píše sa pre programátora/používateľa
- súčasťou dokumentácie
- opisuje účel funkcie, očakávaný vstup a návratovú hodnotu, resp. účinok funkcie
- syntax: v trojitých úvodzovkách hneď v prvom riadku funkcie

# Zabudované funkcie v Pythone

- pretypovacie a typové
- operačné
- vstupno-výstupné

# Pretypovacie a typové funkcie v Pythone

- `bool([x])`
- `chr(i)`
- `complex([real[, imag]])`
- `dict()`
- `enumerate(iterable, start=0)`
- `float([x])`
- `frozenset([iterable])`
- `int([x])`
- `isinstance(object, classinfo)`
- `list([iterable])`
- `set([iterable])`
- `str(object)`
- `tuple([iterable])`
- `type(object)`

# Operačné funkcie v Pythone

- `abs(x)`
- `all(iterable)`
- `any(iterable)`
- `divmod(a, b)`
- `eval(expression, globals=None, locals=None)`
- `filter(function, iterable)`
- `hash(object)`
- `len(s)`
- `map(function, iterable, ...)`
- `max(iterable)`
- `min(iterable)`
- `next(iterator)`
- `range(start, stop[, step])`
- `reversed(seq)`
- `round(number[, ndigits])`
- `sorted(iterable, key=None, reverse=False)`
- `sum(iterable[, start])`
- `zip(*iterables)`

# Vstupno-výstupné funkcie v Pythone

- `format(value[, format_spec])`
- `input([prompt])`
- `open(file, mode='r', encoding=None, ...)`
- `print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)`

# Práca so súbormi - otváranie a vytváranie súborov

```
open(file, mode='r', encoding=None, errors=None,  
newline=None)
```

- file - cesta k súboru (ak cesta neexistuje, súbor sa vytvorí)
- mode - spôsob práce so súborom (text alebo bajty)
  - r – read
  - w – write
  - a – append
  - + – update
  - b/t – binárny/textový mód
- encoding - iba pre texty, určuje spôsob zakódovania textu
- errors - spôsob ako spracovať chyby pri zakódovaní
- newline - znak použitý ako znak pre nový riadok



# Práca so súbormi - čítanie zo súboru

`file.read()` / `file.read(5)`

- bez parametra načíta všetky znaky zo súboru
- parameter udáva počet načítaných znakov

`file.readline()` / `file.readline(5)`

- načítavanie zo súboru po riadkoch
- parameter určuje číslo načítaných riadkov

`file.readlines()`

- načíta všetky riadky zo súboru ako zoznam reťazcov

# Práca so súbormi - zápis do súboru

`write(string)`

- zapíše reťazec do buffra

`writelines(list)`

- zapíše zoznam reťazcov do buffra
- nepridáva znak na koniec riadku

`flush()`

- zapíše obsah buffra do súboru

# Práca so súbormi - zatvorenie súboru

`close()`

- zatvorí súbor
- vždy musí byť zavolaná
- obsah buffra sa zapíše do súboru

# Práca so súbormi

```
my_file = open('path', 'r')  
lines = my_file.readlines()  
  
my_file.close()
```

```
with open('path', 'r') as my_file:  
    lines = my_file.readlines()
```

obsahuje `close()` explicitne

# Funkcia main

```
if __name__ == '__main__':  
    main()
```

Dobré programátorské zvyky:

- vždy vytvorte funkciu `main()`
- funkcia `main()` by mala obsahovať čo najmenej volaní
- telo podmienky `if __name__` by malo byť iba volanie funkcie `main()`, prípadne spracovanie vstupných parametrov

# Odovzdanie parametrov main funkcií

- hodnoty sú v premennej `sys.argv`
- elegantnejšie riešenie - pomocou knižnice `argparse`

```
parser = argparse.ArgumentParser()
parser.add_argument('--path', metavar='path', required=True, help='path
to project folder')
parser.add_argument('--save', metavar='path', required=True, help='path
to save directory')
args = parser.parse_args()
main(workspace=args.path, schema=args.save)
```

# Rekurzívne funkcie

- funkcia definovaná pomocou seba samej
- definícia sa skladá z dvoch častí
  - základný prípad
    - najjednoduchší možný prípad
    - hodnota je pevne daná
  - indukčný/rekurzívny krok
    - definuje spôsob výpočtu pre zložitejšie prípady
    - obsahuje volanie práve definovanej funkcie

# Rekurzia vs. iteratívne riešenia

- rekurzia a cykly sú sémanticky ekvivalentné
  - niektoré problémy sú viac vhodné riešiť rekurzívne, iné iteratívne
  - ak vieme odvodiť spôsob riešenia pomocou jednoduchých prípadov → rekurzia
  - brute force algoritmy → iteratívne riešenie
- 
- typické príklady pre rekurziu: palindrómy, Fibonacciho čísla



# Palindrómy

Palindróm je postupnosť symbolov, ktorú je možné prečítať v oboch smeroch s rovnakým významom, napr.: 1001001, Elze je zle.

Riešenie pomocou rekurzie:

- prázdna postupnosť je palindróm
- postupnosť s jedným znakom je palindróm
- postupnosť je palindróm, ak prvý a posledný znak sú rovnaké a zároveň stredná časť je palindróm

# Fibonacciho čísla

Popisujú postupnosť, v ktorej každý člen je súčtom dvoch predchádzajúcich. Ako príklad uviedol Fibonacci výpočet rast populácie zajacov.

Riešenie pomocou rekurzie:

- prvý člen je 0
- druhý člen je 1
- ďalšie členy sú súčtom predošlých dvoch členov

# Brute force algoritmus - barnyard problem

Na dvore máme sliepky a zajace. Spolu majú 48 hláv a 128 nôh. Koľko máme sliepok a koľko zajacov?

- problém riešime formálne ako sústavu lineárnych rovníc

$$x + y = 48$$

$$2 * x + 4 * y = 128$$

- pre počítač je problém prirodzenejšie riešiť spôsobom brute force, čiže vyskúšaním všetkých možností až kým nenájdeme riešenie

# Fibonacciho čísla - generátory

- generátory sú špeciálne funkcie, ktoré fungujú ako iterátory
- efektívnejšia práca s pamäťou (lazy evaluation)
- použité kľúčové slovo **yield** namiesto **return**
- výsledok je iterátor, ktorý vieme použiť vo **for** cykle
- pomocou metódy **next** dostaneme ďalší záznam v postupnosti

# Použitie generátorov

- práca s veľkými súbormi
- spracovanie veľkých dát
- generovanie nekonečných radov
- pipeliney

# Lambda výraz

- funkcia bez názvu
- telo funkcie je jeden výraz, ktorý definuje návratovú hodnotu funkcie
- definícia musí byť v jednom riadku
- môže mať ľubovoľný počet parametrov

# Definícia lambda funkcií/výrazov

- **lambda** **parametre**: výraz  
**lambda** **x**: 3 \* x + 2
- môže byť uložená do premennej a následne zavolaná pomocou premennej  
f = **lambda** **x**: 3 \* x + 2  
f(2)
- dnes sa takéto použitie neodporúča, lambda výrazy by sme mali používať iba v niekoľkých prípadoch

# Prípady použitia lambda funkcií

- kľúč pre filtrovanie
- kľúč pre triedenie
- triedenie zložitých dátových typov (n-tíc, dictionary, objektov)
- jednoduché funkcie, ktoré nepotrebujeme použiť veľa krát



# Zhrnutie

- definícia funkcií
- štruktúra a časti funkcií
- vybrané zabudované funkcie
- rekurzia
- lambda výrazy a ich použitie