

Funkcionální programování v Pythonu

- Autor Pavel Tišnovský
- Email <tisnik 0x40 centrum 0x2e cz>
- Datum 2023-10-07

Obsah přednášky

- ▶ Programovací jazyk Python
- ▶ Funkcionální programování
- ▶ Praktická část

Programovací jazyk Python

- ▶ Typické použití Pythonu
 - ◆ Nástroje a utility ovládané z příkazového řádku
 - ◆ Aplikace s grafickým uživatelským rozhraním
 - ◆ Client-server
 - serverová část (Flask, Django, CherryPy, ...)
 - klient (Brython, spíše technologické demo)
 - ◆ Numerické výpočty, symbolické výpočty
 - NumPy
 - SciPy
 - Matplotlib
 - ◆ Moderní způsoby využití Pythonu
 - AI
 - Machine Learning (Deep Learning)
PyTorch
 - Big data
 - ◆ Tzv. „glue“ jazyk
 - ◆ Vestavitelný interpret Pythonu

Funkcionální programování

- ▶ Funkce jsou plnohodnotnými typy
- ▶ Čisté funkce (bez vedlejších efektů)
 - ◆ referenční průhlednost (transparency)
 - ◆ volání funkce s danými parametry lze nahradit za výsledek
- ▶ Preferuje se použití neměnitelných hodnot
 - ◆ immutable values/variables
 - ◆ čistě funkcionální datové struktury

Proč funkcionální programování?

- ▶ Funkce bez vedlejších efektů se snadno testují
- ▶ Funkce bez vedlejších efektů se snadno ladí
- ▶ Stav aplikace je izolován
- ▶ Zajištěn souběh či paralelní běh částí programu
- ▶ Neměnné hodnoty: méně možností vytvořit vedlejší efekt

Funkce jsou plnohodnotnými typy

- ▶ "Functions are first-class citizens"
 - ◆ v mnoha materiálech se toto tvrzení rychle přejde
 - ◆ ovšem má mnoho důsledků zasahujících do sémantiky
- ▶ Některé zásadní důsledky pro programovací jazyk
 - ◆ musí být podporovány funkce vyššího řádu
 - ◆ musí být podporovány uzávěry (pokud se nemění sémantika viditelnosti)
 - ◆ typicky se vyžadují nelokální změny stavu aplikace

Funkce jsou plnohodnotnými typy

- ▶ Další vlastnosti jazyka vyplývající z tohoto tvrzení
 - ◆ může být možné skládat funkce (compose)

- ◆ může být možné transformovat funkce
 - někdy i curryfikace jako forma transformace
- ◆ je možné si zapamatovat výsledky funkcí
 - čisté funkce lze chápat jako mapy/slovníky na stereoidech

Python jako funkcionální jazyk?

- ▶ V Pythonu jsou funkce plnohodnotnými datovými typy
 - ◆ se všemi z toho plynoucími důsledky
 - ◆ "funkční" literál poněkud matoucí
- ▶ Neměnitelné datové typy?
 - ◆ standardní jen částečně (řetězce, n-tice)
 - ◆ existují rozšiřující knihovny

Funkcionální koncepty v Pythonu

- ▶ Funkce vyššího řádu
- ▶ Anonymní funkce (lambda)
 - ◆ jen omezeně
- ▶ Uzávěry (closures)
- ▶ Generátorové notace
 - ◆ pro seznamy, množiny, slovníky i n-tice
- ▶ Částečně vyhodnocené funkce
 - ◆ transformace
- ▶ Caching výsledků funkcí

Praktická část

- ▶ Lambda výrazy
- ▶ Funkce vyššího řádu
- ▶ Uzávěry
- ▶ Generátorové notace
- ▶ Částečně vyhodnocené funkce
- ▶ Caching výsledků funkcí
- ▶ Dekorátory
- ▶ Persistentní datové struktury

Lambda výrazy

- ▶ V Pythonu skutečně "jen" výrazy
 - ◆ teoreticky stačí, ovšem sémantika Pythonu je problematická

```
#
# Lambda výraz jako plnohodnotný typ
#
```

```
(lambda x,y:x+y)(1, 2)
```

```
foo=lambda x,y:x+y
foo(1,2)
```

Funkce vyššího řádu

- ▶ Jejich existence plyne z definice funkcí jako plnohodnotného typu
- ▶ Standardní funkce vyššího řádu
 - ◆ map
 - ◆ filter
 - ◆ reduce (z functools)

```
#
# Funkce vyššího řádu vracející jinou funkci
#
```

```
def foo():
    def bar():
        print("BAR")
```

```

        return bar

x = foo()
x()
#
# Funkce vyššího řádu akceptující jinou funkci
#

def add(x, y):
    return x + y

def mul(x, y):
    return x * y

def less_than(x, y):
    return x < y

def get_operator(symbol):
    operators = {
        "+": add,
        "*": mul,
        "<": less_than,
    }
    return operators[symbol]

def calc(operator, x, y):
    return operator(x, y)

z = calc(get_operator("+"), 10, 20)
print(z)

z = calc(get_operator("*"), 10, 20)
print(z)

z = calc(get_operator("<"), 10, 20)
print(z)
#
# Použití funkce vyššího řádu map
#

values = range(-10, 11)

converted = map(abs, values)
print(list(converted))
#
# Použití funkce vyššího řádu filter
#

message = "Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do
eiusmod tempor incididunt ut labore et dolore magna aliqua"
words = message.split()

filtered = filter(lambda word: len(word) > 4, words)
print(list(filtered))
#
# Použití funkce vyššího řádu reduce
#

from functools import reduce

```

```

def multiply(x, y):
    return x * y

x = range(1, 11)
print(x)

y = reduce(multiply, x)
print(y)
#
# Výpočet faktoriálu založený na map a reduce
#

from functools import reduce

n = range(0, 11)

factorials = map(lambda n: reduce(lambda a, b: a*b, range(1, n+1), 1), n)

print(list(factorials))

```

Uzávěry

- ▶ Jejich existence je očekávána protože
 - ◆ funkce jsou plnohodnotnými typy
 - ◆ proměnné vně bloku jsou dostupné (viditelné)
- ▶ V Pythonu poněkud problematické
 - ◆ neglobální, nelokální proměnné
 - ◆ modifikace proměnných

```

#
# Čítač realizovaný uzávěrem: nefunkční varianta
#

def createCounter():
    counter = 0
    def next():
        counter += 1
        return counter
    return next

counter1 = createCounter()
counter2 = createCounter()
for i in range(1,11):
    result1 = counter1()
    result2 = counter2()
    print("Iteration #%d" % i)
    print("    Counter1: %d" % result1)
    print("    Counter2: %d" % result2)
#
# Čítač realizovaný uzávěrem: varianta pro Python 2
#

def createCounter():
    counter = [0]
    def next():
        counter[0] += 1
        return counter[0]
    return next
#
# Čítač realizovaný uzávěrem: varianta pro Python 3
#

```

```
def createCounter():
    counter = 0
    def next():
        nonlocal counter
        counter += 1
        return counter
    return next
```

Generátorové notace

- ▶ Idiomatická syntaxe pro přepis funkcí typu map a filter
 - ◆ pro n-tice
 - ◆ pro seznamy
 - ◆ pro množiny
 - ◆ pro slovníky

Ukázky generátorových notací pro jednotlivé datové typy

```
#
# Generátorová notace: n-tice
#

(x*2 for x in range(11) if x%3 != 0)
#
# Generátorová notace: seznamy
#

message = "Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do
eiusmod tempor incididunt ut labore et dolore magna aliqua"
words = message.split()

lengths = [len(word) for word in words]
#
# Generátorová notace: množiny
#

{x*2 for x in range(11) if x%3 != 0}

{(x, x*2) for x in range(11) if x%3 != 0}
#
# Generátorová notace: slovníky
#

{x: x*2 for x in range(11) if x%3 != 0}
```

Částečně vyhodnocené funkce a metody

- ▶ Opět podporováno knihovnou functools
- ▶ Částečná aplikace parametrů
- ▶ Výsledkem bude nová plně použitelná funkce
- ▶ Lze ovšem aplikovat i na metody

Ukázky tvorby a použití částečně vyhodnocených funkcí

```
#
# Částečně vyhodnocená funkce
#

from functools import partial

def mul(x, y):
    return x * y
```

```

print(mul(6, 7))

print()

doubler = partial(mul, 2)

for i in range(11):
    print(i, doubler(i))
#
# Částečně vyhodnocená funkce
#

from functools import partial

def mul(x=1, y=1, z=1, w=1):
    return x * y * z * w

f1 = mul
f2 = partial(mul, x=2)
f3 = partial(mul, y=2)
f4 = partial(mul, y=2, z=2)
f5 = partial(mul, x=2, y=2, z=2)
f6 = partial(mul, x=2, y=2, z=2, w=2)
#
# Částečně vyhodnocená metoda
#

class Foo:
    def __init__(self):
        self._enabled = False

    def set_enabled(self, state):
        self._enabled = state

    enable = partialmethod(set_enabled, True)
    disable = partialmethod(set_enabled, False)

    def __str__(self):
        return "Foo that is " + ("enabled" if self._enabled else "disabled")

```

Dekorátory

- ▶ Prakticky: transformace funkce
 - ◆ obalení funkce jinou funkcí
 - ◆ velmi užitečné v praxi

Praktické používání dekorátorů

```

#
# Vytvoření a aplikace nového dekorátoru
#

from functools import decorator

@decorator
def wrapper1(function):
    print("-" * 40)
    function()
    print("-" * 40)

```

```
@wrapper1
def hello():
    print("Hello!")
```

```
hello()
#
# Vytvoření a aplikace nového dekorátoru
#
```

```
from funcy import decorator
```

```
@decorator
def wrapper1(function):
    print("-" * 40)
    function()
    print("-" * 40)
```

```
@decorator
def wrapper2(function):
    print("=" * 40)
    function()
    print("=" * 40)
```

```
@wrapper1
@wrapper2
def hello():
    print("Hello!")
```

```
hello()
#
# Vytvoření a aplikace nového dekorátoru: měření doby trvání funkce
#
```

```
from funcy import decorator
import time
```

```
@decorator
def measure_time(func):
    t = time.time()
    res = func()
    print("Function took " + str(time.time() - t) + " seconds to run")
    return res
```

```
@measure_time
def tested_function(n):
    print(f"Sleeping for {n} seconds")
    time.sleep(n)
```

```
tested_function(1)
tested_function(2)
#
# Dekorátor @silent
#
```

```

from funcy import silent

@silent
def divide(a, b):
    return a/b

print(divide(1, 2))
print(divide(1, 0))
#
# Dekorátor @ignore
#

from funcy import ignore

@ignore(errors=ZeroDivisionError, default=-1)
def divide(a, b):
    return a/b

print(divide(1, 2))
print(divide(1, 0))
#
# Dekorátor @ignore
#

from funcy import ignore

@ignore(errors=[ZeroDivisionError, TypeError], default=-1)
def divide(a, b):
    return a/b

print(divide(1, 2))
print(divide(1, 0))
print(divide(None, 1))
#
# Dekorátor @ignore
#

from funcy import ignore

@ignore(errors=ZeroDivisionError, default=0)
@ignore(errors=TypeError, default=-1)
def divide(a, b):
    return a/b

print(divide(1, 2))
print(divide(1, 0))
print(divide(None, 1))
#
# Dekorátor @reraise
#

from funcy import reraise

class MathError(Exception):
    def __init__(self, message):
        self.message = message

@reraise(errors=Exception, into=MathError("neděl nulou!"))
def divide(a, b):

```



```

    return a/b

print(divide(1, 2))
print(divide(1, 0))
#
# Dekorátor @retry
#

from funcy import retry

@retry(3, timeout=1)
def call_function_to_raise_exception():
    print("Trying to call problematic code...")
    raise_exception()

def raise_exception():
    raise Exception("foo")

while True:
    call_function_to_raise_exception()
#
# Dekorátor @retry
#

from funcy import retry

@retry(4, timeout=lambda delay: 2 ** delay, errors=Exception)
def call_function_to_raise_exception():
    print("Trying to call problematic code...")
    raise_exception()

def raise_exception():
    raise Exception("foo")

while True:
    call_function_to_raise_exception()

```

Caching výsledků funkcí

- ▶ Referenčně transparentní funkce
 - ◆ návratová hodnota(y) závisí pouze na parametrech
 - ◆ což znamená, že jde o zobrazení
 - ◆ a může být uloženo do mapy (cache)

Ukázky použití dekorátoru @cache a @lru_cache

```

#
# Dekorátor @cache
#

from time import time
from functools import cache

@cache
def fib(n):
    if n < 2:
        return n

```

```

        return fib(n-1) + fib(n-2)

max_n = 300

for i in range(20):
    if i % 5 == 0:
        fib.cache_clear()
        print(fib.cache_info())
        start = time()
        result = fib(max_n)
        end = time()
        print(result, end - start)
#
# Dekorátor @lru_cache
#

from time import time
from functools import lru_cache

@lru_cache
def fib(n):
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)

```

```

max_n = 300

for _ in range(10):
    start = time()
    result = fib(max_n)
    end = time()
    print(result, end - start)
#
# Dekorátor @lru_cache
#

from time import time
from functools import lru_cache

@lru_cache
def fib(n):
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)

```




```

max_n = 300

for i in range(20):
    if i % 5 == 0:
        fib.cache_clear()
        print(fib.cache_info())
        start = time()
        result = fib(max_n)
        end = time()
        print(result, end - start)

```

Persistentní datové struktury

-  Sdílení struktury
-  Problém přístupu z vláken
-  Knihovna Pyrsistent (a další)

Persistentní datové struktury: Pyrsistent

- ▶ Podporované datové struktury
 - ◆ pvector
 - persistentní vektor
 - obdoba Pythonovského seznamu
 - ◆ pset
 - persistentní množina
 - ◆ pmap
 - persistentní mapa
 - (asociativní pole)
 - ◆ plist
 - persistentní seznam
 - (interně dosti odlišný od vektorů)
 - ◆ pdeque
 - persistentní obousměrná fronta

Persistentní vektory

- ▶ Založeny na RRB stromech
 - ◆ RRB-Trees, Relaxed Radix Balanced Trees
 - ◆ přístup, insert: $\log_{32}(N)$
 - což je prakticky konstantní složitost
 - ◆ dtto pro persistentní množiny

```
#
# Persistentní vektory
#

from pyrsistent import v

vector1 = v(1, 2, 3)
print(vector1)
print(type(vector1))
#
# Persistentní vektory
#

from pyrsistent import v

vector1 = v(1, "foo", (1, 2, 3), None)
print(vector1)
print(type(vector1))

vector2 = vector1.append("Five!")
print(vector1)
print(type(vector1))

print(vector2)
print(type(vector2))
#
# Persistentní vektory
#

from pyrsistent import v

vector1 = v(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
print(vector1)
print(type(vector1))

print(vector1[3:7:2])
print(vector1[3::2])
print(vector1[:7:2])
print(vector1[::2])
```

Persistentní množiny

- ▶ Prvky nejsou přístupné přes index
- ▶ Test na existenci prvku v množině

```
#  
# Persistentní množiny  
#
```

```
from pyrsistent import s
```

```
set1 = s(1, 2, 3)  
set2 = set1.add(4)
```

```
print(set1)  
print(type(set1))
```

```
print(set2)  
print(type(set2))
```

```
#  
# Persistentní množiny  
#
```

```
from pyrsistent import s
```

```
set1 = s(1, 2, 3)  
set2 = set1.remove(1)
```

```
print(set1)  
print(type(set1))
```

```
print(set2)  
print(type(set2))
```

```
#  
# Persistentní množiny  
#
```

```
from pyrsistent import s
```

```
set1 = s(1, 2, 3)  
set2 = s(2, 3, 4)
```

```
print(set1)  
print(set2)
```

```
print("sjednoceni", set1 | set2)  
print("prunik", set1 & set2)  
print("rozdil", set1 - set2)  
print("rozdil", set2 - set1)
```

Persistentní mapy

- ▶ Problematika klíčů, které nejsou řetězci
 - ◆ syntaxe, ne sémantika