České vysoké učení technické v Praze FIT

Programování v Pythonu

Jiří Znamenáček

Příprava studijního programu Informatika je podporována projektem financovaným z Evropského sociálního fondu a rozpočtu hlavního města Prahy.

Praha & EU: Investujeme do vaší budoucnosti

Python – Funkce II

Argumenty napodruhé

I. Pozičních argumentů nemusí být dopředu známý počet – všechny přebytečné „schroustne“ v podobě tuplu speciální argument s operátorem `*`, který způsobí rozbalení předaných hodnot do **n-tice**:

```
def f(arg1, arg2, *args):
    print( "Argument 1: ", arg1 )
    print( "Argument 2: ", arg2 )
    print( "Následující argumenty: ", args )

>>> f( 1, 2, 3, 4, 5 )
Argument 1:  1
Argument 2:  2
Následující argumenty:  (3, 4, 5)
```

→ Protože `*args` si – narozdíl od přiřazování více hodnot najednou – uzurpuje všechny zbývající **poziční** argumenty (srovnej následující), musí být na konci. Cokoliv po něm musí být pojmenovaný (*keyword-only*) argument:

```
def f(arg1, *args, arg2):
    pass

>>> f( 1, 2, 3, 4, 5 )
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: f() needs keyword-only argument arg2
```

II. Podobně jako u pozičních argumentů můžeme použít „zkratku“ také pro získání blíže neurčeného počtu argumentů pojmenovaných. Jelikož ale nyní rozbalujeme dvojice *klíč – hodnota,* místo jedné hvězdičky použijeme operátor `**`, který rozbalí předané hodnoty do **slovníku**:

```
def f(arg1, arg2, **args):
    print( "Argument 1: ", arg1 )
    print( "Argument 2: ", arg2 )
    print( "Následující argumenty: ", args )

>>> f( 1, 2, jedna=1, dvě=2, tři=3 )
Argument 1:  1
Argument 2:  2
Následující argumenty:  {'tři': 3, 'dvě': 2, 'jedna': 1}
```

Právě takto se chová např. vestavěná funkce `print()`, jejíž definice totiž vypadá následovně:

```
print([object, ...], *, sep=' ', end='\n', file=sys.stdout)
```

→ `print()` tedy čeká libovolný – a předem neznámý! – počet pozičních argumentů, proto pojmenované musíme *vždy* předávat jako pojmenované (jinak „probublají“ do pozičních a budou tisknuty).

PS: Kombinací obého můžete funkci na vstupu vnutit přímo tuple / seznam / slovník:

```
def funkce(*args, **kwargs):
    for arg in args:
        print(arg)
    for kwarg in kwargs:
        print(kwarg, kwargs[kwarg])

>>> seznam = [1, 2, 3]
>>> slovnik = { 'jedna': 1, 'dvě': 2, 'tři': 3 }
>>> funkce( *seznam, **slovnik )
1
2
3
tři 3
dvě 2
jedna 1
```

→ Poznámka: Slovník se rozbalí na dvojce *klíč – hodnota*, takže klíč musí být řetězec (a když na to přijde, tak klidně i český, jsme v Python'u 3).

Pokud tento způsob předávání argumentů použijete, je zvykem používat právě uvedené názvy:

- **args** – pro výsledný tuple nepojmenovaných argumentů
- **kwargs** – pro výsledný slovník pojmenovaných argumentů

Viditelnost proměnných

I. Proměnné, funkce a další objekty zavedené (a též moduly importované) uvnitř funkce jsou viditelné právě jen tam – uvnitř funkce:

Program *scope.0.py* **:**

```
# toto je proměnná 'hodnota' na
hodnota = 'venku'

def zmena():
    # toto je proměnná 'hodnota
    hodnota = 'uvnitř'

print(hodnota)
zmena()
print(hodnota)
```

Výstup:

```
venku
venku
```

Pokud byste chtěli skutečně pracovat s proměnnou mimo tělo funkce (tedy zde na globální úrovni), musíte použít klíčové slovo `global`:

Program *scope.1.py* **:**

```
# toto je proměnná 'hodnota' na
hodnota = 'venku'

def zmena():
    # díky klíčovému slovu 'glo
    # proměnná 'hodnota' proměn
    global hodnota
    hodnota = 'uvnitř'

print(hodnota)
zmena()
print(hodnota)
```

Výstup:

```
venku
uvnitř
```

→ To tedy znamená, že si nemusíte hlídat kolize v názvech proměnných – uvnitř funkcí můžete vesele používat stejné názvy, jako mimo ně, a pokaždé budou označovat jiné objekty.

II. Pokud uvnitř funkce definujete další funkci (která tím pádem bude *lokální* vůči nadřazené) a z ní se budete chtít odkazovat na proměnnou v její nadřazené funkci, nikoli však v globálním kontextu, musíte ji specifikovat pomocí klíčového slova `nonlocal`:

Program *scope.2.py* **:**

```
# toto je proměnná 'x' na globá
x = 'vnější'

def outer():
    # toto je proměnná 'x' na m
    x = 'vnitřní'
    print(x)

    def inner():
        # díky klíčovému slovu
        # proměnná 'x' proměnné
        nonlocal x
        x = 'vnitřnější'

    inner()
    print(x)

print(x)
outer()
print(x)
```

Výstup:

```
vnější
vnitřní
vnitřnější
vnější
```

Z hlediska viditelnosti se tedy proměnné (které v Python'u zastupují libovolné objekty – skutečné „proměnné“, složitější datové struktury, ale třeba i funkce a třídy!) dělí na tři skupiny:

- **lokální** – jsou „vidět“ pouze v rámci bloku, v němž byly definovány (tj. např. v těle funkce), a ve všech jeho podřízených blocích
- **globální** – jsou dostupné v celém programu / skriptu / modulu, na jehož nejvyšší úrovni byly zavedeny
- **nelokální** – umožňují z podbloku odkazovat na proměnné stejného jména v nadřazeném bloku, ale nikoli až na globální úroveň

  → PS: Chcete-li přesto odkázat na globální proměnnou, stačí (ne)lokální přiřadit do nové proměnné, která už odkazuje na globální. Týká se to jak neproměnných (*immutable*), tak proměnných (*mutable*) typů.

Uzávěry (closures)

Funkce definované uvnitř jiných funkcí mají jednu podivuhodnou vlastnost – jsou-li z nadřazené funkce vráceny, nesou s sebou kontext platný v době jejich vrácení. Nejlépe osvětlí příklad:

```
# toto je vnější funkce, „generující“ uzávěr
def přičti(kolik):

    # tato vnitřní funkce vidí na proměnné nadřazené funkce, tj. i
    def vnitřní_fce(k_čemu):
        return kolik + k_čemu

    # vnitřní funkci vrátíme „upravenou“ o argument 'kolik' z doby
    return vnitřní_fce


# „Vyrobme“ funkci, která bude ke svému argumentu přičítat 5:
>>> přičti_5 = přičti(5)
# Je to opravdu funkce, a to ta očekávaná:
>>> přičti_5
<function vnitřní_fce at 0x00E2D108>

# Vyzkoušejme to:
>>> přičti_5(3)
8

# Na věci nic nezmění, když vyrobíme další funkci s jiným číslem:
>>> přičti_3 = přičti(3)
>>> přičti_3
<function vnitřní_fce at 0x00E2D0C0>
>>> přičti_3(3)
6
>>> přičti_5(10)
15
```

Jelikož každá z vnitřních funkcí si sebou nese kontext platný v okamžiku jejich vrácení, tedy i hodnoty proměnných (a tudíž i argumentů) v nadřazené funkci v tu chvíli, odpovídají výše uvedené přičítací funkce vlastně následujícím definicím:

```
def přičti_5(k_čemu):
    return 5 + k_čemu

def přičti_3(k_čemu):
    return 3 + k_čemu
```

Takovýmto funkcím, případně takovémuto chování obecně, se říká **uzávěry** (*closures*).

Hodnoty parametrů

Předávání parametrů do funkcí není v Python'u tak jednoduché, jako jinde: *Parametry v Python'u se předávají jako objekty podle svého typu – je-li typ neproměnný, budou předány hodnotou, je-li typ proměnný, budou předány odkazem.*

I. Neproměnný typ => předání hodnotou:

```
x = 3

def f(x):
    x = 2 * x


>>> f(x)
>>> print(x)
3
```

II. Proměnný typ => předání odkazem:

```
x = [3]

def f(x):
    x[0] = 2 * x[0]


>>> f(x)
>>> print(x)
[6]
```

→ Díky `x[0]` se v těle funkce „hrabeme“ přímo v odkazovaném objektu, a proto je výsledek takový, jaký je. Kdybychom místo toho napsali např. `x = 2 * x`, nestane se vůbec nic, protože bychom uvnitř funkce vytvořili novou lokální proměnnou, která by dočasně zastínila globální.

Na první pohled stejně podivně se chovají výchozí hodnoty parametrů – vyhodnocují se totiž pouze při prvním volání funkce, což má poněkud překvapivé důsledky, je-li jejich výchozí hodnotou proměnný (*mutable*) typ:

Program *default.1.py* **:**	**Výstup:**

```
def f(x, xs=[]):
    xs.append(x)
    return xs

print( f(1) )
print( f(2) )
print( f(3) )
```

→ Při prvním zavolání funkce *f* bude proměnná *xs* nastavena jako nálepka na místo paměti vyhrazené proměnnému datovému typu *seznam*. Při každém volání funkce bude ovšem obsah tohoto místa v paměti upraven.

```
[1]
[1, 2]
[1, 2, 3]
```

Pokud výše uvedené chování zrovna není to, co opravdu chceme, tak nejspíš nebudeme chvíli vůbec vědět, která bije. Odpovídající řešení v tomto případě je přepsat funkci následujícím způsobem:

Program *default.2.py* **:**

```
def f(x, xs=None):
    if xs is None:
        xs = []
    xs.append(x)
    return xs

print( f(1) )
print( f(2) )
print( f(3) )
```

Výstup:

```
[1]
[2]
[3]
```

Tahle „podivnost“ má ale i své bezvadné užití: Jsou-li vstupními parametry funkce hešovatelné typy, může si funkce při každém volání postupně budovat slovník vstupů a odpovídajících výstupů a natrefí-li znovu na stejné vstupy, může místo třeba-zrovna-velmi-náročného výpočtu použít již dříve získaný výsledek.

Lambda-funkce

Lambda-funkce jsou anonymní (tj. bezejmenné) funkce definované typicky v rámci jednoho řádku a skládající se pouze z jednoho výrazu, který je vyhodnocován ve chvíli volání funkce:

```
lambda [ARGUMENT[Y]]: VÝRAZ
```

Použití nachází všude tam, kde je požadována funkce, ale přitom není potřeba kvůli tomu zavádět pojmenovanou funkci na obecnější úrovni. Typicky například jako parametry třídících funkcí..

```
sorted( xs, key = lambda x: x[2] )      #

sorted( xs, key = lambda x: len(x) )    # sorted( xs, key=len )
```

..nebo jako parametry funkcí očekávajících funkci:

```
def mymap(xs, fn=lambda x: x):
    return [fn(x) for x in xs]


# Výchozí chování pojmenovaného argumentu 'fn' je „identita“:
>>> mymap( ['a', 'b', 'c', ] )
['a', 'b', 'c']

# „Přepišme“ identitu vestavěnou funkcí 'ord':
>>> mymap( ['a', 'b', 'c', ], ord )
[97, 98, 99]

# Podstrčme funkci nějakou naši vlastní „šílenost“:
>>> mymap( ['a', 'b', 'c', ], lambda x: chr(ord(x)+3) )
['d', 'e', 'f']
```

Příklad z předchozího slajdu o uzávěrech..

```
def přičti(kolik):
    def vnitřní_fce(k_čemu):
        return kolik + k_čemu
    return vnitřní_fce
```

..by šel pomocí lambda-funkce přepsat takto:

```
def přičti(kolik):
    return lambda k_čemu: kolik + k_čemu


>>> přičti_5 = přičti(5)
>>> přičti_5
<function <lambda> at 0x00DC9CD8>
>>> přičti_5(3)
8
```

Funkce jako „first-class citizen“

Chování funkcí z předchozích slajdů je přímým důsledkem toho, že funkce v

Python'u jsou tzv. *first-class object/citizen*. To především znamená, že je možné je předávat jako parametry do jiných funkcí, vracet je jako návratové hodnoty z funkcí a ukládat je v datových strukturách. Schrňme si to:

I. Jméno funkce je – stejně jako u „obyčejných“ proměnných – odkazem na objekt a tudíž:

```
def celociselne_deleni_se_zbytkem(co, cim):
    """Vrať výsledek celočíselného dělení a též zbytek po něm."""
    return co // cim, co % cim

>>> celociselne_deleni_se_zbytkem
<function celociselne_deleni_se_zbytkem at 0x00C06B70>

# Po přiřazení odkazuje „popiska“ fn na stejný objekt, jako původn
>>> fn = celociselne_deleni_se_zbytkem
>>> fn
<function celociselne_deleni_se_zbytkem at 0x00C06B70>

# ..můžeme ji tedy použít stejným způsobem k jejímu zavolání:
>>> fn(13, 4)
(3, 1)
```

II. Funkce mohou tudíž nepřekvapivě být součástí jiných datových struktur:

```
# uložení „odkazů“ na funkce do slovníku
>>> ds = {
...     'metoda_a': abs,
...     'metoda_b': len,
... }
>>> ds
{'metoda_a': <built-in function abs>, 'metoda_b': <built-in functi

# přístup k funkcím slovníkovou notací
>>> ds['metoda_a']( -3.14 )
3.14
>>> ds['metoda_b']( [1, 'dva', 3.0, ] )
3
```

III. Už jsme viděli, že návratová hodnota z funkce může být libovolného typu. A jedním z nich je stejně tak dobře právě i funkce:

```
# A) „normální“ varianta:
def slovo(i):
    def vnitrni(txt):
        return txt.split()[i]
    return vnitrni

# B) totéž za pomoci lambda-funkce:
def slovo(i):
    return lambda txt: txt.split()[i]


# Využití uzávěrů:
>>> slovo_1 = slovo(0)
>>> slovo_1( 'Ahoj, světe! Jak se máš?' )
'Ahoj,'
>>> slovo_3 = slovo(2)
>>> slovo_3( 'Ahoj, světe! Jak se máš?' )
'Jak'
```

→ Zde tedy např. přiřazení `slovo_1 = slovo(2)` efektivně „vyrobí“ objekt `slovo_1` typu funkce, který se bude chovat stejně jako jeho zavedení definicí:

```
def slovo_1(txt):
    return txt.split()[2]
```

IV. Stejně tak může být funkce vstupním parametrem do jiné funkce:

```
def mymap(fn, xs):
    return [ fn(x) for x in xs ]


# Příklad volání:
>>> mymap( abs, [2, -3, 5.4, -7.6, ] )
[2, 3, 5.4, 7.6]
>>> mymap( ord, 'Ahoj, světe!' )
[65, 104, 111, 106, 44, 32, 115, 118, 283, 116, 101, 33]
```

→ Jelikož `abs` i `ord` jsou v Python'u jednoduše odkazy na objekty typu funkce, volání funkce *mymap(FUNKCE, ITERABLE)* vrátí seznam (vybudovaný pomocí generátorové notace) složený z prvků vstupního iterovatelného objektu ITERABLE, na něž je aplikována uvedená FUNKCE.

Výše uvedené koncepty ilustruje souhrnně následující příklad:

```
# funkce pro umocňování, která vrací funkci
def mocni(na):
    return lambda co: co ** na

# seznam složený ze čtyř různých umocňovacích funkcí – x^0, x^1, x
mocneni = [ mocni(na=i) for i in range(4) ]

# „kontrolní“ výpis
>>> mocneni
[<function <lambda> at 0x00DD0B28>,
 <function <lambda> at 0x00DD0B70>,
 <function <lambda> at 0x00DD0BB8>,
 <function <lambda> at 0x00DD0C00>]
>>> mocneni[0]
<function <lambda> at 0x00DD0B28>

# volání
>>> mocneni[0](2)
1
>>> mocneni[1](2)
2
>>> mocneni[2](2)
4
>>> mocneni[3](2)
8
```

Příklad – emulace „switch/case“

Spoustě lidí chybí v Python'u konstrukce pro vícenásobné větvení (*switch/case*). Za pomoci slovníků a lambda-funkcí se dá v některých případech nahradit např. následující konstrukcí:

```
return {
    'klíč-1': VÝRAZ-1,
    'klíč-2': VÝRAZ-2,
    ...
}.get(KLÍČ[, DEFAULT])
```

→ Samozřejmě tato jednoduchá konstrukce neřeší společné chování pro více větví (tedy vynechané *break*). Kdyby ho někdo potřeboval a nechtěl použít standardní *if-elif-else*, pořád může provést vícenásobné mapování pomocí dalšího slovníku…

Příklad:

```
def case(k):
    return {
        'a': 'Stisknuta klávesa A.',
        'b': 'Stisknuta klávesa B.',
        'c': 'Stisknuta klávesa C.',
    }.get(k, 'Stisknuta neznámá klávesa.')


>>> case('')
'Stisknuta neznámá klávesa.'
>>> case('b')
'Stisknuta klávesa B.'
```

Rekurze

V Python'u je samozřejmě k dispozici **rekurze**. Klasický příklad na výpočet faktoriálu:

```
def faktorial(n):
    if n == 0:
        return 1
    else:
        return n * faktorial(n-1)


>>> faktorial(5)
120
>>> faktorial(15)
1307674368000
```

Pro připomenutí: 6! = 6•5! = 6•5•4! = 6•5•4•3! = 6•5•4•3•2! = 6•5•4•3•2•1! = 6•5•4•3•2•1

Rozklad úlohy na podúlohy pomocí rekurze je často ten nejpřirozenější možný. Z hlediska programování však ve stejnou chvíli většinou ten nejnešikovnější: Ve výše uvedeném případě faktoriálu se vlastně pro každé vstupní číslo spočítá zároveň i faktoriál každého čísla menšího. Při každém volání… Pokaždé znovu… A to je ještě výpočet faktoriálu „brnkačka“ v porovnání s takovým rekurzivním řešením hledání *n*-tého členu Fibonacciho posloupnosti:

```
def fib(n):
    if n<= 1:
        return n
    else:
        return fib(n-1) + fib(n-2)


>>> [fib(n) for n in range(11)]
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

V něm totiž každé zavolání funkce vyvolá *dva* nové rekurzivní kroky…

V takovou chvíli nastupuje tzv. **memoizace** (*memoization*) – zapamatování si již jednou nalezených výsledků a jejich použití v příštích výpočtech místo rekurzivního volání. Pro náš faktoriál tak můžeme napsat např.:

```
# slovník zapamatovaných hodnot
fakt = {}

# upravená funkce pro výpočet faktoriálu
def faktorial(n):
    # Nepočítali jsme už žádanou hodnotu?
    try:
        return fakt[n]
    # Ne, ještě ne:
    except KeyError:
        if n == 0:
            return 1
        else:
            # ..spočítejme ji
            val = n * faktorial_s_memoizaci(n-1)
            # ..a nyní už ji jako známou uložme a vraťme
            fakt[n] = val
            return val
```

PS: Počet rekurzivních kroků má z praktických důvodů pevnou hranici (většinou kolem tisíce).

Výpočet členů Fibonacciho posloupnosti jakožto nelineárního problému je zajímavější. Pro srovnání následuje praktické změření výpočetního času pro obě varianty:

Vstup *memoization/memoization.py* **:**

```
# výpočet bez zapamatování
def fibonacci_bez_memoizace(n):
    if n<= 1:
        return n
    else:
        return fibonacci_bez_memoizace(n-1) + fibonacci_bez_memoi


# slovník zapamatovaných hodnot
fib = {}

# upravená funkce pro výpočet n-tého členu Fibonacciho posloupnos
def fibonacci_s_memoizaci(n):
    # Nepočítali jsme už žádanou hodnotu?
    try:
        return fib[n]
    # Ne, ještě ne:
    except KeyError:
        if n <= 1:
            return n
        else:
            # ..spočítejme ji
            val = fibonacci_s_memoizaci(n-1) + fibonacci_s_memoiz
            # ..a nyní už ji jako známou uložme a vraťme
            fib[n] = val
            return val
```

Program *memoization/test.memoization.py* **:**

```
import timeit

"""
t.timeit() – zopakuje volání miliónkrát a spočítá průměr
t.repeat() – zopakuje volání miliónkrát a spočítá průměr; to celé
  =>
My zavoláme kód pouze jednou, ale třítisíckrát ho zopakujeme.
"""

# A)
print('Fibonacci bez zapamatování (5. a 15. a 20. člen):')
t = timeit.Timer("m.fibonacci_bez_memoizace(5); m.fibonacci_bez_m
out = t.repeat(3000, 1)
print( min(out) )

# B)
print('Fibonacci se zapamatováním (5. a 15. a 20. člen):')
t = timeit.Timer("m.fibonacci_s_memoizaci(5); m.fibonacci_s_memoi
out = t.repeat(3000, 1)
print( min(out) )
```

Výstup:

```
Fibonacci bez zapamatování (5. a 15. a 20. člen):
0.050101508701345665
Fibonacci se zapamatováním (5. a 15. a 20. člen):
1.1276665418336052e-05
```