Obecně se považuje za vhodnější vysvětlovat složité na jednoduchém, než naopak. Proto se snažím držet snadno pochopitelných funkcí, aby na nich ukázal příklady některých rysů programovacího jazyka python. Většina z nich se objevuje ve formě letmých zmínek, čímž se pro čtenáře otvírá vítaná příležitost pro samostudium.

Kromě povinného „Hello, Word“ neexistuje známější začátečnické klišé než funkce faktoriál. Bylo by asi nošením dříví do lesa, kdybychom začínali právě s ní…

Takže překvapivě se podívejme na tři způsoby zápisu funkce faktoriál.

Jako první se přirozeně nabízí toto:

# Zápis č.1. Standardní modul

from math import factorial

Za druhé se můžeme blýsknout trochou toho funkcionálního programování:

# Zápis č.2. Funkcionální zápis

from functools import reduce

from operator import mul

factorial = lambda n: reduce(mul, range(n, 0, -1))

A za třetí naprostá klasika, totiž rekurzivní zápis. Pokud jste se s ním ještě nesetkali, palec nahoru, protože jemu se lze vyhnout jen s mimořádným úsilím:

# Zápis č.3. Rekurze

def factorial(n):

return 1 if n == 1 else n \* factorial(n-1)

Nyní si představte, že ve svém programu předpokládáte časté volání funkce faktoriál a napadlo vás, že byste se mohli dostat do časové tísně, tj. například by vás zajímalo, jak dlouho bude trvat například výpočet faktoriálu čísla 1000, když ho budete chtít počítat 100krát za sebou.

Pokud se o to pokusíte právě pomocí rekurzivní funkce, trpce narazíte a představí se vám nesympatická hláška o dosažení maximální hloubky rekurze:

>>>RuntimeError: maximum recursion depth exceeded in comparison

Je velice důležité si uvědomit, že v pythonu nelze rekurzi používat beztrestně, rozhodně ne tam, kde předem nedokážete zaručit maximální počet volání funkce sama sebe.

Zjistit tento limit můžete pomocí:

import sys

sys.getrecursionlimit()

>>> 1000

a dokonce ho můžete i zvýšit pomocí:

sys.setrecursionlimit(1100)

Ale ze zjevných důvodů to není dobrý nápad. Těžko si představit případ, kdy vám nestačí 1000 volání, ale 1100 už ano.

Nejspolehlivější řešení je celou funkci přepsat pomocí iterace například takto:

# Zápis č.4. Iterace

def factorial(n):

f = 1

for i in range(1, n+1):

f \*= i

return f

Protože se nám ale tenhle zápis nehodí do krámu, předpokládejme na okamžik, že toužíme (z důvodů, které jsou jasné jen nám) vytvořit rekurzivní faktoriál pro celá čísla v rozmezí (1, 500), tedy tak, že nás výše uvedené omezení nijak nesvazuje.

Mělo by být zjevné, že:

Faktoriál (3) = 3 \* 2 \* 1

Faktoriál (4) = 4 \* (3 \* 2 \*1) = 4 \* Faktoriál(3)

Pokud se ptáte, co by mělo být zjevné, tak fakt, že faktoriál(3) je nutný pro výpočet faktoriálu(4) atd. Co z toho pro nás plyne? Například nás může napadnout, že jestliže známe faktoriál(3) , nemusíme ho znovu pracně počítat, ale místo toho by bylo lepší si ho uložit a použít ho pro řešení větších čísel. A tak samozřejmě až do nekonečna… Ne, jen do naší zlaté pětistovky, protože u čísla 1000 bychom tak či tak narazili.

Vytvoříme si tedy funkci cache, pomocí které odekorujeme náš rekurzivní faktoriál. Pokud jste se právě vyděsili, protože cizích slov v předchozí větě je víc než malé množství, nepropadejte panice. Vše má své přirozené vysvětlení, jak ostatně zjistíte po malé dávce samostudia. Dekorátory v pythonu jsou lehké a záživné čtení pro dlouhé zimní večery.

Po dvanácti dlouhých měsících…

Nyní když už je vám koncept dekoratérů jasný, nebudeme se zdržovat dalším výkladem a půjdeme rovnou k věci:

def cache(func): # func je funkce, kterou chceme kešovat

memo = {} # slovník, ve kterém klíč(key) je argument funkce faktoriál;

# a hodnota (value) je vypočtený faktoriál

def use\_cache(number): #tzv. closure, tj. funkce ve funkci

if number not in memo:

memo[number] = func(number)

return memo[number]

return use\_cache

Tolik by mělo stačit. Pokud bychom chtěli funkci cache použít bez dekorátoru, napsali bychom to takto:

factorial = cache(factorial)

Tento zápis se ale zdál Guidovi a spol příliš dlouhý, takže dlouze bádali a vedli vášnivé debaty, aby z nich nakonec vylezlo toto:

@cache

def factorial(n):

return 1 if n == 1 else n \* factorial(n-1)

Což je moc pěkné, ale také dost matoucí, jak možná zjistili ti z vás, kteří se s dekoratéry ještě nesetkali.

Teď si napíšeme krátký program, kterým otestujeme, zda jsme skutečně něčeho dosáhli, nebo jen zbytečně ztratili pár minut svého života.

from time import time

from random import randint

scope = (1, 500)

repeat = 1000

values = [randint(\*scope) for i in range(repeat) ]

start = time()

for value in values:

factorial(value)

print("Doba měřená s nekešovanou funkcí je: {} s.".

format(

round(time() - start, 5)))

Na svém počítači jsem získal tento výstup:

>>> Doba měřená s nekešovanou funkcí je: 1.25897 s.

>>> Doba měřená s kešovanou funkcí je: 0.0055 s.

Což je cca 200násobné zrychlení po konkrétní funkční obor.

Můžeme být tedy spokojeni? Pokud si odmyslíme ne zrovna vhodný příklad s funkcí faktoriál, mohlo by nás ještě trápit, že takto napsaný dekoratér není právě universální a lze ho použít jen pro jeden argument. Napravit to můžeme celkem snadno:

def cache(func):

memo = {}

def use\_cache(\*args):

if args not in memo:

memo[args] = func(\*args)

return memo[args]

return use\_cache

Je tedy výsledné řešení naprosto universální? Rozhodně ne, lze ho využít v různých obměnách za určitých podmínek:

* když je kešovaná funkce deterministická (tj. pro stejné argumenty, zadané ve stejném pořadí, musí vracet stejnou hodnotu),
* když její argumenty tvoří pouze tzv. immutable (neměnitelné) typy jako jsou například n-tice (typ tuple), řetězce (typ str) nebo celá čísla (typ int) apod.
* anebo když předpokládáme časté opakované volaní funkce se stejnými argumenty. To u rekurzivního faktoriálu nepředstavuje problém, zvláště pokud si uvědomíme, že při výpočtu faktoriálu 500 naplníme celou keš hodnotami faktoriálů od 1 do 500.

To ale na druhou stranu vyžaduje jistou opatrnost, protože při nevhodném použití keše můžeme snadno zaplnit velkou část paměti kvůli funkci, kterou dejme tomu nebudeme volat až tak často, aby to ospravedlňovalo přehnané nároky aplikace na paměť.

Už několikrát jsme zmínili, že rekurzní zápis není nejvhodnější, jestliže budeme chtít počítat faktoriál od 1000 nahoru. Povězme proto, že použijeme některý zápis 1,2 nebo 4, pro které limit pro počet rekurzí neplatí. Viz tato část z přiloženého zdrojového kódu:

# Odkomentujte funkci, kterou chcete testovat

# factorial = factorial\_1; print("Standardní modul.")

# factorial = factorial\_2; print("Funkcionální programování.")

# factorial = factorial\_3; print("Rekurze.")

factorial = factorial\_4; print("Iterace.")

Co se stane? Keš se nebude zaplňovat tak rychle, takže pokud se má její účinek projevit, musí být totéž číslo dosazeno do funkce alespoň dvakrát.

Takže kdy zvážit nasazení dekorátoru cache? Připomeňme si toto:

scope = (1, 500) # funkční obor

repeat = 10000 # počet volání funkce

Tyto dvě proměnné jsou zavedeny úmyslně. Po krátké úvaze nebo chvíli testování pravděpodobně dojdete k závěru, že funkce cache se nejlépe uplatní tehdy, kdy bude funkční obor, tedy proměnná scope, co nejmenší, zatímco počet volání, tedy proměnná repeat, co nejvyšší.

Je to ale všechno, co bychom měli brát v potaz? Rozhodně ne, do hry zde vstupuje neméně významné veličina, a tou je časová náročnost funkce. Čím je funkce pomalejší, tím pro ni může být kešování, ať už v jakékoliv formě, výhodnější. Časovou náročnost funkce zde ale zanedbáme (i když bychom rozhodně neměli) a zaměříme se jen na proměnné scope a repeat.

V předchozím textu, jsme zmínili, že pokud má mít funkce cache smysl, měla by být pro stejné číslo volána alespoň dvakrát. Matematicky vyjádřeno, zkoumáme faktor vhodnosti:

Vhodnost = repeat/scope

Můžeme předpokládat, že v intervalu od (0,1) nemáme valný důvod o dekorování uvažovat, zatímco od jedné nahoru můžeme. Víc lze zjistit experimenty s přiloženým zdrojovým kódem. Z několika namátkových testů pro různé hodnoty scope a repeat se zdá, že dosažené zrychlení odpovídá vypočítanému faktoru vhodnosti.