`Programování` 1 pro matematiky

7. cvičení, 18-11-2021

tags: `Programovani 1 2021`, `čtvrtek`

Obsah:

Farní oznamy

1. **Materiály k přednáškám** najdete v GitHub repozitáři https://github.com/PKvasnick/Programovani-1. Najdete tam také kód ke cvičením a pdf soubory textů cvičením.
2. **Domácí úkoly** Bude speciální část níže.

Kde se nacházíme Dnes dokončíme Pythonovské funkce, ale nejdříve si něco řekneme k domácím úkolům a dáme kratičkou kapitolku o znakových řetězcích. Opakování bude distribuované v různých částech.

Domácí úkoly

Úlohy z repozitáře

Tento týden jste dostali úkoly, které jsem vybral z repozitáře úkolů v ReCodExu a myslím, že už to víc neudělám. Úlohy mají optimalizované cíle nejen pro konkrétní úkol (například rychlost kód nebo spotřeba paměti), ale často i pro jazyk implementace. Proto jsem osobitně oznamoval, které testy nebudu brát na zřetel. Bohužel, nemám možnost upravit testy v zadání úkolu bez jeho duplikace.

To píšu nehledě na to, že řadu z vás úkol o hledání v seznamech nachytal na hruškách: při použití binárního prohledávání v zásadě nebyl problém splnit všechny testy. To udělalo jenom málo z vás, někteří dokonce ne ani po výslovném upozornění.

Je tady někdo, kdo nemá přístup k textům cvičení v repozitáři?

Vstup z konzole: více čísel na řádce

S tímto neměl problém prakticky nikdo. Bude krátký tutoriál o znakových řetězcích.

Znakové řetězce v Pythonu

Znakový řetězec je objet třídy `str`.

- Přístup k jednotlivým znakům a podřetězcům řetězce je stejný jako u seznamu

- Na rozdíl od řetězce je seznam *neměnný* (immutable), takže nefungují žádné funkce pro modifikaci řetězce. Fungují ale operátory pro vyhledávání v řetězci jako `index` a `count`.
- Funguje operátor `+` a logické operátory `==`/`!=` a `</>`, přičemž se používá lexikografické srovnání (podle UTF-8, ne podle češtiny, takže nebude respekovat např. české pořadí hlásek).

Metody třídy `str`

Upozornění Všechny vracejí novou hodnotu, původní řetězec se nemění.

Metoda	Popis
capitalize()	První písmeno na velké
casefold()	Všechna písmena na malá
center()	Vycentruje řetězec do pole s danou šířkou a výplní
count()	Počet výskytů znaku nebo řetězce v řetězci
endswith()	True pokud řetězec končí daným znakem/řetězcem
expandtabs()	Nahradí `\t` zadaným počtem mezer
find()	Vyhledá podřetězec a vrátí jeho pozici v řetězci
format()	Formátuje zadané hodnoty v řetězci
index()	Zděděné od seznamu, hledá pouze 1 znak
isalnum()	True pokud jsou všechny znaky alfanumerické
isalpha	True pokud jsou všechny znaky písmena
isascii()	True pokud jsou všechny znaky ascii
isdigit()	True pokud jsou všechny znaky číslicemi
islower()	True pokud jsou všechna písmena malá
isnumeric()	True, pokud jsou všechny znaky numerické
isspace()	True, pokud jsou všechny znaky ekvivalentní mezeře
isupper()	True pokud jsou všechna písmena velká
join()	Spojí prvky seznamu do řetězce proloženého daným řetězcem.
ljust()	Vrací doleva zarovnanou verzi řetězce s danou délkou a výplní
lower()	Skonvertuje písmena v řetězci na malá
lstrip()	Vrátí verzi řetězce ořezanou zleva na danou délku
partition()	Vyhledá řetězec a vrátí trojici všechno-před, hledaný-řetězec, všechno-za.
replace()	Vrátí řetězce, kde je podřetězec nahrazený jiným podřetězcem.
rfind()	Vrací pozici posledního výskytu
rindex()	Vrací pozici posledního výskytu
rjust()	Vrací doprava zarovnanou verzi s danou šířkou a výplní
rpartition()	Jako partition, ale hledá zprava
rsplit()	Jako split, ale hledá oddělovač zprava
rstrip()	Vrátí sprava ořezanou verzi řetězce
split()	Rozdělí řetězec u požadovaného separátoru a vrátí seznam

Metoda	Popis
splitlines()	Rozdělí seznam u znaků nového řádku a vrátí seznam
startswith()	True, pokud řetězec začíná daným podřetězcem
strip()	Vrátí ořezanou verzi řetězce
swapcase()	Promění velikost, malá na velká a naopak
upper()	Vrátí řetězec, ve kterém jsou malá písmena nahrazena velkými
zfill()	Vyplní řetězce zleva předepsaným počtem nul

Funkce

Permutace

Chceme vygenerovat všechny permutace množiny (rozlišitelných) prvků. Nejjednodušší je použít rekurzivní metodu:

```
1  def getPermutations(array):
2      if len(array) == 1:
3          return [array]
4      permutations = []
5      for i in range(len(array)):
6          # get all perm's of subarray w/o current item
7          perms = getPermutations(array[:i] + array[i+1:])
8          for p in perms:
9              permutations.append([array[i], *p])
10     return permutations
11 
12 print(getPermutations([1,2,3]))
```

Výhoda je, že dostáváme permutace setříděné podle původního pořadí.

Nevýhoda je, že dostáváme potenciálně obrovský seznam, který se nám musí vejít do paměti. Nešlo by to vyřešit tak, že bychom dopočítávali permutace po jedné podle potřeby?

Funkce jako plnoprávný Pythonovský objekt

Funkce může být přiřazována proměnným, předávána jiným funkcím jako parametr, a může být i návratovou hodnotou.

Funkce map a filter

Funkce map aplikuje hodnotu funkce na každý prvek seznamu nebo jiného iterovatelného objektu. Výsledkem je iterátor, takže pro přímé zobrazení je potReba ho převést na seznam např. pomocí `list`

```
>>> seznam = [[1,2], [2,3,4], [4,5,6,7], [7,8,9], [9,10]]
>>> delky = map(len, seznam)
>>> delky
<map object at 0x00000213118A2DC0>
>>> list(delky)
[2, 3, 4, 3, 2]
>>> list(map(sum, seznam))
[3, 9, 22, 24, 19]
```

Můžeme taky použít vlastní funkci:

```
>>> seznam = [[1,2], [2,3,4], [4,5,6,7], [7,8,9], [9,10]]
>>> def number_from_digits(cislice):
    quotient = 1
    number = 0
    for d in reversed(cislice):
        number = number * 10 + d
    return number

>>> list(map(number_from_digits, seznam))
[21, 432, 7654, 987, 109]
>>>
```

Podobně jako map funguje funkce `filter`: aplikuje na každý prvek seznamu logickou funkci a podle výsledku rozhodne, zda se má hodnota v seznamu ponechat.

```
>>> def len2(cisla) -> bool:
    return len(cisla)>2

>>> filter(len2, seznam)
<filter object at 0x0000021310E6C0A0>
>>> list(filter(len2, seznam))
[[2, 3, 4], [4, 5, 6, 7], [7, 8, 9]]

```

Funkce vytvářející funkce: dekorátory

Pojďme se vrátit k rekurzivní implementaci výpočtu Fibonacciho čísel:

```
# Fibonacci numbers recursive
def fib(n):
    if n < 2:
        return n
    else:
        return fib(n-1) + fib(n-2)

print(fib(5))

```

Chtěli bychom, aby se funkce volala jen v nevyhnutných případech, tedy když se počítá pro novou hodnotu `n`. Pro tento účel nebudeme upravovat funkci zevnitř, ale ji zabalíme:

- Vytvoříme funkci `memoize`, která jako parametr dostane původní "nahou" funkci `fib` a vrátí její upravenou verzi se zapamatováváním.

- Sice zatím nemáme úplně dobrou metodu jak si pamatovat sadu hodnot, pro které známe nějaký údaj, například sadu `n`, pro které známe `fib(n)`, ale můžeme si lehko pomoci dvojicí seznamů.

```python
# Momoised Fibonacci

def memoize(f):
    values = [0,1]
    fibs = [0,1]
    def inner(n):
        if n in values:
            return fibs[values.index(n)]
        else:
            result = f(n)
            values.append(n) # musime aktualizovat najednou
            fibs.append(result)
            return result
    return inner

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

print(fib(100))

```

Abychom si ukázali další použití dekorátorů, zkusme zjistit, jak roste počet volání `fib(n)` u rekurzivní verze. Dekorátor, který na to použijeme, využívá pro ten účel zřízený atribut funkce:

```python
# Dekorátor, počítající počet volání funkce
def counted(f):
    def inner(n):
        inner.calls += 1 # inkrementujeme atribut
        return f(n)
    inner.calls = 0 # zřizujeme atribut funkce inner
    return(inner)

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

@counted # pro porovnání přidáme i nerekurzivní verzi funkce
def fib2(n):
    if n < 2:
        return n
    else:
        f, fp = 1, 0
        for i in range(1,n):
            f, fp = f+fp, f
        return f
```

```
25
26 for i in range(30):
27     fib.calls = 0 # musíme resetovat počítadla
28     fib2.calls = 0
29     print(i, fib(i), fib.calls, fib2(i), fib2.calls)
30
```

Dekorátory umožňují změnit chování funkcí bez toho, aby bolo potřebné měnit kód, který je volá. Je to pokročilé téma, ale učí nás, že s funkcemi je možné dělat divoké věci. Není například problém zkombinovat dekorátory pro memoizaci a počítání volání:

```
1 @counted
2 @memoize
3 def fib(n):
4     ...
```

bude bez problémů fungovat.

Taky funkce...

Lambda-funkce

labmda funkce jsou kapesní bezejmenné funkce, které můžeme definovat na místě potřeby. Šetří práci například u funkcí jako `sort`, `min/max`, `map` a `filter`.

```
1 >>> seznam = [[0,10], [1,9], [2,8], [3,7], [4,6]]
2 >>> seznam.sort(key = lambda s: s[-1])
3 >>> seznam
4 [[4, 6], [3, 7], [2, 8], [1, 9], [0, 10]]
```

Generátory a příkaz yield

Už jsme se setkali s iterátory, např. víme, že dáme-li list comprehension do kulatých závorek místo hranatých, dostaneme namísto seznamu iterátor. Generátorem nazýváme funkci, která může fungovat jako iterátor - lze ji opakovaně volat, a ona pokaždé vrátí následující hodnotu z nějaké posloupnosti.

Příklad 1.

```
1  >>> def my_range(n):
2          k = 0
3          while k < n:
4              yield k
5              k += 1
6          return
7
8  >>> list(my_range(5)
9
10 [0,1,2,3,4]
```

Příklad 2 vám bude povědomý:

```
def read_list():
    while True:
        i = int(input())
        if i == -1:
            break
        yield i
    return

for i in read_list():
    print(f"Načetlo se číslo {i}.")

print("Konec cyklu 1")

for j in read_list():
    print(f"Teď se načetlo číslo {i}")

print("Konec cyklu 2")
```

Tento příklad je důležitý, protože umožňuje výrazně vylepšit strukturu kódu, který jste doteď vytvářeli pro domácí úkoly (ne že bych měl v plánu pronásledovat někoho, kdo bude načítat věci "postaru"). Generátor nám umožňuje oddělit načítání seznamu od jeho zpracování a to i v případě, že zpracováváme seznam sekvenčně, tedy ho nikam neukládáme.

Generátor permutací

Abychom nemuseli držet v pamětí seznam všech permutací, můžeme náš kód upravit tak, aby fungoval jako generátor:

```
def getPermutations(array):
    if len(array) == 1:
        yield array
    else:
        for i in range(len(array)):
            perms = getPermutations(array[:i] + array[i+1:])
            for p in perms:
                yield [array[i], *p]

for p in getPermutations([1,2,3]):
    print(p)
```