`Programování` 1 pro matematiky

8. cvičení, 25-11-2021

tags: `Programovani 1 2021`, `čtvrtek`

Obsah:

Farní oznamy

1. **Doufejme, že nás nezavřou**
2. **Materiály k přednáškám** najdete v GitHub repozitáři https://github.com/PKvasnick/Programovani-1. Najdete tam také kód ke cvičením a pdf soubory textů cvičením.
3. **Domácí úkoly**
   - Předešlý týden byly prázdniny, nedostali jste žádné nové úkoly.
4. **Kódy v repozitáři**

   Prosím pokud nenaleznete některé kódy v adresáři pro dané cvičení, hledejte v adresáři k předchozímu cvičení. Poruchy nastávají kvůli ztrátě synchronizace s druhou skupinou, kterou učím.

Kde se nacházíme

Opakování:

- znakové řetězce
- funkce

Příště: n-tice, množiny a slovníky

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.

- hledání / nahrazování v řetězci: `count`, `find`/`rfind`, `index`/`rindex`, `replace`,
- velikost písmen: `capitalize`, `lower`, `upper`, `title`, `casefold`, `swapcase`

- zarovnání a vyplnění v poli: `center`, `ljust` / `rjust`, `expandtabs`, `strip` / `lstrip` / `rstrip`, `zfill`
- formátování řetězce: `format`
- dotazy na obsah řetězce: , `startswith`, `endswith`, `isalnum`, `isalpha`, `isascii`, `isdecimal`, `isdigit`, `isnumeric`, `islower` / `isupper`, `isspace`
- spojování a rozdělování `join`, `split` / `rsplit`, `splitlines`, `partition` / `rpartition`,

---

Funkce

Příklady

Napište funkci, která

- vrátí řešení rovnice $2^x + x = 11$.

  Začneme tím, že zjevně $0 < x < 4$. a v tomto intervalu bude právě jedno řešení, protože funkce vlevo je rostoucí a vpravo konstantní. Programujeme, to, že řešení vidíte na první pohled, je dobré - máme kontrolu.

  Snažíme se udělat obecnější řešení:

```
def fun(x):
    return 2**x + x - 11

def eqn_solve(f, l, p, eps = 1.0e-6):
    """We expect f(l) < 0 < f(p)"""
    if f(l) > 0 or f(p) < 0:
        print("l a p musí ohraničovat oblast, kde se nachází kořen. ")
        return None
    while abs(l-p) > eps:
        m = (l+p)/2
        if f(m)<0:
            l = m
        else:
            p = m
    return m

def main():
    print(eqn_solve(fun, 0, 4))

main()
```

Lambda-funkce

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

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

Zkuste použít lambda funkci i jako parametr funkce `eqn_solve`.

```
print(eqn_solve(lambda x: 2**x + x - 20, 0, 4))
```

Permutace

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

```
def getPermutations(array):
    if len(array) == 1:
        return [array]
    permutations = []
    for i in range(len(array)):
        # get all perm's of subarray w/o current item
        perms = getPermutations(array[:i] + array[i+1:])
        for p in perms:
            permutations.append([array[i], *p])
    return permutations

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?

Znova Fibonacciho čísla

Jednoduchá rekurzivní implementace vychází z toho, že *Pythonovská funkce zná sebe samu*, takže ji v jejím těle můžeme volat:

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

print(fib(5))

```

Takováto implementace je velice srozumitelná, ale má vadu: zkuste si spočítat `fib(35)`. Důvodem je, že každé volání funkce vede ke dvěma dalším voláním, takže počet volání potřebný pro výpočet `fib(n)` exponenciálně roste. Existují dva způsoby, jak vyřešit takovýto problém s rekurzí:

Jedná se o primitivní, tedy odstranitelnou rekurzi, takže není složité vytvořit nerekurzivní implementaci.

```
# Fibonacci non-recursive

def fib(n):
    if n < 2:
        return n
    else:
```

```
        fpp = 0
        fp = 1
        for i in range(1,n):
            fp, fpp = fp + fpp, fp

    return fp

print(fib(35))

```

Můžeme rekurzivní funkci "vypomoct" zvenčí tak, že si někde zapamatujeme hodnoty, které se již vypočetly, a tyto hodnoty budeme dodávat z paměti a nebudeme na jejich výpočet volat funkci.

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.

```
# 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))

```

Také standardní knihovna Pythonu podporuje takovéto dekorátory:

```python
from functools import cache

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

print(fib(40))
```

Modul `functools` obsahuje také dekorátor `lru_cache`, který funguje jako `cache` ale je možné u něj nastavit, kolik posledních funkčních hodnot si bude pamatovat. Překvapivě dobré výsledky je možné dosáhnout i s poměrně malým zásobníkem - vyzkoušejte.

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

for i in range(30):
    fib.calls = 0 # musíme resetovat počítadla
    fib2.calls = 0
    print(i, fib(i), fib.calls, fib2(i), fib2.calls)

```

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:

```
@counted
@memoize
def fib(n):
    ...
```

bude bez problémů fungovat.

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]
# namísto map můžeme použít list comprehension:
[len(lst) for lst in seznam]
[2, 3, 4, 3, 2]
[sum(lst) for lst in 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]
... ale stejně tak:
>>> [number_from_digits(lst) for lst in seznam]
```

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]]
... ale stejně funguje i list comprehension:
>>> [cisla for cisla in seznam if len2(cisla)]
[[2, 3, 4], [4, 5, 6, 7], [7, 8, 9]]
```

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.

```
>>> def my_range(n):
        k = 0
        while k < n:
            yield k
            k += 1
        return

>>> list(my_range(5)

[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")
```

Lehce vytvoříme i generátor pro vstup čísel, oddělených mezerami.

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)
```

Příklady

A co generátor kombinací? Kombinace jsou něco jiné než permutace - permutace jsou pořadí, kombinace podmnožiny dané velikosti.

Začneme se standardní verzí, vracející seznam všech kombinací velikosti n. Všimněte si prosím odlišnosti oproti permutacím:

```
def combinations(a, n):
    result = []
    if n == 1:
        for x in a:
            result.append([x])
    else:
        for i in range(len(a)):
            for x in combinations(a[i+1:], n-1):
                result.append([a[i], *x])
    return result

print(combinations([1,2,3,4,5],2))

[[1, 2], [1, 3], [1, 4], [1, 5], [2, 3], [2, 4], [2, 5], [3, 4], [3, 5], [4,
5]]
```

Teď už lehce vytvoříme generátor:

```
def combi_gen(a, n):
    if n == 1:
        for x in a:
            yield [x]
    else:
        for i in range(len(a)):
            for x in combi_gen(a[i+1:], n-1):
                yield [a[i]] + x

for c in combi_gen([1,2,3,4,5],3):
    print(c)
```

Další variace: kombinace s opakováním pro bootstrap.