

# Programování 1 pro matematiky

---

## 8. cvičení, 24-11-2021

---

tags: Programování 1 2021, středa

---

### Obsah:

0. Farní oznamy
1. Domácí úkoly
2. Znakové řetězce
3. Funkce: opakování a rozšíření

### 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

- Na poslední úkoly přišlo docela málo řešení. Promluvíme.
- Předěšlý týden byly prázdniny, nedostali jste žádné nové úkoly.

#### 4. Kódy v repozitáři

Prosím pokud nenaleznete některé kódy v adresáři pro cvičení 8, hledejte v adresáři k předchozímu cvičení. Poruchy nastávají kvůli ztrátě synchronizace s druhou skupinou, kterou učím.

### Kde se nacházíme

Promluvíme si o domácích úkolech, narychlo probereme znakové řetězce a vrátíme se k funkcím.

Příště: n-tice, množiny a slovníky

---

## Domácí úkoly

---

### Úlohy z repozitáře

Předminulý 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 nachytl 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. Ale je čas dát krátký tutoriál o znakových řetězcích.

## Znakové řetězce v Pythonu

---

Znakový řetězec je objekt 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 respektovat např. české pořadí hlásek).

Metody třídy `str`

**Upozornění** Všechny vracejí novou hodnotu, původní řetězec se nemění.

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

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

## 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:

```

1  def fun(x):
2      return 2**x + x - 11
3
4  def eqn_solve(f, l, p, eps = 1.0e-6):
5      """We expect f(l) < 0 < f(p)"""
6      if f(l) > 0 or f(p) < 0:
7          print("l a p musí ohraničovat oblast, kde se nachází kořen. ")
8          return None
9      while abs(l-p) > eps:
10         m = (l+p)/2
11         if f(m)<0:
12             l = m
13         else:
14             p = m
15     return m
16
17 def main():
18     print(eqn_solve(fun, 0, 4))
19
20 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`.

```

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]]

```

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

```

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

```

## 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 seříděné 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:

```

1 # Fibonacci numbers recursive
2 def fib(n):
3     if n < 2:
4         return n
5     else:
6         return fib(n-1) + fib(n-2)
7
8 print(fib(5))
9

```

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.

```
1  # Fibonacci non-recursive
2
3  def fib(n):
4      if n < 2:
5          return n
6      else:
7          fpp = 0
8          fp = 1
9          for i in range(1,n):
10             fp, fpp = fp + fpp, fp
11
12     return fp
13
14 print(fib(35))
15
```

Můžeme rekurzivní funkci "vypomocit" 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.

Vyzkoušejte si tento kód:

```
1  from functools import cache
2
3  # Fibonacci numbers recursive
4  @cache
5  def fib(n):
6      if n < 2:
7          return n
8      else:
9          return fib(n-1) + fib(n-2)
10
11 print(fib(40))
```

To funguje a rychle. Funkce `cache` je *dekorátor*, tedy funkce, která nějak upravuje jinou funkci. Ukážeme si, jak to funguje.

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 lehkou pomoci dvojicí seznamů.

```
1  # Memoised Fibonacci
2
3  def memoize(f):
4      values = [0,1]
5      fibs = [0,1]
6      def inner(n):
7          if n in values:
```

```

8         return fibs[values.index(n)]
9     else:
10        result = f(n)
11        values.append(n) # musíme aktualizovat najednou
12        fibs.append(result)
13        return result
14    return inner
15
16    @memoize
17    def fib(n):
18        if n < 2:
19            return n
20        else:
21            return fib(n-1) + fib(n-2)
22
23    print(fib(100))
24

```

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:

```

1  # Dekorátor, počítající počet volání funkce
2  def counted(f):
3      def inner(n):
4          inner.calls += 1 # inkrementujeme atribut
5          return f(n)
6      inner.calls = 0 # zřizujeme atribut funkce inner
7      return(inner)
8
9  @counted
10 def fib(n):
11     if n < 2:
12         return n
13     else:
14         return fib(n-1) + fib(n-2)
15
16 @counted # pro porovnání přidáme i nerekurzivní verzi funkce
17 def fib2(n):
18     if n < 2:
19         return n
20     else:
21         f, fp = 1, 0
22         for i in range(1,n):
23             f, fp = f+fp, f
24         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 bylo 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.

## 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 potřeba ho převést na seznam např. pomocí `list`

```
1 | >>> seznam = [[1,2], [2,3,4], [4,5,6,7], [7,8,9], [9,10]]
2 | >>> delky = map(len, seznam)
3 | >>> delky
4 | <map object at 0x00000213118A2DC0>
5 | >>> list(delky)
6 | [2, 3, 4, 3, 2]
7 | >>> list(map(sum, seznam))
8 | [3, 9, 22, 24, 19]
9 | # namísto map můžeme použít list comprehension:
10 | [len(lst) for lst in seznam]
11 | [2, 3, 4, 3, 2]
12 | [sum(lst) for lst in seznam]
13 | [3, 9, 22, 24, 19]
```

Můžeme taky použít vlastní funkci:

```
1 | >>> seznam = [[1,2], [2,3,4], [4,5,6,7], [7,8,9], [9,10]]
2 | >>> def number_from_digits(cislice):
3 |     quotient = 1
4 |     number = 0
5 |     for d in reversed(cislice):
6 |         number = number * 10 + d
7 |     return number
8 |
9 | >>> list(map(number_from_digits, seznam))
10 | [21, 432, 7654, 987, 109]
11 | ... ale stejně tak:
12 | >>> [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.



```

1  >>> def len2(cisla) -> bool:
2      return len(cisla)>2
3
4  >>> filter(len2, seznam)
5  <filter object at 0x0000021310E6C0A0>
6  >>> list(filter(len2, seznam))
7  [[2, 3, 4], [4, 5, 6, 7], [7, 8, 9]]
8  ... ale stejně funguje i list comprehension:
9  >>> [cisla for cisla in seznam if len2(cisla)]
10 [[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.

```

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:

```

1  def read_list():
2      while True:
3          i = int(input())
4          if i == -1:
5              break
6          yield i
7      return
8
9  for i in read_list():
10     print(f"Načetlo se číslo {i}.")
11
12 print("Konec cyklu 1")
13
14 for j in read_list():
15     print(f"Teď se načetlo číslo {j}")
16
17 print("Konec cyklu 2")

```

## Generátor permutací

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

```
1 def getPermutations(array):
2     if len(array) == 1:
3         yield array
4     else:
5         for i in range(len(array)):
6             perms = getPermutations(array[:i] + array[i+1:])
7             for p in perms:
8                 yield [array[i], *p]
9
10 for p in getPermutations([1,2,3]):
11     print(p)
```