# `Programování` 1 pro matematiky

# 12. cvičení, 03-01-2024

**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** - přišlo hodně málo řešení, ale to nevadí.
3. **Kde se nacházíme** Končíme. Pokud budeme mít cvičení příští týden, bude přehled knihoven Pythonu.

---

# Domácí úkoly

## K-ciferná čísla

> Na jediném řádku standardního vstupu jsou zadána dvě kladná celá čísla *K*, *N*. Napište program, který najde a vypíše všechna *K*-ciferná čísla s ciferným součtem rovným *N*. Každé číslo vypište na samostatný řádek, na jejich pořadí nezáleží. Zápis čísla nesmí začínat vedoucími nulami.

Tady přišlo docela dost neúspěšných řešení.

- Řešení, která zkouší projít všechna k-ciferná čísla a najít mezi nima ty, které mají správný ciferný součet, nemohou pro větší k fungovat.
- Rekurze je přirozený přístup, ale není samospasitelný:  je potřeba ustřežit okrajové případy, jako doplnění nul nebo devítek na konec čísla.

### Rekurzivní řešení

může vypadat nějak takto:

```
import sys


length, digit_sum = [int(s) for s in input().split()]


def list_numbers(prev: list, first: bool = False):
    if first:
```

```
        digits = list(range(1,10))
    else:
        digits = list(range(10))
    missing_digits = length - len(prev)
    missing_sum = digit_sum - sum(prev)
    if missing_sum == 0:
        print("".join(map(str, prev)).ljust(length, "0"))
        return
    if 9 * missing_digits == missing_sum:
        print("".join(map(str, prev)).ljust(length, "9"))
        return
    if 9 * missing_digits <= missing_sum:
        return
    new_digits = [d for d in digits if d <= missing_sum]
    for d in new_digits:
        list_numbers([*prev, d])


list_numbers([], True)
```

## Zásobník

My si ukážeme řešení, využívající zásobník. V hlavním cyklu vybereme číslo ze zásobníku, pokud je dobrým předchůdcem, vytvoříme jeho potomky a uložíme je na zásobník. Pokud je to hledané číslo, vypíšeme jej. Skončíme, když je zásobník prázdný:

```
import sys

length, digit_sum = [int(k) for k in input().split()]
if digit_sum > 9*length:
    print()
    sys.exit()

# We work with strings, not with ints.
def str_sum(s:str) -> int:
    return sum(int(d) for d in s)


digits = [i for i in reversed(range(10))]
# Non-recursive
stack = [str(i) for i in digits[:-1]]

while stack:
    cand = stack.pop()
    cand_sum = str_sum(cand)
    missing_digits = length - len(cand)
    missing_sum = digit_sum - cand_sum
    if missing_digits < 1 and missing_sum > 0:
        continue
    if missing_sum == 0:
        print(cand.ljust(length, "0"))
        continue
    for d in digits:
```

```
        if cand_sum + d <= digit_sum:
            stack.append(cand + str(d))
```

Nebude zásobník příliš velký? V zásadě máme v zásobníku řádově k*10 položek, protože uchováváme pouze jednu větev stromu. Je to rozhodně lepší než odpovídající hloubka rekurze.

Toto je jedna z věcí, které se naučíte v příštím semestru.

## Maticová třída

Tento domácí úkol měl sloužit k procvičení vytváření tříd a jejich metod, a řešení mohlo vypadat nějak takto:

```
class Matrix:

    def __init__(self, ll):
        self.matrix = ll

    def __repr__(self):
        return '\n'.join(' '.join(str(val) for val in row) for row in
self.matrix)

    def vals(self):
        return self.matrix

    def dims(self):
        return (len(self.matrix), len(self.matrix[0]))

    def __add__(self, other):
        result = []
        for i, row in enumerate(self.matrix):
            result.append([x + y for x, y in zip(row, other.matrix[i])])
        return Matrix(result)

    def __sub__(self, other):
        result = []
        for i, row in enumerate(self.matrix):
            result.append([x - y for x, y in zip(row, other.matrix[i])])
        return Matrix(result)

    def __mul__(self, other):
        if isinstance(other, Matrix):
            result = [[0] * len(other.matrix[0]) for _ in
range(len(self.matrix))]
            for i in range(len(self.matrix)):
                for j in range(len(other.matrix[0])):
                    for k in range(len(other.matrix)):
                        result[i][j] += self.matrix[i][k] * other.matrix[k][j]
            return Matrix(result)
        else:
            result = [[x * other for x in row] for row in self.matrix]
            return Matrix(result)

    def is_symmetric(self):
```

```
        if len(self.matrix) != len(self.matrix[0]):
            return False
        for i in range(len(self.matrix)):
            for j in range(i + 1, len(self.matrix[0])):
                if self.matrix[i][j] != self.matrix[j][i]:
                    return False
        return True


def zero_matrix(r, c):
    return Matrix([[0] * c for _ in range(r)])


def identity_matrix(n):
    return Matrix([[1 if i == j else 0 for j in range(n)] for i in range(n)])
```

---

# Funkční objekty

Vysvětlili jsme si už, že funkce jsou plnoprávnými obyvateli Pythonu, a tedy je můžeme přiřazovat proměnným, ukládat do seznamů či slovníků, a používat je jako parametry funkcí.

## Příklad 1. Součet položek ve dvou seznamech

Máme dva seznamy `a` a `b`, chceme seznam s položkami `a[0]+b[0], a[1]+b[1]` atd.

Možností je několik, liší se množstvím a čitelností kódu a také rychlostí.

```
# Součet položek ve dvou seznamech

a = [1, 2, 3, 4, 5]
b = [5, 4, 3, 2, 1]

# Definice funkce - příliš mnoho kódu pro jednoduchou věc
def soucet(x,y):
    return x+y

# List comprehension
print([soucet(x,y) for x,y in zip(a,b)])

# ale vlastně vůbec nepotřebujeme funkci!
print([x+y for x, y in zip(a,b)])

# Funkci ale potřebuje map - to je méně čitelné, ale výkonnější
print(list(map(soucet, a,b)))

# Čitelnější je použít namísto definované funkce soucet lambda funkci:
print(list(map(lambda x,y: x + y, a, b)))

# Základní funkce máme předdefinovány v modulu operator:
import operator
print(list(map(operator.add, a, b)))
```

Modul `operator` obsahuje funkční ekvivalenty pro běžné binární operátory:

| Operation | Syntax | Function |
|---|---|---|
| Addition | `a + b` | `add(a, b)` |
| Concatenation | `seq1 + seq2` | `concat(seq1, seq2)` |
| Containment Test | `obj in seq` | `contains(seq, obj)` |
| Division | `a / b` | `truediv(a, b)` |
| Division | `a // b` | `floordiv(a, b)` |
| Bitwise And | `a & b` | `and_(a, b)` |
| Bitwise Exclusive Or | `a ^ b` | `xor(a, b)` |
| Bitwise Inversion | `~ a` | `invert(a)` |
| Bitwise Or | `a \| b` | `or_(a, b)` |
| Exponentiation | `a ** b` | `pow(a, b)` |
| Identity | `a is b` | `is_(a, b)` |
| Identity | `a is not b` | `is_not(a, b)` |
| Indexed Assignment | `obj[k] = v` | `setitem(obj, k, v)` |
| Indexed Deletion | `del obj[k]` | `delitem(obj, k)` |
| Indexing | `obj[k]` | `getitem(obj, k)` |
| Left Shift | `a << b` | `lshift(a, b)` |
| Modulo | `a % b` | `mod(a, b)` |
| Multiplication | `a * b` | `mul(a, b)` |
| Matrix Multiplication | `a @ b` | `matmul(a, b)` |
| Negation (Arithmetic) | `- a` | `neg(a)` |
| Negation (Logical) | `not a` | `not_(a)` |
| Positive | `+ a` | `pos(a)` |
| Right Shift | `a >> b` | `rshift(a, b)` |
| Slice Assignment | `seq[i:j] = values` | `setitem(seq, slice(i, j), values)` |
| Slice Deletion | `del seq[i:j]` | `delitem(seq, slice(i, j))` |
| Slicing | `seq[i:j]` | `getitem(seq, slice(i, j))` |
| String Formatting | `s % obj` | `mod(s, obj)` |
| Subtraction | `a - b` | `sub(a, b)` |

| Operation | Syntax | Function |
|---|---|---|
| Truth Test | `obj` | `truth(obj)` |
| Ordering | `a < b` | `lt(a, b)` |
| Ordering | `a <= b` | `le(a, b)` |
| Equality | `a == b` | `eq(a, b)` |
| Difference | `a != b` | `ne(a, b)` |
| Ordering | `a >= b` | `ge(a, b)` |
| Ordering | `a > b` | `gt(a, b)` |

V modulu najdete i mnoho dalších užitečných věcí, takže neškodí nahlédnout do dokumentace.

## Příklad 2: Třídění a další operace, vyžadující klíč

```
# Třídění a další operace, vyžadující klíč

# Klíč můžeme lehko definovat pomocí lambda funkce.
>>> k = ["kočka", "sedí", "na", "okně"]
>>> sorted(k, key=lambda x: len(x))

['na', 'sedí', 'okně', 'kočka']

# Funkce min také srovnává a tedy u ní můžeme definovat klíč:
>>> min(k, key=lambda x: len(x))

'na'

# Setřídění podle položky jsme už trénovali:
>>> p = [(1,'leden'), (2,'unor'), (4,'duben')]
>>> sorted(p, key=lambda x: x[1])

[(4, 'duben'), (1, 'leden'), (2, 'unor')]

# Konečně si můžeme vypomoct modulemt operator:
>>> import operator
>>> sorted(p, key = operator.itemgetter(1))

[(4, 'duben'), (1, 'leden'), (2, 'unor')]
```

## Příklad 3: Implementace funkce `itemgetter`

Ukážeme si dvě možné implementace pro `oparator.itemgetter` . První možností je funkce, která vrací funkci:

```
# itemgetter as a function

def itemgetter(k):
    return lambda a: a[k]
```

```
def main():
    a = (1,2)
    print(itemgetter(1)(a))
    u = [(1,5), (2,4), (3,3)]
    print(sorted(u, key = itemgetter(1)))
    print(sorted(u, key = itemgetter(0)))

if __name__ == "__main__":
    main()

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

Druhou možností je implementovat `itemgetter` jako funktor, tedy objekt, který lze volat jako funkci:

```
# itemgetter as functor

class itemgetter:
    def __init__(self, k):
        self.k = k
        self.fun = lambda a: a[self.k]

    def __call__(self, a):
        return self.fun(a)

def main():
    a = (1,2)
    print(itemgetter(1)(a))
    u = [(1,5), (2,4), (3,3)]
    print(sorted(u, key = itemgetter(1)))
    print(sorted(u, key = itemgetter(0)))

if __name__ == "__main__":
    main()

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

## Příklad 4: Kompozice funkcí

Mějme funkce `f(x)` a `g(x)`, chceme implementovat funkci `compose(f,g)`, která vrací složenou funkci `f∘g`, tedy funkci, vracející hodnotu `f(g(x))` a ne hodnotu této funkce pro nějaký argument.

```
def compose(f,g):
    def _comp(x):
        return f(g(x))
    return _comp

print(compose(int, abs)(-4.5))
```

### Vnitřní funkce

Už jsme několikrát viděli, že vytváření funkcí jinými funkcemi je docela silná zbraň, zejména díky tomu, že vytvořené funkce si sebou nesou prostředí mateřské funkce ve stavu svého vzniku. Tím se podobají na třídy.

```
def f():
    n = 0
    def in_f():
        nonlocal n   # n pochází z nadřazeného jmenného prostoru
        n += 1       # musíme explicitně deklarovat, protože používáme
        return n     # na pravé straně výrazu
    return in_f

>>> a = f()
>>> a()
1
>>> a()
2
>>> b = f()
>>> b()
1
>>> b()
2
>>> a()
3
>>>
```

Zjevně funkce `a()` a `b()` mají nezávislé vnitřní stavy.