# `Programování` 1 pro matematiky

# 10. cvičení, 03-12-2024

---

**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**
   - V posledních týdnech chodí méně řešení. Máme poměrně velké zastoupení téměř totožných řešení: v zásadě správných, ale výrazně vylepšitelných.
   - Všichni máte dostatek bodů pro zápočet, ale domácí úkoly budu zadávat až do konce semestru, abyste měli možnost procvičovat programovací svaly.
3. **Konec semestru se blíží**
   - Máme cvičení v lednu? (Nápověda: hodilo by se)
   - Napište mi, pokud **potřebujete revidovat** některé své řešení a máte pocit, že jsem mu zatím nevěnoval dostatečnou pozornost nebo vás hodnotil nespravedlivě.
   - Napište mi také, pokud **nedosahujete potřebný bodový limit** pro zápočet a chtěli byste si jej dodatečně vylepšit.

**Kde se nacházíme**

Dnes se ještě vrátíme k funkcím a funkčním objektům a pak začneme mluvit o třídách v Pythonu.

---

## Domácí úkoly

Měli jste poněkud těžší domácí úkoly, ale řešení, která přišla, byla povětšině velmi dobrá.

**Dobrá čísla**

Toto byl snadný úkol a také jste s ním neměli vážnější problémy. Převažující škola myšlení vyprodukovala toto řešení:

```
def dobre_cislo(n: int) -> bool:
    n_puvodni = n
    while cislo > 0:
```

```
        cislo, cifra = divmod(cislo, 10)
        if cifra == 0 or n_puvodni % cifra != 0:
            return False
    return True


def pocet_dobrych(n: int) -> int:
    pocet = 0
    for i in range(1, n+1):
        if dobre_cislo(i):
            pocet += 1
    return pocet


n = int(input())
print(pocet_dobrych(n))
```

To je v zásadě správný kód:

1. Je logické začít řešení tím, že si definujeme funkci, která zjistí, zda je dané číslo dobré. Takováto funkce je samostatně testovatelný kus kódu s jasně definovanou funkcionalitou.
2. Pak stačí funkci aplikovat na zadaný číselný interval. To je lehké, ale udělat to pomocí další funkce je docela hloupé: vytváříme velký otevřený cyklus, který lze lehko uzavřít:

```
pocet_dobrych = sum(dobre_cislo(i) for i in range(1, n+1))
```

V tomto případě běží cyklus uvnitř funkce, tedy v C, a nemusí se interpretovat Python.

**Chybějící čísla**

To byl druhý jednoduchý úkol a šlo v něm o to, abyste čísla dokázali najít efektivně. V zásadě máte dvě možnosti:

1. Počítadlo:

```
def main() -> None:
    counter = [0] * 101 # pro pohodlí
    while True:
        n = int(input())
        if n == -1:
            break
        counter[n] = 1
    print([i for i, v in enumerate(counter[1:]) if v > 0])


if __name__ == "__main__":
    main()
```

2. Množiny:

```
def main() -> None:
    vse = set(range(1,101))
    videno = set()
    while True:
```

```
        n = int(input())
        if n == -1:
            break
        videno.add(n)
    nevideno = vse - videno
    print(sorted(nevideno))   # máme množinu, musíme utřídit


if __name__ == "__main__":
    main()
```

Našlo se i takovéto řešení:

```
def main() -> None:
    vstup = []
    while True:
        n = int(input())
        if n == -1:
            break
        vstup.append(n)
    chybejici = [i for i in range(1, 101) if i not in vstup]
    print(chybejici)


if __name__ == "__main__":
    main()
```

Je to v podstatě hezký kód, jenomže je to celé špatně.

- úplně zbytečně si pamatujeme vstupná data
- následně 100-krát neefektivně prohledáváme potenciálně velký vstupný seznam.

Ke konci tohoto semestru už víte, že `in` aplikované na seznam znamená prohledávání prvek po prvku.

**K-ciferná čísla**

Toto je těžší úloha na rekurzi. A je to úloha na rekurzi, protože si uvědomíme, že

1. Když už máme první číslice K-ciferného čísla, stojíme před stejným problémem jako na začátku: máme zbylá místa obsadit číslicemi tak, že doplníme ciferný součet na požadovanou hodnotu.
2. Úloha má charakter prohledávání stromu. Kořen stromu je prázdný seznam, jeho potomky jsou seznamy `[1], [2], ... [9]`, jejich potomky seznamy `[1,0], [1,1], ..., [2,0], [2,1], ..., [9,0], [9,1],..., [9,9]`. Koncové uzly obsahují K-ciferná čísla s požadovaným ciferným součtem.

Složitost této úlohy je daná tím, že se požaduje řešení těžkých vstupů, např.

```
- test 2: 8 70
- test 3: 77 693
- test 5: 15 134
```

U testu 2 máme čísla ze samých devítek s jednou 7 nebo dvěma 8. U testu 3 máme čísla ze samých devítek. U testu 5 máte samé devítky a jednu osmičku. Pro řešení to znamená, že musíme dokázat rychle ukončit procházení větví stromu, které obsahují samé devítky (nebo samé nuly). Pak už to jde docela lehko:

```
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)
```

Toto řešení sice funguje, ale v obecném případě je jeho použitelnost omezená na relativně malá K, jinak exploduje počet rekurzivních volání - v každé úrovni volá funkce sebe sama devětkrát.

V příštím semestru se budeme učit, jak tuto úlohu vyřešit nerekurzivně pomocí zásobníku.

---

# Opakování: Funkce a generátory

## Rekurze

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

```
def getPermutations(array):
    if len(array) == 1:   # Base case
        return [array]
    permutations = []
    for i in range(len(array)):
        # Aktuální prvek + všechny permutace pole bez něj
        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?

## Jiný příklad: Kombinace

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:    # Base case: velikost 1 - vracíme seznam prvků
        for x in a:
            result.append([x])
        return result
    for i in range(len(a)):
        # Aktuální prvek + kombinace zbývajících o délce n-1
        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]]
```

#### Složitější kombinatorika

Permutace a kombinace s opakováním: Zatímco u běžných permutací a kombinací pracujeme s množinami, u permutací a kombinací s opakováním pracujeme s multimnožinami:

- Permutace s opakováním je permutace multisetu, tedy některé prvky jsou stejné a pořadí, lišící se záměnou stejných prvků jsou identická.
- Kombinace s opakováním jsou multisety, tedy prvek můžeme vybrat do kombinace vícekrát.

Takováto kombinatorika je užitečná tam, kde nám umožní vybrat menší základní prostor, ve kterém něco hledáme.

Teď se ale vrátíme k problému s velkými seznamy permutací a kombinací a ukážžeme si, že se bez nich umíme obejít.

---

## Generátory a příkaz yield

Dáme-li list comprehension do kulatých závorek místo hranatých, dostaneme namísto seznamu generátor.

```
>>> r = (x for x in range(20) if x % 3 == 2)
>>> r
<generator object <genexpr> at 0x000001BC701E9BD0>
>>> for j in r:
...     print(j)
...
2
5
8
11
14
17
```

Následující ukázka demonstruje, jak Python interaguje s iterátorem:

```
>>> s = (x for x in range(3))
>>> next(s)
0
>>> next(s)
1
>>> next(s)
2
>>> next(s)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

`next(it)` vrací další hodnotu iterátoru, a pokud už další hodnota není, vyvolá iterátor výjimku `StopIteration`. To je standardní chování iterátoru.

**Generátorem** nazýváme funkci, která může fungovat jako iterátor - lze ji opakovaně volat, a ona pokaždé vrátí následující hodnotu z nějaké posloupnosti. Namísto `return` má generátor příkaz `yield` nebo `yield from`:

```
def count(start: int = 0):
    i = start
    while True: # nekonečný iterátor
        yield i
        i += 1

def iterate(numbers: list[int]):
    yield from numbers # ekvivalentní (i for i in numbers)
```

Generátor je funkce, která trpělivě čeká na pozadí, až ji požádáme o další hodnotu. Pokud už další hodnotu nemá, vrátí výjimku "StopIteration".

**Příklad 3** 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 {j}")

print("Konec cyklu 2")
```

Takto můžeme lehce oddělit cyklus zpracování dat od cyklu jejich načítání.

**Příklad**: kombinace a permutace

Použijeme implementace pro permutace a kombinace z minulého cvičení pro implementaci příslušných generátorů.

Funkce pro počítání permutací a kombinací nám dávají potenciálně obrovské seznamy. Nešlo by je implementovat jako generátory?

```
def getPermutations(array):
    if len(array) == 1:   # Base case
        yield array
    for i in range(len(array)):
        # Aktuální prvek + všechny permutace pole bez něj
        for p in getPermutations(array[:i] + array[i+1:]):
            yield [array[i], *p]

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

Podobně pro kombinace:

```
def combinations(a, n):
    if n == 1:    # Base case: velikost 1 - vracíme seznam prvků
        for x in a:
            yield [x]
    for i in range(len(a)):
        # Aktuální prvek + kombinace zbývajících o délce n-1
        for x in combinations(a[i+1:], n-1):
            yield [a[i], *x]

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

Tyto funkce jsou implementovány v modulu `itertools`:

Mnoho iterátorů najdete v modulu `itertools`.

**Infinite iterators:**

| Iterator | Arguments | Results | Example |
|---|---|---|---|
| `count()` | start, [step] | start, start+step, start+2*step, ... | `count(10) --> 10 11 12 13 14 ...` |
| `cycle()` | p | p0, p1, ... plast, p0, p1, ... | `cycle('ABCD') --> A B C D A B C D ...` |
| `repeat()` | elem [,n] | elem, elem, elem, ... endlessly or up to n times | `repeat(10, 3) --> 10 10 10` |

**Iterators terminating on the shortest input sequence:**

| Iterator | Arguments | Results | Example |
|---|---|---|---|
| `accumulate()` | p [,func] | p0, p0+p1, p0+p1+p2, ... | `accumulate([1,2,3,4,5]) --> 1 3 6 10 15` |
| `chain()` | p, q, ... | p0, p1, ... plast, q0, q1, ... | `chain('ABC', 'DEF') --> A B C D E F` |
| `chain.from_iterable()` | iterable | p0, p1, ... plast, q0, q1, ... | `chain.from_iterable(['ABC', 'DEF']) --> A B C D E F` |
| `compress()` | data, selectors | (d[0] if s[0]), (d[1] if s[1]), ... | `compress('ABCDEF', [1,0,1,0,1,1]) --> A C E F` |
| `dropwhile()` | pred, seq | seq[n], seq[n+1], starting when pred fails | `dropwhile(lambda x: x<5, [1,4,6,4,1]) --> 6 4 1` |
| `filterfalse()` | pred, seq | elements of seq where pred(elem) is false | `filterfalse(lambda x: x%2, range(10)) --> 0 2 4 6 8` |
| `groupby()` | iterable[, key] | sub-iterators grouped by value of key(v) | |

| Iterator | Arguments | Results | Example |
|---|---|---|---|
| `islice()` | seq, [start,] stop [, step] | elements from seq[start:stop:step] | `islice('ABCDEFG', 2, None) --> C D E F G` |
| `pairwise()` | iterable | (p[0], p[1]), (p[1], p[2]) | `pairwise('ABCDEFG') --> AB BC CD DE EF FG` |
| `starmap()` | func, seq | func(*seq[0]*), *func*(seq[1]), ... | `starmap(pow, [(2,5), (3,2), (10,3)]) --> 32 9 1000` |
| `takewhile()` | pred, seq | seq[0], seq[1], until pred fails | `takewhile(lambda x: x<5, [1,4,6,4,1]) --> 1 4` |
| `tee()` | it, n | it1, it2, ... itn splits one iterator into n | |
| `zip_longest()` | p, q, ... | (p[0], q[0]), (p[1], q[1]), ... | `zip_longest('ABCD', 'xy', fillvalue='-') --> Ax By C- D-` |

**Combinatoric iterators:**

| Iterator | Arguments | Results |
|---|---|---|
| `product()` | p, q, ... [repeat=1] | cartesian product, equivalent to a nested for-loop |
| `permutations()` | p[, r] | r-length tuples, all possible orderings, no repeated elements |
| `combinations()` | p, r | r-length tuples, in sorted order, no repeated elements |
| `combinations_with_replacement()` | p, r | r-length tuples, in sorted order, with repeated elements |

| Examples | Results |
|---|---|
| `product('ABCD', repeat=2)` | `AA AB AC AD BA BB BC BD CA CB CC CD DA DB DC DD` |
| `permutations('ABCD', 2)` | `AB AC AD BA BC BD CA CB CD DA DB DC` |
| `combinations('ABCD', 2)` | `AB AC AD BC BD CD` |
| `combinations_with_replacement('ABCD', 2)` | `AA AB AC AD BB BC BD CC CD DD` |

**Příklad**: Erastothénovo síto

Co dělá tento kód?

```
from itertools import count

def sieve(s):
    n = next(s)
    yield n
```

```
    yield from sieve(i for i in s if i % n != 0)


primes = sieve(count(start=2))

n = 0
for p in primes:
    print(p)
    n += 1
    if n > 200:
        break
```

Tady máme kombinaci rekurze a iterátoru, takže to může vypadat velice smart, ale není to úplně užitečné, protože kvůli hloubce rekurze a nadbytečné spotřebě paměti přestane toto síto brzy fungovat.

---

## Třídy

Třídy nám umožňují seskupit data a funkce, které na nich operují a zpřístupňují je, a zároveň "schovat" detaily implementace. Třída je datový typ, od kterého si vytváříme instance, přesně tak, jak to děláme u Pythonovských tříd, se kterými jsme se už setkali: `list`, `str`, `tuple` atd.

```
class Zvire():
    pass

>>> pes = Zvire()
>>> pes
<__main__.Zvire object at 0x000001A01A376460>
>>> kocka = Zvire()
>>> kocka
<__main__.Zvire object at 0x000001A01A391B80>
```

Vidíme, že máme dva různé objekty. Takovýto objekt by ale nebyl moc užitečný, pokud neumíme definovat nějaké vlastnosti objektu.

```
# Třídy

class Zvire():

    def __init__(self, jmeno, zvuk):
        self.jmeno = jmeno
        self.zvuk = zvuk

    def slysi_na(self, jmeno):
        return self.jmeno == jmeno

    def ozvi_se(self):
        print(f"{self.jmeno} říká: {self.zvuk}")

...
>>> pes = Zvire("Punťa", "Hafff!")
```

```
>>> pes
<__main__.Zvire object at 0x000001A01A391B80>
>>> pes.slysi_na("Miau")
False
>>> pes.ozvi_se()
Punťa říká: Hafff!
>>> kocka = Zvire("Mourek", "Miau!")
>>> kocka.ozvi_se()
Mourek říká: Miau!
```

`self` nás odkazuje na instanci třídy.

`__init__()` je metoda, která vytváří instanci ze vstupních dat - *konstruktor*.

Metod s dvojitými podtržítky existuje mnoho. Jsou to metody, které definují standardní aspekty objektů.

**Vlastnosti a metody**

```
>>> azor = Zvire("Azor", "Haf!")
>>> azor
<__main__.Zvire object at 0x00000214E4303D00>
>>> azor.jmeno
'Azor'
>>> azor.zvuk
'Haf!'
>>> azor.zvuk = "Haffff!"
>>> azor.slysi_na("azor")
False
>>> azor.ozvi_se()
Azor říká: Haffff!
```

**Identita objektu**

```
>>> jezevcik = Zvire("˘Spagetka", "haf")
>>> bernardyn = Zvire("Bernard", "HAF!!!")
>>> maxipes = bernardyn
>>> maxipes.jmeno = "Fík"
>>> bernardyn.jmeno
'Fík'
>>> type(jezevcik)
<class 'Zvire'>
>>> id(jezevcik), id(bernardyn), id(maxipes)
(737339253592, 737339253704, 737339253704)
>>> bernardyn is maxipes
True
>>> bernardyn is jezevcik
False
```

**Znaková reprezentace objektu**

`__str__()` je to, co používá funkce `print`

`__repr__()` je to, co vypíše Pythonská konzole jako identifikaci objektu.

```
class Zvire():

    def __init__(self, jmeno, zvuk):
        self.jmeno = jmeno
        self.zvuk = zvuk

    ...

    def __str__(self):
        return self.jmeno

    def __repr__(self):
        return f"Zvire({self.jmeno}, {self.zvuk})"

    ...
>>> pes = Zvire("Punta", "haf!")
>>> pes
Zvire(Punta, haf!)
>>> print(pes)
Punta
```

**Protokoly pro operátory**

```
class Zvire():

    def __init__(self, jmeno, zvuk):
        self.jmeno = jmeno
        self.zvuk = zvuk

...

    def __eq__(self, other):
        return self.jmeno == other.jmeno and \
            self.zvuk == other.zvuk

...
>>> pes = Zvire("Punta", "haf!")
>>> kocka = Zvire("Mourek", "Miau!")
>>> pes == kocka
False
```

Podobně lze předefinovat řadu dalších operátorů:

- Konverze na bool, str, int, float
- Indexování `objekt[i], len(i)`, čtení, zápis, mazání.
- Přístup k atributům `objekt.klíč`
- Volání jako funkce `objekt(x)`
- Iterátor pro `for x in objekt:`

**Dokumentační řetězec**

```
class Zvire():
    """Vytvoří zvíře s danými vlastnostmi"""

    def __init__(self, jmeno, zvuk):
        self.jmeno = jmeno
        self.zvuk = zvuk

...

>>> help(Zvire)
>>> lenochod = Zvire("lenochod", "Zzzz...")
>>> help(lenochod.slysi_na)
```

## Dědičnost

```
class Kocka(Zvire):

    def __init__(self, jmeno, zvuk):
        Zvire.__init__(self, jmeno, zvuk)
        self._pocet_zivotu = 9 # interní

    def slysi_na(self, jmeno):
        # Copak kočka slyší na jméno?
        return False
...

>>> k = Kocka("Příšerka", "Mňauuu")
>>> k.slysi_na("Příšerka") (speciální kočičí verze)
False
>>> k.ozvi_se() (původní zvířecí metoda)
Příšerka říká: Mňauuu
```

### Typy

```
>>> type(k) is Kocka
True
>>> type(k) is Zvire
False
>>> isinstance(k, Kocka)
True
>>> isinstance(k, Zvire)
True
>>> issubclass(Kocka, Zvire)
True
```

## Prostory a rozsahy platnosti

Co dělá Python, když chce zjistit, kterou metodu třídy má volat?

Prostory jmen, **namespaces**:

- Zabudované funkce (print) `builtins`

- Globální jména - proměnné a funkce, definované mimo jakoukoli funkci nebo třídu `globals`
- Lokální jména definovaná při aktuálním volání uvnitř aktuální funkce `locals`
- Jména definovaná v aktuální třídě
- Jména definovaná v aktuálním objektu

Oblasti platnosti, **scopes**

Obyčejné jméno se hledá ve všech prostorech jmen, které jsou z daného kontextu vidět - lokální, globální, zabudované proměnné.

`objekt.jméno` se hledá

- mezi atributy objektu
- mezi atributy třídy
- mezi atributy nadřazených tříd

**local, global, nonlocal**

```
def scope_test():
    def do_local():
        spam = "local spam"

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"

    def do_global():
        global spam
        spam = "global spam"

    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)

scope_test()
print("In global scope:", spam)

---
=== RESTART: C:/Users/kvasnicka/AppData/Local/Programs/Python/Python312/sk.py ==
After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
In global scope: global spam
```

---

# Domácí úkoly

- **Pozice prvků ve sloučené posloupnosti** - už jsme slučovali setříděné posloupnosti, teď ale máte zjistit, na kterém indexu skončil ten-který prvek z původních posloupností.
- **Průměrná délka slov** - najít průměrnou délku slov ve vstupním textu.
- **Prvek s maximálním výskytem v posloupnosti** - Název vysvětluje vše. Jednoduchá úloha , u které doufám, že uvidím jednoduchá a efektivní řešení.