# `Programování` 1 pro matematiky

# 10. cvičení, 13-12-2023

---

**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**
   - Sešlo se poměrně málo řešení. Kvalita kódu roste, i když se najdou znepokojivé výjimky.
   - Dám ještě nějaké další domácí úkoly
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.

**Pozice v sloučené posloupnosti**

Podívejte se na tento kód:

```
N, M = map(int, input().split())
A = list(map(int, input().split()))
B = list(map(int, input().split()))

C = A + B
C.sort()
```

```
positions_A = [C.index(a) + 1 for a in A]
positions_B = [C.index(b) + 1 for b in B]

print(" ".join(map(str, positions_A)))
print(" ".join(map(str, positions_B)))
```

Je to velice hezký kód, jenomže je ťo celé špatně.

- `sort` způsobí, že ztrácíte kontrolu nad tím, kam se dostane která položka
- a následně musíte každou pracně vyhledávat.
- dovršení tragedie je, když to děláte krajně neefektivně.

**Obchodní cestující**

I když je toto prototyp výpočetně těžké úlohy, u této miniverze stačí nakódovat, co se žádá. Obvyklá chyba, která se vyskytla ve většině řešení byla, že jste prohledávali zbytečně velký prostor.

Když mám města A, B, C (to není reálné zadání, protože tady neexistuje nejkratší cesta), vytvořím možné trasy takto: A je společné, takže permutuji B, C: ABCA, ACBA. U většiny řešení jsem viděl ABCA, ACBA, BACB, BCAB, CABC, CBAC, tedy n-násobě víc drah, z nichže většina nejsou dráhy předepsané zadáním (i když je možno cyklicky posunout a vytvořit z nich jednu ze “správných” drah).

---

# Opakování: Funkce a generátory

## 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. Co se skrývá pod kapotou? Toto:

**Generátorem** nazýváme funkci, která může fungovat jako iterátor - lze ji opakovaně volat, a ona pokaždé vrátí následující hodnotu z nějaké posloupnosti.

Mnoho generátorů 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` |

| Iterator | Arguments | Results | Example |
|---|---|---|---|
| `groupby()` | iterable[, key] | sub-iterators grouped by value of key(v) | |
| `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**: itertools.count

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
```

**Příklad**: kombinace a permutace

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

---

# 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
```

# Soubory: čtení a zápis

**Souborem** myslíme nějakou skupinu bajtů, uloženou pod svým názvem v souborovém systému.

Budeme se zabývat **textovými soubory**, v nichž bajty reprezenutjí znaky v nějakém kódování.

- *ASCII* (”anglická abeceda“ o 95 znacích)
- *iso-8859-2* (navíc znaky východoevropských jazyků)
- *cp1250* (něco podobné, specifické pro Windows)
- *UTF-8* (vícebajtové znaky, pokrývají většinu glyfů a jazyků světa

Kódování je všudypřítomné, nevyhnete se problémům s explicitním uvedením kódování nebo s převodem. Neexistuje nic takového jako defaultní kódování textového souboru, i když například pro kód v Pythonu je defaultním kódováním **UTF-8**.

Python má rozsáhlou podporu kódování a většinu problémů jde jednoduše řešit.

Python načte textový soubor jako kolekci řádků. Naopak, při zápisu musíme konce řádků zapsat tam, kam patří:

```
f = open("soubor.txt", "w") # "w" jako write, "r" jako read
f.write("Hej, mistře!\n")
f.close()
```

Protože komunikujeme se systémovými službami a operačním systémem, může se při zápisu nebo čtení lehce stát něco neočekávaného - nejde vytvořit soubor, do kterého chcete zapisovat, soubor na čtení neexistuje tam, kde ho hledáte a podobně.  Pokud chceme, aby nám v takovýchto situacích neskončil program s chybou, ale nějak se se situací graciézně vypořádal, potřebujeme nástroje na *obsluhu výjimek* a o nich budeme mluvit za chvíli.

U čtení zápisu bychom rádi měli jistotu, že ať se stane cokoli, soubor se zavře. Proto standardně obsluhujeme soubory pomocí **kontexotvého manažéra** takto:

```
with open("soubor.txt", "w") as f:
    f.write("Hej, mistře!\n")
```

`f.close()` se zavolá automaticky po opuštění bloku `with` a to i v případě, že se stane něco neočekávaného.

## Metody souborů

`f.write(text)` – zapíše text
`f.read(n)` –přečte dalších n znaků, na konci " ".
`f.read()` – přečte zbývající znaky souboru
`f.readline()` – přečte další řádek (včetně "\n") nebo " ".
`f.seek(...)` – přesune se na další pozici v souboru

Další operace:
`print(..., file=f)`
`for line in f:` – cyklus přes řádky souboru
Pozor, řádky končí "\n", hodí se zavolat `rstrip()`.

Vždy je k dispozici:

`sys.stdin` – standardní vstup (odtud čte `input()`)
`sys.stdout` – standardní vstup (sem píše `print()`)
`sys.stderr` – standardní chybový výstup

```
>>> sys.stdout.write("Hej, mistre!\n")
Hej, mistre!
13
```

---

## Chyby a výjimky

```
def divide(x, y):
    return x/y

divide(1, 0)

Traceback (most recent call last):
  File "<pyshell#73>", line 1, in <module>
    divide(1,0)
  File "<pyshell#72>", line 2, in divide
    return x/y
ZeroDivisionError: division by zero
```

**Chyba vygeneruje výjimku**, např.
`ZeroDivisionError` – dělení nulou
`ValueError` – chybný arguement
`IndexError` – přístup k indexu mimo rozsah
`KeyError` – dotaz na hodnotu  neexistujícího klíče ve slovníku
`FileNotFoundError` – pokus o otevření neexistujícího souboru ke čtení
`MemoryError` – vyčerpání dostupné paměti
`KeyboardInterrupt` – běh programu byl přerušen stiskem `Ctrl-C`
`StopIteration` - žádost o novou hodnotu z vyčerpaného iterátoru

```
try:
    x, y = map(int, input().split())
    print(x/y)
except ZeroDivisionError:
    print("Nulou dělit neumím.")
except ValueError as ve:
    print("Chyba:", ve)
    print("Zadejte prosím dvě čísla.")
```

Výjimky jsou objekty, jejich typy jsou třídy.

Výjimka se umí vypsat příkazem `print`

Atributy výjimky obsahují dodatečné informace o tom, co a kde se stalo.

Výjimky tvoří hierarchie, například `FileNotFoundError` je potomkem `IOError`. Můžeme zachytit obecnější typ a doptat se, o kterého potomka se jedná.

```
>>> raise RuntimeError("Jejda!")
Traceback (most recent call last):
  File "<pyshell#75>", line 1, in <module>
    raise RuntimeError("Jejda!")
RuntimeError: Jejda!

>>> assert 1 == 2
Traceback (most recent call last):
  File "<pyshell#78>", line 1, in <module>
    assert 1 == 2
AssertionError

>>> assert 1 == 2, "Pravda už není, co bývala!"
Traceback (most recent call last):
  File "<pyshell#79>", line 1, in <module>
    assert 1 == 2, "Pravda už není, co bývala!"
AssertionError: Pravda už není, co bývala!
```