`Programování` 1 pro matematiky

11. cvičení, 9-12-2025

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**
   - Obě úlohy jste zvládli dobře, i když se vyskytli některé problémy. Víc ve cvičení.
   - Příští cvičení je poslední před svátky. Domácí úkoly ale dostanete, a dostanete je také v prvním lednovém týdnu.

Kde se nacházíme

Ještě máme spoustu povídání ke třídám, ale dnešní porci zkrátím tak, abychom se dostali k prvnímu přeletu čtení a zápisu do souborů a obsluze výjimek.

Tady bývá citát na úvod cvičení. Po kontrole domácích úkolů musím jeden zopakovat:

> Clear beats clever. Everytime. (Tom, the single dad janitor)

Iterátory a generátory jsme neprobírali proto, abyste teď všichni psali nepochopitelný a těžkopádný kód, ale abyste uměli iterovat přes složité objekty.

Kvíz

```
>>> x = {"a": 1, "b": 2}
>>> y = {"b": 3, "c": 4}
>>> z = {**x, **y}
>>> z
{'a': 1, 'b': 3, 'c': 4}
```

Dvě hvězdičky dělají se slovníkem to, co jedna dělá se seznamem: rozdělí slovník na dvojice `klíč : hodnota`, oddělené čárkou.

Jiné způsoby spojování slovníků:

```
>>> x.update(y)
>>> x
{'a': 1, 'b': 3, 'c': 4}
>>> x = x = {"a": 1, "b": 2}    # Návrat k původnímu
>>> x | y
{'a': 1, 'b': 3, 'c': 4}
```

Všimněme si, že ve všech případech se první hodnota `"b"` přepíše druhou.

Někdy potřebujeme řetězit slovníky tak, aby vraceli první nalezenou hodnotu v řetězci. Příklad jsou defaultní a uživatelsky přizpůsobená nastavení:

```
In [47]: from collections import ChainMap

In [48]: default = {"theme" : "light", "font" : "Lucida Sans", "font_size" : 10}

In [49]: user = {"theme" : "dark", "font_size" : 12}

In [50]: settings = ChainMap(user, default)

In [51]: settings
Out[51]: ChainMap({'theme': 'dark', 'font_size': 12}, {'theme': 'light', 'font': 'Lucida
Sans', 'font_size': 10})

In [52]: settings["theme"]
Out[52]: 'dark'

In [53]: settings["font"]
Out[53]: 'Lucida Sans'

In [54]: settings["font_size"]
Out[54]: 12
```

Jinak řečeno, tam, kde si uživatel definoval preferované nastavení, toto nastavení se použije, v opačném případě se použije defaultní nastavení.

Domácí úkoly

Ještě stále chodí hodně řešení, co je celkem potěšující.

Prvek s maximálním výskytem v posloupnosti

Standardní postup pro počítání výskytů věcí v posloupnosti je použít *slovník*. Prototypová úloha byla úloha o nalezení modusu posloupnosti, tedy prvku s největším výskytem.

- Přidávání prvků:
  - odpadá u *defaultdict*
  - u obyčejného slovníku se můžete buď zeptat, zda prvek již ve slovníku existuje, a pokud ne, inicializovat ho:

```
if elem in dic:
    dic[elem] += 1
else:
    dic[elem] = 1
```

Elegantní možnost je použít na pravé straně `dict.get`:

```
dic[elem] = dic.get(elem, 0) + 1
```

- Obyčejný slovník nebo *defaultdict*: modus naleznete jako `max(dic, key = dic.get)`.
- *collections.Counter* má speciální metodu pro vylistování největších prvků.

```
>>> from collections import Counter
>>> Counter('abracadabra').most_common(3)
[('a', 5), ('b', 2), ('r', 2)]
```

Účelem kurzu programování není pouze vás naučit, jak napsat cyklus `for`. Měli byste také znát některé základní postupy, například binární vyhledávání nebo počítání věcí,

Pozice prvků ve sloučené posloupnosti

Toto je lehká úloha jakmile si vzpomenete na slučování setříděných posloupností, což byl jeden z předchozích domácích úkolů. Algoritmus u této úlohy je stejný - pomocí tří indexů procházíme zdrojová pole a výsledné pole. Rozdíl je v tom, že namísto budování výsledného pole si tady pamatujeme index té-které položky ve výsledném poli.

```
def solve(a: list[int], b: list[int]) -> tuple[list[int], list[int]]:
    a_indices = [0]*len(a)
    b_indices = [0]*len(b)
    i = 0
    j = 0
    k = 0
    while i < len(a) and j < len(b):
        if a[i] < b[j]:
            a_indices[i]= k + 1
            i += 1
        else:
            b_indices[j] = k + 1
            j += 1
        k += 1
    while i < len(a):
        a_indices[i] = k + 1
        i += 1
        k += 1
    while j < len(b):
        b_indices[j] = k + 1
        j += 1
        k += 1
    return a_indices, b_indices
```

Co tady nesmíte udělat:

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

C = A + B

C_sorted = sorted(C)
...
```

Tady jste se ztratili, protože jste všechno smíchali dohromady a můžete jedině zkusit zpětně vyhledat, která položka kam přišla. To rozhodně není dobrá cesta k řešení, a to ani když sloučíte posloupnosti známým algoritmem namísto generického `list.sort`.

Jeden způsob záchrany v tomto případě je použít slovník k zapamatování indexů a pak hledat ve slovníku:

```
...
pozice = {}
for i, elem in enumerate(C_sorted):
    pozice[elem] = i

index_a = [0] * len(A)
index_b = [0] * len(A)

for i, a in enumerate(A):
    index_a[i] = pozice[a] + 1
for i, b in enumerate(B):
    index_b[i] = pozice[b] + 1
```

Toto docela napravuje katastrofu způsobenou smícháním posloupností. Takovýto kód je ale křehký, protože zcela závisí od předpokladu, že prvky obou vstupních posloupností jsou unikátní. Co když to ale neplatí?

V takovém případě musí být prvky slovníku ne pozice, ale fronty pozic. První nalezená pozice se také vyvolá jako první při zapisování indexů, pak ji ale musíme odmazat. Protože přidáváme prvky na konci a ubíráme na začátku, ideální je použít namísto obyčejného seznamu `collections.deque`, tedy "dvouhlavou frontu".

```
from collections import deque, defaultdict
...
pozice = defaultdict(deque)
for i, elem in enumerate(C_sorted):
    pozice[elem] = append(i)    # zapisujeme na konec

index_a = [0] * len(A)
index_b = [0] * len(A)

for i, a in enumerate(A):
    index_a[i] = pozice[a].pop_left() + 1   # vyvoláme+odstraníme první
for i, b in enumerate(B):
    index_b[i] = pozice[b].pop_left() + 1
```

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.

```
# 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 chování 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!
>>> azor.barva = "hnědá" # Každému objektu můžeme přidat vlastnosti
>>> azor.__dict__ # lehko zjistíme atributy a metody každého objektu
```

```
{'jmeno': 'Azor', 'zvuk': 'Haf!', 'barva': 'hnědá'}
```

Magické neboli "dunder" metody

Metody s názvy `__metoda__()` jsou metody zděděné od pythonského praobjektu `Object` a definují základní chování každé třídy.

- `__new__()` a `__del__()` - vytvoření a smazání instance (`__new__()` potřebujeme pouze ve speciálních případech, kdy k vytvoření instance nestačí defaultní `__new__` , automaticky volané přes `Object.__init__()`)
- `__str__()` je to, co používá funkce `print`
- `__repr__()` je to, co vypíše Pythonská konzole jako identifikaci objektu.
- Konverze na bool, str, int, float: `__bool__()`, atd., operace `__add__()`, `__subtract__` atd.
- Indexování `objekt[i]`: `__getitem__()`, `__len__()` atd.
- Volání jako funkce `objekt(x)`: `__call__()`
- Iterátor pro `for x in objekt:` `__iter__()`

```
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})"

    def __eq__(self, other):
        return self.jmeno == other.jmeno and \
            self.zvuk == other.zvuk
    ...
>>> pes = Zvire("Punta", "haf!")
>>> pes
Zvire(Punta, haf!)
>>> print(pes)
Punta
>>> kocka = Zvire("Mourek", "Miau!")
>>> pes == kocka
False
```

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)
```

Dataclasses

```
from dataclasses import dataclass

@dataclass
class Osoba:
    jmeno: str
    prijmeni: str
    rok_narozeni: int

petr = Osoba('Petr', 'Novák', 1980)
marie = Osoba('Marie', 'Zahradníková', 1988)

print(petr)
print(marie)
```

`@dataclass` je dekorátor. Je to jenom syntaxe pro napsání, že naši definovanou třídu pošleme jako parametr funkci `dataclass`, která z něj vytvoří jiný objekt, který budeme používat pod jménem `Osoba`.

Dekorátor je funkce, a kromě vstupního objektu může obsahovat itaké další parametry:

```
@dataclass
class C:
    ...

@dataclass()
class C:
    ...

@dataclass(init=True, repr=True, eq=True, order=False, unsafe_hash=False, frozen=False,
match_args=True, kw_only=False, slots=False, weakref_slot=False)
class C:
    ...
```

Účel parametrů je víceméně zřejmý z názvů, zbytek si prosím najděte v dokumentaci. Tam najdete hromadu dalších věcí.

```
from dataclasses import dataclass

@dataclass(order=True)
class Osoba:
    jmeno: str
    prijmeni: str
    rok_narozeni: int

petr = Osoba('Petr', 'Novák', 1980)
marie = Osoba('Marie', 'Zahradníková', 1988)

print(petr)
print(marie)
print(petr<marie)
```

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 ko˘cka sly˘s´ı na jm´eno?
        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

Příklad: atributy třídy a atributy instance: počítadlo objektů

```
class Counted_class:
    _instance_count = 0     # podtržítko znamená interní proměnnou
                            # (není vynucováno)
    def __init__(self):
        type(self)._instance_count += 1     # type, protože jde o
                                            # proměnnou třídy, ne
                                            # instance

    @classmethod            # dekorátor označující metodu třídy
                            # argument ne self, ale cls (silná norma)
    def get_instance_count(cls):
        return cls._instance_count

# Example usage:
obj1 = Counted_class()
obj2 = Counted_class()
print(Counted_class.get_instance_count())  # Output: 2
```

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
```

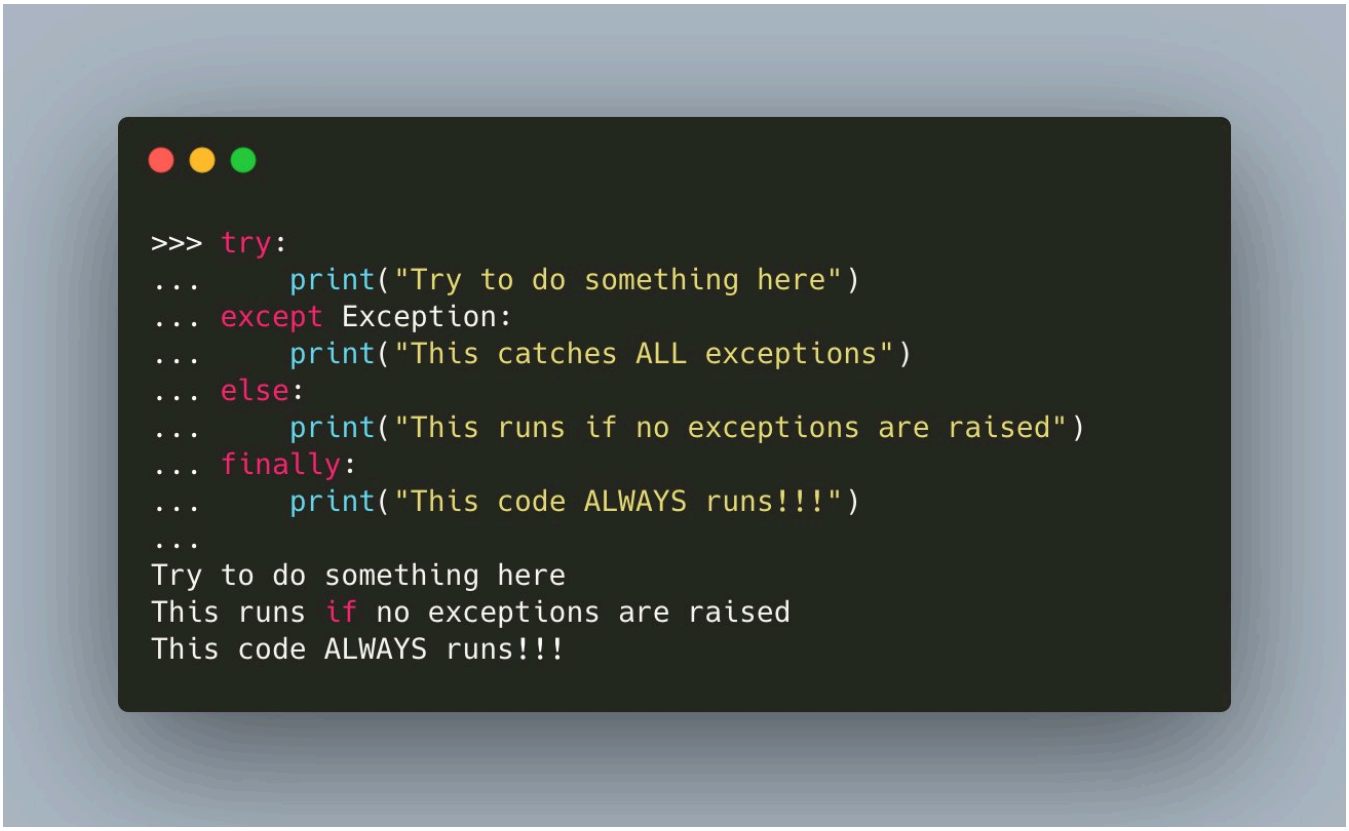
Chyby a výjimky

```
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.")
```

Obecně je syntaxe takováto:



```
>>> try:
...     print("Try to do something here")
... except Exception:
...     print("This catches ALL exceptions")
... else:
...     print("This runs if no exceptions are raised")
... finally:
...     print("This code ALWAYS runs!!!")
...
Try to do something here
This runs if no exceptions are raised
This code ALWAYS runs!!!
```

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!
```

Domácí úkoly

Matrix class - máte vytvořit maticovou třídu a implementovat jednoduché maticové operace. Protože to je víc kódování a testování, dal jsem na tuto úlohu termín dva týdny.

Inverzní permutace - k permutaci čísel 1..N na vstupu máte najít takovou permutaci, která vrátí permutovaná čísla na místo, tedy vytvoří původní posloupnosti 1..N.