Programování 2

7. cvičení, 30-3-2023

tags: `Programovani 2`, `čtvrtek 1`, `čtvrtek 2`

Farní oznamy

1. Tento text a kódy ke cvičení najdete v repozitáří cvičení na https://github.com/PKvasnick/Programovani-2.
2. **Domácí úkoly**: LSS, poměrně lehké, popovídáme si o nich níže.
3. **Zápočtový program** a **zápočtový test**: dostali jsme se do dalšího měsíce výuky a je čas promluvit si o tom, co vás čeká.
    - **Zápočtový program**: Měl by to být větší ucelený kus kódu, řádově stovky řádků na rozdil od desítek pro domácí úkoly. Víc níže.
    - **Zápočtový test** Jeden příklad kategorie domácího úkolu vyřešit v reálném čase u počítače v učebně.

Dnešní program:

- Zápočtový program
- Kvíz
- Mini-tutoriál: výjimky
- Opakování: LSS, domácí úkoly, cyklický zásobník atd.
- Binární strom

Zápočtový program

Zápočtový program je závěrečná výstupní práce každého studenta, vyvrcholení roční výuky programování.

Zatímco průběžné domácí úkoly mají typicky rozsah několika málo desítek řádků kódu a zadaný úkol je pro všechny studenty stejný, zápočtové programy mají obvykle rozsah několika set řádků kódu a studenti zpracovávají různá témata.

- Zadání v polovině letního semestru
- Dokončení: šikovní ke konci semestru, typicky přes prázdniny
- Odevzdání první verze: konec srpna, finální verze: konec září
- Textová dokumentace
    - Zadání
    - Uživatelská část - návod na použití
    - Technická - popis z programátorského hlediska

- **Téma**: Jakékoliv.
    - poslat specifikaci - musíme se dohodnout na rozsahu, aby zadání nebylo příliš sliožité ani příliš jednoduché
        - Sudoku, Piškvorky
        - Výpočet derivací
        - Fyzikální a statistické simulace - difúze částic v složitém prostředí, perkolace, pohyb osob v budově s výtahem, pohyb zákazníků v nákupním středisku, zákazníci obědvající v restauraci, pohyb lidí na Matějské pouti, epidemiologické modely apod.
    - nějaká témata máme, podívejte se třeba na web Martina Mareše: http://mj.ucw.cz/vyuka/zap/
    - tesmín pro zadání závěrečného programu: **do konce dubna**, pak dostanete témata přidělena.

Na zahřátí

> *"Experience is the name everyone gives to their mistakes." – Oscar Wilde*

Vlastní naražený nos poučí lépe než rady učených mistrů. Tak jako kuchař musí zkazit kopu receptů, než se vyučí, i programátor musí udělat kopu chyb. Naučí vás to některé věci automaticky nedělat.

Co dělá tento kód

```
1 [x for x in dir("") if "_" not in x]
```

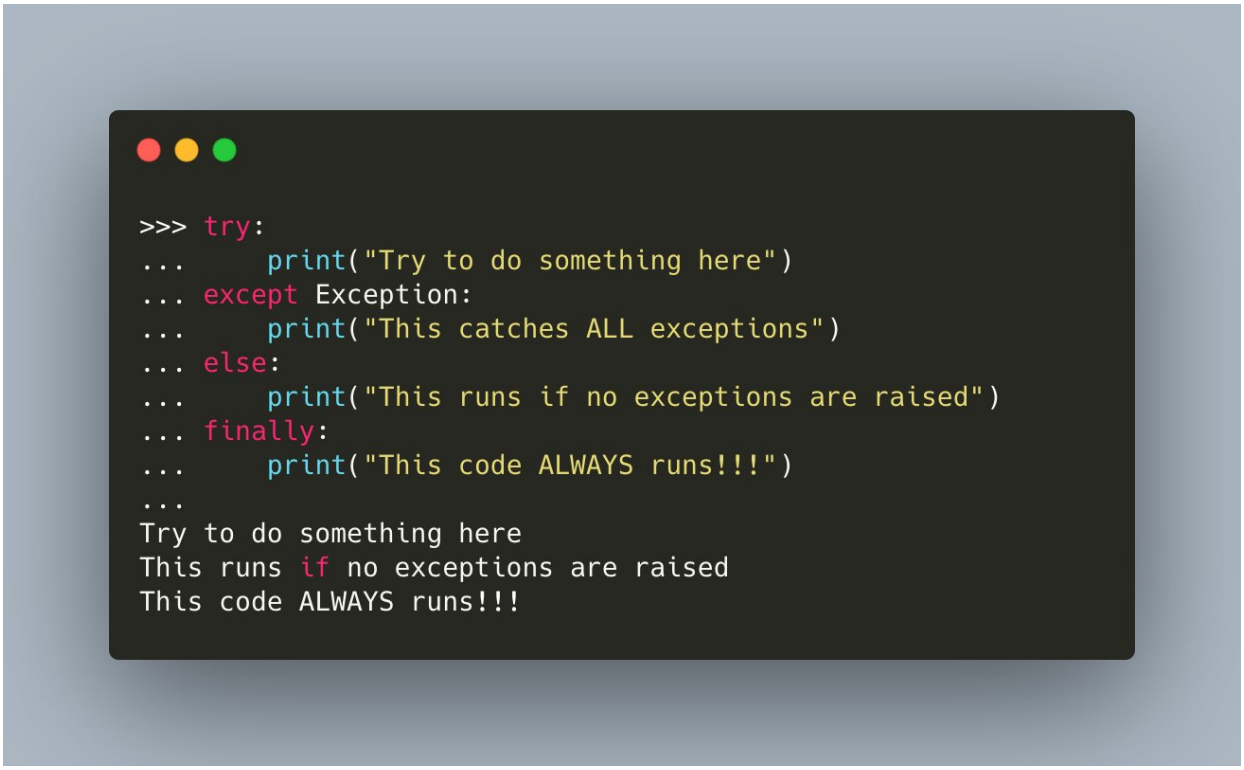
Návod:

`dir(objekt)` vypíše atributy objektu.

Mini tutoriál: Výjimky

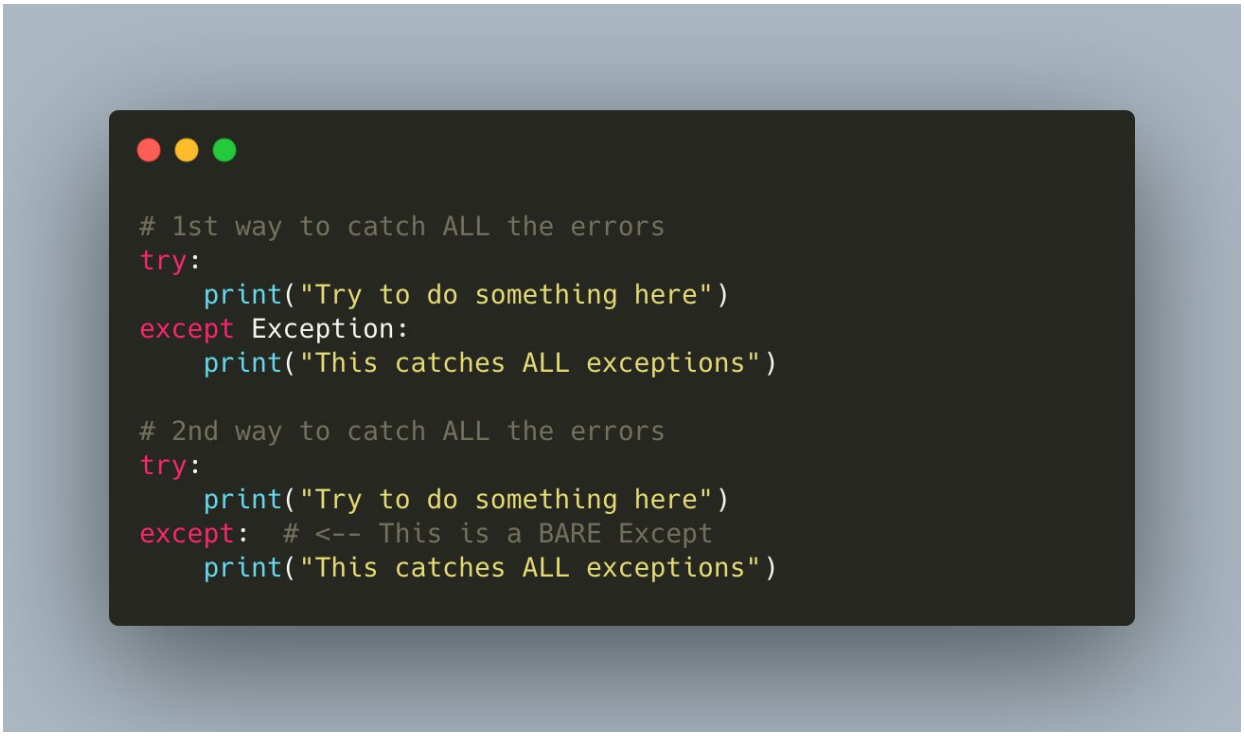
Výjimky jsme měli a i na posledních cvičeních jsme si o nich povídali. Tady několik věcí, které je dobře vědět:

Obsluha výjimek v Pythonu využívá strukturu `try + except`, případně s dodatečnými větvemi `else` a `finally`:



```
>>> 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!!!
```

`Exception` je základní typ výjimky, specifické výjimky jsou jeho podtřídami. Pokud zachytáváme `Exeption`, znamená to, že zachytávame všechny výjimky. V takovém případě nemusíme `Exception` v klauzule `except` vůbec uvádět:



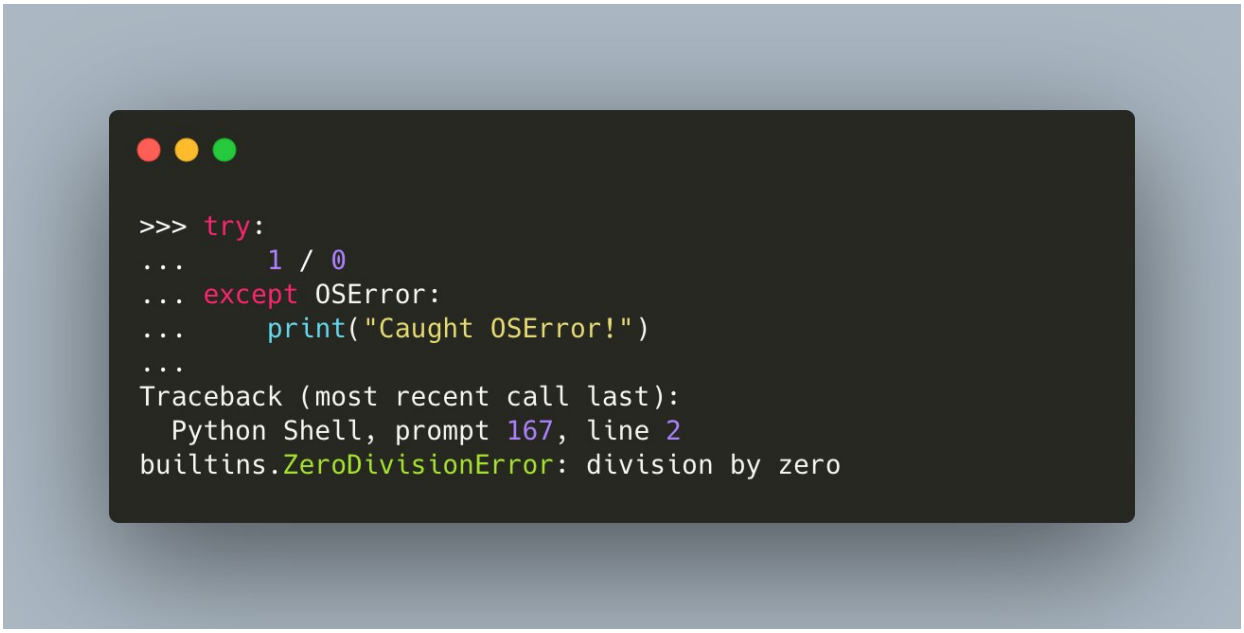
```
# 1st way to catch ALL the errors
try:
    print("Try to do something here")
except Exception:
    print("This catches ALL exceptions")

# 2nd way to catch ALL the errors
try:
    print("Try to do something here")
except:  # <-- This is a BARE Except
    print("This catches ALL exceptions")
```

Úplně nejlepší je ale toto vůbec NIKDY nepoužívat. Zachytávejte ty chybové stavy, které umíte ošetřit. Některé výjimky prostě musíte nechat "přepadnout" do části kódu, která si s ní bude umět poradit.

Co udělat se zachycenou výjimkou? Co potřebujete:

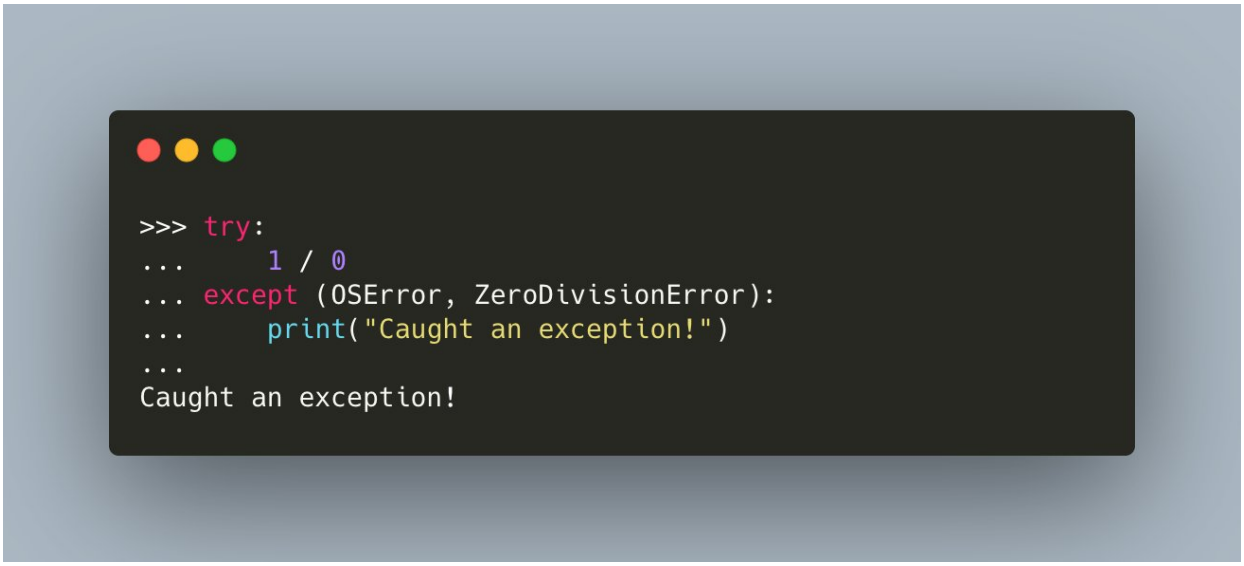
```
>>> try:
...     1 / 0
... except ZeroDivisionError:
...     print("Caught ZeroDivisionError!")
...
Caught ZeroDivisionError!
```

Musíte samozřejmě zachytit správnou výjimku.



```
>>> try:
...     1 / 0
... except OSError:
...     print("Caught OSError!")
...
Traceback (most recent call last):
  Python Shell, prompt 167, line 2
builtins.ZeroDivisionError: division by zero
```

Můžete také zachytit víc výjimek:



```
>>> try:
...     1 / 0
... except (OSError, ZeroDivisionError):
...     print("Caught an exception!")
...
Caught an exception!
```

tady ale vzniká problém: Jak poznat, kterou výjimku jsme zachytili?

Jedna z možností je:

```
>>> try:
...     1 / 0
... except (OSError, ZeroDivisionError) as exception:
...     print(f"{exception=}!")
...
exception=ZeroDivisionError('division by zero')!
```

Praktičtější řešení je použít více klauzulí `except`, každou pro jeden typ výjimky.

`Exception` je třída, má své atributy a můžeme se na ně doptat.

```
>>> try:
...     raise IOError("Broken", "Pipe")
... except IOError as exc:
...     print(type(exc))
...     print(f"{exc.args=}")
...     print(f"{exc=}")
...
<class 'OSError'>
exc.args=('Broken', 'Pipe')
exc=OSError('Broken', 'Pipe')
```

Můžeme si také vytvořit vlastní výjimku:

```
>>> class CustomException(Exception):
...     pass
...
>>> raise CustomException("This is a custom error!")
Traceback (most recent call last):
  Python Shell, prompt 182, line 1
__main__.CustomException: This is a custom error!
```

Klauzule `finally` vám umožňuje provést úklid po operaci nezávisle od toho, zda se operace povedla nebo ne.

Jiný způsob, jak uklidit po operaci se souborem, jsme si ukauzovali v minulém semestru - je to použití kontextového manažera:

```
with open("something.txt", "w") as f:
    passs
```

Toto zaručeně po sobě uklidí, a to i v případě, že se něco pokazí - například pokud se nenajde soubor.

Modul `contextlib` také umožňuje zpracovat výjimky pomocí kontextového manažera namísto `try-except-finally`:

```
1  from contextlib import suppress
2
3  lst = list(range(10))
4
5  i = -1
6  with suppress(IndexError):
7      i = lst.index(12)
8
9  print(i)
```

Pokud chceme, aby program v případě chyby skončil, můžeme v klauzuli `except` použít `sys.exit()` anebo můžete výjimku znova vyvolat:

```
>>> try:
...     raise IOError("Broken", "Pipe")
... except IOError as exc:
...     print("An IOError occurred. Re-raising")
...     raise
...
An IOError occurred. Re-raisingTraceback (most recent call
last):
  Python Shell, prompt 176, line 2
builtins.OSError: [Errno Broken] Pipe
```

`try-except` namísto `if-else`

Pokud potřebujeme zachytit zřídka se vyskytující stav, můžeme namísto `if-else` použít `try-except`. Podmíněný příkaz přidává prodlení ke zpracování obou větví, zatímco `try-except` přidává prodlení prakticky jenom ke větvi `except`.

Není dobré takovýto způsob nadužívat, ale je to Pythonský způsob vyjadřování a neváhejte ho ve vhodné situaci použít.

LSS - ještě naposledy

Spojovaný seznam s hlavou (kód v repozitáří, `code/Ex6/simply_linked_list1.py`)

```python
# Simple linked list

class Node:
    def __init__(self, value):
        """Polozku inicializujeme hodnotou value"""
        self.value = value
        self.next = None

    def __repr__(self):
        """Reprezentace objektu na Pythonovske konzoli"""
        return str(self.value)

```

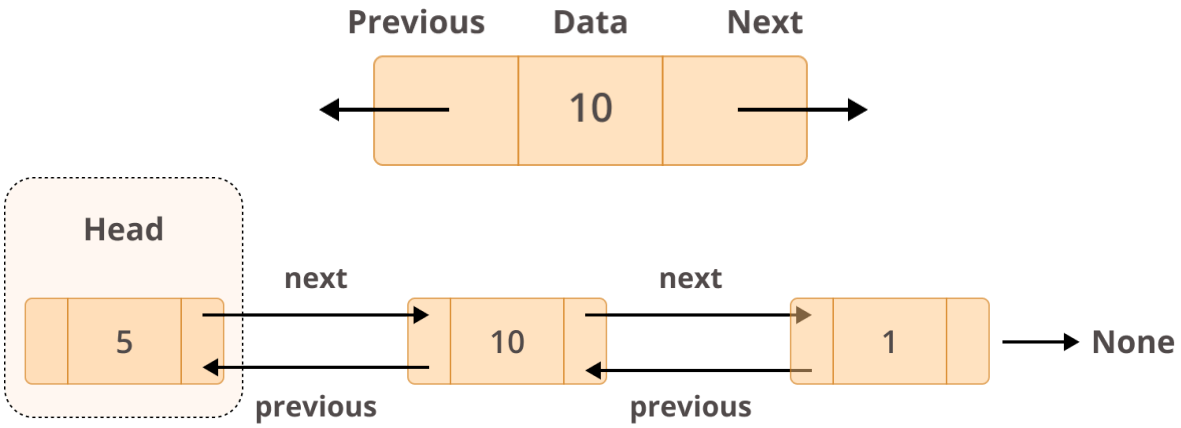
```
13
14  class LinkedList:
15      def __init__(self, values = None):
16          """Spojovany seznam volitelne inicializujeme seznamem hodnot"""
17          if values is None:
18              self.head = None
19              return
20          self.head = Node(values.pop(0)) # pop vrati a odstrani hodnotu z values
21          node = self.head
22          for value in values:
23              node.next = Node(value)
24              node = node.next
25
26      def __repr__(self):
27          """Reprezentace na Pythonovske konzoli:
28          Hodnoty spojene sipkami a na konci None"""
29          values = []
30          node = self.head
31          while node is not None:
32              values.append(str(node.value))
33              node = node.next
34          values.append("None")
35          return " -> ".join(values)
36
37      def __iter__(self):
38          """Iterator prochazejici _hodnotami_ seznamu,
39          napr. pro pouziti v cyklu for"""
40          node = self.head
41          while node is not None:
42              yield node.value
43              node = node.next
44
45      def add_first(self, node):
46          """Prida polozku na zacatek seznamu,
47          tedy na head."""
48          node.next = self.head
49          self.head = node
50
51      def add_last(self, node):
52          """Prida polozku na konec seznamu."""
53          p = self.head
54          prev = None
55          while p is not None:
56              prev, p = p, p.next
57          prev.next = node
58
59
```

Vkládání a odstraňování prvků

- `add_first`, `add_last`
- `add_before`, `add_after`
- `remove`

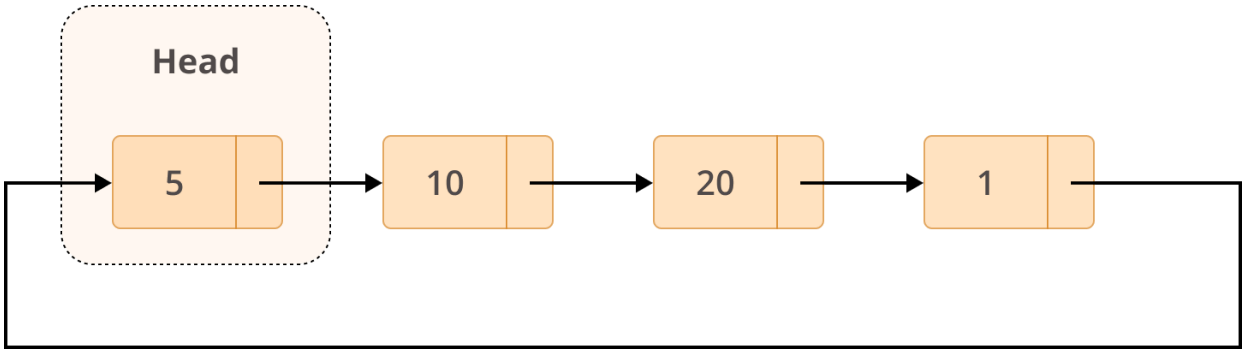
Varianty LSS

- **Dvojitě spojovaný seznam** - pro `deque`




```python
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.previous = None
```

- **Cyklický seznam**



Cyklickým seznamem můžeme procházet počínaje libovolným prvkem:

```python
# Kruhový seznam - pointer u poslední položky ukazuje na začátek seznamu.
from _collections_abc import Generator


class Node:
    def __init__(self, value):
        """Polozku inicializujeme hodnotou value"""
        self.value = value
```

```
        self.next = None

    def __repr__(self):
        """Reprezentace objektu na Pythonovske konzoli"""
        return str(self.value)


class CircularLinkedList:
    def __init__(self, values = None):
        self.head = None
        if values is not None:
            self.head = Node(values.pop(0))
            node = self.head
            for val in values:
                node.next = Node(val)
                node = node.next
            node.next = self.head

    def traverse(self, starting_point: Node = None) -> Generator[Node, None,
None]:
        if starting_point is None:
            starting_point = self.head
        node = starting_point
        while node is not None and (node.next != starting_point):
            yield node
            node = node.next
        yield node

    def print_list(self, starting_point: Node = None) -> None:
        nodes = []
        for node in self.traverse(starting_point):
            nodes.append(str(node))
        print(" -> ".join(nodes))

```

Jak to funguje:

```
>>> clist = CircularLinkedList([1,2,3,4,5])
>>> clist.print_list()
1 -> 2 -> 3 -> 4 -> 5
for node in clist.traverse():
    pass
node
5
node.next
1

```

Domácí úkoly

Kromě Josefa, co byla úloha na kruhový spojovaný seznam, jsme měli dva velmi příbuzné úkoly - implementovat metodu pro průnik a sjednocení hodnot dvou setříděných LSS.

V obou případech je na místě použít algoritmus velmi podobný spojování dvou setříděných seznamů, s malými modifikacemi podle konkrétního zadání.

```
def IntersectionDestruct(a,b):
    """ destruktivni prunik dvou usporadanych seznamu
    * nevytvari zadne nove prvky, vysledny seznam bude poskladany z prvku puvodnich seznamu,
    * vysledek je MNOZINA, takze se hodnoty neopakuji """
    intersection = None
    itail = None

    def add_to_intersection(p:Prvek) -> None:
        pass # implementace nás zatím nezajímá

    while a and b:
        if a.x < b.x:
            a = a.dalsi
        elif b.x < a.x:
            b = b.dalsi
        else:  # a.x == b.x:
            add_to_intersection(a)
            a = a.dalsi
            b = b.dalsi

    if itail:
        itail.dalsi = None
    return intersection

```

a podobně pro sjednocení:

```
def UnionDestruct(a,b):
    """ destruktivni prunik dvou usporadanych seznamu
    * nevytvari zadne nove prvky, vysledny seznam bude poskladany z prvku puvodnich seznamu,
    * vysledek je MNOZINA, takze se hodnoty neopakuji """

    union = None
    p_union = None

    def add_to_result(p):
        pass # implementace nás zatím nezajímá

    while a and b:
        if a.x < b.x:
            add_to_result(a)
```

```
            a = a.dalsi
        elif: b.x < a.x:
            add_to_result(b)
            b = b.dalsi
        else: # a.x == b.x:
            add_to_result(a)
            a = a.dalsi
            b = b.dalsi
    while a:
        add_to_result(a)
        a = a.dalsi
    while b:
        add_to_result(b)
        b = b.dalsi

    return union

```

Zůstává nám jenom implementovat funkci `add_to_result`:

```
    union = None
    p_union = None

    def add_to_result(p):
        nonlocal union, p_union
        if not union:
            union = p
            p_union = p
        else:
            if p.x == p_union.x: # pokud prvek s hodnotou už máme, nechceme další
                return
            p_union.dalsi = p
            p_union = p_union.dalsi
        return

    ...
    return union
```

Mohli bychom také implementovat vnořenou třídu. To by sice bylo čistější (žádné `nonlocal`), ale víc zbytečného kódu.

Stromy: binární stromy

```
class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None
```