

Programování 2

7. cvičení, 30-3-2023

tags: Programování 2, čtvrtek 1, čtvrtek 2

Farní oznamy

1. Tento text a kódy ke cvičení najdete v repozitáři cvičení na <https://github.com/PKvasnick/Progrmovani-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 rozdíl 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.

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: šikovně 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 složité 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 `Exception`, znamená to, že zachytáváme 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 'IOError'>
exc.args=('Broken', 'Pipe')
exc=IOError('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 ukazovali v minulém semestru - je to použití kontextového manažera:

```

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

```

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  import os
2
3  try:
4      os.remove("muj_soubor.txt")
5  except FileNotFoundError:
6      pass
7

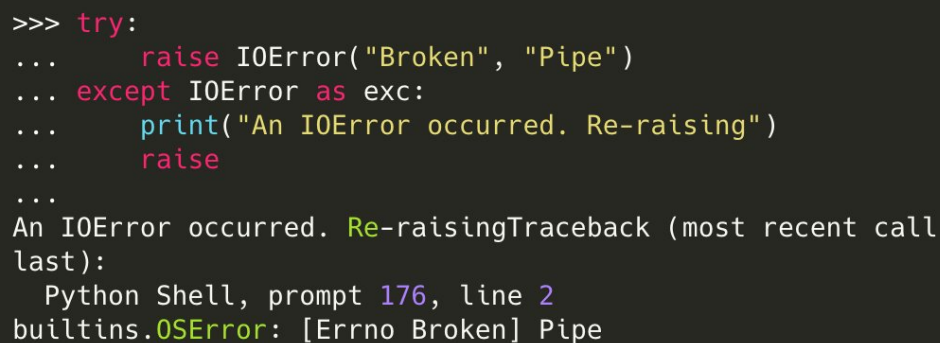
```

Tady jenom chceme, aby se kód nezastavil, když se nenajde soubor, který chceme odstranit.

```
1 from contextlib import suppress
2
3 with suppress(FileNotFoundError):
4     os.remove("muj_soubor.txt")
```

Pro případy, kdy chceme něco udělat, pokud se objeví výjimka, musíme použít `try-except`.

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-raising
Traceback (most recent call last):
  Python Shell, prompt 176, line 2
builtins.IOError: [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 prodlžení ke zpracování obou větví, zatímco `try-except` přidává prodlžení 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

Domácí úkoly

Kromě Josefa, což 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.

```
1 def IntersectionDestruct(a,b):
```

```

2     """ destruktivní pruník dvou usporadanych seznamu
3     * nevytváří žádné nové prvky, výsledný seznam bude poskládán z prvků
    původních seznamu,
4     * výsledek je MNOŽINA, takže se hodnoty neopakují """
5     intersection = None
6     itail = None
7
8     def add_to_intersection(p:Prvek) -> None:
9         pass # implementace nás zatím nezajímá
10
11    while a and b:
12        if a.x < b.x:
13            a = a.dalsi
14        elif b.x < a.x:
15            b = b.dalsi
16        else: # a.x == b.x:
17            add_to_intersection(a)
18            a = a.dalsi
19            b = b.dalsi
20
21    if itail:
22        itail.dalsi = None
23    return intersection
24

```

a podobně pro sjednocení:

```

1     def UnionDestruct(a,b):
2         """ destruktivní pruník dvou usporadanych seznamu
3         * nevytváří žádné nové prvky, výsledný seznam bude poskládán z prvků
    původních seznamu,
4         * výsledek je MNOŽINA, takže se hodnoty neopakují """
5
6         union = None
7         p_union = None
8
9         def add_to_result(p):
10             pass # implementace nás zatím nezajímá
11
12        while a and b:
13            if a.x < b.x:
14                add_to_result(a)
15                a = a.dalsi
16            elif b.x < a.x:
17                add_to_result(b)
18                b = b.dalsi
19            else: # a.x == b.x:
20                add_to_result(a)
21                a = a.dalsi
22                b = b.dalsi
23        while a:

```



```

24         add_to_result(a)
25         a = a.dalsi
26     while b:
27         add_to_result(b)
28         b = b.dalsi
29
30     return union
31

```

Zůstává nám jenom implementovat funkci `add_to_result`:

```

1     union = None
2     p_union = None
3
4     def add_to_result(p):
5         nonlocal union, p_union
6         if not union:
7             union = p
8             p_union = p
9         else:
10            if p.x == p_union.x: # pokud prvek s hodnotou už máme, nechceme
další
11                return
12            p_union.dalsi = p
13            p_union = p_union.dalsi
14        return
15
16    ...
17    return union

```

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

Rekurzivní řešení

Někteří z vás přišli s hezkým rekurzivním řešením:

```

1     def UnionDestruct(a,b):
2         """ destruktivni sjednoceni dvou usporadanych seznamu
3         * nevytvari zadne nove prvky, vysledny seznam bude poskladany z prvku
puvodnich seznamu,
4         * vysledek je MNOZINA, takže se hodnoty neopakují """
5
6         if a is None:
7             return b
8         if b is None:
9             return a
10
11        if a.x == b.x:
12            a.dalsi = UnionDestruct(a.dalsi, b.dalsi)
13        return a

```

```

14     if a.x < b.x:
15         a.dalsi = UnionDestruct(a.dalsi, b)
16         return a
17     if a.x > b.x:
18         b.dalsi = UnionDestruct(a, b.dalsi)
19         return b

```

Toto řešení je dost jemné, je důležité, že např. přiřazení `a.dalsi = ...` se děje až po zpracování zbytku seznamu, takže se struktura seznamu předčasně nezruší.

Podobně pro průnik:

```

1  def IntersectionDestruct(a,b):
2      """ destruktivni prunik dvou usporadanych seznamu
3      * nevytvari zadne nove prvky, vysledny seznam bude poskladan z prvku
4      * vysledek je MNOZINA, takže se hodnoty neopakují """
5
6      if not a or not b:
7          return None
8
9      if a.x == b.x:
10         a.dalsi = IntersectionDestruct(a.dalsi, b.dalsi)
11         return a
12     if a.x < b.x:
13         return IntersectionDestruct(a.dalsi, b)
14     if a.x > b.x:
15         return IntersectionDestruct(a, b.dalsi)
16

```

Implementace LSS

Spojovaný seznam s hlavou (kód v repozitáři, [code/Ex6/simply_linked_list1.py](#))

```

1  # Simple linked list
2
3  class Node:
4      def __init__(self, value):
5          """Polozku inicializujeme hodnotou value"""
6          self.value = value
7          self.next = None
8
9      def __repr__(self):
10         """Reprezentace objektu na Pythonovske konzoli"""
11         return str(self.value)
12
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 spojeny 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

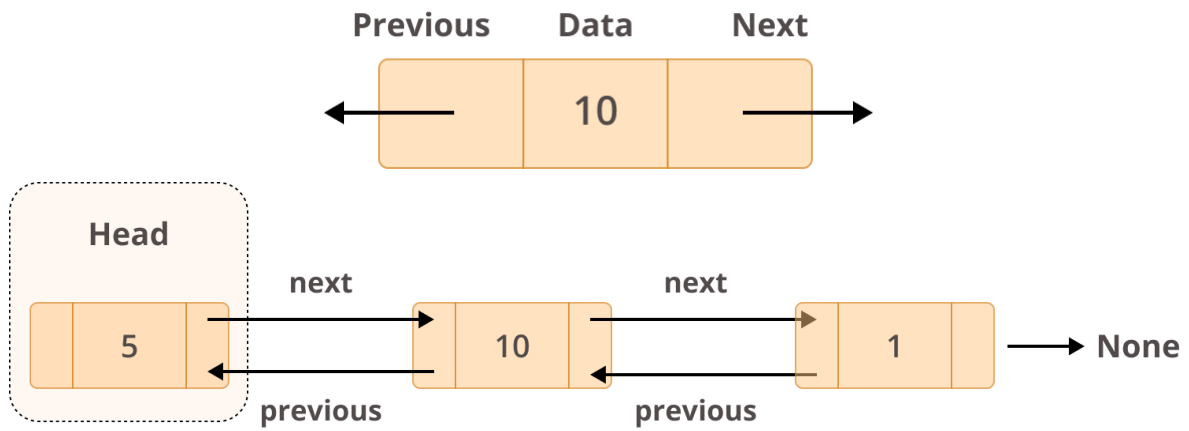
```

Vkládání a odstraňování prvků

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

Varianty LSS

- Dvojitě spojovaný seznam - pro deque

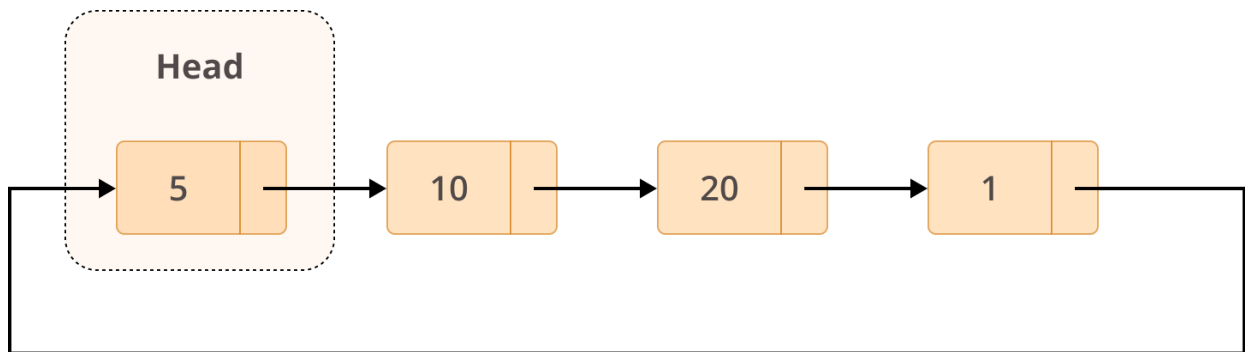


```

1 class Node:
2     def __init__(self, data):
3         self.data = data
4         self.next = None
5         self.previous = None

```

- Cyklický seznam



Cyklickým seznamem můžeme procházet počínaje libovolným prvkem:

```

1 # Kruhový seznam - pointer u poslední položky ukazuje na začátek seznamu.
2 from _collections_abc import Generator
3
4
5 class Node:
6     def __init__(self, value):
7         """Položku inicializujeme hodnotou value"""
8         self.value = value
9         self.next = None
10
11     def __repr__(self):
12         """Reprezentace objektu na Pythonovské konzoli"""
13         return str(self.value)
14
15
16 class CircularLinkedList:
17     def __init__(self, values = None):

```

```

18     self.head = None
19     if values is not None:
20         self.head = Node(values.pop(0))
21         node = self.head
22         for val in values:
23             node.next = Node(val)
24             node = node.next
25         node.next = self.head
26
27     def traverse(self, starting_point: Node = None) -> Generator[Node, None,
None]:
28         if starting_point is None:
29             starting_point = self.head
30         node = starting_point
31         while node is not None and (node.next != starting_point):
32             yield node
33             node = node.next
34         yield node
35
36     def print_list(self, starting_point: Node = None) -> None:
37         nodes = []
38         for node in self.traverse(starting_point):
39             nodes.append(str(node))
40         print(" -> ".join(nodes))
41

```

Jak to funguje:

```

1  >>> clist = CircularLinkedList([1,2,3,4,5])
2  >>> clist.print_list()
3  1 -> 2 -> 3 -> 4 -> 5
4  for node in clist.traverse():
5      pass
6  node
7  5
8  node.next
9  1
10

```