

# Programování 2

---

## 5. cvičení, 16-3-2022

---

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/Programovani-2>.
2. **Domácí úkoly:** 3 nové úkoly od minulého cvičení.
  1. Utřídění seznamu slov pomocí bucket sort
  2. Výpis všech k-ciferných čísel s daným ciferným součtem
  3. Medián

#### Dnešní program:

- Kvíz
  - Python: jak rychlý je můj kód
  - Domácí úkoly
  - Opakování: Třídění
  - Lineární spojovaný seznam
- 

### Na zahřátí

---

*"Before software can be reusable it first has to be usable." – Ralph Johnson*

Opakovaná použitelnost kódu se přeceňuje. Největší využití mívají krátké kousky kódu. Velké knihovny sami po sobě dědíme zřídka.

### Co dělá tento kód

```
1 l1 = [1, 2, 3, 4]
2 for k in l1:
3     k = k+1
4 l1
```

Pokud chcete změnit seznam, iterujte přes indexy a ne přes položky. Viz neměnné typy a pointry. Jaký je správný kód pro přičtení 1 ke všem položkám?

# Jak rychlý je můj kód?

Jak můžeme měřit rychlost kódu?

**Příklad:** ciferný součet

```
1 def cif_soucet_1(cislo: int) -> int:
2     """mod 10 dává číslici, div 10 zbytek"""
3     soucet = 0
4     while cislo:
5         soucet += cislo % 10
6         cislo //= 10
7     return cislo
8
9 def cif_soucet_2(cislo: int) -> int:
10    return sum(map(int, str(cislo)))
```

Který kód je rychlejší?

```
1 from timeit import Timer
2
3 def cif_soucet_1(cislo: int) -> int:
4     """mod 10 dává číslici, div 10 zbytek"""
5     soucet = 0
6     while cislo:
7         soucet += cislo % 10
8         cislo //= 10
9     return soucet
10
11 def cif_soucet_2(cislo: int) -> int:
12     return sum(map(int, str(cislo)))
13
14
15 t1 = Timer("cif_soucet_1(123456789)", "from speedy import cif_soucet_1")
16 print(1, t1.timeit(number = 1000000))
17
18 t2 = Timer("cif_soucet_2(123456789)", "from speedy import cif_soucet_2")
19 print(2, t2.timeit(number = 1000000))
20 -----
21 1 0.7875468000000001
22 2 1.151669
23 1 0.7636549000000001
24 2 1.1202397
25
```

Toto je dost nepohodlné, pohodlnější je použít *magic* `%%timeit` v IPythonu, tedy např.

- v Pythonské konzoli PyCharmu
- v jakémkoli typu Jupyter notebooku - Google Colab, JupyterLab a pod.

# Čtení z konzole

Kód, který v ReCodExu **nefunguje**

```
1 ...
2 s = []
3 line = input()
4 while line != "-end-":
5     s.add(int(line))
6 ...
```

Takovýto kód způsobuje hlášení *End-Of-File Error* v ReCodExu.

`input()` funguje jinak při čtení vstupu z konzole a při čtení vstupu přesměrovaného ze souboru.

Ve vašem IDE se načte "-end-" a kód vám chodí.

V ReCodExu se načte "-end-\n" a dostanete *End-Of-File Error*.

## Řešení:

1. Testujte to, co pozitivně víte: že na posledním řádku se nachází "-end-":

```
1 ...
2 s = []
3 line = input()
4 while "-end-" not in line:
5     s.append(int(line))
6     line = input()
7 ...
```

2. Očistěte poslední řádek od případných dalších znaků (mezer, zanků konce řádku a pod.)

```
1 ...
2 s = []
3 line = input().strip()
4 while "-end-" not in line:
5     s.append(int(line))
6     line = input().strip()
7 ...
```

**Bezpečné řešení:** Nepoužívejte `input()`

```
1 from sys import stdin
2
3 s = []
4 with stdin as vstup:
5     for line in vstup:
6         if "-end-" in line:
7             break
8         s.append(int(line.strip()))
```

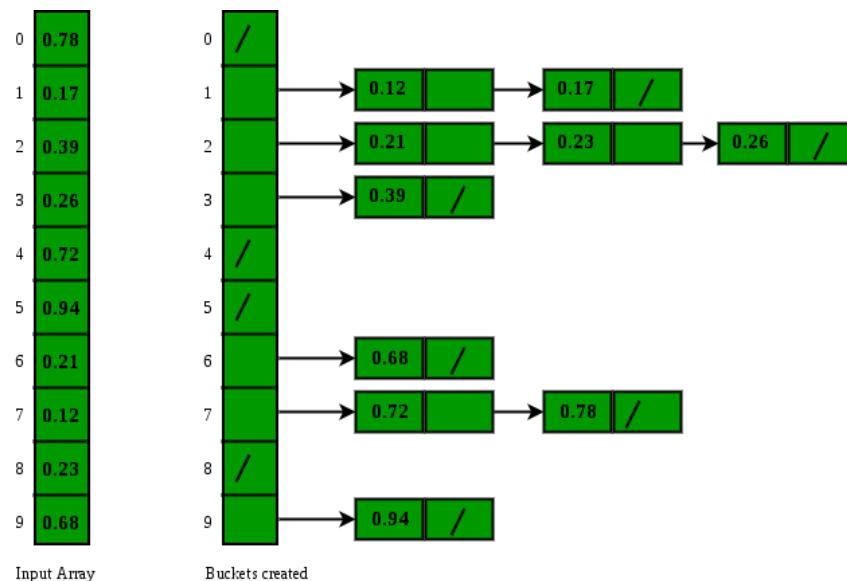
Tady se všechno chová stejně na všech platformách, a pro ukončení vstupu z konzole nemusíte mačkat Ctrl-D.

## Opakování:

## Třídění

### Bucket sort

- Nahrubo si setřídíme čísla do přihrádek
- Setřídíme obsah přihrádek
- Spojíme do výsledného seznamu



- Jednoduchý algoritmus
- Občas musíme popřemýšlet, jak vytvořit přihrádky (viz domácí úkol)

### Quick sort

Souvisí s domácím úkolem o mediánu.

```
1 from random import randint
2
3
4 def quick_sort(b):
5     if len(b) < 2:
6         return b
7     if len(b) == 2:
8         return [min(b), max(b)]
9     pivot = b[randint(0, len(b)-1)]
10    lows = [x for x in b if x < pivot]
11    pivots = [x for x in b if x == pivot]
12    highs = [x for x in b if x > pivot]
13    return quick_sort(lows) + pivots + quick_sort(highs)
```

```

14
15
16 data = [randint(1,100) for _ in range(10)]
17
18 print(data)
19 print(quick_sort(data))

```

## Lineární spojovaný seznam

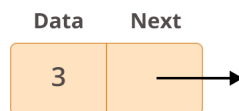
Proč se zabýváme datovými strukturami, když vše máme v Pythonu hotové?

Trénink mozku:

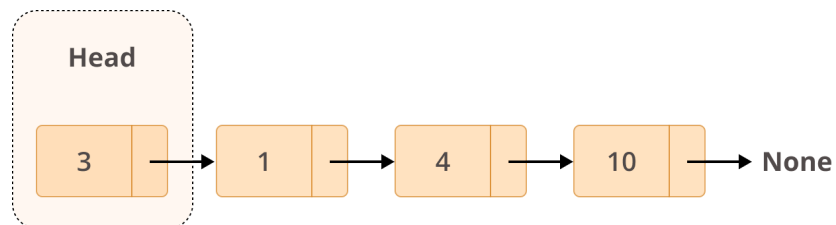
- Musíme umět rozebrat algoritmy na kolečka a šroubky
- Chceme identifikovat společné vzory v algoritmech
- Základní struktury umíme podrobně analyzovat.

### Proč LSS?

"Převratný vynález": **spojení dat a strukturní informace:**

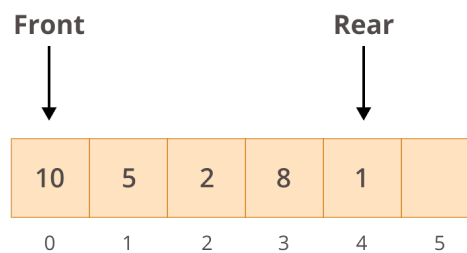


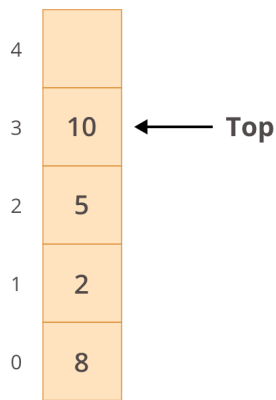
Takovéto jednotky pak umíme spojovat do větších struktur. LSS je nejjednodušší z nich.



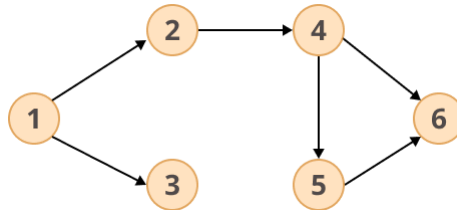
**Aplikace:**

- Fronty a zásobníky





- Grafy



## Spojované seznamy v Pythonu

`list` v Pythonu je [dynamické pole](#).

- Zanedbatelný rozdíl ve využití paměti
- Důležitá je časová efektivnost

### Interface:

- přidávání prvků: `insert` a `append`
- odebírání prvků: `pop` a `remove`

Náročnost:

- na konci:  $O(1)$
- kolem začátku: musíme nejdřív nalézt správné místo,  $O(n)$

Přístup k prvkům: `list`  $O(1)$ , LSS  $O(n)$

## `collections.deque`

Pythonovská implementace nejbližší - co do API - k LSS.

- Rychlé přidávání/odebírání prvků na začátku/konci
- Přístup k prvkům přes začátek / konec

```
1 from collections import deque
2 llist = deque("abcde")
3 llist
4 deque(['a', 'b', 'c', 'd', 'e'])
5
6 llist.append("f")
```

```

7  llist
8  deque(['a', 'b', 'c', 'd', 'e', 'f'])
9
10 llist.appendleft("-")
11 llist
12 deque(['-', 'a', 'b', 'c', 'd', 'e', 'f'])
13
14 llist.pop()
15 'f'
16
17 llist.popleft()
18 '-'
19 llist
20 deque(['a', 'b', 'c', 'd', 'e'])
21
22 # off-brand API
23 llist[1]
24 'b'

```

`deque` podporuje také interface pole a umíme přistupovat k prvkům přes index. Tento přístup je rychlý.

## Implementujeme spojovaný seznam

Existuje víc možností implementace - přes seznam, slovník nebo class.

```

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

```

Samotný LSS má jenom hlavu:

```

1  class LinkedList:
2      def __init__(self):
3          self.head = None

```

Přidáme metody, které nám seznam vytlačí na konzoli a při tom se naučíme seznamem procházet:

```

1  class Node:
2      def __init__(self, value):
3          self.value = value
4          self.next = None
5
6      def __str__(self):
7          return self.data
8
9  class LinkedList:
10     def __init__(self):
11         self.head = None

```

```

12
13     def __str__(self):
14         node = self.head
15         nodes = []
16         while node is not None:
17             nodes.append(node.value)
18             node = node.next
19         nodes.append("None")
20         return " -> ".join(nodes)

```

Nyní můžeme nějaký seznam opravdu vytvořit:

```

1  # Importujeme kód do IPythonové konzole jako modul:
2  >>> from linked_list1 import * # Obecně ne-ne, ale pro náš malý kód OK.
3
4  >>> llist = LinkedList()
5  >>> str(llist)
6  None
7
8  >>> first_node = Node("a")
9  >>> llist.head = first_node
10 >>> str(llist)
11 a -> None
12
13 >>> second_node = Node("b")
14 >>> third_node = Node("c")
15 >>> first_node.next = second_node
16 >>> second_node.next = third_node
17 >>> str(llist)
18 a -> b -> c -> None

```

Vylepšíme `__init__`, abychom mohli vytvářet seznam pohodlněji:

```

1  def __init__(self, values=None):
2      self.head = None
3      if values is not None:
4          node = Node(value=values.pop(0))
5          self.head = node
6          for elem in values:
7              node.next = Node(value=elem)
8              node = node.next

```

## Procházení seznamem

```

1  def __iter__(self):
2      node = self.head
3      while node is not None:
4          yield node
5          node = node.next

```



a vyzkoušíme:

```
1 >>> llist = LinkedList(["a", "b", "c", "d", "e"])
2 >>> str(llist)
3 a -> b -> c -> d -> e -> None
4
5 >>> for node in llist:
6 ...     print(node)
7 a
8 b
9 c
10 d
11 e
```

## Vkládání prvků do seznamu

- add\_first, add\_last

```
1 def add_first(self, node):
2     node.next = self.head
3     self.head = node
```

```
1 def add_last(self, node):
2     if self.head is None:
3         self.head = node
4         return
5     for current_node in self:
6         pass
7     current_node.next = node
```

a vyzkoušíme:

```
1 >>> llist = LinkedList()
2 >>> str(llist)
3 None
4
5 >>> llist.add_first(Node("b"))
6 >>> str(llist)
7 b -> None
8
9 >>> llist.add_first(Node("a"))
10 >>> str(llist)
11 a -> b -> None
```

```

1  >>> llist = LinkedList(["a", "b", "c", "d"])
2  >>> str(llist)
3  a -> b -> c -> d -> None
4
5  >>> llist.add_last(Node("e"))
6  >>> str(llist)
7  a -> b -> c -> d -> e -> None
8
9  >>> llist.add_last(Node("f"))
10 >>> str(llist)
11 a -> b -> c -> d -> e -> f -> None

```

- add\_after, add\_before

Musíme nejdřív nalézt, kam prvek vložit, a přitom uvážit, že umíme seznamem procházet pouze jedním směrem.

```

1  def add_after(self, target_node_value, new_node):
2      if self.head is None:
3          raise Exception("List is empty")
4
5      for node in self:
6          if node.value == target_node_value:
7              new_node.next = node.next
8              node.next = new_node
9              return
10
11     raise Exception("Node with value '%s' not found" % target_node_value)

```

```

1  >>> llist = LinkedList()
2  >>> llist.add_after("a", Node("b"))
3  Exception: List is empty
4
5  >>> llist = LinkedList(["a", "b", "c", "d"])
6  >>> str(llist)
7  a -> b -> c -> d -> None
8
9  >>> llist.add_after("c", Node("cc"))
10 >>> str(llist)
11 a -> b -> c -> cc -> d -> None
12
13 >>> llist.add_after("f", Node("g"))
14 Exception: Node with value 'f' not found

```

```

1  def add_before(self, target_node_value, new_node):
2      if self.head is None:
3          raise Exception("List is empty")
4

```

```

5     if self.head.value == target_node_value:
6         return self.add_first(new_node)
7
8     prev_node = self.head
9     for node in self:
10        if node.value == target_node_value:
11            prev_node.next = new_node
12            new_node.next = node
13            return
14        prev_node = node
15
16    raise Exception("Node with value '%s' not found" % target_node_value)

```

```

1  llist = LinkedList()
2  >>> llist.add_before("a", Node("a"))
3  Exception: List is empty
4
5  >>> llist = LinkedList(["b", "c"])
6  >>> str(llist)
7  b -> c -> None
8
9  >>> llist.add_before("b", Node("a"))
10 >>> str(llist)
11 a -> b -> c -> None
12
13 >>> llist.add_before("b", Node("aa"))
14 >>> llist.add_before("c", Node("bb"))
15 >>> str(llist)
16 a -> aa -> b -> bb -> c -> None
17
18 >>> llist.add_before("n", Node("m"))
19 Exception: Node with value 'n' not found

```

## Odstraňujeme prvky

```

1  def remove_node(self, target_node_value):
2      if self.head is None:
3          raise Exception("List is empty")
4
5      if self.head.value == target_node_value:
6          self.head = self.head.next
7          return
8
9      previous_node = self.head
10     for node in self:
11         if node.value == target_node_value:
12             previous_node.next = node.next
13             return
14         previous_node = node
15

```

16

```
raise Exception("Node with value '%s' not found" % target_node_value)
```