

Programování 2

7. cvičení, 1-4-2025

Farní oznamy

1. Tento text a kódy ke cvičení najdete v repozitáři cvičení na <https://github.com/PKvasnick/Programovani-2>.

2. **Domácí úkoly:**

Prodloužil jsem termín odevzdání: 10b do neděle a pak ale už nic (týká se nových řešení - pokud jste začali řešit před termínem, body dostanete).

- Úmyslně jsem vás s LSS hodil do vody, abyste měli představu o čem budeme mluvit na dnešním cvičení.
- Dostanete jediný další domácí úkol se standardními termíny.

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
- Lineární spojované seznamy a jejich variace

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: šikovná 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 sližité ani příliš jednoduché
 - Jednoduché hry: Sudoku, Piškvorky (PyGame)
 - Matematické knihovny nebo výukové materiály: numerická diferenciace, maticové operace, symbolické manipulace (SymPy), geometrie, diferenciální rovnice, aplikace SAT solverů (piškvorky a pod.), zajímavé algoritmy (dancing links) atd.
 - Strojové učení - neurální sítě, rozhodovací stromy, Gaussovske směsi, logistická regrese a pod.
 - Fyzikální a statistické simulace - difúze částic v složitém prostředí, perkolace, Monte Carlo - Metropolis-Hastings a pod., kSinetika, pohyb osob v budově s výtahem, pohyb zákazníků v nákupním středisku, pohyb lidí na Matějské pouti, epidemiologické modely apod.
 - Statistika, zpracování a prezentace dat, Bayesovské metody (STAN) a pod.
 - Nějaká témata máme, podívejte se třeba na web Martina Mareše: <http://mj.ucw.cz/vyuka/zap/>
 - termín pro zadání závěrečného programu: **do konce dubna**, pak dostanete témata přidělena.

Co se očekává, že uděláte

Do konce dubna:

1. Rozmyslete si anebo vyhledejte vhodné téma, a napište mi e-mail se svou představou. Také napište, pokud nemáte představu, co byste si vybrali.
2. Dohodneme se na zadání (10-15 řádek textu) a odsouhlasíme si ho.

(Nejpozději) do konce srpna:

1. Po odsouhlasení tématu se můžete pustit do díla. Nastudujete si, co potřebujete k implementaci a začnete psát kód.
2. Otestujete, jak kód funguje. Pokud se zaseknete, napíšete mi.
3. Vytvoříte technický popis a uživatelský návod (nemusí být rozsáhlé, pouze výstižné, 1 A4 je spíš rozsáhlejší dokumentace).
4. Zřídíte si pro svůj kód GitHub repozitář, uploadujete ho, dpolníte náležitosti (README.md, setup.py, requirements.txt) a pošlete mi link.

(Nejpozději) do konce září:

1. Pokud od vás neuslyším do poloviny září, začnu zjišťovat, co se děje.
2. Doiterujeme ke konečné verzi (obvykle kolem 80% zadání nepotřebuje iteraci) a dostanete zápočet.

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 chyby. Naučí vás to některé věci automaticky nedělat.

Co dělá tento kód

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

Návod:

`dir(objekt)` vypíše atributy objektu.

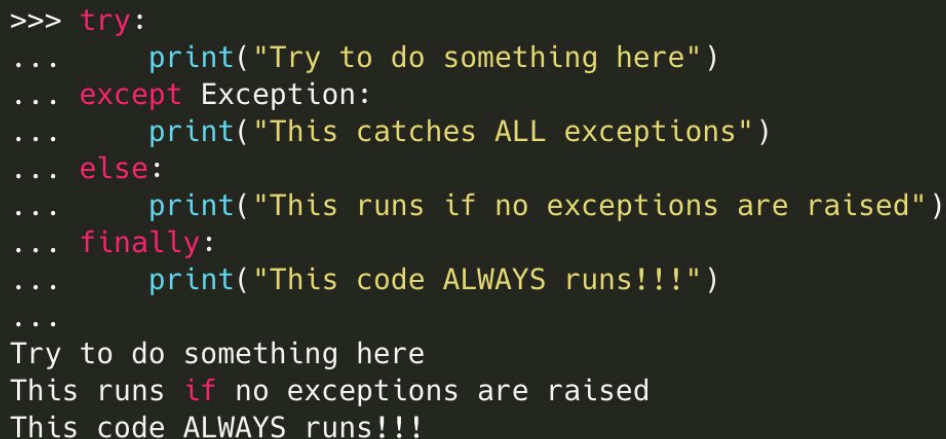
None

`None` je v Pythonu unikátní objekt, představující chybějící hodnotu. Máme jedno jediné `None`, prostupující všechna prázdná místa v Pythonském vesmíru. `None` se ničemu nerovná, jenom chybí. Takže i když v Pythonu lze napsat `if uzel.dalsi == None:`, podstatě věci lépe odpovídá `if uzel.dalsi is None:`.

Mini tutoriál: Výjimky

Výjimky řada z vás běžně používá v kódu, ale dáme si opakování:

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. Také když chyba v jiné části kódu způsobí chybu ve vaší části, chcete vědět, že se to děje a netajit to před původcem.

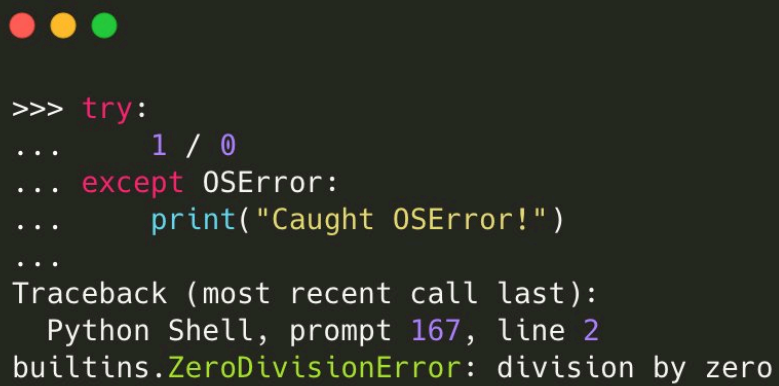
Co udělat se zachycenou výjimkou? Co potřebujete:

```

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

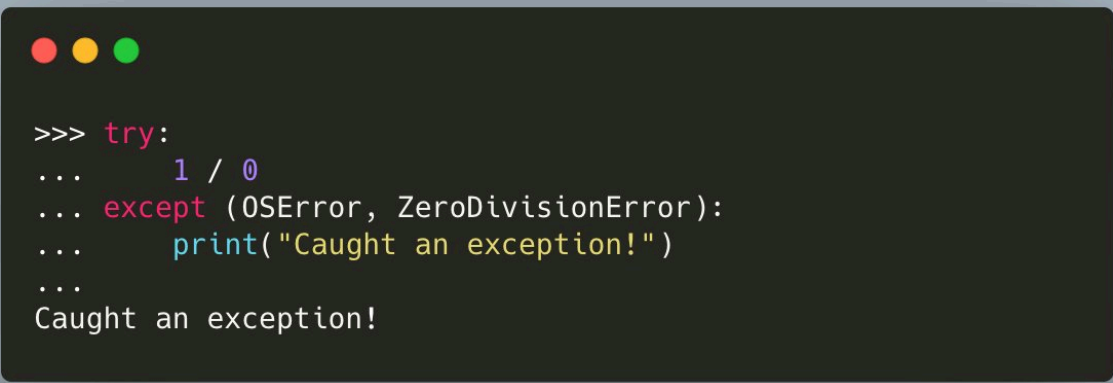
```

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 ukazovali 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`:

```
import os

try:
    os.remove("muj_soubor.txt")
except FileNotFoundError:
    pass
```

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

```
from contextlib import suppress

with suppress(FileNotFoundError):
    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.OSError: [Errno Broken] Pipe
```

`try-except` namísto `if-else`: Radši žádat o odpuštění než o povolení

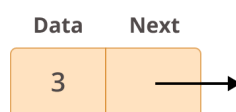
Pokud potřebujeme zachytit zřídka se vyskytující stav, můžeme namísto `if-else` použít `try-except`.

Podmíněný příkaz přidává prodloužení ke zpracování obou větví, zatímco `try-except` přidává prodloužení prakticky jenom ke větvi `except`.

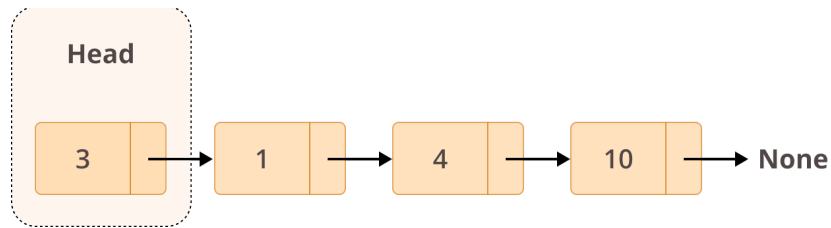
Není dobré takovýto způsob nadužívat, ale je to Pythonský způsob vyjadřování a neváhejte ho ve vhodné situaci použít.

Lineární spojovaný seznam

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

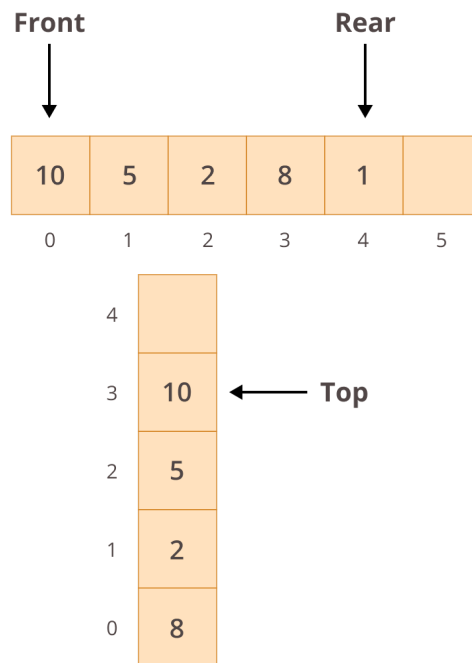


Takovéto jednotky pak umíme spojovat do větších struktur. LSS je nejjednodušší z nich.



Aplikace:

- Fronty a zásobníky



Spojované seznamy v Pythonu

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

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

`collections.deque` je implementace fronty se dvěma konci.

- `append` / `appendleft`
- `pop` / `popleft`

Omezené programování

Dosud jsme se učili hledat optimální prostředky pro implementaci různých věcí - binární vyhledávání a modul *bisect*, haldy a modul *heapq*, množiny a slovníky a pod.

Teď máme jinou situaci: Dobrovolně se zříkáme některých prostředků (např. přístup k položkám přes index) a zkoumáme, co všechno dokážeme udělat.

Implementujeme spojovaný seznam

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

```
# 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)

class LinkedList:
    def __init__(self, values = None):
        """Spojovany seznam volitelne inicializujeme seznamem hodnot"""
        if values is None:
            self.head = None
            return
        self.head = Node(values.pop(0)) # pop vrati a odstrani hodnotu z values
        node = self.head
        for value in values:
            node.next = Node(value)
            node = node.next

    def __repr__(self):
        """Reprezentace na Pythonovske konzoli:
        Hodnoty spojene sipkami a na konci None"""
        values = []
        node = self.head
        while node is not None:
            values.append(str(node.value))
            node = node.next
        values.append("None")
        return " -> ".join(values)

    def __iter__(self):
        """Iterator prochazejici _hodnotami_ seznamu,
        napr. pro pouziti v cyklu for"""
        node = self.head
        while node is not None:
            yield node.value
            node = node.next

    def add_first(self, node):
        """Prida polozku na zacatek seznamu,
        tedy na head."""
        node.next = self.head
        self.head = node
```

```
def add_last(self, node):
    """Prida položku na konec seznamu."""
    p = self.head
    prev = None
    while p is not None:
        prev, p = p, p.next
    prev.next = node
```

Vkládání a odstraňování prvků

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

```
# Simple linked list 2

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

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

class LinkedList:
    def __init__(self, values = None):
        """Spojovaný seznam volitelně inicializujeme seznamem hodnot"""
        if values is None:
            self.head = None
            return
        self.head = Node(values.pop(0)) # pop vrátí a odstraní hodnotu z values
        node = self.head
        for value in values:
            node.next = Node(value)
            node = node.next

    def __repr__(self):
        """Reprezentace na Pythonovské konzoli:
        Hodnoty spojené šipkami a na konci None"""
        values = []
        node = self.head
        while node is not None:
            values.append(str(node.value))
            node = node.next
        values.append("None")
```

```

        return " -> ".join(values)

def __iter__(self):
    """Iterator prochazejici položkami seznamu,
    napr. pro pouziti v cyklu for"""
    node = self.head
    while node is not None:
        yield node
        node = node.next

def values(self):
    vals = []
    for node in self:
        vals.append(node.value)
    return vals

def get_last_node(self):
    for node in self:
        pass
    return node

def __len__(self):
    count = 0
    for node in self:
        count += 1
    return count

def add_first(self, val):
    """Prida položku na zacatek seznamu,
    tedy na head."""
    node = Node(val)
    node.next = self.head
    self.head = node

def add_last(self, val):
    """Prida položku na konec seznamu."""
    for p in self:
        pass
    node = Node(val)
    p.next = node

def get_node(self, target_val):
    for p in self:
        if p.value == target_val:
            return p
    else:
        return None

def add_after(self, target_val, new_val):
    p = self.get_node(target_val)
    if p is None:
        raise ValueError(f"{target_val} se nenachazi v seznamu.")
    node = Node(new_val)

```

```

node.next = p.next
p.next = node

def add_before(self, target_val, new_val):
    if self.head.value == target_val:
        node = Node(new_val)
        node.next = self.head
        self.head = node
        return
    prev = self.head
    p = prev.next
    while (p is not None) and (p.value != target_val):
        prev = p
        p = p.next
    if p is None:
        raise ValueError(f"{target_val} se nenachazi v seznamu.")
    node = Node(new_val)
    node.next = p
    prev.next = node

def remove(self, target_val):
    p = self.head
    if p.value == target_val:
        self.head = p.next
        del p
        return
    prev = p
    p = p.next
    while (p is not None) and (p.value != target_val):
        prev = p
        p = p.next
    if p is None:
        raise ValueError(f"{target_val} se nenachazi v seznamu.")
    prev.next = p.next
    del p

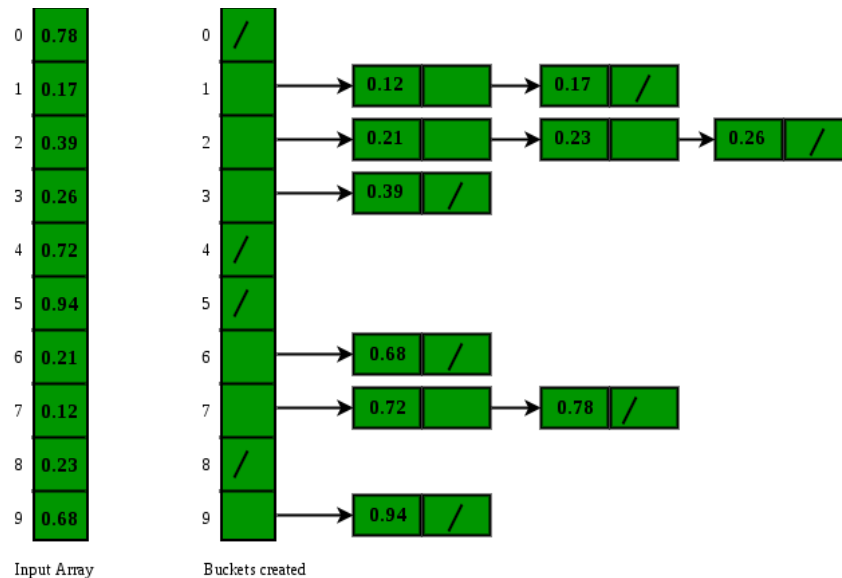
```

Třídění LSS

Utříděný seznam: `add` vloží prvek na správné místo

Jak utřídít již existující seznam?

Bucket sort vyžaduje složitou datovou strukturu



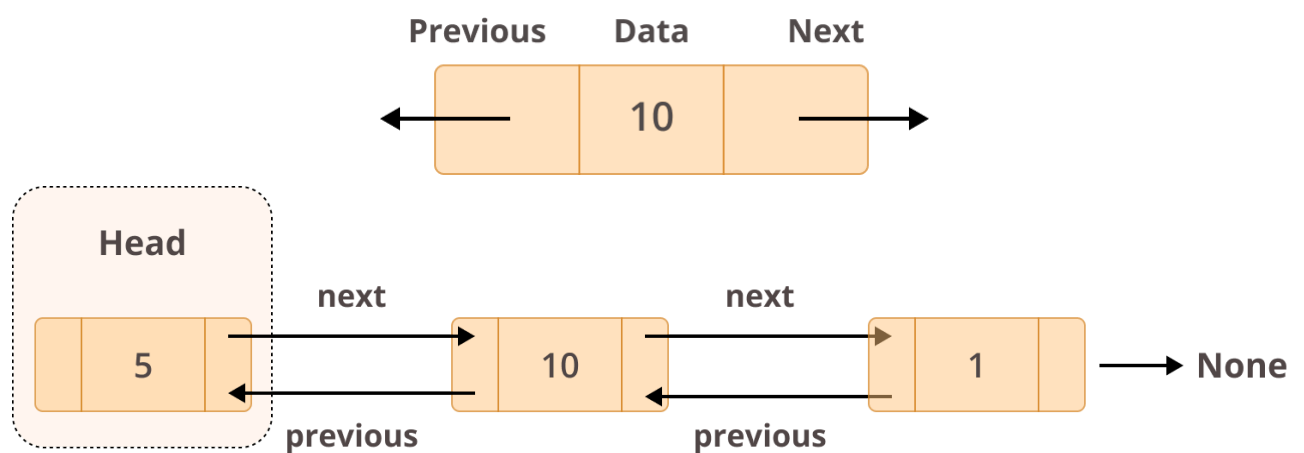
Heapsort potřebuje skákat z k na $2k$ (umíme, ale neradi děláme) a zpátky (neumíme, nebo jenom ztěžka)

Máme třídící algoritmus, který by vystačil s průchody v jednom směru? Umíte ho implementovat v LSS?

Bubble sort, insert sort: volíme tak, abychom se vyhnuli složitým logickým situacím ohledně existence nebo neexistence uzlů.

Varianty LSS

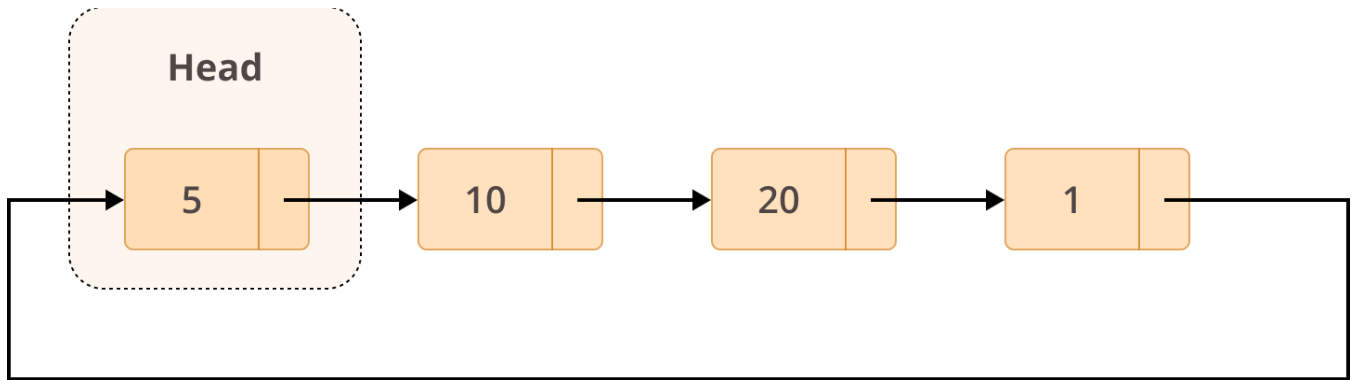
- **Dvouhlavý seznam:** kromě *head* také udržujeme pointer na poslední prvek, *tail*.
- **Dvojitě spojovaný seznam** - pro deque



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

Místy nepříjemné programování kvůli složitější logice v degenerovaných a okrajových případech.

- **Cyklický seznam**



Cyklickým seznamem můžeme procházet počínaje libovolným prvkem:

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

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

    def __repr__(self):
        """Reprezentace objektu na Pythonovské 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:

```
>>> circular_llist = CircularLinkedList()
>>> circular_llist.print_list()
None

>>> a = Node("a")
>>> b = Node("b")
>>> c = Node("c")
>>> d = Node("d")
>>> a.next = b
>>> b.next = c
>>> c.next = d
>>> d.next = a
>>> circular_llist.head = a
>>> circular_llist.print_list()
a -> b -> c -> d

>>> circular_llist.print_list(b)
b -> c -> d -> a

>>> circular_llist.print_list(d)
d -> a -> b -> c
```

Domácí úkol

Protože se řada z vás potřebuje zabývat předešlými úkoly, dostanete jediný další domácí úkol:

Fronta zvířat, kde si dále procvičíte práci s LSS. Tato úloha je o něco jednodušší než úkol o frontě lidí různých věků.