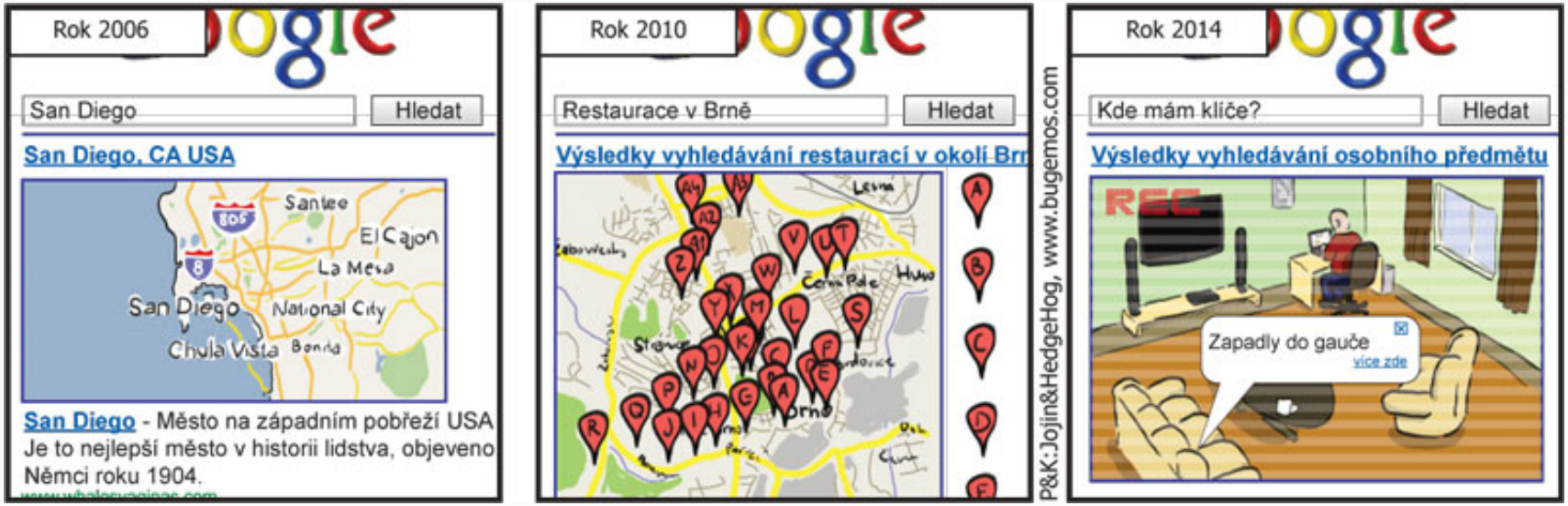
Vyhledávání; datové struktury; proměnné a paměť podrobněji

IB111 Základy programování

Tomáš Vojnar (na základě slajdů Nikoly Beneše)

24. října 2025

Vyhledávání



- Častý problém:
  - web, slovník, informační systémy, databáze, ...
  - Dílčí krok v mnoha algoritmech.

zdroj obrázku: http://www.bugemos.com/

Vyhledávání v obecném seznamu

- Vstup: seznam čísel[1] + dotaz (číslo).
- Výstup: odpověď `True`/`False` – číslo je/není v seznamu.
- Alt. výstup: index čísla v seznamu/nějaká speciální hodnota, např. `None`, pokud číslo v seznamu není.

- V nejhorším případě musíme projít celý seznam.
- Časová složitost: *lineární* (vůči délce seznamu).
- Jde to lépe? V obecném seznamu ne.

[1] Nebo řetězců, nebo čehokoli jiného, co se dá hledat.

Lineární vyhledávání

linear_search.py

```
def contains(haystack: list[int], needle: int) -> bool:
    # ... (viděli jsme minule, list_ops.py)

def index_of(haystack: list[int], needle: int) \
        -> int | None:
    for index, element in enumerate(haystack):
        if element == needle:
            return index
    return None
```

Vyhledávání v Pythonu (odteď i v `ib111.py`)

- Operátor `in` (vrací `bool`).
- `seznam.index(prvek)` – při nenalezení zahlásí `ValueError`.

Příklad: Otrávená studna

- Máme osm studen, víme, že (právě) jedna z nich je otrávená.
- Laboratorní rozbor:
  - pozná přítomnost jedu ve vodě,
  - je drahý.
- Kolik rozborů potřebujeme k tomu, abychom s jistotou našli otrávenou studnu?

Řešení: stačí *tři* rozbory.

- Můžeme smíchat vodu z více studní.
- Každý rozbor nám zmenší „okruh podezřelých“ na polovinu

Obecně: Kolik rozborů potřebujeme pro n studen?

- Tj. kolikrát se dá n dělit dvěma, než spadne na (nebo pod) 1?
- (Dvojkový) logaritmus.

Připomenutí: logaritmus

$x = b^y$ právě tehdy, když $y = \log_b(x)$.

$$\log_{10}(1000) = 3 \qquad \log_3(81) = 4$$
$$\log_2(16) = 4 \qquad \log_2(2) = 1$$
$$\log_2(1024) = 10 \qquad \log_5(1) = 0$$
$$\log_2(\sqrt{2}) = 0.5 \qquad \log_{0.5}(4) = -2$$

- $\log_b(x \cdot y) = \log_b(x) + \log_b(y)$.
- $b^{log_b(x)} = x$.
- $\log_b(x) = \log_a(x)/\log_a(b)$.

Vyhledávání v seřazeném seznamu

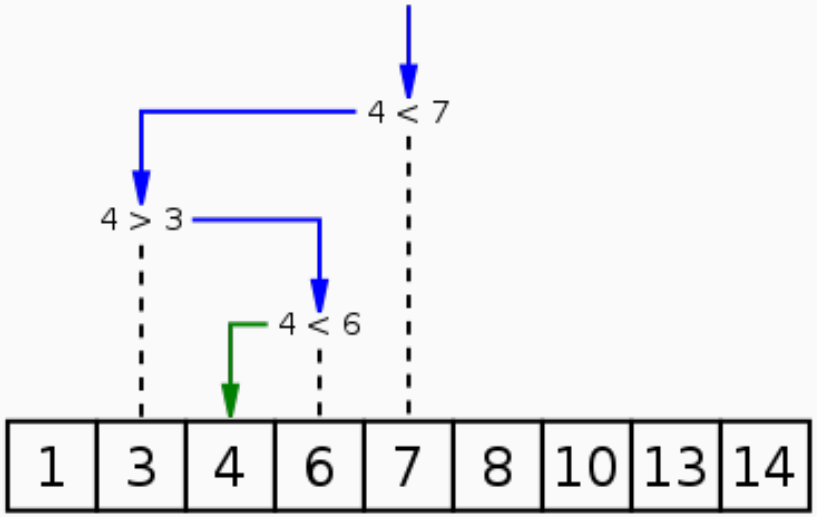
- Vstup: seřazený seznam čísel + dotaz (číslo).
- Výstup: odpověď `True`/`False` – číslo je/není v seznamu, případně index nalezeného prvku.

Řešení

- „Naivní“ řešení: procházení celého seznamu.
  - Časová složitost: *lineární*.
  - Použitelné pro krátké seznamy.
  - Nevhodné pro delší seznamy.
- „Rozumné“ řešení: půlení intervalu.
  - Časová složitost *logaritmická* k délce seznamu.
  - Jak implementovat?

Binární vyhledávání

- Půlení intervalu – podobné hádání čísel, výpočtu odmocniny, ...
- Podíváme se na *prostřední* prvek seznamu,
  - podle jeho hodnoty buď skončíme,
    nebo pokračujeme doleva nebo doprava,
  - udržujeme si „dolní“ (inkluzivní) a „horní“ mez (exkluzivní).



Binární vyhledávání

binary_search.py

```
def binary_search(haystack: list[int], needle: int) \
        -> bool:
    lower = 0
    upper = len(haystack)
    while lower < upper:
        middle = (lower + upper) // 2
        if haystack[middle] == needle:
            return True

        if haystack[middle] < needle:
            lower = middle + 1
        else:
            upper = middle

    return False
```

Binární vyhledávání

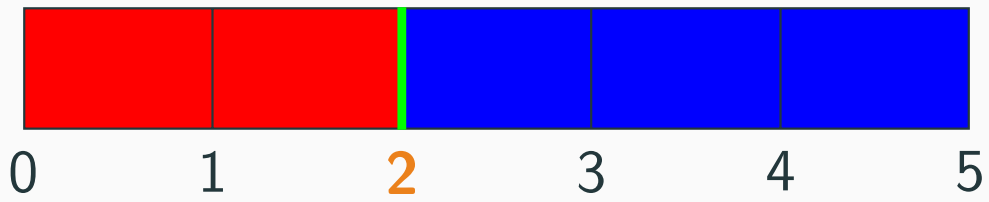
Zajímavé varianty k rozmyšlení

- Co když se v seznamu prvky mohou opakovat?
  - Index prvního prvku s danou hodnotou.
  - Index posledního prvku s danou hodnotou.
  - Jak si zachovat efektivitu?
- Co když prvek v seznamu není, ale zajímalo by nás, kam by se zařadil?
  - Poslední prvek menší než hledaný.
  - První prvek větší než hledaný.

Pro zajímavost: Obecnější binární vyhledávání

Obecnější problém

- Vstup: seznam délky n obsahující prvky dvou druhů:
  - „levé“ a „pravé“.
  - Vstupní podmínka: všechny „levé“ jsou před „pravými“.
- Výstup: číslo i takové, že
  - na indexech v polouzavřeném intervalu $[0, i)$ jsou „levé“ prvky,
  - na indexech v polouzavřeném intervalu $[i, n)$ jsou „pravé“ prvky.
- Jinými slovy: hledáme „dělicí čáru“.



Zpět k původnímu problému

- „levé“ prvky: $<$ hledané číslo.
- „pravé“ prvky: $\geq$ hledané číslo.

```
def find_boundary(nums: list[int], limit: int) -> int:
    lower = 0
    upper = len(nums)
    while lower < upper:
        middle = (lower + upper) // 2
        if nums[middle] < limit:
            lower = middle + 1
        else:
            upper = middle
    return lower
```

Zpět k původnímu problému

- máme index i takový, že
  - na indexech v intervalu $[0, i)$ jsou prvky $<$ hledaný,
  - na indexech v intervalu $[i, n)$ jsou prvky $\geq$ hledaný.
- Jak zjistíme, zda seznam obsahuje hledané číslo?
  - Stačí se podívat na index i,
  - pokud $i < n$.

```
def binary_search_alt(haystack: list[int],
                      needle: int) -> bool:
    b = find_boundary(haystack, needle)
    return b < len(haystack) and haystack[b] == needle
```

Složitost vyhledávání (binární vs. lineární)

Jak moc je logaritmická složitost lepší než lineární?

- Řekněme (pro představu), že
  - lineární vyhledávání provede $2n$ kroků,
  - binární vyhledávání provede $4 \log n$ kroků,
  - jeden krok trvá 1 ns.

n	binární vyhledávání	lineární vyhledávání
8	12 ns	16 ns
16	16 ns	32 ns
256	32 ns	512 ns
65536	64 ns	131 µs
4294967296	128 ns	8,6 s
18446744073709551616	256 ns	1170 **let**

Datové struktury – další operace

Rozšířené operace se seznamy

Modifikace seznamu

- `s.reverse()` obrátí seznam, in situ, vrací `None`.
- `s.extend(t)` přidá prvky seznamu `t` na konec `s`, vrací `None`.
- `s.insert(i, x)` přidá prvek `x` *na* index `i`, vrací `None`.
- `s.pop(i)` odstraní (a vrátí) prvek na indexu `i`.

Vytvoření nového seznamu

- `s + t` vytvoří nový seznam spojením dvou seznamů.

Hledání

- `x in s` vrátí `True`/`False`, jestli je prvek `x` v `s`.
- `s.index(x)` vrátí index 1. výskytu `x` (nebo nastane chyba).

Rozšířené operace se seznamy

Časová složitost

- `.reverse`, `in`, `.index` lineární vůči délce seznamu.
  - Podobně `sum`, `min`, `max`, `.copy`.
- `.extend` lineární vůči délce druhého seznamu.
- `.insert`, `.pop` lineární vůči počtu prvků od indexu ke konci.
  - Je třeba přeskládat seznam od zadaného indexu dál.
- `+` lineární vůči součtu délek obou seznamů.

Rozšířené operace s množinami

Vytvoření nové množiny

- `s | t` sjednocení.
- `s & t` průnik.
- `s - t` množinový rozdíl.

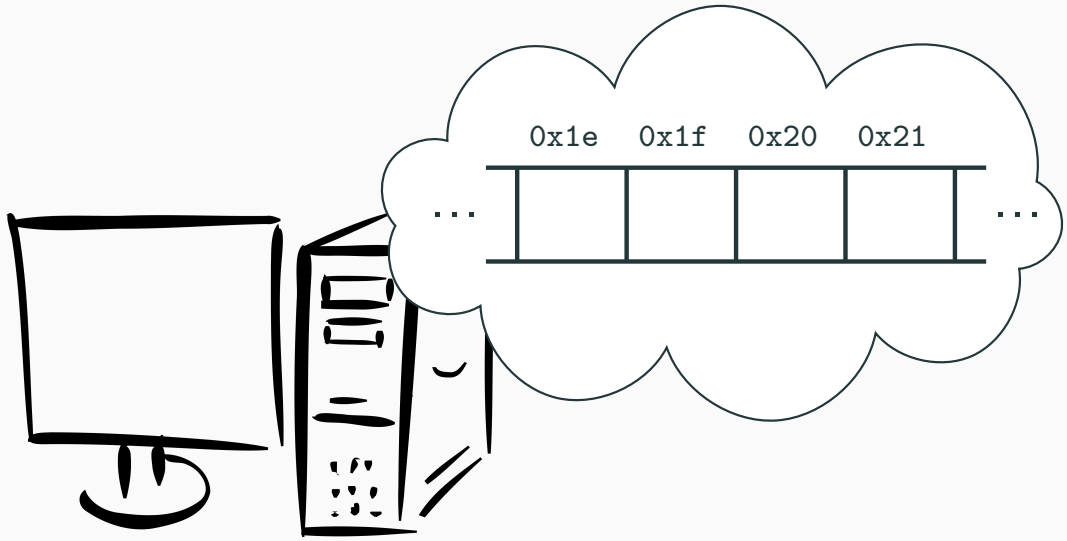
Modifikace množiny

- `s.update(t)` sjednocení.
- `s.intersection_update(t)` průnik.
- `s.difference_update(t)` množinový rozdíl.

Časová složitost

- Považujeme za lineární.
  - U modifikace vůči přidávané/výchozí/menší množině.
- Podrobnosti jsou komplikované
  - Zkuste si implementovat pomocí `.add`, `.remove`, `in`.

Proměnné a paměť



Globální a lokální proměnné

Globální proměnné

- V Pythonu přesněji modulové proměnné.
- Vytvořeny (definovány) přiřazením na úrovni modulu.
  - Tedy mimo funkce a třídy (zavedeme později).
- Jsou viditelné kdekoli v modulu.
  - Mohou být importovány, ale také překryty lokální proměnnou.

Lokální proměnné

- Vytvořeny (definovány) přiřazením uvnitř funkce.
  - Případně v některé speciální konstrukci (např. comprehension) – pak jsou lokální pro příslušnou konstrukci.
- Jsou viditelné jen v příslušné invokaci své funkce.
  - Neřešíme-li tzv. uzávěry – nad rámec IB111.

Globální a lokální proměnné – obecněji

Rozsah viditelnosti – scope

- Část kódu, v níž je jméno proměnné viditelné (použitelné).
  - Syntaktická entita, dříve zmíněný kontext je entita běhová.
- Rozsahem (*scope*) mohou být:
  - moduly (soubory se zdrojovým kódem),
  - třídy (o těch se dozvíme později),
  - funkce,
  - a jiné syntaktické konstrukce (závisí na konkrétním jazyce: např. list comprehension v Pythonu).

Globální a lokální proměnné

Doporučení

- Nepoužívat globální proměnné.
- Globální *konstanty* jsou v pořádku.
  - Konvence pojmenování konstant: velkými písmeny.

Proč?

- Porušují lokalitu kódu.
- Nečitelnost.
- Potenciální chyby.

Alternativy

- Předávání parametrů funkcím a vracení hodnot z funkcí.
- Vlastní datové struktury/třídy (o těch příště).
- Další (nad rámec předmětu):
  - proměnné v uzávěru, statické proměnné v jiných jazycích, ...

Připomenutí

Objekt

- Abstrakce paměti, úložiště/kontejner pro hodnoty.
- Má identitu, typ, hodnotu.

Proměnná

- Má jméno (s určitým rozsahem) a kontext.
- Python: může odkazovat na objekt, vazba se může měnit.
- V jiných jazycích může být vazba pevná.
- V Pythonu může mít typ, v některých jazycích má vždy typ.

Přiřazení

- Python: přesměrování odkazu na objekt.
- C: změna hodnoty uložené v objektu pevně svázaném s proměnnou.

Ilustrace přiřazení

```
int a, b;
a = 1;
```

a	1
b	

```
a = 2;
```

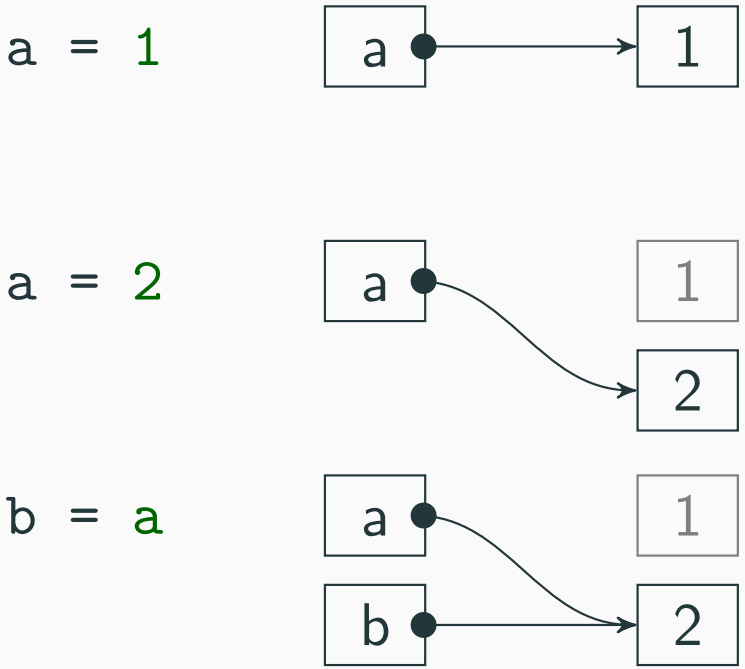
a	2
b	

```
b = a;
```

a	2
b	2

Jazyk C

Přiřazení mění hodnotu



Jazyk Python

Přiřazení přesměruje odkaz

Identita objektu

- Vestavěná funkce `id`:
  - vrátí číselnou hodnotu reprezentující identitu objektu,
  - v implementaci CPython adresa v paměti.
  - (Více ji používat nebudeme, zde jen pro ilustraci.)
- Výraz `a is b` znamená `id(a) == id(b)`.

```
a = 1000
b = a
b += 1  # same as: b = b + 1
assert id(a) != id(b)
s = [1]
t = s  # try changing to: t = s.copy()
s.append(2)
assert id(s) == id(t)
```

Volání podprogramů – předávání parametrů

- Hodnotou – call by value:
  - Formální parametr pevně svázán s novým objektem, do kterého se uloží hodnota skutečného parametru.
  - *Změna hodnoty formálního parametru se neprojeví navenek.*
    - Ani při vnitřním přiřazení, pokud je takové k dispozici.
  - Standardní v Pascalu, C, C++ apod.

- Odkazem – call by reference:
  - Formální parametr se (obvykle pevně) sváže s existujícím objektem pevně svázaným s proměnnou použitou jako skutečný parametr.
  - Předá se tedy vlastně odkaz na proměnnou.
  - *Změna hodnoty formálního parametru je současně změnou hodnoty skutečného parametru.*
  - Lze také svázat s prvkem existující datové struktury.
  - Typicky ale nelze zavolat s libovolným výrazem, který by vytvořil nový (bezejmenný) objekt.
  - Lze použít např. v C++ (`&`), C# (`ref`), Pascalu (`var`) apod.

Volání podprogramů – předávání parametrů

- Python – call by object sharing:
  - Formální parametry se sváží s objekty, na které se vyhodnotí výrazy použité jako skutečné parametry při volání funkce.
  - Může (a nemusí) vzniknout nový objekt.
  - Tento objekt nemusí mít vazbu z nějaké proměnné.
  - *Přiřazení do formálního parametru mění jeho vazbu, nemění hodnotu skutečného parametru.*
    - Ale pozor: vnitřní přiřazení hodnotu skutečného parametru změnit může!
  - Podobně funguje např. Java.
- Jiné možnosti:
  - jménem, hodnotou-výsledkem, ...

Volání podprogramů v Pythonu – příklad

call_by_sharing.py

```
def fun(s):
    s.append(3)
    s = [42, 17]
    s.append(9)
    return s
```

```
t = [1, 2]
u = fun(t)
print(t, u)

v = fun(t + u)
print(t, u, v)
```

Odbočka: Složené přiřazení

Operátory složeného přiřazení (+= apod.)

- V tomto předmětu používáme jen pro čísla (a později řetězce).
- V těchto případech jsou skutečně ekvivalentní přiřazení.
  - `x += 1` je totéž, jako `x = x + 1`.

- Chování pro jiné typy je v Pythonu matoucí:
  - zejména už není ekvivalentní přiřazení.
- Proto preferujeme metody
  `.extend` (seznamy), `.update` (množiny) apod.

Složené přiřazení v Pythonu

plus_equals1.py

```
def increment(x):
    print(x, id(x))
    x += 1
    print(x, id(x))

p = 2020
increment(p)
print(p, id(p))
```

```
def add_to_list(s):
    print(s, id(s))
    s += [1]
    print(s, id(s))

t = [1, 2, 3]
add_to_list(t)
print(t, id(t))
```

- Čeho si můžeme všimnout?

Složené přiřazení v Pythonu

plus_equals2.py

Rozdíl mezi = a += u seznamů!

```
def add_to_list1(s):
    print(s, id(s))
    s += [1]
    print(s, id(s))

t = [1, 2, 3]
add_to_list1(t)
print(t)
```

- V t je [1, 2, 3, 1].

```
def add_to_list2(s):
    print(s, id(s))
    s = s + [1]
    print(s, id(s))

t = [1, 2, 3]
add_to_list2(t)
print(t)
```

- Co je v t?

Proto += apod. s měnitelnými typy v IB111 nepoužíváme!

Správa paměti

Různé přístupy ke správě paměti

- Manuální – vestavěné podprogramy, které je nutno explicitně volat pro přidělení/uvolnění objektů.
- Automatická s uvolněním paměti na konci života proměnné.
  - Typické u pevné vazby proměnné na objekt.
- Automatická s počítáním referencí.
  - Kolik vazeb na daný objekt ještě existuje.
  - Pokud žádná, objekt je uvolněn (nezvládne cyklické vazby).
- Automatická – garbage collection.
  - Jednou za čas se uklidí nepoužívané objekty.

Správa paměti v Pythonu

- Automatická – počítání referencí + někdy i větší úklid.
- Počet referencí `sys.getrefcount(object)`.
  - Opět pouze pro ilustraci, jinak nepoužíváme.

Správa paměti v Pythonu

Zvýšení počtu odkazů

- vytvoření hodnoty — `a = "Hello!"`
- vytvoření aliasu — `b = a`
- předání funkci — `fun(a)`
- vložení do datové struktury — `s = ["Hi!", a]`

Snížení počtu odkazů

- ukončení platnosti lokální proměnné — *konec funkce*
- přiřazení jiné hodnoty aliasu — `b = "Aloha!"`
- odstranění z datové struktury — `s.pop()`
- konec existence datové struktury

Kopírování objektů

Vytvoření aliasů `b = a`

- `a` a `b` se stávají aliasy: odkazují na tentýž objekt.

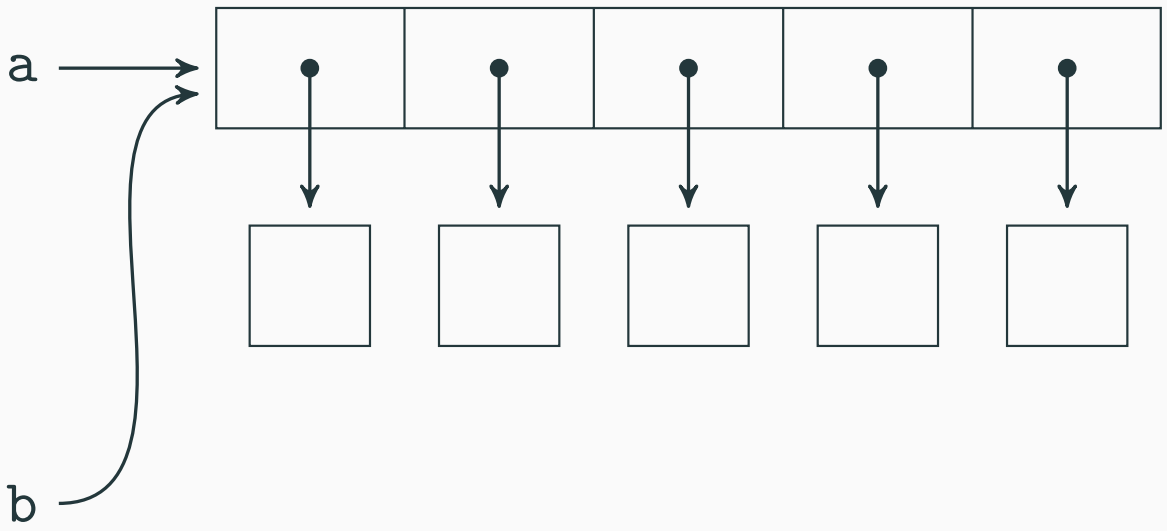
Mělká kopie seznamu `b = a.copy()`

- Vytváříme nový seznam, ale prvky nového a původního seznamu aliasují.
  - OK, pokud jsou prvky neměnných typů (čísla, řetězce, ...).
  - Může a nemusí být žádoucí pro prvky typu seznam apod.
- Podobně pro množiny, slovníky, ...

Hluboká kopie

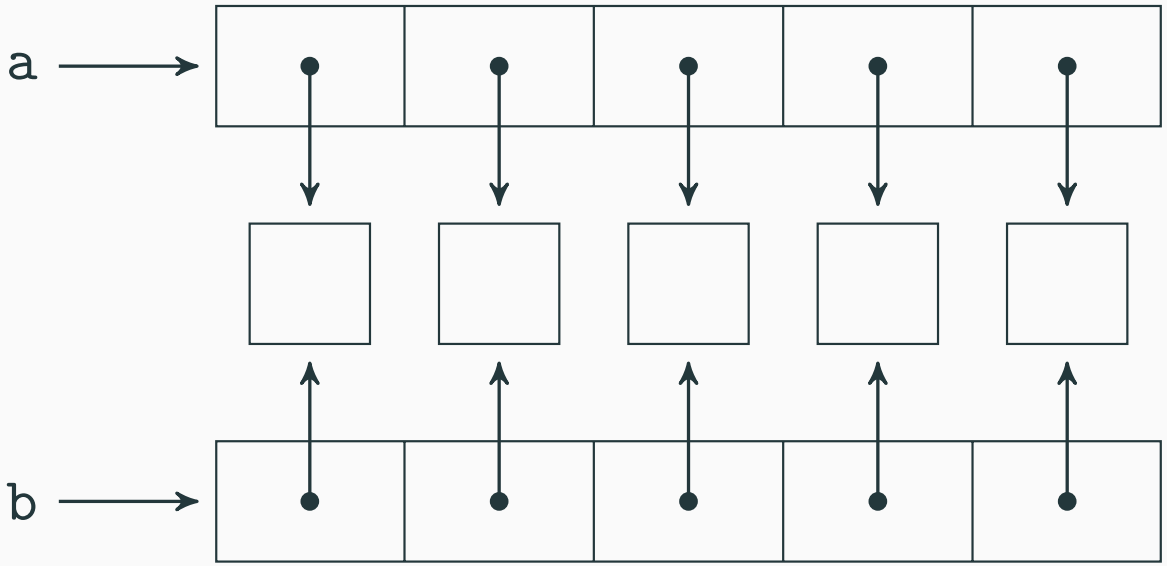
- Kompletní kopie všech dat – jak?

Kopírování objektů




```
b = a
```

Kopírování objektů




```
b = a.copy()
```

Hluboká kopie

deepcopy.py

Seznamy seznamů

```
def list2d_copy(s: list[list[int]]) \
        -> list[list[int]]:
    return [row.copy() for row in s]
```

Slovníky se seznamy jako hodnotami

```
def dict_list_copy(d: dict[str, list[int]]) \
        -> dict[str, list[int]]:
    clone = {}
    for key, value in d.items():
        clone[key] = value.copy()
    return clone
```

K zamyšlení

Datový typ **fronta**

- Prvky v pořadí FIFO (*First In First Out*).
- Operace:
  - vložení,
  - odstranění,
  - test prázdnosti.

- Zkuste si rozmyslet, jak byste implementovali.
  - S prostředky, které zatím máme k dispozici,
  - přemýšlejte o složitosti operací.