Seznamy, ntice

IB111 Základy programování

Tomáš Vojnar (na základě slajdů Nikoly Beneše)

3. října 2025

Největší společný dělitel

(GCD, greatest common divisor)

- *Vstup:* přirozená[1] čísla a, b; alespoň jedno z nich není 0.
- *Výstup:* největší přirozené číslo takové, že dělí a i b.
- Příklady:
  - 504, 540 $\rightarrow$ 36,
  - 17, 0 $\rightarrow$ 17,
  - 10362063572285335032, 85856486393012983 05 $\rightarrow$ 159.

[1] Nezáporná celá.

GCD: Naivní algoritmus

gcd_naive.py

- Pokud je a nebo b rovno 0, vrátíme větší z nich.
- Projdeme všechna čísla od 1 do menšího z a, b.
- Pro každé vyzkoušíme, zda dělí a i b.
  - Jak?
- Pamatujeme si největšího ze společných dělitelů.

```
def gcd_naive(a, b):
    if a == 0 or b == 0:
        return max(a, b)
    best = 0
    for i in range(1, min(a, b) + 1):
        if a % i == 0 and b % i == 0:
            best = i
    return best
```

- Je tento program korektní? Dal by se tento program zefektivnit?

GCD: „Školní“ algoritmus

- Rozložit a i b na součin prvočísel.
- Vybrat, co je společné, vynásobit.

- Příklad:
  - $504 = 2^3 \cdot 3^2 \cdot 7$,
  - $540 = 2^2 \cdot 3^3 \cdot 5$,
  - $\gcd(504, 540) = 2^2 \cdot 3^2 = 36$.

- K zamyšlení: jak toto realizovat programem v Pythonu?
  (Potřebné nástroje máte k dispozici už z minulé kapitoly, zejména není potřeba používat seznamy.)

GCD: Eukleidův algoritmus

gcd_euclid01.py

Základní myšlenka: pokud $a > b$, pak $\gcd(a, b) = \gcd(a - b, b)$.

```
def gcd(a, b):
    if a == 0:
        return b
    while b != 0:
        if a > b:
            a -= b
        else:
            b -= a
    return a
```

Eukleidés z Alexandrie
zdroj: Encyclopædia Britannica

- Proč nám v cyklu stačí podmínka `b != 0`? Proč netestujeme i a?
- V nejhorším případě může být pomalejší než naivní algoritmus; kdy? Co když jedno z čísel bude 1?

GCD: Vylepšený Eukleidův algoritmus

gcd_euclid02.py

Lepší myšlenka: pokud $a > b$, pak $\gcd(a, b) = \gcd(a \bmod b, b)$.
(Místo odčítání použijeme rovnou zbytek po dělení).

```
def gcd(a, b):
    while b != 0:
        aux = a % b
        a = b
        b = aux
    return a
```

- Proč i zde stačí testovat jen `b != 0`?

Poznámka: S využitím ntic (*tuples*, o těch budeme mluvit za chvíli) by se tělo cyklu dalo napsat takto: `a, b = b, a % b`.

GCD: Rekurzivní Eukleidův algoritmus

gcd_euclid03.py

(Pro zajímavost, o rekurzi bude řeč později.)

```
def gcd(a, b):
    return a if b == 0 else gcd(b, a % b)
```

Pro zvídavé: srovnejte se stejnou funkcí v Haskellu:

```
gcd a b = if b == 0 then a else gcd b (a `mod` b)
```

Efektivita algoritmů

- Časová náročnost algoritmu (v nejhorším případě):
  - naivní, „školní“: exponenciální vůči počtu cifer,
  - Eukleidův (s odčítáním): exponenciální vůči počtu cifer,
  - Eukleidův (s modulem): lineární[2] vůči počtu cifer.
- Různé algoritmy mohou řešit tentýž problém různě rychle.
  - Často rozdíl použitelné vs. nepoužitelné.
- Více později (a v dalších předmětech).

[2] Zanedbáváme-li složitost výpočtu % – ve skutečnosti je to trochu složitější.

Seznamy

Motivace pro seznamy

- Chceme zpracovávat větší množství položek.
- Nechceme psát opakovaně stejný kód (DRY).
- Nemusíme předem znát počet položek.

Příklady

- Seznam studentů seřazených nějakým způsobem.
- Úlohy čekající na zpracování.
- Cesta grafem.
- Reprezentace herního plánu (piškvorky, šachy).
  - Zanořené seznamy.
- Podvýrazy propojené určitým operátorem.
- ...

Seznamy

- *Sekvence libovolného počtu položek*.
- Podobné typy běžně dostupné v jiných jazycích:
  - pole (*array*) – pevná délka, všechny položky stejného typu,
  - dynamická pole, různé jiné druhy seznamů, ...
- Seznamy v Pythonu – obecnější než pole:
  - umí měnit velikost,
  - smí obsahovat položky různých typů – *každá položka odkazuje na objekt (má vazbu na objekt)*, jako proměnné.
  - (Většinou se ovšem omezíme na seznamy položek stejného typu, s případnou výjimkou `None`).
- Pole v Pythonu: vestavěné pole `array` a NumPy pole; nad rámec předmětu.

Vytvoření seznamu

lists.py

- Výčtem prvků:

```
s = []
s = [3, 1, 4, 1, 5]
s = ["ABC", 3.14, -7]
s = [[1, 2], [3, 4]]
s = ["pes", "kočka", 0.01, ["velbloud", -13], []]
```

- Tzv. list comprehension (intenzionální zápis seznamu):

```
s = [2 * x for x in range(10)]
s = [x ** 2 for x in range(1, 10) if x % 2 == 0]
s = [3 * x for x in [5, 17, 23, 40]]
s = [[a, b, c] for a in range(1, 10)
     for b in range(1, 10)
     for c in range(1, 10)
     if a ** 2 + b ** 2 == c ** 2]
```

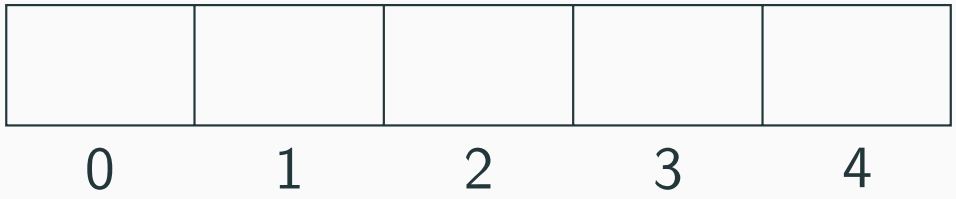
Seznamy – základní operace

- Zjištění délky seznamu: `len(s)`.
- Přidání prvku na konec seznamu: `s.append(x)`.
- Odebrání prvku z konce seznamu: `s.pop()`.

- Indexování (výběr konkrétního prvku) `s[0]`, `s[1]`, …
  - pro čtení i zápis,
  - konkrétnímu prvku můžeme něco přiřadit, např. `s[1] = 42` (tzv. *vnitřní přiřazení*),
- Indexování od konce: `s[-1]`, `s[-2]`, …
  - V jiných jazycích nepříliš časté.

- Kopie seznamu `s.copy()`.
  - Totéž jako `[x for x in s]`.
  - K čemu je to dobré?

Indexování seznamů



Indexování od nuly

- První prvek seznamu je `s[0]`.
- Částečně historicko-technické důvody,
- ale i dobré „matematické“ důvody,
- souvisí s oblibou polouzavřených intervalů $\langle od, do)$.

Pro zajímavost:

- https://en.wikipedia.org/wiki/Zero-based_numbering
- https://www.cs.utexas.edu/users/EWD/transcriptions/EWD08xx/EWD831.html
- https://softwareengineering.stackexchange.com/questions/110804/why-are-zero-based-arrays-the-norm

Procházení seznamu

- Máme seznam `my_list` a chceme postupně projít všechny jeho prvky – jak na to?
- Ne úplně vhodné řešení:

```
for i in range(len(my_list)):
    do_something(my_list[i])
```

- Lepší (čitelnější) řešení:

```
for element in my_list:
    do_something(element)
```

- Podobnou notaci má většina moderních jazyků.
- `range(...)` je *něco jako seznam*,
  - (kvůli efektivitě to není seznam, ale speciální objekt),
  - seznam můžeme vyrobit konverzí `list(range(...))`.

Procházení seznamu

- Co když potřebujeme jak prvky seznamu, tak jejich indexy?

```
for i, element in enumerate(my_list):
    do_something(i, element)
```

- `i` bude postupně nabývat hodnot 0, 1, …,
- `element` bude prvek `my_list[i]`.

- (`enumerate(...)` je další speciální objekt, jehož procházením dostáváme dvojice, viz dále.)

Procházení seznamu

Doporučení

- Nemodifikujte datovou strukturu, kterou zrovna procházíte v cyklu `for`.

```
for elem in my_list:
    if elem > 2:
        my_list.pop()   # BAD
```

- Matoucí, špatně čitelné.
- Nemusí (rozumně/vůbec) fungovat.
  - Se seznamy v Pythonu to *zřídka* nějaký smysl dává...
  - ... ale s jinými datovými strukturami nebo v jiných jazycích ne.

Alternativa

- `while` cyklus s vhodnou podmínkou.
- Procházení *kopie*: `for elem in my_list.copy():`.

Odbočka – Proměnné

Proměnná v Pythonu – připomenutí z 1. přednášky

- Má jméno, existuje v nějakém kontextu, může mít typ a může mít vazbu (odkaz) na objekt („místo v paměti“).

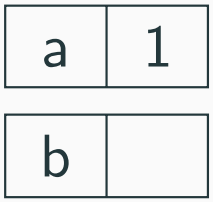
Přiřazení v různých jazycích

- Ve stylu C (Pascal apod.):
  - *změna hodnoty* uložené v objektu
    (vazba proměnné k objektu je pevná).
- Ve stylu Pythonu:
  - *přesměrování odkazu* (vazby) na jiný objekt
    (vazba proměnné k objektu se může měnit).

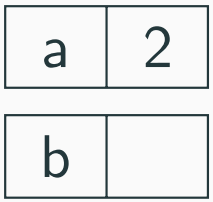
Poznámka: U neměnných (*immutable*) typů (čísla, řetězce) nepozorujeme v Pythonu žádný rozdíl (až na identitu).

Ilustrace přiřazení

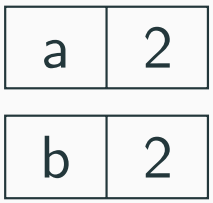
```
int a, b;
a = 1;
```




```
a = 2;
```




```
b = a;
```

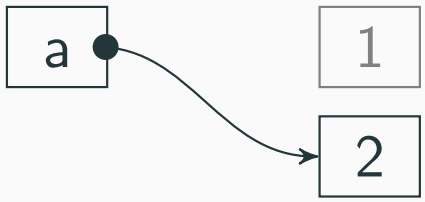


Jazyk C

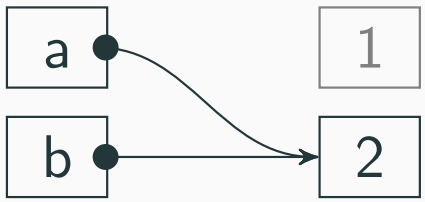
Přiřazení mění hodnotu

```
a = 1
```


```
a = 2
```




```
b = a
```



Jazyk Python

Přiřazení přesměruje odkaz

Seznamy a proměnné v Pythonu

list_refs.py

```
s = [1, 2, 3]
t = s
s.append(4)
```

Jakou hodnotu má proměnná `t`? `[1, 2, 3, 4]`

- Proč?
- Protože přiřazení proměnné v Pythonu mění vazbu (odkaz).
  - Na jeden objekt se může odkazovat více proměnných.
- http://www.pythontutor.com

Už víme, k čemu je dobré `s.copy()`?

- Pokud potřebujeme vytvořit novou *kopii* seznamu nezávislou na původním seznamu.
- Pozor: jedná se o tzv. *mělkou* (shallow) kopii – jen kopie hlavní úrovně seznamu.
- U seznamy seznamů apod. nutno případně *provést i kopie zanořených seznamů*!.

Seznamy a volání funkcí v Pythonu

list_fun_1.py

```
def fun_1(x):
    x = 17

y = 10
fun_1(y)
```

Jakou hodnotu má proměnná `y`? `10`

- `fun_1` nemění hodnotu skutečného parametru.
  (Ani nemůže, čísla jsou v Pythonu neměnná.[3])
- Změní vazbu formálního parametru `x`.

[3] Tj., Python nemá žádnou konstrukci pro změnu hodnoty objektu typu `int`, můžeme pouze vytvářet nové objekty tohoto typu (platí i pro `float`, `bool`, `str`.)

Seznamy a volání funkcí v Pythonu

list_fun_2.py

```
def fun_2(s):
    s.append(17)
```

všimněte si:
funkce `fun_2` není čistá

```
t = [1, 2]
fun_2(t)
```

Jakou hodnotu má proměnná `t`? `[1, 2, 17]`

- `fun_2` mění hodnotu skutečného parametru.
  - Vazba formálního parametru na skutečný funguje stejně jako přiřazení, `s` a `t` pak odkazují na stejný objekt.
- Vyzkoušejte změnit tělo funkce na `s = [17]` nebo `s[0] = 17` a pozorujte efekt.

Seznamy a volání funkcí v Pythonu

list_fun_3.py

```
def fun_3(s):
    s.append(17)
    s = [4, 5]
    s.append(19)

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

Jakou hodnotu má proměnná `t`? `[1, 2, 17]`

- Proč?
- Přiřazení je změna vazby (odkazu).
- Po provedení `s = [4, 5]` už `s` *nemá vazbu na původní seznam*.
- http://www.pythontutor.com

(Vrátíme se k tomu v některé z dalších přednášek.)

Seznamy a volání funkcí v Pythonu

list_fun_4.py

```
def fun_4(t):
    t.append(0)


def fun_5(a, b, c):
    s = [a, b, c]
    fun_4(s)


fun_5(1, 2, 3)
```

Jsou uvedené funkce čisté?

- `fun_4` ne, `fun_5` ano.

Vsuvka: další příklad na čistotu

pure_print.py

```
def fun_6(n, do_print):
    if do_print:
        print(n)
    return n+1


def fun_7(n):
    return fun_6(n, False)


fun_7(0)
```

Jsou uvedené funkce čisté?

- `fun_6` ne, `fun_7` ano.

Seznamy – odvozené operace

`list_ops.py`

- Součet prvků seznamu: `sum(s)`.
  - Součet prázdného seznamu je `0`.
- Maximum/minimum neprázdného seznamu: `max(s)`, `min(s)`.
- (Další přibudou později.)

- Tyto funkce umíme naprogramovat pomocí základních operací.
- Příklad – součet:

```
def my_sum(data):
    total = 0
    for element in data:
        total += element
    return total
```

Ntice

Ntice (*Tuples*)

tuples.py

- Kolekce pevné velikosti, s pevnou hlavní typovou úrovní prvků.
- Hodnoty prvků se dají měnit, pokud to jejich typ připouští.
  - Je-li prvkem např. seznam, vždy jím bude seznam, ale jeho prvky i jejich typy se mohou měnit.
- V tomto předmětu nebudeme ntice indexovat.
- Zápis: *kulaté závorky místo hranatých*,
  - v jistých situacích se kulaté závorky smí vynechat.

```
s = (1, "A", 3)
```

- Typická použití – jednoduchá strukturovaná data:
  - souřadnice `(x, y)`,
  - barva pixelu `(red, green, blue)`,
  - vracení více hodnot z funkce, ...
- Ntice velikosti 1: `(x,)` – Pythonovská specialita.

Použití ntic

tuples.py

- Rozbalení ntice

```
data = ("Rick Sanchez", "C-137", "Earth")
name, dimension, planet = data

children = [
    (1, "Henry"),
    (8, "Kali"),
    (11, "Jane"),
]

for number, name in children:
    pass  # do something with number, name

a, b = 10, 17
```

Použití ntic

tuples.py

- Prohození hodnot proměnných (*swap*)

```
a, b = b, a  # the same as (a, b) = (b, a)
```

- Vracení více hodnot z funkce

```
def minmax(a, b):
    return min(a, b), max(a, b)

x, y = minmax(9.7, 3.14)

quot, rem = divmod(17, 5)  # standard Python function
```

Porovnávání seznamů a ntic

Lexikografické uspořádání („jako ve slovníku“)

- Na *dvojicích*:
  `(a, b) < (c, d)`
  je totéž, co
  `a < c or (a == c and b < d)`.
- Na *nticích stejné délky*:
  podobně; výsledek je podle první dvojice, která se nerovná.
- Na *nticích různé délky*/*seznamech*:
  podobně; je-li `s` vlastním prefixem (začátkem) `t`, pak `s < t`.

Příklad: Řešení hádanky z 1. přednášky

farm3.py

```
def solve_hens_pigs_puzzle(heads, legs):
    for hens in range(heads + 1):
        pigs = heads - hens
        if 2 * hens + 4 * pigs == legs:
            return (hens, pigs)

    return None


# what is the value of solve_hens_pigs_puzzle(20, 56)?
# what is the value of solve_hens_pigs_puzzle(40, 100)?
```

Příklad: Dělitelé čísla

divisors.py

```
def divisors(num):
    result = []
    for divisor in range(1, num + 1):
        if num % divisor == 0:
            result.append(divisor)
    return result
```

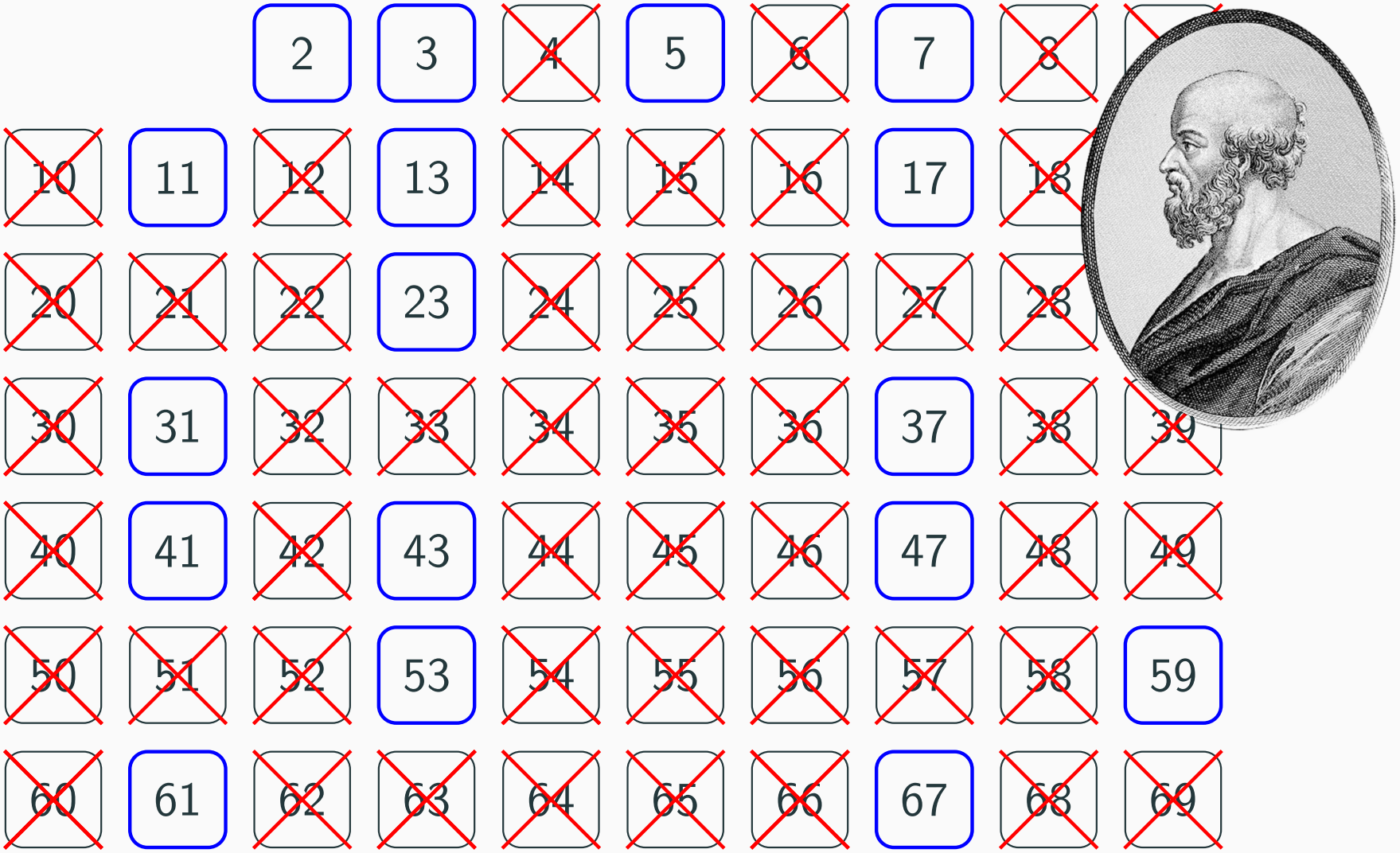
- Dalo by se nějak vylepšit?
  - Počítat jen do `num // 2` (a přidat `num` na konec).
  - Využít toho, že `divisor` a `num // divisor` jsou oba děliteli, menší z těchto dvou dělitelů je $\leq$ odmocnině z `num`.

- Pro zajímavost (použití intenzionálních seznamů):

```
def divisors_alt(num):
    return [divisor for divisor in range(1, num + 1)
            if num % divisor == 0]
```

Příklad: Prvočísla – Eratosthenovo síto

primes.py



Příklad: Prvočísla – Eratosthenovo síto

primes.py

```
def sieve(limit):
    result = []
    is_prime = [True for _ in range(limit)]

    for p in range(2, limit):
        if is_prime[p]:
            result.append(p)
            for mult in range(p * p, limit, p):
                is_prime[mult] = False

    return result
```

Vnořené seznamy

matrix.py

- Prvky seznamů mohou být opět seznamy.
- Použití: vícerozměrná data (např. matice).

```
mat = [[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]]
```

- Hodnotou výrazu `mat[1]` je seznam `[4, 5, 6]`.
- Hodnotou výrazu `mat[1][2]` je tedy číslo `6`.

Příklad: Nulová matice zadaných rozměrů

matrix.py

```
def null_matrix_bad(rows, cols):
    row = [0 for _ in range(cols)]
    matrix = []
    for _ in range(rows):
        matrix.append(row)

    return matrix  # why is this bad?
```

- Proč je toto řešení špatné? (použijte http://pythontutor.com)

Příklad: Nulová matice zadaných rozměrů

matrix.py

```
def null_matrix_good(rows, cols):
    row = [0 for _ in range(cols)]
    matrix = []
    for _ in range(rows):
        matrix.append(row.copy())

    return matrix
```

- Výraz uvnitř intenzionálního seznamu může být zase intenzionální seznam:

```
def null_matrix(rows, cols):
    return [[0 for _ in range(cols)]
            for _ in range(rows)]
```

Příklad: Matice souřadnic

matrix.py

- Chceme matici, jejímiž prvky budou *dvojice* souřadnic
  - ve tvaru *(x, y)*, kde *x* je sloupec, *y* je řádek.

```
def coord_matrix(rows, cols):
    return [[(x, y) for x in range(cols)]
            for y in range(rows)]

cm = coord_matrix(3, 4)
```

- Co bude hodnotou výrazu `cm[1][2]`?

Odbočka: Násobení matic

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \cdot \begin{pmatrix} 0 & 1 & 2 & 3 \\ 9 & 0 & 1 & 4 \\ 8 & 7 & 6 & 5 \end{pmatrix} = \begin{pmatrix} 42 & 22 & 22 & 26 \\ 93 & 46 & 49 & 62 \end{pmatrix}$$

- Počet sloupců levé matice = počet řádků pravé matice.
- Výsledná matice má řádky jako levá, sloupce jako pravá.
- Pro každý řádek levé a sloupec pravé matice
  - spočítáme jejich skalární součin
    (součin po složkách, součet výsledků).

- $1 \cdot 0 + 2 \cdot 9 + 3 \cdot 8 = 0 + 18 + 24 = 42$
- $4 \cdot 0 + 5 \cdot 9 + 6 \cdot 8 = 0 + 45 + 48 = 93$
- $1 \cdot 1 + 2 \cdot 0 + 3 \cdot 7 = 1 + 0 + 21 = 22$
- atd.

Příklad: Násobení matic

matrix.py

```
def matrix_mult(left, right):
    rows = len(left)
    cols = len(right[0])
    common = len(right)
    result = null_matrix(rows, cols)
    for i in range(rows):
        for j in range(cols):
            for k in range(common):
                result[i][j] += left[i][k] * right[k][j]
    return result
```

K zamyšlení: Jak bychom zjistili, zda je vstup platný?

- Jsou na vstupu skutečně matice?
- Jsou matice kompatibilní?