

B4B33RPH: Řešení problémů a hry

Python – základní kameny až skály III

Množiny, list comprehensions, profilování, generátory

Tomáš Svoboda, Petr Pošík, Petr Štibinger

stibipet@fel.cvut.cz

24. října 2023



Katedra kybernetiky
Fakulta elektrotechnická
České vysoké učení technické v Praze

Množina – set

```
>>> a = {1,2,3,3}  
>>> print(a)
```

Jaký bude výpis?

- (a) {1,2,3,3}
- (b) {1,2,3}
- (c) {3,3}

Množina – set

```
>>> a = {1,2,3,3}
>>> print(a)
```

Vyzkoušejte sami:

- >>> a = **set**([1,2,3,3])
- >>> b = {}
- >>> **help**(a)
- >>> **help**(b)

Jaký bude výpis?

- (a) {1,2,3,3}
- (b) {1,2,3}
- (c) {3,3}

Množinové operace, a.union(b)

```
>>> a = {1,2,3,3}
>>> b = {2,3,4}
>>> c = a | b
>>> print(c)
```

Jaký bude výpis?

- (a) {1,2,3,4}
- (b) {1,2,2,3,3,3,4}
- (c) {3}

Množinové operace, a.intersection(b)

```
>>> a = {1,2,3,3}
>>> b = {2,3,4}
>>> c = a & b
>>> print(c)
```

Jaký bude výpis?

- (a) {3}
- (b) {2,3}
- (c) {1,2,2,3,3,2,3,4}

Množina – set

- Vestavěné datové kontejnery – **list**, **tuple**, **dict**, **set**
- Prvky **setu** jsou:
 - Unikátní – každý zastoupen max. jednou
 - Neuspořádané – pořadí není garantováno
 - Neindexované – ale lze využít operátor **in**
 - Neměnné, hashovatelné (viz unikátnost)

Množina – set

- Vestavěné datové kontejnery – **list**, **tuple**, **dict**, **set**
- Prvky **setu** jsou:
 - Unikátní – každý zastoupen max. jednou
 - Neuspořádané – pořadí není garantováno
 - Neindexované – ale lze využít operátor **in**
 - Neměnné, hashovatelné (viz unikátnost)
- Set jako takový ale měnit můžeme
 - **set.add()**
 - **set.remove()**

Kontrolní otázka

```
>>> s1 = {1, 2, 'Hello', 4}
>>> s2 = {(1, 2), [3, 4]}
>>> s3 = {(1, 2), (3,), 4}
>>> s4 = {{1, 2, 3}, 4}
```

Kolik řádků je s chybou?

- (a) žádný
- (b) 1
- (c) 2
- (d) 3

Neměnný set – frozenset

- Speciální případ setu, nelze měnit
- Vlastnosti jako `set`
 - ale neumí `add()`, `remove()`, ...
- Z vestavěných Python objektů má nejbliž ke skutečné konstantě

Příklad použití – kontrola pravidel

```
1 LEGAL_MOVES = frozenset( [ (0,1), (1,0), (1,1) ] )
2
3 p1 = MyPlayer()
4 m = p1.move()
5
6 if m not in LEGAL_MOVES:
7     print('Player just attempted an illegal move:', m)
```

List comprehensions

- Kompaktní vytvoření seznamu bez explicitního bloku smyčky
- Někdy se nazývá *generátorová notace*, protože syntax je podobný
 - ...ale **generátor** je něco trochu jiného!

List comprehensions

list_comp_instant_generation.py – rychlé vytvoření seznamu

```
1 # sampling parabola x^2
2 a = [x**2 for x in range(-10,10)]
3 print(a)
4
5 # sampling square root function sqrt(x)
6 # also with condition in the same command
7 b = [x**0.5 for x in range(-10, 10) if x > 0]
8 print(b)
```

List comprehensions

list_comp_instant_generation.py – výběr z existujících dat

```
1 fruit = ['apple', 'banana', 'lemon', 'plum', 'watermelon']
2
3 # select items containing letter 'a'
4 a = [f for f in fruit if 'a' in f]
5 print(a)
```

Narozeninový problém

- Skupina N osob
- Jaká je pravděpodobnost, že alespoň dva lidé mají narozeniny ve stejný den?
- Pro jak velké N začne být **pravděpodobnější**, že alespoň jeden narozeninový den **není unikátní**?



Narozeninový problém

- Skupina N osob
- Jaká je pravděpodobnost, že alespoň dva lidé mají narozeniny ve stejný den?
- Pro jak velké N začne být **pravděpodobnější**, že alespoň jeden narozeninový den **není unikátní**?

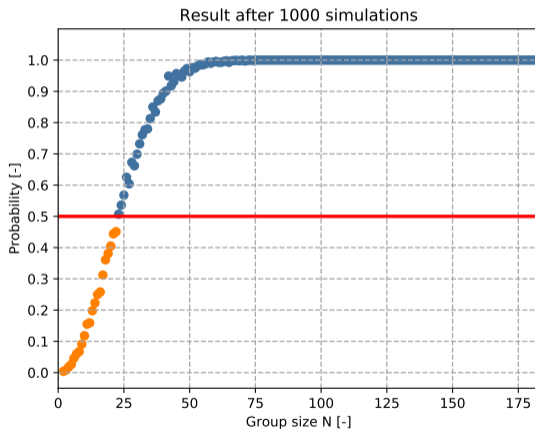


Zkusme programovat „líně“ a využít, co už známe ...

Narozeninový problém

```
1 NUM_TRIALS = 1000
2 UPPER_LIMIT = 183
3
4 prob_matching_dates = {}
5
6 for group_size in range(2, UPPER_LIMIT):
7     prob_matching_dates[group_size] = 0.0
8
9     for trial in range(NUM_TRIALS):
10         birthday_dates = [random.randint(1,365) for i in range(group_size)]
11
12         if len(birthday_dates) != len(set(birthday_dates)):
13             prob_matching_dates[group_size] += 1.0 / NUM_TRIALS
```

Narozeninový problém



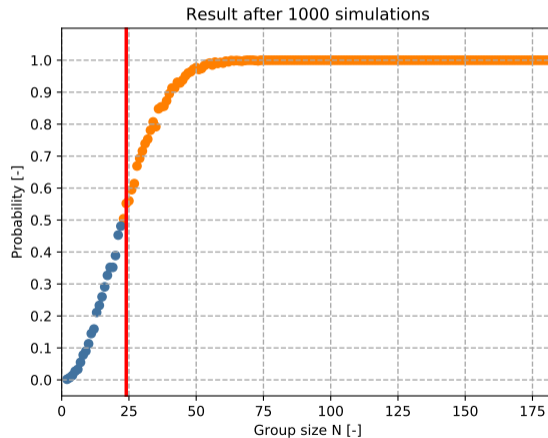
birthday_problem.py - celý kód včetně grafiky

Líné řešení...

- Rychle naprogramované
- Dobře čitelné
- V podstatě přímo zadání, přepsané do Pythonu

Líné řešení je líné

- Počítá se dlouho
- Zbytečné výpočty?
- Asi 80 procent!
- Stačí najít první úspěch
- Efektivní algoritmus



Jak měřit efektivitu

- Hledání slabých míst
- Snaha o optimalizaci
- Měření doby běhu procesu
- Vlastní časování kousků kódu
- Specializované nástroje na **profilování**

Příklad z předchozí přednášky

```
1 import random
2
3 class Memory:
4
5     def __init__(self, size):
6         self.size = size
7         self.data = []
8
9     def add(self, value):
10        self.data.append(value)
11        if len(self.data) > self.size:
12            del self.data[0]
13
14    def get_most_frequent(self):
15        return max(self.data, key=self.data.count)
```

Příklad z předchozí přednášky

```
1 import random
2
3 class Memory:
4
5     def __init__(self, size):
6         self.size = size
7         self.data = []
8
9     def add(self, value):
10        self.data.append(value)
11        if len(self.data) > self.size:
12            del self.data[0]
13
14        def get_most_frequent(self):
15            return max(self.data, key=self.data.count)
```

Příklad z předchozí přednášky

```
1 import random
2
3 class Memory:
4
5     def __init__(self, size):
6         self.size = size
7         self.data = []
8
9     def add(self, value):
10        self.data.append(value)
11        if len(self.data) > self.size:
12            del self.data[0]
13
14        def get_most_frequent(self):
15            return max(self.data, key=self.data.count)
```

Rychle jen vypadá

Líné řešení, neefektivní algoritmus

frequency_example.py – hledání nejčastějšího prvku v poli

```
1 def get_most_frequent_element(data):
2     return max(data, key=data.count)
3
4 if __name__ == '__main__':
5     d1 = ['R', 'P', 'P']
6     print(get_most_frequent_element(d1))
7
8     d2 = 8000 * ['R', 'P', 'P']
9     print(get_most_frequent_element(d2))
```

Měření času – modul `time`

frequency_example_time.py

```
1 import time
2
3 def get_most_frequent_element(data):
4     return max(set(data), key=data.count)
5
6 if __name__ == '__main__':
7     d = 8000 * ['R', 'P', 'P']
8
9     t_start = time.perf_counter()
10    print(get_most_frequent_element(d))
11    t_stop = time.perf_counter()
12
13    print('Elapsed time:', t_stop - t_start, 'seconds')
```


Měření času – modul `time`

frequency_example_time.py

Proč je to tak pomalé?

```
1 import time
2
3 def get_most_frequent_element(data):
4     return max(set(data), key=data.count)
5
6 if __name__ == '__main__':
7     d = 8000 * ['R', 'P', 'P']
8
9     t_start = time.perf_counter()
10    print(get_most_frequent_element(d))
11    t_stop = time.perf_counter()
12
13    print('Elapsed time:', t_stop - t_start, 'seconds')
```

Měření času – modul `time`

frequency_example_time.py

```
1 import time
2
3 def get_most_frequent_element(data):
4     return max(set(data), key=data.count)
5
6 if __name__ == '__main__':
7     d = 8000 * ['R', 'P', 'P']
8
9     t_start = time.perf_counter()
10    print(get_most_frequent_element(d))
11    t_stop = time.perf_counter()
12
13    print('Elapsed time:', t_stop - t_start, 'seconds')
```

Proč je to tak pomalé?

Děláme zbytečné výpočty!

Měření času – modul `time`

frequency_example_time.py

```
1 import time
2
3 def get_most_frequent_element(data):
4     return max(set(data), key=data.count)
5
6 if __name__ == '__main__':
7     d = 8000 * ['R', 'P', 'P']
8
9     t_start = time.perf_counter()
10    print(get_most_frequent_element(d))
11    t_stop = time.perf_counter()
12
13    print('Elapsed time:', t_stop - t_start, 'seconds')
```

Proč je to tak pomalé?

Děláme zbytečné výpočty!

Pro každý prvek v `data`,
projdí celý seznam `data`
a spočítá výskyt tohoto prvku

Měření času – modul `time`

`frequency_example_time_upgrade.py` – využijeme `set`

```
1 import time
2
3 def get_most_frequent_element(data):
4     return max(set(data), key=data.count)
5
6 if __name__ == '__main__':
7     d = 8000 * ['R', 'P', 'P']
8
9     t_start = time.perf_counter()
10    print(get_most_frequent_element(d))
11    t_stop = time.perf_counter()
12
13    print('Elapsed time:', t_stop - t_start, 'seconds')
```

Měření času – modul `time`

frequency_example_time_upgrade.py – využijeme `set`

```
1 import time
2
3 def get_most_frequent_element(data):
4     return max(set(data), key=data.count)
5
6 if __name__ == '__main__':
7     d = 8000 * ['R', 'P', 'P']
8
9     t_start = time.perf_counter()
10    print(get_most_frequent_element(d))
11    t_stop = time.perf_counter()
12
13    print('Elapsed time:', t_stop - t_start, 'seconds')
```

Pro každý prvek v `set(data)`,
projdí celý seznam `data`
a spočítá výskyt tohoto prvku

Měření času – modul `time`

`measure_runtime_func.py` – Funkce na monitorování funkcí

```
1 import time
2
3 def measure_runtime(some_function, function_args, num_repeats):
4     average_time = 0.0
5     for n in range(num_repeats):
6         t_start = time.perf_counter()
7         some_function(*function_args)
8         t_stop = time.perf_counter()
9         average_time += (t_stop - t_start) / num_repeats
10    return average_time
```

measure_runtime_demo.py

```
1 import time
2 from measure_runtime_func import measure_runtime
3
4 def generate_array_A(rows, cols, default_value):
5     return [[default_value] * cols] * rows
6
7 def generate_array_B(rows, cols, default_value):
8     array = []
9     for r in range(rows):
10         new_row = []
11         for c in range(cols):
12             new_row.append(default_value)
13         array.append(new_row)
14     return array
15
16 if __name__ == '__main__':
17     args = (4500, 4500, -1)
18     t = measure_runtime(generate_array_A, args, num_repeats=10)
19     print('Average time A: %.6f seconds' % t)
20     t = measure_runtime(generate_array_B, args, num_repeats=10)
21     print('Average time B: %.6f seconds' % t)
```

measure_runtime_demo.py

```
1 import time
2 from measure_runtime_func import measure_runtime
3
4 def generate_array_A(rows, cols, default_value):
5     return [[default_value] * cols] * rows
6
7 def generate_array_B(rows, cols, default_value):
8     array = []
9     for r in range(rows):
10         new_row = []
11         for c in range(cols):
12             new_row.append(default_value)
13         array.append(new_row)
14     return array
15
16 if __name__ == '__main__':
17     args = (4500, 4500, -1)
18     t = measure_runtime(generate_array_A, args, num_repeats=10)
19     print('Average time A: %.6f seconds' % t)
20     t = measure_runtime(generate_array_B, args, num_repeats=10)
21     print('Average time B: %.6f seconds' % t)
```

Co bude rychlejší?

(a)

(b)

Všechno má svoji cenu, jak vypadají data v paměti?

Python 3.6

[known limitations](#)

```
1 a = [[1]*3]*2
2 print(a)
3 a[0][1] = 9
4 print(a)
```

[Edit this code](#)

→ line that just executed
→ next line to execute



<< First < Prev Next > Last >>

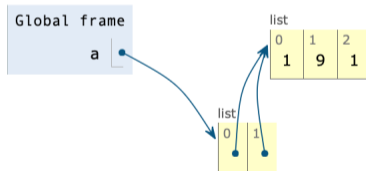
Done running (4 steps)

Print output (drag lower right corner to resize)

```
[[1, 1, 1], [1, 1, 1]]
[[1, 9, 1], [1, 9, 1]]
```

Frames

Objects



Všechno má svoji cenu, jak vypadají data v paměti?

Python 3.6
[known limitations](#)

```

1 matrix = []
2 for r in range(2):
3     row = []
4     for c in range(3):
5         row.append(1)
6     matrix.append(row)
7
8 print(matrix)
9 matrix[0][1] = 9
→ 10 print(matrix)

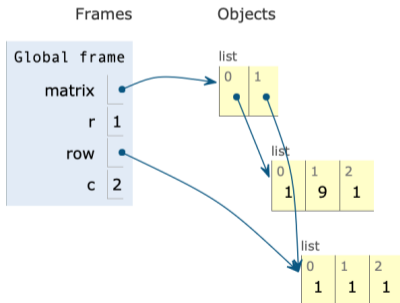
```

[Edit this code](#)

→ line that just executed
→ next line to execute

Print output (drag lower right corner to resize)

```
[[1, 1, 1], [1, 1, 1]]
[[1, 9, 1], [1, 1, 1]]
```



Všechno má svoji cenu, jak vypadají data v paměti?

Python 3.6
[known limitations](#)

```

1 matrix = []
2 for r in range(2):
3     row = [1]*3
4     matrix.append(row)
5
6 print(matrix)
7 matrix[0][1] = 9
→ 8 print(matrix)

```

[Edit this code](#)

→ line that just executed
→ next line to execute



<< First < Prev Next > Last >>

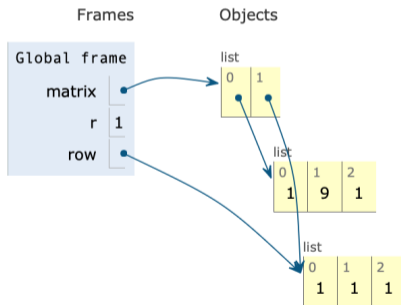
Done running (11 steps)

Print output (drag lower right corner to resize)

```

[[1, 1, 1], [1, 1, 1]]
[[1, 9, 1], [1, 1, 1]]

```



Modul `timeit`

- Vše co jsme si doposud ukázali (a mnohem víc)
- Spouštění z příkazového řádku
- Spouštění uvnitř `.py` programu
- Kód k analýze musí dostat jako string
- Během měření vypíná Garbage Collector → konzistence

Vsuvka – ovládání Garbage Collectoru

Lze dělat i ručně

```
>>> import gc  
>>> gc.disable()  
>>> gc.enable()
```

Vsuvka – ovládání Garbage Collectoru

Lze dělat i ručně

```
>>> import gc  
>>> gc.disable()  
>>> gc.enable()
```

Na vlastní nebezpečí...

gc.disable() be like



Modul timeit

```
1 import timeit
2
3 def generate_array_A(rows, cols, default_value):
4     return [[default_value] * cols] * rows
5
6 def generate_array_B(rows, cols, default_value):
7     array = []
8     for r in range(rows):
9         new_row = []
10        for c in range(cols):
11            new_row.append(default_value)
12        array.append(new_row)
13    return array
14
15 if __name__ == '__main__':
16     NUM_TRIALS = 10
17     a = timeit.timeit('generate_array_A(4500,4500,-1)',
18                      setup='from __main__ import generate_array_A',
19                      number=NUM_TRIALS)
20
21     b = timeit.timeit('generate_array_B(4500,4500,-1)',
22                      setup='from __main__ import generate_array_B',
23                      number=NUM_TRIALS)
24
25     print(f"Average time A: {(a/NUM_TRIALS):.6f} seconds")
26     print(f"Average time B: {(b/NUM_TRIALS):.6f} seconds")
```

timeit_demo.py

Generátory

- Efektivní způsob jak vyrábět sekvence pro smyčky
- Lepší než: vytvoř seznam, potom přes něj iteruj
- Každý prvek vznikne, až když ho potřebujeme
- Menší zátěž na paměť
- Zpřehlednění programu

Generátory; fib_generator.py – výpis N prvků Fibonacciho posloupnosti

$$F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}$$

```
1 def generate_fib(n):
2     a, b = (0, 1)
3     for i in range(n):
4         yield a
5         a, b = (b, a + b)
6
7 if __name__ == '__main__':
8     for num in generate_fib(10):
9         print(num)
```

Krokujte v pythontutor.com.

Generátory – co dělá `yield`

- Podobné použití jako `return`
- Předávání hodnot ven z funkce/metody
- Blok obsahující `yield` bude vracet objekt typu `generator`
- K samotným prvkům z generátoru se dostaneme přes `for` nebo `next`

Generátory – ukázky použití

generator_demo.py

```
1 def squared_sequence(start, stop, step=1):
2     num = start
3     while num < stop:
4         yield num ** 2
5         num += step
6
7 if __name__ == '__main__':
8     s = squared_sequence(3, 16, 1)
9     print(s)
10    print(type(s))
11    print(next(s)) # vygeneruj jeden nový prvek
12    print(next(s))
13    print(list(s)) # generator lze převést na list, tuple, set...
14    # print(next(s))
```

Generátory – ukázky použití

generator_demo.py

```
1 def squared_sequence(start, stop, step=1):
2     num = start
3     while num < stop:
4         yield num ** 2
5         num += step
6
7 if __name__ == '__main__':
8     s = squared_sequence(3, 16, 1)
9     print(s)
10    print(type(s))
11    print(next(s)) # vygeneruj jeden nový prvek
12    print(next(s))
13    print(list(s)) # generator lze převést na list, tuple, set...
14    # print(next(s)) Co se stane zde?
```

Generátory – generátorová notace

generator_comprehension.py

```
1 def squared_sequence(start, stop, step=1):
2     num = start
3     while num < stop:
4         yield num ** 2
5         num += step
6
7 if __name__ == '__main__':
8
9     s1 = squared_sequence(10, 15, 2)
10    s2 = (x**2 for x in range(10, 15, 2))
11
12    r1 = s1 == s2
13    r2 = list(s1) == list(s2)
14
15    print(r1, r2)
```

Generátory – generátorová notace

generator_comprehension.py

```
1 def squared_sequence(start, stop, step=1):
2     num = start
3     while num < stop:
4         yield num ** 2
5         num += step
6
7 if __name__ == '__main__':
8
9     s1 = squared_sequence(10, 15, 2)
10    s2 = (x**2 for x in range(10, 15, 2))
11
12    r1 = s1 == s2
13    r2 = list(s1) == list(s2)
14
15    print(r1, r2)
```

Jaký bude výpis?

- (a) True, True
- (b) True, False
- (c) False, True
- (d) False, False

Funkce `range()`

Pasivně známe, aktivně používáme

```
1 for i in range(-10, 10):  
2     print(i**2)  
3  
4 r = range(0, 20)  
5 print(type(r))
```

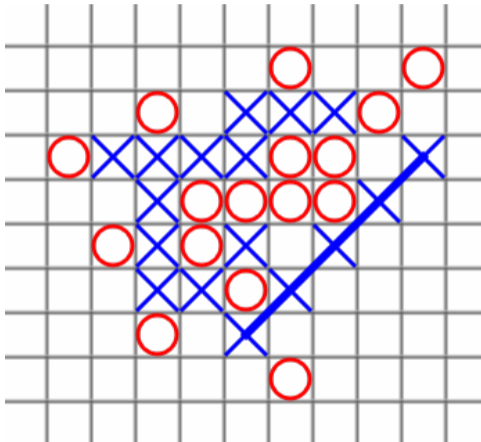
Co je `r` za objekt?

- (a) Generátor
- (b) List
- (c) Tuple
- (d) Range

Objekt range

- **Range není generátor ani tuple**, ale něco mezi tím...
- Neměnná sekvence
- Lze iterovat (víckrát)
- Nemá `next()`, má indexy
- Prvky vytváří, až když je potřebujeme
- Menší spotřeba paměti

Piškvorky



<https://cs.wikipedia.org/wiki/Piškvorky>

Dekompozice problému

- Snaha rozložit složitý problém na jednodušší části
- Ideálně tak jednoduché, že implementace je triviální
- První verze kódu stejně obvykle *nepřežije*
- Je snadnější zlepšovat části, než rozkopat celek

Piškvorcky – pomocné logické funkce

- Vrací **True** nebo **False**
- Jednoduchá operace, kterou ale potřebujeme často
- Zpřehledňují hlavní ideu algoritmu
- Volání funkcí v Pythonu není zadarmo...
- ...ale program obvykle zpomalují spíš jiné věci

```
if is_in_board(r,c):           vs.           if r >= 0 and r < 10 and c >= 0 and c < 10:
```

Piškvorky – příklady logických funkcí

- `is_empty(r,c)`
- `is_full(r,c)`
- `is_winning(r,c)`
- `is_out_of_bounds(r,c)`

Piškvorcky – objektově orientovaný návrh

```
1 import playfield
2
3 class BasePlayer:
4     def __init__(self, my_symbol, opponent_symbol, empty_symbol):
5         self.my_s = my_symbol
6         self.opp_s = opponent_symbol
7         self.empty_s = empty_symbol
8         self.field = playfield.Playfield(empty_symbol=self.empty_s)
9
10    def play(self, game_state):
11        self.field.update(game_state)
12        possible_moves = self.field.get_possible_moves()
13        return self.get_best_move(possible_moves)
14
15    def get_best_move(self, moves):
16        raise NotImplementedError
17
18 class SimplePlayer(BasePlayer):
19    def get_best_move(self, moves):
20        return moves[0]
```

Piškvorcky – práce s herní plochou

- Hra nám předá aktuální stav jako 2D pole (list of lists)
- Izolujeme nástroje, které se týkají přímo hrací plochy
- Uděláme si *wrapper* – objekt s pomocnými funkcemi
- `make_copy()` – pozor na mělké kopie!
- `__str__()` – pro čitelnější výpis
- `is_move_valid(r,c)`
- `is_move_winning(r,c)`
- `get_all_empty_fields()` – vhodný kandidát na generátor?

Piškvorcky – nepatrně lepší volba tahu

```
1 def get_all_empty_fields(self):
2     empty_fields = []
3     for r in range(self.size):
4         for c in range(self.size):
5             if self.is_empty(r,c):
6                 empty_fields.append((r,c))
7     return empty_fields
```

```
1 def get_all_empty_fields(self):
2     for r in range(self.size):
3         for c in range(self.size):
4             if self.is_empty(r,c):
5                 yield r,c
```

Píškorky – nepatrně lepší volba tahu

```
1 def get_best_move(self, moves):
2     my_move = None
3
4     for r,c in moves:
5         if self.field.is_winning(r,c):
6             return r,c # I can win with this move! Do it!
7         my_move = r,c
8
9     # otherwise return the last possible move
10    return my_move
```


Díky za pozornost

Jaká byla dnes rychlost výkladu?

- (A) Příště přidejte.
- (B) Tak akorát.
- (C) Prosím zpomalte.
- (D) Ona už přednáška skončila?

Díky za pozornost

Dnešní látka pro vás byla:

- (A) Lehká až triviální.
- (B) Tak akorát.
- (C) Těžká, ztrácel/a jsem se.
- (D) ...