B4B33RPH: Rešení problémů a hry

Python – základní kameny až skály III

Množiny, list comprehensions, profilování, generátory

Tomáš Svoboda, Petr Pošík, Petr Štibinger

stibipet@fel.cvut.cz

21. října 2024

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)
```

Jaký bude výpis?

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

Vyzkoušejte sami:

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

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 `set`u 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 `set`u 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 nejblíž 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

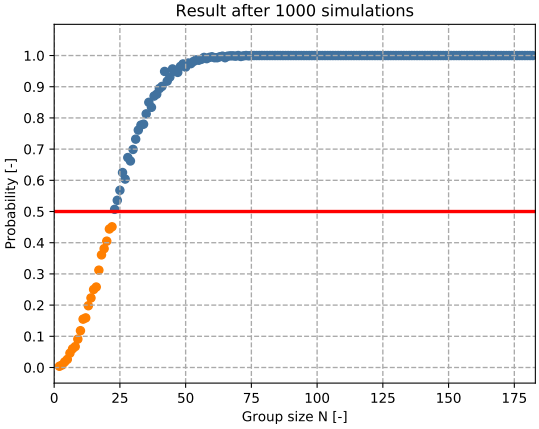
- Skupina N osob
- Jaká je pravděpodobnost, že alespoň dva lidé mají narozeniny ve stejný den?
- Pro jak velké N začne být **pravděpodobnější**, že alespoň jeden narozeninový den **není unikátní**?

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



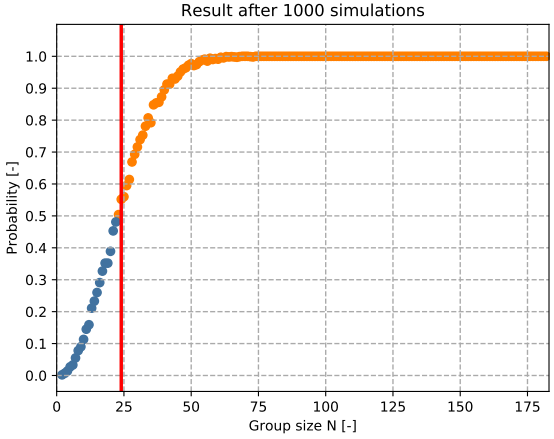
`birthday_problem.py` - celý kód včetně grafiky

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(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(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?

Měření času – modul **time**

frequency_example_time.py

```
 1  import time
 2
 3  def get_most_frequent_element(data):
 4      return max(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(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,
projdi celý seznam data
a spočítej 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)**, projdi celý seznam data a spočítej 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

```python
from measure_runtime_func import measure_runtime

def generate_matrix_A(rows, cols, default_value):
    return [[default_value] * cols] * rows

def generate_matrix_B(rows, cols, default_value):
    matrix = []
    for r in range(rows):
        new_row = []
        for c in range(cols):
            new_row.append(default_value)
        matrix.append(new_row)
    return matrix

if __name__ == '__main__':
    args = (4500, 4500, -1)
    t = measure_runtime(generate_matrix_A, args, num_repeats=10)
    print('Average time A: %.6f seconds' % t)
    t = measure_runtime(generate_matrix_B, args, num_repeats=10)
    print('Average time B: %.6f seconds' % t)
```

measure_runtime_demo.py

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

Co bude rychlejší?

(A)

(B)

measure_runtime_demo.py

```
1  from measure_runtime_func import measure_runtime
2
3  def generate_matrix_A(rows, cols, default_value):
4      return [[default_value] * cols] * rows
5
6  def generate_matrix_B(rows, cols, default_value):
7      return [[default_value] * cols for _ in range(rows)]
8
9  if __name__ == '__main__':
10     args = (4500, 4500, -1)
11     t = measure_runtime(generate_matrix_A, args, num_repeats=10)
12     print('Average time A: %.6f seconds' % t)
13     t = measure_runtime(generate_matrix_B, args, num_repeats=10)
14     print('Average time B: %.6f seconds' % t)
```

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

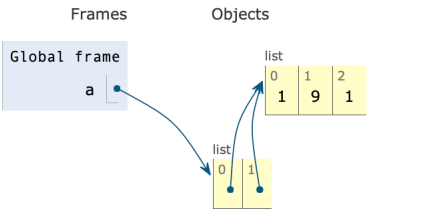
line that just executed
next line to execute

<< 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]]
```



Všechno má svoji cenu, jak vypadají data v paměti?

Python 3.6
known limitations

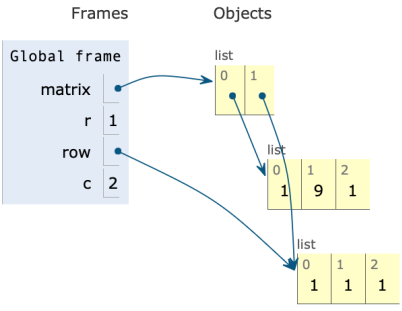
```
1  matrix = []
2  for r in range(2):
3      row = []
4      for c in range(3):
5          row.append(1)
6      matrix.append(row)
7
8  print(matrix)
9  matrix[0][1] = 9
10 print(matrix)
```

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

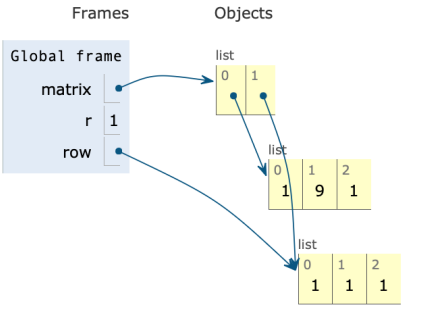
line that just executed
next line to execute

<< 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**

timeit_demo.py

```
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")
```

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 novy prvek
12     print(next(s))
13     print(list(s)) # generator lze prevest na list, tuple, set...
14     s = squared_sequence(3,16,1)
15     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 novy prvek
12     print(next(s))
13     print(list(s)) # generator lze prevest na list, tuple, set...
14     s = squared_sequence(3,16,1)
15     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))
```

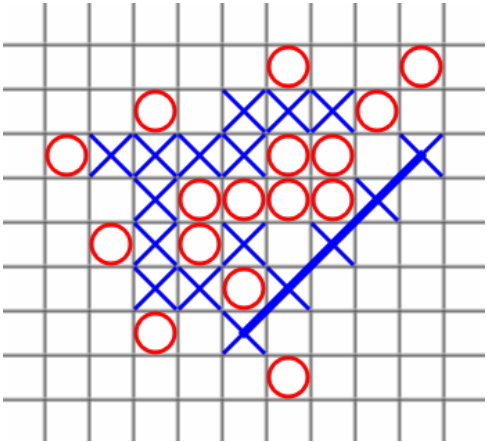
Co je **r** za objekt?

(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

Piškvorky – 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(pos):
```

vs.

```
if pos[0] >= 0 and pos[0] < 10 and pos[1] >= 0 and pos[1] < 10:
```

Piškvorky – příklady logických funkcí

- `is_empty(pos)`
- `is_full(pos)`
- `is_winning(pos)`
- `is_on_board(pos)`

Piškvorky – objektově orientovaný návrh

```
1  import board, random
2  from shared import TTT.cfg
3
4  class BasePlayer:
5      def __init__(self, mine_sym, opp_sym):
6          self.sym = mine_sym
7          self.opp_sym = opp_sym
8          self.empty_sym = TTTcfg.EMPTY_SYM
9          self.board = board.Board()
10
11     def play(self, matrix):
12         self.board.update(matrix)
13         possible_moves = list(self.board.empty_positions())
14         return self.choose_move(possible_moves)
15
16     def choose_move(self, moves):
17         raise NotImplementedError
18
19 class RandomPlayer(BasePlayer):
20     def choose_move(self, moves):
21         return random.choice(moves)
```

Piškvorky – 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(pos)`
- `is_move_winning(pos)`
- `empty_positions()` – vhodný kandidát na generátor?

Piškvorky – nepatrně lepší volba tahu

```
1  def empty_positions(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 empty_positions(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
```

Piškvorky – Greedy a Blocking

```
1  class EvalPlayer(BasePlayer):
2      def choose_move(self, valid_moves):
3          return max(valid_moves, key=self.evaluate)
4      def evaluate(self, move):
5          raise NotImplementedError(f"Implement evaluate() in {self.__class__.__name__
6
7  class GreedyPlayer(EvalPlayer):
8      def evaluate(self, move):
9          return len(self.board.longest_sequence(move, self.sym))
10
11 class BlockingPlayer(EvalPlayer):
12     def evaluate(self, move):
13         return len(self.board.longest_sequence(move, self.opp_sym))
```

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) ...

Temporary page!

LaTeX was unable to guess the total number of pages correctly. As there was some unprocessed data that should have been added to the final page this extra page has been added to receive it.
If you rerun the document (without altering it) this surplus page will go away, becau LaTeX now knows how many pages to expect for this document.