Kurs rozszerzony języka Python

Wykład 4.

Marcin Młotkowski

23 października 2019

Plan wykładu

1. Iteratory i generatory
   - Przetwarzanie iteracyjne kolekcji
   - Operacje na kolekcjach (iterable)
   - Generatory
   - itertools i przykład
2. Wejście/wyjście
   - Pliki tekstowe

Plan wykładu

Protokół iteracyjny

"Producent"

Umiem dostarczać kolejne elementy kolekcji po jednym elemencie, a jak już wszystkie dostarczę to poinformuję o tym.

"Konsument"

Daj kolejny element.

Konsumenci

- instrukcja for–in
- operator in

Naiwna wersja implementacji protokołu

Metody kolekcji

- `iter`: zainicjuj przeglądanie;
- `next`: zwróć element i przesuń wskaźnik; jeśli koniec zwróć None.

Przykładowa implementacja

```
class Kolekcja:
    def __init__(self):
        self.data = ["jeden", "dwa", "trzy"]
    def iter(self):
        self.pointer = 0
    def next(self):
        if self.pointer < len(self.data):
            self.pointer += 1
            return self.data[self.pointer - 1]
        else :
            return None
```

Wady rozwiązania

```
suma = 0
for x in wek_1:
    for y in wek_2:
        suma += x*y
```

Wady rozwiązania

```
wek_1 = wek_2 = Kolekcja()
suma = 0
for x in wek_1:
    for y in wek_2:
            suma += x*y
```

Postulat

Fajnie byłoby, żeby jedną kolekcję dało się przeglądać jednocześnie w kilku miejscach

Postulat

Fajnie byłoby, żeby jedną kolekcję dało się przeglądać jednocześnie w kilku miejscach

Diagnoza problemu

Kłopot jest dlatego, że jest tylko jeden wskaźnik do przeglądania.

Postulat

Fajnie byłoby, żeby jedną kolekcję dało się przeglądać jednocześnie w kilku miejscach

Diagnoza problemu

Kłopot jest dlatego, że jest tylko jeden wskaźnik do przeglądania.

Rozwiązanie problemu

Każdy konsument (pętla, wątek etc.) otrzymuje własny wskaźnik przeglądania.

Implementacja przeglądania kolekcji

Protokół (Python 3.*)

- Na początku wywoływana jest metoda __iter__;

Implementacja przeglądania kolekcji

Protokół (Python 3.*)

- Na początku wywoływana jest metoda __iter__;
- zwróconą wartością powinien być obiekt (enumerator) implementujący metodę __next__() która za każdym wywołaniem zwraca kolejny element kolekcji

Implementacja przeglądania kolekcji

Protokół (Python 3.*)

- Na początku wywoływana jest metoda __iter__;
- zwróconą wartością powinien być obiekt (enumerator) implementujący metodę __next__() która za każdym wywołaniem zwraca kolejny element kolekcji
- Metoda __next__() jest wywoływana tak długo, póki nie zostanie zgłoszony wyjątek StopIteration

Implementacja przeglądania kolekcji

Protokół (Python 3.*)

- Na początku wywoływana jest metoda __iter__;
- zwróconą wartością powinien być obiekt (enumerator) implementujący metodę __next__() która za każdym wywołaniem zwraca kolejny element kolekcji
- Metoda __next__() jest wywoływana tak długo, póki nie zostanie zgłoszony wyjątek StopIteration

Python 2.*

Zamiast __next__ jest next.

Przykład

Zadanie

Implementacja kolekcji zwracającej kolejne liczby od 1 do 10

Przykład

Zadanie

Implementacja kolekcji zwracającej kolejne liczby od 1 do 10

Implementacja

```
class ListaLiczb:
    def __iter__(self):
        self.licznik = 0
        return self
    def __next__(self):
        if self.licznik >= 10: raise StopIteration
        self.licznik += 1
        return self.licznik
```

Nieskończona lista liczb naturalnych

Iterator

```
class IntIterator(object):
    def __init__(self):
        self.licznik = 0

    def __next__(self):
        wynik = self.licznik
        self.licznik += 1
        return wynik
```

Implementacja kolekcji

```
class IntCollection(object):
    def __iter__(self):
        return IntIterator()
```

Zastosowanie

Obliczyć $\max(\Sigma_{i=0} i)$ takie że $\Sigma_{i=0} i < 100$

Zastosowanie

Obliczyć $\max(\Sigma_{i=0} i)$ takie że $\Sigma_{i=0} i < 100$

Rozwiązanie

```
suma = 0
for i in IntCollection():
    if suma + i >= 100: break
    suma += i
```

Jawne użycie iteratorów

```
>>> l = [1,2,3]
>>> it = iter(l)
>>> it.__next__()
1
>>> next(it)
2
>>> next(it)
3
>>> next(it)
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
StopIteration
```

Uwagi

Pytanie

Czy zawsze pożądane jest posiadanie więcej niż jednego iteratora?

Uwagi

Pytanie

Czy zawsze pożądane jest posiadanie więcej niż jednego iteratora?

Kontrprzykład

Przetwarzanie plików.

Ważne

- `filter`
- `map`
- `reduce`

Ważne

- `filter`
- `map`
- `reduce`

Python 2.*

Funkcje zwracają listę

Python 3.*

Funkcje zwracają iterator

Progamowanie funkcjonalne

Operatory (moduł `operator`)

operator.add(x,y)
operator.mul(x,y)
operator.pow(x,y)
...

Progamowanie funkcjonalne

Operatory (moduł `operator`)

operator.add(x,y)
operator.mul(x,y)
operator.pow(x,y)
...

Iloczyn skalarny

```
sum(map(operator.mul, vector1, vector2))
```

Iterator a lista

Python 3.*

```
list(filter(lambda x : x > 2, [1,2,3,4]))
```

Biblioteka *itertools*

Mnóstwo funkcji produkujących generatory

Biblioteka *itertools*

Mnóstwo funkcji produkujących generatory

Kolejne potęgi 2

```
it = map(lambda x : 2**x, itertools.count())
next(it)
next(it)
next(it)
...
```

Definicje

Generator

Generator to funkcja, która zwraca iterator.

Definicje

Generator

Generator to funkcja, która zwraca iterator.

Wyrażenie generatorowe

Wyrażenie generatorowe to wyrażenie, która zwraca iterator.

Jak implementować funkcje generatorowe

yield

Wykorzystanie yield

Implementacja nieskończonej listy potęg 2

```
def power2():
    power = 1
    while True:
        yield power
        power = power * 2
```

```
it = power2()
for x in range(4):
    print (next(it))
```

Wykorzystanie yield

Implementacja nieskończonej listy potęg 2

```
def power2():
    power = 1
    while True:
        yield power
        power = power * 2
```

```
it = power2()
for x in range(4):
    print (next(it))
```

Nieskończona pętla

```
for i in power2(): print (i)
```

Wyrażenia generatorowe

Instrukcja

```
wyr_generatorowe = (i** 2 for i in range(5))
```

jest równoważna

```
def wyr_generatorowe():
    for i in range(5):
        yield i**2
```

Zastosowanie

String szesnastkowo

```
":".join("{:02x}".format(ord(c)) for c in s)
```

yield from

Zamiast

Example

```
for item in iterable: yield item:
```

można pisać

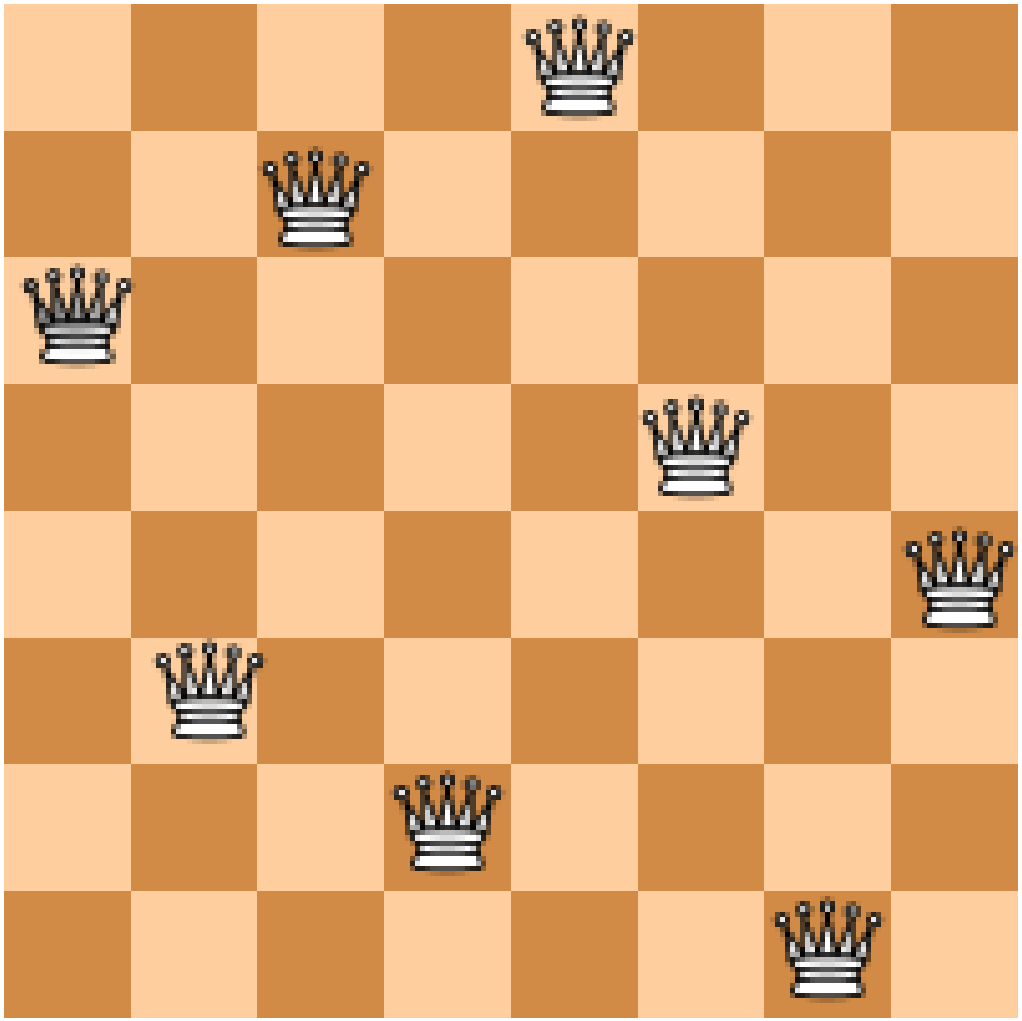
Example

```
yield from iterable
```

Moduł itertools

Generatory permutacji, kombinacji, iloczynów kartezjańskich

Problem 8 hetmanów



Wikipedia

Plan wykładu

Operacje na plikach

Otwarcie i zamknięcie pliku

```
fh = open('plik', 'r')
...
fh.close()
```

Atrybuty otwarcia

'r'	odczyt
'w'	zapis
'a'	dopisanie
'r+'	odczyt i zapis
'rb', 'wb', 'ab'	odczyt i zapis binarny

Metody czytania pliku

Odczyt całego pliku

fh.read()

Odczyt tylko size znaków

fh.read(size)

Odczyt wiersza, wraz ze znakiem '\n'

fh.readline()

Zwraca listę odczytanych wierszy

fh.readlines()

Tryby odczytu/zapisu

Tryb tekstowy

fh.read() zwraca string w kodowaniu takie jak ustawiono przy otwarciu pliku: `open(fname, 'r', encoding=''utf8'')`.

Tryb binarny `open(fname, 'rb')`

fh.read() zwraca ciąg binarny.

Odczyt pliku

Przykład

```
fh = open('test.py', 'r')
while True:
    wiersz = fh.readline()
    if len(wiersz) == 0: break
    print(wiersz)
fh.close()
```

Odczyt pliku

Przykład

```
fh = open('test.py', 'r')
while True:
    wiersz = fh.readline()
    if len(wiersz) == 0: break
    print(wiersz)
fh.close()
```

Inny przykład

```
fh = open('test.py', 'r')
for wiersz in fh:
    print(wiersz)
```

Zapis do pliku

```
fh.write('dane zapisywane do pliku\n')
fh.writelines(['to\n', 'są\n', 'kolejne\n', 'wiersze\n'])
```

Zamykanie pliku

Uwaga

Zawsze należy zamykać pliki.

Przykład

```
try:
    fh = open('nieistniejacy', 'r')
    data = fh.read()
finally:
    fh.close()
```

Zamykanie pliku

Uwaga

Zawsze należy zamykać pliki.

Przykład

```
try:
    fh = open('nieistniejacy', 'r')
    data = fh.read()
finally:
    fh.close()
```

Alternatywne zamykanie pliku

```
del fh
```

Zamykanie pliku

Porada

```
with open('nieistniejacy', 'r') as fh:
    data = fh.read()
```

Formaty danych

- Pliki tekstowe
- pickle
- Pliki z rekordami
- Pliki CSV
- Pliki *.ini
- XML
- ...