Obsah dnesnej prednasky

1. Iterator a generator

2. Lenive vyhodnocovanie (Lazy evaluation)

Iterator a Generator

inspirovane http://www.python-course.eu/python3_generators.php (http://www.python-course.eu/python3_generators.php)

Iterator

- je objekt, ktory ma funkciu `__next__` a funkciu `__iter__`, ktora vracia `self`
- je to vseobecnejsi pojem ako generator
- da sa pouzivat na iterovanie cez kolekciu bez toho, aby sme vedeli aka je jej vnutorna struktura. Staci definovat funkciu `__next__`. Podobny koncept sa da najst vo vela jazykoch. Napriklad aj v Jave.

Iterator sa napriklad implicitne pouziva pri prechadzani kolekcii for cyklom

In [ ]:
```
cities = ["Paris", "Berlin", "Hamburg", "Frankfurt", "London", "Vienna", "Amsterdam", "Den Haag"]
for location in cities:
    print("location: " + location)
```

In [ ]:
```
dir(cities.__iter__())
```

In [ ]:
```
print(type(cities.__iter__()))
print(type(cities.__iter__().__iter__()))
print(cities.__iter__().__next__())
```

Rovnako sa pouzivaju iteratory aj pri prechadzani inych kolekcii

In [ ]:
```
capitals = { "France":"Paris", "Netherlands":"Amsterdam", "Germany":"Berlin", "Switzerland":"Bern", "Austr
ia":"Vienna"}
for country in capitals:
    print("The capital city of " + country + " is " + capitals[country])
```

Generator

- kazdy generator objekt je iterator, ale nie naopak
- tento pojem sa pouziva na pomenovanie funkcie (generator funkcia) ako aj jej navratovej hodnoty (generator objekt)
- generator objekt sa vytvara volanim funkcie (generator funkcie), ktora pouziva `yield`

Generator pouziva vyraz `yield` na zastavenie vykonavania a na vratenie hodnoty

- Vykonavanie sa spusta folanim funkcie `next()` (alebo metody `__next__()`)
- Dalsie volanie zacina od posledneho `yield`
- Medzi volaniami sa hodnoty lokalnych premennych uchovavaju.

Pozor, toto nieje ten isty `yield` ako je v Ruby

- V Ruby je `yield` volanie bloku asociovaneho s metodou
- V Ruby je nieco podobne generatorom napriklad trieda `Enumerator`

http://stackoverflow.com/questions/2504494/are-there-something-like-python-generators-in-ruby__(http://stackoverflow.com/questions/2504494/are-there-something-like-python-generators-in-ruby)

In [ ]:
```
def city_generator():
    yield "Konstanz"
    yield "Zurich"
    yield "Schaffhausen"
    yield "Stuttgart"
```

In [ ]:
```
gen = city_generator()
```

In [ ]:
```
next(gen)
```

Vo vnutri generator funkcie mozem pouzivat cyklus

In [ ]:
```
cities = ["Konstanz", "Zurich", "Schaffhausen", "Stuttgart"]
def city_generator():
    for city in cities:
        yield city
gen = city_generator()
```

In [ ]:
```
next(gen)
```

Tento genereator vlastne len supluje iterator, ktory je nad polom, ale ten cyklus moze robit aj nieco viac a vtedy to uz moze byt zaujimavejsie (ukazem neskor)

Generator funkcia moze prijmat parametre

In [ ]:
```
def city_generator(local_cities):
    for city in local_cities:
        yield city
gen = city_generator(["Konstanz", "Zurich", "Schaffhausen", "Stuttgart"])
```

In [ ]:
```
next(gen)
```

Trik ako napisat generator, ktory velmi casto funguje

Uloha: Mame sekvenciu cisel a chceme vytvorit pohyblivy priemer dvoch po sebe nasledujucich cisel pre celu sekvenciu.

napr:

sekvencie = [1,2,3,4,5]

phyblivy priemer = [(0+1)/2, (1+2)/2, (2+3)/2, (3+4)/2, (4+5)/2] = [0.5, 1.5, 2.5, 3.5, 4.5]

Ako by ste to napisali imperativne ak to chcete len zapisat do konzoly?

In [ ]:
```
sequence = [1,2,3,4,5]
previous = 0
for actual in sequence:
    print((actual + previous) / 2)
    previous = actual
```

Zabalim to do funkcie

In [ ]:
```
sequence = [1,2,3,4,5]
def moving_average(sequence):
    previous = 0
    for actual in sequence:
        print((actual + previous) * 0.5)
        previous = actual
moving_average(sequence)
```

Vymenim print za `yield`

In [ ]:
```
sequence = [1,2,3,4,5]
def moving_average(sequence):
    previous = 0
    for actual in sequence:
        yield (actual + previous) * 0.5
        previous = actual
```

In [ ]:
```
print(list(moving_average(sequence)))
```

Pomocou generatoru by sa dala napriklad spravit funkcia `map`

In [ ]:
```
def map(f, seq):
    for x in seq:
        print(f(x))
```

In [ ]:
```
def map(f, seq):
    for x in seq:
        yield f(x)
```

Porovnajte si ako by vyzerala implementacia `map` v python2 a python3

In [ ]:
```
def map(f, seq): # V pythone 2 map vracia list, implementacia by mohla byt napriklad takato
    result = [] # mame premennu, ktoru postupne upravujeme a nafukujeme
    for x in seq:
        result.append(f(x))
    return result
```

In [ ]:
```
def map(f, seq): # V pythone 3 map je generator a zabera konstantne mnozstvo pamati
    for x in seq:
        yield f(x)
```

Niektore generatory sa daju nahradit funkciou `map`

In [ ]:
```
a, b = 1, 10
def squares(start, stop):
    for i in range(start, stop):
        yield i * i

generator = squares(a, b)
print(generator)
print(next(generator))
print(list(generator))
```

In [ ]:
```
generator = map(lambda i: i*i, range(a, b))
print(generator)
print(next(generator))
print(list(generator))
```

List comprehension tiez moze vytvarat generator

In [ ]:
```
generator = (i*i for i in range(a, b)) # rozdiel oproti kalsickemu LC je v zatvorkach [] => ()
print(generator)
print(next(generator))
print(list(generator))
```

Explicitny generator ma ale vacsiu vyjadrovaciu silu

Nie je obmedzeny len na formu ktoru pouziva funkcia map:

In [ ]:
```
def generator(funkcia, iterator):
    for i in iterator:
        yield funkcia(i)
```

```
        yield funkcia(1)
```

Na co je to cele dobre?

Lenive vyhodnocovanie - Lazy evaluation

Strategie vyhodnocovania

Skratene vyhodocovanie (Short-circuit)

Netrpezlive vyhodocovanie (Eager)

Lenive vyhodnocovanie (Lazy)

Vzdialene vyhodocovanie (Remote)

Ciastocne vyhodocovanie (Partial)

https://en.wikipedia.org/wiki/Evaluation_strategy (https://en.wikipedia.org/wiki/Evaluation_strategy)

Skratene vyhodnocovanie

In [ ]:
```
def fun1(pole):
    print('prva')
    return False

def fun2(pole):
    print('druha')
    return True

if fun1(pole) or fun2(pole):
    pass
```

Lenive vyhodocovanie

Oddaluje vyhodnocovanie az do doby, ked je to treba

In [ ]:
```
pom = (x*x for x in range(5))
next(pom) #prvok z generatora sa vyberie az ked ho je treba a nie pri vytvoreni generatora
```

Nedockave vyhodocovanie

Opak leniveho vyhodnotenia. Vyraz sa vyhodnoti hned ako je priradeny do premennej. Toto je typicky sposob vyhodnocaovania pri vacsine programovacich jazykoch.

In [ ]:
```
pom = [x*x for x in range(5)]
pom[4] # vyraz sa hned vyhodnocuje cely
```

Vyhody nedockaveho vyhodnocovania

- programator moze kontrolovat poradie vykonavania
- nemusi sledovat a planovat poradie vyhodnocovania

Nevyhody

- neumoznuje vynechat vykonavanie kodu, ktory vobec nieje treba (spomente si na priklad so Sparkom)
- neda sa vykonavat kod, ktory je v danej chvili dolezitejsi
- programator musi organizovat kod tak, aby optimalizoval poradie vykonavania

Moderne kompilatory ale uz niektore veci vedia optimalizovat za programatora

Vzdialene vyhodnocovanie

- Vyhodnocovanie na vzdialenom pocitaci.
- Hociaky vypoctovy model, ktory spusta kod na inom stroji.
- Client/Server, Message passing, MapReduce, Remote procedure call (RCP)

Partial evaluation

- Viacero optimalizacnych strategii na to aby sme vytvorili program, ktory bezi rychlejsie ako povodny program.
  - Napriklad predpocitavanie kodu na zaklade dat, ktore su zname uz v case kompilacie.
  - Mamoization (Memoizacia?) - nevykonavanie (cistych) funkcii s rovnakymi vstupmi opakovane. V podstate ide o caschovanie vystupov volani funkcii
  - Partial application - fixovanie niektorych parametrov funkcie a vytvorenie novej s mensim poctom parametrov.

Lenive vyhodnocovanie moze zrychlit vyhodocovanie

In [ ]:
```
%%time
print(2+2)
```

In [ ]:
```
%%time
import time
def slow_square(x):
    time.sleep(0.2)
    return x**2

generator = map(slow_square, range(10))
print(generator)
```

Funkcia slow_square sa zatial nespustila ani raz. Preto je ten cas tak maly

In [ ]:
```
# co sa stane ak budeme chciet transformovat generator na zoznam. Teda spustit ru pomalu funkciu na kazdom
 prvku?
%%time
print(list(generator))
```

In [ ]:
```
# Aj ked chceme len cast pola, tak musime transformovat vsetky prvky
%%time
generator = map(slow_square, range(10))
pole = list(generator)
print(pole[:5])
```

In [ ]:
```
# Mozeme si ale skusit definovat funkciu, ktora nam vyberie len tu cast prvkov, ktore chceme
def head(iterator, n):
    result = []
    for _ in range(n):
        result.append(next(iterator))
    return result
```

In [ ]:
```
%%time

print(head(map(slow_square, range(10)), 5))
```

Ta pomala operacia sa vykonala len tolko krat, kolko sme potrebovali a to co sme nepotrebovali sa nemuselo nikdy vykonat.

In [ ]:
```
%%time
# tuto funkciu sme si ale nemuseli definovat sami. Nieco take uz existuje

from itertools import islice
generator = map(slow_square, range(10000))
print(list(islice(generator, 5)))
```

Lenive vyhodnocovanie setri pamat

In [ ]:
```
from operator import add
from functools import reduce

reduce(add, [x*x for x in range(10000000)])
reduce(add, (x*x for x in range(10000000))) # rozdiel je len v zatvorkach
```

skusim si vyrobit funkciu, ktora mi bude priebezne pocitat a vypisovat aktualnu spotrebu pamati premennych na halde pocas toho ako budeme spocitavat cisla

In [ ]:
```
from functools import reduce
import gc
import os
import psutil
process = psutil.Process(os.getpid())

def print_memory_usage():
    print(process.memory_info().rss)

counter = [0] # Toto je hnusny hack a slubujem, ze nabuduce si povieme ako to spravit lepsie. Spytajte sa
 ma na to!
# Problem je v tom, ze potrebujem pocitadlo, ktore bude dostupne vo funkcii ale zaroven ho potrebujem inic
ializovat mimo tejto funkcie
# Teraz som zaspinil funkciu pouzitim mutable datovej struktury a globalneho priestoru mien. 2xFuj!
def measure_add(a, result, counter=counter):
    if counter[0] % 2000000 == 0:
        print_memory_usage()
    counter[0] = counter[0] + 1
    return a + result
```

In [ ]:
```
gc.collect()
counter[0] = 0
print_memory_usage()
print(reduce(measure_add, [x*x for x in range(10000000)]))
```

In [ ]:
```
gc.collect()
counter[0] = 0
print_memory_usage()
print(reduce(measure_add, (x*x for x in range(10000000))))
```

Ani ked su funkcie povnarane do seba a kolekcia sa predava ako parameter, nikdy nie je cela v pamati

map, filter, reduce aj list comprehension vnutorne pracuju skolekciami ako s iteratormi

In [ ]:
```
gc.collect()
counter[0] = 0
print_memory_usage()
print(reduce(measure_add, filter(lambda x: x%2 == 0, map(lambda x: x*x, range(10000000)))))
# a pokojne by som to mohol vnarat dalej
```

Ked vieme, ze generator sa vyhodnocuje lenivo, tak nam nic nebrani vlozit do neho nekonecny cyklus

In [ ]:
```
def fibonacci():
    """Fibonacci numbers generator"""
    a, b = 1, 1
    while True:
        yield a
        a, b = b, a + b

f = fibonacci()
```

In [ ]:
```
print(list(islice(f, 10)))
```

Voila, nekonecna datova struktura, ktora nezabera skoro ziadnu pamat

Volia, nekonecna datova struktura, ktora nezaberia skoro ziadnu pamat dokedy ju nechcem materializovat celu.

In [ ]:
```
list(fibonacci()) # toto netreba pustat
```

Vedeli by ste to pouzit na:

- generator prvocisel?
- citanie z velmi velkeho suboru, ktory vam nevojde do pamati?
- citanie dat z nejakeho senzoru, ktory produkuje kludne nekonecne mnozstvo dat?

Dalo by sa to pouzit napriklad na cakanie na data

Predstavte si, ze mate subor, do ktoreho nejaky proces zapisuje logy po riadkoch a vy ich spracovavate.

Ako by ste spravili iterovanie cez riadky suboru tak, aby ste cakali na dalsie riadky ak dojdete na koniec suboru?

inspirovane - http://stackoverflow.com/questions/6162002/whats-the-benefit-of-using-generator-in-this-case (http://stackoverflow.com/questions/6162002/whats-the-benefit-of-using-generator-in-this-case)

In [ ]:
```
%%bash
echo -n 'log line' > log.txt
```

In [ ]:
```
import time
```

In [ ]:
```
# s generatorom napriklad takto
def read(file_name):
    with open(file_name) as f:
        while True:
            line = f.readline()
            if not line:
                time.sleep(0.1)
                continue
            yield line

lines = read("log.txt")
print(next(lines))
```

In [ ]:
```
print(next(lines))
```

In [ ]:
```
for line in lines:
    print(line)
```

Toto by som vedel spravit aj bez generatora ale ...

- nemal by som oddelenu logiku cakania a spracovavania riadku
- zneuzivam necistu funkciu print
- nevedel by som priamociaro znovupouzivat generator, vzdy by som to musel kodit odznova
  - jedine, ze by som pouzil funkciu ako parameter
  - stale tam ale zostava problem ako vratit viacero hodnot z jednej funkcie
- nevedel by som pekne transparentne, lenivo iterovat

In [ ]:
```
while True:
    line = logfile.readline()
    if not line:
        time.sleep(0.1)
        continue
    print line
```

Generator moze byt aj trochu zlozitejsi, napriklad rekurzivny

Predstavte si takuto stromovu strukturu

In [ ]:
```
class Node(object):
```

```
    def __init__(self, title, children=None):
        self.title = title
        self.children = children or []

tree = Node(
    'A', [
        Node('B', [
            Node('C', [
                Node('D')
                ]),
            Node('E'),
            ]),
        Node('F'),
        Node('G'),
        ])
```

In [ ]:
```
def node_recurse_generator(node):
    yield node
    for n in node.children:
        for rn in node_recurse_generator(n):
            yield rn

[node.title for node in node_recurse_generator(tree)]
```

http://stackoverflow.com/posts/7634323/edit (http://stackoverflow.com/posts/7634323/edit)

Uloha na volny cas

vedeli by ste vytvorit datovu strukturu `list_r`, ktora by obsahovala prvy prvok zoznamu a jeho zvysok (first, rest)? Vedeli by ste vytvorit rekurzivne funkcie a generatory, ktore by spracovavali zoznam tak ako ste to robili v LISPe?

Ale castokrat sa to da aj bez pouzitia rekurzie

http://stackoverflow.com/questions/26145678/implementing-a-depth-first-tree-iterator-in-python
(http://stackoverflow.com/questions/26145678/implementing-a-depth-first-tree-iterator-in-python)

In [ ]:
```
from collections import deque

def node_stack_generator(node):
    stack = deque([node]) # tu si uchovavam stav prehladavania kedze nepouzivam call stack v rekurzii
    while stack:
        # Pop out the first element in the stack
        node = stack.popleft()
        yield node
        # push children onto the front of the stack.
        # Note that with a deque.extendleft, the first on in is the last
        # one out, so we need to push them in reverse order.
        stack.extendleft(reversed(node.children))

[node.title for node in node_stack_generator(tree)]
```

Uloha na volny cas

Vedeli by ste tieto dva generatory upravit pre binarny strom?

Rekurzivny generator sa da napriklad pouzit na vyrabanie permutacii

In [ ]:
```
def permutations(items):
    n = len(items)
    if n==0:
        yield []
    else:
```