Iterators

Alexander Evgin

28 февраля 2020 г.

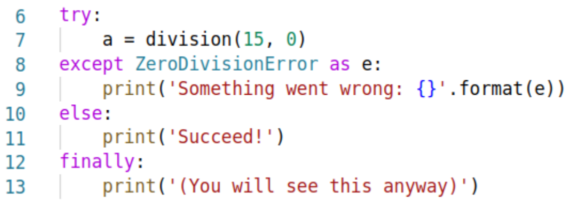
Outline

Recap

- Method vs. function
- `@classmethod`, `@staticmethod`, `__slots__`
- Inheritance
  - MRO
  - `object` — base class for any class

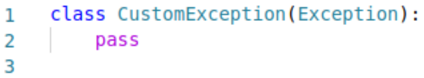
Recap

- Exceptions



```python
try:
    a = division(15, 0)
except ZeroDivisionError as e:
    print('Something went wrong: {}'.format(e))
else:
    print('Succeed!')
finally:
    print('(You will see this anyway)')
```

  - оператор `raise`
  - передача исключения вглубь по ***стеку вызовов***
  - custom exceptions



```python
class CustomException(Exception):
    pass

```

Classes. Conclusion

Enum

```python
from enum import Enum

class Animal(Enum):
    CAT = 0
    DOG = 1
    MONKEY = 2

x = Animal.DOG
print(x)          # Animal.DOG
print(type(x))    # <enum 'Animal'>
print(x.name)     # DOG
print(x.value)    # 1
```

Enum

```python
from enum import Enum, auto

class Animal(Enum):
    CAT = auto()
    DOG = auto()
    MONKEY = auto()
```

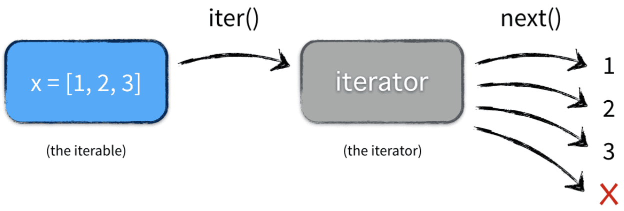
Examples

(dive into code)

- pandas
- werkzeug
- kubernetes-client

Iterators

Iterators



```
set;
iterator = iter(set);
element_1 = next(iterator);
element_2 = next(iterator);
...
```

Iterators

```
for x in xs:
    body
```

Iterators

```python
for x in xs:
    body

it = xs.__iter__()
while True:
    try:
        x = it.__next__()
    except StopIteration:
        break
    body
```

Iterator

```python
class Iterator:
    def __next__(self):
        if self.has_more_elements():
            return self.next_element()
        raise StopIteration

it = Iterator()
elem = next(it, default)
```

Iteratable

```python
class Iterable:
    def __iter__(self):
        return Iterator()

x = Iterable()
it = iter(x)  # calls x.__iter__()
```

Iterat**or** is iterat**able**

```python
class Iterator:
    def __next__(self):
        ...

    def __iter__(self):
        return self
```

```python
class range:
    def __init__(self, start, stop):
        self.start = start
        self.stop = stop

    def __iter__(self):
        return RangeIterator(self.start, self.stop)


class RangeIterator:
    def __init__(self, start, stop):
        self.start = start
        self.stop = stop

    def __iter__(self):
        return self

    def __next__(self):
        if self.start < self.stop:
            res = self.start
            self.start += 1
            return res
        raise StopIteration
```

in

```python
class range:
    ...

class RangeIterator:
    ...

r = range(0, 100)
assert 42 in r # O(N)
```

Exhausted iterator

```
>>> r = [1, 2, 3, 4]
>>> sum(r)
10
>>> sum(r)
10
>>> it = iter(r)
>>> sum(it)
10
>>> sum(it)
0
```

Iterators “length”

```
it = iter([1, 2, 3])
assert it.__length_hint__() == 3  # it not exhausted!
                                  # not precise

length = sum(1 for _ in it)  # it exhausted
```

Collections (containers)

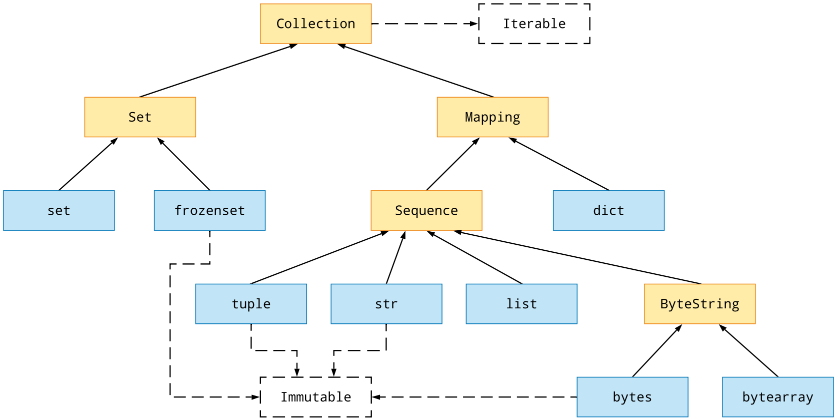
"Some objects contain references to other objects; these are called **containers**. *Examples of containers are tuples, lists and dictionaries."*

Python Language Ref

Вывод: container — это что-то, что contain

Контейнер — имеет `add`, `get`, `remove` в некотором виде

Collections (containers)



Generators

Generators

```python
def range(start, stop):
    while start < stop:
        yield start
        start += 1
```

Generators

```python
def range(start, stop):
    while start < stop:
        yield start # <-- turns function into generator
        start += 1
```

Generators

```python
def range(start, stop):
    while start < stop:
        yield start
        start += 1

# Generator is iterator:
r = range(0, 10)
next(r)  # 0
```

Generators

```python
def g():
    print("started")
    x = 42
    yield x
    print("yielded once")
    x += 1
    yield x
    print("yielded twice, done")

t = g()
for x in it:
    print(x)

# started
# 42
# yielded once
# 43
# yielded twice, done
```

Generators

```python
def g():
    yield

type(g)  # <class 'function'>
type(g())  # <class 'generator'>
dir(g())  # [..., 'close', 'send', 'throw']
```

yield from

```
def f(iterable):
    for item in iterable:
        yield item

def f(iterable):
    yield from iterable  # yield from [1, 2, 3]
```

Examples

```python
def unique(xs):
    seen = set()
    for item in xs:
        if item in seen:
            continue
        seen.add(item)
        yield item

xs = [1, 1, 2, 3]
assert list(unique(xs)) == [1, 2, 3]
```

Examples

```python
def chain(*xss):
    for xs in xss:
        yield from xs

xs = [1, 2, 3]
ys = [92]
assert list(chain(xs, ys)) == [1, 2, 3, 92]
```

Examples

```python
def count(start=0):
    while True: # бесконечный генератор!
        yield start
        start += 1
```

Generator expressions

```
>>> (x * x for x in xs)
<generator object <genexpr> at 0x7ff7437bbeb8>

>>> sum(x**2 for x in range(10))  # нет()
285

>>> map(lambda x: x * x, xs)
<map object at 0x7ff7437c3ac8>
>>> # map(lambda и filter(lambda всегда длиннее
```

Comprehension expressions

```python
xs = [1, 2, 3]
xss = [x ** 2 for x in xs]
assert xss == [1, 4, 9]

dss = {x: x ** 2 for x in xs}
assert dss == {1: 1, 2: 4, 3: 9}
```

Consuming generators

```python
r = range(10)  # O(1) memory used
l = list(r)  # consumed, O(N) memory now
```

Consumers:

- `list`, `set`, `tuple`, ...
- `sum`
- `all`, `any`

itertools

```python
from itertools import islice

xs = range(10)

assert list(islice(xs, 2, 8, 3)) == [2, 5]
```

itertools

```python
from itertools import count, cycle, repeat, islice

def take(n, xs):
    return list(islice(xs, 0, n))
```

itertools

```python
from itertools import count, cycle, repeat, islice

def take(n, xs):
    return list(islice(xs, 0, n))

assert take(3, count(start=1, step=2)) == [1, 3, 5]
assert take(3, cycle(["любит", "не любит"])) ==  \
     ["любит", "не любит", "любит"]
assert take(3, repeat(92)) == list(repeat(92, times=3))
```

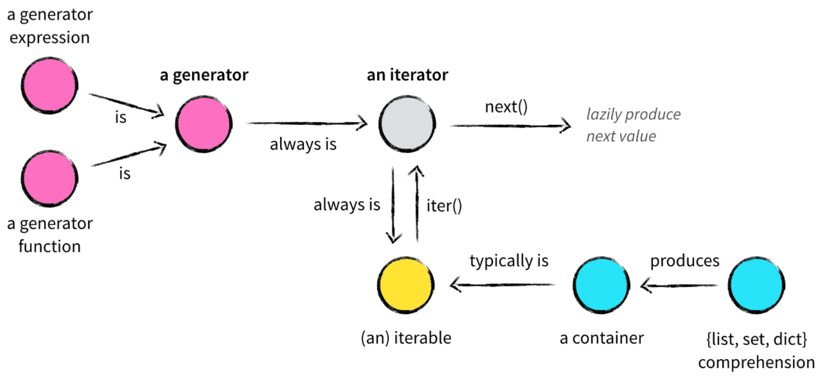
itertools

```python
from itertools import chain

assert list(chain(range(2), "ab")) == [0, 1, "a", "b"]
```

itertools

```python
from itertools import product, combinations, permutations

assert list(product("AB", repeat=2)) \
    == [("A", "A"), ("A", "B"), ("B", "A"), ("B", "B")]

assert list(product("AB", "CD")) \
    == [("A", "C"), ("A", "D"), ("B", "C"), ("B", "D")]

assert list(combinations("ABC", 2)) == \
    [("A", "B"), ("A", "C"), ("B", "C")]

assert list(permutations('ABC', 2)) == \
     [("A", "B"), ("A", "C"), ("B", "A"), ("B", "C"),
      ("C", "A"), ("C", "B")]
```

Sum-up



a generator expression
a generator function
is
is
a generator
always is
an iterator
next()
lazily produce next value
always is
iter()
(an) iterable
typically is
a container
produces
{list, set, dict} comprehension

Generators magic — coroutines

```python
def running_sum():
    acc = 0
    while True:
        acc += yield acc

s = running_sum()
s.send(None)  # 0
s.send(1)  # 1
s.send(1)  # 2
s.send(1)  # 3
```

Q & A

Начнем с: comprehension-ы, `itertools`

Задание

- Реализовать итератор, обходящий список в обратном порядке
- Реализовать итератор, обходящий список от наименьшего элемента до наибольшего
- Реализовать функцию-генератор `map`
- Реализовать генератор чисел Фибоначчи

Самостоятельное задание

- ***(5 баллов)*** Реализовать функцию

  ```
  takesuite(func, iterable)
  ```

  которая находит первые подряд идущие элементы итератора, для которых `func` возвращает `True`.

  ```
  assert takesuite(lambda x: x > 2, [2, 3, 4, 1]) == \
      [3, 4]
  ```