Итераторы

Как работает цикл «for in»?

- Используем:

```
for element in container:
    do_something_with(element)
```

- Фактически:

```
iter_c = iter(container)                # < create iterator
while True:                             # infinite loop
    try:
        element = next(iter_c)          # < take one element
        do_something_with(element)
    except StopIteration:               # loop ends
        break
```

☞ отлавливается исключение StopIteration

☞ Итератор — объект, который имеет два специальных метода, так называемый протокол итератора: `iter()` и `next()`

Пример: итерация по списку

```
l=[2,4,6]
it=iter(l)          # type(it) : <class 'list_iterator'>
print( next(it) )   # 2
print( next(it) )   # 4
print( next(it) )   # 6
print( next(it) )   # exception: StopIteration
```

- Реализация этих протоколов для пользовательских классов осуществляется методами `__iter__()` и `__next__()`

Пример, класс с итератором:

```
class Eratosthenes:
    def __init__(self,nmax):   # create sieve of Eratosthenes
        self.sieve = [0,0] + [1]*(nmax-2)
        for p in range(2,nmax):
            if self.sieve[p] == 0: continue
            for i in range(p*p,nmax,p): self.sieve[i] = 0

    def __iter__(self):        # < create iterator
        self.it=0              # internal variable for iteration
        return self            # must return self !

    def __next__(self):        # < take one element
        try:
            self.it += 1
            while self.sieve[self.it]==0:
                self.it += 1
        except:
                raise StopIteration
        return self.it
```

Тест:

```
for p in Eratosthenes(50):
    print(p,end=',')      # 2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,
print()
print(Eratosthenes(10))  # <__main__.Eratosthenes object at 0x...
print(list(Eratosthenes(10)))  # [2, 3, 5, 7]
```

Пояснения

- ✓ `__iter__()` должен возвращать объект по которому итерируем: `self`
- ✓ `__next__()` должен возвращать следующий элемент, а после последнего элемента возбуждать `StopIteration`

☞ `range()` – функция возвращающая итераторы (генератор)

```
print(range(5))          # range(0, 5)
print(list(range(5)))    # [0, 1, 2, 3, 4]
```

Функции, работающие с итерируемыми объектами

`sum(iter[,start=0])`	сумма чисел
`list(iter)`	возвращает список
`tuple(iter)`	возвращает кортеж
`dict(iter)`	возвращает словарь
`all(iter)`	возвращает `True` если все элементы истинны
`any(iter)`	возвращает `True` если хотя бы один элемент истинен
`enumerate(siquence [,start=0])`	возвращает tuple состоящий из *(number,element)*
`sorted(iter [,key,reverse])`	возвращает сортированный список: `key` – «ключевая функция», `reverse=True` меняет порядок сортировки
`min/max(iter,...)`	возвращает наибольший/наименьший элемент
`zip(iter1[,iter2...])`	возвращает объединяющий итератор: `tuple(1,2...)`
`filter(fun,iter)`	возвращает итератор для которых `fun(i) == True`
`map(fun,iter)`	`fun` к каждому элементу и возвращает итератор

Пример с map() и list()

```
p2=map(lambda a: a*a, Eratosthenes(20))
print('map: ',list(p2)) # map: [4, 9, 25, 49, 121, 169, 289, 361]
```

Пример с zip() и dict()

```
strCard='23456789TJQKA'
lstCard=list(range(2,15)) # [2,3,4,5,6,7,8,9,10,11,12,13,14]
dctCard=dict(zip(strCard,lstCard))
print(dctCard) # {'2': 2, '3': 3, '4': 4,.. 'Q': 12, 'K': 13, 'A': 14}
```

Пример с filter() и list()

```
pf=filter( lambda a: a<10 or \
   len([s for s in str(a) if s in '024568'])==0, \
   Eratosthenes(100))
print('filter: ',list(pf))
# filter:  [2, 3, 5, 7, 11, 13, 17, 19, 31, 37, 71, 73, 79, 97]
```

Генераторы

- Генераторы – функции возвращающие итераторы
- Выглядят как обычные функции, но вместо `return` ставится оператор `yield`

Пояснение на примере:

very simple generator function

```
def simpleGen():
    i=0
    yield i
    i=1
    yield i
```

```
for i in simpleGen():      # 'for in' loop with the generator
    print(i,end=' ')
print() # 0 1
```

Пояснения

- ✓ генератор возвращает объект для которого методы `iter()` и `next()` автоматически созданы (итератор)
- ✓ локальные переменные генератора и текущее состояние сохраняются между вызовами
- ✓ каждый следующий вызов генератора продолжает выполнение с сохраненного состояния
- ✓ при достижении конца функции или оператора `return` возбуждается исключение `StopIteration`

☞ `zip()`, `map()`, `filter()` – генераторы, они возвращают итераторы

Пример: генератор простых чисел

```
def Primes(nmax):
    l=[2]                          # list for prime numbers
    yield l[-1]                    # 'return' last elements of list
    def isPrime(n):                # helper nested function
        for p in l:
            if p*p > n: return True
            if n%p == 0: return False
        return True
    for n in range(3,nmax,2):      # fill in the list
        if isPrime(n):
            l.append(n)
            yield l[-1]            # 'return' last elements of list

print('Primes: ',list(Primes(50)))
Primes:  [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

Генераторные выражения (Generator Expressions)

Сравнение с генератором списка

1. Генераторное выражение: (*expression* `for` *expr* `in` *sequence* `if` *condition*)
2. Генератор списка: [*expression* `for` *expr* `in` *sequence* `if` *condition*]

- Генератор списка создает список
- В генераторном выражении создается анонимная генераторная функция

☞ Реальные вычисления с генераторами «откладываются на потом»: концепция ленивых вычислений (*lazy evaluation*)

```
al = [x**2 for x in range(10)]          # list
print(al)      # [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
ag = (x**2 for x in range(10))          # generator
print(ag)      # <generator object <genexpr> at 0x7f75c649faa0>
print(sum(ag)) # 285
```

Замыкание (closure)

☞ Замыкание это механизм связывающий код функции с ее лексическим окружением: «добавляющий» к функции переменные замыкания

production function

```
def MakeLogPrint():
    n=0
    def LogPrint(msg):      # вложенная функция
        nonlocal n
        n+=1
        print(' log#',n,'>',msg)
    return LogPrint         # вернули ф-ю вместе с замыканием

lp=MakeLogPrint()           # create closure
lp(15)                      # log# 1 > 15
del MakeLogPrint            # delete production function
lp('It works after del')    # log# 2 > It works after del
```

Декораторы

- ✓ Способ добавить функциональность к уже существующей функции
- ✓ Декоратор – функция, которая получает функцию как аргумент, добавляет новые свойства и возвращает новую функцию

Пример: рекурсивная функция для чисел Фибоначчи

```
def fib(n):
    if n < 2:
        return 1
    else:
        return fib(n-1)+fib(n-2)
print( fib(30) )   # 1346269 time ~0.5s
```

- ✾ Если добавить сохранение результатов функции, кеширование, то для `fib()` «тормоз» исчезнет

кеширование вызовов функции

```
def cache(f):   # this function is a decorator for function f
    save={}     # dictionary for caching results
    def cachef(x):          # function closure
        if x not in save:
            save[x] = f(x)
        return save[x]
    return cachef
fib=cache(fib)   # function 'fib' has decorated
print(fib(100))  # 573147844013817084101 time ~0.05s
```

Символ @:

- ✓ Запись: `@name_of_decorator` находящаяся над определением функции означает, что функция будет декорирована:

```
@cache       #
def fib(n):  #  тоже самое, что fib=cache(fib)
    ...
```

Дополнительные слайды