Язык программирования

Лекция № 6

Владимир Владимирович Руцкий
rutsky.vladimir@gmail.com

План занятия

- Классы в Python. Повторение
- Итераторы
- Генераторы
- Декораторы
- Практика

Классы в Python. **Повторение**

Итераторы

- Существует много различных контейнеров (list, tuple, str, bytes, array)
- Частая операция: пройтись по всем элементам контейнера и сделать что-то с каждым элементом
- Для унификации доступа придуман **итератор** — объект, абстрагирующий последовательный доступ к элементам контейнера
  - Контейнер предоставляет метод "создать_объект-итератор()"
  - Объект-итератор имеет метод "получить_следующий_элемент()"

Итераторы в Python

- В Python метод `__iter__()` у контейнеров возвращает объект-итератор
- Итератор имеет метод `__next__()` для получения следующего значения из контейнера
- Если значений больше нет, `__next__()` бросает исключение `StopIteration`

Итераторы в Python. **Пример**

```
>>> a = ['A', 'B', 'C']
>>> a.__iter__
<method-wrapper '__iter__' of list object at 0x...>
>>> it = a.__iter__()
>>> it.__next__()
'A'
>>> it.__next__()
'B'
>>> it.__next__()
'C'
>>> it.__next__()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

Цикл `for`

- Цикл `for` работает используя итераторы:

```
for ELEM in CONTAINER:
    PROCESS(ELEM)
```

эквивалентно:

```
it = CONTAINER.__iter__()
while True:
    try:
        ELEM = it.__next__()
    except StopIteration:
        break

    PROCESS(ELEM)
```

- Замечание: `for` также поддерживает протокол последовательности (`sequence protocol`)

Написание итераторов

```
>>> class MyRange:
...     def __init__(self, stop):
...         self._stop = stop
...         self._next = 0
...     def __iter__(self):
...         return self
...     def __next__(self):
...         if self._next >= self._stop:
...             raise StopIteration
...         else:
...             result, self._next = self._next, self._next + 1
...             return result
...
>>> for i in MyRange(10):
...     print(i, end=" ")
...
0 1 2 3 4 5 6 7 8 9
>>>
```

Передача последовательностей

- Большинство функций работает с контейнерами через итераторы:

```
>>> list(MyRange(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(zip(MyRange(3), MyRange(5)))
[(0, 0), (1, 1), (2, 2)]
```

- *Интерфейс* итератора скрывает детали того, как именно *итерироваться* (проходить) по элементам
- Многие функции возвращают объекты поддерживающие интерфейс итераторов:

```
>>> res = zip(range(10), ['A', 'B'])
>>> it = res.__iter__()
>>> it.__next__()
(0, 'A')
>>>
```

`__iter__()` у итераторов

- Замечание: итераторы также имеют метод __iter__(), который возвращает самого себя
  - В функцию можно передать как контейнер, так и итератор
  - Функция может вернуть как контейнер, так и итератор
- По умолчанию стоит считать, что функция принимает/возвращает контейнер, но работать с ним через интерфейс итератора

Функция `iter()`

- `iter(object[, sentinel])` — обёртка для получения итератора из контейнера
- С одним аргументом возвращает результат `object.__iter__()`
- С двумя аргументами возвращает итератор, который вызывает `object`, пока он не вернёт `sentinel`:

```
f = open('mydata.txt')
for line in iter(f.readline, ''):
    process_line(line)
```

Функция `next()`

- `next(iterator[, default])` — возвращает следующий элемент итератора (вызывая `__next__()`)
  - Если указан default, то когда итератор завершится (бросит исключение StopIteration), будет возвращаться default

Пример

```
>>> r = range(3) # возвращает объект, поддерживающий интерфейс итератора
>>> it = iter(r)
>>> next(it)
0
>>> next(it)
1
>>> next(it)
2
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>> it = iter("ABC")
>>> next(it, "Done!")
'A'
>>> next(it, "Done!")
'B'
>>> next(it, "Done!")
'C'
>>> next(it, "Done!")
'Done!'
>>> next(it, "Done!")
'Done!'
>>>
```

Возможности итераторов

- Интерфейс итератора абстрагирует доступ к элементам контейнера
- Позволяет создавать "виртуальные" последовательности
  - Не обязательно хранить все элементы, если можно создавать их "на лету"

Реализация собственных итераторов через реализацию `__iter__()`, `__next__()` довольно трудоёмкий процесс, можно прощё!

Генераторы

```
>>> def my_generator(n):
...     for i, ch in zip(range(n), "ABCDEFGHIK"):
...         # Если в функции встречается команда yield, то функция — генератор
...         yield ch + str(i)
...
>>> # Генераторы возвращают итератор
... it = my_generator(3)
>>> it
<generator object my_generator at 0x...>
>>> # При вызове генератора код тела функции не выполняется.
... # При попытке получить следующее значение итератора функция выполнится до
... # первого yield
... next(it)
'A0'
>>> # При запросе следующего значения, выполнение функции продолжится
... # с места, где она остановилась на yield в последний раз.  Состояние
... # переменных сохраняется, как будто функция и не прерывалась.
... next(it)
'B1'
>>> next(it)
'C2'
>>> next(it) # Когда тело функции заканчивается, итератор считается исчерпанным
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

Пример

```
>>> # Генераторы очень удобны для создания последовательностей
... # "на лету"
... def my_range(start, stop=None, step=1):
...     if stop is None:
...         start, stop = 0, start
...     while start < stop:
...         yield start
...         start += step
...
>>> list(my_range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(my_range(3, 5))
[3, 4]
>>> list(my_range(3, 10, 2))
[3, 5, 7, 9]
>>>
```

yield from

```
>>> # itertools.chain позволяет объединять последовательности
... import itertools
>>> list(itertools.chain(range(5), "ABC", [4, 3, 2, 1, 0]))
[0, 1, 2, 3, 4, 'A', 'B', 'C', 4, 3, 2, 1, 0]
>>> def my_chain(*iterables):
...     for iterable in iterables:
...         for elem in iterable:
...             yield elem
...
>>> list(my_chain("abc", range(4)))
['a', 'b', 'c', 0, 1, 2, 3]
>>> def my_chain2(*iterables):
...     for iterable in iterables:
...         # В "yield from" генератор будет возвращать значения напрямую из
...         # итератора iterable, пока в нём не закончатся элементы.
...         yield from iterable
...
>>> list(my_chain2("abc", range(4)))
['a', 'b', 'c', 0, 1, 2, 3]
>>>
```

Передача значений в `yield`

```
>>> def echo(value=None):
...     try:
...         while True:
...             try:
...                 # Команда yield возвращает значение, которое может быть
...                 # передано через iterator.send()
...                 value = (yield value)
...             except Exception as e:
...                 value = e
...     except GeneratorExit:
...         print("У итератора вызвали close() — запрос на отмену итерирования")
...
>>> it = echo(1)  # Передаём начальный value=1
>>> print(next(it))
1
>>> print(next(it))  # По умолчанию результат yield — None
None
>>> # Результат yield можно указать, использовав send() вместо __next__()
... print(it.send(2))
2
>>> # Можно вызвать исключение внутри генератора (в месте yield):
... it.throw(TypeError, "spam")
TypeError('spam',)
>>> # Можно запросить остановку итератора (вызывается при уничтожении объекта,
... # который не закончил итерирование):
... print(it.close())
У итератора вызвали close() — запрос на отмену итерирования
None
>>>
```

Выражения генераторы

```
>>> def f(arg, dummy_arg=None):
...     print(type(arg))
...     for i in arg:
...         print(i, end=" ")
...
>>> # Comprehensions, используемые сами по себе, создают контейнеры
... f([x ** 2 for x in range(5)], "test") # будет создан список и передан в f()
<class 'list'>
0 1 4 9 16
>>> # Можно написать генерацию последовательности без скобок контейнера,
... # тогда будет создан генератор:
... f(x ** 2 for x in range(5))
<class 'generator'>
0 1 4 9 16
>>> # Или можно написать круглые скобки:
... f((x ** 2 for x in range(5)), "test")
<class 'generator'>
0 1 4 9 16
>>>
```

Возможности генераторов

- Генераторы позволяют создавать **сопрограммы** (coroutines) — обобщённая подпрограмма, которая имеет несколько точек входа, возможность остановки выполнения и возможность продолжения выполнения из того же состояния
- Возможность остановить работу функции и затем продолжить её позволяет делать различные трюки
  - Фреймворк Twisted на генераторах реализует асинхронное программирование

Декораторы

- При создании функции можно сделать "постобработку" полученной функции с помощью **декоратора**
- Синтаксис:

```
@my_decorator
def func():
    pass
```

эквивалентно:

```
def func():
    pass

func = my_decorator(func)
```

Пример

```
>>> import datetime
>>> import time
>>> def print_time(func):
...     def wrapped_func(*args, **kwargs):
...         # Обёртка засекает время и вызывает оборачиваемую функцию.
...         start_time = datetime.datetime.now()
...         try:
...             return func(*args, **kwargs)
...         finally:
...             # По завершению работы функции обёртка выводит время работы
...             # функции.
...             end_time = datetime.datetime.now()
...             num_secs = (end_time - start_time).total_seconds()
...             print("Took {} seconds".format(num_secs))
...     # Оборачиваемая функция будет заменена возвращаемой декоратором функцией
...     return wrapped_func
...
>>> @print_time
... def long_running_func():
...     for i in range(10000):
...         pass
...     return 123
>>> long_running_func()
Took 0.000243 seconds
123
>>>
```

Комбинирование декораторов

- Можно использовать несколько декораторов и передавать аргументы в декораторы:

```
@dec1
@dec2(1, 2)
@dec3()
def func():
    pass
```

эквивалентно:

```
def func():
    pass

func = dec3()(func)
func = dec2(1, 2)(func)
func = dec1(func)
```

Практика