Python

Лекция 3

Преподаватель: *Дмитрий Косицин*

BSU FAMCS (Fall'20)

Итераторы и генераторы

Последовательности. Итерируемые объекты. Итераторы. Генераторы. Дополнительные способы итерирования.

Sequence and iterable

Последовательность (*sequence*) – упорядоченный *индексируемый* набор объектов, например, **list**, **tuple** и **str**.

У этих объектов переопределены «магические методы» ***__len__*** (длина последовательности) и ***__getitem__*** (отвечает за индексацию).

Итерируемое (*iterable*) – упорядоченный набор объектов, элементы которого можно получать по одному.

У таких объектов реализован метод ***__iter__*** – возвращает итератор, который позволяет обойти итерируемый объект.

Итераторы

Итератор (iterator) представляет собой «поток данных» – он позволяет обойти все элементы *итерируемого* объекта, возвращая их в некоторой последовательности.

В итераторе переопределен метод ***__next__*** (***next*** в Python 2), вызов которого либо возвращает следующий объект, либо бросает исключение ***StopIteration***, если все объекты закончились.

Для явного получения итератора и взятия следующего элемента используются *built-in* методы **iter** и **next**.

```
for item in sequence:
    action(item)
```

```
def for_sequence(sequence, action):  # "for" for sequence
    i, length = 0, len(sequence)
    while i < length:
        item = sequence[i]
        action(item)
        i += 1
```

```
def for_iterable(iterable, action):  # "for" for iterator
    iterator = iter(iterable)
    try:
        while True:
            item = next(iterator)
            action(item)
    except StopIteration:
        pass
```

Итераторы

Итераторы представляют собой классы, содержащие информацию о текущем состоянии итерирования по объекту (например, индекс).

После обхода всех элементов итератор «истощается» (exhausted), бросая исключение ***StopIteration*** при каждом следующем вызове ***__next__***.

Замечание. Функция ***next*** имеет второй параметр – значение по умолчанию, которое будет возвращено, когда итератор исчерпается.

Замечание. У функции ***iter*** также есть второй аргумент – значение, до *получения* которого будет продолжаться итерирование.

Пример реализации итератора

```
class RangeIterator(collections.Iterator):
    def __init__(self, start, stop=None, step=1):
        self._start = start if stop is not None else 0
        self._stop = stop if stop is not None else start
        self._step = step  # positive only

        self._current = self._start

    def __next__(self):
        if self._current >= self._stop:
            raise StopIteration()

        result = self._current
        self._current += self._step
        return result
```

Пример использования итератора

Поскольку итераторы хранят информацию о состоянии, их можно прервать и впоследствии продолжить итерироваться. *Вопрос: что выведет следующий код?*

```
>>> odd_indices_iterator = RangeIterator(1, 10, 2)
>>>
>>> for idx in odd_indices_iterator:
>>>     if idx > 5:
>>>         break
>>>     print(idx)
>>>
>>> for idx in odd_indices_iterator:
>>>     print(idx)
```

Итерируемые и истощаемые

Последовательности *итерируемы* и *не истощаемы* (можно много раз итерироваться по ним).

Итерируемые объекты (не последовательности) могут как не истощаться (**range** в Py3/**xrange** в Py2), так и истощаться (генераторы).

Итераторы *итерируемы* (возвращают сами себя) и *истощаемы* (можно только один раз обойти).

Замечание. Зачастую в классах не реализуют отдельный класс-итератор. В таком случае метод ***__iter__*** возвращает *генератор*.

Замечание. В Python 2 **range** возвращает список, а **xrange** – генератор.

Пример итерируемого объекта

```
class SomeSequence(collections.Iterable):
    def __init__(self, *items):
        self._items = items

    def __iter__(self):
        for item in self._items:
            yield item

    def __iter__(self):
        yield from self._items  # только в Python 3.3+

    def __iter__(self):  # простой и менее гибкий вариант
        return iter(self._items)
```

Напоминание. В модуле **collections** есть и другие базовые классы, например **Sequence**. Эти классы реализуют множество полезных методов, требуя переопределить лишь несколько.

Генератор

Генератор – итератор, с которым можно взаимодействовать (PEP-255).

Каждый следующий объект возвращается с помощью выражения **yield**. Это выражение *приостанавливает* работу генератора и передает значение в вызывающую функцию. При повторном вызове исполнение продолжается с *текущей* позиции либо до следующего **yield**, либо до конца функции.

Генераторы удобно использовать, когда вся последовательность сразу не нужна, а нужно лишь по ней итерироваться.

```
>>> assert all(x % 2 for x in range(1, 10, 2))
```

Замечание. Выражения-генераторы имеют вид comprehensions с круглыми скобками. При передаче в функцию дополнительные круглые скобки не нужны.

Замечания по генераторам

Конструкция **yield from** делегирует, по сути, исполнение некоторому другому итератору (Python 3.3+, PEP-380).

В Python 3 появилась возможность у генераторов (например, **range**) узнать длину генерируемой ими последовательности (метод ***__len__***) и проверить, генерируют ли они определенный элемент (метод ***__contains__***).

В Python 2 ввиду реализации **xrange** не принимает числа типа **long**.

Также в Python 3 есть специальный класс – **collections.ChainMap**, который представляет собой обертку над несколькими mapping'ами.

Дополнительные способы итерирования

В стандартной библиотеке есть модуль **itertools**, в котором реализовано множество итераторов:

- **cycle** – зацикливает некоторый iterable
- **count** – бесконечный счетчик с заданным начальным значением и шагом
- **repeat** – возвращает некоторое значение заданное число раз

Также есть комбинаторные итераторы:

- **product** – итератор по декартову произведению последовательностей (по сути, генерирует кортежи, если бы был реализован вложенный *for*)
- **combinations** – итератор по упорядоченным сочетаниям элементов
- **permutations** – итератор по перестановкам переданных элементов

Дополнительные способы итерирования

- **chain** – итерируется последовательно по нескольким iterable
- **zip_longest** – аналог zip, только прекращает итерироваться, когда исчерпывается не первый, а последний итератор
- **takewhile/dropwhile/filterfalse/compress** – отбирает элементы последовательности в соответствии с предикатом
- **islice** – итераторный аналог slice (не создает списка элементов)
- **groupby** – группирует последовательные элементы
- **starmap** – аналог **map**, только распаковывает аргумент при передаче
- **tee** – создает ***n*** копий итератора

Замечание. В Python 2 доступны **ifilter** и **izip** – итераторные аналоги **filter** и **zip**.

Замечание. В Python 3.2 появилась функция **accumulate**, которая возвращает итератор по кумулятивному массиву.

Менеджеры контекста

...

Менеджеры контекста

В процессе работы с файлами важно корректно работать с исключениями: файл необходимо закрыть в любом случае.

Данный синтаксис позволяет закрыть файл по выходе из блока **with**:

```
with open(file_name) as f:
    # some actions
```

Функция **open** возвращает специальный объект – *context manager*. Менеджер контекста последовательно *инициализирует* контекст, *входит* в него и корректно обрабатывает *выход*.

Пример менеджера контекста

```
class ContextManager(object):
    def __init__(self):
        print('__init__()')

    def __enter__(self):
        print('__enter__()')
        return 'some data'

    def __exit__(self, exc_type, exc_val, exc_tb):
        print('__exit__({}, {})'.format(
            exc_type.__name__, exc_val))


with ContextManager() as c:
    print('inside context "%s"' % c)
```

Менеджер контекста

Менеджер контекста работает следующим образом:

- создается и инициализируется (метод **__init__**)
- организуется вход в контекст (метод **__enter__**) и возвращается объект контекста (в примере с файлом – объект типа **file**)
- выполняются действия внутри контекста (внутри блока **with**)
- организуется выход из контекста с возможной обработкой исключений (метод **__exit__**)

В примере будет выведено следующее:

```
__init__()
__enter__()
inside context "some data"
__exit__(None, None)
```

Менеджер контекста

Замечание. Если исключения не произошло, то параметры, передаваемые в функцию **__exit__** – тип, значение исключения и ***traceback*** – имеют значения **None**.

Замечание. Менеджер контекста, реализуемый функцией **open**, по выходе из контекста просто вызывает метод ***close*** (см. декоратор *contextlib.closing*).

Менеджеры контекста используются:

- для корректной, более простой и переносимой обработки исключений в некотором блоке кода
- Для управления ресурсами

Декоратор *contextlib.contextmanager* позволяет создать менеджер контекста из функции-генератора, что значительно упрощает синтаксис.

Менеджер контекста из генератора

```
>>> @contextlib.contextmanager
>>> def get_context():
>>>     print('__enter__()’)
>>>     try:
>>>         yield 'some data’
>>>     finally:
>>>         print('__exit__()’)
>>>
>>> with get_context() as c:
>>>     print('inside context "%s"' % c)
__enter__()
inside context "some data"
__exit__()
```

Классы. Финальные замечания

Механизм создания классов

Определение класса приводит к следующим действиям:

1. Определяется подходящий ***метакласс*** (класс, который создает другие классы)
2. Подготавливается namespace класса
3. Выполняется тело класса
4. Создается объект класса и присваивается переменной

```
class X(object):
    a = 0

# equivalent: type(name, bases, namespace)
X = type('X', (object, ), {'a': 0})
```

Замечания по созданию классов

Метаклассом по умолчанию является **type**.

Выполнение тела класса приводит к созданию ***словаря*** всех его атрибутов, который передается в **type**. Далее этот словарь доступен через **__dict__** или с помощью built-in функции **vars**.

Замечание. Изменять, добавлять и удалять атрибуты *можно*, модифицируя **__dict__**. Данный способ **менее явный**, нежели использование **getattr** и пр.

Важно! Атрибуты классов при наследовании не перезаписываются, а поиск их происходит последовательно в словарях базовых классов.

Замечания по созданию классов

Вопрос: есть ли разница между реализацией синонима (alias) для функции (функции g и h в примере)?

```
class X(object):
    def f(self):
        return 0

    def g(self):
        return self.f()

    h = f
```

Обычно реализация синонимов необходима при реализации операторов.

Произвольный код в теле класса

Код в модуле выполняется подобно коду телу класса. Неудивительно, ведь модуль – тоже класс! Значит, в теле класса можно писать любые синтаксически корректные конструкции!

```
class C(object):
    if sys.version_info.major == 3:
        def f(self):
            return 1
    else:
        def g(self):
            return 2
```

Замечание. В Python 3 порядок объявления атрибутов сохраняется (PEP-520).

Abstract base classes

В Python есть возможность создавать условные интерфейсы и абстрактные классы. Для этого используется метакласс **ABCMeta** (в Python 3.4 – базовый класс **ABC**) из модуля **abc** (PEP-3119).

Для объявления абстрактного метода используется декоратор **abstractmethod**, абстрактного свойства – **abstractproperty**.

В Python иерархия типов введена для чисел – модуль **numbers** PEP-3141, а также коллекций и функционалов – модуль **collections.abc**.

Создание экземпляра класса

Создание экземпляра класса заключается в вызове метода __ **new** __ для получения объекта класса и метода **__init__** для его инициализации.

```
class C(object):
    def __new__(cls, name):
        return super().__new__(cls)   # make a new class

    def __init__(self, name):
        self.name = name

c = C('class')
```

Создание экземпляра класса

Сигнатура метода **__new__** совпадает с сигнатурой **__init__**.

В методе **__new__** можно возвращать объект ***другого*** класса, модифицировать и присваивать атрибуты!

Метод **__init__** не вызывается автоматически, если **__new__** возвращает объект другого класса.

Метаклассы

```
class Meta(type):
    def __new__(mcs, name, bases, attrs, **kwargs):
        # invoked to create class C itself
        return super().__new__(mcs, name, bases, attrs)

    def __init__(cls, name, bases, attrs, **kwargs):
        # invoked to init class C itself
        return super().__init__(name, bases, attrs)

    def __call__(cls):
        # invoked to create an instance of C
        # -> call __new__ and __init__ inside
        # Note: __call__ must share the signature
        #   with class' __new__ and __init__ method signatures
        return super().__call__()
```

Метаклассы

```
1   class C(metaclass=Meta):
2       def __new__(cls):
3           return super().__new__(cls)
4
5       def __init__(self):
6           pass
7
8   c = C()
```

Строка 1: вызываются методы **__new__** и **__init__** метакласса *Meta* (создается объект – класс).

Строка 8: вызывается метод **__call__** метакласса *Meta,* который вызывает методы **__new__** и **__init__** класса *C*.

Метаклассы

Методы **__new__** и **__init__** метакласса принимают **kwargs – ключевые аргументы. Они используются для настройки класса – вызова метода **__prepare__**, который возвращает *mapping* для сохранения атрибутов класса (см. PEP-3115).

Примером метакласса в стандартной библиотеке является Enum (Py 3.4+).

Замечание. В Python 3.6 появился метод **__init_subclass__** (PEP-487), позволяющий изменить создание классов наследников (например, добавить атрибуты).

В классе присутствуют специальный атрибут **__bases__** (кортеж базовых классов) и функция **__subclasses__**, возвращающая список подклассов.

Замечания о классах

В Python 3.6. можно переопределить метод **__set_name__**(self, owner, name) у дескрипторов для получения имени *name*, под которым дескриптор сохраняется в классе *owner*.

Замечание. При реализации **__getattribute__** в некоторых случаях требуется принимать во внимание дескрипторы.

В классах допустимы некоторые атрибуты, характеризующие класс:

- **__slots__** (используется вместо **__dict__**)
- **__annotations__** (аннотации типов: PEP-318, PEP-481, PEP-3107)
- **__weakref__** («слабые ссылки», docs, PEP-205).

Дескрипторы

Свойства (**property**) и декораторы **staticmethod** и **classmethod** являются *дескрипторами* – специальными объектами, реализованными как атрибуты класса (непосредственного или одного из родителей).

В классах в зависимости от типа дескриптора реализуются методы:

- **__get__**(self, instance, owner) # *owner* – instance class / type
- **__set__** (self, instance, value) # *self* – объект дескриптора
- **__delete__**(self, instance) # *instance* – объект, в котором вызывается дескриптор

Стандартное поведение дескрипторов заключается в работе со словарями объекта, класса или базовых классов.

Пример. Вызов **a.x** приводит к вызову **a.__dict__['x']**, потом **type(a).__dict__['x']** и далее по цепочке наследования.

Пример дескриптора

```
class Descriptor(object):
    def __init__(self, label):
        self.label = label

    def __get__(self, instance, owner):
        return instance.__dict__.get(self.label)

    def __set__(self, instance, value):
        instance.__dict__[self.label] = value

class C(object):
    x = Descriptor('x')

c = C()
c.x = 5
print(c.x)
```

Дескрипторы

Методы и свойства в классе являются дескрипторами. По сути, в каждой функции (неявно) есть метод **__get__**:

```
class Function(object):
    def __get__(self, obj, objtype=None):
        "Simulate func_descr_get() in Objects/funcobject.c"
        return types.MethodType(self, obj, objtype)
```

Декораторы **classmethod** и **staticmethod** модифицируют аргументы вызова:

Transformation	Called from an Object	Called from a Class
function	f(obj, *args)	f(*args)
staticmethod	f(*args)	f(*args)
classmethod	f(type(obj), *args)	f(klass, *args)

Дескрипторы

Метод – объект-функция, который хранится в словаре атрибутов класса. Доступ же обеспечивается с помощью механизма дескрипторов (см. пример, пример).

```
>>> class D(object):
...     def f(self, x):
...         return x
...
>>> d = D()
>>> D.__dict__['f']  # Stored internally as a function
<function f at 0x00C45070>
>>> D.f  # Get from a class becomes an unbound method
<unbound method D.f>
>>> d.f  # Get from an instance becomes a bound method
<bound method D.f of <__main__.D object at 0x00B18C90>>
```

Математические библиотеки и работа с данными

Numpy. SciPy. Sympy. Matplotlib и Seaborn. Pandas.

Numpy

В библиотеке Numpy реализован класс **ndarray** – представление многомерного массива.

Он характеризуется данными (data) и информацией о данных:

- Тип данных и его размер
- Смещение данных в буфере
- Размерности (shape) и размер в байтах
- Количество элементов для перехода к следующему элементу в измерении (по оси – stride)
- Порядок байтов в массиве
- Флаги буфера данных
- Ориентация данных (C-order или Fortran-order)

NDArray

NDArray соответсвует буферу – С-массиву, выровненному по размеру элемента (**itemsize**).

К отдельным элементам массива можно обращаться с помощью методов **item/itemset** – это быстрее, чем по индексу через **__getitem__**.

Очень важно! Все операции с массивом могут быть как с копированием данных, так и без. Некоторые операции возвращают *views* – новые массивы, которые указывают ***на те же данные***, но содержат о них иную информацию. При изменении данных во *view*, данные в оригинальном массиве меняются.

Размерность массива

Форма массива задается атрибутом **shape** – кортеж размерностей.

Изменить размер можно:

- ndarray.reshape() – *view*, но shape обязан быть *compatible* с текущим
- ndarray.resize() – *inplace*, может понадобиться копирование
- ndarray.shape – *inplace*, исключение, если не *compatible*

Разворачивание массива в 1-D:

- ndarray.ravel – *view*
- ndarray.flatten – *copy*
- ndarray.flat – *итератор* по flattened массиву

Замечание. Допустим 0-D массив.

Оси

Оси – составляющие общей размерности массива.

Важно! Нумерация осей (axes) ведется с нуля и соответствует декартовым координатам. Значение **None** в функциях соответствует развернутому массиву.

Изменение осей не приводит к копированию данных.

Методы **transpose** и **swapaxis** позволяют изменить порядок следования осей.

Важно! Эти методы могут сделать отображение данных не непрерывным.

Индексация

Numpy поддерживает два вида индексации: простую и «продвинутую».

Простая индексация – один элемент или слайс:

```
>>> x = numpy.arange(10)
>>> x[1], x[-2], x[3: 7]
```

В многомерном массиве элементы задаются через запятую:

```
>>> x = numpy.arange(100).reshape(5, 5, 4)
>>> x[1, 2, 3], x[1, 1:, -1], x[…, 2] == x[:, :, 2]
```

Замечание. Многоточие '…' – **Ellipsis** позволяет пропустить некоторые измерения, предполагая, что их нужно взять целиком.

Замечание. В numpy применяется index broadcasting: если массив по одной из осей имеет длину 1, он расширяется до необходимой длины (например, можно сложить с числом).

Продвинутая индексация

Очень важно! Запись x[ind_1, …, ind_n] эквивалентна x[(ind_1, …, ind_n)], но кардинально отличается от x[[ind_1, …, ind_n]].

Если в метод **__getitem__** передан ***список*** или ***ndarray***, то используется «продвинутая» индексация: она создает копию массива с указанными элементами.

```
>>> x = numpy.arange(4)
>>> x[[1, 3]] == x[[False, True, False, True]]
```

Список, переданный в качестве индекса, может содержать как индексы, так и массив **bool**, означающий, какие элементы нужно взять.

Замечание. Массив **bool** может иметь длину меньше, чем длина исходного массива. Если длина больше и есть **True** за пределами массива, будет исключение.

Универсальные функции

В numpy реализовано множество функций по работе с данными – сложение, умножение, вычисление синуса и т.п. Все операции над массивами выполняются **поэлементно**!

```
>>> x, y = numpy.arange(3), numpy.linspace(2, 3, num=3, endpoint=True)
>>> x + y == numpy.array([2, 3.5, 5])
>>> numpy.max(x) == x.max()  # 2
```

Универсальные функции (ufunc) выполняются с учетом расположения данных. Во время прохода также используется буферизация! Поэтому они работают быстрее Python built-in. На небольших данных ufunc работают медленнее, ввиду необходимости настройки.

Типы и записи

В numpy используется своя система типов, в частности, numpy.bool отличается от Python built-in bool.

В массивах помимо скалярных типов можно хранить произвольные типы dtype – по сути, C-структуры, содержащие некоторые скалярные типы.

```
>>> dt = np.dtype([('name', np.str_, 16),
                   ('grades', np.float64, (2,))])
>>> dt = np.dtype({'names': ['r','g','b','a'],
                   'formats': [uint8, uint8, uint8, uint8]})
```

В dtypes указаны имена – массивы с такими типами будут являться записями (numpy.recarray), так что к столбцам можно будет обращаться по имени.

Возможности Numpy

В Numpy есть поддержка массивов с пропусками – **MaskedArray**. Значения в таким массивах могут иметь специальное значение **numpy.ma.masked**.

```
>>> x = numpy.array([1, 2, 3, -1, 5])
>>> mx = numpy.ma.masked_array(x, mask=[0, 0, 0, 1, 0])
>>> mx.mean()  # 2.75
```

Такие значения обрабатываются функциями и не влияют на результат.

Работа с произвольными Python-функциями может быть организована:

- **apply_along_axis** (**apply_over_axes**) – применяет функцию вдоль оси (осей)
- **vectorize** – обобщенный класс функций (можно использовать как декоратор, реализован как **for**)
- **frompyfunc** – позволяет создать ufunc из обычной функции

Возможности Numpy

Также в numpy реализована работа

- с матрицами (Matrix)
- со случайными величинами (random)
- со статистическими функциями
- допустима стыковка массивов

SciPy

В тот время как Numpy реализует определенные типы данных и базовые операции, SciPy реализует множество вспомогательных функций:

- Clustering – реализация kmeans и пр.
- Constants – множество констант, математических и физических
- FFTPack – функции, выполняющие дискретное преобразование Фурье
- Integrate – интегрирование (квадратурные формулы, с фиксированной сеткой) и дифференцирование (Рунге-Кутта и пр.)
- IO – работа с форматами данных, в т.ч. MatLab
- Linalg – линейная алгебра (решение СЛАУ, разложения матриц, нахождение собственных значений, матричные функции, специальные виды матриц и пр.)
- NDImage – функции по работе с изображениями (свертки, аффинные преобразования и т.п.)

SciPy

- ODR – orthogonal distance regression
- Optimize – оптимизация (методы первого и второго порядка, метод Ньютона, BGFS, Conjugate Gradient и пр., поиск корней, МНК)
- Signal – методы обработки сигналов (звука – свертки, сплайны, фильтры)
- Sparse и Sparse.Linalg – разреженные матрицы и методы по работе с ними
- CSGraph – методы по работе со сжатыми разреженными графами (в т.ч. алгоритмы Дейкстры, BFS, DFS и т.п.)
- Spatial – работа с точками на плоскости (KDTree и пр.)
- Stats и Stats.Mstats – работа с распределениями и статистиками

Matplotlib, Seaborn, SymPy, Pandas

Для отображения данных используется библиотеки **MatplotLib** и **Seaborn**. Работать с ней так:

- Открыть MPL tutorials (или reference) или SBS tutorials (или API)
- Найти нужный пример
- Модифицировать его под свои нужды

Библиотека **Sympy** позволяет производить символьные вычисления, а **Pandas** – работать с данными (более удобная обертка над **NumPy**, ориентированная на группировку / преобразование данных и статистики).

Полезно: https://github.com/jrjohansson/scientific-python-lectures