Python Underground

Лекция 9

Преподаватель: *Дмитрий Косицин*

Математические библиотеки и работа с данными

Numpy. SciPy. Sympy. Matplotlib. Pandas.

Классы. Финальные замечания

Python. Финальные замечания

Математические библиотеки и работа с данными

Numpy. SciPy. Sympy. Matplotlib. 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(3)
>>> x[[1, 2]] == x[[False, True, 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)
- 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, SymPy, Pandas

Для отображения данных используется библиотека **MatplotLib**. Работать с ней так:

- Открыть tutorials (или обратиться к reference)
- Найти нужный пример
- Модифицировать его под свои нужды

Библиотека **Sympy** позволяет производить символьные вычисления, а **Pandas** – работать с данными (более удобная обертка над **NumPy**).

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

Классы. Финальные замечания

...

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).

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

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

Abstract base class

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**.

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).

```
>>> 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>>
```

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 параметров, PEP-318, PEP-481)
- **__weakref__** («слабые ссылки», docs, PEP-205).

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.

Python. Финальные замечания

Хотелось рассказать

- Реализация интерпретатора
- AST (docs) и Bytecode (dis module, article)
- Интроспекция (модуль inspect)
- Ускорение (Nuitka, PyPy; Cython, Numba)
- Global interpreter lock (presentation)
- Garbage collection and weak references (docs, docs, article)
- Multiprocessing and multithreading (incl. concurrent, subprocess and signal)
- Binary data and object pickling (pickle, bson, dill)
- Coroutines and asynchronous coroutines (docs)
- Type annotations and type hints (e.g. PyCharm IDE docs)

Общие замечания

В Python 3 у генератора **range** переопределен метод **__contains__**. Он может быть переопределен при реализации собственных генераторов.

Выражение **yield from** позволяет генератору передать управление другому генератору (Python 3).

Помимо декорирования функций допустимо декорирование классов (например, **functools.total_ordering**).

В Python 3 порядок следования именованных аргументов (keyword arguments – ****kwargs**) сохраняется.

В Python есть собственный интерактивный отладчик – **pdb** (используется многими IDE).

Общие замечания

Если класс переопределяет **__eq__**, то для метода **__hash__** должно выполняться:

- **__hash__** явно реализован
- явно присвоено **__hash__** = **None** (для изменяемых объектов-коллекций)
- явно присвоено **__hash__** =<ParentClass>.**__hash__** (если не изменен)

Метод **__eq__** должен либо бросать **TypeError**, либо возвращать **NotImplemented**, если передан объект некорректного для сравнения типа.

Профилирование:

- Использование памяти: memory_profiler и др.
- Визуализация профилирования: SnakeViz
- Прочие инструменты: ссылка

Возможности Python

Python позволяет:

- Писать утилиты (работа с файловой системой, простая обработка данных) и прототипы программ
- Работать с данными и производить вычисления
- Реализовывать высокоуровневые интерфейсы (web frameworks)

Python является удобной оберткой над многими низкоуровневыми штуками: позволяет их единообразно использовать и не думать о работе с памятью.

Python Zen

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Readability counts.
Special cases aren't special enough to break the rules.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
…

Spider-Man rule: With Great Power Comes Great Responsibility!

The End

Thank you!