Muslimov Arthur

(since 18.01.20 )

*Введение в библиотеку NumPy*

На удивление, почти всё можно хранить в массивах чисел. Допустим, изображения – это матрица, числа которой отражают яркость пикселов (если картинка чёрно-белая). Вне зависимости от характера данных, первым шагом к их анализу является преобразование в числовые массивы. Тут вам и представляется NumPy. Библиотека NumPy (Numerical Python) даёт массивы, похожие на обычные массивы Python’a, но обеспечивают более эффективное хранение и операции с данными при росте размера массивов. Думаю, что ты сам разберёшься как его установить.

По традиции, при работе с NumPy принято использовать его сокращение np:

import numpy as np

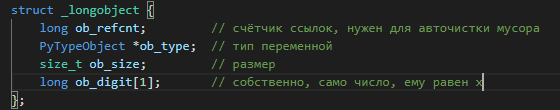
Напоминаю, что в Ipython есть такие штуки, как Tab и ?. Ты знаешь что делать.

*Работа с типами данных в языке Python*

Чтобы работать с данными, следует знать как они хранятся и обрабатываются. Язык Python поддерживает динамическую типизацию, т.е. не надо объявлять переменные, они сами адаптируются под значение. Это значит, что переменные в Python - это что-то больше, чем просто значения. Как говорит автор: «Понимать, как это работает, очень важно, чтобы научиться эффективно анализировать данные с помощью Python».

*Целое число в языке Python – больше, чем просто целое число*

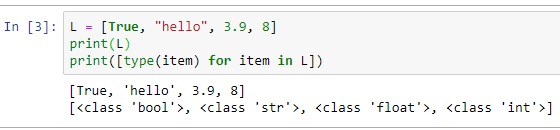
Стандартная реализация Python написана на Си, т.е. объекты Python’a – это структуры Си. Например, x = 10000. x здесь – не простое число, а указатель на структуру, содержащую значение. Давайте узнаем, что именно она хранит:



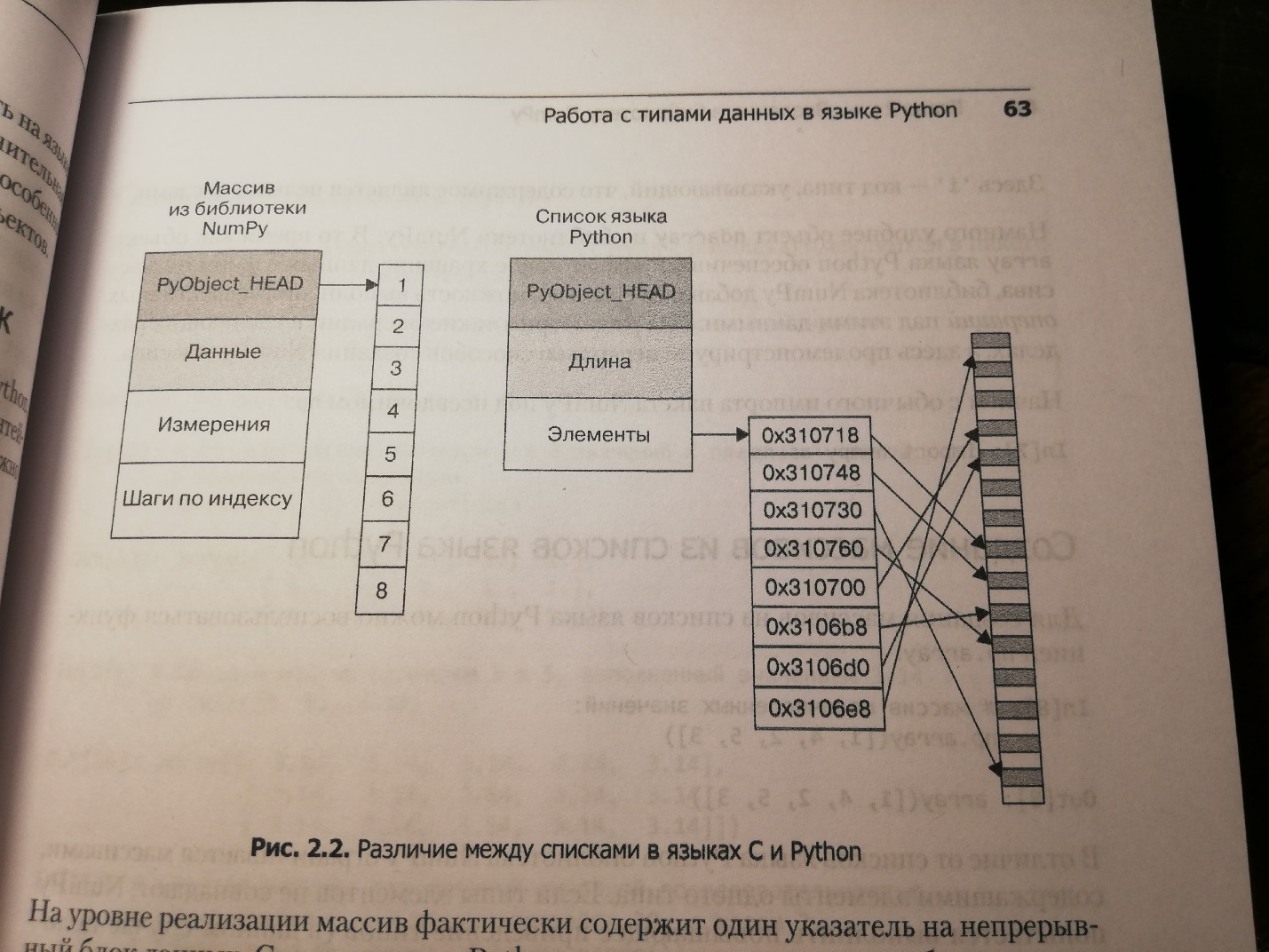
Это значит, что вместе с самим значением x, мы ещё сохраняем и целый служебный набор к нему. Напомню, что целое число в Си – это всего лишь ярлык на место в памяти. Отсюда и столь огромное потребление Python’a по сравнению с низкоуровневым Си. Но это позволило Python’у быть гораздо более простым.

*Список в языке Python – больше, чем просто список*

Теперь давай взглянем на стандартный список, представленный в Python:



Как ты видишь, он умеет быть неоднородным. Однако, за такую гибкость приходится платить. Каждый элемент списка должен хранить информацию о типе, т.е. его элементы – это целые объекты Python. Это очень затратно, если вы используете его, допустим, только для чисел. Но NumPy предлагает свой вариант – массив.

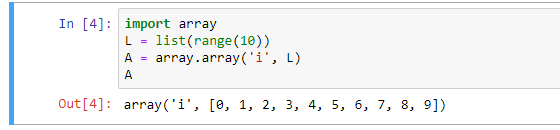


**Да, на такое я тоже готов пойти.**

На уровне реализации, массив – это просто указатель на первый элемент в памяти, за которым непрерывно следуют другие такие же. В Python же список содержит указатель на блок указателей (видимо, этот блок массив), ведущих на целые объекты Python’а. Массивы NumPy гораздо эффективнее хранят данные и работают с ними.

*Массивы с фиксированным типом в языке Python*

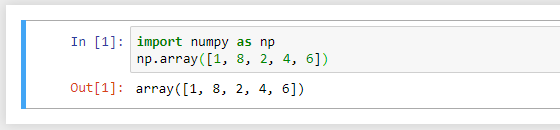
С версии 3.3 в Python появился модуль array. Это значит, что теперь нам доступны полноценные массивы, без всяких излишеств! Вот как его использовать:



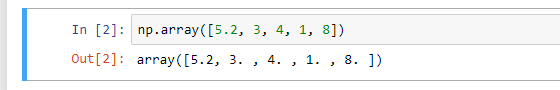
‘i’ здесь – код типа, т.е. int. Уже очень хорошо, можно пользоваться. Но NumPy даёт ещё и эффективные операции над массивами, чем всё-ещё привлекает. Мы рассмотрим их позже, а пока просто опробуем несколько способов их создания. Начнём с традиционного импорта с псевдонимом np.

import numpy as np

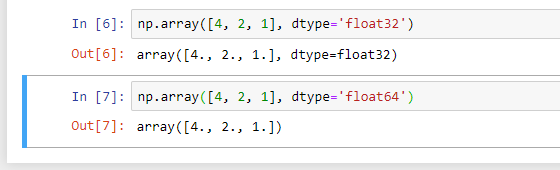
*Создание массивов из списков языка Python*

Наиболее простой способ создания массива – использование функции np.array():

Эта функция создаёт массив из списка. Т.к. NumPy предоставляет только массивы, то все данные в его объектах должны быть однотипными. Если это не так, то он попытается привести их к наибольшему, т.е. если в списке есть float и int, то на выходе все будут float.

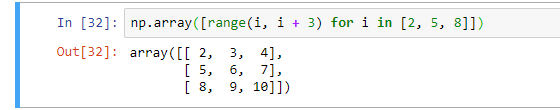


Конечно, можно и задать тип явно, с помощью параметра dtype.



**Как видишь, я запечатлел странность – при аргументе ‘float64’ вывод был без метки типа. Я думаю, что ‘float64’ является типом по умолчанию.**

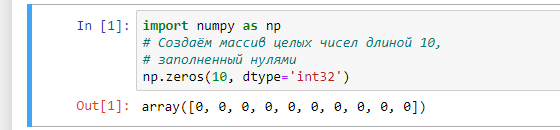
Ещё массивы NumPy приятно выводятся. Взгляни на следующий пример:

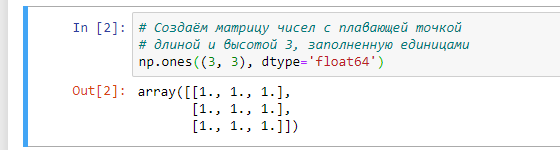


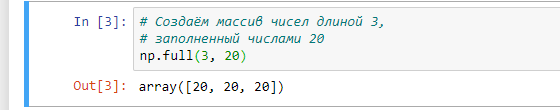
А теперь вспомни, как Python выводит свои списки. Как видишь, тут представление гораздо удобнее. Вложенные списки массивы представлены в виде строк.

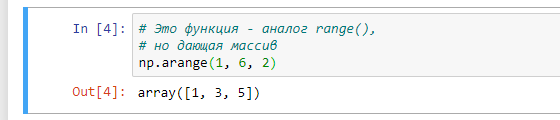
*Создание массивов с нуля*

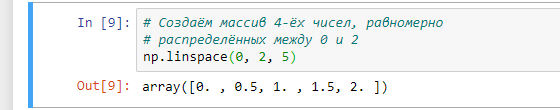
Если вы хотите массив побольше, чем квадратик 3x3, то эффективнее создавать его не из списка, а через встроенные функции NumPy. Вот несколько примеров:



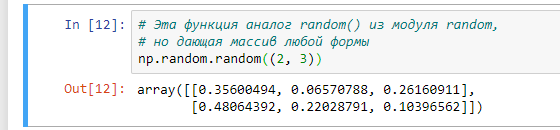


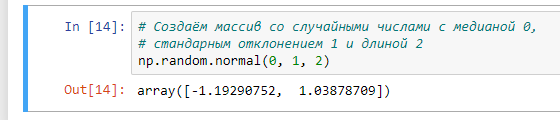




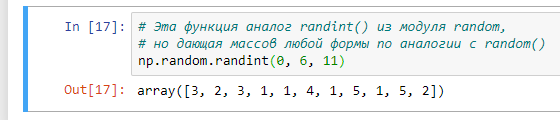


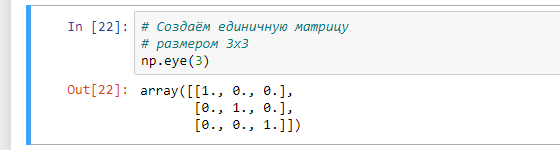
**\*5-ти, из 5-ти чисел.**



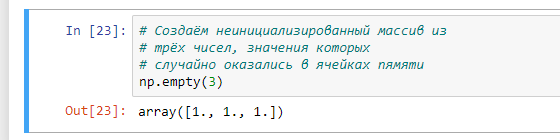


**Тут стоит немного подумать, на счёт медианы и стандартного отклонения.**

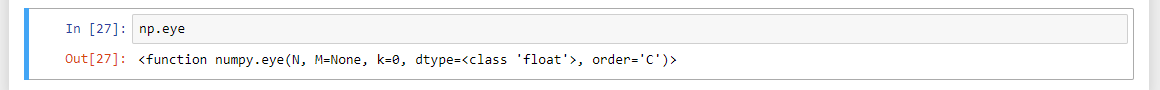




**Единичная матрица – частный случай диагональной матрицы, где на главной диагонали расположены однёрки.**



Если ты хочешь просто увидеть параметры функции, но не хочешь, чтобы её документация заняла половину экрана, достаточно просто ввести её имя без скобок.



Но помни, что такая магия работает только, если код функции написан на Python.

*Стандартные типы данных библиотеки NumPy*

Т.к. массивы NumPy содержат простые типы, надо бы разобраться в них. Т.к. NumPy по большей части написана на С, то многие здешние типы тебе должны бать родными. Напомню, что тип данных массива можно задавать с помощью параметра dtype.

np.zeros(5, dtype=’int8’)

Можно даже не писать строку, а пользоваться готовым.

np.zeros(5, dtype=np.int8)

А вот и все стандартные типы NumPy:

|  |  |
| --- | --- |
| **Тип данных** | **Описание** |
| **bool\_** | **Булев тип (True или False). Занимает целый байт** |
| **int\_** | **Тип целого числа по умолчанию**  **(как long в С, обычно int32 или int64)** |
| **intc** | **Как Int в C (int32 или int64)(даже на Win64 int в C имеет 32 бита)** |
| **intp** | **Как size\_t В C (int32 или int64)(на самом деле unsigned long long)** |
| **int8** | **Байтовый тип (от -128 до 127)** |
| **int16** | **Целое число (от -32 768 до 32 767) (как short в C)** |
| **int32** | **Целое число (от -2 147 483 648 до 2 147 483 647) (как int в C)** |
| **int64** | **Целое число (как long long в C)**  **(от -9 223 372 036 854 775 808 до 9 223 372 036 854 775 808)** |
| **uint8** | **Беззнаковое целое (от 0 до 255)** |
| **uint16** | **Беззнаковое целое (от 0 до 65 535)(как unsigned short в C)** |
| **uint32** | **Беззнаковое целое (от 0 до 4 294 967 295)(как unsigned int в C)** |
| **uint64** | **Беззнаковое целое (от 0 до 18 446 744 073 709 551 615)(llu в C)** |
| **float\_** | **Сокращение float64 (который по умолчанию)** |
| **float16** | **float, у которого 1 бит знак, 5 бит порядок, 10 бита мантисса**  **(если интересно, можешь почитать про строение float)** |
| **float32** | **float, у которого 1 бит знак, 8 бит порядок, 23 бита мантисса** |
| **float64** | **float, у которого 1 бит знак, 11 бит порядок, 52 бита мантисса**  **(double, в общем)** |
| **complex\_** | **Сокращение от complex128** |
| **complex64** | **Комплексное число, состоящее из двух 32-биных чисел**  **(если интересно, можешь почитать о них)** |
| **complex128** | **Комплексное число, состоящее из двух 64-бинтых чисел** |

Ещё в NumPy есть составные типы данных, но о них я тебе поведаю позже.

В NumPy возможно создание своего собственного типа. Путь тебе документацию.

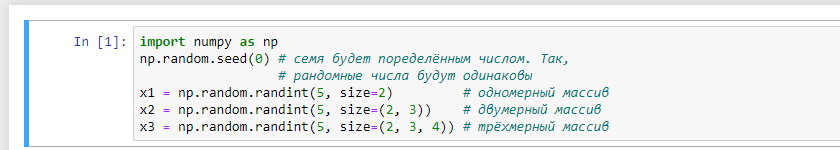
*Введение в массивы библиотеки NumPy*

Работа с данными на Python – всё равно, что работа с NumPy. Даже Pandas основан на этой библиотеке, что уже говорит о многом. Здесь мы начнём работать с основными действиями над массивам, из которых вытекают остальные. И так, ты узнаешь об:

* **атрибутах массивов** – это размер, формы, вес в памяти, тип данных в хранении;
* **индексации массивов –** получение и задание значений его элементов;
* **срезах массивов –** получение части массива;
* **слияние и разбиение массивов** – сложение нескольких массивов в один или его разбиение на меньшие;

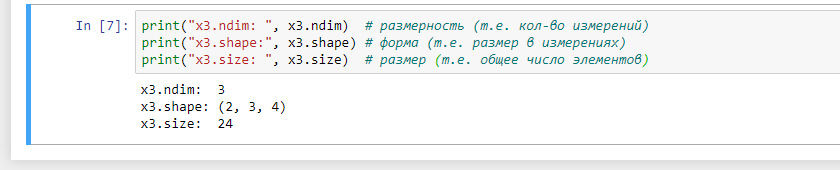
*Атрибуты массивов библиотеки NumPy*

Для начала нам нужны массивы. Для разнообразия я воспользовался генератором рандома NumPy для массивов. Массивы тоже будут разные.

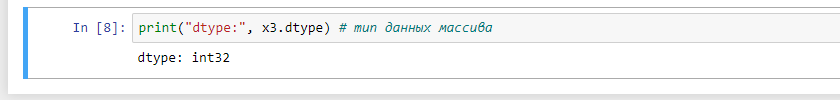


**Я стандартизировал изображения – 560 или 1160 по горизонтали! Ууххуу!**

У каждого массива есть атрибуты. Вот основные:

****

Есть ещё один знакомый тебе атрибут – dtype, т.е. тип данных массива.

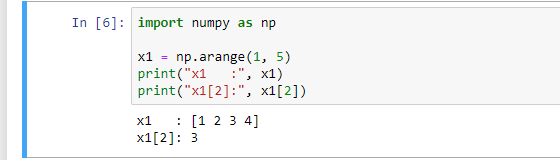


Можно также получить инфо об размере отдельного элемента – с помощью атрибута itemsize, или всего массива в целом – с помощью nbytes (ну или size \* itemsize).

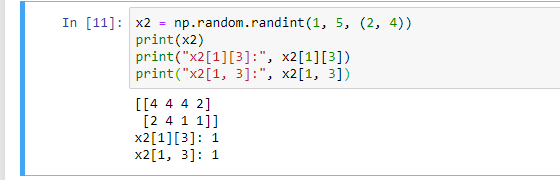


*Индексация массива: доступ к отдельным элементам*

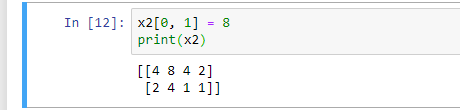
Тут всё просто и по стандарту. Чтобы получить доступ к элементу, используем [i].



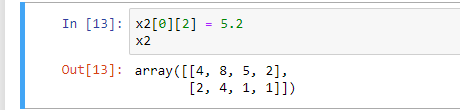
Если же массив многомерный, то тут появляется альтернатива. Вместо [y][x] - [y, x].



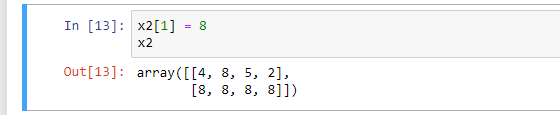
В изменении элементов ничего нового не появляется.



Но помни, что массивы строго типизированы!

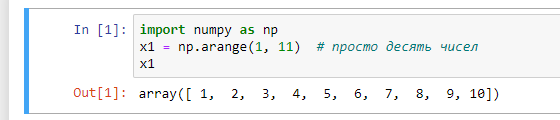


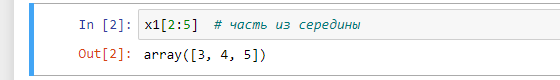
Если ещё поэкспериментировать, то окажется, что Numpy предлагает и такое:

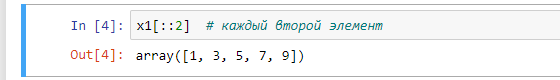


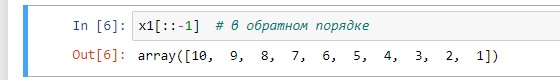
*Срезы массивов: доступ к подмассивам*

Здесь Numpy также поддерживает синтаксис ванильного Python.

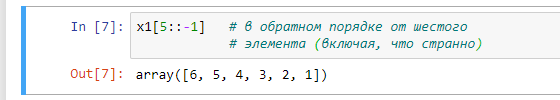




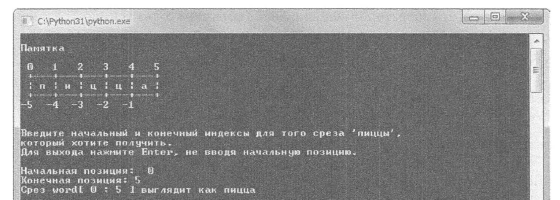




Но будь аккуратен с отрицательным шагом. Ты можешь запутаться!



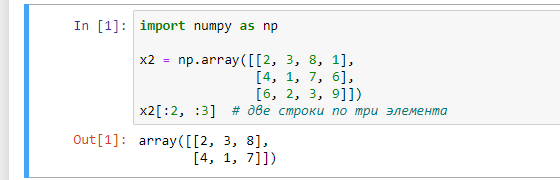
**Тут для меня, в действительности, был сюрприз. Взгляни на часть картинки из замечательной книги Майкла Доусана.**

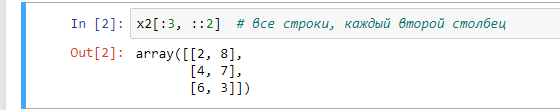
****

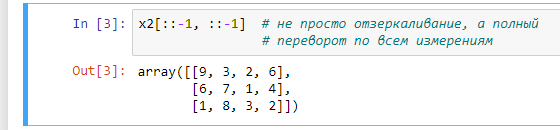
**По идее, в нашем срезе отсчёт должен начаться между 5 и 6, и двигаться к нулю. Т.е. 6-ка не должна быть в срез, но, как видишь, она там есть.**

*Многомерные подмассивы*

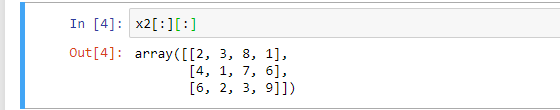
Многомерные массивы срезаются примерно также. Нужны только запятые.



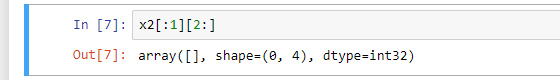




Ты можешь попробовать сделать и «по старинке».



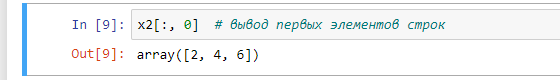
Но этого лучше не делать, т.к. Numpy начинает вести себя непредсказуемо.

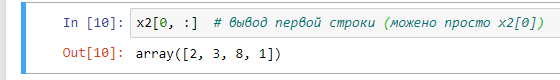


Впрочем, тут уже виноват не Numpy, и даже не IPython, а сам Python.

*Доступ к строкам и столбцам массива*

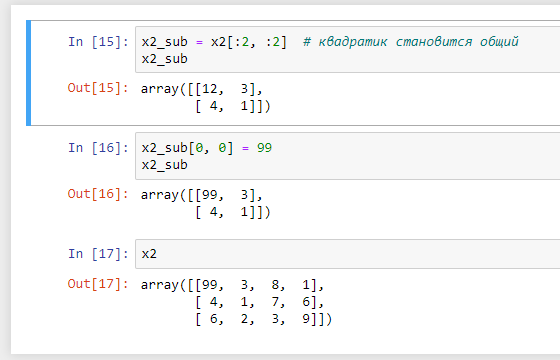
Часто нужен лишь отдельный столбец или строка от массива. Поэтому существует более лаконичный вид записи такого среза.





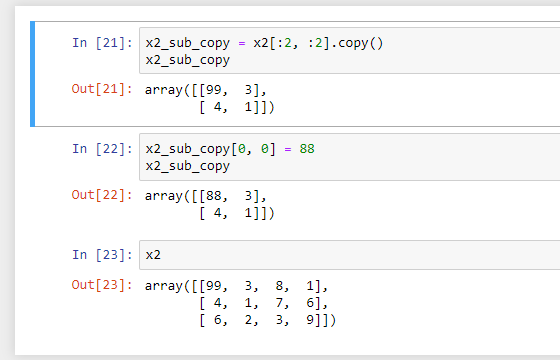
*Подмассивы как предназначенные только для чтения представления*

Списки Python’a возвращают копии. Массивы Numpy возвращают некие представления (views). Даже при срезах. Это значит, что если ты поменяешь значение среза, то это отразится и на исходном массиве. И это действительно удобно! Не тратиться мощность на лишнее копирование.



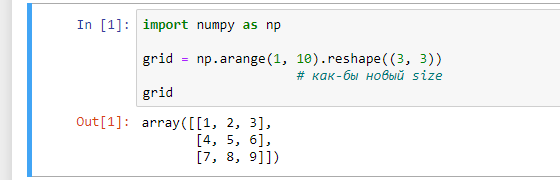
*Создание копий массивов*

Иногда всё же требуется именно копия, не затрагивающая оригинал. Для этого существует метод copy(), явным образом копирующий данные.



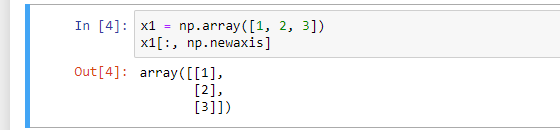
*Изменение формы массивов*

И наконец, мы подошли к самому вкусному (ну, на мой взгляд). Представляю функцию, способную изменить внутренний мир массива – reshape().



Тут главное, чтобы размеры совпадали. Автор говорит, «по возможности, reshape() будет использовать предназначенные только для чтения представления, но непрерывные буферы памяти найти удаётся не всегда». Видимо переводчик тут ошибся, а автор имел ввиду, что reshape() старается создать копию массива и работать с ней, не влияя на оригинал.

Частый паттерн изменения формы – это преобразование одномерного массива в матрицу-строку или матрицу-столбец. Тут на помощь придёт ключевое слово np.newaxis. По факту, это просто пустышка для заполнения измерения, т.е. None.



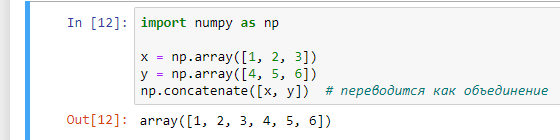
*Слияние и разбиение массивов*

Здесь ты увидишь как массивы умеют складываются и разъединяются.

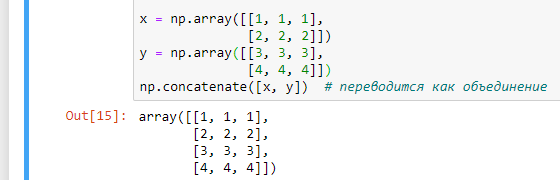
*Слияние массивов*

Для слияния массивов обычно используют функции np.concatenate(). Это основа. Для более частных случаев, когда размерности разные, есть функции np.vstack(), np.hstack.

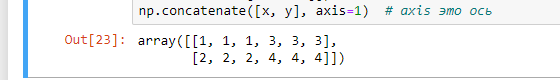
np.concatenate() принимает на вход кортеж или список массивов для объединения.



По умолчанию, np.concatenate() складывает по первой оси (т.е. измерению).

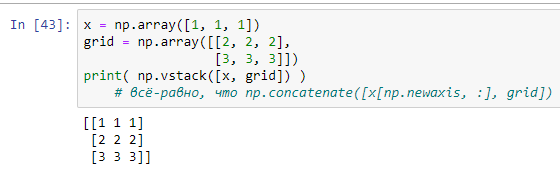


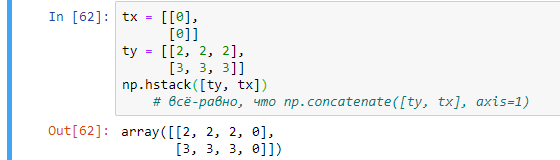
Но можно также выбрать измерение для сложения с помощью параметра axis.



Стоит отметить, что размерность подмассивов x и y здесь должна быть одинакова. Произвольным остаётся только кол-во элементов в выбранной оси.

Для сложения массивов с разными размерностями есть функции np.vstack() (для вертикального объединения), и np.hstack() (для горизонтального).



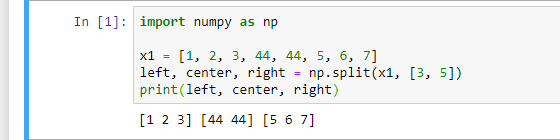


np.dstack() аналогично складывает по третьей оси.

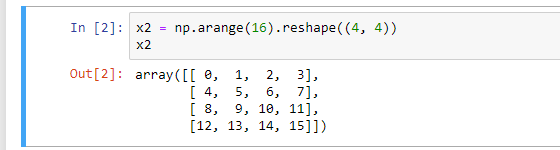
Все эти функции, как я предполагаю, лишь подгоняют массивы под общую форму. Затем вызывают np.concatenate().

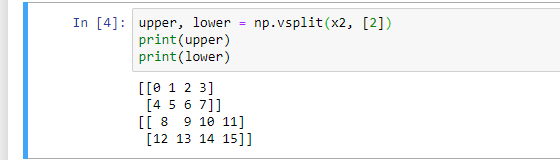
*Разбиение массивов*

Если можно складывать – значит можно и разъединять. Здесь тебе пригодятся np.split(), np.vsplit() и np.hsplit(). Каждой из них нужно передавать список точек разъединения. Вспомни картинку про пиццу в одном из приложении.

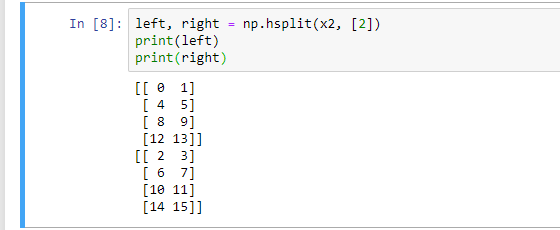


Помни, что если ты делаешь разрез пирога, то получаешь две части, а не одну. Функции np.vsplit() и np.hsplit() работают по вертикали и горизонтали соответственно. Они схожи со своими братьями – функциями np.vstack() и np.hstack().





Отличие между np.split() и np.vsplit() в том, что первой можно задать ещё и axis.



Здесь также есть функция np.dsplit(), разъединяющая по третьей оси.

*Выполнение вычислений над массивами библиотеки NumPy: универсальные функции*

NumPy, помимо самих массивов, даёт ещё и набор оптимизированных функций для них. Лучше не писать свои велосипеды, а пользоваться готовыми эффективными функциями больших дядей, которые, скорее всего, эффективнее и безопаснее твоих. Поэтому мы используем универсальные функции (universal functions, ufuncs) языка Python. Здесь, я думаю, и происходит вся магия.

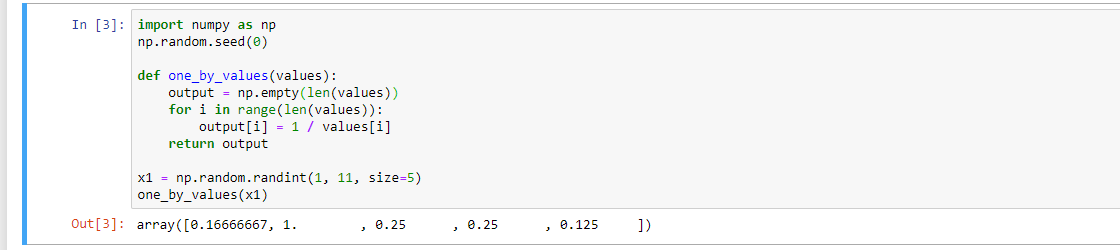
*Медлительность циклов*

Реализация Python по умолчанию (т.е. CPython) местами работает очень медленно. Как я уже говорил, частично в этом виновата динамическая природа языка. Именно это и отнимает возможность компилировать язык в столь быстрый машинный код, что делают языки C и Fortran. Но делаются попытки, призванные исправить это.

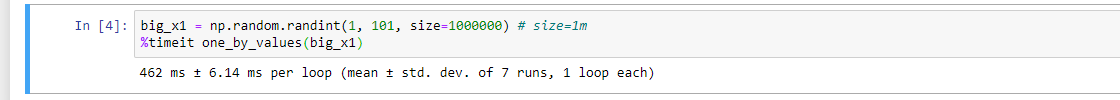
* Проект PyPy (<http://pypy.org>) – интерпретатор Python, написанный на Python. Т.е. он должен быть ещё медленнее, но в когда него добавили JIT компилятор, на лету переводящий код в машинный, скорость выросла в несколько раз.
* Проект Cython (<http://cython.org>) – реализация Python, поддерживающая синтаксис Python плюс строгую типизацию.
* Проект Numba (<http://numba.pydata.org>) – тоже интерпретатор Python, переводящий код в быстрый LLVM-байткод. Тут тоже завязан JIT.

У каждого из этих проектов есть свои сильные и слабые стороны, но самым популярным остаётся «ванильный» Python.

Медлительность Python в основном проявляется в часто повторяющихся операциях, например при переборе массива. Вот допустим, тебе нужно получить обратное значение каждого элемента массива. Очевидное решение может выглядеть так.



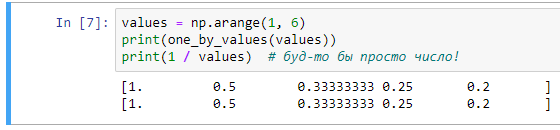
Если ты практиковался на C или Java, то схема тебе знакома. Но мы на Python! Если провести её через %timeit с массивом побольше, то увидим, что она крайне медленна.



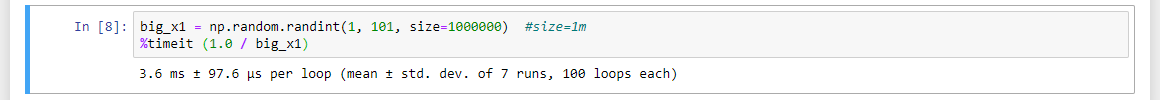
Почти половина секунды! На минуточку, мощности телефонов уже давно измеряются в гигафлопсах (т.е. миллиардах операций с плавающей точкой в секунду). Оказывается, что основная работа приходится типы. Python’у каждый раз приходится проверять тип объекта и вызывать функцию для работы с ним. Если бы этот код был скомпилирован, то всё это было бы известно заранее – а значит, результат вычислялся бы гораздо эффективнее.

*Введение в универсальные функции*

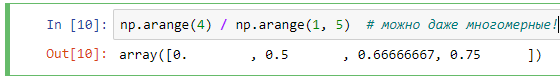
Для подобных случаев в NumPy есть интерфейс с уже скомпилированными функциями. Эти функции называются векторизованными операциями. В общем, суть такая – вы обращаетесь с массивом как с обычным объектом, допустим int. Если вы умножите его на два, то это произойдёт с каждым из его элементов.



Помимо простоты, операции с этим интерфейсом работают ощутимо быстрее!



Векторизованные операции NumPy реализованы посредством универсальных функций (ufuncs), призванных быстро выполнять повторяющиеся операции над элементами массивов. Ещё они очень гибкие. Можно даже работать с массивами!



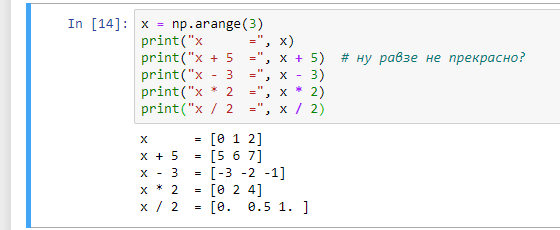
Этот интерфейс включает для многих типов операций свою векторизованную версию. Так что, столкнувшись с подобным, подумай, не использовать ли тебе его.

*Обзор универсальных функций библиотеки NumPy*

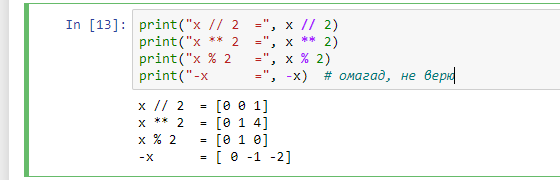
Их существует два вида – унарные, с одним аргументов, и бинарные, с двумя. Ты увидишь как работают оба вида.

*Арифметические функции над массивами*

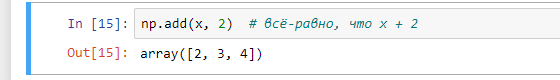
Универсальные функции очень просты, т.к. используют нативные операторы Python.



Помимо абсолютного минимума, поддерживаются также и экзотика.



Все арифметические операции – это удобные адаптеры для функций NumPy. Например, оператор ‘+’ можно заменить простенькой функцией np.add().



А вот и вся таблица реализованных в NumPy арифметических операций.

|  |  |  |
| --- | --- | --- |
| **Оператор** | **Эквивалентная универсальная функция** | **Описание** |
| + | np.add | Сложение (1 + 1 = 2) |
| - | np.subtract | Вычитание ( 3 – 2 = 1) |
| - | np.negative | Унарная операция изменения знака (-2) |
| \* | np.multiply | Умножение (2 \* 3 = 6) |
| / | np.divide | Деление (3 / 2 = 1.5) |
| // | np.floor\_divide | Деление с округлением в меньшую сторону (3 // 2 = 1) |
| \*\* | np.power | Возведение в степень (2 \*\* 3 = 8) |
| % | np.mod | Модуль/остаток (9 % 4 = 1) |

Помимо этого есть ещё и логические, т.е. побитовые операции, но о них позже.

*Абсолютное значение*