NumPy является библиотекой с открытым исходным кодом. Практически в любых вычислениях, начинающихся с сотни тысяч математических операций, лучше использовать NumPy, поскольку она основана на компилируемом языке Си. Ввиду того, что Python — интерпретируемый язык, код на нём обычно срабатывает медленнее, не говоря уже о различных оптимизированных высокоуровневых операциях из линейной алгебры, включённых в библиотеку NumPy. Скорость и удобство работы с многомерными массивами позволили NumPy стать стандартом практически в любой дисциплине, будь то расчёты в физике, вычисления в статистике или приложения машинного обучения. Многие библиотеки машинного обучения, рассматриваемые в дальнейших занятиях, основаны именно на ней, а фреймворки глубокого обучения имеют с ней схожий синтаксис.

Для установки библиотеки мы воспользуемся командой pip install NumPy, которая в jupyter-средах начинается с восклицательного знака. Команда import NumPy as np импортирует функционал библиотеки в переменную np, через которую мы сможем в дальнейшем создавать массивы и вызывать различные полезные функции.

Ключевым объектом библиотеки NumPy является NumPy-массив (np.array), тип данных которого — ndarray, от n-dimensional array (эн-мерного) массива. В большинстве случаев мы имеем дело с 1, 2 или 3-мерными массивами. Создать массив можно с помощью команды np.array(), передав в него список. Помимо списка, инициализацию массивов можно проводить разными способами, о чём мы поговорим ближе к концу урока.

```
1  a = np.array([1, 2, 3])
2  print(a, '|', type(a))
```

```
[1 2 3] | <class 'numpy.ndarray'>
```

Тип данных такого одномерного массива наследуется от того, какие данные лежали в изначальном списке. Если это питоновский тип данных int, то в NumPy-массиве будет np.int64. Тип данных у библиотеки тоже собственный, это нужно для того, чтобы проводить операции с массивами. Целочисленный, вещественный или строковый (object) тип данных и количество бит для хранения одного числа тоже можно задать при инициализации. Важно отметить, что этот аспект часто игнорируют при работе с табличными данными, но в реальных проектах с выборками из десятков миллионов элементов использование 16 бит для хранения чисел каждой характеристики объекта может в разы сократить потребляемую память и скорость вычислений практически без падения в качестве алгоритма. Вывести тип данных массива a можно с помощью атрибута dtype, поменять его можно с помощью .astype (нужный_тип_данных).

```
1  a.dtype
```

```
dtype('int64')
```

Важным отличием NumPy-массива от списка является то, что его размеры должны быть фиксированы. Например, второй элемент (строка) массива не может иметь форму, отличную от первой (или любой другой).

```
1  a = np.array([[1, 2, 3], [4, 5, 6]], float)
2  print(a, '|', type(a))
```

```
[[1. 2. 3.]
 [4. 5. 6.]] | <class 'numpy.ndarray'>
```

Синтаксис индексации и слайсинга очень напоминает список:

```
[ ]  1  a[0]
```

```
array([1., 2., 3.])
```

Обращаясь к нулевому элементу, мы получаем первую строку (индексация начинается с нуля).
Используя запятую, можно производить индексацию (с нуля) по колонкам. По аналогии можно указывать индексы или диапазон индексов для любого измерения в случае многомерных массивов:

```
1  a[1,2]
```

```
6.0
```

Инициализация
Производить инициализацию можно и с помощью встроенных функций. Например, метод zeros принимает на вход желаемую форму массива и возвращает массив из нулей:

```
1  np.zeros(10)
```

```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

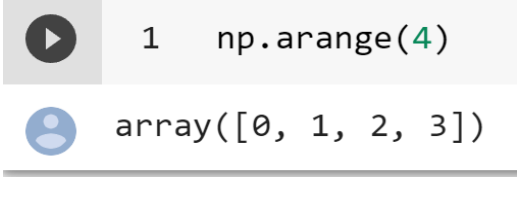
По аналогии, но с единицами, работает и метод ones:

```
1  np.ones(8)
```

```
array([1., 1., 1., 1., 1., 1., 1., 1.])
```

Необязательно инициализировать массив конкретными значениями, можно просто задать форму, а значение будет определено тем, что до этого хранилось в оперативной памяти. Это полезно в ситуациях, когда необходимо часто добавлять что-либо в массив, но количество элементов заранее известно.

Как и в случае с range, в Python существует его аналог в NumPy — np.arange():



Как и в случае с range, можно задавать начало, конец и шаг отсчёта:

```
[ ]   1   np.arange(2, 9, 2)
```

```
array([2, 4, 6, 8])
```

Для генерации значений в вещественном диапазоне можно задать границы (начало и конец) и количество точек между этими границами с помощью linspace:

```
[ ]   1   # от 0 до 10, 5 чисел с равным шагом
      2   np.linspace(0, 10, 5)
```

```
array([ 0. ,  2.5,  5. ,  7.5, 10. ])
```

Для тестирования работы алгоритма бывает полезно создавать значения из распределений: равномерного, нормального и прочих.

```
      1   rng = np.random.default_rng(0)
      2   rng.integers(5, size=(2, 4))
```

```
array([[4, 3, 2, 1],
       [1, 0, 0, 0]])
```

Формы инициализированных массивов в вышеупомянутых функциях можно задавать для любого n-мерного случая (пример с двумерным):

```
[ ]  1  np.ones((3, 2))
```

```
array([[1., 1.],
       [1., 1.],
       [1., 1.]])
```

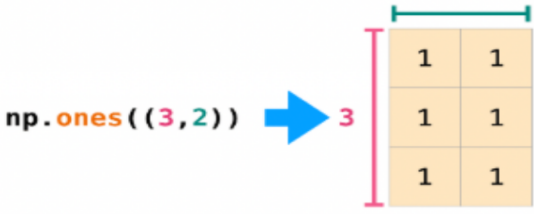
```
[ ]  1  np.zeros((3, 2))
```

```
array([[0., 0.],
       [0., 0.],
       [0., 0.]])
```

```
[ ]  1  rng.random((3, 2))
```

```
array([[0.81327024, 0.91275558],
       [0.60663578, 0.72949656],
       [0.54362499, 0.93507242]])
```

Первое число при инициализации говорит о количестве строк, второе — о количестве столбцов. Третье — о количестве каналов, это часто используется в области анализа изображений (речь идёт о количестве двумерных массивов).



NumPy предлагает множество вариантов сортировки: по убыванию и возрастанию, в лексикографическом формате, частичная сортировка. В простейшем случае используется функция sort, метод сортировки можно указать в аргументах функции:

```
arr = np.array([2, 1, 5, 3, 7, 4, 6, 8])
# Вы можете быстро отсортировать числа в порядке возрастания с помощью:
np.sort(arr)
```

```
array([1, 2, 3, 4, 5, 6, 7, 8])
```

Для удаления элементов авторы библиотеки рекомендуют использовать индексацию, т.е. указать только те индексы, элементы которых вы хотите оставить:

```
a = np.array([1, 2, 3])
# берем элементы на первом и 2 индексе
a = a[[1, 2]]
a
```

```
array([2, 3])
```

С помощью метода append к массиву a можно добавить другой NumPy-массив:

```
1  np.append(a, np.array([4, 5, 6, 7]))
```

```
array([2, 3, 4, 5, 6, 7])
```

Помимо добавления, массивы можно объединять:

```
1  np.concatenate((a,b), axis=0)
```

```
array([[1., 2.],
       [3., 4.],
       [5., 6.],
       [7., 8.]])
```

```
[ ]  1  np.concatenate((a,b), axis=1)
```

```
array([[1., 2., 5., 6.],
       [3., 4., 7., 8.]])
```

Важный аргумент многих функций numpy - axis, который показывает, вдоль какой оси (строки или столбца) мы желаем проводить объединение. Важно помнить, что объединение возможно только в случае, если количество строк или столбцов совпадает. Этот аргумент встречается и далее во многих методах и функциях библиотеки, т.к. применять различные группировки и агрегации можно как ко всем элементам массива (вне

зависимости от положения), так и вдоль определённой оси (учитывая положения). Это бывает полезно, например, при нормализации данных.

Ещё одной особенностью библиотеки является работа с памятью. Присваивая одной переменной (NumPy-массиву) другую, мы создаём массив из ссылок на исходные элементы. Соответственно, изменения значения массива повлекут (т.к. массив C на него ссылается) изменения и самого C. Для создания физической копии значений в оперативной памяти необходимо вызывать метод .copy(). Таким образом NumPy экономит память при работе, так что если вы хотите протестировать ваш алгоритм на небольшом подмножестве (слайсе) ваших данных, не забывайте создавать копию, чтобы не испортить (изменить) исходные данные.

```
a = np.array([1, 2, 3])
b = a
c =  a.copy()

# меняем 0й элемент
a[0] = 0

print(f'Измененный массив a {a}')
print(f'Измененный массив b {b}  <-- изменился по ссылке т.к. поменяти 0й элемент a')
print(f'Измененный массив c {c}  <-- создали копию в памяти')
```

```
Измененный массив a [0 2 3]
Измененный массив b [0 2 3]  <-- изменился по ссылке т.к. поменяти 0й элемент a
Измененный массив c [1 2 3]  <-- создали копию в памяти
```