

ТОМСКИЙ  
ПОЛИТЕХНИЧЕСКИЙ  
УНИВЕРСИТЕТ



# Системный анализ процессов переработки нефти и газа

Лекция №3  
Введение в библиотеку NumPy

Вячеслав Алексеевич Чузлов  
к.т.н., доцент ОХИ ИШПР ТПУ

# Введение

- Библиотека NumPy (**Numerical Python – «числовой Python»**) предоставляет набор эффективных инструментов для хранения и работы с данными.
- Массивы библиотеки NumPy отдаленно напоминают списки Python, однако обеспечивают намного более эффективное хранение и выполнение операций с данными при росте размера массивов.
- Массивы библиотеки NumPy формируют ядро практически всей экосистемы утилит для работы с большими данными (BigData) в Python.



По установившейся традиции, большинство пользователей импортируют пакет NumPy, используя сокращение `np` :

```
>>> import numpy as np
```

# Создание массивов NumPy

# Создание массивов NumPy

- Для того, чтобы создать объект массива NumPy из объекта списка Python, можно использовать функцию `np.array`:

```
>>> import numpy as np  
>>> np.array([1, 3, 5, 4, 2]) # Массив целочисленных значений  
array([1, 3, 5, 4, 2])
```

- В отличие от стандартных списков Python, массивы NumPy могут содержать элементы только одного типа. Если типы элементов не совпадают, NumPy сделает попытку повышающего приведения типов:

```
>>> np.array([3.14, 4, 2, 3, 2.71])  
array([3.14, 4., 2., 3., 2.71])
```

# Создание массивов NumPy

- В тех случаях, когда требуется явно задать тип результирующего массива, необходимо воспользоваться ключевым аргументом `dtype` :

```
>>> np.array([1, 3, 5, 4, 2], dtype='float32')
array([1., 3., 5., 4., 2.], dtype=float32)
```

- В отличие от списков, массивы NumPy можно явным образом описать как многомерные:

```
>>> np.array([range(i, i + 3) for i in [2, 4, 6]])
array([[2, 3, 4],
       [4, 5, 6],
       [6, 7, 8]])
```

# Создание массивов NumPy

Массивы больших размеров эффективнее генерировать с помощью встроенных методов.

- Создаем массив целых чисел длины 10, заполненный нулями:

```
>>> import numpy as np  
>>> np.zeros(10, dtype=int)  
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

- Создадим массив размером  $3 \times 5$  значений с плавающей точкой, заполненный единицами:

```
>>> np.ones((3, 5), dtype=float)  
array([[1., 1., 1., 1., 1.],  
       [1., 1., 1., 1., 1.],  
       [1., 1., 1., 1., 1.]])
```

# Создание массивов NumPy

- Создадим массив размером  $3 \times 5$ , заполненный значением 2.98:

```
>>> np.full((3, 5), 2.98)
array([[2.98, 2.98, 2.98, 2.98, 2.98],
       [2.98, 2.98, 2.98, 2.98, 2.98],
       [2.98, 2.98, 2.98, 2.98, 2.98]])
```

- Создадим массив, заполненный линейной последовательностью, начинающейся с 0 и заканчивающейся 20 (не включая), с шагом 2 (аналогично встроенной функции `range()`):

```
>>> np.arange(0, 20, 2)
array([ 0, 2, 4, 6, 8, 10, 12, 14, 16, 18])
```

- Создадим массив из пяти значений, равномерно располагающихся между 0 и 1:

```
>>> np.linspace(0, 1, 5)
array([0. , 0.25, 0.5 , 0.75, 1. ])
```

# Создание массивов NumPy

- Создадим массив размером  $3 \times 3$  равномерно распределенных случайных значений от 0 до 1 (не включая):

```
>>> np.random.random((3, 3))
array([[0.28303209, 0.54071726, 0.93183376],
       [0.02403954, 0.92295936, 0.62619599],
       [0.06875703, 0.61762719, 0.47795471]])
```

- Создадим массив размером  $3 \times 3$  случайных целых чисел в интервале  $[0; 10)$

```
>>> np.random.randint(0, 10, (3, 3))
array([[3, 6, 7],
       [5, 7, 4],
       [9, 3, 5]])
```



# Атрибуты массивов NumPy для интроспекции

# Атрибуты массивов NumPy для интроспекции

Массив NumPy знает свой ранг, форму, размер, тип `dtype` и другие свойства: их можно определить прямо из специальных атрибутов:

```
>>> import numpy as np
>>> a = np.array(((1, 0, 1), (0, 1, 0)))
>>> a.shape # 2 строки, 3 столбца
(2, 3)
>>> a.ndim # Ранг (число измерений)
2
>>> a.size # Общее количество элементов
6
>>> a.dtype
dtype('int64')
>>> a.data
<memory at 0x102387308>
```

# Атрибуты массивов NumPy для интроспекции

Атрибут	Описание
<code>shape</code>	Измерения массива: размер массива вдоль каждой из его осей, возвращается как кортеж целых чисел
<code>ndim</code>	Количество осей (измерений). Обратите внимание: <code>ndim == len(shape)</code>
<code>size</code>	Общее количество элементов в массиве, равное произведению элементов кортежа <code>shape</code>
<code>dtype</code>	Тип данных массива
<code>data</code>	«Буфер» в памяти, содержащий действительные элементы массива
<code>itemsize</code>	Размер в байтах каждого элемента

# Доступ к элементам массива NumPy



# Индексация элементов массива NumPy

- В одномерном массиве обратиться к  $i$ -му (считая с 0) значению можно по требуемому индексу в квадратных скобках, по аналогии со стандартными списками:

```
>>> import numpy as np
>>> x1 = np.array([5, 0, 3, 3, 7, 9])
>>> x1[0]
5
>>> x1[4]
7
>>> x1[-1]
9
>>> x1[-2]
7
```

# Индексация элементов массива NumPy

- Для обращения к элементам матрицы нужно указать кортеж индексов, разделенных запятыми:

```
>>> x2 = np.array([[3, 5, 2, 4], [7, 6, 8, 8], [1, 6, 7, 7]])
>>> x2
array([[3, 5, 2, 4],
       [7, 6, 8, 8],
       [1, 6, 7, 7]])
>>> x2[0, 0]
3
>>> x2[2, 0]
1
>>> x2[2, -1]
7
```

# Индексация элементов массива NumPy

- При помощи любой из указанных выше нотаций можно изменять значения элементов массива:

```
>>> x2[0, 0] = 24
>>> x2
array([[24,  5,  2,  4],
       [ 7,  6,  8,  8],
       [ 1,  6,  7,  7]])
```

Следует помнить, что, в отличие от списков, массивы NumPy имеют фиксированный тип данных. Если вставить в массив целых чисел значение с плавающей точкой, оно будет **неявно усечено**:

```
>>> x1[0] = 2.71828 # Это значение будет усечено!
>>> x1
array([2, 0, 3, 3, 7, 9])
```

# Изменение формы массива

## Методы `flatten()` и `ravel()`

Предположим, что необходимо «выпрямить» многомерный массив вдоль одной оси. Библиотека NumPy предоставляет для этого два метода: `flatten()` и `ravel()`.

- Оба метода создают одномерный массив в соответствии с его внутренним порядком элементов (по строкам).
- Метод `flatten()` возвращает независимую копию элементов и в общем случае медленнее, чем метод `ravel()`.
- Метод `ravel()` пытается вернуть **представление** преобразованного в одно измерение массива.

**Представление массива** – это массив NumPy, который в данном случае имеет форму, отличающуюся от формы исходного массива, но не содержит «собственных» элементов данных: он содержит **ссылки** на элементы другого массива.

## Методы `flatten()` и `ravel()`

```
>>> a = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> b = a.flatten() # Создает независимую, одномерную копию массива a
>>> b
array([1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> b[3] = 0
>>> b
array([1, 2, 3, 0, 5, 6, 7, 8, 9])
>>> a # Массив a не изменяется
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

Присваивание массиву `b` не изменяет массив `a`, потому что это абсолютно независимые объекты, которые не используют данные совместно.

## Методы `flatten()` и `ravel()`

```
>>> c = a.ravel()  
>>> c  
array([1, 2, 3, 4, 5, 6, 7, 8, 9])  
>>> c[3] = 0  
>>> c  
array([1, 2, 3, 0, 5, 6, 7, 8, 9])  
>>> a  
array([[1, 2, 3],  
       [0, 5, 6],  
       [7, 8, 9]])
```

Необходимо всегда помнить о том, что хотя метод `ravel()` «работает наилучшим образом», возвращая представление внутренних данных, разнообразные операции с массивами (включая вырезание группы элементов (`slicing`)) могут оставлять элементы хранящимися в несмежных локациях памяти.

В этом случае у метода `ravel()` нет другого выбора, кроме создания копии массива.

## Метод `resize()`

- Размер массива может быть изменен (для самого исходного массива) с помощью метода `resize()`, который в качестве аргументов принимает новые значения измерений.

```
>>> a = np.linspace(1, 4, 4)
>>> print(a)
[1. 2. 3. 4.]
>>> a.resize(2, 2) # Изменяет форму исходного массива, возвращаемого значения нет
>>> print(a)
[[1. 2.]
 [3. 4.]]
```

## Метод `reshape()`

- Метод `reshape()` возвращает представление исходного массива с изменением его формы. Исходный массив не изменяется, но оба объекта совместно используют внутренние данные.

```
>>> a = np.linspace(1, 4, 4)
>>> b = a.reshape(2, 2)
>>> print(a)
[1. 2. 3. 4.]
>>> print(b)
[[1. 2.]
 [3. 4.]]
>>> b[0, 0] = -99
>>> print(b)
[[-99. 2.]
 [ 3. 4.]]
>>> print(a)
[-99. 2. 3. 4.]
```

## Метод `reshape()`

- Если требуется поместить числа от 1 до 9 в таблицу  $3 \times 3$ :

```
>>> grid = np.arange(1, 10).reshape((3, 3))
>>> print(grid)
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

*Размер исходного массива должен соответствовать размеру измененного!*

- Другой часто используемый паттерн изменения формы – преобразование одномерного массива в двумерную матрицу-строку или матрицу-столбец. Для этого можно применить метод `reshape()`, но лучше воспользоваться ключевым словом `newaxis` при выполнении операции среза:

```
>>> x = np.array([1, 2, 3])
>>> x.reshape((1, 3)) # преобразование в вектор-строку
array([[1, 2, 3]])
```

## Метод `reshape()`

```
>>> x[np.newaxis, :] # Преобразование в вектор-строку посредством newaxis  
array([[1, 2, 3]])  
>>> x.reshape((3, 1)) # Преобразование в вектор-столбец с помощью reshape  
array([[1],  
       [2],  
       [3]])  
>>> x[:, np.newaxis] # Преобразование в вектор-столбец посредством newaxis  
array([[1],  
       [2],  
       [3]])
```

# Транспонирование массива

- Метод `transpose()` возвращает представление массива с транспонированными осями. Для двумерного массива это обычная операция транспонирования матрицы:

```
>>> a = np.linspace(1, 6, 6).reshape(3, 2)
>>> a
array([[1., 2.],
       [3., 4.],
       [5., 6.]])
>>> a.transpose() # Или просто a.T
array([[1., 3., 5.],
       [2., 4., 6.]])
```

- Транспонирование одномерного массива возвращает этот же неизмененный массив:

```
>>> b = np.array([100, 101, 102, 103])
>>> b.T
array([100, 101, 102, 103])
```

# Объединение массивов

- Слияние, или объединение, двух массивов в библиотеке NumPy выполняется в основном с помощью функций `np.concatenate()`, `np.vstack()` и `np.hstack()`.
- Функция `np.concatenate()` принимает на входе кортеж или список массивов в качестве первого аргумента:

```
>>> x = np.array([1, 2, 3])
>>> y = np.array([3, 2, 1])
>>> np.concatenate([x, y])
array([1, 2, 3, 3, 2, 1])
```

- Можно объединять более двух массивов одновременно:

```
>>> z = [99, 99, 99]
>>> print(np.concatenate([x, y, z]))
[ 1 2 3 3 2 1 99 99 99]
```

# Объединение массивов

- Для объединения двумерных массивов можно также использовать `np.concatenate()` :

```
>>> grid = np.array([[1, 2, 3], [4, 5, 6]])
>>> np.concatenate([grid, grid]) # слияние по первой оси координат
array([[1, 2, 3],
       [4, 5, 6],
       [1, 2, 3],
       [4, 5, 6]])
>>> np.concatenate([grid, grid], axis=1) # слияние по второй оси координат
array([[1, 2, 3, 1, 2, 3],
       [4, 5, 6, 4, 5, 6]])
```

# Объединение массивов

- Для работы с массивами с различающимися измерениями удобнее и понятнее использовать функции `np.vstack()` (вертикальное объединение) и `np.hstack()` (горизонтальное объединение):

```
>>> x = np.array([1, 2, 3])
>>> grid = np.array([[9, 8, 7], [6, 5, 4]])
>>> np.vstack([x, grid]) # Объединяет массивы по вертикали
array([[1, 2, 3],
       [9, 8, 7],
       [6, 5, 4]])
>>> y = np.array([[99], [99]]) # Объединяет массивы по горизонтали
>>> np.hstack([grid, y])
array([[ 9,  8,  7, 99],
       [ 6,  5,  4, 99]])
```

# **Срезы массивов: доступ к подмассивам**



# Срезы массивов: доступ к подмассивам

- Синтаксически срезы массивов NumPy соответствуют срезам стандартных списков Python:

`x[начало:конец:шаг]`

при этом любое из значений можно не указывать, тогда по умолчанию будут приняты следующие значения: `начало = 0`, `конец = размер соответствующего измерения`, `шаг = 1`.

## Одномерные подмассивы

```
>>> x = np.arange(10)
>>> x
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> x[:5]
array([0, 1, 2, 3, 4])
>>> x[4:7]
array([4, 5, 6])
```

## Срезы массивов: доступ к подмассивам

- Массив индексируется кортежем целых чисел, и, как для последовательностей Python, отрицательные индексы отсчитываются от конца оси.
- Срезы и определение шага для них также поддерживаются аналогичным образом. Но следует особо отметить, что операция вырезания группы элементов из массива NumPy возвращает представление (view), а не копию данных, как для списков Python. Для одномерных массивов существует только один индекс:

```
>>> a = np.linspace(1, 6, 6)
>>> print(a)
[1. 2. 3. 4. 5. 6.]
>>> a[1:4:2] # Элементы a[1] и a[3] (с шагом 2)
array([2., 4.])
>>> a[3::-2] # Элементы a[3] и a[1] (с шагом -2)
array([4., 2.])
```

## Срезы массивов: доступ к подмассивам

- Для многомерных массивов индекс существует по каждой оси.
- Если необходимо выбрать каждый элемент по конкретной оси, то нужно заменить ее индекс на один столбец:

```
>>> a = np.linspace(1, 12, 12).reshape(4, 3)
>>> a
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.],
       [ 7.,  8.,  9.],
       [10., 11., 12.]])
>>> a[3, 1]
11.0
>>> a[2, :] # Все элементы в третьей строке
array([7., 8., 9.])
>>> a[:, 1] # Все элементы во втором столбце
array([2., 5., 8., 11.])
>>> a[1:-1, 1:] # Средние строки, начиная со второго столбца
array([[5., 6.],
       [8., 9.]])
```

# Срезы массивов: доступ к подмассивам

1	2	3
4	5	6
7	8	9
10	11	12

a[2, :]

1	2	3
4	5	6
7	8	9
10	11	12

a[:, 1]

1	2	3
4	5	6
7	8	9
10	11	12

a[1:-1, 1:]

1	2	3
4	5	6
7	8	9
10	11	12

a[::-2, :]

1	2	3
4	5	6
7	8	9
10	11	12

a[2:, :-1]

1	2	3
4	5	6
7	8	9
10	11	12

a[1::2, ::2]

## Срезы массивов: доступ к подмассивам

- Специализированная форма записи в виде многоточия (...) удобна для массивов высокого ранга: в индексе она представляет столько столбцов, сколько необходимо для указания всех оставшихся осей.

Например, для четырехмерного массива:

`a[3, 1, ...]` равнозначно `a[3, 1, :, :]`

`a[3, ..., 1]` равнозначно `a[3, :, :, 1]`

Синтаксис с символами двоеточия и многоточия работает и для присваивания:

```
>>> a[:, 1] = 0 # Для всех элементов во втором столбце установить значение ноль
>>> print(a)
[[ 1.  0.  3.]
 [ 4.  0.  6.]
 [ 7.  0.  9.]
 [10.  0. 12.]]
```

# Сложное индексирование

- Массивы NumPy можно также индексировать последовательностями, которые не являются простыми кортежами целых чисел, а именно другими списками, массивами целых чисел и кортежами кортежей.
- Такое «продвинутое индексирование» создает новый массив с собственной копией исходных данных, а не представление:

```
>>> a = np.linspace(0., 0.5, 6)
>>> print(a)
[0. 0.1 0.2 0.3 0.4 0.5]
>>> ia = [1, 4, 5] # Список индексов
>>> print(a[ia])
[0.1 0.4 0.5]
>>> ia = np.array((1, 2), (3, 4))
>>> print(a[ia]) # Массив, формируемый по заданным индексам
[[0.1 0.2]
 [0.3 0.4]]
```

# Сложное индексирование

- Можно проиндексировать многомерный массив с помощью многомерных массивов индексов:

```
>>> a = np.linspace(1, 12, 12).reshape(4, 3)
>>> print(a)
[[ 1.  2.  3.]
 [ 4.  5.  6.]
 [ 7.  8.  9.]
 [10. 11. 12.]]
>>> ia = np.array(((1, 0), (2, 1)))
>>> ja = np.array(((0, 1), (1, 2)))
>>> print(a[ia, ja])
[[4. 2.]
 [8. 6.]]
```

Здесь создается массив  $2 \times 2$  (форма, определяемая массивами индексов) с элементами  $a[1, 0]$ ,  $a[0, 1]$  в верхней строке и элементами  $a[2, 1]$ ,  $a[1, 2]$  в нижней строке.

## Сложное индексирование

- Вместо индексации массива с помощью последовательности целых чисел также возможно использование массива логических значений.
- Элементы `True` такого индексного массива обозначают элементы целевого массива, которые необходимо вернуть:

```
>>> a = np.array([-2, -1, 0, 1, 2])
>>> ia = np.array([False, True, False, True, True])
>>> print(a[ia])
[-1 1 2]
```

# Сложное индексирование

- Поскольку операции сравнения векторизованы по элементам массивов тоно так же, как математические операции, становится возможным применение полезных приемов:

```
>>> print(a)
[-2 -1 0 1 2]
>>> ib = a < 0
>>> print(ib)
[True True False False False]
>>> a[ib] = 0 # Замена всех отрицательных элементов нулями
>>> print(a)
[0 0 0 1 2]
```

- В действительности нет необходимости сохранять вспомогательный массив логических значений `ib`, так как инструкция `a[a < 0] = 0` выполняет ту же самую работу:

```
>>> a = np.array([-2, -1, 0, 1, 2])
>>> a[a < 0] = 0
>>> print(a)
[0 0 0 1 2]
```

# Срезы массивов создают разделяемые ссылки

- Срезы массивов возвращают **разделяемые ссылки** (синонимы), а не **копии** данных массива.
- Этим срезы массивов библиотеки NumPy отличаются от срезов списков языка Python (срезы списков создают новые объекты):

```
>>> x = np.random.randint(1, 10, (3, 4))
>>> x
array([[2, 6, 5, 3],
       [3, 4, 4, 4],
       [8, 2, 9, 9]])
```

Получим из него матрицу  $2 \times 2$ :

```
>>> x2_sub = x[:2, :2]
>>> x2_sub
array([[2, 6],
       [3, 4]])
```

## Срезы массивов создают разделяемые ссылки

Теперь, если изменить значения этой матрицы, исходный массив также изменится:

```
>>> x2_sub[0, 0] = 100
>>> x2_sub
array([[100, 6],
       [ 3, 4]])
>>> x
array([[100, 6, 5, 3],
       [ 3, 4, 4, 4],
       [ 8, 2, 9, 9]])
```

# Создание копий массивов

Для создания копии массива существует метод `copy()`:

```
>>> x2_subcopy = x[:2, :2].copy()
>>> x2_subcopy
array([[100, 6],
       [ 3, 4]])
```

Теперь, если изменить значения этого подмассива, то исходный массив не изменится:

```
>>> x2_subcopy[0, 0] = 2
>>> x2_subcopy
array([[2, 6],
       [3, 4]])
>>> x
array([[100, 6, 5, 3],
       [ 3, 4, 4, 4],
       [ 8, 2, 9, 9]])
```



# Вычисления с массивами NumPy

## Арифметические функции с массивами

Универсальные функции NumPy могут быть легко использованы, т.к. основаны на нативных арифметических операторах Python. Доступны обычные операторы сложения, вычитания, умножения и деления:

```
>>> import numpy as np
>>> x = np.arange(4)
>>> print('x =', x)
x = [0 1 2 3]
>>> print('x + 5 =', x + 5)
x + 5 = [5 6 7 8]
>>> print('x - 5 =', x - 5)
x - 5 = [-5 -4 -3 -2]
>>> print('x * 2 =', x * 2)
x * 2 = [0 2 4 6]
>>> print('x / 2 =', x / 2)
x / 2 = [0. 0.5 1. 1.5]
>>> print('x // 2 =', x // 2)
x // 2 = [0 0 1 1]
```

## Арифметические функции с массивами

Определены также унарная универсальная функция изменения знака, оператор `**` для возведения в степень и оператор `%` для деления по модулю:

```
>>> print('-x =', -x)
-x = [ 0 -1 -2 -3]
>>> print('x ** 2 =', x ** 2)
x ** 2 = [0 1 4 9]
>>> print('x % 2 =', x % 2)
x % 2 = [0 1 0 1]
```

Данные операции могут быть использованы в выражениях любыми способами с соблюдением стандартных приоритетов:

```
>>> -(0.5 * x + 1) ** 2
array([-1. , -2.25, -4. , -6.25])
```

# Арифметические функции с массивами

Все арифметические операторы – удобные аналоги для встроенных функций библиотеки NumPy.

Оператор	Эквивалентная 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: $3 ** 2 = 9$
%	np.mod()	Модуль/остаток: $5 \% 2 = 1$

# Операторы сравнения

- В библиотеке NumPy также реализованы операторы сравнения, такие как `<` и `>` в виде поэлементных универсальных функций.
- Результат этих операторов сравнения всегда представляет собой массив с булевым типом данных.

```
>>> x = np.array([1, 2, 3, 4, 5])
>>> x < 3 # меньше
array([ True,  True, False, False, False])
>>> x > 3 # больше
array([False, False, False, True, True])
>>> x <= 3 # меньше или равно
array([ True,  True,  True, False, False])
>>> x >= 3 # больше или равно
array([False, False, True, True, True])
>>> x != 3 # не равно
array([ True,  True, False, True, True])
>>> x == 3 # равно
array([False, False, True, False, False])
```

# Операторы сравнения

- Можно выполнять поэлементное сравнение двух массивов и использовать составные выражения:

```
>>> 2 * x == x ** 2
array([False, True, False, False, False])
```

- Операторы сравнения могут работать с массивами любого размера и формы:

```
>>> x = np.random.randint(1, 10, (3, 4))
>>> x
array([[6, 6, 4, 4],
       [1, 9, 4, 8],
       [7, 2, 4, 3]])
>>> x < 6
array([[False, False,  True,  True],
       [ True, False,  True, False],
       [False,  True,  True,  True]])
```

## Получение абсолютного значения

Наряду со встроенными арифметическими операторами, с массивами NumPy можно использовать стандартную функцию `abs()` языка Python для получения абсолютного значения:

```
>>> x = np.array([-2, -1, 0, 1, 2])
>>> abs(x)
array([2, 1, 0, 1, 2])
```

Аналогичная универсальная функция NumPy – `np.absolute()`, доступна также под псевдонимом `np.abs()`:

```
>>> np.absolute(x)
array([2, 1, 0, 1, 2])
>>> np.abs(x)
array([2, 1, 0, 1, 2])
```

# Тригонометрические функции

Библиотека NumPy предоставляет набор тригонометрических функций:

```
>>> alpha = np.linspace(0, np.pi, 3)
>>> print('alpha      = ', alpha)
alpha      = [0.          1.57079633 3.14159265]
>>> print('sin(alpha) = ', np.sin(alpha))
sin(alpha) = [0.000000e+00 1.000000e+00 1.2246468e-16]
>>> print('cos(alpha) = ', np.cos(alpha))
cos(alpha) = [ 1.000000e+00 6.123234e-17 -1.000000e+00]
>>> print('tan(alpha) = ', np.tan(alpha))
tan(alpha) = [ 0.0000000e+00 1.63312394e+16 -1.22464680e-16]
```

Значения вычисляются в пределах точности конкретной вычислительной машины, вследствие чего некоторые из них не всегда точно равны нулю, хотя должны.

# Тригонометрические функции

Определены также и обратные тригонометрические функции:

```
>>> x = [-1, 0, 1]
>>> print('x = ', x)
x = [-1, 0, 1]
>>> print('arcsin(x) = ', np.arcsin(x))
arcsin(x) = [-1.57079633  0.           1.57079633]
>>> print('arccos(x) = ', np.arccos(x))
arccos(x) = [3.14159265 1.57079633 0.           ]
>>> print('arctan(x) = ', np.arctan(x))
arctan(x) = [-0.78539816  0.           0.78539816]
```

# Показательные функции и логарифмы

## Показательные функции:

```
>>> x = [1, 2, 3]
>>> print('x = ', x)
x = [1, 2, 3]
>>> print('e^x = ', np.exp(x))
e^x = [ 2.71828183  7.3890561  20.08553692]
>>> print('2^x = ', np.exp2(x))
2^x = [2. 4. 8.]
>>> print('3^x = ', np.power(3, x))
3^x = [ 3  9 27]
```

## Логарифмы:

```
>>> x = [1, 2, 4, 10]
>>> print('ln(x) =', np.log(x))
ln(x) = [0.          0.69314718 1.38629436 2.30258509]
>>> print('log2(x) =', np.log2(x))
log2(x) = [0.          1.          2.          3.32192809]
>>> print('log10(x) =', np.log10(x))
log10(x) = [0.          0.30103    0.60205999 1.          ]
```

# Функции агрегирования



# Суммирование значений массива

В стандартном Python данная задача решается при помощи встроенной функции `sum()`.

```
>>> arr = np.random.random(100)
>>> sum(arr)
51.04318277555585
>>> np.sum(arr)
51.043182775555834
```

NumPy версия функции `np.sum()` работает намного быстрее:

```
>>> big_array = np.random.rand(1000000)
>>> sum(big_array) # 55.4 ms ± 450 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
499951.20542974747
>>> np.sum(big_array) # 377 µs ± 5.09 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
499951.2054297427
```

# Суммирование значений массива

Функции `sum()` и `np.sum()` не идентичны: `np.sum()` может работать с многомерными массивами.

```
>>> x = np.random.randint(1, 10, (3, 4))
>>> x
array([[5, 5, 5, 8],
       [7, 3, 5, 2],
       [3, 2, 4, 4]])
>>> np.sum(x), np.sum(x, axis=0), np.sum(x, axis=1)
(53, array([15, 10, 14, 14]), array([23, 17, 13]))
```

# Сумма и произведение

- Используйте именованный аргумент `where` в тех случаях, когда необходимо вычислить параметры агрегирования для элементов массива, удовлетворяющих какому-либо условию:

```
>>> x = np.arange(1, 11, 1)
>>> x
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
>>> x.sum()
55
>>> x.sum(where=x % 2 == 1) # сумма нечетных элементов
25
>>> x.prod(), x.prod(where=x > 5)
(3628800, 30240)
```

## Минимум и максимум

В стандартном Python определены встроенные функции `min()` и `max()`:

```
>>> big_array = np.random.rand(1000000)
>>> min(big_array), max(big_array)
(6.4005968303249e-08, 0.9999976706393126)
```

Соответствующие функции NumPy имеют аналогичный синтаксис и работают быстрее:

```
>>> np.min(big_array), np.max(big_array)
(6.4005968303249e-08, 0.9999976706393126)
```

## Минимум и максимум

Вычисление некоторых сводных показателей возможно при вызове соответствующих методов:

```
>>> big_array.min(), big_array.max(), big_array.sum()
(6.4005968303249e-08, 0.9999976706393126, 500510.0704873639)
>>> x = np.random.randint(1, 10, (3, 5))
>>> x
array([[6, 6, 6, 4, 7],
       [9, 8, 1, 3, 5],
       [3, 3, 4, 4, 2]])
>>> x.min(axis=0), x.max(axis=1), x.sum(axis=0)
(array([3, 3, 1, 3, 2]), array([7, 9, 4]), array([18, 17, 11, 11, 14]))
```

# Многомерные сводные показатели

- Агрегирование по столбцу или строке – один из часто применяемых видов операций.  
Пусть имеются какие-либо данные, находящиеся в двумерном массиве:

```
>>> m = np.random.random((3, 4))
>>> m
array([[0.1316389 , 0.4891762 , 0.5110874 , 0.80852699],
       [0.56048804, 0.21141372, 0.66407676, 0.59120541],
       [0.20694219, 0.90174667, 0.83109611, 0.10042495]])
```

- По умолчанию все функции агрегирования библиотеки NumPy возвращают сводный показатель по всему массиву:

```
>>> m.sum()
6.0078233599742745
```

# Многомерные сводные показатели

- Но функции агрегирования принимают на входе дополнительный аргумент, позволяющий указать ось, по которой вычисляется сводный показатель.
- Например, можно найти минимальное значение каждого из столбцов, указав `axis=0`:

```
>>> m.min(axis=0)
array([0.1316389 , 0.21141372, 0.5110874 , 0.10042495])
```

Функция возвращает четыре значения, соответствующие четырем столбцам чисел.

- Аналогично можно вычислить максимальное значение в каждой из строк:

```
>>> m.max(axis=1)
array([0.80852699, 0.66407676, 0.90174667])
```

# Другие функции агрегирования

Имя функции	Описание
np.sum()	Вычисляет сумму элементов
np.prod()	Вычисляет произведение элементов
np.mean()	Вычисляет среднее значение элементов
np.std()	Вычисляет стандартное отклонение
np.var()	Вычисляет дисперсию
np.min()	Вычисляет минимальное значение
np.max()	Вычисляет максимальное значение
np.argmax()	Возвращает индекс минимального значения
np.argmax()	Возвращает индекс максимального значения
np.median()	Вычисляет медиану элементов
np.any()	Проверяет, существуют ли элементы со значением True
np.all()	Проверяет, все ли элементы имеют значение True



# Чтение и запись массива в файл

## Функция np.loadtxt()

Сигнатура данной функции:

```
np.loadtxt(fname, dtype=<class 'float'>, comments='#',
           delimiter=None, converters=None, skiprows=0,
           usecols=None, unpack=False, ndmin=0)
```

- `fname` – единственный обязательный аргумент, который может быть именем файла, указателем на открытый файл или генератором, возвращающим строки данных для обработки (парсинга);
- `dtype` – тип данных массива, по умолчанию `float`, но его можно явно установить в этом аргументе.
- `comments` – комментарии в файле обычно начинаются с некоторого символа, как, например, `#` (в исходном коде Python) или `%`. Чтобы библиотека NumPy игнорировала содержимое всех строк, начинающихся с такого символа, используется аргумент `comments`, для которого по умолчанию задано значение `#`;

## Функция np.loadtxt()

- `delimiter` – строка, используемая для разделения столбцов данных в файле. По умолчанию задано значение `None`, означающее, что любое количество пробельных символов (пробелов, табуляций) разделяет данные. Для считывания файлов в формате csv (данные, разделенные запятыми) необходимо установить `delimiter=','`;
- `converters` – необязательный словарь, отображающий индекс столбца в функцию, преобразующую строковые значения из этого столбца в данные (например, типа `float`);
- `skiprows` – целое число, определяющее количество пропускаемых строк в начале файла, прежде чем начать считывание данных (например, для пропуска строк заголовка). По умолчанию равно 0 (заголовка нет);
- `usecols` – последовательность индексов столбцов, определяющая, какие столбцы из файла должны возвращаться как данные. По умолчанию задано значение `None`, означающее, что все столбцы должны быть обработаны и возвращены;

## Функция np.loadtxt()

- `unpack` – по умолчанию таблица данных возвращается в одном массиве из строк и столбцов, отображающих структуру считывающегося файла. При установке `unpack=True` этот массив будет преобразован (транспонирован) так, чтобы можно было выбирать отдельные столбцы и присваивать их различным переменным;
- `ndmin` – минимальное количество измерений, которое должен иметь возвращаемый массив. По умолчанию задано значение 0 (так что файл, содержащий единственное число, считывается как скалярное значение). Можно установить значение 1 или 2.

Например, для считывания первого, третьего и четвертого столбцов из файла `data.txt` в три отдельных одномерных массива выполняется следующая инструкция:

```
col1 , col3 , col4 = np.loadtxt('data.txt', usecols=(0, 2, 3), unpack=True)
```

## Функция np.savetxt()

Сигнатура функции:

```
np.savetxt(fname , X, fmt='%.18e', delimiter=' ',  
newline='\n', header='', footer='', comments='# ')
```

- `fname` – имя файла или ссылка (дескриптор) на открытый файл, в который будут сохраняться данные массива;
- `X` – имя сохраняемого массива;
- `fmt` – строка, определяющая спецификаторы формата в стиле языка С, при выводе данных массива. По умолчанию это строка `'%.18e'`;
- `delimiter` – строка, разделяющая столбцы в файле вывода; по умолчанию – один пробел;
- `newline` – строка, разделяющая строки в файле вывода; по умолчанию используется Unix-стиль `'\n'`. Пользователи Windows могут предпочесть установку для `newline` последовательности символов, применяемую на их платформе: `'\r\n'`;

## Функция np.savetxt()

- `header` – строка (возможно, несколько строк), которая должна быть записана в начале файла вывода;
- `footer` – строка (возможно, несколько строк), которая должна быть записана в конце файла вывода;
- `comments` – строка, которая должна добавляться в строки `header` и `footer`, чтобы пометить их как комментарии. По умолчанию это строка '# '.



**Полиномы**

# Определение и вычисление полинома

Полиномиальный (конечный) степенной ряд в качестве базиса использует степени  $x$ :  $1 (= x^0)$ ,  $x$ ,  $x^2$ ,  $x^3$ , ...,  $x^n$  с коэффициентами  $c_i$ :

$$P(x) = \sum_{i=0}^n c_0 + c_1x + c_2x^2 + c_3x^3 + \dots + c_nx^n$$

Обобщенный класс для описания полинома `numpy.polynomial.Polynomial`. Чтобы импортировать его напрямую, используйте инструкцию:

```
from numpy.polynomial import Polynomial
```

ИЛИ

```
import numpy as np  
Polynomial = np.polynomial.Polynomial
```

## Определение и вычисление полинома

Для определения объекта полинома необходимо передать в конструктор `Polynomial` последовательность коэффициентов для постепенно возрастающих степеней  $x$ , начиная с  $c_0$ .

Например, для представления полинома

$$P(x) = 6 - 5x + x^2$$

определяется объект

```
>>> p = Polynomial([6, -5, 1])
```

# Определение и вычисление полинома

- Проверить коэффициенты объекта `Polynomial` можно функцией `print()` или обращением к его атрибуту `coef`.

```
>>> print(p)
6.0 - 5.0 x + 1.0 x**2
>>> p.coef
array([ 6., -5.,  1.])
```

- Для вычисления многочлена при заданном значении  $x$  необходимо «вызвать» многочлен, как показано ниже:

```
>>> p(4) # Вычисление многочлена p при заданном одиночном значении x
2.0
>>> x = np.linspace(-5, 5, 11)
>>> print(p(x)) # Вычисление p при заданной последовательности значений x
[56. 42. 30. 20. 12.  6.  2.  0.  0.  2.  6.]
```

## Поиск корней

- Корни полинома возвращает метод `roots()`. Одинаковые корни просто повторяются в возвращаемом массиве:

```
>>> q = Polynomial([2, -3])
>>> p = Polynomial([6, -5, 1])
>>> p.roots()
array([2., 3.])
>>> (q * q).roots()
array([0.66666667, 0.66666667])
>>> Polynomial([5, 4, 1]).roots()
array([-2.-1.j, -2.+1.j])
```

- Полиномы можно также создавать по их корням с помощью метода `Polynomial. fromroots()`:

```
>>> print( Polynomial.fromroots([-4, 2, 1]))
8.0 - 10.0 x + 1.0 x**2 + 1.0 x**3
```

| То есть  $(x + 4)(x - 2)(x - 1) = 8 - 10x + x^2 + x^3$ .

# Математический анализ

- Полиномы можно дифференцировать с помощью метода `Polynomial.deriv()`. По умолчанию этот метод возвращает первую производную, но в необязательном аргументе `m` можно определить возврат  $m$ -й производной:

```
>>> print(p)
6.0 - 5.0 x + 1.0 x**2
>>> print(p.deriv())
-5.0 + 2.0 x
>>> print(p.deriv(2))
2.0
```

# Математический анализ

- Объект `Polynomial` можно интегрировать с необязательной нижней границей `L` и постоянной интегрирования `k`:

$$\int_L^x 2 - 3x \, dx = 2x - \frac{3}{2}x^2 \Big|_L^x = 2x - \frac{3}{2}x^2 - 2L + \frac{3}{2}L^2$$

$$\int 2 - 3x \, dx = 2x - \frac{3}{2}x^2 + k$$

По умолчанию `L` и `k` равны нулю, но их значения можно определить в аргументах `lbound` и `k`:

```
>>> print(q)
2.0 - 3.0 x
>>> print(q.integ())
0.0 + 2.0 x - 1.5 x**2
>>> print(q.integ(lbnd=1))
-0.5 + 2.0 x - 1.5 x**2
>>> print(q.integ(k=2))
2.0 + 2.0 x - 1.5 x**2
```

## Метод `Polynomial.fit()`

- Метод класса `Polynomial.fit()` возвращает полином, подогнанный методом наименьших квадратов к данным из выборки значений  $y$ .
- Для метода `fit()` требуются массивы `x` и `y` и значение `deg` – степень полинома.
- Метод возвращает полином, который минимизирует сумму квадратов ошибок:

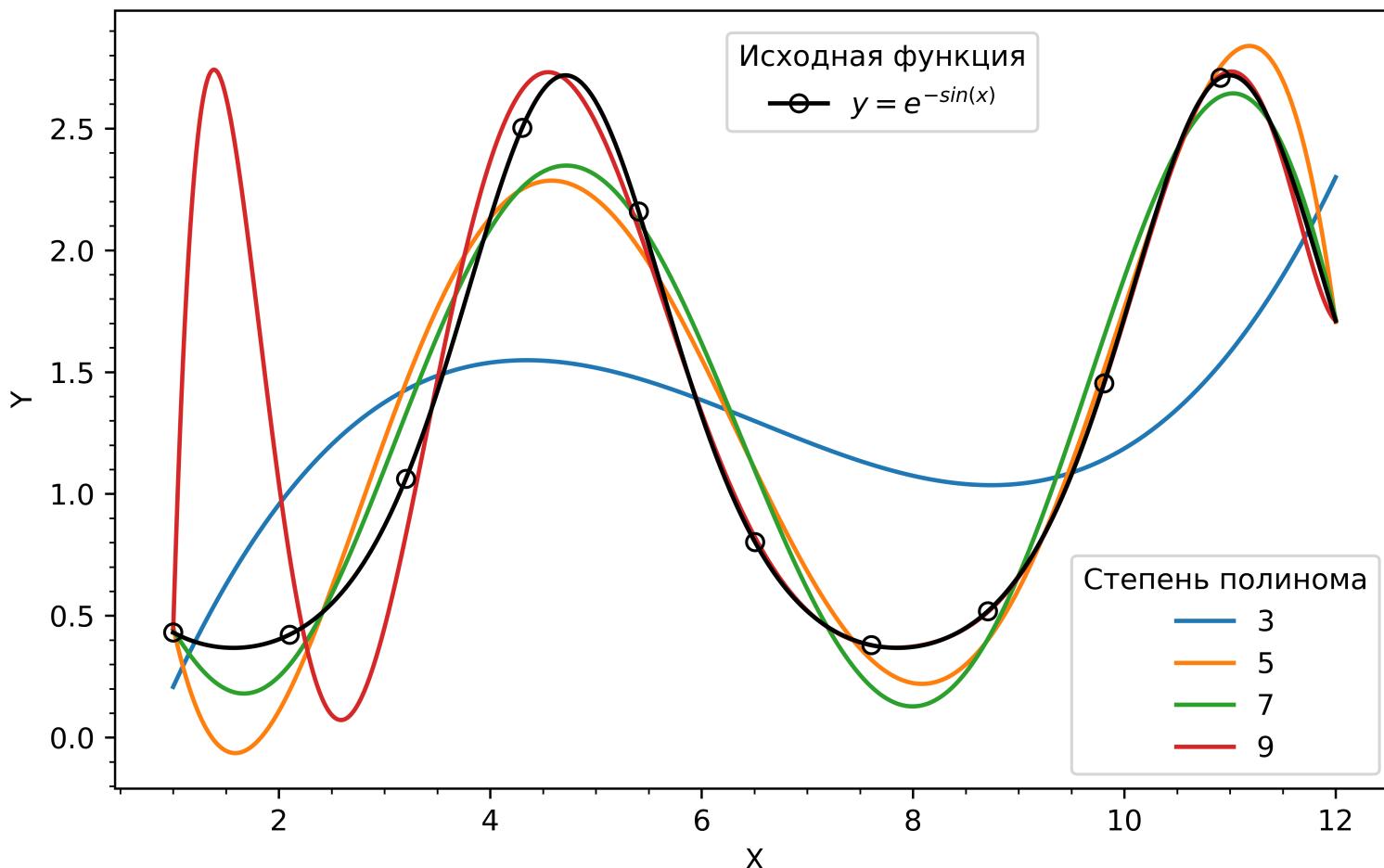
$$E = \sum_{i=1}^n [y_i - p(x_i)]^2$$

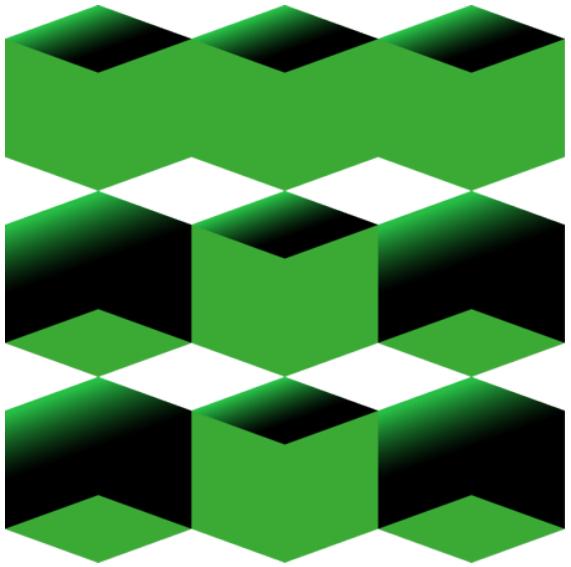
Рассмотрим функцию  $4y = e^{-\sin x}$  на отрезке  $[1; 12]$ .

```
>>> import numpy as np
>>> Polynomial = np.polynomial.Polynomial
>>> x = np.linspace(1, 12, 10)
>>> y = np.exp(-np.sin(x))
>>> p3 = Polynomial.fit(x, y, 3) # Создает кубический полином
>>> print(p3.coef)
[ 1.29952606 -0.96210884 -0.04580373  2.00818378]
```

## Метод `Polynomial.fit()`

Следует помнить о том, что для полинома высокой степени характерно большое количество локальных экстремумов, что создаст проблемы при аппроксимации исходных данных.





**Благодарю за  
внимание!**

Вячеслав Алексеевич Чузлов  
к.т.н., доцент ОХИ ИШПР

