

Научные вычисления в Python: NumPy и SciPy

NumPy и SciPy

- NumPy – основная библиотека для научных вычислений в Python: добавляет матрицы и другие многомерные массивы, а также функции для работы с ними
- SciPy – расширяет возможности NumPy: линейная алгебра, минимизация, интегрирование, специальные функции, преобразования Фурье и другое

Демонстрация возможностей NumPy

```
import numpy as np          # import NumPy
matrix=[[1, 3, 4], [2, 3, 5], [5, 7, 9]]
A=np.array(matrix)          # A is so called ndarray
b=np.array([4, 4, 4])
x=np.linalg.solve(A, b)     # Solve for Ax = b
print(x)                    # [-1.77777778 -0.44444444  1.77777778]
```

NumPy: ndarray

Класс ndarray (aka array):

Непрерывный одномерный сегмент памяти (`ndarray.data`), содержащий однородные данные, в сочетании со схемой индексации:

- `ndarray.ndim` – размерность массива (ранг матрицы)
- `ndarray.shape` – кортеж чисел определяющий размер по каждому измерению
- `ndarray.size` – число элементов в массиве
- `ndarray.dtype` и `ndarray.itemsize` – тип и размер в байтах одного элемента в массиве
- `ndarray.data` – буфер содержащий элементы массива

«Структура» ndarray

```
import numpy as np
a = np.array([[1,2,3],[4,5,6]])
print(a)

type(a)
print(a.ndim)
print(a.shape)
print(a.size)
print(a.dtype, a.itemsize)
```

list -> array
`[[1 2 3]`
`[4 5 6]]` печать без запятых
`<class 'numpy.ndarray'>`
размерность массива: 2
размер по каждому измерению: (2, 3)
полное число элементов: 6
тип элементов: `int64`, в байтах: 8

NumPy: способы создания ndarray

● преобразование: list → array

```
a = np.array([[1,2,3],[4,5,6]])
```

● задание последовательности

```
b = np.arange(0, 2, 0.5) # 0.5 это шаг: [0. 0.5 1. 1.5]
c = np.linspace(0, 2, 5) # 5 число разбиений: [0. 0.5 1. 1.5 2. ]
```

● все нули или единицы

```
z = np.zeros((5, 5)) # shape=(5,5) dtype=float64 по умолчанию
o = np.ones((3, 3), dtype=np.int64) # shape=(3,3) dtype=int64
```

● единичная (identity) матрица

```
I = np.identity(2) # shape=(2,2) || E = np.eye(2,3) # [[1. 0. 0.]
# [0. 1. 0.]
```

NumPy: изменение «формы» ndarray

● `reshape(new_shape)` – задание новой формы

```
a = np.arange(1,7).reshape((2,3))    # [[1 2 3]
                                       #  [4 5 6]]
```

● `ravel()` – приведение к 1D размерности

```
print(a.ravel())                      # [1 2 3 4 5 6]
```

● `transpose()` – транспонирование

```
print(a.transpose())                 # [[1 4] for 2D =a.T
                                       #  [2 5]
                                       #  [3 6]]
```

NumPy: индексы и срезы (slices)

- в многомерных массивах индексы разделяются запятыми
- срезы определяются так же как в списках: `start:stop:step`

```
a=np.arange(12).reshape((3,4)) # [[ 0  1  2  3]
                                #  [ 4  5  6  7]
                                #  [ 8  9 10 11]]

print(a[0,2])                  # 2           индекс,индекс
print(a[1,:-1])                # [4 5 6]     индекс,срез
print(a[0:2,2])                # [2 6]       срез,индекс
print(a[:,2,:2])               # [[ 0  2]     срез,срез
                                #  [ 8 10]]
```

- срез «ссылается» на данные в array: данные не копируются

```
b = a[:,3]      # [ 3  7 11]
b[1] = -7
print(b)        # [ 3 -7 11]
print(a[1,3])   # -7
```

```
a = [[ 0  1  2  3]
      [ 4  5  6  7]
      [ 8  9 10 11]]
```

NumPy: функции `view()` и `copy()`

- присваивание или передача в функцию не создает новых объектов `array`
- `view()` создает новый `array` который ссылается на те же самые данные
- срезы наиболее частый способ создания `view`-объектов: “shallow copy”
- метод `copy()` создает полностью новый объект: “deep copy”

```
x = np.array([[1,2],[3,4]]) # [[1 2]   Матрица для примеров
                          # [3 4]]
```

• `view()`

```
v = x.view()
v.flags.owndata # False
v[0]=5
print(x)        # [[5 2]
                  # [3 4]]
```

• `copy()`

```
c = x.copy()
c.flags.owndata # True
c[0,0]=5
print(x)        # [[1 2]
                  # [3 4]]
```


NumPy: операции с ndarray

- вычисления в NumPy векторизованы: **циклы по элементам не нужны**
- большинство операций выполняются поэлементно

● массив и скаляр

```
v = np.arange(1,5)          # [1 2 3 4]
print(1.1*v)                # [ 1.1  2.2  3.3  4.4]
print(v**2)                 # [ 1  4  9 16]
print(v<3)                  # [ True  True False False]
```

● два одинаковых массива

```
                                # v = [1 2 3 4]
w = np.array([2,3,5,7])        # [2 3 5 7]
print(v*w)                    # [ 2  6 15 28]
print(w**v)                   # [  2   9 125 2401]
```

- в `numpy` имеется встроенный набор математических функций, которые выполняются поэлементно

● Elementwise functions

```
a=np.array([2,4,6])
b=np.array([1,3,5])
print(np.sqrt(a))      # [ 1.41421356  2.          2.44948974]
print(np.exp(b))       # [ 2.71828183 20.08553692 148.4131591 ]
print(np.arctan2(a,b)) # [ 1.10714872  0.92729522  0.87605805]
```

👉 названия функций частично пересекаются с названиями функций в `math`

NumPy: умножение матриц: dot()

dot() – произведение двух матриц

- для 1D: скалярное произведение, для 2D: произведение матриц
- для размерности $n > 2$: сумма произведения элементов последней «оси» матрицы **A** на предпоследнюю «ось» матрицы **B**
- В python-3.5 (NumPy-1.10) появился оператор @ матричного умножения

```
x = np.array([[1,2],[3,4]])      # [[1 2]   [[5 6]
y = np.array([[5,6],[7,8]])      # [3 4]   [7 8]
z = np.array([-1,1])
```

❶ dot(): метод ndarray

```
x.dot(y) # [[19 22] =np.dot(x,y)
          #  [43 50]]
x.dot(z) # [1 1]    =np.dot(x,z)
```

❷ dot(): ф-я двух переменных

```
np.dot(y,x) # [[23 34] =y@x
              #  [31 46]]
np.dot(z,x) # [2 2]    =z@x
```

NumPy: трансляция (broadcasting)

- механизм для выполнения операций с матрицами разных размерностей
- меньшая матрица «транслируется» вдоль недостающих размерностей таким образом что бы получилась размерность большей матрицы

Простой пример:

```
a = np.arange(11,17).reshape((2,3)) # [[11 12 13]
                                     #  [14 15 16]]

b = np.array([2,4,6])                # [2 4 6]

print(a+b)                           # [[13 16 19]
                                     #  [16 19 22]]
```

Более сложный пример трансляции

```
print(a) # [[11 12 13]
          #  [14 15 16]]
```

```
c=np.array([1,3]) # [1 3]
```

```
print(a+c.T) # Error! c.T=[1 3]
```

```
print(a+c.reshape(2,1))
```

```
# [[12 13 14] c.reshape(2,1) = [[1]
```

```
# [17 18 19]]                    [3]]
```

```
print((a.T+c).T)           # [[12 13 14]
                           #  [17 18 19]]
```

NumPy: ввод-вывод


- сохранить в текстовый файл:

```
numpy.savetxt(fname, X, fmt='%.18e', delimiter=' ', newline='\n',  
              header='', footer='', comments='# ', encoding=None)
```

- прочесть из текстового файла:

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

```
arr = np.arange(1,51,dtype=np.int32).reshape((5,10))  
np.savetxt(fname='test_array.out', X=arr, fmt='%3d')  
read_arr = np.loadtxt(fname='test_array.out', dtype=np.int32)  
print(np.all(read_arr == arr))    # True : check arrays are the same
```

 в функциях ввода-вывода предусмотрены опции контроля типов данных, разделители, комментарии, заголовки

NumPy: линейная алгебра

Под-модуль `numpy.linalg`

```
import numpy as np
A = np.array([[3, 1], [1, 4]]) # [[3 1]
                                # [1 4]]
print(np.linalg.det(A))      # Matrix determinant: 11.000000000000002

invA = np.linalg.inv(A)      # Invers matrix: [[ 0.36363636 -0.09090909]
                                # [-0.09090909  0.27272727]]
print(np.dot(A, invA))       # Check: [[ 1.00000000e+00  0.00000000e+00]
                                # [-5.55111512e-17  1.00000000e+00]]

L = np.linalg.cholesky(A)    # Cholesky decomposition: [[1.73205081 0.
                                # [0.57735027 1.91485422]]
print(L.dot(L.T))            # Check: [[3. 1.]
                                # [1. 4.]
```

... продолжение

```
egVal, egVec = np.linalg.eig(A)      # Eigenvalues and Eigenvectors:
    # eig() returns tuple; the first element is the eigen values
    # and the second one is a matrix with eigen vectors
    # column eigenvector[:,i] corresponds to the eigenvalue[i]
print(egVal)                          # [2.38196601 4.61803399]
print(egVec)                          # [[-0.85065081 -0.52573111]
    # [ 0.52573111 -0.85065081]]
print(A.dot(egVec[:,0])/egVal[0])    # Check: [-0.85065081  0.52573111]
print(A.dot(egVec[:,1])/egVal[1])    # [-0.52573111 -0.85065081]
```

- 👉 Пакет `scipy.linalg` имеет дополнительные функции
- 👉 Модули линейной алгебры в NumPy и SciPy находятся в развитии
- 👉 Функции линейной алгебры используют библиотеки BLAS и LAPACK для обеспечения эффективной реализаций стандартных алгоритмов линейной алгебры

SciPy: численное интегрирование

$$\int_0^1 \sin(x)/x \, dx = Si(1) \sim 0.946083$$

```
import numpy as np
from scipy.integrate import quad
res,err = quad(lambda x: np.sin(x)/x, 0,1)
print(f'{res:.6f} estimated error {err:.1e}')
# 0.946083 estimated error 1.1e-14
```

$$\text{Двойной интеграл: } \int_{x=\pi}^{2\pi} \int_{y=0}^{\pi/2} y \sin(x) + x \cos(y) \, dx dy = \frac{5}{4}\pi^2$$

```
import numpy as np
from scipy.integrate import dblquad
res,err = dblquad(lambda y,x: y*np.sin(x)+x*np.cos(y), # integrand(y,x)
                  np.pi, 2*np.pi,                    # limits for x
                  0, np.pi/2)                          # limits for y (may be func(x))
print('{0:.3f} estimated error {1:.0e}'.format(res,err))
# 12.337 estimated error 1e-13
```

SciPy: статистика

Генерация случайных чисел по заданному распределению

```
import numpy as np
from scipy import stats
Nmax = 1_000_000
gs = stats.norm(0,1).rvs(Nmax)      # normal distribution N(0,1)
print(f'Mean: {np.mean(gs):10.6f}')
print(f'Std.devuation: {np.std(gs):10.6f}')
print(f'Median: {np.median(gs):10.6f}')

gs = stats.uniform().rvs(Nmax)      # uniform distribution on [0,1]
```

Normal

Mean: 0.000743
Std.deviation: 0.999652
Median: 0.000090

Uniform

Mean: 0.500514
Std.deviation: 0.288633
Median: 0.500337

Вычисление характеристик распределения

```
n, min_max, mean, var, skew, kurt = stats.describe(gs)
print('Number of elements: {0:d}'.format(n))
print('Min: {0:8.6f} Max: {1:8.6f}'.format(min_max[0], min_max[1]))
print('Mean: {0:8.6f}'.format(mean))
print('Variance: {0:8.6f}'.format(var))
print('Skew : {0:8.6f}'.format(skew))
print('Kurtosis: {0:8.6f}'.format(kurt))
```

Normal

Number of elements: 1000000
Min: -4.805859 Max: 4.654846
Mean: -0.000008
Variance: 1.000193
Skew : 0.000920
Kurtosis: -0.002568

Uniform

Number of elements: 1000000
Min: 0.000000 Max: 0.999998
Mean: 0.499965
Variance: 0.083471
Skew : -0.001275
Kurtosis: -1.201883

SymPy: символьные вычисления

SymPy: система компьютерной алгебры, примеры:

```
import sympy as sym      # подключение
x = sym.Symbol('x')      # описание символов x и y
y = sym.Symbol('y')
a = sym.Rational(1, 2)   # рациональное число 1/2

t=sym.expand((x+a*y)**3) #  $x^3 + 3x^2y/2 + 3xy^2/4 + y^3/8$ 
sym.simplify(t/(x/a+y)) # упрощение:  $x^2/2 + xy/2 + y^2/8$ 

sym.limit(1/t,x,sym.oo)  # предел  $\lim_{x \rightarrow \infty} (1/t)$ : 0
sym.series(sym.tan(x),x) # Тэйлор:  $x + x^3/3 + 2x^5/15 + O(x^6)$ 
sym.diff(t,x)            # производная:  $3x^2 + 3xy + 3y^2/4$ 
sym.integrate(t,y)       #  $x^3y + 3x^2y^2/4 + xy^3/4 + y^4/32$ 
sym.integrate(t,(x,0,1)) #  $y^3/8 + 3y^2/8 + y/2 + 1/4$ 
```

Полезные ресурсы в NumPy, SciPy и Python

Вся актуальная документация на английском

- [Scipy Lecture Notes](#)
- [Real Python Tutorials](#)
- [Python Data Science Handbook](#)