Научное программирование в Python: библиотека NumPy

А.Г. Трофимов
к.т.н., доцент, НИЯУ МИФИ

lab@neuroinfo.ru
http://datalearning.ru

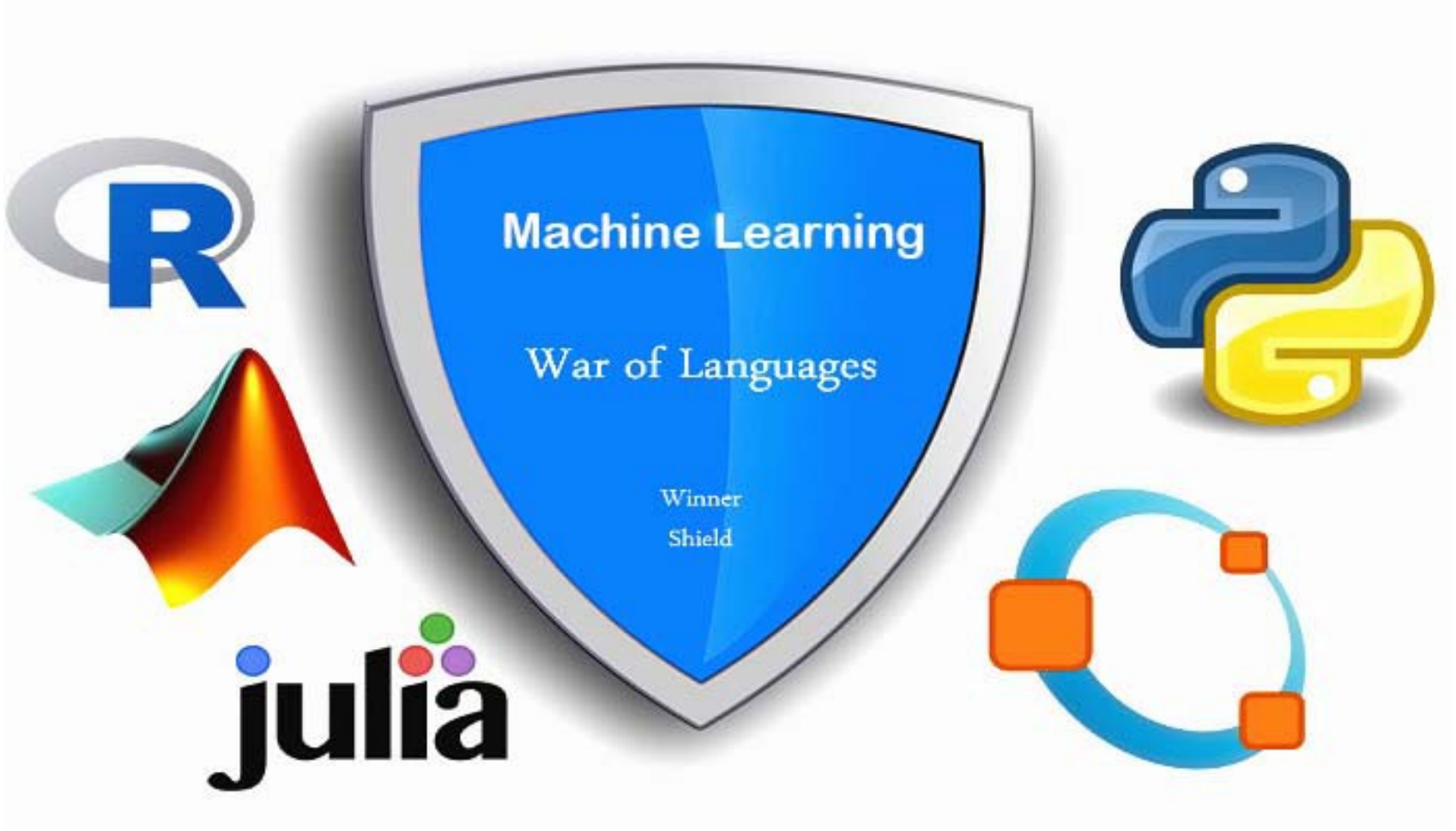
Курс “Программирование в Python”

Май 2018

Языки научного программирования

- **MATLAB**
  Коммерческий, дорогостоящий, труднорасширяемый
- **Python**
  Открытый, свободно распространяемый, язык общего назначения
- **R**
  Язык ориентирован на статистический анализ и визуализацию
- **Julia**
  Относительно новый, высокая скорость вычислений
- **Octave**
  Свободно распространяемый, совместимый с MATLAB язык

Языки научного программирования



Научный Python

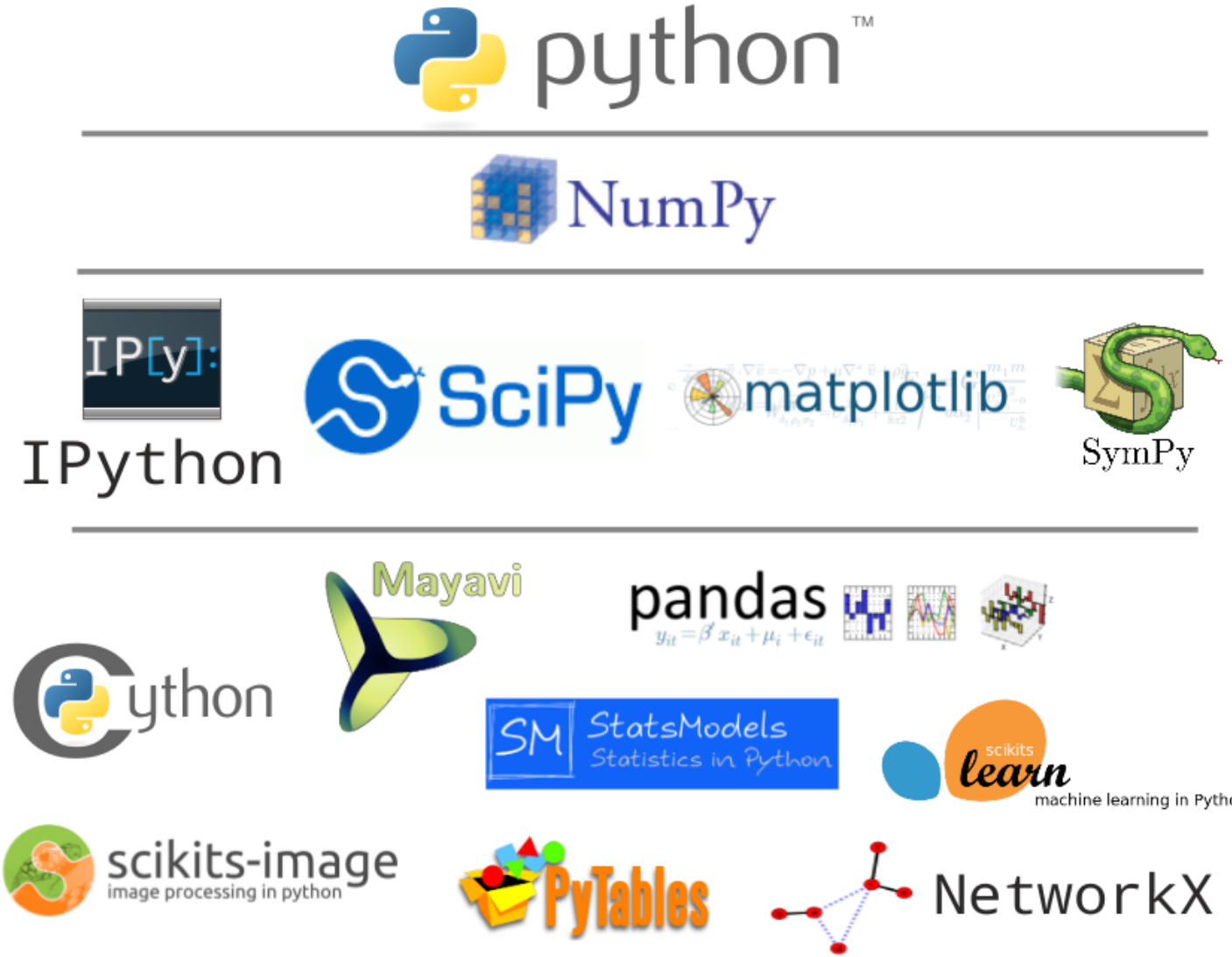
Научные вычисления в Python основаны на использовании библиотек:

- **NumPy**
  Поддержка больших многомерных массивов и матриц, вместе с библиотекой высокоуровневых (и очень быстрых) математических функций для операций с ними
- **SciPy**
  Коллекция алгоритмов для научных и инженерных расчетов, включающая цифровую обработку сигналов, оптимизацию, статистику и пр.
- **matplotlib**
  Библиотека функций визуализации данных

Специализированные библиотеки для научного Python

- **pandas**
  Высокоуровневая библиотека для обработки и анализа данных
- **SymPy**
  Библиотека символьных вычислений
- **scikit-image**
  Коллекция алгоритмов для обработки изображений
- **scikit-learn**
  Коллекция алгоритмов машинного обучения
- **statsmodels**
  Библиотека для оценивания статистических моделей и статистического анализа
- ...

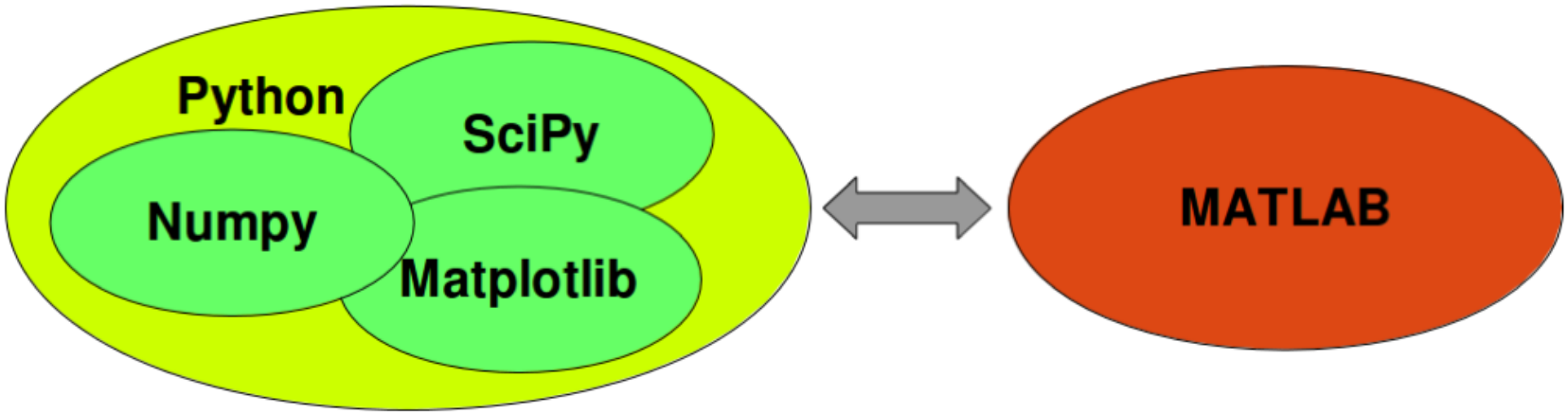
Стек библиотек Python



Python vs MATLAB

Python в комбинации с NumPy, SciPy и matplotlib может рассматриваться как открытая и свободная альтернатива MATLAB

Библиотека Matplotlib реализует возможности визуализации, подобные MATLAB



Массивы в NumPy

Подключение NumPy

```
import numpy as np
```

Основным объектом NumPy является массив элементов (ndarray), как правило, чисел

```
a = np.array([1,2,3]) # from list/tuple/range
type(a) # numpy.ndarray
a.tolist() # convert array to list

a = np.array([range(1,5),range(5,9)]) # 2D-array
print(a)
# [[1 2 3 4]
#  [5 6 7 8]]
```

Создание массивов

```
# array of complex numbers
b = np.array([[1.5,2,3],[4,5,6]], dtype=np.complex)
# [[1.5+0.j  2. +0.j  3. +0.j]
#  [4. +0.j  5. +0.j  6. +0.j]]

np.zeros((3,6)) # zero matrix
np.ones(10) # ones matrix
np.eye(5) # identity matrix
np.empty((3,6)) # reserve memory for matrix

np.arange(10,30,5) # array([10,15,20,25])

np.linspace(0,1,5) # array([0,0.25,0.5,0.75,1])
np.logspace(0,3,4) # array([1,10,100,1000])
```

Размерность массива

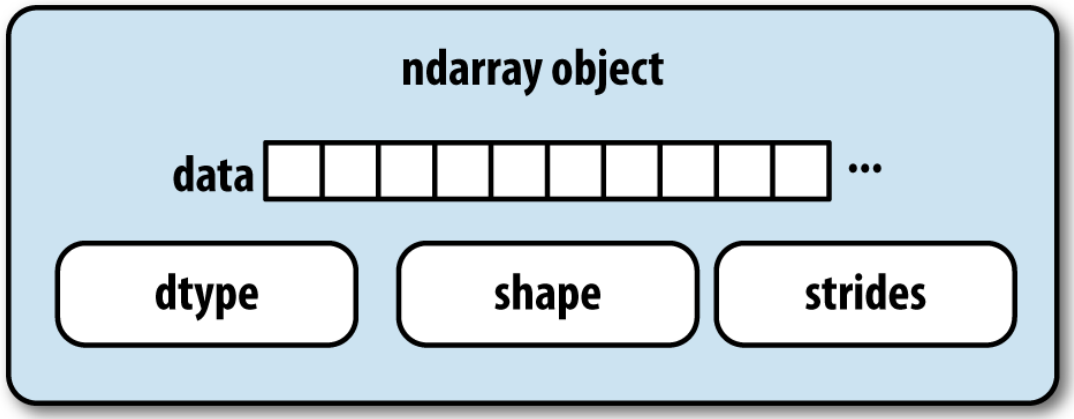
```
# zero-dimensional arrays
a = np.array(5) # scalar
a.ndim # 0

# one-dimensional arrays
a = np.array([1,2,3]) # vector
a.ndim # 1

# multi-dimensional arrays
a = np.array([[1,2,3],[4,5,6]]) # matrix
a.ndim # 2
a = np.array([ [[111,112],[121,122]],
               [[211,212],[221,222]],
               [[311,312],[321,322]] ])
np.ndim(a) # 3
```

Массив NumPy

Массив NumPy – это область памяти (data buffer), содержащая данные, и информация, описывающая способ интерпретации этих данных (view)

Массив NumPy:

- Указатель на область памяти с данными
- Объект `dtype`, определяющий тип элементов массива
- Кортеж `shape`, определяющий форму массива
- Кортеж `strides`, определяющий шаг (в байтах) вдоль каждого измерения массива при увеличении индекса на 1



Создание представлений (view) массива

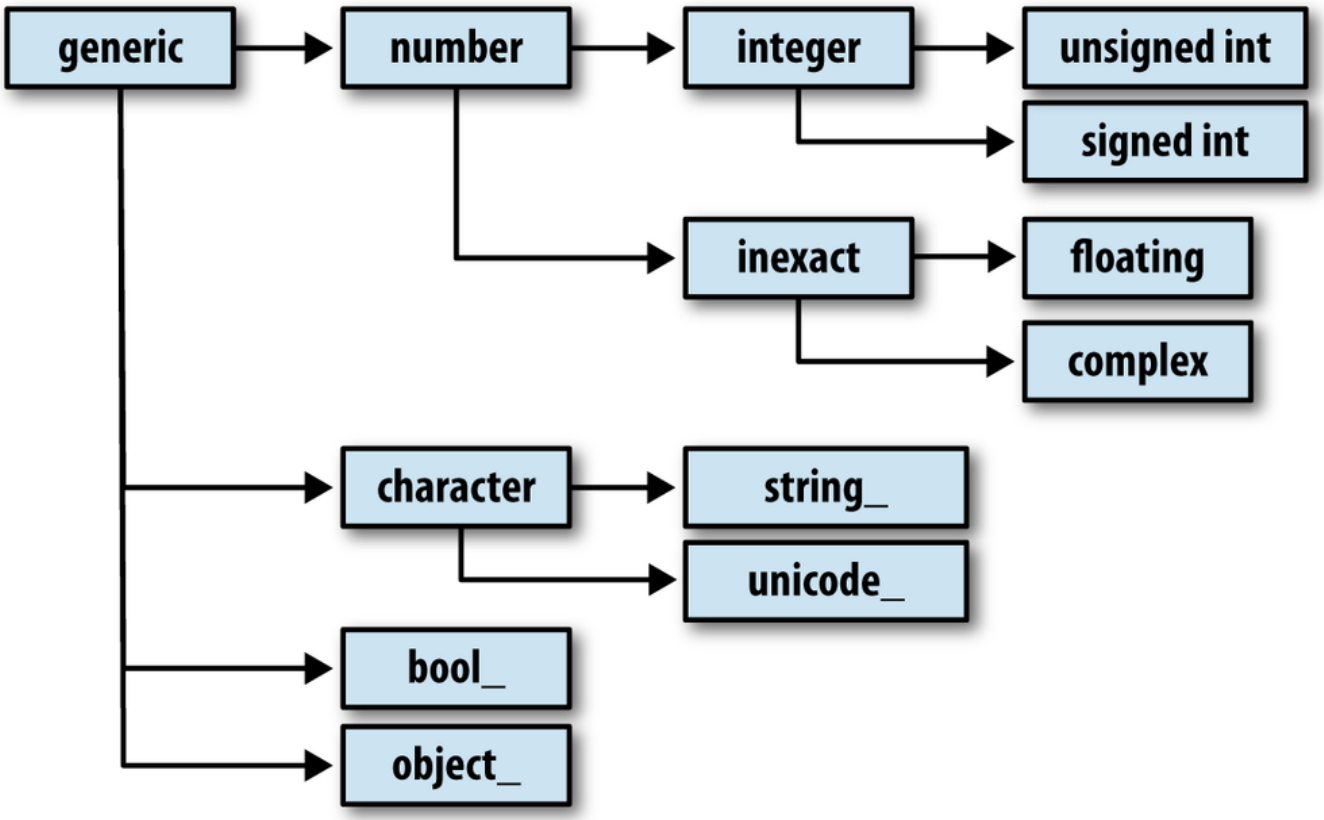
Создание нового представления:

```
a = np.array(range(8)) # [0 1 2 3 4 5 6 7]
b = a.view() # create new view
b.resize((2,4))
b[0,0] = 15
print(a) # [15 1 2 3 4 5 6 7]

c = a.reshape((2,4)) # create new view
c[0,0] = 20
print(a) # [20 1 2 3 4 5 6 7]

a.resize((2,4))
print(a)
[[20 1 2 3]
 [4 5 6 7]]
```

Копирование массива

```
a = np.array(range(8))

b = a.copy()
b = np.array(a,copy=True) # the same
b[0,0] = 15
print(b) # [15  1  2  3  4  5  6  7]
print(a) # [0  1  2  3  4  5  6  7]

c = np.asarray(a) # new view, no copy
c[0,0] = 15 # a affected

# deep copy
import copy
b = copy.deepcopy(a)
```

Типы элементов массива (dtype)



Примеры:

`int8, int16, uint8, unit16, float64, complex128, bool, str, ...`

Структурированные массивы

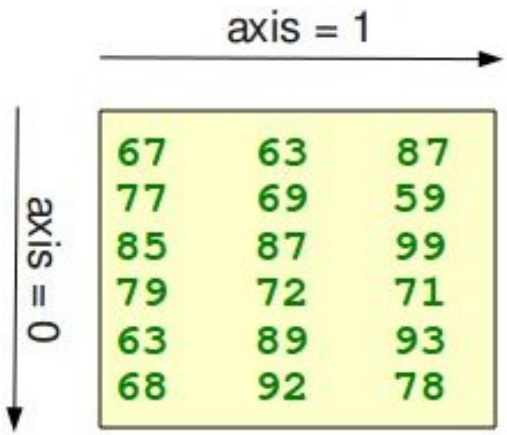
Структурированные массивы (structured arrays) позволяют хранить в виде массива данные разных типов

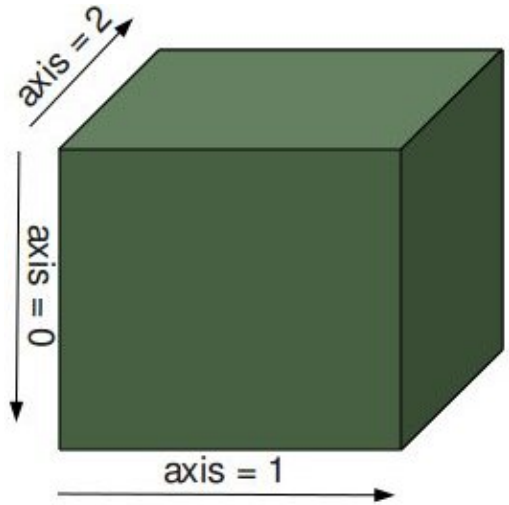
Элемент структурированного массива – аналог структуры языка C

```
dtype = [('x',np.float64),('y',np.int32)]
a = np.array([(1.5,6),(np.pi,-2)],dtype=dtype)

print(a[0]) # (1.5,6)
print(a[0]['y']) # 6
print(a['x']) # [1.5,3.1415] (array)

a[0] = (2,3)
a[1]['x'] = 1.5
```

Форма массива (shape)

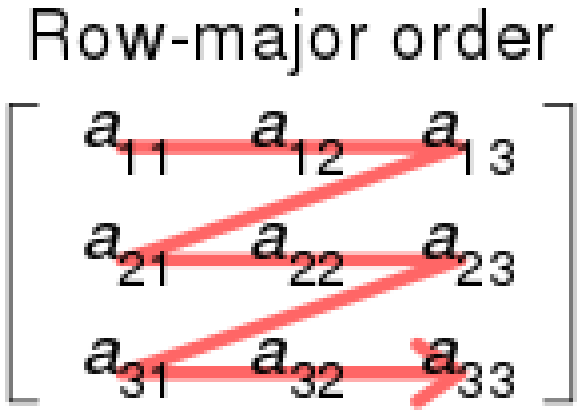
```
a = np.array(5) # scalar
a.shape # ()
a = np.array([1,2,3]) # vector
a.shape # (3,)
a = np.array([[1,2,3],[4,5,6]]) # matrix
a.shape # (2,3)
```



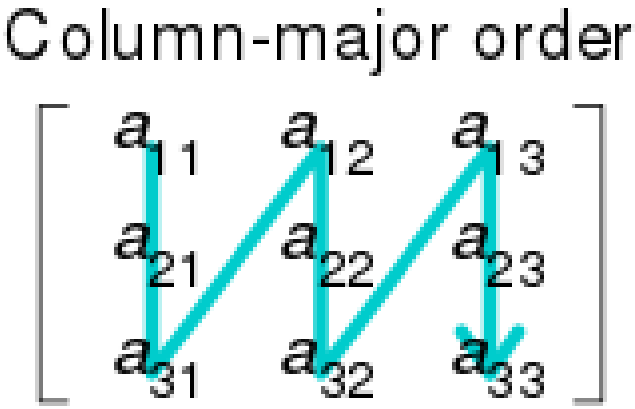


Стратегии хранения массивов в памяти

- **Стратегия C** (row-major order)
  Старшие измерения изменяются первыми



- **Стратегия Fortran** (column-major order)
  Старшие измерения изменяются последними



По умолчанию массивы NumPy создаются в row-major order

Итерирование массивов

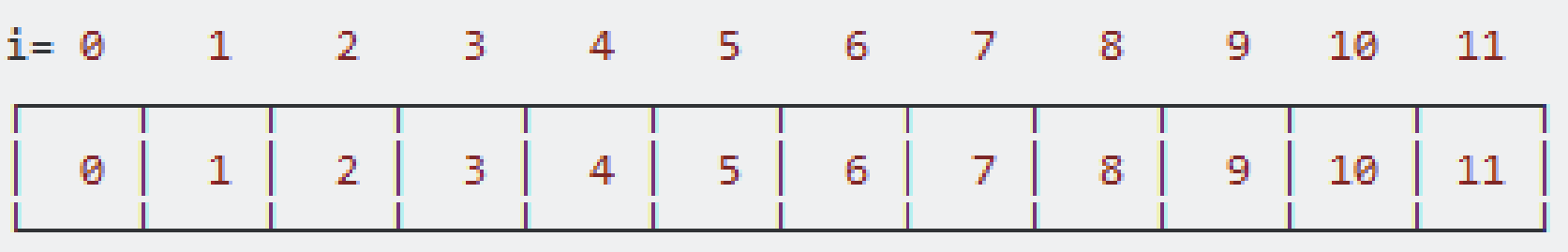
Для итерирования массивов используются итераторы `nditer` и `ndenumerate`

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

for x in np.nditer(a): # default C-order iterating
    print(x)
# 0 1 ... 11

for index,v in np.ndenumerate(a):
    print(index,v)
# (0,0) 0 ... (2,3) 11
```

Изменение формы массива

```
a = np.arange(12) # 1D array

a.dtype # dtype('int32')
a.shape # (12,)
a.strides # (4,)

a[0] # 0
a[11] # 11
a[12] # error: out of range
```

Индексы элементов массива a:



Изменение формы массива

```
b = a.reshape((3,4), order='C') # create new view
c = a.reshape((3,4), order='F') # create new view
```

Индексы элементов массива b:

```
i= 0   0   0   0   1   1   1   1   2   2   2   2
j= 0   1   2   3   0   1   2   3   0   1   2   3
```

0	1	2	3	4	5	6	7	8	9	10	11

Индексы элементов массива c:

```
i= 0   1   2   0   1   2   0   1   2   0   1   2
j= 0   0   0   1   1   1   2   2   2   3   3   3
```

0	1	2	3	4	5	6	7	8	9	10	11

Изменение формы массива

```
d = a.reshape((12,1)) # default order='C'
d[10,0] # 10 (d is 2D array)
e = d.reshape((12,)) # 1D array
```

Индексы элементов массива d:

```
i= 0   1   2   3   4   5   6   7   8   9  10  11
j= 0   0   0   0   0   0   0   0   0   0   0   0
```

0	1	2	3	4	5	6	7	8	9	10	11

Индексы элементов массива e:

```
i= 0   1   2   3   4   5   6   7   8   9  10  11
```

0	1	2	3	4	5	6	7	8	9	10	11

Изменение формы массива

```
f = a.reshape((1,2,1,6,1)) # 5D array, C-order
f[0,1,0,0,0] # 6
```

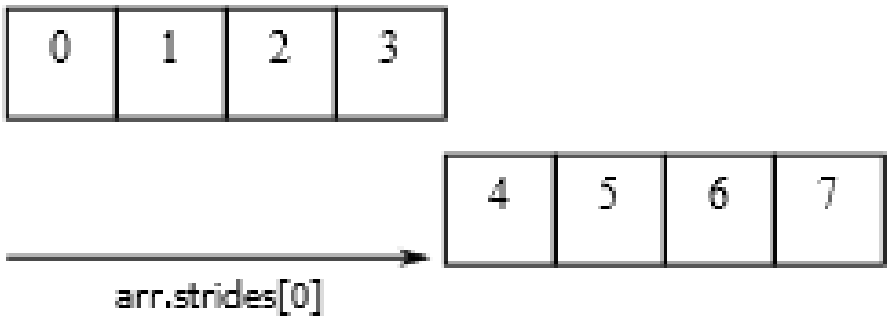
Индексы элементов массива `f`:

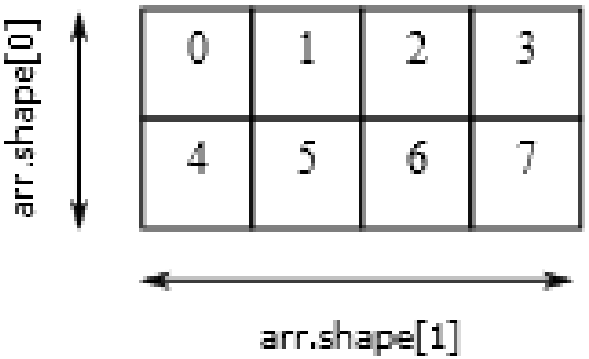
```
i= 0   0   0   0   0   0   0   0   0   0   0   0
j= 0   0   0   0   0   0   1   1   1   1   1   1
k= 0   0   0   0   0   0   0   0   0   0   0   0
l= 0   1   2   3   4   5   0   1   2   3   4   5
m= 0   0   0   0   0   0   0   0   0   0   0   0
```

0	1	2	3	4	5	6	7	8	9	10	11

Страйды массива (strides)

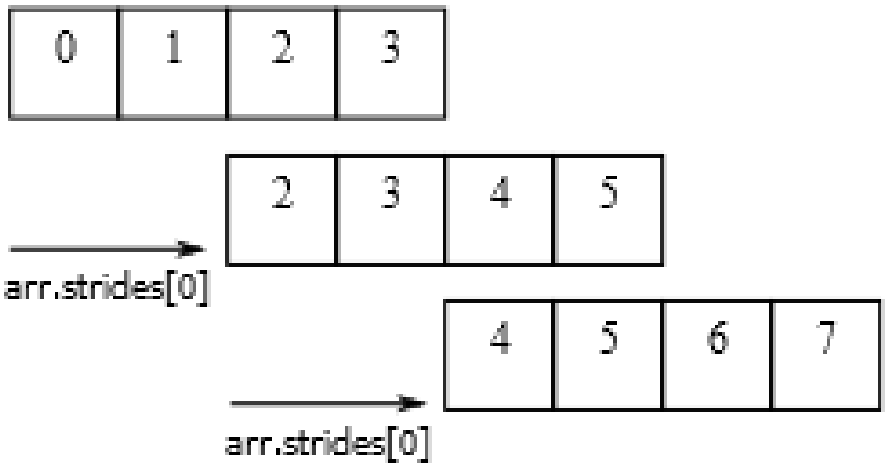
```
a = np.arange(8,dtype=np.uint8)
a.shape # (8,)
a.strides # (1,)

a.shape = (2,4)
a.strides = (4,1)
a[0,1] # 1 (offset=1 byte)
a[1,1] # 4 (offset=4+1 bytes)
```

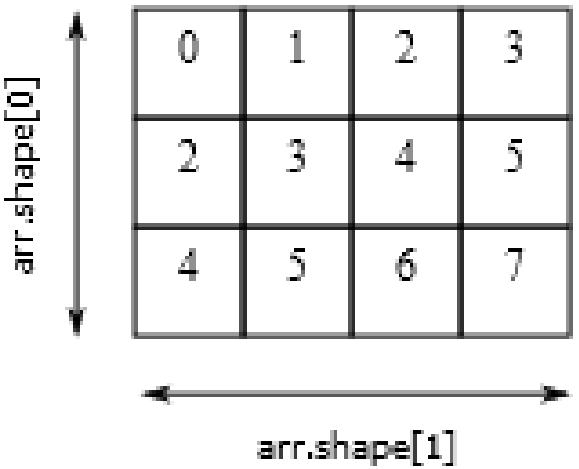




Изменение формы и страйдов массива

```
a.shape = (3,4)
a.strides = (2,1)

a[0,1] # 1 (offset=1 byte)
a[1,1] # 4 (offset=2+1 bytes)
a[2,1] # 4 (offset=2*2+1 bytes)
```





Изменение осей и транспонирование массива

Функции `transpose` и `swapaxes` создают представления массива

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

b = a.transpose()
# [[0,4,8],
#  [1,5,9],
#  [2,6,10],
#  [3,7,11]]
b = a.swapaxes(1,0) # the same
b = a.T # the same
```

Векторизация массива

Векторизация массива (flattening, raveling) – создание одномерного массива из многомерного

Функция `ravel` создаёт представление массива, функция `flatten` – копию массива

```
a = np.arange(8)
b = a.reshape((2,4))
c = b.ravel() # C-order is default
c[0] = 12
print(c) # [12,1,2,3,4,5,6,7]
print(a) # [12,1,2,3,4,5,6,7]
print(b) # [[12,1,2,3],[4,5,6,7]]

c = b.flatten() # C-order is default
c[0] = 21
print(c) # [21,1,2,3,4,5,6,7], a,b not affected
```

Срезы массива (slices)

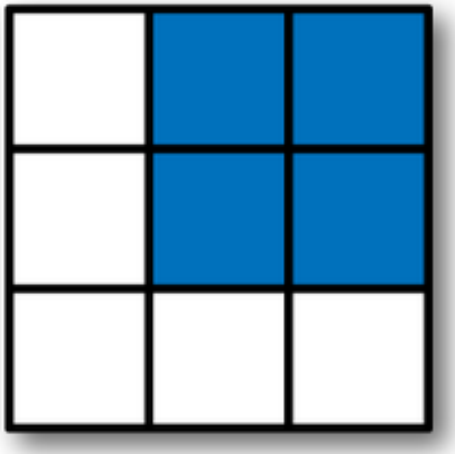
Срезы (slices) – это представления массива

```
a = np.array(range(8))

b = a[:] # [0 1 2 3 4 5 6 7]
b = a[:4] # [0 1 2 3]
b = a[3:] # [3 4 5 6 7]
b = a[::2] # [0 2 4 6]
b = a[2:3] # [2] (numpy.ndarray)
b = a[2] # 2 (int, immutable)
c = a[2:5].copy() # new array [2,3,4]

b = a[2:5] # [2,3,4]
b[0] = 15
print(b) # [15 3 4]
print(a) # [0 1 15 3 4 5 6 7]
```

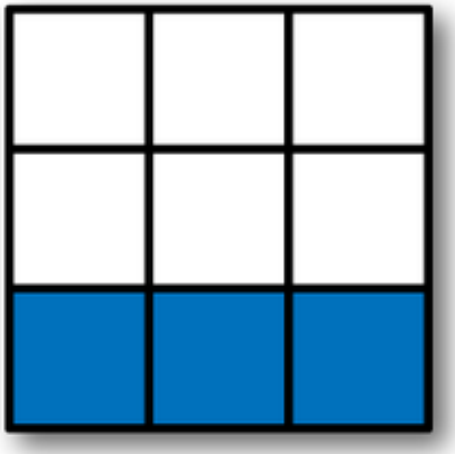
Срезы многомерных массивов

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

b = a[0] # 2D [[1,2,3],[4,5,6]]
c = a[0][1] # 1D [4,5,6]]
c = a[0,1] # the same
d = a[0][1][1] # 0D 5 (int)
d = a[0,1,1] # the same

e = a[i1:j1:s1,i2:j2:s2,i3:j3:s1] # slice
e = a[:1,:,::2] # 3D [[[1,3],[4,6]]]
e = a[0,:,::2] # 2D [[1,3],[4,6]]
e = a[0,0,::2] # 1D [1,3]
```

Срезы многомерных массивов. Примеры

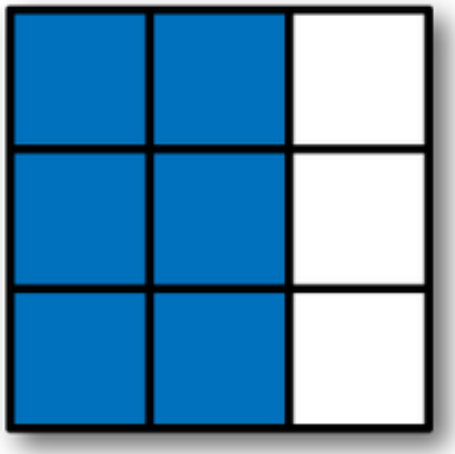


Expression	Shape
`arr[:2, 1:]`	`(2, 2)`

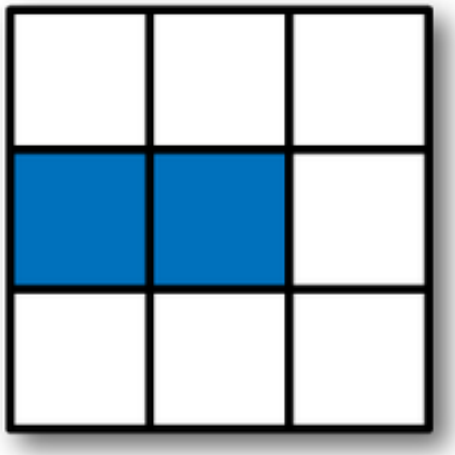


Expression	Shape
`arr[2]`	`(3,)`
`arr[2, :]`	`(3,)`
`arr[2:, :]`	`(1, 3)`

Срезы многомерных массивов. Примеры



Expression	Shape
`arr[:, :2]`	`(3, 2)`



Expression	Shape
`arr[1, :2]`	`(2,)`
`arr[1:2, :2]`	`(1, 2)`

Логические индексы

В качестве индексов могут быть использованы целочисленные или логические массивы (fancy indexing). При использовании fancy indexing всегда создаётся копия массива

```
a = np.arange(8) # [0,1,2,3,4,5,6,7]
b = a>3 # [F,F,F,F,T,T,T,T]
c = a[a>3] # [4,5,6,7]
c[0] = 15 # [15,5,6,7]
print(a) # [0,1,2,3,4,5,6,7]

d = a[(a>2)&(a<5)] # [3,4]
d = a[(a<=1)|(a>5)] # [0,1,6,7]
d = a[a==2] # [2]
d = a[a!=2] # [0,1,3,4,5,6,7]
d = a[~(a==2)] # [0,1,3,4,5,6,7]
```

Функция where

Для выбора элементов из двух массивов по заданному условию используется функция where – векторизованный аналог выражения x **if** condition **else** y (создаётся копия массива)

```
a = np.array([1,2,3,4,5])
b = np.array([11,12,13,14,15])
mask = np.array([True,True,False,False,True])

c = [(x if m else y) for x,y,m in zip(a,b,mask)]
print(c) # [1,2,13,14,5] (list)

c = np.where(mask,a,b)
print(c) # [1,2,13,14,5] (array)

c = np.where(a>3,2,-2) # [-2,-2,-2,2,2]
```

Целочисленные массивы как индексы

В качестве индексов массива могут быть использованы массивы с целочисленными элементами

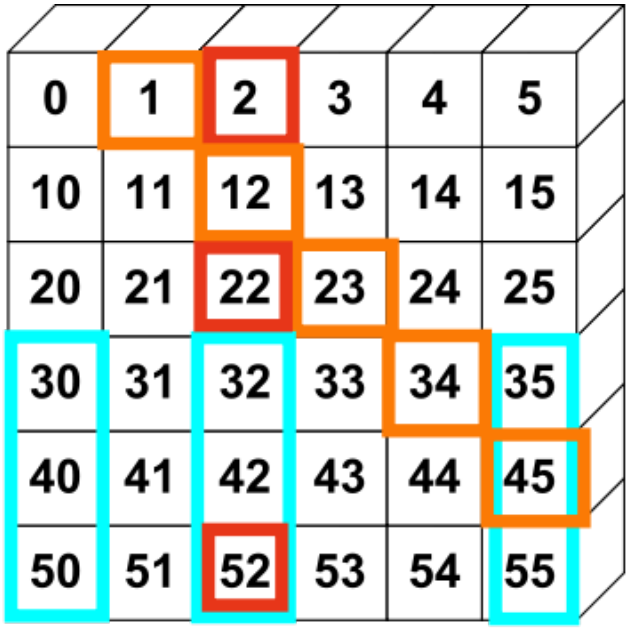
```
a = np.arange(8)**2 # [0,1,4,9,16,25,36,49]
b = np.array([2,4]) # [2,4]
c = a[b] # [4,16] (copy)

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

# select rows 0,2
b = a[[0,2]] # [[0,1,2,3],[8,9,10,11]]
```

Fancy Indexing. Примеры

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

# multiple index arrays
b = a[[0,2,2],[1,2,3]] # [1,10,11]
# a[0,1], a[2,2], a[2,3] selected
a[(1,1)] # 5 (int)

# select rows 0,2 and columns 3,1
b = a[[0,2]][:,[3,1]]
# [[3,1],
#  [11,9]]
```

Fancy Indexing. Примеры

```
>>> a[(0,1,2,3,4),(1,2,3,4,5)]
array([ 1, 12, 23, 34, 45])

>>> a[3:,[0, 2, 5]]
array([[30, 32, 35],
       [40, 42, 45]])
       [50, 52, 55]])

>>> mask = array([1,0,1,0,0,1],
                 dtype=bool)
>>> a[mask,2]
array([2,22,52])
```



В качестве альтернативы fancy indexing могут быть использованы функции `put` и `take`:

```
a = np.arange(8)**2 # [0,1,4,9,16,25,36,49]
b = a.take([2,4]) # [4,16] (copy)
a.put([2,4],[21,22]) # [0,1,21,9,22,25,36,49]
```

Ненулевые элементы массива

```
a = np.array([[0,2,3,0,1],
              [1,0,0,7,0],
              [5,0,0,1,0]])
idxs = a.nonzero() # return tuple of arrays
# ([0,0,0,1,1,2,2], [1,2,4,0,3,0,3])
i,j = a.nonzero() # return 2 arrays
b = np.transpose(idxs) # b.shape=(7,2)

# use fancy indexing
b = a[idxs] # [2,3,1,1,7,5,1]

# indices of elements where condition is True
idxs = np.nonzero(a>2) # ([0,1,2],[2,3,0])
```

Конкатенация массивов

```
a = np.array([[1,2,3],[4,5,6]])
b = np.array([[7,8,9],[10,11,12]])
c = np.concatenate([a,b],axis=0)
# [[1,2,3],
#  [4,5,6],
#  [7,8,9],
#  [10,11,12]]
c = np.vstack([a,b]) # the same

d = np.concatenate([a,b],axis=1)
# [[1,2,3,7,8,9],
#  [4,5,6,10,11,12]]
d = np.hstack([a,b]) # the same
```

Разбиение массивов

```
a = np.arange(10).reshape((5,2))

b1,b2,b3 = np.split(a,[1,4]) # 1,4 - where to slice
# b1: [0,1]
# b2: [[2,3],[4,5],[6,7]]
# b3: [8,9]
b1,b2,b3 = np.vsplit(a,[1,4]) # the same

b1,b2 = np.hsplit(a,[1])
# b1: [[0],[2],[4],[6],[8]]
# b2: [[1],[3],[5],[7],[9]]
```

Дублирование массивов

```
a = np.array([[1,2],[3,4]])
b = np.tile(a,2) # the same as np.tile(a,(1,2))
# [[1,2,1,2],
#  [3,4,3,4]]

c = np.tile(a,(2,1))
# [[1,2],
#  [3,4],
#  [1,2],
#  [3,4]]

d = np.tile(a,(3,4)) # shape=(6,8)

e = np.arange(4).repeat(2) # [0,0,1,1,2,2,3,3]
```

Арифметические операции с массивами

Все арифметические операции со скалярами, а также между массивами одинаковых размеров, выполняются поэлементно

```
a = np.arange(5) # [0,1,2,3,4]
b = np.array([1,1,1,2,2])
c = a+b # [1,2,3,5,6]
c = a-b # [-1,0,1,1,2]
c = a*b # [0,1,2,6,8]
c = a/b # [0,1,2,1.5,2]
c = a**2 # [0,1,4,9,16]

c = a+2 # [2,3,4,5,6]
c = a*2 # [0,2,4,6,8]
c = 1/(a+1) # [1,0.5,0.33,0.25,0.2]
```

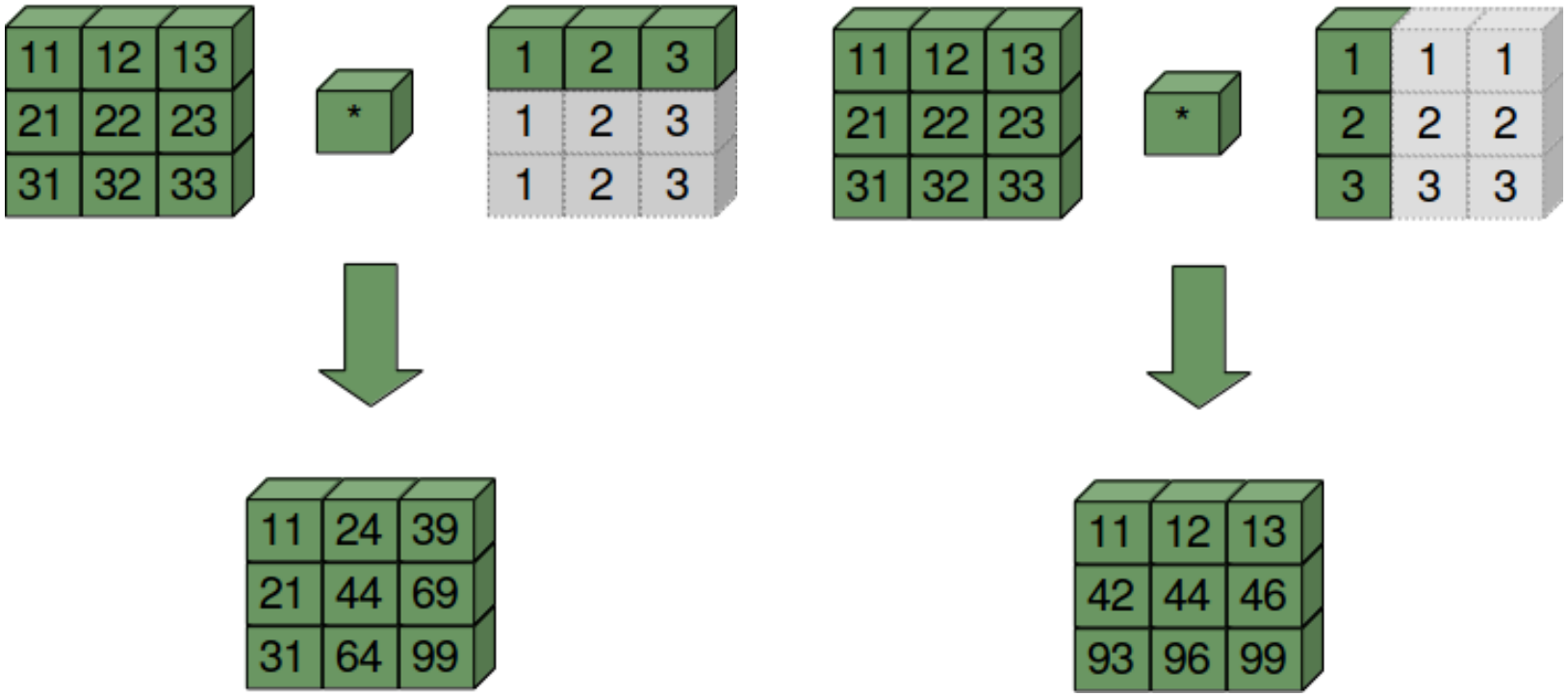
Broadcasting

Broadcasting – механизм приведения массива меньшей размерности к размеру большего массива для возможности выполнения арифметической операции с ними

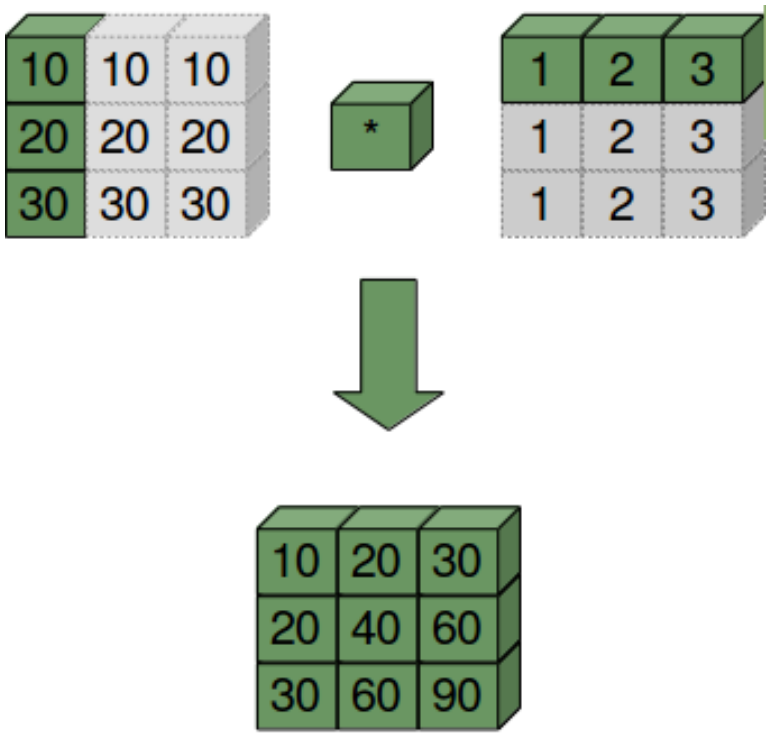
Broadcasting позволяет выполнять арифметические операции с массивами разных размеров

```
a = np.array([[11,12,13],[21,22,23],[31,32,33]])
b = np.array([1,2,3])
b_col = b[:,np.newaxis] # [[1],[2],[3]]

c = a*b # ok, broadcasting used
d = a*b_col # ok, broadcasting used
e = b*b_col # ok, broadcasting used
e = b_col*b # ok, result is the same
```

Broadcasting. Иллюстрация



Условие применимости broadcasting'a



Условие применимости broadcasting'a:

К двум массивам может быть применён broadcasting, если для каждого измерения (начиная с последнего) размеры массивов совпадают либо размер одного из массивов вдоль этого измерения равен 1

```
a = np.arange(60).reshape((3,4,5))
b = a+np.arange(5) # ok
b = a+np.arange(3) # error: broadcast failed
b = a+np.arange(3)[:,np.newaxis,np.newaxis] # ok
```

Присваивание с использованием broadcasting'а

Broadcasting может быть применён для присваивания элементов массиву

```
a = np.zeros((4,3))
a[:] = 5 # broadcast to all elements

b = np.array([1,2,3,4])
a[:] = b # error: broadcast failed
a[:] = b[:,np.newaxis] # ok
# [[1,1,1],[2,2,2],[3,3,3],[4,4,4]]

a[:2] = [[11],[12]]
# [[11,11,11],[12,12,12],[3,3,3],[4,4,4]]
```

Универсальные функции

Универсальные функции (universal function, или ufunc) – это функции, выполняемые для массивов NumPy поэлементно

```
a = np.arange(5) # [0,1,2,3,4]
b = np.arange(5)[::-1] # [4,3,2,1,0]
np.sqrt(a)
np.exp(a)
np.log(a)
np.abs(a)
np.sin(a)
np.isnan(a)
np.isinf(a)

np.maximum(a,b) # [4,3,2,4,3]
np.minimum(a,b) # [0,1,2,1,0]
```

Векторно-матричные операции

```
a = np.array([2,5,1,5])
b = np.array([7,2,15,3])

np.dot(a,b) # dot product
np.unique(a)
np.intersect1d(a,b)
np.union1d(a,b)
np.in1d(a,b)

a = np.array([[1,2],[3,4]])
b = np.array([[2,0],[0,2]])
np.dot(a,b) # matrix multiplication
np.matmul(a,b) # matrix multiplication
a@b # matrix multiplication
```

Математические и статистические методы массивов

Класс ndarray включает методы расчёта среднего значения элементов массива, среднеквадратичного отклонения, кумулятивных сумм, поиск минимального и максимального элементов и пр.

```
a = np.arange(5) # [0,1,2,3,4]
a.sum()
a.mean()
a.std()
a.var()
a.min()
a.argmin()
a.cumsum()
a.cumprod()
...
```

Методы линейной алгебры

Модуль `linalg` реализует методы линейной алгебры, включая обращение матриц, нахождение собственных чисел и собственных векторов, решение систем линейных алгебраических уравнений и пр.

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

b = np.linalg.inv(a)
a.dot(b) # [[1,0],[0,1]]

d = np.linalg.det(a)
l,v = np.linalg.eig(a)
n = np.linalg.norm(a)
x = np.linalg.solve(a,[3,4])
...
```

Матрицы

Объект матрица (matrix) наследует все атрибуты и методы класса `ndarray` и реализует MATLAB-подобное поведение

```
a = np.array([[1,2],[3,4]]) # 2D array, a.shape=(2,2)
b = a[1,:] # 1D array, b.shape=(2,)
c = a[:,1] # 1D array, c.shape=(2,)

b_row = a[1:,:] # 2D array, b_row.shape=(1,2)
c_col = a[:,1:] # 2D array, c_col.shape=(2,1)

A = np.matrix(a)
B = A[1,:] # matrix, B.shape=(1,2)
B = A[1:,:] # matrix, B.shape=(1,2)
C = A[:,1] # matrix, C.shape=(2,2)
C = A[:,1:] # matrix, C.shape=(2,2)
```

Matrix vs Ndarray

Отличия `matrix` от `ndarray`:

- Матрицы всегда 2D
- Операция $*$ означает матричное умножение
- Матрицы всегда 2D
- Любые срезы матриц – это также матрицы
- Дополнительные матричные функции

```
a = np.matrix([[1,2],[3,4]])
b = np.matrix([[2,0],[0,2]])

c = a*b # matrix product
print(c( # [[2,4],[6,8]]
d = a**2 # the same as a*a
a.H # conjugate transpose
a.I # inverse matrix
```

Генерация случайных массивов

```
# random floats in [0.0,1.0)
a = np.random.rand(5) # 1D array, shape=(5,)
a = np.random.rand(2,5) # 2D array, shape=(2,5)

a = np.random.randn(2,5) # 2D array, N(0,1)
a = np.random.normal(10,0.5,(2,5)) # 2D, N(10,0.5)

a = np.random.randint(10) # random int in [0,10)
a = np.random.randint(10,20,(2,5)) # 2D int array

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

Загрузка и сохранение массивов

```
a = np.random.rand(2,5) # 2D array, shape=(2,5)
b = np.random.randint(10) # random int in [0,10)

# uncompressed raw binary format .npy
np.save('myFile',a)
a = np.load('myFile.npy')

# zip archive .npz
np.savez('myFile',a=a,b=b)
npz = np.load('myFile.npz')
a = npz['a']
b = npz['b']

a = np.loadtxt('array_ex.txt',delimiter=',')
```