

ИСПОЛЬЗОВАНИЕ БИБЛИОТЕКИ NUMPY В PYTHON

Введение

NumPy это open-source модуль для python, который предоставляет общие математические и числовые операции в виде пре-скомпилированных, быстрых функций. Они объединяются в высокоуровневые пакеты. Они обеспечивают функционал, который можно сравнить с функционалом MatLab. NumPy (Numeric Python) предоставляет базовые методы для манипуляции с большими массивами и матрицами. SciPy (Scientific Python) расширяет функционал numpy огромной коллекцией полезных алгоритмов, таких как минимизация, преобразование Фурье, регрессия, и другие прикладные математические техники.

Установка

Если у вас есть Python(x, y) (*Примечание переводчика: Python(x, y), это дистрибутив свободного научного и инженерного программного обеспечения для численных расчётов, анализа и визуализации данных на основе языка программирования Python и большого числа модулей (библиотек)*) на платформе Windows, то вы готовы начинать. Если же нет, то после установки python, вам нужно установить пакеты самостоятельно, сначала NumPy потом SciPy. Установка доступна [здесь](#). Следуйте установке на странице, там всё предельно понятно.

Немного дополнительной информации

Сообщество NumPy и SciPy поддерживает онлайн руководство, включающие

гайды и tutorиалы, тут: docs.scipy.org/doc.

Импорт модуля `numpy`

Есть несколько путей импорта. Стандартный метод это — использовать простое выражение:

```
>>> import numpy
```

Тем не менее, для большого количества вызовов функций `numpy`, становится утомительно писать `numpy.X` снова и снова. Вместо этого намного легче сделать это так:

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

Это выражение позволяет нам получать доступ к `numpy` объектам используя `np.X` вместо `numpy.X`. Также можно импортировать `numpy` прямо в используемое пространство имен, чтобы вообще не использовать функции через точку, а вызывать их напрямую:

```
>>> from numpy import *
```

Однако, этот вариант не приветствуется в программировании на `python`, так как убирает некоторые полезные структуры, которые модуль предоставляет. До конца этого туториала мы будем использовать второй вариант импорта (`import numpy as np`).

Массивы

Главной особенностью numpy является объект array. Массивы схожи со списками в python, исключая тот факт, что элементы массива должны иметь одинаковый тип данных, как float и int. С массивами можно проводить числовые операции с большим объемом информации в разы быстрее и, главное, намного эффективнее чем со списками.

Создание массива из списка:

```
a = np.array([1, 4, 5, 8], float)
>>> a
array([ 1.,  4.,  5.,  8.])
>>> type(a)
<class 'numpy.ndarray'>
```

Здесь функция array принимает два аргумента: список для конвертации в массив и тип для каждого элемента. Ко всем элементам можно получить доступ и манипулировать ими также, как вы бы это делали с обычными списками:

```
>>> a[:2]
array([ 1.,  4.])
>>> a[3]
8.0
>>> a[0] = 5.
```

```
>>> a  
array([ 5.,  4.,  5.,  8.]
```

Массивы могут быть и многомерными. В отличии от списков можно задавать команды в скобках. Вот пример двумерного массива (матрица):

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]], float)  
>>> a  
array([[ 1.,  2.,  3.],  
       [ 4.,  5.,  6.]])  
>>> a[0,0]  
1.0  
>>> a[0,1]  
2.0
```

Array slicing работает с многомерными массивами аналогично, как и с одномерными, применяя каждый срез, как фильтр для установленного измерения. Используйте ":" в измерении для указания использования всех элементов этого измерения:

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

Метод `shape` возвращает количество строк и столбцов в матрице:

```
>>> a.shape  
(2, 3)
```

Метод `dtype` возвращает тип переменных, хранящихся в массиве:

```
>>> a.dtype  
dtype('float64')
```

Тут `float64`, это числовой тип данных в `numpy`, который используется для хранения вещественных чисел двойной точности. Так как же `float` в `Python`.

Метод `len` возвращает длину первого измерения (оси):

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

Метод `in` используется для проверки на наличие элемента в массиве:

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]], float)  
>>> 2 in a  
True  
>>> 0 in a
```

False

Массивы можно переформировать при помощи метода, который задает новый многомерный массив. Следуя следующему примеру, мы переформатируем одномерный массив из десяти элементов во двумерный массив, состоящий из пяти строк и двух столбцов:

```
>>> a = np.array(range(10), float)
>>> a
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.])
>>> a = a.reshape((5, 2))
>>> a
array([[ 0.,  1.],
       [ 2.,  3.],
       [ 4.,  5.],
       [ 6.,  7.],
       [ 8.,  9.]])
>>> a.shape
(5, 2)
```

Обратите внимание, метод `reshape` создает новый массив, а не модифицирует оригинальный.

Имейте ввиду, связывание имен в python работает и с массивами. Метод `copy` используется для создания копии существующего массива в памяти:

```
>>> a = np.array([1, 2, 3], float)
>>> b = a
```

```

>>> c = a.copy()
>>> a[0] = 0
>>> a
array([0., 2., 3.])
>>> b
array([0., 2., 3.])
>>> c
array([1., 2., 3.])

```

Списки можно тоже создавать с массивов:

```

>>> a = np.array([1, 2, 3], float)
>>> a.tolist()
[1.0, 2.0, 3.0]
>>> list(a)
[1.0, 2.0, 3.0]

```

Можно также переконвертировать массив в бинарную строку (то есть, не human-readable форму). Используйте метод `tostring` для этого. Метод `fromstring` работает в для обратного преобразования. Эти операции иногда полезны для сохранения большого количества данных в файлах, которые могут быть считаны в будущем.

```

>>> a = array([1, 2, 3], float)
>>> s = a.tostring()
>>> s
'\x00\x00\x00\x00\x00\x00\xf0?\x00\x00\x00\x00\x00\x00@\x00\x00\x00\x00\x00\x00\x08@'

```

```
>>> np.fromstring(s)
array([ 1.,  2.,  3.]
```

Заполнение массива одинаковым значением.

```
>>> a = array([1, 2, 3], float)
>>> a
array([ 1.,  2.,  3.])
>>> a.fill(0)
>>> a
array([ 0.,  0.,  0.]
```

Транспонирование массивов также возможно, при этом создается новый массив:

```
>>> a = np.array(range(6), float).reshape((2, 3))
>>> a
array([[ 0.,  1.,  2.],
       [ 3.,  4.,  5.]])
>>> a.transpose()
array([[ 0.,  3.],
       [ 1.,  4.],
       [ 2.,  5.]])
```

Многомерный массив можно переконвертировать в одномерный при помощи метода `flatten`:


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

Два или больше массивов можно сконкатенировать при помощи метода concatenate:

```
>>> a = np.array([1,2], float)
>>> b = np.array([3,4,5,6], float)
>>> c = np.array([7,8,9], float)
>>> np.concatenate((a, b, c))
array([1., 2., 3., 4., 5., 6., 7., 8., 9.]])
```

Если массив не одномерный, можно задать ось, по которой будет происходить соединение. По умолчанию (не задавая значения оси), соединение будет происходить по первому измерению:

```
>>> a = np.array([[1, 2], [3, 4]], float)
>>> b = np.array([[5, 6], [7,8]], float)
>>> np.concatenate((a,b))
array([[ 1.,  2.],
       [ 3.,  4.],
       [ 5.,  6.],
       [ 7.,  8.]])
>>> np.concatenate((a,b), axis=0)
```

```

array([[ 1.,  2.],
       [ 3.,  4.],
       [ 5.,  6.],
       [ 7.,  8.]])
>>>
np.concatenate((a,b), axis=1)
array([[ 1.,  2.,  5.,  6.],
       [ 3.,  4.,  7.,  8.]])

```

В заключении, размерность массива может быть увеличена при использовании константы `newaxis` в квадратных скобках:

```

>>> a = np.array([1, 2, 3], float)
>>> a
array([1., 2., 3.])
>>> a[:,np.newaxis]
array([[ 1.],
       [ 2.],
       [ 3.]])
>>> a[:,np.newaxis].shape
(3,1)
>>> b[np.newaxis,:]
array([[ 1.,  2.,  3.]])
>>> b[np.newaxis,:].shape
(1,3)

```

Заметьте, тут каждый массив двумерный; созданный при помощи `newaxis` имеет размерность один. Метод `newaxis` подходит для удобного создания надлежаще-мерных массивов в векторной и матричной математике.

Другие пути создания массивов

Функция `arange` аналогична функции `range`, но возвращает массив:

```
>>> np.arange(5, dtype=float)
array([ 0.,  1.,  2.,  3.,  4.])
>>> np.arange(1, 6, 2, dtype=int)
array([1, 3, 5])
```

Функции `zeros` и `ones` создают новые массивы с установленной размерностью, заполненные этими значениями. Это, наверное, самые простые в использовании функции для создания массивов:

```
>>> np.ones((2,3), dtype=float)
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
>>> np.zeros(7, dtype=int)
array([0, 0, 0, 0, 0, 0, 0])
```

Функции `zeros_like` и `ones_like` могут преобразовать уже созданный массив, заполнив его нулями и единицами соответственно:

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]], float)
>>> np.zeros_like(a)
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
```

```
>>> np.ones_like(a)
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
```

Также есть некоторое количество функций для создания специальных матриц. Для создания квадратной матрицы с главной диагональю, которая заполненная единицами, воспользуемся методом `identity`:

```
>>> np.identity(4, dtype=float)
array([[ 1.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.],
       [ 0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  1.]])
```

Функция `eye` возвращает матрицу с единичками на `k`-той диагонали:

```
>>> np.eye(4, k=1, dtype=float)
array([[ 0.,  1.,  0.,  0.],
       [ 0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  1.],
       [ 0.,  0.,  0.,  0.]])
```

Математические операции над массивами

Когда для массивов мы используем стандартные математические операции, должен соблюдаться принцип: элемент--элемент. Это означает, что массивы должны быть одинакового размера во время сложения, вычитания и тому

подобных операций:

```
>>> a = np.array([1,2,3], float)
>>> b = np.array([5,2,6], float)
>>> a + b
array([6., 4., 9.])
>>> a - b
array([-4., 0., -3.])
>>> a * b
array([5., 4., 18.])
>>> b / a
array([5., 1., 2.])
>>> a % b
array([1., 0., 3.])
>>> b**a
array([5., 4., 216.])
```

Для двумерных массивов, умножение остается поэлементным и не соответствует умножению матриц. Для этого существуют специальные функции, которые мы изучим позже.

```
>>> a = np.array([[1,2], [3,4]], float)
>>> b = np.array([[2,0], [1,3]], float)
>>> a * b
array([[2., 0.], [3., 12.]])
```

При несоответствии в размере выбрасываются ошибки:

```
>>> a = np.array([1,2,3], float)
```

```
>>> b = np.array([4,5], float)
```

```
>>> a + b
```

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

ValueError: operands could not be broadcast together with shapes (3,) (2,)

Однако, если размерность массивов не совпадает, они будут преобразованы для выполнения математических операций. Это зачастую означает, что меньший массив будет использован несколько раз для завершения операций. Рассмотрим такой пример:

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

```
>>> b = np.array([-1, 3], float)
```

```
>>> a
```

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

```
       [ 3.,  4.],
```

```
       [ 5.,  6.]])
```

```
>>> b
```

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

```
>>> a + b
```

```
array([[ 0.,  5.],
```

```
       [ 2.,  7.],
```

```
       [ 4.,  9.]])
```

Тут, одномерный массив `b` был преобразован в двухмерный, который соответствует размеру массива `a`. По существу, `b` был повторен несколько

раз, для каждой «строки» а. Иначе его можно представить так:

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

Python автоматически преобразовывает массивы в этом случае. Иногда, однако, когда преобразование играет роль, мы можем использовать константу `newaxis`, чтобы изменить преобразование:

```
>>> a = np.zeros((2,2), float)
>>> b = np.array([ -1.,  3.], float)
>>> a
array([[ 0.,  0.],
       [ 0.,  0.]])
>>> b
array([ -1.,  3.])
>>> a + b
array([[ -1.,  3.],
       [ -1.,  3.]])
>>> a + b[np.newaxis,:]
array([[ -1.,  3.],
       [ -1.,  3.]])
>>> a + b[:,np.newaxis]
array([[ -1., -1.],
       [  3.,  3.]])
```

Вдобавок к стандартным операторам, в `numpy` включена библиотека

стандартных математических функций, которые могут быть применены поэлементно к массивам. Собственно функции: `abs`, `sign`, `sqrt`, `log`, `log10`, `exp`, `sin`, `cos`, `tan`, `arcsin`, `arccos`, `arctan`, `sinh`, `cosh`, `tanh`, `arcsinh`, `arccosh`, и `arctanh`.

```
>>> a = np.array([1, 4, 9], float)
>>> np.sqrt(a)
array([ 1.,  2.,  3.])
```

Функции `floor`, `ceil` и `rint` возвращают нижние, верхние или ближайшие (округлённое) значение:

```
>>> a = np.array([1.1, 1.5, 1.9], float)
>>> np.floor(a)
array([ 1.,  1.,  1.])
>>> np.ceil(a)
array([ 2.,  2.,  2.])
>>> np.rint(a)
array([ 1.,  2.,  2.])
```

Также в `numpy` включены две важные математические константы:

```
>>> np.pi
3.1415926535897931
>>> np.e
2.7182818284590451
```


Перебор элементов массива

Проводить итерацию массивов можно аналогично спискам:

```
>>> a = np.array([1, 4, 5], int)
>>> for x in a:
...     print x
1
4
5
```

Для многомерных массивов итерация будет проводиться по первой оси, так, что каждый проход цикла будет возвращать «строку» массива:

```
>>> a = np.array([[1, 2], [3, 4], [5, 6]], float)
>>> for x in a:
...     print x
[ 1.  2.]
[ 3.  4.]
[ 5.  6.]
```

Множественное присваивание также доступно при итерации:

```
>>> a = np.array([[1, 2], [3, 4], [5, 6]], float)
>>> for (x, y) in a:
...     print x * y
2.0
```

12.0

30.0

Базовые операции над массивами

Для получения каких-либо свойств массивов существует много функций. Элементы могут быть суммированы или перемножены:

```
>>> a = np.array([2, 4, 3], float)
```

```
>>> a.sum()
```

9.0

```
>>> a.prod()
```

24.0

В этом примере были использованы функции массива. Также можно использовать собственные функции numpy:

```
>>> np.sum(a)
```

9.0

```
>>> np.prod(a)
```

24.0

Для большинства случаев могут использоваться оба варианта.

Некие функции дают возможность оперировать статистическими данными.

Это функции mean (среднее арифметическое), вариация и девиация:

```
>>> a = np.array([2, 1, 9], float)
>>> a.mean()
4.0
>>>
a.var()
12.666666666666666
>>> a.std()
3.5590260840104371
```

Можно найти минимум и максимум в массиве:

```
>>> a = np.array([2, 1, 9], float)
>>> a.min()
1.0
>>> a.max()
9.0
```

Функции `argmin` и `argmax` возвращают индекс минимального или максимального элемента:

```
>>> a = np.array([2, 1, 9], float)
>>> a.argmin()
1
>>> a.argmax()
2
```

Для многомерных массивов каждая из функций может принять дополнительный аргумент `axis` и в зависимости от его значения выполнять

функции по определенной оси, помещая результаты исполнения в массив:

```
>>> a = np.array([[0, 2], [3, -1], [3, 5]], float)
>>> a.mean(axis=0)
array([ 2.,  2.])
>>> a.mean(axis=1)
array([ 1.,  1.,  4.])
>>> a.min(axis=1)
array([ 0., -1.,  3.])
>>> a.max(axis=0)
array([ 3.,  5.])
```

Как и списки, массивы можно отсортировать:

```
>>> a = np.array([6, 2, 5, -1, 0], float)
>>> sorted(a)
[-1.0, 0.0, 2.0, 5.0, 6.0]
>>> a.sort()
>>> a
array([-1.,  0.,  2.,  5.,  6.])
```

Значения в массиве могут быть «сокращены», чтобы принадлежать заданному диапазону. Это тоже самое что применять `min(max(x, minval), maxval)` к каждому элементу `x`:

```
>>> a = np.array([6, 2, 5, -1, 0], float)
>>> a.clip(0, 5)
```

```
array([ 5., 2., 5., 0., 0.])
```

Уникальные элементы могут быть извлечены вот так:

```
>>> a = np.array([1, 1, 4, 5, 5, 5, 7], float)
>>> np.unique(a)
array([ 1., 4., 5., 7.])
```

Для двумерных массивов диагональ можно получить так:

```
>>> a = np.array([[1, 2], [3, 4]], float)
>>> a.diagonal()
array([ 1., 4.])
```

Операторы сравнения и тестирование значений

Булево сравнение может быть использовано для поэлементного сравнения массивов одинаковых длин. Возвращаемое значение это массив булевых True/False значений:

```
>>> a = np.array([1, 3, 0], float)
>>> b = np.array([0, 3, 2], float)
>>> a > b
array([ True, False, False], dtype=bool)
>>> a == b
array([False,  True, False], dtype=bool)
```

```
>>> a <= b
array([False,  True,  True], dtype=bool)
```

Результат сравнения может быть сохранен в массиве:

```
>>> c = a > b
>>> c
array([ True, False, False], dtype=bool)
```

Массивы могут быть сравнены с одиночным значением:

```
>>> a = np.array([1, 3, 0], float)
>>> a > 2
array([False,  True, False], dtype=bool)
```

Операторы any и all могут быть использованы для определения истинны ли хотя бы один или все элементы соответственно:

```
>>> c = np.array([ True, False, False], bool)
>>> any(c)
True
>>> all(c)
False
```

Комбинированные булевы выражения могут быть применены к массивам по принципу элемент — элемент используя специальные функции `logical_and`, `logical_or` и `logical_not`:

```

>>> a = np.array([1, 3, 0], float)
>>> np.logical_and(a > 0, a < 3)
array([ True, False, False], dtype=bool)
>>> b = np.array([True, False, True], bool)
>>> np.logical_not(b)
array([False,  True, False], dtype=bool)
>>> c = np.array([False, True, False], bool)
>>> np.logical_or(b, c)
array([ True,  True, False], dtype=bool)

```

Функция `where` создает новый массив из двух других массивов одинаковых длин используя булев фильтр для выбора между двумя элементами. Базовый синтаксис: `where(boolarray, truearray, falsearray)`:

```

>>> a = np.array([1, 3, 0], float)
>>> np.where(a != 0, 1 / a, a)
array([ 1.      , 0.33333333, 0.      ])

```

С функцией `where` так же может быть реализовано «массовое сравнение»:

```

>>> np.where(a > 0, 3, 2)
array([3, 3, 2])

```

Некоторые функции дают возможность тестировать значения в массиве. Функция `nonzero` возвращает кортеж индексов ненулевых значений.

Количество элементов в кортеже равно количеству осей в массиве:

```
>>> a = np.array([[0, 1], [3, 0]], float)
>>> a.nonzero()
(array([0, 1]), array([1, 0]))
```

Также можно проверить значения на конечность и NaN(not a number):

```
>>> a = np.array([1, np.NaN, np.Inf], float)
>>> a
array([ 1., NaN, Inf])
>>> np.isnan(a)
array([False,  True, False], dtype=bool)
>>> np.isfinite(a)
array([ True, False, False], dtype=bool)
```

Хотя здесь мы использовали константы `np.NaN` и `np.Inf` чтобы добавить значения NaN и бесконечность, они могут быть результатами применения стандартных математических операций.

Выбор элементов массива и манипуляция с ними

Мы уже видели, как и у списков, элементы массива можно получить используя операцию доступа по индексу. Однако, в отличие от списков, массивы также позволяют делать выбор элементов используя другие массивы. Это значит, что мы можем использовать массив для фильтрации специфических подмножеств элементов других массивов.

Булевы массивы могут быть использованы как массивы для фильтрации:

```
>>> a = np.array([[6, 4], [5, 9]], float)
>>> a >= 6
array([[ True, False],
       [False,  True]], dtype=bool)
>>> a[a >= 6]
array([ 6.,  9.])
```

Стоит заметить, что когда мы передаем булев массив $a \geq 6$ как индекс для операции доступа по индексу массива a , возвращаемый массив будет хранить только True значения. Также мы можем записать массив для фильтрации в переменную:

```
>>> a = np.array([[6, 4], [5, 9]], float)
>>> sel = (a >= 6)
>>> a[sel]
array([ 6.,  9.])
```

Более замысловатая фильтрация может быть достигнута использованием булевых выражений:

```
>>> a[np.logical_and(a > 5, a < 9)]
>>> array([ 6.])
```

В придачу к булеву выбору, также можно использовать целочисленные

массивы. В этом случае, целочисленный массив хранит индексы элементов, которые будут взяты из массива. Рассмотрим следующий одномерный пример:

```
>>> a = np.array([2, 4, 6, 8], float)
>>> b = np.array([0, 0, 1, 3, 2, 1], int)
>>> a[b]
array([ 2.,  2.,  4.,  8.,  6.,  4.])
```

Иными словами, когда мы используем `b` для получения элементов из `a`, мы берем 0-й, 0-й, 1-й, 3-й, 2-й и 1-й элементы `a` в этом порядке. Списки также могут быть использованы как массивы для фильтрации:

```
>>> a = np.array([2, 4, 6, 8], float)
>>> a[[0, 0, 1, 3, 2, 1]]
array([ 2.,  2.,  4.,  8.,  6.,  4.])
```

Для многомерных массивов, нам необходимо передать несколько одномерных целочисленных массивов в оператор доступа индексу (*Прим. переводчика: в нашем случае индексы это массивы*) для каждой оси. Потом каждый из массивов проходит такую последовательность: первый элемент соответствует индексу строки, который является первым элементом массива `b`, второй элемент соответствует индексу столбца, который является первым элементом массива `c` и так далее. (*Прим. переводчика: первый массив [2, 2] и второй [1, 4], имеем на выходе элементы с индексами [2, 1] и [2, 4]*) Пример:

```
>>> a = np.array([[1, 4], [9, 16]], float)
```

```
>>> b = np.array([0, 0, 1, 1, 0], int)
>>> c = np.array([0, 1, 1, 1, 1], int)
>>> a[b,c]
array([ 1.,  4., 16., 16.,  4.])
```

Специальная функция `take` доступна для выполнения выборки с целочисленными массивами. Это работает также как и использования оператора взятия по индексу:

```
>>> a = np.array([2, 4, 6, 8], float)
>>> b = np.array([0, 0, 1, 3, 2, 1], int)
>>> a.take(b)
array([ 2.,  2.,  4.,  8.,  6.,  4.])
```

Функция `take` также предоставляет аргумент `axis` (ось) для взятия подсекции многомерного массива вдоль какой-либо оси. (*Прим. переводчика: по строкам или столбцам (для двумерных массивов)*).

```
>>> a = np.array([[0, 1], [2, 3]], float)
>>> b = np.array([0, 0, 1], int)
>>> a.take(b, axis=0)
array([[ 0.,  1.],
       [ 0.,  1.],
       [ 2.,  3.]])
>>> a.take(b, axis=1)
array([[ 0.,  0.,  1.],
       [ 2.,  2.,  3.]])
```

В противоположность к функции `take` есть функция `put`, которая будет брать значения из исходного массива и записывать их на специфические индексы в другом `put`-массиве.

```
>>> a = np.array([0, 1, 2, 3, 4, 5], float)
>>> b = np.array([9, 8, 7], float)
>>> a.put([0, 3], b)
>>> a
array([ 9.,  1.,  2.,  8.,  4.,  5.] )
```

Заметим, что значение `7` из исходного массива `b` не было использовано, так как только 2 индекса `[0, 3]` указаны. Исходный массив будет повторен если необходимо в случае не соответствия длин:

```
>>> a = np.array([0, 1, 2, 3, 4, 5], float)
>>> a.put([0, 3], 5)
>>> a
array([ 5.,  1.,  2.,  5.,  4.,  5.] )
```

Векторная и матричная математика

NumPy обеспечивает много функций для работы с векторами и матрицами. Функция `dot` возвращает скалярное произведение векторов:

```
>>> a = np.array([1, 2, 3], float)
```

```
>>> b = np.array([0, 1, 1], float)
>>> np.dot(a, b)
5.0
```

Функция `dot` также может умножать матрицы:

```
>>> a = np.array([[0, 1], [2, 3]], float)
>>> b = np.array([2, 3], float)
>>> c = np.array([[1, 1], [4, 0]], float)
>>> a
array([[ 0.,  1.],
       [ 2.,  3.]])
>>> np.dot(b, a)
array([ 6., 11.])
>>> np.dot(a, b)
array([ 3., 13.])
>>> np.dot(a, c)
array([[ 4.,  0.],
       [14.,  2.]])
>>> np.dot(c, a)
array([[ 2.,  4.],
       [ 0.,  4.]])
```

Также можно получить скалярное, тензорное и внешнее произведение матриц и векторов. Заметим, что для векторов внутреннее и скалярное произведение совпадает.

```
>>> a = np.array([1, 4, 0], float)
```

```
>>> b = np.array([2, 2, 1], float)
>>> np.outer(a, b)
array([[ 2.,  2.,  1.],
       [ 8.,  8.,  4.],
       [ 0.,  0.,  0.]])
>>> np.inner(a, b)
10.0
>>> np.cross(a, b)
array([ 4., -1., -6.]])
```

NumPy также предоставляет набор встроенных функций и методов для работы с линейной алгеброй. Это всё можно найти в под-модуле `linalg`. Этими модулями также можно оперировать с вырожденными и невырожденными матрицами. Определитель матрицы ищется таким образом:

```
>>> a = np.array([[4, 2, 0], [9, 3, 7], [1, 2, 1]], float)
>>> a
array([[ 4.,  2.,  0.],
       [ 9.,  3.,  7.],
       [ 1.,  2.,  1.]])
>>> np.linalg.det(a)
-48.
```

Также можно найти собственный вектор и собственное значение матрицы:

```
>>> vals, vecs = np.linalg.eig(a)
>>> vals
array([ 9.      ,  2.44948974, -2.44948974])
```

```
>>> vecs
array([[ -0.3538921 , -0.56786837,  0.27843404],
       [ -0.88473024,  0.44024287, -0.89787873],
       [ -0.30333608,  0.69549388,  0.34101066]])
```

Невырожденная матрица может быть найдена так:

```
>>> b = np.linalg.inv(a)
>>> b
array([[ 0.14814815,  0.07407407, -0.25925926],
       [ 0.2037037 , -0.14814815,  0.51851852],
       [-0.27777778,  0.11111111,  0.11111111]])
>>> np.dot(a, b)
array([[ 1.00000000e+00,  5.55111512e-17,  2.22044605e-16],
       [ 0.00000000e+00,  1.00000000e+00,  5.55111512e-16],
       [ 1.11022302e-16,  0.00000000e+00,  1.00000000e+00]])
```

Одиночное разложение (аналог диагонализации не квадратной матрицы) может быть достигнут так:

```
>>> a = np.array([[1, 3,4], [5, 2, 3]], float)
>>> U, s, Vh = np.linalg.svd(a)
>>> U
array([[ -0.6113829 , -0.79133492],
       [ -0.79133492,  0.6113829 ]])
>>> s
array([ 7.46791327,  2.86884495])
>>> Vh
```

```
array([[ -0.61169129, -0.45753324, -0.64536587],  
       [ 0.78971838, -0.40129005, -0.46401635],  
       [-0.046676 , -0.79349205,  0.60678804]])
```

Статистика

В придачу к функциям `mean`, `var` и `std`, NumPy предоставляет еще некоторые методы для работы со статистическими данными в массивах.

Медиана может быть найдена так:

```
>>> a = np.array([1, 4, 3, 8, 9, 2, 3], float)  
>>> np.median(a)  
3.0
```

Коэффициент корреляции для некоторых переменных, наблюдается несколько раз и может быть найден из массивов вида: `[[x1, x2, ...], [y1, y2, ...], [z1, z2, ...], ...]`, где `x`, `y`, `z` это разные квантовые наблюдаемые и номера указывают количество «наблюдений»:

```
>>> a = np.array([[1, 2, 1, 3], [5, 3, 1, 8]], float)  
>>> c = np.corrcoef(a)  
>>> c  
array([[ 1.        ,  0.72870505],  
       [ 0.72870505,  1.        ]])
```

Имеем возвращаемый массив `c[i, j]` который хранит корреляционный коэффициент для `i`-тых и `j`-тых квантовых наблюдаемых.

Аналогично, ковариационный момент может быть найден:

```
>>> np.cov(a)
array([[ 0.91666667,  2.08333333],
       [ 2.08333333,  8.91666667]])
```

Случайные числа

Важная часть каждой симуляции это способность генерировать случайные числа. Для этого мы используем встроенный в NumPy генератор псевдослучайных чисел в под-модуле `random`. Числа являются *псевдо* случайными, в том плане что, они сгенерированы детерминистически из порождающего элемента (`seed number`), но рассредоточены в статистическом сходстве с случайным образом. Для генерации NumPy использует особенный алгоритм который имеет название Mersenne Twister.

Задать порождающий элемент последовательности случайных чисел можно так:

```
>>> np.random.seed(293423)
```

Seed это целое число. Каждая программа которая запускается с одинаковым seed`ом будет генерировать одинаковую последовательность чисел каждый раз. Это может быть полезно для отладки, но вообще нам не нужно задавать seed, на самом деле, когда мы запускаем программу несколько раз, мы хотим получать каждый раз разную последовательность чисел. Если эта команда не будет выполнена, то NumPy автоматически выбирает случайный seed

(базирующийся на времени), который является разным при каждом запуске программы.

Массив случайных чисел из полуинтервала $[0.0, 1.0)$ может быть сгенерирован так:

```
>>> np.random.rand(5)
array([ 0.40783762, 0.7550402 , 0.00919317, 0.01713451, 0.95299583])
```

Функция `rand` может быть использована для генерации двумерных массивов, или можно использовать функцию `reshape`:

```
>>> np.random.rand(2,3)
array([[ 0.50431753, 0.48272463, 0.45811345],
       [ 0.18209476, 0.48631022, 0.49590404]])
>>> np.random.rand(6).reshape((2,3))
array([[ 0.72915152, 0.59423848, 0.25644881],
       [ 0.75965311, 0.52151819, 0.60084796]])
```

Для генерации единичного случайного числа на интервале $[0.0, 1.0)$:

```
>>> np.random.random()
0.70110427435769551
```

Для генерации случайного целочисленного числа в диапазоне $[\text{min}, \text{max})$ используем функцию `randint(min, max)`:

```
>>> np.random.randint(5, 10)
```

```
9
```

В каждом нашем примере, мы генерировали числа из непрерывного равномерного распределения. NumPy также включает генераторы для других распределений, таких как: Бета, биномиальное, хи-квадрат, Дирихле, экспоненциальное, Фишера, Гамма, геометрическое, Гамбала, гипергеометрическое, Лапласа, логистическое, логнормальное, логарифмическое, мультиномиальное, многомерное нормальное, отрицательное биномиальное, нецентральное хи-квадрат, нецентральное Фишера, нормальное (Гаусса), Парето, Пуассона, степенное, Рэлея, Коши, Стьюдента, треугольное, Фон-Миса, Вальда, Вейбулла и Ципфа. Рассмотрим два примера.

Для генерации из дискретного распределения Пуассона при $\lambda = 6.0$,

```
>>> np.random.poisson(6.0)
```

```
5
```

Для генерации числа из нормального распределения (Гаусса) при среднем значении $\mu = 1.5$ и стандартной девиации $\sigma = 4.0$:

```
>>> np.random.normal(1.5, 4.0)
```

```
0.83636555041094318
```

Для получения числа из нормального распределения ($\mu = 0$, $\sigma = 1$), без указания аргументов:

```
>>> np.random.normal()
0.27548716940682932
```

Для генерации нескольких значений используем аргумент `size`:

```
>>> np.random.normal(size=5)
array([-1.67215088,  0.65813053, -0.70150614,  0.91452499,  0.71440557])
```

Модуль для генерации случайных чисел также может быть использован для случайного распределения значений в списке. Это может быть полезно если мы хотим случайно распределить значения в списке:

```
>>> l = range(10)
>>> l
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> np.random.shuffle(l)
>>> l
[4, 9, 5, 0, 2, 7, 6, 8, 1, 3]
```

Заметим, что функция `shuffle` модифицирует уже существующий массив и не возвращает новый.

Некоторая дополнительная информация

NumPy включает еще много других функций о которых мы не упоминали здесь. В частности это функции для работы с дискретным преобразованием Фурье, более сложными операциями в линейной алгебре, тестированием

массивов на размер / размерность / тип, разделением и соединением массивов, гистограммами, создания массивов из каких-либо данных разными путями, созданием и оперированием grid-массивов, специальными значениями (NaN, Inf), set-операции, созданием разных видов специальных матриц и вычислением специальных математических функций (Например: функции Бесселя). Также вы можете посмотреть [документацию NumPy](#) для более точных деталей.

Модули SciPy

SciPy очень хорошо расширяет функционал NumPy. Мы не будем говорить о его деталях, но рассмотрим некоторые его возможности. Большинство функций SciPy доступны после импорта модуля:

```
>>> import scipy
```

Функция help обеспечит полезной информацией о SciPy:

```
>>> help(scipy)
```

Help on package scipy:

NAME

scipy

FILE

c:\python25\lib\site-packages\scipy__init__.py

DESCRIPTION

SciPy --- A scientific computing package for Python

=====

Documentation is available in the docstrings and online at <http://docs.scipy.org>.

Contents

SciPy imports all the functions from the NumPy namespace, and in addition provides:

Available subpackages

odr --- Orthogonal Distance Regression [*]

misc --- Various utilities that don't have

another home.sparse.linalg.eigen.arpack --- Eigenvalue solver using iterative methods. [*]

fftpack --- Discrete Fourier Transform algorithms[*]

io --- Data input and output [*]

sparse.linalg.eigen.lobpcg --- Locally Optimal Block Preconditioned Conjugate Gradient Method (LOBPCG) [*]

special --- Airy Functions [*]

lib.blas --- Wrappers to BLAS library [*]

sparse.linalg.eigen --- Sparse Eigenvalue Solvers [*]

stats --- Statistical Functions [*]

lib --- Python wrappers to external libraries [*]

lib.lapack --- Wrappers to LAPACK library [*]

maxentropy --- Routines for fitting maximum entropy models [*]

integrate --- Integration routines [*]

ndimage --- n-dimensional image package [*]

linalg --- Linear algebra routines [*]

spatial --- Spatial data structures and algorithms[*]

interpolate --- Interpolation Tools [*]

sparse.linalg --- Sparse Linear Algebra [*]
sparse.linalg.dsolve.umfpack --- :Interface to the UMFPACK library: [*]
sparse.linalg.dsolve --- Linear Solvers [*]
optimize --- Optimization Tools [*]
cluster --- Vector Quantization / Kmeans [*]
signal --- Signal Processing Tools [*]
sparse --- Sparse Matrices [*]
[*] - using a package requires explicit import (see pkgload)
...

Заметим, что некоторым под-модулям нужен непосредственно дополнительный импорт, которые помечены звездой:

```
>>> import scipy  
>>> import scipy.interpolate
```

Функции в каждом модуле хорошо задокументированы во внутренних docstring`ax и в официальной документации. Большинство из них непосредственно предоставляет функции для работы с числовыми алгоритмами и они очень просты в использовании. Таким образом, SciPy может сохранять гигантское количество времени в научных вычислениях, т.к. он обеспечивает уже написанные и протестированные функции. Мы не будем рассматривать SciPy детально, но таблица ниже покрывает некоторые его возможности:

Модуль	Для чего используется
scipy.constants	Набор математических и физических констант

scipy.special	Много специальных функций для математической физики, такие как: Эйри, эллиптические, Бесселя, гамма, бета, гипергеометрические, параболического цилиндра, Матьё, шаровидной волны, Струве, Кельвина.
scipy.integrate	Функции для работы с численным интегрированием используя методы трапеции, Симпсона, Ромберга и другие. Также предоставляет методы для работы с полными дифференциальными уравнениями.
scipy.optimize	Стандартные методы поиска максимума/минимума для работы с обобщенными пользовательскими функциями. Включенные алгоритмы: Нелдера — Мида, Поулла (<i>Powell's</i>), сопряженных градиентов, Бройдена — Флетчера — Гольдфарба — Шанно, наименьших квадратов, условной оптимизации, имитации отжига, полного перебора, Брента, Ньютона, бисекции, Бройдена, Андерсона и линейного поиска.
scipy.linalg	Более широкий функционал для работы с линейной алгеброй чем в NumPy. Предоставляет больше возможностей для использования специальных и быстрых функций, для специфических объектов (Например: трёхдиагональная матрица). Включенные методы: поиск невырожденной матрицы, поиск определителя, решение линейных систем уравнений, расчета норм и псевдообратной матрицы, спектрального разложения, сингулярного разложения, LU-разложения, разложения Холецкого, QR-разложения, разложения Шура и много других математических операций для работы с матрицами.

scipy.sparse	Функции для работы с большими разреженными матрицами
scipy.interpolate	Методы и классы для интерполяции объектов, которые могут быть использованы для дискретных числовых данных. Линейная и сплайновая (<i>Прим. переводчика: математическое представление плавных кривых</i>) интерполяция доступна для одно- и двух-мерных наборов данных.
scipy.fftpack	Методы для обработки преобразований Фурье.
scipy.signal	Методы для обработки сигналов, например: свертка функций, корреляция, дискретное преобразование Фурье, сглаживающий В-сплайн, фильтрация, и т.д и т.п.
scipy.stats	Большая библиотека статистических функций и распределений для работы с наборами данных.

Большая группа разработчиков непрерывно продолжает разрабатывать новый функционал SciPy. Хороший практический подход это: если вы думаете о реализации каких-либо числовых функций и методов в вашем коде, можете сначала заглянуть в оригинальную документацию SciPy. Есть вероятность того, что это уже кто-то реализовал и внес в SciPy