Введение в Numpy

Что такое Numpy? (1/2)

- Библиотека для работы с **массивами**
- Массив это **индексированная коллекция** переменных **одного типа**
- По умолчанию массивы являются **изменяемыми** (mutable) объектами – хранимые значения можно менять
    - новые значения должны быть **совместимы по типу данных**
    - массив можно сделать **неизменяемым** вызвав метод массива `.setflags(write=False)`

Что такое Numpy? (2/2)

- тип данных (`dtype`):
  - логический (булевский, Boolean) - `bool`
  - целочисленный (Integer) – `int32`, `int64`
  - число с плавающей точкой (Float) – `float16`, `float64`
  - объект (Object) - `object`
  - и др. *https://docs.scipy.org/doc/numpy/user/basics.types.html*
- если тип не указан явно, то он будет **выведен**: общий подходящий тип
- библиотека оптимизирована по **скорости** и **легка** в использовании
- основа для других библиотек, например **Pandas**

Размерность (shape) массива Numpy

- Массив Numpy имеет **размерность** (англ. **shape** – «форма»). Этот параметр показывает размеры массива во всех направлениях
  - **Вектор** содержащий `n` элементов: `(n, )`
  - **Матрица** `n` на `p` элементов: `(n, p)`
  - **Тензор** `n` на `p` на `q` элементов: `(n, p, q)`
  - и так далее …

- **Вектор** это не то же самое, что матрица-столбец `(n, 1)` и матрица-строка `(1, n)`
  - чтобы объединить вектор и матрицу, необходимо изменить размерность (reshape) объектов

Практическое введение

```
import numpy as np

a1 = np.array([[1, 2, 3], [4, 5, 6]])
a1
    > array([[1, 2, 3],
             [4, 5, 6]])

a1.shape
    > (2, 3)

a1.reshape((1, 6))
    > array([[1, 2, 3, 4, 5, 6]])

a1.reshape(6)    # отличается?
```

Получение доступа к элементам: индексирование

```
a4[0, 0]
    >   # элемент по адресу (0, 0)
a4[1, :]
    >   # все элементы второй *строки*
a4[:, 2].shape
    >   # возвращает *вектор*
a4[1:]
    >   # индексирует строки
a4[0:2]
    >   # исключая последнюю строка
a2[:-1]
```

Векторизованные операции

- операторы эффективно реализуют поэлементные операции
  - под капотом вызывается код на C («векторизация»)

```
a1 = np.array([[1, 2, 3], [4, 5, 6]])
a2 = np.array([[10, 20, 30], [40, 50, 60]])
a1 + a2
    > array([[11, 22, 33],
             [44, 55, 66]])

a1 * a2
a1 / a2
```

Что произойдет, если размерности (shape) различаются?

```
a2 = np.array([1, 2, 3])    # shape?
a2 * 2
    > array([2, 4, 6])

a3 = np.array([[10], [20], [30]])    # shape?
a4 = a2 + a3
    > array([[11, 12, 13],
             [21, 22, 23],
             [31, 32, 33]])
```

Broadcasting

- трюк в Numpy, который позволяет использовать **векторизованные операции** без необходимости создавать большие временные матрицы
    - умножение матрицы на x не требует создания матрицы, содержащей нужное количество повторений x
    - вычисления являются векторизованными и потому быстрыми (реализованы на C)

- возможно только в том случае, когда не совпадающая размерность одного объекта может быть **“растянута”** до нужного размера другого объекта
    - в реальности такое “растяжение” не происходит, чтобы сэкономить память

- Примеры:
    - матрица (n, p) и скаляр –> a matrix (n, p)
    - матрица-строка (1, n) и матрица-столбец (p, 1) -> матрица (n, p)

Изменение элементов массива

```
a4[0] = 10
    >    # всем элементам в строке 0 назначить значение 10

a4[:, 1] = [1, 2, 3]
    >    # назначить столбцу 1 значение [1, 2, 3]

a4[1:, 1:] = [[1, 2], [3, 4]]
    >    # назначить сегменту матрицы новые значения

a4[0, 0] = "a string"
    >    # что произойдет, если попытаться назначить какому-нибудь
         # элементу матрицы целых чисел строковое значение?

a4[2] = 2.7
    >    # что произойдет, если попытаться назначить какому-нибудь
         # элементу матрицы значение типа float (число с плавающей
         # точкой)?
```

Булевские маски

```
mask = a4 > 22
mask
    > array([[False, False, False],
             [False, False,  True],
             [ True,  True,  True]])

a4[mask].shape
    > (4,)

a4[mask]
    > array([23, 31, 32, 33])

a4[mask] = 5
    > array([[11, 12, 13],
             [21, 22,  5],
             [ 5,  5,  5]])
```

Создание нового массива

```
# создать массив, содержащий равномерно распределённые значения
# из некоторого интервала

np.arange(10)
np.linspace(0, 1, 100)


# другие способы создания массивов

np.ones(100) * 10
np.zeros(100)
np.full(100, 10)
np.eye(10)
np.random.randn(10, 5)
```

Математика

```
# к массиву можно применять функции

np.sqrt(a1)
np.exp(a2)
np.log(a3)
np.sin(np.linspace(0, 1, 100) * 2 * np.pi))
np.transpose(a4)

$ ...
```

Укладывание массивов в стопку (stacking)

```
np.vstack([a1, a4])
    > array([[ 2, 2,  3],
             [10, 1, 10],
             [21, 1,  2],
             [ 2, 3,  4]])

np.hstack([a1, a2])
    > array([2, 2, 3, 1, 2, 3])
```

Методы массива Numpy

```
a1.mean()
a1.sum()
a1.sum(axis=0)
a2.dot(a3)    # a2 @ a3
a1.cumsum()
a2.max()
a2.argmax()
a1.T
a1.reshape(-1)
…
```

Векторизация (1/2)

- Numpy предоставляет много инструментов, позволяющих писать **эффективный** код
  - индексирование, broadcasting, фильтрация (выбор элементов), функции, методы, …

- иной раз необходимо **изменить подход к решению задачи**, чтобы сделать решение пригодным для numpy
  - использовать вектора/матрицы вместо циклов
  - использовать механизм broadcasting и другие...

- если этого все ещё недостаточно, то можно **«векторизовать»** пользовательские функции
  - под капотом для организации циклов будет использоваться Питон

Векторизация (2/2)

```
# создаем векторизованную реализацию некоторой функции
vect_func = np.vectorize(func)

# применяем эту функцию к массиву
vect_func(a4)
```

Практика

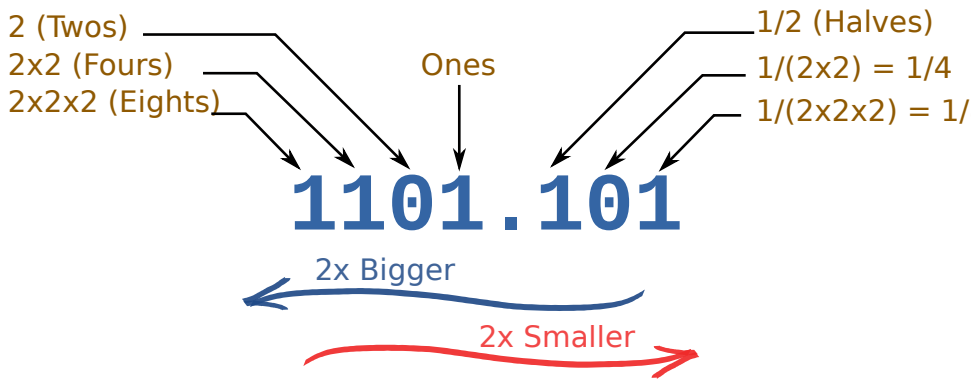
numpy.ipynb

Работа с числами с плавающей точкой (float)

Работа с числами с плавающей точкой (1/4)

- В компьютере числа представлены в **двоичном** (**бинарном**) виде, то есть, при помощи 1 и 0
  - десятичные дроби не могут быть представлены точно
  - необходим компромисс между точностью и скоростью вычислений

- В десятичной системе есть разряд **единиц** (10^0), **десятков** (10^1), **сотен** (10^2), **тысяч** (10^3) и т. д.
- В двоичной системе есть разряд **единиц** (2^0), **двоек** (2^1), **четверок** (2^2), **восьмерок** (2^3) и т. д.



Это равно 1×8 + 1×4 + 0×2 + 1 + 1×(1/2) + 0×(1/4) + 1×(1/8) = **13.625** в десятичной системе

Источник: https://www.google.com/url?q=https://www.mathsisfun.com/binary-number-system.html&sa=D&ust=1575479348491000&usg=AFQjCNFeq1E5TRyyXDMf1Iox5JgziB1q0w

Работа с числами с плавающей точкой (2/4)

● Numpy поддерживает значительно больше численных типов данных чем чистый Python, как например:

`float16` - `float32` - `float64` — `longdouble` и `float`

○ при необходимости, можно использовать тип данных, предоставляющих бóльшую точность

○ метод **`.finfo`** предоставляет информацию о диапазоне значений и их точности

```
import numpy as np

np.finfo('float64')
    > finfo(resolution=1e-15, min=-1.7976931348623157e+308,
            max=1.7976931348623157e+308, dtype=float64)

np.finfo('float16')
    > finfo(resolution=0.001, min=-6.55040e+04,
            max=6.55040e+04, dtype=float16)
```

Работа с числами с плавающей точкой (3/4)

- все элементы некоторого массива имеют один и тот же тип данных (**`dtype`**)
- Numpy может сам вывести тип данных (**`dtype`**) из данных в массиве или тип данных можно указать явно

```
x = np.array([0.33, 2, 3])
print(x[0] == 0.33)   # dtype по умолчанию – float64
    >>> True

x.dtype = 'float32'   # численный тип с уменьшенной точностью
print(x[0] == 0.33)
    >>> False
```

Работа с числами с плавающей точкой (4/4)

● имейте в виду, что, работая с числами в чистом Питоне и в Numpy, в частности при сравнении или округлении значений, вы можете заметить некоторые странные и неожиданные вещи.

```
np_val = np.float64(435)/100
py_val = 435/100
print(round(np_val, 1))
    > 4.4
print(round(float(np_val), 1))
    > 4.3
print(round(np.float64(py_val), 1))
    > 4.4
print(round(py_val, 1))
    > 4.3

4.35 типа float64 это 4.34999999999999964472863211994990706443786621093750
```