Information Technologies ¶

Блокнот №10

Библиотеки numpy и matplotlib

Библиотека numpy: эффективные массивы

Писать программы на Python легко и приятно. Гораздо легче и приятнее, чем на низкоуровневых языках программирования, таких как C или C++. Но, увы, чудес не бывает: за простоту написания кода мы платим скоростью его исполнения.

In [ ]:
```
numbers = [1.2] * 10000
numbers[:10]
```

Для проверки, с какой скоростью выполняется некоторый фрагмент кода, полезно использовать магическое слово `%%timeit`. Оно говорит, что ячейку нужно выполнить несколько раз и засечь, сколько времени на это ушло.

In [ ]:
```
%%timeit

squares = [x**2 for x in numbers]
```

Больше миллисекунды на один проход! (Кстати, `x*x` будет в два раза быстрее — попробуйте!) Не очень-то быстро, на самом деле. Для «тяжелой» математики, часто возникающей при обработке больших массивов данных, хочется использовать все возможности компьютера.

Но не надо отчаиваться: для быстрой работы с числами есть специальные библиотеки, и главная из них — `numpy`.

In [ ]:
```
import numpy as np
```

Главный объект, с которым мы будем работать — это `np.array` (на самом деле он называется `np.ndarray`):

In [ ]:
```
np_numbers = np.array(numbers)
```

In [ ]:
```
np_numbers
```

`np.array` — это специальный тип данных, похожий на список, но содержащий данные только одного типа (в данном случае — только вещественные числа).

In [ ]:
```
np_numbers[3]
```

In [ ]:
```
len(np_numbers)
```

С математической точки зрения, `np.array` — это что-то, похожее на вектор. Но практически все операции выполняются поэлементно. Например, возведение в квадрат каждого элемента можно

реализовать как `np_numbers**2` .

In [ ]:
```
np_squares = np_numbers**2
np_squares
```

Посмотрим, как быстро работает эта операция:

In [ ]:
```
%%timeit

np_squares = np_numbers**2
```

Здесь 5 микросекунд, в 200 раз быстрее! Правда, нас предупреждают, что это может быть последствия кеширования — но в любом случае, работа с массивами чисел с помощью `numpy` происходит гораздо быстрее, чем с помощью обычных списков и циклов.

Давайте посмотрим на `np.array` более подробно.

Массивы похожи на списки.

In [ ]:
```
from numpy import array
```

In [ ]:
```
q = array([4, 5, 8, 9])
```

Будем дальше называть `np.array` массивами (в отличие от списков, которые мы так в Python не называем). Итак, можно обращаться к элементам массива по индексам, как и к спискам.

In [ ]:
```
q[0]
```

И менять их тоже можно

In [ ]:
```
q[0] = 12
q
```

Можно итерировать элементы списка, хотя этого следует по возможности избегать — массивы numpy нужны как раз для того, чтобы не использовать циклы для выполнения массовых операций. Ниже будет понятно, как это можно делать.

In [ ]:
```
for x in q:
    print(x)
```

Можно делать срезы (но с ними тоже есть хитрости, об этом ниже).

In [ ]:
```
q[1:3]
```

Давайте заведём ещё один массив.

In [ ]:
```
w = array([2, 3, 6, 10])
```

Все арифметические операции над массивами выполняются поэлементно. Поэтому `+` означает сложение, а не конкатенацию. В отличие от обычных списков.

In [ ]:
```
q + w
```

Если вы хотели сделать конкатенацию, то нужно использовать не оператор сложения, а специальную функцию.

In [ ]:
```
np.concatenate( [q, w] )
```

Аналогично сложению работают и другие операции. Например, умножение:

In [ ]:
```
q * w
```

Если у массивов будет разная длина, то ничего не получится:

In [ ]:
```
q = np.array([14, 8, 14, 9])
w = np.array([1, 3, 4])
q + w
```

Можно применять различные математические операции к массивам.

In [ ]:
```
x = array([1,2,3,4,5])
y = array([4, 5, 6, 2, 1])
```

In [ ]:
```
np.exp(x)
```

Заметим, что мы должны были использовать функцию `exp` из `numpy`, а не из обычного `math`. Если бы мы взяли эту функцию из `math`, ничего бы не сработало.

In [ ]:
```
import math
```

In [ ]:
```
math.sqrt(x)
```

Вообще в `numpy` много математических функций. Вот, например, квадратный корень:

In [ ]:
```
np.sqrt(x)
```

Типы элементов в массивах

Вообще, в массивах могут храниться не только числа.

In [ ]:
```
mixed_array = np.array([1, 2, 3, "Hello"])
```

Однако, все элементы, лежащие в одном массиве, должны быть одного типа.

In [ ]:
```
mixed_array
```

Здесь видно, что числа `1`, `2`, `3` превратились в строчки `'1'`, `'2'`, `'3'`. Параметр `dtype` содержит информацию о типе объектов, хранящихся в массиве. `<U21` означает юникодную строку длиной максимум 21 байт. При попытке записать более длинную строку она будет обрезана.

In [ ]:
```
mixed_array[0] = 'Hello, World, This is a Test'
mixed_array[0]
```

Вообще `numpy` при создании массива старается не терять информацию и выбирает самый «вместительный» тип.

In [ ]:
```
np.array([1,2,3])
```

In [ ]:
```
array([1,2,3, 5.])
```

Все числа превратились в вещественные, поскольку в исходном списке, из которого делается массив, есть вещественное число.

Срезы

Давайте посмотрим внимательно на срезы.

In [ ]:
```
x = array([1.1, 2.2, 3.3, 4.4, 5.5])
```

In [ ]:
```
s = x[1:3]
```

In [ ]:
```
s
```

Пока всё идёт как обычно: мы создали срез, начинающийся с элемента с индексом 1 (то есть второй элемент, нумерация с нуля) и заканчивающийся элементом с индексом 3 (последний элемент всегда не включается).

Теперь попробуем изменить значение элемента *в срезе*:

In [ ]:
```
s[0] = 100
```

In [ ]:
```
s
```

Как вы думаете, что произойдёт с исходным массивом `x`?

In [ ]:
```
x
```

Он тоже изменился! Раньше мы видели подобную штуку в ситуациях, когда один список имел несколько имён (то есть несколько переменных на него ссылались), но создание среза раньше приводило к копированию информации. Оказывается, в numpy создание среза ничего не копирует: срез — это не новый массив, содержащий те же элементы, что старый, а так называемый *view* (вид), то есть своего рода интерфейс к старому массиву. Грубо говоря, наш срез `s` просто помнит, что «его» элемент с индексом 0 — это на самом деле элемент с индексом 1 от исходного массива `x`, а его элемент с индексом 1 — это на самом деле элемент с индексом 2 от исходного массива, а других элементов у него нет. Можно думать про срез как про такие специальные очки, через которые мы смотрим на исходный массив.

Преимущество и этого подхода два: во-первых, непосвященные сломают голову, пытаясь понять, что тут происходит, а во-вторых если у нас есть огромный массив данных, то нам не придётся тратить ресурсы на то, чтобы его скопировать, если нам нужно сделать срез. Недостатки тоже есть, но они являются продолжением первого из преимуществ.

Если вам всё-таки нужно сделать копию массива, нужно использовать метод `copy()`.

In [ ]:
```
y = x.copy()
```

In [ ]:
```
y
```

In [ ]:
```
x
```

In [ ]:
```
y[0] = 12
```

In [ ]:
```
x
```

In [ ]:
```
y
```

In [ ]:
```
x
```

Продвинутая индексация

Помимо коварных срезов есть ещё некоторые возможности создания новых массивов из старых. Например, можно выбрать из массива элементы с нужными номерами вот так:

In [ ]:
```
x
```

In [ ]:
```
y = x[ [1, 3, 4] ]
```

In [ ]:
```
y
```

Можно даже использовать одинаковые номера.

In [ ]:
```
y = x[ [1, 1, 1] ]
```

In [ ]:
```
y
```

Кстати, создаваемый таким образом объект уже является честной копией исходного, а не каким-нибудь коварным видом.

In [ ]:
```
y[0] = 123
```

In [ ]:
```
y
```

In [ ]:
```
x
```

In [ ]:
```
x
```

Есть ещё один хитрый способ выбора элементов из массива. Допустим, мы хотим выбрать только те элементы, которые обладают каким-то свойством — скажем, меньше 50. Можно было бы использовать цикл с условием или аналогичный ему list comprehension, но в `numpy` используют другой синтаксис.

In [ ]:
```
y = x[ x < 50 ]
```

In [ ]:
```
x
```

In [ ]:
```
y
```

Как он работает? Очень просто. (Ну ок, не очень.) Для начала, что такое `x < 50` ? Это результат применения операции «сравнение с 50» к каждому элементу массива. То есть это новый массив.

In [ ]:
```
x < 50
```

Если в каком-то месте стоит `True` , то это означает, что на соответствующем месте в `x` стоит элемент, который удовлетворяет условию, а если `False` , то не удовлетворяет.

Теперь можно попробовать подставить массив из `True` и `False` в качестве индекса в `x` .

In [ ]:
```
x[ array([True, True, False, False, True]) ]
```

Эта штука выбирает ровно те элементы, на чьих местах стоит `True` — то есть ровно те, для которых выполнялось условие. То, что нам нужно!

А вот так можно проверить несколько условий одновременно:

In [ ]:
```
x[ (x < 50) & (x > 2) ]
```

Скобочки очень важны, иначе ничего не заработает. Операция `&` соответствует логическому И и опять же выполняется поэлементно.

In [ ]:
```
(x < 50) & (x > 2)
```

Для логического ИЛИ мы бы исползовали `|` , а для отрицания `~` .

In [ ]:
```
(x < 50) | (x > 2)
```

In [ ]:
```
~ (x < 50)
```

Результатом такого выбора снова является *вид*, и это очень удобно, потому что позволяет заменять одни элементы на другие в зависимости от условий и таким образом избавляться от операторов `if` .

In [ ]:
```
x
```

In [ ]:
```
x[ x>50 ] = 0
# заменить все элементы, большие 50, на 0
```

In [ ]:
```
x
```

Кстати, чтобы узнать, правда ли, что два массива равны (в том числе, что состоят из одних и тех же элементов, находящихся в одном и том же порядке), теперь нельзя использовать `==` — ведь это тоже поэлементная операция!

In [ ]:
```
np.array([1, 2, 3]) == np.array([1, 2, 3])
```

Чтобы понять, правда ли, что массивы равны, можно использовать такой синтаксис:

In [ ]:
```
(np.array([1, 2, 3]) == np.array([1,2,3])).all()
```

Здесь мы сначала сравниваем массивы поэлементно, а потом применяем к результату метод `all()`, возвращающий истину только если все элементы являются истинными. Этот подход часто используется, хотя имеет свои подводные камни (см. на stackoverflow (http://stackoverflow.com/questions/10580676/comparing-two-numpy-arrays-for-equality-element-wise)). Есть и специализированная функция для проверки на равенство:

In [ ]:
```
np.array_equal(np.array([1, 2, 3]), np.array([1, 2, 3]))
```

Построение графиков в matplotlib

В Python существует много способов строить графики. Мы сейчас рассмотрим самый простой из них, а позже поговорим про более сложные. Для этого нам потребуется библиотека `matplotlib`, а точнее её часть под названием `pyplot`. Стандартный способ её импорта выглядит вот так:

In [ ]:
```
import matplotlib.pyplot as plt
```

Чтобы графики рисовались прямо в ноутбуке, нужно дать вот такую магическую команду:

In [ ]:
```
%matplotlib inline
```

Простейшее рисование — это функция `plot`, она принимает на вход список x-координат, список y-координат и рисует соответствующую картинку либо в виде ломаной:

In [ ]:
```
plt.plot([1, 2, 3, 4], [1, 4, 9, 16])
```

либо в виде отдельный точек:

In [ ]:
```
plt.plot([1, 2, 3, 4], [1, 4, 9, 16], 'o')
```

Либо ещё кучей способов.

In [ ]:
```
plt.plot([1, 2, 3, 4], [1, 4, 9, 16], '-o')
```

Посмотрим, как `numpy` работает в связке с `matplotlib.pyplot`. Вообще это всё очень похое на *MATLAB*.

In [ ]:
```
x = np.linspace(-5, 5, 200)
# это массив из 200 элементов, состоящий из равномерно разбросанных чисел от -5 до 5
```

In [ ]:
```
len(x)
```

In [ ]:
```
x[:10]
```

Вот так можно нарисовать параболу:

In [ ]:
```
plt.plot(x, x**2)
```

Действительно, `x**2` — это массив, элементами которого являются квадраты чисел, лежащих в `x`. Значит, построив график, состоящий из точек, x-координаты которых записаны в `x`, а y-координаты с `x**2`, мы построим график функции $y = x^2$.

А вот, например, синусоида:

In [ ]:
```
plt.plot(x, np.sin(x))
```

А вот что-то посложнее:

In [ ]:
```
plt.plot(x, np.sin(x**2))
```

Вот так можно построить несколько графиков и сделать подписи.

In [ ]:
```
x = np.linspace(-1,1,201)
plt.plot(x,x**2, label = '$y = x^2$')
plt.plot(x,x**3, label = '$y = x^3$')
plt.legend(loc='best')
```

Знак `$` в `label` используется для того, чтобы записывать формулы — это делается в LaTeX (https://en.wikibooks.org/wiki/LaTeX/Mathematics)-нотации и долларами там обозначается начало и конец формулы. (Кстати, в IPython Notebook в ячейках типа `Markdown` тоже можно записывать формулы в LaTeX-нотации.)

Конечно, мы могли бы получить `x` и `y` не в результате вычисления значений какой-то функции, а откуда-то извне. Возьмём для примера случайные числа.

In [ ]:
```
x = np.random.random(100)
```

In [ ]:
```
y = np.random.random(100)
plt.plot(x,y, 'o')
```

Есть и специализированная функция для создания *scatter plot*.

In [ ]:
```
plt.scatter(x,y)
```

Ещё можно построить гистограмму.

In [ ]:
```
plt.hist(x)
```

Можно строить трёхмерные картинки, но тут уже нужна магия и я не буду вдаваться в детали.

In [ ]:
```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure()
```

In [ ]:
```
%matplotlib inline
```

In [ ]:
```
x,y = np.mgrid[-1:1:0.01, -1:1:0.01]
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(x,y,x**2+y**2)
fig
```

Наконец, можно строить интерактивные картинки!

In [ ]:
```
from ipywidgets import interact, interactive, fixed
import ipywidgets as widgets
import numpy as np
import matplotlib.pyplot as plt
```

In [ ]:
```
%matplotlib inline
```

In [ ]:
```
def plot_pic(a, b):
    x = np.linspace(-3,3,200)
    plt.plot(x, np.sin(x*a+b))
```

In [ ]:
```
interact(plot_pic, a=[0, 3, 0.1], b=[0, 3, 0.1])
```

Функция `interact` создаёт несколько бегунков и позволяет с их помощью задавать параметры у функции (в данном случае `plot_pic`), которая строит график.

> Эта картинка не будет интерактивной при просмотре IPython Notebook, но если вы скачаете его и запустите у себя на компьютере, то там будет.

Ещё `interact` можно вызывать так:

In [ ]:
```
@interact(a=[0, 3, 0.1], b=[0, 3, 0.1])
def plot_pic(a, b):
    x = np.linspace(-3,3,200)
    plt.plot(x, np.sin(x*a+b))
```