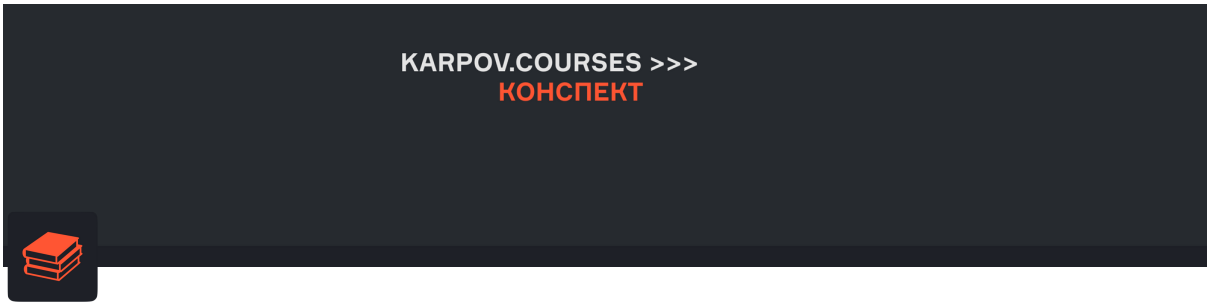


> Конспект > 4 урок > Обзор numpy, pandas, Jupyter. Основы Jupyter

> Оглавление

> Список основных библиотек

На данном этапе уже можно поближе познакомиться с основными инструментами в DS: `numpy`, `pandas`, `matplotlib`. Кроме того, уже есть достаточно знаний в `Jupyter Notebook`, чтобы пользоваться его продвинутыми командами.

Почему будем изучать именно эти библиотеки?

- Без `pandas` не проходит и дня в работе DS: он используется для исследования данных, их преобразования, проверки гипотез, загрузки/выгрузки и многого другого. `pandas` работает довольно быстро, намного быстрее встроенных в `python` возможностей.

- Высокую скорость обработки в pandas обеспечивает библиотека `numpy`. Хотя `numpy` сам по себе используется чуть реже в анализе данных, он очень важен — многие библиотеки для ML «под капотом» используют `numpy` для быстрых вычислений. О нем более подробно поговорим в блоке про машинное обучение.

- Наконец, для любого хорошего отчета необходима визуализация результатов. Для построения графиков чаще всего используется `matplotlib` и основанные на ней библиотеки (`seaborn`, к примеру). Про визуализацию подробнее поговорим в модуле «Машинное обучение».

> Ячейки в Jupyter

Весь `Jupyter Notebook` состоит из ячеек (англ. _*cell*_).

Ячейки в ноутбуке бывают трех типов:

- `code` — содержат Python-код, который Jupyter выполнит.
- `text` — содержат текст в разметке Markdown.
- `raw` — содержат любые символы, которые Jupyter не будет пытаться как-либо выполнить (на практике используется редко).

Немного о Markdown:

Jupyter Notebook поддерживает язык разметки Markdown. Это простой способ создания красиво выглядящих документов с небольшим количеством команд, которые нужно запомнить.

«Язык разметки» — это просто набор соглашений, правил.

Допустим, что вы общаетесь с другом по СМС. В них нельзя сделать текст жирным или наклонным. Вы договариваетесь с другом: если я пишу `*что-то*` вот так между звездочками, то считай, что это наклонный текст. А если я пишу `**что-то**` между двумя звездочками, то считай, что это жирный текст. Вы придумали правила.

Markdown — это набор подобных правил.

Пример использования Markdown в Jupyter:

```
*Этот текст будет наклонным (курсив)*

**Этот текст будет жирным**

_Можно **вставлять** один тип в другой_
```

Результат будет:

Этот текст будет наклонным (курсив)

Этот текст будет жирным

*Можно **вставлять** один тип в другой*

Больше информации

> Горячие клавиши в Jupyter

В Jupyter Notebook много горячих клавиш (англ. _*hotkey*_). Это сочетание клавиш наподобие `Ctrl + C`, которые делают определенное действие.

Чтобы пользоваться горячими клавишами, нужно выйти из режима редактирования ячейки — нажать `esc`. Курсор в тексте должен пропасть. Горячих клавиш много, поэтому дадим список самых популярных:

- `esc` — выйти из режима редактирования ячейки
- `b` (как _*below*_) — создать ячейку внизу с типом `code`
- `a` (как _*above*_) — создать ячейку сверху с типом `code`
- `y` — сделать выбранной ячейку (это ячейка, вокруг которой зеленая рамка) тип `code`
- `m` (как _*markdown*_) — сделать выбранной ячейке тип `text`
- `dd` (как _*delete*_, только дважды) — удалить выбранную ячейку
- `x` — удалить ячейку и сохранить во временную память (не `Ctrl + C` — у Jupyter на ячейки отдельный буфер обмена)
- `c` — скопировать ячейку и сохранить во временную память
- `v` — вставить ячейку из временной памяти под текущей
- `V` (`shift + v`) — вставить ячейку из временной памяти над текущей
- `Ctrl + Enter` — исполнить текущую ячейку
- `Shift + Enter` — исполнить текущую ячейку и выбрать ячейку ниже
- стрелка вниз — выбрать ячейку ниже
- стрелка вверх — выбрать ячейку выше

Больше информации

> Магические команды в Jupyter

Магические команды (англ. _*magic commands*_) — это дополнительная функциональность Jupyter, которая дает возможность менять поведение кода, добавлять подсчеты и чуть-чуть упрощает работу.

Существует два типа магических команд:

- Строчные, обозначенные одним символом `%`. (Команда работает на одной строке кода)
- Ячеечные, обозначенные двойным символом `% %`. (Команда работает над всей ячейкой)

Посмотреть доступные магические команды можно с помощью `%lsmagic`. Возможный результат работы команды:

```
Available line magics:
%alias  %alias_magic  %autoawait  %autocall  %automagic  %autosave  %bookmark  %cat  %cd  %clear  %colors  %conda
%config  %connect_info  %cp  %debug  %dhist  %dirs  %doctest_mode  %ed  %edit  %env  %gui  %hist  %history
%killbgscripts  %ldir  %less  %lf  %lk  %ll  %load  %load_ext  %loadpy  %logoff  %logon  %logstart
%logstate  %logstop  %ls  %lsmagic  %lx  %macro  %magic  %man  %matplotlib  %mkdir  %more  %mv  %notebook
%page  %pastebin  %pdb  %pdef  %pdoc  %pfile  %pinfo  %pinfo2  %pip  %popd  %pprint  %precision  %prun  %psearch
%psource  %pushd  %pwd  %pycat  %pylab  %qtconsole  %quickref  %recall  %rehashx  %reload_ext  %rep  %rerun  %reset
%reset_selective  %rm  %rmdir  %run  %save  %sc  %set_env  %store  %sx  %system  %tb  %time  %timeit  %unalias
%unload_ext  %who  %who_ls  %whos  %xdel  %xmode

Available cell magics:
%%!  %%HTML  %%SVG  %%bash  %%capture  %%debug  %%file  %%html  %%javascript  %%js  %%latex  %%markdown
```

```
%%perl  %%prun  %%pypy  %%python  %%python2  %%python3  %%ruby  %%script  %%sh  %%svg  %%sx  %%system
%%time  %%timeit  %%writefile

Automagic is ON, % prefix IS NOT needed for line magics.
```

Рассмотрим несколько примеров:

```
%%time
# Cell magic - это магия, которая действует на всю ячейку
# cell magic обязательно должна быть первой строкой в ячейке
for i in range(int(1e6)):
    a = i**2
```

Возможный результат:

```
CPU times: user 5.71 s, sys: 59.7 ms, total: 5.77 s
Wall time: 5.84 s
```

```
# Через ! можно сказать jupyter, что вся команду надо выполнить
# в терминале системы
# Jupyter поймет, что это не Python код
!pip install notebook
```

```
import numpy as np
%timeit np.array([i**2 for i in range(int(1e6))])
```

Возможный результат:

```
719 ms ± 48.1 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Оставим ссылки на документацию:

Документация на magic

Хорошая статья с полезными magic-командами

> Kernel в Jupyter

В Jupyter можно выполнять не только Python-код, но и код на других языках! Мы этим пользоваться не будем, но про возможность полезно знать.

За выполнение кода Python в Jupyter отвечает ядро (англ. *kernel*). Нам здесь нужно знать две вещи: ядро можно менять на другое (скажем, то, где больше установленных пакетов) и ядро может зависать.

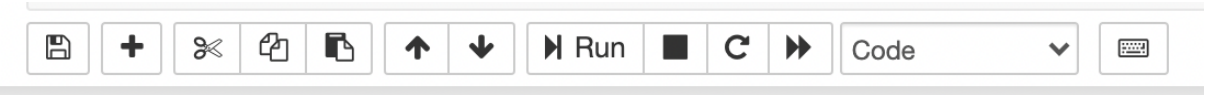
Когда мы выполняем код в Jupyter, часто видим `In [*]`. На деле, эта запись означает другое: она говорит, что ядро занято.

Если ядро зависло или команда выполняется слишком долго, можно его прервать (англ. _interrupt_). Для этого на панели наверху есть кнопка с изображением квадратика.

Ядро содержит в ОЗУ* все когда-либо объявленные переменные ноутбука и может «жить» долго. В больших проектах это может привести к тому, что ядро станет занимать очень много ОЗУ.

Если вам кажется, что ядро «раздулось», до больших размеров, то его можно перезапустить кнопкой, находящейся справа от кнопки прерывания. Перезапуск ядра приведет к тому, что все переменные удалятся и все расчеты потеряются. Будьте готовы.

Кстати, на панельке сверху есть немало операций, которые мы изучили в секции «Горячие клавиши».



*ОЗУ - это компонент, который позволяет компьютеру кратковременно хранить данные и осуществлять быстрый доступ к ним. Компьютер загружает программу или затребованный документ в память из хранилища, а далее обращается к каждой единице информации в оперативной памяти. Многие операции зависят от памяти, поэтому имеющийся объем ОЗУ играет критическую роль в производительности вашей системы.

Больше информации

> Numpy

Numpy — это библиотека для работы с матрицами. Название происходит от Numeric Python. «Под капотом» написана на C++ (это такой язык программирования, очень быстрый в умелых руках), а то, что видим в Python — это лишь «рычаги управления».

Официальный сайт Numpy

Используется как фундамент для построения популярных библиотек в ML: `pandas`, `scikit-learn` и т.п. Вы познакомитесь с `numpy` поближе в модуле «Машинное обучение», а пока мы обзорно посмотрим на его возможности:

```
import numpy as np

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

Результат:

```
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])
```

```
# Матрицу можно транспонировать ("повернуть на бок")
a.T
```

Результат:

```
array([[ 1,  4,  7, 10],
       [ 2,  5,  8, 11],
       [ 3,  6,  9, 12]])
```

```
# Можно считать скалярные произведения
np.dot(a, np.array([1, 2, 3]))
```

Результат:

```
array([14, 32, 50, 68])
```

```
# Обычные умножения делаются поэлементно. Немного непривычно для математиков
b = np.array([[-1, -2, -3], [-4, -5, -6], [-7, -8, -9], [-10, -11, -12]])
a * b
```

Результат:

```
array([[  -1,   -4,   -9],
       [ -16,  -25,  -36],
       [ -49,  -64,  -81],
       [-100, -121, -144]])
```

```
# матричные умножения делаются через np.dot
np.dot(a, b.T)
```

Результат:

```
array([[ -14,  -32,  -50,  -68],
       [ -32,  -77, -122, -167],
       [ -50, -122, -194, -266],
       [ -68, -167, -266, -365]])
```

```
# можно вытащить размерность через .shape - строки x столбцы
a.shape
```

Результат:

```
(4, 3)
```

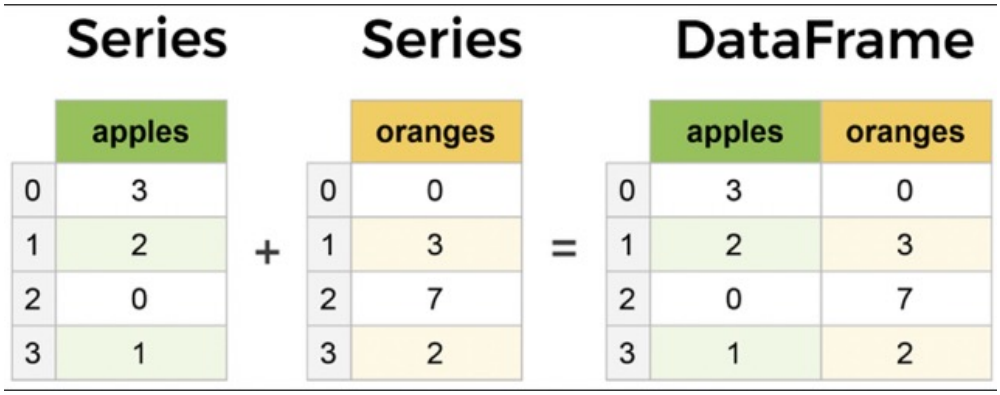
Больше информации

> Pandas

Библиотека для работы с табличными данными. Подробнее познакомимся в следующем уроке, здесь же пройдемся обзорно.

Основные компоненты `pandas` – `Series` и `DataFrame`.

Series – что-то вроде столбца с данными, DataFrame – это таблица, созданная из столбцов Series.



На данном этапе нужно знать, что тип данных DataFrame содержит сотни методов и других операций, которые имеют решающее значение для любого анализа.

Ниже рассмотрим некоторые из них. Что-то пока может быть не совсем понятно, но после следующего блока, посвященного `pandas`, все сложится в единую картину.

```
import pandas as pd  # чаще всего импортируют под именем pd

# Создание датафрейма с двумя колонками из списка кортежей
df = pd.DataFrame([("Python", 6), ("Python", 8), ("ML", 20)],
                  columns=['block', 'lessons'])
df
```

Результат:

	block	lessons
0	Python	6
1	Python	8
2	ML	20

```
# Можно фильтровать записи через логические условия
df[df['block'] == 'Python']
```

Результат:

	block	lessons
0	Python	6
1	Python	8

```
# создавать колонки
result = []
for i in range(1, df.shape[0] + 1):
    result.append(f'course_{i}')

df['#'] = result
df
```

Результат:

	block	lessons	#
0	Python	6	course_1
1	Python	8	course_2
2	ML	20	course_3

```
# Сгруппировать по колонке "block" и вывести сумму в пределах группы для каждой колонки
df.groupby('block').sum()
```

Результат:

	lessons
block	
ML	20
Python	14

```
# Сохраним результат в файл
df.to_csv('1.csv', index=False)

# Прочитаем файл
df_1 = pd.read_csv('1.csv', index_col=0)
df_1
```

Результат:

	lessons	#
block		
Python	6	course_1
Python	8	course_2
ML	20	course_3

Больше информации

> Matplotlib

Популярная библиотека для визуализации данных. Тесно интегрируется с `numpy`. Будем активно использовать в блоке «Машинное обучение», пока что приведем обзор возможностей.

Новичкам, которые пытаются изучать `matplotlib` самостоятельно, рано или поздно начинает казаться, что все слишком запутано и сложно. С одной стороны — все примеры с официального руководства работают, но с другой стороны — что-то свое сделать никак не получается.

Библиотека `matplotlib` предназначена для создания научной графики. Знакомство лучше начать с двух простых вопросов:

- Какие бывают графики?
- Как они строятся?

График линии

Метод построения линии очень прост:

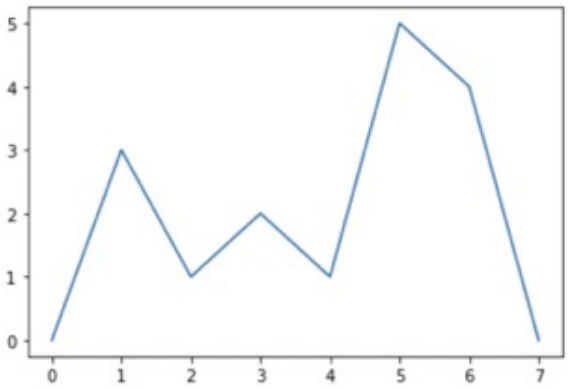
- есть массив абсцисс (`x`);
- есть массив ординат (`y`);
- элементы с одинаковым индексом в этих массивах — это координаты точек на плоскости;
- последовательные точки соединяются линией.

Под массивами подразумеваются списки, кортежи или массивы `NumPy`.

Приведем пример:

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.plot((0, 1, 2, 3, 4, 5, 6, 7), (0, 3, 1, 2, 1, 5, 4, 0))
plt.show()
```

В результате получим следующий график:



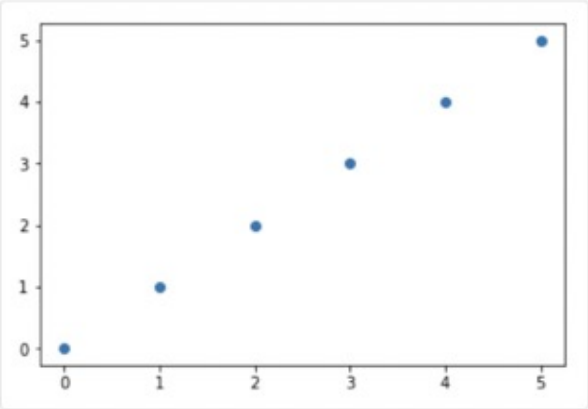
Метод `plt.plot()` в простейшем случае принимает один аргумент — последовательность чисел, которая соответствует оси ординат `(y)`, ось абсцисс `(x)` строится автоматически от `0` до `n`, где `n` — это длинна массива ординат.

График множества точек

Единственное отличие графика множества точек от графика линии — точки не соединяются линией. Вот и все.

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.scatter([0, 1, 2, 3, 4 , 5], [0, 1, 2, 3, 4 , 5])
plt.show()
```

В результате получим следующий график:



Все как и прежде — двум соответствующим значениям из массивов соответствуют координаты точки.

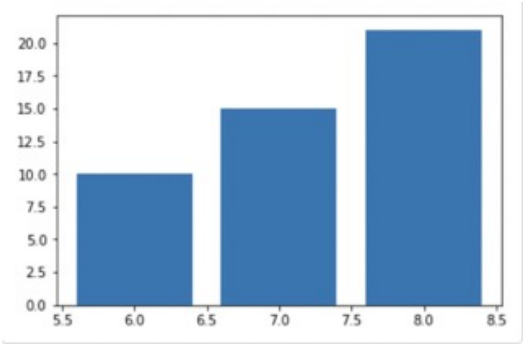
Гистограммы

Очень часто данные удобно представлять в виде гистограмм. В самом простом случае гистограмма — это множество прямоугольников, площадь которых (или высота) пропорциональна какой-нибудь величине. Например, осадки за 3 месяца: в июне выпало 10 мм, в июле — 15мм, в августе — 21 мм.

```
%matplotlib inline
import matplotlib.pyplot as plt

plt.bar([6, 7, 8], [10, 15, 21])
plt.show()
```

В результате получим следующий график:



Первый массив содержит номера месяцев, а второй массив — значения показателей.

В данном степе были приведены основные виды графиков. Больше про виды графиков и про matplotlib можно почитать здесь.