KARPOV.COURSES >>>
КОНСПЕКТ

> Конспект > 6 урок > PYTHON

> Оглавление 6 урока

1. numpy
2. Оконные функции2.1. Скользящее среднее 2.2. Экспоненциальное сглаживание
3. Итерация по нескольким спискам одновременно
4. Общая настройка графиков
5. Настройка графика
6. Построение сложного графика
7. Дефолтные аргументы функций
8. Объединение таблиц по нескольким полям
9. Преобразование континуальной переменной в категориальную
10. distplot
11. plotly
12. Модули и импорты

> NumPy

Модуль для работы с данными (преимущественно численными), на котором основан пандас. Часть методов у этих модулей пересекается, и функции нампая хорошо работают с сериями и датафрэймами.
Традиционно сокращается при импорте как `np`

```
import numpy as np
```

Для примера возьмём 10-й логарифм от серии:

```
x
```

```
0          45061
1         121288
2         102737
3         107564
4           4922
           ...
999995    112583
999996    112583
999997    112583
999998    113350
999999    117353
Name: smth, Length: 1000000, dtype: int64
```

```
np.log10(x)
```

```
0         4.653801
1         5.083818
2         5.011727
3         5.031667
4         3.692142
             ...
999995    5.051473
999996    5.051473
999997    5.051473
999998    5.054422
999999    5.069494
Name: smth, Length: 1000000, dtype: float64
```

То есть, достаточно просто передать внутрь функции объект, к элементам которого мы хотим применить функцию.

Документация

> Оконные функции

Иногда (например, при работе с временными данными) нужно произвести агрегирующие вычисления, захватывающие определённый промежуток данных (не всю колонку). То есть мы будем работать в определённом "окне" значений колонки. Окна бывают разные, простой вариант — *скользящее*.

Для вычисления 1-го значения берётся, допустим, 4 предыдущих значения и то, которое рассчитываем. Для вычисления 2-го берутся значения со сдвигом на 1 (то есть со 2-го по 6-е) и так далее.

```
conversion_df['Conversion_rate'].head(10)
```

```
Date
2014-01-01    0.229167
2014-01-02    0.229968
2014-01-03    0.192201
2014-01-04    0.188793
2014-01-05    0.179035
2014-01-06    0.170029
2014-01-07    0.172693
2014-01-08    0.186263
2014-01-09    0.185567
2014-01-10    0.181296
Name: Conversion_rate, dtype: float64
```

То есть, для скользящего окна с периодом 5 значение 2014-01-05 будет вычисляться по значениям с 01 по 05 включительно. А для 2014-01-06 — по значениям с 02 по 06.

```
conversion_df['Conversion_rate'].rolling(5).mean().head(10)
```

```
Date
2014-01-01         NaN
2014-01-02         NaN
2014-01-03         NaN
2014-01-04         NaN
2014-01-05    0.203833
2014-01-06    0.192005
2014-01-07    0.180550
2014-01-08    0.179363
2014-01-09    0.178717
2014-01-10    0.179170
Name: Conversion_rate, dtype: float64
```

Скользящее окно призывается с помощью метода `rolling`, принимающего период окна. Эта функция похожа на группировку тем, что результат не просматривается, и необходимо применить агрегирующую функцию (в примере выше— mean)

4 первых значения не были вычислены, так как для них нет 5 значений, по которым они бы вычислялись.

Избавление от NaN'ов

Если расчёт каждого нового значения ровно по периоду не важен, можно использовать параметр `min_periods`. Он принимает число, которое указывает, сколько как минимум должно быть значений, чтобы посчитать результат.

```
conversion_df['Conversion_rate'].rolling(5, min_periods=1).mean().head(7)
```

```
Date
2014-01-01    0.229167
2014-01-02    0.229567
2014-01-03    0.217112
2014-01-04    0.210032
2014-01-05    0.203833
2014-01-06    0.192005
2014-01-07    0.180550
Name: Conversion_rate, dtype: float64
```

Документация

> Скользящее среднее

Посмотрим на скользящее среднее более подробно. Периодически требуется не только преобразовать данные, но и сгладить сам ряд. Сглаживание временных рядов позволяет избавиться от шума в данных и более точно увидеть линию общего тренда. Один из способов — использование **скользящего среднего**. Что же это такое?

В pandas есть уже готовая реализация этого метода — `pd.rolling()`. Функция принимает несколько параметров, первый и самый важный из которых `window` — размер окна, также называемый шириной окна. Данный параметр отвечает за число наблюдений, которые используются для подсчета скользящего среднего. К примеру,

$$SMA_t = \frac{1}{n}\sum_{i=0}^{n-1} x_{t-i} = \frac{x_t + x_{t-1} + ... + x_{t-(n-1)}}{n}$$

где

n — размер окна,

t — момент времени.

Предположим, имеется 5 наблюдений:

```
df = pd.DataFrame({'value': [0, 1, 2, 3, 4]})
df
```

```
  value
0     0
1     1
2     2
3     3
4     4
```

Для подсчета скользящего среднего с размером окна 2 вычисления будут выглядеть следующим образом:

$$SMA_t = \frac{x_t + x_{t-1}}{2}$$

где *xt*— значение в текущий момент времени,

xt−1 — в предыдущий.

```
df.rolling(window=2).mean()
```

```
   value
0    NaN
1    0.5
2    1.5
3    2.5
4    3.5
```

Обратите внимание, что теперь на месте самого первого значения стоит NaN, поскольку указанный размер окна 2 подразумевает наличие двух значений для подсчета, и вполне логично, что скользящее среднее для первого наблюдения вычислить не получится, поскольку других значений перед ним нет. Если увеличить размер окна до 3, то NaN будет стоять и вместо второго наблюдения, и так далее.

Еще один вариант использования `pd.rolling()` — **центрированное скользящее среднее**. В таком случае используются наблюдения до, после и во время t*t*. В pandas за это отвечает параметр `center`, который по умолчанию равен False.

Для подсчета центрированного среднего с размером окна 3:

$$SMA_t = \frac{x_{t-1} + x_t + x_{t+1}}{3}$$

где *xt* — значение в текущий момент времени, *xt*−1 — в предыдущий, *xt*+1 — последующий.

```
df.rolling(window=3, center=True).mean()
# (0+1+2)/3=1# (1+2+3)/3=2
```

```
  value
0   NaN
1   1.0
2   2.0
3   3.0
4   NaN
```

> Экспоненциальное сглаживание

Следующий шаг — экспоненциальное сглаживание. В предыдущем подходе (скользящее среднее), использовались n*n* последних наблюдений и все они имели равный вес. В данном случае веса экспоненциально уменьшаются, т.е. более "старые" события имеют меньший вес, а для подсчета используются все имеющиеся наблюдения.

$$EMA_t = \alpha * P_t + (1 - \alpha) * EMA_{t-1}$$

где:

- *α* — веса от 0 до 1; чем выше, тем больший вес имеют новые значения и меньше старые,
- *Pt* — значение в момент времени *t*,
- *EMAt*−1 — значение скользящего среднего в момент *t*−1.

Более подробно про расчет EMA можно почитать здесь.

Для подсчета экспоненциального скользящего среднего в pandas есть метод `ewm()`. Например,

```
  value
0     0
1     1
2     2
3     3
4     4
```

```
df.ewm(span=2).mean()
```

```
      value
0  0.000000
1  0.750000
2  1.615385
3  2.550000
4  3.520661
```

> Итерация по нескольким спискам одновременно

Время от времени вам будет нужно пройтись по нескольким спискам одновременно — то есть получать элементы с одним индексом из каждого из них. Можно сделать это так:

```
nums = [3, 5, 7]
letters = ['all', 'for', 'exoplanets colonization']

for i in range(len(nums)):
    num = nums[i]
    letter = letters[i]
    # do something with them
```

Но есть более удобный способ — функция `zip()`. По сути она позволяет перебирать элементы списков одновременно, возвращая на каждой итерации кортеж с элементами из одинаковой позиции соответствующих списков.

```
nums = [3, 5, 7]
letters ['all', 'for', 'exoplanets colonization']

for i in zip(nums, letters):
    num = i[0]
    letter = i[1]
    # do something with them
```

Кажется, что лучше не стало — в такой записи, и правда, стало ненамного лучше. Но мы можем сразу извлечь `num` и `letter` из кортежа с помощью записи:

```
nums = [3, 5, 7]
letters ['all', 'for', 'exoplanets colonization']

for num, letter in zip(nums, letters):
    # do something with them
```

Связано это с тем, что в питоне можно распаковывать кортежи и списки в переменные, например:

```
num, letter = (3, 'all')  # num == 3 and letter == 'all'
```

Здесь мы присваиваем элементы из кортежа `(3, 'all')` в соответствующие переменные.

Больше информации

> Общая настройка графика

Многие настройки рисования можно установить 1 раз для скрипта (обычно в его начале), избавившись таким образом от повторов:

```
# Font size will be increased, background of figures will be white, grid will be prese
nt and size of plots will be increased

sns.set(
    font_scale=2,
    style="whitegrid",
    rc={'figure.figsize':(20,7)}
        )
```

Документация

> Кастомизация графика

Хорошие графики обладают подписанными осями, заголовком и начинаются от 0 при сравнении между собой нескольких значений (например, на барплоте)!

Есть несколько способов провести кастомизацию графика:

Через объект графика

При создании графика возвращается объект, который хранит о нём информацию. Через него можно задать параметры кастомизации:

```
ax = conversion_df.plot()  # create plot

ax.set_xlabel('Date of orders')  # Label of x axis
ax.set_ylabel('Conversion rate')  # Label of y axis
ax.set_title('Conversion rate by date')  # Title of the plot

y_labels = [str(int(i * 100)) + '%' for i in ax.get_yticks()]  # Prepare custom labels
of axis values

ax.set_yticklabels(y_labels)  # Set new labels
ax.set_xticklabels(labels=conversion_df.index, rotation=90)  # Same just for x axis an
d totate labels to perpendicular configuration

sns.despine()  # Get rid of axis on the plot
```

Через методы matplotlib

```
ax = conversion_df.plot()

plt.xlabel('Date of orders')
plt.ylabel('Conversion rate')
plt.title('Conversion rate by date')

y_labels = [str(int(i * 100)) + '%' for i in ax.get_yticks()]

ax.set_yticklabels(y_labels)
ax.set_xticklabels(labels=conversion_df.index, rotation=90)

sns.despine()
```

Больше информации

> Рецепт для построения графика

Давайте подробнее посмотрим на форматирование графиков, а именно на их расположение, подписи осей, рамки и сетки.

Как построить несколько графиков рядом друг с другом?

Для этого можно использовать `plt.subplots()`. Благодаря этому методу мы разобьём общее полотно графика на части и получим матрицу, где в каждой ячейке можно разместить свой график. Функция принимает несколько параметров:

- `nrows` — число рядов/строк, на которые мы разделим полотно
- `ncols` — число колонок, на которые мы разделим полотно
- `figsize` — размер картинки, общего полотна
- `sharey` — будет ли ось y общей для графиков (пошаренной) и между какими (можно задать общую ось между всеми графиками, между графиками одной строки или одного столбца)
- `sharex` — то же самое для оси x

```
import matplotlib.dates as mdates


fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(20, 10), sharey='col', sharex=True
)
```

Мы указала 2 строки и 2 колонки для графиков (то есть, у нас будет матрица 2 на 2, где можно уместить до 4-х рисунков). Размер картинки указывается как кортеж с шириной и высотой графика. `'col'` в `sharey` указывает на то, что у графиков в каждой колонке будет одинаковая ось у `True` в `sharex` означает, что ось x будет одинаковая у всех графиков

В результате возвращается 2 объекта — сама картинка и оси каждого из графиков (plt — довольно сложная библиотека в плане организации графика), которые мы сразу записываем в переменные `fig` и `axes`

В последнем находится массив из объектов axes:

```
array([[<matplotlib.axes._subplots.AxesSubplot object at 0x11d35dd50>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x11e71df10>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x11e783410>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x11e7bc310>]],
      dtype=object)
```

Это по сути numpy массив 2 на 2 — то есть, матрица. В каждой её ячейке можно нарисовать график независимый от других ячеек (или зависимый по осям, если их зашарить).

Рисуем

Для отрисовки графиков используем цикл. Сначала соединяем с помощью `zip()`:

- значения для параметра `window`
- оси `axes`
- список цветов для каждого из графиков

```
windows = [2, 20, 30, 40]
colors = ['coral', 'blue', 'green', 'purple']

for window, ax, color in zip(windows, axes.flatten(), colors):
    ax.plot(avocado_mean.rolling(window=window).mean(), label=window, color=color)
```

Здесь мы используем метод `.flatten()` у `axes`, чтобы перевести матрицу в одномерный массив и по нему проитерироваться.
Содержимое `axes.flatten()`

```
array([<matplotlib.axes._subplots.AxesSubplot object at 0x11d35dd50>,
       <matplotlib.axes._subplots.AxesSubplot object at 0x11e71df10>,
       <matplotlib.axes._subplots.AxesSubplot object at 0x11e783410>,
       <matplotlib.axes._subplots.AxesSubplot object at 0x11e7bc310>],
      dtype=object)
```

Форматируем

Теперь для каждого графика:

```
for ax in axes.flatten():
    # удаляем рамку
    ax.set_frame_on(False)
    # устанавливаем major locator – 4 января для каждого года
    ax.xaxis.set_major_locator(mdates.MonthLocator(bymonth=1, bymonthday=4))
    # показывать в формате сокращенного названия месяца и дня (Jan 04)
    ax.xaxis.set_major_formatter(mdates.DateFormatter('%b %d'))
    # под major locator - minor locator, т.е. редактируем minor ticks
    ax.xaxis.set_minor_locator(mdates.YearLocator(month=1, day=1))
    # показываем год
    ax.xaxis.set_minor_formatter(mdates.DateFormatter('\n%Y'))
    # делаем сетку графика совсем немного серой и наполовину прозрачной
    ax.grid(True, color='#e2e2e2', alpha=0.5)
```

В результате мы определим, где будут находиться лэйблы тиков и как они будут выглядеть.

Добавляем названия

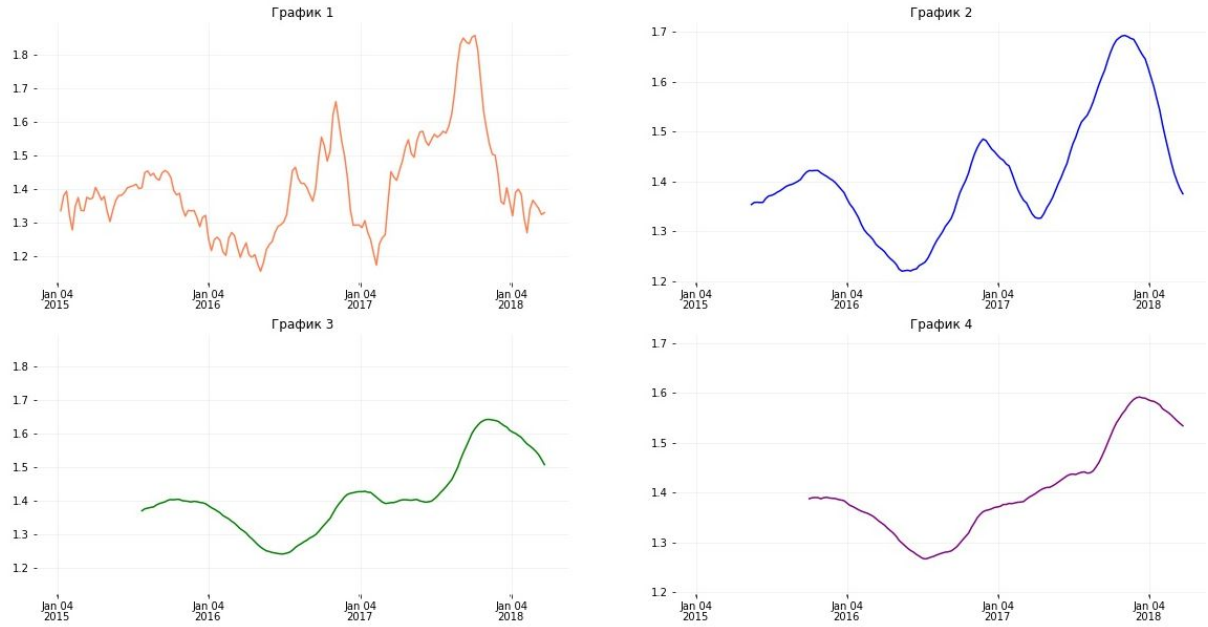
Для каждого рисунка устанавливаем название с помощью `ax.set()` и форматирования строк. Также используем `ax.tick_params()` для изменения внешнего вида ticks.

```
for name, ax in zip(['1','2','3','4'], axes.flatten()):
    ax.set(title='График {}'.format(name))
    ax.tick_params(labelbottom=True, which='both')
```

Здесь мы проитерировались по нашим графикам, задали заголовок каждого (`title`), а также указали, что хотим нарисовать названия у тиков снизу.

Строим 📊

Объединяем кусочки кода в одну ячейку, `plt.show()` и готово!



Документация plt.subplots()
Документация np.array.flatten()

> Дефолтные аргументы функций

Если какой-то из аргументов вашей функции часто имеет одно значение, имеет смысл прописать его по умолчанию. Для этого поставьте после имени аргумента = и укажите значение, которое будет принимать аргумент, если он не получит его при вызове функции. Аргументы по умолчанию должны идти после обычных.

```
def greetings(name='Ra'):
    print('Hi,', name)
```

```
greetings('Tolya')
```

```
Hi, Tolya
```

```
greetings()
```

```
Hi, Ra
```

Больше информации

> Объединение таблиц по нескольким полям

Если указать в параметре `on` функции `pd.merge` список из колонок, то произойдёт объединение по их комбинации. То есть будут объединены строки, где совпадают значения во всех указанных в `on` колонках:

```
# Merge by combination of Company Id and Company Name
orders_with_sales_team = pd.merge(order_leads, sales_team, on=['Company Id','Company N
ame'])
```

Документация

> Преобразование континуальной переменной в категориальную

Если вам нужно перейти от чисел к категориям, воспользуйтесь функцией `pd.cut`. Она принимает массив значений и число интервалов/ список из границ интервалов:

```
values.head()
```

```
0    1
1    6
2    1
3    2
4    4
dtype: int64
```

```
pd.cut(values.head(), 3)
```

```
0    (0.995, 2.667]
1      (4.333, 6.0]
2    (0.995, 2.667]
3    (0.995, 2.667]
4    (2.667, 4.333]
dtype: category
Categories (3, interval[float64]): [(0.995, 2.667] < (2.667, 4.333] < (4.333, 6.0]]
```

```
pd.cut(values.head(), [0, 3, 5, 10])
```

```
0     (0, 3]
1    (5, 10]
2     (0, 3]
3     (0, 3]
4     (3, 5]
dtype: category
Categories (3, interval[int64]): [(0, 3] < (3, 5] < (5, 10]]
```

Как видите, числа заменились на интервалы.

Изменение названий

Для добавления своих названий используется аргумент `labels`, куда подаётся список из названий интервалов:

```
pd.cut(values.head(), [0, 3, 5, 10], labels=['low', 'medium', 'high'])
```

```
0       low
1      high
2       low
3       low
4    medium
dtype: category
Categories (3, object): [low < medium < high]
```

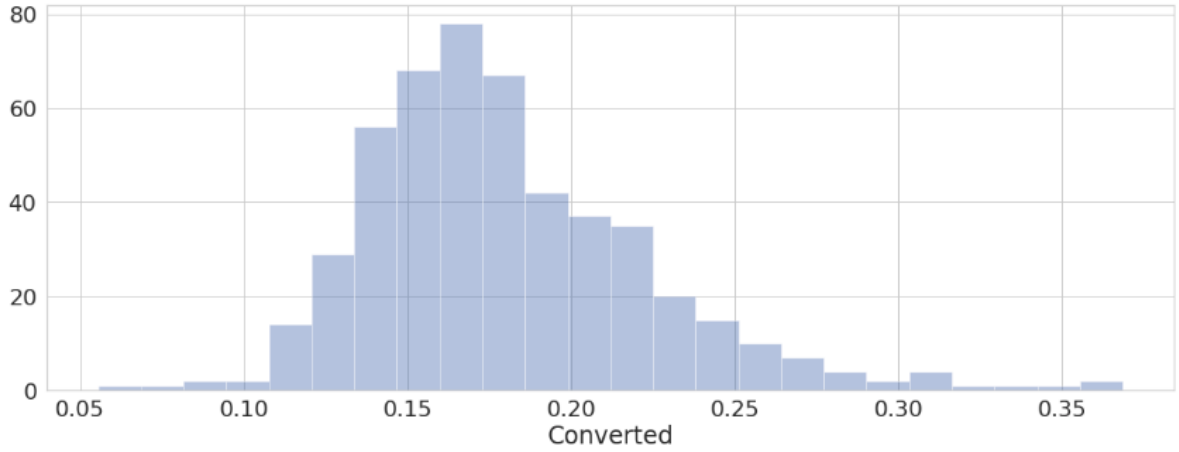
Документация

> distplot

Графики, отражающие распределение значений:

```
sns.distplot(conversion_by_sales, kde=False)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f5e9bec8470>
```

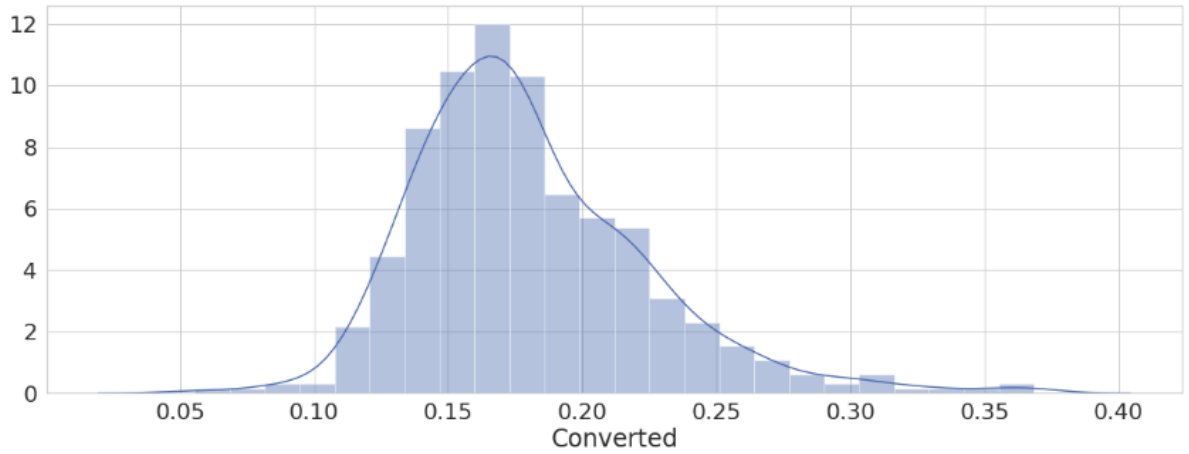


kde

Kernel Density Estimation — аргумент добавляет аппроксимацию распределения, по умолчанию `True`. При этом распределение шкалируется и становится графиком плотности распределения вероятностей *(probability density function)*.

```
sns.distplot(conversion_by_sales['Converted'])
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f5e82ce6cc0>
```



Документация

> plotly

Библиотека для создания интерактивных графиков. Например:

```
import plotly.express as px

px.line(conversion_df, conversion_df.index, conversion_df['Conversion_rate'])
```

Документация

> Модули

Модульность — важное свойство программ, которое обеспечивается языком программирования. Мы можем разбить код программы по нескольким обособленным по смыслу файлам. Это значительно облегчает разработку и тестирование сколько-нибудь сложных приложений.

Небольшой пример: вы написали 10 функций, решающих ваши задачи. Держать их в одном файле и там же вызывать — не очень хорошая идея, потому что получается бардак. Хорошим шагом будет разделение функций от скриптов, где вы непосредственно применяете их

Реализация

Каждый питоновский скрипт (с расширением `py`, не юпитер ноутбук) является модулем. Чтобы получить доступ к его содержимому необходимо произвести импорт. При импортировании файла он целиком исполняется — как если бы вы выполнили его в юпитер ноутбуке.

Для получения своего модуля:

- создайте файл
- напишите там код, который хотите (стандартный вариант — функцию)
- сохраните файл с английским названием, чтобы он начинался с буквы и не содержал пробелов, в конце `.py`

Расположение модулей

Чтобы модуль можно было импортировать, он должен быть виден питону. Он смотрит модули в нескольких местах:

- питоновская папка, куда устанавливаются библиотеки — она находится в глубинах питона
- папка, откуда запущен питон — просто рабочая папка, где вы работаете с ноутбуком
- кастомные места, прописанные в `sys.path` — любое место, которое вы запишете

Самый простой вариант — держать ваш файл-модуль в той же папке, где работаете

Виды импорта

Помимо импорта видов

`import pandas as pd`

и

`import os`

Существуют другие варианты:

- `from my_module import my_function` — импортировать функцию `my_function`, содержащуюся в `my_module.py`После этого её можно использовать в коде как `my_function`
- `from my_module import *`Краткая запись для импорта всех имён из модуля, `my_function` также можно использовать сразу в коде по её имени. Все остальные переменные/функции тоже были выгружены из модуля. Это не рекомендуемая практика при импорте большинства модулей, так как скрипт захламляется именами из модуля (меньше свободных имён для ваших переменных). Но вполне валидно для модулей с небольшим числом имён.

Больше информации