Анализ и визуализация данных

Краткое введение в массивы NumPy и датафреймы Pandas

Массивы NumPy

Сегодня мы познакомимся с библиотекой NumPy (сокращение от *Numeric Python*), которая часто используется в задачах, связанных с анализом данных и машинным обучением.

Чтобы мы смогли на конкретных примерах увидеть, зачем эта библиотека используется, давайте ее импортируем. Если вы уже устанавливали Anaconda, то библиотека NumPy также была установлена на ваш компьютер. Проверим: импортируем библиотеку с сокращенным названием, так часто делают, чтобы не писать каждый раз в коде длинное название. Сокращение `np` для библиотеки `numpy` – общепринятое, его часто можно увидеть в документации или официальных руководствах.

```
import numpy as np
```

Основным объектом NumPy является *Ndarray* – это n-мерный массив (от *n-dimensional array*), структура данных, которая позволяет хранить набор элементов одного типа: либо целые числа, либо числа с плавающей точкой, либо строки, либо логические значения `True` и `False`. Массивы могут быть одномерными, то есть визуально ничем не отличаться от простого списка значений:

```
np.array([0, 2, 3, 4])
```

```
array([0, 2, 3, 4])
```

А могут быть двумерными, то есть представлять собой таблицу, похожую на вложенный список или «список списков»):

```
np.array([[1, 2],
          [1, 0]])
```

```
array([[1, 2],
       [1, 0]])
```

Массивы могут быть и большей размерности (список таблиц или что-то более объемное – вкладывать списки в списки мы можем довольно долго), но на практике они нужны редко.

Зачем изучать массивы? Во-первых, с массивами гораздо приятнее работать, чем со списками, плюс, они занимают меньше памяти. Во-вторых, особенности массивов позволят нам лучше понять, как устроены столбцы в датафреймах (таблицах с данными), с которыми нам предстоит работать дальше.

Для того, чтобы увидеть, почему массивы удобнее списков, рассмотрим такую задачу. У нас есть список `money_k`, который содержит некоторые суммы в кнатах (волшебная валюта).

```
money_k = [210, 265, 570, 120, 180, 194]
```

Как получить новый список `money_s`, где те же суммы записаны в сиклях (1 сикль = 29 кнатов)? Либо создать пустой список и заполнить его через цикл for, либо использовать списковые включения (генераторы списков). Пойдем по второму пути:

```
money_s = [i/29 for i in money_k]
money_s

[7.241379310344827,
 9.137931034482758,
 19.655172413793103,
 4.137931034482759,
 6.206896551724138,
 6.689655172413793]
```

Вроде бы быстро, но без цикла все равно не обошлось. Поступим проще – сделаем из списка массив:

```
Money_k = np.array([210, 265, 570, 120, 180, 194])
Money_k

array([210, 265, 570, 120, 180, 194])
```

А теперь просто разделим его на 29:

```
Money_s = Money_k / 29
Money_s

array([ 7.24137931,  9.13793103, 19.65517241,  4.13793103,
6.20689655,
        6.68965517])
```

Почему такое возможно? Потому что подобные операции производятся поэлементно, то есть над каждым элементом массива в отдельности. Такие операции еще назвают *векторизованными*. То же будет работать и для нескольких массивов. Допустим, у нас есть два нюхлера (ниффлера), которые в течение 3 часов собирают монетки:

```
Niff_one = np.array([83, 73, 65])
Niff_two = np.array([34, 56, 40])
```

Посчитаем, сколько они насобирали вместе за каждый час:

```
Niff_sum = Niff_one + Niff_two
Niff_sum
```

```
array([117, 129, 105])
```

Довольно быстро и удобно!

Важно! Запомните эту особенность массивов, нам она очень пригодится, когда будем работать с датафреймами pandas. Если мы решим сложить столбцы в таблице, они тоже будут складываться поэлементно.

Типы данных в массивах и преобразование типов

Чуть раньше мы зафиксировали, что массивы могут состоять только из элементов одного типа. Посмотрим, что это за типы:

```
# integer
Niff_sum.dtype
```

```
dtype('int32')
```

```
 # float
Money_s.dtype
```

```
dtype('float64')
```

```
# boolean
YN = np.array([True, False])
YN.dtype
```

```
dtype('bool')
```

Числа 64 или 32, дописанные в конце названия типа, зависят от системы (32-битная или 64-битная), на это можно не обращать внимания. А вот на что стоит обратить внимание, так это на то, что после `.dtype` нет круглых скобок. Раньше, когда мы дописывали что-то к объекту после точки, это «что-то» было методом (вспомните методы `.lower()` и `.capitalize()` на строках). Здесь `dtype` – это не метод, а *атрибут* массива, то есть какая-то его характеристика.

Три типа рассмотрели, остались строки. Создадим массив со строками:

```
creatures = np.array(["niffler", "kneazle", "puffskein"])
creatures.dtype
```

```
dtype('<U9')
```

Получили таинственную запись. Но все просто. Буква `U` здесь означает *Unicode* (в этом формате кодируются строки), а 9 – это максимальное число символов в строке внутри массива. Поэтому можем считать это строковым типом, где все строки не длиннее 9 символов.

В завершение разговора о типах посмотрим, что будет, если мы попытаемся поместить в массив объекты разных типов. Пусть у нас будут названия мячей в квиддиче и число очков, которые они приносят:

```
balls = np.array(["quaffle", 10, "snitch", 150])
balls
```

```
array(['quaffle', '10', 'snitch', '150'], dtype='<U11')
```

Как и ожидалось, строковый тип оказался сильнее и вытеснил числа. Если это допустимо, можем один тип превратить в другой. Впомним про массив `YN`:

```
YN
```

```
array([ True, False])
```

Превратим `True` и `False` в целые числа 1 и 0:

```
YN2 = YN.astype('int')
YN2
```

```
array([1, 0])
```

А теперь в обычные строки:

```
YN3 = YN.astype('str')
YN3
```

```
array(['True', 'False'], dtype='<U5')
```

Важно! Запомните этот полезный метод `.astype()`, он нам еще очень пригодится, когда будем работать с датафреймами.

Фильтрация значений по условиям и булевы массивы

Представим себе, что у нас есть массив `points` с числом очков, которые заработала команда за одну игру в квиддич:

```
points = np.array([150, 0, 20, 0, 30, 20, 0])
```

Убедимся, что число игроков в команде правильное – должно быть 7 человек. Вызовем атрибут `size`:

```
points.size  # все ок
```

```
7
```

Теперь поинтересуемся, кто из участников набрал больше 0 очков:

```
points > 0
```

```
array([ True, False,  True, False,  True,  True, False])
```

Неравенство выше было автоматически применено к каждому элементу массива, поэтому мы получили новый массив из `True` и `False`, которые сообщают нам, выполнено ли это условие для конкретного элемента или нет. Как посчитать число игроков, которые заработали больше 0 очков? Посчитать число `True`. А если учесть, что вместо `True` Python видит 1, а вместо `False` – 0? Посчитать сумму всех элементов массива:

```
(points > 10).sum()
```

```
4
```

А как получить массив, в котором будут только те элементы `points`, которые удовлетворяют некоторому условию? Записать это условие в квадратных скобках, как раньше мы указывали индекс элемента:

```
points[points > 10]
```

```
array([150,  20,  30,  20])
```

Запись выше означает, что из `points` Python должен выбрать те элементы, где `points > 10` возвращает `True`.

Если условия сложные, то их нужно формулировать с помощью операторов `&` (одновременное выполнение условий) или `|` (хотя бы одно из условий верно).

```
points[(points > 10) & (points < 30)]
```

```
array([20, 20])
```

«Словесные» операторы `and` и `or` здесь не подойдут. Плюс, всегда нужно ставить скобки вокруг каждой части условия, иначе Python начнет «раскручивать» условие со знаков `&` или `|`, что закончится ошибкой:

```
points[points > 10 & points < 30]  # пытался сопоставить 10 и массив points
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[25], line 1
----> 1 points[points > 10 & points < 30]

ValueError: The truth value of an array with more than one element is ambiguous. Use a.any() or a.all()
```

Если нужны индексы элементов, удовлетворяющих условиям, можно воспользоваться методом `where`:

```
np.where(points > 0)
```

```
(array([0, 2, 4, 5], dtype=int64),)
```

Последовательности pandas Series

Теперь перейдем к объектам из библиотеки pandas. Библиотека pandas – библиотека для более удобной работы с данными в табличном виде (например, файлы Excel) или базами данных.

Как и библиотека NumPy, библиотека pandas была загружена вместе с Anaconda. Импортируем библиотеку с сокращенным названием:

```
import pandas as pd
```

Теперь рассмотрим объект, структуру данных, которая называется *Series* или последовательность pandas. Эта структура является своеобразным звеном между массивом и датафреймом (таблицей). Датафрейм pandas – это набор объектов типа *Series*, а *Series* – это один столбец в таблице.

Создадим пустой *Series*:

```
pd.Series()
```

```
Series([], dtype: object)
```

Series – объект, который связан с массивом NumPy и который наследует многие его атрибуты и методы. Так, у массива, как мы выяснили, были атрибуты `dtype` и `size`, значит, у *Series* тоже такие атрибуты будут. Мы могли привести массив одного типа к другому с помощью метода `.astype()`, значит, с *Series* сможем проделать то же самое. И так далее.

Но объект *Series* похож не только на массив. Давайте создадим массив `Points` со значениями числа очков, которые набрали члены одной команды по квиддичу за одну игру:

```
Points = pd.Series([150, 0, 20, 0, 30, 20, 0])
Points
```

```
0    150
1      0
2     20
3      0
4     30
5     20
6      0
dtype: int64
```

На что похож `Points`? На словарь! На словарь, где ключами являются индесы строк, а значениями – сами значения в столбце. Чтобы стало совсем похоже на те словари, которые мы обсуждали, вместо абстрактных индексов строк добавим имена игроков. Добавим аргумент `index`, в котором перечислим новые названия строк:

```
Points = pd.Series([150, 0, 20, 0, 30, 20, 0],
                   index = ["Harry", "Fred", "Alicia",
                            "George", "Katie", "Angelina", "Oliver"])

Points
```

```
Harry       150
Fred          0
Alicia       20
George        0
Katie        30
Angelina     20
Oliver        0
dtype: int64
```

И, как у словаря, у *Series* есть атрибут `.values`:

```
Points.values
```

```
array([150,   0,  20,   0,  30,  20,   0], dtype=int64)
```

Теперь перейдем к самому главному – к датафреймам (таблицам), которые создаются с помощью библиотеки pandas. Самый простой способ получить датафрейм – загрузить данные из файла csv или Excel и сохранить их как датафрейм. Однако мы начнем с обратной задачи: создадим датафрейм из более простой структуры в Python и выгрузим его в файл.

Рассмотрим словарь `data_dict` с именами игроков, набранными ими очками и их ролью в команде:

```
data_dict = {"Name" : ["Harry", "Fred", "Alicia",
                       "George", "Katie", "Angelina", "Oliver"],
             "Score" : [150,   0,  20,   0,  30,  20,   0],
             "Status": ["seeker", "beater", "chaser",
                        "beater", "chaser", "chaser", "keeper"]}
```

Получим из него датафрейм `df`:

```
df = pd.DataFrame(data_dict)
df
```

```
       Name  Score  Status
0     Harry    150  seeker
1      Fred      0  beater
2    Alicia     20  chaser
3    George      0  beater
4     Katie     30  chaser
5  Angelina     20  chaser
6    Oliver      0  keeper
```

Теперь экспортируем полученный датафрейм в файл Excel и назовем этот файл `scores.xlsx`:

```
df.to_excel("scores.xlsx")
```

Файл `scores.xlsx` автоматически был сохранен в рабочую папку. Вспомним, как проверить, какая папка является рабочей:

```
import os
os.getcwd()  # cwd – current working directory
```

```
'C:\\Users\\Boris Kondratenko\\!2025\\T.2'
```

В моем случае файл `scores.xlsx` нужно искать в папке `Users`.

Из списка списков тоже можно сделать датафрейм:

```
L = [[150,   0,  20,   0,  30,  20,   0],
["seeker", "beater", "chaser",
"beater", "chaser", "chaser", "keeper"]]

pd.DataFrame(L)
```

```
        0       1       2       3       4       5       6
0     150       0      20       0      30      20       0
1  seeker  beater  chaser  beater  chaser  chaser  keeper
```

Однако в таком случае значения будут записываться по строкам. Чтобы это поправить, можем транспонировать полученный датафрейм – поменять местами строки и столбцы:

```
pd.DataFrame(L).T  # T – транспонирование
```

```
     0       1
0  150  seeker
1    0  beater
2   20  chaser
3    0  beater
4   30  chaser
5   20  chaser
6    0  keeper
```

Со списком кортежей будет та же история:

```
L2 = [(150,   0,  20,   0,  30,  20,   0),
("seeker", "beater", "chaser",
"beater", "chaser", "chaser", "keeper")]

pd.DataFrame(L2).T
```

```
     0       1
0  150  seeker
1    0  beater
2   20  chaser
3    0  beater
4   30  chaser
5   20  chaser
6    0  keeper
```

Наконец, датафрейм можно получить из списка словарей:

```
D = [{"name" : "Anna",
    "age" : 23},
    {"name" : "Katie",
    "age" : 23}]

dat = pd.DataFrame(D)
dat
```

```
    name  age
0   Anna   23
1  Katie   23
```

Итак, на что похожи столбцы датафрейма и как создать датафрейм почти «с нуля», мы обсудили. Далее перейдём к загрузке данных из файла и их описанию - откройте следующий ноутбук `pandas-intro`