ОФО Раздел 1 ЛК 3. Библиотеки Numpy и Pandas

Лекция 3. Библиотеки Numpy и Pandas. Часть 1¶

У языка программирования Python очень много возможностей. Для удобства использования многие более специализированные элементы находятся в **библиотеках**, иначе называемых **модулями**. Например, при изучении курса "Алгоритмизация и программирование" мы использовали модуль `math`, предназначенным для работы с математическими функциями.

При решении задач из области искусственного интеллекта не обойтись без двух модулей, обеспечивающих обработку данных.

`numpy` — это модуль Python, предназначенный для работы с матрицами и многомерными массивами. Встроенных двумерных массивов в Python нет.

Мы будем использовать возможности этого модуля совместно с другим модулем,

`pandas` предназначен для обработки и анализа данных без использования специализированных языков программирования. Основная цель — очистка и первичная оценка данных по общим показателям. Многие функции модуля `pandas` базируются на `numpy`.

Библиотека Numpy¶

Numpy — модуль, предоставляющий доступ к математическим операциям и структурам данных. Является базой для всех других библиотек, работающих с искусственным интеллектом.

Установка¶

Если вы используете Google Colab или Anaconda, то Numpy уже установлена на виртуальном сервере и вы можете им пользоваться.

В других средах необходимо выполнить следующую команду:
In [ ]:

```
!pip install numpy
```

Перед использованием библиотеку обязательно нужно подключить. Это делается с помощью команды `import`. По соглашению для подключения библиотеки `NumPy` используется короткое имя `np`.
In [ ]:

```
import numpy as np
```

Особые константы¶

Numpy реализует несколько особых значений через константы. Например:
In [ ]:

```
np.NaN
# not a number - Не числовое значение
```

Out[ ]:

```
nan
```

In [ ]:

```
np.Inf
# infinity - бесконечность
```

Out[ ]:

```
inf
```

Массивы¶

Главная особенность библиотеки `Numpy` — это массивы (array).

Массив — структура данных, позволяющая хранить набор элементов, имеющих один и тот же тип данных, например, целое число или строка.

К элементам массива можно обращаться по их номерам, называемым **индексами**.
In [ ]:

```
# Создание массива из списка
a = np.array([1, 2, 3, 4, 7], float)
print('Array:', a)
print('Тип: ',type(a))
# Обратное преобразование из массива в список
print(a.tolist())
```

```
Array: [1. 2. 3. 4. 7.]
Тип:  <class 'numpy.ndarray'>
[1.0, 2.0, 3.0, 4.0, 7.0]
```

С массивами можно выполнять стандартные операции, доступные для обычных списков: обращаться к элементу по индексу, в том числе отрицательному, делать срезы, присваивать новое значение элементу.
In [ ]:

```
a = np.array([1, 2, 7, 4])

print('1: ', a[0])
```

```
print('2: ', a[1:3])
print('3: ', a[-1])
a[0] = 5
print('4: ', a[0])
```

```
1:  1
2:  [2 7]
3:  4
4:  5
```

Многомерные массивы¶

В numpy можно работать с многомерными массивами.

Многомерный массив — это массив, элементами которого являются другие массивы. Многомерные массивы имеют два или более индексов.

Частным случаем многомерного массива является двумерный массив, который можно представить в виде таблицы или матрицы.

Многомерные массивы широко используются на практике для хранения разнообразных данных. Например, любое черно-белое изображение можно хранить в виде двумерного массива, где координатам каждого пикселя будут соответствовать индексы элемента массива, а значение этого элемента — насыщенность цвета. Цветное изображение можно сохранить с помощью трехмерного массива, где третьей координатой будет цвет пикселя. При обучении нейронных сетей для работы с компьютерным зрением используются четырёхмерные массивы.
Далее мы будем работать в основном с одномерными и двумерными массивами.

Рассмотрим пример создания двумерного массива из нескольких списков. Обратите внимание, что количество элементов в этих списках должно быть одинаковым, лишний или недостающий элемент в одном из них вызовут ошибку.
In [ ]:

```
a = np.array([[1, 2, 3, 2], [4, 5, 6, 8]], int)
print(a)
# Обращение к элементам двумерного массива по индексам
print('1: ', a[0,0])
print('2: ', a[1,0])
print('3: ', a[0,1])
```

```
[[1 2 3 2]
 [4 5 6 8]]
1:  1
2:  4
3:  2
```

В двумерных массивах можно делать срезы, результатами которых могут быть как одномерные, так и двумерные массивы. Срезы выполняются по каждому индексу отдельно.
In [ ]:

```
print('4: ', a[1,:])
print('5: ', a[:,2])
print('6: ', a[-1:, -2:])
```

```
4:  [4 5 6 8]
5:  [3 6]
6:  [[6 8]]
```

Функции и методы для работы с массивами¶

С помощью команды shape можно определять размерности массива.
In [ ]:

```
a.shape
```

Out[ ]:

```
(2, 4)
```

Команда dtype() позволяет узнать тип данных элементов массива. Напомним, что все элементы массива должны иметь одинаковый тип данных.
In [ ]:

```
a.dtype
```

Out[ ]:

```
dtype('int64')
```

Изменение размеров массива¶

In [ ]:

```
a = np.array([[1, 2, 3], [4, 5, 6]], int)
print(a)
```

```
[[1 2 3]
 [4 5 6]]
```

Команда reshape() позволяет изменить форму массива.
In [ ]:

```
b = a.reshape(1,6)
```

Out[ ]:

```
array([[1, 2, 3, 4, 5, 6]])
```

In [ ]:

```
# Обратите внимание, что в процессе изменения размера создан новый массив, а старый не изменился
a
```

Out[ ]:

```
array([[1, 2, 3],
       [4, 5, 6]])
```

С помощью команды `flatten()` можно вытянуть многомерный массив в одномерную строку.
In [ ]:

```
a.flatten()
```

Out[ ]:

```
array([1, 2, 3, 4, 5, 6])
```

In [ ]:

```
# Обратите внимание, что в процессе изменения размера создан новый массив, а не изменён старый
a
```

Out[ ]:

```
array([[1, 2, 3],
       [4, 5, 6]])
```

Способы заполнения массивов¶

В библиотеке Numpy существует аналог команды `range()` — команда `np.arange()`.
In [ ]:

```
print(np.arange(5))
print(np.arange(1, 6, 2))
```

```
[0 1 2 3 4]
[1 3 5]
```

Можно создать массив, заполненный единицами
In [ ]:

```
np.ones((4,4))
```

Out[ ]:

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

Или заполненный нулями
In [ ]:

```
np.zeros((4, 4))
```

Out[ ]:

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

Также можно создать единичную матрицу (матрицу, у которой все элементы равны нулю, и только элементы главной диагонали равны единице). Единичная матрица может быть только квадратной, поэтому у команды `np.identity()` только один параметр.
In [ ]:

```
np.identity(4)
```

Out[ ]:

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

Единицами можно заполнить и другие диагонали.
In [ ]:

```
# k - номер диагонали, заполненный единицами
np.eye(4, 4, k=-1)
```

Out[ ]:

Out[ ]:

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

Перебор элементов массива¶

Перебор элементов массива выполняется с помощью циклов.
In [ ]:

```
a = np.array([1, 4, 5], int)
for x in a:
   print(x)
```

```
1
4
5
```

Простой перебор для многомерного случая перебирает только по первой размерности.
In [ ]:

```
a = np.array([[1, 2], [3, 4], [5, 6]], float)
for x in a:
   print(x)
```

```
[1. 2.]
[3. 4.]
[5. 6.]
```

Поэтому для многомерного случая используется перебор по каждой из размерностей. Для этого используются вложенные циклы.
In [ ]:

```
# перебор правильным способом
for x in range(a.shape[0]):
  for y in range(a.shape[1]):
    print(a[x, y])
```

```
1.0
2.0
3.0
4.0
5.0
6.0
```

Операции над массивами¶

Математические операции над массивами¶

С массивами можно использовать стандартные математические операции. Они будут работать поэлементно. Для матричных операций используются отдельные команды.

Стандартные математические операции применимы только к массивам одинаковых размеров.
In [ ]:

```
a = np.arange(1, 4, 1, dtype=int)
b = np.arange(6, 9, 1, dtype=int)
print('a: ', a)
print('b: ', b)
```

```
a:  [1 2 3]
b:  [6 7 8]
```

In [ ]:

```
a + b
```

Out[ ]:

```
array([ 7,  9, 11])
```

In [ ]:

```
a - b
```

Out[ ]:

```
array([-5, -5, -5])
```

In [ ]:

```
a * b
```

Out[ ]:

```
array([ 6, 14, 24])
```

In [ ]:

```
b / a
```

Out[ ]:

```
array([6.        , 3.5       , 2.66666667])
```

In [ ]:

```
a % b
```

Out[ ]:

```
array([1, 2, 3])
```

In [ ]:

```
b**a
```

Out[ ]:

```
array([  6,  49, 512])
```

In [ ]:

```
a // b
```

Out[ ]:

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

Кроме того, поэлементно могут быть применены другие математические операции
In [ ]:

```
# корень
np.sqrt(a)
```

Out[ ]:

```
array([1.        , 1.41421356, 1.73205081])
```

In [ ]:

```
a = np.array([1.1, 2.5, 1.9], float)
```

In [ ]:

```
# округление вниз
np.floor(a)
```

Out[ ]:

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

In [ ]:

```
# округление вверх
np.ceil(a)
```

Out[ ]:

```
array([2., 3., 2.])
```

In [ ]:

```
# округление по правилам математики (значения вида xxx.5 округляются до ближайшего четного)
np.rint(a)
```

Out[ ]:

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

Простые операции над массивами¶

Одномерные массивы¶

In [ ]:

```
a = np.arange(1, 6, 1)
print(a)
print('Сумма: ', a.sum())
print('Произведение: ', a.prod())
```

```
[1 2 3 4 5]
Сумма:  15
Произведение:  120
```

Статистические характеристики:
In [ ]:

```
# среднее (математическое ожидание)
```

```
# среднее (математическое ожидание)
print(a.mean())
# дисперсия (смещенная - это будет важно в дальнейшем)
print(a.var())
# стандартное отклонение (несмещенное)
print(a.std())
```

```
3.0
2.0
1.4142135623730951
```

Минимальный элемент массива
In [ ]:

```
a.min()
```

Out[ ]:

```
1
```

Номер минимального элемента
In [ ]:

```
a.argmin()
```

Out[ ]:

```
0
```

Команда `clip` позволяет "отрезать" значения сверху и снизу. Значения, не входящие в диапазон, заменяются на границы диапазона.
In [ ]:

```
a = np.array([6, 2, 5, -1, 0, 6, 2, 5, 4], float)
a.clip(0, 5)
```

Out[ ]:

```
array([5., 2., 5., 0., 0., 5., 2., 5., 4.])
```

Выбор только уникальных значений из массива.
In [ ]:

```
np.unique(a)
```

Out[ ]:

```
array([-1.,  0.,  2.,  4.,  5.,  6.])
```

Многомерные массивы¶

Для работы с многомерными массивами можно использовать параметр `axis` (оси).
In [ ]:

```
a = np.array([[5, 2], [4, 1], [3, -1]])
print(a)
print(a.mean(axis=0))
print(a.mean(axis=1))
a.mean()
```

```
[[ 5  2]
 [ 4  1]
 [ 3 -1]]
[4.         0.66666667]
[3.5 2.5 1. ]
```

Out[ ]:

```
2.3333333333333335
```

Логические операции над массивами¶

К массивам можно применять логические операции, например, проверять, какие элементы удовлетворяют некоторому условию.
In [ ]:

```
a = np.array([1, 3, 0])
b = np.array([0, 3, 2])

print(a > b)
```

```
[ True False False]
```

Для удобства условие можно сохранить в отдельной переменной.
In [ ]:

```
c = a > 2
c
```

Out[ ]:

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

In [ ]:

```
# проверяем, что хотя бы один элемент истинен
print(any(c))
# проверяем, что все элементы истинны
print(all(c))
```

```
True
False
```

Такж можно использовать специальные методы

```
np.logical_and(_, _)
np.logical_or(_, _)
np.logical_not(_)
```

In [ ]:

```
(a < 3) * (a > 0)
```

Out[ ]:

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

In [ ]:

```
np.logical_and(a > 0, a < 3)
```

Out[ ]:

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

С помощью np.where можно создать массив на основании условий. Синтаксис:

```
where(boolarray, truearray, falsearray)
```

In [ ]:

```
a = np.array([1, 3, 0, 7, 2, 5, 0])
a
```

Out[ ]:

```
array([1, 3, 0, 7, 2, 5, 0])
```

In [ ]:

```
b = np.where(a != 0, 1 / a, a)
b
```

```
<ipython-input-55-9523604cc2b4>:1: RuntimeWarning: divide by zero encountered in divide
  b = np.where(a != 0, 1 / a, a)
```

Out[ ]:

```
array([1.        , 0.33333333, 0.        , 0.14285714, 0.5       ,
       0.2       , 0.        ])
```

Можно проверять элементы массива на наличие NaN и бесконечностей.
In [ ]:

```
a = np.array([1, np.NaN, np.Inf], float)
a
```

Out[ ]:

```
array([ 1., nan, inf])
```

In [ ]:

```
np.isnan(a)
```

Out[ ]:

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

In [ ]:

```
np.isfinite(a)
```

Out[ ]:

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

Выбор элементов массива по условию¶

Очень важной особенностью массивов является то, что массивы можно создавать, выбирая элементы из других массивов.
In [ ]:

In [ ]:

```
# это результат применения логической операции к многомерному массиву
a = np.array([[6, 4, 2, 1], [5, 7, 8, 9]], float)
a >= 6
```

Out[ ]:

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

In [ ]:

```
# а это результат фильтрации элементов
# обратите внимание, получился одномерный массив, содержащий только элементы, удовлетворяющие условию
a[a >= 6]
```

Out[ ]:

```
array([6., 7., 8., 9.])
```

In [ ]:

```
a[np.logical_and(a > 5, a < 9)]
```

Out[ ]:

```
array([6., 7., 8.])
```

Обратите внимание, что если передать целочисленные значения в качестве условий, то результат будет другой. Будут выбраны элементы массива, стоящие на указанных в массиве условий местах.
In [ ]:

```
a = np.array([2, 4, 6, 8], float)
b = np.array([0, 0, 1, 3, 2], int)
a[b]
```

Out[ ]:

```
array([2., 2., 4., 8., 6.])
```

In [ ]:

```
# Для выбора значений из многомерных массивов необходимо передать целочисленные массивы, которые определяют индексы по каждому из нап
a = np.array([[1, 4], [9, 16]], float)
b = np.array([0, 0, 1, 1, 0], int)
c = np.array([0, 1, 1, 1, 1], int)
a[b,c]
```

Out[ ]:

```
array([ 1.,  4., 16., 16.,  4.])
```

Векторная и матричная математика с использованием numpy¶

Векторные вычисления и матричные вычисления позволяют значительно ускорить обработку численной информации. Мы не будем углубляться в теорию, просто рассмотрим основные операции для работы с матрицами и векторами.
Для двух векторов a и b одинаковой длины **скалярное произведение** считается по следующей формуле:

$ a*b = \sum_{i=0}^{len(a)} a_i*b_i $

В библиотеке numpy имеется функция `dot()`, позволяющая вычислить скалярное произведение векторов.
In [ ]:

```
a = np.array([1, 2, 3], float)
b = np.array([0, 1, 1], float)
np.dot(a, b)
```

Out[ ]:

```
5.0
```

Произведение матриц вычисляется также при помощи команды `dot()`
In [ ]:

```
a = np.array([[0, 1], [2, 3]], float)
b = np.array([2, 3], float)
d = np.array([[1, 1, 1], [1, 1, 1], [1, 1, 1]], float)
```

In [ ]:

```
np.dot(b, a)
```

Out[ ]:

```
array([ 6., 11.])
```

In [ ]:

```
np.dot(a, b)
```

Out[ ]:

```
array([ 3., 13.])
```

In [ ]:

```
# следите за размерностью, иначе ничего не получится. Попытка перемножить эти матрицы вызовет ошибку!
np.dot(b, d)
```

Многие математические операции, связанные с линейной алгеброй, реализованы в модуле linalg, содержащемся в numpy. Например, **определитель** матрицы можно вычислить так
In [ ]:

```
np.linalg.det(a)
```

Out[ ]:

```
-2.0
```

Лекция 3. Библиотеки Numpy и Pandas. Часть 2¶

Модуль pandas¶

pandas предназначен для обработки и анализа данных без использования специализированных языков программирования. Основная цель — очистка и первичная оценка данных по общим показателям.

Поскольку pandas работает на основе библиотеки numpy, перед использованием необходимо подключить обе библиотеки. Для подключения модулей необходимо выполнить следующие команды.
In [ ]:

```
import pandas as pd
import numpy as np
```

Ряды (Series) и датафреймы (Dataframe)¶

Series представляют собой одномерные массивы, у каждого элемента такого массива имеется индекс. Если индекс явно не задан, то pandas автоматически создаёт RangeIndex от 0 до N-1, где N — количество элементов. Ряды очень похожи на словари Python, т.к. ключами могут выступ элементы любых типов.

Для рядов можно делать выборки, фильтровать, применять математические операции и т.д.
In [ ]:

```
s1 = pd.Series(data=[10, -11, 12, 13, -14], index=[1, 'a', 3, 5, 7]) # Явное задание данных и индексов для ряда
s2 = pd.Series(['Red', 'Green', 'Blue', 'Black']) # Создание ряда из списка
l = [[1, 2]]
s3 = pd.Series(l*5) # Создание ряда из списка списков с использованием арифметических операций
# Создание ряда из словаря
s4 = pd.Series({'Homer': 'Dad',
                'Marge': 'Mom',
                'Bart': 'Son',
                'Lisa': 'Daughter',
                'Maggie': 'Daughter'})
print(s1)
print(s2)
print(s3)
print(s4)

print(s1[5]) # Обращение к элементу ряда
print(s2[[0, 2]]) # Выборка элементов ряда
print(s1[s1 > 0]) # Фильтрация только положительных элементов ряда
print(s1 > 0)
```

```
1    10
a   -11
3    12
5    13
7   -14
dtype: int64
0      Red
1    Green
2     Blue
3    Black
dtype: object
0    [1, 2]
1    [1, 2]
```

```
1    [1, 2]
2    [1, 2]
3    [1, 2]
4    [1, 2]
dtype: object
Homer           Dad
Marge           Mom
Bart            Son
Lisa       Daughter
Maggie     Daughter
dtype: object
13
0     Red
2    Blue
dtype: object
1    10
3    12
5    13
dtype: int64
1     True
a    False
3     True
5     True
7    False
dtype: bool
```

`DataFrame` — табличная структура данных, где столбцами являются объекты `Series`.
In [ ]:

```
# Создадим датафрейм из словаря
df = pd.DataFrame({
   'country': ['Kazakhstan', 'Russia', 'Belarus', 'Uzbekistan'],
    'population': [17.04, 143.5, 9.5, 36],
    'area': [2724902, 17125191, 207600, 448924]
})
df # Это тоже вывод, доступный в блокнотах, однако его можно использовать только один раз на ячейку
```

Out[ ]:

	country	population	area
0	Kazakhstan	17.04	2724902
1	Russia	143.50	17125191
2	Belarus	9.50	207600
3	Uzbekistan	36.00	448924

Каждый столбец является рядом. Чтобы в это убедиться, узнаем тип данных у любого столбца.
In [ ]:

```
type(df['country'])
```

Out[ ]:

```
pandas.core.series.Series
```

У объекта `DataFrame` два индекса, по строкам и по столбцам. Если индекс по строкам явно не задан, то `pandas` автоматически создает его так же, как и для рядов: `RangeIndex` от `0` до `N-1`, где `N` — количество элементов.
In [ ]:

```
df = pd.DataFrame([[10, 11], [20, 21], [30, 31]],
                  columns=['A', 'B'])
df
```

Out[ ]:

	A	B
0	10	11
1	20	21
2	30	31

Функции numpy для заполнения рядов и датафреймов¶

В numpy содержатся функции, которые можно использовать для заполнения рядов или столбцов датафрейма.

`np.arange(start, stop, step)` — создает последовательность чисел от `start` до `stop-1` с шагом `step`. Синтаксис:
In [ ]:

```
s = pd.Series(np.arange(15,25,2))
s
```

Out[ ]:

```
0    15
1    17
2    19
3    21
4    23
dtype: int64
```

`np.linspace(start, stop, number)` — создает последовательность из `number` значений начиная от `start` и заканчивая `stop`.
In [ ]:

```
s = pd.Series(np.linspace(0, 7, 5))
s
```

Out[ ]:

```
0    0.00
1    1.75
2    3.50
3    5.25
4    7.00
dtype: float64
```

Для работы со случайными значениями нужно задать зерно для генератора с помощью команды

`np.random.seed(целое_положительное_число)`.

Это поможет в будущем воспроизводить результаты при повторных запусках. Далее можно использовать команду `np.random.normal` для задани
нормально распределенных случайных чисел.

Создадим ряд из 7 нормально распределенных случайных чисел.
In [ ]:

```
np.random.seed(42)
s = pd.Series(np.random.normal(size=50))
s
```

Out[ ]:

```
0     0.496714
1    -0.138264
2     0.647689
3     1.523030
4    -0.234153
5    -0.234137
6     1.579213
7     0.767435
8    -0.469474
9     0.542560
10   -0.463418
11   -0.465730
12    0.241962
13   -1.913280
14   -1.724918
15   -0.562288
16   -1.012831
17    0.314247
18   -0.908024
19   -1.412304
20    1.465649
21   -0.225776
22    0.067528
23   -1.424748
24   -0.544383
25    0.110923
26   -1.150994
27    0.375698
28   -0.600639
29   -0.291694
30   -0.601707
31    1.852278
32   -0.013497
33   -1.057711
34    0.822545
35   -1.220844
36    0.208864
37   -1.959670
38   -1.328186
39    0.196861
40    0.738467
```

```
41    0.171368
42   -0.115648
43   -0.301104
44   -1.478522
45   -0.719844
46   -0.460639
47    1.057122
48    0.343618
49   -1.763040
dtype: float64
```

Создадим датафрейм из 4 строк и 3 столбцов, содержащих случайные значения. Здесь мы помимо генерации последовательности из 12 чисел используем ее преобразование в двумерный массив с помощью метода reshape.
In [ ]:

```
np.random.seed(42)
df = pd.DataFrame(np.random.normal(size=12).reshape(4, 3),
                  index=['ind_1', 'ind_2', 'ind_3', 'ind_4'],
                  columns=['col_1', 'col_2', 'col_3'])
df
```

Out[ ]:

	col_1	col_2	col_3
ind_1	0.496714	-0.138264	0.647689
ind_2	1.523030	-0.234153	-0.234137
ind_3	1.579213	0.767435	-0.469474
ind_4	0.542560	-0.463418	-0.465730

Работа с файлами данных¶

На практике, как правило, датафреймы используются для анализа таблиц данных, содержащихся в файлах. Для работы с файлами необходимо предварительно подключить работу с гугл-диском.
In [ ]:

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

Чаще всего файлы с данными имеют формат *.csv (Comma Separated Values — разделенные запятой значения). Преимущество таких файлов в том, что они по структуре являются текстовыми, т.е. их удобно открывать и читать из программы, а также осуществлять в них запись. С другой стороны, файлы *.csv благодаря своей четкой структуре могут быть открыты в табличных процессорах (MS Excel и др.), что удобно для аналитиков. Правда, сохранить в них дополнительные элементы (формулы, диаграммы и т.п.) не удастся, но можно скопировать данные и анализировать их уже в другом файле нужного формата.

В Python для работы с такими файлами используются объекты типа Dataframe.

На гугл-диске курса содержится файл sp500.csv. Это реальные данные, предоставленные для учебного использования Корпоративным университетом Сбербанка. На примере этого файла мы будем рассматривать работу с датафреймами. Для запуска примера у себя вам нужно скачать файл sp500.csv и загрузить его на свой гугл-диск в подходящую папку.

pd.read_csv — считывает данные из файла с указанными параметрами и возвращает объект типа DataFrame (sep — разделитель).
In [ ]:

```
sp500 = pd.read_csv("/content/drive/My Drive/Colab Notebooks/sp500.csv", sep = ',')
sp500.head() # Выводит на экран только первые пять записей. Удобно для того, чтобы убедиться в открытии файла
```

Если нужно прочитать не весь файл, а только некоторые столбцы, можно воспользоваться параметрами pd.read_csv. Это проще, чем читать вс удалять лишнее.
In [ ]:

```
sp500 = pd.read_csv(filepath_or_buffer = "/content/drive/My Drive/Colab Notebooks/sp500.csv", # Имя файла
                    sep = ',', # Разделитель
                    usecols=['Symbol', 'Sector', 'Price', 'Book Value'], # Нужные столбцы
                    index_col='Symbol') # Чтобы использовать значения столбца в качестве индекса
sp500
```

Out[ ]:

Symbol	Sector	Price	Book Value
MMM	Industrials	141.14	26.668
ABT	Health Care	39.60	15.573
ABBV	Health Care	53.95	2.954
ACN	Information Technology	79.79	8.326

ACE	Financials	102.91	86.897
...	...	...	...
YHOO	Information Technology	35.02	12.768
YUM	Consumer Discretionary	74.77	5.147
ZMH	Health Care	101.84	37.181
ZION	Financials	28.43	30.191
ZTS	Health Care	30.53	2.150

500 rows × 3 columns

У датафреймов есть возможность выбирать отдельные столбцы (результат будет иметь тип `Series`) и группы столбцов (в результате получим `DataFrame`).

In [ ]:

```
sp500['Sector'].head()
```

Out[ ]:

```
Symbol
MMM                Industrials
ABT                Health Care
ABBV               Health Care
ACN     Information Technology
ACE                 Financials
Name: Sector, dtype: object
```

In [ ]:

```
type(sp500['Sector'])
```

Out[ ]:

```
pandas.core.series.Series
```

In [ ]:

```
sp500[['Price', 'Book Value']].head()
```

Out[ ]:

	Price	Book Value
Symbol		
MMM	141.14	26.668
ABT	39.60	15.573
ABBV	53.95	2.954
ACN	79.79	8.326
ACE	102.91	86.897

In [ ]:

```
type(sp500[['Price', 'Book Value']])
```

Out[ ]:

```
pandas.core.frame.DataFrame
```

Аналогичным образом происходит выборка по строкам. Для этого можно использовать метод `loc`.

In [ ]:

```
sp500.loc['MMM']
```

Out[ ]:

```
Sector        Industrials
Price              141.14
Book Value         26.668
Name: MMM, dtype: object
```

In [ ]:

```
sp500.loc[['MMM', 'MSFT']]
```

Out[ ]:

	Sector	Price	Book Value
Symbol			
MMM	Industrials	141.14	26.668
MSFT	Information Technology	40.12	10.584

Для отбора строк по индексу можно использовать метод `iloc`.

In [ ]:

```
sp500.iloc[[0, 3]]
```

Out[ ]:

Symbol	Sector	Price	Book Value
MMM	Industrials	141.14	26.668
ACN	Information Technology	79.79	8.326

Также можно выполнять выборку сразу и по строкам, и по столбцам.
In [ ]:

```
sp500.loc[['ABT', 'ZTS']][['Sector', 'Price']]
```

Out[ ]:

Symbol	Sector	Price
ABT	Health Care	39.60
ZTS	Health Care	30.53

Срезы данных¶

Срезы данных нужны для получения выборок по заданным условиям.
In [ ]:

```
# Выбираем первые 7 строк
sp500.iloc[:7]
```

Out[ ]:

Symbol	Sector	Price	Book Value
MMM	Industrials	141.14	26.668
ABT	Health Care	39.60	15.573
ABBV	Health Care	53.95	2.954
ACN	Information Technology	79.79	8.326
ACE	Financials	102.91	86.897
ACT	Health Care	213.77	55.188
ADBE	Information Technology	64.30	13.262

In [ ]:

```
# Выбираем строки, начиная с метки ABT и заканчивая меткой ACN
sp500.loc['ABT':'ACN']
```

Out[ ]:

Symbol	Sector	Price	Book Value
ABT	Health Care	39.60	15.573
ABBV	Health Care	53.95	2.954
ACN	Information Technology	79.79	8.326

Копирование и удаление:
In [ ]:

```
sp500_copy = sp500.copy()
del sp500_copy['Price']
sp500_copy.iloc[:2]
```

Out[ ]:

Symbol	Sector	Book Value
MMM	Industrials	26.668
ABT	Health Care	15.573

Фильтрация по условию¶

In [ ]:

```
# Выясняем, у каких строк цена меньше 100
sp500.Price < 100
```

Out[ ]:

```
Symbol
MMM     False
ABT      True
ABBV     True
ACN      True
```

```
ACE     False
        ...
YHOO     True
YUM      True
ZMH     False
ZION     True
ZTS      True
Name: Price, Length: 500, dtype: bool
```

In [ ]:

```
# Отбираем нужные строки
sp1 = sp500[sp500.Price < 100]
sp1
```

Out[ ]:

Symbol	Sector	Price	Book Value
ABT	Health Care	39.60	15.573
ABBV	Health Care	53.95	2.954
ACN	Information Technology	79.79	8.326
ADBE	Information Technology	64.30	13.262
AES	Utilities	13.61	5.781
...	...	...	...
XYL	Industrials	38.42	12.127
YHOO	Information Technology	35.02	12.768
YUM	Consumer Discretionary	74.77	5.147
ZION	Financials	28.43	30.191
ZTS	Health Care	30.53	2.150

407 rows × 3 columns

In [ ]:

```
# извлекаем лишь те строки, в которых значение Price < 10 и > 6
r = sp500[(sp500['Price'] < 10) &
          (sp500.Price > 6)] ['Price']
r
```

Out[ ]:

```
Symbol
HCBK    9.80
HBAN    9.10
SLM     8.82
WIN     9.38
Name: Price, dtype: float64
```

In [ ]:

```
# извлекаем строки, в которых переменная Sector принимает значение Health Care, а переменная Price больше или равна 100.00
r = sp500[(sp500.Sector == 'Health Care') &
          (sp500.Price >= 100.00)] [['Price', 'Sector']]
r
```

Out[ ]:

Symbol	Price	Sector
ACT	213.77	Health Care
ALXN	162.30	Health Care
AGN	166.92	Health Care
AMGN	114.33	Health Care
BCR	146.62	Health Care
BDX	115.70	Health Care
BIIB	299.71	Health Care
CELG	150.13	Health Care
HUM	124.49	Health Care
ISRG	363.86	Health Care
JNJ	100.98	Health Care
LH	100.75	Health Care
MCK	183.75	Health Care
PRGO	138.63	Health Care
REGN	297.77	Health Care
TMO	115.74	Health Care
WAT	100.54	Health Care
WLP	108.82	Health Care

ZMH	101.84	Health Care

In [ ]:

```
# Проверка, есть ли строчки из категорий 'Information Technology' и 'Financials' и вывод этих строчек
s_tmp = sp500.Sector.isin(['Information Technology', 'Financials'])
sp500[s_tmp]#.head()
```

Out[ ]:

Symbol	Sector	Price	Book Value
ACN	Information Technology	79.79	8.326
ACE	Financials	102.91	86.897
ADBE	Information Technology	64.30	13.262
AFL	Financials	61.31	34.527
AKAM	Information Technology	53.65	15.193
...	...	...	...
XRX	Information Technology	12.06	10.471
XLNX	Information Technology	46.03	10.247
XL	Financials	32.47	37.451
YHOO	Information Technology	35.02	12.768
ZION	Financials	28.43	30.191

146 rows × 3 columns

In [ ]:

```
# Использование метода query() -- создание запроса
q = sp500.query("Sector=='Health Care' & Price >= 100")[['Price', 'Sector']]
q
```

Out[ ]:

Symbol	Price	Sector
ACT	213.77	Health Care
ALXN	162.30	Health Care
AGN	166.92	Health Care
AMGN	114.33	Health Care
BCR	146.62	Health Care
BDX	115.70	Health Care
BIIB	299.71	Health Care
CELG	150.13	Health Care
HUM	124.49	Health Care
ISRG	363.86	Health Care
JNJ	100.98	Health Care
LH	100.75	Health Care
MCK	183.75	Health Care
PRGO	138.63	Health Care
REGN	297.77	Health Care
TMO	115.74	Health Care
WAT	100.54	Health Care
WLP	108.82	Health Care
ZMH	101.84	Health Care

В библиотеке pandas есть еще много различных функций для анализа данных, дополнительную информацию можно посмотреть в официальной документации.

Назад Дальше