Занятие 3. Библиотеки numpy и pandas

Введение

Библиотека NumPy (Numerical Python), впервые появившаяся в 2006 году, считается основной реализацией массивов Python. Она предоставляет высокопроизводительный, полнофункциональный тип n-мерного массива, который называется **ndarray** (в дальнейшем мы будем называть его синонимом **array**).
NumPy — одна из многих библиотек с открытым кодом, устанавливаемых в дистрибутиве Anaconda Python. Операции с array выполняются на два порядка быстрее, чем операции со списками. В мире больших данных, в которых приложениям приходится выполнять серьезную обработку огромных объёмов данных на базе массивов, этот прирост быстродействия может оказаться критичным. Согласно libraries.io, свыше 450 библиотек Python зависят от NumPy. Многие популярные библиотеки data science, такие как pandas, SciPy (Scientific Python) и Keras (глубокое обучение), построены на базе NumPy или зависят от последней.
В этом занятии исследуются базовые возможности array.
Как известно, списки могут быть многомерными. Обычно многомерные списки обрабатываются во вложенных циклах или с использованием генераторов списков с несколькими секциями for. Сильной стороной NumPy является «программирование, ориентированное на массивы», использующее программирование в функциональном стиле с внутренними итерациями; работа с массивами становится компактной и прямолинейной, а код избавляется от ошибок, которые могут возникнуть во внешних итерациях или в явно запрограммированных циклах.
В разделе «Введение в data science» этого занятия начнется ваше знакомство с библиотекой **pandas**, которая будет использоваться во многих практических примерах при изучении data science. В приложениях больших данных часто возникает необходимость в коллекциях более гибких, чем массивы NumPy, — коллекциях с поддержкой смешанных типов данных, нестандартного индексирования, отсутствующих данных, данных с нарушенной структурой и данных, которые должны быть приведены к форме, более подходящей для баз данных и пакетов анализа данных, с которыми вы работаете.

Создание массивов на основе существующих данных

Документация NumPy рекомендует импортировать *модуль numpy* под именем np, чтобы к его компонентам можно было обращаться с префиксом "np.

In [1]:

```
1 import numpy as np
```

Модуль numpy предоставляет различные функции для создания массивов. В данном случае будет использоваться функция array, которая получает в аргументе массив или другую коллекцию и возвращает новый массив с элементами своего аргумента. Передадим при вызове список:

In [2]:

```
1 numbers = np.array([2, 3, 5, 7, 11])
```

Функция array копирует содержимое своего аргумента в массив. Проверим тип объекта, возвращенного функцией array, и выведем его содержимое:

In [4]:

```
1 type(numbers)
```

Out[4]:

```
numpy.ndarray
```

In [5]:

```
1 numbers
```

Out[5]:

```
array([ 2,  3,  5,  7, 11])
```

Заметим, что для объекта указан тип numpy.ndarray, но при выводе массива используется обозначение «array». При выводе array NumPy отделяет каждое значение от следующего запятой и пробелом и выравнивает все значения по правому краю поля постоянной ширины. Ширина поля определяется на основании значения, занимающего наибольшее количество знаков при выводе. В данном случае значение 11 занимает два знака, поэтому все значения форматируются по полям из двух символов (поэтому символы [и 2 отделены друг от друга начальным пробелом).

Многомерные аргументы

Функция array копирует размерности своего аргумента. Создадим объект array на основе списка из двух строк и трех столбцов:

In [6]:

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

Out[6]:

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

NumPy автоматически форматирует array на основании количества их измерений и выравнивает столбцы в каждой строке.

Атрибуты array

Объект array предоставляет атрибуты для получения информации об их структуре и содержимом. В этом разделе будут использоваться следующие объекты array:

In [7]:

```
import numpy as np
integers = np.array([[1, 2, 3], [4, 5, 6]])
integers
```

Out[7]:

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

In [8]:

```
floats = np.array([0.0, 0.1, 0.2, 0.3, 0.4])
floats
```

Out[8]:

```
array([0. , 0.1, 0.2, 0.3, 0.4])
```

Функция array определяет тип элемента array на основании элементов ее аргументов. Для проверки типа элементов можно воспользоваться атрибутом dtype типа array:

In [9]:

```
integers.dtype
```

Out[9]:

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

In [10]:

```
floats.dtype
```

Out[10]:

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

Как будет показано в следующем разделе, различные функции создания array получают ключевой аргумент dtype для определения типа элементов array. Ради высокого быстродействия библиотека NumPy написана на языке программирования C и в ней используются типы данных C. По умолчанию NumPy сохраняет целые числа в виде значений типа int64 библиотеки NumPy, соответствующих 64-разрядным (8-байтовым) целым числам C, а числа с плавающей точкой — в виде значений типа float64 библиотеки NumPy. В наших примерах чаще всего будут встречаться типы int64, float64, bool (логический тип) и object для нечисловых данных (например, строк). Полный список поддерживаемых типов приведен по адресу https://docs.scipy.org/doc/numpy/user/basics.types.html (https://docs.scipy.org/doc/numpy/user/basics.types.html).

Определение размерности array

Атрибут ndim содержит количество измерений array, а атрибут shape содержит кортеж, определяющий размерность array:

In [11]:

```
integers.ndim
```

Out[11]:

```
2
```

In [12]:

```
floats.ndim
```

Out[12]:

```
1
```

In [13]:

```
integers.shape
```

Out[13]:

```
(2, 3)
```

In [14]:

```
floats.shape
```

Out[14]:

```
(5,)
```

Здесь integers состоит из двух строк и трех столбцов (6 элементов), а структура данных floats является одномерной, поэтому фрагмент выше выводит кортеж из одного элемента (на что указывает запятая) с количеством элементов floats (5)

Определение количества элементов и размера элементов

Общее количество элементов в array можно получить из атрибута size, а количество байтов, необходимое для хранения каждого элемента, — из атрибута itemsize:

In [15]:

```
integers.size
```

Out[15]:

```
6
```

In [16]:

```
integers.itemsize # 4 для компиляторов C с 32-разрядными int
```

Out[16]:

```
4
```

In [17]:

```
floats.size
```

Out[17]:

```
5
```

In [18]:

```
floats.itemsize
```

Out[18]:

```
8
```

Обратите внимание: размер integers равен произведению значений из кортежа shape — две строки по три элемента, итого шесть элементов. В каждом случае значение itemsize равно 8, потому что integers содержит значения int64, а floats содержит значения float64; каждое занимает 8 байт.

Перебор элементов многомерной коллекции array

Обычно для работы с array используются компактные конструкции программирования в функциональном стиле. Так как array являются итерируемыми объектами, при желании можно использовать внешние итерации:

In [19]:

```
for row in integers:
    for column in row:
        print(column, end=' ')
    print()
```

```
1 2 3
4 5 6
```

Чтобы перебрать элементы многомерной коллекции array так, как если бы она была одномерной, используйте атрибут flat:

In [20]:

```
for i in integers.flat:
    print(i, end=' ')
```

```
1 2 3 4 5 6
```

Заполнение array конкретными значениями

NumPy предоставляет функции zeros, ones и full для создания коллекций array, содержащих 0, 1 или заданное значение соответственно. По умолчанию функции zeros и ones создают коллекции array, содержащие значения float64.
Вскоре я покажу, как настроить тип элементов. Первым аргументом этих функций должно быть целое число или кортеж целых чисел, определяющий нужные размеры. Для целого числа каждая функция возвращает одномерную коллекцию array с заданным количеством элементов:

In [21]:

```
1 import numpy as np
```

In [22]:

```
1 np.zeros(5)
```

Out[22]:

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

Для кортежа целых чисел эти функции возвращают многомерную коллекцию array с заданными размерами. При вызове функций zeros и ones можно задать тип элементов при помощи ключевого аргумента dtype:

In [23]:

```
1 np.ones((2, 4), dtype=int)
```

Out[23]:

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

Коллекция array, возвращаемая full, содержит элементы со значением и типом второго аргумента:

In [24]:

```
1 np.full((3, 5), 13)
```

Out[24]:

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

Создание коллекций array по диапазонам

NumPy предоставляет оптимизированные функции для создания коллекций array на базе диапазонов. Мы ограничимся простыми равномерными диапазонами целых чисел и чисел с плавающей точкой, помня, впрочем, что NumPy поддерживает и нелинейные диапазоны.

Создание диапазонов функцией arange

Воспользуемся функцией arange библиотеки NumPy для создания целочисленных диапазонов — по аналогии со встроенной функцией range. Функция arange сначала определяет количество элементов в полученной коллекции array, выделяет память, а затем сохраняет заданный диапазон значений в array:

In [25]:

```
1 import numpy as np
2 np.arange(5)
```

Out[25]:

```
array([0, 1, 2, 3, 4])
```

In [26]:

```
1 np.arange(5, 10)
```

Out[26]:

```
array([5, 6, 7, 8, 9])
```

In [27]:

```
1 np.arange(10, 1, -2)
```

Out[27]:

```
array([10,  8,  6,  4,  2])
```

И хотя при создании array можно передавать в аргументах range, всегда используйте функцию arange, так как она оптимизирована для array. Скоро я покажу, как определить время выполнения различных операций, чтобы сравнить их быстродействие.

Создание диапазонов чисел с плавающей точкой функцией linspace

Для создания равномерно распределенных диапазонов чисел с плавающей точкой можно воспользоваться функцией linspace библиотеки NumPy. Первые два аргумента функции определяют начальное и конечное значение диапазона, при этом конечное значение включается в array. Необязательный ключевой аргумент num задает количество равномерно распределенных генерируемых значений (по умолчанию используется значение 50):

In [28]:

```
1 np.linspace(0.0, 1.0, num=5)
```

Out[28]:

```
array([0.  , 0.25, 0.5 , 0.75, 1.  ])
```

Изменение размерности array

Коллекцию array также можно создать на базе диапазона элементов, а затем воспользоваться методом reshape для преобразования одномерной коллекции array в многомерную. Создадим коллекцию array со значениями от 1 до 20, а затем преобразуем ее к двумерной структуре из четырех строк и пяти столбцов:

In [29]:

```
1 np.arange(1, 21).reshape(4, 5)
```

Out[29]:

```
array([[ 1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10],
       [11, 12, 13, 14, 15],
       [16, 17, 18, 19, 20]])
```

Обратите внимание на *сцепленные вызовы* методов в приведенном фрагменте.
Сначала arange создает коллекцию array со значениями 1–20. После этого вызов reshape для полученной коллекции array создает коллекцию 4 × 5, показанную выше.
Размерность можно изменить для любой коллекции array — при условии что по количеству элементов новая версия не отличается от оригинала. Таким образом, одномерная коллекция array из шести элементов может превратиться в коллекцию 3 × 2 или 2 × 3, и наоборот, но попытка преобразовать коллекцию array из 15 элементов в коллекцию 4 × 4 (16 элементов) приводит к ошибке ValueError

Вывод больших коллекций array

При выводе коллекции array, содержащей 1000 и более элементов, NumPy исключает из вывода строки и/или столбцы в середине.
Следующие фрагменты генерируют 100 000 элементов. В первом примере показаны все четыре строки, каждая из 25 000 столбцов (знак многоточия … представляет отсутствующие данные).

In [30]:

```
1 np.arange(1, 100001).reshape(4, 25000)
```

Out[30]:

```
array([[     1,      2,      3, ...,  24998,  24999,  25000],
       [ 25001,  25002,  25003, ...,  49998,  49999,  50000],
       [ 50001,  50002,  50003, ...,  74998,  74999,  75000],
       [ 75001,  75002,  75003, ...,  99998,  99999, 100000]])
```

Во втором примере приведены три первых и три последних строки (из 100 строк), причем каждая из шести насчитывает 1000 столбцов:

In [31]:

```
1 np.arange(1, 100001).reshape(100, 1000)
```

Out[31]:

```
array([[     1,      2,      3, ...,    998,    999,   1000],
       [  1001,   1002,   1003, ...,   1998,   1999,   2000],
       [  2001,   2002,   2003, ...,   2998,   2999,   3000],
       ...,
       [ 97001,  97002,  97003, ...,  97998,  97999,  98000],
       [ 98001,  98002,  98003, ...,  98998,  98999,  99000],
       [ 99001,  99002,  99003, ...,  99998,  99999, 100000]])
```

Сравнение быстродействия списков и array

Многие операции с коллекциями array выполняются намного быстрее, чем соответствующие операции со списками. Для демонстрации мы воспользуемся магической командой IPython %timeit, измеряющей среднюю продолжительность операций. Учтите, что время в вашей системе может отличаться от моих результатов.

Хронометраж создания списка с результатами 6 000 000 бросков кубика

Воспользуемся функцией randrange модуля random с генератором списка, чтобы создать список с результатами 6 000 000 бросков и провести хронометраж операции командой %timeit. Обратите внимание на использование символа продолжения строки (\) для разбиения команды из фрагмента ниже на две строки:

In [32]:

```
import random
%timeit rolls_list = \
    [random.randrange(1, 7) for i in range(0, 6_000_000)]
```

```
8.92 s ± 370 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

По умолчанию %timeit выполняет команду в цикле, который прогоняется семь раз. Если количество итераций не задано, то %timeit выбирает подходящее значение. В нашем тестировании операции, занимавшие более 500 милли секунд, выполнялись всего один раз, тогда как операции, занимавшие менее 500 миллисекунд, повторялись 10 раз и более.

После выполнения команды %timeit выводит среднее время ее выполнения, а также стандартное отклонение по всем выполнениям. В среднем %timeit указывает, что создание списка заняло 8.92 секунды со стандартным отклонением 370 миллисекунд (мс). В сумме семикратное выполнение фрагмента заняло около 1 минуты.

Хронометраж создания коллекции array с результатами 6 000 000 бросков

Теперь воспользуемся функцией *randint* из модуля numpy.random для создания коллекции array с 6 000 000 бросков:

In [33]:

```
import numpy as np
%timeit rolls_array = np.random.randint(1, 7, 6_000_000)
```

```
113 ms ± 4.65 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

В среднем %timeit показывает, что создание array заняло всего 113 миллисекунд со стандартным отклонением 4.65 микросекунд (мкс). В сумме выполнение предыдущего фрагмента заняло на нашем компьютере менее половины секунды — около 1/100 от времени выполнения фрагмента [32]. Таким образом, с array операция действительно выполняется на два порядка быстрее!

Эти данные наглядно показывают, почему для операций, связанных с интенсивными вычислениями, обычно отдается предпочтение коллекциям array перед списками. В своих практических примерах data science мы войдем в мир больших данных и искусственного интеллекта с его высокими требованиями к производительности. Вы узнаете, как сочетание современного оборудования, программных продуктов, коммуникаций и структур алгоритмов позволяет справиться с колоссальными требованиями современных приложений к вычислительным мощностям.

Операторы array

NumPy предоставляет обширный набор операторов, позволяющих писать простые выражения для выполнения операций с целыми коллекциями array. В этом разделе продемонстрированы математические операции между коллекциями array и числовыми значениями, а также между коллекциями array одинакового размера.

Арифметические операции с array и числовыми значениями

Начнем с поэлементных арифметических операций с array и числовыми значениями, использующих арифметические операторы и расширенное присваивание. Поэлементные операции применяются к каждому элементу, так что фрагмент [34] умножает каждый элемент на 2, а фрагмент [35] возводит каждый элемент в куб. В каждом случае возвращается новая коллекция array, содержащая результат:

In [34]:

```
import numpy as np
numbers = np.arange(1, 6)
numbers * 2
```

Out[34]:

```
array([ 2,  4,  6,  8, 10])
```

In [35]:

```
numbers ** 3
```

Out[35]:

```
array([  1,   8,  27,  64, 125], dtype=int32)
```

In [36]:

```
numbers # numbers не изменяется арифметическими операторами
```

Out[36]:

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

Фрагмент [36] показывает, что арифметические операторы не изменили numbers.
Операторы + и * коммутативны, поэтому фрагмент [34] также можно было записать в виде 2 * numbers.

Расширенные присваивания изменяют каждый элемент левого операнда.

In [38]:

```
numbers += 10
numbers
```

Out[38]:

```
array([21, 22, 23, 24, 25])
```

Арифметические операции между коллекциями array

С коллекциями array, имеющими одинаковые размеры, можно выполнять арифметические операции и расширенные присваивания. Перемножим одномерные коллекции array numbers и numbers2 (созданные ниже), каждая из которых содержит пять элементов:

In [39]:

```
numbers2 = np.linspace(1.1, 5.5, 5)
numbers2
```

Out[39]:

```
array([1.1, 2.2, 3.3, 4.4, 5.5])
```

In [40]:

```
numbers * numbers2
```

Out[40]:

```
array([ 23.1,  48.4,  75.9, 105.6, 137.5])
```

Результат представляет собой новый объект array, полученный поэлементным перемножением обоих операндов — 11 * 1.1, 12 * 2.2, 13 * 3.3 и т. д. Результаты выполнения арифметических операций между коллекциями array с целыми числами и числами с плавающей точкой — коллекция чисел с плавающей точкой.

Сравнение коллекций array

Коллекции array можно сравнивать как с отдельными значениями, так и с другими коллекциями array. Сравнения выполняются поэлементно. В результате таких сравнений создаются коллекции array с логическими значениями, каждое из которых обозначает результат сравнения соответствующих элементов:

In [41]:

```
numbers
```

Out[41]:

```
array([21, 22, 23, 24, 25])
```

In [42]:

```
numbers >= 13
```

Out[42]:

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

In [43]:

```
numbers2
```

Out[43]:

```
array([1.1, 2.2, 3.3, 4.4, 5.5])
```

In [44]:

```
numbers2 < numbers
```

Out[44]:

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

Вычислительные методы NumPy

Коллекция array содержит различные методы для выполнения вычислений. По умолчанию эти методы игнорируют размеры array и используют в вычислениях все элементы. Например, при вычислении среднего значения для array суммируются все элементы независимо от размера, после чего сумма делится на общее количество элементов. Эти вычисления можно выполнять и с отдельными измерениями. Например, в двумерной коллекции array можно вычислить среднее значение по каждой строке и по каждому столбцу.

Возьмем коллекцию array, представляющую оценки четырех студентов на трех экзаменах:

In [45]:

```
import numpy as np
grades = np.array([[87, 96, 70], [100, 87, 90],
                   [94, 77, 90], [100, 81, 82]])
grades
```

Out[45]:

```
array([[ 87,  96,  70],
       [100,  87,  90],
       [ 94,  77,  90],
       [100,  81,  82]])
```

При помощи различных методов можно вычислить сумму (sum), наименьшее (min) и наибольшее (max) значение, математическое ожидание (mean), стандартное отклонение (std) и дисперсию (var) — каждая подобная операция в контексте программирования в функциональном стиле является *сверткой*:

In [46]:

```
grades.sum()
```

Out[46]:

```
1054
```

In [47]:

```
grades.min()
```

Out[47]:

```
70
```

In [48]:

```
grades.max()
```

Out[48]:

```
100
```

In [49]:

```
grades.mean()
```

Out[49]:

```
87.83333333333333
```

In [50]:

```
grades.std()
```

Out[50]:

```
8.792357792739987
```

In [51]:

```
grades.var()
```

Out[51]:

```
77.30555555555556
```

Вычисления по строкам или по столбцам

Многие вычислительные методы также могут применяться к конкретным размерностям array (они называются осями array). Такие методы получают ключевой аргумент axis, определяющий размерность, используемую в вычислениях; это позволяет быстро проводить вычисления по строкам или по столбцам в двумерной коллекции array.

Допустим, вы хотите вычислить среднюю оценку по каждому *экзамену*, представленному одним из столбцов. Аргумент axis=0 выполняет вычисления по всем значениям строк внутри каждого столбца:

In [52]:

```
grades.mean(axis=0)
```

Out[52]:

```
array([95.25, 85.25, 83.  ])
```

Значение 95.25 представляет собой среднее значение для оценок первого столбца (87, 100, 94 и 100), 85.25 — среднее значение для оценок второго столбца (96, 87, 77 и 81), а 83 — среднее значение для оценок третьего столбца (70, 90, 90 и 82). Как и прежде, NumPy не отображает завершающие нули в дробной части: '83.'. Также стоит заметить, что все значения элементов выводятся в полях постоянной ширины, именно поэтому за '83.' следуют два пробела.

С аргументом axis=1 вычисления будут выполняться со всеми значениями столбца внутри каждой отдельной строки. Так, для вычисления средней оценки каждого студента по всем экзаменам можно использовать следующую команду:

In [53]:

```
grades.mean(axis=1)
```

Out[53]:

```
array([84.33333333, 92.33333333, 87.        , 87.66666667])
```

Этот фрагмент вычисляет четыре средних значения — по одному для каждой строки. Таким образом, 84.33333333 представляет собой среднее значение для оценок строки 0 (87, 96 и 70), и так далее для остальных строк.

Коллекции array библиотеки NumPy поддерживают много других вычислительных методов. За полным списком обращайтесь по адресу: https://docs.scipy.org/doc/numpy/reference/arrays.ndarray.html (https://docs.scipy.org/doc/numpy/reference/arrays.ndarray.html)

Универсальные функции

NumPy предоставляет десятки автономных *универсальных функций* для выполнения различных поэлементных операций. Каждая функция выполняет свою задачу с использованием одного или двух аргументов, которыми могут быть коллекции array или аналогичные структуры (например, списки). Некоторые из этих функций вызываются при применении к array таких операторов, как + и *. Каждая функция возвращает новую коллекцию array с результатами.

Создадим коллекцию array и вычислим квадратный корень всех ее значений при помощи универсальной функции sqrt:

In [54]:

```
import numpy as np
numbers = np.array([1, 4, 9, 16, 25, 36])
np.sqrt(numbers)
```

Out[54]:

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

А теперь просуммируем две коллекции array одинакового размера при помощи универсальной функции add:

In [55]:

```
numbers2 = np.arange(1, 7) * 10
np.add(numbers, numbers2) # всё равно что numbers + numbers2
```

Out[55]:

```
array([11, 24, 39, 56, 75, 96])
```

Индексирование и сегментация

К одномерным коллекциям array могут применяться операции индексирования и сегментации; при этом используются синтаксис и приемы, продемонстрированные в первом занятии. В этом разделе мы сосредоточимся на средствах индексирования и сегментации, специфичных для array.

Индексирование с двумерными коллекциями array

Чтобы выбрать элемент двумерной коллекции array, укажите кортеж с индексами строки и столбца элемента в квадратных скобках.

In [56]:

```
import numpy as np
grades = np.array([[87, 96, 70], [100, 87, 90],
                   [94, 77, 90], [100, 81, 82]])
grades[0, 1] # строка 0, столбец 1
```

Out[56]:

```
96
```

Выбор подмножества строк двумерной коллекции array

Чтобы выбрать одну строку, укажите только один индекс в квадратных скобках:

In [57]:

```
grades[1]
```

Out[57]:

```
array([100,  87,  90])
```

Для выбора нескольких смежных строк используется синтаксис сегмента:

In [58]:

```
grades[0:2]
```

Out[58]:

```
array([[ 87,  96,  70],
       [100,  87,  90]])
```

Для выбора нескольких несмежных строк используется список индексов строк:

In [59]:

```
grades[[1, 3]]
```

Out[59]:

```
array([[100,  87,  90],
       [100,  81,  82]])
```

Выбор подмножества столбцов двумерной коллекции array

Чтобы выбрать подмножество столбцов, укажите кортеж, который определяет выбираемые строки и столбцы. Каждым элементом может быть конкретный индекс, сегмент или список. Выберем элементы только первого столбца:

In [60]:

```
grades[:, 0]
```

Out[60]:

```
array([ 87, 100,  94, 100])
```

Ноль после запятой указывает, что выбирается только столбец 0. Двоеточие (:) перед запятой указывает, какие строки в этом столбце должны выбираться. В данном случае : является сегментом, представляющим все строки. Здесь также может использоваться номер конкретной строки, сегмент, представляющий подмножество строк, или список индексов конкретных строк.

Для выбора нескольких смежных столбцов используется синтаксис сегмента:

In [61]:

```
grades[:, 1:3]
```

Out[61]:

```
array([[96, 70],
       [87, 90],
       [77, 90],
       [81, 82]])
```

Для выбора конкретных столбцов используется *список* индексов этих столбцов:

In [62]:

```
grades[:, [0, 2]]
```

Out[62]:

```
array([[ 87,  70],
       [100,  90],
       [ 94,  90],
       [100,  82]])
```

Изменение размеров и транспонирование

Мы использовали метод reshape для получения двумерных коллекций array по одномерным диапазонам. NumPy предоставляет различные способы изменения размера массивов.

Методы reshape и resize

Методы reshape и resize коллекций array позволяют изменить размеры коллекции. Метод reshape возвращает представление (поверхностную копию) исходной коллекции array с новыми размерами. Исходная коллекция array при этом не изменяется:

In [63]:

```
import numpy as np
grades = np.array([[87, 96, 70], [100, 87, 90]])
grades
```

Out[63]:

```
array([[ 87,  96,  70],
       [100,  87,  90]])
```

In [64]:

```
grades.reshape(1, 6)
```

Out[64]:

```
array([[ 87,  96,  70, 100,  87,  90]])
```

In [65]:

```
grades # массив не изменился
```

Out[65]:

```
array([[ 87,  96,  70],
       [100,  87,  90]])
```

Метод resize изменяет размер исходной коллекции array:

In [66]:

```
grades.resize(1, 6)
```

In [67]:

```
grades # массив изменился: он стал одномерным
```

Out[67]:

```
array([[ 87,  96,  70, 100,  87,  90]])
```

Горизонтальное и вертикальное добавление

Коллекции array можно объединять, добавляя новые столбцы или новые строки, — эти два механизма называются горизонтальным или вертикальным дополнением. Создадим еще одну коллекцию array с размерами 2 × 3:

In [70]:

```
grades = [[100, 96, 70], [100, 87, 90]]
grades2 = np.array([[94, 77, 90], [100, 81, 82]])
```

Допустим, содержимое grades2 представляет результаты еще трех экзаменов для двух студентов из коллекции grades. Мы можем объединить grades и grades2 функцией hstack из библиотеки NumPy; для этого функции передается кортеж с объединяемыми коллекциями. Дополнительные круглые скобки необходимы из-за того, что функция hstack ожидает получить один аргумент:

In [71]:

```
np.hstack((grades, grades2))
```

Out[71]:

```
array([[100,  96,  70,  94,  77,  90],
       [100,  87,  90, 100,  81,  82]])
```

Теперь предположим, что grades2 представляет оценки двух других студентов на трех экзаменах. В этом случае grades и grades2 можно объединить функцией vstack библиотеки NumPy:

In [72]:

```
np.vstack((grades, grades2))
```

Out[72]:

```
array([[100,  96,  70],
       [100,  87,  90],
       [ 94,  77,  90],
       [100,  81,  82]])
```

Введение в data science: коллекции Series и DataFrame библиотеки pandas

Массив NumPy оптимизирован для однородных числовых данных, при обращении к которым используются целочисленные индексы. Область data science создает уникальные требования, для которых необходимы более специализированные структуры данных. Приложения больших данных должны поддерживать смешанные типы данных, нестандартное индексирование, отсутствующие данные, данные с нарушенной структурой и данные, которые

должны быть приведены к форме, более подходящей для баз данных и пакетов анализа данных, с которыми вы работаете.

Pandas — самая популярная библиотека для работы с такими данными. Она предоставляет две ключевые коллекции, которые мы будем использовать во всём курсе и в особенности на экзамене — Series для одномерных коллекций и DataFrame для двумерных коллекций. Коллекция MultiIndex библиотеки pandas может использоваться для работы с многомерными данными в контексте Series и DataFrame.

Уэс Маккинни (Wes McKinney) создал pandas в 2008 году, пока он работал в отрасли. Название pandas происходит от термина «panel data», то есть данные измерений по времени (например, биржевые котировки или исторические температурные данные). Маккинни понадобилась библиотека, в которой одни и те же структуры данных могли бы обрабатывать как временные, так и не временные данные, с поддержкой выравнивания данных, отсутствующих данных, стандартных операций в стиле баз данных и т. д.

Между NumPy иPandas существует тесная связь. Коллекции Series и DataFrame используют array в своей внутренней реализации. Series и DataFrame являются допустимыми аргументами для многих операций NumPy. С другой стороны, коллекции array являются допустимыми аргументами для многих операций Series и DataFrame.

Коллекция Series

Series представляет собой расширенную одномерную версию array. Если array использует только целочисленные индексы, начинающиеся с нуля, то коллекция Series поддерживает нестандартное индексирование, включая даже использование нецелочисленных индексов (например, строк). Коллекция Series также предоставляет дополнительные возможности, которые делают ее более удобной для многих задач, ориентированных на область data science. Например, коллекция Series может содержать отсутствующие данные, которые по умолчанию игнорируются многими операциями Series.

Создание Series с индексами по умолчанию

По умолчанию Series использует целочисленные индексы с последовательной нумерацией от 0. Следующий фрагмент создает коллекцию Series на базе списка целых чисел:

In [73]:

```
1 import pandas as pd
2 grades = pd.Series([87, 100, 94])
```

Инициализатором также может быть кортеж, словарь, array, другая коллекция Series или одиночное значение (последний случай продемонстрирован ниже).

Вывод коллекции Series

Pandas выводит Series в формате из двух столбцов: индексы выравниваются по левому краю в левом столбце, а значения — по правому краю в правом столбце.
После элементов Series pandas выводит тип данных (dtype) элементов используемой коллекции array:

In [74]:

```
1 grades
```

Out[74]:

```
0     87
1    100
2     94
dtype: int64
```

Обратите внимание, насколько просто выводится Series в этом формате по сравнению с соответствующим кодом вывода списка в двухстолбцовом формате.

Создание коллекции Series с одинаковыми значениями

Вы можете создать коллекцию Series, все элементы которой имеют одинаковые значения:

In [75]:

```
1 pd.Series(98.6, range(3))
```

Out[75]:

```
0    98.6
1    98.6
2    98.6
dtype: float64
```

Второй аргумент представляет собой одномерный итерируемый объект (такой как список, массив или диапазон) с индексами Series. Количество индексов определяет количество элементов.

Обращение к элементам Series

Чтобы обратиться к элементу Series, укажите его индекс в квадратных скобках:

In [76]:

```
grades[0]
```

Out[76]:

87

Вычисление описательных статистик для Series

Коллекция Series предоставляет многочисленные методы для выполнения часто встречающихся операций, включая получение различных характеристик описательной статистики. В этом разделе продемонстрированы методы count, mean, min, max и std (стандартное отклонение):

In [78]:

```
# количество значений
grades.count()
```

Out[78]:

3

In [79]:

```
# среднее значение, или математическое ожидание
grades.mean()
```

Out[79]:

93.66666666666667

In [80]:

```
grades.min()
```

Out[80]:

87

In [81]:

```
grades.max()
```

Out[81]:

100

In [82]:

```
grades.std()
```

Out[82]:

6.506407098647712

Каждая из этих характеристик является сверткой в стиле функционального программирования. Вызов методов Series создает не только названные характеристики, но и многие другие:

In [83]:

```
grades.describe()
```

Out[83]:

```
count      3.000000
mean      93.666667
std        6.506407
min       87.000000
25%       90.500000
50%       94.000000
75%       97.000000
max      100.000000
dtype: float64
```

Строки 25%, 50% и 75% содержат **квартили**:

- 50% — медиана отсортированных значений;
- 25% — медиана первой половины отсортированных значений;
- 75% — медиана второй половины отсортированных значений.

Если в квартиле существуют два средних элемента, то медианой этого квартиля становится их среднее значение.
Наша коллекция Series состоит из трех значений, так что 25-процентным квартилем становится среднее значение 87 и 94, а 75-процентным квартилем — среднее значение 94 и 100.
Еще одной дисперсионной метрикой (наряду со стандартным отклонением и дисперсией) является **интерквартильный диапазон** — разность между 75-процентным квартилем и 25-процентным квартилем. Конечно, квартили и интерквартильный диапазон приносят больше пользы в больших наборах данных.

Создание коллекции Series с нестандартными индексами

Для назначения *нестандартных* индексов используется ключевой аргумент index:

In [84]:

```
grades = pd.Series([87, 100, 94], index=['Wally', 'Eva', 'Sam'])
grades
```

Out[84]:

```
Wally     87
Eva      100
Sam       94
dtype: int64
```

В данном случае используются строковые индексы, но можно использовать и другие неизменяемые типы, включая целые числа, не начинающиеся с 0, а также непоследовательные целые числа. Стоит при этом заметить, как удобно и компактно pandas форматирует коллекции Series для вывода.

Словари как инициализаторы

Если коллекция Series инициализируется словарем, то ключи становятся индексами Series, а значения — значениями элементов Series:

In [85]:

```
grades = pd.Series({'Wally': 87, 'Eva': 100, 'Sam': 94})
```

In [86]:

```
grades
```

Out[86]:

```
Wally     87
Eva      100
Sam       94
dtype: int64
```

Обращение к элементам Series с использованием нестандартных индексов

Чтобы обратиться к отдельному элементу в коллекции Series с нестандартными индексами, укажите значение нестандартного индекса в квадратных скобках:

In [87]:

```
grades['Eva']
```

Out[87]:

```
100
```

Если нестандартными индексами являются строки, которые могут представлять допустимые идентификаторы Python, то pandas автоматически добавляет их в Series как атрибуты, к которым можно обращаться через точку (.):

In [88]:

```
grades.Wally
```

Out[88]:

```
87
```

Коллекция Series также содержит встроенные атрибуты. Например, атрибут dtype возвращает тип элемента базовой коллекции array:

In [89]:

```
grades.dtype
```

Out[89]:

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

Атрибут values возвращает базовую коллекцию array:

In [90]:

```
grades.values
```

Out[90]:

```
array([ 87, 100,  94], dtype=int64)
```

Создание коллекции Series со строковыми элементами

Если коллекция Series содержит строки, то можно воспользоваться ее атри бутом str для вызова методов строк элементов. Сначала создадим коллекцию Series со строками:

In [91]:

```
hardware = pd.Series(['Hammer', 'Saw', 'Wrench'])
hardware
```

Out[91]:

```
0    Hammer
1       Saw
2    Wrench
dtype: object
```

Обратите внимание: pandas также выравнивает по правому краю строковые значения элементов, а типом данных (dtype) для строк является object.

Теперь вызовем метод contains для каждого элемента, чтобы определить, содержит ли значение каждого элемента букву 'a' нижнего регистра:

In [92]:

```
hardware.str.contains('a')
```

Out[92]:

```
0     True
1     True
2    False
dtype: bool
```

DataFrame

DataFrame — расширенная двумерная версия array. Как и Series, DataFrame может иметь нестандартные индексы строк и столбцов и предоставляет дополнительные операции и средства, с которыми удобнее использовать коллекцию DataFrame для многих задач, ориентированных на специфику data science. DataFrame также поддерживает пропущенные данные. Каждый столбец DataFrame является коллекцией Series. Коллекция Series, представляющая каждый столбец, может содержать элементы разных типов, как будет вскоре показано при рассмотрении загрузки наборов данных в DataFrame.

Создание DataFrame на базе словаря

Создадим коллекцию DataFrame на базе словаря, представляющего оценки студентов на трех экзаменах:

In [93]:

```
import pandas as pd
grades_dict = {'Wally': [87, 96, 70], 'Eva': [100, 87, 90],
               'Sam': [94, 77, 90], 'Katie': [100, 81, 82],
               'Bob': [83, 65, 85]}
grades = pd.DataFrame(grades_dict)
grades
```

Out[93]:

	Wally	Eva	Sam	Katie	Bob
0	87	100	94	100	83
1	96	87	77	81	65
2	70	90	90	82	85

Pandas выводит содержимое DataFrame в табличном формате с индексами, выровненными по левому краю в столбце индексов, а значения остальных столбцов выравниваются по правому краю. Ключи словаря становятся именами столбцов, а значения, связанные с каждым ключом, становятся значениями элементов соответствующего столбца. Вскоре мы покажем, как поменять местами строки и столбцы. По умолчанию индексы строк задаются автоматически сгенерированными целыми числами, начиная с 0.

Настройка индексов DataFrame с использованием атрибута index

При создании DataFrame можно задать нестандартные индексы при помощи ключевого аргумента index:

In [94]:

```
pd.DataFrame(grades_dict, index=['Test1', 'Test2', 'Test3'])
```

Out[94]:

	Wally	Eva	Sam	Katie	Bob
Test1	87	100	94	100	83
Test2	96	87	77	81	65
Test3	70	90	90	82	85

Воспользуемся атрибутом index для преобразования индексов DataFrame из последовательных целых чисел в текстовые метки:

In [96]:

```
grades.index = ['Test1', 'Test2', 'Test3']
grades
```

Out[96]:

	Wally	Eva	Sam	Katie	Bob
Test1	87	100	94	100	83
Test2	96	87	77	81	65
Test3	70	90	90	82	85

При задании индексов необходимо передать одномерную коллекцию, количество элементов в которой совпадает с количеством строк данных в DataFrame; в противном случае происходит ошибка ValueError. Series также предоставляет атрибут index для изменения существующих индексов Series.

Обращение к столбцам DataFrame

Одна из сильных сторон pandas — возможность быстро и удобно просматривать данные многими разными способами, включая отдельные части данных. Начнем с получения оценок студента Eva по имени и вывода столбца в формате Series:

Если строки с именами столбцов DataFrame являются допустимыми идентификаторами Python, они могут использоваться как атрибуты. Получим оценки студента Sam при помощи атрибута Sam:

In [97]:

```
grades['Sam']
```

Out[97]:

```
Test1    94
Test2    77
Test3    90
Name: Sam, dtype: int64
```

Выбор строк с использованием атрибутов loc и iloc

Хотя коллекции DataFrame поддерживают возможность индексирования в синтаксисе [], документация pandas рекомендует использовать атрибуты loc, iloc, at и iat, которые оптимизированы для обращения к DataFrame и предоставляют дополнительные средства, выходящие за рамки того, что можно сделать исключительно с []. В документации указано, что при индексировании [] часто создается копия данных, что может привести к логической ошибке, если вы попытаетесь присвоить новые значения DataFrame результату операции [].

Чтобы обратиться к строке по ее текстовой метке, воспользуйтесь атрибутом loc коллекции DataFrame. Ниже выводятся все оценки из строки 'Test1':

In [98]:

```
grades.loc['Test1']
```

Out[98]:

```
Wally     87
Eva      100
Sam       94
Katie    100
Bob       83
Name: Test1, dtype: int64
```

К строкам также можно обращаться по целочисленным индексам, начинающимся с 0, при помощи атрибута iloc (буква i в iloc означает, что атрибут используется с целочисленными индексами). Следующий пример выводит все оценки из второй строки:

In [99]:

```
grades.iloc[1]
```

Out[99]:

```
Wally    96
Eva      87
Sam      77
Katie    81
Bob      65
Name: Test2, dtype: int64
```

Выбор строк с использованием атрибутов loc и iloc

Индексом может быть и *сегмент*. При использовании с атрибутом loc сегментов, содержащих метки, заданный диапазон *включает* верхний индекс ('Test3'):

In [100]:

```
grades.loc['Test1':'Test3']
```

Out[100]:

	Wally	Eva	Sam	Katie	Bob
Test1	87	100	94	100	83
Test2	96	87	77	81	65
Test3	70	90	90	82	85

При использовании с атрибутом iloc сегментов, содержащих целочисленные индексы, заданный диапазон *не включает* верхний индекс (2):

In [101]:

```
grades.iloc[0:2]
```

Out[101]:

	Wally	Eva	Sam	Katie	Bob
Test1	87	100	94	100	83
Test2	96	87	77	81	65

Выбор подмножеств строк и столбцов

До настоящего момента мы выбирали только целые строки. Вы также можете ограничиться меньшими подмножествами DataFrame и выбирать строки и столбцы с использованием двух сегментов, двух списков или сочетания сегментов и списков.

Предположим, вы хотите просмотреть только оценки студентов Eva и Katie для экзаменов Test1 и Test2. Для этого можно воспользоваться атрибутом loc с сегментом для двух последовательных строк и списком для двух непоследовательных столбцов:

In [102]:

```
grades.loc['Test1':'Test2', ['Eva', 'Katie']]
```

Out[102]:

	Eva	Katie
Test1	100	100
Test2	87	81

Сегмент 'Test1':'Test2' выбирает строки для Test1 и Test2. Список ['Eva', 'Katie'] выбирает только соответствующие оценки из этих двух столбцов.

Воспользуемся iloc со списком и сегментом, чтобы выбрать первый и третий экзамены и первые три столбца для этих экзаменов:

In [103]:

```
grades.iloc[[0, 2], 0:3]
```

Out[103]:

	Wally	Eva	Sam
Test1	87	100	94
Test3	70	90	90

Логическое индексирование

Одним из самых мощных средств выбора в pandas является логическое индексирование. Для примера выберем все оценки, большие или равные 90:

In [104]:

```
grades[grades >= 90]
```

Out[104]:

	Wally	Eva	Sam	Katie	Bob
Test1	NaN	100.0	94.0	100.0	NaN
Test2	96.0	NaN	NaN	NaN	NaN
Test3	NaN	90.0	90.0	NaN	NaN

Pandas проверяет каждую оценку и, если она больше или равна 90, включает ее в новую коллекцию DataFrame. Оценки, для которых условие равно False, представлены в новой коллекции DataFrame значением NaN («Not A Number», то есть «не число».) NaN используется pandas для обозначения отсутствующих значений.

Выберем все оценки в диапазоне 80–89:

In [105]:

```
grades[(grades >= 80) & (grades < 90)]
```

Out[105]:

	Wally	Eva	Sam	Katie	Bob
Test1	87.0	NaN	NaN	NaN	83.0
Test2	NaN	87.0	NaN	81.0	NaN
Test3	NaN	NaN	NaN	82.0	85.0

Логические индексы Pandas объединяют несколько условий оператором Python & (поразрядная операция И) — не путайте с логическим оператором and. Для условий or используется оператор | (поразрядная операция ИЛИ). NumPy также поддерживает логическое индексирование для массивов, но всегда возвращает одномерный массив, содержащий только те значения, для которых выполняется условие.

Описательная статистика

Коллекции Series и DataFrame содержат метод describe, который вычисляет основные характеристики описательной статистики для данных и возвращает их в форме DataFrame. В DataFrame статистики вычисляются по столбцам.

In [106]:

```
grades.describe()
```

Out[106]:

	Wally	Eva	Sam	Katie	Bob
count	3.000000	3.000000	3.000000	3.000000	3.000000
mean	84.333333	92.333333	87.000000	87.666667	77.666667
std	13.203535	6.806859	8.888194	10.692677	11.015141
min	70.000000	87.000000	77.000000	81.000000	65.000000
25%	78.500000	88.500000	83.500000	81.500000	74.000000
50%	87.000000	90.000000	90.000000	82.000000	83.000000
75%	91.500000	95.000000	92.000000	91.000000	84.000000
max	96.000000	100.000000	94.000000	100.000000	85.000000

Как видно из вывода, метод describe позволяет быстро получить сводную картину данных. Он неплохо демонстрирует мощь программирования, ориентированного на массивы, на примере понятного, компактного вызова в функциональном стиле. Pandas берет на себя все подробности вычисления этих статистик для каждого столбца. Возможно, вам захочется получить аналогичную статистику для экзаменов, чтобы уяснить, какие результаты показали студенты на экзаменах Test1, Test2 и Test3, — вскоре мы покажем, как это делается.

По умолчанию pandas вычисляет характеристики описательной статистики в формате чисел с плавающей точкой и выводит их с шестью знаками точности. Для управления точностью и другими настройками по умолчанию может использоваться функция pandas set_option:

In [108]:

```
pd.set_option('precision', 2)
grades.describe()
```

Out[108]:

	Wally	Eva	Sam	Katie	Bob
count	3.00	3.00	3.00	3.00	3.00
mean	84.33	92.33	87.00	87.67	77.67
std	13.20	6.81	8.89	10.69	11.02
min	70.00	87.00	77.00	81.00	65.00
25%	78.50	88.50	83.50	81.50	74.00
50%	87.00	90.00	90.00	82.00	83.00
75%	91.50	95.00	92.00	91.00	84.00
max	96.00	100.00	94.00	100.00	85.00

Вероятно, для оценок самой важной характеристикой является математическое ожидание. Чтобы вычислить его для каждого студента, достаточно вызвать mean для коллекции DataFrame:

In [109]:

```
grades.mean()
```

Out[109]:

```
Wally    84.33
Eva      92.33
Sam      87.00
Katie    87.67
Bob      77.67
dtype: float64
```

Сортировка строк по индексам

Данные часто сортируются для удобства чтения. DataFrame можно сортировать по строкам и по столбцам как по индексам, так и по значениям. Отсортируем строки по убыванию индексов при помощи функции sort_index, которой передается ключевой аргумент **ascending=False** (по умолчанию сортировка выполняется по возрастанию). Функция возвращает новую коллекцию DataFrame с отсортированными данными:

In [110]:

```
grades.sort_index(ascending=False)
```

Out[110]:

	Wally	Eva	Sam	Katie	Bob
Test3	70	90	90	82	85
Test2	96	87	77	81	65
Test1	87	100	94	100	83

Сортировка по индексам столбцов

Теперь отсортируем столбцы по возрастанию (слева направо) по именам столбцов. Ключевой аргумент **axis=1** означает, что сортировка должна выполняться по индексам столбцов, а не по индексам строк — аргумент **axis=0** (по умолчанию) сортирует индексы строк:

In [111]:

```
grades.sort_index(axis=1)
```

Out[111]:

	Bob	Eva	Katie	Sam	Wally
Test1	83	100	100	94	87
Test2	65	87	81	77	96
Test3	85	90	82	90	70

Сортировка по значениям столбцов

Предположим, вы хотите просмотреть оценки Test1 по убыванию, чтобы имена студентов следовали от наибольшей оценки к наименьшей. Вызов метода sort_values выглядит так:

In [112]:

```
grades.sort_values(by='Test1', axis=1, ascending=False)
```

Out[112]:

	Eva	Katie	Sam	Wally	Bob
Test1	100	100	94	87	83
Test2	87	81	77	96	65
Test3	90	82	90	70	85

Ключевые аргументы by и axis совместно работают для определения того, какие значения будут сортироваться. В данном случае сортировка осуществляется по значениям столбцов (axis=1) для Test1.

Возможно, имена студентов и оценки станет удобнее читать при выводе в столбец, так что отсортировать можно и транспонированную коллекцию DataFrame.
В следующем примере указывать ключевой аргумент axis не нужно, потому что sort_values по умолчанию сортирует данные в заданном столбце:

In [113]:

```
grades.T.sort_values(by='Test1', ascending=False)
```

Out[113]:

	Test1	Test2	Test3
Eva	100	87	90
Katie	100	81	82
Sam	94	77	90
Wally	87	96	70
Bob	83	65	85

Наконец, поскольку сортируются только оценки экзамена Test1, не исключено, что другие экзамены выводить вообще не нужно. Объединим операцию выбора с сортировкой:

In [114]:

```
grades.loc['Test1'].sort_values(ascending=False)
```

Out[114]:

```
Eva      100
Katie    100
Sam       94
Wally     87
Bob       83
Name: Test1, dtype: int64
```

Задачи на закрепление

Упражнения на методы numpy

При работе можно (и даже нужно пользоваться документацией библиотеки numpy).

1. Создайте нулевой вектор размера 10;
2. Как найти размер памяти занимаемый вектором?
3. Как создать вектор размера 10, пятый элемент которого = 1?
4. Как инвертировать вектор (первый элемент становится последним)?
5. Создайте матрицу 3x3 со значениями от 0 до 8.
6. Создайте массив 10x10 со случайными значениями и найдите минимальное и максимальное значения.
7. Создайте двумерный-массив с 1(единицами) на границе и 0(нулями) внутри.
8. Как добавить границу (заполненную нулями) вокруг существующего массива (например: массив 5 на 5 из единиц окружить нулями, чтобы он превратился в массив 7 на 7?
9. Умножьте матрицу 5x3 на матрицу 3x2 (матричное произведение).
10. Вычтите среднее значение каждой строки матрицы. Пояснение к заданию: необходимо вычислить среднее значение каждой строки матрицы и вычесть его из элементов соответствующей строки.

In [ ]:

```

```

Упражнения на методы pandas

При выполнении создать свой DataFrame.

1. Напишите программу для подсчета количества столбцов в DataFrame.

2. Напишите программу для преобразования заданного списка списков (как пример: [[2, 4], [1, 3]]) в Dataframe.
3. Напишите программу, чтобы проверить, присутствует ли данный столбец в DataFrame или нет.
4. Напишите программу для объединения двух Series в DataFrame.
5. Напишите программу для удаления списка строк из указанного DataFrame.
6. Напишите программу для подсчета значений NaN в одном или нескольких столбцах в DataFrame.
7. Напишите программу для добавления одной строки в существующий DataFrame.

In [ ]:

```
1
```