**Numpy and Pandas**

import pandas as pd

import numpy as np

|  |  |
| --- | --- |
| **Numpy** |  |
| a = np.array([ [1,2,3],[4,5,6] ]) | Создание списка NumPy |
| a.ndim >>> 2 | **количество измерений** массива (линейный, матрица, 3-мерный и т.д.) |
| a.size >>> 6 | **количество элементов** в массиве |
| a.shape >>> (2,3) | **форма** списка (2 строки по 3 элемента) |
| a.dtype >>> ‘int32’ | **Формат** **данных** в списке |
| a.sum() >>> 21  a.prod() >>> 720  a.sum(axis=0) >>> array([5,7,9])  a.sum(axis=1) >>> array([6,15])  a.min()  a.max()  np.sqrt(a) | сумма всех элементов  произведение всех элементов  сумма элементов в каждом столбце  сумма значений в каждой строчке  минимальное значение  максимальное значение  квадратный корень из каждого значения |
| A @ B | Произведение матриц (не поэлементное) |
| a.arange(15) | создать одномерный массив из range(15) - от 0 до 14, по умолчанию шаг =1 |
| a.arange(15).reshape(3,5) | **переформировать** массив из 15 элементов в матрицу с 3 строками по 5 элементов |
| a.reshape(a.shape[::-1]) | **Транспонирование массива** |
| a.arange(0, 4, 0.5)  a.arange(2.3, 5.7, 0.3) | создать одномерный массив - от 0 до 4 с шагом 0.5, по умолчанию шаг =1,  то же самое для дробных значений |
| a=np.linspace(11, 60, 25) | сделать массив float-значений? конечное число - включительно, **с автоматически рассчитываемым шагом** при заданном количестве значений |
| def f(x,y):  return 10\*x+y  np.fromfunction(f, SHAPE, dtype=int) | создание массива из функции, ИСПОЛЬЗУЮЩЕЙ КООРДИНАТЫ ЭЛЕМЕНТА МАССИВА (x,y,z) |
| a[0,3] >>> 3  a[1, -1] >>> 6 | Выбор значения ячейки по индексам строк, столбцов… |
| Слайс (срез) numpy не создает копию данных (в отличие от списков python), а возвращает так называемый view.  Это значит, что если поменять значение в слайсе (срезе), то значение в оригинальном массиве также изменится | |
| a [3:9:2]  a [::-1]  a [1, ::2] >>> [4,6]  a [..., 1] >>> [2,5] | **Срезы**: с 3 по 9 не включительно с шагом 2,  от начала до конца в обратную сторону (развернутый),  из второго элемента срез от начала до конца с шагом 2,  выбор в каждом подмассиве (через троеточие, называемое **Ellipsis**) элемента под индексом 1 |
|  |  |
| size = 10  SHAPE = (2,5)  z = np.zeros(( size, )).reshape(SHAPE)  z = np.zeros(( size, ), dtype=np.int\_).reshape(SHAPE) | создать массив float нулей количеством с заданное число size с заданной формой SHAPE  или то же самое с заданным типом integer |
| z.shape = 5, **-1** | придать массиву форму с автоматическим значением одного из измерений - нужно указать -1 на месте неизвестного |
| с = np.zeros\_like(a) | сделать массив с формой, схожей на форму другого массива (в данном случае массива а) |
| A = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12], [13, 14, 15, 16]])  idx1 = [[0, 1], [3, 2]]  idx2 = [[0, 2], [1, 1]]  A[idx1, idx2] → array([[ 1, 7], [14, 10]]) | Обращение к элементам массива по индексам, указанным в других массивах. |
| **a.flat**  for elem in a.flat: | генератор по ВСЕМ элементам многомерного массива c перебором всех уровней (можно итерироваться) |
| a.ravel() | перевод значений массива в одномерный массив |
| np.vstack((b,c)) | массив, вертикально (по второму индексу) соединяющий два других массива в единый, сохраняя количество измерений |
| np.hstack((b,c))  либо  np.column\_stack((b,c)) | массив, горизонтально (по первому индексу) соединяющий два других массива в единый, сохраняя количество измерений |
| np.hsplit(a, 2)  np.vsplit(a, 5) | разделить на 2 массива горизонтально  или вертикально |
| A[0, 0] = 100  A[:, 2] = [0, 0, 0] | **Установление значений для ячейки**  **Установление значений 3-го столбца** |
| np.all(a==b) | **Проверка равенства** всех элементов массивов a и b |
| a + b, a - b, a \* b, a / b, a \* 2, a\*\*4 | поэлементное сложение, вычитание, умножение, деление данных в массивах a и b; поэлементное умножение на 6, возведение в 4 степень |
| np.diff(a) | сделать массив из **разниц между соседними значениями** переданного массива |
| b - a \* c | вычитание из значений массива и произведения значений массивов a и c |
| с = a > 5 | сделать **массив-маску из булевых значений True-False,** указывающих, какие элементы больше 5 |
| b= a[a > 5] | создание массива из наложения маски (>5) на массив a, в и ,elen все элементы a, отвечающие условию |
|  |  |
| v = [1,0,1]  vv = np.**tile**(v, (4,1)) | broadcasting:  сделать **массив большой из повторения маленького** (уложить плиточкой в виде маленького массива), указав количество строк и повторений в каждой строке |
| a[:, np.newaxis] | создать новое измерение и по всем измерениям разделить содержимое массива |
| x = np.ones((4, 3))  y = np.random.rand(3)  (x \* y).shape → (4,3) | Произведение массивов разного размера возможно, если расходится только 1 размернотсь (например, 4\*3 и 3\*1) |
| ax, bx, cx = np**.ix\_(**a,b,c)  ax + bx \* cx | ix\_ функция для действий между массивами разного размера (количества элементов)  С преобразованными массивами можно производить поэлементные действия |
|  |  |
|  |  |
|  |  |
| **Pandas** |  |
| **СОЗДАНИЕ DATAFRAME** | |
| series = pd.Series([1,2], index=['a','b'])  df = pd.DataFrame([series]) | Dataframe from Series |
| cols = ['a','b']  list\_of\_series = [pd.Series([1,2],index=cols), pd.Series([3,4],index=cols)]  df = pd.DataFrame(list\_of\_series, columns=cols) | Dataframe from multiple Series |
| df\_new = DataFrame.copy(deep=True) | Скопировать Датафрфм. Получаем абсолютно независимый новый Dataframe. |
| df\_new = DataFrame.copy(deep=False) | Сделать ссылочную копию Датафрейма. Изменения в оригинале тут же отразятся в копии и наоборот. |
| df = pd.DataFrame(list\_of\_dicts) | Создать df из листа словарей. DataFrame from list of dicts |
| df = pd.DataFrame(  np.random.randint(0,100,size=(100, 4)), columns=list('ABCD')) | Создать df с рандомными интеджер значениями в указанном диапазоне. |
| fines\_df = pd.DataFrame.from\_dict(fines) | Создать Датафрейм из словаря или **листа словарей** |
| df = pd.DataFrame({  'country': ['Russia', 'Kazakhstan', 'Ukraine', 'Belarus'],  'population': [144.5, 18.3, 42.4, 9.5],  'square': [17\_125\_200, 2\_724\_900, 603\_628, 207\_595],  }, index=['RU', 'KZ', 'UA', 'BY']) | Создание таблицы-ДатаФрейма с именами столбцов и индексами в виде кодов стран |
| df = pandas.DataFrame({'any': [1,2,3,4,5,6,7], 'string': ['первое; второе', '', 'третье', 'четвертое; пятое; шестое; 7', '', '[]', '']}) |  |
| df.to\_csv('countries\_data.csv') | Выгрузить данные в csv файл с указанием имени файла |
| pd.read\_csv('data.csv')  pd.read\_csv('data.csv', index\_col=0) | Прочитать данные из csv-файла с колонкой индексов или без неё |
| df = pd.read\_csv('data.csv', na\_values=”NA”, decimal=”,”, delimiter=';', skiprows=1, names=[“Регион”,”2017”], index\_col= ”Регион”) | na\_values - указываем, что поставить на место отсутствующих данных.  decimal - указывает, какой знак ставить для разделения целых и дробных частей. Позволяет сразу ещё при чтении файла перевести данные из формата «строка» в число.  delimiter - позволяет указать разделитель, по которому строки в файле надо делить на значения столбцов.  skiprows=1 - пропустит первый ряд, ряд-заголовок,  names - именя колонок  index\_col - сделать одну из колонок индексом. |
| user1 = pd.**read\_csv**('dataset/1.csv', **names**=['TIME', 'X', 'Y', 'Z'], header=None) | Создать Data Frame из csv-файла **с указанием имён колонок**. |
| cols = ['beer\_servings', 'continent']  small\_drinks = pd.read\_csv('http://bit.ly/drinksbycountry', usecols=cols)  small\_drinks.info(memory\_usage='deep') | **Прочитать** из файла не все данные, а **только указанные колонки**. Надо знать их имена.  Уменьшает используемую память. |
| dtypes = {'continent':'category'}  smaller\_drinks = pd.read\_csv('http://bit.ly/drinksbycountry', usecols=cols, dtype=dtypes)  smaller\_drinks.info(memory\_usage='deep') | Прочитать из файла данные, **отформатировав категорийные значения** (ячейки, в которых стоят слова, буквы, символы) из формата «object» в формат данных «category».  **Резко уменьшает используемую память** (иногда в 10 раз) |
| from glob import glob  stock\_files = sorted(glob('data/stocks\*.csv'))  d.**concat**((pd.read\_csv(file) for file in stock\_files), ignore\_index=True) | Объединить две таблицы. Создать **DataFrame из нескольких файлов**, в каждом из которых данные отдельных дней (другие строки для DataFrame)  data/stocks3.csv, data/stocks2.csv, data/stocks1.csv |
| pd.**concat**((pd.read\_csv(file) for file in drink\_files), axis='columns') | То же самое, но из разных файлов подтягиваем разные колонки, а не строки. |
| with open('fines.json') as f:  json\_str = f.read()  data\_json = json.loads(json\_str)['Value']  fines\_list = json.loads(data\_json)['Fines']  for fine in fines\_list:  fines.append({  'Name': fine['ApnDetail'][0]['Value'].replace('\t', ' - '),  'Place': fine['ApnDetail'][3]['Value'],  'Fine sum': fine['FineSum'],  }) | **Расшифровка многоуровнего Json-файла до простого словаря**: считываем из файла, выгружаем значения.  Из значений - штрафы.  Потом заменяем символ каретки на тире. |
| df = pd.read\_clipboard() | Создать DataFrame из БУФЕРА ОБМЕНА. Предварительно можно выделить данные, например в Excel и скопировать в буфер. |
| data = pd.ExcelFile (‘http://video.ittensive.com/py/load.timings.xlsx’) | Импорт данных в DataFrame из Экселя (**Excel**) |
| data = pd.ExcelFile(path)  data = data.parse(sheet\_name=0, usecols='A:C', names=[“Дата”, “Время отрисовки”, “Время загрузки”], converters={“Дата”: pd.to\_datetime, “ Время отрисовки ”: int, “Время загрузки”: int}) | Импорт данных с конкретного листа Эксель-файла, указываем лист и называем колонки по-своему.  Тут же преобразовываем типы данных через словарь типов.  usecols - какие колонки выгружать (можно указать диапазон буквами, можно интеджерами и по-другому)  index\_col - колонка с названиями строк. указывается номер колонки начиная с 0-го.  header : int, list of int, default 0. Row (0-indexed) to use for the column labels of the parsed DataFrame. If a list of integers is passed those row positions will be combined into a MultiIndex. Use None if there is no header.  skiprows :list-like. Rows to skip at the beginning (0-indexed)  nrows :int, default None. Number of rows to parse.  skipfooter : int, default 0. Rows at the end to skip (0-indexed). |
|  |  |
| **Преобразование и работа с DATAFRAME** | |
| Когда мы берём df[‘имя колонки’] мы можем получать значение, которое там сидит, например объект datetime. Но иногда имеем тип данных Series. Чтобы обратиться к методам не Пандас, а внутреннего типа данных, нужно ставить **префикс .dt для datetime, .str для строк**.  df[‘имя колонки’].dt.month, df[‘имя колонки’].str.lower() | |
| list\_new = df.tolist() | Сделать лист из Dataframe. Dataframe to list. List from dataframe. |
|  |  |
| for column in df:  print(df[column]) | Итерация по колонкам. Итерация по столбцам Dataframe.  Iterate over columns |
| df = pd.DataFrame({'A': [1, 2, 3], 'B': ['a', 'b', 'f']})  mask = df.isin([1, 3, 12, 'a'])  df = df.where(mask, other=30)  df  Out[59]:  A B  0 1 a  1 30 30  2 3 30 | Наложить маску. Use Boolean mask. |
| df.toclipboard() | Отправить DataFrame to clipboard. В буфер обмена. |
| with pd.ExcelWriter(path ='path\_to\_file.xlsx') as writer:  df.to\_excel(writer, sheet\_name='Выход') | Записать DataFrame в файл Эксель. Экспорт в Excel.  path - Path to xls or xlsx file.  engine:str (optional)- Engine to use for writing. If None, defaults to io.excel.<extension>.writer. NOTE: can only be passed as a keyword argument.  date\_format: str, default None. Format string for dates written into Excel files (e.g. ‘YYYY-MM-DD’).  datetime\_format: str, default None. Format string for datetime objects written into Excel files. (e.g. ‘YYYY-MM-DD HH:MM:SS’).  mode{‘w’, ‘a’}, default ‘w’. File mode to use (write or append).  sheet\_name - имя листа, на который записать таблицу  startrow: int, default 0 - Upper left cell row to dump data frame.  startcol: int, default 0- Upper left cell column to dump data frame.  header: bool or list of str, default True, Write out the column names. If a list of string is given it is assumed to be aliases for the column names.  index: bool, default True. Write row names (index). |
| s = pd.Series(range(7, 16)) | создаем массив (Series) с данными из диапазона 7 включ-но -16 не включительно |
| s2 = pd.Series(data, index=[ascii\_lowercase[i] for i in data]) | создаем словарь данных из массива data с индексами в виде букв |
| s3.index = [ascii\_uppercase[i] for i in range(10)] | назначить индексами другие символы |
| df.reset\_index(drop=True) | Переносит индексы текущие в отдельный столбец DataFrame, **создаёт стандартные индексы** с интеджер-значениями.  Чтобы новую колонку не создавать, а просто сбросить индексы, ставим drop=True |
| myDataFrame.set\_index('column\_name') | Сделать указанную колонку индексами. Создать индексы из колонки. |
| df['new\_column'] = df.index | Наоборот, сделать из индексов отдельную колонку |
| df.columns = map(str.lower, df.columns) | Переименовать названия колонок - сделать их lowercase |
| for column in df:  print(df[column]) | Итерирование по колонкам |
| for name, values in df.iteritems():  print('{name}: {value}'.format(name=name, value=values[0])) | Итерирование по колонкам, перебор колонок |
| df.columns.get\_loc("pear") | Узнать индекс колонки по имени колонки |
| colname = df.columns[pos] | Узнать имя колонки по индексу колонки |
| s2['a'] | выбор значения по ключу |
| s2[['a', 'd', 'g']] = 13 | присваиваем значение 13 ячейкам под указанными ключами |
| s4.name = 'Numbers'  s4.index.name = 'letters' | Присвоить имена столбцам значений и индексов |
| df = df.rename({'col one':'col\_one', 'col two':'col\_two'}, axis='columns') | **Поменять имена конкретных колонок** |
| list\_of\_column\_names = df.columns.tolist() | **Вывести лист имен колонок** |
| cols = df.columns.tolist()  cols = cols[-1:] + cols[:-1] # Меняем порядок, как хотим (левый включительно, правый невключительно)  df = df[cols] | **Поменять порядок колонок DataFrame. Поменять местами колонки, переставить колонки:**  [**https://stackoverflow.com/questions/13148429/how-to-change-the-order-of-dataframe-columns**](https://stackoverflow.com/questions/13148429/how-to-change-the-order-of-dataframe-columns) |
| df.columns = ['col\_one', 'col\_two'] | Переименовать ВСЕ колонки одним листом. |
| df.columns = df.columns.str.replace(' ', '\_')  df.add\_prefix('X\_')  df.add\_suffix('\_Y') | Поменять в имени колонок какой-то символ  Добавить к именам колонок префикс  Добавить к именам колонок суффикс |
| df.fillna()  df.fillna(0, inplace=True) | Заполнить ячейки со значениями NaN чем-то конкретным (передается в скобках) |
| DataFrame.sort\_values(by, axis=0, ascending=True, inplace=False, kind='quicksort', na\_position='last', ignore\_index=False, key=None)  df.sort\_values(by=['col1'], ascending=True, inplace=True) | **Сортировать** данные по указанной колонке (sort, order by).  Помни про inplace=True!  by=... str or list of str  Name or list of names to sort by.  if axis is 0 or ‘index’ then by may contain index levels and/or column labels.  if axis is 1 or ‘columns’ then by may contain column levels and/or index labels.  ascending=True - в порядке увеличения, если False - будет по убыванию  axis{0 or ‘index’, 1 or ‘columns’}, default 0  inplace bool, default False  ignore\_index bool, default False  If True, the resulting axis will be labeled 0, 1, …, n - 1.  na\_position{‘first’, ‘last’}, default ‘last’. Puts NaNs at the beginning if first; last puts NaNs at the end.  kind{‘quicksort’, ‘mergesort’, ‘heapsort’}, default ‘quicksort’. Choice of sorting algorithm. See also ndarray.np.sort for more information. mergesort is the only stable algorithm. For DataFrames, this option is only applied when sorting on a single column or label.  key callable, optional- Apply the key function to the values before sorting. This is similar to the key argument in the builtin sorted() function, with the notable difference that this key function should be vectorized. It should expect a Series and return a Series with the same shape as the input. It will be applied to each column in by independently.  <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.sort_values.html> |
| df['density'] = df.population / df.square \* 1\_000\_000 | **Добавление колонки** с просчётом в ней математического выражения на основании других колонок |
| pd.**concat**([df, df\_new], axis='columns') | **Добавить колнки из другого DataFrame.** Соединить несколько DataFrame. |
| data = pd.**merge**(data\_2017, data\_2018, left\_index= True, right\_index= True) | Объединить/соединить два DataFrame в одну таблицу. |
| df.name.**str.split**(' ', expand=True)  df[['first', 'middle', 'last']] = df.name.str.split(' ', expand=True) | **Разбить колонку name с текстовыми данными на несколько колонок**.Разделить данные в колонке на несколько колонок.  Например, ФИО на 3 колонки: Фамилия, Имя, Отчество**.** |
| df.drop(['density'], axis='columns')  df = df.drop(['density'], axis='columns') | Показать Датафрейм **без колонки** ‘density’.  Сбросить колонку, удалить колонку. |
| df = df.drop(df.columns[[0, 1, 3]], axis=1) | # df.columns is zero-based pd.Index сбросить несокько колонок по индексу |
| del df['<ColumnName>'] | **Удалить** колонку |
| df.append(pd.Series(df.sum(), name='Full summary')) | **Добавить строку** со значением, равным сумме значений верхних строк и именем ‘Full summary’ |
| area\_indexes = data[data[“Регион”].str.contains(“округ”)].index | Найти индексы строк, значение столбца «Регион» в которых содержит слово «округ». |
| df.drop\_duplicates(subset='col2')  df.drop\_duplicates(‘Column2’) | Удалить повторения (дубликаты) |
| a = pandas.Series([1,2,2,3,2], index=[1,2,3,4,5])  a.drop\_duplicates() - УДАЛЯЕТ И ВТОРОЙ, И ПОСЛЕДНИЙ!  Out[4]:  1 1  2 2  4 3  a.loc[a.shift(-1) != a] - УДАЛЯЕТ ИМЕННО СОСЕДНИЕ ДУБЛИКАТЫ  Out[3]:  1 1  3 2  4 3  5 2 | Удалить последовательно идущие дубликаты. Удалить повторяющиеся ряды.  <https://stackoverflow.com/questions/19463985/pandas-drop-consecutive-duplicates> |
| df.agg(‘sum’) | **показать суммы** значений по всем колонкам |
| df.agg([‘min’,’max’]) | показать минимумы и максимумы по всем колонкам |
| df.**idxmax**() | показать **индексы максимальных значений** для каждой колонки |
| df.loc[df.idxmax()] | показать максимальные значения в каждой колонке |
| df.apply( max, axis=1 ) | **Максимум по каждой строке.** |
| df[‘New column’] = df.apply(function, args=(df2, number,), axis=1) | Создать новую колонку, применив к каждому ряду функцию function с дополнительными аргументами - df2 и number. Первым аргументом становится каждый ряд df, который передается в фукцию как Series. |
| import pandas as pd  import swifter  df.**swifter.apply**(lambda x: x.sum() - x.min()) | <https://yandex.ru/turbo/nuancesprog.ru/s/p/6043/>  **Векторизация** функции.  Распараллеливание по ядрам и применение сразу к массивам данных, а не к каждой строке поочерёдно. |
| df['new\_group'] =  np.where( df['tax\_class'] == '1', 1, 2 ) | Функция np.where по умолчанию использует **векторизацию**. Принимает первым аргументом условие, вторым — результат выполнения условия, третьим — результат невыполнения условия.  <https://python-school.ru/speed-up-pandas/> |
| conditions = [  df['year\_built'] > 2000,  df['neighborhood'].str.startswith('WILLIAMSBURG'),  df['neighborhood'].isin(list1),  df['neighborhood'].isin(list2) ]  choices = [  'New built',  'Old WILLIAMSBURG',  'Good place',  'Nice place' ]  df['new\_group'] = np.select(conditions, choices, default='Not satisfied') | Функция np.select по умолчанию использует **векторизацию** |
| def haversine(): … | Примеры скорости работы разных расчётов  https://engineering.upside.com/a-beginners-guide-to-optimizing-pandas-code-for-speed-c09ef2c6a4d6 |
| df['distance'] = df.**apply**(lambda row: haversine(40.671, -73.985, row['latitude'], row['longitude']), axis=1) | Apply работает помедленне векторизации, но лучшечем iterrows или тупой for-loop по строкам (никогда не делай for-loop по строкам pandas) 90.6 ms ± 7.55 ms |
| df['distance'] = haversine(40.671, -73.985,  df['latitude'], df['longitude']) | # Vectorized implementation of Haversine applied on Pandas series  **ВЕКТОРИЗИРОВАННЫЙ** расчёт новой колонки в Pandas с использованием данных других колонок. ОЧЕНЬ БЫСТРЫЙ!  1.62 ms ± 41.5 µs |
| df['distance'] = haversine(40.671, -73.985, df['latitude'].values, df['longitude'].values) | **Векторизованный** расчёт на массивах Numpy.  **САМЫЙ БЫСТРЫЙ!**  We convert our latitude and longitude arrays from Pandas series to NumPy arrays **simply by using the values method of the series**.  **370 µs ± 18 µs** |
| haversine\_series = []  for index, row in df.iterrows():  haversine\_series.append(haversine(40.671, -73.985, row['latitude'], row['longitude']))  df['distance'] = haversine\_series | # Haversine applied on rows via iteration  Итерация по строкам через iterrows – очень медленная!  166 ms ± 2.42 ms |
| movies[(movies.genre == 'Action') |  (movies.genre == 'Drama') |  (movies.genre == 'Western')]  movies[movies.genre.**isin**(['Action', 'Drama', 'Western'])]  movies[**~**movies.genre.isin(['Action', 'Drama', 'Western'])] | Фильтр (отбор) по колонке на множество значений. Способ 1.  Способ 2. isin()  Реверсивный isin() - исключает из выборки указанные значения. Достаточно поставить тильду в начало. |
| df[(df.sys\_initial <= 120) & (df.sys\_final > 115)] | Отбор по 2 условиям в разных колонках (несколько условий) |
|  |  |
|  |  |
| df[‘City’].**value\_counts**(dropna=False) | Посчитать, сколько в данном столбце **каждого вида значений**. |
| df[‘City’].fillna(value=’VARIOUS’, inplace=True] | Вставить в пустые места столбца конкретные значения. |
| df[‘City’].fillna(df[‘City’].median(), inplace=True)  df[‘City’].fillna(df[‘City’].mean(), inplace=True) | Вставить в пустые места столбца средневзвешенное значение (медиану) этого же столбца. Или просто среднее арифметическое. |
|  |  |
| df['Sex'].**replace**(['male', 'female'], [1,0], inplace=True) | **Заменить данные категорийные на булевные / числовые в столбце** DataFrame  inplace = True заменит значения в указанном df, False - оставит df нетронутым, но создаст новый DataFrame |
| df.iat[0,4] = 1 | **Установить конкретное значение** в ячейке указанной по индексам |
| df.**at**[‘Diana’, ’Sex’] = 0 | **Установить конкретное значение** в ячейке указанной по **именам** строки и колонки |
|  |  |
| df.isnull().sum() | Проверяет, сколько в каждом столбце пропущенных значений (NaN) |
| df.dropna(how=’any’)  df.dropna(how=’all’) | Выкинуть строки, где хотя бы одно значение NaN  / где ВСЕ значения пустые. |
| df.dropna(subset=[’City’, ‘Color’], how=’any’) | Выкинуть строки, где пустые значения есть в указанных столбцах. |
|  |  |
| <https://datatofish.com/if-condition-in-pandas-dataframe/> | IF CONDIDTIONS |
|  |  |
| **ТИПЫ и ФОРМАТЫ** | |
| df.dtypes | Выводит типы значений каждой колонки |
| drinks.select\_dtypes(include='number')  drinks.select\_dtypes(include=['number', 'object', 'category', 'datetime'])  drinks.select\_dtypes(exclude='number') | Выбрать только колонки с цифровыми значениями.  Или с типами значений по списку.  Или все КРОМЕ цифровых. |
| df.astype({'col\_one':'float', 'col\_two':'float'})  pd.to\_numeric(df.col\_three, errors='coerce')  pd.to\_numeric(df.col\_three, errors='coerce').fillna(0)  df = df.apply(pd.to\_numeric, errors='coerce').fillna(0) | Поменять формат со строк на числа  Если знаем, что означают NaN значения, можем вставить их сразу.  apply this function to the entire DataFrame all at once by using the apply() method |
| pd.cut(titanic.Age, bins=[0, 18, 25, 99], labels=['child', 'young adult', 'adult']) | Поменять формат с чисел на строки, категории. |
| df.City.astype(int) | **Сменить тип данных** столбца на заданный |
| df.price.str.replace(‘$’, ‘’).astype(float).mean() | **Меняем формат** «валюта» (currency, decimal) на float в столбце “price”, переходя на функции строк, выдёргивая знак доллара, затем меняя формат. Заодно считаем среднюю цену. |
| pd.**set\_option**('display.float\_format', '{:.2f}'.format) | Отформатировать вывод всех float-значений. Меняем количество знаков после запятой (точки). Применяется к модулю Pandas в целом, а не к Dataframe. |
| format\_dict = {'Date':'{:%m/%d/%y}', 'Close':'${:.2f}', 'Volume':'{:,}'}  df.style.format(format\_dict) | Форматируем сразу несколько столбцов с помощью словаря. |
| stocks.style.format(format\_dict).hide\_index()  .highlight\_min('Close', color='red')  .highlight\_max('Close', color='lightgreen')  stocks.style.format(format\_dict).hide\_index()  .background\_gradient(subset='Volume', cmap='Blues') | Ещё методы форматирования вывода. |
| df['Дата'] = pd.to\_datetime(df['Дата']) | Перевести строки с датами в даты |
| df['date'] = df["date"].dt.strftime('%Y-%m') | Изменить формат из типа данных „datetime64“ в «строку из даты» нужного формата |
| df['Дата'] = df['Дата'].dt.date | Перевести формат даты из полного (с датой и временем) в сокращённый — только дата |
| mask = pd.to\_datetime(df.columns, format='%d-%b-%Y', errors='coerce').notna()  df.loc[:, mask] | Найти ячейки, в которых есть дата **в указанном формате**. |
|  |  |
| **АНАЛИЗ DATAFRAME** | |
| **import pandas\_profiling**  **pandas\_profiling.ProfileReport(df)**  profile = ProfileReport(df, title='Pandas Profiling Report', explorative=True)  profile.to\_file("your\_report.html") | **Мощный инструмент изучить DataFrame**. Множество метрик и важных данных сканируется одной командой.  В jupyter notebook выведет данные на экран.  Эти 2 строки надо добавить для вывода в файл при использовании, например, в IDE. |
| df.info() | Стандартная информация о DF |
| df.head()  df.tail() | Показать первые или последние 5 элементов ДФ |
| s4.median()  s4.max()  s4.min() | показать среднее, максимальное и минимальное значения Series |
| A B C  0 1 0 0  1 2 2 0  df.any()  A True  B True  C False | Whether each column contains at least one True element (the default).  По умолчанию - проверяет столбцы. В доках сказано «reduce the index», что означает, что проверяет столбцы. Т.е. любое значение из столбца должно соответствовать условию. |
| A B  0 True 1  1 False 0  df.any(axis='columns')  0 True  1 False | Хотя бы одно значение соответсвует условию. Любое значение соответствует условию.  axis='columns' означает «reduce the columns». Итог будет по строкам. |
| numerics = ['int16', 'int32', 'int64', 'float16', 'float32', 'float64']  newdf = df.select\_dtypes(include=numerics) | Выбрать только колонки с числовыми данными |
| df.select\_dtypes(include=np.number).columns.tolist() | Выбрать только колонки с числовыми данными |
| df.\_get\_numeric\_data() | То же самое.  Выбрать только колонки с числовыми значениями |
| mask = pd.to\_datetime(df.columns, format='%d-%b-%Y', errors='coerce').notna()  df.loc[:, mask] | Найти ячейки, в которых есть дата в указанном формате. |
|  |  |
| df.nsmallest(3, 'Col2') | Показать **три наименьших значения**. Равносильно df.sort\_values.tail(3) |
| df.nlargest(3, 'Col2') | Показать **три наибольших значения**. Равносильно df.sort\_values.head(3) |
| df.columns | **лист имён колонок** |
| df.index | Лист индексов (имен строк) |
| boolean = not df["Student"].is\_unique # True (credit to @Carsten)  boolean = df['Student'].duplicated().any() # True | Два способа проверить, **есть ли дубликаты у значений в колонке**  (для каждого значения/ряда, как всё в пандас) |
| df.columns[(df == 38.15).iloc[0]]  In [52]:  (df == 38.15).idxmax(axis=1)[0]  Out[52]:  'col7' | Вывести имена колонок, в которых есть конкретное значение  <https://stackoverflow.com/questions/38331568/return-the-column-names-for-a-specific-value-in-a-pandas-dataframe> в указанном ряду |
| cols = df.columns[(df==‘искомое значение’).iloc[i]] | Лист с именами колонок, содержащих искомое значение в конкретной строке с индексом i  Отбор по значению без указания имен колонок. Отбор во всех колонках. |
| df[‘cols’] = str(df.columns) | Создать стролбец со списком колонок |
| df1 = df[['a', 'b']] | Выбрать несколько колонок DataFrame |
| ~~df[(df==‘искомое значение’)].columns.any()~~ | ~~Выбрать колонки, значение в которых совпадает с указанным, содержащие указанное значение.~~ |
| ~~df[[ ‘Column3’, ‘Column7’ ]] [(df==‘искомое значение’)].columns.any()~~ | ~~Сначала сократить список колонок, а потом указать колонки с искомым значением из данного списка.~~ |
| len(df.index)  df.shape[0]  df.shape[1] | Количество строк  опять же количетсво строк  Количество столбцов |
| df[df['date'].astype(str).str.**contains**('07311954')] | Найти строки в столбце, СОДЕРЖАЩИЕ определенное значение. |
| df2 = df.genre.value\_counts()  df [df.genre.isin(df2.nlargest(3).index)].head() | Количество строк в каждом значении колонки «жанр» (сколько раз встречается каждое значение)  Показать строки, входящие в 3 наибольших жанра |
| df2.col.isin(df1.col) | Проверяет для каждого значения солонки col df2, входит ли значение в колонку col df1 |
|  |  |
| df.ne()  df.ne(0) | Возвращает True или False в зависимости, соответствует ли значение тому, что в атрибутах метода. Например, 0 в ячейке или нет - True / False. |
|  |  |
|  | |
|  |  |
|  |  |
|  |  |
| df['Calls'].mean().round() | Округлить значение. Математическое округление. |
| **СРЕЗЫ** | |
| df ['country'].unique() | показать **уникальные значения в колонке** |
| dataFrame.loc[<ROWS RANGE> , <COLUMNS RANGE>]  columnsData = dfObj.loc[ : , 'Age' ]  columnsData = dfObj.loc[ : , ['Age', 'Name'] ]  subset = dfObj.loc[ ['c' , 'b'] ,['Age', 'Name'] ]  subset = dfObj.loc[ 'a' : 'c' ,'Age' : 'City' ] | Срезы по именам.   * все строки, колонка Age * все строки, несколько колонок: Age и Name * строки с и b, колонки Age и Name * ДИАПАЗОНЫ рядов с c до b, колонок c Age по Name |
| max\_speed shield  cobra 1 2  viper 4 5  sidewinder 7 8  >>> df.loc['viper']  max\_speed 4  shield 5  Name: viper, dtype: int64 | Получить ряд по его имени.  Или группу рядов по их именам |
| >>> df.loc['cobra', 'shield']  2 | **Получить одну ячейку по имени ряда и имени колонки** |
| dataFrame.iloc[<ROWS INDEX RANGE> , <COLUMNS INDEX RANGE>]  dfObj.iloc[: , [0, 2]] | Срезы по номерам строк и столбцов.   * все строки, диапазон колонок с индекса 0 до индекса 2 |
| df['population'] | выбор столбца Датафрейма по его имени, при присваивании другому столбцу будет для каждой строки другого выбирать 1 элемент из указанного из того же ряда. |
| df.loc['KZ'] | выбор строки Датафрейма по её имени |
| df.loc['KZ']['population']  df.loc['KZ', 'population'] | выбор ячейки датафрейма по именам строки и столбца |
| df.loc['KZ':, 'square'] | **Срез** Датафрейма от строки Казахстана до последней строки  и по столбцу square |
| df.loc[:, ['sys\_final', 'dia\_final']] | Срез ВСЕХ строчек Датафрейма по 2 колонкам (несколько колонок) |
| df1 = df[['a', 'b']] | Выбрать несколько колонок DataFrame |
| df1 = df.iloc[:, 0:2] | **Срез колонок с первой по 2 (в срезах правая цифра не включительно)** |
| df.iloc[:3] | **Срез** Датафрейма по ИНДЕКСУ строки от начала до 3 элемента не включительно. Получаем DataFrame до 3 строки. |
| df.iloc[:, [1]] | **Срез** Датафрейма по ИНДЕКСУ столбца (индексу колонки) - второй столбец в данном случае. |
| **df['Колонка 3'].iloc[n]** | **Выбираем столбец по названию (str), а строку - по индексу (int).** Получаем конкретную ячейку по имени колонки и индексу строки. |
| **df.iloc[0][‘A’]** | **Выбрать значение ячейки по индексу строки и названию колонки** |
| df.loc[::-1] | «Питоновский» способ развернуть ряды в DataFrame |
| df.loc[:, ::-1] | «Питоновский» способ развернуть задом наперёд колонки в DataFrame (reverse) |
|  |  |
| df[df.population > 42]**[[**'country', 'square'**]]** | **Отбор из Датафрейма строк**, ГДЕ колонка ‘population’ содержит значение больше 42.  Для них показать срез колонок ‘country’ и ‘square’ (Две пары скобок!) |
|  |  |
| **ЛОГИЧЕСКИЕ ОПЕРАТОРЫ, УСЛОВИЯ IF** | |
| (a == b) & (c > d)  df[‘new’] = np.where((check[‘one’]==check[‘two’]) & (check[‘three’]>check[‘four’]), ‘ИСТИНА’, ‘ЛОЖЬ’) | Во-первых, в Pandas логические условия записываются каждое в скобках, а логические операторы следующие:  **exp1 and exp2 # Logical AND**  **exp1 or exp2 # Logical OR**  **not exp1 # Logical NOT**  **...will translate to...**  **exp1 & exp2 # Element-wise logical AND**  **exp1 | exp2 # Element-wise logical OR**  **~exp1 # Element-wise logical NOT**  ~~use eq instead of ==~~ |
| Apply - самый «питоновский» способ.  def alert(c):  mapping = {1.0: 'Full', 0.0: 'Empty'}  return mapping.get(c['used'], 'Partial')  df = pd.DataFrame(data={'portion':[1, 2, 3, 4], 'used':[1.0, 0.3, 0.0, 0.8]})  df['alert'] = df.**apply**(alert, axis=1)  # portion used alert  # 0 1 1.0 Full  # 1 2 0.3 Partial  # 2 3 0.0 Empty  # 3 4 0.8 Partial | Столбец, заполняемый по условию, делается через отдельную функцию. |
| df.loc[(УСЛОВИЕ), отбираемые столбцы]  df.loc[df['used'].eq(1.0), 'alert'] = 'Full'  df.loc[df['used'].eq( 0.0), 'alert'] = 'Empty'  df.loc[(df['used'] >0.0) & (df['used'] < 1.0), 'alert'] = 'Partial' | Столбец, заполняемый по условию, делается через отдельную функцию.  Более быстрое решение для интерпретатора.  <https://stackoverflow.com/questions/27041724/using-conditional-to-generate-new-column-in-pandas-dataframe> |
| df['alert'] = np.where(df.used == 1, 'Full',  np.where(df.used == 0, 'Empty', 'Partial')) | Более понятное для программиста. Тоже довольно быстрое. |
| np.where(m, A, B)  is roughly equivalent to  A.where(m, B) in pandas | Пока в np получается криво, а в pandas не выходит вовсе |
|  |  |
|  |  |
| **GROUPBY** | |
| orders.groupby('order\_id').item\_price.sum() | **Groupby**. Отобрать строки, относящиеся к одной группе (заказ №1) и просуммировать их цену. |
| orders.groupby('order\_id').item\_price.**agg**(['sum', 'count']) | Посчитать по группе сразу два значения, применить две функции, две выборки - общую цену и количество элементов в группе. |
| df['total\_price'] = df.groupby('order\_id').item\_price.**transform**('sum') | Общая цена. Присоединить **новый столбец, в котором будет подсчитана сумма** заказа для каждой строчки. Transform приводит длину столбца sum к необходимой. |
| titanic.groupby(['Sex', 'Pclass']).Survived.mean() | Группировка по 2 признакам (Мультииндекс) |
| titanic.groupby(['Sex', 'Pclass']).Survived.mean().**unstack**() | **Создание отдельного DataFrame по Мультииндексу** с строками и колонками, указаннами в мультииндексе groupby |
| titanic.**pivot\_table**(index='Sex', columns='Pclass', values='Survived', aggfunc='mean') | То же самое, но через «повёрнутую таблицу» - **pivot\_table**  Создаёт таблицу связи одного значения с другим (разных колонок). |
| summary = df.groupby('Name')['Fine sum'].**agg**(  ['count', 'sum']).**sort\_values**(['count'], ascending=False) | **Группировка** штрафов из ДФ по названию, вывод на экран суммы штрафов, обработанной функциями подсчета количества и суммы всех значений. Сортировка - по количеству. По убыванию (не по возрастанию) |
|  |  |
| df.groupby(['<Column1Name>','<Column2Name>'])['<Uniqe Column>'].**count**() | подсчитать **количество** по уникальной колонке, сгруппировав по двум признакам (колонки 1 и 2) |
| pvt = titanic.pivot\_table(  index=['Sex'],  columns=['PClass'],  values='Name',  aggfunc='count'  ) | **Создать группировку** по полу, классу и подсчитать количество в данных группах |
|  |  |
| df1\_grouped = df1.groupby('atable')  # iterate over each group  for **group\_name, df\_group** in df1\_grouped:  print('\nCREATE TABLE {}('.format(group\_name))  for **row\_index, row** in df\_group.iterrows():  col = … | **Итерация по группам** в DataFrame |
| Suppose you have this series:  delivery  2001-01-02 0 2  1 3  6 2  7 2  9 3  2001-01-03 3 2  6 1  7 1  8 3  9 1  dtype: int64  If you want one delivery per date with the maximum value, you could use idxmax:  dates = series.index.get\_level\_values(0)  series.loc[series.groupby(dates).idxmax()] | Работа с Мультииндексным Dataframe (группы, вложенные в группы)  <https://stackoverflow.com/questions/27914360/python-pandas-idxmax-for-multiple-indexes-in-a-dataframe> |
| **ВИЗУАЛИЗАЦИЯ** | |
| df['hospital\_id'].value\_counts().plot.pie(figsize=(5, 5)) | Сделать круговую диаграмму (диаграмма-пирог) |
| df['hospital\_id'].value\_counts().plot.bar() | Сделать столбчатую диаграмму |
| pd.crosstab(df.drug\_admst, df.gender).plot.bar() | Стобчатая диаграмма из 2 ДАТАФРЕЙМОВ (2 таблицы в 1 график) |
|  |  |
| pd.set\_option('display.width', 40)  print df | Установить ширину дисплея терминала. По умолчанию 80.  Показать таблицу в терминале, растянув на ширину экрана. |
| pd.set\_option('display.max\_columns', 30)  print df | Установить показываемое количество колонок. По умолчанию 20.  Показать таблицу в терминале, растянув на ширину экрана. |
|  |  |
|  |  |
| pd.show\_versions()  pd.\_\_version\_\_ | Показать версии установленных библиотек.  Показать версию только Панды |
| 1) df = pd.DataFrame({'col one':[100, 200], 'col two':[300, 400]})  2) pd.DataFrame(np.random.rand(4, 8))  3) pd.DataFrame(np.random.rand(4, 8), columns=list('abcdefgh')) | Варианты **создать тестовый DataFrame** с рандомными значениями |
| ufo.isna().sum()  ufo.isna().mean() | **Показать количество пустых ячеек** (без значений, NaN, null, none) в каждой колонке.  То же самое, но показать процент пустых ячеек. |
| df.dropna(axis='columns')  df.dropna(thresh=len(df)**\*0.9**, axis='columns') | **Отбросить колонки с пустыми ячейками**.  Отбросить колонки, в которых больше 10% ячеек пустые. |
|  |  |
|  |  |
| movies\_1 = movies.sample(frac=0.75, random\_state=1234) | **Выбрать случайные строки** из DataFrame. frac - доля строк от общего числа (длина выборки, random\_state - значение для псевдорандомизации выборки) |
| **SKLearn** |  |
| titanic.describe() | Описать столбцы DataFrame стандартными метриками: количество элементов, среднее значение, минимальное значение, максимальное, значения, равные 25, 50 и 75%. |
| s = (data.dtypes == 'object')  object\_cols = list(s[s].index) | Найти колонки с данными типа “object” (обычно str, categorical variables) |
| object\_cols = [col for col in X\_train.columns if X\_train[col].dtype == "object"] |
| good\_label\_cols = [col for col in object\_cols if  set(X\_train[col]) == set(X\_valid[col])] | Проверка совпадения значений колонок тренировочной и вввалидационной базы данных |
| bad\_label\_cols = list(set(object\_cols)-set(good\_label\_cols)) | # Problematic columns that will be dropped from the dataset |
| clean\_X\_train = X\_train.select\_dtypes(exclude=['object']) | 1. Удалить колонки с данными типа “object” |
| from sklearn.preprocessing import LabelEncoder  label\_encoder = LabelEncoder()  for col in object\_cols:  label\_X\_train[col] = label\_encoder.fit\_transform(X\_train[col])  label\_X\_valid[col] = label\_encoder.transform(X\_valid[col]) | 1. Применить встроенную sklearn функцию, меняющую данные object на int-заменители. |
| from sklearn.preprocessing import OneHotEncoder  OH\_encoder = OneHotEncoder(handle\_unknown='ignore', sparse=False)  OH\_cols\_train = pd.DataFrame(OH\_encoder.fit\_transform(X\_train[object\_cols]))  OH\_cols\_valid = pd.DataFrame(OH\_encoder.transform(X\_valid[object\_cols]))  OH\_cols\_train.index = X\_train.index  OH\_cols\_valid.index = X\_valid.index  num\_X\_train = X\_train.drop(object\_cols, axis=1)  num\_X\_valid = X\_valid.drop(object\_cols, axis=1)  OH\_X\_train = pd.concat([num\_X\_train, OH\_cols\_train], axis=1)  OH\_X\_valid = pd.concat([num\_X\_valid, OH\_cols\_valid], axis=1) | 1. применить функцию, **вставляющую вмето object несолько столбцов, по 1 на каждое значение** столбца с categorical variables   # Apply one-hot encoder to each column with categorical data  # One-hot encoding removed index; put it back  # Remove categorical columns (will replace with one-hot encoding)  # Add one-hot encoded columns to numerical features |
|  |  |
|  |  |
|  |  |

Трудности Pandas:

1. Не работает синтаксис Python - нет if, for, list comerhansions…
2. Нет выпадающих подсказок с возможными методами, которые можно было бы применить через точку к DataFrame.
3. Описание ошибкок очень часто не содержательно. Или вовсе ошибочно, например для мультииндексов выходило «нет метода count()», хотя стоит сделать левее в pipe-line groupby для одной колонки, а не для двух - и вдруг count(), стоящий на несколько команд правее - вдруг находился(да и куда он вообще может деться?).

|  |  |
| --- | --- |
| **Spark** |  |
| import findspark  findspark.init ('/media/anton/Toshiba/Programming/Python/BigData\_Spark/spark-3.0.0-bin-hadoop2.7')  from pyspark.sql import SparkSession  spark = SparkSession.builder.appName('ops').getOrCreate()  df = spark.read.csv('sales\_info.csv', inferSchema=True, header=True) | Подключение библиотеки spark и запуск сессии |
| df = spark.read.json('people.json') | Прочитать json |
| df.**show**() | **Показать** весь df |
| mean\_val = df.select(mean(df['Sales'])).**collect**() | Сохранить данные в переменную (почему не достаточно просто присвоить прееменной?) |
| df.printSchema() | Показать схему df |
| df.describe() | Короткое описание df |
| df.describe().show() | Статистическая справка по колонкам (более подробное описание df) |
| data\_schema = [StructField('age', IntegerType(), True),  StructField('name', StringType(), True)]  final\_struc = StructType(fields=data\_schema)  df = spark.read.json('people.json', schema=final\_struc) | Открываем json, сразу приводя колонки к нужным параметрам через «схему» |
|  |  |
| df.select('age') | Сделать DataFrame из одной колонки |
| df = df.withColumn('newage', df['age']\*2) | Создание новой колонки из данных других колонок (not inPlace) |
| df = df.withColumnRenamed('age', 'newName').show() | Переименовать колонку |
| df.select(‘age’, ‘oldName’.alias(‘NewName’)) | Переименовать колонку при отборе |
| df.createOrReplaceTempView('people') | Создать SQL представление ДатаФрейма с указанным именем |
| new\_results = spark.sql('SELECT \* FROM people WHERE age=30') | Отбор из созданного SQL представления |
|  |  |
| df.filter('Close < 500').show() | Фильтр по колонке “Close”, значения меньше 500 |
| df.filter('Close < 500').select('Open').show()  f.filter('Close < 500').select(['Open', ‘Close’]).show() | Тот же фильтр, но селектим (показываем) из отобранного колонку “Open” |
| df.filter((df['Close'] < 200) & ~ (df['Open']>200)).show() | Отбор по 2 условиям – каждое берется в скобки, используются логические операторы & - and, | - or ~ - not |
| df.filter(df['Low'] == 197.16).show() | Отбор по конкретному значению в колонке |
| result = df.filter(df['Low'] == 197.16).collect() | Collect выдаёт лист объектов ‘Row’ или ‘Column’, сформированных по нашим фильтрам, который можно сохранить в переменную и использовать. |
| row = result[0]  row.asDict()[‘Volume’] | Первый элемент нашего результата выводим как словарь.  И выбираем из него значение ключа Volume |
|  |  |
| df.orderBy('Sales').show() | Упорядочить по указанному столбцу |
| df.orderBy(df['Sales'].desc()).show() | Упорядочить в обратном порядке по Sales |
|  |  |
|  |  |
| **GROUP BY** | |
| df.groupBy('Company').mean().show()  df.groupBy('Company').count().show()  df.groupBy('Company').agg({'Sales': 'count'}).show() | Показать средние продажи по компаниям  Количество продаж по компаниям (в данном случае продажи - единственная оставшаяся колонка)  То же самое: количество продаж по компаниям (группам) |
| df.agg({'Sales': 'sum'}).show() | Показать СУММУ (‘sum’) продаж (‘Sales’) по всему ДатаФрейму |
| **from pyspark.sql.functions import countDistinct, avg, stddev**  df.select(countDistinct('Sales')).show()  df.select(avg('Sales')).show() | Применить функцию через select  посчитать по ‘Sales’ уникальные значения  средние значения |
| df.select(avg('Sales').**alias**('Average Sales')).show() | Дать имя отбору (временной колонке) |
| sales\_std = df.select(stddev('Sales').alias('std')).show()  **from pyspark.sql.functions import format\_number**  sales\_std.select(format\_number('std',2)).show() | Показать среднюю девиацию.  Отформатировать числа, второй аргумент – количество знаков после запятой |
|  |  |
| **MISSING DATA** | |
| df.na.drop().show() | Просто выбросить строки с пропущенными lанными (равными null) |
| df.na.drop(thresh=2).show() | Сбросить строки в которых как минимум две ячейки с нормальными данными (non null, ценные) |
| df.na.drop(how='any').show()  df.na.drop(how='all').show() | Сбросить строки, в которых как минимум одна любая ячейка null  Сбросить только те, где все null |
| df.na.drop(**subset**=['Sales']).show() | Сбросить только те, где null **в указанном столбце** Sales |
| df.na.fill('FILL VALUE').show() | Заполнить пустые ячейки указаннм значением, ЕСЛИ совпадает тип данных (в примере - str) |
| df.na.fill('No Name', **subset**=['Name']).show() | Заполнить пустые ячейки в указанной колонке указаннм значением |
| mean\_val = df.select(mean(df['Sales'])).collect()  mean\_sales = mean\_val[0][0]  df.na.fill(mean\_sales, ['Sales']).show() | Находим среднее значение Sales и заполняем пустые ячейки в колонке средним значением |
| df.na.fill(df.select(mean(df['Sales'])).collect()[0][0], ['Sales']).show() | Заполняем средним значением - в одну строку |
|  |  |
|  |  |
| **DATES AND TIMESTAMPS** | |
| from pyspark.sql.functions import (dayofmonth, hour, dayofyear, month, year, weekofyear, format\_number, date\_format) | Функции для извлечения частей даты и Timestamp |
| df.select(dayofmonth(df['Date'])).show() | Показать из даты только день месяца (второй, десятый, третий...) |
| newdf = df.withColumn('Year', year(df['Date']))  newdf.groupBy('Year').mean().show()  newdf.groupBy('Year').mean().select(['Year','avg(Close)']).show() | Создать новый столбец «Year»  Сделать DataFrame со средними значениями ВСЕХ колонок, по годам. Средние за 2010, за 2011, 2012 и т.д.  Из сделанного Датафрейма средних значений показать Year и Aavg(Close) |
| result.select('Year', format\_number('avg(Close)',2).alias('AVG Close')).show() | Выбрать 2 колонки, отформатировать значения в колонке  и переименовать колонку |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |