Pandas Tutorial

Jihuan Zhang

November 25, 2025

The main purpose of this tutorial is to help you pass the online assessments for the data analyst/scientist positions. This tutorial begins with the essential parts of Pandas. Everything else is presented in the last section, which may includes detailed usages, advanced usages, useful methods/attributes, and other useful functions.

It is not possible to memorize all the details in the last section. However, it is necessary that you memorize and understand everything except the last section.

1 Preliminaries

1.1 Attributes and Methods

Attributes have no parentheses, e.g., df.shape, df.values. Methods have parentheses, e.g., df.sum(), df.dropna().

1.2 DataFrames and Series

1.2.1 Series

A Series has attributes `name`, `index`, `values`, etc[1]. A Series has no attribute `columns`!

```
Series.values # equivalent to Series.iloc[0:]
```

1.2.2 DataFrames

A DataFrame has attributes `columns`, `index`, `values`, etc. Each column of a DataFrame is a Series with `name` attribute equal to the corresponding column name.

2 Creating DataFrames/Series

```
pd.DataFrame([scalar]) # from a scalar
pd.DataFrame(Series) # from a Series
pd.DataFrame([[1,2,3],[4,5,6]]) # from a list; # each element will be a row in the DataFrame
pd.DataFrame({'col1':[rows], 'col2':[rows], ...} ) # from a Dictionary
pd.Series(..., name = 'name') # create a Series; Series has no 'column'
```

3 Displaying Sub-DataFrame/Series

3.1 Indexing

3.1.1 Brackets []

```
df[['col1','col2']] # DataFrame of two columns
df[['col1']] # dataframe of one column
df['col1'] # Series
```

Slice is allowed in `[]` too.

```
df[1:5] # sub-DataFrame consists of row 1 to 4
df[-3:] # last three rows
Series[1:5] # sub-Series
```

[1] Both `df1 +=*/**// df2` and `Series1 +=*/**// Series2` work and are elementwise, and the result is still a `df` and a `Series`, respectively; for calculations with vector, e.g, inner and outer product, we need to use the attribute '`.values`' which is an array, e.g., `df['col1'].values @ df['col2'].values`

3.1.2 `iloc`: Integer-Location Based Indexing

A slice object is Python's internal representation of the colon notation `start:stop:step` (`:step` is optional if `step` is one). For example, `0:5:1 == 0:5 == slice(0,5,1) == slice(0,5)`. A slice cannot be an element of a list. `DataFrame.iloc[]` and `Series.iloc[]` only support integer, list of integer(s), and slice.

```
df.iloc[:,[0]] # df.iloc() is wrong
df.iloc[-3:,0:5] # df.iloc[-3:,[0:5]] is wrong
df.iloc[:,0] # this returns a Series
df.iloc[0,:] # this is also a Series! df.iloc[0,:].name will be the corresponding index of df; df.iloc
    [0,:].index will be the corresponding columns of df
df.iloc[0,0] # this returns a scalar
Series.iloc[0:5] # sub-Series
Series.iloc[1] # scalar, e.g., df['col1'].iloc[0]
```

3.1.3 `loc`: Location (Index/Label) Based Indexing

```
df.loc[index_1:index_n,'col_1':'col_n'] # location (label) based indexing; if the index is labeled by
    0,1,2,..., then df.loc[1:3,:] works and returns row 1 to 3 (INCLUDING 3); if the index is labeled
    differently, say 'a','b','c',..., then df.loc[1:3,:] will return an error
df.loc[index_1,:] # similarly, this returns a Series instead of a dataframe
df.loc[index_1,'col_1'] # similarly, this returns a scalar
Series.loc[index_1:index_n]
```

3.2 Boolean Indexing by Filtering Conditions

`DataFrame[]` and `Series[]` also allow boolean array/Series of conformable dimensions.

3.2.1 Single Condition

```
df[df['col'] <= val].iloc[:,1:3] # if you use R, you might have intuitively tried df.iloc[[df['col'] <= val
    ],1:3]. This is wrong in pandas
df[df['col'].between(val1, val2)] # between; df[1 <= df['col'] <= 10] is wrong syntax
```

`isin()`: the input can be a list, tuple, set, dictionary (check keys only), or a Series.

```
Series.isin(...) # "Series in ..." is wrong
```

Strings Filtering:

```
df[df['col1'].str.startswith('abc')]
df[df['col1'].str.endswith('abc')]
df[df['col1'].str.contains('abc')]
```

3.2.2 Multiple Conditions

For DataFrames/Series[2], operators must be &, |, and ~, instead of `and`, `or`, and `not`. If there are more than one condition, then each condition must be in parenthesis `()`.

```
df[(df['col'] < 1000) | (df['col2'].isna())] # df[df['col'] < 1000 or df['col2'].isna()] is wrong
```

3.2.3 Column Filtering

This is not commonly used.

```
df.iloc[:,[i%2==1 for i in range(len(df.columns))]]
df.iloc[:,[j for j in range(1,len(df.columns)+1,2)]]
df.loc[:, df.columns.str.endswith('sting')]
```

[2]Operators also works for DataFrames. The results will be DataFrames of Bools

4 Modifying DataFrames/Series

4.1 Concatenation

```
pd.concat([df1, df2], axis=0, # default; axis=1 means you concatenate along column.
  join='outer', # default; ='inner' only concat unique rows
  ignore_index=False)  # default; =True to re-index
pd.concat([Seires1, Series2], axis=1) # concat two Series by column
```

4.2 Changing Column Names

4.2.1 Changing Some Column Names

```
df.rename(columns={'old_name': 'new_name'})
```

4.2.2 Changing All Column Names Using f-String

```
# f"..." is a formatted string literal (an f-string). It lets you put expressions inside "..." that are
    evaluated and converted to text.
# enumerate(df.columns) convert column names to Iterable, whose elements are [0,'colname1'], [1,'colname2
    '],...
df.columns = [f"X_{i+1}" if i <= 100 else c for i, c in enumerate(df.columns)]

# equivalently
n = 2  # want to modify column 0 to column n-1
new_names = [f"X_{i}" for i in range(1, n+1)]
df.rename(columns=dict(zip(df.columns[:n], new_names)), inplace=True)

# equivalently
n = 2
new_names = list(df.columns)
for i in range(0,n):
  new_names[i] = f'X_{i+1}'
df.columns = new_names
```

4.3 Deleting Rows/Columns

```
pd.DataFrame.drop(['col1', 'col2', ...], axis=1, inplace=True) # drop columns
```

4.4 Modifying Strings

```
# string values
Series.str.title() # Converts first character of each word to uppercase and remaining to lowercase.
Series.str.capitalize() # Converts first character to uppercase and remaining to lowercase.
Series.str.swapcase() # Converts uppercase to lowercase and lowercase to uppercase.
```

4.5 Sort

```
df.sort_values(by=['col1', 'col2', ...], # In lexicographic order
  axis=0, # default; =1 to sort columns (rarely used)
  ascending=True, # default; if using =[True, False, ...], must match length in 'by'
  inplace=False, # default
  na_position='last', # default; ='first' puts NaNs at the beginning
)
```

The index will be sorted too. To reset the index (i.e., make the index still 1,2,... after sort), use `.reset_index(drop=True)`. `drop=True` means the reset index will not be made as a new column.

```
df.sort_values(by=['col',...]).reset_index(drop=True)
```

4.6 Adding New Columns

We can directly add a new column too a DataFrame:

```
df['new_colname'] = df['col1']*12
```

4.6.1 Assign

Unlike direct assignment (`df['col'] = value`), `assign()` returns a new DataFrame without modifying the original.

```
df.assign(new_col1 = value1, new_col2 = value2, ...) # create new columns with identical values
df.assign(new_col1=df['col1']*12, new_col2=df['new_col1']/1000, new_col3=df['new_col2']>60000, ...)
```

Create new columns based on anonymous funcitons:

```
df.assign(
  bonus=lambda x: x['salary'] * 0.1,
  total_comp=lambda x: x['salary'] + x['bonus'],  # Uses 'bonus' created above!
  tax=lambda x: x['total_comp'] * 0.25
)
```

4.6.2 Advanced Methods

`pd.cut()`

```
# Custom numerical Grouping with pd.cut()
df['age_group'] = pd.cut(df['age'], bins=[0, 10, 20, 30], right=True, include_lowest=True) # right side
    inclusive; include_lowest=True: Without include_lowest, 0 would be NaN
# Instead of equal-width bins, create equal-frequency bins (quantiles)
df['age_quartile'] = pd.qcut(df['age'], q=4, labels=['Q1', 'Q2', 'Q3', 'Q4'])
```

`Series.map()`

```
# Custom string Grouping with map()
rating_groups = {
  'Good': 'High',
  'Excellent': 'High',
  'Normal': 'Low',
  'Bad': 'Low'
}
df['rating_group'] = df['rating'].map(rating_groups) # Create new column with groups
```

4.7 Dropping NAs/Duplicates

A duplicate is define by having the same row across all columns.

```
df.dropna() # drops all rows that contain at least one NA
df.drop_duplicates() # drop all duplicates except for the first occurrence
Series.unique() # returns an array; Series.drop_duplicates() also works but it returns a Series

df.dropna(axis=0, # default
  how='any', # default; ='all' will only drop rows that are straight NAs
  subset=None, # default
  inplace=False, # default
)

df.drop_duplicates(subset=None,
  keep='first', # {'first', 'last', False}, default ''first Determines which duplicates (if any) to keep. '
      first': Drop duplicates except for the first occurrence. 'last': Drop duplicates except for the last
      occurrence. False: Drop all duplicates.
  inplace=False # default
)
```

4.8 Displaying NAs/Duplicates (Advanced)

```
df.isna() # return a dataframe where missing values are False and others are True
df.isna().sum() # return a Series of count of missing values per column
df.isna().sum().sum() # return a scalar of total count of missing values

df.isna().any() # per column across rows: return a Series with unnamed name and indices given by columns
    names; False means the corresponding COLUMN contains no NA
df.isna().any(axis=1) # per row across columns: return a Series with unnamed name and indices given by the
    indices of df; False means the corresponding ROW contains no NA
df.isna().all() # per column across rows: return a Series with unnamed name and indices given by columns
    names; True means all rows in the corresponding COLUMN are NAs

df[df.isna().any(axis=1)] # return a dataframe containing rows with NA; axis=1 means by row, default is by
    column; of course, df[df.isna().any()] would not work
df[df[['col1','col2']].isna().any(axis=1)] # return a dataframe whose 'col1' or 'col2' contains rows with
    NA; axis=1 means by row, default is by column
df[df.isna().sum(axis=1) > N] # Rows with more than N NAs
df.isna().sum(axis=1) # Count NAs per row
df.columns[df.isna().any()] # column names that contain NAs

df.fillna('*') # return a dataframe whose missing values are set to be '*'
Series.fillna(int(df['col'].mean()), inplace=True) # can replace NA by some numbers

df.duplicated() # return a series of bools, True mean this row is a duplicate
df[df.duplicated()] # return duplicated rows
df[df.duplicated(['col1','col2'])] # find rows whose values in 'col1' and 'col2' as a tuple, is duplicated
```

4.9 Ranking

```
# return a Series
df['new_col'] = df['col_to_be_ranked'].rank(axis=0, method='average', numeric_only=False, na_option='keep',
     ascending=True, pct=False) # method:{'average', 'min', 'max', 'first', 'dense'}, default 'average'; '
    dense' ensures rank has no gap; ascending=True means the smallest number gets the highest rank.
```

Lexicographically rank multiple columns:

```
# Lexicographic ranking
df['_temp'] = list(zip(df['col1'], df['col2'])) # create a Series of 2-tuple
df['rank'] = df['_temp'].rank(ascending=False, method='min').astype(int) # 2-tuple is by default, ranked in
     lexicographic order
```

5 Aggregation and Transformation Functions

5.1 Aggregation Functions

Aggregation functions (reduce length): `.sum()`, `.mean()`, `.max()`, `.min()`, `.count()`, `.nunique()`, `.first()`, `.last()`, etc. These collapse each group into a single row.

5.2 Transformation Functions

Transformation functions (preserve length): `.rolling()`, `.cumprod()`, `.pct_change()`, `.shift()`, `.cumsum()`, `.cummax()`, `.cummin()`, `.rank()`, `diff()`, `ffill()`, `bfill()`, etc. These return a value for each original row

6 Groupby

`df.groupby(['col1', 'col2', ...])` returns a `DataFrameGroupBy` object, that is, `df` is partitioned into several sub-DataFrames, based on each unique combination of `['col1', 'col2', ...]`.

By default, if group keys contain NA values ('any', not 'all'), the corresponding rows will be dropped. If `dropna = False`, NA values will also be treated as the key in groups.

```
df.groupby(by = ['col1', 'col2', ...], dropna = True)
```

6.1 Groupby Specific Attributes

```
df.groupby(['col1', 'col2', ...]).ngroup # number of groups
```

6.2 Groupby Specific Methods

6.2.1 `size()`

`size()` is a groupby specific aggregation function. It returns a Series, with index given by `['col1', 'col2', ...]`, and values given by count of rows for each group, taking `None` into account. Make sure you understand its differences from `count()`.

```
df.groupby(['col1', 'col2', ...]).size()
```

6.3 `agg()`

`df.grouby(...)` returns a GroupBy object, which is different from DataFrame and Series.

Conduct single operation for a single column

```
df.grouby(['col1','col2'])['col3'].min() # group by ('col1','col2') and calculate min in 'col3' for each
    group; this returns a Series with name 'col3' and index given by each group ['col1','col2']
df.grouby(['col1','col2'])['col3'].min().reset_index() # this reset the index, changing ['col1','col2'] to
    columns, which make the previous Series to a DataFrame
df.grouby(['col1','col2'])[['col3']].min() # [['col3']] makes the result a DataFrame
df.grouby(['col1','col2'])['col3'].agg(['min']) # agg('min') also works (new column name will be different)

# rename the new column
df.grouby(['col1','col2'])['col3'].agg(new_colname='min') # return a dataframe even though we use ['col3']
    instead of [['col3']]

# one can customize functions in agg()
df.grouby(['col1','col2'])['col3'].agg([lambda x: x.isna().sum()]) # here x represents column2 in each
    group; x.isna() sets entries to be True for NA and False otherwise; sum()=sum(axis=0) by default, per
    column across rows
```

Conduct multiple operations for a single column

```
df.groupby('category')['value'].agg([
'count',    # Count of non-null values
'sum',      # Sum of values
'mean',     # Average (mean)
'median',   # Median value
'min',      # Minimum value
'max',      # Maximum value
'std',      # Standard deviation
'var',      # Variance
'first',    # First value
'last',     # Last value
'nunique',  # Number of unique values
])

# rename each new column
df.groupby(['col',...])['col2'].agg(new_colname1='min', new_colname2=lambda x:x.quantile(0.25), ...)
```

Conduct multiple operations for multiple columns

```
df.groupby(['col',...]).agg(new_colname1=('col1','min'), new_colname2=('col2',lambda x:x.quantile(0.25)),
    ...) # always return a DataFrame; if you want a Series, you need df.groupby(['col',...])['col1'].min()
```

`.reset_index()`

6.4 `transform()`

The syntax of `transform()` is slightly different from `agg()` as you can't assign new column names or conduct multiple transformations at once.

Conduct a single transformation for a single column

```
df.groupby(['col',...])['col2'].cumsum()
df.groupby(['col',...])['col2'].transform('cumsum')
df.groupby(['col',...])['col2'].shift(-1)
df.groupby(['col',...])['col2'].transform(lambda x:x.shift(-1))
df.groupby(['col',...])['col2'].transform(lambda x:(x-x.mean())/x.std())
```

Conduct multiple transformations for multiple columns

```
# you need to do it one by one
df['new_colname1'] = df.groupby(['col',...])['col2'].cumsum()
df['new_colname2'] = df.groupby(['col',...])['col3'].shift(-1)

# or, you can use assign(), but this is really unnecessary as there will be even more lines of code
df = df.assign(
  new_colname1 = df.groupby(['col',...])['col2'].cumsum(),
  new_colname2 = df.groupby(['col',...])['col3'].shift(-1)
)
```

7 Merge

```
pd.merge(left, right, how='inner', # default; can also be 'left', 'right', 'outer'
  on=['key1', ...], # key can contain duplicates (unlike unique 'key' for database)
  left_on=None, right_on=None, # in case left right keys are different colnames
  suffixes=('_x', '_y') # default; handles overlapping column names between left and right DataFrames
)
```

8 Time/Date Data

8.1 Creating Time/Date Data

```
df['time_column'] = pd.to_datetime(df['time_column']) # convert a column to time data type
date = pd.to_datetime('2015-01-25') # format = "%Y-%m-%d"
datetime = pd.to_datetime('2015-01-25, 15:33') # format = '%Y-%m-%d, %H:%M'
```

8.2 Displaying Time Information

```
date.day_name() # returns 'Sunday'
date.weekday() # returns 6; Monday is 0, Sunday is 6
date.year
date.quarter
date.isocalendar().week # i-th week of the year
date.day # i-th day of the month
date.month
date.month_name()
```

8.3 Calculation of Time/Date

8.3.1 Time/Date Difference

```
# for a Series, ".dt" stands for "datetime accessor"
(df['date1'] - df['date2']).dt.days # difference as number of days
(pd.to_datetime('2015-01-25') - pd.to_datetime('2015-01-14')).days # scalar needs no ".dt"

# Common .dt operations:
df['date'].dt.year       # Extract year
df['date'].dt.month      # Extract month
df['date'].dt.day        # Extract day
df['date'].dt.hour       # Extract hour
df['date'].dt.dayofweek  # Day of week (0=Monday)
df['date'].dt.strftime('%Y-%m-%d')  # Format as string
df['date'].dt.days       # Days from Timedelta
```

The following is a `Timedelta` object:

```
(pd.to_datetime('2015-01-25') - pd.to_datetime('2015-01-14')) == pd.Timedelta(days=2) # True; Available
    kwargs: {days, seconds, microseconds, milliseconds, minutes, hours, weeks}. years/months does not work
    because Timedelta represents a fixed duration, so it can't handle variable-length periods
```

But the following is just an integer:

```
(date1 - date2).days == 2 # true
```

8.3.2 Time/Date Addition

```
pd.to_datetime('2015-01-23') + pd.Timedelta(days=2) == pd.to_datetime('2015-01-25') # True; pd.to_datetime
    ('2015-01-31').day + 2 == 33, so we don;t want to do that
```

8.3.3 Time/Date Comparison

Pandas automatically converts the string to datetime for comparison.

```
pd['date1'] <= '2024-02-01'
```

8.3.4 `rolling()`

```
df = df.sort_values(['col1','date_col']) # must sort first
df.groupby(['col',...]).rolling(window='7D',on='date_col')['col1'].agg(...)

# .agg() on rolling doesn't support named aggregation syntax
posts.groupby('user_id').rolling(window='7D', on='post_date').agg(seven_day_posts = ('post_id','count')).
    reset_index() # this is WRONG

posts.groupby('user_id').rolling(window='7D', on='post_date')['post_id'].count().reset_index().rename(
    columns={'post_id':'seven_day_posts'}) # this is correct
```

Example:

```
import pandas as pd
import numpy as np

def rolling_volatility(prices, window=20, trading_days=252, log_returns=True):
  """
  prices: list or Series of prices
  window: rolling window length
  trading_days: number of trading days per year
  log_returns: whether to use log returns
  """
  prices = pd.Series(prices)

  # 1) Compute returns
  if log_returns:
    returns = np.log(prices / prices.shift(1))        # log return
  else:
    returns = prices.pct_change()                     # simple return

  # 2) Rolling volatility = rolling std of returns
  rolling_vol = returns.rolling(window).std()

  # 3) Annualized volatility
  annualized_vol = rolling_vol * np.sqrt(trading_days)

  return rolling_vol, annualized_vol
```

9 Everything else

9.1 Reading/Saving Datasets

9.1.1 Reading Datasets

```
data = pd.read_csv("../parent_folder/file.csv", # '_excel' for .xlsx; '_table' for .txt; '_json' for .json
header=None, # 1st row is data; =0: 1st row is column name
nrows=20, # only read first 20 rows
skiprows=0, # skip nothing; =20: skip first 20 rows; =range(0,21,2); =lambda x: x%2==1: skip odd rows
usecols=[0,1,2], # only read columns 0,1,2; can also be a list of column names
index_col=['positionId'], # use specific column for index; ow a new column will be created for index
keep_default_na=True, # set missing values as NaN; =False: set missing values as blank
na_values=[0,'NA'], # set all zero and 'NA' as NaN
na_filter=False, # do not do anything to the missing values, display the original values
dtype={'positionId':str, 'companyName':str}, # customize data type
parse_dates=['Year'] # set column 'Year' to be time series data
)
```

9.1.2 Saving Datasets

```
df.to_csv("out.csv", # output file name
encoding='utf-8',
index=False, # a new column will be created for index
na_rep='missing' # customize NaN
)
```

9.2 Other Functions/Methods

```
len(df) # this actually works and gives you the number of rows
isinstance(n, int) and n > 0 # check if n is a natural number
f'row_{i}' # makes i changeable
OrdinalEncoder().fit_transform(DataFrame) # factorize nonnumerical data
```

9.3 Descriptive Statistics

```
DataFrame/Series.sum()          # Sum of all values
DataFrame/Series.mean()         # Average
DataFrame/Series.median()       # Median value
DataFrame/Series.min()          # Minimum value
DataFrame/Series.max()          # Maximum value
DataFrame/Series.std()          # Standard deviation
DataFrame/Series.var()          # Variance
DataFrame/Series.count()        # Count non-null values
DataFrame/Series.nunique()      # Count unique values
DataFrame/Series.quantile(0.75) # 75th percentile
DataFrame/Series.prod()         # Product of all values
```

9.4 Data Inspection

```
df.head(n)            # First n rows (default 5)
df.tail(n)            # Last n rows (default 5)
df.shape              # (rows, columns)
df.info()             # Data types and memory usage
df.describe()         # Statistical summary
df.columns            # Column names
df.dtypes             # Data types of each column
df.index              # Index information
df.values             # Numpy array of values
df.memory_usage()     # Memory usage per column
```

9.5 Summary Descriptive Statistics

```
df.shape[0] # df.shape is a tuple. df.shape[0] is number of rows, df.shape[1] is number of columns
df.describe() # descriptive statistics only for numeric-type columns
df.describe(include='O') # descriptive statistics only for object-type columns (strings, list, dict, mixed)
    ; count: Number of non-null values; unique: Number of unique values; top: Most frequently occurring
    value; freq: Frequency of the most common value
df.describe(include='all')
df.describe().round(4) # round the results in 4 decimal spaces
df.describe().round(4).T # transpose
```

9.6 Pivit Table

```
df.pivot_table(values=None, # Column(s) to aggregate
index=None, # Column(s) for row labels
columns=None, # Column(s) for column labels
aggfunc='mean', # default ""mean
fill_value=None, # Value to replace NaN
margins=False, # If margins=True, special All columns and rows will be added with partial group aggregates
    across the categories on the rows and columns.
dropna=True # If True, rows with a NaN value in any column will be omitted before computing margins
)
```

9.6.1 Naming Rule for Columns by Pivit Tables

```
# Level 0: one of the aggfunc (if more than more)
# Level 1: one of the values
# Level 2: columns[0]
# Level 3: columns[1]
# Level 4: columns[2]  (if you have 3 columns parameters)
# ...

# When: single values + single columns + single aggfunc, result has simple column names
# ['product_val_1', 'product_val_2', ...]

# When: multiple values + multiple columns + single aggfun, result is a tuple
# [('value_name_1','col1_val1','col2_val1',...), ..., ('value_name_2','col1_val1','col2_val1',...), ...]

# When: multiple values + multiple columns + multiple aggfun, result is a tuple
# [('aggfunc_1', 'value_name_1','col1_val1','col2_val1',...), ..., ('aggfunc_2', 'value_name_1','col1_val1
    ','col2_val1',...), ...]
```

9.7 Grouper

```
df.grouby(pd.Grouper(key='date',freq='ME'))
```

9.8 Advanced Usages

```
pd.DataFrame(Counter(array).elements()) # dict
pd.DataFrame(Counter(array).values()) # dict

pd.DataFrame(range(5)) # Iterable
pd.DataFrame(permutations([1,2,3,4,5])) # Iterable
```