Pandas Notes

Jihuan Zhang

1 Reading/Saving Datasets

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
  )
```

1.2 Creating Datasets

```
data = [[1,2,3],[4,5,6]] # each element represent a row in dataframe
pd.DataFrame(data) # ndarray (structured or homogeneous), Iterable, dict, or DataFrame
pd.DataFrame(data, index=['row1','row2'], columns=['col1','col2','col3'])
pd.DataFrame(data, index=range(len(data)))
data = [1,2,3,4,5]
pd.DataFrame(Counter(data).elements()) # dict
pd.DataFrame(Counter(data).values()) # dict
pd.DataFrame(range(5)) # Iterable
pd.DataFrame(permutations(data)) # Iterable
```

1.3 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
  )
```

2 Attributes and Methods

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

2.1 Descriptive Statistics

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

2.2 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
```

3 Displaying Datasets

3.1 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
len(df) # this actually works and gives you the number of rows
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
```

3.2 Displaying sub-dataframe by Indexing

```
df[['col1','col2']] # display a sub-dataframe of two columns
df[['col1']] # display a sub-dataframe of one column
df['col1'] # this returns a Series instead of a dataframe; a Series in Pandas has attributes 'name' (instead of '
    columns'), 'index' and 'values'; Series has no 'columns' attribute; df['col1']+=*/**//df['col2'] is
    elementwise, and the result is still a Series; for calculations with vector, e.g, inner and outer product, we
    need to use 'df.values' which is an array, e.g., df['col1'].values @ df['col2'].values

df.iloc[:,[0]] # integer-location based indexing; df.iloc() is wrong
df.iloc[:,0] # similarly, this returns a Series instead of a dataframe
```

```python
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

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.to_frame() # convert Series to DataFrame
pd.DataFrame(series) # convert Series to DataFrame, more general
pd.DataFrame([scalar]) # convert scalar to DataFrame, more general
```

3.3 Displaying sub-dataframe by Filtering Conditions

```python
df[df['salary']<=70000].iloc[:,1:3] # if you use R, you might have intuitively tried df.iloc[df['salary
    ']<=70000],1:3]. This is wrong in pandas
df[df['name'].isin(['Joe','Henry','Delta'])] # df['name'] in (['Joe','Henry','Delta']) is wrong

df['col1'].iloc[0] # a scalar; since df['col1'] is a Series, iloc only has one dimension; same as df[['col1']].
    iloc[0,0]

# for Series, operator must be &,| instead of and,or, and each condition must be in parenthesis ()
df[(df['col'] < 1000) | (df['col2'].isna())] # df[df['col'] < 1000 or df['col2'].isna()] is wrong

df[df['col1'].between(val1,val2)] # between

# filtering strings
df['col1'].str.startswith('abc')
df['col1'].str.endswith('abc')
df['col1'].str.contains('abc')

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('数')]
```

4 Modifying Dataframes

4.1 Changing Column Names

```python
df.rename(columns={'old_name': 'new_name'}) # change a small number of column names

# changing a large number of column names
# 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  # 要改的列数: 第0~(2-1)列
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
```

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

4.2 Modifying Values

```python
# 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.
```

4.3 Sort

```python
df.sort_values(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.
  axis=0, # "{0 or 'index', 1 or 'columns'}", default 0. Axis to be sorted.
  ascending=True, # bool or list of bool, default True. Sort ascending vs. descending. Specify list for multiple
      sort orders. If this is a list of bools, must match the length of the by.
  inplace=False # bool, default False. If True, perform operation in-place.
  na_position='last', # {'first', 'last'}, default 'last'. Puts NaNs at the beginning if first; last puts
      NaNs at the end.
)
```

4.4 Adding New Columns

4.4.1 Assign

`assign()` is a method for adding new columns to a DataFrame in a functional, chainable way. Unlike direct assignment (`df['col'] = value`), `assign()` returns a new DataFrame without modifying the original.

```python
df.assign(new_col1 = value1, new_col2 = value2, ...) # create new columns with identical values
df3 = df.assign(new_col1=df['col1']*12, new_col2=df['new_col1']/1000, new_col3=df['new_col2']>60000, ...) # you
    can create new columns using other new columns

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
) # Creating New Columns Based on Anonymous Funcitons
```

4.4.2 Advanced Methods

```python
# 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'])

# Custom string Grouping with map()
```

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

4.5 Dropping NAs/Duplicates

```python
df.dropna(axis=0, # {0 or 'index', 1 or 'columns'}, default 0; =0 or 'index': drop rows which contain missing
    values; =1 or 'columns': drop columns which contain missing values; only a single axis is allowed.
  how='any', # {'any', 'all'}, default 'any'
  subset=None, # default None; Labels along other axis to consider, e.g. if you are dropping rows these would be a
      list of columns to include.
  inplace=False, # bool, default False; Whether to modify the DataFrame rather than creating a new one.
)
df.dropna() # return a dataframe that drops all rows that contain at least one NA

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 # bool, default False; Whether to modify the DataFrame rather than creating a new one.
)
df.drop_duplicates() # drop all duplicates except for the first occurrence; a duplicate is define by having the
    same row across all columns
```

4.6 Displaying NAs/Duplicates (Advanced)

```python
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 '*'

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
```

4.7 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'
```

5 Groupby

```
# df.grouby(...) returns a GroupBy object, which is different from DataFrame and Series
df.grouby(['col1','col2'])['col3'].agg('min') # group by ('col1','col2') and calculate min for 'col3' for each
    group; this returns a Series with name 'col3' and index ['col1','col2']
df.grouby(['col1','col2'])['col3'].agg('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']].agg('min') # note that [['col3']] makes the result a DataFrame instead of a
    Series.
df.grouby(['col1','col2'])['col3'].agg(['min']) # this changes the name 'col3' into 'min', so it is slightly
    different from .agg(['min'])
df.grouby(['col1','col2'])['col3'].min() # indeed, if we only want one descriptive statistic, we don't have to use
     .agg() method

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
])

df.groupby('col1').agg(new_colname1 = ('col1','min'), new_colname2 = ('col2','max'), ...)
df.groupby('col1')['col2'].agg([('new_col1','min'), ('new_col2','max'), ...])

# one can customize functions in agg()
df.groupby('category')['value'].agg(null_count=lambda x: x.isna().sum()) # here x represent each group, which is a
     Series in this case (['value'] instead of [['value']]); x.isna() is the same class as x, where every entry is
     set True for NA and False otherwise; sum()=sum(axis=0) by default, per column across rows

sales.groupby('region')['revenue'].agg([
'mean',
('25th_percentile', lambda x: x.quantile(0.25)),
('median', 'median'),
('75th_percentile', lambda x: x.quantile(0.75))
])
```

6 Merge

```
pd.merge(left, right, how='inner', on=['key1', 'key2'])
pd.merge(left, right, on=['key1', 'key2']) # how='inner' is default
pd.merge(left, right, how='outer', on=['key1', 'key2'])
```

```python
pd.merge(left, right, how='right', on=['key1', 'key2'])
pd.merge(left, right, on='k', suffixes=['_l', '_r'])
left.merge(right, how='inner', on=None, left_on=None, right_on=None, left_index=False, right_index=False, sort=
    False, suffixes=('_x', '_y'), copy=None, indicator=False, validate=None)
```

7 Pivit Table

```python
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
)
```

7.1 Naming Rule for Columns by Pivit Tables

```python
# 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',...), ...]
```

8 Time Data

8.1 Creating Time Data

```python
date = pd.to_datetime('2015-01-25')
```

8.2 Displaying Time Information

```python
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 Converting Time Unit

```
df['col1'].dt.total_seconds() # convert the time into seconds
```

8.4 Calculating Times

```
df['time_col'].iloc[1] - df['time_col'].iloc[0] == pd.Timedelta(days=1) # Available kwargs: {days, seconds,
    microseconds, milliseconds, minutes, hours, weeks}. Values for construction in compat with datetime.timedelta.
     Numpy ints and floats will be coerced to python ints and floats. years/months does not work because Timedelta
     represents a fixed duration, so it can't handle variable-length periods

pd.to_datetime('2015-01-30') + pd.DateOffset(months=1) # returns Timestamp('2015-02-28 00:00:00'), automatically
    offset backwards; Available kwargs: {years, months, days, seconds, microseconds, milliseconds, nanoseconds,
    minutes, hours, weeks}.
```

8.5 Grouper

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

9 Other Functions/Methods

```
isinstance(n, int) and n > 0 # check if n is a natural number
f'row_{i}' # makes i changeable
```