Introduction

Pandas is a package built on top of NumPy, and provides an efficient implementation of many features :
- DataFrames
- Series
- Data Alignement
- Handling Missing Data
- Grouping and Aggregation
- Data Input and Output
- Handling Time Series

Pandas General Methods

Accessing values	pd_ds.values	DataFrame.values Series.values
Accessing Indices	pd_ds.index	DataFrame.index Series.index
Accessing specific element	pd_ds[idx]	DataFrame[1] Series[1]
Accessing range of elements	pd_ds[start : end]	DataFrame[1:4] Series[2:5]
Implicit Indexing	df.iloc[rows , cols]	data.iloc[1:3] #last index is not included
Explicit Indexing	df.loc[rows,cols]	data.loc['California' : 'Texas'] #last index is included

Pandas Series

Creating Series with lists	pd.Series([values], index= list)	data = pd.Series([0.25, 0.5, 0.75, 1.0], index=['a', 'b', 'c', 'd'])
Creating Series with dictionaries	pd.Series({index:value})	population_dict = {'California': 38332521, 'Texas': 26448193, 'New York': 19651127, 'Florida': 19552860, 'Illinois': 12882135} population = pd.Series(population_dict)
Slicing Series	Series[from_idx : to_idx]	population['Texas':'Florida']
Slicing Indices with Dictionary Series	pd.Series({index:value}, index=[])	pd.Series({2:'a', 1:'b', 3:'c'}, index=[3, 2]) # only returns the third and second index respectfully

Pandas Series Index Can be a list of string or list of integers (or any desired type) unlike numpy arrays

Pandas DataFrames

Creating DataFrame	pd.DataFrame({index : iterable})	pd.DataFrame({'population': population, 'area': area})

Pandas DataFrames (cont)

Adding Column names	pd.DataFrame(dict , columns = [list_of_col-_names])	pd.DataFrame(population, columns=['population'])
Slicing DataFrame Index	pd.DataFrame(dict , columns = [ ] , index = [ ])	pd.DataFrame(np.random.rand(3, 2), columns=['foo', 'bar'], index=-['a', 'b', 'c'])
Reading CSV Files	pd.read_csv(source , index_col = col)	pd.read_csv("data/president_heights.csv", index_col="order")
Saving DF to CSV	dataframe.to_csv(source)	df.to_csv("data/president_heights_copy.csv")
Reading Excel Files	pd.read_excel(source)	pd.read_excel("data/president_heights.xlsx")
Saving DF to Excel	dataframe.to_excel(source)	df.to_excel("data/president_heights_copy.xlsx")
Access DataFrame Columns	dataframe.columns	df.columns
Transposing DataFrames	dataframe.T	df.T
Subsetting Using loc	dataframe.loc[condition , cols]	data.loc[data.density > 100, ['pop', 'density']]
Masking	dataframe[mask]	data[data.density > 100]

Pandas Index

Creating Index	pd.Index(list)	pd.Index([2, 3, 5, 7, 11])
Accessing Index	Index[idx]	ind[1]
Slicing Index	Index[from : to : step]	ind[: : 2]
Intersection Between Indices	index_1.intersection(index_2)	indA = pd.Index([1, 3, 5, 7, 9]) indB = pd.Index([2, 3, 5, 7, 11]) indA.intersection(indB)
Union Between Indices	index_1.union(index_2)	indA.union(indB)
Symmetric Difference	index_1. symmetric_difference(index_2)	indA.symmetric_difference(indB)

The **Index** has many of the attributes familiar from NumPy arrays such as :
ind.size, ind.shape, ind.ndim, ind.dtype

Pandas Universal Functions

+	add()
-	sub() , subtract()
*	mul(), multiply()
/	truediv(), div(), divide()
//	floordiv()
%	mod()

Pandas Universal Functions (cont)

**	pow()

These universal functions are used in the following form :
- data_struct.uf(data_struct_2)
- data_struct.uf()

Datatype Conversions (NaN or None)

Float	No change
Object	No change
Integer	Upcast to float64
Boolean	Upcast to object

These are data type conversion when there is missing values

Operating On Missing Values

Nullability Check	data_struc.is_null()	data = pd.Series([1, np.nan, 'hello', None]) data.isnull()
Non-Nullability Check	data_struc.not_null()	data.not_null()
Slicing Non-Null Values	data_struct[data_struc.not_null()]	data[data.not_null()]
Dropping Null Values	data_struct.dropna(axis=0/1 , how = 'any'/'all' , thresh = n)	data.dropna(axis = 0 , thresh = 2) # the tresh means each row has at least 2 non-null values
Filling Missing Values	data_struct.fillna(value , method = 'ffil'/'bfill' , axis = 0/1)	df.fillna(method='ffill', axis=1)
Filling Using A Function (Interpolation)	data_struct.interpolate(method='linear'/'polynomial'...)	df.interpolate() # the method is linear by default

When Working with missing values methods , axis = 0 means rows and 1 columns

Pandas Multi-Indexing

Creating Multi-Index From Tuples	pd.MultiIndex.from_tuples(tuple)	index = pd.MultiIndex.from_tuples([('California', 2000), ('California', 2010)])
Creating Multi-Index From Arrays	pd.MultiIndex.from_arrays(list)	pd.MultiIndex.from_arrays([['a', 'a', 'b', 'b'], [1, 2, 1, 2]])
Creating Multi-Index From Product	pd.MultiIndex.from_product([index1_list,index2_list])	pd.MultiIndex.from_product([['a', 'b'], [1, 2]])
Creating Multi-Index From DataFrame Values	pd.MultiIndex.from_frame(dataframe)	df = pd.DataFrame([['a', 'b'], [1, 2]]) pd.MultiIndex.from_frame(df)
Applying Muli-Index	data_struct.reindex(index)	pop = pop.reindex(index)

Pandas Multi-Indexing (cont)		
Setting Index From Columns	data_struct.set_index([cols])	pop_flat.set_index(['population'])
Accessing Multi-Indexed Data Structures	data_struct[first_index,second_index,....., col]	pop[:, 2010] # gets all rows from first index and only 2010 rows from second index
Unstacking	data_struct.unstack()	pop.unstack() # this converts the last index (if we have 2 then the second one) values into cols
Stacking	data_struct.stack()	pop.stack() # this converts columns into a second index
Naming Multi-Indexes	data_struct.index.names = list	pop.index.names = ['state', 'year']
Swapping Multi-Indexes	data_struct.swaplevel(0,1)	pop_df = pop_df.swaplevel(0,1)
Dropping Multi-Indexes	data_struct.droplevel(level=index)	pop_df.droplevel(level=0)
Multi-Index In Columns	pd.DataFrame(data, index=multi_index_-rows, columns=multi_indx_cols)	columns = pd.MultiIndex.from_product([['Bob', 'Guido', 'Sue'], ['HR', 'Temp']], names=['subject', 'type']) health_data = pd.DataFrame(data, index=index, columns=c-olumns)
Slicing Using Multi-Index Column Values	dataframe[multi_ind_col_value]	health_data['Guido']
Slicing Multi-Index Cols & Rows Using IndexSlice	idx = pd.IndexSlice df.loc[idx[index_row1,index_row2], idx[in-dex_col1,index_col2]]	idx = pd.IndexSlice health_data.loc[idx[:, 1], idx[:, 'HR']]
Resetting Multi-Index to Cols	data_struct.reset_index()	pop.reset_index(name='population')
Sorting Multi-Index	data_struct.sort_index()	data.sort_index()

It is a good practice to sort the values after swapping Multi-index Levels

Concatenation , Merging and Joins		
Concatenation	pd.concat([data_struc , data_struct2] , ignore_index = True/False)	pd.concat([ser1, ser2])
Adding MultiIndex Keys	pd.concat([data_struc , data_struct2] , keys = ['a','b'])	display('x', 'y', "pd.concat([x, y], keys=['x', 'y'])")

Concatenation , Merging and Joins (cont)		
Concatenation with Joins	pd.concat([data_struc , data_struct2] , join = 'outer'/'-inner')	pd.concat([df5, df6], join='inner') # The intersection of 2 DFs
Merging	pd.merge(data_struc , data_struct2)	df3 = pd.merge(df1, df2)
Merging on Columns	pd.merge(data_struc , data_struct2 , on ='col_name')	pd.merge(df1, df2, on='employee')
Specific Merging	pd.merge(data_struc , data_struct2 , right_on ='col_-name' , left_on ='col_name')	pd.merge(df1, df3, left_on="employee", right_on="nam-e").drop('name', axis=1) # When using right and left on we have to drop one of the cols (avoid redundancy)
Joining (Default Merge to indices only)	data_struct.join(data_struct2)	df1a.join(df2a)
Merging on Indices	pd.merge(data_struc , data_struct2 , left_index=True , right_index = True)	pd.merge(df1a, df3, left_index=True, right_on='name')
Merging with methods	pd.merge(data_struct, data_struct2, how='inner'/'ou-ter'/'left'/'right')	pd.merge(df6, df7, how='inner')
Merging Conflicting Col Names	pd.merge(data_struct, data_struct2, suffixes = ['_suff1', _'suff2'])	pd.merge(df8, df9, on="name", suffixes=["_Sem1", "_Se-m_2"])

- Note that when adding multi-index keys in a concatenation , the number of keys should be the same as the number of data structures being concatenated

Advanced Group By Methods		
Aggregation using a list (General)	df.groupby('col').aggregate([list_of_methods]	df.groupby('key').aggregate(['min', np.median, max])
Aggregation using a dict (Specific)	df.groupby('col').aggregate({'col' : 'method' , 'col2' : 'method'})	df.groupby('key').aggregate({'data1': 'min', 'data2': 'max'})
Filtering	df.groupby('col').filter(func)	def filter_func(x): return x['data2'].std() > 4 df.groupby('key').filter(filter_func)
Transformation	df.groupby('col').transform(lambda_func)	df.groupby('key').transform(lambda x: x - x.mean())

Advanced Group By Methods (cont)		
Apply	df.groupby('col').apply(user_func)	def norm_by_data2(x): # x is a DataFrame of group values x['data1'] /= x['data2'].sum() return x df.groupby('key').apply(norm_by_data2)
Grouping By Custom Mapping	df.groupby(mapping).method	mapping = {'A': 'vowel', 'B': 'consonant', 'C': 'consonant'} df2.groupby(mapping).sum()

aggregate() : this method allows using more than one function with groupby for different columns
filter() : this method allows user-defined filter functions to be applied with a groupby (uses boolean operations only)
transform(): mostly uses lambda functions to returned new and transformed version of a columns
apply(): this method allows you to apply arbitrary user-defined functions with groupby

By **Taissir Boukrouba**
(taissir2002)

cheatography.com/taissir2002/

Not published yet.
Last updated 19th November, 2023.