PANDAS CHEAT SHEET - "The Poster of Pandas!"

		ITERATIVE	MAPPED	VECTORIZED	NOTES
Arithmetic	Basic operations	`for i, row in test_df.iterrows():` `  test_df.loc[i, "int_col"] += 5`	`test_df["dest_col"] = test_df["int_col"].apply(lambda x: x + 5)` `test_df["dest_col"] = test_df["int_col"].map(lambda x: x + 5)`	`test_df["dest_col"] = test_df["dest_col"] + 5` `test_df["dest_col"] += 5`	

	Common functions
	(floor / ceil / round / power, etc)
		`for i, row in test_df.iterrows():`
		`  test_df.loc[i, "dest_col"] = (`
		`  math.floor(test_df.loc[i, "float_col"]))`
			`test_df["dest_col"] = test_df["float_col"].apply(lambda x: math.ceil(x))`
			
			`test_df["dest_col"] = test_df["float_col"].apply(math.floor)`
			`#EX 2 :`
			`test_df["dest_col"] = test_df["int_col"].apply(lambda x: math.pow(x, 2))`
				`test_df["dest_col"] = np.round(df["float_col"], 2)`
				
				`test_df["dest_col"] = np.ceil(df["int_col"])`
				`#ex 2`
				`test_df["dest_col"] = test_df["int_col"] ** 2`	

	Simple aggregations	`sum_val = 0`
		`average_val = 0.0`
		`min_val = None`
		
		`for i, row in test_df.iterrows():`
		`  sum_val += row["int_col_1"]`
		`  average_val = float(sum_val) / float(i)`
		`  if min_val is None or row["int_col_1"] < min_val:`
		`  min_val = row["int_col_1"]`
			`# Since these form of aggregation functions fundamentally have cross-`
			`# row dependencies, there is no real map() / apply() equivalent,`
			`# without extremely contrived examples (E.g. using a lambda function`
			`#  that references a 'global' variable external to it. )`
				`min_val = df["int_col_1"].min()`
				`sum_val = df["int_col_1"].sum()`
				`average_val = df["int_col_1"].mean()`
				
				`#You can also get a series of mins / maxes /etc across multiple`
				`#  columns by df[["col1", "col2", "col3"]].min(), which will return`
				`#  a series of 3 values - the min of each column listed.  Calling min()`
				`# again would result in the min of those 3 values.`
					* Most of the vectorized functions take common optional arguments:
					- axis - 0 or 1 to indicate "Across rows (1) or columns (0, default)"
					- skipna (bool to skip NaN values) (default True)
					- numeric_only

String Operations	Capitalize, replace, etc.	`for i, row in test_df.iterrows():`
		`  test_df.loc[i, "dest_name"] = (`
		`  test_df[i, "str_name"].capitalize()))`
			`test_df["dest_col"] = test_df["str_col"].apply(lambda x: x.capitalize())`
			`# Ex 2`
			`test_df["dest_col"] = (`
			`  test_df["str_col"].apply(lambda x: x.replace("orig", "new_text")))`
				`test_df["dest_col"] = test_df["str_col"].str.capitalize()`
				
				`test_df["dest_col"] = test_df["str_col"].str.replace("orig", "new_text")`
					* Replace can also take a dictionary input, or use regexes
					* You can call replace() on the dataframe itself to replace ALL instances of a value across any columns

Boolean Operations	Find rows where col < 5	`matching_rows = list()`
		`for i, row in test_df.iterrows():`
		`  if (test_df.loc[i, "int_col"]  <  5):`
		`  matching_rows.append(i)`
			`test_df["matches"] = test_df["int_col"].apply(lambda x: True`
			`  if x < 5 else False)`
				`bool_map_matches = (df["int_col"] < 5)`	

	Find rows where Col A >10 and Col B < 5	`matching_rows = list()`
		`for i, row in test_df.iterrows():`
		`  if (row["a"]  >  10 and row["b"] < 5):`
		`  matching_rows.append(i)`
			`test_df["matches"] = test_df.apply(`
			`  lambda row: True if row["a"] > 10`
			`  and row["b"] < 5 else False,`
			`  axis=1)`
				`bool_map_matches = ((df["a"] > 10) & (df["b"] < 5) )`	* Boolean maps can be combined via boolean operators such as &, |, ~
					* "axis=" tells apply() to operate on columns (default / 0), or rows (axis=1). When operating on a row, all columns in the row are acessable

	Boolean aggregations (any.... All ... in set... etc.)	`#Find any rows with integer values > 5 in one of 3 columns`
		`#Similar variants for All > 5 etc.`
		`matching_row_nums = list()`
		
		`for i, row in df.iterrows():`
		`  if row["int_c1"] > 5 or row["int_c2"] > 5 or row["int_c3"] > 5:`
		`  matching_row_nums.append(i)`
		
		`#Ex 2 - Check if a value is contained in a set`
		`rows_w_val_in_set = list()`
		
		`for i, row in df.iterrows():`
		`  if row["val_to_check"] in list_vals_to_match:`
		`  rows_w_val_in_set.append(i)`
			`#Ex 1 - Any Greater than 5`
			`df["any_gt_5"] = df.apply(lambda row: any(row[col] > 5`
			`  for col in ["int_c1", "int_c2", "int_c3"]),`
			`  axis=1)`
			
			`#Ex2`
			`df["val_in_set"] = \`
			`  df["val_to_check"].apply(lambda x: x in list_vals_to_match)`
				`#Any value in row > 5`
				`df["any_gt_5"] = (df[["int_c1", "int_c2", "int_c3"]] > 5).any(axis=1)`
				
				`#Value is in set (using isin)`
				`df["val_in_set"] = df["val_to_check"].isin(list_vals_to_match)`
				
				`#BONUS:  Between values (inclusive)`
				`df["val_between_3_7"] = df["val_to_check"].between(3, 7)`
				`#Equivalent to (df["val_to_check"] >= 3) & (df["val_to_check"] <= 7)`
					* between() includes the endpoints by default, add "inclusive=False" to extcude them.

Conditional Updates	Perform a conditional calc	`for i, row in test_df.iterrows():`
		`  if (test_df.at[i, "int_col"]  <  5):`
		`  test_df.at[i, "dest_col"] = 0`
		`  else:`
		`  test_df.at[i, "dest_col"] = test_df.loc[i, "int_col"]`
			`test_df["dest_col"] = test_df["int_col"].apply(lambda x: 0`
			`  if x < 5 else x)`
				`test_df.loc[test_df["int_col"] < 5, "dest_col"] = 0`
				`test_df.loc[test_df["int_col"] >= 5, "dest_col"] = (`
				`  test_df.loc[test_df["int_col"] >= 5, "int_col"])`
				`# OR, EVEN BETTER:`
				`test_df["dest_col"] = np.where(test_df["int_col"] < 5, 0,`
				`  test_df["int_col"])`	

	Conditional calc with multiple replacement values	`for i, row in test_df.iterrows():`
		`  if row["int_col"] < 25:`
		`  test_df.at[i, "dest_col"] = "Low"`
		`  elif row["int_col"] < 50:`
		`  test_df.at[i, "dest_col"] = "Medium"`
		`  elif row["int_col"] < 75:`
		`  test_df.at[i, "dest_col"] = "High"`
		`  else:`
		`  test_df.at[i, "dest_col"] = "Critical"`
			`def categorize(x):`
			`  if x < 25:`
			`  return "Low"`
			`  elif x < 50:`
			`  return "Medium"`
			`  elif x < 75:`
			`  return "High"`
			`  else:`
			`  return "Critical"`
			
			`df["dest_col"] = df["int_col"].apply(categorize)`
				`conditions = [`
				`  df["int_col"] < 25,`
				`  (df["int_col"] >= 25) & (df["int_col"] < 50),`
				`  (df["int_col"] >= 50) & (df["int_col"] < 75),`
				`  df["int_col"] >= 75`
				`]`
				`choices = ["Low", "Medium", "High", "Critical"]`
				`df["dest_col"] = np.select(conditions, choices, default='Unknown')`
					* Select() takes a list of boolean maps as inputs 1, and a list of replacement values as input 2, and returns a column with values subbed in for matching rows.

	Conditional calc with multiple replacement values - example 2	`#Given a dictionary, severity_weights, mapping text keys  (e.g.`
		`#"High", "Medium") to numeric values (e.g 4, 3 ) called`
		`# 'severity_weight_dict'`
		`for i, row in df.iterrows():`
		`  severity_text = row["severity"]`
		`  df.at[i, "severity_weight"] = severity_weight_dict[severity_text]`
			`#Given a dictionary, severity_weights, mapping text keys  (e.g.`
			`#"High", "Medium") to numeric values (e.g 4, 3 ) called`
			`# 'severity_weight_dict'`
			
			`df["severity_weight"] = df["severity"].apply(  \`
			`  lambda x: severity_weight_dict[x])`
				`#Given a dictionary, severity_weights, mapping text keys  (e.g.`
				`#"High", "Medium") to numeric values (e.g 4, 3 ) called`
				`# 'severity_weight_dict'`
				
				`df["sev_weight"] = df["sev_text"] replace(severity_weight_dict)`
					* replace can operate on a whole row or column, a whole dataframe, or anything in between.

Aggregation / grouping	Create sums by a grouped value - similar to a database GROUP BY	`# Below is a little contrived as it's assuming we want to end up with a`
		`# dataframe in the end- might be more natural to update a dictionary.`
		`issue_types = ["DataIssue", "ConfigIssue", "LogicIssue"]`
		`result_df = pd.DataFrame("issue_type": issue_types,`
		`  "num_issues": [0, 0, 0])`
		`for index, row in test_df.iterrows():`
		`  result_df.loc[result_df["issue_type"] == row["issue_type"],`
		`  "num_issues"] += 1`
			`#Any examples here are fairly contrived and fundamentally fall`
			`# through to a vecroeized approach under the hood. - e.g.`
			`#`
			`#def sum_group(group_df):`
			`#  return group_df['value'].sum()`
			`#`
			`#result_df = (`
			`  df.groupby('group').apply(sum_group).reset_index(name='sum'))`
				`result_df = df.groupby("risk_label")["risk_value"].sum()`
				
				`# example 2:`
				`result_df2 = df.groupby("issue_type").count()`
				
				`# Example 3:`
				`df.groupby("department").agg("Total_Issues="issue_id", "count"),`
				`  Agg_Risk_Rating=("risk_value", "sum"))`
					* Great Ex: : https://www.geeksforgeeks.org/python-pandas-dataframe-groupby/
					* Group by multiple columns as well - groupby([col1, col2])
					* Common groupby sub-functions:
					- sum, mean, min, max
					- count
					- agg - multiple aggregations on different columns, as specified in inputs
					- first - returns the first item in the group

	Filter a grouped DF	`#See Note Above`	`#See Note Above`	`# Rows for teams where the sum of risk_val of those rows would be >= 1000`
				
				`result_df = (`
				`  df.groupby("team").filter(lambda x: x["risk_value"].sum() >= 1000))`
					You can follow up with a filter() or transform() and filter() to gte similar results to SQLs GROUP BY ... HAVING

Filtering / Subframes	Subframe with only selected columns	`#Iterative approaches to this problem only make sense if you're selecting`
		`# columns based on calculations, etc. - e.g. something like`
		`risk_cols = [col for col in df.columns if col.startswith('risk_')]`
		`new_df = df[risk_cols]`
			`# There is no real "apply()" or map() equivalent.  You can use functions`
			`# like filter() in a similar fashion though:`
			`new_df = df.filter(regex='^risk_')`
				`#Make a dataframe with only columns int_c1, int_c2, and float_c1`
				`#  from the source dataframe df`
				
				`sub_df = df[["int_c1", "int_c2", "int_c3", "float_c1"]]`	

	Taking a Sub-frame with only rows matching certain conditions	`# Assume we have a list result() that we're using`
		`for i, row in test_df.iterrows():`
		`  if 20 <= row["int_col1"] <= 50 and row["int_col2"] > 70:`
		`  result.append(row)`
		`filtered_df = pd.DataFrame(result)`
			`# This is pretty contrived, but…`
			`filtered_df = test_df[test_df.apply(lambda row: 20 <= row["int_col1"] <= 50`
			`  and row["int_col2"] > 70,`
			`  axis=1)]`
				`mask = ((df["col1"] >= 20) & (df["col1"] <= 50) & (df["int_col2"] > 70))`
				`filtered_df = test_df[mask]`
				`# Or one of the below (each slightly slower but more readable)`
				`new_df = df[(df["col1"].between(20, 50)) & (df["int_col2"] > 70)]`
				`filtered_df = df.query("20 <= int_col1 <= 50 and int_col2 > 70")`
					* Basically, passing a boolean series mask to a dataframe's [] gives a filtered dataframe containing the rows where the boolean mask was True