dfply Documentation

Release 0.3

Kiefer Katovich

Aug 28, 2018

Contents:

CHAPTER 1

Sphinx AutoAPI Index

This page is the top-level of your generated API documentation. Below is a list of all items that are documented here.

1.1 dfply

1.1.1 Subpackages

dfply.data

Package Contents

dfply.data.**root**

dfply.data.**diamonds**

1.1.2 Submodules

dfply.base

Module Contents

dfply.base.**_recursive_apply**(*f*, *l*)

dfply.base.**contextualize**(*arg*, *context*)

dfply.base.**flatten**(*l*)

dfply.base.**_check_delayed_eval**(*args*, *kwargs*)

dfply.base.**_context_args**(*args*)

dfply.base.**_context_kwargs**(*kwargs*)

dfply.base.**_delayed_function**(*function*, *args*, *kwargs*)

dfply.base.**make_symbolic**(*f*)

class dfply.base.**Intention**(*function=lambda x: x*, *invert=False*)
Bases:object

evaluate(*self*, *context*)

__getattr__(*self*, *attribute*)

__invert__(*self*)

__call__(*self*, **args*, ***kwargs*)

dfply.base.**_magic_method_names = ['__abs__', '__add__', '__and__', '__cmp__', '__complex__**

dfply.base.**_set_magic_method**(*name*)

dfply.base.**X**

class dfply.base.**pipe**(*function*)
Bases:object

__name__ = pipe

__rshift__(*self*, *other*)

__rrshift__(*self*, *other*)

__call__(*self*, **args*, ***kwargs*)

class dfply.base.**IntentionEvaluator**(*function*, *eval_symbols=True*, *eval_as_label=[]*, *eval_as_selector=[]*)
Bases:object

Parent class for symbolic argument decorators. Default behavior is to recursively turn the arguments and keyword arguments of a decorated function into *symbolic.Call* objects that can be evaluated against a pandas DataFrame as it comes down a pipe.

__name__ = IntentionEvaluator

_evaluate(*self*, *df*, *arg*)

_evaluate_label(*self*, *df*, *arg*)

_evaluate_selector(*self*, *df*, *arg*)

_evaluator_loop(*self*, *df*, *arg*, *eval_func*)

_symbolic_eval(*self*, *df*, *arg*)

_symbolic_to_label(*self*, *df*, *arg*)

_symbolic_to_selector(*self*, *df*, *arg*)

_recursive_arg_eval(*self*, *df*, *args*)

_recursive_kwarg_eval(*self*, *df*, *kwargs*)

_find_eval_args(*self*, *request*, *args*)

_find_eval_kwargs(*self*, *request*, *kwargs*)

__call__(*self*, **args*, ***kwargs*)

dfply.base.**symbolic_evaluation**(*function=None*, *eval_symbols=True*, *eval_as_label=[]*, *eval_as_selector=[]*)

class dfply.base.**group_delegation**(*function*)
Bases:object

__name__ = group_delegation

_apply(*self*, *df*, **args*, ***kwargs*)

__call__(*self*, **args*, ***kwargs*)

dfply.base.**dfpipe**(*f*)

dfply.group

Module Contents

dfply.group.**group_by**(*df*, **args*)

dfply.group.**ungroup**(*df*)

dfply.join

Module Contents

dfply.join.**get_join_parameters**(*join_kwargs*)
Convenience function to determine the columns to join the right and left DataFrames on, as well as any suffixes for the columns.

dfply.join.**inner_join**(*df*, *other*, ***kwargs*)
Joins on values present in both DataFrames.

Args: df (pandas.DataFrame): Left DataFrame (passed in via pipe) other (pandas.DataFrame): Right DataFrame

Kwargs:

by (str or list): Columns to join on. If a single string, will join on that column. If a list of lists which contain strings or integers, the right/left columns to join on.

suffixes (list): String suffixes to append to column names in left and right DataFrames.

Example: a >> inner_join(b, by='x1')

x1 x2 x3

0 A 1 True 1 B 2 False

dfply.join.**full_join**(*df*, *other*, ***kwargs*)
Joins on values present in either DataFrame. (Alternate to *outer_join*)

Args: df (pandas.DataFrame): Left DataFrame (passed in via pipe) other (pandas.DataFrame): Right DataFrame

Kwargs:

by (str or list): Columns to join on. If a single string, will join on that column. If a list of lists which contain strings or integers, the right/left columns to join on.

suffixes (list): String suffixes to append to column names in left and right DataFrames.

Example: a >> outer_join(b, by='x1')

x1 x2 x3

0 A 1.0 True 1 B 2.0 False 2 C 3.0 NaN 3 D NaN True

dfply.join.**outer_join**(*df*, *other*, ***kwargs*)

Joins on values present in either DataFrame. (Alternate to *full_join*)

Args: df (pandas.DataFrame): Left DataFrame (passed in via pipe) other (pandas.DataFrame): Right DataFrame

Kwargs:

by (str or list): Columns to join on. If a single string, will join on that column. If a list of lists which contain strings or integers, the right/left columns to join on.

suffixes (list): String suffixes to append to column names in left and right DataFrames.

Example: a >> full_join(b, by='x1')

x1 x2 x3

0 A 1.0 True 1 B 2.0 False 2 C 3.0 NaN 3 D NaN True

dfply.join.**left_join**(*df*, *other*, ***kwargs*)

Joins on values present in in the left DataFrame.

Args: df (pandas.DataFrame): Left DataFrame (passed in via pipe) other (pandas.DataFrame): Right DataFrame

Kwargs:

by (str or list): Columns to join on. If a single string, will join on that column. If a list of lists which contain strings or integers, the right/left columns to join on.

suffixes (list): String suffixes to append to column names in left and right DataFrames.

Example: a >> left_join(b, by='x1')

x1 x2 x3

0 A 1 True 1 B 2 False 2 C 3 NaN

dfply.join.**right_join**(*df*, *other*, ***kwargs*)

Joins on values present in in the right DataFrame.

Args: df (pandas.DataFrame): Left DataFrame (passed in via pipe) other (pandas.DataFrame): Right DataFrame

Kwargs:

by (str or list): Columns to join on. If a single string, will join on that column. If a list of lists which contain strings or integers, the right/left columns to join on.

suffixes (list): String suffixes to append to column names in left and right DataFrames.

Example: a >> right_join(b, by='x1')

x1 x2 x3

0 A 1.0 True 1 B 2.0 False 2 D NaN True

dfply.join.**semi_join**(*df*, *other*, ***kwargs*)

Returns all of the rows in the left DataFrame that have a match in the right DataFrame.

Args: df (pandas.DataFrame): Left DataFrame (passed in via pipe) other (pandas.DataFrame): Right DataFrame

Kwargs:

by (str or list): Columns to join on. If a single string, will join on that column. If a list of lists which contain strings or integers, the right/left columns to join on.

Example: a >> semi_join(b, by='x1')

x1 x2

0 A 1 1 B 2

dfply.join.**anti_join**(*df*, *other*, ***kwargs*)

Returns all of the rows in the left DataFrame that do not have a match in the right DataFrame.

Args: df (pandas.DataFrame): Left DataFrame (passed in via pipe) other (pandas.DataFrame): Right DataFrame

Kwargs:

by (str or list): Columns to join on. If a single string, will join on that column. If a list of lists which contain strings or integers, the right/left columns to join on.

Example: a >> anti_join(b, by='x1')

x1 x2

2 C 3

dfply.join.**bind_rows**(*df*, *other*, *join='outer'*, *ignore_index=False*)

Binds DataFrames "vertically", stacking them together. This is equivalent to *pd.concat* with *axis=0*.

Args: df (pandas.DataFrame): Top DataFrame (passed in via pipe). other (pandas.DataFrame): Bottom DataFrame.

Kwargs:

join (str): One of *"outer"* or *"inner"*. Outer join will preserve columns not present in both DataFrames, whereas inner joining will drop them.

ignore_index (bool): Indicates whether to consider pandas indices as part of the concatenation (defaults to *False*).

dfply.join.**bind_cols**(*df*, *other*, *join='outer'*, *ignore_index=False*)

Binds DataFrames "horizontally". This is equivalent to *pd.concat* with *axis=1*.

Args: df (pandas.DataFrame): Left DataFrame (passed in via pipe). other (pandas.DataFrame): Right DataFrame.

Kwargs:

join (str): One of *"outer"* or *"inner"*. Outer join will preserve rows not present in both DataFrames, whereas inner joining will drop them.

ignore_index (bool): Indicates whether to consider pandas indices as part of the concatenation (defaults to *False*).

dfply.reshape

Module Contents

dfply.reshape.**arrange**(*df*, **args*, ***kwargs*)

Calls *pandas.DataFrame.sort_values* to sort a DataFrame according to criteria.

See: http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.sort_values.html

For a list of specific keyword arguments for sort_values (which will be the same in arrange).

Args:

***args: Symbolic, string, integer or lists of those types indicating** columns to sort the DataFrame by.

Kwargs:

****kwargs: Any keyword arguments will be passed through to the pandas** *DataFrame.sort_values* function.

dfply.reshape.**rename**(*df*, ***kwargs*)

Renames columns, where keyword argument values are the current names of columns and keys are the new names.

Args: df (`pandas.DataFrame`): DataFrame passed in via >> pipe.

Kwargs:

****kwargs: key:value pairs where keys are new names for columns and** values are current names of columns.

dfply.reshape.**gather**(*df*, *key*, *values*, **args*, ***kwargs*)

Melts the specified columns in your DataFrame into two key:value columns.

Args: key (str): Name of identifier column. values (str): Name of column that will contain values for the key. *args (str, int, symbolic): Columns to "melt" into the new key and

value columns. If no args are specified, all columns are melted into they key and value columns.

Kwargs:

add_id (bool): Boolean value indicating whether to add a *"_ID"* column that will preserve information about the original rows (useful for being able to re-widen the data later).

Example: diamonds >> gather('variable', 'value', ['price', 'depth','x','y','z']) >> head(5)

carat cut color clarity table variable value

0 0.23 Ideal E SI2 55.0 price 326.0 1 0.21 Premium E SI1 61.0 price 326.0 2 0.23 Good E VS1 65.0 price 327.0 3 0.29 Premium I VS2 58.0 price 334.0 4 0.31 Good J SI2 58.0 price 335.0

dfply.reshape.**convert_type**(*df*, *columns*)

Helper function that attempts to convert columns into their appropriate data type.

dfply.reshape.**spread**(*df*, *key*, *values*, *convert=False*)

Transforms a "long" DataFrame into a "wide" format using a key and value column.

If you have a mixed datatype column in your long-format DataFrame then the default behavior is for the spread columns to be of type *object*, or string. If you want to try to convert dtypes when spreading, you can set the convert keyword argument in spread to True.

Args: key (str, int, or symbolic): Label for the key column. values (str, int, or symbolic): Label for the values column.

Kwargs:

convert (bool): Boolean indicating whether or not to try and convert the spread columns to more appropriate data types.

Example: widened = elongated >> spread(X.variable, X.value) widened >> head(5)

_ID carat clarity color cut depth price table x y z

0 0 0.23 SI2 E Ideal 61.5 326 55 3.95 3.98 2.43 1 1 0.21 SI1 E Premium 59.8 326 61 3.89 3.84 2.31 2 10 0.3 SI1 J Good 64 339 55 4.25 4.28 2.73 3 100 0.75 SI1 D Very Good 63.2 2760 56 5.8 5.75 3.65 4 1000 0.75 SI1 D Ideal 62.3 2898 55 5.83 5.8 3.62

dfply.reshape.**separate**(*df*, *column*, *into*, *sep='[\W_]+'*, *remove=True*, *convert=False*, *extra='drop'*, *fill='right'*)

Splits columns into multiple columns.

Args: df (pandas.DataFrame): DataFrame passed in through the pipe. column (str, symbolic): Label of column to split. into (list): List of string names for new columns.

Kwargs:

sep (str or list): If a string, the regex string used to split the column. If a list, a list of integer positions to split strings on.

remove (bool): Boolean indicating whether to remove the original column. convert (bool): Boolean indicating whether the new columns should be

converted to the appropriate type.

extra (str): either *'drop'*, where split pieces beyond the specified new columns are dropped, or *'merge'*, where the final split piece contains the remainder of the original column.

fill (str): either *'right'*, where *np.nan* values are filled in the right-most columns for missing pieces, or *'left'* where *np.nan* values are filled in the left-most columns.

dfply.reshape.**unite**(*df*, *colname*, **args*, ***kwargs*)

Does the inverse of *separate*, joining columns together by a specified separator.

Any columns that are not strings will be converted to strings.

Args: df (pandas.DataFrame): DataFrame passed in through the pipe. colname (str): the name of the new joined column. *args: list of columns to be joined, which can be strings, symbolic, or

integer positions.

Kwargs: sep (str): the string separator to join the columns with. remove (bool): Boolean indicating whether or not to remove the

original columns.

na_action (str): can be one of *'maintain'* (the default), *'ignore'*, or *'as_string'*. The default will make the new column row a *NaN* value if any of the original column cells at that row contained *NaN*. *'ignore'* will treat any *NaN* value as an empty string during joining. *'as_string'* will convert any *NaN* value to the string *'nan'* prior to joining.

dfply.select

Module Contents

dfply.select.**selection_context**(*arg*, *context*)

dfply.select.**selection_filter**(*f*)

dfply.select.**resolve_selection**(*df*, **args*, *drop=False*)

dfply.select.**select**(*df*, **args*)

dfply.select.**drop**(*df*, **args*)

dfply.select.**select_if**(*df*, *fun*)

Selects columns where fun(ction) is true Args:

fun: a function that will be applied to columns

dfply.select.**drop_if**(*df*, *fun*)
Drops columns where fun(ction) is true Args:

fun: a function that will be applied to columns

dfply.select.**starts_with**(*columns*, *prefix*)

dfply.select.**ends_with**(*columns*, *suffix*)

dfply.select.**contains**(*columns*, *substr*)

dfply.select.**matches**(*columns*, *pattern*)

dfply.select.**everything**(*columns*)

dfply.select.**num_range**(*columns*, *prefix*, *range*)

dfply.select.**one_of**(*columns*, *specified*)

dfply.select.**columns_between**(*columns*, *start_col*, *end_col*, *inclusive=True*)

dfply.select.**columns_from**(*columns*, *start_col*)

dfply.select.**columns_to**(*columns*, *end_col*, *inclusive=False*)

dfply.set_ops

Module Contents

dfply.set_ops.**validate_set_ops**(*df*, *other*)
Helper function to ensure that DataFrames are valid for set operations. Columns must be the same name in the same order, and indices must be of the same dimension with the same names.

dfply.set_ops.**union**(*df*, *other*, *index=False*, *keep='first'*)
Returns rows that appear in either DataFrame.

Args: df (pandas.DataFrame): data passed in through the pipe. other (pandas.DataFrame): other DataFrame to use for set operation with

the first.

Kwargs:

index (bool): Boolean indicating whether to consider the pandas index as part of the set operation (default *False*).

keep (str): Indicates which duplicate should be kept. Options are *'first'* and *'last'*.

dfply.set_ops.**intersect**(*df*, *other*, *index=False*, *keep='first'*)
Returns rows that appear in both DataFrames.

Args: df (pandas.DataFrame): data passed in through the pipe. other (pandas.DataFrame): other DataFrame to use for set operation with

the first.

Kwargs:

index (bool): Boolean indicating whether to consider the pandas index as part of the set operation (default *False*).

keep (str): Indicates which duplicate should be kept. Options are *'first'* and *'last'*.

dfply.set_ops.**set_diff**(*df*, *other*, *index=False*, *keep='first'*)
Returns rows that appear in the first DataFrame but not the second.

Args: df (pandas.DataFrame): data passed in through the pipe. other (pandas.DataFrame): other DataFrame to use for set operation with

the first.

Kwargs:

index (bool): Boolean indicating whether to consider the pandas index as part of the set operation (default *False*).

keep (str): Indicates which duplicate should be kept. Options are *'first'* and *'last'*.

dfply.subset

Module Contents

dfply.subset.**head**(*df*, *n=5*)

dfply.subset.**tail**(*df*, *n=5*)

dfply.subset.**sample**(*df*, **args*, ***kwargs*)

dfply.subset.**distinct**(*df*, **args*, ***kwargs*)

dfply.subset.**row_slice**(*df*, *indices*)

dfply.subset.**mask**(*df*, **args*)

dfply.subset.**filter_by**

dfply.subset.**top_n**(*df*, *n=None*, *ascending=True*, *col=None*)

dfply.subset.**pull**(*df*, *column=-1*)

dfply.summarize

Module Contents

dfply.summarize.**summarize**(*df*, ***kwargs*)

dfply.summarize.**summarize_each**(*df*, *functions*, **args*)

dfply.summary_functions

Module Contents

dfply.summary_functions.**mean**(*series*)
Returns the mean of a series.

Args: series (pandas.Series): column to summarize.

dfply.summary_functions.**first**(*series*, *order_by=None*)
Returns the first value of a series.

Args: series (pandas.Series): column to summarize.

Kwargs:

order_by: a pandas.Series or list of series (can be symbolic) to order the input series by before summarization.

dfply.summary_functions.**last**(*series*, *order_by=None*)
Returns the last value of a series.

Args: series (pandas.Series): column to summarize.

Kwargs:

order_by: a pandas.Series or list of series (can be symbolic) to order the input series by before summarization.

dfply.summary_functions.**nth**(*series*, *n*, *order_by=None*)
Returns the nth value of a series.

Args: series (pandas.Series): column to summarize. n (integer): position of desired value. Returns *NaN* if out of range.

Kwargs:

order_by: a pandas.Series or list of series (can be symbolic) to order the input series by before summarization.

dfply.summary_functions.**n**(*series*)
Returns the length of a series.

Args: series (pandas.Series): column to summarize.

dfply.summary_functions.**n_distinct**(*series*)
Returns the number of distinct values in a series.

Args: series (pandas.Series): column to summarize.

dfply.summary_functions.**IQR**(*series*)
Returns the inter-quartile range (IQR) of a series.

The IRQ is defined as the 75th quantile minus the 25th quantile values.

Args: series (pandas.Series): column to summarize.

dfply.summary_functions.**colmin**(*series*)
Returns the minimum value of a series.

Args: series (pandas.Series): column to summarize.

dfply.summary_functions.**colmax**(*series*)
Returns the maximum value of a series.

Args: series (pandas.Series): column to summarize.

dfply.summary_functions.**median**(*series*)
Returns the median value of a series.

Args: series (pandas.Series): column to summarize.

dfply.summary_functions.**var**(*series*)
Returns the variance of values in a series.

Args: series (pandas.Series): column to summarize.

dfply.summary_functions.**sd**(*series*)
Returns the standard deviation of values in a series.

Args: series (pandas.Series): column to summarize.

dfply.transform

Module Contents

dfply.transform.**mutate**(*df*, ***kwargs*)

Creates new variables (columns) in the DataFrame specified by keyword argument pairs, where the key is the column name and the value is the new column value(s).

Args: df (pandas.DataFrame): data passed in through the pipe.

Kwargs:

***kwargs: keys are the names of the new columns, values indicate** what the new column values will be.

Example: diamonds >> mutate(x_plus_y=X.x + X.y) >> select_from('x') >> head(3)

x y z x_plus_y

0 3.95 3.98 2.43 7.93 1 3.89 3.84 2.31 7.73 2 4.05 4.07 2.31 8.12

dfply.transform.**mutate_if**(*df*, *predicate*, *fun*)

Modifies columns in place if the specified predicate is true. Args:

df (pandas.DataFrame): data passed in through the pipe. predicate: a function applied to columns that returns a boolean value fun: a function that will be applied to columns where predicate returns True

Example:

diamonds >> mutate_if(lambda col: min(col) < 1 and mean(col) < 4, lambda row: 2 * row) >> head(3)
carat cut color clarity depth table price x y z

0 0.46 Ideal E SI2 61.5 55.0 326 3.95 3.98 4.86 1 0.42 Premium E SI1 59.8 61.0 326 3.89 3.84 4.62 2 0.46 Good E VS1 56.9 65.0 327 4.05 4.07 4.62 (columns 'carat' and 'z', both having a min < 1 and mean < 4, are doubled, while the other rows remain as they were)

dfply.transform.**transmute**(*df*, **keep_columns*, ***kwargs*)

Creates columns and then returns those new columns and optionally specified original columns from the DataFrame.

This works like *mutate*, but designed to discard the original columns used to create the new ones.

Args:

***keep_columns: Column labels to keep. Can be string, symbolic, or** integer position.

Kwargs:

***kwargs: keys are the names of the new columns, values indicate** what the new column values will be.

Example: diamonds >> transmute(x_plus_y=X.x + X.y, y_div_z=(X.y / X.z)) >> head(3)

y_div_z x_plus_y

0 1.637860 7.93 1 1.662338 7.73 2 1.761905 8.12

dfply.vector

Module Contents

dfply.vector.**order_series_by**(*series*, *order_series*)

Orders one series according to another series, or a list of other series. If a list of other series are specified, ordering is done hierarchically like when a list of columns is supplied to *.sort_values()*.

Args: series (pandas.Series): the pandas Series object to be reordered. order_series: either a pandas Series object or a list of pandas Series

objects. These will be sorted using *.sort_values()* with *ascending=True*, and the new order will be used to reorder the Series supplied in the first argument.

Returns: reordered *pandas.Series* object

dfply.vector.**desc**(*series*)

Mimics the functionality of the R desc function. Essentially inverts a series object to make ascending sort act like descending sort.

Args:

series (pandas.Series): pandas series to be inverted prior to ordering/sorting.

Returns:

inverted *pandas.Series*. The returned series will be numeric (integers), regardless of the type of the original series.

Example:

First group by cut, then find the first value of price when ordering by price ascending, and ordering by price descending using the *desc* function.

diamonds >> group_by(X.cut) >> summarize(carat_low=first(X.price, order_by=X.price),

carat_high=first(X.price, order_by=desc(X.price)))

cut carat_high carat_low

0 Fair 18574 337 1 Good 18788 327 2 Ideal 18806 326 3 Premium 18823 326 4 Very Good 18818 336

dfply.vector.**coalesce**(**series*)

Takes the first non-NaN value in order across the specified series, returning a new series. Mimics the coalesce function in dplyr and SQL.

Args:

***series: Series objects, typically represented in their symbolic form** (like X.series).

Example:

df = pd.DataFrame({ 'a':[1,np.nan,np.nan,np.nan,np.nan], 'b':[2,3,np.nan,np.nan,np.nan], 'c':[np.nan,np.nan,4,5,np.nan], 'd':[6,7,8,9,np.nan]

}) df >> transmute(coal=coalesce(X.a, X.b, X.c, X.d))

coal

0 1 1 3 2 4 3 5 4 np.nan

dfply.vector.**case_when**(**conditions*)

Functions as a switch statement, creating a new series out of logical conditions specified by 2-item lists where the left-hand item is the logical condition and the right-hand item is the value where that condition is true.

Conditions should go from the most specific to the most general. A conditional that appears earlier in the series will "overwrite" one that appears later. Think of it like a series of if-else statements.

The logicals and values of the condition pairs must be all the same length, or length 1. Logicals can be vectors of booleans or a single boolean (*True*, for example, can be the logical statement for the final conditional to catch all remaining.).

Args:

***conditions: Each condition should be a list with two values. The first** value is a boolean or vector of booleans that specify indices in which the condition is met. The second value is a vector of values or single value specifying the outcome where that condition is met.

Example:

df = pd.DataFrame({ 'num':np.arange(16)

}) df >> mutate(strnum=case_when([X.num % 15 == 0, 'fizzbuzz'],

[X.num % 3 == 0, 'fizz'], [X.num % 5 == 0, 'buzz'], [True, X.num.astype(str)]))

num strnum

0 0 fizzbuzz 1 1 1 2 2 2 3 3 fizz 4 4 4 5 5 buzz 6 6 fizz 7 7 7 8 8 8 9 9 fizz 10 10 buzz 11 11 11 12 12 fizz 13 13 13 14 14 14 15 15 fizzbuzz

dfply.vector.**if_else**(*condition*, *when_true*, *otherwise*)

Wraps creation of a series based on if-else conditional logic into a function call.

Provide a boolean vector condition, value(s) when true, and value(s) when false, and a vector will be returned the same length as the conditional vector according to the logical statement.

Args:

condition: A boolean vector representing the condition. This is often a logical statement with a symbolic series.

when_true: A vector the same length as the condition vector or a single value to apply when the condition is *True*.

otherwise: A vector the same length as the condition vector or a single value to apply when the condition is *False*.

Example: df = pd.DataFrame

dfply.vector.**na_if**(*series*, **values*)

If values in a series match a specified value, change them to *np.nan*.

Args: series: Series or vector, often symbolic. *values: Value(s) to convert to *np.nan* in the series.

dfply.window_functions

Module Contents

dfply.window_functions.**lead**(*series*, *i=1*)

Returns a series shifted forward by a value. *NaN* values will be filled in the end.

Same as a call to *series.shift(i)*

Args: series: column to shift forward. i (int): number of positions to shift forward.

dfply.window_functions.**lag**(*series*, *i=1*)
Returns a series shifted backwards by a value. *NaN* values will be filled in the beginning.

Same as a call to *series.shift(-i)*

Args: series: column to shift backward. i (int): number of positions to shift backward.

dfply.window_functions.**between**(*series*, *a*, *b*, *inclusive=False*)
Returns a boolean series specifying whether rows of the input series are between values *a* and *b*.

Args: series: column to compare, typically symbolic. a: value series must be greater than (or equal to if *inclusive=True*)

for the output series to be *True* at that position.

b: value series must be less than (or equal to if *inclusive=True*) for the output series to be *True* at that position.

Kwargs:

inclusive (bool): If *True*, comparison is done with >= and <=. If *False* (the default), comparison uses > and <.

dfply.window_functions.**dense_rank**(*series*, *ascending=True*)
Equivalent to *series.rank(method='dense', ascending=ascending)*.

Args: series: column to rank.

Kwargs: ascending (bool): whether to rank in ascending order (default is *True*).

dfply.window_functions.**min_rank**(*series*, *ascending=True*)
Equivalent to *series.rank(method='min', ascending=ascending)*.

Args: series: column to rank.

Kwargs: ascending (bool): whether to rank in ascending order (default is *True*).

dfply.window_functions.**cumsum**(*series*)
Calculates cumulative sum of values. Equivalent to *series.cumsum()*.

Args: series: column to compute cumulative sum for.

dfply.window_functions.**cummean**(*series*)
Calculates cumulative mean of values. Equivalent to *series.expanding().mean()*.

Args: series: column to compute cumulative mean for.

dfply.window_functions.**cummax**(*series*)
Calculates cumulative maximum of values. Equivalent to *series.expanding().max()*.

Args: series: column to compute cumulative maximum for.

dfply.window_functions.**cummin**(*series*)
Calculates cumulative minimum of values. Equivalent to *series.expanding().min()*.

Args: series: column to compute cumulative minimum for.

dfply.window_functions.**cumprod**(*series*)
Calculates cumulative product of values. Equivalent to *series.cumprod()*.

Args: series: column to compute cumulative product for.

dfply.window_functions.**cumany**(*series*)
Calculates cumulative any of values. Equivalent to *series.expanding().apply(np.any).astype(bool)*.

Args: series: column to compute cumulative any for.

dfply.window_functions.**cumall**(*series*)
Calculates cumulative all of values. Equivalent to *series.expanding().apply(np.all).astype(bool)*.

Args: series: column to compute cumulative all for.

dfply.window_functions.**percent_rank**(*series*, *ascending=True*)

dfply.window_functions.**row_number**(*series*, *ascending=True*)
Returns row number based on column rank Equivalent to *series.rank(method='first', ascending=ascending)*.

Args: series: column to rank.

Kwargs: ascending (bool): whether to rank in ascending order (default is *True*).

Usage: diamonds >> head() >> mutate(rn=row_number(X.x))

carat cut color clarity depth table price x y z rn

0 0.23 Ideal E SI2 61.5 55.0 326 3.95 3.98 2.43 2.0 1 0.21 Premium E SI1 59.8 61.0 326 3.89 3.84 2.31 1.0 2 0.23 Good E VS1 56.9 65.0 327 4.05 4.07 2.31 3.0 3 0.29 Premium I VS2 62.4 58.0 334 4.20 4.23 2.63 4.0 4 0.31 Good J SI2 63.3 58.0 335 4.34 4.35 2.75 5.0

1.1.3 Package Contents

dfply.**diamonds**

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

d

Index

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

X