Lecture#4

May 18, 2023

1 Python for Signal Processing

Danilo Greco, PhD - danilo.greco@uniparthenope.it - University of Naples Parthenope

1.0.1 Lecture 4

This lecture will provide an overview on Pandas data management library:

1. installation,
2. documentation,
3. main functions and applications,
4. practical examples.

1.0.2 What is Pandas?

Pandas is a fundamental library for data analysis and manipulation tool in Python.

It provides some impressive features:

- a fast and efficient DataFrame object for data manipulation with integrated indexing
- tools for reading and writing data between in-memory data structures and different formats: CSV, Microsoft Excel, etc.
- powerful indexing, reshaping and slicing
- dataset merge and aggregation functions
- time series support
- basic data analytic functions

and much more.

Pandas is based on the DataFrame object which encapsulates one or more Series, that are 1D ndarray with axis labels (including time series).

2 Install

It can be installed by executing the following command:

```
pip install pandas
```

or

```
python -m pip install pandas
```

NOTE: when installing with python version 3.x, replace pip or python with pip3 or python3 in the commands above.

3 Documentation

The official pandas documentation is available on this website and provides an extensive guide for both users and developers, including setup and absolute beginners tutorials.

There is also an interesting book explaining how to use pandas for data analysis: >Wes McKinney, Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython, O'Reilly Media

4 Main functions and applications

The first thing to do is inform the python interpreter that we are using the package. The following command tells python to import the library and use an alias for quicker references within our code.

It's useful to import numpy as well even if it is not strictly necessary.

[318]:
```
import numpy as np
import pandas as pd
```

We can create a Series by passing a list of values as for numpy:

[319]:
```
s = pd.Series([1, 3, 5, np.nan, 6, 8])
s
```

[319]:
```
0    1.0
1    3.0
2    5.0
3    NaN
4    6.0
5    8.0
dtype: float64
```

We used np.nan as a numpy constant to assign *Not a Number* as a value in our data. Missing data are generally assigned to NaN when loading a Series or a DataFrame.

Next, we want to create a random dataset indexed on dates:

[320]:
```
count = 10
dates = pd.date_range('20200101', periods=count)
columns = ["SensorA", "SensorB", "SensorC"]

df = pd.DataFrame(np.random.randn(count, len(columns)), index=dates,␣
 ↪columns=columns)
df
```

[320]:
```
              SensorA   SensorB   SensorC
2020-01-01 -0.111036  0.938855 -0.239526
2020-01-02 -1.219236 -0.727935 -1.405468
```

```
2020-01-03  1.416288  0.297426  0.582218
2020-01-04 -0.952213  0.237028  0.078740
2020-01-05 -0.461383  1.614169  0.146575
2020-01-06  0.732611  0.403459  0.280104
2020-01-07  1.287054  0.507770  0.327374
2020-01-08 -0.064964  1.143019 -0.630793
2020-01-09  0.378133 -1.610900 -0.576368
2020-01-10  1.147492  0.168867  0.777388
```

Inspecting the dataset when it contains lots of data can be a cumbersome process therefore we can have a look at the data by using the following methods that display the first or last 5 rows respectively:

[321]:
```
df.head(2)
```

[321]:
```
             SensorA   SensorB   SensorC
2020-01-01 -0.111036  0.938855 -0.239526
2020-01-02 -1.219236 -0.727935 -1.405468
```

[322]:
```
df.tail(5)
```

[322]:
```
             SensorA   SensorB   SensorC
2020-01-06  0.732611  0.403459  0.280104
2020-01-07  1.287054  0.507770  0.327374
2020-01-08 -0.064964  1.143019 -0.630793
2020-01-09  0.378133 -1.610900 -0.576368
2020-01-10  1.147492  0.168867  0.777388
```

Dataset index and header columns can be viewed with

[323]:
```
df.index
```

[323]:
```
DatetimeIndex(['2020-01-01', '2020-01-02', '2020-01-03', '2020-01-04',
               '2020-01-05', '2020-01-06', '2020-01-07', '2020-01-08',
               '2020-01-09', '2020-01-10'],
              dtype='datetime64[ns]', freq='D')
```

[324]:
```
df.columns
```

[324]:
```
Index(['SensorA', 'SensorB', 'SensorC'], dtype='object')
```

Columns may have different data types and use the default index, as shown down here:

[325]:
```
df2 = pd.DataFrame({
        'ColumnA': np.random.randint(10,size=4),
        'ColumnB': pd.Categorical(['Sunny','Cloudy','Rainy','Windy']),
        'ColumnC': 1.3
        })
```

```
print(df2)
df2.dtypes
```

```
   ColumnA ColumnB  ColumnC
0        5   Sunny      1.3
1        5  Cloudy      1.3
2        7   Rainy      1.3
3        4   Windy      1.3
```

[325]:
```
ColumnA       int32
ColumnB    category
ColumnC     float64
dtype: object
```

If a DataFrame contains homogeneus data types, it can be converted to a plain numpy ndarray

[326]:
```
dir(pd.DataFrame)
```

[326]:
```
['T',
 '_AXIS_LEN',
 '_AXIS_NAMES',
 '_AXIS_NUMBERS',
 '_AXIS_ORDERS',
 '_AXIS_TO_AXIS_NUMBER',
 '_HANDLED_TYPES',
 '__abs__',
 '__add__',
 '__and__',
 '__annotations__',
 '__array__',
 '__array_priority__',
 '__array_ufunc__',
 '__array_wrap__',
 '__bool__',
 '__class__',
 '__contains__',
 '__copy__',
 '__deepcopy__',
 '__delattr__',
 '__delitem__',
 '__dict__',
 '__dir__',
 '__divmod__',
 '__doc__',
 '__eq__',
 '__finalize__',
 '__floordiv__',
 '__format__',
```

```
'__ge__',
'__getattr__',
'__getattribute__',
'__getitem__',
'__getstate__',
'__gt__',
'__hash__',
'__iadd__',
'__iand__',
'__ifloordiv__',
'__imod__',
'__imul__',
'__init__',
'__init_subclass__',
'__invert__',
'__ior__',
'__ipow__',
'__isub__',
'__iter__',
'__itruediv__',
'__ixor__',
'__le__',
'__len__',
'__lt__',
'__matmul__',
'__mod__',
'__module__',
'__mul__',
'__ne__',
'__neg__',
'__new__',
'__nonzero__',
'__or__',
'__pos__',
'__pow__',
'__radd__',
'__rand__',
'__rdivmod__',
'__reduce__',
'__reduce_ex__',
'__repr__',
'__rfloordiv__',
'__rmatmul__',
'__rmod__',
'__rmul__',
'__ror__',
'__round__',
```

```
'__rpow__',
'__rsub__',
'__rtruediv__',
'__rxor__',
'__setattr__',
'__setitem__',
'__setstate__',
'__sizeof__',
'__str__',
'__sub__',
'__subclasshook__',
'__truediv__',
'__weakref__',
'__xor__',
'_accessors',
'_accum_func',
'_add_numeric_operations',
'_agg_by_level',
'_agg_examples_doc',
'_agg_summary_and_see_also_doc',
'_align_frame',
'_align_series',
'_append',
'_arith_method',
'_as_manager',
'_box_col_values',
'_can_fast_transpose',
'_check_inplace_and_allows_duplicate_labels',
'_check_inplace_setting',
'_check_is_chained_assignment_possible',
'_check_label_or_level_ambiguity',
'_check_setitem_copy',
'_clear_item_cache',
'_clip_with_one_bound',
'_clip_with_scalar',
'_cmp_method',
'_combine_frame',
'_consolidate',
'_consolidate_inplace',
'_construct_axes_dict',
'_construct_axes_from_arguments',
'_construct_result',
'_constructor',
'_constructor_sliced',
'_convert',
'_count_level',
'_data',
```

```
'_dir_additions',
'_dir_deletions',
'_dispatch_frame_op',
'_drop_axis',
'_drop_labels_or_levels',
'_ensure_valid_index',
'_find_valid_index',
'_from_arrays',
'_from_mgr',
'_get_agg_axis',
'_get_axis',
'_get_axis_name',
'_get_axis_number',
'_get_axis_resolvers',
'_get_block_manager_axis',
'_get_bool_data',
'_get_cleaned_column_resolvers',
'_get_column_array',
'_get_index_resolvers',
'_get_item_cache',
'_get_label_or_level_values',
'_get_numeric_data',
'_get_value',
'_getitem_bool_array',
'_getitem_multilevel',
'_gotitem',
'_hidden_attrs',
'_indexed_same',
'_info_axis',
'_info_axis_name',
'_info_axis_number',
'_info_repr',
'_init_mgr',
'_inplace_method',
'_internal_names',
'_internal_names_set',
'_is_copy',
'_is_homogeneous_type',
'_is_label_or_level_reference',
'_is_label_reference',
'_is_level_reference',
'_is_mixed_type',
'_is_view',
'_iset_item',
'_iset_item_mgr',
'_iset_not_inplace',
'_iter_column_arrays',
```

```
'_ixs',
'_join_compat',
'_logical_func',
'_logical_method',
'_maybe_cache_changed',
'_maybe_update_cacher',
'_metadata',
'_min_count_stat_function',
'_needs_reindex_multi',
'_protect_consolidate',
'_reduce',
'_reduce_axis1',
'_reindex_axes',
'_reindex_columns',
'_reindex_index',
'_reindex_multi',
'_reindex_with_indexers',
'_rename',
'_replace_columnwise',
'_repr_data_resource_',
'_repr_fits_horizontal_',
'_repr_fits_vertical_',
'_repr_html_',
'_repr_latex_',
'_reset_cache',
'_reset_cacher',
'_sanitize_column',
'_series',
'_set_axis',
'_set_axis_name',
'_set_axis_nocheck',
'_set_is_copy',
'_set_item',
'_set_item_frame_value',
'_set_item_mgr',
'_set_value',
'_setitem_array',
'_setitem_frame',
'_setitem_slice',
'_slice',
'_stat_axis',
'_stat_axis_name',
'_stat_axis_number',
'_stat_function',
'_stat_function_ddof',
'_take_with_is_copy',
'_to_dict_of_blocks',
```

```
'_typ',
'_update_inplace',
'_validate_dtype',
'_values',
'_where',
'abs',
'add',
'add_prefix',
'add_suffix',
'agg',
'aggregate',
'align',
'all',
'any',
'append',
'apply',
'applymap',
'asfreq',
'asof',
'assign',
'astype',
'at',
'at_time',
'attrs',
'axes',
'backfill',
'between_time',
'bfill',
'bool',
'boxplot',
'clip',
'columns',
'combine',
'combine_first',
'compare',
'convert_dtypes',
'copy',
'corr',
'corrwith',
'count',
'cov',
'cummax',
'cummin',
'cumprod',
'cumsum',
'describe',
'diff',
```

```
'div',
'divide',
'dot',
'drop',
'drop_duplicates',
'droplevel',
'dropna',
'dtypes',
'duplicated',
'empty',
'eq',
'equals',
'eval',
'ewm',
'expanding',
'explode',
'ffill',
'fillna',
'filter',
'first',
'first_valid_index',
'flags',
'floordiv',
'from_dict',
'from_records',
'ge',
'get',
'groupby',
'gt',
'head',
'hist',
'iat',
'idxmax',
'idxmin',
'iloc',
'index',
'infer_objects',
'info',
'insert',
'interpolate',
'isin',
'isna',
'isnull',
'items',
'iteritems',
'iterrows',
'itertuples',
```

```
'join',
'keys',
'kurt',
'kurtosis',
'last',
'last_valid_index',
'le',
'loc',
'lookup',
'lt',
'mad',
'mask',
'max',
'mean',
'median',
'melt',
'memory_usage',
'merge',
'min',
'mod',
'mode',
'mul',
'multiply',
'ndim',
'ne',
'nlargest',
'notna',
'notnull',
'nsmallest',
'nunique',
'pad',
'pct_change',
'pipe',
'pivot',
'pivot_table',
'plot',
'pop',
'pow',
'prod',
'product',
'quantile',
'query',
'radd',
'rank',
'rdiv',
'reindex',
'reindex_like',
```

```
'rename',
'rename_axis',
'reorder_levels',
'replace',
'resample',
'reset_index',
'rfloordiv',
'rmod',
'rmul',
'rolling',
'round',
'rpow',
'rsub',
'rtruediv',
'sample',
'select_dtypes',
'sem',
'set_axis',
'set_flags',
'set_index',
'shape',
'shift',
'size',
'skew',
'slice_shift',
'sort_index',
'sort_values',
'sparse',
'squeeze',
'stack',
'std',
'style',
'sub',
'subtract',
'sum',
'swapaxes',
'swaplevel',
'tail',
'take',
'to_clipboard',
'to_csv',
'to_dict',
'to_excel',
'to_feather',
'to_gbq',
'to_hdf',
'to_html',
```

```
 'to_json',
 'to_latex',
 'to_markdown',
 'to_numpy',
 'to_parquet',
 'to_period',
 'to_pickle',
 'to_records',
 'to_sql',
 'to_stata',
 'to_string',
 'to_timestamp',
 'to_xarray',
 'to_xml',
 'transform',
 'transpose',
 'truediv',
 'truncate',
 'tshift',
 'tz_convert',
 'tz_localize',
 'unstack',
 'update',
 'value_counts',
 'values',
 'var',
 'where',
 'xs']
```

[327]:
```
df.values
```

[327]:
```
array([[-0.11103589,  0.93885463, -0.23952558],
       [-1.21923645, -0.72793451, -1.40546812],
       [ 1.41628812,  0.29742617,  0.58221825],
       [-0.95221329,  0.23702819,  0.07874023],
       [-0.46138256,  1.61416912,  0.14657472],
       [ 0.73261097,  0.4034592 ,  0.28010365],
       [ 1.2870541 ,  0.50776983,  0.32737421],
       [-0.06496448,  1.14301852, -0.63079325],
       [ 0.37813318, -1.61090044, -0.57636789],
       [ 1.14749157,  0.16886671,  0.77738771]])
```

To get a quick statistical summary of our dataset, the following method can be invoked:

[328]:
```
df.describe()
```

[328]:

	SensorA	SensorB	SensorC
count	10.000000	10.000000	10.000000

```
mean    0.215275   0.297176  -0.065976
std     0.932026   0.920689   0.656244
min    -1.219236  -1.610900  -1.405468
25%    -0.373796   0.185907  -0.492157
50%     0.156584   0.350443   0.112657
75%     1.043771   0.831083   0.315557
max     1.416288   1.614169   0.777388
```

Data can be easily sorted in ascending or descending order with:

[329]:
```
df.sort_values(by='SensorB',ascending=True)
```

[329]:
```
              SensorA   SensorB   SensorC
2020-01-09  0.378133 -1.610900 -0.576368
2020-01-02 -1.219236 -0.727935 -1.405468
2020-01-10  1.147492  0.168867  0.777388
2020-01-04 -0.952213  0.237028  0.078740
2020-01-03  1.416288  0.297426  0.582218
2020-01-06  0.732611  0.403459  0.280104
2020-01-07  1.287054  0.507770  0.327374
2020-01-01 -0.111036  0.938855 -0.239526
2020-01-08 -0.064964  1.143019 -0.630793
2020-01-05 -0.461383  1.614169  0.146575
```

Selection in a DataFrame can be done in the standard python slicing ways but they're not optimized for pandas which relies on four access methods: >loc, iloc, at, iat.

Here are some examples:

[330]:
```
print(dates[2])
df.loc[dates[2]]   # selection by label
```

```
2020-01-03 00:00:00
```

[330]:
```
SensorA    1.416288
SensorB    0.297426
SensorC    0.582218
Name: 2020-01-03 00:00:00, dtype: float64
```

[331]:
```
df.loc[:,["SensorA","SensorC"]].tail() # selection by label on multiple columns
```

[331]:
```
              SensorA   SensorC
2020-01-06  0.732611  0.280104
2020-01-07  1.287054  0.327374
2020-01-08 -0.064964 -0.630793
2020-01-09  0.378133 -0.576368
2020-01-10  1.147492  0.777388
```

[332]:
```
df.loc['2020-01-03'].at['SensorC']  # access the value of given row and column
```

[332]:
```
0.5822182512962807
```

[333]:
```
print(dates[2])
df.at[pd.Timestamp('2020-01-03'),'SensorC']
```

```
2020-01-03 00:00:00
```

[333]:
```
0.5822182512962807
```

[334]:
```
df.at[dates[2],'SensorC']  # access the value of given row and column
```

[334]:
```
0.5822182512962807
```

[335]:
```
df.iloc[2]  # selection by position
```

[335]:
```
SensorA    1.416288
SensorB    0.297426
SensorC    0.582218
Name: 2020-01-03 00:00:00, dtype: float64
```

[336]:
```
df.iloc[-3:,[0,2]]
```

[336]:
```
              SensorA   SensorC
2020-01-08 -0.064964 -0.630793
2020-01-09  0.378133 -0.576368
2020-01-10  1.147492  0.777388
```

[337]:
```
df.iat[2,1]
```

[337]:
```
0.29742617283525397
```

All previous commands can also be used to assign values at slices or individual elements, such as:

[338]:
```
df.iat[2,1] = 0.0
df.head()
```

[338]:
```
              SensorA   SensorB   SensorC
2020-01-01 -0.111036  0.938855 -0.239526
2020-01-02 -1.219236 -0.727935 -1.405468
2020-01-03  1.416288  0.000000  0.582218
2020-01-04 -0.952213  0.237028  0.078740
2020-01-05 -0.461383  1.614169  0.146575
```

Filtering is also possible with boolean conditions and the result can be used to create a new DataFrame with:

[339]:
```
df3 = df[df['SensorA'] > 0].copy()
df3
```

[339]:
```
              SensorA   SensorB   SensorC
2020-01-03  1.416288  0.000000  0.582218
2020-01-06  0.732611  0.403459  0.280104
2020-01-07  1.287054  0.507770  0.327374
2020-01-09  0.378133 -1.610900 -0.576368
2020-01-10  1.147492  0.168867  0.777388
```

Alternatively, filtering can be applied with the query method, even combining multiple conditions:

[340]:
```
df.query('SensorA > 0 & SensorB > 0')
```

[340]:
```
              SensorA   SensorB   SensorC
2020-01-06  0.732611  0.403459  0.280104
2020-01-07  1.287054  0.507770  0.327374
2020-01-10  1.147492  0.168867  0.777388
```

4.0.1 Dealing with missing data

Missing data are always getting trouble to data analysts but Pandas allows to deal with them easily.

But first, we need to go through CSV data import, which is one of the most frequently used formats and it natively supported in Pandas.

Many other formats are readable but they won't be addressed in this lecture.

In order to read data from a CSV file, we can use the pd.read_csv library function, with some specific parameters to correctly import the index.

If we do not specify the proper parameters, the simplest import produces an undesired result where the index column is automatically assigned to automatical integer values:

[341]:
```
df_missing = pd.read_csv('missing.csv')
df_missing
```

[341]:
```
   Unnamed: 0   SensorA   SensorB   SensorC
0  2020-01-03  0.470622 -1.172069  1.661895
1  2020-01-04  1.797044       NaN  0.164888
2  2020-01-09  2.229225  0.751242       NaN
3  2020-01-10  0.839191  1.247696       NaN
4  2020-01-13  0.424159 -0.611664  0.179591
5  2020-01-16  1.222883 -0.992786  1.642943
6  2020-01-20  0.116415  0.354019  0.277395
```

The Unnamed: 0 column is likely to be the original index of the dataset, hence we can set the index with:

[342]:
```
df_missing.set_index('Unnamed: 0')
```

[342]:
```
              SensorA   SensorB   SensorC
Unnamed: 0
2020-01-03  0.470622 -1.172069  1.661895
```

```
2020-01-04  1.797044       NaN  0.164888
2020-01-09  2.229225  0.751242       NaN
2020-01-10  0.839191  1.247696       NaN
2020-01-13  0.424159 -0.611664  0.179591
2020-01-16  1.222883 -0.992786  1.642943
2020-01-20  0.116415  0.354019  0.277395
```

WARNING: some transformations are not persisted unless you assign the result to a variable If we now display the df_missing dataset we find out that the index is not set as expected.

[343]:
```
df_missing
```

[343]:
```
   Unnamed: 0   SensorA   SensorB   SensorC
0  2020-01-03  0.470622 -1.172069  1.661895
1  2020-01-04  1.797044       NaN  0.164888
2  2020-01-09  2.229225  0.751242       NaN
3  2020-01-10  0.839191  1.247696       NaN
4  2020-01-13  0.424159 -0.611664  0.179591
5  2020-01-16  1.222883 -0.992786  1.642943
6  2020-01-20  0.116415  0.354019  0.277395
```

To make sure our changes are persisted we can either assign a variable or use the inplace = True parameter in the set_index call. This parameter is generally available for many data transformation functions in Pandas.

[344]:
```
df_new = df_missing.set_index('Unnamed: 0')
df_new
```

[344]:
```
              SensorA   SensorB   SensorC
Unnamed: 0
2020-01-03  0.470622 -1.172069  1.661895
2020-01-04  1.797044       NaN  0.164888
2020-01-09  2.229225  0.751242       NaN
2020-01-10  0.839191  1.247696       NaN
2020-01-13  0.424159 -0.611664  0.179591
2020-01-16  1.222883 -0.992786  1.642943
2020-01-20  0.116415  0.354019  0.277395
```

[345]:
```
df_missing.set_index('Unnamed: 0',inplace=True)
df_missing
```

[345]:
```
              SensorA   SensorB   SensorC
Unnamed: 0
2020-01-03  0.470622 -1.172069  1.661895
2020-01-04  1.797044       NaN  0.164888
2020-01-09  2.229225  0.751242       NaN
2020-01-10  0.839191  1.247696       NaN
```

```
2020-01-13  0.424159 -0.611664  0.179591
2020-01-16  1.222883 -0.992786  1.642943
2020-01-20  0.116415  0.354019  0.277395
```

We can specify the index column at read time in order to save time:

[346]:
```
df_missing = pd.read_csv('missing.csv',index_col=0)
df_missing
```

[346]:
```
             SensorA   SensorB   SensorC
2020-01-03  0.470622 -1.172069  1.661895
2020-01-04  1.797044       NaN  0.164888
2020-01-09  2.229225  0.751242       NaN
2020-01-10  0.839191  1.247696       NaN
2020-01-13  0.424159 -0.611664  0.179591
2020-01-16  1.222883 -0.992786  1.642943
2020-01-20  0.116415  0.354019  0.277395
```

Going back to missing data, they are identified by NaN values in our dataset.

Should we want to remove all data rows containing missing values, the dropna function is used:

[347]:
```
df_missing.dropna(how='any')
```

[347]:
```
             SensorA   SensorB   SensorC
2020-01-03  0.470622 -1.172069  1.661895
2020-01-13  0.424159 -0.611664  0.179591
2020-01-16  1.222883 -0.992786  1.642943
2020-01-20  0.116415  0.354019  0.277395
```

Conversely, to maintain those rows and columns and replace missing values with some other valid ones, we use:

[348]:
```
df_missing.fillna(0.0)
```

[348]:
```
             SensorA   SensorB   SensorC
2020-01-03  0.470622 -1.172069  1.661895
2020-01-04  1.797044  0.000000  0.164888
2020-01-09  2.229225  0.751242  0.000000
2020-01-10  0.839191  1.247696  0.000000
2020-01-13  0.424159 -0.611664  0.179591
2020-01-16  1.222883 -0.992786  1.642943
2020-01-20  0.116415  0.354019  0.277395
```

These are only basic examples because many optional parameters are available for those functions that give us more flexibility on how to deal with missing values. Finally, if we want to have a boolean map of missing values the pd.isna function is returns True where NaN values are located.

[349]:
```
pd.isna(df_missing)
```

[349]:
```
            SensorA  SensorB  SensorC
2020-01-03    False    False    False
2020-01-04    False     True    False
2020-01-09    False    False     True
2020-01-10    False    False     True
2020-01-13    False    False    False
2020-01-16    False    False    False
2020-01-20    False    False    False
```

It is also possible to apply functions to the dataset, such as:

[350]:
```
df_missing.apply(np.cumsum) # incrementally sum over columns
```

[350]:
```
             SensorA   SensorB   SensorC
2020-01-03  0.470622 -1.172069  1.661895
2020-01-04  2.267666       NaN  1.826783
2020-01-09  4.496891 -0.420826       NaN
2020-01-10  5.336082  0.826869       NaN
2020-01-13  5.760241  0.215206  2.006374
2020-01-16  6.983124 -0.777581  3.649317
2020-01-20  7.099539 -0.423561  3.926712
```

[351]:
```
df_missing.apply(lambda x: x.max() - x.min()) # uses lambda functions over the␣
↪columns
```

[351]:
```
SensorA    2.112809
SensorB    2.419764
SensorC    1.497007
dtype: float64
```

Other aggregations can be done with groupby:

[352]:
```
df = pd.DataFrame({'Class': ['A','B','A','A','B','A','B','A','B','B'],
                   'Data1': np.random.randn(10),
                   'Data2': np.random.randn(10)})

df.set_index('Class',inplace=True) # optional
df
```

[352]:
```
          Data1     Data2
Class
A     -2.132686  0.262920
B     -0.454462  1.782832
A      1.645823 -0.284874
A     -0.104982  0.175540
B      0.833925 -0.179474
A      0.926685 -0.105481
B      1.712343 -0.342803
A      0.253101  0.528295
```

```
B      0.601537  0.434794
B     -0.059516  0.172697
```

[353]:
```
df.groupby('Class').sum()
```

[353]:
```
           Data1     Data2
Class
A       0.587941  0.576401
B       2.633827  1.868046
```

[354]:
```
df.groupby('Class').mean()
```

[354]:
```
           Data1     Data2
Class
A       0.117588  0.115280
B       0.526765  0.373609
```

4.0.2 Pivot Tables

Creation of pivot tables from data is natively supported in Pandas and this provides a quick and powerful way to aggregate data.

In the examples below, we have a list of row entries representing acquisitions from field sensors and we produce two pivot tables averaging the values (other aggregation functions may be specified) according to the desired rows and columns:

[355]:
```
dfp = pd.DataFrame({
        'Room':['LivingRoom', 'BedRoom', 'BathRoom']*4, #repeat this definition␣
 ↪for 4 times
        'Position': ['Floor','Floor','Floor','Ceiling','Ceiling','Ceiling']*2,␣
 ↪#repeat this definition for 2 times
        'Temperature': np.random.uniform(low=15.0, high=35.0, size=(12,)),
        'Humidity': np.random.random(12) #np.random.random is a function␣
 ↪provided by the NumPy library that generates random numbers.
                                         #It returns random floats between 0.0␣
 ↪and 1.0.
        })

dfp
```

[355]:
```
          Room Position  Temperature  Humidity
0   LivingRoom    Floor    16.814480  0.891609
1      BedRoom    Floor    32.502342  0.514669
2     BathRoom    Floor    32.682362  0.349457
3   LivingRoom  Ceiling    22.208808  0.768508
4      BedRoom  Ceiling    16.131473  0.172980
5     BathRoom  Ceiling    23.054688  0.020702
6   LivingRoom    Floor    17.566295  0.884646
```

```
7       BedRoom    Floor    19.420697  0.497103
8      BathRoom    Floor    28.748401  0.588229
9    LivingRoom  Ceiling    20.462798  0.526227
10      BedRoom  Ceiling    29.165841  0.139532
11     BathRoom  Ceiling    16.932523  0.625002
```

[356]:
```
pd.pivot_table(dfp, values=['Temperature','Humidity'], index=['Room'],␣
↪columns=['Position'])
```

[356]:

	Humidity		Temperature	
Position	Ceiling	Floor	Ceiling	Floor
Room				
BathRoom	0.322852	0.468843	19.993606	30.715382
BedRoom	0.156256	0.505886	22.648657	25.961519
LivingRoom	0.647368	0.888127	21.335803	17.190387

[357]:
```
pd.pivot_table(dfp, values=['Temperature','Humidity'],␣
↪index=['Position','Room'])
```

[357]:

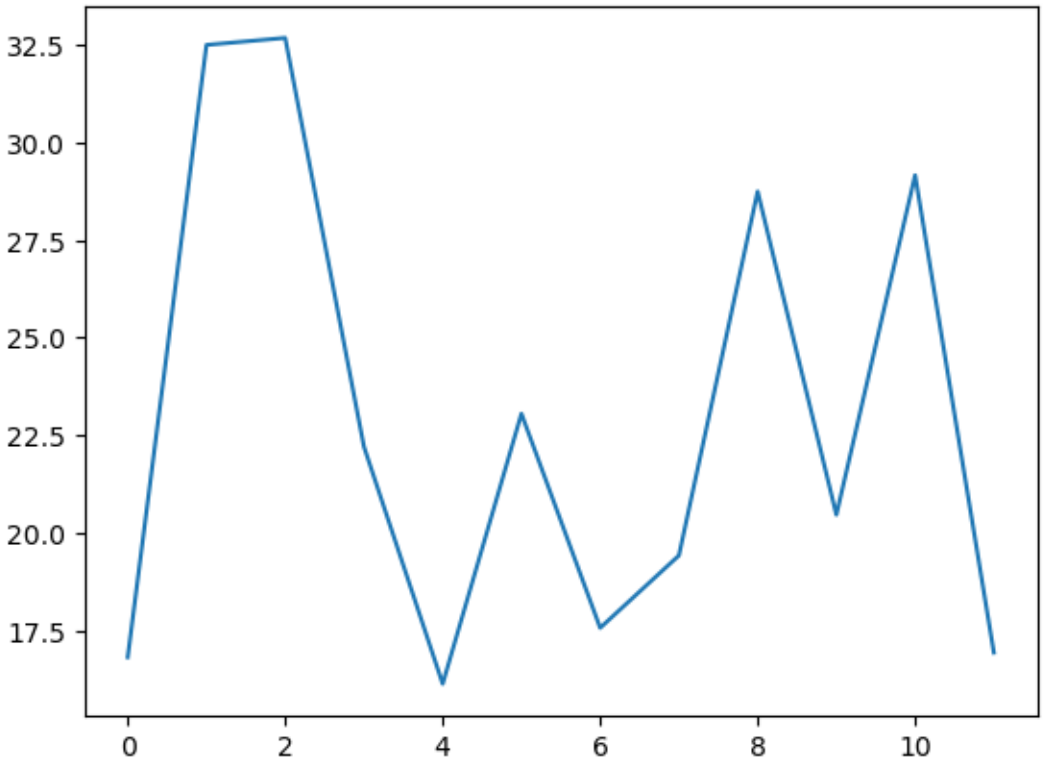
		Humidity	Temperature
Position	Room		
Ceiling	BathRoom	0.322852	19.993606
	BedRoom	0.156256	22.648657
	LivingRoom	0.647368	21.335803
Floor	BathRoom	0.468843	30.715382
	BedRoom	0.505886	25.961519
	LivingRoom	0.888127	17.190387

4.0.3 Plotting data

Pandas includes some interesting features to graphically represent DataFrames:

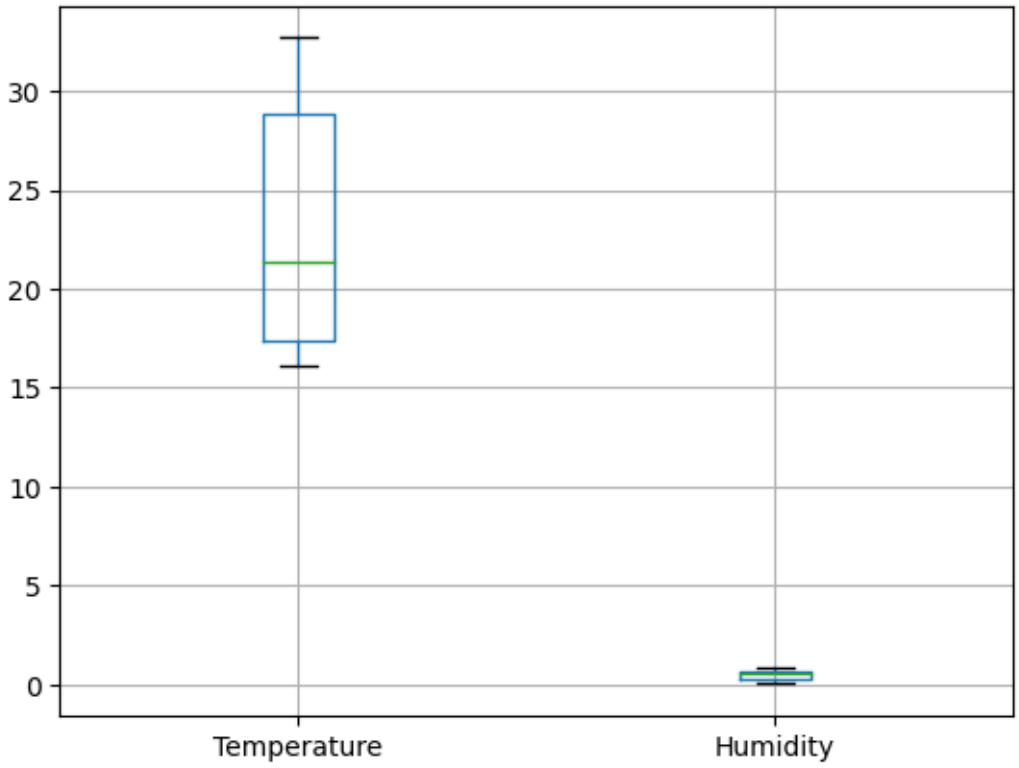
[358]:
```
dfp['Temperature'].plot()
```

[358]:
```
<AxesSubplot:>
```



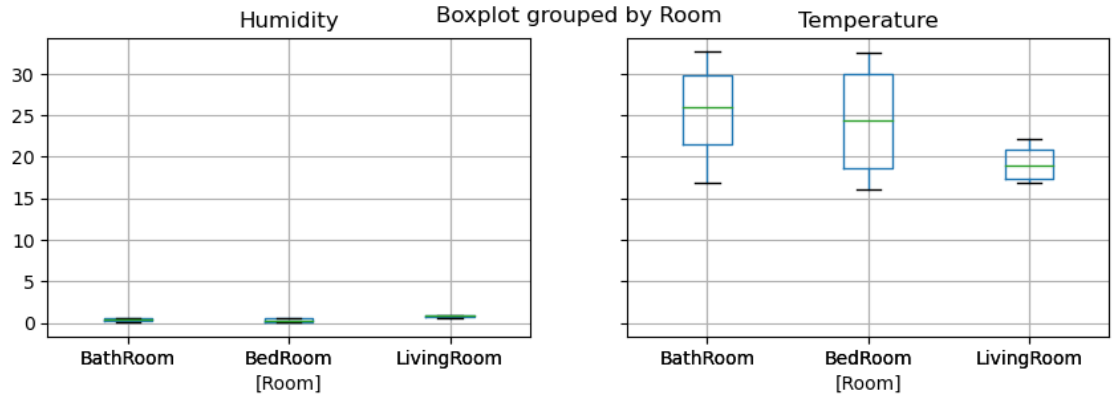
[359]:
```
dfp.boxplot()
```

[359]: <AxesSubplot:>



[360]:
```
dfp.boxplot(by=['Room'],figsize=(10,3))
```

[360]:
```
array([<AxesSubplot:title={'center':'Humidity'}, xlabel='[Room]'>,
       <AxesSubplot:title={'center':'Temperature'}, xlabel='[Room]'>],
      dtype=object)
```



The numerical values for the plots displayed above can be calculated through the following methods:

[361]:
```
dfp.groupby('Room')['Temperature'].mean()
```

[361]:
```
Room
BathRoom      25.354494
BedRoom       24.305088
LivingRoom    19.263095
Name: Temperature, dtype: float64
```

The previous call returns a Pandas Series as shown by

[362]:
```
dfp.groupby('Room')['Temperature'].mean().__class__
```

[362]:
```
pandas.core.series.Series
```

whereas the alternative option returns a Pandas DataFrame:

[363]:
```
dfp.groupby('Room').agg({'Temperature':'mean'})
```

[363]:
```
             Temperature
Room
BathRoom      25.354494
BedRoom       24.305088
LivingRoom    19.263095
```

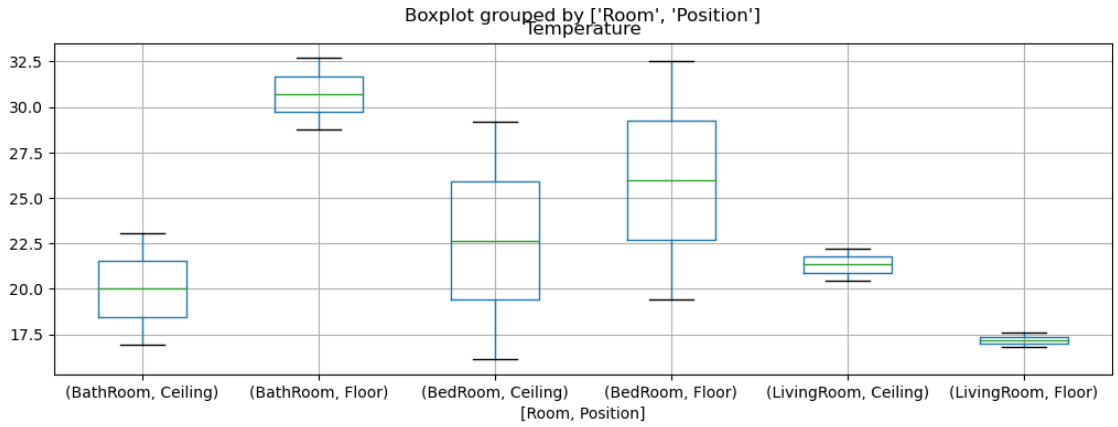
[364]:
```
dfp.groupby('Room').agg({'Temperature':'mean'}).__class__
```

[364]:
```
pandas.core.frame.DataFrame
```

Another example plot displays Temperature data with two levels of aggregation:

[365]:
```
dfp.boxplot(column=['Temperature'], by=['Room','Position'],figsize=(12,4))
```

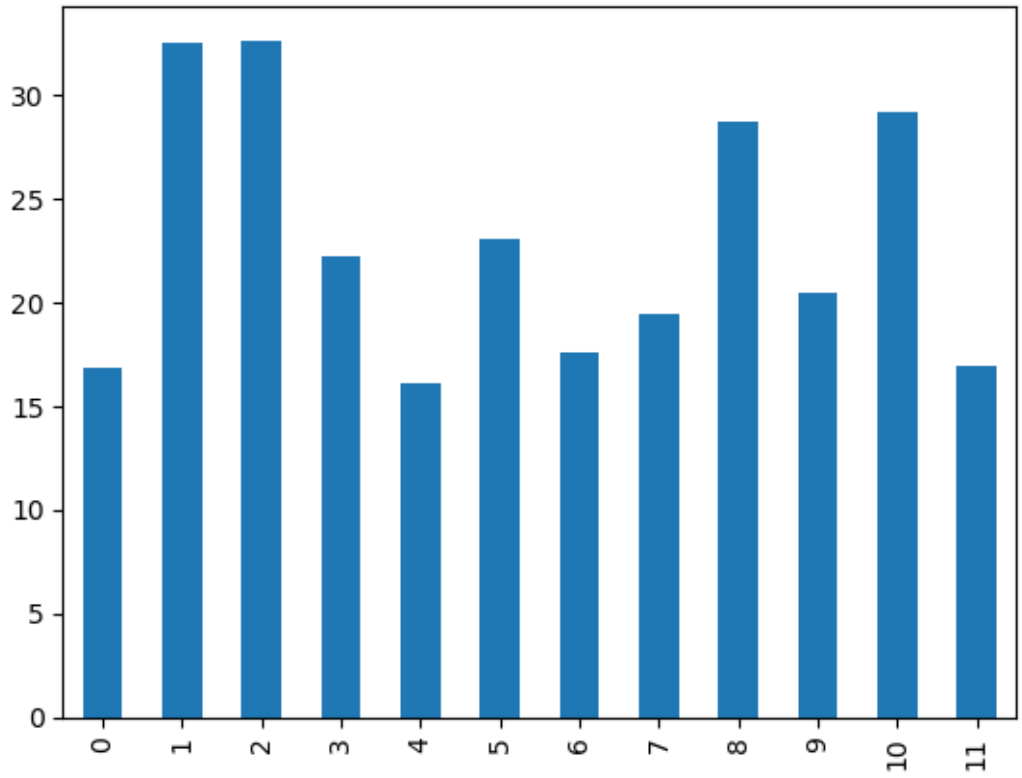
[365]:
```
<AxesSubplot:title={'center':'Temperature'}, xlabel='[Room, Position]'>
```



Many other options are available for the plot method by selecting the proper kind among a set of possible values: bar, hist, kde, box, etc.

[366]:
```
dfp['Temperature'].plot(kind='bar')
```

[366]:
```
<AxesSubplot:>
```



4.0.4 A complete data exploration task

The following cells demonstrate a possible use of Pandas to perform data exploration and visualization using its internals.

For this purpose, we'll be using a dataset about diamonds characteristics to show the basic steps of data exploration:

[367]:
```
import pandas as pd
import numpy as np

filename = 'diamonds.csv'
dmds = pd.read_csv(filename)
data = dmds.copy()
data.tail()
```

[367]:

	carat	cut	color	clarity	depth	table	price	x	y	z
53935	0.72	Ideal	D	SI1	60.8	57.0	2757	5.75	5.76	3.50
53936	0.72	Good	D	SI1	63.1	55.0	2757	5.69	5.75	3.61
53937	0.70	Very Good	D	SI1	62.8	60.0	2757	5.66	5.68	3.56
53938	0.86	Premium	H	SI2	61.0	58.0	2757	6.15	6.12	3.74
53939	0.75	Ideal	D	SI2	62.2	55.0	2757	5.83	5.87	3.64

The info command displays the column names with their relevant data types and memory occupation for the loaded dataset.

[368]:
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 53940 entries, 0 to 53939
Data columns (total 10 columns):
 #   Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   carat    53940 non-null  float64
 1   cut      53940 non-null  object
 2   color    53940 non-null  object
 3   clarity  53940 non-null  object
 4   depth    53940 non-null  float64
 5   table    53940 non-null  float64
 6   price    53940 non-null  int64
 7   x        53940 non-null  float64
 8   y        53940 non-null  float64
 9   z        53940 non-null  float64
dtypes: float64(6), int64(1), object(3)
memory usage: 4.1+ MB
```

The describe method displays statistical information about the dataset for all numerical columns. As shown in the table below the categorical variables are disregarded by the function:

[369]:
```
data.describe()
```

[369]:

	carat	depth	table	price	x	y	z
count	53940.000000	53940.000000	53940.000000	53940.000000	53940.000000	53940.000000	53940.000000
mean	0.797940	61.749405	57.457184	3932.799722	5.731157	5.734526	3.538734
std	0.474011	1.432621	2.234491	3989.439738	1.121761	1.142135	0.705699
min	0.200000	43.000000	43.000000	326.000000	0.000000		
25%	0.400000	61.000000	56.000000	950.000000	4.710000		
50%	0.700000	61.800000	57.000000	2401.000000	5.700000		
75%	1.040000	62.500000	59.000000	5324.250000	6.540000		
max	5.010000	79.000000	95.000000	18823.000000	10.740000		

```
min        0.000000      0.000000
25%        4.720000      2.910000
50%        5.710000      3.530000
75%        6.540000      4.040000
max       58.900000     31.800000
```

Specific percentile values can be displayed by adding the list of desired percentiles in the percentiles parameter of the describe method:

[370]:
```
percs = [.1, .3, .7, .9]
data.describe(percentiles=percs)
```

[370]:
```
              carat         depth         table         price             x  \
count  53940.000000  53940.000000  53940.000000  53940.000000  53940.000000
mean       0.797940     61.749405     57.457184   3932.799722      5.731157
std        0.474011      1.432621      2.234491   3989.439738      1.121761
min        0.200000     43.000000     43.000000    326.000000      0.000000
10%        0.310000     60.000000     55.000000    646.000000      4.360000
30%        0.420000     61.200000     56.000000   1087.000000      4.820000
50%        0.700000     61.800000     57.000000   2401.000000      5.700000
70%        1.010000     62.400000     58.000000   4662.000000      6.420000
90%        1.510000     63.300000     60.000000   9821.000000      7.310000
max        5.010000     79.000000     95.000000  18823.000000     10.740000

                 y             z
count  53940.000000  53940.000000
mean       5.734526      3.538734
std        1.142135      0.705699
min        0.000000      0.000000
10%        4.360000      2.690000
30%        4.830000      2.980000
50%        5.710000      3.530000
70%        6.420000      3.980000
90%        7.300000      4.520000
max       58.900000     31.800000
```

In order to display additional information about categorical variables an additional parameter should be specified to obtain record count, the number of unique values of each variable, the most frequent value and its occurrence count.

[371]:
```
data.describe(include=np.object)
```

```
C:\Users\GRCDNL71D14D969B\AppData\Local\Temp\ipykernel_13892\307950999.py:1:
DeprecationWarning: `np.object` is a deprecated alias for the builtin `object`.
To silence this warning, use `object` by itself. Doing this will not modify any
behavior and is safe.
Deprecated in NumPy 1.20; for more details and guidance:
https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
  data.describe(include=np.object)
```

[371]:
```
           cut  color clarity
count    53940  53940   53940
unique       5      7       8
top      Ideal      G     SI1
freq     21551  11292   13065
```

In order to display the unique values for categorical features, the following command can be used:

[372]:
```
data['cut'].unique()
```

[372]:
```
array(['Ideal', 'Premium', 'Good', 'Very Good', 'Fair'], dtype=object)
```

[373]:
```
data['color'].unique()
```

[373]:
```
array(['E', 'I', 'J', 'H', 'F', 'G', 'D'], dtype=object)
```

[374]:
```
data['clarity'].unique()
```

[374]:
```
array(['SI2', 'SI1', 'VS1', 'VS2', 'VVS2', 'VVS1', 'I1', 'IF'],
      dtype=object)
```

One important transformation that might be useful for data manipulation is mapping all categorical variables into numerical equivalents. To do this, we first create the proper dictionaries:

[375]:
```
cut_map = {v: c for c, v in enumerate(data['cut'].unique())}
color_map = {v: c for c, v in enumerate(data['color'].unique())}
clarity_map = {v: c for c, v in enumerate(data['clarity'].unique())}
```

[376]:
```
cut_map, color_map, clarity_map
```

[376]:
```
({'Ideal': 0, 'Premium': 1, 'Good': 2, 'Very Good': 3, 'Fair': 4},
 {'E': 0, 'I': 1, 'J': 2, 'H': 3, 'F': 4, 'G': 5, 'D': 6},
 {'SI2': 0,
  'SI1': 1,
  'VS1': 2,
  'VS2': 3,
  'VVS2': 4,
  'VVS1': 5,
  'I1': 6,
  'IF': 7})
```

Then we map the dataframe columns using the dictionaries to replace the textual values with the numerical ones:

[377]:
```
data['cut'] = data['cut'].map(cut_map)
data['color'] = data['color'].map(color_map)
data['clarity'] = data['clarity'].map(clarity_map)
data.tail()
```

[377]:

	carat	cut	color	clarity	depth	table	price	x	y	z
53935	0.72	0	6	1	60.8	57.0	2757	5.75	5.76	3.50
53936	0.72	2	6	1	63.1	55.0	2757	5.69	5.75	3.61
53937	0.70	3	6	1	62.8	60.0	2757	5.66	5.68	3.56
53938	0.86	1	3	0	61.0	58.0	2757	6.15	6.12	3.74
53939	0.75	0	6	0	62.2	55.0	2757	5.83	5.87	3.64

It is often necessary to clean the dataset from NaN values before applying any machine learning algorithm, therefore we first need to check and eventually count these values across the columns:

[378]:
```
data.isna().sum()
```

[378]:
```
carat      0
cut        0
color      0
clarity    0
depth      0
table      0
price      0
x          0
y          0
z          0
dtype: int64
```

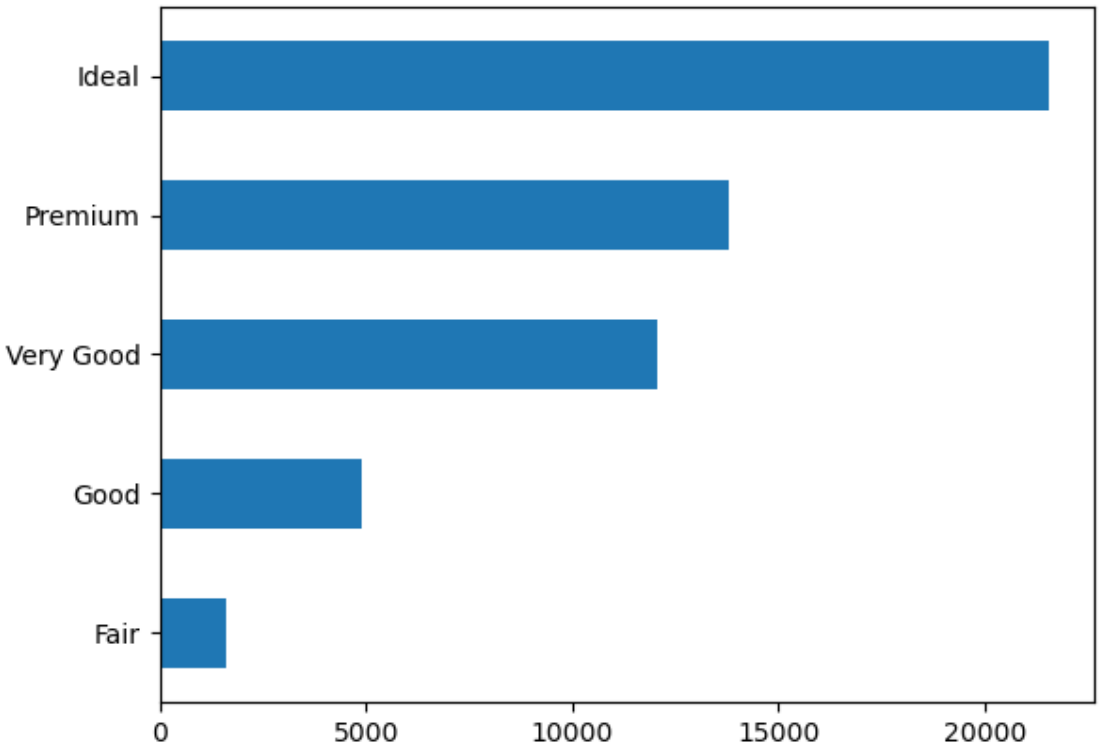
A summary of the number of occurrencies of each symbol in a colum in descending order is easily obtained by:

[379]:
```
count = dmds['cut'].value_counts()
count
```

[379]:
```
Ideal        21551
Premium      13791
Very Good    12082
Good          4906
Fair          1610
Name: cut, dtype: int64
```

[380]:
```
count.sort_values(ascending=True).plot(kind='barh')
```

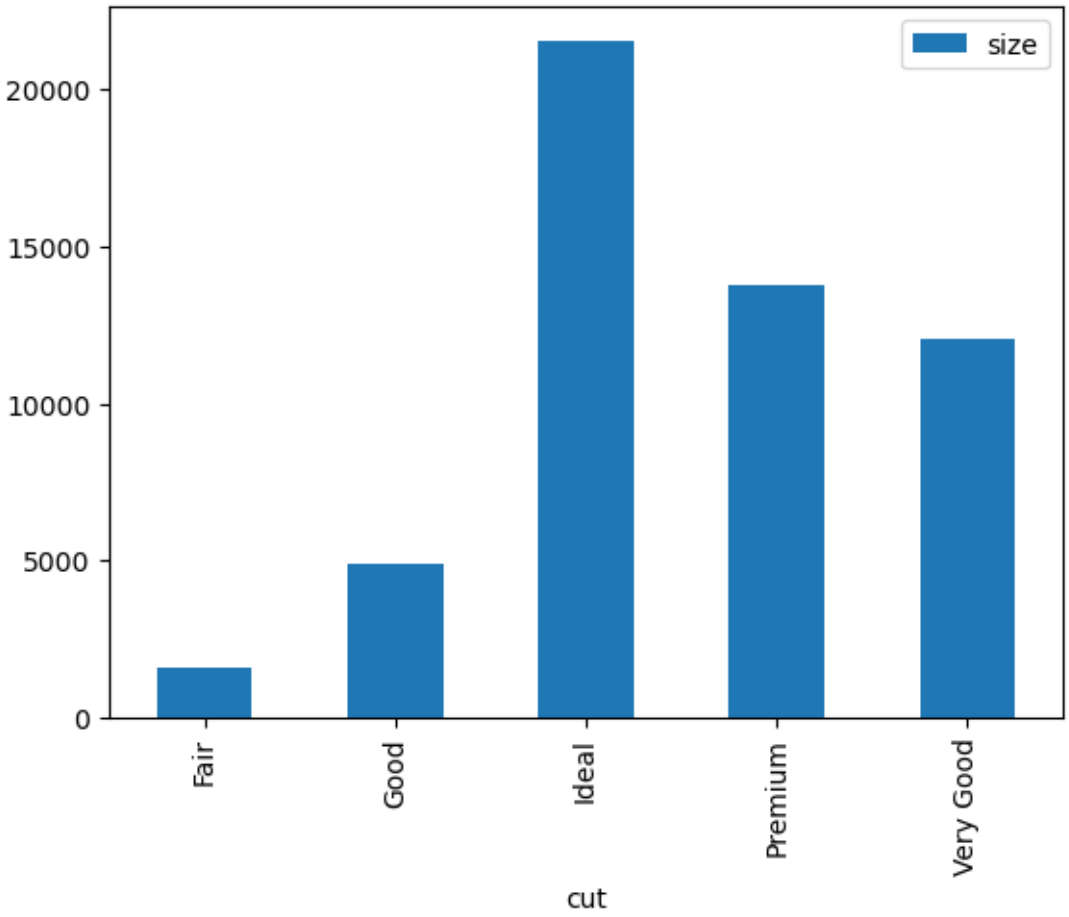
[380]:
```
<AxesSubplot:>
```



A group by clause can be used to display more detailed information about a given column, preserving the natural ordering of the aggregation column, such as:

[381]:
```
count = dmds.groupby(by='cut',as_index=False).size()
count.plot(kind='bar',x='cut',y='size')
```

[381]:
```
<AxesSubplot:xlabel='cut'>
```



Now we can filter out all diamonds that do not match our quality requirements by combining multiple conditions on the various columns: we select clarity **IF**, color **D** and cut **Ideal**.

[382]:
```
topq = dmds[(data['carat'] > 1.0) & (data['clarity'] == 7) & (data['color'] ==␣
↪6) & (data['cut'] == 0)]
topq
```

[382]:

	carat	cut	color	clarity	depth	table	price	x	y	z
25622	1.04	Ideal	D	IF	61.8	57.0	14494	6.49	6.52	4.02
25718	1.04	Ideal	D	IF	61.8	57.0	14626	6.52	6.49	4.02
26198	1.02	Ideal	D	IF	63.0	57.0	15575	6.39	6.35	4.01
26311	1.06	Ideal	D	IF	61.2	57.0	15813	6.57	6.61	4.03
26965	1.07	Ideal	D	IF	60.9	54.0	17042	6.66	6.73	4.08
27226	1.03	Ideal	D	IF	62.0	56.0	17590	6.55	6.44	4.03

Hence the percentage of top quality diamonds can be calculated by means of:

[383]:
```
print('The top quality diamonds percentage is {:.3f} at the average price of {:.
↪2f}$.'.format(topq.size/dmds.size*100,topq['price'].mean()))
```

```
The top quality diamonds percentage is 0.011 at the average price of 15856.67$.
```

Now let's visualize the diamonds with a pivot table to create a hierarchical view by clarity, color and cut:

[384]:
```
thepivot = dmds.
↪pivot_table(values=['carat','price'],index=['clarity','color','cut'],aggfunc=np.
↪sum)
thepivot
```

[384]:

clarity	color	cut	carat	price
I1	D	Fair	7.51	29532
		Good	8.32	27926
		Ideal	12.48	45850
		Premium	13.86	45825
		Very Good	4.75	13114
…			…	…
VVS2	J	Fair	1.01	2998
		Good	12.18	56825
		Ideal	47.01	222584
		Premium	42.57	218394
		Very Good	31.96	172853

[276 rows x 2 columns]

The pivot table is organized as a multi-index dataframe, whose data can be accessed through the .loc as shown in the cell below that reports the total carats and price of all top quality diamonds:

[385]:
```
thepivot.loc['IF','D','Ideal']
```

[385]:
```
carat        17.24
price    183881.00
Name: (IF, D, Ideal), dtype: float64
```

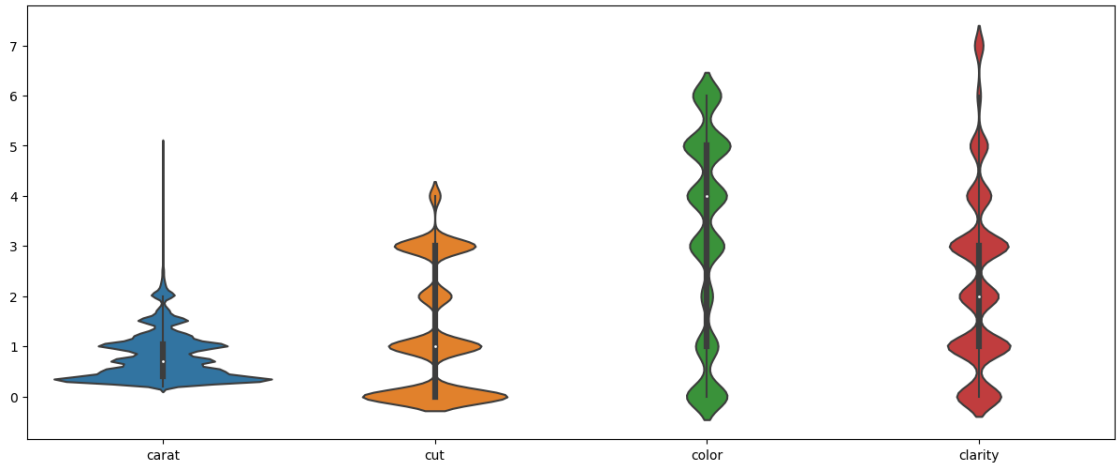
As already shown in the present example, pandas provides its own data plotting features, but it is still possible to extend the plots variety by means of additional packages, such as **seaborn** that will be discussed in further detail in lecture #4.

The following figure displays violin plots for some important features of our dataset.

[386]:
```
import seaborn as sns
import matplotlib.pyplot as plt

fig, ax = plt.subplots(figsize=(15,6))
sns.violinplot(data=data.loc[:,['carat','cut','color','clarity']], ax=ax)
```

[386]:
```
<AxesSubplot:>
```



Finally, it is possible to calculate the correlation matrix, a symmetric matrix that shows the correlation among pairs of data features: values close to 1.0 highlight strong positive correlation between the selected features meaning that both features move in the same direction, whereas negative values imply variations in opposite directions.

[387]:
```
data.loc[:,['carat','cut','color','clarity']].corr()
```

[387]:

	carat	cut	color	clarity
carat	1.000000	0.114426	-0.065386	-0.281218
cut	0.114426	1.000000	-0.029128	-0.118670
color	-0.065386	-0.029128	1.000000	0.032589
clarity	-0.281218	-0.118670	0.032589	1.000000

[ ]: