Chapter 6: pandas in Depth: Data Manipulation

Saul SL

July 2023

1 Introduction

In Pandas two data structures are defined, `Series` and `Data Frame`. The former is a uni-dimensional object similar to an array whereas, the later is designed to hold multi-dimensional data.

```
import pandas as pd
import numpy as np
```

2 Data preparation

2.1 Merging

Merging (`pandas.merge()`) works over rows using a common column. By default only common elements are kept.

```
# Define data frames
# Define data frames
frame1 = pd.DataFrame({
    "id": ["ball", "pencil", "pen", "mug", "ashtray"],
    "price": [12.33, 11.44, 33.21, 13.23, 33.62],
})
frame2 = pd.DataFrame({
    "id": ["pencil", "pencil", "ball", "pen"],
    "color": ["white", "red", "red", "black"],
})

# Merge. It uses 'id'. 'mug' and 'ashtray' from frame1 are excluded
frame1_2_merged = pd.merge(frame1, frame2)
```

```
>>> frame1
        id  price
0     ball  12.33
1   pencil  11.44
2      pen  33.21
3      mug  13.23
4  ashtray  33.62

>>> frame2
       id  color
0  pencil  white
1  pencil    red
2    ball    red
3     pen  black

>>> frame1_2_merged
       id  price  color
0    ball  12.33    red
1  pencil  11.44  white
```

```
2  pencil  11.44    red
3     pen  33.21  black
```

In case there are more than one common column (usually and `ID` column and another one) one has to specify over which column the merge will take place. Note in the examples (`merge_df_id` and `merge_df_brand`) that the results differ depending on which column is used for merging. Note that is also possible to merge on multiple columns but it is likely that the merging method (how=<method> is needed, see below)

```python
# Define data frames with 2 columns in common
frame1 = pd.DataFrame({
    'id': ['ball', 'pencil', 'pen', 'mug', 'ashtray'],
    'color': ['white', 'red', 'red', 'black', 'green'],
    'brand': ['OMG', 'ABC', 'ABC', 'POD', 'POD']
})
frame2 = pd.DataFrame({
    'id': ['pencil', 'pencil', 'ball', 'pen'],
    'brand': ['OMG', 'POD', 'ABC', 'POD']
})

# Merge specifying the column to work on
merged_df_id = pd.merge(frame1, frame2, on='id')
merged_df_brand = pd.merge(frame1, frame2, on='brand')

# Merge using multiple columns (merging method 'how' is needed)
merged_df_all = pd.merge(frame1, frame2, on=['id', 'brand'], how='outer')
```

```
>>> frame1
        id  color brand
0     ball  white   OMG
1   pencil    red   ABC
2      pen    red   ABC
3      mug  black   POD
4  ashtray  green   POD

>>> frame2
       id brand
0  pencil   OMG
1  pencil   POD
2    ball   ABC
3     pen   POD

>>> merged_df_id
       id  color brand_x brand_y
0    ball  white     OMG     ABC
1  pencil    red     ABC     OMG
2  pencil    red     ABC     POD
3     pen    red     ABC     POD

>>> merged_df_brand
      id_x  color brand    id_y
0     ball  white   OMG  pencil
1   pencil    red   ABC    ball
2      pen    red   ABC    ball
3      mug  black   POD  pencil
4      mug  black   POD     pen
5  ashtray  green   POD  pencil
6  ashtray  green   POD     pen
```

In the opposite case where there is no common column but we know that the elements of two of them could be used for merging, it is possible to specify which columns from each data frame will be used.

To keep all the information from each data frame, the merging method should be specified as `outer`. This can be understood as a *union* between two sets. Other methods of merging are `left` and `right`. The former will keep all the elements of the data frame that was mentioned first whereas, the latter will keep all the elements from the second data frame.

```
# rename columns on dataframe2
frame2.columns = ['sid', 'brand']

# Merge specifying which columns to use
pd.merge(frame1, frame2, left_on='id', right_on='sid')

# revert changes in dataframe2
frame2.columns = ['id', 'brand']

# merge using 'outer' method
merged_df_id_outer = pd.merge(frame1, frame2, on='id', how='outer')
```

```
>>> merged_df_id
       id  color brand_x brand_y
0    ball  white     OMG     ABC
1  pencil    red     ABC     OMG
2  pencil    red     ABC     POD
3     pen    red     ABC     POD

>>> merged_df_id_outer
        id  color brand_x brand_y
0     ball  white     OMG     ABC
1   pencil    red     ABC     OMG
2   pencil    red     ABC     POD
3      pen    red     ABC     POD
4      mug  black     POD     NaN
5  ashtray  green     POD     NaN
```

It is also possible to merge using indices rather than values. Note that using the `join()` method uses much less code but requires the columns to be uniqu

```
# merge by index using merge() method
pd.merge(frame1, frame2, right_index=True, left_index=True)

# merge by index using the join() method.
# requires unique columns
frame2.columns = ['id2', 'brand2']
frame1.join(frame2)
```

2.2 Concatenating

Concatenation `pandas.concat()` can be thought as a less stringent form of data combination where the main requirement is the specification of an axis (row, `axis=0` default or column, `axis=1`) over which to perform the combination. The concatenation uses indices, overlapping indices will be kept while missing values will be filled with `NaN`. When concatenating series, the argument `keys` can be used to distinguish the elements from each object

```
# Define series
ser1 = pd.Series(np.random.rand(4), index=[1, 2, 3, 4])
ser2 = pd.Series(np.random.rand(4), index=[5, 6, 7, 8])

# Concatenates series, key arg is used to distinguish them
pd.concat([ser1, ser2], keys=[1, 2])

# Build data frames. Note indices on frame2
frame1 = pd.DataFrame(np.random.rand(9).reshape(3, 3),
                      index=[1, 2, 3],
                      columns=["A", "B", "C"])
frame2 = pd.DataFrame(np.random.rand(9).reshape(3, 3),
                      index=[4, 5, 6],
```

```
                      columns=["A", "B", "C"])
# Combine over rows (axis=0 is not needed, default)
comb_df_rows = pd.concat([frame1, frame2], axis=0)

# Combine over columns
comb_df_cols = pd.concat([frame1, frame2], axis=1)
```

```
>>> comb_df_rows
          A         B         C
1  0.344394  0.131850  0.284119
2  0.047269  0.181124  0.697911
3  0.612473  0.271074  0.851305
4  0.727531  0.486195  0.356041
5  0.147917  0.063598  0.208063
6  0.442059  0.443724  0.656471

>>> comb_df_cols
          A         B         C         A         B         C
1  0.344394  0.131850  0.284119       NaN       NaN       NaN
2  0.047269  0.181124  0.697911       NaN       NaN       NaN
3  0.612473  0.271074  0.851305       NaN       NaN       NaN
4       NaN       NaN       NaN  0.727531  0.486195  0.356041
5       NaN       NaN       NaN  0.147917  0.063598  0.208063
6       NaN       NaN       NaN  0.442059  0.443724  0.656471
```

2.3 Combining

Combine `pandas.DataFrame.combine_first()`, while concatenation will keep overlapping indices, the combine function will only keep the indices from the first object to be combined. Missing indices will be not be filled with `Nan`. On (Nelli 2018, p.193) a way to combine 'parts' two series is mentioned however using this code on newer versions of python (mine being 3.10.11) raises a warning. However, the same result can be obtained with the `iloc()` method.

```
# Define series with overlapping indices
ser1 = pd.Series([10, 20, 30, 40, 50], index=[1, 2, 3, 4, 5])
ser2 = pd.Series([21, 41, 51, 61], index=[2, 4, 5, 7])

# combine series while keeping those of ser1
comb_ser1_2 = ser1.combine_first(ser2)

# combine series focusing on ser2
comb_ser2_1 = ser2.combine_first(ser1)

# Combine parts of ser1 and ser2. Note the command below
# ser1[:3].combine_first(ser2[:3])  # Deprecate
ser1.iloc[:3].combine_first(ser2.iloc[:3])
```

```
>>> comb_ser1_2
1    10.0
2    20.0
3    30.0
4    40.0
5    50.0
6    61.0
dtype: float64

>>> comb_ser2_1
1    10.0
2    21.0
3    30.0
4    41.0
```

```
5    51.0
6    61.0
```

2.4 Pivoting

Pivoting involves changing the orientation of a table, rearranging the data by columns of vice versa. On (Nelli 2018, Ch. 4) this was addressed using the `stack()` and `unstack()` functions which produce a long series and a wide data frame, respectively. Passing the level of the index as an integer signals it to be unstacked (*i.e.* converted to a column).

```python
# stacking and unstacking
frame1 = pd.DataFrame(
    np.arange(9).reshape(3, 3),
    index=["white", "black", "red"],
    columns=["ball", "pen", "pencil"],
)
# Create a long table
frame_stacked = frame1.stack()

# Revert to a wide table
frame_unstacked = frame_stacked.unstack()

# Specify which level to be unstacked
frame_unstacked_l0 = frame_stacked.unstack(0)
frame_unstacked_l1 = frame_stacked.unstack(1)  # same as no specifying level
```

```
>>> frame_stacked
white  ball      0
       pen       1
       pencil    2
black  ball      3
       pen       4
       pencil    5
red    ball      6
       pen       7
       pencil    8
dtype: int64

>>> frame_unstacked_l0
        white  black  red
ball        0      3    6
pen         1      4    7
pencil      2      5    8

>>> frame_unstacked_l1
       ball  pen  pencil
white     0    1       2
black     3    4       5
red       6    7       8
```

Note in the examples that the stacking a data frame produces a **`series`** object. Also, that un-stacking needs a **`series`** object (with hierarchical index) to work on; if a **`dataframe`** is passed the result is not the expected. In order to pivot from a 'long' to a 'wide' format using a data frame the `pivot()` function is used specifying the column that would be used as `index` and the one that would be used as `columns`. Note that in the resulting data frame `columns` will have two levels, `item` and an undefined one.

```python
longframe = pd.DataFrame({
    "color": [
        "white", "white", "white", "red", "red", "red", "black", "black",
        "black"
    ],
    "item": ["ball", "pen", "mug", "ball", "pen", "mug", "ball", "pen", "mug"],
```

```
    "value":
    np.random.rand(9),
})
wideframe = longframe.pivot(index='color', columns='item')
```

```
>>> longframe
   color  item     value
0  white  ball  0.387997
1  white   pen  0.126525
2  white   mug  0.241254
3    red  ball  0.271296
4    red   pen  0.291455
5    red   mug  0.052206
6  black  ball  0.154831
7  black   pen  0.072519
8  black   mug  0.328268

>>> wideframe
          value
item       ball       mug       pen
color
black  0.154831  0.328268  0.072519
red    0.271296  0.052206  0.291455
white  0.387997  0.241254  0.126525

>>> wideframe.columns
MultiIndex([('value', 'ball'),
            ('value',  'mug'),
            ('value',  'pen')],
           names=[None, 'item'])
```

2.5 Removing

Removing columns and rows is carried with the `del` command and the `drop()` function, respectively. Note that the former modifies the data frame passed whereas, the latter does not.

```
# Define dataframe
frame1 = pd.DataFrame(np.arange(9).reshape(3, 3),
                      index=['white', 'black', 'red'],
                      columns=['ball', 'pen', 'pencil'])

# Remove column
del frame1['ball']

# Remove row
frame1_norow = frame1.drop('white')
```

```
>>> frame1
       pen  pencil
white    1       2
black    4       5
red      7       8

>>> frame1_norow
       pen  pencil
black    4       5
red      7       8
```

3 Data transformation

3.1 Removing duplicates

The duplicated function returns a boolean where true values indicate the presence of a duplicate; the first occurrence is reported as false. This result can be used to remove duplicates using index. Note that the resulting boolean (`duplicates`) needs to be negated. Alternatively, the function, `drop_duplicates()` returns a data frame with only unique rows.

```
dframe = pd.DataFrame({
    'color': ['white', 'white', 'red', 'red', 'white'],
    'value': [2, 1, 3, 3, 2]
})

# Find duplicates. Returns a bool
duplicates = dframe.duplicated()
dframe_dedup = dframe[~duplicates]
dframe_dedup2 = dframe.drop_duplicates()  # same as dframe_dedup
```

```
>>> dframe
   color  value
0  white      2
1  white      1
2    red      3
3    red      3
4  white      2

>>> dframe_dedup
   color  value
3    red      3
4  white      2
```

3.2 Mapping

> ”Mapping is nothing more than the creation of a list of matches between two different values, with the ability to bind a value to a particular label or string.” (Nelli 2018, p.199)

3.2.1 Replace

A dictionary is created with keys corresponding to data frame values that we want to replace and dictionary values with the new values. These are applied to the data frame using the `replace()` function.

```
frame = pd.DataFrame({
    'item': ['ball', 'mug', 'pen', 'pencil', 'ashtray'],
    'color': ['white', 'rosso', 'verde', 'black', 'yellow'],
    'price': [5.56, 4.20, 1.30, 0.56, 2.75]
})

newcolors = {'rosso': 'red', 'verde': 'green'}

frame_v2 = frame.replace(newcolors)
```

```
>>> frame
      item   color  price
0     ball   white   5.56
1      mug   rosso   4.20
2      pen   verde   1.30
3   pencil   black   0.56
4  ashtray  yellow   2.75
```

```
>>> frame_v2
      item   color  price
0     ball   white   5.56
1      mug     red   4.20
2      pen   green   1.30
3   pencil   black   0.56
4  ashtray  yellow   2.75
```

In the example above `replace()` was used to replace colors written in another language; other uses could be replacing `NaN` values.

3.2.2 Adding values

Mapping can also be used to add another column (values of the dictionary) based on matching the dictionary keys to column values. This is performed with the `map()` function

```
# Define a dictionary with item: prices
prices = {
    'ball': 5.56,
    'mug': 4.20,
    'bottle': 1.30,
    'scissors': 3.41,
    'pen': 1.30,
    'pencil': 0.56,
    'ashtray': 2.75
}

# create a new 'price' column and match the items to prices
frame['price'] = frame['item'].map(prices)
```

```
>>> frame
      item   color  price
0     ball   white   5.56
1      mug   rosso   4.20
2      pen   verde   1.30
3   pencil   black   0.56
4  ashtray  yellow   2.75
```

3.2.3 Rename the index of axes

Similarly to adding values, indices can be modified with the `rename()` function. Using the `index` and `columns` arguments both indices can be modified. Note that the changes are stored in new variables, to modify the same data frame use the `inplace=True` option. Also, for simple changes the dictionary with maps can be passed to function itself.

```
# Dictionary with new indices
reindex = {0: 'first', 1: 'second', 2: 'third', 3: 'fourth', 4: 'fifth'}
recolumn = {'item': 'object', 'price': 'value'}

# Rename indices
frame_v2 = frame.rename(reindex)

# Rename indices and columns
frame_v3 = frame.rename(index=reindex, columns=recolumn)

# Inplace replacement
frame.rename(columns={'item': 'object'}, inplace=True)

# map defined in function
frame.rename(index={1: 'first', 2: 'dos'}, columns={'item': 'object'})
```

```
frame_v2
           item   color  price
first      ball   white   5.56
second      mug   rosso   4.20
third       pen   verde   1.30
fourth   pencil   black   0.56
fifth   ashtray  yellow   2.75

>>> frame_v3
    object   color  value
0     ball   white   5.56
1      mug   rosso   4.20
2      pen   verde   1.30
3   pencil   black   0.56
4  ashtray  yellow   2.75
```

3.3 Discretization and Binning

Discretization involved grouping a series of data into discrete categories. This could be useful to handle large quantity of data. The pandas function `cut()` will return a categorical array defined by the second parameter (`bins`). The values of the input array are coded in integers between 0 and the number of defined bins.

If instead of defining the edges of the bins an integer is provided as the second argument of the `cut()` function, then the edges of the bins will be calculated based on the minimum and maximum values of the array.

In addition to the `cut()` function there is the `qcut()` one which divides the input array into quantiles.

```
# generate a list of 100 random integers between 1:100
random_integers = np.random.randint(1, 101, size=100)

# define categories (bins)
bins = [0, 25, 50, 75, 100]

# divide (cut) the array into the defined bins
cat = pd.cut(random_integers, bins)

# Add labels to bins
bin_names = ['unlikely', 'less likely', 'likely', 'highly likely']
cat_labels = pd.cut(random_integers, bins, labels=bin_names)

# Divide results based on the min and max value into n intervals
cat_2 = pd.cut(random_integers, 5)

# check that the categories match the bins
cat.categories

# check that the code numbers match the number of categories
pd.Series(cat.codes).unique()

# Count the number of ocurrences
pd.value_counts(cat)

# Divide the array into quartiles
quartiles = pd.qcut(random_integers, 4)
```

```
>>> cat.categories
IntervalIndex([(0, 25], (25, 50], (50, 75], (75, 100]], dtype='interval[int64, right]')

>>> pd.Series(cat.codes).unique()
array([3, 1, 2, 0], dtype=int8)

>>> pd.value_counts(cat)
```

```
(25, 50]     34
(50, 75]     23
(75, 100]    23
(0, 25]      20

>>> pd.value_counts(quartiles)
(0.999, 29.5]     25
(29.5, 46.0]      25
(46.0, 71.25]     25
(71.25, 100.0]    25
dtype: int64
```

3.4 Detecting and Filtering Outliers

The example provided creates a data frame with 3 columns and 1000 rows filled with random numbers. Then it filters those that have an absolute value that is higher than 3 times the standard deviation. What is interesting is the use of the `any()` function which returns a boolean over the specified axis (1=columns). From the documentation of the function, *"Returns False unless there is at least one element within a series or along a Dataframe axis that is True or equivalent"* That is if one of the comparison between the cell value and the threshold (3 times the standard deviation) is carried column wise, then for the resulting data of boolean, if any of the row values is true it returns `True`. This can be seen comparing the threshold value to the filtered data frame. For example, for the first row only the cell corresponding to the second column is higher than the threshold. If the `any()` function is omitted then the test would require all the row values to be true to return true

```python
# Create a working data frame
randframe = pd.DataFrame(np.random.randn(1000, 3))

# Calculate threshold
thr = 3 * randframe.std()

# Filter
randframe_3sd = randframe[(np.abs(randframe) > thr).any(axis=1)]

# Modify data frame to add a row with all values above threshold
randframe_mod = randframe.copy()  # Makes an independent copy
randframe_mod.loc[1] = pd.Series([4, 4, 4])  # Add custom values
randframe_mod_3sd = randframe_mod[(np.abs(randframe_mod)
                                   > thr)]  # Uses same threshold
randframe_mod_3sd = randframe_mod_3sd.dropna()  # Removes NaN
```

```
>>> thr
0    2.990998
1    2.947070
2    2.990297
dtype: float64

>>> randframe_3sd
            0         1         2
198 -0.885098 -3.161628 -0.830367
458  0.643790  3.373517 -0.577254
531 -3.014660 -0.275209  0.618381
672  0.205011  2.997098  0.525500
675  0.320883 -0.654335 -3.080121

>>> randframe_mod_3sd
     0    1    2
1  4.0  4.0  4.0
```

3.5 Reordering and sub-sampling

To re-order the rows of a data frame randomly, an array that represents the index's values (in random order) is created used the `np.random.permutation()` function and then, mapped to the working data frame using the `take()` function

```
# Working data frame
nframe = pd.DataFrame(np.arange(35).reshape(7, 5))

# Array of index in random order
new_order = np.random.permutation(7)

# Re-order data frame
nframe_v2 = nframe.take(new_order)
```

```
>>> nframe
    0   1   2   3   4
0   0   1   2   3   4
1   5   6   7   8   9
2  10  11  12  13  14
3  15  16  17  18  19
4  20  21  22  23  24
5  25  26  27  28  29
6  30  31  32  33  34

>>> new_order
array([0, 3, 2, 1, 6, 4, 5])

>>> nframe_v2
    0   1   2   3   4
0   0   1   2   3   4
3  15  16  17  18  19
2  10  11  12  13  14
1   5   6   7   8   9
6  30  31  32  33  34
4  20  21  22  23  24
5  25  26  27  28  29
```

The `take()` function can also be applied to obtain a random sub-sample. In this case the array of index values is obtained with the `np.random.randint()`

```
# Define and apply subsample
subsample = np.random.randint(0, len(nframe), size=3)
nframe_sub = nframe.take(subsample)
```

```
>>> nframe_sub
    0   1   2   3   4
5  25  26  27  28  29
1   5   6   7   8   9
0   0   1   2   3   4
```

3.6 String Manipulation

Simple methods for working with strings are presented. These include

- Split specifying a character (`,` in the example). The result usually contains spaces which are removed with the `strip()` function.
- Concatenate or join, the former uses the `+` symbol and string objects, the latter specifies the character followed by the `join()` function and a list of strings.

- Search, depending on the method it may return a boolean, or the index value of the string that was search (start of string if is longer than 1). Note that `index()` will raise an error if the string is not found.
- Count, returns an integer with the number of occurrences. Zero if not found
- Replace, takes the string to be replaced and its replacement. Unchanged if not found.

```
# Split string on a character
text = '16 Bolton Avenue , Boston'
text_split = [s.strip() for s in text.split(',')]
address, city = [s.strip() for s in text.split(',')]

# concatenate - join strings
text_concat = address + ',' + city  # Cumbersome for many objects
text_join = ','.join(text_split)  # Requires a list

# search for text
'Boston' in text  # Returns a boolean
text.index('Boston')  # Returns index, error if not found
text.find('Boston')  # Returns index, '-1' if not found

# Count ocurrences
text.count('e')  # Returns '2'
text.count('Avenue')  # Returns '1'

# Replace strings
text_v1 = text.replace('Avenue', 'Street')
text_v2 = text.replace('1', '')  # Deletes '1'
```

```
>>> text
'16 Bolton Avenue , Boston'

>>> text_concat
'16 Bolton Avenue,Boston'

>>> text_v1
'16 Bolton Street , Boston'

>>> text_v2
'6 Bolton Avenue , Boston'
```

3.7 Regular Expressions

The examples provide a brief overview of the uses of python's `re` module for processing text using regular expressions (*regex*). As defined in Nagy (2018, p.2) "a regex is a finite character sequence defining a search pattern". The character sequence is formed according to the regular expression dialect which in python (as in many other languages) corresponds to PCRE (perl compatible regular expression). Section **??** list the functions and regular expressions available in the `re` module. The first three functions (`findall()`, `search()=/=match()` and `split()`) are used in the examples along with the `compile()` function. The latter is used to produce a regex object that can be re-used for faster performance. As for the regular expressions, the examples show the use of `\s` which matches any white space followed by `+` which indicates one or more occurrences of the previous character. Another example uses the `[A,a]` which matches the letter 'a' (either lower or upper case) followed by any word character that is present one or more times (`\w+`), this is used to match the words, Avenue and address.

Things worth noting, in the text used in the book ('This is my address: 16 Bolton Avenue, Boston') there are no words that have a letter a in the middle. When the text is modified to include other occurrences of the letter 'a' ('This a temporary address: 16 Bolton Avenue, Boston') the regular expression does not work any more. Furthermore if we (for some reason) would like to get all the strings starting with the letter 'a' or 'A', whether they occur at the beginning of the word or not, then a new regular expression needs to be used. This illustrates some of the complexities of working with regular expressions.

```python
import re

# Dummy example of a text with white spaces
text = "This is   an\t odd  \n text!"

# Remove spaces (split() function)
text_noSpaces = re.split('\s+', text)

# Compile a regular expression for re-use
regex = re.compile('\s+')

# Remove spaces (similar to text_noSpaces)
text_noSpaces_v2 = regex.split(text)

# Define text
# address = 'This is my address: 16 Bolton Avenue, Boston'  # Original sentence
address = 'This a temporary address: 16 Bolton Avenue, Boston'

# Find all words starting with 'A' or 'a'
# text_aA = re.findall('[A,a]\w+',text) # Doesn't work
pattern1 = r'\b[Aa]\w*'
text_aA = re.findall(pattern1, address)

# Find all the words containing an 'a' or 'A'
pattern2 = r'\b[Aa]\w+|a\w+|a'
text_aA_v2 = re.findall(pattern2, address)

# Find the first occurrence of a word that starts with A or a
index_aA_first = re.search(pattern1, address)  # returns a match object

# Extract the first match
text_aA_first = address[index_aA_first.start():index_aA_first.end()]

# Search only at the beginning of the string (match() function)
text_T_first = re.match('T\w+', text)
text_T_first_v2 = re.search('^T\w+', address)  # same as text_T_first
```

```
>>> text_noSpaces
['This', 'is', 'an', 'odd', 'text!']

>>> text_noSpaces_v2
['This', 'is', 'an', 'odd', 'text!']

>>> address
'This a temporary address: 16 Bolton Avenue, Boston'

>>> pattern1
'\\b[Aa]\\w*'

>>> text_aA
['a', 'address', 'Avenue']

>>> pattern2
'\\b[Aa]\\w+|a\\w+|a'

>>> text_aA_v2
['a', 'ary', 'address', 'Avenue']
```

4 Data aggregation

As described in (Nelli 2018, p.217), "data aggregation involves a transformation that produces a single integer from an array", examples include calculating the mean of a series of numbers. Aggregation often includes a step of separating data into categories before applying a function, for example calculate the total price of notebooks grouped by paper size (**??**). These steps (*split, apply and combine*) are carried in part through the function `GroupBy`.

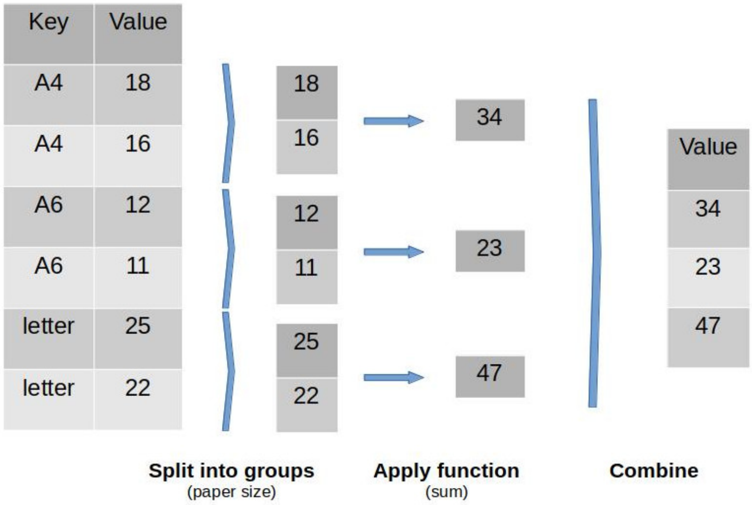

Figure 1: The split-apply-combine mechanism applied to a series for notebook prices based on paper size (Based on Nelli (2018))

`GroupBy` takes a `series` or a `dataframe` and produces a `group` object based on the series' index, another series or even a dictionary. The first example uses frame's `price1` series and groups it based on the values of the `color` column. The keyword `by` as well as, `axis=0` can be omitted as the first argument is supposed to be the criteria for grouping and it is used by default on indices (`axis=0`). The second example, shows how multiple columns can be selected, either explicitly or implicitly. In the latter case it is recommended to pass the argument `numeric_only=True` to include only `float`, `int`, or `boolean` columns.

```
# Define working dataframe
frame = pd.DataFrame({
    'color': ['white', 'red', 'green', 'red', 'green'],
    'object': ['pen', 'pencil', 'pencil', 'ashtray', 'pen'],
    'price1': [5.56, 4.20, 1.30, 0.56, 2.75],
    'price2': [4.75, 4.12, 1.60, 0.75, 3.15]
})

# Group price1 column based on color
group = frame['price1'].groupby(by=frame['color'], axis=0)

# Calculate the mean price of objects grouped by color
price1_mean_color = group.mean()

# Group multiple columns explicitly or implicitly
# The second raises a warning if the mean argument is not provided
prices_mean_color = frame[['price1', 'price2']].groupby(frame['color']).mean()
prices_mean_color_v2 = frame.groupby(frame['color']).mean(numeric_only=True)
```

```
>>> frame
   color   object  price1  price2
0  white      pen    5.56    4.75
1    red   pencil    4.20    4.12
2  green   pencil    1.30    1.60
3    red  ashtray    0.56    0.75
4  green      pen    2.75    3.15

>>> price1_mean_color
color
green    2.025
```

```
red      2.380
white    5.560
Name: price1, dtype: float64

>>> prices_mean_color
       price1  price2
color
green   2.025   2.375
red     2.380   2.435
white   5.560   4.750
```

The next example show how the group object is composed of two tupples, one for the name of the group and the other for the values contained in the group. More importantly, it shows how can one iterate over these objects and potentially change their value by applying some function

```python
group2 = frame.groupby('color')
for name, group in group2:
    print(name)
    group['total'] = group['price1'] + group['price2']
    print(group)
```

```
green
   color  object  price1  price2  total
2  green  pencil    1.30    1.60    2.9
4  green     pen    2.75    3.15    5.9
red
  color   object  price1  price2  total
1   red   pencil    4.20    4.12   8.32
3   red  ashtray    0.56    0.75   1.31
white
   color object  price1  price2  total
0  white    pen    5.56    4.75  10.31
```

Then, it is shown how a given column can be selected at different stages of the split-apply-combine process. All the commands yield the same result.

```python
# Select column before grouping
mean_1 = frame['price1'].groupby(frame['color']).mean()

# Select column after grouping
mean_2 = frame.groupby(frame['color'])['price1'].mean()

# Select column after applying a function
mean_3 = (frame.groupby(frame['color']).mean(numeric_only=True))['price1']
```

```
>>> mean_1
color
green    2.025
red      2.380
white    5.560
Name: price1, dtype: float64

>>> mean_2
color
green    2.025
red      2.380
white    5.560
Name: price1, dtype: float64

>>> mean_3
color
green    2.025
```

```
red      2.380
white    5.560
```

The next example shows how more than one function (including custom ones) can be applied to a group object. This is done with the `agg()` function passing a list of functions or functions' names; the latter passed as a string (*e.g.* `'mean'`).

```
# working group object
group = frame.groupby('color')


# Define custom function
def range(series):
    return series.max() - series.min()


# apply multiple functions with agg()
isummary = group['price1'].agg(['mean', 'std', range])
```

```
>>> isummary
        mean       std  range
color
green  2.025  1.025305   1.45
red    2.380  2.573869   3.64
white  5.560       NaN   0.00
```

The last part of the section and the chapter briefly discuss the use of the functions `transform()` and `apply()`. The former applies a function on a series or data frame producing an object with the same axis shape as the input data frame. In the example a data frame with the sum of prices grouped by color; these can then be merged on the original data frame. As for the `apply()` function, the example also uses a `lambda()` function which allows to define an anonymous function in the same line, in this case one for calculating the maximum value. The result is a dataframe with a hierarchical index which is redundant with the columns thus, it is removed first by dropping a level and then, renaming the index.

```
# Working dataframe 1
frame = pd.DataFrame({
    'color': ['white', 'red', 'green', 'red', 'green'],
    'price1': [5.56, 4.20, 1.30, 0.56, 2.75],
    'price2': [4.75, 4.12, 1.60, 0.75, 3.15]
})

# Calculate totals per color
totals = frame.groupby('color').transform(np.sum).add_prefix('tot_')

# Merge totals to working dataframe
frame_updated = pd.merge(frame, totals, right_index=True, left_index=True)

# Working dataframe 2
frame2 = pd.DataFrame({
    'color': ['white', 'black', 'white', 'white', 'black', 'black'],
    'status': ['up', 'up', 'down', 'down', 'down', 'up'],
    'value1': [12.33, 14.55, 22.34, 27.84, 23.40, 18.33],
    'value2': [11.23, 31.80, 29.99, 31.18, 18.25, 22.44]
})

# Calculate max value per color and status
max_values = frame2.groupby(['color', 'status'
                             ]).apply(lambda x: x.max()).add_prefix('max_')

# Modify multi-index as it is redundant with columns
```

```python
max_values = max_values.droplevel(0)
max_values.index = range(4)
```

```
>>> frame_updated
   color  price1  price2  tot_price1  tot_price2
0  white    5.56    4.75        5.56        4.75
1    red    4.20    4.12        4.76        4.87
2  green    1.30    1.60        4.05        4.75
3    red    0.56    0.75        4.76        4.87
4  green    2.75    3.15        4.05        4.75

>>> max_values
  max_color max_status  max_value1  max_value2
0     black       down       23.40       18.25
1     black         up       18.33       31.80
2     white       down       27.84       31.18
3     white         up       12.33       11.23
```

5 References

References

Nagy, Zsolt (2018). *Regex Quick Syntax Reference: Understanding and Using Regular Expressions*. 1st ed. 2018. Berkeley, CA: Apress : Imprint: Apress. 1 p. ISBN: 978-1-4842-3876-9. DOI: 10.1007/978-1-4842-3876-9.

Nelli, Fabio (2018). *Python Data Analytics: With Pandas, NumPy, and Matplotlib*. 2nd ed. 2018. Berkeley, CA: Apress : Imprint: Apress. 1 p. ISBN: 978-1-4842-3913-1. DOI: 10.1007/978-1-4842-3913-1.

6 List of `re` defined regular expressions and functions

Obtained from w3schools

Table 1: Regex functions

Function	**Description**
`findall`	Returns a list containing all matches
`search`	Returns a Match object if there is a match anywhere in the string
`match`	Returns a Match object if there is a match at the beginning of the string
`split`	Returns a list where the string has been split at each match
`sub`	Replaces one or many matches with a string

Table 2: Metacharacters

Character	Description	Example
`[]`	A set of characters	`[a-m]`
`\`	Signals a special sequence (can also be used to escape special characters)	`\d`
`.`	Any character (except newline character)	`he..o`
`^`	Starts with	`^hello`
`$`	Ends with	`planet$`
`*`	Zero or more occurrences	`he.*o`
`+`	One or more occurrences	`he.+o`
`?`	Zero or one occurrences	`he.?o`
`{}`	Exactly the specified number of occurrences	`he.{2}o`
|	Either or	falls| stays
`()`	Capture and group	

Table 3: Special sequences

Character	Description	Example
`\A`	Returns a match if the specified characters are at the beginning of the string	`\AThe`
`\b`	Returns a match where the specified characters are at the beginning or at the end of a word (the `r` in the beginning is making sure that the string is being treated as a "raw string")	`r'\bain' r'ain\b'`
`\B`	Returns a match where the specified characters are present, but NOT at the beginning (or at the end) of a word (the `r` in the beginning is making sure that the string is being treated as a "raw string")	`r'\Bain' r'ain\B'`
`\d`	Returns a match where the string contains digits (numbers from 0-9)	`\d`
`\D`	Returns a match where the string DOES NOT contain digits	`\D`
`\s`	Returns a match where the string contains a white space character	`\s`
`\S`	Returns a match where the string DOES NOT contain a white space character	`\S`
`\w`	Returns a match where the string contains any word characters (characters from a to Z, digits from 0-9, and the underscore _ character)	`\w`
`\W`	Returns a match where the string DOES NOT contain any word characters	`\W`
`\Z`	Returns a match if the specified characters are at the end of the string	`Spain\Z`

Table 4: Sets

Set	Description
`[arn]`	Returns a match where one of the specified characters (a, r, or n) is present
`[a-n]`	Returns a match for any lower case character, alphabetically between a and n
`[^arn]`	Returns a match for any character EXCEPT a, r, and n
`[0123]`	Returns a match where any of the specified digits (0, 1, 2, or 3) are present
`[0-9]`	Returns a match for any digit between 0 and 9
`[0-5][0-9]`	Returns a match for any two-digit numbers from 00 and 59
`[a-zA-Z]`	Returns a match for any character alphabetically between a and z, lower case OR upper case
`[+]`	In sets, `+`, `*`, `.`, | , `()`, `$`,={}= has no special meaning, so `[+]` means: return a match for any `+` character in the string