Pandas

November 22, 2020

1 *Pandas*

1.1 Introduction

Pandas is a *python* library that provides high-performance, easy-to-use data structures and **data analysis tools**.

These structures heavily rely on *NumPy* and its arrays and they are suitable for:

- Tabular data with heterogeneously-typed columns
- Ordered and unordered time series data
- Arbitrary matrix data

Among others, *pandas* can read data from Excel spreadsheets, CSV or TSV files of even from SQL databases.

Pandas is conventionally imported as ***pd***

[1]:
```
import pandas as pd
```

1.2 Data structures

The main data structures of *Pandas* are the *Series* and the *DataFrame*.

1.2.1 *Series*

The *Series* is just a **one-dimensional labeled array** of any data type.

Series behaves as a **dict-like** object. The **axis labels** are collectively referred to as the ***index***.

The *index* is used to get and set values and to align the values when operations between *Series* are performed. In this case, the series do not need to have the same length.

Indexes may not be unique, but operations that require unicity will raise an exception.

Note that operations such as slicing will also slice the *index*.

Under the hood, *Series* are an **extended** *ndarray*, which makes it a valid argument to most *NumPy* functions, but also makes it **fixed size**.

Series may have a name. This is useful, for instance, to identify them.

***Series* creation** You just need some *data* or the shape of the *Series* to create one.

If no *index* is provided at creation, one will be created having contiguous integer values starting from zero (known as *RangeIndex*).

Note that indexes can be longer than the actual data size and that **missing values** will be marked as *NaN*.

Data can be provided in many ways. Let's see some with a few examples.

From a scalar

[2]:
```
series = pd.Series(data=4)
print("Series with default index:")
display(series)
```

```
Series with default index:

0    4
dtype: int64
```

From a list or *ndarray*

[3]:
```
series = pd.Series(list(range(5)), index=['a', 'b', 'c', 'd', 'e'])
display(series)
```

```
a    0
b    1
c    2
d    3
e    4
dtype: int64
```

[4]:
```
import numpy as np

series = pd.Series(np.fromiter(range(5), dtype=int), index=['a', 'b', 'c', 'd',␣
 ↪'e'])
display(series)
```

```
a    0
b    1
c    2
d    3
e    4
dtype: int64
```

From a dictionary *Series* index and values will be the key-values pair of the dictionary.

[5]:
```
# Create a dictionary with zip
keys = ['a', 'b', 'c', 'd', 'e']
values = list(range(5))
```

```
dictionary = dict(zip(keys, values))
print("Dict:", dictionary)

series = pd.Series(dictionary)
print("Series:")
display(series)
```

```
Dict: {'a': 0, 'b': 1, 'c': 2, 'd': 3, 'e': 4}
Series:
```

```
a    0
b    1
c    2
d    3
e    4
dtype: int64
```

Note that missing values are supported!

[6]:
```
# Remove key 'c'
del dictionary["c"]
print("Dict:", dictionary)

# Pass the dictionary and the full keys as arguments
series = pd.Series(dictionary, index=keys)
print("Series:")
display(series)
```

```
Dict: {'a': 0, 'b': 1, 'd': 3, 'e': 4}
Series:
```

```
a    0.0
b    1.0
c    NaN
d    3.0
e    4.0
dtype: float64
```

1.2.2 *DataFrame*

The *DataFrame* is a **two-dimensional labeled data structure** that resembles a spreadsheet, a SQL table or a dictionary of *Series*.

The **row labels** are collectively referred to as the *index* and the columns can potentially be of different types.

Note that while rows cannot be added or removed (because of the *ndarray* implementation), columns can be added\removed any time.

Arithmetic operations between *DataFrames* align on both row and column labels.

***DataFrame* creation** Just like *Series*, *DataFrames* can be created in many ways.

One way is to use the *DataFrame(data=None, index=None, columns=None, dtype=None, copy=False)* class.

If *data* argument is an iterable of iterables, each entry is used to fill the rows.

The *index* (row labels) or *columns* (column labels) arguments can be used for both naming the rows and columns and filtering (if *data* is a dict of dicts). In the latter case, the final *DataFrame* will not include all data not matching.

From a list of *Series*, ndarrays, lists or tuples

[7]:
```
rows = []
for i in range(4):
    rows.append(list(range(5)))
    np.random.shuffle(rows[i])

print("Rows:\n", rows)

dataframe = pd.DataFrame(data=rows)
print("DataFrame from list of lists:")
display(dataframe)
```

```
Rows:
 [[1, 3, 0, 4, 2], [0, 4, 2, 1, 3], [0, 2, 3, 4, 1], [1, 4, 2, 3, 0]]
DataFrame from list of lists:
```

	0	1	2	3	4
0	1	3	0	4	2
1	0	4	2	1	3
2	0	2	3	4	1
3	1	4	2	3	0

Let's provide the index and column names.

[8]:
```
dataframe = pd.DataFrame(rows,
                         index=["A", "B", "C", "D"],
                         columns=["one", "two", "three", "four", "five"]
                        )
# print("DataFrame from list of lists with indices and labels:")
display(dataframe)
```

	one	two	three	four	five
A	1	3	0	4	2
B	0	4	2	1	3
C	0	2	3	4	1
D	1	4	2	3	0

From a dict of *Series* or dicts Let's init a *DataFrame* from dictionaries

[9]:
```
# Init two dictionaries
keys = ['a', 'b', 'c', 'd', 'e']
dictionary1 = dict(zip(keys, list(range(5))))
dictionary2 = dict(zip(keys, list(range(5, 10))))

# Init a column dictionary
columns = {
    "one": dictionary1,
    "two": dictionary2
}

# From a dictionary of dictionaries
dataframe = pd.DataFrame(columns)
```

What if we provide all the column names with some order?

[10]:
```
# We can provide column order
dataframe = pd.DataFrame(data=columns, columns=["two", "one"])
display(dataframe)
```

	two	one
a	5	0
b	6	1
c	7	2
d	8	3
e	9	4

Let's try to filter the rows and the columns.

[11]:
```
# We can provide the only rows we're interested in.
dataframe = pd.DataFrame(data=columns, index=["a", "b"])
display(dataframe)
```

	one	two
a	0	5
b	1	6

[12]:
```
# We can provide the only columns we're interested in.
dataframe = pd.DataFrame(data=columns, columns=["two"])
display(dataframe)
```

	two
a	5
b	6
c	7
d	8
e	9

What about a dictionary of *Series*?

[13]:
```
for key, value in columns.items():
    columns[key] = pd.Series(value)

dataframe = pd.DataFrame(columns)
display(dataframe)
```

	one	two
a	0	5
b	1	6
c	2	7
d	3	8
e	4	9

From other sources Pandas has connectors and helpers to read data from many sources.

For instance, you can read CSV or Excel files, SQL databases, and more.

Check the docs for more.

1.3 *DataFrame* selection

How can we select data from *DataFrame*'s columns and/or rows?

1.3.1 Columns selection

You can access the columns of a *DataFrame* by using the indexing ([]) operator.

Column selection can be performed by label. If you provide:

- a single label, it will return the *Series* of that column
- multiple labels, it will return a sub-*DataFrame* with specified columns

[14]:
```
# Reinit the dataframe
rows = []
for i in range(4):
    rows.append(list(range(5)))
    np.random.shuffle(rows[i])

dataframe = pd.DataFrame(rows,
                         index=["A", "B", "C", "D"],
                         columns=["one", "two", "three", "four", "five"]
                        )

display(dataframe)
```

	one	two	three	four	five
A	0	2	1	3	4
B	0	2	4	1	3
C	2	4	0	3	1
D	1	2	3	4	0

Get the column 'one'

[15]:
```
display(dataframe["one"])

print("What type is the result?", type(dataframe["one"]))
```

```
A    0
B    0
C    2
D    1
Name: one, dtype: int64
```

```
What type is the result? <class 'pandas.core.series.Series'>
```

Get the columns 'one' and 'two'

[16]:
```
display(dataframe[["one", "two"]])

print("What type is the result?", type(dataframe[["one", "two"]]))
```

```
   one  two
A    0    2
B    0    2
C    2    4
D    1    2
```

```
What type is the result? <class 'pandas.core.frame.DataFrame'>
```

1.3.2 Rows selection

Row selection can be performed using the *.loc[]* property.

Again, row selection can be performed by (index) label. If you provide:

- a single index, it will return the *Series* of that row
- multiple indexes, it will return a sub-*DataFrame* with specified rows

You can also specify what columns to get as a second positional argument.

Single index label Let's get the 'A' row…

[17]:
```
display(dataframe.loc["A"])
```

```
one      0
two      2
three    1
four     3
five     4
Name: A, dtype: int64
```

… and filter the columns

[18]:
```
display("Indexing: 'A' row and get column 'two'", dataframe.loc["A", "two"])
display("Indexing: 'A' row and get columns ['one', 'two']", dataframe.loc["A",␣
↪["one", "two"]])
```

```
"Indexing: 'A' row and get column 'two'"
```

```
2
```

```
"Indexing: 'A' row and get columns ['one', 'two']"
```

```
one    0
two    2
Name: A, dtype: int64
```

List of index labels Let's get the 'A' and 'B' rows...

[19]:
```
display(dataframe.loc[["A", "B"]])
```

	one	two	three	four	five
A	0	2	1	3	4
B	0	2	4	1	3

Slicing on index labels *DataFrame*s support slicing!

Let's fetch the rows from 'A' to 'C'.

[20]:
```
display(dataframe.loc["A":"C"])
```

	one	two	three	four	five
A	0	2	1	3	4
B	0	2	4	1	3
C	2	4	0	3	1

Boolean indexing *DataFrames* also support boolean indexing.

Filtering with a boolean *DataFrame*

Let's begin with filtering every entry independently.

[21]:
```
# Get a boolean mask, just like we would using NumPy
indexing_df = dataframe > 1

print("Boolean DataFrame > 1")
display(indexing_df)

print("What type is the filtering index?", type(indexing_df))
display(indexing_df.dtypes)
```

```
Boolean DataFrame > 1
```

```
     one   two  three   four   five
A  False  True  False   True   True
B  False  True   True  False   True
C   True  True  False   True  False
D  False  True   True   True  False
```

```
What type is the filtering index? <class 'pandas.core.frame.DataFrame'>
```

```
one      bool
two      bool
three    bool
four     bool
five     bool
dtype: object
```

[22]:
```
print("Boolean indexing with the DataFrame")
display(dataframe[indexing_df])
```

```
Boolean indexing with the DataFrame
```

```
   one  two  three  four  five
A  NaN    2    NaN   3.0   4.0
B  NaN    2    4.0   NaN   3.0
C  2.0    4    NaN   3.0   NaN
D  NaN    2    3.0   4.0   NaN
```

Filtering with a boolean *Series*

Let's filter the *DataFrame* depending on the values of a given column.

[23]:
```
# Get a boolean mask, just like we would using NumPy
indexing_series = dataframe["four"] > 1

print("Boolean Series > 1")
display(indexing_series)

print("What type is the filtering index?", type(indexing_series))
display(indexing_series.dtype)
```

```
Boolean Series > 1
```

```
A     True
B    False
C     True
D     True
Name: four, dtype: bool
```

```
What type is the filtering index? <class 'pandas.core.series.Series'>
```

```
dtype('bool')
```

[24]:
```
print("Boolean indexing with the Series")
display(dataframe[indexing_series])
```

```
Boolean indexing with the Series
```

	one	two	three	four	five
A	0	2	1	3	4
C	2	4	0	3	1
D	1	2	3	4	0

Filter the rows

We can also filter the rows according to some condition. For instance:

[25]:
```
# Let's get the rows where column 'two' is > 1
indexing_series = dataframe.loc[:, "two"] > 0
display(indexing_series)

print("What type is the filtering index?", type(indexing_series))
display(indexing_series.dtype)
```

```
A    True
B    True
C    True
D    True
Name: two, dtype: bool
```

```
What type is the filtering index? <class 'pandas.core.series.Series'>
```

```
dtype('bool')
```

[26]:
```
print("Boolean indexing with the Series")
display(dataframe[indexing_series])
```

```
Boolean indexing with the Series
```

	one	two	three	four	five
A	0	2	1	3	4
B	0	2	4	1	3
C	2	4	0	3	1
D	1	2	3	4	0

Position You can also access rows by integer index (position) using the *.iloc* attribute.

Integer indexes are 0-based and may be tricky. Other selection methods should be preferred.

As a trivial example, if we want to get the second entry we can…

[27]:
```
print("DataFrame at position 1:")
display(dataframe.iloc[1])
```

```
DataFrame at position 1:
```

```
one      0
two      2
three    4
four     1
five     3
Name: B, dtype: int64
```

1.4 *DataFrame* manupulation

After an overview on data creation and selection, let's introduce some data manipulation.

1.4.1 Show index and columns

How can we show index and column names?

DataFrames have the *.index* and *.columns* attributes that may also be set.

[28]:
```
print("Index:", dataframe.index)
print("Columns:", dataframe.columns)
```

```
Index: Index(['A', 'B', 'C', 'D'], dtype='object')
Columns: Index(['one', 'two', 'three', 'four', 'five'], dtype='object')
```

1.4.2 Get *ndarray*

What if we want to get the underlying *ndarray*?

We can use the *.to_numpy()* method

[29]:
```
display(dataframe.to_numpy())
```

```
array([[0, 2, 1, 3, 4],
       [0, 2, 4, 1, 3],
       [2, 4, 0, 3, 1],
       [1, 2, 3, 4, 0]])
```

1.4.3 Arithmetic operations

Arithmetic operations on *DataFrames* (and *Series* also) are as simple as writing the equation.

Remember that values will be aligned by indices, not position!

Let's sum two columns (*Series*)

[30]:
```
dataframe["two"] + dataframe["one"]
```

[30]:
```
A    2
B    2
C    6
D    3
dtype: int64
```

Let's sum two full *DataFrames*

[31]:
```
dataframe + dataframe
```

[31]:

	one	two	three	four	five
A	0	4	2	6	8
B	0	4	8	2	6
C	4	8	0	6	2
D	2	4	6	8	0

1.4.4 Add and remove columns

We can add columns by just defining a new one.

[32]:
```
dataframe["six"] = dataframe["two"] + dataframe["one"]
display(dataframe)
```

	one	two	three	four	five	six
A	0	2	1	3	4	2
B	0	2	4	1	3	2
C	2	4	0	3	1	6
D	1	2	3	4	0	3

Column removal can be performed in two ways

[33]:
```
# del dataframe["six"]

dataframe.drop('six', axis=1, inplace=True)

display(dataframe)
```

	one	two	three	four	five
A	0	2	1	3	4
B	0	2	4	1	3
C	2	4	0	3	1
D	1	2	3	4	0

1.4.5 Sort by an axis

DataFrames can be easily sorted by the *index* or by column labels or values.

Sort by labels

[34]:
```
print("Sort by index labels")
display(dataframe.sort_index(axis=0, ascending=False, inplace=False))

print("Sort by column labels")
display(dataframe.sort_index(axis=1, ascending=False, inplace=False))
```

```
Sort by index labels
```

```
   one  two  three  four  five
D    1    2      3     4     0
C    2    4      0     3     1
B    0    2      4     1     3
A    0    2      1     3     4
```

```
Sort by column labels
```

```
   two  three  one  four  five
A    2      1    0     3     4
B    2      4    0     1     3
C    4      0    2     3     1
D    2      3    1     4     0
```

Sort by values

[35]:
```
print("Sorted by 'one' column values")
display(dataframe.sort_values("one", ascending=False, inplace=False))
print("Sorted by ['two', 'one'] column values")
display(dataframe.sort_values(['two', 'one'], ascending=False, inplace=False))
```

```
Sorted by 'one' column values
```

```
   one  two  three  four  five
C    2    4      0     3     1
D    1    2      3     4     0
A    0    2      1     3     4
B    0    2      4     1     3
```

```
Sorted by ['two', 'one'] column values
```

```
   one  two  three  four  five
C    2    4      0     3     1
D    1    2      3     4     0
A    0    2      1     3     4
B    0    2      4     1     3
```

1.4.6 Reindex the *DataFrame*

Reindexing the *DataFrame* consists of changing the *index*.

Set another column as index You can set, for instance, another column as the new index.

If you set the *drop* parameter to *True*, the columns used as the new index will be removed.

[36]:
```
display(dataframe.set_index("five", drop=True))
```

```
      one  two  three  four
five
4       0    2      1     3
3       0    2      4     1
```

```
1       2    4      0     3
0       1    2      3     4
```

Change the index labels You can also rename the index labels using the *.reindex* method.

This operation will place NA/NaN in locations having no value in the previous index.

[37]:
```
display(dataframe.reindex(["G", "B", "C", "D"]))
```

```
   one  two  three  four  five
G  NaN  NaN    NaN   NaN   NaN
B  0.0  2.0    4.0   1.0   3.0
C  2.0  4.0    0.0   3.0   1.0
D  1.0  2.0    3.0   4.0   0.0
```

Reset the index You can also reset the *index* column by replacing it with a *RangeIndex*.

If you set the *drop* parameter to *True*, old index will be removed from the *DataFrame*.

[38]:
```
display(dataframe.reset_index(drop=False))
```

```
  index  one  two  three  four  five
0     A    0    2      1     3     4
1     B    0    2      4     1     3
2     C    2    4      0     3     1
3     D    1    2      3     4     0
```

1.4.7 Query the *DataFrame*

You can query the *DataFrame* using the *.query* method.

The parameter is a query string that will be evaluated as a boolean expression.

You can refer: - Columns, even with spaces by surrounding them with backticks (‘) - Environment variables (prefixed with @)

For instance:

[39]:
```
dataframe.query("four > 3")
```

[39]:
```
   one  two  three  four  five
D    1    2      3     4     0
```

1.4.8 Merge *DataFrames*

DataFrames can be merged by concatenating them along some axis (index or columns), performing union or intersection of them.

There are many functions and methods for this kind of operation:

- *concat*: this function is the most general
- *.merge*: this method performs a SQL-style join

- *.append*: this method appends the provided rows to the end of the caller object
- *.join*: this method joins the columns of two *DataFrames*

Let's focus on the *concat* function.

After splitting the rows of a *DataFrame*, recreate the original by concatenating them.

[40]:
```
pieces = [dataframe[:2], dataframe[2:4]]

for i, df in enumerate(pieces, start=1):
    print("Piece {}".format(i))
    display(df)

print("Concatenated pieces:")
display(pd.concat(pieces))
```

```
Piece 1
```

	one	two	three	four	five
A	0	2	1	3	4
B	0	2	4	1	3

```
Piece 2
```

	one	two	three	four	five
C	2	4	0	3	1
D	1	2	3	4	0

```
Concatenated pieces:
```

	one	two	three	four	five
A	0	2	1	3	4
B	0	2	4	1	3
C	2	4	0	3	1
D	1	2	3	4	0

After splitting the columns of a *DataFrame*, recreate the original by concatenating them.

[41]:
```
pieces = [dataframe[["one", "two"]], dataframe[["three", "four"]]]

for i, df in enumerate(pieces, start=1):
    print("Piece {}".format(i))
    display(df)

print("Concatenated pieces:")
display(pd.concat(pieces, axis=1))
```

```
Piece 1
```

	one	two
A	0	2
B	0	2

```
C    2    4
D    1    2
```

```
Piece 2
```

```
   three  four
A      1     3
B      4     1
C      0     3
D      3     4
```

```
Concatenated pieces:
```

```
   one  two  three  four
A    0    2      1     3
B    0    2      4     1
C    2    4      0     3
D    1    2      3     4
```

1.5 Group By

Group By is meant to split-apply-combine data in *pandas*.

The full chain consists of the following steps:

- **Splitting** the data into groups using some criteria
- **Applying** some function to each group independently
  - **Aggregation** : compute some summary statistics for each group
  - **Transformation** : perform some group-specific computations and return a like-indexed object
  - **Filtration** : discard some groups, according to a group-wise computation that evaluates *True* or *False*
- **Combining** the results into a data structure

1.5.1 Introduction by example

Let's consider the following *DataFrame*.

[42]:
```
df = pd.DataFrame({
    'Animal': ['Falcon', 'Falcon', 'Parrot', 'Parrot'],
    'Max Speed': [380., 370., 24., 26.]
})

display(df)
```

```
   Animal  Max Speed
0  Falcon      380.0
1  Falcon      370.0
2  Parrot       24.0
3  Parrot       26.0
```

Split As a first step, we want to split the data and group our *DataFrame* using the column values.

[43]:
```
groups = df.groupby(['Animal'])

for animal, group_df in groups:
    print(animal, "group")
    display(group_df)
```

```
Falcon group
```

```
   Animal  Max Speed
0  Falcon      380.0
1  Falcon      370.0
```

```
Parrot group
```

```
   Animal  Max Speed
2  Parrot       24.0
3  Parrot       26.0
```

Aggregate and combine Apply an aggregation by computing the mean value and get the new *DataFrame* as the concatenation of the results.

[44]:
```
groups.mean()
```

[44]:
```
         Max Speed
Animal
Falcon      375.0
Parrot       25.0
```

1.6 Pivoting

Pivoting consists in reshaping a *DataFrame* and organizing it using some given index and/or column values.

This is useful, for instance, when we need some specific representation for our data.

Example Let's try to clarify the concept with an example.

Given another *DataFrame*.

[45]:
```
df = pd.DataFrame({
    'foo': ['one', 'one', 'one', 'two', 'two', 'two'],
    'bar': ['A', 'B', 'C', 'A', 'B', 'C'],
    'baz': [1, 2, 3, 4, 5, 6],
    'zoo': ['x', 'y', 'z', 'q', 'w', 't']
})

df
```

[45]:
```
   foo bar  baz zoo
0  one   A    1   x
1  one   B    2   y
2  one   C    3   z
3  two   A    4   q
4  two   B    5   w
5  two   C    6   t
```

We would like to pivot it and have:

- the values of the "foo" column as new index
- the values of the "bar" column as new columns labels
- the values of "baz" values as new values

[46]:
```
df.pivot(index='foo', columns='bar', values='baz')
```

[46]:
```
bar  A  B  C
foo
one  1  2  3
two  4  5  6
```

We can also use two or mode columns as values.

[47]:
```
df.pivot(index='foo', columns='bar', values=['baz', 'zoo'])
```

[47]:
```
    baz       zoo
bar   A  B  C   A  B  C
foo
one   1  2  3   x  y  z
two   4  5  6   q  w  t
```

1.7 Native *dtypes*

While *dtypes* are inherited from *NumPy*, new some are introduced.

Specifically, the main ones are:

1.7.1 Time Series

Time series types allow to parse time information from various formats, manipulate them and time-zones, resampling, perform date and time arithmetic, etc…

Create a time index One way to create a time index is via the *date_range* function.

date_range(start=None, end=None, periods=None, freq=None, tz=None, …): - start: first date - end: last date - periods: number of periods to generate - freq: frequency of periods

[48]:
```
print("DateRange starting 1/1/2020 with 6M frequency and 10 periods")
date_range = pd.date_range('1/1/2020', periods=8, freq='6M')
display(date_range)
```

```
DateRange starting 1/1/2020 with 6M frequency and 10 periods

DatetimeIndex(['2020-01-31', '2020-07-31', '2021-01-31', '2021-07-31',
               '2022-01-31', '2022-07-31', '2023-01-31', '2023-07-31'],
              dtype='datetime64[ns]', freq='6M')
```

[49]:
```
values = np.random.randint(0, 20, len(date_range))
time_series = pd.Series(values, index=date_range)
print("Time Series:")
display(time_series)
```

```
Time Series:

2020-01-31     9
2020-07-31    11
2021-01-31    15
2021-07-31    18
2022-01-31    13
2022-07-31    14
2023-01-31    19
2023-07-31     2
Freq: 6M, dtype: int64
```

[50]:
```
# Create a DataFrame from the time series
time_dataframe = pd.DataFrame(time_series, columns=["value"])
display(time_dataframe)
```

```
            value
2020-01-31      9
2020-07-31     11
2021-01-31     15
2021-07-31     18
2022-01-31     13
2022-07-31     14
2023-01-31     19
2023-07-31      2
```

Resample the time series You can resample a time series to change the frequency.

For instance, reduce the accuracy to 1Y.

[51]:
```
display(time_series.to_period("Y"))
```

```
2020     9
2020    11
2021    15
2021    18
2022    13
2022    14
2023    19
```

```
2023     2
Freq: A-DEC, dtype: int64
```

[52]:
```
display(time_dataframe.to_period("Y"))
```

```
      value
2020      9
2020     11
2021     15
2021     18
2022     13
2022     14
2023     19
2023      2
```

Basic aggregation What if we want to aggregate the values over the year?

[53]:
```
print("Aggregate by sum to 1Y Frequency:")
display(time_dataframe.to_period("Y").groupby(axis=0, level=0).sum())

# display(time_series.resample('1Y').sum())
```

```
Aggregate by sum to 1Y Frequency:
```

```
      value
2020     20
2021     33
2022     27
2023     21
```

Assignation via time index Being the *Time Series* the index, we can locate entries via dates.

[54]:
```
time_dataframe.loc["2020-01-31", "two"] = 10
display(time_dataframe)
```

```
            value   two
2020-01-31      9  10.0
2020-07-31     11   NaN
2021-01-31     15   NaN
2021-07-31     18   NaN
2022-01-31     13   NaN
2022-07-31     14   NaN
2023-01-31     19   NaN
2023-07-31      2   NaN
```

1.7.2 Categoricals

Categoricals correspond to ***categorical variables*** in statistics.

A *categorical variable* takes on a limited, and usually fixed, number of possible values, with some intrinsic order possible.

As an example, consider the following *DataFrame*.

[55]:
```
df = pd.DataFrame({
    "id": [1, 2, 3, 4, 5, 6],
    "raw_grade": ['a', 'b', 'b', 'a', 'a', 'e']
})
```

Transform a column to *Category* We can define the categories by assigning the type "category" to the column.

[56]:
```
df["grade"] = df["raw_grade"].astype("category")
display(df)
```

```
   id raw_grade grade
0   1         a     a
1   2         b     b
2   3         b     b
3   4         a     a
4   5         a     a
5   6         e     e
```

Rename the categories You can rename the categories very easyly by setting the *.cat.categories* attribute.

[57]:
```
df["grade"].cat.categories = ["very good", "good", "very bad"]
display(df["grade"])
```

```
0    very good
1         good
2         good
3    very good
4    very good
5     very bad
Name: grade, dtype: category
Categories (3, object): [very good, good, very bad]
```

Add new categories You can also add new categories.

[58]:
```
df["grade"].cat.set_categories(["very good", "good", "medium", "bad", "very␣
↪bad"], inplace=True)
```

Sort by Sort the *DataFrame* with the intrinsic category order

[59]:
```
display(df.sort_values(by="grade"))
```

```
   id raw_grade      grade
0   1         a  very good
3   4         a  very good
4   5         a  very good
1   2         b       good
2   3         b       good
5   6         e   very bad
```

1.8 Exercise

Given the CSV file from the previous lecture, load it and perform some statistical analysis.

[60]:
```
import matplotlib.pyplot as plt
%matplotlib inline
```

[61]:
```
file = "albumlist.csv"

df = pd.read_csv(file,
                 encoding="ISO-8859-15",
                 index_col="Number"
                )

display(df.head())
```

```
        Year                                  Album          Artist  \
Number
1       1967  Sgt. Pepper's Lonely Hearts Club Band     The Beatles
2       1966                            Pet Sounds  The Beach Boys
3       1966                              Revolver     The Beatles
4       1965                  Highway 61 Revisited       Bob Dylan
5       1965                           Rubber Soul     The Beatles

            Genre                       Subgenre
Number
1            Rock  Rock & Roll, Psychedelic Rock
2            Rock     Pop Rock, Psychedelic Rock
3            Rock     Psychedelic Rock, Pop Rock
4            Rock          Folk Rock, Blues Rock
5       Rock, Pop                       Pop Rock
```

What is the most present year in the chart?

[62]:
```
display(df["Year"].value_counts().head())
```

```
1970    26
1972    24
1973    23
1969    22
```

```
1968    21
Name: Year, dtype: int64
```

What are the unique genres in the chart?

[63]:
```
df["Genre"].unique()
```

[63]:
```
array(['Rock', 'Rock, Pop', 'Funk / Soul', 'Rock, Blues', 'Jazz',
       'Jazz, Rock, Blues, Folk, World, & Country', 'Funk / Soul, Pop',
       'Blues', 'Pop', 'Rock, Folk, World, & Country',
       'Folk, World, & Country', 'Classical, Stage & Screen', 'Reggae',
       'Hip Hop', 'Jazz, Funk / Soul', 'Rock, Funk / Soul, Pop',
       'Electronic, Rock',
       'Jazz, Rock, Funk / Soul, Folk, World, & Country',
       'Jazz, Rock, Funk / Soul, Pop, Folk, World, & Country',
       'Funk / Soul, Stage & Screen',
       'Electronic, Rock, Funk / Soul, Stage & Screen',
       'Rock, Funk / Soul', 'Rock, Reggae', 'Jazz, Pop',
       'Funk / Soul, Folk, World, & Country', 'Latin, Funk / Soul',
       'Funk / Soul, Blues',
       'Reggae,ÊPop,ÊFolk, World, & Country,ÊStage & Screen',
       'Electronic,ÊStage & Screen', 'Jazz, Rock, Funk / Soul, Blues',
       'Jazz, Rock', 'Rock, Latin, Funk / Soul', 'Electronic, Rock, Pop',
       'Hip Hop, Rock, Funk / Soul', 'Electronic, Pop',
       'Rock, Blues, Pop', 'Electronic, Rock, Funk / Soul, Pop',
       'Rock, Funk / Soul, Folk, World, & Country', 'Rock,ÊBlues',
       'Rock, Pop, Folk, World, & Country', 'Rock, Latin',
       'Rock, Stage & Screen', 'Rock, Blues, Folk, World, & Country',
       'Electronic', 'Electronic, Funk / Soul, Pop',
       'Pop, Folk, World, & Country', 'Electronic, Hip Hop, Pop',
       'Blues, Folk, World, & Country',
       'Electronic, Hip Hop, Funk / Soul, Pop',
       'Rock, Funk / Soul, Blues, Pop, Folk, World, & Country',
       'Jazz, Pop, Folk, World, & Country', 'Jazz, Rock, Pop',
       'Hip Hop, Funk / Soul', 'Hip Hop, Rock',
       'Electronic, Hip Hop, Funk / Soul',
       'Funk / Soul,ÊFolk, World, & Country',
       'Electronic, Hip Hop, Reggae, Pop', 'Electronic, Reggae',
       'Electronic, Funk / Soul', 'Rock, Funk / Soul, Blues', 'Rock,ÊPop',
       'Electronic, Rock, Funk / Soul, Blues, Pop', 'Rock, Reggae, Latin'],
      dtype=object)
```

Let's filter the genres by main

[64]:
```
df["Genre"] = df["Genre"].apply(lambda x: x.strip().split(",")[0].strip())
```

[65]:
```
df["Genre"].unique()
```

[65]:
```
array(['Rock', 'Funk / Soul', 'Jazz', 'Blues', 'Pop', 'Folk', 'Classical',
       'Reggae', 'Hip Hop', 'Electronic', 'Latin'], dtype=object)
```

Set the genre as a category

[66]:
```
df["Genre"] = df["Genre"].astype("category")
```

Any correlation between year and number?

[67]:
```
df.reset_index(drop=False, inplace=True)
```

[68]:
```
df[["Number", "Year"]].corr()
```

[68]:

	Number	Year
Number	1.000000	0.325667
Year	0.325667	1.000000

What is the best rated genre?

[69]:
```
grouped_by_genre = df.groupby("Genre")

display(grouped_by_genre)
```

```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x7fb641c15e10>
```

[70]:
```
mean_position = grouped_by_genre["Number"].mean()

mean_position
```

[70]:
```
Genre
Blues          243.222222
Classical       45.000000
Electronic     307.133333
Folk           262.230769
Funk / Soul    208.078431
Hip Hop        301.411765
Jazz           187.421053
Latin          107.000000
Pop            145.000000
Reggae         191.857143
Rock           250.393082
Name: Number, dtype: float64
```

What are the bands that have the most albums in the chart?

[71]:
```
# As a first step, let's filter the columns
album_df = df[["Artist", "Album"]]
```

```python
# Then, group by the artist
bands = album_df.groupby("Artist", axis=0)

# Now we count how many occurrences there are for each and sort by that value
sorted_bands = bands.count().sort_values("Album", ascending=False)

display(sorted_bands.head())
```

Artist	Album
The Beatles	10
Bob Dylan	10
The Rolling Stones	10
Bruce Springsteen	8
The Who	7