Working with Data, Continued

Sebastian Ernst, PhD

Course: Data Engineering, EAIiIBISIS.Ii8K.5dfa09851a120.22

More data cleaning

Dealing with missing data

- Pandas uses `np.na` to **represent** missing values; `isna()` and `notna()` can functions can be used to **detect** them, e.g.:

  ```
  df.isna().sum()
  ```

- Values can be **filled** using `fillna()`, which takes a scaler, dict, `Series` or `Dataframe`.
- We can **drop** rows with missing values using `dropna()`.
- The same can be done with **rows** by changing the *axis*:

  ```
  df.dropna(axis=1)
  ```

Applying lambdas (a.k.a the functional approach)

- Any **function** can be **applied** to a Series or an entire DataFrame
- The function can be a **Python lambda**:

```
df.one.apply(lambda x: 3.14*x)
```

- When applying to a **Series**, the function is called with **scalars** as arguments.
- Applied to a **DataFrame**, the function is called with **Series**:
  - **column-by-column**, argument's index is `df.index`:

```
df.apply(some_function)
df.apply(some_function, axis=0)
```

  - **row-by-row**, argument's index is `df.columns`:

```
df.apply(some_function, axis=1)
```

Iterating over pandas objects (a.k.a the procedural approach)

- DataFrames can be **iterated over row-by-row** using `df.iterrows()`:

```
for i, r in df.iterrows():
    print(i)  # index
    print(r)  # row as Series, indexed with df.columns
```

- Similarly, we can use `df.items()` to iterate **column-by-column**
- As `df.iterrows()` returns Series, it **does not preserve** dtypes; use `df.itertupes()` if that's needed

Reshaping data

Pivoting

- **Pivoting** is a common reshaping procedure, provided by `df.pivot()`.
- Essentially, DataFrames are converted from a **long format** (where attributes for an entity are described by separate rows) to a **wide format** (where we have columns for attributes):

`df`

	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

```
df.pivot(index='foo',
         columns='bar',
         values='baz')
```

bar	A	B	C
foo			
one	1	2	3
two	4	5	6

Melting

- A **wide** dataframe can be **unpivoted** back to the **long** format using `df.melt()`:

`df3`

	first	last	height	weight
0	John	Doe	5.5	130
1	Mary	Bo	6.0	150

`df3.melt(id_vars=['first', 'last'])`

	first	last	variable	value
0	John	Doe	height	5.5
1	Mary	Bo	height	6.0
2	John	Doe	weight	130
3	Mary	Bo	weight	150

More (un)pivoting

- The `pivot_table()` function is a **generalisation** of `pivot()` that can handle **duplicate values** by adding **aggregation**.
- If **column names** contain **useful values** (e.g. prefixed), such data can be processed using `wide_to_long()`.

Aggregation and grouping

Aggregation

- Pandas provides many **aggregation functions**, such as `sum()`, `max()` or `mean()`.
- The `agg()` function lets us **combine** various aggregation functions in one call and takes:
  - a **list of function names**:

    ```
    df.agg(['sum', 'min'])
    ```

  - a **dictionary** keyed by column names:

    ```
    df.agg({'one': 'mean', 'four': 'sum'})
    ```

  - a **tuples** as **keyword arguments** (to rename columns):

    ```
    df.agg(one_mean=('one', 'mean'), four_sum=('four', 'sum'))
    ```

    tip: add `sum(axis=1)` to get a `Series` instead of a `DataFrame` here

Grouping

- **Grouping** is performed using `groupby()` which **returns** a *groupby object*.
- Calls to `groupby()` are usually chained with a *groupby*-compatible **aggregate function**, such as `sum()`, `min()`, `max()`, `mean()` or `agg()`.
- A *groupby object* can also be **iterated** over:

```
for i, g in df.groupby('six'):
    print(i, g)
```

- Groups can also be **filtered** (equivalent of SQL `HAVING`) using `filter()`, which returns a `DataFrame`:

```
df.groupby('six').filter(lambda x: len(x) > 2)
```

Grouping categoricals

When grouping by **categorical** dtypes, groups are created for **all categories**, even the **empty ones**:

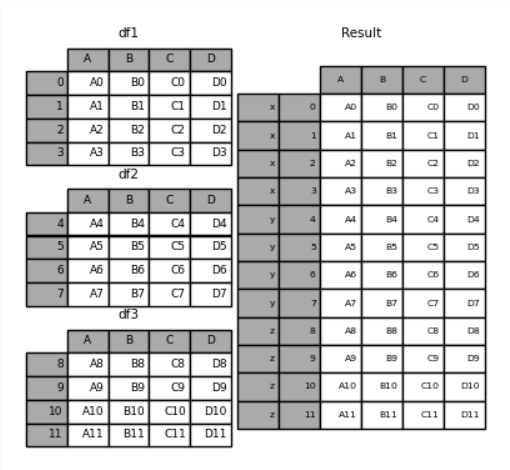
```
>>> # "very good", "good", "very bad"
>>> df["grade"] = df["grade"].cat.set_categories(
...    ["very bad", "bad", "medium", "good", "very good"]
... )
>>> df.groupby("grade").size()
grade
very bad     1
bad          0
medium       0
good         2
very good    3
dtype: int64
```

Merge, join, concatenate and compare

Concatenating DataFrames

- **Concatenation** is a simple operation of **"gluing"** DataFrames along **one of the axes**.
- **Implemented** using `pd.concat()`, which takes a **list** of objects as argument.
- Common **use case** is appending several **homogenous DFs**:

```
dfs = []
dfs.append(df1)
dfs.append(df2)
dfs.append(df3)
pd.concat(dfs)
```



Concatenating along column axis

Objects can be concatenated along **any axis**:

```
pd.concat([df1, df4], axis=1)
```

df1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

df4

	B	D	F
2	B2	D2	F2
3	B3	D3	F3
6	B6	D6	F6
7	B7	D7	F7

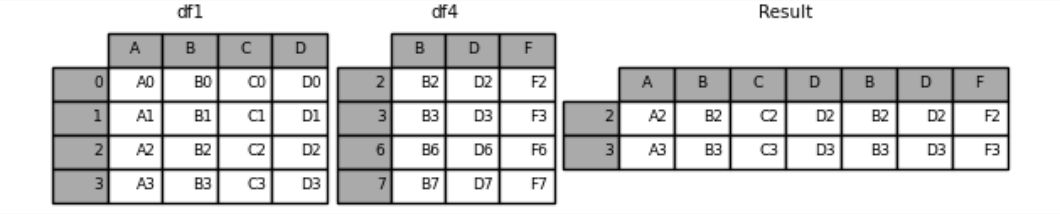
Result

	A	B	C	D	B	D	F
0	A0	B0	C0	D0	NaN	NaN	NaN
1	A1	B1	C1	D1	NaN	NaN	NaN
2	A2	B2	C2	D2	B2	D2	F2
3	A3	B3	C3	D3	B3	D3	F3
6	NaN	NaN	NaN	NaN	B6	D6	F6
7	NaN	NaN	NaN	NaN	B7	D7	F7

Inner concatenation

To only include **common index entries**, we can add `join='inner'`:

```
pd.concat([df1, df4], axis=1, join="inner")
```



df1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

df4

	B	D	F
2	B2	D2	F2
3	B3	D3	F3
6	B6	D6	F6
7	B7	D7	F7

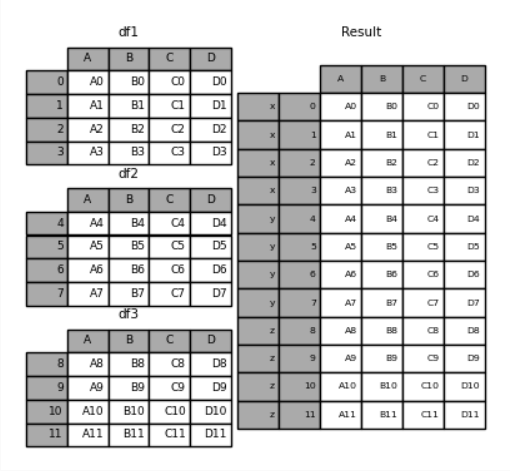
Result

	A	B	C	D	B	D	F
2	A2	B2	C2	D2	B2	D2	F2
3	A3	B3	C3	D3	B3	D3	F3

Using group keys

Group keys make it easy to remember which source DataFrame a row came from:

```
pd.concat([df1, df2, df2],
          keys=["x", "y", "z"])
```



Join

- Performs a join of **two or more** DataFrames using their **indexes** (or, optionally, one **key column**):

```
df1.join(df2)                          # index-to-index
df1.join(df2, on='my_column')          # df1.my_column to df2.index
df1.join(df2.set_index('some_column'), on='my_column')
                                       # key-to-key, sort of...
df1.join([df2, df3])                   # multiple DataFrames
```

- Join can be **inner**, **outer** (left/right) or **cross**:

```
df1.join(df2, how='outer')
```

- **Overlapping columns** can be suffixed using `lsuffix` and `rsuffix` (only when merging 2!)

Merge

- **Similar** to join, available as `pd.merge()` and `df.merge()`
- Only handles **exactly two** DataFrames
- Easier **key-to-key** joins using `left_on` and `right_on`:

```
df1.join(df2.set_index('some_column'), on='my_column')
df1.merge(df2, left_on='my_column', right_on='some_column')
```

- Can `validate` **relationship type** (`one_to_one`, `one_to_many`, `many_to_one`, `many_to_many`)

Visualising data

Plotting in pandas

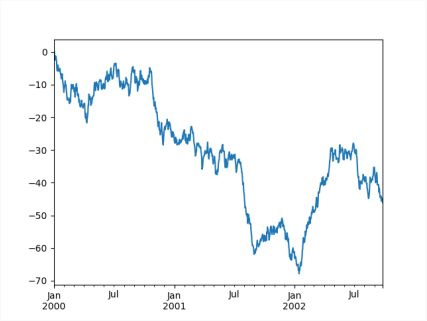
- In **Python**, one of the most common **plotting tools** is matplotlib and its pyplot interface:

```
import matplotlib.pyplot as plt
```

- **Pandas objects** provide the `plot()` method, which **by default** is a wrapper around `plt.plot()`:

```
ts = pd.Series(np.random.randn(1000),
               index=pd.date_range(
                 "1/1/2000", periods=1000))
ts = ts.cumsum()
ts.plot()
```

- Besides the default line plot, **available types** include bar, histogram, box, area, scatter, hex bin and pie plots.



More plotting

- Pandas provides **plotting tools** in the `pd.plotting` package:

```
df = pd.DataFrame(np.random.randn(1000, 4),
                  columns=["a", "b", "c", "d"])
pd.plotting.scatter_matrix(df, alpha=0.2,
                           figsize=(10, 10),
                           diagonal="hist")
```

- Pandas also supports different **plotting backends**, such as Plotly Express, Bokeh, Altair or hvPlot:

```
pd.options.plotting.backend = "plotly"
```

- Many **other tools** (such as seaborn) support **pandas objects**.