Lecture 5
Advanced Pandas

Data Visualization · 1-DAV-105

Lecture by Broňa Brejová

More details in the notebook version

Pandas functions seen so far

- Creating `Series` and `DataFrame`, `pd.read_csv`
- Accessing parts `iloc[]`, `loc[]`, `query`, `head`
- Working with indexes `set_index`, `reset_index`
- Other operations `rename`, `pivot`, `copy`, `sort_values`, ...
- Basic statistics: `mean, median, var, std, min, max, quantile, corr, describe`

Also recall:

- long and wide tables
- numerical and categorical variables

Data sets for today

- Country indicators from World Bank (as in L03)
- 2049 movies (as in L04)
- Some artificial examples or small subsets
- Details see https://fmfi-compbio.github.io/viz/data/

Hierarchical index (MultiIndex)

An index with duplicate labels

Pandas allows setting index on a column with duplicate labels

example_long

	Country	Year	Population
0	Slovak Republic	2010	5,391,428.00
1	Austria	2010	8,363,404.00
2	Slovak Republic	2020	5,458,827.00
3	Austria	2020	8,916,864.00

Country	Year	Population
Slovak Republic	2010	5,391,428.00
Austria	2010	8,363,404.00
Slovak Republic	2020	5,458,827.00
Austria	2020	8,916,864.00

```
# set country name as index in a copy of the table
example_long_indexed = example_long.set_index('Country')
```

An index with duplicate labels

Selecting a country returns multiple rows

Country	Year	Population
Slovak Republic	2010	5,391,428.00
Austria	2010	8,363,404.00
Slovak Republic	2020	5,458,827.00
Austria	2020	8,916,864.00

```
example_long_indexed.loc['Slovak Republic']
```

Country	Year	Population
Slovak Republic	2010	5,391,428.00
Slovak Republic	2020	5,458,827.00

The hierarchical index (multiindex)

- More natural index uses a pair (country, year), which uniquely specifies a row
- Created by `set_index` with a list of columns to use as index
- For faster operations, sort the table by the index using `sort_index`

```
# create MultiIndex by choosing a list of columns
example_multiindexed = example_long.set_index(['Country', 'Year']).sort_index()
```

		Population
Country	**Year**	
Austria	**2010**	8,363,404.00
	2020	8,916,864.00
Slovak Republic	**2010**	5,391,428.00
	2020	5,458,827.00

Accessing table using multiindex

- `loc`: a tuple with one value per level, or only several initial levels
- `xs`: specify other levels

```
example_multiindexed.loc[('Slovak Republic', 2010)]
```

```
Population   5,391,428.00
Name: (Slovak Republic, 2010), dtype: float64
```

Accessing table using multiindex

- `loc`: a tuple with one value per level, or only several initial levels
- `xs`: specify other levels

```
example_multiindexed.loc[('Slovak Republic',)]
```

	Population
Year	
2010	5,391,428.00
2020	5,458,827.00

Accessing table using multiindex

- `loc`: a tuple with one value per level, or only several initial levels
- `xs`: specify other levels

```
example_multiindexed.xs(2010, level='Year')
```

	Population
Country	
Austria	8,363,404.00
Slovak Republic	5,391,428.00

Names of index levels can be used in query

```
example_multiindexed.query('Year > 2015')
```

		Population
Country	**Year**	
Austria	**2020**	8,916,864.00
Slovak Republic	**2020**	5,458,827.00

Combining tables

Toy example: two small tables

Each table a few countries, data for different year

Country	Area	Population	Year
Slovak Republic	49,030.00	5,391,428.00	2010
Austria	83,879.00	8,363,404.00	2010
Hungary	93,030.00	10,000,022.00	2010

Country	Area	Population	Year
Slovak Republic	49,030.00	5,458,827.00	2020
Austria	83,879.00	8,916,864.00	2020
Ukraine	603,550.00	44,132,050.00	2020

Concatenating tables using `concat`

Default: combine along axis 0

The rows of second table are added after the rows of the first table

```
pd.concat([example_countries2010, example_countries2020])
```

Country	Area	Population	Year
Slovak Republic	49,030.00	5,391,428.00	2010
Austria	83,879.00	8,363,404.00	2010
Hungary	93,030.00	10,000,022.00	2010
Slovak Republic	49,030.00	5,458,827.00	2020
Austria	83,879.00	8,916,864.00	2020
Ukraine	603,550.00	44,132,050.00	2020

long table

Concatenating tables using `concat`

With axis=1 finds rows with the same index and combines their columns

```
pd.concat([example_countries2010, example_countries2020], axis=1)
```

Country	Area	Population	Year	Area	Population	Year
Slovak Republic	49,030.00	5,391,428.00	2,010.00	49,030.00	5,458,827.00	2,020.00
Austria	83,879.00	8,363,404.00	2,010.00	83,879.00	8,916,864.00	2,020.00
Hungary	93,030.00	10,000,022.00	2,010.00	NaN	NaN	NaN
Ukraine	NaN	NaN	NaN	603,550.00	44,132,050.00	2,020.00

This is not ideal - why?

Concatenating tables using `concat`

Add year as multiindex for columns with `keys` argument

```
pd.concat([example_countries2010, example_countries2020],
          axis=1, keys=['2010', '2020'])
```

	2010			2020		
	Area	Population	Year	Area	Population	Year
Country						
Slovak Republic	49,030.00	5,391,428.00	2,010.00	49,030.00	5,458,827.00	2,020.00
Austria	83,879.00	8,363,404.00	2,010.00	83,879.00	8,916,864.00	2,020.00
Hungary	93,030.00	10,000,022.00	2,010.00	NaN	NaN	NaN
Ukraine	NaN	NaN	NaN	603,550.00	44,132,050.00	2,020.00

Concatenating Series to a wide table along axis 1

```
Country
Slovak Republic    5,391,428.00
Austria            8,363,404.00
Hungary           10,000,022.00
Name: Population2010, dtype: float64
```

```
Country
Slovak Republic    5,458,827.00
Austria            8,916,864.00
Ukraine           44,132,050.00
Name: Population2020, dtype: float64
```

```
pd.concat([example2_countries2010,
           example2_countries2020],
          axis=1)
```

Country	Population2010	Population2020
Slovak Republic	5,391,428.00	5,458,827.00
Austria	8,363,404.00	8,916,864.00
Hungary	10,000,022.00	NaN
Ukraine	NaN	44,132,050.00

Keep only shared rows with `join='inner'`

```
Country
Slovak Republic    5,391,428.00
Austria            8,363,404.00
Hungary           10,000,022.00
Name: Population2010, dtype: float64
```

```
Country
Slovak Republic    5,458,827.00
Austria            8,916,864.00
Ukraine           44,132,050.00
Name: Population2020, dtype: float64
```

```
pd.concat([example2_countries2010,
           example2_countries2020],
          axis=1, join='inner')
```

Country	Population2010	Population2020
Slovak Republic	5,391,428.00	5,458,827.00
Austria	8,363,404.00	8,916,864.00

Merging tables with `merge`

- works similarly as concat with axis=1,
- matches lines of two tables using specified columns, not necessarily index
- if values in these columns repeat, it combines all matching pairs of rows

```
pd.merge(tab1, tab2, on='name')
```

	name	value
0	a	1
1	a	2
2	a	3
3	b	4

	name	value
0	a	10
1	a	20
2	b	30

	name	value_x	value_y
0	a	1	10
1	a	1	20
2	a	2	10
3	a	2	20
4	a	3	10
5	a	3	20
6	b	4	30

Example of using `merge` on countries

```
example_membership = pd.merge(example_countries, membership,
                              on='Country')
```

Country	Population2010	Population2020
Slovak Republic	5,391,428.00	5,458,827.00
Austria	8,363,404.00	8,916,864.00

	Country	Member
0	Slovak Republic	NATO
1	Slovak Republic	EU
2	Slovak Republic	UN
3	Austria	UN
4	Austria	EU

What will be the result?

Example of using `merge` on countries

```
example_membership = pd.merge(example_countries, membership,
                              on='Country')
```

	Country	Population2010	Population2020	Member
0	Slovak Republic	5,391,428.00	5,458,827.00	NATO
1	Slovak Republic	5,391,428.00	5,458,827.00	EU
2	Slovak Republic	5,391,428.00	5,458,827.00	UN
3	Austria	8,363,404.00	8,916,864.00	UN
4	Austria	8,363,404.00	8,916,864.00	EU

This may seem like a nonsense (country data needlessly repeated) but it is useful for computing statistics of organizations

Total population of countries in an organization

... imagine we have membership for all countries of the world

```
example_membership.query('Member == "EU"')['Population2020'].sum()
```

	Country	Population2010	Population2020	Member
0	Slovak Republic	5,391,428.00	5,458,827.00	NATO
1	Slovak Republic	5,391,428.00	5,458,827.00	EU
2	Slovak Republic	5,391,428.00	5,458,827.00	UN
3	Austria	8,363,404.00	8,916,864.00	UN
4	Austria	8,363,404.00	8,916,864.00	EU

Computing population sums for all organizations

using `groupby` to be covered next...

beware: only 2 countries used here, so the sums are not meaningful

```
example_membership.groupby('Member')['Population2020'].sum()
```

```
Member
EU      14,375,691.00
NATO     5,458,827.00
UN      14,375,691.00
Name: Population2020, dtype: float64
```

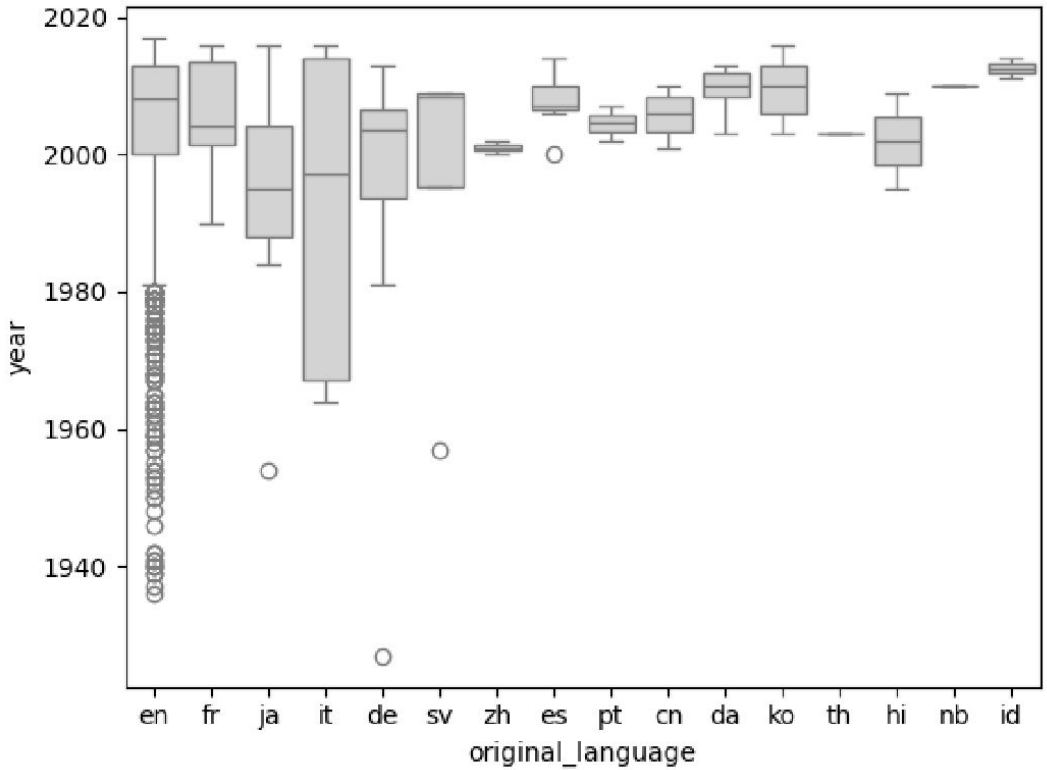
Connection to relational databases

- Relational databases allow processing of large data stored on disk (not in memory), also simultaneous access of many processes
- Database course in the third year
- Operation `merge` is called join in databases, used frequently
- Operation `groupby` (next) is also frequent
- Database would store original tables (one for countries, one for relation between countries and organizations), merged table is computed only temporarily

Panda's merge does inner join by default (only rows present in both tables) other options via `how='left'/'right'/'outer'/...`

Aggregation, split-apply-combine (`groupby`)

Recall boxplots splitting data by category



What if we want to use these statistics per group as numbers?

A simple example of groupby

Compute the number of countries belonging to each region of the world

```
countries.groupby('Region').size()
```

```
Region
East Asia & Pacific            37
Europe & Central Asia          58
Latin America & Caribbean      42
Middle East & North Africa     21
North America                   3
South Asia                      8
Sub-Saharan Africa             48
dtype: int64
```

- `groupby` splits countries into groups by `Region`
- `size` computes the number of rows in each group

General principles: split-apply-combine strategy

- **Split:** split data into groups, often by values in some column
- **Apply:** apply some computation on each group, obtaining a single value, Series or DataFrame
- **Combine:** concatenate results for all groups together to a new table

Introduced in R by Hadley Wickham

General principles: split-apply-combine strategy

- **Split:** split data into groups, often by values in some column
- **Apply:** apply some computation on each group, obtaining a single value, Series or DataFrame
- **Combine:** concatenate results for all groups together to a new table

Typical operations in the **apply step**:

- **aggregation**: e.g. compute group size, mean, median etc.
- **transformation**: e.g. compute percentage or rank of each item within a group
- **filtering**: e.g. include only groups that are large enough

In Pandas: `groupby` followed by a function for apply. Combine implicit.

Simple aggregation in the apply step

Apply functions such as `sum`, `mean`, `median`, `min`, `max`, `size`, `count`, `describe` after `groupby`

- `size` gives the number of rows in the group
- `count` gives the number of non-missing values in each column.

```
countries.groupby('Region').sum(numeric_only=True)
```

Region	Population2000	Population2010	Population2020	Area	GDP2000	...
East Asia & Pacific	2,025,976,167.00	2,187,065,378.00	2,340,350,517.00	24,794,669.42	233,980.83	
Europe & Central Asia	862,786,208.00	889,169,626.00	922,353,365.00	28,813,751.77	883,386.74	
...						

Total of GDP not very useful

Specifically sum only population in 2020 per region

```
countries.groupby('Region')['Population2020'].sum()
```

```
Region
East Asia & Pacific           2,340,350,517.00
Europe & Central Asia           922,353,365.00
Latin America & Caribbean       650,534,988.00
Middle East & North Africa      479,966,650.00
North America                   369,582,572.00
South Asia                    1,882,531,621.00
Sub-Saharan Africa            1,151,302,077.00
Name: Population2020, dtype: float64
```

Transformation in the apply step

- `transform`: gets a function to be used on every group; it should produce a group with the same index
- Custom function (e.g. a lambda expression) or built-in functions specified by a string

```
countries.groupby('Region')['Population2020'].transform('sum')
```

```
Country
Afghanistan          1,882,531,621.00
Albania                922,353,365.00
Algeria                479,966,650.00
American Samoa       2,340,350,517.00
Andorra                922,353,365.00
                             ...
Virgin Islands         650,534,988.00
West Bank and Gaza     479,966,650.00
Yemen                  479,966,650.00
Zambia               1,151,302,077.00
Zimbabwe             1,151,302,077.00
Name: Population2020, Length: 217, dtype: float64
```

What happened here?

Transformation in the apply step

- `transform`: gets a function to be used on every group; it should produce a group with the same index
- Custom function (e.g. a lambda expression) or built-in functions specified by a string

```
countries.groupby('Region')['Population2020'].transform('sum')
```

```
Country
Afghanistan          1,882,531,621.00
Albania                922,353,365.00
Algeria                479,966,650.00
American Samoa       2,340,350,517.00
Andorra                922,353,365.00
                             ...
Virgin Islands         650,534,988.00
West Bank and Gaza     479,966,650.00
Yemen                  479,966,650.00
Zambia               1,151,302,077.00
Zimbabwe             1,151,302,077.00
Name: Population2020, Length: 217, dtype: float64
```

Sum population of each region and assign to each country in the region

Transformation in the apply step

```
countries.groupby('Region')['Population2020'].transform('sum')
```

```
Country
Afghanistan          1,882,531,621.00
Albania                922,353,365.00
Algeria                479,966,650.00
American Samoa       2,340,350,517.00
Andorra                922,353,365.00
                             ...
Virgin Islands         650,534,988.00
West Bank and Gaza     479,966,650.00
Yemen                  479,966,650.00
Zambia               1,151,302,077.00
Zimbabwe             1,151,302,077.00
Name: Population2020, Length: 217, dtype: float64
```

Sum population of each region and assign to each country in the region

```
countries['Population2020'] / region_sums
```

```
Afghanistan      0.02
Albania          0.00
Algeria          0.09
 ...
```

What fraction is country's population within its region?

Another way using lambda expression

```
(countries.groupby('Region')['Population2020']
 .transform(lambda x : x / x.sum()))
```

Lambda function takes a Series `x` of country sizes within a region and divides them by the sum of x

The same with named function definition:

```
def group_fraction(x):
    return x / x.sum()
(countries.groupby('Region')['Population2020']
 .transform(group_fraction))
```

Add region name using `concat`

```
pop_within_group2 = pd.concat([pop_within_group,
                               countries['Region']], axis=1)
```

```
Afghanistan      0.02
Albania          0.00
Algeria          0.09
```

Country	Population2020	Region
Afghanistan	0.02	South Asia
Albania	0.00	Europe & Central Asia
Algeria	0.09	Middle East & North Africa

Lookup value for Slovakia

```
pop_within_group2.loc["Slovak Republic"]
```

```
Population2020                          0.01
Region                 Europe & Central Asia
Name: Slovak Republic, dtype: object
```

Check that the sum of each region is 1

```
pop_within_group2.groupby('Region').sum()
```

Region	Population2020
East Asia & Pacific	1.00
Europe & Central Asia	1.00
Latin America & Caribbean	1.00
Middle East & North Africa	1.00
North America	1.00
South Asia	1.00
Sub-Saharan Africa	1.00

Filtering in the apply step

- `groupby` can be followed by `filter` to select only some of the groups
- here get all countries in regions that have at least one billion inhabitants

```
(countries.groupby("Region")
 .filter(lambda x : x['Population2020'].sum() > 1e9))
```

Country	ISO3	Region	Income Group	Population2000	Population2010	Population2020	Area
Afghanistan	AFG	South Asia	Low income	19,542,983.00	28,189,672.00	38,972,231.00	652,860.00
American Samoa	ASM	East Asia & Pacific	High income	58,229.00	54,849.00	46,189.00	200.00
Angola	AGO	Sub-Saharan Africa	Lower middle income	16,394,062.00	23,364,186.00	33,428,486.00	1,246,700.00
Australia	AUS	East Asia & Pacific	High income	19,028,802.00	22,031,750.00	25,649,247.00	7,741,220.00
Bangladesh	BGD	South Asia	Lower middle income	129,193,327.00	148,391,139.00	167,420,950.00	147,570.00

Check sums in the the selected regions

```
filtered.groupby('Region')['Population2020'].sum()
```

```
Region
East Asia & Pacific   2,340,350,517.00
South Asia            1,882,531,621.00
Sub-Saharan Africa    1,151,302,077.00
Name: Population2020, dtype: float64
```

Grouping by multiple values

In `groupby` we can use: a single column, list of columns, a separate `Series`

```
countries.groupby(['Region', "Income Group"])['Population2020'].sum()
```

```
Region                     Income Group
East Asia & Pacific        High income            223,971,823.00
                           Low income              25,867,467.00
                           Lower middle income    301,779,468.00
                           Upper middle income  1,788,731,759.00
Europe & Central Asia      High income            522,292,344.00
                           Lower middle income     94,487,207.00
                           Upper middle income    305,573,814.00
Latin America & Caribbean  High income             34,033,357.00
                           Lower middle income     40,120,621.00
                           Upper middle income    547,890,556.00
Middle East & North Africa High income             68,156,525.00
                           Low income              53,056,642.00
                           Lower middle income    304,739,289.00
                           Upper middle income     54,014,194.00
North America              High income            369,582,572.00
South Asia                 Low income              38,972,231.00
                           Lower middle income  1,843,044,952.00
                           Upper middle income        514,438.00
Sub-Saharan Africa         High income                98,462.00
                           Low income             549,157,331.00
                           Lower middle income    533,054,222.00
                           Upper middle income     68,992,062.00
Name: Population2020, dtype: float64
```

A separate series to be used in `groupby`

Classify each country as small (< 1million), medium and large (>100 million)

```
bin_ends = [0, 1e6, 1e8, 1e10]
bin_labels = ["small", "medium", "large"]
size_groups = (pd.cut(countries['Population2020'],
                      bins=bin_ends, labels=bin_labels)
               .rename("SizeCategory"))
```

```
Country
Afghanistan       medium
Albania           medium
Algeria           medium
American Samoa     small
Andorra            small
Name: SizeCategory, dtype: category
Categories (3, object): ['small' < 'medium' < 'large']
```

A separate series in to be used `groupby`

Now use this Series in `groupby`

Parameter `observed=True` added because `size_groups` has a categorical variable type (explained next)

```
countries.groupby(size_groups, observed=True).size()
```

```
SizeCategory
small      57
medium    146
large      14
dtype: int64
```

```
countries.groupby(['Region', size_groups], observed=True).size()
```

```
Region                        SizeCategory
East Asia & Pacific           small           18
                              medium          15
                              large            4
Europe & Central Asia         small           12
                              medium          45
                              large            1
Latin America & Caribbean     small           19
                              medium          21
                              large            2
Middle East & North Africa    small            1
                              medium          19
                              large            1
North America                 small            1
                              medium           1
                              large            1
South Asia                    small            2
                              medium           3
                              large            3
Sub-Saharan Africa            small            4
                              medium          42
                              large            2
dtype: int64
```

Categorical variables

Categorical variables

Categorical variables: values from a small set, such as region and income group

So far represented as strings, but we can explicitly convert them to a **categorical data type** in Pandas

- Strings internally replaced by numerical IDs within the table, potentially saving memory
- Categories can be ordered and then sorting, minimum, maximum etc. works as desired, not alphabetically
- Pandas is aware of the full set of possible values; categories without members can appear in the `groupby` results

Converting income group to a categorical type

```
# creating a categorical type
cat_type = pd.api.types.CategoricalDtype(categories=["Low income",
                                                     "Lower middle income",
                                                     "Upper middle income",
                                                     "High income"],
                                         ordered=True)
# converting Income Group column to cat_type in a new DataFrame
countries_cat = countries.astype({'Income Group': cat_type})
```

```
countries_cat['Income Group'].min()
```

```
'Low income'
```

```
countries['Income Group'].dropna().min()
```

```
'High income'
```

Unordered category, empty groups in `groupby`

```
# convert region to an unordered category
countries_cat2 = countries_cat.astype({'Region': 'category'})
# count the number of countries for each combination of income group and region
countries_cat2.groupby(['Income Group', 'Region'], observed=False).size()
```

```
Income Group          Region
Low income            East Asia & Pacific            1
                      Europe & Central Asia          0
                      Latin America & Caribbean      0
                      Middle East & North Africa     2
                      North America                  0
                      South Asia                     1
                      Sub-Saharan Africa            22
Lower middle income   East Asia & Pacific           13
                      Europe & Central Asia          4
                      Latin America & Caribbean      4
 ...
```

Dates and times

Dates and times

Often used in time series, extensive support in Pandas

Here only an example:

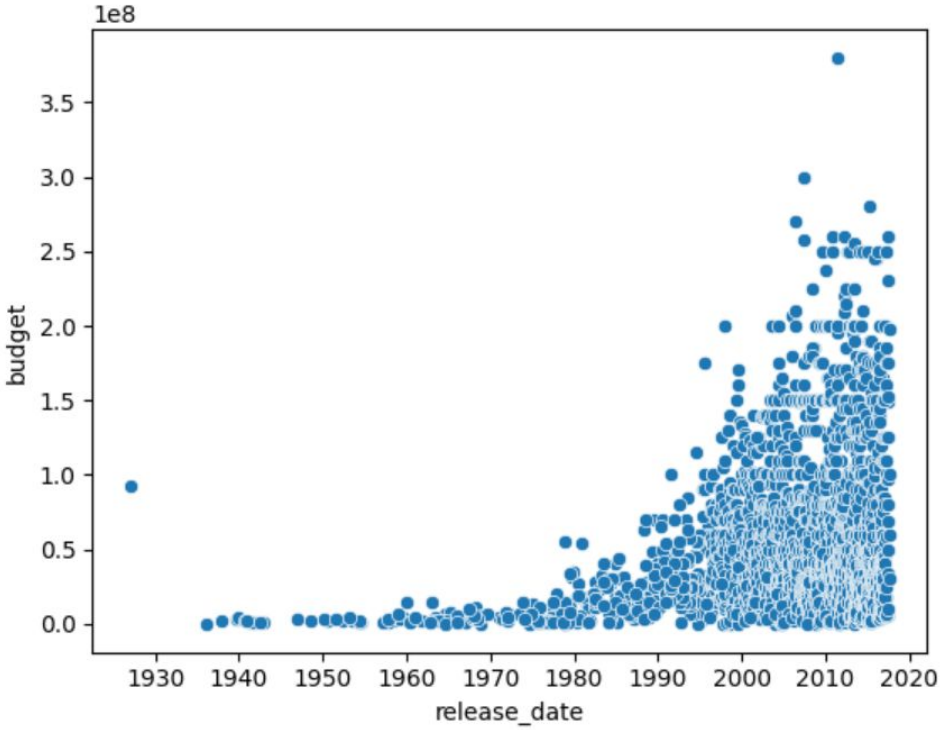
- In the movie data set, the column release_date is recognized as date by passing parse_dates parameter to read_csv.
- We get the day of the week using day_name() and use value_counts to see which days are most frequent as movie release dates.

```
# import data, including parsing of dates
url = 'https://fmfi-compbio.github.io/viz/data/Movies_small.csv'
movies = pd.read_csv(url, parse_dates=['release_date'])
# get days of week for realse dates
days = movies['release_date'].apply(lambda x : x.day_name())
days.value_counts()
```

```
release_date
Friday       639
Thursday     515
Wednesday    474
Tuesday      175
Saturday      94
Monday        87
Sunday        65
Name: count, dtype: int64
```

Date as a coordinate in a plot

```
sns.scatterplot(data=movies, x='release_date', y='budget')
```



Missing values

Missing valuea

- Data sets are often incomplete
- Pandas provides techniques for working with missing data
- Missing data are typically imported as np.nan (not-a-number)
- Ints are converted to floats in the presence of missing values
- Next: several basic functions for handling missing data

Behaviour of summary statistics with missing values

```
a = pd.Series([1, 2, np.nan, 3])
```

a.sum() skips missing values:

```
6.0
```

a.count() counts non-missing values:

```
3
```

a.mean() also considers only non-missing:

```
2.0
```

Comparisons with missing values

```
a = pd.Series([1, 2, np.nan, 3])
```

`a > 2` evaluates missing values as `False`, similarly `<`, `==`:

```
0    False
1    False
2    False
3     True
dtype: bool
```

`a == np.nan` also evaluates as `False`:

```
0    False
1    False
2    False
3    False
dtype: bool
```

Comparisons with missing values

```
a = pd.Series([1, 2, np.nan, 3])
```

`a.isna()` can be used to detect missing values:

```
0    False
1    False
2     True
3    False
dtype: bool
```

`a.dropna()` omits missing values:

```
0   1.00
1   2.00
3   3.00
dtype: float64
```

`a.fillna(-1)` replaces them with a specified value:

```
0    1.00
1    2.00
2   -1.00
3    3.00
dtype: float64
```

Pandas efficiency

Pandas efficiency

We will see:

- Different implementations of the same operation can have very different running time on large data
- Pandas functions are usually much faster than manual iteration.
- However, if you do not work on huge data sets, the difference is not so important

```
# generate a Series of million random numbers
# and also convert it to Python list
length = int(1e6)
xs = pd.Series(np.random.uniform(0,100, length))
xl = list(xs)
```

Jupyter command `%timeit`

```
display(Markdown("**Method `sum` on `Series` `xs.sum()`:**"))
%timeit result = xs.sum()
```

Method `sum` on `Series` `xs.sum()`:

```
433 µs ± 20.7 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

Results for computing sum of 1 million numbers

- Method sum on Series `xs.sum()`: 0.433ms
- Python sum on Python list `sum(xl)`: 7.77ms
- Python sum on Series `sum(xs)`: 56.5 ms

You will get different results on different computers / software versions

Pure Python can be accelerated e.g. by Numba library

Results for computing squares of 1 million numbers

- Pandas Series multiplication `x2s = xs * xs`: 0.848ms
- Python list comprehension on a list `x2l = [x*x for x in xl]`: 36.7 ms
- Pandas apply function `x2s = xs.apply(lambda x : x * x)`: 155 ms
- Setting values of Series by a for loop: 8,280 ms (estimated from n=10,000)
- Adding values to Series by a for loop: 160,000 ms (estimated from n=1,000)

```
length2 = 10000
xs_small = xs.iloc[0:length2]
def assignments(len, x):
  x2 = pd.Series([0.0] * len)
  for i in range(len):
    x2[i] = x[i] * x[i]
  return x2
```

```
length3 = 1000
xs_tiny = xs.iloc[0:length3]
def assignments(len, x):
  x2 = pd.Series([0.0])
  for i in range(len):
    x2[i] = x[i] * x[i]
  return x2
```

Summary

- index with repeated values, multiindex
- concat and merge for combining tables
- groupby for aggregating / transforming / filtering groups of data
- categorical data types
- date and time data type
- missing values
- efficiency