EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Philosophische Fakultät
Seminar für Sprachwissenschaft

Data Science for Linguists

Session 3: Pandas and Data Frames

Johannes Dellert

10 November, 2023

Table of Contents

Pandas: Purpose and Usage

- the `pandas` package provides the most popular Python implementation of a **data frame** (a multidimensional array of heterogeneous types with attached row and column labels)
- `pandas.DataFrame` objects provide extensive functionality for
  - ▷ working with missing data
  - ▷ operations that do not map well to NumPy-style element-wise broadcasting
  - ▷ efficient data structures for time slices and indexing
- convention for import (this is what we will assume throughout this course):

```
import pandas as pd
```

Pandas: Fundamental Data Structures

- two main workhorse data structures:
  - `pd.Series` for a sequence of values of the same type
  - `pd.DataFrame` for a rectangular table of data composed of one series per column
- Pandas data structures combine NumPy arrays with `Index` objects which hold
  - axis labels (including column name)
  - other metadata (e.g. axis name or names)

pd.Series

- a `Series` combines a sequence of values (a one-dimensional NumPy array) with an explicit sequence of indices (of type `pd.Index`) which allows much broader usage
- construction from list literal: `data = pd.Series([0.25, 0.5, 0.75, 1.0])`
- `data.values` is the array, `data.index` is a `RangeIndex(start=0, stop=4, step=1)`
- data can be accessed by the associated index using slicing notation: `data[1:3]`
- crucial difference to a NumPy array: explicitly defined index associated with the values, which gives the `Series` object additional capabilities:
  - ▷ index need not be an integer, but can consist of values of any desired type
  - ▷ for instance, this allows us to make every datapoint accessible under string ids:
    ```
    data = pd.Series([0.25, 0.5, 0.75], index=['a', 'b', 'c'])
    stored_value = data['b']
    ```
- `data.reindex()` allows us to create a new `Series` with values rearranged to align with the new index (plus support for filling with default values or interpolation)

`pd.Series` as a Specialized Dictionary

- this capability turns a Pandas `Series` into a specialised Python dictionary:
  - ▷ a Python dictionary maps arbitrary keys to a set of arbitrary values
  - ▷ a `Series` maps typed keys to a set of typed values (which makes it more efficient!)
- there is a constructor which allows to construct a `Series` directly from a Python dictionary:

```
iso_to_lang = {'aa': 'Afar', 'ab': 'Abkhaz', 'ae': 'Avestan',
               'af': 'Afrikaans', 'ak': 'Akan', 'am': 'Amharic'}
```

- the index is ordered, which makes array-style operations such as slicing possible:

```
In [ ]: iso_to_lang['ae':'ak']
Out[ ]: ae Avestan
        af Afrikaans
        ak Akan
```

- the equivalent to `del dct[key]` is `data.drop(list_of_indices)`

Constructing `pd.Series` Objects

- general form of the constructor: `pd.Series(data, index=index)`
- `data` can be one of many entities:
  - ▷ list or NumPy array, in which case `index` defaults to an integer sequence
  - ▷ a scalar, which is repeated to fill the specified index
  - ▷ a dictionary, in which case `index` defaults to dictionary keys
- in each case, an index can be explicity set to control the order or subset of keys used

```
In [ ]: pd.Series({2:'a', 1:'b', 3:'c'}, index=[1, 2])
Out[ ]: 1 b
        2 a
```

pd.DataFrame as Generalized NumPyArray

- a DataFrame combines a two-dimensional array of values with explict row and column indices
- it can be thought of as a sequence of Series that are aligned (share the same index)

```
In [ ]: lang = pd.Series(iso_to_lang)
        speakers = pd.Series({'aa': 2500000, 'ab': 190000, 'ae': 0,
                              'af': 17500000, 'ak': 11000000, 'am': 57000000})
        data = pd.DataFrame({'lang_name': lang, 'num_speakers': speakers})
        data
Out[ ]:
    lang_name  num_speakers
aa       Afar       2500000
ab     Abkhaz        190000
ae    Avestan             0
af  Afrikaans      17500000
ak       Akan      11000000
am    Amharic      57000000
```

`pd.DataFrame` as Generalized NumPyArray

- like `Series`, a `DataFrame` object has an `index` attribute
- additionally, it has a `columns` attribute containing an `Index` object with the column labels:

```
In [ ]: data.columns
Out[ ]: Index(['lang_name', 'num_speakers'], dtype='object')
```

- therefore, the `DataFrame` can be seen as a generalisation of a two-dimensional NumPy array where both rows and columns have a generalised index for data access

`pd.DataFrame` as Specialized Dictionary

- dictionary maps key to value, `DataFrame` maps column name to a `Series` of column data

Constructing `pd.DataFrame` Objects

- a single-column `DataFrame` can be constructed from a single Series
- any list of dictionaries can be turned into a `DataFrame`
- from a dictionary of `Series` objects (with column names as keys)
- by adding column and index names to a two-dimensional NumPy array

`pd.Index` as Immutable Array

- the `Index` object which implements the indices in a `Series` or `DataFrame` can be thought of as an **immutable array**
- an `Index` object `ind` in many ways operates like an array:
  - ▷ `ind[1]` provides the key stored in the second position of the index
  - ▷ `ind[::2]` creates a new index which only contains every second element
  - ▷ `ind.size` returns the number of entries in the index
  - ▷ `ind.shape` returns the length in both dimensions
  - ▷ `ind.ndim` returns the number of index dimensions
  - ▷ `ind.dtype` returns the type of the index keys
- the important difference is that the indices are immutable
  - ▷ no possibility to add a new key for a new position
  - ▷ no possibility to assign a different key to a position
  - ▷ no possibility to delete an entry
- immutability decreases the risk of unintended side effects

`pd.Index` as Ordered Set

- Pandas is designed to facilitate set operations across datasets
- as we will see, this primarily works through set operations on indices:
  - ▷ `indA.intersection(indB)` will contain only those keys which occur in both `indA` and `indB`, with the order depending on `indA`
  - ▷ `indA.intersection(indB)` will contain keys which occur in `indA` or `indB`, with enough copies to cover all repeated elements, in default order
  - ▷ `indA.symmetric_difference(indB)` will contain only those keys with occur in either `indA` or `indB`, but not in both

Table of Contents

Data Selection in Series as Dictionary

- `data[key]`
- `key in data`, `data.keys()`, `data.items()`
- `data[key] = value` to change or add value under a key

Data Selection in Series as One-Dimensional Array

- slicing by explicit index: `data["a":"c"]` (final index included!)
- slicing by implicit integer index: `data[0:2]` (final index excluded!)
- masking: `data[(data > 0.2) & (data < 0.8)]`

Indexers `loc` and `iloc`

- `data.loc` for indexing and slicing with reference to explicit index
- `data.iloc` for indexing and slicing with implicit python-style index
- especially relevant when working with integer indices
  (different index for `data[x]` and `data[a:b]`!)

Data Selection in Data Frame as Dictionary

- access to individual series via column name indexing: `data[num_speakers]`
- equivalently, the column names are available as fields: `data.num_speakers]` (only for strings, and in the absence of naming conflicts!)
- dictionary-style syntax can also be used to add new columns:
  `data["density"] = data["pop"] / data["area"]`
  (exploiting element-by-element arithmetic between `Series` objects)

Data Selection in Data Frame as Two-Dimensional Array

- underlying raw NumPy array accessible via `data.values`
- much array-based logic is available for data frames as well (e.g. `data.T`)
- index semantics is different, array-style indexing works via `data.iloc`
- `data.iloc` also makes broadcasting etc. available

Additional Indexing Conventions

- by default, indexing refers to columns, but slicing to rows
- direct masking operations are interpreted row-wise rather than column-wise

Table of Contents

Index Preservation under Universal Functions

- applying any NumPy ufunc on a Pandas object will return another Pandas object with the indices preserved

Index Alignment in Series

•

Index Alignment in Data Frames

-

Operations Between Series and Data Frames

-

Table of Contents

Data Loading and Storage

- there are many functions of shape `pd.read_X`, e.g. `pd.read_csv(filename)`

Table of Contents

Data Exploration and Summary Statistics

- `data.count()` retirms the number of non-NA values
- `data.sum()` returns a `Series` containing column sums
- `data.sum(axis="columns")` returns a `Series` containing row sums
- `data.mean()` returns the average of the values in each column
- `data.median()` returns the median of the values in each column
- `data.idxmax()` returns a `Series` of index values where the maximum values are attained
- `data.pct_change()` computes percent changes
- `data.corrwith(series)` computes pairwise correlations of each column with `series`
- `data.describe()` produces multiple summary statistics in one shot
- `data.unique()` produces an array of the unique values in a `Series`
- `data.value_counts()` computes a `Series` containing value frequencies
- `data.isin()` performs a vectorised set membership check and returns a boolean mask for filtering, e.g. `data[data.isin(["b", "c"])]`
- `pd.notnull(series)` returns a boolean Series of values where the value is NaN
- option `level` to reduce grouped by level if axis is hierarchically indexed

Sorting and Ranking

- `data.sort_index()` sorts the rows by their index
- `data.sort_values()` sorts the rows by their values
- `data.rank()`

Table of Contents

Assignment 3: Tasks

1) Create a new Jupyter notebook and import `pandas`. Load the Glottolog language database from the file `languages_and_dialects_geo.tsv` into a `DataFrame` object.
2) Explore the dataset and gain a first overview. What are the columns and their value ranges? How many entries does this database of languages and dialects have?
3) Extract the full inventory of macroareas into which this dataset partitions languages.
4) How many languages and dialects have an ISO 639-3 code associated with them?
5) For how many languages and dialects do we have latitude and longitude data?
6) Extract the name and macroarea of the northernmost language for which we have data.
7) Use the `pd.Series.quantile()` function to find out the latitude range which covers the central half of the world's languages. Is it symmetric around the equator?
8) Between which two languages do we find the largest gap in longitude, i.e. the longest stretch in west-east direction which is not inhabited by any other language?
9) How about the largest gap in longitude among languages of the Southern Hemisphere?

Preliminary Course Plan

1	**27/10**	**IPython and Jupyter**
2	**03/11**	**Introduction to NumPy**
3	**10/11**	**Pandas and Data Frames**
4	17/11	Data Cleaning and Preparation
5	24/11	Linguistic Preprocessing
6	01/12	Data Wrangling: Join, Combine, Reshape
7	08/12	Data Aggregation and Grouping
8	15/12	Visualisation with Seaborn
9	22/12	Modeling and Prediction
10	12/01	Classification
11	19/01	Clustering
12	26/01	Pattern Extraction and Density Estimation
13	02/02	Statistical Inference
14	09/02	Data Science Projects

Questions

Questions?

Comments?

Suggestions?