Python: pandas

Intro. to computing

`https://mandarm.github.io/computing.html`

Background: NumPy vs. Pandas

NumPy

- 'Clean', mostly numerical data
  (*structured arrays* also support general data)
- Elements indexed by non-negative integers

Pandas

- 'Noisy', real-world data with possibly missing entries
- Elements indexed by *labels* in addition to integers

Boilerplate

```
import numpy as np
import pandas as pd
rng = np.random.default_rng() # optional
```

Descriptions of many functions have been simplified in the following slides. For a complete description, see the Pandas documentation.

Basic data structures: Series, DataFrames, Index

Series I

- Series $\equiv$ values + index
  - values / data: one-dimensional array holding items of the same type (integers / strings / Python objects)
  - *index*: sequence of labels (0, 1, ... by default)
- NumPy arrays $\approx$ lists; Pandas Series $\approx$ dicts (see examples below)
- May be created from lists, NumPy ndarrays, scalars, dicts, etc.
- Index may be explicitly specified.
  - scalars repeated as necessary
  - length has to match for lists, arrays
  - entries *do not* have to be distinct

```
# Default index (0, 1, ...)
data = pd.Series([0.25, 0.5, 0.75, 1.0])
print(f'Values: {data.values}, Index: {data.index}')
print(data[1:4], "\n", data[0])
```

Series II

```
# Explicit index with repeating elements
new_index = ("A", "B", "C", "D", "C")
new_data = pd.Series([0.25, 0.5, 0.75, 1.0, 1.25], index=new_index)
print(f'Values: {new_data.values}, Index: {new_data.index}')

# dict like behaviour
print("A" in new_data)
print(new_data["F"])        # ERROR (KeyError)
print(new_data.get("F"))    # returns None (no error)

# non-dict-like behaviour
print(new_data['C'])       # 2 elements
print(new_data['B':'D'])   # slicing works
print(new_data['B':'C'])   # but not for repeated keys!

# From a scalar
x = pd.Series(5, index=[100, 200, 300])
```

Series III

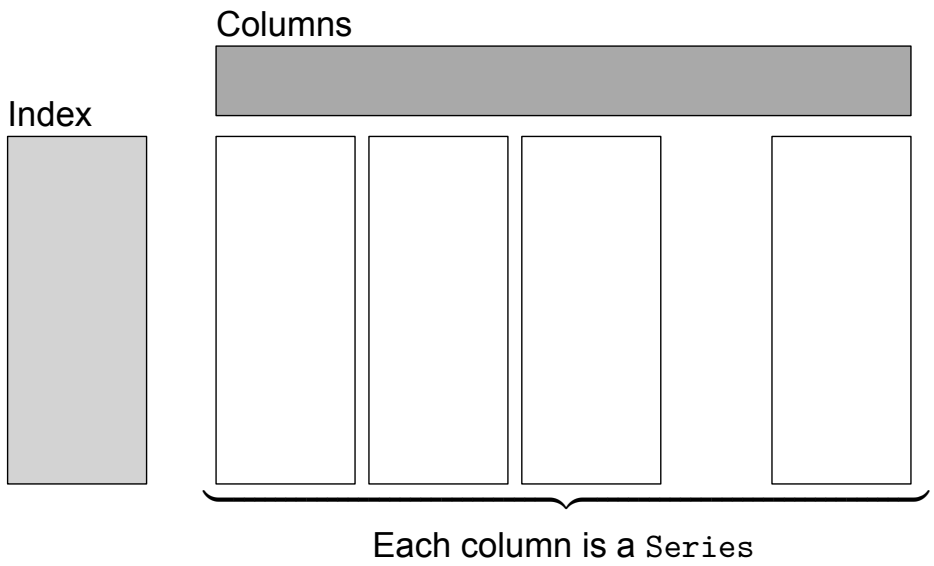
- Series from dictionaries
  - default index $\equiv$ dictionary keys
  - if index is explicitly provided, corresponding dict elements will be pulled out, and keys not in dict will be assigned NaN

```
# From a dictionary; default index  dict keys
y = pd.Series({2:'a', 1:'b', 3:'c'})
# may not be practical, but doable
y1 = pd.Series({2:'a', 1:'b', 3:'c', 'd':100})

# From a dictionary with explicit index  keys
z = pd.Series({2:'a', 1:'b', 3:'c'}, index=[3,2,10])
```

DataFrame I

- Two-dimensional array + row label (`index`) + column labels (`columns`)
  $\equiv$ sequence of `Series` objects that share the same index
- Interpretation: rows $\equiv$ data items, columns $\equiv$ attributes
- Different columns may have different types
- 2-D NumPy arrays $\approx$ lists of lists
  Pandas DataFrames $\approx$ dicts (keys $\equiv$ column names, values $\equiv$ Series)
  of dicts (keys = index, values = values)



DataFrame II

- May be created from
  - 2-D ndarrays
  - dict with lists / dicts / ndarrays / Series as values

```
# From 2-D NumPy arrays
X = rng.random((4,3))
Y = pd.DataFrame(X)       # default index, columns
print(X[0]), "\n", Y[0])

Z = pd.DataFrame(X, columns=['Bangalore', 'Delhi', 'Kolkata'],
                 index=list('abcd')])
```

Warning

- For 2-D `ndarrays`, `X[i]` is the i-th **row**.
- For `DataFrames`, `X[<label>]` is a **column**

DataFrame III

```
# From Series objects
population_dict = {'California': 39538223,
    'Texas': 29145505,
    'New York': 20201249,
    'Pennsylvania': 13002700,
    'Florida': 21538187, }
population = pd.Series(population_dict)

area_dict = {'California': 423967,
    'Florida': 170312,
    'New York': 141297,
    'Pennsylvania': 119280,
    'Texas': 695662, }
area = pd.Series(area_dict)

# Single-column DataFrame from single Series
dummy = pd.DataFrame(population, columns=['population'])
```

DataFrame IV

```
# Multi-column DataFrame from a dict of Series
states = pd.DataFrame({'population': population, 'area': area})
print(f'Row names: {states.index}\nColumn names: {states.columns}')
print(type(states['population'])) # Series

# Multi-column DataFrame from a dict of dicts
states = pd.DataFrame({'population': population_dict, 'area': area_dict
    })
print(f'Row names: {states.index}\nColumn names: {states.columns}')
print(type(states['population'])) # Series
```

DataFrame V

```
# Multi-column DataFrame from a heterogeneous dict
df2 = pd.DataFrame({
    "A": 1.0,     # scalars repeated as needed
    "C": pd.Series(1, index=list(range(4)), dtype="float32"), # these
    "D": np.array([3] * 4, dtype="int32"),                    # must be of
    "E": pd.Categorical(["test", "train", "test", "train"]),  # same length
    "F": "foo",   # scalars repeated as needed
})
```

Warning

Creating DataFrame from **dict** of dicts and **list** of dicts is very different!

```
confusing_example1 = pd.DataFrame([population_dict, area_dict])
confusing_example2 = pd.DataFrame([{'a': 1, 'b': 2}, {'b': 3, 'c': 4}])
```

Dictionary keys become columns!

Index

- Index $\equiv$ *immutable* array or *ordered* (multi)set
- Standard indexing, restricted slicing operations work.
- Share many attributes with NumPy arrays (`size`, `shape`, `dtype`)
- Supports set operations like union, intersection, difference
  - reduces all multisets to sets before operations

```
i1 = pd.Index([2, 3, 5, 7, 11])
i2 = pd.Index([1, 11, 9, 3, 2])

print(i1.union(i2))
print(i1.intersection(i2))
```

Reading and writing file data I

- Functions: `df = pd.read_X()`, `df.to_X()`
  where `X` may be `csv`, `excel`, `sql`, `json`, etc.
- `pd.read_csv(infile)`
  - `infile`: compulsory positional argument, may be
    - string, e.g. "/home/mandar/data/data.csv"
    - open file handle, e.g., `f = open(filename)`
  - `sep=` or `delimiter=` (default: comma (','))
  - `skiprows=`
    - number of lines to skip (single integer) at the start of the file, or
    - list of line numbers to skip (0-indexed)
  - `header=` line number that contains column names/labels (0-indexed)
    - counting starts after `skiprows` is processed
    - commented lines are also ignored (not counted) ???
    - pass `None` if column names is absent from input file
      (column names should be passed explicitly using `names`)

Reading and writing file data II

- `names=` user-defined column names
- `index_col=` label or index of column that should be used as row label
- and *many* more.

Examples: (see course page for example csv files)

```
# `Correct' way of reading various example data files
df = pd.read_csv("example1.csv", header=None)
df = pd.read_csv("example2.csv")
# Different rows have different no. of fields in example3.csv
# => no easy way to read it correctly
df = pd.read_csv("example4.csv", index_col=0)
```

Attributes, operators, functions

Digression: a convenience function

Reference: [JVDP] Ch. 18

```
def make_df(cols, ind):
    """Quickly make a DataFrame
       cols: list of column names
       ind:  list of indices"""
    data = {c: [str(c) + str(i) for i in ind] for c in cols}
    return pd.DataFrame(data)
# Example
df1 = make_df('ABC', range(3))


A = pd.DataFrame(rng.integers(1,50,(4,3)), index=range(4),
                 columns=list("ABC"))
```

DataFrames: attributes and basic functions I

- `df.shape`, `df.size`, `df.index`, `df.columns`, `df.T`
- `df.head(<optional no. of rows>)`: first few rows
  `df.tail(<optional no. of rows>)`: last few rows
- `df.describe()`: statistical summary
- `df.mean(axis=0)`, `df.var(axis=0)`

`axis` ≡ *direction of aggregation*

- `axis=0`, `axis='rows'` ≡ column-wise operation
- `axis=1`, `axis='columns'` ≡ row-wise operation

aliases are Pandas specific, not for NumPy

DataFrames: attributes and basic functions II

- `df.value_counts()`: Series containing frequency of each distinct row in `df`
  - `subset`: column label or list of labels (optional)
  - `normalize=False`: return relative frequencies instead of counts
  - `sort=True`: sort by frequencies (`False` ⇒ sort by column values??)

may be passed as positional parameters in this order

- `df.to_numpy()`: extract underlying NumPy array, without the index or column labels

Accessing elements I

Indexing (avoidable)

- `df["label"]`: returns a column as a Series
  - if label is valid identifier, `df.label` also works
- `df[["label2", "label3", "label1"]]`: returns multiple columns as a DataFrame

Accessing elements II

Using dedicated functions: ~~return *views*, not copies~~ (recommended)

- `df.loc[<row selector>, <column selector>]`
  - column selector is optional
  - select rows and/or columns by single label / list of labels / label slice
    - **Warning:** for slicing with `loc[]`, **BOTH** start and stop are included.
  - to select all rows / columns, use `:` as selector
- `df.iloc[<row selector>, <column selector>]`
  - as above, but select rows and/or columns using single integer index / list of integer indices / slicing
  - for slicing, stop is excluded as usual.
- `df.at()`, `df.iat()`
  - similar to `loc/iloc`, but can be used only for getting / setting single value
  - faster than `loc/iloc`

Note the **CORRECTION**

LATER: also Booleans

Accessing elements III

Examples:

```
df = pd.DataFrame(rng.random((4,4)), columns=list("wxyz"))
print(df.loc[:, "x"])
print(df.loc[:, ["x", "z"]])    # In all these examples,
print(df.loc[:, "x":"z"])       # the column "z" is included
v = df.loc[2:3, "x":"z"]        # in the result.
v.loc[2,"y"] = 100              # v is view, so df changes (see below)

df.iloc[3]
df.iloc[3:5, 0:2]               # Usual slicing (stop not included)
df.iloc[[1, 2, 4], [0, 2]]
```

CORRECTION

For recent versions of Pandas (3.0 and newer):

- `df.loc[...] = ...` changes `df`
- `v = df.loc[...]` (effectively) creates a copy ⇒ changes to `v` do not affect `df`

Ufuncs I

- Similar to NumPy ufuncs
- For unary operations: index and column labels preserved

Example:

```
df = pd.DataFrame(rng.random((4,4)), columns=list("wxyz"))
print(np.sin(df * np.pi / 4))
```

- For binary arithmetic operations / comparisons:
  - between two Series: *automatically aligns indices*
  - between two DataFrames: *automatically aligns both index and columns*
  - between DataFrame and Series: operates **row-wise** by default
- Missing data $\Rightarrow$ NaN

Justification:
rows $\equiv$ data items,
columns $\equiv$ attributes (possibly heterogeneous)

Ufuncs II

Examples:

```
# Examples with Series
area = pd.Series({'Alaska': 1723337, 'Texas': 695662,
                  'California': 423967})
population = pd.Series({'California': 39538223, 'Texas': 29145505,
                        'Florida': 21538187})
print(population / area) # Works on the union
# Compare area.index.union(population.index)

# Examples with DataFrames
A = pd.DataFrame(rng.integers(0, 20, (2, 2)), columns=['a', 'b'])
B = pd.DataFrame(rng.integers(0, 10, (3, 3)), columns=['b', 'a', 'c'])
print(A + B)   # handles "a" and "b" occurring in different orders
```

Ufuncs III

Python operator	Pandas method(s)
+	add
-	sub, subtract
*	mul, multiply
/	truediv, div, divide
//	floordiv
%	mod
**	pow

Names differ slightly from NumPy

- Methods take additional optional arguments to change default behaviour
  - `fill_value`: value used in place of existing missing (NaN) values, and any new element (needed for successful alignment) before computation

    Example: `A.add(B, fill_value=A.values.mean())`

    NOTE: if data in *both* locations is missing, result will still be NaN.
  - `axis`

Ufuncs IV

Alignment between DataFrame and Series

- `axis` specifies which axis is used for **matching / alignment**, i.e., whether Series index is matched with row index (`index`) or column index (`columns`)
- Possible values: 0 or `"rows"` or `"index"`, **or** 1 or `"columns"` (default)

```
A = pd.DataFrame(rng.integers(10, size=(3, 4)), columns=list("QRST"))

# Default: column-wise alignment, row-wise subtraction
print(A - A.iloc[0])

# axis=0 ⇒ row-wise alignment, col-wise subtraction
print(A.subtract(A.loc[:, "R"], axis=0))

halfrow = A.iloc[0, ::2]
```

Ufuncs V

Exercises:

- What is the type of `halfrow`?
- What is the value of `A - halfrow`?
- Why does `A - A.loc[:, "R"]` produce a 3×7 array of NaNs?
- Why does `A.subtract(halfrow, axis=0)` produce a 5×4 array of NaNs?

Boolean operations and comparisons I

- `df.XX(other, axis='columns')` (`XX` may be `eq`, `ne`, `le`, `lt`, `ge`, `gt`)
  - equivalent to usual comparison operators
  - `other` can be scalar, sequence, Series, or DataFrame
- `df.any(axis=0)`, `df.all(axis=0)`
  (`axis` $\equiv$ direction for aggregation)

Two warnings

- DataFrame does not have truth value; `if A: pass` raises `ValueError`
- NaN $\neq$ NaN `np.nan == np.nan` gives `False`

- `df1.equals(df2)`: tests equality, with NaNs in corresponding locations treated as equal
  (index order also needs to be same)

Boolean operations and comparisons II

```
# Comparisons with a scalar
print(A > 0)
print((A - halfrow) > 0)
print((A > 0).all())           # also try any()
print((A > 0).all(axis=None)) # aggregate over all elements

# Difference between == and equals() when NaN is present
((A - halfrow) == (A - halfrow)).all(axis=None)
(A - halfrow).equals(A - halfrow)

# Comparisons between DataFrames
A = pd.DataFrame(rng.integers(10, size=(3, 4)), columns=list("QRST"))
B = A.reindex([2,0,1])         # create a copy with reordered index
C = A.reindex(list("TQRS"), axis='columns')  # as above, with cols

# Operators align out-of-order columns / indices
print(A.eq(B).all(axis=None))

# equals() requires index and columns to match
print(A.equals(B))
```

Boolean indexing: selection using conditions I

- For selecting elements from Series, Boolean expressions work as for ndarrays
- For selecting rows from DataFrame, Boolean vector must have same length as index

Example:

```
# With Series
s = pd.Series(range(-3, 4))
print(s[s > 0])
print(s[(s < -1) | (s > 0.5)])
print(s[~(s < 0)])
```

Digression: `map()`

`map(function, iterable, /, *iterables, strict=False)`
Make an iterator that applies `function` to arguments taken from each of the iterables, one by one. Stops when the shortest iterable is exhausted.
`strict=True` $\Rightarrow$ raise a `ValueError` if one argument is exhausted before others

Boolean indexing: selection using conditions II

```
df = pd.DataFrame(rng.integers(50, size=(8, 4)), columns=list("ABCD"))
print(df.loc[df.loc[:,'A'] > 0]) # "Avoidable" style: (df[df['A'] > 0])

# More complex examples
df2 = pd.DataFrame({'a': ['one','one','two','three','two','one','six'],
                    'b': ['x', 'y', 'y', 'x', 'y', 'x', 'x'],
                    'c': np.random.randn(7)})

# Select only 'two' and 'three'
criterion = df2['a'].map(lambda x: x.startswith('t'))
print(df2[criterion])
# equivalent but slower
print(df2[[x.startswith('t') for x in df2['a']]])

# Multiple criteria
print(df2[criterion & (df2['b'] == 'x')])
print(df2.loc[criterion & (df2['b'] == 'x'), 'b':'c'])
```

Boolean indexing: selection using conditions III

Warning: `loc` vs. `iloc`

- For `loc`, selection can be done by array **or** Series.
- For `iloc`, selection can **only** be done by array, **not** Series.
  (use `.values` if needed)

```
df = pd.DataFrame([[1, 2], [3, 4], [5, 6]], index=list('abc'),
                  columns=['A', 'B'])
s = (df['A'] > 2)     # s is a Series
df.loc[s, 'B']        # works
df.iloc[s, 1]         # does not work
df.iloc[s.values, 1]  # works
```

Exercise: Change all the "avoidable" style indexing operations above to use `.loc[]`.

Adding/deleting rows/columns I

- Assignment via `df.at`, `df.loc`

Examples:

```
df.loc[5,:] = rng.random(4)
df.loc[6,] = rng.random(4)
df.loc[3,"a"] = "hello"
```

  - missing values filled with NaN as usual
  - *will not work with* `df.iat`, `df.iloc` (index out of bound error)

Adding/deleting rows/columns II

- `pd.concat(<list of Series or DataFrames>, axis=0)`
  append rows or columns (with `axis=1`)
  - `axis`: direction in which things to be concatenated are arranged
    `axis=0` $\Rightarrow$ rows, `axis=1` $\Rightarrow$ cols
  - aligns indexes / columns, unmatched 'cells' filled with NaN
  - returns copy; does not change arguments
  - other optional arguments
    - `join='outer'` (or `'inner'`): result contains all (union) labels, or only common (intersection) labels
    - `ignore_index`: create a new integer index (0, 1, ...)
    - `verify_integrity=False`: causes error if `index`es overlap

Adding/deleting rows/columns III

Examples:

```
df1 = pd.DataFrame(rng.random((4,4)), columns=list("wxyz"))
df2 = pd.DataFrame(rng.random((4,4)), columns=list("wxyz"))
df3 = pd.DataFrame(rng.random((4,4)), columns=list("abcd"))
print(pd.concat([df1,df2]))             # Don't forget to put
print(pd.concat([df1,df3]))             # DFs to be concatenated
print(pd.concat([df1,df3],axis=1))      # in a list.
```

Note from the official documentation

Adding a column to a DataFrame is relatively fast. However, adding a row requires a copy, and may be expensive. We recommend passing a pre-built list of records to the DataFrame constructor instead of building a DataFrame by iteratively appending records to it.

Adding/deleting rows/columns IV

- `df.drop( labels, *, axis=0, index=None, columns=None)`
  - `labels`: single row/column label or list of row/column labels (don't use tuples)
  - `index=L` may be used as replacement for `labels=L, axis=0`
  - `columns=C` may be used as replacement for `labels=C, axis=1`
- `df.pop(<column name>)`: return column as Series, and drop from frame

Sorting

- `df.sort_index(axis=0, ascending=True, inplace=False, key=)`: sort by *row labels* (`axis=0`) or *column labels* (`axis=1`)
- `df.sort_values(by=<column label OR list of column labels>)`
  (also takes ascending and inplace as above)

```
df = pd.DataFrame([[1, 8, 6], [9, 4, 5], [7, 5, 3]])
df1 = df.sort_index(axis=0, ascending=False)
df2 = df.sort_index(axis=1, ascending=False)
df3 = df.sort_values(by=0, ascending=False)
df4 = df.sort_values(by=[1,2], ascending=False)
```

[SKIP] Functions for handling missing data

- `X.isna()`, `X.notna()`, `X.isnull()`, `X.notnull()`: generates a Boolean mask indicating *missing* or *valid* values, resp.
- `X.dropna()`
  - `axis`: remove rows (`axis=0`) or columns (`axis=1`) which contain missing values
  - `inplace=False`: create new copy or modify in place
- `X.fillna(value, *, axis=0, inplace=False, method=None)`
  - `value`: scalar (constant, e.g., 0)

    (dict / Series / DataFrame for more complex filling)
  - `method`: `'bfill'` or `'ffill'` (use next / previous valid value to fill gap)

[SKIP] Representation of missing data I

- Use *sentinels*: bit patterns representing (type-specific) invalid values
- Nullable Dtypes, `pd.NA`
- Sentinels used by NumPy: `NaN`, `None`
  `NaN`: for floating-point (all numeric?) types, `None`: for all other types (???)
- Pandas handles `NaN`, `None` almost interchangeably

```
s = pd.Series([1, np.nan, 2, None]) # 1., NaN, 2., NaN (dtype: float64)
x = pd.Series(range(2), dtype=int)  # 0, 1 (dtype: int64)
x[0] = None  # x = NaN, 1.0 (dtype: float64)

%timeit np.arange(1E6, dtype=int).sum()
%timeit np.arange(1E6, dtype=float).sum()
%timeit np.arange(1E6, dtype=object).sum()

# Exercise: save the above arrays, change one element to NaN and None
# in turn and rerun the timeit commands.
```

[SKIP] Representation of missing data II

NaN vs. None

`None` (Python specific)

- Containing ndarray must have `dtype=object`
- Handling/data operations have to be done at the Python level ⇒ slow (see example below)
- Arithmetic operations not supported ⇒ `sum()`, `min()` do not work

`NaN`

- Special floating-point value supported by **ALL** (IEEE 754) standards-compliant systems
- Handling/data operations can be done in hardware (floating point processors) ⇒ fast
- All arithmetic operations supported by default (*)
- No equivalent for integers, strings, or other types

[SKIP] Representation of missing data III

More about NaN

- Result of almost all arithmetic operations with NaN is also NaN.
- NaN-aware NumPy aggregation functions: `np.nanX()`, where `X` is `sum`, `max`, `min`, etc.

[SKIP] Database like join operations I

`df.merge(data)` or `pd.merge(d1, d2)`

- Merging is done on a *single* column
- *One-to-one joins*: values in the common column are all distinct
- *Many-to-one joins*: values in the left column are repeated
- *Many-to-many joins*: values in both left and right columns are repeated
- Indexes are generally discarded, unless merging by `left_index` or `right_index`
- Optional keyword arguments
  - `on`, `left_on`, `right_on`: specify which column to merge on
  - `how`: `"inner"` (intersection, default) or `"outer"` (union), `"left"`, `"right"`

[SKIP] Database like join operations II

```
left = pd.DataFrame({"key": ["hello", "hello"], "lval": [1, 2]})
right = pd.DataFrame({"key": ["hello", "hello"], "rval": [4, 5]})

pd.merge(left, right, on="key")
# Yields
#     key       lval  rval
# 0   hello     1     4
# 1   hello     1     5
# 2   hello     2     4
# 3   hello     2     5

left = pd.DataFrame({"key": ["hello", "bye"], "lval": [1, 2]})
right = pd.DataFrame({"key": ["hello", "bye"], "rval": [4, 5]})
pd.merge(left, right, on="key")
# Yields
#     key    lval  rval
# 0   hello  1     4
# 1   bye    2     5
```

[SKIP] Database like join operations III

```
df = df1.merge(df2, left_on='<column name>', right_on='<column name>')
```

[SKIP] Iterating over rows / columns I

`df.apply(f)`: apply function `f()` to each row/column in turn

- row data (`axis=1`) or column data (`axis=0`) is passed to `f()` as a Series
- Other optional arguments
  - `axis=0`: specifies which axis' index is the index of the Series passed to `f()`
  - `raw=False`: determines if row or column is passed as a Series or ndarray object
  - `args=()`: positional arguments to pass to `f()` in addition to the array/series

[SKIP] Iterating over rows / columns II

Example:

```
>>> df
A  B
0  4  9
1  4  9
2  4  9

>>> df.apply(np.sum, axis=0) # axis = 0 => for each column (default)
A    12
B    27
dtype: int64

>>> df.apply(np.sum, axis=1) # axis = 1 => for each row
0    13
1    13
2    13
dtype: int64
```

References

- [JVDP] *Python Data Science Handbook*, Jake VanderPlas, 2nd ed. (2023), O'Reilly Media; Part III, Chapters 13–16, 18, 19. https://jakevdp.github.io/PythonDataScienceHandbook/
- Official documentation https://pandas.pydata.org/docs/getting_started/intro_tutorials
- Cheatsheet: https://pandas.pydata.org/Pandas_Cheat_Sheet.pdf
- Topics to skip
  - special data types like date ranges
  - hierarchical / multi index
  - pivot tables
  - time series