tafra

Release 1.0.10

David S. Fulford

Nov 22, 2022

CONTENTS

The `tafra` began life as a thought experiment: how could we reduce the idea of a da*tafra*me (as expressed in libraries like `pandas` or languages like R) to its useful essence, while carving away the cruft? The original proof of concept stopped at "group by".

This library expands on the proof of concept to produce a practically useful `tafra`, which we hope you may find to be a helpful lightweight substitute for certain uses of `pandas`.

A `tafra` is, more-or-less, a set of named *columns* or *dimensions*. Each of these is a typed `numpy` array of consistent length, representing the values for each column by *rows*.

The library provides lightweight syntax for manipulating rows and columns, support for managing data types, iterators for rows and sub-frames, *pandas*-like "transform" support and conversion from *pandas* Dataframes, and SQL-style "group by" and join operations.

Tafra	Tafra
Aggregations	Union, GroupBy, Transform, IterateBy, InnerJoin, LeftJoin, CrossJoin
Aggregation Helpers	union, union_inplace, group_by, transform, iterate_by, inner_join, left_join, cross_join
Constructors	as_tafra, from_dataframe, from_series, from_records
SQL Readers	read_sql, read_sql_chunks
Destructors	to_records, to_list, to_tuple, to_array, to_pandas
Properties	rows, columns, data, dtypes, size, ndim, shape
Iter Methods	iterrows, itertuples, itercols
Functional Methods	row_map, tuple_map, col_map, pipe
Dict-like Methods	keys, values, items, get, update, update_inplace, update_dtypes, update_dtypes_inplace
Other Helper Methods	select, head, copy, rename, rename_inplace, coalesce, coalesce_inplace, _coalesce_dtypes, delete, delete_inplace
Printer Methods	pprint, pformat, to_html
Indexing Methods	_slice, _index, _ndindex

CHAPTER

ONE

GETTING STARTED

Install the library with pip:

```
pip install tafra
```

1.1 A short example

```
>>> from tafra import Tafra

>>> t = Tafra({
...     'x': np.array([1, 2, 3, 4]),
...     'y': np.array(['one', 'two', 'one', 'two'], dtype='object'),
... })

>>> t.pformat()
Tafra(data = {
 'x': array([1, 2, 3, 4]),
 'y': array(['one', 'two', 'one', 'two'])},
dtypes = {
 'x': 'int', 'y': 'object'},
rows = 4)

>>> print('List:', '\n', t.to_list())
List:
 [array([1, 2, 3, 4]), array(['one', 'two', 'one', 'two'], dtype=object)]

>>> print('Records:', '\n', tuple(t.to_records()))
Records:
 ((1, 'one'), (2, 'two'), (3, 'one'), (4, 'two'))

>>> gb = t.group_by(
...      ['y'], {'x': sum}
... )

>>> print('Group By:', '\n', gb.pformat())
Group By:
Tafra(data = {
 'x': array([4, 6]), 'y': array(['one', 'two'])},
dtypes = {
```

(continues on next page)

(continued from previous page)

```
 'x': 'int', 'y': 'object'},
rows = 2)
```

1.2 Flexibility

Have some code that works with `pandas`, or just a way of doing things that you prefer? `tafra` is flexible:

```
>>> df = pd.DataFrame(np.c_[
...     np.array([1, 2, 3, 4]),
...     np.array(['one', 'two', 'one', 'two'])
... ], columns=['x', 'y'])

>>> t = Tafra.from_dataframe(df)
```

And going back is just as simple:

```
>>> df = pd.DataFrame(t.data)
```

CHAPTER

TWO

TIMINGS

In this case, lightweight also means performant. Beyond any additional features added to the library, `tafra` should provide the necessary base for organizing data structures for numerical processing. One of the most important aspects is fast access to the data itself. By minimizing abstraction to access the underlying `numpy` arrays, `tafra` provides an order of magnitude increase in performance.

- **Import note** If you assign directly to the `Tafra.data` or `Tafra._data` attributes, you *must* call `Tafra._coalesce_dtypes` afterwards in order to ensure the typing is consistent.

Construct a `Tafra` and a `DataFrame`:

```
>>> tf = Tafra({
...     'x': np.array([1., 2., 3., 4., 5., 6.]),
...     'y': np.array(['one', 'two', 'one', 'two', 'one', 'two'], dtype='object'),
...     'z': np.array([0, 0, 0, 1, 1, 1])
... })

>>> df = pd.DataFrame(t.data)
```

2.1 Read Operations

Direct access:

```
>>> %timemit x = t._data['x']
55.3 ns ± 5.64 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)
```

Indirect with some penalty to support `Tafra` slicing and `numpy`'s advanced indexing:

```
>>> %timemit x = t['x']
219 ns ± 71.6 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

`pandas` timing:

```
>>> %timemit x = df['x']
1.55 µs ± 105 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

This is the fastest methed for accessing the numpy array among alternatives of `df.values()`, `df.to_numpy()`, and `df.loc[]`.

2.2 Assignment Operations

Direct access is not recommended as it avoids the validation steps, but it does provide fast access to the data attribute:

```
>>> x = np.arange(6)

>>> %timeit tf._data['x'] = x
65 ns ± 5.55 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)
```

Indidrect access has a performance penalty due to the validation checks to ensure consistency of the `tafra`:

```
>>> %timeit tf['x'] = x
7.39 µs ± 950 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

Even so, there is considerable performance improvement over `pandas`.

`pandas` timing:

```
>>> %timeit df['x'] = x
47.8 µs ± 3.53 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

2.3 Grouping Operations

`tafra` also excels at aggregation methods, the primary of which are a SQL-like `GROUP BY` and the split-apply-combine equivalent to a SQL-like `GROUP BY` following by a `LEFT JOIN` back to the original table.

```
>>> %timeit tf.group_by(['y', 'z'], {'x': sum})
138 µs ± 4.03 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)

>>> %timeit tf.transform(['y', 'z'], {'sum_x': (sum, 'x')})
161 µs ± 2.31 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

The equivalent `pandas` functions are given below. They require a chain of several object methods to perform the same role, and the transform requires a copy operation and assignment into the copied `DataFrame` in order to preserve immutability.

```
>>> %timeit df.groupby(['y','z']).agg({'x': 'sum'}).reset_index()
2.5 ms ± 177 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

>>> %%timeit
... tdf = df.copy()
... tdf['x'] = df.groupby(['y', 'z'])[['x']].transform(sum)
2.81 ms ± 143 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

CHAPTER

THREE

CONTENTS

3.1 API Reference

3.1.1 Summary

Tafra

Tafra(data, dtypes, validate, check_rows)	A minimalist dataframe.

Aggregations

Union()	Union two `Tafra` together.
GroupBy(group_by_cols, aggregation, iter_fn)	Aggregation by a set of unique values.
Transform(group_by_cols, aggregation, iter_fn)	Apply a function to each unique set of values and join to the original table.
IterateBy(group_by_cols)	A generator that yields a `Tafra` for each set of unique values.
InnerJoin(on, select)	An inner join.
LeftJoin(on, select)	A left join.
CrossJoin(on, select)	A cross join.

Methods

from_records(records, columns[, dtypes])	Construct a `Tafra` from an Iterator of records, e.g.
from_dataframe(df[, dtypes])	Construct a `Tafra` from a `pandas.DataFrame`.
from_series(s[, dtype])	Construct a `Tafra` from a `pandas.Series`.
read_sql(query, cur)	Execute a SQL SELECT statement using a `pyodbc.Cursor` and return a Tuple of column names and an Iterator of records.
read_sql_chunks(query, cur[, chunksize])	Execute a SQL SELECT statement using a `pyodbc.Cursor` and return a Tuple of column names and an Iterator of records.
read_csv(csv_file[, guess_rows, missing, dtypes])	Read a CSV file with a header row, infer the types of each column, and return a Tafra containing the file's contents.

continues on next page

Table 1 – continued from previous page

as_tafra(maybe_tafra)	Returns the unmodified *tafra* ` if already a Tafra, else construct a Tafra from known types or subtypes of DataFrame or *dict*.
to_records([columns, cast_null])	Return a Iterator of Tuple, each being a record (i.e.
to_list([columns, inner])	Return a list of homogeneously typed columns (as numpy.ndarray).
to_tuple([columns, name, inner])	Return a NamedTuple or Tuple.
to_array([columns])	Return an object array.
to_pandas([columns])	Construct a pandas.DataFrame.
to_csv(filename[, columns])	Write the Tafra to a CSV.
rows	The number of rows of the first item in data.
columns	The names of the columns.
data	The Tafra data.
dtypes	The Tafra dtypes.
size	The Tafra size.
ndim	The Tafra number of dimensions.
shape	The Tafra shape.
head([n])	Display the head of the Tafra.
keys()	Return the keys of data, i.e. like dict.keys().
values()	Return the values of data, i.e. like dict.values().
items()	Return the items of data, i.e. like dict.items().
get(key[, default])	Return from the get() function of data, i.e. like dict.get().
iterrows()	Yield rows as Tafra.
itertuples([name])	Yield rows as NamedTuple, or if name is None, yield rows as tuple.
itercols()	Yield columns as Tuple[str, np.ndarray], where the str is the column name.
row_map(fn, *args, **kwargs)	Map a function over rows.
tuple_map(fn, *args, **kwargs)	Map a function over rows.
col_map(fn, *args, **kwargs)	Map a function over columns.
key_map(fn, *args, **kwargs)	Map a function over columns like :meth:col_map, but return Tuple of the key with the function result.
pipe(fn, *args, **kwargs)	Apply a function to the Tafra and return the resulting Tafra.
select(columns)	Use column names to slice the Tafra columns analogous to SQL SELECT.
copy([order])	Create a copy of a Tafra.
update(other)	Update the data and dtypes of this Tafra with another Tafra.
update_inplace(other)	Inplace version.
update_dtypes(dtypes)	Apply new dtypes.
update_dtypes_inplace(dtypes)	Inplace version.
parse_object_dtypes()	Parse the object dtypes using the ObjectFormatter instance.
parse_object_dtypes_inplace()	Inplace version.
rename(renames)	Rename columns in the Tafra from a dict.
rename_inplace(renames)	In-place version.
coalesce(column, fills)	Fill None values from fills.
coalesce_inplace(column, fills)	In-place version.
_coalesce_dtypes()	Update dtypes with missing keys that exist in data.

continues on next page

Table 1 – continued from previous page

delete(columns)	Remove a column from *data* and *dtypes*.
delete_inplace(columns)	In-place version.
pprint([indent, width, depth, compact])	Pretty print.
pformat([indent, width, depth, compact])	Format for pretty printing.
to_html([n])	Construct an HTML table representation of the `Tafra` data.
_slice(_slice)	Use a `slice` to slice the `Tafra`.
_iindex(index)	Use a :class`int` to slice the `Tafra`.
_aindex(index)	Use numpy advanced indexing to slice the `Tafra`.
_ndindex(index)	Use `numpy.ndarray` indexing to slice the `Tafra`.

Helper Methods

union(other)	Helper function to implement *tafra.group.Union.apply()*.
union_inplace(other)	Inplace version.
group_by(columns[, aggregation, iter_fn])	Helper function to implement *tafra.group.GroupBy.apply()*.
transform(columns[, aggregation, iter_fn])	Helper function to implement *tafra.group.Transform.apply()*.
iterate_by(columns)	Helper function to implement *tafra.group.IterateBy.apply()*.
inner_join(right, on[, select])	Helper function to implement *tafra.group.InnerJoin.apply()*.
left_join(right, on[, select])	Helper function to implement *tafra.group.LeftJoin.apply()*.
cross_join(right[, select])	Helper function to implement *tafra.group.CrossJoin.apply()*.

Object Formatter

ObjectFormatter	A dictionary that contains mappings for formatting objects.

3.1.2 Detailed Reference

Tafra

Methods

class tafra.base.**Tafra**(*data: ~dataclasses.InitVar = <property object>*, *dtypes: ~dataclasses.InitVar = <property object>*, *validate: ~dataclasses.InitVar = True*, *check_rows: bool = True*)

A minimalist dataframe.

Constructs a *Tafra* from `dict` of data and (optionally) dtypes. Types on parameters are the types of the constructed *Tafra*, but attempts are made to parse anything that "looks" like the correct data structure, including `Iterable`, `Iterator`, `Sequence`, and `Mapping` and various combinations.

Parameters are given as an `InitVar`, defined as:

```
InitVar = Union[Tuple[str, Any], _Mapping, Sequence[_Element],
Iterable[_Element], Iterator[_Element], enumerate]

_Mapping = Union[Mapping[str, Any], Mapping[int, Any], Mapping[float, Any],
Mapping[bool, Any]

_Element = Union[Tuple[Union[str, int, float, np.ndarray], Any], List[Any],
Mapping]
```

Parameters

- **data** (*InitVar*) – The data of the Tafra.
- **dtypes** (*InitVar*) – The dtypes of the columns.
- **validate** (*bool = True*) – Run validation checks of the data. False will improve performance, but *data* and *dtypes* will not be validated for conformance to expected data structures.
- **check_rows** (*bool = True*) – Run row count checks. False will allow columns of differing lengths, which may break several methods.

Returns
tafra – The constructed *Tafra*.

Return type
Tafra

classmethod from_dataframe(*df: DataFrame*, *dtypes: Optional[Dict[str, Any]] = None*, ***kwargs: Any*) → *Tafra*

Construct a *Tafra* from a `pandas.DataFrame`. If `dtypes` are not given, take from `pandas.DataFrame.dtypes`.

Parameters

- **df** (*pandas.DataFrame*) – The dataframe used to build the *Tafra*.
- **dtypes** (*Optional[Dict[str, Any]] = None*) – The dtypes of the columns.

Returns
tafra – The constructed *Tafra*.

Return type
Tafra

classmethod from_series(*s: Series*, *dtype: Optional[str] = None*, ***kwargs: Any*) → *Tafra*

Construct a *Tafra* from a `pandas.Series`. If `dtype` is not given, take from `pandas.Series.dtype`.

Parameters

- **df** (*pandas.Series*) – The series used to build the *Tafra*.
- **dtype** (*Optional[str] = None*) – The dtypes of the column.

Returns
tafra – The constructed *Tafra*.

Return type
Tafra

classmethod from_records(*records: Iterable[Iterable[Any]]*, *columns: Iterable[str]*, *dtypes: Optional[Iterable[Any]] = None*, ***kwargs: Any*) → *Tafra*

Construct a *Tafra* from an Iterator of records, e.g. from a SQL query. The records should be a nested Iterable, but can also be fed a cursor method such as `cur.fetchmany()` or `cur.fetchall()`.

Parameters

- **records** (*ITerable[Iteralble[str]]*) – The records to turn into a *Tafra*.
- **columns** (*Iterable[str]*) – The column names to use.
- **dtypes** (*Optional[Iterable[Any]] = None*) – The dtypes of the columns.

Returns
tafra – The constructed *Tafra*.

Return type
Tafra

classmethod read_sql(*query: str*, *cur: Cursor*) → *Tafra*

Execute a SQL SELECT statement using a `pyodbc.Cursor` and return a Tuple of column names and an Iterator of records.

Parameters

- **query** (*str*) – The SQL query.
- **cur** (*pyodbc.Cursor*) – The pyodbc cursor.

Returns
tafra – The constructed *Tafra*.

Return type
Tafra

classmethod read_sql_chunks(*query: str*, *cur: Cursor*, *chunksize: int = 100*) → Iterator[*Tafra*]

Execute a SQL SELECT statement using a `pyodbc.Cursor` and return a Tuple of column names and an Iterator of records.

Parameters

- **query** (*str*) – The SQL query.
- **cur** (*pyodbc.Cursor*) – The pyodbc cursor.

Returns
tafra – The constructed *Tafra*.

Return type
Tafra

classmethod read_csv(*csv_file: Union[str, Path, TextIOWrapper, IO[str]]*, *guess_rows: int = 5*, *missing: Optional[str] = ''*, *dtypes: Optional[Dict[str, Any]] = None*, ***csvkw: Dict[str, Any]*) → *Tafra*

Read a CSV file with a header row, infer the types of each column, and return a Tafra containing the file's contents.

Parameters

- **csv_file** (*Union[str, TextIOWrapper]*) – The path to the CSV file, or an open file-like object.
- **guess_rows** (*int*) – The number of rows to use when guessing column types.

- **dtypes** (*Optional[Dict[str, str]]*) – dtypes by column name; by default, all dtypes will be inferred from the file contents.
- ****csvkw** (*Dict[str, Any]*) – Additional keyword arguments passed to csv.reader.

Returns
tafra – The constructed *Tafra*.

Return type
Tafra

classmethod as_tafra(*maybe_tafra: Union[*Tafra*, DataFrame, Series, Dict[str, Any], Any]*) → Optional[*Tafra*]

Returns the unmodified *tafra* ` if already a *Tafra*, else construct a *Tafra* from known types or subtypes of `DataFrame` or *dict*. Structural subtypes of `DataFrame` or `Series` are also valid, as are classes that have `cls.__name__ == 'DataFrame'` or `cls.__name__ == 'Series'`.

Parameters
maybe_tafra (*Union['tafra', DataFrame]*) – The object to ensure is a *Tafra*.

Returns
tafra – The *Tafra*, or None is `maybe_tafra` is an unknown type.

Return type
Optional[*Tafra*]

to_records(*columns: Optional[Iterable[str]] = None*, *cast_null: bool = True*) → Iterator[Tuple[Any, ...]]

Return a `Iterator` of `Tuple`, each being a record (i.e. row) and allowing heterogeneous typing. Useful for e.g. sending records back to a database.

Parameters

- **columns** (*Optional[Iterable[str]] = None*) – The columns to extract. If `None`, extract all columns.
- **cast_null** (*bool*) – Cast `np.nan` to None. Necessary for :mod:`pyodbc`

Returns
records

Return type
Iterator[Tuple[Any, ...]]

to_list(*columns: Optional[Iterable[str]] = None*, *inner: bool = False*) → Union[List[ndarray], List[List[Any]]]

Return a list of homogeneously typed columns (as `numpy.ndarray`). If a generator is needed, use *to_records()*. If `inner == True` each column will be cast from `numpy.ndarray` to a `List`.

Parameters

- **columns** (*Optional[Iterable[str]] = None*) – The columns to extract. If `None`, extract all columns.
- **inner** (*bool = False*) – Cast all `np.ndarray` to :class`List`.

Returns
list

Return type
Union[List[np.ndarray], List[List[Any]]]

to_tuple(*columns: Optional[Iterable[str]] = None*, *name: Optional[str] = 'Tafra'*, *inner: bool = False*) → Union[Tuple[ndarray], Tuple[Tuple[Any, ...]]]

Return a `NamedTuple` or `Tuple`. If a generator is needed, use *to_records()*. If `inner == True` each column will be cast from `np.ndarray` to a `Tuple`. If *name* is *None*, returns a `Tuple` instead.

Parameters

- **columns** (*Optional[Iterable[str]] = None*) – The columns to extract. If `None`, extract all columns.
- **name** (*Optional[str] = 'Tafra'*) – The name for the `NamedTuple`. If `None`, construct a `Tuple` instead.
- **inner** (*bool = False*) – Cast all `np.ndarray` to :class`List`.

Returns
list

Return type
Union[Tuple[np.ndarray], Tuple[Tuple[Any, ...]]]

to_array(*columns: Optional[Iterable[str]] = None*) → ndarray

Return an object array.

Parameters
columns (*Optional[Iterable[str]] = None*) – The columns to extract. If `None`, extract all columns.

Returns
array

Return type
np.ndarray

to_pandas(*columns: Optional[Iterable[str]] = None*) → DataFrame

Construct a `pandas.DataFrame`.

Parameters
columns (*Iterable[str]*) – The columns to write. IF `None`, write all columns.

Returns
dataframe

Return type
`pandas.DataFrame`

to_csv(*filename: Union[str, Path, TextIOWrapper, IO[str]]*, *columns: Optional[Iterable[str]] = None*) → None

Write the *Tafra* to a CSV.

Parameters

- **filename** (*Union[str, Path]*) – The path of the filename to write.
- **columns** (*Iterable[str]*) – The columns to write. IF `None`, write all columns.

rows

The number of rows of the first item in *data*. The `len()` of all items have been previously validated.

Returns
rows – The number of rows of the *Tafra*.

Return type
int

columns

The names of the columns. Equivalent to *Tafra*.keys().

Returns
columns – The column names.

Return type
Tuple[str, ...]

data: InitVar

The *Tafra* data.

Returns
data – The data.

Return type
Dict[str, np.ndarray]

dtypes: InitVar

The *Tafra* dtypes.

Returns
dtypes – The dtypes.

Return type
Dict[str, str]

size

The *Tafra* size.

Returns
size – The size.

Return type
int

ndim

The *Tafra* number of dimensions.

Returns
ndim – The number of dimensions.

Return type
int

shape

The *Tafra* shape.

Returns
shape – The shape.

Return type
int

head(*n: int = 5*) → *Tafra*

Display the head of the *Tafra*.

Parameters
n (*int = 5*) – The number of rows to display.

Returns
None

Return type
None

keys() → KeysView[str]

Return the keys of *data*, i.e. like `dict.keys()`.

Returns
data keys – The keys of the data property.

Return type
KeysView[str]

values() → ValuesView[ndarray]

Return the values of *data*, i.e. like `dict.values()`.

Returns
data values – The values of the data property.

Return type
ValuesView[np.ndarray]

items() → ItemsView[str, ndarray]

Return the items of *data*, i.e. like `dict.items()`.

Returns
items – The data items.

Return type
ItemsView[str, np.ndarray]

get(*key: str*, *default: Optional[Any] = None*) → Any

Return from the *get()* function of *data*, i.e. like `dict.get()`.

Parameters

- **key** (*str*) – The key value in the data property.
- **default** (*Any*) – The default to return if the key does not exist.

Returns
value – The value for the key, or the default if the key does not exist.

Return type
Any

iterrows() → Iterator[*Tafra*]

Yield rows as *Tafra*. Use *itertuples()* for better performance.

Returns
tafras – An iterator of *Tafra*.

Return type
Iterator[*Tafra*]

itertuples(*name: Optional[str] = 'Tafra'*) → Iterator[Tuple[Any, ...]]

Yield rows as `NamedTuple`, or if `name` is `None`, yield rows as `tuple`.

Parameters
name (*Optional[str] = 'Tafra'*) – The name for the `NamedTuple`. If `None`, construct a `Tuple` instead.

Returns
tuples – An iterator of `NamedTuple`.

Return type
Iterator[NamedTuple[Any, ...]]

`itercols()` → Iterator[Tuple[str, ndarray]]

Yield columns as `Tuple[str, np.ndarray]`, where the `str` is the column name.

Returns
tuples – An iterator of *Tafra*.

Return type
Iterator[Tuple[str, np.ndarray]]

`row_map`(*fn: Callable[[...], Any]*, **args: Any*, ***kwargs: Any*) → Iterator[Any]

Map a function over rows. To apply to specific columns, use *select()* first. The function must operate on *Tafra*.

Parameters

- **fn** (*Callable[..., Any]*) – The function to map.
- ***args** (*Any*) – Additional positional arguments to `fn`.
- ****kwargs** (*Any*) – Additional keyword arguments to `fn`.

Returns
iter_tf – An iterator to map the function.

Return type
Iterator[Any]

`tuple_map`(*fn: Callable[[...], Any]*, **args: Any*, ***kwargs: Any*) → Iterator[Any]

Map a function over rows. This is faster than *row_map()*. To apply to specific columns, use *select()* first. The function must operate on `NamedTuple` from *itertuples()*.

Parameters

- **fn** (*Callable[..., Any]*) – The function to map.
- **name** (*Optional[str] = 'Tafra'*) – The name for the `NamedTuple`. If `None`, construct a `Tuple` instead. Must be given as a keyword argument.
- ***args** (*Any*) – Additional positional arguments to `fn`.
- ****kwargs** (*Any*) – Additional keyword arguments to `fn`.

Returns
iter_tf – An iterator to map the function.

Return type
Iterator[Any]

`col_map`(*fn: Callable[[...], Any]*, **args: Any*, ***kwargs: Any*) → Iterator[Any]

Map a function over columns. To apply to specific columns, use *select()* first. The function must operate on `Tuple[str, np.ndarray]`.

Parameters

- **fn** (*Callable[..., Any]*) – The function to map.
- ***args** (*Any*) – Additional positional arguments to `fn`.
- ****kwargs** (*Any*) – Additional keyword arguments to `fn`.

Returns
iter_tf – An iterator to map the function.

Return type
Iterator[Any]

key_map(*fn: Callable[[...], Any]*, **args: Any*, ***kwargs: Any*) → Iterator[Tuple[str, Any]]

Map a function over columns like :meth:col_map, but return `Tuple` of the key with the function result. To apply to specific columns, use *select()* first. The function must operate on `Tuple[str, np.ndarray]`.

Parameters

- **fn** (*Callable[..., Any]*) – The function to map.
- ***args** (*Any*) – Additional positional arguments to `fn`.
- ****kwargs** (*Any*) – Additional keyword arguments to `fn`.

Returns
iter_tf – An iterator to map the function.

Return type
Iterator[Any]

pipe(*fn: Callable[[*Tafra*, P], *Tafra*], **args: Any*, ***kwargs: Any*) → *Tafra*

Apply a function to the *Tafra* and return the resulting *Tafra*. Primarily used to build a tranformer pipeline.

Parameters

- **fn** (*Callable[[], 'Tafra']*) – The function to apply.
- ***args** (*Any*) – Additional positional arguments to `fn`.
- ****kwargs** (*Any*) – Additional keyword arguments to `fn`.

Returns
tafra – A new *Tafra* result of the function.

Return type
Tafra

__rshift__(*other: Callable[[*Tafra*], *Tafra*]*) → *Tafra*

select(*columns: Iterable[str]*) → *Tafra*

Use column names to slice the *Tafra* columns analogous to SQL SELECT. This does not copy the data. Call *copy()* to obtain a copy of the sliced data.

Parameters
columns (*Iterable[str]*) – The column names to slice from the *Tafra*.

Returns
tafra – the *Tafra* with the sliced columns.

Return type
Tafra

copy(*order: str = 'C'*) → *Tafra*

Create a copy of a *Tafra*.

Parameters
order (*str = 'C' {'C', 'F', 'A', 'K'}*) – Controls the memory layout of the copy. 'C' means C-order, 'F' means F-order, 'A' means 'F' if a is Fortran contiguous, 'C' otherwise. 'K' means match the layout of a as closely as possible.

Returns
tafra – A copied *Tafra*.

Return type
Tafra

update(*other:* Tafra) → *Tafra*

Update the data and dtypes of this *Tafra* with another *Tafra*. Length of rows must match, while data of different `dtype` will overwrite.

Parameters
other (Tafra) – The other *Tafra* from which to update.

Returns
None

Return type
None

update_inplace(*other:* Tafra) → None

Inplace version.

Update the data and dtypes of this *Tafra* with another *Tafra*. Length of rows must match, while data of different `dtype` will overwrite.

Parameters
other (Tafra) – The other *Tafra* from which to update.

Returns
None

Return type
None

update_dtypes(*dtypes: Dict[str, Any]*) → *Tafra*

Apply new dtypes.

Parameters
dtypes (*Dict[str, Any]*) – The dtypes to update. If `None`, create from entries in *data*.

Returns
tafra – The updated *Tafra*.

Return type
Optional[*Tafra*]

update_dtypes_inplace(*dtypes: Dict[str, Any]*) → None

Inplace version.

Apply new dtypes.

Parameters
dtypes (*Dict[str, Any]*) – The dtypes to update. If `None`, create from entries in *data*.

Returns
tafra – The updated *Tafra*.

Return type
Optional[*Tafra*]

parse_object_dtypes() → *Tafra*

Parse the object dtypes using the `ObjectFormatter` instance.

parse_object_dtypes_inplace() → None

Inplace version.

Parse the object dtypes using the `ObjectFormatter` instance.

rename(*renames: Dict[str, str]*) → *Tafra*

Rename columns in the *Tafra* from a `dict`.

Parameters
renames (*Dict[str, str]*) – The map from current names to new names.

Returns
tafra – The *Tafra* with update names.

Return type
Optional[*Tafra*]

rename_inplace(*renames: Dict[str, str]*) → None

In-place version.

Rename columns in the *Tafra* from a `dict`.

Parameters
renames (*Dict[str, str]*) – The map from current names to new names.

Returns
tafra – The *Tafra* with update names.

Return type
Optional[*Tafra*]

coalesce(*column: str*, *fills: Iterable[Iterable[Union[None, str, int, float, bool, ndarray]]]*) → ndarray

Fill `None` values from `fills`. Analogous to `SQL COALESCE` or `pandas.fillna()`.

Parameters

- **column** (*str*) – The column to coalesce.
- **fills** (*Iterable[Union[str, int, float, bool, np.ndarray]:*) –

Returns
data – The coalesced data.

Return type
np.ndarray

coalesce_inplace(*column: str*, *fills: Iterable[Iterable[Union[None, str, int, float, bool, ndarray]]]*) → None

In-place version.

Fill `None` values from `fills`. Analogous to `SQL COALESCE` or `pandas.fillna()`.

Parameters

- **column** (*str*) – The column to coalesce.
- **fills** (*Iterable[Union[str, int, float, bool, np.ndarray]:*) –

Returns
data – The coalesced data.

Return type
np.ndarray

`_coalesce_dtypes`() → None

Update *dtypes* with missing keys that exist in *data*.

Must be called if :attr:`data` or :attr:`data` is directly modified!

Returns
None

Return type
None

`delete`(*columns: Iterable[str]*) → *Tafra*

Remove a column from *data* and *dtypes*.

Parameters
column (*str*) – The column to remove.

Returns
tafra – The *Tafra* with the deleted column.

Return type
Optional[*Tafra*]

`delete_inplace`(*columns: Iterable[str]*) → None

In-place version.

Remove a column from *data* and *dtypes*.

Parameters
column (*str*) – The column to remove.

Returns
tafra – The *Tafra* with the deleted column.

Return type
Optional[*Tafra*]

`pprint`(*indent: int = 1*, *width: int = 80*, *depth: Optional[int] = None*, *compact: bool = False*) → None

Pretty print. Parameters are passed to `pprint.PrettyPrinter`.

Parameters

- **indent** (*int*) – Number of spaces to indent for each level of nesting.
- **width** (*int*) – Attempted maximum number of columns in the output.
- **depth** (*Optional[int]*) – The maximum depth to print out nested structures.
- **compact** (*bool*) – If true, several items will be combined in one line.

Returns
None

Return type
None

`pformat`(*indent: int = 1*, *width: int = 80*, *depth: Optional[int] = None*, *compact: bool = False*) → str

Format for pretty printing. Parameters are passed to `pprint.PrettyPrinter`.

Parameters

- **indent** (*int*) – Number of spaces to indent for each level of nesting.
- **width** (*int*) – Attempted maximum number of columns in the output.

- **depth** (*Optional[int]*) – The maximum depth to print out nested structures.
- **compact** (*bool*) – If true, several items will be combined in one line.

Returns
formatted string – A formatted string for pretty printing.

Return type
str

to_html(*n: int = 20*) → str

Construct an HTML table representation of the *Tafra* data.

Parameters
n (*int = 20*) – Number of items to print.

Returns
HTML – The HTML table representation.

Return type
str

_slice(*_slice: slice*) → *Tafra*

Use a `slice` to slice the *Tafra*.

Parameters
_slice (*slice*) – The `slice` object.

Returns
tafra – The sliced *Tafra*.

Return type
Tafra

_iindex(*index: int*) → *Tafra*

Use a :class`int` to slice the *Tafra*.

Parameters
index (*int*) –

Returns
tafra – The sliced *Tafra*.

Return type
Tafra

_aindex(*index: Sequence[Union[int, bool]]*) → *Tafra*

Use numpy advanced indexing to slice the *Tafra*.

Parameters
index (*Sequence[Union[int, bool]]*) –

Returns
tafra – The sliced *Tafra*.

Return type
Tafra

_ndindex(*index: ndarray*) → *Tafra*

Use `numpy.ndarray` indexing to slice the *Tafra*.

Parameters
index (*np.ndarray*) –

Returns
tafra – The sliced *Tafra*.

Return type
Tafra

Helper Methods

class tafra.base.**Tafra**

union(*other:* Tafra) → *Tafra*

Helper function to implement *tafra.group.Union.apply()*.

Union two *Tafra* together. Analogy to SQL UNION or *pandas.append*. All column names and dtypes must match.

Parameters
other (*Tafra*) – The other tafra to union.

Returns
tafra – A new tafra with the unioned data.

Return type
Tafra

union_inplace(*other:* Tafra) → None

Inplace version.

Helper function to implement *tafra.group.Union.apply_inplace()*.

Union two *Tafra* together. Analogy to SQL UNION or *pandas.append*. All column names and dtypes must match.

Parameters
other (*Tafra*) – The other tafra to union.

Returns
None

Return type
None

group_by(*columns: Iterable[str], aggregation: Mapping[str, Union[Callable[[ndarray], Any], Tuple[Callable[[ndarray], Any], str]]] = {}, iter_fn: Mapping[str, Callable[[ndarray], Any]] = {}*) → *Tafra*

Helper function to implement *tafra.group.GroupBy.apply()*.

Aggregation by a set of unique values.

Analogy to SQL `GROUP BY`, not `pandas.DataFrame.groupby()`.

Parameters

- **columns** (*Iterable[str]*) – The column names to group by.
- **aggregation** (*Mapping[str, Union[Callable[[np.ndarray], Any], Tuple[Callable[[np.ndarray], Any], str]]]*) – Optional. A mapping for columns and aggregation functions. Should be given as {'column': fn} or {'new_column': (fn, 'column')}.

- **iter_fn** (*Mapping[str, Callable[[np.ndarray], Any]]*) – Optional. A mapping for new columns names to the function to apply to the enumeration. Should be given as {'new_column': fn}.

Returns
tafra – The aggregated *Tafra*.

Return type
Tafra

transform(*columns: Iterable[str]*, *aggregation: Mapping[str, Union[Callable[[ndarray], Any], Tuple[Callable[[ndarray], Any], str]]] = {}*, *iter_fn: Dict[str, Callable[[ndarray], Any]] = {}*) → *Tafra*

Helper function to implement *tafra.group.Transform.apply()*.

Apply a function to each unique set of values and join to the original table. Analogy to `pandas.DataFrame.groupby().transform()`, i.e. a SQL `GROUP BY` and `LEFT JOIN` back to the original table.

Parameters

- **group_by** (*Iterable[str]*) – The column names to group by.
- **aggregation** (*Mapping[str, Union[Callable[[np.ndarray], Any], Tuple[Callable[[np.ndarray], Any], str]]]*) – Optional. A mapping for columns and aggregation functions. Should be given as {'column': fn} or {'new_column': (fn, 'column')}.
- **iter_fn** (*Mapping[str, Callable[[np.ndarray], Any]]*) – Optional. A mapping for new columns names to the function to apply to the enumeration. Should be given as {'new_column': fn}.

Returns
tafra – The transformed *Tafra*.

Return type
Tafra

iterate_by(*columns: Iterable[str]*) → Iterator[Tuple[Tuple[Any, ...], ndarray, *Tafra*]]

Helper function to implement *tafra.group.IterateBy.apply()*.

A generator that yields a *Tafra* for each set of unique values. Analogy to *pandas.DataFrame.groupby()*, i.e. an `Iterator` of *Tafra*.

Yields tuples of ((unique grouping values, ...), row indices array, subset tafra)

Parameters
group_by (*Iterable[str]*) – The column names to group by.

Returns
tafras – An iterator over the grouped *Tafra*.

Return type
Iterator[GroupDescription]

inner_join(*right:* Tafra, *on: Iterable[Tuple[str, str, str]]*, *select: Iterable[str] = []*) → *Tafra*

Helper function to implement *tafra.group.InnerJoin.apply()*.

An inner join.

Analogy to SQL INNER JOIN, or *pandas.merge(..., how='inner')*,

Parameters

- **right** (Tafra) – The right-side *Tafra* to join.

- **on** (*Iterable[Tuple[str, str, str]]*) – The columns and operator to join on. Should be given as ('left column', 'right column', 'op') Valid ops are:

  '==' : equal to '!=' : not equal to '<' : less than '<=' : less than or equal to '>' : greater than '>=' : greater than or equal to

- **select** (*Iterable[str] = []*) – The columns to return. If not given, all unique columns names are returned. If the column exists in both :class`Tafra`, prefers the left over the right.

Returns
tafra – The joined *Tafra*.

Return type
Tafra

left_join(*right:* Tafra, *on: Iterable[Tuple[str, str, str]]*, *select: Iterable[str] = []*) → *Tafra*

Helper function to implement *tafra.group.LeftJoin.apply()*.

A left join.

Analogy to SQL LEFT JOIN, or *pandas.merge(..., how='left')*,

Parameters

- **right** (Tafra) – The right-side *Tafra* to join.
- **on** (*Iterable[Tuple[str, str, str]]*) – The columns and operator to join on. Should be given as ('left column', 'right column', 'op') Valid ops are:

  '==' : equal to '!=' : not equal to '<' : less than '<=' : less than or equal to '>' : greater than '>=' : greater than or equal to

- **select** (*Iterable[str] = []*) – The columns to return. If not given, all unique columns names are returned. If the column exists in both :class`Tafra`, prefers the left over the right.

Returns
tafra – The joined *Tafra*.

Return type
Tafra

cross_join(*right:* Tafra, *select: Iterable[str] = []*) → *Tafra*

Helper function to implement *tafra.group.CrossJoin.apply()*.

A cross join.

Analogy to SQL CROSS JOIN, or *pandas.merge(..., how='outer') using temporary columns of static value to intersect all rows*.

Parameters

- **right** (Tafra) – The right-side *Tafra* to join.
- **select** (*Iterable[str] = []*) – The columns to return. If not given, all unique columns names are returned. If the column exists in both :class`Tafra`, prefers the left over the right.

Returns
tafra – The joined *Tafra*.

Return type
Tafra

Aggregations

class tafra.group.**Union**

Union two `Tafra` together. Analogy to SQL UNION or *pandas.append*. All column names and dtypes must match.

apply(*left:* Tafra, *right:* Tafra) → *Tafra*

Apply the `Union_` to the `Tafra`.

Parameters

- **left** (`Tafra`) – The left `Tafra` to union.
- **right** (`Tafra`) – The right `Tafra` to union.

Returns
tafra – The unioned :class`Tafra`.

Return type
Tafra

apply_inplace(*left:* Tafra, *right:* Tafra) → None

In-place version.

Apply the `Union_` to the `Tafra`.

Parameters

- **left** (`Tafra`) – The left `Tafra` to union.
- **right** (`Tafra`) – The right `Tafra` to union.

Returns
tafra – The unioned :class`Tafra`.

Return type
Tafra

class tafra.group.**GroupBy**(*group_by_cols: Iterable[str]*, *aggregation: InitVar*, *iter_fn: Mapping[str, Callable[[ndarray], Any]]*)

Aggregation by a set of unique values.

Analogy to SQL `GROUP BY`, not `pandas.DataFrame.groupby()`.

Parameters

- **columns** (*Iterable[str]*) – The column names to group by.
- **aggregation** (*Mapping[str, Union[Callable[[np.ndarray], Any], Optional. Tuple[Callable[[np.ndarray], Any], str]]]*) – A mapping for columns and aggregation functions. Should be given as {'column': fn} or {'new_column': (fn, 'column')}.
- **iter_fn** (*Mapping[str, Callable[[np.ndarray], Any]]*) – Optional. A mapping for new columns names to the function to apply to the enumeration. Should be given as {'new_column': fn}.

apply(*tafra:* Tafra) → *Tafra*

Apply the *GroupBy* to the `Tafra`.

Parameters
tafra (`Tafra`) – The tafra to apply the operation to.

Returns
tafra – The aggregated `Tafra`.

Return type
Tafra

class tafra.group.**Transform**(*group_by_cols: Iterable[str]*, *aggregation: InitVar*, *iter_fn: Mapping[str, Callable[[ndarray], Any]]*)

Apply a function to each unique set of values and join to the original table.

Analogy to `pandas.DataFrame.groupby().transform()`, i.e. a SQL `GROUP BY` and `LEFT JOIN` back to the original table.

Parameters

- **group_by** (*Iterable[str]*) – The column names to group by.
- **aggregation** (*Mapping[str, Union[Callable[[np.ndarray], Any], Tuple[Callable[[np.ndarray], Any], str]]]*) – Optional. A mapping for columns and aggregation functions. Should be given as {'column': fn} or {'new_column': (fn, 'column')}.
- **iter_fn** (*Mapping[str, Callable[[np.ndarray], Any]]*) – Optional. A mapping for new columns names to the function to apply to the enumeration. Should be given as {'new_column': fn}.

apply(*tafra:* Tafra) → *Tafra*

Apply the *Transform* to the `Tafra`.

Parameters
tafra (Tafra) – The tafra to apply the operation to.

Returns
tafra – The transformed `Tafra`.

Return type
Tafra

class tafra.group.**IterateBy**(*group_by_cols: Iterable[str]*)

A generator that yields a `Tafra` for each set of unique values.

Analogy to *pandas.DataFrame.groupby()*, i.e. an Sequence of *Tafra* objects. Yields tuples of ((unique grouping values, ...), row indices array, subset tafra)

Parameters
group_by (*Iterable[str]*) – The column names to group by.

apply(*tafra:* Tafra) → Iterator[Tuple[Tuple[Any, ...], ndarray, *Tafra*]]

Apply the *IterateBy* to the `Tafra`.

Parameters
tafra (Tafra) – The tafra to apply the operation to.

Returns
tafras – An iterator over the grouped `Tafra`.

Return type
Iterator[GroupDescription]

class tafra.group.**InnerJoin**(*on: Iterable[Tuple[str, str, str]]*, *select: Iterable[str]*)

An inner join.

Analogy to SQL INNER JOIN, or *pandas.merge(..., how='inner')*,

Parameters

- **right** (Tafra) – The right-side Tafra to join.
- **on** (*Iterable[Tuple[str, str, str]]*) – The columns and operator to join on. Should be given as ('left column', 'right column', 'op') Valid ops are:

  '==' : equal to '!=' : not equal to '<' : less than '<=' : less than or equal to '>' : greater than '>=' : greater than or equal to
- **select** (*Iterable[str] = []*) – The columns to return. If not given, all unique columns names are returned. If the column exists in both :class`Tafra`, prefers the left over the right.

apply(*left_t:* Tafra, *right_t:* Tafra) → *Tafra*

Apply the *InnerJoin* to the Tafra.

Parameters

- **left_t** (Tafra) – The left tafra to join.
- **right_t** (Tafra) – The right tafra to join.

Returns
tafra – The joined Tafra.

Return type
Tafra

class tafra.group.**LeftJoin**(*on: Iterable[Tuple[str, str, str]]*, *select: Iterable[str]*)

A left join.

Analogy to SQL LEFT JOIN, or *pandas.merge(..., how='left')*,

Parameters

- **right** (Tafra) – The right-side Tafra to join.
- **on** (*Iterable[Tuple[str, str, str]]*) – The columns and operator to join on. Should be given as ('left column', 'right column', 'op') Valid ops are:

  '==' : equal to '!=' : not equal to '<' : less than '<=' : less than or equal to '>' : greater than '>=' : greater than or equal to
- **select** (*Iterable[str] = []*) – The columns to return. If not given, all unique columns names are returned. If the column exists in both :class`Tafra`, prefers the left over the right.

apply(*left_t:* Tafra, *right_t:* Tafra) → *Tafra*

Apply the *LeftJoin* to the Tafra.

Parameters

- **left_t** (Tafra) – The left tafra to join.
- **right_t** (Tafra) – The right tafra to join.

Returns
tafra – The joined Tafra.

Return type
Tafra

class tafra.group.**CrossJoin**(*on: Iterable[Tuple[str, str, str]]*, *select: Iterable[str]*)

A cross join.

Analogy to SQL CROSS JOIN, or *pandas.merge(..., how='outer') using temporary columns of static value to intersect all rows*.

Parameters

- **right** (Tafra) – The right-side Tafra to join.
- **select** (*Iterable[str] = []*) – The columns to return. If not given, all unique columns names are returned. If the column exists in both :class`Tafra`, prefers the left over the right.

apply(*left_t:* Tafra, *right_t:* Tafra) → *Tafra*

Apply the *CrossJoin* to the Tafra.

Parameters

- **left_t** (Tafra) – The left tafra to join.
- **right_t** (Tafra) – The right tafra to join.

Returns
tafra – The joined Tafra.

Return type
Tafra

Object Formatter

class tafra.formatter.**ObjectFormatter**

A dictionary that contains mappings for formatting objects. Some numpy objects should be cast to other types, e.g. the decimal.Decimal type cannot operate with np.float. These mappings are defined in this class.

Each mapping must define a function that takes a np.ndarray and returns a np.ndarray.

The key for each mapping is the name of the type of the actual value, looked up from the first element of the np.ndarray, i.e. type(array[0]).__name__.

__getitem__(*dtype: str*) → Callable[[ndarray], ndarray]

Get the dtype formatter.

__setitem__(*dtype: str*, *value: Callable[[ndarray], ndarray]*) → None

Set the dtype formatter.

__delitem__(*dtype: str*) → None

Delete the dtype formatter.

3.2 Numerical Performance

3.2.1 Summary

One of the goals of tafra is to provide a fast-as-possible data structure for numerical computing. To achieve this, all function returns are written as generator expressions wherever possible.

Additionally, because the data contains values of ndarrays, the map functions may also take functions that operate on ndarrays. This means that they are able to take numba @jit'ed functions as well.

pandas is essentially a standard package for anyone performing data science with Python, and it provides a wide variety of useful features. However, it's not particularly aimed at maximizing performance. Let's use an example of a dataframe of function arguments, and a function that maps scalar arguments into a vector result. Any function of time serves this purpose, so let's use a hyperbolic function.

First, let's randomnly generate some function arguments and construct both a Tafra and a pandas.DataFrame:

```
>>> from tafra import Tafra
>>> import pandas as pd
>>> import numpy as np

>>> from typing import Tuple, Union, Any

>>> tf = Tafra({
...     'wellid': np.arange(0, 100),
...     'qi': np.random.lognormal(np.log(2000.), np.log(3000. / 1000.) / (2 * 1.28),␣
↪100),
...     'Di': np.random.uniform(.5, .9, 100),
...     'bi': np.random.normal(1.0, .2, 100)
... })

>>> df = pd.DataFrame(tf.data)

>>> tf.head(5)
```

index	wellid	qi	Di	bi
dtype	int32	float64	float64	float64
0	0	2665.82	0.54095	1.07538
1	1	1245.85	0.81711	0.48448
2	2	1306.56	0.61570	0.54587
3	3	2950.33	0.81956	0.66440
4	4	1963.93	0.56918	0.74165

Next, we define our hyperbolic function and the time array to evaluate:

```
>>> import math

>>> def tan_to_nominal(D: float) -> float:
...     return -math.log1p(-D)

>>> def sec_to_nominal(D: float, b: float) -> float:
...     if b <= 1e-4:
...         return tan_to_nominal(Di)
...
...     return ((1.0 - D) ** -b - 1.0) / b

>>> def hyp(qi: float, Di: float, bi: float, t: np.ndarray) -> np.ndarray:
...     Dn = sec_to_nominal(Di, bi)
...
...     if bi <= 1e-4:
...         return qi * np.exp(-Dn * t)
...
...    return qi / (1.0 + Dn * bi * t) ** (1.0 / bi)

>>> t = 10 ** np.linspace(0, 4, 101)
```

And let's build a generic `mapper` function to map over the named columns:

```
>>> def mapper(tf: Union[Tafra, pd.DataFrame]) -> Tuple[int, np.ndarray]:
...     return tf['wellid'], hyp(tf['qi'], tf['Di'], tf['bi'], t)
```

We can call this with the following style. The `pandas` syntax is a bit verbose, but `pandas.DataFrame.from_items()` is deprecated in newer versions, so this is the recommended way. Let's time each approach:

```
>>> %timeit tdcs = Tafra(tf.row_map(mapper))
3.38 ms ± 129 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)


>>> %timeit pdcs = pd.DataFrame(dict(df.apply(mapper, axis=1).to_list()))
6.86 ms ± 408 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

We see `Tafra` is about twice as fast. Mapping a function this way is convenient, but there is some indirection occuring that we can do away with to obtain direct access to the data of the `Tafra`, and there is a faster method for `pandas` as well as opposed to `pandas.DataFrame.apply()`. Instead of constructing a new `Tafra` or `pd.DataFrame` for each row, we can instead return a `NamedTuple`, which is faster to construct. Doing so:

```
>>> def tuple_mapper(tf: Tuple[Any, ...]) -> Tuple[int, np.ndarray]:
...     return tf.wellid, hyp(tf.qi, tf.Di, tf.bi, t)

>>> %timeit Tafra(tf.tuple_map(tuple_mapper))
1.68 ms ± 84.3 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)

>>> %timeit pd.DataFrame(dict((tuple_mapper(row)) for row in df.itertuples()))
3.14 ms ± 121 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

And once again, `Tafra` is about twice as fast.

One of the upcoming features of `pandas` is the ability to apply `numba` `@jit`'ed functions to `pandas.DataFrame.apply()`. The performance improvement should be significant, especially for long-running functions, but there will still be overhead in the abstraction before the function is called. We can demonstrate this by `@jit`'ing our hyperbolic function and mapping it over the dataframes, and get an idea of how much improvement is possible:

```
>>> from numba import jit
>>> jit_kw = {'fastmath': True}

>>> @jit(**jit_kw)
...  def tan_to_nominal(D: float) -> float:
...     return -math.log1p(-D)

>>> @jit(**jit_kw)
... def sec_to_nominal(D: float, b: float) -> float:
...     if b <= 1e-4:
...         return tan_to_nominal(D)
...
...     return ((1.0 - D) ** -b - 1.0) / b

>>> @jit(**jit_kw)
... def hyp(qi: float, Di: float, bi: float, t: np.ndarray) -> np.ndarray:
...     Dn = sec_to_nominal(Di, bi)
...
...     if bi <= 1e-4:
...         return qi * np.exp(-Dn * t)
```

(continues on next page)

(continued from previous page)

```
...
...     return qi / (1.0 + Dn * bi * t) ** (1.0 / bi)

>>> %timeit Tafra(tf.tuple_map(tuple_mapper))
884 µs ± 41.9 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)

>>> %timeit pd.DataFrame(dict((tuple_mapper(row)) for row in df.itertuples()))
3.09 ms ± 115 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Interestingly, we see that `pandas` does not get much benefit from this, as the limit occurs not in the performance of the functions but in the performance of `pandas` itself. We can validate this by skipping the dataframe construction step:

```
>>> %timeit [tf.tuple_map(tuple_mapper)]
81.9 µs ± 2.91 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)

>>> %timeit [(tuple_mapper(row)) for row in df.itertuples()]
614 µs ± 14.8 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Last, we might as the question "If `pandas` is incurring some performance penalty, what is the performance penalty of `Tafra`?" We'll write a function that operates on the `numpy.ndarray`s themselves rather than using the helper `Tafra.tuple_map()`. We can also use `numpy`'s built in apply function, `numpy.apply_along_axis()`, but it is considerably slower than a `@jit`'ed function.

```
>>> @jit(**jit_kw)
... def ndarray_map(qi: np.ndarray, Di: np.ndarray, bi: np.ndarray, t: np.ndarray) -> np.
↪ndarray:
...     out = np.zeros((qi.shape[0], t.shape[0]))
...     for i in range(qi.shape[0]):
...         out[i, :] = hyp(qi[i], Di[i], bi[i], t)
...
...     return out
81.2 µs ± 9.7 µs per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

And the timing is neglible, meaning `Tafra`'s `Tafra.tuple_map()` is essentially as performant as we are able to achieve in Python.

3.3 Testing

Testing is set to evaluate:

- style with flake8,
- typing with mypy,
- valid function return values and behaviors with hypothesis, and
- test coverage using coverage.

3.3.1 Windows

Run `test.bat` in the `test` directory.

3.3.2 Linux

Run `test.sh` in the `test` directory.

3.4 Version History

Tafra: a minimalist dataframe

3.4.1 Author

Derrick W. Turk David S. Fulford

Notes

Created on April 25, 2020

3.4.2 1.0.10

- Add `pipe` and overload >> operator for Tafra objects

3.4.3 1.0.9

- Add test files to build

3.4.4 1.0.8

- Check rows in constructor to ensure equal data length

3.4.5 1.0.7

- Handle missing or NULL values in `read_csv()`.
- Cast empty elements to None when updating dtypes to avoid failure of `np.astype()`.
- Update some typing, minor refactoring for performance

3.4.6 1.0.6

- Additional validations in constructor, primary to evaluate Iterables of values
- Split `col_map` to `col_map` and `key_map` as the original function's return signature depending upon an argument.
- Fix some documentation typos

3.4.7 1.0.5

- Add `tuple_map` method
- Refactor all iterators and `..._map` functions to improve performance
- Unpack `np.ndarray` if given as keys to constructor
- Add `validate=False` in `__post_init__` if inputs are **known** to be valid to improve performance

3.4.8 1.0.4

- Add `read_csv`, `to_csv`
- Various refactoring and improvement in data validation
- Add `typing_extensions` to dependencies
- Change method of `dtype` storage, extract `str` representation from `np.dtype()`

3.4.9 1.0.3

- Add `read_sql` and `read_sql_chunks`
- Add `to_tuple` and `to_pandas`
- Cleanup constructor data validation

3.4.10 1.0.2

- Add object_formatter to expose user formatting for dtype=object
- Improvements to indexing and slicing

3.4.11 1.0.1

- Add iter functions
- Add map functions
- Various constructor improvements

3.4.12 1.0.0

- Initial Release

3.5 Index

INDEX

Symbols

A

C

D

F

G

H

I

K

L

N

O

P