typeddfs

Release 0.17.0

Douglas Myers-Turnbull

Jul 27, 2022

CONTENTS

CHAPTER

ONE

GUIDE

We're going to wrangle and analyze data input from a bird-watching group.

Let's just read a CSV. It looks like this:

```
species,person,date,notes
Blue Jay,Kerri Johnson,2021-05-14,perched in a tree
```

We'd like to declare what this should look like.

```
import typeddfs as tdf

Sightings = (
    tdf.typed("Sightings")
    .require("species", "person", "date")
    .reserve("notes")
    .strict()
    .build()
)
```

Let's try reading a malformed CSV that is missing the "date" column.

```
Sightings.read_csv("missing_col.csv")
```

This will raise a `typeddfs.errors.MissingColumnError`.

Much more to come…

1.1 Serialization

1.2 Typing rules

1.3 Construction and customization

1.4 New functions

Natural sorting.

1.5 Matrix types

1.6 Imperative declaration

1.7 Data types and freezing

1.8 Checksums and caching

1.9 Advanced serialization

1.10 Generating CLI-style help

1.11 Utilities

CHAPTER

TWO

MISC EXAMPLES

2.1 Simple example

```
from typeddfs import TypedDfs

MyDfType = (
    TypedDfs.typed("MyDfType")
    .require("name", index=True)  # always keep in index
    .require("value", dtype=float)  # require a column and type
    .drop("_temp")  # auto-drop a column
    .verify(lambda ddf: len(ddf) == 12)  # require exactly 12 rows
).build()

df = MyDfType.read_file(input("filename? [.feather/.csv.gz/.tsv.xz/etc.]"))
df = df.sort_natural()
df.write_file("myfile.feather", mkdirs=True)
# want to write to a sha1sum-like (.sha256) file?
df.write_file("myfile.feather", file_hash=True)
# verify it?
MyDfType.read_file("myfile.feather", check_hash="file")
```

2.2 A matrix-style DataFrame

```
import numpy as np
from typeddfs import TypedDfs

Symmetric64 = (
    TypedDfs.matrix("Symmetric64", doc="A symmetric float64 matrix")
    .dtype(np.float64)
    .verify(lambda df: df.values.sum().sum() == 1.0)
    .add_methods(product=lambda df: df.flatten().product())
).build()

mx = Symmetric64.read_file("input.tab")
print(mx.product())  # defined above
if mx.is_symmetric():
    mx = mx.triangle()  # it's symmetric, so we only need half
```

(continues on next page)

(continued from previous page)

```
    long = mx.drop_na().long_form()  # columns: "row", 'column", and "value"
    long.write_file("long-form.xml")
```

2.3 Example in terms of CSV

For a CSV like this:

```
key,value,note
abc,123,?
```

```
from typeddfs import TypedDfs

# Build me a Key-Value-Note class!
KeyValue = (
    TypedDfs.typed("KeyValue")  # With enforced reqs / typing
    .require("key", dtype=str, index=True)  # automagically add to index
    .require("value")  # required
    .reserve("note")  # permitted but not required
    .strict()  # disallow other columns
).build()

# This will self-organize and use "key" as the index:
df = KeyValue.read_csv("example.csv")

# For fun, let"s write it and read it back:
df.to_csv("remake.csv")
df = KeyValue.read_csv("remake.csv")
print(df.index_names(), df.column_names())  # ["key"], ["value", "note"]

# And now, we can type a function to require a KeyValue,
# and let it raise an `InvalidDfError` (here, a `MissingColumnError`):
def my_special_function(df: KeyValue) -> float:
    return KeyValue(df)["value"].sum()
```

CHAPTER

THREE

Q & A

3.1 What are the different types of typed DataFrames?

You should generally use two: *typeddfs.typed_dfs.TypedDf* and *typeddfs.matrix_dfs.MatrixDf*. There is also a specialized matrix type, *typeddfs.matrix_dfs.AffinityMatrixDf*. You can construct these easily with *typeddfs._entries.TypedDfs.typed()*, *typeddfs._entries.TypedDfs.matrix()*, and *typeddfs._entries.TypedDfs.affinity_matrix()*. There is a final type, defined to have no typing rules, that can be constructed with *typeddfs._entries.TypedDfs.untyped()*. You can convert a vanilla Pandas DataFrame to an "untyped" variant via *typeddfs._entries.TypedDfs.wrap()* to give it the additional methods.

```
from typeddfs import TypedDfs

MyDf = TypedDfs.typed("MyDf").build()
```

3.2 What is the hierarchy of DataFrames?

It's confusing. In general, you won't need to know the difference.

typeddfs.typed_dfs.TypedDf and *typeddfs.matrix_dfs.MatrixDf* inherit from *typeddfs.base_dfs.BaseDf*, which inherits from *typeddfs.abs_dfs.AbsDf*, which inherits from *typeddfs._core_dfs.CoreDf*. (Technically, `CoreDf` inherits from *typeddfs._pretty_dfs.PrettyDf*.) The difference is:

- *typeddfs.base_dfs.BaseDf* has methods `convert` and `of` (generally overridden).
- *typeddfs.abs_dfs.AbsDf* contains `typeddfs.abs_dfs.AbsDf.get_typing()`, overrides IO methods from DataFrame, and adds *typeddfs.abs_dfs.AbsDf.read_file()* and *typeddfs.abs_dfs.AbsDf.write_file()*.
- *typeddfs._core_dfs.CoreDf* wraps DataFrame methods to retain the same type for returned DataFrames and adds a few extra methods.

3.3 What is the difference between `__init__`, `convert`, and `of`?

These three methods in *typeddfs.typed_dfs.TypedDf* (and its superclasses) are a bit different. `typeddfs.typed_dfs.TypedDf.__init__()` does NOT attempt to reorganize or validate your DataFrame, while *typeddfs.typed_dfs.TypedDf.convert()* and `typeddfs.typed_dfs.TypedDf.of()` do.``of`` is simply more flexible than `convert`: `convert` only accepts a DataFrame, while `of` will take anything that `DataFrame.__init__` will.

3.4 When do typed DFs "detype" during chained invocations?

Most DataFrame-level functions that ordinarily return DataFrames themselves try to keep the same type. This includes `typeddfs.abs_dfs.AbsDf.reindex()`, `typeddfs.abs_dfs.AbsDf.drop_duplicates()`, `typeddfs.abs_dfs.AbsDf.sort_values()`, and `typeddfs.abs_dfs.AbsDf.set_index()`. This is to allow for easy chained invocation, but it's important to note that the returned DataFrame might not conform to your requirements. Call `typeddfs.abs_dfs.AbsDf.retype()` at the end to reorganize and verify.

```
from typeddfs import TypedDfs

MyDf = TypedDfs.typed("MyDf").require("valid").build()
my_df = MyDf.read_csv("x.csv")
my_df_2 = my_df.drop_duplicates().rename_cols(valid="ok")
print(type(my_df_2))  # type(MyDf)
# but this fails!
my_df_3 = my_df.drop_duplicates().rename_cols(valid="ok").retype()
# MissingColumnError "valid"
```

You can call `typeddfs.abs_dfs.AbsDf.dtype()` to remove any typing rules and `typeddfs.abs_dfs.AbsDf.vanilla()` if you need a plain DataFrame, though this should rarely be needed.

3.5 How does one get the typing info?

Call `typeddfs.base_dfs.BaseDf.get_typing()`

```
from typeddfs import TypedDfs

MyDf = TypedDfs.typed("MyDf").require("valid").build()
MyDf.get_typing().required_columns  # ["valid"]
```

3.6 How are toml documents read and written?

These are limited to a single array of tables (AOT). The AOT is named `row` by default (set with `aot=`). On read, you can pass `aot=None` to have it use the unique outermost key. `

3.7 How are INI files read and written?

These require exactly 2 columns after `reset_index()`. Parsing is purposefully minimal because these formats are flexible. Trailing whitespace and whitespace surrounding = is ignored. Values are not escaped, and keys may not contain =. Line continuation with \ is not allowed. Quotation marks surrounding values are not dropped, unless `drop_quotes=True` is passed. Comments begin with `;`, along with # if `hash_sign=True` is passed.

On read, section names are prepended to the keys. For example, the key name will be `section.key` in this example:

```
[section]
key = value
```

On write, the inverse happens.

3.8 What about .properties?

These are similar to INI files. Only hash signs are allowed for comments, and reserved chars *are* escaped in keys. This includes \\,``, ``\=, and `\:` These are not escaped in values.

3.9 What is "flex-width format"?

This is a format that shows up a lot in the wild, but doesn't seem to have a name. It's just a text format like TSV or CSV, but where columns are preferred to line up in a fixed-width font. Whitespace is ignored on read, but on write the columns are made to line up neatly. These files are easy to view. By default, the delimiter is three vertical bars (`|||`).

3.10 When are read and write guaranteed to be inverses?

In principle, this invariant holds when you call `.strict()` to disallow additional columns and specify `dtype=` in all calls to `.require` and `.reserve`. In practice, this might break down for certain combinations of DataFrame structure, dtypes, and serialization format. It seems pretty solid for Feather, Parquet, and CSV/TSV-like variants, especially if the dtypes are limited to bools, real values, int values, and strings. There may be corner cases for XML, TOML, INI, Excel, OpenDocument, and HDF5, as well as for categorical and miscellaneous `object` dtypes.

3.11 How do I include another filename suffix?

Use `.suffix()` to register a suffix or remap it to another format.

```
from typeddfs import TypedDfs, FileFormat

MyDf = TypedDfs.typed("MyDf").suffix(tabbed="tsv").build()
# or:
MyDf = TypedDfs.typed("MyDf").suffix(**{".tabbed": FileFormat.tsv}).build()
```

3.12 How do the checksums work?

There are simple convenience flags to write sha1sum-like files while writing files, and to verify them when reading.

```
from pathlib import Path
from typeddfs import TypedDfs

MyDf = TypedDfs.typed("MyDf").build()
df = MyDf()
df.write_file("here.csv", file_hash=True)
# a hex-encoded hash and filename
Path("here.csv.sha256").read_text(encoding="utf8")
MyDf.read_file("here.csv", file_hash=True)  # verifies that it matches
```

You can change the hash algorithm with `.hash()`. The second variant is `dir_hash`.

```
from pathlib import Path
from typeddfs import TypedDfs, Checksums

MyDf = TypedDfs.typed("MyDf").build()
df = MyDf()
path = Path("dir", "here.csv")
df.write_file(path, dir_hash=True, mkdirs=True)
# potentially many hex-encoded hashes and filenames; always appended to
MyDf.read_file(path, dir_hash=True)  # verifies that it matches
# read it
sums = Checksums.parse_hash_file_resolved(Path("my_dir", "my_dir.sha256"))
```

CHAPTER

FOUR

NUTS AND BOLTS

CHAPTER

FIVE

API REFERENCE

This page contains auto-generated API reference documentation[1].

5.1 typeddfs

Metadata and top-level declarations for typeddfs.

5.1.1 Subpackages

typeddfs._mixins

Submodules

typeddfs._mixins._csv_like_mixin

Mixin for CSV and TSV.

Module Contents

class typeddfs._mixins._csv_like_mixin.**_CsvLikeMixin**

classmethod read_csv(**args*, ***kwargs*) → __qualname__

Reads from CSV, converting to this type. Using to_csv() and read_csv() from BaseFrame, this property holds:

```
df.to_csv(path)
df.__class__.read_csv(path) == df
```

Passing `index` on `to_csv` or `index_col` on `read_csv` explicitly will break this invariant.

Parameters

- **args** – Passed to `pd.read_csv`; should start with a path or buffer
- **kwargs** – Passed to `pd.read_csv`.

classmethod read_tsv(**args*, ***kwargs*) → __qualname__

Reads tab-separated data. See *read_csv()* for more info.

[1] Created with sphinx-autoapi

to_csv(**args*, ***kwargs*) → Optional[str]

to_tsv(**args*, ***kwargs*) → Optional[str]

Writes tab-separated data. See *to_csv()* for more info.

typeddfs._mixins._dataclass_mixin

Dataclass mixin.

Module Contents

class typeddfs._mixins._dataclass_mixin.**TypedDfDataclass**

Just a `dataclass` for TypedDfs. Contains *get_df_type()* to point to the original DataFrame.

get_as_dict() → Mapping[str, Any]

Returns a mapping from the dataclass field name to the value.

abstract classmethod get_df_type() → Type[*TypedDf*]

Returns the original DataFrame type.

classmethod get_fields() → Sequence[dataclasses.Field]

Returns the fields of this dataclass.

class typeddfs._mixins._dataclass_mixin.**_DataclassMixin**

classmethod _create_dataclass(*fields: Sequence[Tuple[str, Type[Any]]]*) → Type[*TypedDfDataclass*]

classmethod create_dataclass(*reserved: bool = True*) → Type[*TypedDfDataclass*]

Creates a best-effort immutable `dataclass` for this type. The fields will depend on the columns and index levels present in `get_typing()`. The type of each field will correspond to the specified dtype (*typeddfs.df_typing.DfTyping.auto_dtypes()*), falling back to `Any` if none is specified.

Note: If this type can support additional columns (*typeddfs.df_typing.DfTyping.is_strict()* is the default, `False`), the dataclass will not be able to support extra fields. For most cases, `typeddfs.abs_dfs.AbsDf.to_dataclass_instances()` is better.

Parameters **reserved** – Include reserved columns and index levels

Returns A subclass of `typeddfs.abs_dfs.TypedDfDataclass`

classmethod from_dataclass_instances(*instances: Sequence[*TypedDfDataclass*]*) → __qualname__

Creates a new instance of this DataFrame type from dataclass instances. This mostly delegates to `pd.DataFrame.__init__`, calling `cls.of(instances)`. It is provided for consistency with *to_dataclass_instances()*.

Parameters **instances** – A sequence of dataclass instances. Although typed as `typeddfs.abs_dfs.TypedDfDataclass`, any type created by Python's `dataclass` module should work.

Returns A new instance of this type

to_dataclass_instances() → Sequence[*TypedDfDataclass*]

Creates a dataclass from this DataFrame and returns instances. Also see *from_dataclass_instances()*.

Note: Dataclass elements are equal if fields and values match, even if they are of different types. This was done by overriding `__eq__` to enable comparing results from separate calls to this method. Specifically, `typeddfs.abs_dfs.TypedDfDataclass.get_as_dict()` must return True.

Caution: Fields cannot be included if columns are not present. If `self.get_typing().is_strict is False`, then the dataclass created by two different DataFrames of type `self.__class__` may have different fields.

Caution: A new dataclass is created per call, so `df.to_dataclass_instances()[0] is not df.to_dataclass_instances()[0]`.

typeddfs._mixins._excel_mixins

Mixin for Excel/ODF IO.

Module Contents

class typeddfs._mixins._excel_mixins.**_ExcelMixin**

classmethod read_excel(*io*, *sheet_name: _SheetNamesOrIndices = 0*, **args*, ***kwargs*) → __qualname__

classmethod read_ods(*io*, *sheet_name: _SheetNamesOrIndices = 0*, ***kwargs*) → __qualname__

Reads OpenDocument ODS/ODT files. Prefer this method over *read_excel()*.

classmethod read_xls(*io*, *sheet_name: _SheetNamesOrIndices = 0*, ***kwargs*) → __qualname__

Reads legacy XLS Excel files. Prefer this method over *read_excel()*.

classmethod read_xlsb(*io*, *sheet_name: _SheetNamesOrIndices = 0*, ***kwargs*) → __qualname__

Reads XLSB Excel files. This is a relatively uncommon format. Prefer this method over *read_excel()*.

classmethod read_xlsx(*io*, *sheet_name: _SheetNamesOrIndices = 0*, ***kwargs*) → __qualname__

Reads XLSX Excel files. Prefer this method over *read_excel()*.

to_excel(*excel_writer*, **args*, ***kwargs*) → Optional[str]

to_ods(*ods_writer*, **args*, ***kwargs*) → Optional[str]

Writes OpenDocument ODS/ODT files. Prefer this method over `write_excel()`.

to_xls(*excel_writer*, **args*, ***kwargs*) → Optional[str]

Reads legacy XLS Excel files. Prefer this method over `write_excel()`.

to_xlsb(*excel_writer*, **args*, ***kwargs*) → Optional[str]

Writes XLSB Excel files. This is a relatively uncommon format. Prefer this method over `write_excel()`.

to_xlsx(*excel_writer*, **args*, ***kwargs*) → Optional[str]

Writes XLSX Excel files. Prefer this method over `write_excel()`.

typeddfs._mixins._feather_parquet_hdf_mixin

Mixin for Feather, Parquet, and HDF5.

Module Contents

class typeddfs._mixins._feather_parquet_hdf_mixin.**_FeatherParquetHdfMixin**

classmethod read_feather(**args*, ***kwargs*) → __qualname__

classmethod read_hdf(**args*, *key: Optional[str] = None*, ***kwargs*) → __qualname__

classmethod read_parquet(**args*, ***kwargs*) → __qualname__

to_feather(*path_or_buf*, **args*, ***kwargs*) → Optional[str]

to_hdf(*path: typeddfs.utils._utils.PathLike*, *key: Optional[str] = None*, ***kwargs*) → None

to_parquet(*path_or_buf*, **args*, ***kwargs*) → Optional[str]

typeddfs._mixins._flexwf_mixin

Mixin for flex-wf.

Module Contents

class typeddfs._mixins._flexwf_mixin.**_FlexwfMixin**

classmethod read_flexwf(*path_or_buff*, *sep: str = '\\|\\|\\|'*, ***kwargs*) → __qualname__

Reads a "flexible-width format". The delimiter (`sep`) is important. **Note that ``sep`` is a regex pattern if it contains more than 1 char.**

These are designed to read and write (`to_flexwf`) as though they were fixed-width. Specifically, all of the columns line up but are separated by a possibly multi-character delimiter.

The files ignore blank lines, strip whitespace, always have a header, never quote values, and have no default index column unless given by `required_columns()`, etc.

Parameters

- **path_or_buff** – Path or buffer
- **sep** – The delimiter, a regex pattern
- **kwargs** – Passed to `read_csv`; may include 'comment' and 'skip_blank_lines'

to_flexwf(*path_or_buff=None*, *sep: str = '|||'*, *mode: str = 'w'*, ***kwargs*) → Optional[str]

Writes a fixed-width formatter, optionally with a delimiter, which can be multiple characters.

See `read_flexwf` for more info.

Parameters

- **path_or_buff** – Path or buffer
- **sep** – The delimiter, 0 or more characters
- **mode** – write or append (w/a)

- **kwargs** – Passed to `Utils.write`; may include 'encoding'

Returns The string data if `path_or_buff` is a buffer; None if it is a file

typeddfs._mixins._formatted_mixin

Mixin for formats like HTML and RST.

Module Contents

class typeddfs._mixins._formatted_mixin.**_FormattedMixin**

_tabulate(*fmt: Union[str, tabulate.TableFormat]*, ***kwargs*) → str

classmethod read_html(*path: typeddfs.utils._utils.PathLike*, **args*, ***kwargs*) → __qualname__

Similar to `pd.read_html`, but requires exactly 1 table and returns it.

Raises

- **lxml.etree.XMLSyntaxError** – If the HTML could not be parsed
- ***NoValueError*** – If no tables are found
- ***ValueNotUniqueError*** – If multiple tables are found

to_html(**args*, ***kwargs*) → Optional[str]

to_markdown(**args*, ***kwargs*) → Optional[str]

to_rst(*path_or_none: Optional[typeddfs.utils._utils.PathLike] = None*, *style: str = 'simple'*, *mode: str = 'w'*) → Optional[str]

Writes a reStructuredText table. :param path_or_none: Either a file path or `None` to return the string :param style: The type of table; currently only "simple" is supported :param mode: Write mode

typeddfs._mixins._full_io_mixin

Combines various IO mixins.

Module Contents

class typeddfs._mixins._full_io_mixin.**_FullIoMixin**

classmethod _call_read(*clazz*, *path: Union[pathlib.Path, str]*, *storage_options: Optional[pandas._typing.StorageOptions] = None*) → pandas.DataFrame

_call_write(*path: Union[pathlib.Path, str]*, *storage_options: Optional[pandas._typing.StorageOptions] = None*, *atomic: bool = False*) → Optional[str]

classmethod _check_io_ok(*path: pathlib.Path*, *fmt: Optional[*typeddfs.file_formats.FileFormat*]*)

classmethod _get_fmt(*path: pathlib.Path*) → Optional[*typeddfs.file_formats.FileFormat*]

classmethod _get_io(*on*, *path: pathlib.Path*, *fmt:* typeddfs.file_formats.FileFormat, *custom*, *prefix: str*)

classmethod _get_read_kwargs(*fmt: Optional[typeddfs.file_formats.FileFormat]*, *path: pathlib.Path*, *storage_options: Optional[pandas._typing.StorageOptions]*) → Mapping[str, Any]

classmethod _get_write_kwargs(*fmt: Optional[typeddfs.file_formats.FileFormat]*, *path: pathlib.Path*, *storage_options: Optional[pandas._typing.StorageOptions]*) → Mapping[str, Any]

pretty_print(*fmt: Union[None, str, tabulate.TableFormat] = None*, ***, *to: Optional[typeddfs.utils._utils.PathLike] = None*, *mode: str = 'w'*, ***kwargs*) → str

Outputs a pretty table using the tabulate package.

Parameters

- **fmt** – A tabulate format; if None, chooses according to `to`, falling back to `"plain"`
- **to** – Write to this path (.gz, .zip, etc. is inferred)
- **mode** – Write mode: 'w', 'a', or 'x'
- **kwargs** – Passed to tabulate

Returns The formatted string

typeddfs._mixins._fwf_mixin

Mixin for fixed-width format.

Module Contents

class typeddfs._mixins._fwf_mixin.**_FwfMixin**

classmethod read_fwf(**args*, ***kwargs*) → __qualname__

to_fwf(*path_or_buff=None*, *mode: str = 'w'*, *colspecs: Optional[Sequence[Tuple[int, int]]] = None*, *widths: Optional[Sequence[int]] = None*, *na_rep: Optional[str] = None*, *float_format: Optional[str] = None*, *date_format: Optional[str] = None*, *decimal: str = '.'*, ***kwargs*) → Optional[str]

Writes a fixed-width text format. See `read_fwf` and `to_flexwf` for more info.

Parameters

- **path_or_buff** – Path or buffer
- **mode** – write or append (w/a)
- **colspecs** – A list of tuples giving the extents of the fixed-width fields of each line as half-open intervals (i.e., [from, to[ )
- **widths** – A list of field widths which can be used instead of `colspecs` if the intervals are contiguous
- **na_rep** – Missing data representation
- **float_format** – Format string for floating point numbers
- **date_format** – Format string for datetime objects
- **decimal** – Character recognized as decimal separator. E.g. use ',' for European data.
- **kwargs** – Passed to `typeddfs.utils.Utils.write()`

Returns The string data if `path_or_buff` is a buffer; None if it is a file

typeddfs._mixins._ini_like_mixin

Mixin for INI, .properties, and TOML.

Module Contents

class typeddfs._mixins._ini_like_mixin.**_IniLikeMixin**

classmethod _assert_can_write_properties_class() → None

_assert_can_write_properties_instance() → None

classmethod _properties_files_apply() → bool

classmethod _read_properties_like(*unescape_keys*, *unescape_values*, *comment_chars: Set[str]*, *strip_quotes: bool*, *path_or_buff*, ***kwargs*) → __qualname__

Reads a .properties-like file.

_to_properties_like(*escape_keys*, *escape_values*, *sep: str*, *comment_char: str*, *path_or_buff=None*, *mode: str = 'w'*, *comment: Union[None, str, Sequence[str]] = None*, ***kwargs*) → Optional[str]

Writes a .properties-like file.

classmethod read_ini(*path_or_buff*, *hash_sign: bool = False*, *strip_quotes: bool = False*, ***kwargs*) → __qualname__

Reads an INI file.

> **Caution:** This is provided as a preview. It may have issues and may change.

Parameters

- **path_or_buff** – Path or buffer
- **hash_sign** – Allow # to denote a comment (as well as ;)
- **strip_quotes** – Remove quotation marks (”” or ‘’) surrounding the values
- **kwargs** – Passed to `typeddfs.utils.Utils.read()`

classmethod read_properties(*path_or_buff*, *strip_quotes: bool = False*, ***kwargs*) → __qualname__

Reads a .properties file. Backslashes, colons, spaces, and equal signs are escaped in keys and values.

> **Caution:** This is provided as a preview. It may have issues and may change. It currently does not support continued lines (ending with an odd number of backslashes).

Parameters

- **path_or_buff** – Path or buffer
- **strip_quotes** – Remove quotation marks (“”) surrounding values

- **kwargs** – Passed to `read_csv`; avoid setting

classmethod read_toml(*path_or_buff*, *aot: Optional[str] = 'row'*, *aot_only: bool = True*, ***kwargs*) → __qualname__

Reads a TOML file.

> **Caution:** This is provided as a preview. It may have issues and may change.

Parameters

- **path_or_buff** – Path or buffer
- **aot** – The name of the array of tables (i.e. `[[ table ]]`) If None, finds the unique outermost TOML key, implying `aot_only`.
- **aot_only** – Fail if any outermost keys other than the AOT are found
- **kwargs** – Passed to `Utils.read`

to_ini(*path_or_buff=None*, *comment: Union[None, str, Sequence[str]] = None*, *mode: str = 'w'*, ***kwargs*) → __qualname__

Writes an INI file.

> **Caution:** This is provided as a preview. It may have issues and may change.

Parameters

- **path_or_buff** – Path or buffer
- **comment** – Comment line(s) to add at the top of the document
- **mode** – 'w' (write) or 'a' (append)
- **kwargs** – Passed to `typeddfs.utils.Utils.write()`

to_properties(*path_or_buff=None*, *mode: str = 'w'*, ***, *comment: Union[None, str, Sequence[str]] = None*, ***kwargs*) → Optional[str]

Writes a .properties file. Backslashes, colons, spaces, and equal signs are escaped in keys. Backslashes are escaped in values. The separator is always =.

> **Caution:** This is provided as a preview. It may have issues and may change.

Parameters

- **path_or_buff** – Path or buffer
- **comment** – Comment line(s) to add at the top of the document
- **mode** – Write ('w') or append ('a')
- **kwargs** – Passed to `typeddfs.utils.Utils.write()`

Returns The string data if `path_or_buff` is a buffer; None if it is a file

to_toml(*path_or_buff=None*, *aot: str = 'row'*, *comment: Union[None, str, Sequence[str]] = None*, *mode: str = 'w'*, ***kwargs*) → __qualname__

Writes a TOML file.

> **Caution:** This is provided as a preview. It may have issues and may change.

Parameters

- **path_or_buff** – Path or buffer
- **aot** – The name of the array of tables (i.e. `[[ table ]]`)
- **comment** – Comment line(s) to add at the top of the document
- **mode** – 'w' (write) or 'a' (append)
- **kwargs** – Passed to `typeddfs.utils.Utils.write()`

typeddfs._mixins._json_xml_mixin

Mixin for JSON and XML.

Module Contents

class typeddfs._mixins._json_xml_mixin.**_JsonXmlMixin**

classmethod read_json(**args*, ***kwargs*) → __qualname__

classmethod read_xml(**args*, ***kwargs*) → __qualname__

to_json(*path_or_buf=None*, **args*, ***kwargs*) → Optional[str]

to_xml(*path_or_buf=None*, **args*, ***kwargs*) → Optional[str]

typeddfs._mixins._lines_mixin

Mixin for line-by-line text files.

Module Contents

class typeddfs._mixins._lines_mixin.**_LinesMixin**

classmethod _lines_files_apply() → bool

_tabulate(*fmt: Union[str, tabulate.TableFormat]*, ***kwargs*) → str

classmethod read_lines(*path_or_buff*, ***kwargs*) → __qualname__

Reads a file that contains 1 row and 1 column per line. Skips lines that are blank after trimming whitespace. Also skips comments if `comment` is set.

> **Caution:** For technical reasons, values cannot contain a 6-em space (U+2008). Their presence will result in undefined behavior.

Parameters

- **path_or_buff** – Path or buffer
- **kwargs** – Passed to `pd.DataFrame.read_csv` E.g. 'comment', 'encoding', 'skip_blank_lines', and 'line_terminator'

`to_lines`(*path_or_buff=None*, *mode: str = 'w'*, ***kwargs*) → Optional[str]

Writes a file that contains one row per line and 1 column per line. Associated with `.lines` or `.txt`.

> **Caution:** For technical reasons, values cannot contain a 6-em space (U+2008). Their presence will result in undefined behavior.

Parameters

- **path_or_buff** – Path or buffer
- **mode** – Write ('w') or append ('a')
- **kwargs** – Passed to `pd.DataFrame.to_csv`

Returns The string data if `path_or_buff` is a buffer; None if it is a file

typeddfs._mixins._new_methods_mixin

Mixin with misc new DataFrame methods.

Module Contents

class typeddfs._mixins._new_methods_mixin.**_NewMethodsMixin**

`cfirst`(*cols: Union[str, int, Sequence[str]]*) → __qualname__

Returns a new DataFrame with the specified columns appearing first.

Parameters **cols** – A list of columns, or a single column or column index

`drop_cols`(**cols: Union[str, Iterable[str]]*) → __qualname__

Drops columns, ignoring those that are not present.

Parameters **cols** – A single column name or a list of column names

`iter_row_col`() → Generator[Tuple[Tuple[int, int], Any], None, None]

Iterates over `((row, col), value)` tuples. The row and column are the row and column numbers, 1-indexed.

`only`(*column: str*, *exclude_na: bool = False*) → Any

Returns the single unique value in a column. Raises an error if zero or more than one value is in the column.

Parameters

- **column** – The name of the column

- **exclude_na** – Exclude None/pd.NA values

rename_cols(***cols*) → __qualname__

Shorthand for `.rename(columns=)`.

set_attrs(***attrs*) → __qualname__

Sets `pd.DataFrame.attrs`, returning a copy.

sort_natural(*column: str*, ***, *alg: Union[None, int, Set[str]] = None*, *reverse: bool = False*) → __qualname__

Calls `natsorted` on a single column.

Parameters

- **column** – The name of the (single) column to sort by
- **alg** – Input as the `alg` argument to `natsorted` If `None`, the "best" algorithm is chosen from the dtype of `column` via `typeddfs.utils.Utils.guess_natsort_alg()`. Otherwise, :meth:typeddfs.utils.Utils.exact_natsort_alg` is called with `Utils.exact_natsort_alg(alg)`.
- **reverse** – Reverse the sort order (e.g. 'z' before 'a')

sort_natural_index(***, *alg: int = None*, *reverse: bool = False*) → __qualname__

Calls natsorted on this index. Works for multi-index too.

Parameters

- **alg** – Input as the `alg` argument to `natsorted` If `None`, the "best" algorithm is chosen from the dtype of `column` via `typeddfs.utils.Utils.guess_natsort_alg()`. Otherwise, :meth:typeddfs.utils.Utils.exact_natsort_alg` is called with `Utils.exact_natsort_alg(alg)`.
- **reverse** – Reverse the sort order (e.g. 'z' before 'a')

st(**array_conditions: Sequence[bool]*, ***dict_conditions: Mapping[str, Any]*) → __qualname__

Short for "such that" – an alternative to slicing with `.loc`.

Parameters

- **array_conditions** – Conditions like `df["score"]<2`
- **dict_conditions** – Equality conditions, mapping column names to their values (ex `score=2`)

Returns A new DataFrame of the same type

strip_control_chars() → __qualname__

Removes all control characters (Unicode group 'C') from all string-typed columns.

typeddfs._mixins._pickle_mixin

Mixin for pickle.

Module Contents

class typeddfs._mixins._pickle_mixin._PickleMixin

classmethod read_pickle(*filepath_or_buffer*, **args*, ***kwargs*) → __qualname__

to_pickle(*path*, **args*, ***kwargs*) → None

typeddfs._mixins._pretty_print_mixin

Mixin that just overrides _repr_html.

Module Contents

class typeddfs._mixins._pretty_print_mixin._PrettyPrintMixin

A DataFrame with an overridden _repr_html_ and some simple additional methods.

_dims() → str

Returns a string describing the dimensionality.

Returns A text description of the dimensions of this DataFrame

_repr_html_() → str

Renders HTML for display() in Jupyter notebooks. Jupyter automatically uses this function.

Returns Just a string containing HTML, which will be wrapped in an HTML object

typeddfs._mixins._retype_mixin

Mixin that overrides Pandas functions to retype.

Module Contents

class typeddfs._mixins._retype_mixin._RetypeMixin

__add__(*other*)

__divmod__(*other*)

__mod__(*other*)

__mul__(*other*)

__pow__(*other*)

__radd__(*other*)

__rdivmod__(*other*)

__rmod__(*other*)

__rmul__(*other*)

__rpow__(*other*)

__rsub__(*other*)

__rtruediv__(*other*)

__sub__(*other*)

__truediv__(*other*)

classmethod _change(*df*) → __qualname__

classmethod _change_if_df(*df*)

classmethod _convert_typed(*df: pandas.DataFrame*)

_no_inplace(*kwargs*)

abs() → __qualname__

append(**args*, ***kwargs*) → __qualname__

applymap(**args*, ***kwargs*) → __qualname__

asfreq(**args*, ***kwargs*) → __qualname__

assign(***kwargs*) → __qualname__

astype(**args*, ***kwargs*) → __qualname__

bfill(***kwargs*) → __qualname__

convert_dtypes(**args*, ***kwargs*) → __qualname__

copy(*deep: bool = False*) → __qualname__

drop(**args*, ***kwargs*) → __qualname__

drop_duplicates(***kwargs*) → __qualname__

dropna(**args*, ***kwargs*) → __qualname__

ffill(***kwargs*) → __qualname__

fillna(**args*, ***kwargs*) → __qualname__

infer_objects(**args*, ***kwargs*) → __qualname__

reindex(**args*, ***kwargs*) → __qualname__

rename(**args*, ***kwargs*) → __qualname__

replace(**args*, ***kwargs*) → __qualname__

reset_index(**args*, ***kwargs*) → __qualname__

set_index(*keys*, *drop=True*, *append=False*, *inplace=False*, *verify_integrity=False*) → __qualname__

shift(**args*, ***kwargs*) → __qualname__

sort_values(**args*, ***kwargs*) → __qualname__

to_period(**args*, ***kwargs*) → __qualname__

to_timestamp(**args*, ***kwargs*) → __qualname__

transpose(**args*, ***kwargs*) → __qualname__

truncate(**args*, ***kwargs*) → __qualname__

tz_convert(**args*, ***kwargs*) → __qualname__

tz_localize(**args*, ***kwargs*) → __qualname__

typeddfs.utils

Tools that could possibly be used outside of typed-dfs.

Submodules

typeddfs.utils._format_support

Handles optional packages required for formats.

Module Contents

typeddfs.utils._format_support.**DfFormatSupport**

typeddfs.utils._format_support.**fastparquet**

typeddfs.utils._format_support.**openpyxl**

typeddfs.utils._format_support.**pyarrow**

typeddfs.utils._format_support.**pyxlsb**

typeddfs.utils._format_support.**tables**

typeddfs.utils._format_support.**tomlkit**

typeddfs.utils._utils

Internal utilities for typeddfs.

Module Contents

typeddfs.utils._utils.**PathLike**

typeddfs.utils._utils.**_AUTO_DROPPED_NAMES**

typeddfs.utils._utils.**_DEFAULT_ATTRS_SUFFIX = .attrs.json**

typeddfs.utils._utils.**_DEFAULT_HASH_ALG = sha256**

typeddfs.utils._utils._FAKE_SEP =

typeddfs.utils._utils._FORBIDDEN_NAMES

typeddfs.utils._utils._SENTINEL

typeddfs.utils.checksum_models

Models for shasum-like files.

Module Contents

class typeddfs.utils.checksum_models.**ChecksumFile**

delete() → None

Deletes the hash file by calling `pathlib.Path.unlink(self.hash_path)`.

Raises **OSError** – Accordingly

property file_path → pathlib.Path

property hash_value → str

load() → __qualname__

Returns a copy of `self` read from `hash_path`.

classmethod new(*hash_path: typeddfs.utils._utils.PathLike*, *file_path: typeddfs.utils._utils.PathLike*, *hash_value: str*) → *ChecksumFile*

Use this as a constructor.

classmethod parse(*path: pathlib.Path*, *, *lines: Optional[Sequence[str]] = None*) → __qualname__

Reads hash file contents.

Parameters

- **path** – The path of the checksum file; required to resolve paths relative to its parent
- **lines** – The lines in the checksum file; reads `path` if None

Returns A ChecksumFile

rename(*path: pathlib.Path*) → __qualname__

Replaces `self.file_path` with `path`. This will affect the filename written in a .shasum-like file. No OS operations are performed.

update(*value: str*, *overwrite: Optional[bool] = True*) → __qualname__

Modifies the hash.

Parameters

- **value** – The new hex-encoded hash
- **overwrite** – If `None`, requires that the value is the same as before (no operation is performed). If `False`, this method will always raise an error.

verify(*computed: str*) → None

Verifies the checksum.

Parameters **computed** – A pre-computed hex-encoded hash

Raises *HashDidNotValidateError* – If the hashes are not equal

write() → None

Writes the hash file.

Raises **OsError** – Accordingly

class typeddfs.utils.checksum_models.**ChecksumMapping**

__add__(*other: Union[*ChecksumMapping*, Mapping[typeddfs.utils._utils.PathLike, str], __qualname__]*) → __qualname__

Performs a symmetric addition.

Raises **ValueError** – If `other` intersects (shares keys) with `self`

See also:

append()

__contains__(*path: pathlib.Path*) → bool

__getitem__(*path: pathlib.Path*) → str

__len__() → int

__sub__(*other: Union[typeddfs.utils._utils.PathLike, Iterable[typeddfs.utils._utils.PathLike],* ChecksumMapping*]*) → __qualname__

Removes entries.

See also:

remove()

append(*append: Mapping[typeddfs.utils._utils.PathLike, str]*, *, *overwrite: Optional[bool] = False*) → __qualname__

Append paths to a dir hash file. Like *update()* but less flexible and only for adding paths.

property **entries** → Mapping[pathlib.Path, str]

get(*key: pathlib.Path*, *default: Optional[str] = None*) → Optional[str]

items() → AbstractSet[Tuple[pathlib.Path, str]]

keys() → AbstractSet[pathlib.Path]

load(*missing_ok: bool = False*) → __qualname__

Replaces this map with one read from the hash file.

Parameters **missing_ok** – If the hash path does not exist, treat it has having no items

classmethod **new**(*hash_path: typeddfs.utils._utils.PathLike*, *dct: Mapping[typeddfs.utils._utils.PathLike, str]*) → *ChecksumMapping*

Use this as the constructor.

classmethod parse(*path: pathlib.Path*, *, *lines: Optional[Sequence[str]] = None*, *missing_ok: bool = False*, *subdirs: bool = False*) → __qualname__

Reads hash file contents.

Parameters

- **path** – The path of the checksum file; required to resolve paths relative to its parent
- **lines** – The lines in the checksum file; reads `path` if None
- **missing_ok** – If `path` does not exist, assume it contains no items
- **subdirs** – Permit files within subdirectories specified with `/` Most tools do not support these.

Returns A mapping from raw string filenames to their hex hashes. Any node called `./` in the path is stripped.

remove(*remove: Union[typeddfs.utils._utils.PathLike, Iterable[typeddfs.utils._utils.PathLike]]*, *, *missing_ok: bool = False*) → __qualname__

Strips paths from this hash collection. Like *update()* but less flexible and only for removing paths.

Raises ***typeddfs.df_errors.PathNotRelativeError*** – To avoid, try calling `resolve` first

update(*update: Union[Callable[[pathlib.Path], Optional[typeddfs.utils._utils.PathLike]], Mapping[typeddfs.utils._utils.PathLike, Optional[typeddfs.utils._utils.PathLike]]]*, *, *missing_ok: bool = True*, *overwrite: Optional[bool] = True*) → __qualname__

Returns updated hashes from a dir hash file.

Parameters

- **update** – Values to overwrite. May be a function or a dictionary from paths to values. If `None` is returned, the entry will be removed; otherwise, updates with the returned hex hash.
- **missing_ok** – Require that the path is already listed
- **overwrite** – Allow overwriting an existing value. If `None`, only allow if the hash is the same.

values() → ValuesView[str]

verify(*path: typeddfs.utils._utils.PathLike*, *computed: str*, *, *resolve: bool = False*, *exist: bool = False*) → None

Verifies a checksum. The file `path` must be listed.

Parameters

- **path** – The file to look for
- **computed** – A pre-computed hex-encoded hash; if set, do not calculate from `path`
- **resolve** – Resolve paths before comparison
- **exist** – Require that `path` exists

Raises

- **FileNotFoundError** – If `path` does not exist
- ***HashFileMissingError*** – If the hash file does not exist
- ***HashDidNotValidateError*** – If the hashes are not equal
- **HashVerificationError`** – Superclass of `HashDidNotValidateError` if the filename is not listed, etc.

write(*, *sort: Union[bool, Callable[[Sequence[pathlib.Path]], Sequence[pathlib.Path]]] = False*, *rm_if_empty: bool = False*) → None

Writes to the hash (.shasum-like) file.

Parameters

- **sort** – Sort with this function, or `sorted` if True
- **rm_if_empty** – Delete with `pathlib.Path.unlink` if this contains no items

Raises **OSError** – Accordingly

typeddfs.utils.checksums

Tools for shasum-like files.

Module Contents

class typeddfs.utils.checksums.**Checksums**

alg :str

calc_hash(*path: typeddfs.utils._utils.PathLike*) → str

Calculates the hash of a file and returns it, hex-encoded.

classmethod default_algorithm() → str

delete_any(*path: typeddfs.utils._utils.PathLike*, *, *rm_if_empty: bool = False*) → None

Deletes the filesum and removes `path` from the dirsum. Ignores missing files.

generate_dirsum(*directory: typeddfs.utils._utils.PathLike*, *glob: str = '*'*) → *typeddfs.utils.checksum_models.ChecksumMapping*

Generates a new hash mapping, calculating hashes for extant files.

Parameters

- **directory** – Base directory
- **glob** – Glob pattern under `directory` (cannot be recursive)

Returns A ChecksumMapping; use `.write` to write it

get_dirsum_of_dir(*path: typeddfs.utils._utils.PathLike*) → pathlib.Path

Returns the path required for the per-directory hash of `path`.

Example

```
Utils.get_hash_file("my_dir") # Path("my_dir", "my_dir.sha256")
```

get_dirsum_of_file(*path: typeddfs.utils._utils.PathLike*) → pathlib.Path

Returns the path required for the per-directory hash of `path`.

Example

```
Utils.get_hash_file(Path("my_dir, my_file.txt.gz")) # Path("my_dir", "my_dir.
sha256")
```

get_filesum_of_file(*path: typeddfs.utils._utils.PathLike*) → pathlib.Path

Returns the path required for the per-file hash of `path`.

Example

```
Utils.get_hash_file("my_file.txt.gz") # Path("my_file.txt.gz.sha256")
```

classmethod guess_algorithm(*path: typeddfs.utils._utils.PathLike*) → str

Guesses the hashlib algorithm used from a hash file.

Parameters **path** – The hash file (e.g. my-file.sha256)

Example

```
Utils.guess_algorithm("my_file.sha1") # "sha1"
```

load_dirsum_exact(*path: typeddfs.utils._utils.PathLike*, *, *missing_ok: bool = True*) → *typeddfs.utils.checksum_models.ChecksumMapping*

load_dirsum_of_dir(*path: typeddfs.utils._utils.PathLike*, *, *missing_ok: bool = True*) → *typeddfs.utils.checksum_models.ChecksumMapping*

load_dirsum_of_file(*path: typeddfs.utils._utils.PathLike*, *, *missing_ok: bool = True*) → *typeddfs.utils.checksum_models.ChecksumMapping*

load_filesum_exact(*path: typeddfs.utils._utils.PathLike*) → *typeddfs.utils.checksum_models.ChecksumFile*

load_filesum_of_file(*path: typeddfs.utils._utils.PathLike*) → *typeddfs.utils.checksum_models.ChecksumFile*

classmethod resolve_algorithm(*alg: str*) → str

Finds a hash algorithm by name in `hashlib`. Converts to lowercase and removes hyphens.

Raises ***HashAlgorithmMissingError*** – If not found

verify_any(*path: typeddfs.utils._utils.PathLike*, *, *file_hash: bool*, *dir_hash: bool*, *computed: Optional[str]*) → Optional[str]

verify_hex(*path: typeddfs.utils._utils.PathLike*, *expected: str*) → Optional[str]

Verifies a hash directly from a hex string.

write_any(*path: typeddfs.utils._utils.PathLike*, *, *to_file: bool*, *to_dir: bool*, *overwrite: Optional[bool] = True*) → Optional[str]

Adds and/or appends the hex hash of `path`.

Parameters

- **path** – Path to the file to hash
- **to_file** – Whether to save a per-file hash
- **to_dir** – Whether to save a per-dir hash

- **overwrite** – If True, overwrite the file hash and any entry in the dir hash. If False, never overwrite either. If None, never overwrite, but ignore if equal to any existing entries.

typeddfs.utils.cli_help

Utils for getting nice CLI help on DataFrame inputs.

Attention: The exact text used in this module are subject to change.

Note: Two consecutive newlines (\n\n) are used to separate sections. This is consistent with a number of formats, including Markdown, reStructuredText, and Typer.

Module Contents

class typeddfs.utils.cli_help.**DfCliHelp**

classmethod help(*clazz: Type[*typeddfs.abs_dfs.AbsDf*]*) → *DfHelp*

Returns info suitable for CLI help.

Display this info as the help description for an argument that's a path to a table file that will be read with *typeddfs.abs_dfs.AbsDf.read_file()* for clazz.

Parameters **clazz** – The typeddfs.typed_dfs.AbsDf subclass

classmethod list_formats(**, flexwf_sep: str = _FLEXWF_SEP, hdf_key: str = _HDF_KEY, toml_aot: str = _TOML_AOT*) → *DfFormatsHelp*

Lists all file formats with descriptions.

For example, *typeddfs.file_formats.FileFormat.ods* is "OpenDocument Spreadsheet".

class typeddfs.utils.cli_help.**DfFormatHelp**

Help text on a specific file format.

desc :str

fmt :typeddfs.file_formats.FileFormat

property all_suffixes → Sequence[str]

Returns all suffixes, naturally sorted.

property bare_suffixes → Sequence[str]

Returns all suffixes, excluding compressed variants (etc. .gz), naturally sorted.

get_text() → str

Returns a 1-line string of the suffixes and format description.

class typeddfs.utils.cli_help.**DfFormatsHelp**

Help on file formats only.

Initialize self. See help(type(self)) for accurate signature.

get_long_text(*, *recommended_only: bool = False*, *nl: str = '\n'*, *bullet: str = '- '*, *indent: str = ' '*) → str

Returns a multi-line text listing of allowed file formats.

Parameters

- **recommended_only** – Skip non-recommended file formats
- **nl** – Newline characters; use "n", "\n", or " "
- **bullet** – Prepended to each item
- **indent** – Spaces for nested indent

Returns

Something like:: [[Supported formats]]:

.csv[.bz2/.gz/.xz/.zip]: comma-delimited

.parquet/.snappy: Parquet

.h5/.hdf/.hdf5: HDF5 (key 'df') [discouraged]

.pickle/.pkl: Python Pickle [discouraged]

get_short_text(*, *recommended_only: bool = False*) → str

Returns a single-line text listing of allowed file formats.

Parameters **recommended_only** – Skip non-recommended file formats

Returns

Something like:: .csv, .tsv/.tab, or .flexwf [.gz,/.xz,/.zip/.bz2]; .feather, .pickle, or .snappy …

class typeddfs.utils.cli_help.**DfHelp**

Info on a TypedDf suitable for CLI help.

clazz :Type[typeddfs.abs_dfs.AbsDf]

formats :DfFormatsHelp

get_header_text(*, *use_doc: bool = True*, *nl: str = '\n'*) → str

Returns a multi-line header of the DataFrame name and docstring.

Parameters

- **use_doc** – Include the docstring, as long as it is not None
- **nl** – Newline characters; use "n", "nn", or " "

Returns

Something like:: Path to a Big Table file.

This is a big table for big things.

get_long_text(*, *use_doc: bool = True*, *recommended_only: bool = False*, *nl: str = '\n'*, *bullet: str = '- '*, *indent: str = ' '*) → str

Returns a multi-line text description of the DataFrame. Includes its required and optional columns, and supported file formats.

Parameters

- **use_doc** – Include the docstring of the DataFrame type
- **recommended_only** – Only include recommended formats

- **nl** – Newline characters; use “n”, “nn”, or ” “
- **bullet** – Prepended to each item
- **indent** – Spaces for nested indent

abstract get_long_typing_text() → str

Returns multi-line text on only the required columns / structure.

get_short_text(*, *use_doc: bool = True*, *recommended_only: bool = False*, *nl: str = '\n'*) → str

Returns a multi-line description with compressed text.

Parameters

- **use_doc** – Include the docstring of the DataFrame type
- **recommended_only** – Only include recommended formats
- **nl** – Newline characters; use “n”, “\n”, or ” “

abstract get_short_typing_text() → str

Returns 1-line text on only the required columns / structure.

property typing → *typeddfs.df_typing.DfTyping*

typeddfs.utils.dtype_utils

Data type tools for typed-dfs.

Module Contents

class typeddfs.utils.dtype_utils.**DtypeUtils**

is_bool

is_bool_dtype

is_categorical

is_categorical_dtype

is_complex

is_complex_dtype

is_datetime64_any_dtype

is_datetime64tz_dtype

is_extension_type

is_float

is_float_dtype

is_integer

is_integer_dtype

is_interval

is_interval_dtype

is_number

is_numeric_dtype

is_object_dtype

is_period_dtype

is_scalar

is_string_dtype

classmethod describe_dtype(*t: Type[Any]*, *, *short: bool = False*) → Optional[str]

Returns a string name for a Pandas-supported dtype.

Parameters

- **t** – Any Python type
- **short** – Use shorter strings (e.g. "int" instead of "integer")

Returns A string like "floating-point" or "zoned datetime". Returns `None` if no good name is found or if `t` is `None`.

typeddfs.utils.io_utils

Tools for IO.

Module Contents

class typeddfs.utils.io_utils.**IoUtils**

classmethod get_encoding(*encoding: str = 'utf-8'*) → str

Returns a text encoding from a more flexible string. Ignores hyphens and lowercases the string. Permits these nonstandard shorthands:

- "`platform`": use `sys.getdefaultencoding()` on the fly
- "`utf8(bom)`": use "`utf-8-sig`" on Windows; "`utf-8`" otherwise
- "`utf16(bom)`": use "`utf-16-sig`" on Windows; "`utf-16`" otherwise
- "`utf32(bom)`": use "`utf-32-sig`" on Windows; "`utf-32`" otherwise

classmethod get_encoding_errors(*errors: Optional[str]*) → Optional[str]

Returns the value passed as``errors=`` in `open`. :raises ValueError: If invalid

classmethod is_binary(*path: typeddfs.utils._utils.PathLike*) → bool

classmethod path_or_buff_compression(*path_or_buff*, *kwargs*) → *typeddfs.file_formats.CompressionFormat*

classmethod read(*path_or_buff*, *, *mode: str = 'r'*, ***kwargs*) → str

Reads using Pandas's `get_handle`. By default (unless `compression=` is set), infers the compression type from the filename suffix. (e.g. `.csv.gz`).

classmethod tmp_path(*path: typeddfs.utils._utils.PathLike*, *extra: str = 'tmp'*) → pathlib.Path

classmethod verify_can_read_files(**paths: Union[str, pathlib.Path]*, *missing_ok: bool = False*, *attempt: bool = False*) → None

Checks that all files can be written to, to ensure atomicity before operations.

Parameters

- ***paths** – The files
- **missing_ok** – Don't raise an error if a path doesn't exist
- **attempt** – Actually try opening

Returns If a path is not a file (modulo existence) or doesn't have 'W' set

Return type *ReadPermissionsError*

classmethod verify_can_write_dirs(**paths: Union[str, pathlib.Path]*, *missing_ok: bool = False*) → None

Checks that all directories can be written to, to ensure atomicity before operations.

Parameters

- ***paths** – The directories
- **missing_ok** – Don't raise an error if a path doesn't exist

Returns If a path is not a directory (modulo existence) or doesn't have 'W' set

Return type *WritePermissionsError*

classmethod verify_can_write_files(**paths: Union[str, pathlib.Path]*, *missing_ok: bool = False*, *attempt: bool = False*) → None

Checks that all files can be written to, to ensure atomicity before operations.

Parameters

- ***paths** – The files
- **missing_ok** – Don't raise an error if a path doesn't exist
- **attempt** – Actually try opening

Returns If a path is not a file (modulo existence) or doesn't have 'W' set

Return type *WritePermissionsError*

classmethod write(*path_or_buff*, *content*, ***, *mode: str = 'w'*, *atomic: bool = False*, ***kwargs*) → Optional[str]

Writes using Pandas's `get_handle`. By default (unless `compression=` is set), infers the compression type from the filename suffix (e.g. `.csv.gz`).

typeddfs.utils.json_utils

Tools that could possibly be used outside of typed-dfs.

Module Contents

class typeddfs.utils.json_utils.**JsonDecoder**

from_bytes(*data: ByteString*) → Any

from_str(*data: str*) → Any

class typeddfs.utils.json_utils.**JsonEncoder**

bytes_options :int

default :Callable[[Any], Any]

prep :Optional[Callable[[Any], Any]]

str_options :int

as_bytes(*data: Any*) → ByteString

as_str(*data: Any*) → str

class typeddfs.utils.json_utils.**JsonUtils**

classmethod decoder() → *JsonDecoder*

classmethod encoder(**fallbacks: Optional[Callable[[Any], Any]]*, *indent: bool = True*, *sort: bool = False*, *preserve_inf: bool = True*, *last: Optional[Callable[[Any], Any]] = str*) → *JsonEncoder*

Serializes to string with orjson, indenting and adding a trailing newline. Uses `orjson_default()` to encode more types than orjson can.

Parameters

- **indent** – Indent by 2 spaces
- **preserve_inf** – Preserve infinite values with `orjson_preserve_inf()`
- **sort** – Sort keys with `orjson.OPT_SORT_KEYS`; only for `typeddfs.json_utils.JsonEncoder.as_str()`
- **last** – Last resort option to encode a value

classmethod misc_types_default() → Callable[[Any], Any]

classmethod new_default(**fallbacks: Optional[Callable[[Any], Any]]*, *first: Optional[Callable[[Any], Any]] = _misc_types_default*, *last: Optional[Callable[[Any], Any]] = str*) → Callable[[Any], Any]

Creates a new method to be passed as `default=` to `orjson.dumps`. Tries, in order: `orjson_default()`, `fallbacks`, then `str`.

Parameters

- **first** – Try this first
- **fallbacks** – Tries these, in order, after `first`, skipping any None

- **last** – Use this as the last resort; consider `str` or `repr`

classmethod preserve_inf(*data: Any*) → Any

Recursively replaces infinite float and numpy values with strings. Orjson encodes NaN, inf, and +inf as JSON null. This function converts to string as needed to preserve infinite values. Any float scalar (`np.floating` and `float`) will be replaced with a string. Any `np.ndarray`, whether it contains an infinite value or not, will be converted to an ndarray of strings. The returned result may still not be serializable with orjson or `orjson_bytes()`. Trying those methods is the best way to test for serializablity.

typeddfs.utils.misc_utils

Misc tools for typed-dfs.

Module Contents

class typeddfs.utils.misc_utils.**MiscUtils**

classmethod choose_table_format(*, *path: typeddfs.utils._utils.PathLike*, *fmt: Union[None, tabulate.TableFormat, str] = None*, *default: str = 'plain'*) → Union[str, tabulate.TableFormat]

Makes a best-effort guess of a good tabulate format from a path name.

classmethod delete_file(*path: typeddfs.utils._utils.PathLike*, *, *missing_ok: bool = False*, *alg: str = _DEFAULT_HASH_ALG*, *attrs_suffix: str = _DEFAULT_ATTRS_SUFFIX*, *rm_if_empty: bool = True*) → None

Deletes a file, plus the checksum file and/or directory entry, and `.attrs.json`.

Parameters

- **path** – The path to delete
- **missing_ok** – ok if the path does not exist (will still delete any associated paths)
- **alg** – The checksum algorithm
- **attrs_suffix** – The suffix for attrs file (normally .attrs.json)
- **rm_if_empty** – Remove the dir checksum file if it contains no additional paths

Raises *typeddfs.df_errors.PathNotRelativeError* – To avoid, try calling `resolve` first

classmethod freeze(*v: Any*) → Any

Returns `v` or a hashable view of it. Note that the returned types must be hashable but might not be ordered. You can generally add these values as DataFrame elements, but you might not be able to sort on those columns.

Parameters **v** – Any value

Returns Either `v` itself, a `typeddfs.utils.FrozeSet` (subclass of `typing.AbstractSet`), a `typeddfs.utils.FrozeList` (subclass of `typing.Sequence`), or a `typeddfs.utils.FrozeDict` (subclass of `typing.Mapping`). int, float, str, np.generic, and tuple are always returned as-is.

Raises

- **AttributeError** – If `v` is not hashable and could not converted to a FrozeSet, FrozeList, or FrozeDict, *or* if one of the elements for one of the above types is not hashable.
- **TypeError** – If `v` is an `Iterator` or *collections.deque* `

classmethod join_to_str(**items: Any*, *last: str*, *sep: str = ', '*) → str

Joins items to something like "cat, dog, and pigeon" or "cat, dog, or pigeon".

Parameters

- **items** – Items to join; `str(item) for item in items` will be used
- **last** – Probably "and", "or", "and/or", or "" Spaces are added/removed as needed if `suffix` is alphanumeric or "and/or", after stripping whitespace off the ends.
- **sep** – Used to separate all words; include spaces as desired

Examples

- `join_to_str(["cat", "dog", "elephant"], last="and") # cat, dog, and elephant`
- `join_to_str(["cat", "dog"], last="and") # cat and dog`
- `join_to_str(["cat", "dog", "elephant"], last="", sep="/") # cat/dog/elephant`

classmethod plain_table_format(***, *sep: str = ' '*, ***kwargs*) → tabulate.TableFormat

Creates a simple tabulate style using a column-delimiter `sep`.

Returns A tabulate `TableFormat`, which can be passed as a style

classmethod table_format(*fmt: str*) → tabulate.TableFormat

Gets a tabulate style by name.

Returns A TableFormat, which can be passed as a style

classmethod table_formats() → Sequence[str]

Returns the names of styles for tabulate.

typeddfs.utils.parse_utils

Misc tools for typed-dfs.

Module Contents

class typeddfs.utils.parse_utils.**ParseUtils**

classmethod _re_leaf(*at: str*, *items: Mapping[str, Any]*) → Generator[Tuple[str, Any], None, None]

classmethod _un_leaf(*to: MutableMapping[str, Any]*, *items: Mapping[str, Any]*) → None

classmethod dict_to_dots(*items: Mapping[str, Any]*) → Mapping[str, Any]

Performs the inverse of *dots_to_dict()*.

Example

```
Utils.dict_to_dots({"genus":  {"species":  "fruit bat"}}) == {"genus.species":
"fruit bat"}
```

classmethod dicts_to_toml_aot(*dicts: Sequence[Mapping[str, Any]]*)

Make a tomlkit Document consisting of an array of tables ("AOT").

Parameters **dicts** – A sequence of dictionaries

Returns //github.com/sdispater/tomlkit/blob/master/tomlkit/items.py>`_ (i.e. `[[array]]`)

Return type A tomlkit`AoT<https

classmethod dots_to_dict(*items: Mapping[str, Any]*) → Mapping[str, Any]

Make sub-dictionaries from substrings in `items` delimited by `.`. Used for TOML.

Example

```
Utils.dots_to_dict({"genus.species":  "fruit bat"}) == {"genus":  {"species":
"fruit bat"}}
```

See also:

dict_to_dots()

classmethod property_key_escape(*s: str*) → str

Escapes a key in a .property file.

classmethod property_key_unescape(*s: str*) → str

Un-escapes a key in a .property file.

classmethod property_value_escape(*s: str*) → str

Escapes a value in a .property file.

classmethod property_value_unescape(*s: str*) → str

Un-escapes a value in a .property file.

classmethod strip_control_chars(*s: str*) → str

Strips all characters under the Unicode 'Cc' category.

typeddfs.utils.sort_utils

Tools for sorting.

Module Contents

class typeddfs.utils.sort_utils.**SortUtils**

classmethod _ns_info_from_int_flag(*val: int*) → NatsortFlagsAndValue

classmethod all_natsort_flags() → Mapping[str, int]

Returns all flags defined by natsort, including combined and default flags. Combined flags are, e.g., `ns_enum.ns.REAL ns_enum.nsFLOAT | ns_enum.ns.SIGNED.`. Default flags are, e.g., `ns_enum.ns.UNSIGNED`.

See also:

`std_natsort_flags()`

Returns A mapping from flag name to int value

classmethod core_natsort_flags() → Mapping[str, int]

Returns natsort flags that are not combinations or defaults.

See also:

all_natsort_flags()

Returns A mapping from flag name to int value

classmethod exact_natsort_alg(*flags: Union[None, int, Collection[Union[int, str]]]*) → NatsortFlagsAndValue

Gets the flag names and combined `alg=` argument for natsort.

Examples

- `exact_natsort_alg({"REAL"}) == ({"FLOAT", "SIGNED"}, ns.FLOAT | ns.SIGNED)`
- `exact_natsort_alg({}) == ({}, 0)`
- `exact_natsort_alg(ns.LOWERCASEFIRST) == ({"LOWERCASEFIRST"}, ns.LOWERCASEFIRST)`
- `exact_natsort_alg({"localenum", "numafter"}) == ({"LOCALENUM", "NUMAFTER"}, ns.LOCALENUM | ns.NUMAFTER)`

Parameters **flags** – Can be either: - a single integer `alg` argument - a set of flag ints and/or names in `natsort.ns`

Returns A tuple of the set of flag names, and the corresponding input to `natsorted` Only uses standard flag names, never the "combined" ones. (E.g. `exact_natsort_alg({"REAL"})` will return `({"FLOAT", "SIGNED"}, ns.FLOAT | ns.SIGNED)`.

classmethod guess_natsort_alg(*dtype: Type[Any]*) → NatsortFlagsAndValue

Guesses a good natsorted flag for the dtype.

Here are some specifics:

- integers INT and SIGNED
- floating-point FLOAT and SIGNED
- strings COMPATIBILITYNORMALIZE and GROUPLETTERS
- datetime GROUPLETTERS (only affects 'Z' vs. 'z'; shouldn't matter)

Parameters **dtype** – Probably from `pd.Series.dtype`

Returns A tuple of (set of flags, int) – see *exact_natsort_alg()*

classmethod natsort(*lst: Iterable[T]*, *dtype: Type[T]*, ***, *alg: Union[None, int, Set[str]] = None*, *reverse: bool = False*) → Sequence[T]

Perform a natural sort consistent with the type `dtype`. Uses natsort.

See also:

guess_natsort_alg()

Parameters

- **lst** – A sequence of things to sort
- **dtype** – The type; must be a subclass of each element in `lst`
- **alg** – A specific natsort algorithm or set of flags
- **reverse** – Sort in reverse (e.g. Z to A or 9 to 1)

Package Contents

class typeddfs.utils.**Utils**

json_decoder

json_encoder

classmethod banned_names() → Set[str]

Lists strings that cannot be used for column names or index level names.

classmethod default_hash_algorithm() → str

classmethod insecure_hash_functions() → Set[str]

5.1.2 Submodules

typeddfs._core_dfs

Module Contents

class typeddfs._core_dfs.**CoreDf**(*data=None*, *index=None*, *columns=None*, *dtype=None*, *copy=False*)

An abstract Pandas DataFrame subclass with additional methods.

abstract classmethod new_df(***kwargs*) → __qualname__

Creates a new, somewhat arbitrary DataFrame of this type. Calling this with no arguments should always be supported.

Parameters ****kwargs** – These should be narrowed by the overriding method as needed.

Raises

- *UnsupportedOperationError* – Can be raised if a valid DataFrame is too difficult to create.
- *InvalidDfError* – May be raised if the type requires specific constraints and did not overload this method to account for them. While programmers using the type should be aware of this possibility, consuming code, in general, should assume that `new_df` will always work.

vanilla() → pandas.DataFrame

Makes a copy that's a normal Pandas DataFrame.

Returns A shallow copy with its `__class__` set to pd.DataFrame

vanilla_reset() → pandas.DataFrame

Same as *vanilla()*, but resets the index – but dropping the index if it has no name. This means that an effectively index-less dataframe will not end up with an extra column called "index".

typeddfs._entries

Convenient code for import.

Module Contents

typeddfs._entries.**affinity_matrix**

typeddfs._entries.**example**

typeddfs._entries.**matrix**

typeddfs._entries.**typed**

typeddfs._entries.**untyped**

typeddfs._entries.**wrap**

class typeddfs._entries.**FinalDf**(*data=None*, *index=None*, *columns=None*, *dtype=None*, *copy=False*)

An untyped DataFrame meant for general use.

class typeddfs._entries.**TypedDfs**

The only thing you need to import from `typeddfs`.

Contains static factory methods to build new DataFrame subclasses. In particular, see:

```
- :meth:`typed`
- :meth:`untyped`
- :meth:`matrix`
- :meth:`affinity_matrix`
```

Checksums

ClashError

CompressionFormat

FileFormat

FilenameSuffixError

FinalDf

FrozeDict

FrozeList

FrozeSet

InvalidDfError

MissingColumnError

NoValueError

NonStrColumnError

NotSingleColumnError

UnexpectedColumnError

UnexpectedIndexNameError

UnsupportedOperationError

Utils

ValueNotUniqueError

VerificationFailedError

_logger

classmethod affinity_matrix(*name: str*, *doc: Optional[str] = None*) → *typeddfs.builders.AffinityMatrixDfBuilder*

Creates a new subclass of an `typeddfs.matrix_dfs.AffinityMatrixDf`.

Parameters

- **name** – The name that will be used for the new class
- **doc** – The docstring for the new class

Returns A builder instance (builder pattern) to be used with chained calls

classmethod example() → Type[*typeddfs.typed_dfs.TypedDf*]

Creates a new example TypedDf subclass. The class has:

- required index "key"
- required column "value"
- reserved column "note"
- no other columns

Returns The created class

classmethod matrix(*name: str*, *doc: Optional[str] = None*) → *typeddfs.builders.MatrixDfBuilder*

Creates a new subclass of an `typeddfs.matrix_dfs.MatrixDf`.

Parameters

- **name** – The name that will be used for the new class
- **doc** – The docstring for the new class

Returns A builder instance (builder pattern) to be used with chained calls

classmethod typed(*name: str*, *doc: Optional[str] = None*) → *typeddfs.builders.TypedDfBuilder*

Creates a new type with flexible requirements. The class will enforce constraints and subclass *typeddfs.typed_dfs.TypedDf*.

Parameters

- **name** – The name that will be used for the new class
- **doc** – The docstring for the new class

Returns A builder instance (builder pattern) to be used with chained calls

Example

```
TypedDfs.typed("MyClass").require("name", index=True).build()
```

classmethod untyped(*name: str*, *doc: Optional[str] = None*) → Type[*typeddfs.untyped_dfs.UntypedDf*]

Creates a new subclass of `UntypedDf`. The returned class will not enforce constraints but will have some extra methods. In general *typed()* should be preferred because it has more consistent behavior, especially for IO.

Parameters

- **name** – The name that will be used for the new class
- **doc** – The docstring for the new class

Returns A class instance

Example

```
MyClass = TypedDfs.untyped("MyClass")
```

classmethod wrap(*df: pandas.DataFrame*) → *FinalDf*

Just wraps a DataFrame into a simple untyped DataFrame. Useful to quickly access a function only defined on typeddfs DataFrames.

Example

```
TypedDfs.wrap(df).write_file("abc.feather")
```

typeddfs._pretty_dfs

Defines a DataFrame with simple extra functions like `column_names`.

Module Contents

class typeddfs._pretty_dfs.**PrettyDf**(*data=None*, *index: Axes | None = None*, *columns: Axes | None = None*, *dtype: Dtype | None = None*, *copy: bool | None = None*)

A DataFrame with an overridden `_repr_html_` and some simple additional methods.

property _constructor_expanddim

column_names() → List[str]

Returns the list of columns.

Returns A Python list

index_names() → List[str]

Returns the list of index names. Unlike `self.index.names`, returns `[]` instead of `[None]` if there is no index.

Returns A Python list

is_multindex() → bool

Returns whether this is a `pd.MultiIndex`.

n_columns() → int

Returns the number of columns.

n_indices() → int

Returns the number of index names.

n_rows() → int

Returns the number of rows.

typeddfs.abs_dfs

Defines a low-level DataFrame subclass. It overrides a lot of methods to auto-change the type back to `cls`.

Module Contents

class typeddfs.abs_dfs.**AbsDf**(*data=None*, *index=None*, *columns=None*, *dtype=None*, *copy=False*)

An abstract Pandas DataFrame subclass with additional methods.

classmethod _check(*df*) → None

Should raise an *typeddfs.df_errors.InvalidDfError* or subclass for issues.

classmethod can_read() → Set[*typeddfs.file_formats.FileFormat*]

Returns all formats that can be read using `read_file`. Some depend on the availability of optional packages. The lines format (`.txt`, `.lines`, etc.) is only included if this DataFrame *can* support only 1 column+index. See *typeddfs.file_formats.FileFormat.can_read()*.

classmethod can_write() → Set[*typeddfs.file_formats.FileFormat*]

Returns all formats that can be written to using `write_file`. Some depend on the availability of optional packages. The lines format (`.txt`, `.lines`, etc.) is only included if this DataFrame type *can* support only 1 column+index. See *typeddfs.file_formats.FileFormat.can_write()*.

classmethod from_records(**args*, ***kwargs*) → __qualname__

Convert structured or record ndarray to DataFrame.

Creates a DataFrame object from a structured ndarray, sequence of tuples or dicts, or DataFrame.

Parameters

- **data** (*structured ndarray, sequence of tuples or dicts, or DataFrame*) – Structured input data.
- **index** (*str, list of fields, array-like*) – Field of array to use as the index, alternately a specific set of input labels to use.

- **exclude** (*sequence, default None*) – Columns or fields to exclude.
- **columns** (*sequence, default None*) – Column names to use. If the passed data do not have names associated with them, this argument provides names for the columns. Otherwise this argument indicates the order of the columns in the result (any names not found in the data will become all-NA columns).
- **coerce_float** (*bool, default False*) – Attempt to convert values of non-string, non-numeric objects (like decimal.Decimal) to floating point, useful for SQL result sets.
- **nrows** (*int, default None*) – Number of rows to read if data is an iterator.

Return type DataFrame

See also:

DataFrame.from_dict DataFrame from dict of array-like or dicts.

DataFrame DataFrame object creation using constructor.

Examples

Data can be provided as a structured ndarray:

```
>>> data = np.array([(3, 'a'), (2, 'b'), (1, 'c'), (0, 'd')],
...                 dtype=[('col_1', 'i4'), ('col_2', 'U1')])
>>> pd.DataFrame.from_records(data)
   col_1 col_2
0      3     a
1      2     b
2      1     c
3      0     d
```

Data can be provided as a list of dicts:

```
>>> data = [{'col_1': 3, 'col_2': 'a'},
...         {'col_1': 2, 'col_2': 'b'},
...         {'col_1': 1, 'col_2': 'c'},
...         {'col_1': 0, 'col_2': 'd'}]
>>> pd.DataFrame.from_records(data)
   col_1 col_2
0      3     a
1      2     b
2      1     c
3      0     d
```

Data can be provided as a list of tuples with corresponding columns:

```
>>> data = [(3, 'a'), (2, 'b'), (1, 'c'), (0, 'd')]
>>> pd.DataFrame.from_records(data, columns=['col_1', 'col_2'])
   col_1 col_2
0      3     a
1      2     b
2      1     c
3      0     d
```

classmethod read_file(*path: Union[pathlib.Path, str]*, ***, *file_hash: Optional[bool] = None*, *dir_hash: Optional[bool] = None*, *hex_hash: Optional[str] = None*, *attrs: Optional[bool] = None*, *storage_options: Optional[pandas._typing.StorageOptions] = None*) → __qualname__

Reads from a file (or possibly URL), guessing the format from the filename extension. Delegates to the `read_*` functions of this class.

You can always write and then read back to get the same dataframe. .. code-block:

```
# df is any DataFrame from typeddfs
# path can use any suffix
df.write_file(path))
df.read_file(path)
```

Text files always allow encoding with .gz, .zip, .bz2, or .xz.

Supports:

- .csv, .tsv, or .tab
- .json
- .xml
- .feather
- .parquet or .snappy
- .h5 or .hdf
- .xlsx, .xls, .odf, etc.
- .toml
- .properties
- .ini
- .fxf (fixed-width)
- .flexwf (fixed-but-unspecified-width with an optional delimiter)
- .txt, .lines, or .list

See also:

read_url() write_file()

Parameters

- **path** – Only path-like strings or pathlib objects are supported, not buffers (because we need a filename).
- **file_hash** – Check against a hash file specific to this file (e.g. <path>.sha1)
- **dir_hash** – Check against a per-directory hash file
- **hex_hash** – Check against this hex-encoded hash
- **attrs** – Set dataset attributes/metadata (`pd.DataFrame.attrs`) from a JSON file. If True, uses `typeddfs.df_typing.DfTyping.attrs_suffix`. If a str or Path, uses that file. If None or False, does not set.
- **storage_options** – Passed to Pandas

Returns An instance of this class

classmethod read_url(*url: str*) → __qualname__

Reads from a URL, guessing the format from the filename extension. Delegates to the `read_*` functions of this class.

See also:

read_file()

Returns An instance of this class

write_file(*path: Union[pathlib.Path, str]*, *, *overwrite: bool = True*, *mkdirs: bool = False*, *file_hash: Optional[bool] = None*, *dir_hash: Optional[bool] = None*, *attrs: Optional[bool] = None*, *storage_options: Optional[pandas._typing.StorageOptions] = None*, *atomic: bool = False*) → Optional[str]

Writes to a file, guessing the format from the filename extension. Delegates to the `to_*` functions of this class (e.g. `to_csv`). Only includes file formats that can be read back in with corresponding `to` methods.

Supports, where text formats permit optional .gz, .zip, .bz2, or .xz:

- .csv, .tsv, or .tab
- .json
- .feather
- .fwf (fixed-width)
- .flexwf (columns aligned but using a delimiter)
- .parquet or .snappy
- .h5, .hdf, or .hdf5
- .xlsx, .xls, and other variants for Excel
- .odt and .ods (OpenOffice)
- .xml
- .toml
- .ini
- .properties
- .pkl and .pickle
- .txt, .lines, or .list; see `to_lines()` and `read_lines()`

See also:

read_file()

Parameters

- **path** – Only path-like strings or pathlib objects are supported, not buffers (because we need a filename).
- **overwrite** – If False, complain if the file already exists
- **mkdirs** – Make the directory and parents if they do not exist
- **file_hash** – Write a hash for this file. The filename will be path+"."+algorithm. If None, chooses according to `self.get_typing().io.hash_file`.

- **dir_hash** – Append a hash for this file into a list. The filename will be the directory name suffixed by the algorithm; (i.e. path.parent/(path.parent.name+"."+algorithm)). If None, chooses according to `self.get_typing().io.hash_dir`.
- **attrs** – Write dataset attributes/metadata (`pd.DataFrame.attrs`) to a JSON file. uses `typeddfs.df_typing.DfTyping.attrs_suffix`. If None, chooses according to `self.get_typing().io.use_attrs`.
- **storage_options** – Passed to Pandas
- **atomic** – Write to a temporary file, then renames

Returns Whatever the corresponding method on `pd.to_*` returns. This is usually either str or None

Raises

- *InvalidDfError* – If the DataFrame is not valid for this type
- **ValueError** – If the type of a column or index name is non-str

typeddfs.base_dfs

Defines the superclasses of the types `TypedDf` and `UntypedDf`.

Module Contents

class typeddfs.base_dfs.**BaseDf**(*data=None*, *index=None*, *columns=None*, *dtype=None*, *copy=False*)

An abstract DataFrame type that has a way to convert and de-convert. A subclass of *typeddfs.abs_dfs.AbsDf*, it has methods *convert()* and `vanilla()`. but no implementation or enforcement of typing.

__getitem__(*item*)

Finds an index level or column, returning the Series, DataFrame, or value. Note that typeddfs forbids duplicate column names, as well as column names and index levels sharing names.

classmethod convert(*df: pandas.DataFrame*) → __qualname__

Converts a vanilla Pandas DataFrame to cls.

Note: The argument `df` will have its `__class__` changed to `cls` but will otherwise be unaffected.

Returns A copy

classmethod of(*df*, **args*, *keys: Optional[Iterable[str]] = None*, ***kwargs*) → __qualname__

Construct or convert a DataFrame, returning this type. Delegates to *convert()* for DataFrames, or tries first constructing a DataFrame by calling `pd.DataFrame(df)`. If `df` is a list (`Iterable`) of DataFrames, will call `pd.concat` on them; for this, `ignore_index=True` is passed. If the list is empty, will return `new_df()`.

May be overridden to accept more types, such as a string for database lookup. For example, `Customers.of("john")` could return a DataFrame for a database customer, or return the result of `Customers.convert(...)` if a DataFrame instance is provided. You may add and process keyword arguments, but keyword args for `pd.DataFrame.__init__` should be passed along to that constructor.

Parameters

- **df** – A DataFrame, list of DataFrames, or something to be passed to `pd.DataFrame`.

- **keys** – Labels for the DataFrames (if passed a sequence of them) to use as attr keys; if None, attrs will be empty ({}) if concatenating
- **kwargs** – Passed to `pd.DataFrame.__init__`; can be handled directly by this method for specialized construction, database lookup, etc.

Returns A new DataFrame; see *convert()* for more info.

retype() → __qualname__

Calls `self.__class__.convert` on this DataFrame. This is useful to call at the end of a chain of DataFrame functions, where the type is preserved but the DataFrame may no longer be valid under this type's rules. This can occur because, for performance, typeddfs does not call `convert` on most calls.

Examples

- `df = MyDf(data).apply(my_fn, axis=1).retype() # make sure it's still valid`
- `df = MyDf(data).groupby(...).retype() # we maybe changed the index; fix it`

Returns A copy

typeddfs.builders

Defines a builder pattern for `TypedDf`.

Module Contents

class typeddfs.builders.**AffinityMatrixDfBuilder**(*name: str*, *doc: Optional[str] = None*)

A builder pattern for *typeddfs.matrix_dfs.AffinityMatrixDf*.

Constructs a new builder.

Parameters

- **name** – The name of the resulting class
- **doc** – The docstring of the resulting class

Raises **TypeError** – If `name` or `doc` non-string

build() → Type[*typeddfs.matrix_dfs.AffinityMatrixDf*]

Builds this type.

Returns A newly created subclass of *typeddfs.matrix_dfs.AffinityMatrixDf*.

Raises

- ***typeddfs.df_errors.ClashError*** – If there is a contradiction in the specification
- ***typeddfs.df_errors.FormatInsecureError*** – If `hash()` set an insecure hash format and `secure()` was set.

Note: Copies, so this builder can be used to create more types without interference.

class typeddfs.builders.**MatrixDfBuilder**(*name: str*, *doc: Optional[str] = None*)

A builder pattern for *typeddfs.matrix_dfs.MatrixDf*.

Constructs a new builder.

Parameters

- **name** – The name of the resulting class
- **doc** – The docstring of the resulting class

Raises **TypeError** – If name or doc non-string

_check_final() → None

build() → Type[*typeddfs.matrix_dfs.MatrixDf*]

Builds this type.

Returns A newly created subclass of *typeddfs.matrix_dfs.MatrixDf*.

Raises

- *ClashError* – If there is a contradiction in the specification
- *FormatInsecureError* – If hash() set an insecure hash format and secure() was set.

Note: Copies, so this builder can be used to create more types without interference.

Raises *DfTypeConstructionError* – for some errors

dtype(*dt: Type[Any]*) → __qualname__

Sets the type of value for all matrix elements. This should almost certainly be a numeric type, and it must be ordered.

Returns This builder for chaining

class typeddfs.builders.**TypedDfBuilder**(*name: str*, *doc: Optional[str] = None*)

A builder pattern for *typeddfs.typed_dfs.TypedDf*.

Example

```
TypedDfBuilder.typed().require("name").build()
```

Constructs a new builder.

Parameters

- **name** – The name of the resulting class
- **doc** – The docstring of the resulting class

Raises **TypeError** – If name or doc non-string

_check(*names: Sequence[str]*) → None

_check_final() → None

Final method in the chain. Creates a new subclass of TypedDf.

Returns The new class

Raises *typeddfs.df_errors.ClashError* – If there is a contradiction in the specification

build() → Type[*typeddfs.typed_dfs.TypedDf*]

Builds this type.

Returns A newly created subclass of `typeddfs.typed_dfs.TypedDf`.

Raises `DfTypeConstructionError` – If there is a contradiction in the specification

---

Note: Copies, so this builder can be used to create more types without interference.

---

drop(**names: str*) → __qualname__

Adds columns (and index names) that should be automatically dropped.

Parameters **names** – Varargs list of names

Returns This builder for chaining

require(**names: str*, *dtype: Optional[Type] = None*, *index: bool = False*) → __qualname__

Requires column(s) or index name(s). DataFrames will fail if they are missing any of these.

Parameters

- **names** – A varargs list of columns or index names
- **dtype** – An automatically applied transformation of the column values using `.astype`
- **index** – If True, put these in the index

Returns This builder for chaining

Raises `typeddfs.df_errors.ClashError` – If a name was already added or is forbidden

reserve(**names: str*, *dtype: Optional[Type] = None*, *index: bool = False*) → __qualname__

Reserves column(s) or index name(s) for optional inclusion. A reserved column will be accepted even if `strict` is set. A reserved index will be accepted even if `strict` is set; additionally, it will be automatically moved from the list of columns to the list of index names.

Parameters

- **names** – A varargs list of columns or index names
- **dtype** – An automatically applied transformation of the column values using `.astype`
- **index** – If True, put these in the index

Returns This builder for chaining

Raises `typeddfs.df_errors.ClashError` – If a name was already added or is forbidden

series_names(*index: Union[None, bool, str] = False*, *columns: Union[None, bool, str] = False*) → __qualname__

Sets `pd.DataFrame.index.name` and/or `pd.DataFrame.columns.name`. Valid values are `False` to not set (default), `None` to set to `None`, or a string to set to.

Returns This builder for chaining

strict(*index: bool = True*, *cols: bool = True*) → __qualname__

Disallows any columns or index names not in the lists of reserved/required.

Parameters

- **index** – Disallow additional names in the index
- **cols** – Disallow additional columns

Returns This builder for chaining

typeddfs.datasets

Near-replica of example from the readme.

Module Contents

class typeddfs.datasets.**ExampleDfs**

DataFrames derived from Seaborn and other sources.

anagrams

anscombe

attention

brain_networks

car_crashes

diamonds

dots

exercise

flights

fmri

gammas

geyser

iris

mpg

penguins

planets

taxis

tips

titanic

class typeddfs.datasets.**LazyDf**(*name: str*, *source: str*, *clazz: Type[T]*, *_df: Optional[T]*)

A *typeddfs.abs_dfs.AbsDf* that is lazily loaded from a source. Create normally via *from_source()*. Create with *from_df()* to wrap an extant DataFrame into a LazyDataFrame.

Example

```
lazy = LazyDataFrame.from_source("https://google.com/dataframe.csv")
```

property clazz → Type[T]

property df → T

classmethod from_df(*df: X*, *name: Optional[str] = None*) → *LazyDf*[X]

classmethod from_source(*source: str*, *clazz: Type[S] = PlainTypedDf*, *name: Optional[str] = None*) → *LazyDf*[S]

property name → str

typeddfs.df_errors

Exceptions used by typeddfs.

Module Contents

exception typeddfs.df_errors.**ClashError**(**args*, *keys: Optional[AbstractSet[str]] = None*)

Duplicate columns or other keys were added.

keys

The clashing name(s)

Initialize self. See help(type(self)) for accurate signature.

exception typeddfs.df_errors.**DfTypeConstructionError**

An inconsistency prevents creating the DataFrame type.

Initialize self. See help(type(self)) for accurate signature.

exception typeddfs.df_errors.**FilenameSuffixError**(**args*, *key: Optional[str] = None*, *filename: Optional[str] = None*)

A filename extension was not recognized.

key

The unrecognized suffix

filename

The bad filename

Initialize self. See help(type(self)) for accurate signature.

exception typeddfs.df_errors.**FormatDiscouragedError**(**args*, *key: Optional[str] = None*)

A requested format is not recommended.

key

The problematic format name

Initialize self. See help(type(self)) for accurate signature.

exception typeddfs.df_errors.**FormatInsecureError**(**args*, *key: Optional[str] = None*)

A requested format is less secure than required or requested.

key

The problematic format name

Initialize self. See help(type(self)) for accurate signature.

exception typeddfs.df_errors.**HashAlgorithmMissingError**(**args*, *key: Optional[str] = None*)

The hash algorithm was not found in `hashlib`.

key

The missing hash algorithm

Initialize self. See help(type(self)) for accurate signature.

exception typeddfs.df_errors.**HashContradictsExistingError**(**args*, *key: Optional[str] = None*, *original: Optional[str] = None*, *new: Optional[str] = None*)

A hash for the filename already exists in the directory hash list, but they differ.

key

The filename (excluding parents)

original

Hex hash found listed for the file

new

Hex hash that was to be written

filename

The filename of the listed file

Initialize self. See help(type(self)) for accurate signature.

exception typeddfs.df_errors.**HashDidNotValidateError**(**args*, *actual: Optional[str] = None*, *expected: Optional[str] = None*)

The hashes did not validate (expected != actual).

actual

The actual hex-encoded hash

expected

The expected hex-encoded hash

Initialize self. See help(type(self)) for accurate signature.

exception typeddfs.df_errors.**HashEntryExistsError**(**args*, *key: Optional[str] = None*)

The file is already listed in the hash dir, and it cannot be overwritten.

key

The existing hash dir path

Initialize self. See help(type(self)) for accurate signature.

exception typeddfs.df_errors.**HashError**

Something went wrong with hash file writing or reading.

Initialize self. See help(type(self)) for accurate signature.

exception typeddfs.df_errors.**HashExistsError**(**args*, *key: Optional[str] = None*, *original: Optional[str] = None*, *new: Optional[str] = None*)

A hash for the filename already exists in the directory hash list.

key

The filename (excluding parents)

original

Hex hash found listed for the file

new

Hex hash that was to be written

filename

The filename of the listed file

Initialize self. See help(type(self)) for accurate signature.

exception typeddfs.df_errors.**HashFileExistsError**(**args*, *key: Optional[str] = None*)

The hash file already exists and cannot be overwritten.

key

The existing hash file path or filename

Initialize self. See help(type(self)) for accurate signature.

exception typeddfs.df_errors.**HashFileInvalidError**(**args*, *key: Union[None, pathlib.PurePath, str] = None*)

The hash file could not be parsed.

key

The path to the hash file

Initialize self. See help(type(self)) for accurate signature.

exception typeddfs.df_errors.**HashFileMissingError**(**args*, *key: Optional[str] = None*)

The hash file does not exist.

key

The path or filename of the file corresponding to the expected hash file(s)

Initialize self. See help(type(self)) for accurate signature.

exception typeddfs.df_errors.**HashFilenameMissingError**(**args*, *key: Optional[str] = None*)

The filename was not found listed in the hash file.

key

The filename

Initialize self. See help(type(self)) for accurate signature.

exception typeddfs.df_errors.**HashVerificationError**

Something went wrong when validating a hash.

Initialize self. See help(type(self)) for accurate signature.

exception typeddfs.df_errors.**HashWriteError**

Something went wrong when writing a hash file.

Initialize self. See help(type(self)) for accurate signature.

exception typeddfs.df_errors.**InvalidDfError**

A general typing failure of typeddfs.

Initialize self. See help(type(self)) for accurate signature.

exception typeddfs.df_errors.**LengthMismatchError**(**args*, *key: Optional[str] = None*, *lengths: AbstractSet[int]*)

The lengths of at least two lists do not match.

key

The key used for lookup

lengths

The lengths

Initialize self. See help(type(self)) for accurate signature.

exception typeddfs.df_errors.**MissingColumnError**(**args*, *key: Optional[str] = None*)

A required column is missing.

key

The name of the missing column

Initialize self. See help(type(self)) for accurate signature.

exception typeddfs.df_errors.**MultipleHashFilenamesError**(**args*, *key: Optional[str] = None*)

There are multiple filenames listed in the hash file where only 1 was expected.

key

The filename with duplicate entries

Initialize self. See help(type(self)) for accurate signature.

exception typeddfs.df_errors.**NoValueError**(**args*, *key: Optional[str] = None*)

No value because the collection is empty.

key

The key used for lookup

Initialize self. See help(type(self)) for accurate signature.

exception typeddfs.df_errors.**NonStrColumnError**

A column name is not a string.

Initialize self. See help(type(self)) for accurate signature.

exception typeddfs.df_errors.**NotSingleColumnError**

A DataFrame needs to contain exactly 1 column.

Initialize self. See help(type(self)) for accurate signature.

exception typeddfs.df_errors.**PathNotRelativeError**(**args*, *key: Optional[str] = None*)

The filename is not relative to the hash dir.

key

The filename

Initialize self. See help(type(self)) for accurate signature.

exception typeddfs.df_errors.**ReadPermissionsError**(**args*, *key: Optional[str] = None*)

Couldn't read from a file.

Initialize self. See help(type(self)) for accurate signature.

exception typeddfs.df_errors.**RowColumnMismatchError**(**args*, *rows: Optional[Sequence[str]] = None*, *columns: Optional[Sequence[str]] = None*)

The row and column names differ.

rows

The row names, in order

columns

The column names, in order

Initialize self. See help(type(self)) for accurate signature.

exception typeddfs.df_errors.**UnexpectedColumnError**(**args*, *key: Optional[str] = None*)

An extra/unrecognized column is present.

key

The name of the unexpected column

Initialize self. See help(type(self)) for accurate signature.

exception typeddfs.df_errors.**UnexpectedIndexNameError**(**args*, *key: Optional[str] = None*)

An extra/unrecognized index level is present.

key

The name of the unexpected index level

Initialize self. See help(type(self)) for accurate signature.

exception typeddfs.df_errors.**UnsupportedOperationError**

Something could not be performed, in general.

Initialize self. See help(type(self)) for accurate signature.

exception typeddfs.df_errors.**ValueNotUniqueError**(**args*, *key: Optional[str] = None*, *values: Optional[AbstractSet[str]] = None*)

There is more than 1 unique value.

key

The key used for lookup

values

The set of values

Initialize self. See help(type(self)) for accurate signature.

exception typeddfs.df_errors.**VerificationFailedError**(**args*, *key: Optional[str] = None*)

A custom typing verification failed.

key

The key name of the verification that failed

Initialize self. See help(type(self)) for accurate signature.

exception typeddfs.df_errors.**WritePermissionsError**(**args*, *key: Optional[str] = None*)

Couldn't write to a file.

Initialize self. See help(type(self)) for accurate signature.

typeddfs.df_typing

Information about how DataFrame subclasses should be handled.

Module Contents

typeddfs.df_typing.**FINAL_DF_TYPING**

typeddfs.df_typing.**FINAL_IO_TYPING**

class typeddfs.df_typing.**DfTyping**

Contains all information about how to type a DataFrame subclass.

_auto_dtypes :Optional[Mapping[str, Type[Any]]]

_column_series_name :Union[bool, None, str]

_columns_to_drop :Optional[Set[str]]

_index_series_name :Union[bool, None, str]

_io_typing :IoTyping

_more_columns_allowed :bool = True

_more_index_names_allowed :bool = True

_order_dclass :bool = True

_post_processing :Optional[Callable[[T], Optional[T]]]

_required_columns :Optional[Sequence[str]]

_required_index_names :Optional[Sequence[str]]

_reserved_columns :Optional[Sequence[str]]

_reserved_index_names :Optional[Sequence[str]]

_value_dtype :Optional[Type[Any]]

_verifications :Optional[Sequence[Callable[[T], Union[None, bool, str]]]]

property auto_dtypes → Mapping[str, Type[Any]]

A mapping from column/index names to the expected dtype. These are used via `pd.Series.as_type` for automatic conversion. An error will be raised if a `as_type` call fails. Note that Pandas frequently just does not perform the conversion, rather than raising an error. The keys should be contained in `known_names`, but this is not strictly required.

property column_series_name → Union[bool, None, str]

Intelligently returns `df.columns.name`. Returns a value that will be forced into `df.columns.name` on calling `convert`. If `None`, will set `df.columns.name = None`. If `False`, will not set. (`True` is treated the same as `None`.)

property columns_to_drop → Set[str]

Returns the list of columns that are automatically dropped by `convert`. This does NOT include "level_0" and "index, which are ALWAYS dropped.

copy(***kwargs*) → *DfTyping*

property index_series_name → Union[bool, None, str]

Intelligently returns `df.index.name`. Returns a value that will be forced into `df.index.name` on calling `convert`, *only if* the DataFrame is multi-index. If `None`, will set `df.index.name = None` if `df.index.names != [None]`. If `False`, will not set. (`True` is treated the same as `None`.)

property io → *IoTyping*

property is_strict → bool

Returns True if this allows unspecified index levels **or** columns.

property known_column_names → Sequence[str]

Returns all columns that are required or reserved. The sort order positions required columns first.

property known_index_names → Sequence[str]

Returns all index levels that are required or reserved. The sort order positions required columns first.

property known_names → Sequence[str]

Returns all index and column names that are required or reserved. The sort order is: required index, reserved index, required columns, reserved columns.

property more_columns_allowed → bool

Returns whether the DataFrame allows columns that are not reserved or required.

property more_indices_allowed → bool

Returns whether the DataFrame allows index levels that are neither reserved nor required.

property order_dataclass → bool

Whether the corresponding dataclass can be sorted (has `__lt__`).

property post_processing → Optional[Callable[[T], Optional[T]]]

A function to be called at the final stage of `convert`. It is called immediately before `verifications` are checked. The function takes a copy of the input `BaseDf` and returns a new copy.

Note: Although a copy is passed as input, the function should not modify it. Technically, doing so will cause problems only if the DataFrame's internal values are modified. The value passed is a *shallow* copy (see `pd.DataFrame.copy`).

---

property required_columns → Sequence[str]

Returns the list of required column names.

property required_index_names → Sequence[str]

Returns the list of required column names.

property required_names → Sequence[str]

Returns all index and column names that are required. The sort order is: required index, required columns.

property reserved_columns → Sequence[str]

Returns the list of reserved (optional) column names.

property reserved_index_names → Sequence[str]

Returns the list of reserved (optional) index levels.

property reserved_names → Sequence[str]

Returns all index and column names that are **not** required. The sort order is: reserved index, reserved columns.

property value_dtype → Optional[Type[Any]]

A type for "values" in a simple DataFrame. Typically numeric.

property verifications → Sequence[Callable[[T], Union[None, bool, str]]]

Additional requirements for the DataFrame to be conformant.

Returns A sequence of conditions that map the DF to None or True if the condition passes, or False or the string of an error message if it fails

class typeddfs.df_typing.**IoTyping**

Abstract base class for generic types.

A generic type is typically declared by inheriting from this class parameterized with one or more type variables. For example, a generic mapping type might be defined as:

```
class Mapping(Generic[KT, VT]):
    def __getitem__(self, key: KT) -> VT:
        ...
    # Etc.
```

This class can then be used as follows:

```
def lookup_name(mapping: Mapping[KT, VT], key: KT, default: VT) -> VT:
    try:
        return mapping[key]
    except KeyError:
        return default
```

_attrs_json_kwargs :Optional[Mapping[str, Any]]

_attrs_suffix :str = .attrs.json

_custom_readers :Optional[Mapping[str, Callable[[pathlib.Path], pandas.DataFrame]]]

_custom_writers :Optional[Mapping[str, Callable[[pandas.DataFrame, pathlib.Path], None]]]

_hash_alg :Optional[str] = sha256

_hdf_key :str = df

_read_kwargs :Optional[Mapping[typeddfs.file_formats.FileFormat, Mapping[str, Any]]]

_recommended :bool = False

_remap_suffixes :Optional[Mapping[str, typeddfs.file_formats.FileFormat]]

_remapped_read_kwargs :Optional[Mapping[str, Any]]

_remapped_write_kwargs :Optional[Mapping[str, Any]]

_save_hash_dir :bool = False

_save_hash_file :bool = False

_secure :bool = False

_text_encoding :str = utf-8

_use_attrs :bool = False

_write_kwargs :Optional[Mapping[typeddfs.file_formats.FileFormat, Mapping[str, Any]]]

property attrs_json_kwargs → Mapping[str, Any]

Keyword arguments for `typeddfs.json_utils.JsonUtils.encoder`. Used when writing attrs.

property attrs_suffix → str

File filename suffix detailing where to save/load per-DataFrame "attrs" (metadata). Will be appended to the DataFrame filename.

copy(***kwargs*) → *IoTyping*

property custom_readers → Mapping[str, Callable[[pathlib.Path], pandas.DataFrame]]

Mapping from filename suffixes (module compression) to custom reading methods.

property custom_writers → Mapping[str, Callable[[pandas.DataFrame, pathlib.Path], None]]

Mapping from filename suffixes (module compression) to custom reading methods.

property dir_hash → bool

Whether to save (append) to per-directory hash files by default. Specifically, in `typeddfs.abs_df.AbsDf.write_file()`.

property file_hash → bool

Whether to save per-file hash files by default. Specifically, in `typeddfs.abs_df.AbsDf.write_file()`.

property flexwf_sep → str

The delimiter used when reading "flex-width" format.

> **Caution:** Only checks the read keyword arguments, not write

property hash_algorithm → Optional[str]

The hash algorithm used for checksums.

property hdf_key → str

The default key used in `typeddfs.abs_df.AbsDf.to_hdf()`. The key is also used in `typeddfs.abs_df.AbsDf.read_hdf.()`

property is_text_encoding_utf → bool

property read_kwargs → Mapping[*typeddfs.file_formats.FileFormat*, Mapping[str, Any]]

Passes kwargs into read functions from `read_file`. These are keyword arguments that are automatically added into specific `read_` methods when called by `read_file`.

Note: This should rarely be needed

---

property read_suffix_kwargs → Mapping[str, Mapping[str, Any]]

Per-suffix kwargs into read functions from `read_file`. Modulo compression (e.g. .tsv is equivalent to .tsv.gz).

property recommended → bool

Whether to forbid discouraged formats like fixed-width and HDF5. Excludes all insecure formats.

property remap_suffixes → Mapping[str, *typeddfs.file_formats.FileFormat*]

Returns filename formats that have been re-mapped to file formats. These are used in `read_file` and `write_file`.

Note: This should rarely be needed. An exception might be `.txt` to tsv rather than lines; Excel uses this.

---

property secure → bool

Whether to forbid insecure operations and formats.

property text_encoding → str

Can be an exact encoding like utf-8, "platform", "utf8(bom)" or "utf16(bom)". See the docs in `TypedDfs.typed().encoding` for details.

property toml_aot → str

The name of the Array of Tables (AoT) used when reading TOML.

Caution: Only checks the read keyword arguments, not write

property use_attrs → bool

Whether to read and write `pd.DataFrame.attrs` when passing `attrs=None`.

property write_kwargs → Mapping[*typeddfs.file_formats.FileFormat*, Mapping[str, Any]]

Passes kwargs into write functions from `to_file`. These are keyword arguments that are automatically added into specific `to_` methods when called by `write_file`.

Note: This should rarely be needed

---

property write_suffix_kwargs → Mapping[str, Mapping[str, Any]]

Per-suffix kwargs into read functions from `write_file`. Modulo compression (e.g. .tsv is equivalent to .tsv.gz).

typeddfs.example

Near-replica of example from the readme.

Module Contents

typeddfs.example.**run**() → None

Runs an example usage of typeddfs.

typeddfs.file_formats

File formats for reading/writing to/from DFs.

Module Contents

class typeddfs.file_formats.**BaseCompression**

base :pathlib.Path

compression :CompressionFormat

class typeddfs.file_formats.**BaseFormatCompression**

base :pathlib.Path

compression :CompressionFormat

format :Optional[FileFormat]

class typeddfs.file_formats.**CompressionFormat**

A compression scheme or no compression: gzip, zip, bz2, xz, and none. These are the formats supported by Pandas for read and write. Provides a few useful functions for calling code.

Examples

- `CompressionFormat.strip("my_file.csv.gz") # Path("my_file.csv")`
- `CompressionFormat.from_path("myfile.csv") # CompressionFormat.none`

bz2 = []

gz = []

none = []

xz = []

zip = []

zstd = []

classmethod all_suffixes() → Set[str]

Returns all suffixes for all compression formats.

classmethod from_path(*path: typeddfs.utils._utils.PathLike*) → *CompressionFormat*

Returns the compression scheme from a path suffix.

classmethod from_suffix(*suffix: str*) → *CompressionFormat*

Returns the recognized compression scheme from a suffix.

property full_name → str

Returns a more-complete name of this format. For example, “gzip” “bzip2”, “xz”, and “none”.

property is_compressed → bool

Shorthand for `fmt is not CompressionFormat.none`.

classmethod list() → Set[*CompressionFormat*]

Returns the set of CompressionFormats. Works with static type analysis.

classmethod list_non_empty() → Set[*CompressionFormat*]

Returns the set of CompressionFormats, except for `none`. Works with static type analysis.

property name_or_none → Optional[str]

Returns the name, or `None` if it is not compressed.

classmethod of(*t: Union[str,* CompressionFormat*]*) → *CompressionFormat*

Returns a FileFormat from a name (e.g. "gz" or "gzip"). Case-insensitive.

Example

```
CompressionFormat.of("gzip").suffix # ".gz"
```

property pandas_value → Optional[str]

Returns the value that should be passed to Pandas as `compression`.

classmethod split(*path: typeddfs.utils._utils.PathLike*) → *BaseCompression*

classmethod strip_suffix(*path: typeddfs.utils._utils.PathLike*) → pathlib.Path

Returns a path with any recognized compression suffix (e.g. ".gz") stripped.

property suffix → str

Returns the single Pandas-recognized suffix for this format. This is just "" for CompressionFormat.none.

class typeddfs.file_formats.**FileFormat**

A computer-readable format for reading **and** writing of DataFrames in typeddfs. This includes CSV, Parquet, ODT, etc. Some formats also include compressed variants. E.g. a ".csg.gz" will map to `FileFormat.csv`. This is used internally by `typeddfs.abs_df.read_file()` and `typeddfs.abs_df.write_file()`, but it may be useful to calling code directly.

Examples

- `FileFormat.from_path("my_file.csv.gz").is_text() # True`
- `FileFormat.from_path("my_file.csv.gz").can_read() # always True`
- `FileFormat.from_path("my_file.xlsx").can_read() # true if required package is installed`

csv = []

feather = []

flexwf = []

fwf = []

hdf = []

ini = []

json = []

lines = []

ods = []

parquet = []

pickle = []

properties = []

toml = []

tsv = []

xls = []

xlsb = []

xlsx = []

xml = []

classmethod all_readable() → Set[*FileFormat*]

Returns all formats that can be read on this system. Note that the result may depend on whether supporting packages are installed. Includes insecure and discouraged formats.

classmethod all_writable() → Set[*FileFormat*]

Returns all formats that can be written to on this system. Note that the result may depend on whether supporting packages are installed. Includes insecure and discouraged formats.

property can_always_read → bool

Returns whether this format can be read as long as typeddfs is installed. In other words, regardless of any optional packages.

property can_always_write → bool

Returns whether this format can be written to as long as typeddfs is installed. In other words, regardless of any optional packages.

property can_read → bool

Returns whether this format can be read. Note that the result may depend on whether supporting packages are installed.

property can_write → bool

Returns whether this format can be written. Note that the result may depend on whether supporting packages are installed.

compressed_variants(*suffix: str*) → Set[str]

Returns all allowed suffixes.

Example

```
```

FileFormat.json.compressed_variants(".json") # {".json", ".json.gz", ".json.zip", ...}

classmethod from_path(*path: typeddfs.utils._utils.PathLike*, *, *format_map: Optional[Mapping[str, Union[*FileFormat*, str]]] = None*) → *FileFormat*

Guesses a FileFormat from a filename.

See also:

from_suffix()

Parameters

- **path** – A string or `pathlib.Path` to a file.
- **format_map** – A mapping from suffixes to formats; if `None`, uses *suffix_map()*.

Raises ***typeddfs.df_errors.FilenameSuffixError*** – If not found

classmethod from_path_or_none(*path: typeddfs.utils._utils.PathLike*, *, *format_map: Optional[Mapping[str, Union[*FileFormat*, str]]] = None*) → Optional[*FileFormat*]

Same as *from_path()*, but returns None if not found.

classmethod from_suffix(*suffix: str*, *, *format_map: Optional[Mapping[str, Union[*FileFormat*, str]]] = None*) → *FileFormat*

Returns the FileFormat corresponding to a filename suffix.

See also:

from_path()

Parameters

- **suffix** – E.g. ".csv.gz" or ".feather"
- **format_map** – A mapping from suffixes to formats; if `None`, uses *suffix_map()*.

Raises ***typeddfs.df_errors.FilenameSuffixError*** – If not found

classmethod from_suffix_or_none(*suffix: str*, *, *format_map: Optional[Mapping[str, Union[*FileFormat*, str]]] = None*) → Optional[*FileFormat*]

Same as *from_suffix()*, but returns None if not found.

property is_binary → bool

Returns whether this format is text-encoded. Note that this does *not* consider whether the file is compressed.

property is_recommended → bool

Returns whether the format is good. Includes CSV, TSV, Parquet, etc. Excludes all insecure formats along with fixed-width, INI, properties, TOML, and HDF5.

property is_secure → bool

Returns whether the format does NOT have serious security issues. These issues only apply to reading files, not writing. Excel formats that support Macros are not considered secure. This includes .xlsm, .xltm, and .xls. These can simply be replaced with xlsx. Note that .xml is treated as secure: Although some parsers are subject to entity expansion attacks, good ones are not.

property is_text → bool

Returns whether this format is text-encoded. Note that this does *not* consider whether the file is compressed.

classmethod list() → Set[*FileFormat*]

Returns the set of FileFormats. Works with static type analysis.

matches(*, *supported: bool*, *secure: bool*, *recommended: bool*) → bool

Returns whether this format meets some requirements.

Parameters

- **supported** – *can_read* and *can_write* are True
- **secure** – *is_secure* is True
- **recommended** – *is_recommended* is True

classmethod of(*t: Union[str,* FileFormat*]*) → *FileFormat*

Returns a FileFormat from an exact name (e.g. "csv").

See also:

from_suffix() from_path()

classmethod split(*path: typeddfs.utils._utils.PathLike*, *, *format_map: Optional[Mapping[str, Union[*FileFormat*, str]]] = None*) → *BaseFormatCompression*

Splits a path into the base path, format, and compression.

See also:

split_or_none() strip() from_path()

Raises ***FilenameSuffixError*** – If the suffix is not found

Returns A 3-tuple of (base base excluding suffixes, file format, compression format)

classmethod split_or_none(*path: typeddfs.utils._utils.PathLike*, *, *format_map: Optional[Mapping[str, Union[*FileFormat*, str]]] = None*) → *BaseFormatCompression*

Splits a path into the base path, format, and compression.

See also:

split() strip() from_path()

Returns A 3-tuple of (base base excluding suffixes, file format, compression format)

classmethod strip(*path: typeddfs.utils._utils.PathLike*, *, *format_map: Optional[Mapping[str, Union[*FileFormat*, str]]] = None*) → pathlib.Path

Strips a recognized, optionally compressed, suffix from `path`.

See also:

split()

Example

```
FileFormat.strip("abc/xyz.csv.gz")  # Path("abc") / "xyz"
```

classmethod suffix_map() → MutableMapping[str, *FileFormat*]

Returns a mapping from all suffixes to their respective formats. See *suffixes()*.

property suffixes → Set[str]

Returns the suffixes that are tied to this format. These will not overlap with the suffixes for any other format. For example, .txt is for `FileFormat.lines`, although it could be treated as tab- or space-separated.

property supports_encoding → bool

Returns whether this format supports a text encoding of some sort. This may not correspond to an `encoding=` parameter, and the format may be binary. For example, XLS and XML support encodings.

typeddfs.frozen_types

Hashable and ordered collections.

Module Contents

class typeddfs.frozen_types.**FrozeDict**(*dct: Mapping[K, V]*)

An immutable dictionary/mapping. Hashable and ordered.

EMPTY :FrozeDict

__contains__(*item: K*) → bool

__eq__(*other:* FrozeDict*[K, V]*) → bool

Return self==value.

__getitem__(*item: K*) → T

__hash__() → int

Return hash(self).

__iter__()

__len__() → int

__lt__(*other: Mapping[K, V]*)

Compares this dict to another, with partial ordering.

The algorithm is:

1. Sort `self` and `other` by keys
2. If `sorted_self < sorted_other`, return `False`
3. If the reverse is true (`sorted_other < sorted_self`), return `True`
4. (The keys are now known to be the same.) For each key, in order: If `self[key] < other[key]`, return `True`
5. Return `False`

__make_other(*other: Union[*FrozeDict*[K, V], Mapping[K, V]]*) → Dict[K, V]

__repr__() → str

Return repr(self).

__str__() → str

Return str(self).

get(*key: K*, *default: Optional[V] = None*) → Optional[V]

D.get(k[,d]) -> D[k] if k in D, else d. d defaults to None.

property is_empty → bool

items() → AbstractSet[tuple[K, V]]

D.items() -> a set-like object providing a view on D's items

keys() → AbstractSet[K]

D.keys() -> a set-like object providing a view on D's keys

property length → int

req(*key: K*, *default: Optional[V] = None*) → V

Returns the value corresponding to `key`. Short for "require". Falls back to `default` if `default` is not None and `key` is not in this dict.

Raise: KeyError: If `key` is not in this dict and `default` is `None`

to_dict() → MutableMapping[K, V]

values() → ValuesView[V]

D.values() -> an object providing a view on D's values

class typeddfs.frozen_types.**FrozeList**(*lst: Sequence[T]*)

An immutable list. Hashable and ordered.

EMPTY :FrozeList

__eq__(*other: Union[*FrozeList*[T], Sequence[T]]*) → bool

Return self==value.

__getitem__(*item: int*)

__hash__() → int

Return hash(self).

__iter__() → Iterator[T]

__len__() → int

__lt__(*other: Union[*FrozeList*[T], Sequence[T]]*)

Return self<value.

__make_other(*other: Union[*FrozeList*[T], Sequence[T]]*) → List[T]

__repr__() → str

Return repr(self).

__str__() → str

Return str(self).

get(*item: T*, *default: Optional[T] = None*) → Optional[T]

property is_empty → bool

property length → int

req(*item: T*, *default: Optional[T] = None*) → T

Returns the requested list item, falling back to a default. Short for "require".

Raises KeyError – If `item` is not in this list and `default` is `None`

to_list() → List[T]

class typeddfs.frozen_types.**FrozeSet**(*lst: AbstractSet[T]*)

An immutable set. Hashable and ordered. This is almost identical to `typing.FrozenSet`, but it's behavior was made equivalent to those of FrozeDict and FrozeList.

EMPTY :FrozeSet

__contains__(*x: T*) → bool

__eq__(*other:* FrozeSet*[T]*) → bool

Return self==value.

__getitem__(*item: T*) → T

__hash__() → int

Return hash(self).

__iter__() → Iterator[T]

__len__() → int

__lt__(*other: Union[*FrozeSet*[T], AbstractSet[T]]*)

Compares `self` and `other` for partial ordering. Sorts `self` and `other`, then compares the two sorted sets.

Approximately:: return list(sorted(self)) < list(sorted(other))

__make_other(*other: Union[*FrozeSet*[T], AbstractSet[T]]*) → Set[T]

__repr__() → str

Return repr(self).

__str__() → str

Return str(self).

get(*item: T*, *default: Optional[T] = None*) → Optional[T]

property is_empty → bool

property length → int

req(*item: T*, *default: Optional[T] = None*) → T

Returns `item` if it is in this set. Short for "require". Falls back to `default` if `default` is not `None`.

Raises KeyError – If `item` is not in this set and `default` is `None`

to_frozenset() → AbstractSet[T]

to_set() → AbstractSet[T]

typeddfs.matrix_dfs

DataFrames that are essentially n-by-m matrices.

Module Contents

class typeddfs.matrix_dfs.**AffinityMatrixDf**(*data=None*, *index=None*, *columns=None*, *dtype=None*, *copy=False*)

A similarity or distance matrix. The rows and columns must match, and only 1 index is allowed.

__repr__() → str

Return repr(self).

__str__() → str

Return str(self).

classmethod _check(*df:* typeddfs.base_dfs.BaseDf)

Should raise an *typeddfs.df_errors.InvalidDfError* or subclass for issues.

classmethod get_typing() → *typeddfs.df_typing.DfTyping*

classmethod new_df(*n: Union[int, Sequence[str]] = 0*, *fill: Union[int, float, complex] = 0*) → __qualname__

Returns a DataFrame that is empty but valid.

Parameters

- **n** – Either a number of rows/columns or a sequence of labels. If a number is given, will choose (str-type) labels '0', '1', ...
- **fill** – A value to fill in every cell. Should match `self.required_dtype`.

Raises

- ***InvalidDfError*** – If a function in `verifications` fails (returns False or a string).
- **IntCastingNaNError** – If `fill` is NaN or inf and `self.required_dtype` does not support it.

symmetrize() → __qualname__

Averages with its transpose, forcing it to be symmetric.

class typeddfs.matrix_dfs.**LongFormMatrixDf**(*data=None*, *index=None*, *columns=None*, *dtype=None*, *copy=False*)

A long-form matrix with columns "row", "column", and "value".

classmethod get_typing() → *typeddfs.df_typing.DfTyping*

class typeddfs.matrix_dfs.**MatrixDf**(*data=None*, *index=None*, *columns=None*, *dtype=None*, *copy=False*)

A dataframe that is best thought of as a simple matrix. Contains a single index level and a list of columns, with numerical values of a single dtype.

classmethod get_typing() → *typeddfs.df_typing.DfTyping*

classmethod new_df(*rows: Union[int, Sequence[str]] = 0*, *cols: Union[int, Sequence[str]] = 0*, *fill: Union[int, float, complex] = 0*) → __qualname__

Returns a DataFrame that is empty but valid.

Parameters

- **rows** – Either a number of rows or a sequence of labels. If a number is given, will choose (str-type) labels '0', '1', ...
- **cols** – Either a number of columns or a sequence of labels. If a number is given, will choose (str-type) labels '0', '1', ...
- **fill** – A value to fill in every cell. Should match `self.required_dtype`. String values are

Raises

- ***InvalidDfError*** – If a function in `verifications` fails (returns False or a string).
- **IntCastingNaNError** – If `fill` is NaN or inf and `self.required_dtype` does not support it.

typeddfs.typed_dfs

Defines DataFrames with convenience methods and that enforce invariants.

Module Contents

class typeddfs.typed_dfs.**PlainTypedDf**(*data=None*, *index=None*, *columns=None*, *dtype=None*, *copy=False*)

A trivial TypedDf that behaves like an untyped one.

class typeddfs.typed_dfs.**TypedDf**(*data=None*, *index=None*, *columns=None*, *dtype=None*, *copy=False*)

A concrete BaseFrame that enforces conditions. Each subclass has required and reserved (optional) columns and index names. They may or may not permit additional columns or index names.

The constructor will require the conditions to pass but will not rearrange columns and indices. To do that, call `convert`.

Overrides a number of DataFrame methods that preserve the subclass. For example, calling `df.reset_index()` will return a `TypedDf` of the same type as `df`. If a condition would then fail, call `untyped()` first.

For example, suppose `MyTypedDf` has a required index name called "xyz". Then this will be fine as long as `df` has a column or index name called `xyz`: `MyTypedDf.convert(df)`. But calling `MyTypedDf.convert(df).reset_index()` will fail. You can put the column "xyz" back into the index using `convert`: `MyTypedDf.convert(df.reset_index())`. Or, you can get a plain DataFrame (UntypedDf) back: `MyTypedDf.convert(df).untyped().reset_index()`.

To summarize: Call `untyped()` before calling something that would result in anything invalid.

classmethod _check(*df*) → None

Should raise an *typeddfs.df_errors.InvalidDfError* or subclass for issues.

classmethod _check_has_required(*df: pandas.DataFrame*) → None

classmethod _check_has_unexpected(*df: pandas.DataFrame*) → None

classmethod convert(*df: pandas.DataFrame*) → __qualname__

Converts a vanilla Pandas DataFrame (or any subclass) to `cls`. Explicitly sets the new copy's __class__ to cls. Rearranges the columns and index names. For example, if a column in `df` is in `self.reserved_index_names()`, it will be moved to the index.

The new index names will be, in order:

- `required_index_names()`, in order

- `reserved_index_names()`, in order
- any extras in `df`, if `more_indices_allowed` is True

Similarly, the new columns will be, in order:

- `required_columns()`, in order
- `reserved_columns()`, in order
- any extras in `df` in the original, if `more_columns_allowed` is True

Note: Any column called `index` or `level_0` will be dropped automatically.

Parameters **df** – The Pandas DataFrame or member of cls; will have its __class_ change but will otherwise not be affected

Returns A copy

Raises

- ***InvalidDfError*** – If a condition such as a required column or symmetry fails (specific subclasses)
- **TypeError** – If `df` is not a DataFrame

classmethod get_typing() → *typeddfs.df_typing.DfTyping*

meta() → __qualname__

Drops the columns, returning only the index but as the same type.

Returns A copy

Raises ***InvalidDfError*** – If the result does not pass the typing of this class

classmethod new_df(*reserved: Union[bool, Sequence[str]] = False*) → __qualname__

Returns a DataFrame that is empty but has the correct columns and indices.

Parameters **reserved** – Include reserved index/column names as well as required. If True, adds all reserved index levels and columns; You can also specify the exact list of columns and index names.

Raises ***InvalidDfError*** – If a function in `verifications` fails (returns False or a string).

untyped() → *typeddfs.untyped_dfs.UntypedDf*

Makes a copy that's an UntypedDf. It won't have enforced requirements but will still have the convenience functions.

Returns A shallow copy with its __class__ set to an UntypedDf

See: `vanilla()`

typeddfs.untyped_dfs

Defines DataFrames with convenience methods but that do not enforce invariants.

Module Contents

class typeddfs.untyped_dfs.**UntypedDf**(*data=None*, *index=None*, *columns=None*, *dtype=None*, *copy=False*)

A concrete DataFrame that does not require columns or enforce conditions. Overrides a number of DataFrame methods that preserve the subclass. For example, calling `df.reset_index()` will return a `UntypedDf` of the same type as `df`.

classmethod get_typing() → *typeddfs.df_typing.DfTyping*

classmethod new_df(*rows: int = 0*, *cols: Union[int, Sequence[str]] = 0*, *fill: Any = 0*) → __qualname__

Creates a new, semi-arbitrary DataFrame of the specified rows and columns. The DataFrame will have no index.

Parameters

- **rows** – Number of rows
- **cols** – Number of columns or a sequence of column labels
- **fill** – Fill every cell with this value

5.1.3 Package Contents

typeddfs.**__pkg**

typeddfs.**logger**

typeddfs.**metadata**

typeddfs.**metadata**

PYTHON MODULE INDEX

t

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Z