Triad

Han Wang

Mar 19, 2023

API:

1 triad **1**

1.1 triad.collections . . . 1
- 1.1.1 triad.collections.dict . . . 1
- 1.1.2 triad.collections.fs . . . 4
- 1.1.3 triad.collections.function_wrapper . . . 5
- 1.1.4 triad.collections.schema . . . 7

1.2 triad.utils . . . 10
- 1.2.1 triad.utils.assertion . . . 10
- 1.2.2 triad.utils.class_extension . . . 11
- 1.2.3 triad.utils.convert . . . 13
- 1.2.4 triad.utils.dispatcher . . . 17
- 1.2.5 triad.utils.entry_points . . . 22
- 1.2.6 triad.utils.hash . . . 22
- 1.2.7 triad.utils.iter . . . 22
- 1.2.8 triad.utils.json . . . 24
- 1.2.9 triad.utils.pandas_like . . . 24
- 1.2.10 triad.utils.pyarrow . . . 27
- 1.2.11 triad.utils.rename . . . 30
- 1.2.12 triad.utils.schema . . . 30
- 1.2.13 triad.utils.string . . . 32
- 1.2.14 triad.utils.threading . . . 32

1.3 triad.constants . . . 34

1.4 triad.exceptions . . . 34

Python Module Index **35**

Index **37**

CHAPTER

ONE

TRIAD

1.1 triad.collections

1.1.1 triad.collections.dict

class triad.collections.dict.**IndexedOrderedDict**(**args*, ***kwds*)

Bases: collections.OrderedDict, Dict[triad.collections.dict.KT, triad.collections.dict.VT]

Subclass of OrderedDict that can get and set with index

clear()

Return type None

copy()

Return type *triad.collections.dict.IndexedOrderedDict*

equals(*other*, *with_order*)

Compare with another object

Parameters

- **other** (typing.Any) – for possible types, see *to_kv_iterable()*
- **with_order** (bool) – whether to compare order

Returns whether they equal

get_item_by_index(*index*)

Get key value pair by index

Parameters **index** (int) – index of the item

Return type typing.Tuple[typing.TypeVar(KT), typing.TypeVar(VT)]

Returns key value tuple at the index

get_key_by_index(*index*)

Get key by index

Parameters **index** (int) – index of the key

Return type typing.TypeVar(KT)

Returns key value at the index

get_value_by_index(*index*)
Get value by index

Parameters **index** (int) – index of the item

Return type typing.TypeVar(VT)

Returns value at the index

index_of_key(*key*)
Get index of key

Parameters **key** (typing.Any) – key value

Return type int

Returns index of the key value

move_to_end(**args*, ***kwds*)
Move an existing element to the end (or beginning if last is false).

Raise KeyError if the element does not exist.

Return type None

pop(**args*, ***kwds*)
value. If key is not found, d is returned if given, otherwise KeyError is raised.

Return type typing.TypeVar(VT)

pop_by_index(*index*)
Pop item at index

Parameters **index** (int) – index of the item

Return type typing.Tuple[typing.TypeVar(KT), typing.TypeVar(VT)]

Returns key value tuple at the index

popitem(**args*, ***kwds*)
Remove and return a (key, value) pair from the dictionary.

Pairs are returned in LIFO order if last is true or FIFO order if false.

Return type typing.Tuple[typing.TypeVar(KT), typing.TypeVar(VT)]

property readonly: bool
Whether this dict is readonly

Return type bool

set_readonly()
Make this dict readonly

Return type None

set_value_by_index(*index*, *value*)
Set value by index

Parameters

- **index** (int) – index of the item
- **value** (typing.TypeVar(VT)) – new value

Return type None

class triad.collections.dict.**ParamDict**(*data=None*, *deep=True*)

Bases: *triad.collections.dict.IndexedOrderedDict*[str, Any]

Parameter dictionary, a subclass of IndexedOrderedDict, keys must be string

Parameters

- **data** (typing.Optional[typing.Any]) – for possible types, see *to_kv_iterable()*
- **deep** (bool) – whether to deep copy data

IGNORE = 2

OVERWRITE = 0

THROW = 1

get(*key*, *default*)

Get value by key, and the value must be a subtype of the type of default``(which can't be None). If the ``key is not found, return default.

Parameters **key** (typing.Union[int, str]) – the key to search

Raises

- ***NoneArgumentError*** – if default is None
- **TypeError** – if the value can't be converted to the type of default

Return type typing.Any

Returns the value by key, and the value must be a subtype of the type of default. If key is not found, return *default*

get_or_none(*key*, *expected_type*)

Get value by *key*, and the value must be a subtype of expected_type

Parameters

- **key** (typing.Union[int, str]) – the key to search
- **expected_type** (type) – expected return value type

Raises **TypeError** – if the value can't be converted to expected_type

Return type typing.Any

Returns if key is not found, None. Otherwise if the value can be converted to expected_type, return the converted value, otherwise raise exception

get_or_throw(*key*, *expected_type*)

Get value by key, and the value must be a subtype of expected_type. If key is not found or value can't be converted to expected_type, raise exception

Parameters

- **key** (typing.Union[int, str]) – the key to search
- **expected_type** (type) – expected return value type

Raises

- **KeyError** – if key is not found
- **TypeError** – if the value can't be converted to expected_type

Return type typing.Any

Returns only when `key` is found and can be converted to `expected_type`, return the converted value

`to_json`(*indent=False*)
Generate json expression string for the dictionary

Parameters **indent** (bool) – whether to have indent

Return type str

Returns json string

`update`(*other*, *on_dup=0*, *deep=True*)
Update dictionary with another object (for possible types, see *to_kv_iterable()*)

Parameters

- **other** (typing.Any) – for possible types, see *to_kv_iterable()*
- **on_dup** (int) – one of ParamDict.OVERWRITE, ParamDict.THROW and ParamDict.IGNORE

Raises

- **KeyError** – if using ParamDict.THROW and other contains existing keys
- **ValueError** – if on_dup is invalid

Return type *triad.collections.dict.ParamDict*

Returns itself

1.1.2 triad.collections.fs

`class triad.collections.fs.FileSystem`(*auto_close=True*)
Bases: fs.mountfs.MountFS

A unified filesystem based on PyFileSystem2. The special requirement for this class is that all paths must be absolute path with scheme. To customize different file systems, you should override *create_fs* to provide your own configured file systems.

Examples

```
fs = FileSystem()
fs.writetext("mem://from/a.txt", "hello")
fs.copy("mem://from/a.txt", "mem://to/a.txt")
```

Note: If a path is not a local path, it must include the scheme and *netloc* (the first element after *://*) :param auto_close: If *True* (the default), the child filesystems will be closed when *MountFS* is closed.

`create_fs`(*root*)
create a PyFileSystem instance from *root*. *root* is in the format of / if local path, else <*scheme*>*://*<*netloc*>. You should override this method to provide custom instances, for example, if you want to create an S3FS with certain parameters. :type root: str :param root: / if local path, else <*scheme*>*://*<*netloc*>

Return type fs.base.FS

property glob
A globber object

makedirs(*path*, *permissions=None*, *recreate=False*)
Make a directory, and any missing intermediate directories.

Note: This overrides the base `makedirs`

Parameters

- **path** (`str`) – path to directory from root.
- **permissions** (`typing.Optional[typing.Any]`) – initial permissions, or *None* to use defaults.

Recreate if *False* (the default), attempting to create an existing directory will raise an error. Set to *True* to ignore existing directories.

Return type `fs.subfs.SubFS`

Returns a sub-directory filesystem.

Raises

- **fs.errors.DirectoryExists** – if the path is already a directory, and `recreate` is *False*.
- **fs.errors.DirectoryExpected** – if one of the ancestors in the path is not a directory.

1.1.3 triad.collections.function_wrapper

class triad.collections.function_wrapper.**AnnotatedParam**(*param*)
Bases: `object`

An abstraction of annotated parameter

class triad.collections.function_wrapper.**FunctionWrapper**(*func*, *params_re='.*'*, *return_re='.*'*)
Bases: `object`

Create a function wrapper that can recognize and validate all input types.

Parameters

- **func** (`typing.Callable`) – the function to be wrapped
- **params_re** (`str`) – paramter types regex expression
- **return_re** (`str`) – return types regex expression

Examples

Here is a simple example to show how to use FunctionWrapper. Assuming we want to validate the functions with 2 pandas dataframes as the first two input and then arbitray other input, and with 1 pandas dataframe as the return

```
import pandas as pd

@function_wrapper(None)  # all param defintions are here, no entrypoint
class MyFuncWrapper(FunctionWrapper):
    def __init__(self, func):
```

(continues on next page)

(continued from previous page)

```
        super().__init__(
            func,
            params_re="^dd.*",  # starts with two dataframe parameters
            return_re="^d$",  # returns a dataframe
        )

@MyFuncWrapper.annotated_param(pd.DataFrame, code="d")
class MyDataFrameParam(AnnotatedParam):
    pass

def f1(a:pd.DataFrame, b:pd.DataFrame, c) -> pd.DataFrame:
    return a

def f2(a, b:pd.DataFrame, c):
    return a

# f1 is valid
MyFuncWrapper(f1)

# f2 is invalid because of the first parameter
# TypeError will be thrown
MyFuncWrapper(f2)
```

---

classmethod annotated_param(*annotation*, *code=None*, *matcher=None*, *child_can_reuse_code=False*)

The decorator to register a type annotation for this function wrapper

Parameters

- **annotation** (typing.Any) – the type annotation
- **code** (typing.Optional[str]) – the single char code to represent this type annotation , defaults to None, meaning it will try to use its parent class' code, this is allowed only if child_can_reuse_code is set to True on the parent class.
- **matcher** (typing.Optional[typing.Callable[[typing.Any], bool]]) – a function taking in a type annotation and decide whether it is acceptable by the *AnnotatedParam* , defaults to None, meaning it will just do a simple == check.
- **child_can_reuse_code** (bool) – whether the derived types of the current AnnotatedParam can reuse the code (if not specifying a new code) , defaults to False

property input_code: str

The input parameters code expression

Return type str

property output_code: str

The output code expression

Return type str

classmethod parse_annotation(*annotation*, *param=None*, *none_as_other=True*)

Return type *triad.collections.function_wrapper.AnnotatedParam*

class triad.collections.function_wrapper.**KeywordParam**(*param*)

Bases: *triad.collections.function_wrapper.AnnotatedParam*

For keyword parameters

class triad.collections.function_wrapper.**NoneParam**(*param*)
Bases: *triad.collections.function_wrapper.AnnotatedParam*

The case where there is no annotation for a parameter

class triad.collections.function_wrapper.**OtherParam**(*param*)
Bases: *triad.collections.function_wrapper.AnnotatedParam*

Any annotation that is not recognized

class triad.collections.function_wrapper.**PositionalParam**(*param*)
Bases: *triad.collections.function_wrapper.AnnotatedParam*

For positional parameters

class triad.collections.function_wrapper.**SelfParam**(*param*)
Bases: *triad.collections.function_wrapper.AnnotatedParam*

For the self parameters in member functions

triad.collections.function_wrapper.**function_wrapper**(*entrypoint*)
The decorator to register a new *FunctionWrapper* type.

Parameters **entrypoint** (typing.Optional[str]) – the entrypoint to load in setup.py in order to find the registered *AnnotatedParam* under this *FunctionWrapper*

1.1.4 triad.collections.schema

class triad.collections.schema.**Schema**(**args*, ***kwargs*)
Bases: *triad.collections.dict.IndexedOrderedDict*[str, pyarrow.lib.Field]

A Schema wrapper on top of pyarrow.Fields. This has more features than pyarrow.Schema, and they can convert to each other.

This class can be initialized from schema like objects. Here is a list of schema like objects:

- pyarrow.Schema or Schema objects
- pyarrow.Field: single field will be treated as a single column schema
- schema expressions: *expression_to_schema()*
- Dict[str,Any]: key will be the columns, and value will be type like objects
- Tuple[str,Any]: first item will be the only column name of the schema, and the second has to be a type like object
- List[Any]: a list of Schema like objects
- pandas.DataFrame: it will extract the dataframe's schema

Here is a list of data type like objects:

- pyarrow.DataType
- pyarrow.Field: will only use the type attribute of the field
- type expression or other objects: for *to_pa_datatype()*

Examples

```
Schema("a:int,b:int")
Schema("a:int","b:int")
Schema(a=int,b=str) # == Schema("a:long,b:str")
Schema(dict(a=int,b=str)) # == Schema("a:long,b:str")
Schema([(a,int),(b,str)]) # == Schema("a:long,b:str")
Schema((a,int),(b,str)) # == Schema("a:long,b:str")
Schema("a:[int],b:{x:int,y:{z:[str],w:byte}},c:[{x:str}]")
```

Note:

- For supported pyarrow.DataTypes see *is_supported()*
- If you use python type as data type (e.g. *Schema(a=int,b=str)*) be aware the data type different. (e.g. python *int* type -> pyarrow *long/int64* type)
- When not readonly, only *append* is allowed, *update* or *remove* are disallowed
- When readonly, no modification on the existing schema is allowed
- *append*, *update* and *remove* are always allowed when creating a new object
- InvalidOperationError will be raised for disallowed operations
- At most one of **args* and ***kwargs* can be set

Parameters

- **args** (typing.Any) – one or multiple schema like objects, which will be combined in order
- **kwargs** (typing.Any) – key value pairs for the schema

append(*obj*)

Append schema like object to the current schema. Only new columns are allowed.

Raises ***SchemaError*** – if a column exists or is invalid or obj is not convertible

Return type *triad.collections.schema.Schema*

Returns the Schema object itself

assert_not_empty()

Return type *triad.collections.schema.Schema*

copy()

Clone Schema object

Return type *triad.collections.schema.Schema*

Returns cloned object

exclude(*other*, *require_type_match=True*, *ignore_type_mismatch=False*)

Return type *triad.collections.schema.Schema*

extract(*obj*, *ignore_key_mismatch=False*, *require_type_match=True*, *ignore_type_mismatch=False*)

Return type *triad.collections.schema.Schema*

property fields: List[pyarrow.lib.Field]
List of pyarrow.Fields

Return type typing.List[pyarrow.lib.Field]

intersect(*other*, *require_type_match=True*, *ignore_type_mismatch=True*, *use_other_order=False*)

Return type *triad.collections.schema.Schema*

property names: List[str]
List of column names

Return type typing.List[str]

property pa_schema: pyarrow.lib.Schema
convert as pyarrow.Schema

Return type pyarrow.lib.Schema

property pandas_dtype: Dict[str, numpy.dtype]
convert as *dtype* dict for pandas dataframes. Currently, struct type is not supported

Return type typing.Dict[str, numpy.dtype]

property pd_dtype: Dict[str, numpy.dtype]
convert as *dtype* dict for pandas dataframes. Currently, struct type is not supported

Return type typing.Dict[str, numpy.dtype]

property pyarrow_schema: pyarrow.lib.Schema
convert as pyarrow.Schema

Return type pyarrow.lib.Schema

remove(*obj*, *ignore_key_mismatch=False*, *require_type_match=True*, *ignore_type_mismatch=False*)

Return type *triad.collections.schema.Schema*

rename(*columns*, *ignore_missing=False*)
Rename the current schema and generate a new one

Parameters **columns** (typing.Dict[str, str]) – dictionary to map from old to new column names

Return type *triad.collections.schema.Schema*

Returns renamed schema object

transform(**args*, ***kwargs*)
Transform the current schema to a new schema

Raises ***SchemaError*** – if there is any exception

Return type *triad.collections.schema.Schema*

Returns transformed schema

Examples

```
s=Schema("a:int,b:int,c:str")
s.transform("x:str") # x:str
# add
```

(continues on next page)

(continued from previous page)

```
s.transform("*,x:str") # a:int,b:int,c:str,x:str
s.transform("*","x:str") # a:int,b:int,c:str,x:str
s.transform("*",x=str) # a:int,b:int,c:str,x:str
# subtract
s.transform("*-c,a") # b:int
s.transform("*-c-a") # b:int
s.transform("*~c,a,x") # b:int  # ~ means exlcude if exists
s.transform("*~c~a~x") # b:int  # ~ means exlcude if exists
# + means overwrite existing and append new
s.transform("*+e:str,b:str,d:str") # a:int,b:str,c:str,e:str,d:str
# you can have multiple operations
s.transform("*+b:str-a") # b:str,c:str
# callable
s.transform(lambda s:s.fields[0]) # a:int
s.transform(lambda s:s.fields[0], lambda s:s.fields[2]) # a:int,c:str
```

property types: List[pyarrow.lib.DataType]

List of pyarrow.DataTypes

Return type typing.List[pyarrow.lib.DataType]

union(*other*, *require_type_match=False*)

Return type *triad.collections.schema.Schema*

union_with(*other*, *require_type_match=False*)

Return type *triad.collections.schema.Schema*

exception triad.collections.schema.**SchemaError**(*message*)

Bases: Exception

Exceptions related with construction and modifying schemas

1.2 triad.utils

1.2.1 triad.utils.assertion

triad.utils.assertion.**assert_arg_not_none**(*obj*, *arg_name=''*, *msg=''*)

Assert an argument is not None, otherwise raise exception

Parameters

- **obj** (typing.Any) – argument value
- **arg_name** (str) – argument name, if None or empty, it will use *msg*
- **msg** (str) – only when *arg_name* is None or empty, this value is used

Raises *NoneArgumentError* – with *arg_name* or *msg*

Return type None

triad.utils.assertion.**assert_or_throw**(*bool_exp*, *exception=None*)

Assert on expression and throw custom exception

Parameters

- **bool_exp** (bool) – boolean expression to assert on
- **exception** (typing.Optional[typing.Any]) – a custom Exception instance, or any other object that will be stringfied and instantiate an AssertionError, or a function that can generate the supported data types

Examples

```
assert_or_throw(True, "assertion error")
assert_or_throw(False)  # raise AssertionError
assert_or_throw(False, "assertion error")  # raise AssertionError
assert_or_throw(False, TypeError("assertion error"))  # raise TypeError

# Lazy evaluations is useful when constructing the error
# itself is expensive or error-prone. With lazy evaluations, happy
# path will be fast and error free.
def fail():  # a function that is slow and wrong
    sleep(10)
    raise TypeError

assert_or_throw(True, fail())  # (unexpectedly) raise TypeError
assert_or_throw(True, fail)  # no exception
assert_or_throw(True, lambda: "a" + fail())  # no exception
assert_or_throw(False, lambda: "a" + fail())  # raise TypeError
```

Return type None

1.2.2 triad.utils.class_extension

triad.utils.class_extension.**extensible_class**(*class_type*)

The decorator making classes extensible by external methods

Parameters **class_type** (typing.Type) – the class under the decorator

Return type typing.Type

Returns the class_type

Examples

```
@extensible_class
class A:

    # It's recommended to implement __getattr__ so that
    # PyLint will not complain about the dynamically added methods
    def __getattr__(self, name):
        raise NotImplementedError

@extension_method
def method(obj:A):
    return 1
```

(continues on next page)

(continued from previous page)

```
assert 1 == A().method()
```

Note: If the method name is already in the original class, a ValueError will be thrown. You can't modify any built-in attribute.

---

triad.utils.class_extension.**extension_method**(*func=None*, *class_type=None*, *name=None*, *on_dup='error'*)

The decorator to add functions as members of the correspondent classes.

Parameters

- **func** (typing.Optional[typing.Callable]) – the function under the decorator
- **class_type** (typing.Optional[typing.Type]) – the parent class type, defaults to None
- **name** (typing.Optional[str]) – the specified class method name, defaults to None. If None then `func.__name__` will be used as the method name
- **on_dup** (str) – action on name duplication, defaults to `error`. `error` will throw a ValueError; `ignore` will take no action; `overwrite` will use the current method to overwrite.

Return type typing.Callable

Returns the underlying function

Examples

```
@extensible_class
class A:

    # It's recommended to implement __getattr__ so that
    # PyLint will not complain about the dynamically added methods
    def __getattr__(self, name):
        raise NotImplementedError

# The simplest way to use this decorator, the first argument of
# the method must be annotated, and the annotated type is the
# class type to add this method to.
@extension_method
def method1(obj:A):
    return 1

assert 1 == A().method1()

# Or you can be explicit of the class type and the name of the
# method in the class. In this case, you don't have to annotate
# the first argument.
@extension_method(class_type=A, name="m3")
def method2(obj, b):
    return 2 + b

assert 5 == A().m3(3)
```

Note: If the method name is already in the original class, a ValueError will be thrown. You can't modify any built-in attribute.

1.2.3 triad.utils.convert

triad.utils.convert.**as_type**(*obj*, *target*)

Convert *obj* into *target* type

Parameters

- **obj** (typing.Any) – input object
- **target** (type) – target type

Return type typing.Any

Returns object in the target type

triad.utils.convert.**get_caller_global_local_vars**(*global_vars=None*, *local_vars=None*, *start=- 1*, *end=- 1*)

Get the caller level global and local variables.

Parameters

- **global_vars** (typing.Optional[typing.Dict[str, typing.Any]]) – overriding global variables, if not None, will return this instead of the caller's globals(), defaults to None
- **local_vars** (typing.Optional[typing.Dict[str, typing.Any]]) – overriding local variables, if not None, will return this instead of the caller's locals(), defaults to None
- **start** (int) – start stack level (from 0 to any negative number), defaults to -1 which is one level above where this function is invoked
- **end** (int) – end stack level (from `start` to any smaller negative number), defaults to -1 which is one level above where this function is invoked

Return type typing.Tuple[typing.Dict[str, typing.Any], typing.Dict[str, typing.Any]]

Returns tuple of *global_vars* and *local_vars*

Examples

```
def caller():
    x=1
    assert 1 == get_value("x")

def get_value(var_name):
    _, l = get_caller_global_local_vars()
    assert var_name in l
    assert var_name not in locals()
    return l[var_name]
```

Notice

This is for internal use, users normally should not call this directly.

If merging multiple levels, the variables on closer level (to where it is invoked) will overwrite the further levels values if there is overlap.

Examples

```
def f1():
    x=1

    def f2():
        x=2

        def f3():
            _, l = get_caller_global_local_vars(start=-1,end=-2)
            assert 2 == l["x"]

            _, l = get_caller_global_local_vars(start=-2,end=-2)
            assert 1 == l["x"]

    f2()
f1()
```

triad.utils.convert.**get_full_type_path**(*obj*)

Get the full module path of the type (if *obj* is class or function) or type of the instance (if *obj* is an object instance)

Parameters **obj** (typing.Any) – a class/function type or an object instance

Raises **TypeError** – if *obj* is None, lambda, or neither a class or a function

Return type str

Returns full path string

triad.utils.convert.**str_to_instance**(*s*, *expected_base_type=None*, *args=[]*, *kwargs={}*, *global_vars=None*, *local_vars=None*)

Use *str_to_type()* to find a matching type and instantiate

Parameters

- **s** (str) – see *str_to_type()*
- **expected_base_type** (typing.Optional[type]) – see *str_to_type()*
- **args** (typing.List[typing.Any]) – args to instantiate the type
- **kwargs** (typing.Dict[str, typing.Any]) – kwargs to instantiate the type
- **global_vars** (typing.Optional[typing.Dict[str, typing.Any]]) – overriding global variables, if None, it will use the caller's globals(), defaults to None
- **local_vars** (typing.Optional[typing.Dict[str, typing.Any]]) – overriding local variables, if None, it will use the caller's locals(), defaults to None

Return type typing.Any

Returns the instantiated the object

triad.utils.convert.**str_to_object**(*expr*, *global_vars=None*, *local_vars=None*)

Convert string expression to object. The string expression must express a type with relative or full path, or express a local or global instance without brackets or operators.

Parameters

- **expr** (str) – string expression, see examples below
- **global_vars** (typing.Optional[typing.Dict[str, typing.Any]]) – overriding global variables, if None, it will use the caller's globals(), defaults to None
- **local_vars** (typing.Optional[typing.Dict[str, typing.Any]]) – overriding local variables, if None, it will use the caller's locals(), defaults to None

Return type typing.Any

Returns the object

Raises ValueError – unable to find a matching object

Examples

```
class _Mock(object):
    def __init__(self, x=1):
        self.x = x

m = _Mock()
assert 1 == str_to_object("m.x")
assert 1 == str_to_object("m2.x", local_vars={"m2": m})
assert RuntimeError == str_to_object("RuntimeError")
assert _Mock == str_to_object("_Mock")
```

Note: This function is to dynamically load an object from string expression. If you write that string expression as python code at the same location, it should generate the same result.

triad.utils.convert.**str_to_type**(*s*, *expected_base_type=None*, *global_vars=None*, *local_vars=None*)

Given a string expression, find the first/last type from all import libraries. If the expression contains ., it's supposed to be a relative or full path of the type including modules.

Parameters

- **s** (str) – type expression, for example *triad.utils.iter.Slicer* or *str*
- **expected_base_type** (typing.Optional[type]) – base class type that must satisfy, defaults to None
- **global_vars** (typing.Optional[typing.Dict[str, typing.Any]]) – overriding global variables, if None, it will use the caller's globals(), defaults to None
- **local_vars** (typing.Optional[typing.Dict[str, typing.Any]]) – overriding local variables, if None, it will use the caller's locals(), defaults to None

Raises TypeError – unable to find a matching type

Return type type

Returns found type

triad.utils.convert.**to_bool**(*obj*)

Convert an object to python bool value. It can handle values like *True*, *true*, *yes*, *1*, etc

Parameters **obj** (typing.Any) – object

Raises TypeError – if failed to convert

Return type bool

Returns bool value

triad.utils.convert.**to_datetime**(*obj*)

Convert an object to python datetime. If the object is a string, then if ciso8601 is installed then it will use `ciso8601.parse_datetime` to parse else it will use `pandas.to_datetime` to parse, which can be a lot slower.

Parameters **obj** (typing.Any) – object

Raises **TypeError** – if failed to convert

Return type datetime.datetime

Returns datetime value

triad.utils.convert.**to_function**(*func*, *global_vars=None*, *local_vars=None*)

For an expression, it tries to find the matching function.

Params s a string expression or a callable

Parameters

- **global_vars** (typing.Optional[typing.Dict[str, typing.Any]]) – overriding global variables, if None, it will use the caller's globals(), defaults to None
- **local_vars** (typing.Optional[typing.Dict[str, typing.Any]]) – overriding local variables, if None, it will use the caller's locals(), defaults to None

Raises **AttributeError** – if unable to find such a function

Return type typing.Any

Returns the matching function

triad.utils.convert.**to_instance**(*s*, *expected_base_type=None*, *args=[]*, *kwargs={}*, *global_vars=None*, *local_vars=None*)

If s is str or type, then use *to_type()* to find matching type and instantiate. Otherwise return s if it matches constraints

Parameters

- **s** (typing.Any) – see *to_type()*
- **expected_base_type** (typing.Optional[type]) – see *to_type()*
- **args** (typing.List[typing.Any]) – args to instantiate the type
- **kwargs** (typing.Dict[str, typing.Any]) – kwargs to instantiate the type
- **global_vars** (typing.Optional[typing.Dict[str, typing.Any]]) – overriding global variables, if None, it will use the caller's globals(), defaults to None
- **local_vars** (typing.Optional[typing.Dict[str, typing.Any]]) – overriding local variables, if None, it will use the caller's locals(), defaults to None

Raises

- **ValueError** – if s is an instance but not a (sub)type of *expected_base_type*
- **TypeError** – if s is an instance, args and kwargs must be empty

Return type typing.Any

Returns the instantiated object

triad.utils.convert.**to_size**(*exp*)

Convert input value or expression to size For expression string, it must be in the format of <*value*> or <*value*><*unit*>. Value must be 0 or positive, default unit is byte if not provided. Unit can be *b*, *byte*, *k*, *kb*, *m*, *mb*, *g*, *gb*, *t*, *tb*.

Args: exp (Any): expression string or numerical value

Raises: ValueError: for invalid expression ValueError: for negative values

Returns: int: size in byte

Return type int

triad.utils.convert.**to_timedelta**(*obj*)

Convert an object to python datetime.

If the object is a string, *min* or *-inf* will return *timedelta.min*, *max* or *inf* will return *timedelta.max*; if the object is a number, the number will be used as the seconds argument; Otherwise it will use *pandas.to_timedelta* to parse the object.

Parameters **obj** (typing.Any) – object

Raises TypeError – if failed to convert

Return type datetime.timedelta

Returns timedelta value

triad.utils.convert.**to_type**(*s*, *expected_base_type=None*, *global_vars=None*, *local_vars=None*)

Convert an object *s* to *type* * if *s* is *str*: see *str_to_type()* * if *s* is *type*: check *expected_base_type* and return itself * else: check *expected_base_type* and return itself

Parameters

- **s** (typing.Any) – see *str_to_type()*
- **expected_base_type** (typing.Optional[type]) – see *str_to_type()*
- **global_vars** (typing.Optional[typing.Dict[str, typing.Any]]) – overriding global variables, if None, it will use the caller's globals(), defaults to None
- **local_vars** (typing.Optional[typing.Dict[str, typing.Any]]) – overriding local variables, if None, it will use the caller's locals(), defaults to None

Raises TypeError – if no matching type found

Return type type

Returns the matching type

1.2.4 triad.utils.dispatcher

class triad.utils.dispatcher.**ConditionalDispatcher**(*default_func*, *entry_point=None*)

Bases: object

A conditional function dispatcher based on custom matching functions. This is a more general solution compared to `functools.singledispatch`. You can write arbitrary matching functions according to all the inputs of the function.

Note: Please use the decorators *conditional_dispatcher()* and *conditional_broadcaster()* instead of directly using this class.

Parameters

- **default_func** (typing.Callable[..., typing.Any]) – the parent function that will dispatch the execution based on matching functions
- **entry_point** (typing.Optional[str]) – the entry point to preload children functions, defaults to None

candidate(*matcher*, *priority=1.0*)

A decorator to register a child function with matcher and priority.

Note: The order to be matched is determined by both the priority and the order of registration.

- The default priority is 1.0
- Children with higher priority values will be matched earlier
- When `priority>0` then later registrations will be matched earlier
- When `priority<=0` then earlier registrations will be matched earlier

So if you want to 'overwrite' the existed matches, set priority to be greater than 1.0. If you want to 'ignore' the current if there are other matches, set priority to 0.0.

See also:

Please see examples in *conditional_dispatcher()* and *conditional_broadcaster()*.

Parameters

- **matcher** (typing.Callable[..., bool]) – a function determines whether it is a match based on the same input as the parent function
- **priority** (float) – it determines the order to be matched, **higher value means higher priority**, defaults to 1.0

Return type typing.Callable

register(*func*, *matcher*, *priority=1.0*)

Register a child function with matcher and priority.

Note: The order to be matched is determined by both the priority and the order of registration.

- The default priority is 1.0
- Children with higher priority values will be matched earlier
- When `priority>0` then later registrations will be matched earlier
- When `priority<=0` then earlier registrations will be matched earlier

So if you want to 'overwrite' the existed matches, set priority to be greater than 1.0. If you want to 'ignore' the current if there are other matches, set priority to 0.0.

Parameters

- **func** (typing.Callable[..., typing.Any]) – a child function to be used when matching
- **matcher** (typing.Callable[..., bool]) – a function determines whether it is a match based on the same input as the parent function
- **priority** (float) – it determines the order to be matched, **higher value means higher priority**, defaults to 1.0

Return type None

run(**args*, ***kwargs*)
Execute all matching children functions as a generator.

Note: Only when there is matching functions, the default implementation will be invoked.

Return type typing.Iterable[typing.Any]

run_top(**args*, ***kwargs*)
Execute the first matching child function

Return type typing.Any

Returns the return of the child function

triad.utils.dispatcher.**conditional_broadcaster**(*default_func=None*, *entry_point=None*)
Decorating a conditional broadcaster that will run **all** registered functions in other modules/packages.

Examples

Assume in `pkg1.module1`, you have:

```
from triad import conditional_broadcaster

@conditional_broadcaster(entry_point="my.plugins")
def myprint(obj):
    raise NotImplementedError

@conditional_broadcaster(entry_point="my.plugins")
def myprint2(obj):
    raise NotImplementedError
```

In another package `pkg2`, in `setup.py`, you define an entry point as:

```
setup(
    ...,
    entry_points={
        "my.plugins": [
            "my = pkg2.module2"
        ]
    },
)
```

And in `pkg2.module2`:

```
from pkg1.module1 import get_len

@myprint.candidate(lambda obj: isinstance(obj, str))
def myprinta(obj:str) -> None:
    print(obj, "a")

@myprint.candidate(lambda obj: isinstance(obj, str) and obj == "x")
def myprintb(obj:str) -> None:
    print(obj, "b")
```

Now, both functions will be automatically registered when `pkg2` is installed in the environement. In another `pkg3`:

```
from pkg1.module1 import get_len

myprint("x")  # calling both myprinta and myprinta
myprint("y")  # calling myprinta only
myprint2("x")  # raise NotImplementedError due to no matching candidates
```

Note: Only when no matching candidate found, the implementation of the original function will be used. If you don't want to throw an error, then use `pass` in the original function instead.

See also:

Please read *candidate()* for details about the matching function and priority settings.

Parameters

- **default_func** (typing.Optional[typing.Callable[..., typing.Any]]) – the function to decorate
- **entry_point** (typing.Optional[str]) – the entry point to preload dispatchers, defaults to None

Return type typing.Callable

triad.utils.dispatcher.**conditional_dispatcher**(*default_func=None*, *entry_point=None*)

Decorating a conditional dispatcher that will run the **first matching** registered functions in other modules/packages. This is a more general solution compared to `functools.singledispatch`. You can write arbitrary matching functions according to all the inputs of the function.

Examples

Assume in `pkg1.module1`, you have:

```
from triad import conditional_dispatcher

@conditional_dispatcher(entry_point="my.plugins")
def get_len(obj):
    raise NotImplementedError
```

In another package `pkg2`, in `setup.py`, you define an entry point as:

```
setup(
    ...,
    entry_points={
        "my.plugins": [
            "my = pkg2.module2"
        ]
    },
)
```

And in `pkg2.module2`:

```
from pkg1.module1 import get_len

@get_len.candidate(lambda obj: isinstance(obj, str))
def get_str_len(obj:str) -> int:
    return len(obj)

@get_len.candidate(lambda obj: isinstance(obj, int) and obj == 10)
def get_int_len(obj:int) -> int:
    return obj
```

Now, both functions will be automatically registered when `pkg2` is installed in the environement. In another `pkg3`:

```
from pkg1.module1 import get_len

assert get_len("abc") == 3  # calling get_str_len
assert get_len(10) == 10  # calling get_int_len
get_len(20)  # raise NotImplementedError due to no matching candidates
```

See also:

Please read *candidate()* for details about the matching function and priority settings.

Parameters

- **default_func** (typing.Optional[typing.Callable[..., typing.Any]]) – the function to decorate
- **entry_point** (typing.Optional[str]) – the entry point to preload dispatchers, defaults to None

Return type typing.Callable

triad.utils.dispatcher.**run_at_def**(*run_at_def_func=None*, ***kwargs*)

Decorator to run the function at declaration. This is useful when we want import to trigger a function run (which can guarantee it runs only once).

Examples

Assume the following python file is a module in your package, then when you `import package.module`, the two functions will run.

```
from triad import run_at_def

@run_at_def
def register_something():
    print("registered")

@run_at_def(a=1)
def register_something2(a):
    print("registered", a)
```

Parameters

- **run_at_def_func** (typing.Optional[typing.Callable]) – the function to decorate
- **kwargs** (typing.Any) – the parameters to call this function

Return type typing.Callable

1.2.5 triad.utils.entry_points

1.2.6 triad.utils.hash

triad.utils.hash.**to_uuid**(**args*)

Determine the uuid by input arguments. It will search the input recursively. If an object contains *__uuid__* method, it will call that method to get the uuid for that object.

Examples

```
to_uuid([1,2,3])
to_uuid(1,2,3)
to_uuid(dict(a=1,b="z"))
```

Parameters **args** (typing.Any) – arbitrary input

Return type str

Returns uuid string

1.2.7 triad.utils.iter

class triad.utils.iter.**EmptyAwareIterable**(*it*)

Bases: Iterable[triad.utils.iter.T]

A wrapper of iterable that can tell if the underlying iterable is empty, it can also peek a non-empty iterable.

Parameters **it** (typing.Union[typing.Iterable[typing.TypeVar(T)], typing.Iterator[typing.TypeVar(T)]]) – the underlying iterable

Raises StopIteration – raised by the underlying iterable

property empty: bool

Check if the underlying iterable has more items

Return type bool

Returns whether it is empty

peek()

Return the next of the iterable without moving

Raises StopIteration – if it's empty

Return type typing.TypeVar(T)

Returns the *next* item

class triad.utils.iter.**Slicer**(*sizer=None*, *row_limit=None*, *size_limit=None*, *slicer=None*)

Bases: object

A better version of slice_iterable()

Parameters

- **sizer** (typing.Optional[typing.Callable[[typing.Any], int]]) – the function to get size of an item
- **row_limit** (typing.Optional[int]) – max row for each slice, defaults to None
- **size_limit** (typing.Optional[typing.Any]) – max byte size for each slice, defaults to None
- **slicer** (typing.Optional[typing.Callable[[int, typing.TypeVar(T), typing.Optional[typing.TypeVar(T)]], bool]]) – taking in current number, current value, last value, it decides if it's a new slice

Raises AssertionError – if *size_limit* is not None but *sizer* is None

slice(*orig_it*)

Slice the original iterable into slices by the combined slicing logic

Parameters **orig_it** (typing.Iterable[typing.TypeVar(T)]) – ther original iterable

Yield an iterable of EmptyAwareIterable

Return type typing.Iterable[*triad.utils.iter.EmptyAwareIterable*[typing.TypeVar(T)]]

triad.utils.iter.**make_empty_aware**(*it*)

Make an iterable empty aware, or return itself if already empty aware

Parameters **it** (typing.Union[typing.Iterable[typing.TypeVar(T)], typing.Iterator[typing.TypeVar(T)]]) – underlying iterable

Return type *triad.utils.iter.EmptyAwareIterable*[typing.TypeVar(T)]

Returns EmptyAwareIterable[T]

triad.utils.iter.**slice_iterable**(*it*, *slicer*)

Slice the original iterable into slices by slicer

Parameters

- **it** (typing.Union[typing.Iterable[typing.TypeVar(T)], typing.Iterator[typing.TypeVar(T)]]) – underlying iterable
- **slicer** (typing.Callable[[int, typing.TypeVar(T), typing.Optional[typing.TypeVar(T)]], bool]) – taking in current number, current value, last value, it decides if it's a new slice

Yield an iterable of iterables (_SliceIterable[T])

Return type typing.Iterable[triad.utils.iter._SliceIterable[typing.TypeVar(T)]]

triad.utils.iter.**to_kv_iterable**(*data*, *none_as_empty=True*)

Convert data to iterable of key value pairs

Parameters

- **data** (typing.Any) – input object, it can be a dict or Iterable[Tuple[Any, Any]] or Iterable[List[Any]]
- **none_as_empty** (bool) – if to treat None as empty iterable

Raises

- **ValueError** – if input is None and *none_as_empty==False*
- **ValueError** – if input is a set
- **TypeError or ValueError** – if input data type is not acceptable

Yield iterable of key value pair as tuples

Return type typing.Iterable[typing.Tuple[typing.Any, typing.Any]]

1.2.8 triad.utils.json

triad.utils.json.**check_for_duplicate_keys**(*ordered_pairs*)

Raise ValueError if a duplicate key exists in provided ordered list of pairs, otherwise return a dict.

Example:

```
>>> json.loads('{"x": 1, "x": 2}', object_pairs_hook=check_for_duplicate_keys)
```

Raises **KeyError** – if there is duplicated key

Return type typing.Dict[typing.Any, typing.Any]

triad.utils.json.**loads_no_dup**(*json_str*)

Load json string, and raise KeyError if there are duplicated keys

Parameters **json_str** (str) – json string

Raises **KeyError** – if there are duplicated keys

Return type typing.Any

Returns the parsed object

1.2.9 triad.utils.pandas_like

class triad.utils.pandas_like.**PandasLikeUtils**(**args*, ***kwds*)

Bases: Generic[triad.utils.pandas_like.T]

A collection of utils for general pandas like dataframes

as_array_iterable(*df*, *schema=None*, *columns=None*, *type_safe=False*)

Convert pandas like dataframe to iterable of rows in the format of list.

Parameters

- **df** (typing.TypeVar(T, bound= typing.Any)) – pandas like dataframe

- **schema** (typing.Optional[pyarrow.lib.Schema]) – schema of the input. With None, it will infer the schema, it can't infer wrong schema for nested types, so try to be explicit
- **columns** (typing.Optional[typing.List[str]]) – columns to output, None for all columns
- **type_safe** (bool) – whether to enforce the types in schema, if False, it will return the original values from the dataframe

Return type typing.Iterable[typing.List[typing.Any]]

Returns iterable of rows, each row is a list

Notice

If there are nested types in schema, the conversion can be slower

as_arrow(*df*, *schema=None*)

Convert pandas like dataframe to pyarrow table

Parameters

- **df** (typing.TypeVar(T, bound= typing.Any)) – pandas like dataframe
- **schema** (typing.Optional[pyarrow.lib.Schema]) – if specified, it will be used to construct pyarrow table, defaults to None

Return type pyarrow.lib.Table

Returns pyarrow table

empty(*df*)

Check if the dataframe is empty

Parameters **df** (typing.TypeVar(T, bound= typing.Any)) – pandas like dataframe

Return type bool

Returns if it is empty

enforce_type(*df*, *schema*, *null_safe=False*)

Enforce the pandas like dataframe to comply with *schema*.

Parameters

- **df** (typing.TypeVar(T, bound= typing.Any)) – pandas like dataframe
- **schema** (pyarrow.lib.Schema) – pyarrow schema
- **null_safe** (bool) – whether to enforce None value for int, string and bool values

Return type typing.TypeVar(T, bound= typing.Any)

Returns converted dataframe

Notice

When *null_safe* is true, the native column types in the dataframe may change, for example, if a column of *int64* has None values, the output will make sure each value in the column is either None or an integer, however, due to the behavior of pandas like dataframes, the type of the columns may no longer be *int64*

This method does not enforce struct and list types

ensure_compatible(*df*)

Check whether the datafame is compatible with the operations inside this utils collection, if not, it will raise ValueError

Parameters **df** (typing.TypeVar(T, bound= typing.Any)) – pandas like dataframe

Raises ValueError – if not compatible

Return type None

fillna_default(*col*)

Fill column with default values according to the dtype of the column.

Parameters col (typing.Any) – series of a pandas like dataframe

Return type typing.Any

Returns filled series

is_compatile_index(*df*)

Check whether the datafame is compatible with the operations inside this utils collection

Parameters df (typing.TypeVar(T, bound= typing.Any)) – pandas like dataframe

Return type bool

Returns if it is compatible

safe_groupby_apply(*df*, *cols*, *func*, *key_col_name='__safe_groupby_key__'*, ***kwargs*)

Safe groupby apply operation on pandas like dataframes. In pandas like groupby apply, if any key is null, the whole group is dropped. This method makes sure those groups are included.

Parameters

- **df** (typing.TypeVar(T, bound= typing.Any)) – pandas like dataframe
- **cols** (typing.List[str]) – columns to group on, can be empty
- **func** (typing.Callable[[typing.TypeVar(T, bound= typing.Any)], typing.TypeVar(T, bound= typing.Any)]) – apply function, df in, df out
- **key_col_name** – temp key as index for groupu. default "__safe_groupby_key__"

Return type typing.TypeVar(T, bound= typing.Any)

Returns output dataframe

Notice

The dataframe must be either empty, or with type pd.RangeIndex, pd.Int64Index or pd.UInt64Index and without a name, otherwise, *ValueError* will raise.

to_schema(*df*)

Extract pandas dataframe schema as pyarrow schema. This is a replacement of pyarrow.Schema.from_pandas, and it can correctly handle string type and empty dataframes

Parameters df (typing.TypeVar(T, bound= typing.Any)) – pandas dataframe

Raises ValueError – if pandas dataframe does not have named schema

Return type pyarrow.lib.Schema

Returns pyarrow.Schema

Notice

The dataframe must be either empty, or with type pd.RangeIndex, pd.Int64Index or pd.UInt64Index and without a name, otherwise, *ValueError* will raise.

class triad.utils.pandas_like.**PandasUtils**(**args*, ***kwds*)

Bases: *triad.utils.pandas_like.PandasLikeUtils*[pandas.core.frame.DataFrame]

A collection of pandas utils

1.2.10 triad.utils.pyarrow

class triad.utils.pyarrow.**SchemaedDataPartitioner**(*schema*, *key_positions*, *sizer=None*, *row_limit=0*, *size_limit=None*)

Bases: object

Partitioner for stream of array like data with given schema. It uses :func"*~triad.utils.iter.Slicer* to partition the stream

Parameters

- **schema** (pyarrow.lib.Schema) – the schema of the data stream to process
- **key_positions** (typing.List[int]) – positions of partition keys on *schema*
- **sizer** (typing.Optional[typing.Callable[[typing.Any], int]]) – the function to get size of an item
- **row_limit** (int) – max row for each slice, defaults to None
- **size_limit** (typing.Optional[typing.Any]) – max byte size for each slice, defaults to None

partition(*data*)

Partition the given data stream

Parameters **data** (typing.Iterable[typing.Any]) – iterable of array like objects

Yield iterable of <partition_no, slice_no, slice iterable> tuple

Return type typing.Iterable[typing.Tuple[int, int, *triad.utils.iter.EmptyAwareIterable*[typing.Any]]]

triad.utils.pyarrow.**apply_schema**(*schema*, *data*, *copy=True*, *deep=False*, *str_as_json=True*)

Use *pa.Schema* to convert a row(list) to the correspondent types.

Notice this function is to convert from python native type to python native type. It is used to normalize data input, which could be generated by different logics, into the correct data types.

Notice this function assumes each item of *data* has the same length with *schema* and will not do any extra validation on that.

Parameters

- **schema** (pyarrow.lib.Schema) – pyarrow schema
- **data** (typing.Iterable[typing.List[typing.Any]]) – and iterable of rows, represented by list or tuple
- **copy** (bool) – whether to apply inplace (copy=False), or create new instances
- **deep** (bool) – whether to do deep conversion on nested (struct, list) types
- **str_as_json** (bool) – where to treat string data as json for nested types

Raises

- **ValueError** – if any value can't be converted to the datatype
- **NotImplementedError** – if any field type is not supported by Triad

Yield converted rows

Return type typing.Iterable[typing.List[typing.Any]]

triad.utils.pyarrow.**expression_to_schema**(*expr*)

Convert schema expression to *pyarrow.Schema*.

Format: *col_name:col_type[,col_name:col_type]+*

If col_type is a list type, the syntax should be *[element_type]*

If col_type is a struct type, the syntax should be *{col_name:col_type[,col_name:col_type]+}*

If col_type is a map type, the syntax should be *<key_type,value_type>*

Whitespaces will be removed. The format of the expression is json without any double quotes

Examples

```
expression_to_schema("a:int,b:int")
expression_to_schema("a:[int],b:{x:<int,int>,y:{z:[str],w:byte}}")
```

Parameters **expr** (str) – schema expression

Raises SyntaxError – if there is syntax issue or unknown types

Return type pyarrow.lib.Schema

Returns pyarrow.Schema

triad.utils.pyarrow.**get_alter_func**(*from_schema*, *to_schema*, *safe*)

Generate the alteration function based on `from_schema` and `to_schema`. This function can be applied to arrow tables with `from_schema`, the outout will be in `to_schema`'s order and types

Parameters

- **from_schema** (pyarrow.lib.Schema) – the source schema
- **to_schema** (pyarrow.lib.Schema) – the destination schema
- **safe** (bool) – whether to check for conversion errors such as overflow

Return type typing.Callable[[pyarrow.lib.Table], pyarrow.lib.Table]

Returns a function that can be applied to arrow tables with `from_schema`, the outout will be in `to_schema`'s order and types

triad.utils.pyarrow.**get_eq_func**(*data_type*)

Generate equality function for a give datatype

Parameters **data_type** (pyarrow.lib.DataType) – pyarrow data type supported by Triad

Return type typing.Callable[[typing.Any, typing.Any], bool]

Returns the function

triad.utils.pyarrow.**is_supported**(*data_type*, *throw=False*)

Whether *data_type* is currently supported by Triad

Parameters

- **data_type** (pyarrow.lib.DataType) – instance of pa.DataType
- **throw** (bool) – whether to raise exception if not supported

Return type bool

Returns if it is supported

triad.utils.pyarrow.**schema_to_expression**(*schema*)
Convert pyarrow.Schema to Triad schema expression see *expression_to_schema()*

Parameters **schema** (pyarrow.lib.Schema) – pyarrow schema

Raises **NotImplementedError** – if there some type is not supported by Triad

Return type pyarrow.lib.Schema

Returns schema string expression

triad.utils.pyarrow.**schemas_equal**(*a*, *b*, *check_order=True*, *check_metadata=True*)
check if two schemas are equal

Parameters

- **a** (pyarrow.lib.Schema) – first pyarrow schema
- **b** (pyarrow.lib.Schema) – second pyarrow schema
- **compare_order** – whether to compare order
- **compare_order** – whether to compare metadata

Return type bool

Returns if the two schema equal

triad.utils.pyarrow.**to_pa_datatype**(*obj*)
Convert an object to pyarrow DataType

Parameters **obj** (typing.Any) – any object

Raises **TypeError** – if unable to convert

Return type pyarrow.lib.DataType

Returns an instance of pd.DataType

triad.utils.pyarrow.**to_pandas_dtype**(*schema*, *use_extension_types=False*)
convert as *dtype* dict for pandas dataframes. Currently, struct type is not supported

Parameters

- **schema** (pyarrow.lib.Schema) – the pyarrow schema
- **use_extension_types** (bool) – whether to use pandas extension data types, defaults to False

Return type typing.Dict[str, numpy.dtype]

Returns the pandas data type dictionary

triad.utils.pyarrow.**to_single_pandas_dtype**(*pa_type*, *use_extension_types=False*)
convert a pyarrow data type to a pandas datatype. Currently, struct type is not supported

Parameters

- **schema** – the pyarrow schema
- **use_extension_types** (bool) – whether to use pandas extension data types, defaults to False

Return type typing.Dict[str, numpy.dtype]

Returns the pandas data type

1.2.11 triad.utils.rename

triad.utils.rename.**normalize_names**(*names*)

Normalize dataframe column names to follow Fugue column naming rules. It only operates on names that are not valid to Fugue.

It tries to minimize the changes to the original name. Special characters will be converted to _, but if this does not provide a valid and unique column name, more transformation will be done.

Note: This is a temporary solution before *Schema* can take arbitrary names

Examples

- `[0,1] => {0:"_0", 1:"_1"}`
- `["1a","2b"] => {"1a":"_1a", "2b":"_2b"}`
- `["*a","-a"] => {"*a":"_a", "-a":"_a_1"}`

Parameters **names** (typing.List[typing.Any]) – the columns names of a dataframe

Return type typing.Dict[typing.Any, str]

Returns the rename operations as a dict, key is the original column name, value is the new valid name.

1.2.12 triad.utils.schema

triad.utils.schema.**move_to_unquoted**(*expr*, *p*, *quote='`'*)

When p is on a quote, find the position next to the end of the quoted part

Parameters

- **expr** (str) – the original string
- **p** (int) – the current position of expr, and it should be a quote
- **quote** – the quote character

Raises SyntaxError – if there is an open quote detected

Return type int

Returns the position next to the end of the quoted part

triad.utils.schema.**quote_name**(*name*, *quote='`'*)

Add quote ` for strings that are not a valid triad var name.

Parameters

- **name** (str) – the name string
- **quote** (str) – the quote char, defaults to `

Return type str

Returns the quoted(if necessary) string

triad.utils.schema.**safe_replace_out_of_quote**(*s*, *find*, *replace*, *quote='`'*)

Replace strings out of the quoted part

Parameters

- **s** (str) – the original string
- **find** (str) – the string to find
- **replace** (str) – the string used to replace
- **quote** – the quote character

Return type str

Returns the string with the replacements

triad.utils.schema.**safe_search_out_of_quote**(*s*, *chars*, *quote='`'*)

Search for chars out of the quoted parts

Parameters

- **s** (str) – the original string
- **chars** (str) – the charaters to find
- **quote** – the quote character

Yield the tuple in format of `position, char`

Return type typing.Iterable[typing.Tuple[int, str]]

triad.utils.schema.**safe_split_and_unquote**(*s*, *sep_char=','*, *quote='`'*, *on_unquoted_empty='keep'*)

Split the string and unquote every part

Examples

```
" a , ` b ` , c " => ["a", " b ","c"]
```

Parameters

- **s** (str) – the original string
- **sep_char** (str) – the split character, defaults to ","
- **quote** (str) – the quote character
- **on_unquoted_empty** (str) – can be `keep`, `ignore` or `throw`, defaults to "keep"

Raises ValueError – if there are empty but unquoted parts and `on_unquoted_empty` is `throw`

Return type typing.List[str]

Returns the unquoted parts.

triad.utils.schema.**safe_split_out_of_quote**(*s*, *sep_chars*, *max_split=- 1*, *quote='`'*)

Return type typing.List[str]

triad.utils.schema.**split_quoted_string**(*s*, *quote='`'*)

Split `s` to a sequence of quoted and unquoted parts.

Parameters

- **s** (str) – the original string
- **quote** – the quote character

Yield the tuple in the format of `is_quoted, start, end`

Return type typing.Iterable[typing.Tuple[bool, int, int]]

triad.utils.schema.**unquote_name**(*name*, *quote='`'*)

If the input is quoted, then get the inner string, otherwise do nothing.

Parameters

- **name** (str) – the name string
- **quote** (str) – the quote char, defaults to `

Return type str

Returns the value without `

1.2.13 triad.utils.string

triad.utils.string.**assert_triad_var_name**(*expr*)

Check if *expr* is a valid Triad variable name based on Triad standard: it has to be a valid python identifier and it can't be purely _ :type expr: str :param expr: column name expression :raises AssertionError: if the expression is invalid :rtype: str :return: the expression string

triad.utils.string.**validate_triad_var_name**(*expr*)

Check if *expr* is a valid Triad variable name based on Triad standard: it has to be a valid python identifier and it can't be purely _

Note: Any valid triad var name can be used as column names without quote ````

Parameters **expr** (str) – column name expression

Return type bool

Returns whether it is valid

1.2.14 triad.utils.threading

class triad.utils.threading.**RunOnce**(*func*, *key_func=None*, *lock_type=<function RLock>*)

Bases: object

Run *func* once, the uniqueness is defined by *key_func*. This implementation is serialization safe and thread safe.

Note: Please use the decorator *run_once()* instead of directly using this class

---

Parameters

- **func** (typing.Callable) – the function to run only once with this wrapper instance
- **key_func** (typing.Optional[typing.Callable]) – the unique key determined by arguments of *func*, if not set, it will use the same hasing logic as **:external+python:func:`functools.lru_cache`**
- **lock_type** (typing.Type) – lock class type for thread safe

class triad.utils.threading.**SerializableRLock**

Bases: object

A serialization safe wrapper of **:external+python:class:`threading.RLock`**

triad.utils.threading.**run_once**(*func=None*, *key_func=None*, *lock_type=<function RLock>*)

The decorator to run *func* once, the uniqueness is defined by *key_func*. This implementation is serialization safe and thread safe.

Parameters

- **func** (typing.Optional[typing.Callable]) – the function to run only once with this wrapper instance
- **key_func** (typing.Optional[typing.Callable]) – the unique key determined by arguments of *func*, if not set, it will use the same hasing logic as **:external+python:func:`functools.lru_cache`**
- **lock_type** (typing.Type) – lock class type for thread safe, it doesn't need to be serialization safe

---

Examples

```
@run_once
def r(a):
    return max(a)

a1 = [0, 1]
a2 = [0, 2]
assert 1 == r(a1) # will trigger r
assert 1 == r(a1) # will get the result from cache
assert 2 == r(a2) # will trigger r again because of different arguments

# the following example ignores arguments
@run_once(key_func=lambda *args, **kwargs: True)
def r2(a):
    return max(a)

assert 1 == r(a1) # will trigger r
assert 1 == r(a2) # will get the result from cache
```

Note:

- Hash collision is the concern of the user, not this class, your *key_func* should avoid any potential collision
- *func* can have no return
- For concurrent calls of this wrapper, only one will trigger *func* other calls will be blocked until the first call returns an result
- This class is cloudpicklable, but unpickled instance does NOT share the same context with the original one
- This is not to replace **:external+python:func:`functools.lru_cache`**, it is not supposed to cache a lot of items

---

Return type typing.Callable

1.3 triad.constants

1.4 triad.exceptions

exception triad.exceptions.**InvalidOperationError**(*message=''*)

Bases: Exception

Exception on invalid operations

exception triad.exceptions.**NoneArgumentError**(*message*)

Bases: ValueError

Exception on None argument

PYTHON MODULE INDEX

t

triad.collections.dict, 1
triad.collections.fs, 4
triad.collections.function_wrapper, 5
triad.collections.schema, 7
triad.constants, 34
triad.exceptions, 34
triad.utils.assertion, 10
triad.utils.class_extension, 11
triad.utils.convert, 13
triad.utils.dispatcher, 17
triad.utils.entry_points, 22
triad.utils.hash, 22
triad.utils.iter, 22
triad.utils.json, 24
triad.utils.pandas_like, 24
triad.utils.pyarrow, 27
triad.utils.rename, 30
triad.utils.schema, 30
triad.utils.string, 32
triad.utils.threading, 32

INDEX

A

annotated_param() (*triad.collections.function_wrapper.FunctionWrapper class method*), 6
AnnotatedParam (*class in triad.collections.function_wrapper*), 5
append() (*triad.collections.schema.Schema method*), 8
apply_schema() (*in module triad.utils.pyarrow*), 27
as_array_iterable() (*triad.utils.pandas_like.PandasLikeUtils method*), 24
as_arrow() (*triad.utils.pandas_like.PandasLikeUtils method*), 25
as_type() (*in module triad.utils.convert*), 13
assert_arg_not_none() (*in module triad.utils.assertion*), 10
assert_not_empty() (*triad.collections.schema.Schema method*), 8
assert_or_throw() (*in module triad.utils.assertion*), 10
assert_triad_var_name() (*in module triad.utils.string*), 32

C

candidate() (*triad.utils.dispatcher.ConditionalDispatcher method*), 18
check_for_duplicate_keys() (*in module triad.utils.json*), 24
clear() (*triad.collections.dict.IndexedOrderedDict method*), 1
conditional_broadcaster() (*in module triad.utils.dispatcher*), 19
conditional_dispatcher() (*in module triad.utils.dispatcher*), 20
ConditionalDispatcher (*class in triad.utils.dispatcher*), 17
copy() (*triad.collections.dict.IndexedOrderedDict method*), 1
copy() (*triad.collections.schema.Schema method*), 8
create_fs() (*triad.collections.fs.FileSystem method*), 4

E

empty (*triad.utils.iter.EmptyAwareIterable property*), 22
empty() (*triad.utils.pandas_like.PandasLikeUtils method*), 25
EmptyAwareIterable (*class in triad.utils.iter*), 22
enforce_type() (*triad.utils.pandas_like.PandasLikeUtils method*), 25
ensure_compatible() (*triad.utils.pandas_like.PandasLikeUtils method*), 25
equals() (*triad.collections.dict.IndexedOrderedDict method*), 1
exclude() (*triad.collections.schema.Schema method*), 8
expression_to_schema() (*in module triad.utils.pyarrow*), 27
extensible_class() (*in module triad.utils.class_extension*), 11
extension_method() (*in module triad.utils.class_extension*), 12
extract() (*triad.collections.schema.Schema method*), 8

F

fields (*triad.collections.schema.Schema property*), 8
FileSystem (*class in triad.collections.fs*), 4
fillna_default() (*triad.utils.pandas_like.PandasLikeUtils method*), 26
function_wrapper() (*in module triad.collections.function_wrapper*), 7
FunctionWrapper (*class in triad.collections.function_wrapper*), 5

G

get() (*triad.collections.dict.ParamDict method*), 3
get_alter_func() (*in module triad.utils.pyarrow*), 28
get_caller_global_local_vars() (*in module triad.utils.convert*), 13
get_eq_func() (*in module triad.utils.pyarrow*), 28
get_full_type_path() (*in module triad.utils.convert*), 14
get_item_by_index() (*triad.collections.dict.IndexedOrderedDict method*), 1
get_key_by_index() (*triad.collections.dict.IndexedOrderedDict method*), 1

get_or_none() (*triad.collections.dict.ParamDict method*), 3
get_or_throw() (*triad.collections.dict.ParamDict method*), 3
get_value_by_index() (*triad.collections.dict.IndexedOrderedDict method*), 1
glob (*triad.collections.fs.FileSystem property*), 4

I

IGNORE (*triad.collections.dict.ParamDict attribute*), 3
index_of_key() (*triad.collections.dict.IndexedOrderedDict method*), 2
IndexedOrderedDict (*class in triad.collections.dict*), 1
input_code (*triad.collections.function_wrapper.FunctionWrapper property*), 6
intersect() (*triad.collections.schema.Schema method*), 9
InvalidOperationError, 34
is_compatile_index() (*triad.utils.pandas_like.PandasLikeUtils method*), 26
is_supported() (*in module triad.utils.pyarrow*), 28

K

KeywordParam (*class in triad.collections.function_wrapper*), 6

L

loads_no_dup() (*in module triad.utils.json*), 24

M

make_empty_aware() (*in module triad.utils.iter*), 23
makedirs() (*triad.collections.fs.FileSystem method*), 5
module
    triad.collections.dict, 1
    triad.collections.fs, 4
    triad.collections.function_wrapper, 5
    triad.collections.schema, 7
    triad.constants, 34
    triad.exceptions, 34
    triad.utils.assertion, 10
    triad.utils.class_extension, 11
    triad.utils.convert, 13
    triad.utils.dispatcher, 17
    triad.utils.entry_points, 22
    triad.utils.hash, 22
    triad.utils.iter, 22
    triad.utils.json, 24
    triad.utils.pandas_like, 24
    triad.utils.pyarrow, 27
    triad.utils.rename, 30
    triad.utils.schema, 30
    triad.utils.string, 32
    triad.utils.threading, 32
move_to_end() (*triad.collections.dict.IndexedOrderedDict method*), 2
move_to_unquoted() (*in module triad.utils.schema*), 30

N

names (*triad.collections.schema.Schema property*), 9
NoneArgumentError, 34
NoneParam (*class in triad.collections.function_wrapper*), 7
normalize_names() (*in module triad.utils.rename*), 30

O

OtherParam (*class in triad.collections.function_wrapper*), 7
output_code (*triad.collections.function_wrapper.FunctionWrapper property*), 6
OVERWRITE (*triad.collections.dict.ParamDict attribute*), 3

P

pa_schema (*triad.collections.schema.Schema property*), 9
pandas_dtype (*triad.collections.schema.Schema property*), 9
PandasLikeUtils (*class in triad.utils.pandas_like*), 24
PandasUtils (*class in triad.utils.pandas_like*), 26
ParamDict (*class in triad.collections.dict*), 2
parse_annotation() (*triad.collections.function_wrapper.FunctionWrapper class method*), 6
partition() (*triad.utils.pyarrow.SchemaedDataPartitioner method*), 27
pd_dtype (*triad.collections.schema.Schema property*), 9
peek() (*triad.utils.iter.EmptyAwareIterable method*), 23
pop() (*triad.collections.dict.IndexedOrderedDict method*), 2
pop_by_index() (*triad.collections.dict.IndexedOrderedDict method*), 2
popitem() (*triad.collections.dict.IndexedOrderedDict method*), 2
PositionalParam (*class in triad.collections.function_wrapper*), 7
pyarrow_schema (*triad.collections.schema.Schema property*), 9

Q

quote_name() (*in module triad.utils.schema*), 30

R

readonly (*triad.collections.dict.IndexedOrderedDict property*), 2
register() (*triad.utils.dispatcher.ConditionalDispatcher method*), 18

remove() (*triad.collections.schema.Schema method*), 9
rename() (*triad.collections.schema.Schema method*), 9
run() (*triad.utils.dispatcher.ConditionalDispatcher method*), 19
run_at_def() (*in module triad.utils.dispatcher*), 21
run_once() (*in module triad.utils.threading*), 33
run_top() (*triad.utils.dispatcher.ConditionalDispatcher method*), 19
RunOnce (*class in triad.utils.threading*), 32

S

safe_groupby_apply() (*triad.utils.pandas_like.PandasLikeUtils method*), 26
safe_replace_out_of_quote() (*in module triad.utils.schema*), 30
safe_search_out_of_quote() (*in module triad.utils.schema*), 31
safe_split_and_unquote() (*in module triad.utils.schema*), 31
safe_split_out_of_quote() (*in module triad.utils.schema*), 31
Schema (*class in triad.collections.schema*), 7
schema_to_expression() (*in module triad.utils.pyarrow*), 28
SchemaedDataPartitioner (*class in triad.utils.pyarrow*), 27
SchemaError, 10
schemas_equal() (*in module triad.utils.pyarrow*), 29
SelfParam (*class in triad.collections.function_wrapper*), 7
SerializableRLock (*class in triad.utils.threading*), 32
set_readonly() (*triad.collections.dict.IndexedOrderedDict method*), 2
set_value_by_index() (*triad.collections.dict.IndexedOrderedDict method*), 2
slice() (*triad.utils.iter.Slicer method*), 23
slice_iterable() (*in module triad.utils.iter*), 23
Slicer (*class in triad.utils.iter*), 23
split_quoted_string() (*in module triad.utils.schema*), 31
str_to_instance() (*in module triad.utils.convert*), 14
str_to_object() (*in module triad.utils.convert*), 14
str_to_type() (*in module triad.utils.convert*), 15

T

THROW (*triad.collections.dict.ParamDict attribute*), 3
to_bool() (*in module triad.utils.convert*), 15
to_datetime() (*in module triad.utils.convert*), 16
to_function() (*in module triad.utils.convert*), 16
to_instance() (*in module triad.utils.convert*), 16
to_json() (*triad.collections.dict.ParamDict method*), 4
to_kv_iterable() (*in module triad.utils.iter*), 24
to_pa_datatype() (*in module triad.utils.pyarrow*), 29
to_pandas_dtype() (*in module triad.utils.pyarrow*), 29
to_schema() (*triad.utils.pandas_like.PandasLikeUtils method*), 26
to_single_pandas_dtype() (*in module triad.utils.pyarrow*), 29
to_size() (*in module triad.utils.convert*), 16
to_timedelta() (*in module triad.utils.convert*), 17
to_type() (*in module triad.utils.convert*), 17
to_uuid() (*in module triad.utils.hash*), 22
transform() (*triad.collections.schema.Schema method*), 9
triad.collections.dict
    module, 1
triad.collections.fs
    module, 4
triad.collections.function_wrapper
    module, 5
triad.collections.schema
    module, 7
triad.constants
    module, 34
triad.exceptions
    module, 34
triad.utils.assertion
    module, 10
triad.utils.class_extension
    module, 11
triad.utils.convert
    module, 13
triad.utils.dispatcher
    module, 17
triad.utils.entry_points
    module, 22
triad.utils.hash
    module, 22
triad.utils.iter
    module, 22
triad.utils.json
    module, 24
triad.utils.pandas_like
    module, 24
triad.utils.pyarrow
    module, 27
triad.utils.rename
    module, 30
triad.utils.schema
    module, 30
triad.utils.string
    module, 32
triad.utils.threading
    module, 32
types (*triad.collections.schema.Schema property*), 10

U

union() (*triad.collections.schema.Schema method*), 10
union_with() (*triad.collections.schema.Schema method*), 10
unquote_name() (*in module triad.utils.schema*), 32
update() (*triad.collections.dict.ParamDict method*), 4

V

validate_triad_var_name() (*in module triad.utils.string*), 32