Nano-Utils Documentation

Release 2.3.5

B. F. van Beek

Sep 21, 2023

CONTENTS

CHAPTER

ONE

NANO-UTILS

Utility functions used throughout the various nlesc-nano repositories.

1.1 Installation

- PyPi: `pip install Nano-Utils`
- GitHub: `pip install git+https://github.com/nlesc-nano/Nano-Utils@master`

1.2 API

1.2.1 nanoutils.utils

General utility functions.

Index

PartialPrepend	A `partial()` subclass where the `*args` are appended rather than prepended.
SetAttr(obj, name, value)	A context manager for temporarily changing an attribute's value.
VersionInfo(major[, minor, micro])	A `namedtuple()` representing the version of a package.
group_by_values(iterable)	Take an iterable, yielding 2-tuples, and group all first elements by the second.
get_importable(string[, validate])	Get an importable object.
construct_api_doc(glob_dict[, decorators])	Format a **Nano-Utils** module-level docstring.
split_dict(dct[, preserve_order, keep_keys, ...])	Pop all items from **dct** which are in not in **keep_keys** and use them to construct a new dictionary.
get_func_name(func[, prepend_module, ...])	Extract and return the name of **func**.
set_docstring(docstring)	A decorator for assigning docstrings.
raise_if(exception)	A decorator which raises the passed exception whenever calling the decorated function.
ignore_if(exception[, warn])	A decorator which, if an exception is passed, ignores calls to the decorated function.
SequenceView(sequence)	A read-only view of an underlying `Sequence`.
CatchErrors(*exceptions)	A re-usable context manager for catching and storing all exceptions of a given type.
LazyImporter(module, imports)	A class for lazilly importing objects.
MutableLazyImporter(module, imports)	A subclass of *LazyImporter* with mutable `imports`.
positional_only(func)	A decorator for converting mangled parameters to positional-only.
UserMapping([iterable])	Base class for user-defined immutable mappings.
MutableUserMapping([iterable])	Base class for user-defined mutable mappings.
warning_filter(action[, message, category, ...])	A decorator for wrapping function calls with `warnings.filterwarnings()`.

API

class nanoutils.**PartialPrepend**

A `partial()` subclass where the `*args` are appended rather than prepended.

Examples

```
>>> from functools import partial
>>> from nanoutils import PartialPrepend

>>> func1 = partial(isinstance, 1)  # isinstance(1, ...)
>>> func2 = PartialPrepend(isinstance, float)  # isinstance(..., float)

>>> func1(int)  # isinstance(1, int)
True

>>> func2(1.0)  # isinstance(1.0, float)
True
```

class nanoutils.**SetAttr**(*obj*, *name*, *value*)

A context manager for temporarily changing an attribute's value.

The *SetAttr* context manager is thread-safe, reusable and reentrant.

> **Warning:** Note that while *SetAttr.__enter__()* and *SetAttr.__exit__()* are thread-safe, the same does *not* hold for *SetAttr.__init__()*.

Examples

```
>>> from nanoutils import SetAttr

>>> class Test:
...     a = False

>>> print(Test.a)
False

>>> set_attr = SetAttr(Test, 'a', True)
>>> with set_attr:
...     print(Test.a)
True
```

property obj

Get the to-be modified object.

Type
object

property name

Get the name of the to-be modified attribute.

Type
str

property value

Get the value to-be assigned to the *name* attribute of *SetAttr.obj*.

Type
object

property attr

Get or set the *name* attribute of *SetAttr.obj*.

Type
object

class nanoutils.**VersionInfo**(*major*, *minor=0*, *micro=0*)

A namedtuple() representing the version of a package.

Examples

```
>>> from nanoutils import VersionInfo

>>> version = '0.8.2'
>>> VersionInfo.from_str(version)
VersionInfo(major=0, minor=8, micro=2)
```

---

major

The semantic major version.

Type
int

minor

The semantic minor version.

Type
int

micro

The semantic micro version.

Type
int

property patch

An alias for *VersionInfo.micro*.

Type
int

property maintenance

An alias for *VersionInfo.micro*.

Type
int

property bug

An alias for *VersionInfo.micro*.

Type
int

classmethod from_str(*version*, *, *fullmatch=True*)

Construct a *VersionInfo* from a string.

Parameters

- **version** (str) – A PEP 440-compatible version string(*e.g.* `version = "0.8.2"`). Note that version representations are truncated at up to three integers.
- **fullmatch** (bool) – Whether to perform a full or partial match on the passed string.

Returns
A new VersionInfo instance.

Return type
nanoutils.VersionInfo

See also:

PEP 440
This PEP describes a scheme for identifying versions of Python software distributions, and declaring dependencies on particular versions.

nanoutils.**group_by_values**(*iterable*)

Take an iterable, yielding 2-tuples, and group all first elements by the second.

Examples

```
>>> str_list: list = ['a', 'a', 'a', 'a', 'a', 'b', 'b', 'b', 'c']
>>> iterator = enumerate(str_list, 1)

>>> new_dict: dict = group_by_values(iterator)
>>> print(new_dict)
{'a': [1, 2, 3, 4, 5], 'b': [6, 7, 8], 'c': [9]}
```

Parameters
iterable (Iterable[tuple[VT, KT]]) – An iterable yielding 2 elements upon iteration (*e.g.* dict.items() or enumerate()). The second element must be Hashable and will be used as key in the to-be returned mapping.

Returns
A grouped dictionary.

Return type
dict[KT, list[VT]]

nanoutils.**get_importable**(*string*, *validate=None*)

Get an importable object.

Examples

```
>>> from inspect import isclass
>>> from nanoutils import get_importable

>>> dict_type = get_importable('builtins.dict', validate=isclass)
>>> print(dict_type)
<class 'dict'>
```

Parameters

- **string** (str) – A string representing an importable object. Note that the string *must* contain the object's module.
- **validate** (Callable, optional) – A callable for validating the imported object. Will raise a RuntimeError if its output evaluates to False.

Returns
The imported object

Return type
object

nanoutils.**construct_api_doc**(*glob_dict*, *decorators=frozenset({})*)

Format a **Nano-Utils** module-level docstring.

Examples

```
>>> __doc__ = """
... Index
... -----
... .. autosummary::
... {autosummary}
...
... API
... ---
... {autofunction}
...
... """

>>> from nanoutils import construct_api_doc

>>> __all__ = ['obj', 'func', 'Class']

>>> obj = ...

>>> def func(obj: object) -> None:
...     pass

>>> class Class(object):
...     pass

>>> doc = construct_api_doc(locals())
>>> print(doc)

Index
-----
.. autosummary::
    obj
    func
    Class

API
---
.. autodata:: obj
.. autofunction:: func
.. autoclass:: Class
    :members:
```

Parameters

- **glob_dict** (Mapping[str, object]) – A mapping containg a module-level namespace. Note that the mapping *must* contain the "__doc__" and "__all__" keys.
- **decorators** (Container[str]) – A container with the names of all decorators. If not

specified, all functions will use the Sphinx `autofunction` domain.

Returns
The formatted string.

Return type
`str`

nanoutils.**split_dict**(*dct*, *preserve_order=False*, ***, *keep_keys=None*, *disgard_keys=None*)

Pop all items from **dct** which are in not in **keep_keys** and use them to construct a new dictionary.

Note that, by popping its keys, the passed **dct** will also be modified inplace.

Examples

```
>>> from nanoutils import split_dict

>>> dict1 = {1: 'a', 2: 'b', 3: 'c', 4: 'd'}
>>> dict2 = split_dict(dict1, keep_keys={1, 2})

>>> print(dict1, dict2, sep='\n')
{1: 'a', 2: 'b'}
{3: 'c', 4: 'd'}

>>> dict3 = split_dict(dict1, disgard_keys={1, 2})
>>> print(dict1, dict3, sep='\n')
{}
{1: 'a', 2: 'b'}
```

Parameters

- **dct** (`MutableMapping[KT, VT]`) – A mutable mapping.
- **preserve_order** (`bool`) – If `True`, preserve the order of the items in **dct**. Note that `preserve_order = False` is generally faster.
- **keep_keys** (`Iterable[KT]`, keyword-only) – An iterable with keys that should remain in **dct**. Note that **keep_keys** and **disgard_keys** are mutually exclusive.
- **disgard_keys** (`Iterable[KT]`, keyword-only) – An iterable with keys that should be removed from **dct**. Note that **disgard_keys** and **keep_keys** are mutually exclusive.

Returns
A new dictionaries with all key/value pairs from **dct** not specified in **keep_keys**.

Return type
`dict[KT, VT]`

nanoutils.**get_func_name**(*func*, *prepend_module=False*, *repr_fallback=False*)

Extract and return the name of **func**.

A total of three attempts are performed at retrieving the passed functions name:

1. Return the functions qualified name (`__qualname__`).
2. Return the functions name (`__name__`).
3. Return the (called) name of the functions type.

Examples

```
>>> from functools import partial
>>> from nanoutils import get_func_name

>>> def func1():
...     pass

>>> class Class():
...     def func2(self):
...         pass

>>> func3 = partial(len)

>>> get_func_name(func1)
'func1'

>>> get_func_name(func1, prepend_module=True)
'__main__.func1'

>>> get_func_name(Class.func2)
'Class.func2'

>>> get_func_name(func3)
'partial(...)'

>>> get_func_name(func3, repr_fallback=True)
'functools.partial(<built-in function len>)'
```

Parameters

- **func** (*Callable*) – A callable object.
- **prepend_module** (*bool*) – If `True` prepend the objects module (`__module__`), if available, to the to-be returned string.
- **repr_fallback** (*bool*) – By default, when the passed function has neither a `__qualname__` or `__name__` attribute the (called) name of the functions class is returned. If `True` then use `repr()` instead.

Returns

A string representation of the name of **func**.

Return type

str

`@nanoutils.`**set_docstring**

A decorator for assigning docstrings.

Examples

```
>>> from nanoutils import set_docstring
```

(continues on next page)

(continued from previous page)

```
>>> number = "#10"

>>> @set_docstring(f"Fancy docstring {number}.")
... def func():
...     pass

>>> print(func.__doc__)
Fancy docstring #10.
```

Parameters

docstring (str, optional) – The to-be assigned docstring.

@nanoutils.**raise_if**

A decorator which raises the passed exception whenever calling the decorated function.

If **exception** is None then the decorated function will be called as usual.

Examples

```
>>> from nanoutils import raise_if

>>> ex1 = None
>>> ex2 = TypeError("This is an exception")

>>> @raise_if(ex1)
... def func1() -> bool:
...     return True

>>> @raise_if(ex2)
... def func2() -> bool:
...     return True

>>> func1()
True

>>> func2()
Traceback (most recent call last):
  ...
TypeError: This is an exception
```

Parameters

exception (BaseException, optional) – An exception. If None is passed then the decorated function will be called as usual.

See also:

nanoutils.ignore_if()

A decorator which, if an exception is passed, ignores calls to the decorated function.

@nanoutils.**ignore_if**(*exception*, *warn=True*)

A decorator which, if an exception is passed, ignores calls to the decorated function.

If **exception** is `None` then the decorated function will be called as usual.

Examples

```
>>> import warnings
>>> from nanoutils import ignore_if

>>> ex1 = None
>>> ex2 = TypeError("This is an exception")

>>> @ignore_if(ex1)
... def func1() -> bool:
...     return True

>>> @ignore_if(ex2)
... def func2() -> bool:
...     return True

>>> func1()
True

>>> func2()

# Catch the warning and a raise it as an exception
>>> with warnings.catch_warnings():
...     warnings.simplefilter("error", UserWarning)
...     func2()
Traceback (most recent call last):
  ...
UserWarning: Skipping call to func2()
```

Parameters

- **exception** (`BaseException`, optional) – An exception. If `None` is passed then the decorated function will be called as usual.
- **warn** (`bool`) – If `True` issue a `UserWarning` whenever calling the decorated function

See also:

nanoutils.raise_if()
A decorator which raises the passed exception whenever calling the decorated function.

class nanoutils.**SequenceView**(*sequence*)

A read-only view of an underlying `Sequence`.

Examples

```
>>> from nanoutils import SequenceView

>>> lst = [1, 2, 3]
>>> view = SequenceView(lst)
>>> print(view)
SequenceView([1, 2, 3])

>>> lst.append(4)
>>> print(view)
SequenceView([1, 2, 3, 4])

>>> print(len(view))
4

>>> del view[0]
Traceback (most recent call last):
    ...
TypeError: 'SequenceView' object doesn't support item deletion
```

---

pprint_kwargs = {'compact': True, 'sort_dicts': False, 'width': 67}

A class variable containing a dictionary with keyword arguments for `pprint.pformat()`.

index(*value*, *start=0*, *stop=9223372036854775807*, */*)

Return the first index of **value**.

count(*value*, */*)

Return the number of times **value** occurs in the instance.

class nanoutils.**CatchErrors**(**exceptions*)

A re-usable context manager for catching and storing all exceptions of a given type.

Examples

```
>>> from nanoutils import CatchErrors

>>> context = CatchErrors(AssertionError)

>>> for i in range(3):
...     with context as exc_view:
...         assert False, i
...         print(exc_view)
SequenceView([AssertionError(0)])
SequenceView([AssertionError(0), AssertionError(1)])
SequenceView([AssertionError(0), AssertionError(1), AssertionError(2)])
```

See also:

contextlib.suppress

Context manager to suppress specified exceptions.

property exceptions

Get the to-be caught exception types.

property caught_exceptions

Get a read-only view of all caught exceptions.

clear()

Clear all caught exceptions.

class nanoutils.**LazyImporter**(*module*, *imports*)

A class for lazilly importing objects.

Parameters

- **module** (types.ModuleType) – The to-be wrapped module.
- **imports** (Mapping[str, str]) – A mapping that maps the names of to-be lazzily imported objects to the names of their modules.

Examples

```
>>> from nanoutils import LazyImporter

>>> __getattr__ = LazyImporter.from_name("nanoutils", {"Any": "typing"})
>>> print(__getattr__)
LazyImporter(module=nanoutils, imports={'Any': 'typing'})

>>> __getattr__("Any")
typing.Any
```

---

property module

Get the wrapped module.

Type
types.ModuleType

property imports

Get a mapping that maps object names to their module name.

Type
types.MappingProxyType[str, str]

classmethod from_name(*name*, *imports*)

Construct a new instance from the module **name**.

Parameters

- **name** (str) – The name of the to-be wrapped module.
- **imports** (Mapping[str, str]) – A mapping that maps the names of to-be lazzily imported objects to the names of their modules.

Returns
A new LazyImporter instance or a subclass thereof.

Return type
nanoutils.LazyImporter

class nanoutils.**MutableLazyImporter**(*module*, *imports*)

A subclass of *LazyImporter* with mutable *imports*.

Parameters

- **module** (types.ModuleType) – The to-be wrapped module.
- **imports** (Mapping[str, str]) – A mapping that maps the names of to-be lazzily imported objects to the names of their modules.

Examples

```
>>> from nanoutils import MutableLazyImporter

>>> __getattr__ = MutableLazyImporter.from_name("nanoutils", {"Any": "typing"})
>>> print(__getattr__)
MutableLazyImporter(module=nanoutils, imports={'Any': 'typing'})

>>> __getattr__("Any")
typing.Any

>>> del __getattr__.imports["Any"]
>>> print(__getattr__)
MutableLazyImporter(module=nanoutils, imports={})

>>> __getattr__.imports = {"Hashable": "collections.abc"}
>>> __getattr__("Hashable")
<class 'collections.abc.Hashable'>
```

property imports

Get or set the dictionary that maps object names to their module name.

Setting a value will assign it as a copy.

Type
dict[str, str]

@nanoutils.**positional_only**

A decorator for converting mangled parameters to positional-only.

Sets the `__signature__` attribute of the decorated function.

Examples

```
>>> from nanoutils import positional_only
>>> import inspect

>>> def func1(__a, b=None):
...     pass

>>> print(inspect.signature(func1))
(__a, b=None)

>>> @positional_only
... def func2(__a, b=None):
...     pass

>>> print(inspect.signature(func2))
(a, /, b=None)
```

Parameters

func (Callable) – The to-be decorated function whose `__signature__` attribute will be added or updated.

class nanoutils.**UserMapping**(*iterable=None*, */*, ***kwargs*)

Base class for user-defined immutable mappings.

copy()

Return a deep copy of this instance.

keys()

Return a set-like object containing all keys.

items()

Return a set-like object containing all key/value pairs.

values()

Return a collection containing all values.

get(*key*, *default=None*)

Return the value for key if the key is present, else default.

classmethod fromkeys(*iterable*, *value=None*)

Create a new dictionary with keys from iterable and values set to default.

class nanoutils.**MutableUserMapping**(*iterable=None*, */*, ***kwargs*)

Base class for user-defined mutable mappings.

clear()

Remove all items from the mapping.

popitem()

Remove and return a (key, value) pair as a 2-tuple.

Pairs are returned in LIFO (last-in, first-out) order. Raises a KeyError if the mapping is empty.

pop(*key*, *default=<object object>*)

Remove the specified key and return the corresponding value.

If the key is not found, default is returned if given, otherwise a KeyError is raised.

update(*iterable=None*, */*, ***kwargs*)

Update the mapping from the passed mapping or iterable.

@nanoutils.**warning_filter**(*action*, *message=''*, *category=<class 'Warning'>*, *module=''*, *lineno=0*, *append=False*)

A decorator for wrapping function calls with `warnings.filterwarnings()`.

Examples

```
>>> from nanoutils import warning_filter
>>> import warnings

>>> @warning_filter("error", category=UserWarning)
... def func():
```

(continues on next page)

(continued from previous page)

```
...     warnings.warn("test", UserWarning)

>>> func()
Traceback (most recent call last):
    ...
UserWarning: test
```

Parameters

- **action** (str) – One of the following strings:
  - "default": Print the first occurrence of matching warnings for each location (module + line number) where the warning is issued
  - "error": Turn matching warnings into exceptions
  - "ignore": Never print matching warnings
  - "always": Always print matching warnings
  - "module": Print the first occurrence of matching warnings for each module where the warning is issued (regardless of line number)
  - "once": Print only the first occurrence of matching warnings, regardless of location
- **message** (str, optional) – A string containing a regular expression that the start of the warning message must match. The expression is compiled to always be case-insensitive.
- **category** (type[Warning]) – The to-be affected Warning (sub-)class.
- **module** (str, optional) – A string containing a regular expression that the module name must match. The expression is compiled to be case-sensitive.
- **lineno** (int) – An integer that the line number where the warning occurred must match, or 0 to match all line numbers.
- **append** (bool) – Whether the warning entry is inserted at the end.

See also:

warnings.filterwarnings()
: Insert a simple entry into the list of warnings filters (at the front).

1.2.2 nanoutils.empty

A module with empty (immutable) containers and iterables.

Can be used as default arguments for functions.

Index

EMPTY_CONTAINER	An empty Container.
EMPTY_COLLECTION	An empty Collection.
EMPTY_SET	An empty Set.
EMPTY_SEQUENCE	An empty Sequence.
EMPTY_MAPPING	An empty Mapping.

API

nanoutils.**EMPTY_CONTAINER: Container = frozenset()**

An immutable empty Container.

nanoutils.**EMPTY_COLLECTION: Collection = frozenset()**

An immutable empty Collection.

nanoutils.**EMPTY_SET: Set = frozenset()**

An immutable empty Set.

nanoutils.**EMPTY_SEQUENCE: Sequence = tuple()**

An immutable empty Sequence.

nanoutils.**EMPTY_MAPPING: Mapping = types.MappingProxyType({})**

An immutable empty Mapping.

1.2.3 nanoutils.schema

A module with schema-related utility functions.

See also:

schema is a library for validating Python data structures, such as those obtained from config-files, forms, external services or command-line parsing, converted from JSON/YAML (or something else) to Python data-types.

Index

Default(value[, call])	A validation class akin to the likes of schemas.Use.
Formatter(msg)	A str subclass used for creating schema error messages.
supports_float(value)	Check if a float-like object has been passed (SupportsFloat).
supports_int(value)	Check if an int-like object has been passed (SupportsInt).
isinstance_factory(class_or_tuple[, module])	Return a function which checks if the passed object is an instance of **class_or_tuple**.
issubclass_factory(class_or_tuple[, module])	Return a function which checks if the passed class is a subclass of **class_or_tuple**.
import_factory(validate[, module])	Return a function which calls *nanoutils.get_importable()* with the **validate** argument.

API

class nanoutils.**Default**(*value*, *call=True*)

A validation class akin to the likes of `schemas.Use`.

Upon executing *Default.validate()* returns the stored *value*. If *call* is `True` and the value is a callable, then it is called before its return.

Examples

```
>>> from schema import Schema, And
>>> from nanoutils import Default

>>> schema1 = Schema(And(int, Default(True)))
>>> schema1.validate(1)
True

>>> schema2 = Schema(And(int, Default(dict)))
>>> schema2.validate(1)
{}

>>> schema3 = Schema(And(int, Default(dict, call=False)))
>>> schema3.validate(1)
<class 'dict'>
```

value

The to-be return value for when *Default.validate()* is called. If *Default.call* is `True` then the value is called (if possible) before its return.

Type
object

call

Whether to call *Default.value* before its return (if possible) or not.

Type
bool

validate(**args*, ***kwargs*)

Validate the passed **data**.

Parameters
***args/**kwargs** – Variadic (keyword) arguments to ensure signature compatibility. Supplied values will not be used.

Returns
Return *Default.value*. The to-be returned value will be called if it is a callable and *Default.call* is `True`.

Return type
object

class nanoutils.**Formatter**(*msg*)

A `str` subclass used for creating `schema` error messages.

Examples

```
>>> from nanoutils import Formatter

>>> string = Formatter("{name}: {type} = {value}")
>>> string.format(1)
'value: int = 1'
```

---

format(*obj*)

Return a formatted version of `Formatter._msg`.

Parameters
obj (`object`) – The to-be formatted object.

Returns
A formatted string.

Return type
`str`

nanoutils.**supports_float**(*value*)

Check if a float-like object has been passed (`SupportsFloat`).

Examples

```
>>> from nanoutils import supports_float

>>> supports_float(1.0)
True

>>> supports_float(1)
True

>>> supports_float('1.0')
True

>>> supports_float('not a float')
False
```

Parameters
value (`object`) – The to-be evaluated object.

Returns
Whether or not the passed **value** is float-like or not.

Return type
`bool`

nanoutils.**supports_int**(*value*)

Check if an int-like object has been passed (`SupportsInt`).

Examples

```
>>> from nanoutils import supports_int

>>> supports_int(1.0)
True

>>> supports_int(1.5)
False

>>> supports_int(1)
True

>>> supports_int('1')
True

>>> supports_int('not a int')
False
```

Parameters
value (object) – The to-be evaluated object.

Returns
Whether or not the passed **value** is int-like or not.

Return type
bool

nanoutils.**isinstance_factory**(*class_or_tuple*, *module='nanoutils.schema'*)

Return a function which checks if the passed object is an instance of **class_or_tuple**.

Examples

```
>>> from nanoutils import isinstance_factory

>>> func = isinstance_factory(int)

>>> func(1)  # isinstance(1, int)
True

>>> func(1.0)  # isinstance(1.0, int)
False
```

Parameters

- **class_or_tuple** (type or Tuple[type, ...]) – A type object or tuple of type objects.
- **module** (str) – The __module__ of the to-be returned function.

Returns
A function which asserts the passed object is an instance of **class_or_tuple**.

Return type
Callable[[object], bool]

See also:

`isinstance()`
Return whether an object is an instance of a class or of a subclass thereof.

nanoutils.**issubclass_factory**(*class_or_tuple*, *module='nanoutils.schema'*)

Return a function which checks if the passed class is a subclass of **class_or_tuple**.

Examples

```
>>> from nanoutils import issubclass_factory

>>> func = issubclass_factory(int)

>>> func(bool)  # issubclass(bool, int)
True

>>> func(float)  # issubclass(float, int)
False
```

Parameters

- **class_or_tuple** (type or Tuple[type, ...]) – A type object or tuple of type objects.
- **module** (str) – The __module__ of the to-be returned function.

Returns
A function which asserts the passed type is a subclass of **class_or_tuple**.

Return type
Callable[[type], bool]

See also:

`issubclass()`
Return whether **cls** is a derived from another class or is the same class.

nanoutils.**import_factory**(*validate*, *module='nanoutils.schema'*)

Return a function which calls *nanoutils.get_importable()* with the **validate** argument.

Examples

```
>>> from inspect import isclass
>>> from nanoutils import import_factory

>>> func = import_factory(isclass)
>>> func('builtins.dict')
<class 'dict'>

>>> func('builtins.len')
Traceback (most recent call last):
  ...
RuntimeError: Passing <built-in function len> to isclass() failed to return True
```

Parameters

- **validate** (Callable[[T], bool]) – A callable used for validating the passed object.
- **module** (str) – The __module__ of the to-be returned function.

Returns
A function for importing the passed object and validating it using **validate**.

Return type
Callable[[str], T]

See also:

nanoutils.get_importable()
Import an object and, optionally, validate it using **validate**.

1.2.4 nanoutils.typing_utils

Types related to the builtin typing module.

Contains aliases for python >= 3.8 exclusive objects related to typing.

See also:

The typing module: Support for gradual typing as defined by PEP 484.

At large scale, the structure of the module is following:

- Imports and exports, all public names should be explicitly added to __all__.
- Internal helper functions: these should never be used in code outside this module.
- _SpecialForm and its instances (special forms): Any, NoReturn, ClassVar, Union and Optional.
- Two classes whose instances can be type arguments in addition to types: ForwardRef and TypeVar.
- The core of internal generics API: _GenericAlias and _VariadicGenericAlias, the latter is currently only used by Tuple and Callable. All subscripted types like X[int], Union[int, str], *etc.*, are instances of either of these classes.
- The public counterpart of the generics API consists of two classes: Generic and Protocol.
- Public helper functions: get_type_hints(), overload(), cast(), no_type_check(), no_type_check_decorator().
- Generic aliases for collections.abc ABCs and few additional protocols.
- Special types: NewType(), NamedTuple and TypedDict.
- Wrapper submodules for re and io related types.

Index

Literal	Special typing form to define literal types (a.k.a. value types).
Final	Special typing construct to indicate final names to type checkers.
final()	A decorator to indicate final methods and final classes.
Protocol	Base class for protocol classes.
SupportsIndex	An ABC with one abstract method __index__().
TypedDict	A simple typed name space. At runtime it is equivalent to a plain dict.
runtime_checkable	Mark a protocol class as a runtime protocol, so that it an be used with isinstance() and issubclass().
PathType	An annotation for path-like objects.
ArrayLike	Objects that can be converted to arrays (see numpy.ndarray).
DTypeLike	Objects that can be converted to dtypes (see numpy.dtype).
ShapeLike	Objects that can serve as valid array shapes.

API

nanoutils.**PathType = typing.Union[str, bytes, os.PathLike]**

An annotation for path-like objects.

1.2.5 nanoutils.numpy_utils

Utility functions related to numpy.

Note that these functions require the NumPy package.

See also:

NumPy is the fundamental package needed for scientific computing with Python. It provides:

- a powerful N-dimensional array object
- sophisticated (broadcasting) functions
- tools for integrating C/C++ and Fortran code
- useful linear algebra, Fourier transform, and random number capabilities

Index

as_nd_array(array, dtype[, ndmin, copy])	Construct a numpy array from an iterable or array-like object.
array_combinations(array[, r, axis])	Construct an array with all combinations() of **ar** along a use-specified axis.
fill_diagonal_blocks(array, i, j[, val])	Fill diagonal blocks in **array** of size (i, j).
DTypeMapping([iterable])	A mapping for creating structured dtypes.
MutableDTypeMapping([iterable])	A mutable mapping for creating structured dtypes.

API

nanoutils.**as_nd_array**(*array*, *dtype*, *ndmin=1*, *copy=False*)

Construct a numpy array from an iterable or array-like object.

Examples

```
>>> from nanoutils import as_nd_array

>>> as_nd_array(1, int)
array([1])

>>> as_nd_array([1, 2, 3, 4], int)
array([1, 2, 3, 4])

>>> iterator = iter([1, 2, 3, 4])
>>> as_nd_array(iterator, int)
array([1, 2, 3, 4])
```

Parameters

- **array** (Iterable or array-like) – An array-like object or an iterable consisting of scalars.
- **dtype** (type or numpy.dtype) – The data type of the to-be returned array.
- **ndmin** (int) – The minimum dimensionality of the to-be returned array.
- **copy** (bool) – If True, always return a copy.

Returns

A numpy array constructed from **value**.

Return type

numpy.ndarray

nanoutils.**array_combinations**(*array*, *r=2*, *axis=-1*)

Construct an array with all combinations() of **ar** along a use-specified axis.

Examples

```
>>> from nanoutils import array_combinations

>>> array = [[1, 2, 3, 4],
...          [5, 6, 7, 8]]

>>> array_combinations(array, r=2)
array([[[1, 2],
        [5, 6]],

       [[1, 3],
        [5, 7]],

       [[1, 4],
        [5, 8]],
```

(continues on next page)

(continued from previous page)

```
       [[2, 3],
        [6, 7]],

       [[2, 4],
        [6, 8]],

       [[3, 4],
        [7, 8]]])
```

Parameters

- **array** (array-like, shape $(m, \dots)$) – An n dimensional array-like object.
- **r** (int) – The length of each combination.
- **axis** (int) – The axis used for constructing the combinations.

Returns

A $n + 1$ dimensional array with all **ar** combinations (of length **r**) along the user-specified **axis**. The variable k herein represents the number of combinations: $k = \dfrac{m!/r!}{(m-r)!}$.

Return type

numpy.ndarray, shape $(k, \dots, r)$

nanoutils.**fill_diagonal_blocks**(*array*, *i*, *j*, *val=nan*)

Fill diagonal blocks in **array** of size (i, j).

The blocks are filled along the last 2 axes in **array**. Performs an inplace update of **array**.

Examples

```
>>> import numpy as np
>>> from nanoutils import fill_diagonal_blocks

>>> array = np.zeros((10, 15), dtype=int)
>>> i = 2
>>> j = 3

>>> fill_diagonal_blocks(array, i, j, val=1)
>>> print(array)
[[1 1 1 0 0 0 0 0 0 0 0 0 0 0 0]
 [1 1 1 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 1 1 1 0 0 0 0 0 0 0 0 0]
 [0 0 0 1 1 1 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 1 1 1 0 0 0 0 0 0]
 [0 0 0 0 0 0 1 1 1 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 1 1 1 0 0 0]
 [0 0 0 0 0 0 0 0 0 1 1 1 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 1 1 1]
 [0 0 0 0 0 0 0 0 0 0 0 0 1 1 1]]
```

Parameters

- **array** (numpy.ndarray) – A >= 2D NumPy array whose diagonal blocks are to be filled. Gets modified in-place.
- **i** (int) – The size of the diagonal blocks along axis -2.
- **j** (int) – The size of the diagonal blocks along axis -1.
- **val** (float) – Value to be written on the diagonal. Its type must be compatible with that of the array **a**.

Return type
None

class nanoutils.**DTypeMapping**(*iterable=None*, */*, ***fields*)

A mapping for creating structured dtypes.

Examples

```
>>> from nanoutils import DTypeMapping
>>> import numpy as np

>>> DType1 = DTypeMapping({"x": float, "y": float, "z": float, "symbol": (str, 2)})
>>> print(DType1)
DTypeMapping(
    x      = float64,
    y      = float64,
    z      = float64,
    symbol = <U2,
)

>>> DType1.x
dtype('float64')

>>> DType1.symbol
dtype('<U2')

>>> @DTypeMapping.from_type
... class DType2:
...     xyz = (float, 3)
...     symbol = (str, 2)
...     charge = np.int64

>>> print(DType2)
DTypeMapping(
    xyz    = ('<f8', (3,)),
    symbol = <U2,
    charge = int64,
)
```

property dtype

Get a structured dtype constructed from dtype mapping.

classmethod from_type(*type_obj*)

Construct a new dtype mapping from all public attributes of the decorated type object.

Example

```
>>> from nanoutils import DTypeMapping

>>> @DTypeMapping.from_type
... class AtomsDType:
...     xyz = (float, 3)
...     symbol = (str, 2)
...     charge = np.int64

>>> print(AtomsDType)
DTypeMapping(
    xyz    = ('<f8', (3,)),
    symbol = <U2,
    charge = int64,
)
```

Parameters

type_obj (type) – A type object or any object that supports `vars()`.

classmethod fromkeys(*iterable*, *value=None*)

Create a new dictionary with keys from iterable and values set to value.

class nanoutils.**MutableDTypeMapping**(*iterable=None*, */*, ***fields*)

A mutable mapping for creating structured dtypes.

Examples

```
>>> from nanoutils import DTypeMapping
>>> import numpy as np

>>> DType1 = MutableDTypeMapping({"x": float, "y": float, "z": float, "symbol":␣
↪(str, 2)})
>>> print(DType1)
MutableDTypeMapping(
    x      = float64,
    y      = float64,
    z      = float64,
    symbol = <U2,
)

>>> @MutableDTypeMapping.from_type
... class DType2:
...     xyz = (float, 3)
...     symbol = (str, 2)
...     charge = np.int64

>>> print(DType2)
```

(continues on next page)

(continued from previous page)

```
MutableDTypeMapping(
    xyz    = ('<f8', (3,)),
    symbol = <U2,
    charge = int64,
)
```

---

property dtype

Get a structured dtype constructed from the mapping.

update(*iterable=None*, */*, ***fields*)

Update the mapping from the passed mapping or iterable.

1.2.6 nanoutils.file_container

An abstract container for reading and writing files.

Index

file_to_context(file, **kwargs)	Take a path- or file-like object and return an appropiate context manager.
AbstractFileContainer(*args, **kwargs)	An abstract container for reading and writing files.

API

nanoutils.**file_to_context**(*file*, ***kwargs*)

Take a path- or file-like object and return an appropiate context manager.

Passing a path-like object will supply it to `open()`, while passing a file-like object will pass it to `contextlib.nullcontext()`.

Examples

```
>>> from io import StringIO
>>> from nanoutils import file_to_context

>>> path_like = 'file_name.txt'
>>> file_like = StringIO('this is a file-like object')

>>> context1 = file_to_context(file_like)
>>> with context1 as f1:
...     ...

>>> context2 = file_to_context(path_like)
>>> with context2 as f2:
...     ... # insert operations here
```

Parameters

- **file** (str, bytes, os.PathLike or IO) – A path- or file-like object.
- ****kwargs** (Any) – Further keyword arguments for open(). Only relevant if **file** is a path-like object.

Returns
An initialized context manager. Entering the context manager will return a file-like object.

Return type
ContextManager[IO]

class nanoutils.**AbstractFileContainer**(**args*, ***kwargs*)

An abstract container for reading and writing files.

Two public methods are defined within this class:

- *AbstractFileContainer.read()*: Construct a new instance from this object's class by reading the content to a file or file object. How the content of the to-be read file is parsed has to be defined in the *AbstractFileContainer._read()* abstract method. Additional post processing, after the new instance has been created, can be performed with *AbstractFileContainer._read_postprocess()*
- *AbstractFileContainer.write()*: Write the content of this instance to an opened file or file object. How the content of the to-be exported class instance is parsed has to be defined in the *AbstractFileContainer._write()*

Examples

```
>>> from io import StringIO
>>> from nanoutils import AbstractFileContainer

>>> class SubClass(AbstractFileContainer):
...     def __init__(self, value: str):
...         self.value = value
...
...     @classmethod
...     def _read(cls, file_obj, decoder):
...         value = decoder(file_obj.read())
...         return {'value': value}
...
...     def _write(self, file_obj, encoder):
...         value = encoder(self.value)
...         file_obj.write(value)

>>> file1 = StringIO('This is a file-like object')
>>> file2 = StringIO()

>>> obj = SubClass.read(file1)
>>> obj.write(file2)

>>> print(file2.getvalue())
This is a file-like object
```

---

classmethod **read**(*file*, *bytes_decoding=None*, ***kwargs*)

Construct a new instance from this object's class by reading the content of **file**.

Parameters

- **file** (str, bytes, os.PathLike or IO) – A path- or file-like object.
- **bytes_decoding** (str, optional) – The type of encoding to use when reading from **file** when it will be/is be opened in bytes mode. This value should be left empty otherwise.
- ****kwargs** (Any) – Further keyword arguments for open(). Only relevant if **file** is a path-like object.

Returns
A new instance constructed from **file**.

Return type
nanoutils.AbstractFileContainer

abstract classmethod _read(*file_obj*, *decoder*)

A helper function for *read()*.

Parameters

- **file_obj** (IO[AnyStr]) – A file-like object opened in read mode.
- **decoder** (Callable[[AnyStr], str]) – A function for converting the items of **file_obj** into strings.

Returns
A dictionary with keyword arguments for a new instance of this objects' class.

Return type
Dict[str, Any]

See also:

AbstractFileContainer.read()
Construct a new instance from this object's class by reading the content of **file**.

_read_postprocess()

Post process new instances created by *read()*.

Return type
None

See also:

AbstractFileContainer.read()
Construct a new instance from this object's class by reading the content of **file**.

write(*file=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*, *bytes_encoding=None*, ***kwargs*)

Write the content of this instance to **file**.

Parameters

- **file** (str, bytes, os.PathLike or IO) – A path- or file-like object. Defaults to sys.stdout if not specified.
- **bytes_encoding** (str, optional) – The type of encoding to use when writing to **file** when it will be/is be opened in bytes mode. This value should be left empty otherwise.
- ****kwargs** (Any) – Further keyword arguments for open(). Only relevant if **file** is a path-like object.

Return type
None

abstract _write(*file_obj*, *encoder*)

A helper function for *write()*.

Parameters

- **file_obj** (IO[AnyStr]) – A file-like object opened in write mode.
- **encoder** (Callable[[str], AnyStr]) – A function for converting strings into either str or bytes, the exact type matching that of **file_obj**.

Return type
None

See also:

AbstractFileContainer.write()
Write the content of this instance to **file**.

1.2.7 nanoutils.testing_utils

Utility functions related to unit-testing.

Index

FileNotFoundWarning	A ResourceWarning subclass for when a file or directory is requested but doesn't exist.
delete_finally(*paths[, prefix, warn])	A decorater which deletes the specified files and/or directories after calling the deocrated function.

API

exception nanoutils.**FileNotFoundWarning**

A ResourceWarning subclass for when a file or directory is requested but doesn't exist.

@nanoutils.**delete_finally**(**paths*, *prefix=None*, *warn=True*)

A decorater which deletes the specified files and/or directories after calling the deocrated function.

Examples

```
>>> import os
>>> from nanoutils import delete_finally

>>> file1 = 'file1.txt'
>>> dir1 = 'dir1/'
>>> os.path.isfile(file1) and os.path.isdir(dir1)
True

>>> @delete_finally(file1, dir1)
... def func():
```

(continues on next page)

(continued from previous page)

```
...     pass

>>> func()
>>> os.path.isfile(file1) or os.path.isdir(dir1)
False
```

Parameters

- ***paths** (str, bytes or os.PathLike) – Path-like objects with the names of to-be deleted files and/or directories.
- **prefix** (str, bytes or os.PathLike, optional) – The directory where all user specified **paths** are located. If None, asume the files/directories are either absolute or located in the current working directory.
- **warn** (bool) – If True issue a *FileNotFoundWarning* if a to-be deleted file or directory cannot be found.

1.2.8 nanoutils.hdf5_utils

An module with various h5py-related utilities.

Index

RecursiveKeysView(f)	Create a recursive view of all dataset names.
RecursiveValuesView(f)	Create a recursive view of all <Datasets>h5py.Dataset.
RecursiveItemsView(f)	Create a recursive view of all name/Dataset pairs.

API

class nanoutils.**RecursiveKeysView**(*f*)

Create a recursive view of all dataset names.

Examples

```
>>> import h5py
>>> from nanoutils import RecursiveKeysView

>>> filename: str = ...

>>> with h5py.File(filename, 'a') as f:
...     a = f.create_group('a')
...     b = f['a'].create_group('b')
...
...     dset1 = f.create_dataset('dset1', (10,), dtype=float)
...     dset2 = f['a'].create_dataset('dset2', (10,), dtype=float)
...     dset3 = f['a']['b'].create_dataset('dset3', (10,), dtype=float)
```

(continues on next page)

(continued from previous page)

```
...
...     print(RecursiveKeysView(f))
<RecursiveKeysView ['/a/b/dset3',
                    '/a/dset2',
                    '/dset1']>
```

Parameters
f (h5py.Group) – The to-be iterated h5py group of file.

Returns
A recursive view of all dataset names within the passed group or file.

Return type
KeysView[str]

class nanoutils.**RecursiveValuesView**(*f*)

Create a recursive view of all <Datasets>h5py.Dataset.

Examples

```
>>> import h5py
>>> from nanoutils import RecursiveValuesView

>>> filename: str = ...

>>> with h5py.File(filename, 'a') as f:
...     a = f.create_group('a')
...     b = f['a'].create_group('b')
...
...     dset1 = f.create_dataset('dset1', (10,), dtype=float)
...     dset2 = f['a'].create_dataset('dset2', (10,), dtype=float)
...     dset3 = f['a']['b'].create_dataset('dset3', (10,), dtype=float)
...
...     print(RecursiveValuesView(f))
<RecursiveValuesView [<HDF5 dataset "dset3": shape (10,), type "<f8">,
                      <HDF5 dataset "dset2": shape (10,), type "<f8">,
                      <HDF5 dataset "dset1": shape (10,), type "<f8">]>
```

Parameters
f (h5py.Group) – The to-be iterated h5py group of file.

Returns
A recursive view of all datasets within the passed group or file.

Return type
ValuesView[h5py.Dataset]

class nanoutils.**RecursiveItemsView**(*f*)

Create a recursive view of all name/Dataset pairs.

Examples

```
>>> import h5py
>>> from nanoutils import RecursiveItemsView

>>> filename: str = ...

>>> with h5py.File(filename, 'a') as f:
...     a = f.create_group('a')
...     b = f['a'].create_group('b')
...
...     dset1 = f.create_dataset('dset1', (10,), dtype=float)
...     dset2 = f['a'].create_dataset('dset2', (10,), dtype=float)
...     dset3 = f['a']['b'].create_dataset('dset3', (10,), dtype=float)
...
...     print(RecursiveItemsView(f))
<RecursiveItemsView [('/a/b/dset3', <HDF5 dataset "dset3": shape (10,), type "<f8">
↪),
                     ('/a/dset2', <HDF5 dataset "dset2": shape (10,), type "<f8">),
                     ('/dset1', <HDF5 dataset "dset1": shape (10,), type "<f8">)]>
```

Parameters
f (h5py.Group) – The to-be iterated h5py group of file.

Returns
A recursive view of all name/dataset pairs within the passed group or file.

Return type
ItemsView[str, h5py.Dataset]

1.2.9 nanoutils.yaml_utils

A `yaml.Loader` subclass that dissallows for duplicate keys.

Index

UniqueLoader(stream)	A `yaml.Loader` subclass that dissallows for duplicate keys.

API

class nanoutils.**UniqueLoader**(*stream*)
A `yaml.Loader` subclass that dissallows for duplicate keys.

Examples

```
>>> import yaml
>>> from nanoutils import UniqueLoader

>>> STR = """
```

(continues on next page)

(continued from previous page)

```
... a: 0
... a: 1
... """

>>> yaml.load(STR, Loader=yaml.SafeLoader)
{'a': 1}

>>> yaml.load(STR, Loader=UniqueLoader)
Traceback (most recent call last):
  ...
yaml.constructor.ConstructorError: while constructing a mapping
  in "<unicode string>", line 2, column 1
found a duplicate key
  in "<unicode string>", line 3, column 1
```

1.2.10 nanoutils.SetAttr

class nanoutils.**SetAttr**(*obj*, *name*, *value*)

A context manager for temporarily changing an attribute's value.

The *SetAttr* context manager is thread-safe, reusable and reentrant.

> **Warning:** Note that while *SetAttr.__enter__()* and *SetAttr.__exit__()* are thread-safe, the same does *not* hold for *SetAttr.__init__()*.

Examples

```
>>> from nanoutils import SetAttr

>>> class Test:
...     a = False

>>> print(Test.a)
False

>>> set_attr = SetAttr(Test, 'a', True)
>>> with set_attr:
...     print(Test.a)
True
```

__init__(*obj*, *name*, *value*)

Initialize the *SetAttr* context manager.

Parameters

- **obj** (object) – The to-be modified object. See *SetAttr.obj*.
- **name** (str) – The name of the to-be modified attribute. See *SetAttr.name*.

- **value** (object) – The value to-be assigned to the **name** attribute of **obj**. See *SetAttr.value*.

Return type
None

__repr__()
Implement str(self) and repr(self).

__eq__(*value*)
Implement self == value.

__reduce__()
A helper for pickle.

> **Warning:** Unsupported operation, raises a TypeError.

__copy__()
Implement copy.copy(self).

__deepcopy__(*memo=None*)
Implement copy.deepcopy(self, memo=memo).

__hash__()
Implement hash(self).

> **Warning:** A TypeError will be raised if *SetAttr.value* is not hashable.

__enter__()
Enter the context manager, modify *SetAttr.obj*.

__exit__(*exc_type*, *exc_value*, *traceback*)
Exit the context manager, restore *SetAttr.obj*.

property obj
Get the to-be modified object.

Type
object

property name
Get the name of the to-be modified attribute.

Type
str

property value
Get the value to-be assigned to the *name* attribute of *SetAttr.obj*.

Type
object

property attr
Get or set the *name* attribute of *SetAttr.obj*.

Type
object

PYTHON MODULE INDEX

n

INDEX

Symbols

A

B

C

D

E

F

G

I

K

L

M

N

O

P

R

S

U

V

W