
kwarray Documentation

Release 0.6.16

Kitware Inc. Jon Crall

Nov 18, 2023

CONTENTS

1	Function Usefulness	3
1.1	kwarray package	4
1.1.1	Submodules	4
1.1.2	Module contents	127
2	Indices and tables	193
	Bibliography	195
	Python Module Index	197
	Index	199

The `kwarray` module implements a small set of pure-python extensions to `numpy` and `torch` along with a few select algorithms. Each module contains module level docstring that gives a rough idea of the utilities in each module, and each function or class itself contains a docstring with more details and examples.

KWarray is part of Kitware's computer vision Python suite:

<https://gitlab.kitware.com/computer-vision>

FUNCTION USEFULNESS

Function name	Usefulness
<code>kwarray.ensure_rng</code>	475
<code>kwarray.ArrayAPI</code>	202
<code>kwarray.atleast_nd</code>	98
<code>kwarray.stats_dict</code>	77
<code>kwarray.group_indices</code>	72
<code>kwarray.DataFrameArray</code>	60
<code>kwarray.normalize</code>	59
<code>kwarray.isect_flags</code>	53
<code>kwarray.shuffle</code>	48
<code>kwarray.SlidingWindow</code>	33
<code>kwarray.robust_normalize</code>	31
<code>kwarray.Stitcher</code>	30
<code>kwarray.one_hot_embedding</code>	29
<code>kwarray.boolmask</code>	27
<code>kwarray.embed_slice</code>	25
<code>kwarray.RunningStats</code>	20
<code>kwarray.setcover</code>	19
<code>kwarray.argmaxima</code>	15
<code>kwarray.padded_slice</code>	14
<code>kwarray.seed_global</code>	14
<code>kwarray.standard_normal</code>	10
<code>kwarray.find_robust_normalizers</code>	10
<code>kwarray.DataFrameLight</code>	10
<code>kwarray.group_items</code>	10
<code>kwarray.maxvalue_assignment</code>	9
<code>kwarray.apply_grouping</code>	9
<code>kwarray.group_consecutive</code>	7
<code>kwarray.mincost_assignment</code>	6
<code>kwarray.uniform</code>	6
<code>kwarray.iter_reduce_ufunc</code>	5
<code>kwarray.FlatIndexer</code>	5
<code>kwarray.arglexmax</code>	3
<code>kwarray.dtype_info</code>	3
<code>kwarray.group_consecutive_indices</code>	1
<code>kwarray.equal_with_nan</code>	1
<code>kwarray.unique_rows</code>	0
<code>kwarray.uniform32</code>	0
<code>kwarray.standard_normal64</code>	0

continues on next page

Table 1 – continued from previous page

Function name	Usefulness
<code>karray.standard_normal32</code>	0
<code>karray.random_product</code>	0
<code>karray.random_combinations</code>	0
<code>karray.one_hot_lookup</code>	0
<code>karray.mindist_assignment</code>	0
<code>karray.generalized_logistic</code>	0
<code>karray.argminima</code>	0
<code>karray.apply_embedded_slice</code>	0
<code>karray.NoSupportError</code>	0
<code>karray.LocLight</code>	0

1.1 karray package

1.1.1 Submodules

`karray.algo_assignment` module

A convenient interface to solving assignment problems with the Hungarian algorithm (also known as Munkres or maximum linear-sum-assignment).

The core implementation of `munkres` in `scipy`. Recent versions are written in C, so their speed should be reflected here.

Todo:

- [] Implement linear-time maximum weight matching approximation algorithm from this paper: <https://web.eecs.umich.edu/~pettie/papers/ApproxMWM-JACM.pdf>
-

`karray.algo_assignment.mindist_assignment`(*vecs1*, *vecs2*, *p*=2)

Finds minimum cost assignment between two sets of D dimensional vectors.

Parameters

- **vecs1** (*np.ndarray*) – NxD array of vectors representing items in *vecs1*
- **vecs2** (*np.ndarray*) – MxD array of vectors representing items in *vecs2*
- **p** (*float*) – L-p norm to use. Default is 2 (aka Euclidean)

Returns

tuple containing assignments of rows in *vecs1* to rows in *vecs2*, and the total distance between assigned pairs.

Return type

Tuple[list, float]

Note: Thin wrapper around `mincost_assignment`

CommandLine

```
xdoctest -m ~/code/kwarray/kwarray/algo_assignment.py mindist_assignment
```

CommandLine

```
xdoctest -m ~/code/kwarray/kwarray/algo_assignment.py mindist_assignment
```

Example

```
>>> # xdoctest: +REQUIRES(module:scipy)
>>> # Rows are detections in img1, cols are detections in img2
>>> rng = np.random.RandomState(43)
>>> vecs1 = rng.randint(0, 10, (5, 2))
>>> vecs2 = rng.randint(0, 10, (7, 2))
>>> ret = mindist_assignment(vecs1, vecs2)
>>> print('Total error: {:.4f}'.format(ret[1]))
Total error: 8.2361
>>> print('Assignment: {}'.format(ret[0])) # xdoc: +IGNORE_WANT
Assignment: [(0, 0), (1, 3), (2, 5), (3, 2), (4, 6)]
```

`kwarray.algo_assignment.mincost_assignment(cost)`

Finds the minimum cost assignment based on a NxM cost matrix, subject to the constraint that each row can match at most one column and each column can match at most one row. Any pair with a cost of infinity will not be assigned.

Parameters

cost (*ndarray*) – NxM matrix, `cost[i, j]` is the cost to match `i` and `j`

Returns

tuple containing a list of assignment of rows
and columns, and the total cost of the assignment.

Return type

`Tuple[list, float]`

CommandLine

```
xdoctest -m ~/code/kwarray/kwarray/algo_assignment.py mincost_assignment
```

Example

```
>>> # xdoctest: +REQUIRES(module:scipy)
>>> # Costs to match item i in set1 with item j in set2.
>>> cost = np.array([
>>>     [9, 2, 1, 9],
>>>     [4, 1, 5, 5],
>>>     [9, 9, 2, 4],
>>> ])
>>> ret = mincost_assignment(cost)
>>> print('Assignment: {}'.format(ret[0]))
>>> print('Total cost: {}'.format(ret[1]))
Assignment: [(0, 2), (1, 1), (2, 3)]
Total cost: 6
```

Example

```
>>> # xdoctest: +REQUIRES(module:scipy)
>>> cost = np.array([
>>>     [0, 0, 0, 0],
>>>     [4, 1, 5, -np.inf],
>>>     [9, 9, np.inf, 4],
>>>     [9, -2, np.inf, 4],
>>> ])
>>> ret = mincost_assignment(cost)
>>> print('Assignment: {}'.format(ret[0]))
>>> print('Total cost: {}'.format(ret[1]))
Assignment: [(0, 2), (1, 3), (2, 0), (3, 1)]
Total cost: -inf
```

Example

```
>>> # xdoctest: +REQUIRES(module:scipy)
>>> cost = np.array([
>>>     [0, 0, 0, 0],
>>>     [4, 1, 5, -3],
>>>     [1, 9, np.inf, 0.1],
>>>     [np.inf, np.inf, np.inf, 100],
>>> ])
>>> ret = mincost_assignment(cost)
>>> print('Assignment: {}'.format(ret[0]))
>>> print('Total cost: {}'.format(ret[1]))
Assignment: [(0, 2), (1, 1), (2, 0), (3, 3)]
Total cost: 102.0
```

`kwarray.algo_assignment.maxvalue_assignment(value)`

Finds the maximum value assignment based on a NxM value matrix. Any pair with a non-positive value will not be assigned.

Parameters

value (*ndarray*) – NxM matrix, `value[i, j]` is the value of matching `i` and `j`

Returns

tuple containing a list of assignment of rows
and columns, and the total value of the assignment.

Return type

Tuple[list, float]

CommandLine

```
xdoctest -m ~/code/kwarray/kwarray/algo_assignment.py maxvalue_assignment
```

Example

```
>>> # xdoctest: +REQUIRES(module:scipy)
>>> # Costs to match item i in set1 with item j in set2.
>>> value = np.array([
>>>     [9, 2, 1, 3],
>>>     [4, 1, 5, 5],
>>>     [9, 9, 2, 4],
>>>     [-1, -1, -1, -1],
>>> ])
>>> ret = maxvalue_assignment(value)
>>> # Note, depending on the scipy version the assignment might change
>>> # but the value should always be the same.
>>> print('Total value: {}'.format(ret[1]))
Total value: 23.0
>>> print('Assignment: {}'.format(ret[0])) # xdoc: +IGNORE_WANT
Assignment: [(0, 0), (1, 3), (2, 1)]
```

```
>>> ret = maxvalue_assignment(np.array([[np.inf]]))
>>> print('Assignment: {}'.format(ret[0]))
>>> print('Total value: {}'.format(ret[1]))
Assignment: [(0, 0)]
Total value: inf
```

```
>>> ret = maxvalue_assignment(np.array([[0]]))
>>> print('Assignment: {}'.format(ret[0]))
>>> print('Total value: {}'.format(ret[1]))
Assignment: []
Total value: 0
```

kwarray.algo_setcover module

Algorithms to find a solution to the setcover problem

SeeAlso:

<https://github.com/keon/algorithms>

`kwarray.algo_setcover.setcover(candidate_sets_dict, items=None, set_weights=None, item_values=None, max_weight=None, algo='approx')`

Finds a feasible solution to the minimum weight maximum value set cover. The quality and runtime of the solution will depend on the backend algorithm selected.

Parameters

- **candidate_sets_dict** (*Dict[KT, List[VT]]*) – a dictionary where keys are the candidate set ids and each value is a candidate cover set.
- **items** (*Optional[VT]*) – the set of all items to be covered, if not specified, it is inferred from the candidate cover sets
- **set_weights** (*Optional[Dict[KT, float]]*) – maps candidate set ids to a cost for using this candidate cover in the solution. If not specified the weight of each candidate cover defaults to 1.
- **item_values** (*Optional[Dict[VT, float]]*) – maps each item to a value we get for returning this item in the solution. If not specified the value of each item defaults to 1.
- **max_weight** (*Optional[float]*) – if specified, the total cost of the returned cover is constrained to be less than this number.
- **algo** (*str*) – specifies which algorithm to use. Can either be ‘approx’ for the greedy solution or ‘exact’ for the globally optimal solution. Note the ‘exact’ algorithm solves an integer-linear-program, which can be very slow and requires the *pulp* package to be installed.

Returns

a subdict of candidate_sets_dict containing the chosen solution.

Return type

Dict

Example

```
>>> candidate_sets_dict = {
>>>     'a': [1, 2, 3, 8, 9, 0],
>>>     'b': [1, 2, 3, 4, 5],
>>>     'c': [4, 5, 7],
>>>     'd': [5, 6, 7],
>>>     'e': [6, 7, 8, 9, 0],
>>> }
>>> greedy_soln = setcover(candidate_sets_dict, algo='greedy')
>>> print('greedy_soln = {}'.format(ub.urepr(greedy_soln, nl=0)))
greedy_soln = {'a': [1, 2, 3, 8, 9, 0], 'c': [4, 5, 7], 'd': [5, 6, 7]}
>>> # xdoc: +REQUIRES(module:pulp)
>>> exact_soln = setcover(candidate_sets_dict, algo='exact')
>>> print('exact_soln = {}'.format(ub.urepr(exact_soln, nl=0)))
exact_soln = {'b': [1, 2, 3, 4, 5], 'e': [6, 7, 8, 9, 0]}
```

```
kwarray.algo_setcover._setcover_greedy_old(candidate_sets_dict, items=None, set_weights=None,
                                           item_values=None, max_weight=None)
```

Benchmark

```
items = np.arange(10000) candidate_sets_dict = {} for i in range(1000):
    candidate_sets_dict[i] = np.random.choice(items, 200).tolist()

_setcover_greedy_new(candidate_sets_dict) == _setcover_greedy_old(candidate_sets_dict)
_ = nh.util.profile_onthefly(_setcover_greedy_new)(candidate_sets_dict) _ =
nh.util.profile_onthefly(_setcover_greedy_old)(candidate_sets_dict)

import ubelt as ub for timer in ub.Timerit(3, bestof=1, label='time'):
    with timer:
        len(_setcover_greedy_new(candidate_sets_dict))

import ubelt as ub for timer in ub.Timerit(3, bestof=1, label='time'):
    with timer:
        len(_setcover_greedy_old(candidate_sets_dict))

kwarray.algo_setcover._setcover_greedy_new(candidate_sets_dict, items=None, set_weights=None,
                                           item_values=None, max_weight=None)
```

Implements Johnson's / Chvatal's greedy set-cover approximation algorithm.

The approximation gaurentees depend on specifications of set weights and item values

Running time:

N = number of universe items C = number of candidate covering sets

Worst case running time is: $O(C^2 * CN)$

(note this is via simple analysis, the big-oh might be better)

Set Cover: $\log(\text{len}(\text{items}) + 1)$ approximation algorithm Weighted Maximum Cover: $1 - 1/e \approx .632$ approximation algorithm Generalized maximum coverage is not implemented [[WikiMaxCov](#)].

References

Example

```
>>> candidate_sets_dict = {
>>>     'a': [1, 2, 3, 8, 9, 0],
>>>     'b': [1, 2, 3, 4, 5],
>>>     'c': [4, 5, 7],
>>>     'd': [5, 6, 7],
>>>     'e': [6, 7, 8, 9, 0],
>>> }
>>> greedy_soln = _setcover_greedy_new(candidate_sets_dict)
>>> #print(repr(greedy_soln))
...
>>> print('greedy_soln = {}'.format(ub.urepr(greedy_soln, nl=0)))
greedy_soln = {'a': [1, 2, 3, 8, 9, 0], 'c': [4, 5, 7], 'd': [5, 6, 7]}
```

Example

```
>>> candidate_sets_dict = {
>>>     'a': [1, 2, 3, 8, 9, 0],
>>>     'b': [1, 2, 3, 4, 5],
>>>     'c': [4, 5, 7],
>>>     'd': [5, 6, 7],
>>>     'e': [6, 7, 8, 9, 0],
>>> }
>>> items = list(set(it.chain(*candidate_sets_dict.values()))
>>> set_weights = {i: 1 for i in candidate_sets_dict.keys()}
>>> item_values = {e: 1 for e in items}
>>> greedy_soln = _setcover_greedy_new(candidate_sets_dict,
>>>                                     item_values=item_values,
>>>                                     set_weights=set_weights)
>>> print('greedy_soln = {}'.format(ub.urepr(greedy_soln, nl=0)))
greedy_soln = {'a': [1, 2, 3, 8, 9, 0], 'c': [4, 5, 7], 'd': [5, 6, 7]}
```

Example

```
>>> candidate_sets_dict = {}
>>> greedy_soln = _setcover_greedy_new(candidate_sets_dict)
>>> print('greedy_soln = {}'.format(ub.urepr(greedy_soln, nl=0)))
greedy_soln = {}
```

`kwarray.algo_setcover._setcover_ilp(candidate_sets_dict, items=None, set_weights=None, item_values=None, max_weight=None, verbose=False)`

Set cover / Weighted Maximum Cover exact algorithm using an integer linear program.

Todo:

- [] Use CPLEX solver if available
-

Example

```
>>> # xdoc: +REQUIRES(module:pulp)
>>> candidate_sets_dict = {}
>>> exact_soln = _setcover_ilp(candidate_sets_dict)
>>> print('exact_soln = {}'.format(ub.urepr(exact_soln, nl=0)))
exact_soln = {}
```

Example

```
>>> # xdoc: +REQUIRES(module:pulp)
>>> candidate_sets_dict = {
>>>     'a': [1, 2, 3, 8, 9, 0],
>>>     'b': [1, 2, 3, 4, 5],
>>>     'c': [4, 5, 7],
>>>     'd': [5, 6, 7],
>>>     'e': [6, 7, 8, 9, 0],
>>> }
>>> items = list(set(it.chain(*candidate_sets_dict.values()))))
>>> set_weights = {i: 1 for i in candidate_sets_dict.keys()}
>>> item_values = {e: 1 for e in items}
>>> exact_soln1 = _setcover_ilp(candidate_sets_dict,
>>>                             item_values=item_values,
>>>                             set_weights=set_weights)
>>> exact_soln2 = _setcover_ilp(candidate_sets_dict)
>>> assert exact_soln1 == exact_soln2
```

kwarray.arrayapi module

The ArrayAPI is a common API that works exactly the same on both torch.Tensors and numpy.ndarrays.

The ArrayAPI is a combination of efficiency and convenience. It is convenient because you can just use an operation directly, it will type check the data, and apply the appropriate method. But it is also efficient because it can be used with minimal type checking by accessing a type-specific backend.

For example, you can do:

```
impl = kwarray.ArrayAPI.coerce(data)
```

And then impl will give you direct access to the appropriate methods without any type checking overhead. e.g. impl.<op-you-want>(data)

But you can also do kwarray.ArrayAPI.<op-you-want>(data) on anything and it will do type checking and then do the operation you want.

Idea:

Perhaps we could separate this into its own python package (maybe called “onearray”), where the module itself behaves like the ArrayAPI. Design goals are to provide easy to use (as drop-in as possible) replacements for torch or numpy function calls. It has to have near-zero overhead, or at least a way to make that happen.

Example

```
>>> # xdoctest: +REQUIRES(module:torch)
>>> import kvarray
>>> import torch
>>> import numpy as np
>>> data1 = torch.rand(10, 10)
>>> data2 = data1.numpy()
>>> # Method 1: grab the appropriate sub-impl
>>> impl1 = kvarray.ArrayAPI.impl(data1)
>>> impl2 = kvarray.ArrayAPI.impl(data2)
>>> result1 = impl1.sum(data1, axis=0)
>>> result2 = impl2.sum(data2, axis=0)
>>> res1_np = kvarray.ArrayAPI.numpy(result1)
>>> res2_np = kvarray.ArrayAPI.numpy(result2)
>>> print('res1_np = {!r}'.format(res1_np))
>>> print('res2_np = {!r}'.format(res2_np))
>>> assert np.allclose(res1_np, res2_np)
>>> # Method 2: choose the impl on the fly
>>> result1 = kvarray.ArrayAPI.sum(data1, axis=0)
>>> result2 = kvarray.ArrayAPI.sum(data2, axis=0)
>>> res1_np = kvarray.ArrayAPI.numpy(result1)
>>> res2_np = kvarray.ArrayAPI.numpy(result2)
>>> print('res1_np = {!r}'.format(res1_np))
>>> print('res2_np = {!r}'.format(res2_np))
>>> assert np.allclose(res1_np, res2_np)
```

Example

```
>>> # xdoctest: +REQUIRES(module:torch)
>>> import torch
>>> import numpy as np
>>> data1 = torch.rand(10, 10)
>>> data2 = data1.numpy()
```

class kvarray.arrayapi._ImplRegistry

Bases: `object`

_register(*func*, *func_type*, *impl*)

_implmethod(*func=None*, *func_type='data_func'*, *impl=None*)

_apimethod(*key=None*, *func_type='data_func'*)

Creates wrapper for a “data method” — i.e. a ArrayAPI function that has only one main argument, which is an array.

_ensure_datamethods_names_are_registered()

Checks to make sure all methods are implemented in both torch and numpy implementations as well as exposed in the ArrayAPI.

kvarray.arrayapi.**_torchmethod**(*func=None*, *func_type='data_func'*, *, *impl='torch'*)

kvarray.arrayapi.**_numpymethod**(*func=None*, *func_type='data_func'*, *, *impl='numpy'*)

`kwarray.arrayapi._apimethod(key=None, func_type='data_func')`

Creates wrapper for a “data method” — i.e. a ArrayAPI function that has only one main argument, which is an array.

class `kwarray.arrayapi.TorchImpls`

Bases: `object`

Torch backend for the ArrayAPI API

is_tensor = `True`

is_numpy = `False`

static `result_type(*arrays_and_dtypes)`

Return type from promotion rules

dtype, promote_types, min_scalar_type, can_cast

static `cat(datas, axis=-1)`

static `hstack(datas)`

Concatenates along axis=0 if inputs are 1-D otherwise axis=1

Example

```
>>> # xdoctest: +REQUIRES(module:torch)
>>> import torch
>>> datas1 = [torch.arange(10), torch.arange(10)]
>>> datas2 = [d.numpy() for d in datas1]
>>> ans1 = TorchImpls.hstack(datas1)
>>> ans2 = NumpyImpls.hstack(datas2)
>>> assert np.all(ans1.numpy() == ans2)
```

static `vstack(datas)`

Ensures that inputs datas are at least 2D (prepending a dimension of 1) and then concats along axis=0.

Example

```
>>> # xdoctest: +REQUIRES(module:torch)
>>> import torch
>>> datas1 = [torch.arange(10), torch.arange(10)]
>>> datas2 = [d.numpy() for d in datas1]
>>> ans1 = TorchImpls.vstack(datas1)
>>> ans2 = NumpyImpls.vstack(datas2)
>>> assert np.all(ans1.numpy() == ans2)
```

static `atleast_nd(arr, n, front=False)`

static `view(data, *shape)`

static `take(data, indices, axis=None)`

static `compress(data, flags, axis=None)`

Example

```
>>> # xdoctest: +REQUIRES(module:torch)
>>> import kvarray
>>> import torch
>>> data = torch.rand(10, 4, 2)
>>> impl = kvarray.ArrayAPI.coerce(data)
```

```
>>> axis = 0
>>> flags = (torch.arange(data.shape[axis]) % 2) == 0
>>> out = impl.compress(data, flags, axis=axis)
>>> assert tuple(out.shape) == (5, 4, 2)
```

```
>>> axis = 1
>>> flags = (torch.arange(data.shape[axis]) % 2) == 0
>>> out = impl.compress(data, flags, axis=axis)
>>> assert tuple(out.shape) == (10, 2, 2)
```

```
>>> axis = 2
>>> flags = (torch.arange(data.shape[axis]) % 2) == 0
>>> out = impl.compress(data, flags, axis=axis)
>>> assert tuple(out.shape) == (10, 4, 1)
```

```
>>> axis = None
>>> data = torch.rand(10)
>>> flags = (torch.arange(data.shape[0]) % 2) == 0
>>> out = impl.compress(data, flags, axis=axis)
>>> assert tuple(out.shape) == (5,)
```

`static tile(data, reps)`

Implement np.tile in torch

Example

```
>>> # xdoctest: +SKIP
>>> # xdoctest: +REQUIRES(module:torch)
>>> data = torch.arange(10)[: , None]
>>> ans1 = ArrayAPI.tile(data, [1, 2])
>>> ans2 = ArrayAPI.tile(data.numpy(), [1, 2])
>>> assert np.all(ans1.numpy() == ans2)
```

Doctest

```

>>> # xdoctest: +SKIP
>>> # xdoctest: +REQUIRES(module:torch)
>>> shapes = [(3,), (3, 4,), (3, 5, 7), (1,), (3, 1, 3)]
>>> for shape in shapes:
>>>     data = torch.rand(*shape)
>>>     for axis in range(len(shape)):
>>>         for reps in it.product(*[range(0, 4)] * len(shape)):
>>>             ans1 = ArrayAPI.tile(data, reps)
>>>             ans2 = ArrayAPI.tile(data.numpy(), reps)
>>>             #print('ans1.shape = {!r}'.format(tuple(ans1.shape)))
>>>             #print('ans2.shape = {!r}'.format(tuple(ans2.shape)))
>>>             assert np.all(ans1.numpy() == ans2)

```

static `repeat(data, repeats, axis=None)`

I'm not actually sure how to implement this efficiently

Example

```

>>> # xdoctest: +SKIP
>>> data = torch.arange(10)[: , None]
>>> ans1 = ArrayAPI.repeat(data, 2, axis=1)
>>> ans2 = ArrayAPI.repeat(data.numpy(), 2, axis=1)
>>> assert np.all(ans1.numpy() == ans2)

```

Doctest

```

>>> # xdoctest: +SKIP
>>> shapes = [(3,), (3, 4,), (3, 5, 7)]
>>> for shape in shapes:
>>>     data = torch.rand(*shape)
>>>     for axis in range(len(shape)):
>>>         for repeats in range(0, 4):
>>>             ans1 = ArrayAPI.repeat(data, repeats, axis=axis)
>>>             ans2 = ArrayAPI.repeat(data.numpy(), repeats, axis=axis)
>>>             assert np.all(ans1.numpy() == ans2)

```

`ArrayAPI.repeat(data, 2, axis=0)` `ArrayAPI.repeat(data.numpy(), 2, axis=0)`

`x = np.array([[1,2],[3,4]])` `np.repeat(x, [1, 2], axis=0)`

`ArrayAPI.repeat(data.numpy(), [1, 2])`

static `T(data)`

static `transpose(data, axes)`

Example

```
>>> # xdoctest: +REQUIRES(module:torch)
>>> import torch
>>> data1 = torch.rand(2, 3, 5)
>>> data2 = data1.numpy()
>>> res1 = ArrayAPI.transpose(data1, (2, 0, 1))
>>> res2 = ArrayAPI.transpose(data2, (2, 0, 1))
>>> assert np.all(res1.numpy() == res2)
```

```
static numel(data)

static full_like(data, fill_value, dtype=None)

static empty_like(data, dtype=None)

static zeros_like(data, dtype=None)

static ones_like(data, dtype=None)

static full(shape, fill_value, dtype=<class 'float'>)

static empty(shape, dtype=<class 'float'>)

static zeros(shape, dtype=<class 'float'>)

static ones(shape, dtype=<class 'float'>)

static argmax(data, axis=None)

static argmin(data, axis=None)

static argsort(data, axis=-1, descending=False)
```

Example

```
>>> # xdoctest: +REQUIRES(module:torch)
>>> from kvarray.arrayapi import * # NOQA
>>> import torch
>>> rng = np.random.RandomState(0)
>>> data2 = rng.rand(5, 5)
>>> data1 = torch.from_numpy(data2)
>>> res1 = ArrayAPI.argsort(data1)
>>> res2 = ArrayAPI.argsort(data2)
>>> assert np.all(res1.numpy() == res2)
>>> res1 = ArrayAPI.argsort(data1, axis=1)
>>> res2 = ArrayAPI.argsort(data2, axis=1)
>>> assert np.all(res1.numpy() == res2)
>>> res1 = ArrayAPI.argsort(data1, axis=1, descending=True)
>>> res2 = ArrayAPI.argsort(data2, axis=1, descending=True)
>>> assert np.all(res1.numpy() == res2)
>>> data2 = rng.rand(5)
>>> data1 = torch.from_numpy(data2)
>>> res1 = ArrayAPI.argsort(data1, axis=0, descending=True)
```

(continues on next page)

(continued from previous page)

```
>>> res2 = ArrayAPI.argsort(data2, axis=0, descending=True)
>>> assert np.all(res1.numpy() == res2)
```

static `max(data, axis=None)`

Example

```
>>> # xdoctest: +REQUIRES(module:torch)
>>> from kwarray.arrayapi import * # NOQA
>>> from kwarray.arrayapi import _TORCH_HAS_MAX_BUG
>>> import torch
>>> import pytest
>>> if _TORCH_HAS_MAX_BUG:
>>>     pytest.skip('torch max has a bug, which was fixed in 1.7')
>>> data1 = torch.rand(5, 5, 5, 5, 5, 5)
>>> data2 = data1.numpy()
>>> res1 = ArrayAPI.max(data1)
>>> res2 = ArrayAPI.max(data2)
>>> assert np.all(res1.numpy() == res2)
>>> res1 = ArrayAPI.max(data1, axis=(4, 0, 1))
>>> res2 = ArrayAPI.max(data2, axis=(4, 0, 1))
>>> assert np.all(res1.numpy() == res2)
>>> res1 = ArrayAPI.max(data1, axis=(5, -2))
>>> res2 = ArrayAPI.max(data2, axis=(5, -2))
>>> assert np.all(res1.numpy() == res2)
```

static `min(data, axis=None)`

Example

```
>>> # xdoctest: +REQUIRES(module:torch)
>>> from kwarray.arrayapi import * # NOQA
>>> from kwarray.arrayapi import _TORCH_HAS_MAX_BUG
>>> import pytest
>>> import torch
>>> if _TORCH_HAS_MAX_BUG:
>>>     pytest.skip('torch min has a bug, which was fixed in 1.7')
>>> data1 = torch.rand(5, 5, 5, 5, 5, 5)
>>> data2 = data1.numpy()
>>> res1 = ArrayAPI.min(data1)
>>> res2 = ArrayAPI.min(data2)
>>> assert np.all(res1.numpy() == res2)
>>> res1 = ArrayAPI.min(data1, axis=(4, 0, 1))
>>> res2 = ArrayAPI.min(data2, axis=(4, 0, 1))
>>> assert np.all(res1.numpy() == res2)
>>> res1 = ArrayAPI.min(data1, axis=(5, -2))
>>> res2 = ArrayAPI.min(data2, axis=(5, -2))
>>> assert np.all(res1.numpy() == res2)
```

static `max_argmax(data, axis=None)`

Parameters

- **data** (*Tensor*) – data to perform operation on
- **axis** (*None* | *int*) – axis to perform operation on

Returns

The max value(s) and argmax position(s).

Return type

Tuple[Numeric, Numeric]

Note: In modern versions of torch and numpy if there are multiple maximum values the index of the instance is returned. This is not true in older versions of torch. I'm unsure when this gaurentee was added to numpy.

Example

```
>>> # xdoctest: +REQUIRES(module:torch)
>>> from kvarray.arrayapi import * # NOQA
>>> from kvarray.arrayapi import ArrayAPI
>>> import torch
>>> data = 1 / (1 + (torch.arange(12) - 6).view(3, 4) ** 2)
>>> ArrayAPI.max_argmax(data)
(tensor(1...), tensor(6))
>>> # When the values are all the same, there doesn't seem
>>> # to be a reliable spec on which one is returned first.
>>> np.ones(10).argmax() # xdoctest: +IGNORE_WANT
0
>>> # Newer versions of torch (e.g. 1.12.0)
>>> torch.ones(10).argmax() # xdoctest: +IGNORE_WANT
tensor(0)
>>> # Older versions of torch (e.g 1.6.0)
>>> torch.ones(10).argmax() # xdoctest: +IGNORE_WANT
tensor(9)
```

static min_argmin(data, axis=None)

Parameters

- **data** (*Tensor*) – data to perform operation on
- **axis** (*None* | *int*) – axis to perform operation on

Returns

The min value(s) and argmin position(s).

Return type

Tuple[Numeric, Numeric]

Note: In modern versions of torch and numpy if there are multiple minimum values the index of the instance is returned. This is not true in older versions of torch. I'm unsure when this gaurentee was added to numpy.

Example

```

>>> # xdoctest: +REQUIRES(module:torch)
>>> from kwarray.arrayapi import * # NOQA
>>> from kwarray.arrayapi import ArrayAPI
>>> import torch
>>> data = (torch.arange(12) - 6).view(3, 4) ** 2
>>> ArrayAPI.min_argmin(data)
(tensor(0), tensor(6))
>>> # Issue demo:
>>> # When the values are all the same, there doesn't seem
>>> # to be a reliable spec on which one is returned first.
>>> np.ones(10).argmin() # xdoctest: +IGNORE_WANT
0
>>> # Newer versions of torch (e.g. 1.12.0)
>>> torch.ones(10).argmin() # xdoctest: +IGNORE_WANT
tensor(0)
>>> # Older versions of torch (e.g 1.6.0)
>>> torch.ones(10).argmin() # xdoctest: +IGNORE_WANT
tensor(9)

```

static maximum(data1, data2, out=None)

Example

```

>>> # xdoctest: +REQUIRES(module:torch)
>>> import sys, ubelt
>>> from kwarray.arrayapi import * # NOQA
>>> import torch
>>> data1 = torch.rand(5, 5)
>>> data2 = torch.rand(5, 5)
>>> result1 = TorchImpls.maximum(data1, data2)
>>> result2 = NumpyImpls.maximum(data1.numpy(), data2.numpy())
>>> assert np.allclose(result1.numpy(), result2)
>>> result1 = TorchImpls.maximum(data1, 0)
>>> result2 = NumpyImpls.maximum(data1.numpy(), 0)
>>> assert np.allclose(result1.numpy(), result2)

```

static minimum(data1, data2, out=None)

Example

```

>>> # xdoctest: +REQUIRES(module:torch)
>>> import torch
>>> data1 = torch.rand(5, 5)
>>> data2 = torch.rand(5, 5)
>>> result1 = TorchImpls.minimum(data1, data2)
>>> result2 = NumpyImpls.minimum(data1.numpy(), data2.numpy())
>>> assert np.allclose(result1.numpy(), result2)

```

Example

```
>>> # xdoctest: +REQUIRES(module:torch)
>>> import sys, ubelt
>>> from kvarray.arrayapi import * # NOQA
>>> import torch
>>> data1 = torch.rand(5, 5)
>>> data2 = torch.rand(5, 5)
>>> result1 = TorchImpls.minimum(data1, data2)
>>> result2 = NumpyImpls.minimum(data1.numpy(), data2.numpy())
>>> assert np.allclose(result1.numpy(), result2)
>>> result1 = TorchImpls.minimum(data1, 0)
>>> result2 = NumpyImpls.minimum(data1.numpy(), 0)
>>> assert np.allclose(result1.numpy(), result2)
```

static `array_equal(data1, data2, equal_nan=False) → bool`

Returns True if all elements in the array are equal. This mirrors the behavior of `numpy.array_equal()`.

Parameters

- **data1** (*Tensor*) – first array
- **data2** (*Tensor*) – second array
- **equal_nan** (*bool*) – Whether to compare NaN's as equal.

Returns

bool

Example

```
>>> # xdoctest: +REQUIRES(module:torch)
>>> from kvarray.arrayapi import * # NOQA
>>> import torch
>>> data1 = torch.rand(5, 5)
>>> data2 = data1 + 1
>>> result1 = TorchImpls.array_equal(data1, data2)
>>> result2 = NumpyImpls.array_equal(data1.numpy(), data2.numpy())
>>> result3 = TorchImpls.array_equal(data1, data1)
>>> result4 = NumpyImpls.array_equal(data1.numpy(), data1.numpy())
>>> assert result1 is False
>>> assert result2 is False
>>> assert result3 is True
>>> assert result4 is True
```


Example

```

>>> # xdoctest: +REQUIRES(module:torch)
>>> from kwarray.arrayapi import * # NOQA
>>> import torch
>>> data1 = torch.rand(5, 5)
>>> data1[0] = np.nan
>>> data2 = data1
>>> result1 = TorchImpls.array_equal(data1, data2)
>>> result2 = NumpyImpls.array_equal(data1.numpy(), data2.numpy())
>>> result3 = TorchImpls.array_equal(data1, data2, equal_nan=True)
>>> result4 = NumpyImpls.array_equal(data1.numpy(), data2.numpy(), equal_
↳ nan=True)
>>> assert result1 is False
>>> assert result2 is False
>>> assert result3 is True
>>> assert result4 is True

```

static `matmul(data1, data2, out=None)`

static `sum(data, axis=None)`

static `nan_to_num(x, copy=True)`

static `copy(data)`

static `log(data)`

static `log2(data)`

static `any(data)`

static `all(data)`

static `nonzero(data)`

static `astype(data, dtype, copy=True)`

static `tensor(data, device=None)`

static `numpy(data)`

static `tolist(data)`

static `contiguous(data)`

static `pad(data, pad_width, mode='constant')`

static `asarray(data, dtype=None)`

Cast data into a tensor representation

Example

```
>>> data = np.empty((2, 0, 196, 196), dtype=np.float32)
```

static `ensure(data, dtype=None)`

Cast data into a tensor representation

Example

```
>>> data = np.empty((2, 0, 196, 196), dtype=np.float32)
```

static `dtype_kind(data)`

returns the numpy code for the data type kind

static `floor(data, out=None)`

static `ceil(data, out=None)`

static `ifloor(data, out=None)`

static `iceil(data, out=None)`

static `round(data, decimals=0, out=None)`

Example

```
>>> # xdoctest: +REQUIRES(module:torch)
>>> import kwarray
>>> import torch
>>> rng = kwarray.ensure_rng(0)
>>> np_data = rng.rand(10) * 100
>>> pt_data = torch.from_numpy(np_data)
>>> a = kwarray.ArrayAPI.round(np_data)
>>> b = kwarray.ArrayAPI.round(pt_data)
>>> assert np.all(a == b.numpy())
>>> a = kwarray.ArrayAPI.round(np_data, 2)
>>> b = kwarray.ArrayAPI.round(pt_data, 2)
>>> assert np.all(a == b.numpy())
```

static `iround(data, out=None, dtype=<class 'int'>)`

static `clip(data, a_min=None, a_max=None, out=None)`

static `softmax(data, axis=None)`

class `kwarray.arrayapi.NumpyImpls`

Bases: `object`

Numpy backend for the ArrayAPI API

is_tensor = `False`

is_numpy = `True`

static hstack(*tup*, *, *dtype=None*, *casting='same_kind'*)

Stack arrays in sequence horizontally (column wise).

This is equivalent to concatenation along the second axis, except for 1-D arrays where it concatenates along the first axis. Rebuilds arrays divided by *hsplit*.

This function makes most sense for arrays with up to 3 dimensions. For instance, for pixel-data with a height (first axis), width (second axis), and r/g/b channels (third axis). The functions *concatenate*, *stack* and *block* provide more general stacking and concatenation operations.

Parameters

- **tup** (*sequence of ndarrays*) – The arrays must have the same shape along all but the second axis, except 1-D arrays which can be any length.
- **dtype** (*str or dtype*) – If provided, the destination array will have this dtype. Cannot be provided together with *out*.
- **.. versionadded:: 1.24**
- **casting** ({*'no'*, *'equiv'*, *'safe'*, *'same_kind'*, *'unsafe'*}, *optional*) – Controls what kind of data casting may occur. Defaults to *'same_kind'*.
- **.. versionadded:: 1.24**

Returns

stacked – The array formed by stacking the given arrays.

Return type

ndarray

See also:

concatenate

Join a sequence of arrays along an existing axis.

stack

Join a sequence of arrays along a new axis.

block

Assemble an nd-array from nested lists of blocks.

vstack

Stack arrays in sequence vertically (row wise).

dstack

Stack arrays in sequence depth wise (along third axis).

column_stack

Stack 1-D arrays as columns into a 2-D array.

hsplit

Split an array into multiple sub-arrays horizontally (column-wise).

Examples

```
>>> a = np.array((1,2,3))
>>> b = np.array((4,5,6))
>>> np.hstack((a,b))
array([1, 2, 3, 4, 5, 6])
>>> a = np.array([[1],[2],[3]])
>>> b = np.array([[4],[5],[6]])
>>> np.hstack((a,b))
array([[1, 4],
       [2, 5],
       [3, 6]])
```

static vstack(*tup*, *, *dtype=None*, *casting='same_kind'*)

Stack arrays in sequence vertically (row wise).

This is equivalent to concatenation along the first axis after 1-D arrays of shape $(N,)$ have been reshaped to $(1,N)$. Rebuilds arrays divided by *vsplit*.

This function makes most sense for arrays with up to 3 dimensions. For instance, for pixel-data with a height (first axis), width (second axis), and r/g/b channels (third axis). The functions *concatenate*, *stack* and *block* provide more general stacking and concatenation operations.

`np.row_stack` is an alias for *vstack*. They are the same function.

Parameters

- **tup** (*sequence of ndarrays*) – The arrays must have the same shape along all but the first axis. 1-D arrays must have the same length.
- **dtype** (*str or dtype*) – If provided, the destination array will have this dtype. Cannot be provided together with *out*.
- **.. versionadded:: 1.24**
- **casting** (*{'no', 'equiv', 'safe', 'same_kind', 'unsafe'}, optional*) – Controls what kind of data casting may occur. Defaults to 'same_kind'.
- **.. versionadded:: 1.24**

Returns

stacked – The array formed by stacking the given arrays, will be at least 2-D.

Return type

ndarray

See also:

concatenate

Join a sequence of arrays along an existing axis.

stack

Join a sequence of arrays along a new axis.

block

Assemble an nd-array from nested lists of blocks.

hstack

Stack arrays in sequence horizontally (column wise).

dstack

Stack arrays in sequence depth wise (along third axis).

column_stack

Stack 1-D arrays as columns into a 2-D array.

vsplit

Split an array into multiple sub-arrays vertically (row-wise).

Examples

```
>>> a = np.array([1, 2, 3])
>>> b = np.array([4, 5, 6])
>>> np.vstack((a,b))
array([[1, 2, 3],
       [4, 5, 6]])
```

```
>>> a = np.array([[1], [2], [3]])
>>> b = np.array([[4], [5], [6]])
>>> np.vstack((a,b))
array([[1],
       [2],
       [3],
       [4],
       [5],
       [6]])
```

static result_type(*arrays_and_dtypes)

Return type from promotion rules

SeeAlso:

`numpy.find_common_type()` `numpy.promote_types()` `numpy.result_type()`

static cat(datas, axis=-1)**static atleast_nd(arr, n, front=False)****static view(data, *shape)****static take(data, indices, axis=None)****static compress(data, flags, axis=None)****static repeat(data, repeats, axis=None)****static tile(data, reps)****static T(data)****static transpose(data, axes)****static numel(data)****static empty_like(data, dtype=None)****static full_like(data, fill_value, dtype=None)**

```

static zeros_like(data, dtype=None)
static ones_like(data, dtype=None)
static full(shape, fill_value, dtype=<class 'float'>)
static empty(shape, dtype=<class 'float'>)
static zeros(shape, dtype=<class 'float'>)
static ones(shape, dtype=<class 'float'>)
static argmax(data, axis=None)
static argmin(data, axis=None)
static argsort(data, axis=-1, descending=False)
static max(data, axis=None)
static min(data, axis=None)
static max_argmax(data, axis=None)

```

Parameters

- **data** (*ArrayLike*) – data to perform operation on
- **axis** (*None* | *int*) – axis to perform operation on

Returns

The max value(s) and argmax position(s).

Return type

Tuple[Numeric, Numeric]

```
static min_argmin(data, axis=None)
```

Parameters

- **data** (*ArrayLike*) – data to perform operation on
- **axis** (*None* | *int*) – axis to perform operation on

Returns

The min value(s) and argmin position(s).

Return type

Tuple[Numeric, Numeric]

```
static sum(data, axis=None)
```

```
static maximum(data1, data2, out=None)
```

```
static minimum(data1, data2, out=None)
```

```
matmul = <ufunc 'matmul'>
```

static nan_to_num(*x*, *copy=True*, *nan=0.0*, *posinf=None*, *neginf=None*)

Replace NaN with zero and infinity with large finite numbers (default behaviour) or with the numbers defined by the user using the *nan*, *posinf* and/or *neginf* keywords.

If *x* is inexact, NaN is replaced by zero or by the user defined value in *nan* keyword, infinity is replaced by the largest finite floating point values representable by *x.dtype* or by the user defined value in *posinf* keyword and -infinity is replaced by the most negative finite floating point values representable by *x.dtype* or by the user defined value in *neginf* keyword.

For complex dtypes, the above is applied to each of the real and imaginary components of *x* separately.

If *x* is not inexact, then no replacements are made.

Parameters

- **x** (*scalar or array_like*) – Input data.
- **copy** (*bool, optional*) – Whether to create a copy of *x* (True) or to replace values in-place (False). The in-place operation only occurs if casting to an array does not require a copy. Default is True.

New in version 1.13.

- **nan** (*int, float, optional*) – Value to be used to fill NaN values. If no value is passed then NaN values will be replaced with 0.0.

New in version 1.17.

- **posinf** (*int, float, optional*) – Value to be used to fill positive infinity values. If no value is passed then positive infinity values will be replaced with a very large number.

New in version 1.17.

- **neginf** (*int, float, optional*) – Value to be used to fill negative infinity values. If no value is passed then negative infinity values will be replaced with a very small (or negative) number.

New in version 1.17.

Returns

out – *x*, with the non-finite values replaced. If *copy* is False, this may be *x* itself.

Return type

ndarray

See also:

isinf

Shows which elements are positive or negative infinity.

isneginf

Shows which elements are negative infinity.

isposinf

Shows which elements are positive infinity.

isnan

Shows which elements are Not a Number (NaN).

isfinite

Shows which elements are finite (not NaN, not infinity)

Notes

NumPy uses the IEEE Standard for Binary Floating-Point for Arithmetic (IEEE 754). This means that Not a Number is not equivalent to infinity.

Examples

```
>>> np.nan_to_num(np.inf)
1.7976931348623157e+308
>>> np.nan_to_num(-np.inf)
-1.7976931348623157e+308
>>> np.nan_to_num(np.nan)
0.0
>>> x = np.array([np.inf, -np.inf, np.nan, -128, 128])
>>> np.nan_to_num(x)
array([ 1.79769313e+308, -1.79769313e+308,  0.00000000e+000, # may vary
        -1.28000000e+002,  1.28000000e+002])
>>> np.nan_to_num(x, nan=-9999, posinf=33333333, neginf=33333333)
array([ 3.3333333e+07,  3.3333333e+07, -9.9990000e+03,
        -1.2800000e+02,  1.2800000e+02])
>>> y = np.array([complex(np.inf, np.nan), np.nan, complex(np.nan, np.inf)])
array([ 1.79769313e+308, -1.79769313e+308,  0.00000000e+000, # may vary
        -1.28000000e+002,  1.28000000e+002])
>>> np.nan_to_num(y)
array([ 1.79769313e+308 +0.00000000e+000j, # may vary
        0.00000000e+000 +0.00000000e+000j,
        0.00000000e+000 +1.79769313e+308j])
>>> np.nan_to_num(y, nan=111111, posinf=222222)
array([222222.+111111.j, 111111.      +0.j, 111111.+222222.j])
```

static array_equal(a1, a2, equal_nan=False)

True if two arrays have the same shape and elements, False otherwise.

Parameters

- **a1, a2** (*array_like*) – Input arrays.
- **equal_nan** (*bool*) – Whether to compare NaN's as equal. If the dtype of a1 and a2 is complex, values will be considered equal if either the real or the imaginary component of a given value is nan.

New in version 1.19.0.

Returns

b – Returns True if the arrays are equal.

Return type

`bool`

See also:

allclose

Returns True if two arrays are element-wise equal within a tolerance.

array_equiv

Returns True if input arrays are shape consistent and all elements equal.

Examples

```
>>> np.array_equal([1, 2], [1, 2])
True
>>> np.array_equal(np.array([1, 2]), np.array([1, 2]))
True
>>> np.array_equal([1, 2], [1, 2, 3])
False
>>> np.array_equal([1, 2], [1, 4])
False
>>> a = np.array([1, np.nan])
>>> np.array_equal(a, a)
False
>>> np.array_equal(a, a, equal_nan=True)
True
```

When `equal_nan` is `True`, complex values with nan components are considered equal if either the real *or* the imaginary components are nan.

```
>>> a = np.array([1 + 1j])
>>> b = a.copy()
>>> a.real = np.nan
>>> b.imag = np.nan
>>> np.array_equal(a, b, equal_nan=True)
True
```

`log = <ufunc 'log'>`

`log2 = <ufunc 'log2'>`

static any(*a*, *axis=None*, *out=None*, *keepdims=<no value>*, *, *where=<no value>*)

Test whether any array element along a given axis evaluates to `True`.

Returns single boolean if *axis* is `None`

Parameters

- **a** (*array_like*) – Input array or object that can be converted to an array.
- **axis** (*None or int or tuple of ints, optional*) – Axis or axes along which a logical OR reduction is performed. The default (`axis=None`) is to perform a logical OR over all the dimensions of the input array. *axis* may be negative, in which case it counts from the last to the first axis.

New in version 1.7.0.

If this is a tuple of ints, a reduction is performed on multiple axes, instead of a single axis or all the axes as before.
- **out** (*ndarray, optional*) – Alternate output array in which to place the result. It must have the same shape as the expected output and its type is preserved (e.g., if it is of type float, then it will remain so, returning 1.0 for `True` and 0.0 for `False`, regardless of the type of *a*). See [Output type determination](#) for more details.
- **keepdims** (*bool, optional*) – If this is set to `True`, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input array.

If the default value is passed, then *keepdims* will not be passed through to the *any* method of sub-classes of *ndarray*, however any non-default value will be. If the sub-class' method does not implement *keepdims* any exceptions will be raised.

- **where** (*array_like of bool, optional*) – Elements to include in checking for any *True* values. See *~numpy.ufunc.reduce* for details.

New in version 1.20.0.

Returns

any – A new boolean or *ndarray* is returned unless *out* is specified, in which case a reference to *out* is returned.

Return type

bool or *ndarray*

See also:

ndarray.any

equivalent method

all

Test whether all elements along a given axis evaluate to *True*.

Notes

Not a Number (NaN), positive infinity and negative infinity evaluate to *True* because these are not equal to zero.

Examples

```
>>> np.any([[True, False], [True, True]])
True
```

```
>>> np.any([[True, False], [False, False]], axis=0)
array([ True, False])
```

```
>>> np.any([-1, 0, 5])
True
```

```
>>> np.any(np.nan)
True
```

```
>>> np.any([[True, False], [False, False]], where=[[False], [True]])
False
```

```
>>> o=np.array(False)
>>> z=np.any([-1, 4, 5], out=o)
>>> z, o
(array(True), array(True))
>>> # Check now that z is a reference to o
>>> z is o
True
```

(continues on next page)

(continued from previous page)

```
>>> id(z), id(o) # identity of z and o
(191614240, 191614240)
```

static all(*a*, *axis=None*, *out=None*, *keepdims=<no value>*, ***, *where=<no value>*)

Test whether all array elements along a given axis evaluate to True.

Parameters

- **a** (*array_like*) – Input array or object that can be converted to an array.
- **axis** (*None or int or tuple of ints, optional*) – Axis or axes along which a logical AND reduction is performed. The default (**axis=None**) is to perform a logical AND over all the dimensions of the input array. *axis* may be negative, in which case it counts from the last to the first axis.

New in version 1.7.0.

If this is a tuple of ints, a reduction is performed on multiple axes, instead of a single axis or all the axes as before.

- **out** (*ndarray, optional*) – Alternate output array in which to place the result. It must have the same shape as the expected output and its type is preserved (e.g., if `dtype(out)` is float, the result will consist of 0.0's and 1.0's). See [Output type determination](#) for more details.
- **keepdims** (*bool, optional*) – If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input array.

If the default value is passed, then *keepdims* will not be passed through to the *all* method of sub-classes of *ndarray*, however any non-default value will be. If the sub-class' method does not implement *keepdims* any exceptions will be raised.

- **where** (*array_like of bool, optional*) – Elements to include in checking for all *True* values. See `~numpy.ufunc.reduce` for details.

New in version 1.20.0.

Returns

all – A new boolean or array is returned unless *out* is specified, in which case a reference to *out* is returned.

Return type

ndarray, [bool](#)

See also:

ndarray.all

equivalent method

[any](#)

Test whether any element along a given axis evaluates to True.

Notes

Not a Number (NaN), positive infinity and negative infinity evaluate to *True* because these are not equal to zero.

Examples

```
>>> np.all([[True,False],[True,True]])
False
```

```
>>> np.all([[True,False],[True,True]], axis=0)
array([ True, False])
```

```
>>> np.all([-1, 4, 5])
True
```

```
>>> np.all([1.0, np.nan])
True
```

```
>>> np.all([[True, True], [False, True]], where=[[True], [False]])
True
```

```
>>> o=np.array(False)
>>> z=np.all([-1, 4, 5], out=o)
>>> id(z), id(o), z
(28293632, 28293632, array(True)) # may vary
```

static copy(*a*, *order*='K', *subok*=False)

Return an array copy of the given object.

Parameters

- **a** (*array_like*) – Input data.
- **order** ({'C', 'F', 'A', 'K'}, *optional*) – Controls the memory layout of the copy. 'C' means C-order, 'F' means F-order, 'A' means 'F' if *a* is Fortran contiguous, 'C' otherwise. 'K' means match the layout of *a* as closely as possible. (Note that this function and `ndarray.copy()` are very similar, but have different default values for their *order*= arguments.)
- **subok** (*bool*, *optional*) – If True, then sub-classes will be passed-through, otherwise the returned array will be forced to be a base-class array (defaults to False).

New in version 1.19.0.

Returns

arr – Array interpretation of *a*.

Return type

ndarray

See also:

ndarray.copy

Preferred method for creating an array copy

Notes

This is equivalent to:

```
>>> np.array(a, copy=True)
```

Examples

Create an array x, with a reference y and a copy z:

```
>>> x = np.array([1, 2, 3])
>>> y = x
>>> z = np.copy(x)
```

Note that, when we modify x, y changes, but not z:

```
>>> x[0] = 10
>>> x[0] == y[0]
True
>>> x[0] == z[0]
False
```

Note that, np.copy clears previously set WRITEABLE=False flag.

```
>>> a = np.array([1, 2, 3])
>>> a.flags["WRITEABLE"] = False
>>> b = np.copy(a)
>>> b.flags["WRITEABLE"]
True
>>> b[0] = 3
>>> b
array([3, 2, 3])
```

Note that np.copy is a shallow copy and will not copy object elements within arrays. This is mainly important for arrays containing Python objects. The new array will contain the same object which may lead to surprises if that object can be modified (is mutable):

```
>>> a = np.array([1, 'm', [2, 3, 4]], dtype=object)
>>> b = np.copy(a)
>>> b[2][0] = 10
>>> a
array([1, 'm', list([10, 3, 4])], dtype=object)
```

To ensure all elements within an object array are copied, use *copy.deepcopy*:

```
>>> import copy
>>> a = np.array([1, 'm', [2, 3, 4]], dtype=object)
>>> c = copy.deepcopy(a)
>>> c[2][0] = 10
>>> c
array([1, 'm', list([10, 3, 4])], dtype=object)
>>> a
array([1, 'm', list([2, 3, 4])], dtype=object)
```

static nonzero(*a*)

Return the indices of the elements that are non-zero.

Returns a tuple of arrays, one for each dimension of *a*, containing the indices of the non-zero elements in that dimension. The values in *a* are always tested and returned in row-major, C-style order.

To group the indices by element, rather than dimension, use *argwhere*, which returns a row for each non-zero element.

Note: When called on a zero-d array or scalar, `nonzero(a)` is treated as `nonzero(atleast_1d(a))`.

Deprecated since version 1.17.0: Use *atleast_1d* explicitly if this behavior is deliberate.

Parameters

a (*array_like*) – Input array.

Returns

tuple_of_arrays – Indices of elements that are non-zero.

Return type

tuple

See also:

flatnonzero

Return indices that are non-zero in the flattened version of the input array.

ndarray.nonzero

Equivalent ndarray method.

count_nonzero

Counts the number of non-zero elements in the input array.

Notes

While the nonzero values can be obtained with `a[nonzero(a)]`, it is recommended to use `x[x.astype(bool)]` or `x[x != 0]` instead, which will correctly handle 0-d arrays.

Examples

```
>>> x = np.array([[3, 0, 0], [0, 4, 0], [5, 6, 0]])
>>> x
array([[3, 0, 0],
       [0, 4, 0],
       [5, 6, 0]])
>>> np.nonzero(x)
(array([0, 1, 2, 2]), array([0, 1, 0, 1]))
```

```
>>> x[np.nonzero(x)]
array([3, 4, 5, 6])
>>> np.transpose(np.nonzero(x))
array([[0, 0],
       [1, 1],
```

(continues on next page)

(continued from previous page)

```
[2, 0],
[2, 1]])
```

A common use for `nonzero` is to find the indices of an array, where a condition is True. Given an array *a*, the condition *a* > 3 is a boolean array and since False is interpreted as 0, `np.nonzero(a > 3)` yields the indices of the *a* where the condition is true.

```
>>> a = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> a > 3
array([[False, False, False],
       [ True,  True,  True],
       [ True,  True,  True]])
>>> np.nonzero(a > 3)
(array([1, 1, 1, 2, 2, 2]), array([0, 1, 2, 0, 1, 2]))
```

Using this result to index *a* is equivalent to using the mask directly:

```
>>> a[np.nonzero(a > 3)]
array([4, 5, 6, 7, 8, 9])
>>> a[a > 3] # prefer this spelling
array([4, 5, 6, 7, 8, 9])
```

`nonzero` can also be called as a method of the array.

```
>>> (a > 3).nonzero()
(array([1, 1, 1, 2, 2, 2]), array([0, 1, 2, 0, 1, 2]))
```

static `astype(data, dtype, copy=True)`

static `tensor(data, device=NoParam)`

static `numpy(data)`

static `tolist(data)`

static `contiguous(data)`

static `pad(data, pad_width, mode='constant')`

static `asarray(data, dtype=None)`

Cast data into a numpy representation

static `ensure(data, dtype=None)`

Cast data into a numpy representation

static `dtype_kind(data)`

static `floor(data, out=None)`

static `ceil(data, out=None)`

static `ifloor(data, out=None)`

static `iceil(data, out=None)`

static `round(data, decimals=0, out=None)`

static `iround(data, out=None, dtype=<class 'int'>)`

static `clip(a, a_min, a_max, out=None, **kwargs)`

Clip (limit) the values in an array.

Given an interval, values outside the interval are clipped to the interval edges. For example, if an interval of `[0, 1]` is specified, values smaller than 0 become 0, and values larger than 1 become 1.

Equivalent to but faster than `np.minimum(a_max, np.maximum(a, a_min))`.

No check is performed to ensure `a_min < a_max`.

Parameters

- **a** (*array_like*) – Array containing elements to clip.
- **a_min, a_max** (*array_like or None*) – Minimum and maximum value. If `None`, clipping is not performed on the corresponding edge. Only one of `a_min` and `a_max` may be `None`. Both are broadcast against `a`.
- **out** (*ndarray, optional*) – The results will be placed in this array. It may be the input array for in-place clipping. `out` must be of the right shape to hold the output. Its type is preserved.
- ****kwargs** – For other keyword-only arguments, see the [ufunc docs](#).

New in version 1.17.0.

Returns

clipped_array – An array with the elements of `a`, but where values `< a_min` are replaced with `a_min`, and those `> a_max` with `a_max`.

Return type

`ndarray`

See also:

[Output type determination](#)

Notes

When `a_min` is greater than `a_max`, `clip` returns an array in which all values are equal to `a_max`, as shown in the second example.

Examples

```
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.clip(a, 1, 8)
array([1, 1, 2, 3, 4, 5, 6, 7, 8, 8])
>>> np.clip(a, 8, 1)
array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1])
>>> np.clip(a, 3, 6, out=a)
array([3, 3, 3, 3, 4, 5, 6, 6, 6, 6])
>>> a
array([3, 3, 3, 3, 4, 5, 6, 6, 6, 6])
>>> a = np.arange(10)
>>> a
```

(continues on next page)

(continued from previous page)

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.clip(a, [3, 4, 1, 1, 1, 4, 4, 4, 4, 4], 8)
array([3, 4, 2, 3, 4, 5, 6, 7, 8, 8])
```

static softmax(data, axis=None)

static kron(a, b)

Outer product

Parameters

- **a** (*ndarray*) – Array of shape (r0, r1, ... rN)
- **b** (*ndarray*) – Array of shape (s0, s1, ... sN)

Returns

with shape (r0*s0, r1*s1, ..., rN*sN).

Return type

ndarray

Note: # From numpy doc $\text{kron}(a,b)[k_0,k_1,\dots,k_N] = a[i_0,i_1,\dots,i_N] * b[j_0,j_1,\dots,j_N]$

$kt = it * st + jt, t = 0, \dots, N$

Notes

If $a.\text{shape} = (r_0, r_1, \dots, r_N)$ and $b.\text{shape} = (s_0, s_1, \dots, s_N)$, the Kronecker product has shape $(r_0*s_0, r_1*s_1, \dots, r_N*s_N)$.

class kwarray.arrayapi.ArrayAPI

Bases: [object](#)

Compatibility API between torch and numpy.

The API defines classmethods that work on both Tensors and ndarrays. As such the user can simply use `kwarray.ArrayAPI.<funcname>` and it will return the expected result for both Tensor and ndarray types.

However, this is inefficient because it requires us to check the type of the input for every API call. Therefore it is recommended that you use the [ArrayAPI.coerce\(\)](#) function, which takes as input the data you want to operate on. It performs the type check once, and then returns another object that defines with an identical API, but specific to the given data type. This means that we can ignore type checks on future calls of the specific implementation. See examples for more details.

Example

```
>>> # Use the easy-to-use, but inefficient array api
>>> # xdoctest: +REQUIRES(module:torch)
>>> import kwarray
>>> import torch
>>> take = kwarray.ArrayAPI.take
>>> np_data = np.arange(0, 143).reshape(11, 13)
>>> pt_data = torch.LongTensor(np_data)
>>> indices = [1, 3, 5, 7, 11, 13, 17, 21]
```

(continues on next page)

(continued from previous page)

```

>>> idxs0 = [1, 3, 5, 7]
>>> idxs1 = [1, 3, 5, 7, 11]
>>> assert np.allclose(take(np_data, indices), take(pt_data, indices))
>>> assert np.allclose(take(np_data, idxs0, 0), take(pt_data, idxs0, 0))
>>> assert np.allclose(take(np_data, idxs1, 1), take(pt_data, idxs1, 1))

```

Example

```

>>> # Use the easy-to-use, but inefficient array api
>>> # xdoctest: +REQUIRES(module:torch)
>>> import kwarray
>>> import torch
>>> compress = kwarray.ArrayAPI.compress
>>> np_data = np.arange(0, 143).reshape(11, 13)
>>> pt_data = torch.LongTensor(np_data)
>>> flags = (np_data % 2 == 0).ravel()
>>> f0 = (np_data % 2 == 0)[: , 0]
>>> f1 = (np_data % 2 == 0)[0, :]
>>> assert np.allclose(compress(np_data, flags), compress(pt_data, flags))
>>> assert np.allclose(compress(np_data, f0, 0), compress(pt_data, f0, 0))
>>> assert np.allclose(compress(np_data, f1, 1), compress(pt_data, f1, 1))

```

Example

```

>>> # Use ArrayAPI to coerce an identical API that doesnt do type checks
>>> # xdoctest: +REQUIRES(module:torch)
>>> import kwarray
>>> import torch
>>> np_data = np.arange(0, 15).reshape(3, 5)
>>> pt_data = torch.LongTensor(np_data)
>>> # The new ``impl`` object has the same API as ArrayAPI, but works
>>> # specifically on torch Tensors.
>>> impl = kwarray.ArrayAPI.coerce(pt_data)
>>> flat_data = impl.view(pt_data, -1)
>>> print('flat_data = {!r}'.format(flat_data))
flat_data = tensor([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
>>> # The new ``impl`` object has the same API as ArrayAPI, but works
>>> # specifically on numpy ndarrays.
>>> impl = kwarray.ArrayAPI.coerce(np_data)
>>> flat_data = impl.view(np_data, -1)
>>> print('flat_data = {!r}'.format(flat_data))
flat_data = array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])

```

_torch

alias of *TorchImpls*

_numpy

alias of *NumpyImpls*

static impl(data)

Returns a namespace suitable for operating on the input data type

Parameters

data (*ndarray* | *Tensor*) – data to be operated on

static coerce(data)

Coerces some form of inputs into an array api (either numpy or torch).

static result_type(*arrays_and_dtypes)

Return type from promotion rules

Example

```
>>> import kwarray
>>> kwarray.ArrayAPI.result_type(float, np.uint8, np.float32, np.float16)
>>> # xdoctest: +REQUIRES(module:torch)
>>> import torch
>>> kwarray.ArrayAPI.result_type(torch.int32, torch.int64)
```

static cat(datas, *args, **kwargs)

static hstack(datas, *args, **kwargs)

static vstack(datas, *args, **kwargs)

static take(data, *args, **kwargs)

static compress(data, *args, **kwargs)

static repeat(data, *args, **kwargs)

static tile(data, *args, **kwargs)

static view(data, *args, **kwargs)

static numel(data, *args, **kwargs)

static atleast_nd(data, *args, **kwargs)

static full_like(data, *args, **kwargs)

static ones_like(data, *args, **kwargs)

static zeros_like(data, *args, **kwargs)

static empty_like(data, *args, **kwargs)

static sum(data, *args, **kwargs)

static argsort(data, *args, **kwargs)

static argmax(data, *args, **kwargs)

static argmin(data, *args, **kwargs)

static max(data, *args, **kwargs)

```
static min(data, *args, **kwargs)
static max_argmax(data, *args, **kwargs)
static min_argmin(data, *args, **kwargs)
static maximum(data, *args, **kwargs)
static minimum(data, *args, **kwargs)
static matmul(data, *args, **kwargs)
static astype(data, *args, **kwargs)
static nonzero(data, *args, **kwargs)
static nan_to_num(data, *args, **kwargs)
static tensor(data, *args, **kwargs)
static numpy(data, *args, **kwargs)
static tolist(data, *args, **kwargs)
static asarray(data, *args, **kwargs)
static T(data, *args, **kwargs)
static transpose(data, *args, **kwargs)
static contiguous(data, *args, **kwargs)
static pad(data, *args, **kwargs)
static dtype_kind(data, *args, **kwargs)
static any(data, *args, **kwargs)
static all(data, *args, **kwargs)
static array_equal(data, *args, **kwargs)
static log2(data, *args, **kwargs)
static log(data, *args, **kwargs)
static copy(data, *args, **kwargs)
static ceil(data, *args, **kwargs)
static ifloor(data, *args, **kwargs)
static floor(data, *args, **kwargs)
static ceil(data, *args, **kwargs)
static round(data, *args, **kwargs)
static iround(data, *args, **kwargs)
static clip(data, *args, **kwargs)
```

static softmax(*data*, **args*, ***kwargs*)

kwarray.arrayapi.TorchNumpyCompat

alias of [ArrayAPI](#)

kwarray.arrayapi._torch_dtype_lut()

kwarray.arrayapi.dtype_info(*dtype*)

Lookup datatype information

Parameters

dtype (*type*) – a numpy, torch, or python numeric data type

Returns

an iinfo of finfo structure depending on the input type.

Return type

[numpy.iinfo](#) | [numpy.finfo](#) | [torch.iinfo](#) | [torch.finfo](#)

References

..[DtypeNotes] https://higra.readthedocs.io/en/stable/_modules/higra/hg_utils.html#dtype_info

Example

```
>>> from kwarray.arrayapi import * # NOQA
>>> try:
>>>     import torch
>>> except ImportError:
>>>     torch = None
>>> results = []
>>> results += [dtype_info(float)]
>>> results += [dtype_info(int)]
>>> results += [dtype_info(complex)]
>>> results += [dtype_info(np.float32)]
>>> results += [dtype_info(np.int32)]
>>> results += [dtype_info(np.uint32)]
>>> if hasattr(np, 'complex256'):
>>>     results += [dtype_info(np.complex256)]
>>> if torch is not None:
>>>     results += [dtype_info(torch.float32)]
>>>     results += [dtype_info(torch.int64)]
>>>     results += [dtype_info(torch.complex64)]
>>> for info in results:
>>>     print('info = {!r}'.format(info))
>>> for info in results:
>>>     print('info.bits = {!r}'.format(info.bits))
```

kwarray.dataframe_light module

A faster-than-pandas pandas-like interface to column-major data, in the case where the data only needs to be accessed by index.

For data where more complex ids are needed you must use pandas.

class kwarray.dataframe_light.LocLight(*parent*)

Bases: `object`

class kwarray.dataframe_light.DataFrameLight(*data=None, columns=None*)

Bases: `NiceRepr`

Implements a subset of the pandas.DataFrame API

The API is restricted to facilitate speed tradeoffs

Note: Assumes underlying data is Dict[list|ndarray]. If the data is known to be a Dict[ndarray] use DataFrameArray instead, which has faster implementations for some operations.

Note: pandas.DataFrame is slow. DataFrameLight is faster. It is a tad more restrictive though.

Example

```
>>> self = DataFrameLight({})
>>> print('self = {!r}'.format(self))
>>> self = DataFrameLight({'a': [0, 1, 2], 'b': [2, 3, 4]})
>>> print('self = {!r}'.format(self))
>>> item = self.iloc[0]
>>> print('item = {!r}'.format(item))
```

Benchmark

```
>>> # BENCHMARK
>>> # xdoc: +REQUIRES(--bench)
>>> from kwarray.dataframe_light import * # NOQA
>>> import ubelt as ub
>>> NUM = 1000
>>> print('NUM = {!r}'.format(NUM))
>>> # to_dict conversions
>>> print('=====')
>>> print('===== to_dict conversions =====')
>>> _keys = ['list', 'dict', 'series', 'split', 'records', 'index']
>>> results = []
>>> df = DataFrameLight._demodata(num=NUM).pandas()
>>> ti = ub.Timerit(verbose=False, unit='ms')
>>> for key in _keys:
>>>     result = ti.reset(key).call(lambda: df.to_dict(orient=key))
>>>     results.append((result.mean(), result.report()))
>>> key = 'series+numpy'
```

(continues on next page)

(continued from previous page)

```

>>> result = ti.reset(key).call(lambda: {k: v.values for k, v in df.to_dict(orient=
↳ 'series').items()})
>>> results.append((result.mean(), result.report()))
>>> print('\n'.join([t[1] for t in sorted(results)]))
>>> print('=====')
>>> print('===== DFLight Conversions =====')
>>> ti = ub.Timerit(verbose=True, unit='ms')
>>> key = 'self.pandas'
>>> self = DataFrameLight(df)
>>> ti.reset(key).call(lambda: self.pandas())
>>> key = 'light-from-pandas'
>>> ti.reset(key).call(lambda: DataFrameLight(df))
>>> key = 'light-from-dict'
>>> ti.reset(key).call(lambda: DataFrameLight(self._data))
>>> print('=====')
>>> print('===== BENCHMARK: .LOC[] =====')
>>> ti = ub.Timerit(num=20, bestof=4, verbose=True, unit='ms')
>>> df_light = DataFrameLight._demodata(num=NUM)
>>> # xdoctest: +REQUIRES(module:pandas)
>>> df_heavy = df_light.pandas()
>>> series_data = df_heavy.to_dict(orient='series')
>>> list_data = df_heavy.to_dict(orient='list')
>>> np_data = {k: v.values for k, v in df_heavy.to_dict(orient='series').items()}
>>> for timer in ti.reset('DF-heavy.iloc'):
>>>     with timer:
>>>         for i in range(NUM):
>>>             df_heavy.iloc[i]
>>> for timer in ti.reset('DF-heavy.loc'):
>>>     with timer:
>>>         for i in range(NUM):
>>>             df_heavy.iloc[i]
>>> for timer in ti.reset('dict[SERIES].loc'):
>>>     with timer:
>>>         for i in range(NUM):
>>>             {key: series_data[key].loc[i] for key in series_data.keys()}
>>> for timer in ti.reset('dict[SERIES].iloc'):
>>>     with timer:
>>>         for i in range(NUM):
>>>             {key: series_data[key].iloc[i] for key in series_data.keys()}
>>> for timer in ti.reset('dict[SERIES][]'):
>>>     with timer:
>>>         for i in range(NUM):
>>>             {key: series_data[key][i] for key in series_data.keys()}
>>> for timer in ti.reset('dict[NDARRAY][]'):
>>>     with timer:
>>>         for i in range(NUM):
>>>             {key: np_data[key][i] for key in np_data.keys()}
>>> for timer in ti.reset('dict[list][]'):
>>>     with timer:
>>>         for i in range(NUM):
>>>             {key: list_data[key][i] for key in np_data.keys()}
>>> for timer in ti.reset('DF-Light.iloc/loc'):

```

(continues on next page)

(continued from previous page)

```
>>>     with timer:
>>>         for i in range(NUM):
>>>             df_light.iloc[i]
>>> for timer in ti.reset('DF-Light._getrow'):
>>>     with timer:
>>>         for i in range(NUM):
>>>             df_light._getrow(i)
NUM = 1000
=====
===== to_dict conversions =====
Timed best=0.022 ms, mean=0.022 ± 0.0 ms for series
Timed best=0.059 ms, mean=0.059 ± 0.0 ms for series+numpy
Timed best=0.315 ms, mean=0.315 ± 0.0 ms for list
Timed best=0.895 ms, mean=0.895 ± 0.0 ms for dict
Timed best=2.705 ms, mean=2.705 ± 0.0 ms for split
Timed best=5.474 ms, mean=5.474 ± 0.0 ms for records
Timed best=7.320 ms, mean=7.320 ± 0.0 ms for index
=====
===== DFLight Conversions =====
Timed best=1.798 ms, mean=1.798 ± 0.0 ms for self.pandas
Timed best=0.064 ms, mean=0.064 ± 0.0 ms for light-from-pandas
Timed best=0.010 ms, mean=0.010 ± 0.0 ms for light-from-dict
=====
===== BENCHMARK: .LOC[] =====
Timed best=101.365 ms, mean=101.564 ± 0.2 ms for DF-heavy.iloc
Timed best=102.038 ms, mean=102.273 ± 0.2 ms for DF-heavy.loc
Timed best=29.357 ms, mean=29.449 ± 0.1 ms for dict[SERIES].loc
Timed best=21.701 ms, mean=22.014 ± 0.3 ms for dict[SERIES].iloc
Timed best=11.469 ms, mean=11.566 ± 0.1 ms for dict[SERIES][]
Timed best=0.807 ms, mean=0.826 ± 0.0 ms for dict[NDARRAY][]
Timed best=0.478 ms, mean=0.492 ± 0.0 ms for dict[list][]
Timed best=0.969 ms, mean=0.994 ± 0.0 ms for DF-Light.iloc/loc
Timed best=0.760 ms, mean=0.776 ± 0.0 ms for DF-Light._getrow
```

property `iloc`

property values

property `loc`

to_string(*args, **kwargs)

Return type

str

to_dict(orient='dict', into=<class 'dict'>)

Convert the data frame into a dictionary.

Parameters

- **orient** (*str*) – Currently naively supports orient in { 'dict', 'list' }, otherwise we fallback to pandas conversion and call its to_dict method.
- **into** (*type*) – type of dictionary to transform into

Returns

dict

Example

```
>>> from kwarray.dataframe_light import * # NOQA
>>> self = DataFrameLight._demodata(num=7)
>>> print(self.to_dict(orient='dict'))
>>> print(self.to_dict(orient='list'))
```

pandas()

Convert back to pandas if you need the full API

Return type

pd.DataFrame

Example

```
>>> # xdoctest: +REQUIRES(module:pandas)
>>> df_light = DataFrameLight._demodata(num=7)
>>> df_heavy = df_light.pandas()
>>> got = DataFrameLight(df_heavy)
>>> assert got._data == df_light._data
```

_pandas()

Deprecated, use self.pandas instead

classmethod _demodata(num=7)**Example**

```
>>> self = DataFrameLight._demodata(num=7)
>>> print('self = {!r}'.format(self))
>>> other = DataFrameLight._demodata(num=11)
>>> print('other = {!r}'.format(other))
>>> both = self.union(other)
>>> print('both = {!r}'.format(both))
>>> assert both is not self
>>> assert other is not self
```

property columns**sort_values**(key, inplace=False, ascending=True)**keys()****_getrow**(index)**_getcol**(key)**_getcols**(keys)

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

Get item for given key. Returns default value if not found.

clear()

Removes all rows inplace

compress(*flags*, *inplace=False*)

NOTE: NOT A PART OF THE PANDAS API

take(*indices*, *inplace=False*)

Return the elements in the given *positional* indices along an axis.

Parameters

inplace (*bool*) – NOT PART OF PANDAS API

Note: assumes axis=0

Example

```
>>> df_light = DataFrameLight._demodata(num=7)
>>> indices = [0, 2, 3]
>>> sub1 = df_light.take(indices)
>>> # xdoctest: +REQUIRES(module:pandas)
>>> df_heavy = df_light.pandas()
>>> sub2 = df_heavy.take(indices)
>>> assert np.all(sub1 == sub2)
```

copy()

extend(*other*)

Extend self inplace using another dataframe array

Parameters

other (*DataFrameLight* | *dict[str, Sequence]*) – values to concat to end of this object

Note: Not part of the pandas API

Example

```
>>> self = DataFrameLight(columns=['foo', 'bar'])
>>> other = {'foo': [0], 'bar': [1]}
>>> self.extend(other)
>>> assert len(self) == 1
```

union(**others*)

Note: Note part of the pandas API

classmethod `concat(others)`

classmethod `from_pandas(df)`

classmethod `from_dict(records)`

reset_index(drop=False)

noop for compatability, the light version doesnt store an index

groupby(*by=None, *args, **kwargs*)

Group rows by the value of a column. Unlike pandas this simply returns a zip object. To ensure compatiability call list on the result of groupby.

Parameters

- **by** (*str*) – column name to group by
- ***args** – if specified, the dataframe is coerced to pandas
- ***kwargs** – if specified, the dataframe is coerced to pandas

Example

```
>>> df_light = DataFrameLight._demodata(num=7)
>>> res1 = list(df_light.groupby('bar'))
>>> # xdoctest: +REQUIRES(module:pandas)
>>> df_heavy = df_light.pandas()
>>> res2 = list(df_heavy.groupby('bar'))
>>> assert len(res1) == len(res2)
>>> assert all([np.all(a[1] == b[1]) for a, b in zip(res1, res2)])
```

rename(*mapper=None, columns=None, axis=None, inplace=False*)

Rename the columns (index renaming is not supported)

Example

```
>>> df_light = DataFrameLight._demodata(num=7)
>>> mapper = {'foo': 'fi'}
>>> res1 = df_light.rename(columns=mapper)
>>> res3 = df_light.rename(mapper, axis=1)
>>> # xdoctest: +REQUIRES(module:pandas)
>>> df_heavy = df_light.pandas()
>>> res2 = df_heavy.rename(columns=mapper)
>>> res4 = df_heavy.rename(mapper, axis=1)
>>> assert np.all(res1 == res2)
>>> assert np.all(res3 == res2)
>>> assert np.all(res3 == res4)
```

iterrows()

Iterate over rows as (index, Dict) pairs.

Yields

Tuple[int, Dict] – the index and a dictionary representing a row

Example

```
>>> from kwarray.dataframe_light import * # NOQA
>>> self = DataFrameLight._demodata(num=3)
>>> print(ub.urepr(list(self.iterrows()), sort=1))
[
  (0, {'bar': 0, 'baz': 2.73, 'foo': 0}),
  (1, {'bar': 1, 'baz': 2.73, 'foo': 0}),
  (2, {'bar': 2, 'baz': 2.73, 'foo': 0}),
]
```

Benchmark

```
>>> # xdoc: +REQUIRES(--bench)
>>> from kwarray.dataframe_light import * # NOQA
>>> import ubelt as ub
>>> df_light = DataFrameLight._demodata(num=1000)
>>> df_heavy = df_light.pandas()
>>> ti = ub.Timerit(21, bestof=3, verbose=2, unit='ms')
>>> ti.reset('light').call(lambda: list(df_light.iterrows()))
>>> ti.reset('heavy').call(lambda: list(df_heavy.iterrows()))
>>> # xdoctest: +IGNORE_WANT
Timed light for: 21 loops, best of 3
  time per loop: best=0.834 ms, mean=0.850 ± 0.0 ms
Timed heavy for: 21 loops, best of 3
  time per loop: best=45.007 ms, mean=45.633 ± 0.5 ms
```

class kwarray.dataframe_light.DataFrameArray(data=None, columns=None)

Bases: *DataFrameLight*

DataFrameLight assumes the backend is a Dict[list] DataFrameArray assumes the backend is a Dict[ndarray]

Take and compress are much faster, but extend and union are slower

extend(other)

compress(flags, inplace=False)

take(indices, inplace=False)

kwarray.distributions module

Defines data structures for efficient repeated sampling of specific distributions (e.g. Normal, Uniform, Binomial) with specific parameters.

Inspired by `~/code/imgaug/imgaug/parameters.py`

Similar Libraries:

- <https://docs.pymc.io/api/distributions.html>
- <https://github.com/phobson/paramnormal>

Todo:

- [] change sample shape to just a single num.
- [] Some Distributions will output vectors. Maybe we could just postpend the dimensions?
- [] Expose as kwstats?
- [] Improve coerce syntax for concise distribution specification

References

<https://stackoverflow.com/questions/21100716/fast-arbitrary-distribution-random-sampling> <https://stackoverflow.com/questions/4265988/generate-random-numbers-with-a-given-numerical-distribution> <https://codereview.stackexchange.com/questions/196286/inverse-transform-sampling>

class kwarray.distributions.Value(*default=None, min=None, max=None, help=None, constraints=None, type=None, name=None*)

Bases: NiceRepr

Container for class `__params__` values.

Used to store metadata about distribution arguments, including default values, numeric constraints, typing, and help text.

Example

```
>>> from kwarray.distributions import * # NOQA
>>> self = Value(43.5)
>>> print(Value(name='lucy'))
>>> print(Value(name='jeff', default=1))
>>> self = Value(name='fred', default=1.0)
>>> print('self = {}'.format(ub.urepr(self, nl=1)))
>>> print(Value(name='bob', default=1.0, min=-5, max=5))
>>> print(Value(name='alice', default=1.0, min=-5))
```

sample(*rng*)

Get a random value for this parameter.

kwarray.distributions._issubclass2(*child, parent*)

Uses string comparisons to avoid ipython reload errors. Much less robust though.

kwarray.distributions._isinstance2(*obj, cls*)

Internal hacked version of `isinstance` for debugging

`obj = self`
`cls = distributions.Distribution`

`child = obj.__class__`
`parent = cls`

class kwarray.distributions.Parameterized

Bases: NiceRepr

Keeps track of all registered params and classes with registered params

_setparam(*key, value*)

_setchild(*key, value*)

children()

seed(*rng=None*)

parameters()

Returns parameters in this object and its children

_body_str()

idstr(*nl=None, thresh=80*)

Example

```
>>> # xdoctest: +REQUIRES(module:scipy)
>>> self = TruncNormal()
>>> self.idstr()
>>> #
>>> #
>>> class Dummy(Distribution):
>>>     def __init__(self):
>>>         super(Dummy, self).__init__()
>>>         self._setparam('a', 3)
>>>         self.b = Normal()
>>>         self.c = Uniform()
>>> self = Dummy()
>>> print(self.idstr())
>>> #
>>> class Tail5(Distribution):
>>>     def __init__(self):
>>>         super(Tail5, self).__init__()
>>>         self._setparam('a_parameter', 3)
>>>         for i in range(5):
>>>             self._setparam(chr(i + 97), i)
>>> #
>>> class Tail6(Distribution):
>>>     def __init__(self):
>>>         super(Tail6, self).__init__()
>>>         for i in range(9):
>>>             self._setparam(chr(i + 97) + '_parameter', i)
>>> #
>>> class Dummy2(Distribution):
>>>     def __init__(self):
>>>         super(Dummy2, self).__init__()
>>>         self._setparam('x', 3)
>>>         self._setparam('y', 3)
>>>         self.d = Dummy()
>>>         self.f = Tail6()
>>>         self.y = Tail5()
>>> self = Dummy2()
>>> print(self.idstr())
>>> print(ub.urepr(self.json_id()))
```

json_id()

```
_make_body(self_part, child_part, nl=None, thresh=80)
```

```
class kwarray.distributions.ParameterizedList(items)
```

Bases: [Parameterized](#)

Example

```
>>> from kwarray import distributions as stoch
>>> self1 = stoch.ParameterizedList([
>>>     stoch.Normal(),
>>>     stoch.Uniform(),
>>> ])
>>> print(self1.idstr())
>>> self = stoch.ParameterizedList([stoch.ParameterizedList([
>>>     stoch.Normal(),
>>>     stoch.Uniform(),
>>>     self1,
>>> ])]])
>>> print(self.idstr())
>>> print(self.idstr(0))
```

```
_setparam(key, value)
```

```
append(item)
```

```
idstr(nl=None, thresh=80)
```

```
class kwarray.distributions._BinOpMixin
```

Bases: [object](#)

Allows binary operations to be performed on distributions to create composed distributions.

```
int()
```

```
round(ndigits=None)
```

```
clip(a_min=None, a_max=None)
```

```
log()
```

```
log10()
```

```
exp()
```

```
sqrt()
```

```
abs()
```

```
class kwarray.distributions._RBinOpMixin
```

Bases: [_BinOpMixin](#)

<https://docs.python.org/3/reference/datamodel.html>

```
class kwarray.distributions.Distribution(*args, **kwargs)
```

Bases: [Parameterized](#), [_RBinOpMixin](#)

Base class for all distributions.

There are 3 main subtypes:

ContinuousDistribution DiscreteDistribution MixedDistribution

Note: In [DiscVsCont] notes that there are only 3 types of random variables: discrete, continuous, or mixed. And these types are mutually exclusive.

Note: When inheriting from this class, you typically do not need to define an `__init__` method. Instead, overwrite the `__params__` class attribute with an `OrderedDict[str, Value]` to indicate what the signature of the `__init__` method should be. This allows for (1) concise expression of new distributions and (2) for new distributions to inherit a `random` classmethod that works according to constraints specified in each parameter Value.

If you do overwrite `__init__`, be sure to call `super()`.

References

seed(*rng=None*)

sample(**shape*)

classmethod random(*rng=None*)

Returns a random distribution

Parameters

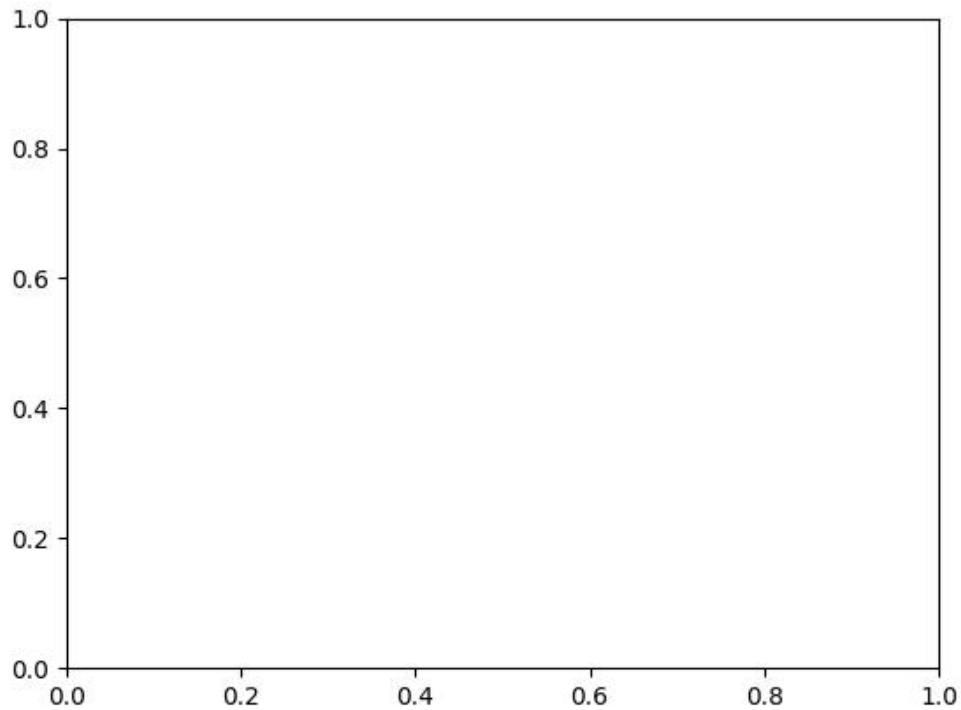
rng (*int* | *float* | *None* | *numpy.random.RandomState* | *random.Random*) – random coercable

CommandLine

```
xdoctest -m /home/joncrall/code/kwarray/kwarray/distributions.py Distribution.  
↪ random --show
```

Example

```
>>> # xdoctest: +REQUIRES(module:scipy)  
>>> from kwarray.distributions import * # NOQA  
>>> self = Distribution.random()  
>>> print('self = {!r}'.format(self))  
>>> # xdoctest: +REQUIRES(--show)  
>>> import kwplot  
>>> kwplot.autompl()  
>>> kwplot.figure(fnum=1, doclf=True)  
>>> self.plot('0.001s', bins=256)  
>>> kwplot.show_if_requested()
```

classmethod `coerce(arg, rng=None)`

classmethod `cast(arg)`

classmethod `seeded(rng=0)`

plot(`n='0.01s'`, `bins='auto'`, `stat='count'`, `color=None`, `kde=True`, `ax=None`, `**kwargs`)

Plots n samples from the distribution.

Parameters

- **bins** (`int` | `List[Number]` | `str`) – number of bins, bin edges, or special numpy method for finding the number of bins.
- **stat** (`str`) – density, count, probability, frequency
- ****kwargs** – other args passed to `seaborn.histplot()`

Example

```
>>> from kwarray.distributions import Normal # NOQA
>>> self = Normal()
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autopl()
>>> self.plot(n=1000)
```

```
_show(n, bins=None, ax=None, color=None, label=None)
```

plot samples monte-carlo style

```
kwarray.distributions._coerce_timedelta(data)
```

```
kwarray.distributions._generate_on_a_time_budget(func, maxiters, budget)
```

budget = 60

```
class kwarray.distributions.DiscreteDistribution(*args, **kwargs)
```

Bases: [Distribution](#)

```
class kwarray.distributions.ContinuousDistribution(*args, **kwargs)
```

Bases: [Distribution](#)

```
class kwarray.distributions.MixedDistribution(*args, **kwargs)
```

Bases: [Distribution](#)

```
class kwarray.distributions.Mixture(pdf, weights=None, rng=None)
```

Bases: [MixedDistribution](#)

Creates a mixture model of multiple distributions

Contains a set of distributions with associated weights. Sampling is done by first choosing a distribution with probability proportional to its weighing, and then sampling from the chosen distribution.

In general, a mixture model generates data by first first we sample from z , and then we sample the observables x from a distribution which depends on z , i.e. $p(z, x) = p(z) p(x | z)$ [[GrosseMixture](#)] [[StephensMixture](#)].

Parameters

- **pdfs** (*List[Distribution]*) – list of distributions
- **weights** (*List[float]*) – corresponding weights of each distribution
- **rng** (*np.random.RandomState*) – seed random number generator

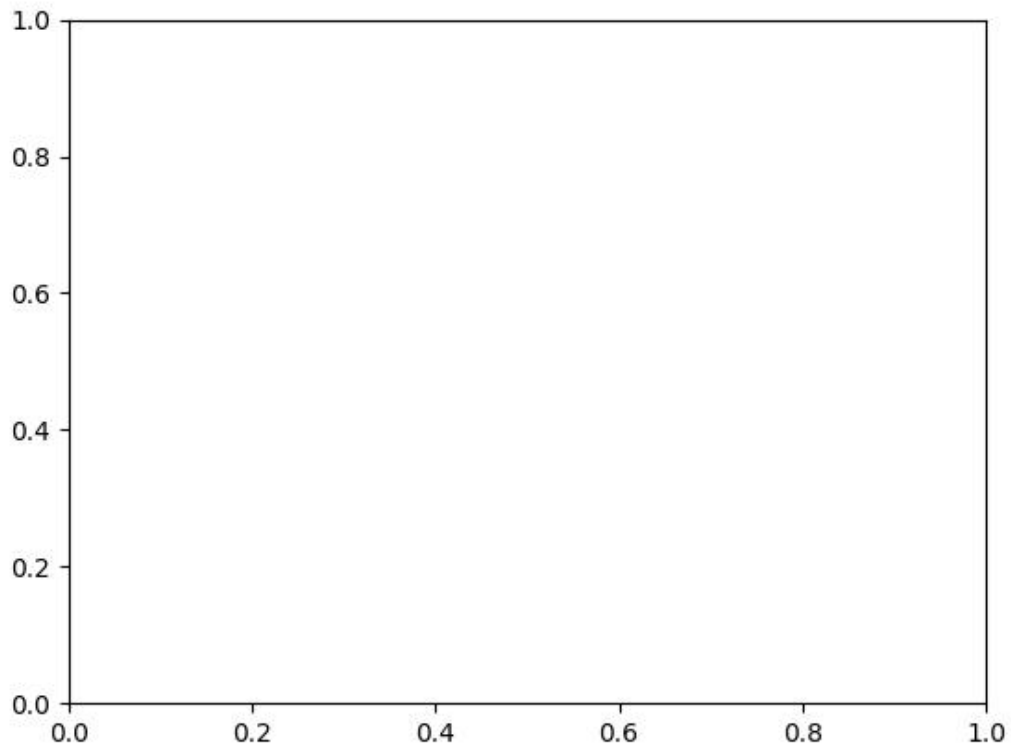
References

CommandLine

```
xdoctest -m kwarray.distributions Mixture:0 --show
```

Example

```
>>> # In this examle we create a bimodal mixture of normals
>>> from kwarray.distributions import * # NOQA
>>> pdfs = [Normal(mean=10, std=2), Normal(18, 2)]
>>> self = Mixture(pdfs)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.figure(fnum=1, doclf=True)
>>> self.plot(500, bins=25)
>>> kwplot.show_if_requested()
```



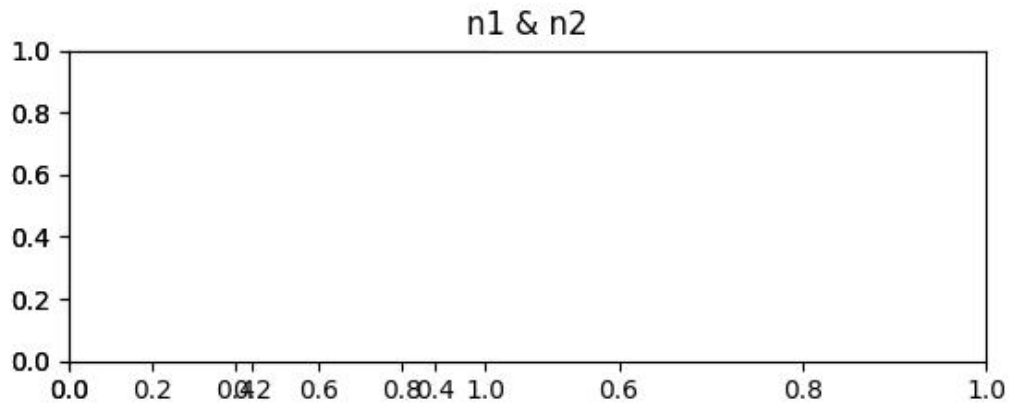
Example

```
>>> # Compare Composed versus Mixture Distributions
>>> # Given two normal distributions,
>>> from kwarray.distributions import Normal # NOQA
>>> from kwarray.distributions import * # NOQA
>>> n1 = Normal(mean=11, std=3)
>>> n2 = Normal(mean=53, std=5)
>>> composed = (n1 * 0.3) + (n2 * 0.7)
>>> mixture = Mixture([n1, n2], [0.3, 0.7])
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.figure(fnum=1, pnum=(2, 2, 1))
>>> ax = kwplot.figure(pnum=(2, 1, 1), title='n1 & n2').gca()
>>> n = 10000
>>> plotkw = dict(stat='density', kde=1, bins=1000)
>>> plotkw = dict(stat='count', kde=1, bins=1000)
>>> #plotkw = dict(stat='frequency', kde=1, bins='auto')
>>> n1.plot(n, ax=ax, **plotkw)
>>> n2.plot(n, ax=ax, **plotkw)
>>> ax=kwplot.figure(pnum=(2, 2, 3), title='composed').gca()
>>> composed.plot(n, ax=ax, **plotkw)
```

(continues on next page)

(continued from previous page)

```
>>> ax=kwplot.figure(pnum=(2, 2, 4), title='mixture').gca()
>>> mixture.plot(n, ax=ax, **plotkw)
>>> kwplot.show_if_requested()
```



sample(*shape)

Sampling from a mixture of k distributions with weights w_k is equivalent to picking a distribution with probability w_k , and then sampling from the picked distribution. *SUser6655984*
<<https://stackoverflow.com/a/47762586/887074>>

classmethod random(rng=None, n=3)

Parameters

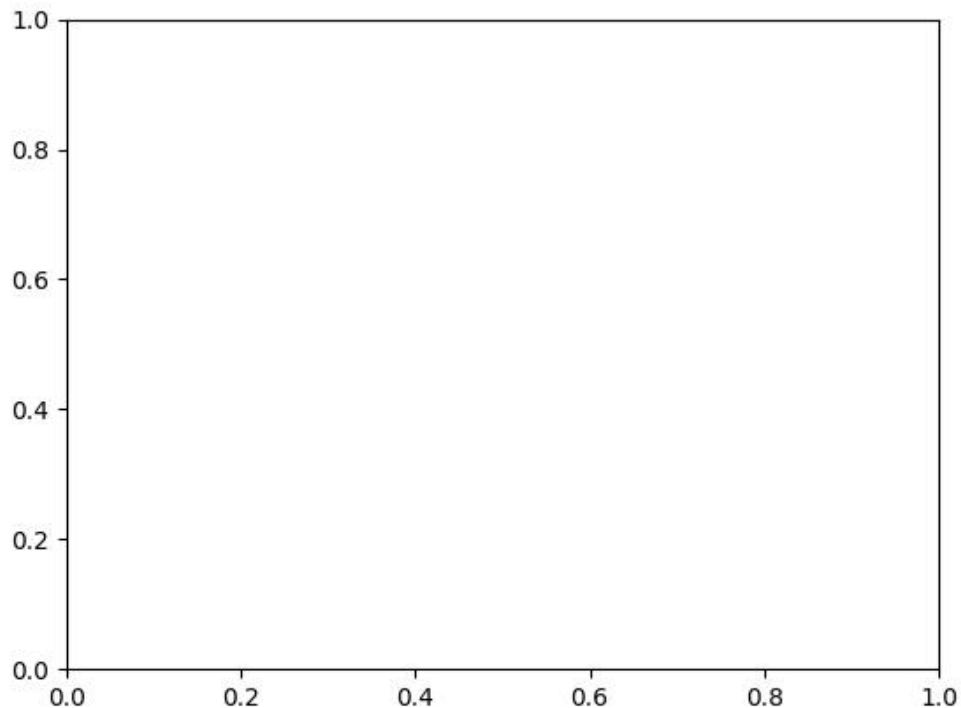
- **rng** (*int* | *float* | *None* | *numpy.random.RandomState* | *random.Random*) – random coercable
- **n** (*int*) – number of random distributions in the mixture

Example

```

>>> # xdoctest: +REQUIRES(module:scipy)
>>> from kwarray.distributions import * # NOQA
>>> print('Mixture = {!r}'.format(Mixture))
>>> print('Mixture = {!r}'.format(dir(Mixture)))
>>> self = Mixture.random(3)
>>> print('self = {!r}'.format(self))
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.figure(fnum=1, doclf=True)
>>> self.plot('0.1s', bins=256)
>>> kwplot.show_if_requested()

```



class kwarray.distributions.**Composed**(*args, **kwargs)

Bases: [MixedDistribution](#)

A distribution generated by composing different base distributions or numbers (which are considered as constant distributions).

Given the operation and its arguments, the sampling process of a “Composed” distribution will sample from each of the operands, and then apply the operation to the sampled points. For instance if we add two Normal distributions, this will first sample from each distribution and then add the results.

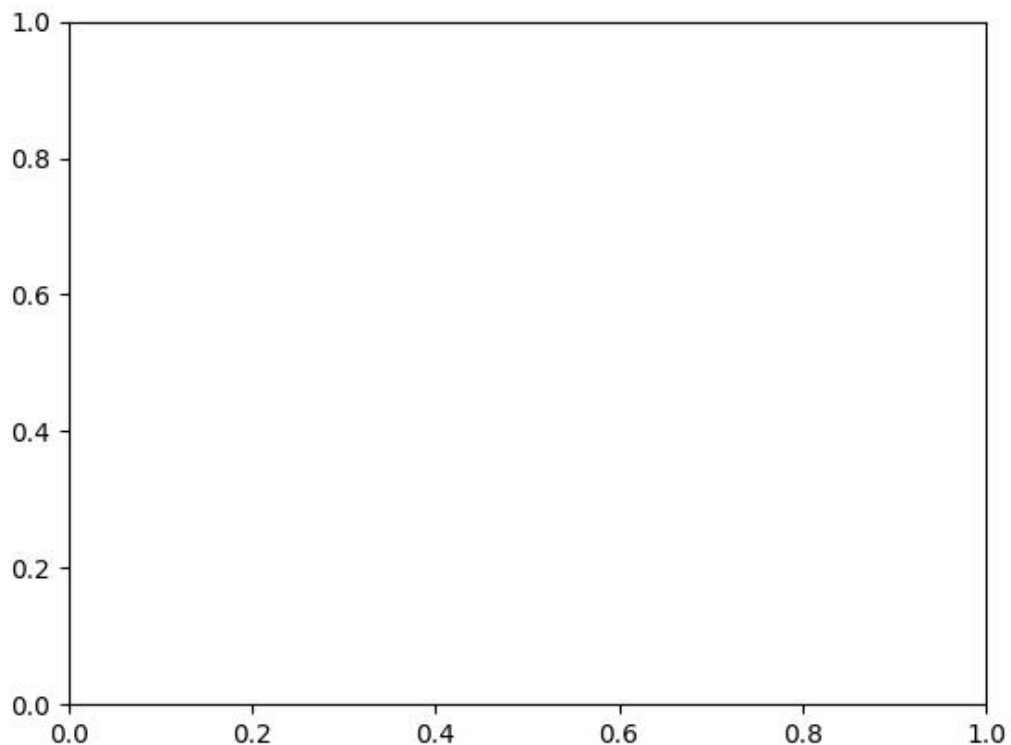
Note: This is not the same as mixing distributions!

Variables

- **self.operation** (*Function*) – operation (add / sub / mult / div) to perform on operands
- **self.operands** (*Sequence*[*Distribution* | *Number*]) – arguments passed to operation

Example

```
>>> # In this examle you can see that the sum of two Normal random
>>> # variables is also normal
>>> from kwarray.distributions import * # NOQA
>>> operands = [Normal(mean=10, std=2), Normal(15, 2)]
>>> operation = np.add
>>> self = Composed(operation, operands)
>>> data = self.sample(5)
>>> print(ub.urepr(list(data), nl=0, precision=5))
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.figure(fnum=1, doclf=True)
>>> self.plot(1000, bins=100)
```

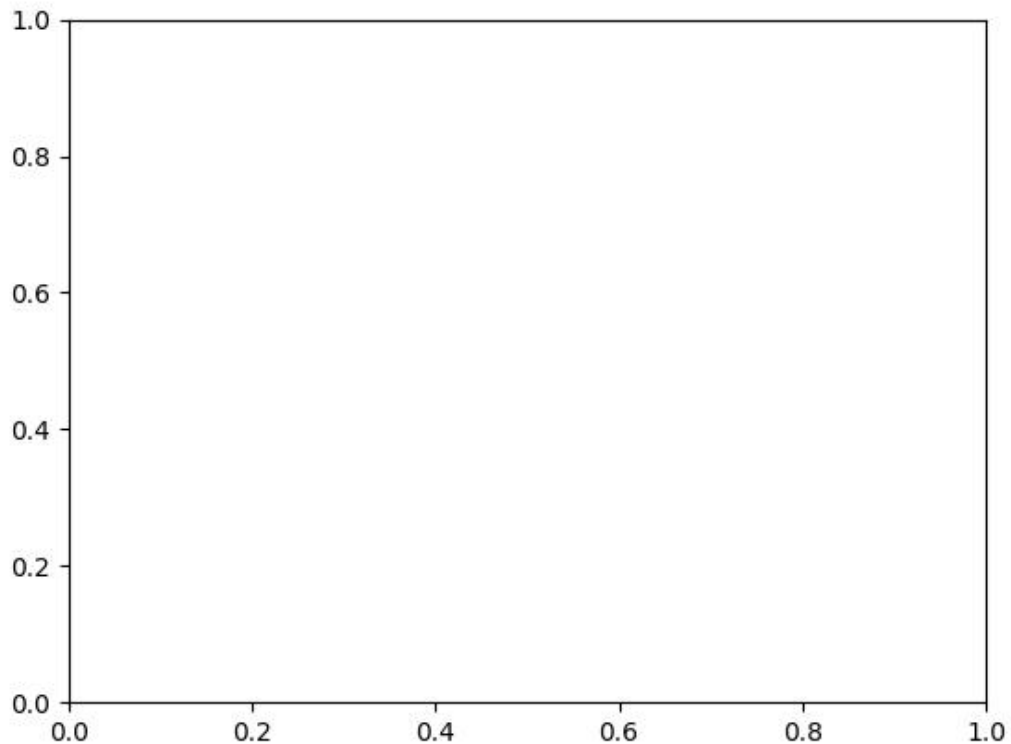


Example

```

>>> # Binary operations result in composed distributions
>>> # We can make a (bounded) exponential distribution using a uniform
>>> from kwarray.distributions import * # NOQA
>>> X = Uniform(.001, 7)
>>> lam = .7
>>> e = np.exp(1)
>>> self = lam * e ** (-lam * X)
>>> data = self.sample(5)
>>> print(ub.urepr(list(data), nl=0, precision=5))
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.figure(fnum=1, doclf=True)
>>> self.plot(5000, bins=100)

```



Parameters

- **operation** (*Any*) – no help given. Defaults to None.
- **operands** (*Any*) – no help given. Defaults to None.

sample(**shape*)

kwarray.distributions._**trysample**(*arg*, *shape*)

samples if *arg* is a distribution, otherwise returns *arg*

exception kwarray.distributions.CoerceError

Bases: [ValueError](#)

kwarray.distributions.CastError

alias of [CoerceError](#)

class kwarray.distributions.Uniform(*args, **kwargs)

Bases: [ContinuousDistribution](#)

Defaults to a uniform distribution over floats between 0 and 1

Example

```
>>> from kwarray.distributions import * # NOQA
>>> self = Uniform(rng=0)
>>> self.sample()
0.548813...
>>> float(self.sample(1))
0.7151...
```

Benchmark

```
>>> import ubelt as ub
>>> self = Uniform()
>>> for timer in ub.Timerit(100, bestof=10):
>>>     with timer:
>>>         [self() for _ in range(100)]
>>> for timer in ub.Timerit(100, bestof=10):
>>>     with timer:
>>>         self(100)
```

Parameters

- **high** (*int*) – no help given. Defaults to 1.
- **low** (*int*) – no help given. Defaults to 0.

sample(*shape)

classmethod **coerce**(arg)

class kwarray.distributions.Exponential(*args, **kwargs)

Bases: [ContinuousDistribution](#)

The exponential distribution is the probability distribution of the time between events in a Poisson point process, i.e., a process in which events occur continuously and independently at a constant average rate¹.

References:

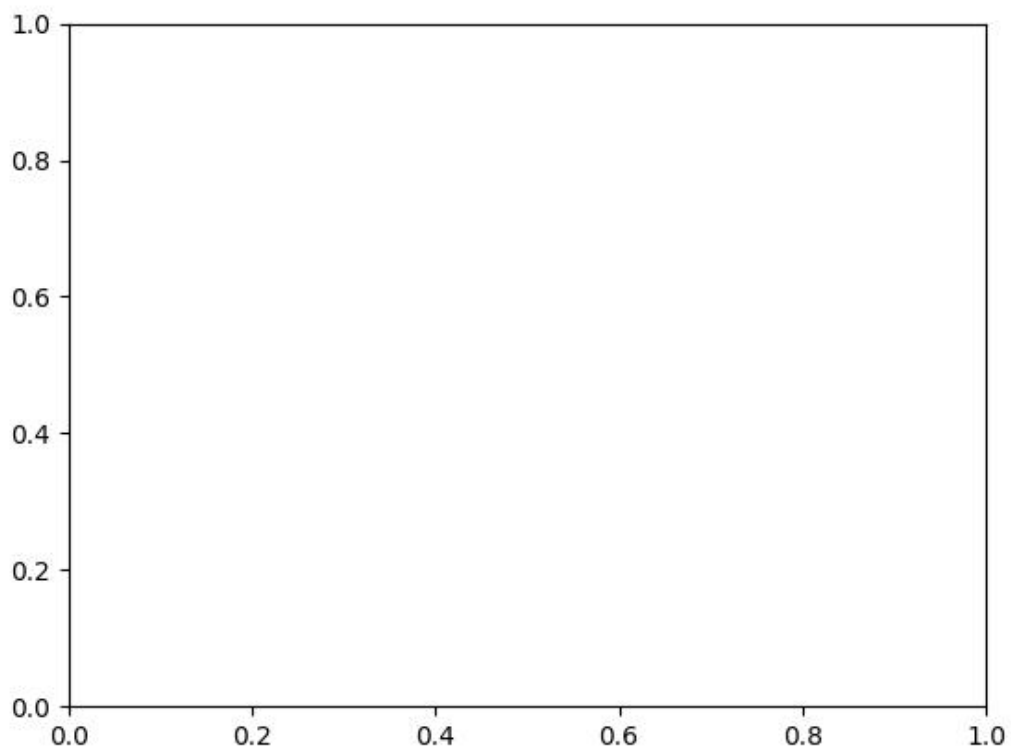
¹ https://en.wikipedia.org/wiki/Exponential_distribution

Example

```

>>> from kwarray.distributions import * # NOQA
>>> self = Exponential(rng=0)
>>> self.sample()
>>> self.sample(2, 3)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.figure(fnum=1, doclf=True)
>>> self.plot(500, bins=25)

```



Parameters

scale (*int*) – no help given. Defaults to 1.

sample (**shape*)

class kwarray.distributions.**Constant** (**args*, ***kwargs*)

Bases: *DiscreteDistribution*

Example

```
>>> self = Constant(42, rng=0)
>>> self.sample()
42
>>> self.sample(3)
array([42, 42, 42])
```

Parameters

value (*int*) – constant value. Defaults to 1.

sample(**shape*)

class kwarray.distributions.**DiscreteUniform**(**args, **kwargs*)

Bases: [DiscreteDistribution](#)

Uniform distribution over integers.

Parameters

- **min** (*int*) – inclusive minimum
- **max** (*int*) – exclusive maximum

Example

```
>>> self = DiscreteUniform.coerce(4)
>>> self.sample(100)
```

sample(**shape*)

classmethod **coerce**(*arg, rng=None*)

class kwarray.distributions.**Normal**(**args, **kwargs*)

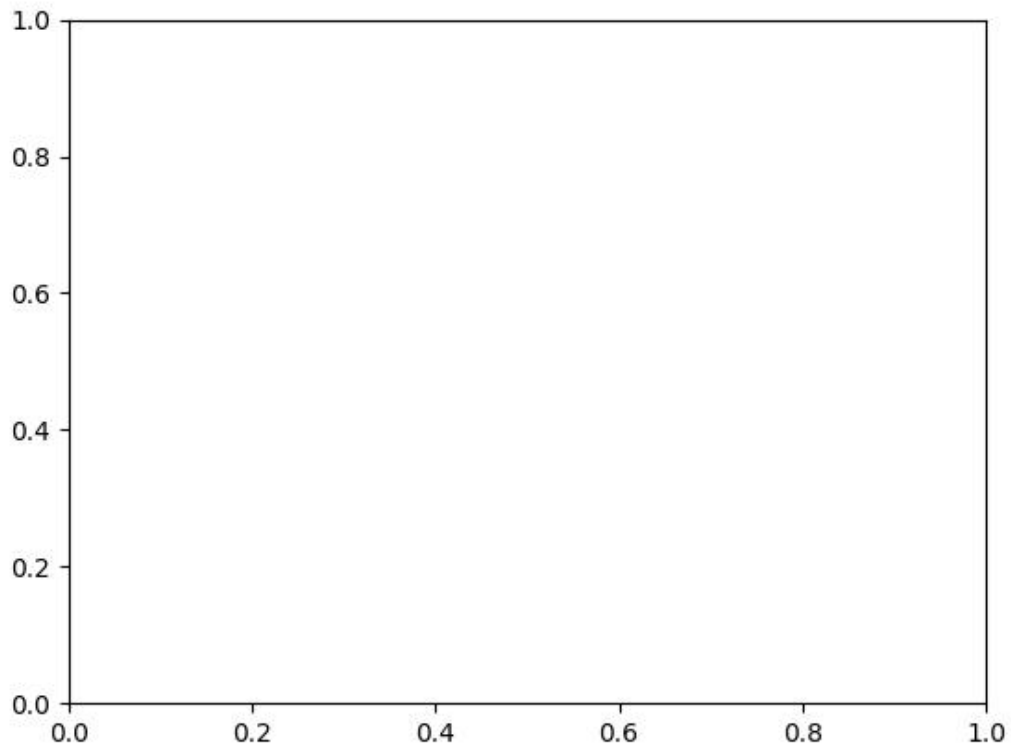
Bases: [ContinuousDistribution](#)

A normal distribution. See [\[WikiNormal\]](#) [\[WikiCLT\]](#).

References

Example

```
>>> from kwarray.distributions import * # NOQA
>>> self = Normal(mean=100, rng=0)
>>> self.sample()
>>> self.sample(100)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.figure(fnum=1, doclf=True)
>>> self.plot(500, bins=25)
```

**Parameters**

- **mean** (*float*) – no help given. Defaults to 0.0.
- **std** (*float*) – no help given. Defaults to 1.0.

sample(**shape*)

classmethod random(*rng=None*)

class kwarray.distributions.**TruncNormal**(**args*, ***kwargs*)

Bases: [ContinuousDistribution](#)

A truncated normal distribution.

A normal distribution, but bounded by low and high values. Note this is much different from just using a clipped normal.

Parameters

- **mean** (*float*) – mean of the distribution
- **std** (*float*) – standard deviation of the distribution
- **low** (*float*) – lower bound
- **high** (*float*) – upper bound
- **rng** (*np.random.RandomState*)

References

https://en.wikipedia.org/wiki/Truncated_normal_distribution
generated/scipy.stats.truncnorm.html

<https://docs.scipy.org/doc/scipy/reference/>

CommandLine

```
xdoctest -m /home/joncrall/code/kwarray/kwarray/distributions.py TruncNormal
```

Example

```
>>> # xdoctest: +REQUIRES(module:scipy)
>>> self = TruncNormal(rng=0)
>>> self() # output of this changes before/after scipy version 1.5
...0.1226...
```

Example

```
>>> # xdoctest: +REQUIRES(module:scipy)
>>> from kwarray.distributions import * # NOQA
>>> low = -np.pi / 16
>>> high = np.pi / 16
>>> std = np.pi / 8
>>> self = TruncNormal(low=low, high=high, std=std, rng=0)
>>> shape = (3, 3)
>>> data = self(*shape)
>>> print(ub.urepr(data, precision=5))
np.array([[ 0.01841,  0.0817 ,  0.0388 ],
          [ 0.01692, -0.0288 ,  0.05517],
          [-0.02354,  0.15134,  0.18098]], dtype=np.float64)
```

_update_internals()

classmethod random(rng=None)

sample(*shape)

class kwarray.distributions.Bernoulli(*args, **kwargs)

Bases: *DiscreteDistribution*

The Bernoulli distribution is the discrete probability distribution of a random variable which takes the value 1 with probability p and the value 0 with probability $q = 1 - p$.

References

https://en.wikipedia.org/wiki/Bernoulli_distribution

Parameters

p (*float*) – probability of success. Defaults to 0.5.

sample(**shape*)

classmethod **coerce**(*arg*)

class kwarray.distributions.**Binomial**(**args*, ***kwargs*)

Bases: *DiscreteDistribution*

The Binomial distribution represents the discrete probabilities of obtaining some number of successes in n “binary-experiments” each with a probability of success p and a probability of failure 1 - p.

References

https://en.wikipedia.org/wiki/Binomial_distribution

Parameters

- **p** (*float*) – probability of success. Defaults to 0.5.
- **n** (*int*) – probability of success. Defaults to 1.

sample(**shape*)

class kwarray.distributions.**Categorical**(*categories*, *weights=None*, *rng=None*)

Bases: *DiscreteDistribution*

Example

```
>>> categories = [3, 5, 1]
>>> weights = [.05, .5, .45]
>>> self = Categorical(categories, weights, rng=0)
>>> self.sample()
5
>>> list(self.sample(2))
[1, 1]
>>> self.sample(2, 3)
array([[5, 5, 1],
       [5, 1, 1]])
```

Parameters

- **categories** (*Any*) – no help given. Defaults to None.
- **weights** (*Any*) – no help given. Defaults to None.

sample(**shape*)

class kwarray.distributions.**NonlinearUniform**(*min*, *max*, *nonlinearity*=None, *reverse*=False, *rng*=None)

Bases: [ContinuousDistribution](#)

Weighted sample between two points depending on some nonlinearity

Todo: could refactor part of this into a PowerLaw distribution

Parameters

nonlinearity (*func* or *str*) – needs to be a function that maps the range 0-1 to the range 0-1

Example

```
>>> self = NonlinearUniform(0, 100, np.sqrt, rng=0)
>>> print(ub.urepr(list(self.sample(2)), precision=2, nl=0))
[74.08, 84.57]
>>> print(ub.urepr(self.sample(2, 3), precision=2, nl=1))
np.array([[77.64, 73.82, 65.09],
          [80.37, 66.15, 94.43]], dtype=np.float64)
```

sample(**shape*)

class kwarray.distributions.**CategoryUniform**(*categories*=[None], *rng*=None)

Bases: [DiscreteUniform](#)

Discrete Uniform over a list of categories

Parameters

- **min** (*int*) – no help given. Defaults to 0.
- **max** (*int*) – no help given. Defaults to 1.

sample(**shape*)

class kwarray.distributions.**PDF**(*x*, *p*, *rng*=None)

Bases: [Distribution](#)

BROKEN?

Similar to Categorical, but interpolates to approximate a continuous random variable.

Returns a value *x* with probability *p*.

References

<http://www.nehalemlabs.net/prototype/blog/2013/12/16/how-to-do-inverse-transformation-sampling-in-scipy-and-numpy/>

Parameters

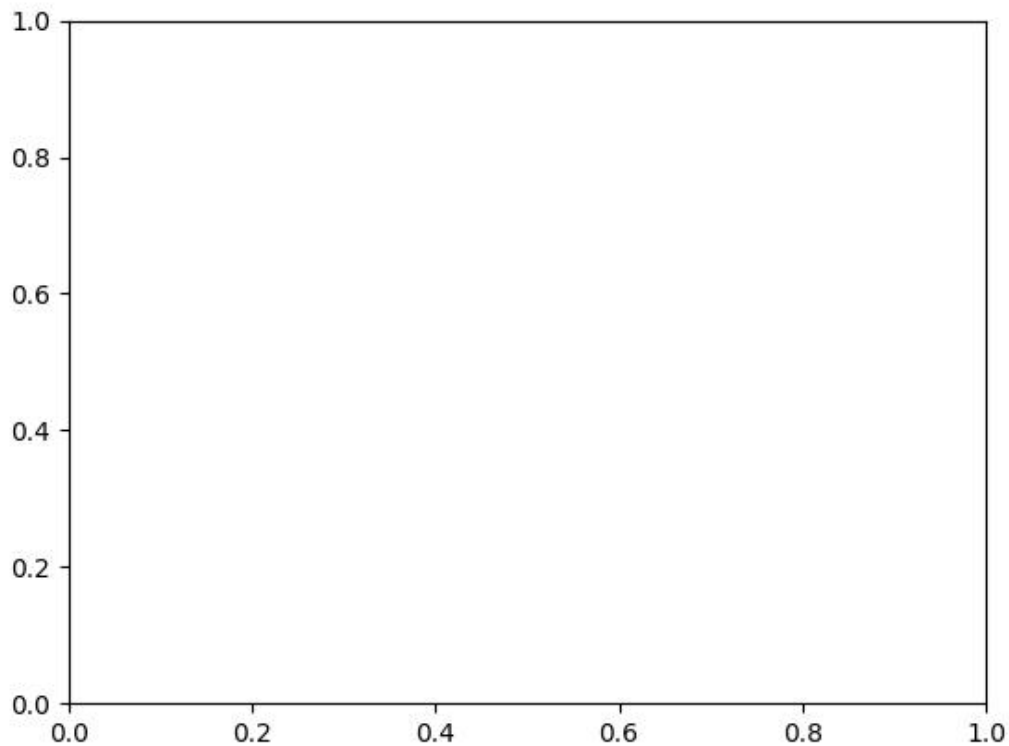
- **x** (*list* or *tuple*) – domain in which this PDF is defined
- **p** (*list*) – probability sample for each domain sample

Example

```

>>> # xdoctest: +REQUIRES(module:scipy)
>>> from kwarray.distributions import PDF # NOQA
>>> x = np.linspace(800, 4500)
>>> p = np.log10(x)
>>> p = x ** 2
>>> self = PDF(x, p)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.figure(fnum=1, doclf=True)
>>> self.plot(5000, bins=50)

```



sample(*shape)

class kwarray.distributions.**Seeded**(rng=None, cls=None)

Bases: `object`

Helper for grabbing pre-seeded distributions

kwarray.distributions.**_test_distributions**()

kwarray.distributions.**_process_docstrings**()

Iterate over the definitions with `__params__` defined and dynamically add relevant information to their docstrings. We should modify this so it can rewrite the docstrings statically. I don't like dynamic docstrings at runtime.

CommandLine

```
xdoctest -m kwarray.distributions _process_docstrings
```

Example

```
>>> # Show the results of the docstring formatting
>>> from kwarray import distributions
>>> candidates = []
>>> for val in distributions.__dict__.values():
>>>     if hasattr(val, '__params__') and val.__params__ is not NotImplemented:
>>>         candidates.append(val)
>>> for val in candidates:
>>>     print('=====' )
>>>     print(val)
>>>     print('-----')
>>>     print(val.__doc__)
>>>     print('=====' )
```

kwarray.fast_rand module

Fast 32-bit random functions for numpy as of 2018. (More recent versions of numpy may have these natively supported).

```
kwarray.fast_rand.uniform(low=0.0, high=1.0, size=None, dtype=<class 'numpy.float32'>, rng=<module
    'numpy.random' from
    '/home/docs/checkouts/readthedocs.org/user_builds/kwarray/envs/latest/lib/python3.11/site-
    packages/numpy/random/__init__.py'>)
```

Draws float32 samples from a uniform distribution.

Samples are uniformly distributed over the half-open interval `[low, high)` (includes low, but excludes high).

Parameters

- **low** (*float*) – Lower boundary of the output interval. All values generated will be greater than or equal to low. Defaults to 0.
- **high** (*float*) – Upper boundary of the output interval. All values generated will be less than high. Default to 1.
- **size** (*int* | *Tuple[int, ...]* | *None*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is *None* (default), a single value is returned if `low` and `high` are both scalars. Otherwise, `np.broadcast(low, high).size` samples are drawn.
- **dtype** (*type*) – either `np.float32` or `np.float64`. Defaults to `float32`
- **rng** (*numpy.random.RandomState*) – underlying random state

Returns

uniformly distributed random numbers with chosen size and dtype
Extended typing
`NDArray[Literal[size], Literal[dtype]]`

Return type

`ndarray`

Benchmark

```
>>> from timerit import Timerit
>>> import kwarray
>>> size = (300, 300, 3)
>>> for timer in Timerit(100, bestof=10, label='dtype=np.float32'):
>>>     rng = kwarray.ensure_rng(0)
>>>     with timer:
>>>         ours = standard_normal(size, rng=rng, dtype=np.float32)
>>> # Timed best=4.705 ms, mean=4.75 ± 0.085 ms for dtype=np.float32
>>> for timer in Timerit(100, bestof=10, label='dtype=np.float64'):
>>>     rng = kwarray.ensure_rng(0)
>>>     with timer:
>>>         theirs = standard_normal(size, rng=rng, dtype=np.float64)
>>> # Timed best=9.327 ms, mean=9.794 ± 0.4 ms for rng=np.float64
```

`kwarray.fast_rand.standard_normal`(size, mean=0, std=1, dtype=<class 'float'>, rng=<module 'numpy.random' from '/home/docs/checkouts/readthedocs.org/user_builds/kwarray/envs/latest/lib/python3.11/site-packages/numpy/random/__init__.py'>)

Draw samples from a standard Normal distribution with a specified mean and standard deviation.

Parameters

- **size** (*int* | *Tuple[int, ...]*) – shape of the returned ndarray
- **mean** (*float*) – mean of the normal distribution. defaults to 0
- **std** (*float*) – standard deviation of the normal distribution. defaults to 1.
- **dtype** (*type*) – either `np.float32` (default) or `np.float64`
- **rng** (*numpy.random.RandomState*) – underlying random state

Returns

normally distributed random numbers with chosen size and dtype Extended typing
`NDArray[Literal[size], Literal[dtype]]`

Return type

ndarray

Benchmark

```
>>> from timerit import Timerit
>>> import kwarray
>>> size = (300, 300, 3)
>>> for timer in Timerit(100, bestof=10, label='dtype=np.float32'):
>>>     rng = kwarray.ensure_rng(0)
>>>     with timer:
>>>         ours = standard_normal(size, rng=rng, dtype=np.float32)
>>> # Timed best=4.705 ms, mean=4.75 ± 0.085 ms for dtype=np.float32
>>> for timer in Timerit(100, bestof=10, label='dtype=np.float64'):
>>>     rng = kwarray.ensure_rng(0)
>>>     with timer:
>>>         theirs = standard_normal(size, rng=rng, dtype=np.float64)
>>> # Timed best=9.327 ms, mean=9.794 ± 0.4 ms for rng=np.float64
```

```
kwarray.fast_rand.standard_normal32(size, mean=0, std=1, rng=<module 'numpy.random' from  
                                     '/home/docs/checkouts/readthedocs.org/user_builds/kwarray/envs/latest/lib/python3.11/s  
                                     packages/numpy/random/__init__.py'>)
```

Fast normally distributed random variables.

Uses the Box–Muller transform [[WikiBoxMuller](#)].

The difference between this function and `numpy.random.standard_normal()` is that we use float32 arrays in the backend instead of float64. Halving the amount of bits that need to be manipulated can significantly reduce the execution time, and 32-bit precision is often good enough.

Parameters

- **size** (*int* | *Tuple[int, ...]*) – shape of the returned ndarray
- **mean** (*float*, *default=0*) – mean of the normal distribution
- **std** (*float*, *default=1*) – standard deviation of the normal distribution
- **rng** (*numpy.random.RandomState*) – underlying random state

Returns

normally distributed random numbers with chosen size.

Return type

ndarray[Any, Float32]

References

SeeAlso:

- `standard_normal()`
- `standard_normal64()`

Example

```
>>> # xdoctest: +REQUIRES(module:scipy)
>>> import scipy
>>> import scipy.stats
>>> pts = 1000
>>> # Our numbers are normally distributed with high probability
>>> rng = np.random.RandomState(28041990)
>>> ours_a = standard_normal32(pts, rng=rng)
>>> ours_b = standard_normal32(pts, rng=rng) + 2
>>> ours = np.concatenate((ours_a, ours_b)) # numerical stability?
>>> p = scipy.stats.normaltest(ours)[1]
>>> print('Probability our data is non-normal is: {:.4g}'.format(p))
Probability our data is non-normal is: 1.573e-14
>>> rng = np.random.RandomState(28041990)
>>> theirs_a = rng.standard_normal(pts)
>>> theirs_b = rng.standard_normal(pts) + 2
>>> theirs = np.concatenate((theirs_a, theirs_b))
>>> p = scipy.stats.normaltest(theirs)[1]
>>> print('Probability their data is non-normal is: {:.4g}'.format(p))
Probability their data is non-normal is: 3.272e-11
```

Example

```
>>> pts = 1000
>>> rng = np.random.RandomState(28041990)
>>> ours = standard_normal32(pts, mean=10, std=3, rng=rng)
>>> assert np.abs(ours.std() - 3.0) < 0.1
>>> assert np.abs(ours.mean() - 10.0) < 0.1
```

Example

```
>>> # Test an even and odd numbers of points
>>> assert standard_normal32(3).shape == (3,)
>>> assert standard_normal32(2).shape == (2,)
>>> assert standard_normal32(1).shape == (1,)
>>> assert standard_normal32(0).shape == (0,)
>>> assert standard_normal32((3, 1)).shape == (3, 1)
>>> assert standard_normal32((3, 0)).shape == (3, 0)
```

`kwarray.fast_rand.standard_normal64(size, mean=0, std=1, rng=<module 'numpy.random' from
'/home/docs/checkouts/readthedocs.org/user_builds/kwarray/envs/latest/lib/python3.11/site-
packages/numpy/random/__init__.py'>)`

Simple wrapper around `rng.standard_normal` to make an API compatible with `standard_normal32()`.

Parameters

- **size** (*int* | *Tuple[int, ...]*) – shape of the returned ndarray
- **mean** (*float*) – mean of the normal distribution. defaults to 0
- **std** (*float*) – standard deviation of the normal distribution. defaults to 1.
- **rng** (*numpy.random.RandomState*) – underlying random state

Returns

normally distributed random numbers with chosen size.

Return type

ndarray[Any, Float64]

SeeAlso:

- `standard_normal`
- `standard_normal32`

Example

```
>>> pts = 1000
>>> rng = np.random.RandomState(28041994)
>>> out = standard_normal64(pts, mean=10, std=3, rng=rng)
>>> assert np.abs(out.std() - 3.0) < 0.1
>>> assert np.abs(out.mean() - 10.0) < 0.1
```

```
kwarray.fast_rand.uniform32(low=0.0, high=1.0, size=None, rng=<module 'numpy.random' from  
    '/home/docs/checkouts/readthedocs.org/user_builds/kwarray/envs/latest/lib/python3.11/site-  
    packages/numpy/random/__init__.py'>)
```

Draws float32 samples from a uniform distribution.

Samples are uniformly distributed over the half-open interval `[low, high)` (includes low, but excludes high).

Parameters

- **low** (*float, default=0.0*) – Lower boundary of the output interval. All values generated will be greater than or equal to low.
- **high** (*float, default=1.0*) – Upper boundary of the output interval. All values generated will be less than high.
- **size** (*int | Tuple[int, ...] | None*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if low and high are both scalars. Otherwise, `np.broadcast(low, high).size` samples are drawn.

Returns

uniformly distributed random numbers with chosen size.

Return type

`ndarray[Any, Float32]`

Example

```
>>> rng = np.random.RandomState(0)
>>> uniform32(low=0.0, high=1.0, size=None, rng=rng)
0.5488...
>>> uniform32(low=0.0, high=1.0, size=2000, rng=rng).sum()
1004.94...
>>> uniform32(low=-10, high=10.0, size=2000, rng=rng).sum()
202.44...
```

Benchmark

```
>>> from timerit import Timerit
>>> import kwarray
>>> size = 512 * 512
>>> for timer in Timerit(100, bestof=10, label='theirs: dtype=np.float64'):
>>>     rng = kwarray.ensure_rng(0)
>>>     with timer:
>>>         theirs = rng.uniform(size=size)
>>> for timer in Timerit(100, bestof=10, label='theirs: dtype=np.float32'):
>>>     rng = kwarray.ensure_rng(0)
>>>     with timer:
>>>         theirs = rng.rand(size).astype(np.float32)
>>> for timer in Timerit(100, bestof=10, label='ours: dtype=np.float32'):
>>>     rng = kwarray.ensure_rng(0)
>>>     with timer:
>>>         ours = uniform32(size=size)
```

kwarray.util_averages module

Currently just defines “stats_dict”, which is a nice way to gather multiple numeric statistics (e.g. max, min, median, mode, arithmetic-mean, geometric-mean, standard-deviation, etc...) about data in an array.

`kwarray.util_averages.stats_dict(inputs, axis=None, nan=False, sum=False, extreme=True, n_extreme=False, median=False, shape=True, size=False, quantile='auto')`

Describe statistics about an input array

Parameters

- **inputs** (*ArrayLike*) – set of values to get statistics of
- **axis** (*int*) – if **inputs** is ndarray then this specifies the axis
- **nan** (*bool*) – report number of nan items (TODO: rename to skipna)
- **sum** (*bool*) – report sum of values
- **extreme** (*bool*) – report min and max values
- **n_extreme** (*bool*) – report extreme value frequencies
- **median** (*bool*) – report median
- **size** (*bool*) – report array size
- **shape** (*bool*) – report array shape
- **quantile** (*str | bool | List[float]*) – defaults to ‘auto’. Can also be a list of quantiles to compute. if truthy computes quantiles.

Returns

dictionary of common numpy statistics (min, max, mean, std, nMin, nMax, shape)

Return type

`collections.OrderedDict`

SeeAlso:

`scipy.stats.describe()` `pandas.DataFrame.describe()`

Example

```
>>> # xdoctest: +IGNORE_WHITESPACE
>>> from kwarray.util_averages import * # NOQA
>>> axis = 0
>>> rng = np.random.RandomState(0)
>>> inputs = rng.rand(10, 2).astype(np.float32)
>>> stats = stats_dict(inputs, axis=axis, nan=False, median=True)
>>> import ubelt as ub # NOQA
>>> result = str(ub.urepr(stats, nl=1, precision=4, with_dtype=True))
>>> print(result)
{
  'mean': np.array([[0.5206, 0.6425]], dtype=np.float32),
  'std': np.array([[0.2854, 0.2517]], dtype=np.float32),
  'min': np.array([[0.0202, 0.0871]], dtype=np.float32),
  'max': np.array([[0.9637, 0.9256]], dtype=np.float32),
  'q_0.25': np.array([0.4271, 0.5329], dtype=np.float64),
```

(continues on next page)

(continued from previous page)

```

'q_0.50': np.array([0.5584, 0.6805], dtype=np.float64),
'q_0.75': np.array([0.7343, 0.8607], dtype=np.float64),
'med': np.array([0.5584, 0.6805], dtype=np.float32),
'shape': (10, 2),
}

```

Example

```

>>> # xdoctest: +IGNORE_WHITESPACE
>>> from kwarray.util_averages import * # NOQA
>>> axis = 0
>>> rng = np.random.RandomState(0)
>>> inputs = rng.randint(0, 42, size=100).astype(np.float32)
>>> inputs[4] = np.nan
>>> stats = stats_dict(inputs, axis=axis, nan=True, quantile='auto')
>>> import ubelt as ub # NOQA
>>> result = str(ub.urepr(stats, nl=0, precision=1, strkeys=True))
>>> print(result)

```

Example

```

>>> import kwarray
>>> import ubelt as ub
>>> rng = kwarray.ensure_rng(0)
>>> orig_inputs = rng.rand(1, 1, 2, 3)
>>> param_grid = ub.named_product({
>>>     #'axis': (None, 0, (0, 1), -1),
>>>     'axis': [None],
>>>     'percent_nan': [0, 0.5, 1.0],
>>>     'nan': [True, False],
>>>     'sum': [1],
>>>     'extreme': [True],
>>>     'n_extreme': [True],
>>>     'median': [1],
>>>     'size': [1],
>>>     'shape': [1],
>>>     'quantile': ['auto'],
>>> })
>>> for params in param_grid:
>>>     kwargs = params.copy()
>>>     percent_nan = kwargs.pop('percent_nan', 0)
>>>     if percent_nan:
>>>         inputs = orig_inputs.copy()
>>>         inputs[rng.rand(*inputs.shape) < percent_nan] = np.nan
>>>     else:
>>>         inputs = orig_inputs
>>>     stats = kwarray.stats_dict(inputs, **kwargs)
>>>     print('---')

```

(continues on next page)

(continued from previous page)

```
>>> print('params = {}'.format(ub.urepr(params, nl=1)))
>>> print('stats = {}'.format(ub.urepr(stats, nl=1)))
```

`kwarray.util_averages._gmean(a, axis=0, dtype=None, clobber=False)`

Compute the geometric mean along the specified axis.

Modification of the scikit-learn method to be more memory efficient

Example

```
>>> rng = np.random.RandomState(0)
>>> C, H, W = 8, 32, 32
>>> axis = 0
>>> a = [rng.rand(C, H, W).astype(np.float16),
>>>       rng.rand(C, H, W).astype(np.float16)]
```

exception `kwarray.util_averages.NoSupportError`

Bases: `RuntimeError`

class `kwarray.util_averages.RunningStats(nan_policy='omit', check_weights=True, **kwargs)`

Bases: `NiceRepr`

Track mean, std, min, and max values over time with constant memory.

Dynamically records per-element array statistics and can summarize them per-element, across channels, or globally.

Todo:

- [] This may need a few API tweaks and good documentation
-

Example

```
>>> import kwarray
>>> run = kwarray.RunningStats()
>>> ch1 = np.array([[0, 1], [3, 4]])
>>> ch2 = np.zeros((2, 2))
>>> img = np.dstack([ch1, ch2])
>>> run.update(np.dstack([ch1, ch2]))
>>> run.update(np.dstack([ch1 + 1, ch2]))
>>> run.update(np.dstack([ch1 + 2, ch2]))
>>> # No marginalization
>>> print('current-ave = ' + ub.urepr(run.summarize(axis=ub.NoParam), nl=2,
↳ precision=3))
>>> # Average over channels (keeps spatial dims separate)
>>> print('chann-ave(k=1) = ' + ub.urepr(run.summarize(axis=0), nl=2, precision=3))
>>> print('chann-ave(k=0) = ' + ub.urepr(run.summarize(axis=0, keepdims=0), nl=2,
↳ precision=3))
>>> # Average over spatial dims (keeps channels separate)
>>> print('spatial-ave(k=1) = ' + ub.urepr(run.summarize(axis=(1, 2)), nl=2,
↳ precision=3))
>>> print('spatial-ave(k=0) = ' + ub.urepr(run.summarize(axis=(1, 2), keepdims=0),
↳
```

(continues on next page)

(continued from previous page)

```

    nl=2, precision=3))
>>> # Average over all dims
>>> print('alldim-ave(k=1) = ' + ub.urepr(run.summarize(axis=None), nl=2,
    nl=2, precision=3))
>>> print('alldim-ave(k=0) = ' + ub.urepr(run.summarize(axis=None, keepdims=0),
    nl=2, precision=3))

```

Parameters

nan_policy (*str*) – indicates how we will handle nan values

- if “omit” - set weights of nan items to zero.
- if “propagate” - propagate nans.
- if “raise” - then raise a ValueError if nans are given.

check_weights (*bool*):

if True, we check the weights for zeros (which can also implicitly occur when data has nans). Disabling this check will result in faster computation, but it is your responsibility to ensure all data passed to update is valid.

property shape

_update_from_other (*other*)

Combine this runner with another one.

update_many (*data*, *weights=1*)

Assumes first data axis represents multiple observations

Example

```

>>> import kwarray
>>> rng = kwarray.ensure_rng(0)
>>> run = kwarray.RunningStats()
>>> data = rng.randn(1, 2, 3)
>>> run.update_many(data)
>>> print(run.current())
>>> data = rng.randn(2, 2, 3)
>>> run.update_many(data)
>>> print(run.current())
>>> data = rng.randn(3, 2, 3)
>>> run.update_many(data)
>>> print(run.current())
>>> run.update_many(1000)
>>> print(run.current())
>>> assert np.all(run.current()['n'] == 7)

```


Example

```

>>> import kwarray
>>> rng = kwarray.ensure_rng(0)
>>> run = kwarray.RunningStats()
>>> data = rng.randn(1, 2, 3)
>>> run.update_many(data.ravel())
>>> print(run.current())
>>> data = rng.randn(2, 2, 3)
>>> run.update_many(data.ravel())
>>> print(run.current())
>>> data = rng.randn(3, 2, 3)
>>> run.update_many(data.ravel())
>>> print(run.current())
>>> run.update_many(1000)
>>> print(run.current())
>>> assert np.all(run.current()['n'] == 37)

```

update(data, weights=1)

Updates statistics across all data dimensions on a per-element basis

Example

```

>>> import kwarray
>>> data = np.full((7, 5), fill_value=1.3)
>>> weights = np.ones((7, 5), dtype=np.float32)
>>> run = kwarray.RunningStats()
>>> run.update(data, weights=1)
>>> run.update(data, weights=weights)
>>> rng = np.random
>>> weights[rng.rand(*weights.shape) > 0.5] = 0
>>> run.update(data, weights=weights)

```

Example

```

>>> import kwarray
>>> run = kwarray.RunningStats()
>>> data = np.array([[1, np.nan, np.nan], [0, np.nan, 1.]])
>>> run.update(data)
>>> print('current = {}'.format(ub.urepr(run.current(), nl=1)))
>>> print('summary(axis=None) = {}'.format(ub.urepr(run.summarize(), nl=1)))
>>> print('summary(axis=1) = {}'.format(ub.urepr(run.summarize(axis=1), nl=1)))
>>> print('summary(axis=0) = {}'.format(ub.urepr(run.summarize(axis=0), nl=1)))
>>> data = np.array([[2, 0, 1], [0, 1, np.nan]])
>>> run.update(data)
>>> data = np.array([[3, 1, 1], [0, 1, np.nan]])
>>> run.update(data)
>>> data = np.array([[4, 1, 1], [0, 1, 1.]])
>>> run.update(data)
>>> print('----')

```

(continues on next page)

(continued from previous page)

```
>>> print('current = {}'.format(ub.urepr(run.current(), nl=1)))
>>> print('summary(axis=None) = {}'.format(ub.urepr(run.summarize(), nl=1)))
>>> print('summary(axis=1) = {}'.format(ub.urepr(run.summarize(axis=1), nl=1)))
>>> print('summary(axis=0) = {}'.format(ub.urepr(run.summarize(axis=0), nl=1)))
```

_sumsq_std(total, squares, n)

Sum of squares method to compute standard deviation

summarize(axis=None, keepdims=True)

Compute summary statistics across a one or more dimension

Parameters

- **axis** (*int* | *List[int]* | *None* | *NoParamType*) – axis or axes to summarize over, if *None*, all axes are summarized. if *ub.NoParam*, no axes are summarized the current result is returned.
- **keepdims** (*bool*, *default=True*) – if *False* removes the dimensions that are summarized over

Returns

containing minimum, maximum, mean, std, etc..

Return type

Dict

Raises

NoSupportError – if update was never called with valid data

Example

```
>>> # Test to make sure summarize works across different shapes
>>> base = np.array([1, 1, 1, 1, 0, 0, 0, 1])
>>> run0 = RunningStats()
>>> for _ in range(3):
>>>     run0.update(base.reshape(8, 1))
>>> run1 = RunningStats()
>>> for _ in range(3):
>>>     run1.update(base.reshape(4, 2))
>>> run2 = RunningStats()
>>> for _ in range(3):
>>>     run2.update(base.reshape(2, 2, 2))
>>> #
>>> # Summarizing over everything should be exactly the same
>>> s0N = run0.summarize(axis=None, keepdims=0)
>>> s1N = run1.summarize(axis=None, keepdims=0)
>>> s2N = run2.summarize(axis=None, keepdims=0)
>>> #assert ub.util_indexable.indexable_allclose(s0N, s1N, rel_tol=0.0, abs_
↳ tol=0.0)
>>> #assert ub.util_indexable.indexable_allclose(s1N, s2N, rel_tol=0.0, abs_
↳ tol=0.0)
>>> assert s0N['mean'] == 0.625
```

current()

Returns current statics on a per-element basis (not summarized over any axis)

`kwarray.util_averages._combine_mean_stds(means, stds, nums=None, axis=None, keepdims=False, bessel=True)`

Parameters

- **means** (*array*) – means[i] is the mean of the ith entry to combine
- **stds** (*array*) – stds[i] is the std of the ith entry to combine
- **nums** (*array* | *None*) – nums[i] is the number of samples in the ith entry to combine. if *None*, assumes sample sizes are infinite.
- **axis** (*int* | *Tuple[int]* | *None*) – axis to combine the statistics over
- **keepdims** (*bool*) – if *True* return arrays with the same number of dimensions they were given in.
- **bessel** (*int*) – Set to 1 to enables *bessel* correction to unbiased the combined std estimate. Only disable if you have the true population means, or you think you know what you are doing.

References

<https://stats.stackexchange.com/questions/55999/is-it-possible-to-find-the-combined-standard-deviation>

Sympy

```
>>> # xdoctest: +REQUIRES(env:SHOW_SYMPY)
>>> # What about the case where we don't know population size of the
>>> # estimates. We could treat it as a fixed number, or perhaps take the
>>> # limit as n -> infinity.
>>> import sympy
>>> import sympy as sym
>>> from sympy import symbols, sqrt, limit, IndexedBase, summation
>>> from sympy import Indexed, Idx, symbols
>>> means = IndexedBase('m')
>>> stds = IndexedBase('s')
>>> nums = IndexedBase('n')
>>> i = symbols('i', cls=Idx)
>>> k = symbols('k', cls=Idx)
>>> #
>>> combo_mean = symbols('C')
>>> #
>>> bessel = 1
>>> total = summation(nums[i], (i, 1, k))
>>> combo_mean_expr = summation(nums[i] * means[i], (i, 1, k)) / total
>>> p1 = summation((nums[i] - bessel) * stds[i], (i, 1, k))
>>> p2 = summation(nums[i] * ((means[i] - combo_mean) ** 2), (i, 1, k))
>>> #
>>> combo_std_expr = sqrt((p1 + p2) / (total - bessel))
>>> print('-----')
>>> print('General Combined Mean / Std Formulas')
>>> print('C = combined mean')
>>> print('S = combined std')
>>> print('-----')
>>> print(ub.hzcat(['C = ', sym.pretty(combo_mean_expr, use_unicode=True, use_
```

(continues on next page)

(continued from previous page)

```

↪unicode_sqrt_char=True)))))
>>> print(ub.hzcat(['S = ', sym.pretty(combo_std_expr, use_unicode=True, use_
↪unicode_sqrt_char=True)))))
>>> print('')
>>> print('-----')
>>> print('Now assuming all sample sizes are the same constant value N')
>>> print('-----')
>>> # Now assume all n[i] = N (i.e. a constant value)
>>> N = symbols('N')
>>> combo_mean_const_n_expr = combo_mean_expr.copy().xreplace({nums[i]: N})
>>> combo_std_const_n_expr = combo_std_expr.copy().xreplace({nums[i]: N})
>>> p1_const_n = p1.copy().xreplace({nums[i]: N})
>>> p2_const_n = p2.copy().xreplace({nums[i]: N})
>>> total_const_n = total.copy().xreplace({nums[i]: N})
>>> #
>>> print(ub.hzcat(['C = ', sym.pretty(combo_mean_const_n_expr, use_unicode=True,
↪use_unicode_sqrt_char=True)))))
>>> print(ub.hzcat(['S = ', sym.pretty(combo_std_const_n_expr, use_unicode=True,
↪use_unicode_sqrt_char=True)))))
>>> #
>>> print('')
>>> print('-----')
>>> print('Take the limit as N -> infinity')
>>> print('-----')
>>> #
>>> # Limit doesnt directly but we can break it into parts
>>> lim_C = limit(combo_mean_const_n_expr, N, float('inf'))
>>> lim_p1 = limit(p1_const_n / (total_const_n - bessell), N, float('inf'))
>>> lim_p2 = limit(p2_const_n / (total_const_n - bessell), N, float('inf'))
>>> lim_expr = sym.sqrt(lim_p1 + lim_p2)
>>> print(ub.hzcat(['lim(C, N->inf) = ', sym.pretty(lim_C)]))
>>> print(ub.hzcat(['lim(S, N->inf) = ', sym.pretty(lim_expr)]))

```

Example

```

>>> from kwarray.util_averages import * # NOQA
>>> from kwarray.util_averages import _combine_mean_stds
>>> means = np.array([1.2, 3.2, 4.1])
>>> stds = np.array([4.2, 0.2, 2.1])
>>> nums = np.array([10, 100, 10])
>>> _combine_mean_stds(means, stds, nums)
>>> means = np.array([1, 2, 3])
>>> stds = np.array([1, 2, 3])
>>> #
>>> nums = np.array([1, 1, 1]) / 3
>>> print(_combine_mean_stds(means, stds, nums, bessell=True), '- .3 B')
>>> print(_combine_mean_stds(means, stds, nums, bessell=False), '- .3')
>>> nums = np.array([1, 1, 1])
>>> print(_combine_mean_stds(means, stds, nums, bessell=True), '- 1 B')
>>> print(_combine_mean_stds(means, stds, nums, bessell=False), '- 1')

```

(continues on next page)

(continued from previous page)

```

>>> nums = np.array([10, 10, 10])
>>> print(_combine_mean_stds(means, stds, nums, bessel=True), '- 10 B')
>>> print(_combine_mean_stds(means, stds, nums, bessel=False), '- 10')
>>> nums = np.array([1000, 1000, 1000])
>>> print(_combine_mean_stds(means, stds, nums, bessel=True), '- 1000 B')
>>> print(_combine_mean_stds(means, stds, nums, bessel=False), '- 1000')
>>> #
>>> nums = None
>>> print(_combine_mean_stds(means, stds, nums, bessel=True), '- inf B')
>>> print(_combine_mean_stds(means, stds, nums, bessel=False), '- inf')

```

Example

```

>>> from kwarray.util_averages import * # NOQA
>>> from kwarray.util_averages import _combine_mean_stds
>>> means = np.stack([np.array([1.2, 3.2, 4.1])] * 100, axis=0)
>>> stds = np.stack([np.array([4.2, 0.2, 2.1])] * 100, axis=0)
>>> nums = np.stack([np.array([10, 100, 10])] * 100, axis=0)
>>> cm1, cs1, _ = _combine_mean_stds(means, stds, nums, axis=None)
>>> print('combo_mean = {}'.format(ub.urepr(cm1, nl=1)))
>>> print('combo_std = {}'.format(ub.urepr(cs1, nl=1)))
>>> means = np.stack([np.array([1.2, 3.2, 4.1])] * 1, axis=0)
>>> stds = np.stack([np.array([4.2, 0.2, 2.1])] * 1, axis=0)
>>> nums = np.stack([np.array([10, 100, 10])] * 1, axis=0)
>>> cm2, cs2, _ = _combine_mean_stds(means, stds, nums, axis=None)
>>> print('combo_mean = {}'.format(ub.urepr(cm2, nl=1)))
>>> print('combo_std = {}'.format(ub.urepr(cs2, nl=1)))
>>> means = np.stack([np.array([1.2, 3.2, 4.1])] * 5, axis=0)
>>> stds = np.stack([np.array([4.2, 0.2, 2.1])] * 5, axis=0)
>>> nums = np.stack([np.array([10, 100, 10])] * 5, axis=0)
>>> cm3, cs3, combo_num = _combine_mean_stds(means, stds, nums, axis=1)
>>> print('combo_mean = {}'.format(ub.urepr(cm3, nl=1)))
>>> print('combo_std = {}'.format(ub.urepr(cs3, nl=1)))
>>> assert np.allclose(cm1, cm2) and np.allclose(cm2, cm3)
>>> assert not np.allclose(cs1, cs2)
>>> assert np.allclose(cs2, cs3)

```

Example

```

>>> from kwarray.util_averages import * # NOQA
>>> from kwarray.util_averages import _combine_mean_stds
>>> means = np.random.rand(2, 3, 5, 7)
>>> stds = np.random.rand(2, 3, 5, 7)
>>> nums = (np.random.rand(2, 3, 5, 7) * 10) + 1
>>> cm, cs, cn = _combine_mean_stds(means, stds, nums, axis=1, keepdims=1)
>>> assert cm.shape == cs.shape == cn.shape
>>> print(f'cm.shape={cm.shape}')
>>> cm, cs, cn = _combine_mean_stds(means, stds, nums, axis=(0, 2), keepdims=1)
>>> assert cm.shape == cs.shape == cn.shape

```

(continues on next page)

(continued from previous page)

```

>>> print(f'cm.shape={cm.shape}')
>>> cm, cs, cn = _combine_mean_stds(means, stds, nums, axis=(1, 3), keepdims=1)
>>> assert cm.shape == cs.shape == cn.shape
>>> print(f'cm.shape={cm.shape}')
>>> cm, cs, cn = _combine_mean_stds(means, stds, nums, axis=None)
>>> assert cm.shape == cs.shape == cn.shape
>>> print(f'cm.shape={cm.shape}')
cm.shape=(2, 1, 5, 7)
cm.shape=(1, 3, 1, 7)
cm.shape=(2, 1, 5, 1)
cm.shape=()

```

`kwarray.util_averages._no_keepdim_indexer(result, axis)`

Computes an indexer to postprocess a result with `keepdims=True` that will modify the result as if `keepdims=False`

`kwarray.util_averages._postprocess_keepdims(original, result, axis)`

Can update the result of a function that does not support `keepdims` to look as if `keepdims` was supported.

kwarray.util_groups module

Functions for partitioning numpy arrays into groups.

`kwarray.util_groups.group_items(item_list, groupid_list, assume_sorted=False, axis=None)`

Groups a list of items by group id.

Works like `ubelt.group_items()`, but with numpy optimizations. This can be quite a bit faster than using `itertools.groupby()`¹².

In cases where there are many lists of items to group (think column-major data), consider using `group_indices()` and `apply_grouping()` instead.

Parameters

- **item_list** (*NDArray*) – The input array of items to group. Extended typing `NDArray[Any, VT]`
- **groupid_list** (*NDArray*) – Each item is an id corresponding to the item at the same position in `item_list`. For the fastest runtime, the input array must be numeric (ideally with integer types). This list must be 1-dimensional. Extended typing `NDArray[Any, KT]`
- **assume_sorted** (*bool*) – If the input array is sorted, then setting this to `True` will avoid an unnecessary sorting operation and improve efficiency. Defaults to `False`.
- **axis** (*int | None*) – Group along a particular axis in `items` if it is n-dimensional.

Returns

mapping from groupids to corresponding items. Extended typing `Dict[KT, NDArray[Any, VT]]`.

Return type

`Dict[Any, NDArray]`

¹ <http://stackoverflow.com/questions/4651683/>

² `numpy-grouping-using-itertools-groupby-performance`

References

Example

```
>>> from kwarray.util_groups import * # NOQA
>>> items = np.array([0, 1, 2, 3, 4, 5, 6, 7, 1, 1])
>>> keys = np.array([2, 2, 1, 1, 0, 1, 0, 1, 1, 1])
>>> grouped = group_items(items, keys)
>>> print('grouped = ' + ub.urepr(grouped, nl=1, with_dtype=False, sort=1))
grouped = {
    0: np.array([4, 6]),
    1: np.array([2, 3, 5, 7, 1, 1]),
    2: np.array([0, 1]),
}
```

`kwarray.util_groups.group_indices(idx_to_groupid, assume_sorted=False)`

Find unique items and the indices at which they appear in an array.

A common use case of this function is when you have a list of objects (often numeric but sometimes not) and an array of “group-ids” corresponding to that list of objects.

Using this function will return a list of indices that can be used in conjunction with [apply_grouping\(\)](#) to group the elements. This is most useful when you have many lists (think column-major data) corresponding to the group-ids.

In cases where there is only one list of objects or knowing the indices doesn’t matter, then consider using `func:group_items` instead.

Parameters

- **idx_to_groupid** (*NDArray*) – The input array, where each item is interpreted as a group id. For the fastest runtime, the input array must be numeric (ideally with integer types). If the type is non-numeric then the less efficient `ubelt.group_items()` is used.
- **assume_sorted** (*bool*) – If the input array is sorted, then setting this to True will avoid an unnecessary sorting operation and improve efficiency. Defaults to False.

Returns

(**keys**, **groupxs**) -

keys (*NDArray*):

The unique elements of the input array in order

groupxs (*List[NDArray]*):

Corresponding list of indexes. The i-th item is an array indicating the indices where the item `key[i]` appeared in the input array.

Return type

Tuple[*NDArray*, List[*NDArray*]]

Example

```

>>> # xdoctest: +IGNORE_WHITESPACE
>>> import kwarray
>>> import ubelt as ub
>>> idx_to_groupid = np.array([2, 1, 2, 1, 2, 1, 2, 3, 3, 3, 3])
>>> (keys, groupxs) = kwarray.group_indices(idx_to_groupid)
>>> print('keys = ' + ub.urepr(keys, with_dtype=False))
>>> print('groupxs = ' + ub.urepr(groupxs, with_dtype=False))
keys = np.array([1, 2, 3])
groupxs = [
    np.array([1, 3, 5]),
    np.array([0, 2, 4, 6]),
    np.array([ 7,  8,  9, 10]),
]

```

Example

```

>>> # xdoctest: +IGNORE_WHITESPACE
>>> import kwarray
>>> import ubelt as ub
>>> # 2d arrays must be flattened before coming into this function so
>>> # information is on the last axis
>>> idx_to_groupid = np.array([[ 24], [ 129], [ 659], [ 659], [ 24],
...     [659], [ 659], [ 822], [ 659], [ 659], [24]]).T[0]
>>> (keys, groupxs) = kwarray.group_indices(idx_to_groupid)
>>> # Different versions of numpy may produce different orderings
>>> # so normalize these to make test output consistent
>>> # [gxs.sort() for gxs in groupxs]
>>> print('keys = ' + ub.urepr(keys, with_dtype=False))
>>> print('groupxs = ' + ub.urepr(groupxs, with_dtype=False))
keys = np.array([ 24, 129, 659, 822])
groupxs = [
    np.array([ 0,  4, 10]),
    np.array([1]),
    np.array([2, 3, 5, 6, 8, 9]),
    np.array([7]),
]

```

Example

```

>>> # xdoctest: +IGNORE_WHITESPACE
>>> import kwarray
>>> import ubelt as ub
>>> idx_to_groupid = np.array([True, True, False, True, False, False, True])
>>> (keys, groupxs) = kwarray.group_indices(idx_to_groupid)
>>> print(ub.urepr(keys, with_dtype=False))
>>> print(ub.urepr(groupxs, with_dtype=False))
np.array([False,  True])
[

```

(continues on next page)

(continued from previous page)

```
np.array([2, 4, 5]),
np.array([0, 1, 3, 6]),
]
```

Example

```
>>> # xdoctest: +IGNORE_WHITESPACE
>>> import ubelt as ub
>>> import kvarray
>>> idx_to_groupid = [('a', 'b'), ('d', 'b'), ('a', 'b'), ('a', 'b')]
>>> (keys, groupxs) = kvarray.group_indices(idx_to_groupid)
>>> print(ub.urepr(keys, with_dtype=False))
>>> print(ub.urepr(groupxs, with_dtype=False))
[
  ('a', 'b'),
  ('d', 'b'),
]
[
  np.array([0, 2, 3]),
  np.array([1]),
]
```

`kvarray.util_groups.apply_grouping(items, groupxs, axis=0)`

Applies grouping from `group_indices`.

Typically used in conjunction with `group_indices()`.

Parameters

- **items** (*NDAarray*) – items to group
- **groupxs** (*List[NDAarray[None, Int]]*) – groups of indices
- **axis** (*None|int, default=0*)

Returns

grouped items

Return type

List[NDAarray]

Example

```
>>> # xdoctest: +IGNORE_WHITESPACE
>>> idx_to_groupid = np.array([2, 1, 2, 1, 2, 1, 2, 3, 3, 3, 3])
>>> items          = np.array([1, 8, 5, 5, 8, 6, 7, 5, 3, 0, 9])
>>> (keys, groupxs) = group_indices(idx_to_groupid)
>>> grouped_items = apply_grouping(items, groupxs)
>>> result = str(grouped_items)
>>> print(result)
[array([8, 5, 6]), array([1, 5, 8, 7]), array([5, 3, 0, 9])]
```

`kwarray.util_groups.group_consecutive(arr, offset=1)`

Returns lists of consecutive values. Implementation inspired by³.

Parameters

- **arr** (*NDArray*) – array of ordered values
- **offset** (*float, default=1*) – any two values separated by this offset are grouped. In the default case, when `offset=1`, this groups increasing values like: 0, 1, 2. When `offset` is 0 it groups consecutive values that are the same, e.g.: 4, 4, 4.

Returns

a list of arrays that are the groups from the input

Return type

List[NDArray]

Note: This is equivalent (and faster) to using: `apply_grouping(data, group_consecutive_indices(data))`

References

Example

```
>>> arr = np.array([1, 2, 3, 5, 6, 7, 8, 9, 10, 15, 99, 100, 101])
>>> groups = group_consecutive(arr)
>>> print('groups = {}'.format(list(map(list, groups))))
groups = [[1, 2, 3], [5, 6, 7, 8, 9, 10], [15], [99, 100, 101]]
>>> arr = np.array([0, 0, 3, 0, 0, 7, 2, 3, 4, 4, 4, 1, 1])
>>> groups = group_consecutive(arr, offset=1)
>>> print('groups = {}'.format(list(map(list, groups))))
groups = [[0], [0], [3], [0], [0], [7], [2, 3, 4], [4], [4], [1], [1]]
>>> groups = group_consecutive(arr, offset=0)
>>> print('groups = {}'.format(list(map(list, groups))))
groups = [[0, 0], [3], [0, 0], [7], [2], [3], [4, 4, 4], [1, 1]]
```

`kwarray.util_groups.group_consecutive_indices(arr, offset=1)`

Returns lists of indices pointing to consecutive values

Parameters

- **arr** (*NDArray*) – array of ordered values
- **offset** (*float, default=1*) – any two values separated by this offset are grouped.

Returns

groupxs: a list of indices

Return type

List[NDArray]

SeeAlso:

[`group\_consecutive\(\)`](#)

[`apply\_grouping\(\)`](#)

³ <http://stackoverflow.com/questions/7352684/groups-consecutive-elements>

Example

```

>>> arr = np.array([1, 2, 3, 5, 6, 7, 8, 9, 10, 15, 99, 100, 101])
>>> groupxs = group_consecutive_indices(arr)
>>> print('groupxs = {}'.format(list(map(list, groupxs))))
groupxs = [[0, 1, 2], [3, 4, 5, 6, 7, 8], [9], [10, 11, 12]]
>>> assert all(np.array_equal(a, b) for a, b in zip(group_consecutive(arr, 1),
↳ apply_grouping(arr, groupxs)))
>>> arr = np.array([0, 0, 3, 0, 0, 7, 2, 3, 4, 4, 4, 1, 1])
>>> groupxs = group_consecutive_indices(arr, offset=1)
>>> print('groupxs = {}'.format(list(map(list, groupxs))))
groupxs = [[0], [1], [2], [3], [4], [5], [6, 7, 8], [9], [10], [11], [12]]
>>> assert all(np.array_equal(a, b) for a, b in zip(group_consecutive(arr, 1),
↳ apply_grouping(arr, groupxs)))
>>> groupxs = group_consecutive_indices(arr, offset=0)
>>> print('groupxs = {}'.format(list(map(list, groupxs))))
groupxs = [[0, 1], [2], [3, 4], [5], [6], [7], [8, 9, 10], [11, 12]]
>>> assert all(np.array_equal(a, b) for a, b in zip(group_consecutive(arr, 0),
↳ apply_grouping(arr, groupxs)))

```

kwarray.util_misc module

Misc tools that should find a better home

class kwarray.util_misc.FlatIndexer(*lens*)

Bases: [NiceRepr](#)

Creates a flat “view” of a jagged nested indexable object. Only supports one offset level.

Parameters

lens (*List[int]*) – a list of the lengths of the nested objects.

Doctest

```

>>> self = FlatIndexer([1, 2, 3])
>>> len(self)
>>> self.unravel(4)
>>> self.ravel(2, 1)

```

classmethod [fromlist](#)(*items*)

Convenience method to create a [FlatIndexer](#) from the list of items itself instead of the array of lengths.

Parameters

items (*List[list]*) – a list of the lists you want to flat index over

Returns

FlatIndexer

unravel(*index*)

Parameters

index (*int* | *List[int]*) – raveled index

Returns

outer and inner indices

Return type

Tuple[int, int]

Example

```
>>> import kwarray
>>> rng = kwarray.ensure_rng(0)
>>> items = [rng.rand(rng.randint(0, 10)) for _ in range(10)]
>>> self = kwarray.FlatIndexer.fromlist(items)
>>> index = np.arange(0, len(self))
>>> outer, inner = self.unravel(index)
>>> recon = self.ravel(outer, inner)
>>> # This check is only possible because index is an arange
>>> check1 = np.hstack(list(map(sorted, kwarray.group_indices(outer)[1])))
>>> check2 = np.hstack(kwarray.group_consecutive_indices(inner))
>>> assert np.all(check1 == index)
>>> assert np.all(check2 == index)
>>> assert np.all(index == recon)
```

ravel(outer, inner)**Parameters**

- **outer** – index into outer list
- **inner** – index into the list referenced by outer

Returns

the raveled index

Return type

List[int]

kwarray.util_numpy module

Numpy specific extensions

kwarray.util_numpy.boolmask(indices, shape=None)

Constructs an array of booleans where an item is True if its position is in **indices** otherwise it is False. This can be viewed as the inverse of `numpy.where()`.

Parameters

- **indices** (*NDArray*) – list of integer indices
- **shape** (*int* | *tuple*) – length of the returned list. If not specified the minimal possible shape to incorporate all the indices is used. In general, it is best practice to always specify this argument.

Returns

mask - mask[idx] is True if idx in indices

Return type

NDArray[Any, Int]

Example

```
>>> indices = [0, 1, 4]
>>> mask = boolmask(indices, shape=6)
>>> assert np.all(mask == [True, True, False, False, True, False])
>>> mask = boolmask(indices)
>>> assert np.all(mask == [True, True, False, False, True])
```

Example

```
>>> import kwarray
>>> import ubelt as ub # NOQA
>>> indices = np.array([(0, 0), (1, 1), (2, 1)])
>>> shape = (3, 3)
>>> mask = kwarray.boolmask(indices, shape)
>>> result = ub.urepr(mask, with_dtype=0)
>>> print(result)
np.array([[ True, False, False],
          [False,  True, False],
          [False,  True, False]])
```

`kwarray.util_numpy.iter_reduce_ufunc(ufunc, arrs, out=None, default=None)`

constant memory iteration and reduction

Applies ufunc from left to right over the input arrays

Parameters

- **ufunc** (*Callable*) – called on each pair of consecutive ndarrays
- **arrs** (*Iterator[NDArray]*) – iterator of ndarrays
- **default** (*object*) – return value when iterator is empty

Returns

if `len(arrs) == 0`, returns `default` if `len(arrs) == 1`, returns `arrs[0]`, if `len(arrs) >= 2`, returns `ufunc(...ufunc(ufunc(arrs[0], arrs[1]), arrs[2]),...arrs[n-1])`

Return type

NDArray

Example

```
>>> arr_list = [
...     np.array([0, 1, 2, 3, 8, 9]),
...     np.array([4, 1, 2, 3, 4, 5]),
...     np.array([0, 5, 2, 3, 4, 5]),
...     np.array([1, 1, 6, 3, 4, 5]),
...     np.array([0, 1, 2, 7, 4, 5])
... ]
>>> memory = np.array([9, 9, 9, 9, 9, 9])
>>> gen_memory = memory.copy()
>>> def arr_gen(arr_list, gen_memory):
...     for arr in arr_list:
```

(continues on next page)

(continued from previous page)

```

...     gen_memory[:] = arr
...     yield gen_memory
>>> print('memory = %r' % (memory,))
>>> print('gen_memory = %r' % (gen_memory,))
>>> ufunc = np.maximum
>>> res1 = iter_reduce_ufunc(ufunc, iter(arr_list), out=None)
>>> res2 = iter_reduce_ufunc(ufunc, iter(arr_list), out=memory)
>>> res3 = iter_reduce_ufunc(ufunc, arr_gen(arr_list, gen_memory), out=memory)
>>> print('res1      = %r' % (res1,))
>>> print('res2      = %r' % (res2,))
>>> print('res3      = %r' % (res3,))
>>> print('memory     = %r' % (memory,))
>>> print('gen_memory = %r' % (gen_memory,))
>>> assert np.all(res1 == res2)
>>> assert np.all(res2 == res3)

```

kwarray.util_numpy.**isect_flags**(arr, other)

Check which items in an array intersect with another set of items

Parameters

- **arr** (*NDArray*) – items to check
- **other** (*Iterable*) – items to check if they exist in arr

Returns

booleans corresponding to arr indicating if any item in other
is also contained in other.

Return type

NDArray

Example

```

>>> arr = np.array([
>>>     [1, 2, 3, 4],
>>>     [5, 6, 3, 4],
>>>     [1, 1, 3, 4],
>>> ])
>>> other = np.array([1, 4, 6])
>>> mask = isect_flags(arr, other)
>>> print(mask)
[[ True False False  True]
 [False  True False  True]
 [ True  True False  True]]

```

kwarray.util_numpy.**atleast_nd**(arr, n, front=False)

View inputs as arrays with at least n dimensions.

Parameters

- **arr** (*ArrayLike*) – An array-like object. Non-array inputs are converted to arrays. Arrays that already have n or more dimensions are preserved.
- **n** (*int*) – number of dimensions to ensure

- **front** (*bool*) – if True new dimensions are added to the front of the array. otherwise they are added to the back. Defaults to False.

Returns

An array with `a.ndim >= n`. Copies are avoided where possible, and views with three or more dimensions are returned. For example, a 1-D array of shape `(N,)` becomes a view of shape `(1, N, 1)`, and a 2-D array of shape `(M, N)` becomes a view of shape `(M, N, 1)`.

Return type

NDArray

See also:

`numpy.atleast_1d`, `numpy.atleast_2d`, `numpy.atleast_3d`

Example

```
>>> n = 2
>>> arr = np.array([1, 1, 1])
>>> arr_ = atleast_nd(arr, n)
>>> import ubelt as ub # NOQA
>>> result = ub.urepr(arr_.tolist(), nl=0)
>>> print(result)
[[1], [1], [1]]
```

Example

```
>>> n = 4
>>> arr1 = [1, 1, 1]
>>> arr2 = np.array(0)
>>> arr3 = np.array([[[[1]]]])
>>> arr1_ = atleast_nd(arr1, n)
>>> arr2_ = atleast_nd(arr2, n)
>>> arr3_ = atleast_nd(arr3, n)
>>> import ubelt as ub # NOQA
>>> result1 = ub.urepr(arr1_.tolist(), nl=0)
>>> result2 = ub.urepr(arr2_.tolist(), nl=0)
>>> result3 = ub.urepr(arr3_.tolist(), nl=0)
>>> result = '\n'.join([result1, result2, result3])
>>> print(result)
[[[1]], [[1]], [[1]]]
[[[0]]]
[[[[1]]]]
```

Note: Extensive benchmarks are in `kwarray/dev/bench_atleast_nd.py`

These demonstrate that this function is statistically faster than the numpy variants, although the difference is small. On average this function takes 480ns versus numpy which takes 790ns.

`kwarray.util_numpy.argmaxima(arr, num, axis=None, ordered=True)`

Returns the top num maximum indicies.

This can be significantly faster than using `argsort`.

Parameters

- **arr** (*NDArray*) – input array
- **num** (*int*) – number of maximum indices to return
- **axis** (*int* | *None*) – axis to find maxima over. If *None* this is equivalent to using `arr.ravel()`.
- **ordered** (*bool*) – if *False*, returns the maximum elements in an arbitrary order, otherwise they are in decending order. (Setting this to *false* is a bit faster).

Todo:

- `[]` if num is *None*, return arg for all values equal to the maximum
-

Returns

NDArray

Example

```
>>> # Test cases with axis=None
>>> arr = (np.random.rand(100) * 100).astype(int)
>>> for num in range(0, len(arr) + 1):
>>>     idxs = argmaxima(arr, num)
>>>     idxs2 = argmaxima(arr, num, ordered=False)
>>>     assert np.all(arr[idxs] == np.array(sorted(arr)[::-1][:len(idxs)])),
↳ 'ordered=True must return in order'
>>>     assert sorted(idxs2) == sorted(idxs), 'ordered=False must return the right_
↳ idxs, but in any order'
```

Example

```
>>> # Test cases with axis
>>> arr = (np.random.rand(3, 5, 7) * 100).astype(int)
>>> for axis in range(len(arr.shape)):
>>>     for num in range(0, len(arr) + 1):
>>>         idxs = argmaxima(arr, num, axis=axis)
>>>         idxs2 = argmaxima(arr, num, ordered=False, axis=axis)
>>>         assert idxs.shape[axis] == num
>>>         assert idxs2.shape[axis] == num
```

`kwarray.util_numpy.argmaxinima(arr, num, axis=None, ordered=True)`

Returns the top num minimum indicies.

This can be significantly faster than using `argsort`.

Parameters

- **arr** (*NDArray*) – input array
- **num** (*int*) – number of minimum indices to return
- **axis** (*int*|*None*) – axis to find minima over. If *None* this is equivalent to using `arr.ravel()`.

- **ordered** (*bool*) – if False, returns the minimum elements in an arbitrary order, otherwise they are in ascending order. (Setting this to false is a bit faster).

Example

```
>>> arr = (np.random.rand(100) * 100).astype(int)
>>> for num in range(0, len(arr) + 1):
>>>     idxs = argminima(arr, num)
>>>     assert np.all(arr[idxs] == np.array(sorted(arr)[:len(idxs)])),
↳ 'ordered=True must return in order'
>>>     idxs2 = argminima(arr, num, ordered=False)
>>>     assert sorted(idxs2) == sorted(idxs), 'ordered=False must return the right_
↳ idxs, but in any order'
```

Example

```
>>> # Test cases with axis
>>> from kwarray.util_numpy import * # NOQA
>>> arr = (np.random.rand(3, 5, 7) * 100).astype(int)
>>> # make a unique array so we can check argmax consistency
>>> arr = np.arange(3 * 5 * 7)
>>> np.random.shuffle(arr)
>>> arr = arr.reshape(3, 5, 7)
>>> for axis in range(len(arr.shape)):
>>>     for num in range(0, len(arr) + 1):
>>>         idxs = argminima(arr, num, axis=axis)
>>>         idxs2 = argminima(arr, num, ordered=False, axis=axis)
>>>         print('idxs = {!r}'.format(idxs))
>>>         print('idxs2 = {!r}'.format(idxs2))
>>>         assert idxs.shape[axis] == num
>>>         assert idxs2.shape[axis] == num
>>>         # Check if argmin agrees with -argmax
>>>         idxs3 = argmaxima(-arr, num, axis=axis)
>>>         assert np.all(idxs3 == idxs)
```

Example

```
>>> arr = np.arange(20).reshape(4, 5) % 6
>>> argminima(arr, axis=1, num=2, ordered=False)
>>> argminima(arr, axis=1, num=2, ordered=True)
>>> argmaxima(-arr, axis=1, num=2, ordered=True)
>>> argmaxima(-arr, axis=1, num=2, ordered=False)
```

`kwarray.util_numpy.unique_rows(arr, ordered=False, return_index=False)`

Like `unique`, but works on rows

Parameters

- **arr** (*NDArray*) – must be a contiguous C style array
- **ordered** (*bool*) – if true, keeps relative ordering

References

<https://stackoverflow.com/questions/16970982/find-unique-rows-in-numpy-array>

Example

```
>>> import kwarray
>>> from kwarray.util_numpy import * # NOQA
>>> rng = kwarray.ensure_rng(0)
>>> arr = rng.randint(0, 2, size=(22, 3))
>>> arr_unique = unique_rows(arr)
>>> print('arr_unique = {!r}'.format(arr_unique))
>>> arr_unique, idxs = unique_rows(arr, return_index=True, ordered=True)
>>> assert np.all(arr[idxs] == arr_unique)
>>> print('arr_unique = {!r}'.format(arr_unique))
>>> print('idxs = {!r}'.format(idxs))
>>> arr_unique, idxs = unique_rows(arr, return_index=True, ordered=False)
>>> assert np.all(arr[idxs] == arr_unique)
>>> print('arr_unique = {!r}'.format(arr_unique))
>>> print('idxs = {!r}'.format(idxs))
```

`kwarray.util_numpy.arglexmax(keys, multi=False)`

Find the index of the maximum element in a sequence of keys.

Parameters

- **keys** (*tuple*) – a k-tuple of k N-dimensional arrays. Like `np.lexsort` the last key in the sequence is used for the primary sort order, the second-to-last key for the secondary sort order, and so on.
- **multi** (*bool*) – if `True`, returns all indices that share the max value

Returns

either the index or list of indices

Return type

`int` | `NDArray[Any, Int]`

Example

```
>>> k, N = 100, 100
>>> rng = np.random.RandomState(0)
>>> keys = [(rng.rand(N) * N).astype(int) for _ in range(k)]
>>> multi_idx = arglexmax(keys, multi=True)
>>> idxs = np.lexsort(keys)
>>> assert sorted(idxs[::-1][:len(multi_idx)]) == sorted(multi_idx)
```

Benchark:

```
>>> import ubelt as ub
>>> k, N = 100, 100
>>> rng = np.random
>>> keys = [(rng.rand(N) * N).astype(int) for _ in range(k)]
```

(continues on next page)

(continued from previous page)

```

>>> for timer in ub.Timerit(100, bestof=10, label='arglexmax'):
>>>     with timer:
>>>         arglexmax(keys)
>>> for timer in ub.Timerit(100, bestof=10, label='lexsort'):
>>>     with timer:
>>>         np.lexsort(keys)[-1]

```

`kwarray.util_numpy.generalized_logistic(x, floor=0, capacity=1, C=1, y_intercept=None, Q=None, growth=1, v=1)`

A generalization of the logistic / sigmoid functions that allows for flexible specification of S-shaped curve.

This is also known as a “Richards curve” [[WikiRichardsCurve](#)].

Parameters

- **x** (*NDArray*) – input x coordinates
- **floor** (*float*) – the lower (left) asymptote. (Also called **A** in some texts). Defaults to 0.
- **capacity** (*float*) – the carrying capacity. When **C**=1, this is the upper (right) asymptote. (Also called **K** in some texts). Defaults to 1.
- **C** (*float*) – Has influence on the upper asymptote. Defaults to 1. This is typically not modified.
- **y_intercept** (*float | None*) – specify where the the y intercept is at x=0. Mutually exclusive with **Q**.
- **Q** (*float | None*) – related to the value of the function at x=0. Mutually exclusive with **y_intercept**. Defaults to 1.
- **growth** (*float*) – the growth rate (also calle **B** in some texts). Defaults to 1.
- **v** (*float*) – Positive number that influences near which asymptote the growth occurs. Defaults to 1.

Returns

the values for each input

Return type

NDArray

References

Example

```

>>> from kwarray.util_numpy import * # NOQA
>>> # xdoctest: +REQUIRES(module:pandas)
>>> import pandas as pd
>>> import ubelt as ub
>>> x = np.linspace(-3, 3, 30)
>>> basis = {
>>>     # 'y_intercept': [0.1, 0.5, 0.8, -1],
>>>     # 'y_intercept': [0.1, 0.5, 0.8],
>>>     'v': [0.5, 1.0, 2.0],
>>>     'growth': [-1, 0, 2],

```

(continues on next page)

(continued from previous page)

```

>>> }
>>> grid = list(ub.named_product(basis))
>>> datas = []
>>> for params in grid:
>>>     y = generalized_logistic(x, **params)
>>>     data = pd.DataFrame({'x': x, 'y': y})
>>>     key = ub.urepr(params, compact=1)
>>>     data['key'] = key
>>>     for k, v in params.items():
>>>         data[k] = v
>>>     datas.append(data)
>>> all_data = pd.concat(datas).reset_index()
>>> # xdoctest: +REQUIRES(--show)
>>> # xdoctest: +REQUIRES(module:kwplot)
>>> import kwplot
>>> plt = kwplot.autoplt()
>>> sns = kwplot.autosns()
>>> plt.gca().cla()
>>> sns.lineplot(data=all_data, x='x', y='y', hue='growth', size='v')

```

kwarray.util_numpy.equal_with_nan(a1, a2)

Numpy has array_equal with equal_nan=True, but this is elementwise

Parameters

- **a1** (*ArrayLike*) – input array
- **a2** (*ArrayLike*) – input array

Example

```

>>> import kwarray
>>> a1 = np.array([
>>>     [np.nan, 0, np.nan],
>>>     [np.nan, 0, 0],
>>>     [np.nan, 1, 0],
>>>     [np.nan, 1, np.nan],
>>> ])
>>> a2 = np.array([np.nan, 0, np.nan])
>>> flags = kwarray.equal_with_nan(a1, a2)
>>> assert np.array_equal(flags, np.array([
>>>     [ True, False,  True],
>>>     [ True, False, False],
>>>     [ True,  True, False],
>>>     [ True,  True,  True]
>>> ]))

```

kwarray.util_random module

Handle and interchange between different random number generators (numpy, python, torch, ...). Also defines useful random iterator functions and [ensure_rng\(\)](#).

Random Number Generator Patterns

If you need a seeded random number generator `kwarray.ensure_rng` is helpful with that: [kwarray.util_random.ensure_rng\(\)](#)

If the input is a number it returns a seeded random number generator. If it is `None` it returns whatever the system level RNG is. If the input is an existing RNG it returns it without changing it. It also has the ability to switch between Python's random module RNG and numpy's `np.random` RNG (it can translate the internal state between the two).

When I write randomized functions / class, a coding pattern I like is to have a default keyword argument `rng=None`. Then `kwarray.ensure_rng` coerces whatever the input is into a `random.Random()` or `numpy.random.RandomState()` object.

```
def some_random_function(*args, rng=None):
    rng = kwarray.ensure_rng(rng)
```

Then if this random function calls any other random function, it passes the coerced `rng` to all other subfunctions. This ensures that seeing the RNG at the top level produces a completely deterministic process.

For a more involved example

```
import pandas as pd
import numpy
import kwarray

def random_subfunc1(rng=None):
    rng = kwarray.ensure_rng(rng, api='python')
    value: float = rng.betavariate(3, 2.3)
    return value

def random_subfunc2(rng=None):
    rng = kwarray.ensure_rng(rng, api='numpy')
    arr: np.ndarray = rng.choice([1, 2, 3, 4], size=3, replace=0)
    return arr

def random_method(rng=None):
    value = random_subfunc1(rng=rng)
    arr = random_subfunc2(rng=rng)
    final = (arr * value).sum()
    return final

def demo():
    results = []
    num = 10
    for _ in range(num):
        rng = np.random.RandomState(3)
        row = {}
```

(continues on next page)

(continued from previous page)

```
row['system'] = random_method(None)
row['seeded'] = random_method(0)
row['exiting'] = random_method(rng)
results.append(row)

df = pd.DataFrame(results)
print(df)
```

This results in:

	system	seeded	exiting
0	3.642700	6.902354	4.869275
1	3.127890	6.902354	4.869275
2	4.317397	6.902354	4.869275
3	3.382259	6.902354	4.869275
4	1.999498	6.902354	4.869275
5	5.293688	6.902354	4.869275
6	2.984741	6.902354	4.869275
7	6.455160	6.902354	4.869275
8	5.161900	6.902354	4.869275
9	2.810358	6.902354	4.869275

`kwarray.util_random.seed_global(seed, offset=0)`

Seeds the python, numpy, and torch global random states

Parameters

- **seed** (*int*) – seed to use
- **offset** (*int*) – if specified, uses a different seed for each global random state separated by this offset. Defaults to 0.

`kwarray.util_random.shuffle(items, rng=None)`

Shuffles a list inplace and then returns it for convinience

Parameters

- **items** (*list* | *ndarray*) – data to shuffle
- **rng** (*int* | *float* | *None* | *numpy.random.RandomState* | *random.Random*) – seed or random number gen

Returns

this is the input, but returned for convinience

Return type

list

Example

```
>>> list1 = [1, 2, 3, 4, 5, 6]
>>> list2 = shuffle(list(list1), rng=1)
>>> assert list1 != list2
>>> result = str(list2)
>>> print(result)
[3, 2, 5, 1, 4, 6]
```

`kwarray.util_random.random_combinations(items, size, num=None, rng=None)`

Yields num combinations of length size from items in random order

Parameters

- **items** (*List*) – pool of items to choose from
- **size** (*int*) – Number of items in each combination
- **num** (*int* | *None*) – Number of combinations to generate. If *None*, generate them all.
- **rng** (*int* | *float* | *None* | *numpy.random.RandomState* | *random.Random*) – seed or random number generator. Defaults to the global state of the python random module.

Yields

Tuple – a random combination of items of length size.

Example

```
>>> # xdoctest: +REQUIRES(module:scipy)
>>> import ubelt as ub
>>> items = list(range(10))
>>> size = 3
>>> num = 5
>>> rng = 0
>>> # xdoctest: +IGNORE_WANT
>>> combos = list(random_combinations(items, size, num, rng))
>>> print('combos = {}'.format(ub.urepr(combos, nl=1)))
combos = [
    (0, 6, 9),
    (4, 7, 8),
    (4, 6, 7),
    (2, 3, 5),
    (1, 2, 4),
]
```

Example

```
>>> # xdoctest: +REQUIRES(module:scipy)
>>> import ubelt as ub
>>> items = list(zip(range(10), range(10)))
>>> # xdoctest: +IGNORE_WANT
>>> combos = list(random_combinations(items, 3, num=5, rng=0))
>>> print('combos = {}'.format(ub.urepr(combos, nl=1)))
combos = [
    ((0, 0), (6, 6), (9, 9)),
    ((4, 4), (7, 7), (8, 8)),
    ((4, 4), (6, 6), (7, 7)),
    ((2, 2), (3, 3), (5, 5)),
    ((1, 1), (2, 2), (4, 4)),
]
```

`kwarray.util_random.random_product(items, num=None, rng=None)`

Yields num items from the cartesian product of items in a random order.

Parameters

- **items** (*List[Sequence]*) – items to get cartesian product of packed in a list or tuple. (note this deviates from api of `itertools.product()`)
- **num** (*int | None*) – maximum number of items to generate. If None generate them all
- **rng** (*int | float | None | numpy.random.RandomState | random.Random*) – Seed or random number generator. Defaults to the global state of the python random module.

Yields

Tuple – a random item in the cartesian product

Example

```
>>> import ubelt as ub
>>> items = [(1, 2, 3), (4, 5, 6, 7)]
>>> rng = 0
>>> # xdoctest: +IGNORE_WANT
>>> products = list(random_product(items, rng=0))
>>> print(ub.urepr(products, nl=0))
[(3, 4), (1, 7), (3, 6), (2, 7), ... (1, 6), (2, 5), (2, 4)]
>>> products = list(random_product(items, num=3, rng=0))
>>> print(ub.urepr(products, nl=0))
[(3, 4), (1, 7), (3, 6)]
```


Example

```
>>> # xdoctest: +REQUIRES(--profile)
>>> rng = ensure_rng(0)
>>> items = [np.array([15, 14]), np.array([27, 26]),
>>>          np.array([21, 22]), np.array([32, 31])]
>>> num = 2
>>> for _ in range(100):
>>>     list(random_product(items, num=num, rng=rng))
```

`kwarray.util_random._npstate_to_pystate(npstate)`

Convert state of a NumPy RandomState object to a state that can be used by Python's Random. Derived from [\[SO44313620\]](#).

References

Example

```
>>> py_rng = random.Random(0)
>>> np_rng = np.random.RandomState(seed=0)
>>> npstate = np_rng.get_state()
>>> pystate = _npstate_to_pystate(npstate)
>>> py_rng.setstate(pystate)
>>> assert np_rng.rand() == py_rng.random()
```

`kwarray.util_random._pystate_to_npstate(pystate)`

Convert state of a Python Random object to state usable by NumPy RandomState. Derived from [\[SO44313620\]](#).

References

Example

```
>>> py_rng = random.Random(0)
>>> np_rng = np.random.RandomState(seed=0)
>>> pystate = py_rng.getstate()
>>> npstate = _pystate_to_npstate(pystate)
>>> np_rng.set_state(npstate)
>>> assert np_rng.rand() == py_rng.random()
```

`kwarray.util_random._coerce_rng_type(rng)`

Internal method that transforms input seeds into an integer form.

`kwarray.util_random.ensure_rng(rng=None, api='numpy')`

Coerces input into a random number generator.

This function is useful for ensuring that your code uses a controlled internal random state that is independent of other modules.

If the input is None, then a global random state is returned.

If the input is a numeric value, then that is used as a seed to construct a random state.

If the input is a random number generator, then another random number generator with the same state is returned. Depending on the api, this random state is either return as-is, or used to construct an equivalent random state with the requested api.

Parameters

- **rng** (*int* | *float* | *None* | *numpy.random.RandomState* | *random.Random*) – if *None*, then defaults to the global rng. Otherwise this can be an integer or a *RandomState* class. Defaults to the global random.
- **api** (*str*) – specify the type of random number generator to use. This can either be ‘numpy’ for a *numpy.random.RandomState* object or ‘python’ for a *random.Random* object. Defaults to numpy.

Returns

rng - either a numpy or python random number generator, depending on the setting of api.

Return type

(*numpy.random.RandomState* | *random.Random*)

Example

```
>>> rng = ensure_rng(None)
>>> ensure_rng(0).randint(0, 1000)
684
>>> ensure_rng(np.random.RandomState(1)).randint(0, 1000)
37
```

Example

```
>>> num = 4
>>> print('--- Python as PYTHON ---')
>>> py_rng = random.Random(0)
>>> pp_nums = [py_rng.random() for _ in range(num)]
>>> print(pp_nums)
>>> print('--- Numpy as PYTHON ---')
>>> np_rng = ensure_rng(random.Random(0), api='numpy')
>>> np_nums = [np_rng.rand() for _ in range(num)]
>>> print(np_nums)
>>> print('--- Numpy as NUMPY---')
>>> np_rng = np.random.RandomState(seed=0)
>>> nn_nums = [np_rng.rand() for _ in range(num)]
>>> print(nn_nums)
>>> print('--- Python as NUMPY---')
>>> py_rng = ensure_rng(np.random.RandomState(seed=0), api='python')
>>> pn_nums = [py_rng.random() for _ in range(num)]
>>> print(pn_nums)
>>> assert np_nums == pp_nums
>>> assert pn_nums == nn_nums
```

Example

```

>>> # Test that random modules can be coerced
>>> import random
>>> import numpy as np
>>> ensure_rng(random, api='python')
>>> ensure_rng(random, api='numpy')
>>> ensure_rng(np.random, api='python')
>>> ensure_rng(np.random, api='numpy')

```

kwarray.util_robust module

Functions relating to robust statistical methods for normalizing data.

`kwarray.util_robust.find_robust_normalizers(data, params='auto')`

Finds robust normalization statistics a set of scalar observations.

The idea is to estimate “fense” parameters: minimum and maximum values where anything under / above these values are likely outliers. For non-linear normalizaiton schemes we can also estimate an likely middle and extent of the data.

Parameters

- **data** (*ndarray*) – a 1D numpy array where invalid data has already been removed
- **params** (*str* | *dict*) – normalization params.

When passed as a dictionary valid params are:

scaling (str):

This is the “mode” that will be used in the final normalization. Currently has no impact on the Defaults to ‘linear’. Can also be ‘sigmoid’.

extrema (str):

The method for determening what the extrama are. Can be “quantile” for strict quantile clipping Can be “adaptive-quantile” for an IQR-like adjusted quantile method. Can be “tukey” or “IQR” for an exact IQR method.

low (float): This is the low quantile for likely inliers.

mid (float): This is the middle quantlie for likely inliers.

high (float): This is the high quantile for likely inliers.

Can be specified as a concise string.

The string “auto” defaults to:

```
dict(extrema='adaptive-quantile', scaling='linear', low=0.01,
mid=0.5, high=0.9).
```

The string “tukey” defaults to:

```
dict(extrema='tukey', scaling='linear').
```

Returns

normalization parameters that can be passed to `kwarray.normalize()` containing the keys:

type (str): which is always ‘normalize’

mode (str): the value of `params['scaling']`

min_val (float): the determined “robust” minimum inlier value.

max_val (float): the determined “robust” maximum inlier value.

beta (float): the determined “robust” middle value for use in non-linear normalizers.

alpha (float): the determined “robust” extent value for use in non-linear normalizers.

Return type

Dict[str, str | float]

Note: The defaults and methods of this function are subject to change.

Todo:

- [] No (or minimal) Magic Numbers! Use first principles to determine defaults.
 - [] Probably a lot of literature on the subject.
 - [] <https://arxiv.org/pdf/1707.09752.pdf>
 - [] <https://www.tandfonline.com/doi/full/10.1080/02664763.2019.1671961>
 - [] <https://www.rips-irsp.com/articles/10.5334/irsp.289/>
 - [] **This function is not possible to get right in every case**
(probably can prove this with a NFL theroem), might be useful to allow the user to specify a “model” which is specific to some domain.
-

Example

```
>>> from kwarray.util_robust import * # NOQA
>>> data = np.random.rand(100)
>>> norm_params1 = find_robust_normalizers(data, params='auto')
>>> norm_params2 = find_robust_normalizers(data, params={'low': 0, 'high': 1.0})
>>> norm_params3 = find_robust_normalizers(np.empty(0), params='auto')
>>> print('norm_params1 = {}'.format(ub.urepr(norm_params1, nl=1)))
>>> print('norm_params2 = {}'.format(ub.urepr(norm_params2, nl=1)))
>>> print('norm_params3 = {}'.format(ub.urepr(norm_params3, nl=1)))
```

Example

```
>>> # xdoctest: +REQUIRES(module:scipy)
>>> from kwarray.util_robust import * # NOQA
>>> from kwarray.distributions import Mixture
>>> import ubelt as ub
>>> # A random mixture distribution for testing
>>> data = Mixture.random(6).sample(3000)
```

`kwarray.util_robust._tukey_quantile_fence(data, clip=False)`

One might wonder where the 1.5 in the above interval comes from – Paul Velleman, a statistician at Cornell University, was a student of John Tukey, who invented this test for outliers. He wondered the same thing. When he asked Tukey, “Why 1.5?”, Tukey answered, “Because 1 is too small and 2 is too large.” [OxfordShapeSpread].

References

`kvarray.util_robust._quantile_extreme_estimator(data, params)`

`kvarray.util_robust._custom_quantile_extreme_estimator(data, params)`

`kvarray.util_robust.robust_normalize(imdata, return_info=False, nodata=None, axis=None, dtype=<class 'numpy.float32'>, params='auto', mask=None)`

Normalize data intensities using heuristics to help put sensor data with extremely high or low contrast into a visible range.

This function is designed with an emphasis on getting something that is reasonable for visualization.

Todo:

- [x] Move to kvarray and renamed to robust_normalize?
 - [] Support for M-estimators?
-

Parameters

- **imdata** (*ndarray*) – raw intensity data
- **return_info** (*bool*) – if True, return information about the chosen normalization heuristic.
- **params** (*str* | *dict*) – Can contain keys, low, high, or mid, scaling, extrema e.g. {'low': 0.1, 'mid': 0.8, 'high': 0.9, 'scaling': 'sigmoid'} See documentation in [find_robust_normalizers\(\)](#).
- **axis** (*None* | *int*) – The axis to normalize over, if unspecified, normalize jointly
- **nodata** (*None* | *int*) – A value representing nodata to leave unchanged during normalization, for example 0
- **dtype** (*type*) – can be float32 or float64
- **mask** (*ndarray* | *None*) – A mask indicating what pixels are valid and what pixels should be considered nodata. Mutually exclusive with nodata argument. A mask value of 1 indicates a VALID pixel. A mask value of 0 indicates an INVALID pixel. Note this is the opposite of a masked array.

Returns

a floating point array with values between 0 and 1. if return_info is specified, also returns extra data

Return type

ndarray | *Tuple*[*ndarray*, *Any*]

Note: This is effectively a combination of [find_robust_normalizers\(\)](#) and [normalize\(\)](#).

Example

```
>>> # xdoctest: +REQUIRES(module:scipy)
>>> from kwarray.util_robust import * # NOQA
>>> from kwarray.distributions import Mixture
>>> import ubelt as ub
>>> # A random mixture distribution for testing
>>> data = Mixture.random(6).sample(3000)
>>> param_basis = {
>>>     'scaling': ['linear', 'sigmoid'],
>>>     'high': [0.6, 0.8, 0.9, 1.0],
>>> }
>>> param_grid = list(ub.named_product(param_basis))
>>> param_grid += ['auto']
>>> param_grid += ['tukey']
>>> rows = []
>>> rows.append({'key': 'orig', 'result': data})
>>> for params in param_grid:
>>>     key = ub.urepr(params, compact=1)
>>>     result, info = robust_normalize(data, return_info=True, params=params)
>>>     print('key = {}'.format(key))
>>>     print('info = {}'.format(ub.urepr(info, nl=1)))
>>>     rows.append({'key': key, 'info': info, 'result': result})
>>> # xdoctest: +REQUIRES(--show)
>>> import seaborn as sns
>>> import kwplot
>>> kwplot.autompl()
>>> pnum_ = kwplot.PlotNums(nSubplots=len(rows))
>>> for row in rows:
>>>     ax = kwplot.figure(fnum=1, pnum=pnum_()).gca()
>>>     sns.histplot(data=row['result'], kde=True, bins=128, ax=ax, stat='density')
>>>     ax.set_title(row['key'])
```

Example

```
>>> # xdoctest: +REQUIRES(module:kwimage)
>>> from kwarray.util_robust import * # NOQA
>>> import ubelt as ub
>>> import kwimage
>>> import kwarray
>>> s = 512
>>> bit_depth = 11
>>> dtype = np.uint16
>>> max_val = int(2 ** bit_depth)
>>> min_val = int(0)
>>> rng = kwarray.ensure_rng(0)
>>> background = np.random.randint(min_val, max_val, size=(s, s), dtype=dtype)
>>> poly1 = kwimage.Polygon.random(rng=rng).scale(s / 2)
>>> poly2 = kwimage.Polygon.random(rng=rng).scale(s / 2).translate(s / 2)
>>> foreground = np.zeros_like(background, dtype=np.uint8)
>>> foreground = poly1.fill(foreground, value=255)
```

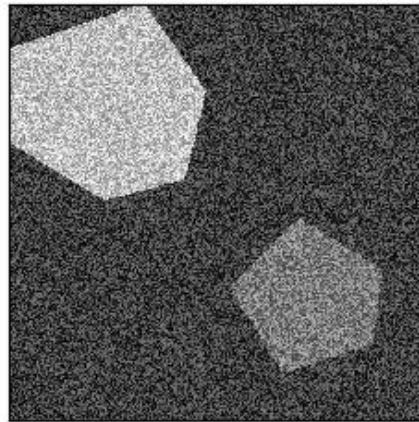
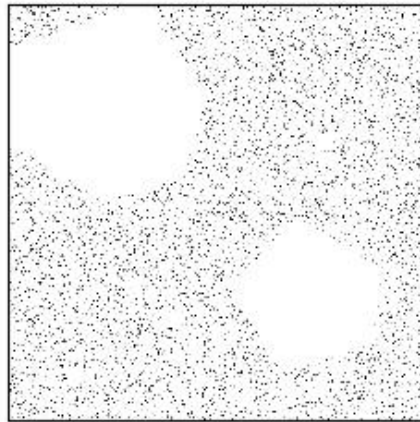
(continues on next page)

(continued from previous page)

```

>>> foreground = poly2.fill(forground, value=122)
>>> foreground = (kwimage.ensure_float01(forground) * max_val).astype(dtype)
>>> imdata = background + foreground
>>> normed, info = kwarray.robust_normalize(imdata, return_info=True)
>>> print('info = {}'.format(ub.urepr(info, nl=1)))
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(imdata, pnum=(1, 2, 1), fnum=1)
>>> kwplot.imshow(normed, pnum=(1, 2, 2), fnum=1)

```



`kwarray.util_robust._apply_robust_normalizer(normalizer, imdata, imdata_valid, mask, dtype, copy=True)`

Todo: abstract into a scikit-learn-style Normalizer class which can fit/predict different types of normalizers.

`kwarray.util_robust.normalize(arr, mode='linear', alpha=None, beta=None, out=None, min_val=None, max_val=None)`

Normalizes input values based on a specified scheme.

The default behavior is a linear normalization between 0.0 and 1.0 based on the min/max values of the input. Parameters can be specified to achieve more general constrat stretching or signal rebalancing. Implements the linear and sigmoid normalization methods described in [WikiNorm].

Parameters

- **arr** (*NDArray*) – array to normalize, usually an image
- **out** (*NDArray* | *None*) – output array. Note, that we will create an internal floating point copy for integer computations.
- **mode** (*str*) – either linear or sigmoid.
- **alpha** (*float*) – Only used if mode=sigmoid. Division factor (pre-sigmoid). If unspecified computed as: $\max(\text{abs}(\text{old_min} - \text{beta}), \text{abs}(\text{old_max} - \text{beta})) / 6.212606$. Note this parameter is sensitive to if the input is a float or uint8 image.
- **beta** (*float*) – subtractive factor (pre-sigmoid). This should be the intensity of the most interesting bits of the image, i.e. bring them to the center (0) of the distribution. Defaults to $(\text{max} - \text{min}) / 2$. Note this parameter is sensitive to if the input is a float or uint8 image.
- **min_val** – inputs lower than this minimum value are clipped
- **max_val** – inputs higher than this maximum value are clipped.

SeeAlso:

[*find_robust_normalizers\(\)*](#) - determine robust parameters for normalize to mitigate the effect of outliers.

[*robust_normalize\(\)*](#) - finds and applies robust normalization parameters

References

Example

```
>>> raw_f = np.random.rand(8, 8)
>>> norm_f = normalize(raw_f)
```

```
>>> raw_f = np.random.rand(8, 8) * 100
>>> norm_f = normalize(raw_f)
>>> assert isclose(norm_f.min(), 0)
>>> assert isclose(norm_f.max(), 1)
```

```
>>> raw_u = (np.random.rand(8, 8) * 255).astype(np.uint8)
>>> norm_u = normalize(raw_u)
```

Example

```
>>> # xdoctest: +REQUIRES(module:kwimage)
>>> import kwimage
>>> arr = kwimage.grab_test_image('lowcontrast')
>>> arr = kwimage.ensure_float01(arr)
>>> norms = {}
>>> norms['arr'] = arr.copy()
>>> norms['linear'] = normalize(arr, mode='linear')
>>> # xdoctest: +REQUIRES(module:scipy)
```

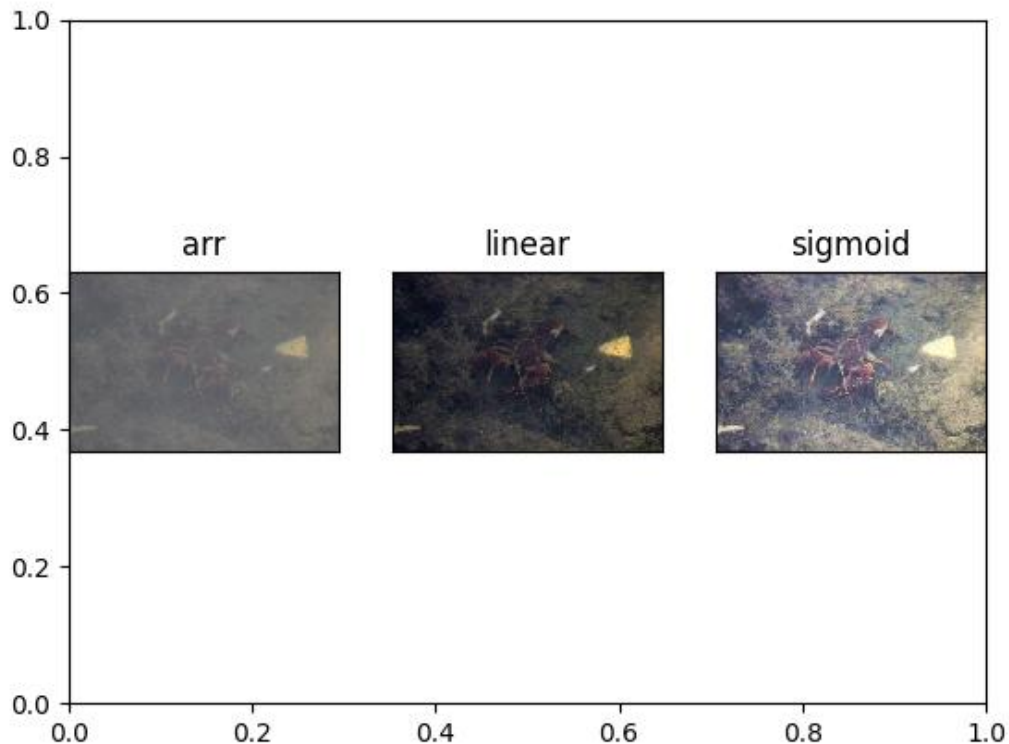
(continues on next page)

(continued from previous page)

```

>>> norms['sigmoid'] = normalize(arr, mode='sigmoid')
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.figure(fnum=1, doclf=True)
>>> pnum_ = kwplot.PlotNums(nSubplots=len(norms))
>>> for key, img in norms.items():
>>>     kwplot.imshow(img, pnum=pnum_(), title=key)

```



Example

```

>>> # xdoctest: +REQUIRES(module:kwimage)
>>> arr = np.array([np.inf])
>>> normalize(arr, mode='linear')
>>> # xdoctest: +REQUIRES(module:scipy)
>>> normalize(arr, mode='sigmoid')
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.figure(fnum=1, doclf=True)
>>> pnum_ = kwplot.PlotNums(nSubplots=len(norms))
>>> for key, img in norms.items():

```

(continues on next page)

(continued from previous page)

```
>>> kwplot.imshow(img, pnum=pnum_(), title=key)
```

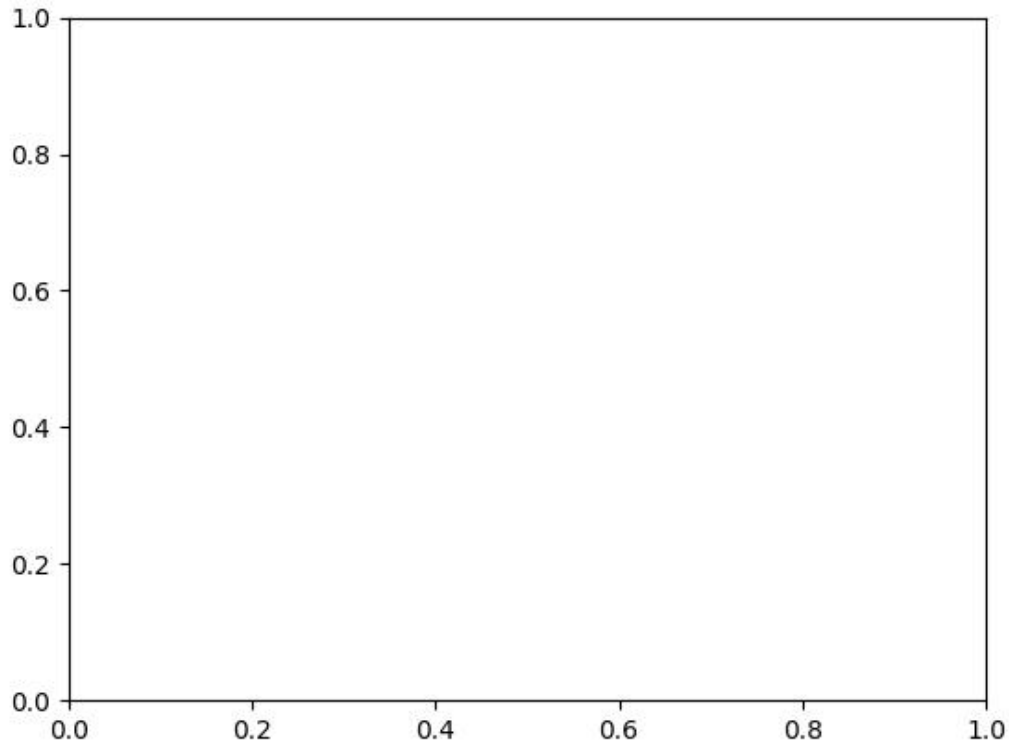
Benchmark

```
>>> # Our method is faster than standard in-line implementations for
>>> # uint8 and competitive with in-line float32, in addition to being
>>> # more concise and configurable. In 3.11 all inplace variants are
>>> # faster.
>>> # xdoctest: +REQUIRES(module:kwimage)
>>> import timerit
>>> import kwimage
>>> import kwarray
>>> ti = timerit.Timerit(1000, bestof=10, verbose=2, unit='ms')
>>> arr = kwimage.grab_test_image('lowcontrast', dsize=(512, 512))
>>> #
>>> arr = kwimage.ensure_float01(arr)
>>> out = arr.copy()
>>> for timer in ti.reset('inline_naive(float)':
>>>     with timer:
>>>         (arr - arr.min()) / (arr.max() - arr.min())
>>> #
>>> for timer in ti.reset('inline_faster(float)':
>>>     with timer:
>>>         max_ = arr.max()
>>>         min_ = arr.min()
>>>         result = (arr - min_) / (max_ - min_)
>>> #
>>> for timer in ti.reset('kwarray.normalize(float)':
>>>     with timer:
>>>         kwarray.normalize(arr)
>>> #
>>> for timer in ti.reset('kwarray.normalize(float, inplace)':
>>>     with timer:
>>>         kwarray.normalize(arr, out=out)
>>> #
>>> arr = kwimage.ensure_uint255(arr)
>>> out = arr.copy()
>>> for timer in ti.reset('inline_naive(uint8)':
>>>     with timer:
>>>         (arr - arr.min()) / (arr.max() - arr.min())
>>> #
>>> for timer in ti.reset('inline_faster(uint8)':
>>>     with timer:
>>>         max_ = arr.max()
>>>         min_ = arr.min()
>>>         result = (arr - min_) / (max_ - min_)
>>> #
>>> for timer in ti.reset('kwarray.normalize(uint8)':
>>>     with timer:
>>>         kwarray.normalize(arr)
```

(continues on next page)

(continued from previous page)

```
>>> #
>>> for timer in ti.reset('kwarray.normalize(uint8, inplace)':
>>>     with timer:
>>>         kwarray.normalize(arr, out=out)
>>> print('ti.rankings = {}'.format(ub.urepr(
>>>     ti.rankings, nl=2, align=':', precision=5)))
```



kwarray.util_slices module

Utilities related to slicing

References

<https://stackoverflow.com/questions/41153803/zero-padding-slice-past-end-of-array-in-numpy>

Todo:

- [] Could have a kwarray function to expose this inverse slice functionality. Also having a top-level call to apply an embedded slice would be good.

`kwarray.util_slices.padded_slice(data, slices, pad=None, padkw=None, return_info=False)`

Allows slices with out-of-bound coordinates. Any out of bounds coordinate will be sampled via padding.

Parameters

- **data** (*Sliceable*) – data to slice into. Any channels must be the last dimension.
- **slices** (*slice* | *Tuple[slice, ...]*) – slice for each dimensions
- **ndim** (*int*) – number of spatial dimensions
- **pad** (*List[int|Tuple]*) – additional padding of the slice
- **padkw** (*Dict*) – if unspecified defaults to {'mode': 'constant'}
- **return_info** (*bool*, *default=False*) – if True, return extra information about the transform.

Note: Negative slices have a different meaning here then they usually do. Normally, they indicate a wrap-around or a reversed stride, but here they index into out-of-bounds space (which depends on the pad mode). For example a slice of -2:1 literally samples two pixels to the left of the data and one pixel from the data, so you get two padded values and one data value.

SeeAlso:

embed_slice - finds the embedded slice and padding

Returns

data_sliced: subregion of the input data (possibly with padding,
depending on if the original slice went out of bounds)

Tuple[Sliceable, Dict] :

data_sliced : as above

transform : information on how to return to the original coordinates

Currently a dict containing:

st_dims: a list indicating the low and high space-time
coordinate values of the returned data slice.

The structure of this dictionary mach change in the future

Return type

Sliceable

Example

```
>>> import kwarray
>>> data = np.arange(5)
>>> slices = [slice(-2, 7)]
```

```
>>> data_sliced = kwarray.padded_slice(data, slices)
>>> print(ub.urepr(data_sliced, with_dtype=False))
np.array([0, 0, 0, 1, 2, 3, 4, 0, 0])
```

```
>>> data_sliced = kwarray.padded_slice(data, slices, pad=[(3, 3)])
>>> print(ub.urepr(data_sliced, with_dtype=False))
np.array([0, 0, 0, 0, 0, 0, 1, 2, 3, 4, 0, 0, 0, 0, 0])
```

```
>>> data_sliced = kwarray.padded_slice(data, slice(3, 4), pad=[(1, 0)])
>>> print(ub.urepr(data_sliced, with_dtype=False))
np.array([2, 3])
```

`kwarray.util_slices.apply_embedded_slice(data, data_slice, extra_padding, **padkw)`

Apply a precomputed embedded slice.

This is used as a subroutine in `padded_slice`.

Parameters

- **data** (*ndarray*) – data to slice
- **data_slice** (*Tuple[slice]*)
- **extra_padding** (*Tuple[slice]*)

Returns

ndarray

`kwarray.util_slices._apply_padding(array, pad_width, **padkw)`

Alternative to numpy pad with different short-cut semantics for the “pad_width” argument.

Unlike numpy pad, you must specify a (start, stop) tuple for each dimension. The shortcut is that you only need to specify this for the leading dimensions. Any unspecified trailing dimension will get an implicit (0, 0) padding.

TODO: does this get exposed as a public function?

`kwarray.util_slices.embed_slice(slices, data_dims, pad=None)`

Embeds a “padded-slice” inside known data dimension.

Returns the valid data portion of the slice with extra padding for regions outside of the available dimension.

Given a slices for each dimension, image dimensions, and a padding get the corresponding slice from the image and any extra padding needed to achieve the requested window size.

Todo:

- [] Add the option to return the inverse slice
-

Parameters

- **slices** (*Tuple[slice, ...]*) – a tuple of slices for to apply to data data dimension.
- **data_dims** (*Tuple[int, ...]*) – n-dimension data sizes (e.g. 2d height, width)
- **pad** (*int | List[int | Tuple[int, int]]*) – extra pad applied to (start / end) / (both) sides of each slice dim

Returns

data_slice - *Tuple[slice]* a slice that can be applied to an array

with with shape *data_dims*. This slice will not correspond to the full window size if the requested slice is out of bounds.

extra_padding - extra padding needed after slicing to achieve the requested window size.

Return type

Tuple

Example

```
>>> # Case where slice is inside the data dims on left edge
>>> import kwarray
>>> slices = (slice(0, 10), slice(0, 10))
>>> data_dims = [300, 300]
>>> pad = [10, 5]
>>> a, b = kwarray.embed_slice(slices, data_dims, pad)
>>> print('data_slice = {!r}'.format(a))
>>> print('extra_padding = {!r}'.format(b))
data_slice = (slice(0, 20, None), slice(0, 15, None))
extra_padding = [(10, 0), (5, 0)]
```

Example

```
>>> # Case where slice is bigger than the image
>>> import kwarray
>>> slices = (slice(-10, 400), slice(-10, 400))
>>> data_dims = [300, 300]
>>> pad = [10, 5]
>>> a, b = kwarray.embed_slice(slices, data_dims, pad)
>>> print('data_slice = {!r}'.format(a))
>>> print('extra_padding = {!r}'.format(b))
data_slice = (slice(0, 300, None), slice(0, 300, None))
extra_padding = [(20, 110), (15, 105)]
```

Example

```
>>> # Case where slice is inside than the image
>>> import kwarray
>>> slices = (slice(10, 40), slice(10, 40))
>>> data_dims = [300, 300]
>>> pad = None
>>> a, b = kwarray.embed_slice(slices, data_dims, pad)
>>> print('data_slice = {!r}'.format(a))
>>> print('extra_padding = {!r}'.format(b))
data_slice = (slice(10, 40, None), slice(10, 40, None))
extra_padding = [(0, 0), (0, 0)]
```

Example

```
>>> # Test error cases
>>> import kwarray
>>> import pytest
>>> slices = (slice(0, 40), slice(10, 40))
>>> data_dims = [300, 300]
>>> with pytest.raises(ValueError):
>>>     kwarray.embed_slice(slices, data_dims[0:1])
```

(continues on next page)

(continued from previous page)

```

>>> with pytest.raises(ValueError):
>>>     kwarray.embed_slice(slices[0:1], data_dims)
>>> with pytest.raises(ValueError):
>>>     kwarray.embed_slice(slices, data_dims, pad=[(1, 1)])
>>> with pytest.raises(ValueError):
>>>     kwarray.embed_slice(slices, data_dims, pad=[1])

```

`kwarray.util_slices._coerce_pad(pad, ndims)`

kwarray.util_slider module

Defines the *SlidingWindow* and *Sticher* classes.

The *SlidingWindow* generates a grid of slices over an `numpy.ndarray()`, which can then be used to compute on subsets of the data. The *Sticher* can then take these results and recombine them into a final result that matches the larger array.

class `kwarray.util_slider.SlidingWindow(shape, window, overlap=None, stride=None, keepbound=False, allow_overshoot=False)`

Bases: `NiceRepr`

Slide a window of a certain shape over an array with a larger shape.

This can be used for iterating over a grid of sub-regions of 2d-images, 3d-volumes, or any n-dimensional array.

Yields slices of shape *window* that can be used to index into an array with shape *shape* via `numpy / torch fancy indexing`. This allows for fast fast iteration over subregions of a larger image. Because we generate a grid-basis using only shapes, the larger image does not need to be in memory as long as its width/height/depth/etc...

Parameters

- **shape** (*Tuple[int, ...]*) – shape of source array to slide across.
- **window** (*Tuple[int, ...]*) – shape of window that will be slid over the larger image.
- **overlap** (*float, default=0*) – a number between 0 and 1 indicating the fraction of overlap that parts will have. Specifying this is mutually exclusive with *stride*. Must be $0 \leq \text{overlap} < 1$.
- **stride** (*int, default=None*) – the number of cells (pixels) moved on each step of the window. Mutually exclusive with *overlap*.
- **keepbound** (*bool, default=False*) – if True, a non-uniform stride will be taken to ensure that the right / bottom of the image is returned as a slice if needed. Such a slice will not obey the overlap constraints. (Defaults to False)
- **allow_overshoot** (*bool, default=False*) – if False, we will raise an error if the window doesn't slide perfectly over the input shape.

Variables

- **strides** (*basis_shape - shape of the grid corresponding to the number of*) – the sliding window will take.
- **dimension** (*basis_slices - slices that will be taken in every*) –

Yields

Tuple[slice, ...] –

slices used for numpy indexing, the number of slices
in the tuple

Note: For each dimension, we generate a basis (which defines a grid), and we slide over that basis.

Todo:

- [] have an option that is allowed to go outside of the window bounds on the right and bottom when the slider overshoots.
-

Example

```
>>> from kwarray.util_slider import * # NOQA
>>> shape = (10, 10)
>>> window = (5, 5)
>>> self = SlidingWindow(shape, window)
>>> for i, index in enumerate(self):
>>>     print('i={}, index={}'.format(i, index))
i=0, index=(slice(0, 5, None), slice(0, 5, None))
i=1, index=(slice(0, 5, None), slice(5, 10, None))
i=2, index=(slice(5, 10, None), slice(0, 5, None))
i=3, index=(slice(5, 10, None), slice(5, 10, None))
```

Example

```
>>> from kwarray.util_slider import * # NOQA
>>> shape = (16, 16)
>>> window = (4, 4)
>>> self = SlidingWindow(shape, window, overlap=(.5, .25))
>>> print('self.stride = {!r}'.format(self.stride))
self.stride = [2, 3]
>>> list(ub.chunks(self.grid, 5))
[[ (0, 0), (0, 1), (0, 2), (0, 3), (0, 4)],
  [(1, 0), (1, 1), (1, 2), (1, 3), (1, 4)],
  [(2, 0), (2, 1), (2, 2), (2, 3), (2, 4)],
  [(3, 0), (3, 1), (3, 2), (3, 3), (3, 4)],
  [(4, 0), (4, 1), (4, 2), (4, 3), (4, 4)],
  [(5, 0), (5, 1), (5, 2), (5, 3), (5, 4)],
  [(6, 0), (6, 1), (6, 2), (6, 3), (6, 4)]]
```


Example

```
>>> # Test shapes that dont fit
>>> # When the window is bigger than the shape, the left-aligned slices
>>> # are returned.
>>> self = SlidingWindow((3, 3), (12, 12), allow_overshoot=True, keepbound=True)
>>> print(list(self))
[(slice(0, 12, None), slice(0, 12, None))]
>>> print(list(SlidingWindow((3, 3), None, allow_overshoot=True, keepbound=True)))
[(slice(0, 3, None), slice(0, 3, None))]
>>> print(list(SlidingWindow((3, 3), (None, 2), allow_overshoot=True,
↪ keepbound=True)))
[(slice(0, 3, None), slice(0, 2, None)), (slice(0, 3, None), slice(1, 3, None))]
```

_compute_stride(overlap, stride, shape, window)

Ensures that stride has overlap the correct shape. If stride is not provided, compute stride from desired overlap.

_iter_basis_frac()

property grid

Generate indices into the “basis” slice for each dimension. This enumerates the nd indices of the grid.

Yields

`Tuple[int, ...]`

property slices

Generate slices for each window (equivalent to `iter(self)`)

Example

```
>>> shape = (220, 220)
>>> window = (10, 10)
>>> self = SlidingWindow(shape, window, stride=5)
>>> list(self)[41:45]
[(slice(0, 10, None), slice(205, 215, None)),
 (slice(0, 10, None), slice(210, 220, None)),
 (slice(5, 15, None), slice(0, 10, None)),
 (slice(5, 15, None), slice(5, 15, None))]
>>> print('self.overlap = {!r}'.format(self.overlap))
self.overlap = [0.5, 0.5]
```

property centers

Generate centers of each window

Yields

`Tuple[float, ...]` – the center coordinate of the slice

Example

```
>>> shape = (4, 4)
>>> window = (3, 3)
>>> self = SlidingWindow(shape, window, stride=1)
>>> list(zip(self.centers, self.slices))
[((1.0, 1.0), (slice(0, 3, None), slice(0, 3, None))),
 ((1.0, 2.0), (slice(0, 3, None), slice(1, 4, None))),
 ((2.0, 1.0), (slice(1, 4, None), slice(0, 3, None))),
 ((2.0, 2.0), (slice(1, 4, None), slice(1, 4, None)))]
>>> shape = (3, 3)
>>> window = (2, 2)
>>> self = SlidingWindow(shape, window, stride=1)
>>> list(zip(self.centers, self.slices))
[((0.5, 0.5), (slice(0, 2, None), slice(0, 2, None))),
 ((0.5, 1.5), (slice(0, 2, None), slice(1, 3, None))),
 ((1.5, 0.5), (slice(1, 3, None), slice(0, 2, None))),
 ((1.5, 1.5), (slice(1, 3, None), slice(1, 3, None)))]
```

class kwarray.util_slider.Stitcher(shape, device='numpy', dtype='float32', nan_policy='propagate')

Bases: [NiceRepr](#)

Stitches multiple possibly overlapping slices into a larger array.

This is used to invert the SlidingWindow. For semenatic segmentation the patches are probability chips. Overlapping chips are averaged together.

SeeAlso:

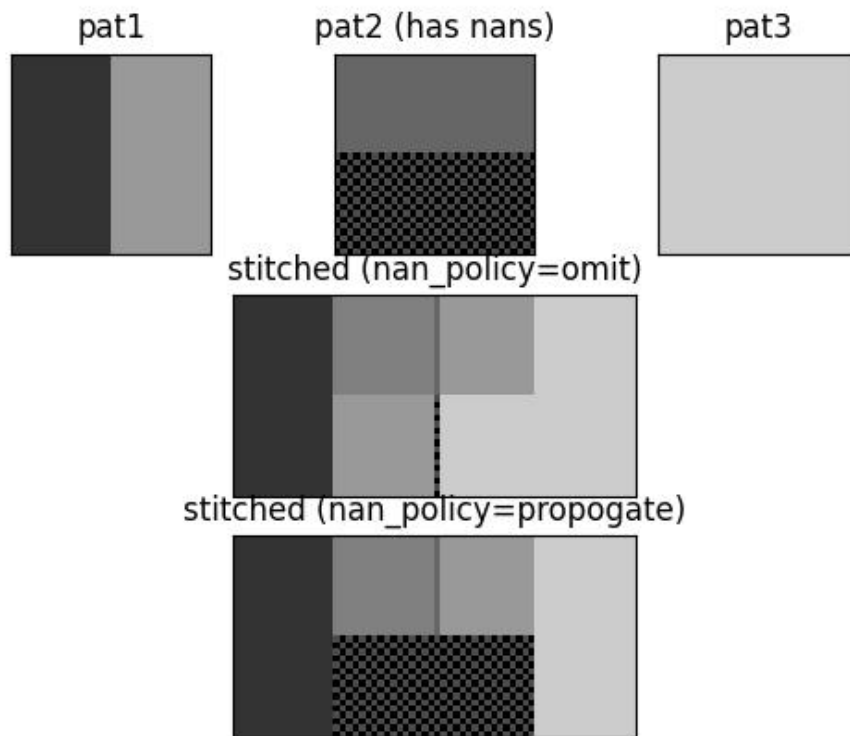
[*kwarray.RunningStats*](#) - similarly performs running means, but can also track other statistics.

Example

```
>>> from kwarray.util_slider import * # NOQA
>>> import sys
>>> # Build a high resolution image and slice it into chips
>>> highres = np.random.rand(5, 200, 200).astype(np.float32)
>>> target_shape = (1, 50, 50)
>>> slider = SlidingWindow(highres.shape, target_shape, overlap=(0, .5, .5))
>>> # Show how Sticher can be used to reconstruct the original image
>>> sticher = Stitcher(slider.input_shape)
>>> for sl in list(slider):
...     chip = highres[sl]
...     sticher.add(sl, chip)
>>> assert sticher.weights.max() == 4, 'some parts should be processed 4 times'
>>> recon = sticher.finalize()
```

Example

```
>>> from kwarray.util_slider import * # NOQA
>>> import sys
>>> # Demo stitching 3 patterns where one has nans
>>> pat1 = np.full((32, 32), fill_value=0.2)
>>> pat2 = np.full((32, 32), fill_value=0.4)
>>> pat3 = np.full((32, 32), fill_value=0.8)
>>> pat1[:, 16:] = 0.6
>>> pat2[16:, :] = np.nan
>>> # Test with nan_policy=omit
>>> stitcher = Stitcher(shape=(32, 64), nan_policy='omit')
>>> stitcher[0:32, 0:32](pat1)
>>> stitcher[0:32, 16:48](pat2)
>>> stitcher[0:32, 33:64](pat3[:, 1:])
>>> final1 = stitcher.finalize()
>>> # Test without nan_policy=propagate
>>> stitcher = Stitcher(shape=(32, 64), nan_policy='propagate')
>>> stitcher[0:32, 0:32](pat1)
>>> stitcher[0:32, 16:48](pat2)
>>> stitcher[0:32, 33:64](pat3[:, 1:])
>>> final2 = stitcher.finalize()
>>> # Checks
>>> assert np.isnan(final1).sum() == 16, 'only should contain nan where no data was_
↳ stitched'
>>> assert np.isnan(final2).sum() == 512, 'should contain nan wherever a nan was_
↳ stitched'
>>> # xdoctest: +REQUIRES(--show)
>>> # xdoctest: +REQUIRES(module:kwplot)
>>> import kwplot
>>> import kwimage
>>> kwplot.autompl()
>>> kwplot.imshow(pat1, title='pat1', pnum=(3, 3, 1))
>>> kwplot.imshow(kwimage.nodata_checkerboard(pat2, square_shape=1), title='pat2_
↳ (has nans)', pnum=(3, 3, 2))
>>> kwplot.imshow(pat3, title='pat3', pnum=(3, 3, 3))
>>> kwplot.imshow(kwimage.nodata_checkerboard(final1, square_shape=1), title=
↳ 'stitched (nan_policy=omit)', pnum=(3, 1, 2))
>>> kwplot.imshow(kwimage.nodata_checkerboard(final2, square_shape=1), title=
↳ 'stitched (nan_policy=propagate)', pnum=(3, 1, 3))
```



Example

```
>>> # Example of weighted stitching
>>> # xdoctest: +REQUIRES(module:kwimage)
>>> from kwarray.util_slider import * # NOQA
>>> import kwimage
>>> import kwarray
>>> import sys
>>> data = kwimage.Mask.demo().data.astype(np.float32)
>>> data_dims = data.shape
>>> window_dims = (8, 8)
>>> # We are going to slide a window over the data, do some processing
>>> # and then stitch it all back together. There are a few ways we
>>> # can do it. Lets demo the params.
>>> basis = {
>>>     # Vary the overlap of the slider
>>>     'overlap': (0, 0.5, .9),
>>>     # Vary if we are using weighted stitching or not
>>>     'weighted': ['none', 'gauss'],
>>>     'keepbound': [True, False]
>>> }
>>> results = []
>>> gauss_weights = kwimage.gaussian_patch(window_dims)
```

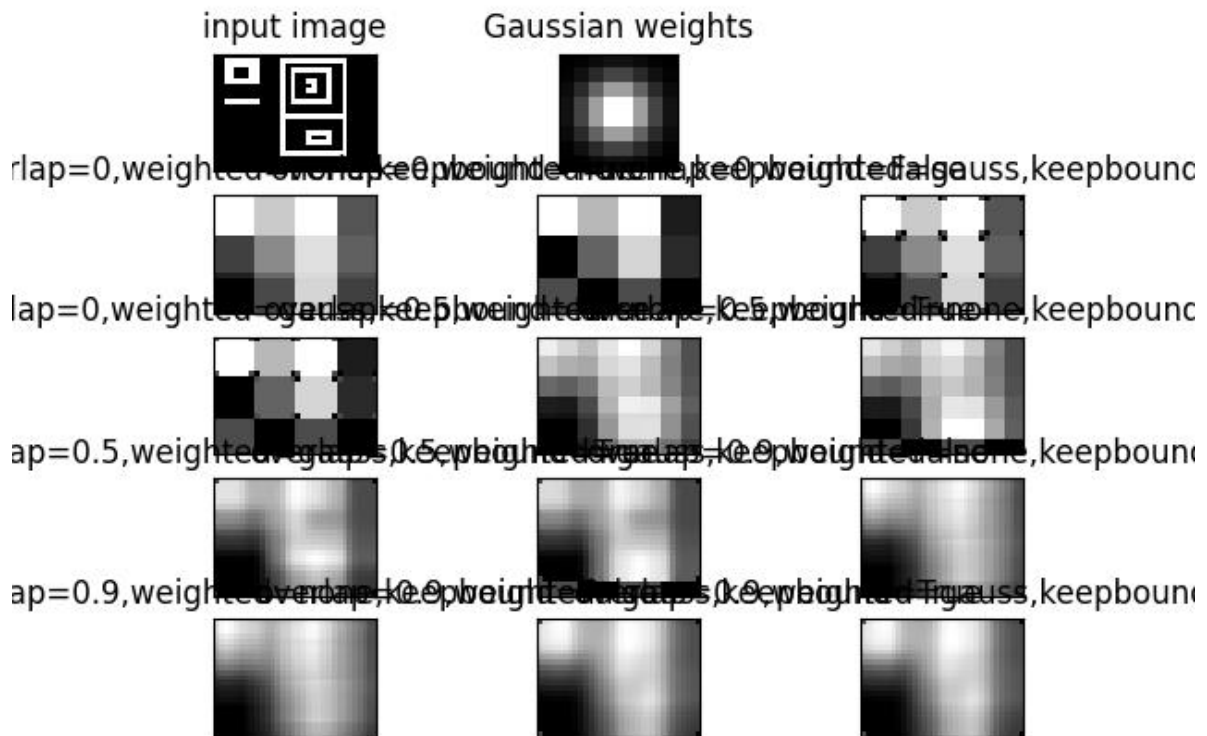
(continues on next page)

(continued from previous page)

```

>>> gauss_weights = kwimage.normalize(gauss_weights)
>>> for params in ub.named_product(basis):
>>>     if params['weighted'] == 'none':
>>>         weights = None
>>>     elif params['weighted'] == 'gauss':
>>>         weights = gauss_weights
>>>     # Build the slider and stitcher
>>>     slider = kwarray.SlidingWindow(
>>>         data_dims, window_dims, overlap=params['overlap'],
>>>         allow_overshoot=True,
>>>         keepbound=params['keepbound'])
>>>     stitcher = kwarray.Stitcher(data_dims)
>>>     # Loop over the regions
>>>     for sl in list(slider):
>>>         chip = data[sl]
>>>         # This is our dummy function for thie example.
>>>         predicted = np.ones_like(chip) * chip.sum() / chip.size
>>>         stitcher.add(sl, predicted, weight=weights)
>>>     final = stitcher.finalize()
>>>     results.append({
>>>         'final': final,
>>>         'params': params,
>>>     })
>>> # xdoctest: +REQUIRES(--show)
>>> # xdoctest: +REQUIRES(module:kwplot)
>>> import kwplot
>>> kwplot.autompl()
>>> pnum_ = kwplot.PlotNums(nCols=3, nSubplots=len(results) + 2)
>>> kwplot.imshow(data, pnum=pnum_(), title='input image')
>>> kwplot.imshow(gauss_weights, pnum=pnum_(), title='Gaussian weights')
>>> pnum_()
>>> for result in results:
>>>     param_key = ub.urepr(result['params'], compact=1)
>>>     final = result['final']
>>>     canvas = kwarray.normalize(final)
>>>     canvas = kwimage.fill_nans_with_checkers(canvas)
>>>     kwplot.imshow(canvas, pnum=pnum_(), title=param_key)

```



Parameters

- **shape** (*tuple*) – dimensions of the large image that will be created from the smaller pixels or patches.
- **device** (*str* | *int* | *torch.device*) – default is ‘numpy’, but if given as a torch device, then underlying operations will be done with torch tensors instead.
- **dtype** (*str*) – the datatype to use in the underlying accumulator.
- **nan_policy** (*str*) – if omit, check for nans and convert any to zero weight items in stitching.

add(*indices*, *patch*, *weight=None*)

Incorporate a new (possibly overlapping) patch or pixel using a weighted sum.

Parameters

- **indices** (*slice* | *tuple* | *None*) – typically a `Tuple[slice]` of pixels or a single pixel, but this can be any numpy fancy index.
- **patch** (*ndarray*) – data to patch into the bigger image.
- **weight** (*float* | *ndarray*) – weight of this patch (default to 1.0)

average()

Averages out contributions from overlapping adds using weighted average

Returns

out - the stitched image

Return type

ndarray

finalize(*indices=None*)

Averages out contributions from overlapping adds

Parameters**indices** (*None* | *slice* | *tuple*) – if *None*, finalize the entire block, otherwise only finalize a subregion.**Returns**

final - the stitched image

Return type

ndarray

kwarray.util_slider._slices1d(*margin, stop, step=None, start=0, keepbound=False, check=True*)

Helper to generates slices in a single dimension.

Parameters

- **margin** (*int*) – the length of the slice (window)
- **stop** (*int*) – the length of the image dimension
- **step** (*int, default=None*) – the length of each step / distance between slices
- **start** (*int, default=0*) – starting point (in most cases set this to 0)
- **keepbound** (*bool*) – if *True*, a non-uniform step will be taken to ensure that the right / bottom of the image is returned as a slice if needed. Such a slice will not obey the overlap constraints. (Defaults to *False*)
- **check** (*bool*) – if *True* an error will be raised if the window does not cover the entire extent from start to stop, even if *keepbound* is *True*.

Yields*slice* – slice in one dimension of size (*margin*)**Example**

```
>>> stop, margin, step = 2000, 360, 360
>>> keepbound = True
>>> strides = list(_slices1d(margin, stop, step, keepbound, check=False))
>>> assert all([(s.stop - s.start) == margin for s in strides])
```

Example

```
>>> stop, margin, step = 200, 46, 7
>>> keepbound = True
>>> strides = list(_slices1d(margin, stop, step, keepbound=False, check=True))
>>> starts = np.array([s.start for s in strides])
>>> stops = np.array([s.stop for s in strides])
>>> widths = stops - starts
>>> assert np.all(np.diff(starts) == step)
>>> assert np.all(widths == margin)
```

Example

```
>>> import pytest
>>> stop, margin, step = 200, 36, 7
>>> with pytest.raises(ValueError):
...     list(_slices1d(margin, stop, step))
```

karray.util_torch module

Torch specific extensions

`karray.util_torch._is_in_onnx_export()`

`karray.util_torch.one_hot_embedding(labels, num_classes, dim=1)`

Embedding labels to one-hot form.

Parameters

- **labels** – (LongTensor) class labels, sized [N,].
- **num_classes** – (int) number of classes.
- **dim** (*int*) – dimension which will be created, if negative

Returns

encoded labels, sized [N,#classes].

Return type

Tensor

References

<https://discuss.pytorch.org/t/convert-int-into-one-hot-format/507/4>

Example

```
>>> # each element in target has to have 0 <= value < C
>>> # xdoctest: +REQUIRES(module:torch)
>>> import torch
>>> labels = torch.LongTensor([0, 0, 1, 4, 2, 3])
>>> num_classes = max(labels) + 1
>>> t = one_hot_embedding(labels, num_classes)
>>> assert all(row[y] == 1 for row, y in zip(t.numpy(), labels.numpy()))
>>> import ubelt as ub
>>> print(ub.urepr(t.numpy().tolist()))
[
    [1.0, 0.0, 0.0, 0.0, 0.0],
    [1.0, 0.0, 0.0, 0.0, 0.0],
    [0.0, 1.0, 0.0, 0.0, 0.0],
    [0.0, 0.0, 0.0, 0.0, 1.0],
    [0.0, 0.0, 1.0, 0.0, 0.0],
    [0.0, 0.0, 0.0, 1.0, 0.0],
]
```

(continues on next page)

(continued from previous page)

```

>>> t2 = one_hot_embedding(labels.numpy(), num_classes)
>>> assert np.all(t2 == t.numpy())
>>> from kwarray.util_torch import _torch_available_devices
>>> devices = _torch_available_devices()
>>> if devices:
>>>     device = devices[0]
>>>     try:
>>>         t3 = one_hot_embedding(labels.to(device), num_classes)
>>>     except RuntimeError:
>>>         pass
>>>     assert np.all(t3.cpu().numpy() == t.numpy())

```

Example

```

>>> # xdoctest: +REQUIRES(module:torch)
>>> import torch
>>> nC = num_classes = 3
>>> labels = (torch.rand(10, 11, 12) * nC).long()
>>> assert one_hot_embedding(labels, nC, dim=0).shape == (3, 10, 11, 12)
>>> assert one_hot_embedding(labels, nC, dim=1).shape == (10, 3, 11, 12)
>>> assert one_hot_embedding(labels, nC, dim=2).shape == (10, 11, 3, 12)
>>> assert one_hot_embedding(labels, nC, dim=3).shape == (10, 11, 12, 3)
>>> labels = (torch.rand(10, 11) * nC).long()
>>> assert one_hot_embedding(labels, nC, dim=0).shape == (3, 10, 11)
>>> assert one_hot_embedding(labels, nC, dim=1).shape == (10, 3, 11)
>>> labels = (torch.rand(10) * nC).long()
>>> assert one_hot_embedding(labels, nC, dim=0).shape == (3, 10)
>>> assert one_hot_embedding(labels, nC, dim=1).shape == (10, 3)

```

`kwarray.util_torch.one_hot_lookup(data, indices)`

Return value of a particular column for each row in data.

Each item in labels corresponds to a row in data. Returns the index specified at each row.

Parameters

- **data** (*ArrayLike*) – N x C float array of values
- **indices** (*ArrayLike*) – N integer array between 0 and C. This is a column index for each row in data.

Returns

the selected probability for each row

Return type

ArrayLike

Note: This is functionally equivalent to `[row[c] for row, c in zip(data, indices)]` except that it works with pure matrix operations.

Todo:

- [] Allow the user to specify which dimension indices should be zipped over. By default it should be dim=0
 - [] Allow the user to specify which dimension indices should select from. By default it should be dim=1.
-

Example

```
>>> from kwarray.util_torch import * # NOQA
>>> data = np.array([
>>>     [0, 1, 2],
>>>     [3, 4, 5],
>>>     [6, 7, 8],
>>>     [9, 10, 11],
>>> ])
>>> indices = np.array([0, 1, 2, 1])
>>> res = one_hot_lookup(data, indices)
>>> print('res = {!r}'.format(res))
res = array([ 0,  4,  8, 10])
>>> alt = np.array([row[c] for row, c in zip(data, indices)])
>>> assert np.all(alt == res)
```

Example

```
>>> # xdoctest: +REQUIRES(module:torch)
>>> import torch
>>> data = torch.from_numpy(np.array([
>>>     [0, 1, 2],
>>>     [3, 4, 5],
>>>     [6, 7, 8],
>>>     [9, 10, 11],
>>> ]))
>>> indices = torch.from_numpy(np.array([0, 1, 2, 1])).long()
>>> res = one_hot_lookup(data, indices)
>>> print('res = {!r}'.format(res))
res = tensor([ 0,  4,  8, 10]...)
>>> alt = torch.LongTensor([row[c] for row, c in zip(data, indices)])
>>> assert torch.all(alt == res)
```

`kwarray.util_torch._torch_available_devices()`

An attempt to determine what devices this version of torch can use

Try and check that cuda is available AND we have a good kernel image

1.1.2 Module contents

The `kwarray` module implements a small set of pure-python extensions to `numpy` and `torch` along with a few select algorithms. Each module contains module level docstring that gives a rough idea of the utilities in each module, and each function or class itself contains a docstring with more details and examples.

KWarray is part of Kitware's computer vision Python suite:

<https://gitlab.kitware.com/computer-vision>

class `kwarray.ArrayAPI`

Bases: `object`

Compatibility API between `torch` and `numpy`.

The API defines classmethods that work on both Tensors and ndarrays. As such the user can simply use `kwarray.ArrayAPI.<funcname>` and it will return the expected result for both Tensor and ndarray types.

However, this is inefficient because it requires us to check the type of the input for every API call. Therefore it is recommended that you use the `ArrayAPI.coerce()` function, which takes as input the data you want to operate on. It performs the type check once, and then returns another object that defines with an identical API, but specific to the given data type. This means that we can ignore type checks on future calls of the specific implementation. See examples for more details.

Example

```
>>> # Use the easy-to-use, but inefficient array api
>>> # xdoctest: +REQUIRES(module:torch)
>>> import kwarray
>>> import torch
>>> take = kwarray.ArrayAPI.take
>>> np_data = np.arange(0, 143).reshape(11, 13)
>>> pt_data = torch.LongTensor(np_data)
>>> indices = [1, 3, 5, 7, 11, 13, 17, 21]
>>> idxs0 = [1, 3, 5, 7]
>>> idxs1 = [1, 3, 5, 7, 11]
>>> assert np.allclose(take(np_data, indices), take(pt_data, indices))
>>> assert np.allclose(take(np_data, idxs0, 0), take(pt_data, idxs0, 0))
>>> assert np.allclose(take(np_data, idxs1, 1), take(pt_data, idxs1, 1))
```

Example

```
>>> # Use the easy-to-use, but inefficient array api
>>> # xdoctest: +REQUIRES(module:torch)
>>> import kwarray
>>> import torch
>>> compress = kwarray.ArrayAPI.compress
>>> np_data = np.arange(0, 143).reshape(11, 13)
>>> pt_data = torch.LongTensor(np_data)
>>> flags = (np_data % 2 == 0).ravel()
>>> f0 = (np_data % 2 == 0)[: , 0]
>>> f1 = (np_data % 2 == 0)[0, :]
>>> assert np.allclose(compress(np_data, flags), compress(pt_data, flags))
```

(continues on next page)

(continued from previous page)

```
>>> assert np.allclose(compress(np_data, f0, 0), compress(pt_data, f0, 0))
>>> assert np.allclose(compress(np_data, f1, 1), compress(pt_data, f1, 1))
```

Example

```
>>> # Use ArrayAPI to coerce an identical API that doesnt do type checks
>>> # xdoctest: +REQUIRES(module:torch)
>>> import kwarray
>>> import torch
>>> np_data = np.arange(0, 15).reshape(3, 5)
>>> pt_data = torch.LongTensor(np_data)
>>> # The new ``impl`` object has the same API as ArrayAPI, but works
>>> # specifically on torch Tensors.
>>> impl = kwarray.ArrayAPI.coerce(pt_data)
>>> flat_data = impl.view(pt_data, -1)
>>> print('flat_data = {!r}'.format(flat_data))
flat_data = tensor([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
>>> # The new ``impl`` object has the same API as ArrayAPI, but works
>>> # specifically on numpy ndarrays.
>>> impl = kwarray.ArrayAPI.coerce(np_data)
>>> flat_data = impl.view(np_data, -1)
>>> print('flat_data = {!r}'.format(flat_data))
flat_data = array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

_torch

alias of *TorchImpls*

_numpy

alias of *NumpyImpls*

static impl(data)

Returns a namespace suitable for operating on the input data type

Parameters

data (*ndarray* | *Tensor*) – data to be operated on

static coerce(data)

Coerces some form of inputs into an array api (either numpy or torch).

static result_type(*arrays_and_dtypes)

Return type from promotion rules

Example

```
>>> import kwarray
>>> kwarray.ArrayAPI.result_type(float, np.uint8, np.float32, np.float16)
>>> # xdoctest: +REQUIRES(module:torch)
>>> import torch
>>> kwarray.ArrayAPI.result_type(torch.int32, torch.int64)
```

static cat(datas, *args, **kwargs)

```
static hstack(datas, *args, **kwargs)
static vstack(datas, *args, **kwargs)
static take(data, *args, **kwargs)
static compress(data, *args, **kwargs)
static repeat(data, *args, **kwargs)
static tile(data, *args, **kwargs)
static view(data, *args, **kwargs)
static numel(data, *args, **kwargs)
static atleast_nd(data, *args, **kwargs)
static full_like(data, *args, **kwargs)
static ones_like(data, *args, **kwargs)
static zeros_like(data, *args, **kwargs)
static empty_like(data, *args, **kwargs)
static sum(data, *args, **kwargs)
static argsort(data, *args, **kwargs)
static argmax(data, *args, **kwargs)
static argmin(data, *args, **kwargs)
static max(data, *args, **kwargs)
static min(data, *args, **kwargs)
static max_argmax(data, *args, **kwargs)
static min_argmin(data, *args, **kwargs)
static maximum(data, *args, **kwargs)
static minimum(data, *args, **kwargs)
static matmul(data, *args, **kwargs)
static astype(data, *args, **kwargs)
static nonzero(data, *args, **kwargs)
static nan_to_num(data, *args, **kwargs)
static tensor(data, *args, **kwargs)
static numpy(data, *args, **kwargs)
static tolist(data, *args, **kwargs)
static asarray(data, *args, **kwargs)
```

```

static T(data, *args, **kwargs)

static transpose(data, *args, **kwargs)

static contiguous(data, *args, **kwargs)

static pad(data, *args, **kwargs)

static dtype_kind(data, *args, **kwargs)

static any(data, *args, **kwargs)

static all(data, *args, **kwargs)

static array_equal(data, *args, **kwargs)

static log2(data, *args, **kwargs)

static log(data, *args, **kwargs)

static copy(data, *args, **kwargs)

static ceil(data, *args, **kwargs)

static ifloor(data, *args, **kwargs)

static floor(data, *args, **kwargs)

static ceil(data, *args, **kwargs)

static round(data, *args, **kwargs)

static iround(data, *args, **kwargs)

static clip(data, *args, **kwargs)

static softmax(data, *args, **kwargs)

```

```
class kwarray.DataFrameArray(data=None, columns=None)
```

Bases: [DataFrameLight](#)

DataFrameLight assumes the backend is a Dict[list] DataFrameArray assumes the backend is a Dict[ndarray]

Take and compress are much faster, but extend and union are slower

```
extend(other)
```

```
compress(flags, inplace=False)
```

```
take(indices, inplace=False)
```

```
class kwarray.DataFrameLight(data=None, columns=None)
```

Bases: [NiceRepr](#)

Implements a subset of the pandas.DataFrame API

The API is restricted to facilitate speed tradeoffs

Note: Assumes underlying data is Dict[list|ndarray]. If the data is known to be a Dict[ndarray] use DataFrameArray instead, which has faster implementations for some operations.

Note: pandas.DataFrame is slow. DataFrameLight is faster. It is a tad more restrictive though.

Example

```
>>> self = DataFrameLight({})
>>> print('self = {!r}'.format(self))
>>> self = DataFrameLight({'a': [0, 1, 2], 'b': [2, 3, 4]})
>>> print('self = {!r}'.format(self))
>>> item = self.iloc[0]
>>> print('item = {!r}'.format(item))
```

Benchmark

```
>>> # BENCHMARK
>>> # xdoc: +REQUIRES(--bench)
>>> from kwarray.dataframe_light import * # NOQA
>>> import ubelt as ub
>>> NUM = 1000
>>> print('NUM = {!r}'.format(NUM))
>>> # to_dict conversions
>>> print('=====')
>>> print('===== to_dict conversions =====')
>>> _keys = ['list', 'dict', 'series', 'split', 'records', 'index']
>>> results = []
>>> df = DataFrameLight._demodata(num=NUM).pandas()
>>> ti = ub.Timerit(verbose=False, unit='ms')
>>> for key in _keys:
>>>     result = ti.reset(key).call(lambda: df.to_dict(orient=key))
>>>     results.append((result.mean(), result.report()))
>>> key = 'series+numpy'
>>> result = ti.reset(key).call(lambda: {k: v.values for k, v in df.to_dict(orient=
↳ 'series').items()})
>>> results.append((result.mean(), result.report()))
>>> print('\n'.join([t[1] for t in sorted(results)]))
>>> print('=====')
>>> print('===== DFLight Conversions =====')
>>> ti = ub.Timerit(verbose=True, unit='ms')
>>> key = 'self.pandas'
>>> self = DataFrameLight(df)
>>> ti.reset(key).call(lambda: self.pandas())
>>> key = 'light-from-pandas'
>>> ti.reset(key).call(lambda: DataFrameLight(df))
>>> key = 'light-from-dict'
>>> ti.reset(key).call(lambda: DataFrameLight(self._data))
>>> print('=====')
>>> print('===== BENCHMARK: .LOC[] =====')
>>> ti = ub.Timerit(num=20, bestof=4, verbose=True, unit='ms')
>>> df_light = DataFrameLight._demodata(num=NUM)
>>> # xdoctest: +REQUIRES(module:pandas)
```

(continues on next page)

(continued from previous page)

```
>>> df_heavy = df_light.pandas()
>>> series_data = df_heavy.to_dict(orient='series')
>>> list_data = df_heavy.to_dict(orient='list')
>>> np_data = {k: v.values for k, v in df_heavy.to_dict(orient='series').items()}
>>> for timer in ti.reset('DF-heavy.iloc'):
>>>     with timer:
>>>         for i in range(NUM):
>>>             df_heavy.iloc[i]
>>> for timer in ti.reset('DF-heavy.loc'):
>>>     with timer:
>>>         for i in range(NUM):
>>>             df_heavy.iloc[i]
>>> for timer in ti.reset('dict[SERIES].loc'):
>>>     with timer:
>>>         for i in range(NUM):
>>>             {key: series_data[key].loc[i] for key in series_data.keys()}
>>> for timer in ti.reset('dict[SERIES].iloc'):
>>>     with timer:
>>>         for i in range(NUM):
>>>             {key: series_data[key].iloc[i] for key in series_data.keys()}
>>> for timer in ti.reset('dict[SERIES][]'):
>>>     with timer:
>>>         for i in range(NUM):
>>>             {key: series_data[key][i] for key in series_data.keys()}
>>> for timer in ti.reset('dict[NDARRAY][]'):
>>>     with timer:
>>>         for i in range(NUM):
>>>             {key: np_data[key][i] for key in np_data.keys()}
>>> for timer in ti.reset('dict[list][]'):
>>>     with timer:
>>>         for i in range(NUM):
>>>             {key: list_data[key][i] for key in np_data.keys()}
>>> for timer in ti.reset('DF-Light.iloc/loc'):
>>>     with timer:
>>>         for i in range(NUM):
>>>             df_light.iloc[i]
>>> for timer in ti.reset('DF-Light._getrow'):
>>>     with timer:
>>>         for i in range(NUM):
>>>             df_light._getrow(i)
NUM = 1000
=====
===== to_dict conversions =====
Timed best=0.022 ms, mean=0.022 ± 0.0 ms for series
Timed best=0.059 ms, mean=0.059 ± 0.0 ms for series+numpy
Timed best=0.315 ms, mean=0.315 ± 0.0 ms for list
Timed best=0.895 ms, mean=0.895 ± 0.0 ms for dict
Timed best=2.705 ms, mean=2.705 ± 0.0 ms for split
Timed best=5.474 ms, mean=5.474 ± 0.0 ms for records
Timed best=7.320 ms, mean=7.320 ± 0.0 ms for index
=====
===== DFLight Conversions =====
```

(continues on next page)

(continued from previous page)

```

Timed best=1.798 ms, mean=1.798 ± 0.0 ms for self.pandas
Timed best=0.064 ms, mean=0.064 ± 0.0 ms for light-from-pandas
Timed best=0.010 ms, mean=0.010 ± 0.0 ms for light-from-dict
=====
===== BENCHMARK: .LOC[] =====
Timed best=101.365 ms, mean=101.564 ± 0.2 ms for DF-heavy.iloc
Timed best=102.038 ms, mean=102.273 ± 0.2 ms for DF-heavy.loc
Timed best=29.357 ms, mean=29.449 ± 0.1 ms for dict[SERIES].loc
Timed best=21.701 ms, mean=22.014 ± 0.3 ms for dict[SERIES].iloc
Timed best=11.469 ms, mean=11.566 ± 0.1 ms for dict[SERIES][]
Timed best=0.807 ms, mean=0.826 ± 0.0 ms for dict[NDARRAY][]
Timed best=0.478 ms, mean=0.492 ± 0.0 ms for dict[list][]
Timed best=0.969 ms, mean=0.994 ± 0.0 ms for DF-Light.iloc/loc
Timed best=0.760 ms, mean=0.776 ± 0.0 ms for DF-Light._getrow

```

property `iloc`**property** `values`**property** `loc`**to_string**(*args, **kwargs)**Return type**`str`**to_dict**(orient='dict', into=<class 'dict'>)

Convert the data frame into a dictionary.

Parameters

- **orient** (*str*) – Currently naively supports orient in { 'dict', 'list' }, otherwise we fallback to pandas conversion and call its to_dict method.
- **into** (*type*) – type of dictionary to transform into

Returns`dict`

Example

```

>>> from kwarray.dataframe_light import * # NOQA
>>> self = DataFrameLight._demodata(num=7)
>>> print(self.to_dict(orient='dict'))
>>> print(self.to_dict(orient='list'))

```

pandas()

Convert back to pandas if you need the full API

Return type`pd.DataFrame`

Example

```
>>> # xdoctest: +REQUIRES(module:pandas)
>>> df_light = DataFrameLight._demodata(num=7)
>>> df_heavy = df_light.pandas()
>>> got = DataFrameLight(df_heavy)
>>> assert got._data == df_light._data
```

`_pandas()`

Deprecated, use `self.pandas` instead

classmethod `_demodata(num=7)`

Example

```
>>> self = DataFrameLight._demodata(num=7)
>>> print('self = {!r}'.format(self))
>>> other = DataFrameLight._demodata(num=11)
>>> print('other = {!r}'.format(other))
>>> both = self.union(other)
>>> print('both = {!r}'.format(both))
>>> assert both is not self
>>> assert other is not self
```

property `columns`

`sort_values(key, inplace=False, ascending=True)`

`keys()`

`_getrow(index)`

`_getcol(key)`

`_getcols(keys)`

`get(key, default=None)`

Get item for given key. Returns default value if not found.

`clear()`

Removes all rows inplace

`compress(flags, inplace=False)`

NOTE: NOT A PART OF THE PANDAS API

`take(indices, inplace=False)`

Return the elements in the given *positional* indices along an axis.

Parameters

`inplace` (*bool*) – NOT PART OF PANDAS API

Note: assumes axis=0

Example

```
>>> df_light = DataFrameLight._demodata(num=7)
>>> indices = [0, 2, 3]
>>> sub1 = df_light.take(indices)
>>> # xdoctest: +REQUIRES(module:pandas)
>>> df_heavy = df_light.pandas()
>>> sub2 = df_heavy.take(indices)
>>> assert np.all(sub1 == sub2)
```

copy()

extend(*other*)

Extend self inplace using another dataframe array

Parameters

other (*DataFrameLight* | *dict[str, Sequence]*) – values to concat to end of this object

Note: Not part of the pandas API

Example

```
>>> self = DataFrameLight(columns=['foo', 'bar'])
>>> other = {'foo': [0], 'bar': [1]}
>>> self.extend(other)
>>> assert len(self) == 1
```

union(others*)**

Note: Note part of the pandas API

classmethod concat(*others*)

classmethod from_pandas(*df*)

classmethod from_dict(*records*)

reset_index(*drop=False*)

noop for compatability, the light version doesnt store an index

groupby(*by=None, *args, **kwargs*)

Group rows by the value of a column. Unlike pandas this simply returns a zip object. To ensure compatability call list on the result of groupby.

Parameters

- **by** (*str*) – column name to group by
- ***args** – if specified, the dataframe is coerced to pandas
- ***kwargs** – if specified, the dataframe is coerced to pandas

Example

```
>>> df_light = DataFrameLight._demodata(num=7)
>>> res1 = list(df_light.groupby('bar'))
>>> # xdoctest: +REQUIRES(module:pandas)
>>> df_heavy = df_light.pandas()
>>> res2 = list(df_heavy.groupby('bar'))
>>> assert len(res1) == len(res2)
>>> assert all([np.all(a[1] == b[1]) for a, b in zip(res1, res2)])
```

rename(mapper=None, columns=None, axis=None, inplace=False)

Rename the columns (index renaming is not supported)

Example

```
>>> df_light = DataFrameLight._demodata(num=7)
>>> mapper = {'foo': 'fi'}
>>> res1 = df_light.rename(columns=mapper)
>>> res3 = df_light.rename(mapper, axis=1)
>>> # xdoctest: +REQUIRES(module:pandas)
>>> df_heavy = df_light.pandas()
>>> res2 = df_heavy.rename(columns=mapper)
>>> res4 = df_heavy.rename(mapper, axis=1)
>>> assert np.all(res1 == res2)
>>> assert np.all(res3 == res2)
>>> assert np.all(res3 == res4)
```

iterrows()

Iterate over rows as (index, Dict) pairs.

Yields

Tuple[int, Dict] – the index and a dictionary representing a row

Example

```
>>> from kwarrray.dataframe_light import * # NOQA
>>> self = DataFrameLight._demodata(num=3)
>>> print(ub.urepr(list(self.iterrows()), sort=1))
[
  (0, {'bar': 0, 'baz': 2.73, 'foo': 0}),
  (1, {'bar': 1, 'baz': 2.73, 'foo': 0}),
  (2, {'bar': 2, 'baz': 2.73, 'foo': 0}),
]
```

Benchmark

```
>>> # xdoc: +REQUIRES(--bench)
>>> from kwarray.dataframe_light import * # NOQA
>>> import ubelt as ub
>>> df_light = DataFrameLight._demodata(num=1000)
>>> df_heavy = df_light.pandas()
>>> ti = ub.Timerit(21, bestof=3, verbose=2, unit='ms')
>>> ti.reset('light').call(lambda: list(df_light.iterrows()))
>>> ti.reset('heavy').call(lambda: list(df_heavy.iterrows()))
>>> # xdoctest: +IGNORE_WANT
Timed light for: 21 loops, best of 3
    time per loop: best=0.834 ms, mean=0.850 ± 0.0 ms
Timed heavy for: 21 loops, best of 3
    time per loop: best=45.007 ms, mean=45.633 ± 0.5 ms
```

class kwarray.**FlatIndexer**(*lens*)

Bases: `NiceRepr`

Creates a flat “view” of a jagged nested indexable object. Only supports one offset level.

Parameters

lens (*List[int]*) – a list of the lengths of the nested objects.

Doctest

```
>>> self = FlatIndexer([1, 2, 3])
>>> len(self)
>>> self.unravel(4)
>>> self.ravel(2, 1)
```

classmethod **fromlist**(*items*)

Convenience method to create a `FlatIndexer` from the list of items itself instead of the array of lengths.

Parameters

items (*List[list]*) – a list of the lists you want to flat index over

Returns

`FlatIndexer`

unravel(*index*)

Parameters

index (*int* | *List[int]*) – raveled index

Returns

outer and inner indices

Return type

`Tuple[int, int]`

Example

```
>>> import kvarray
>>> rng = kvarray.ensure_rng(0)
>>> items = [rng.rand(rng.randint(0, 10)) for _ in range(10)]
>>> self = kvarray.FlatIndexer.fromlist(items)
>>> index = np.arange(0, len(self))
>>> outer, inner = self.unravel(index)
>>> recon = self.ravel(outer, inner)
>>> # This check is only possible because index is an arange
>>> check1 = np.hstack(list(map(sorted, kvarray.group_indices(outer)[1])))
>>> check2 = np.hstack(kvarray.group_consecutive_indices(inner))
>>> assert np.all(check1 == index)
>>> assert np.all(check2 == index)
>>> assert np.all(index == recon)
```

ravel(outer, inner)

Parameters

- **outer** – index into outer list
- **inner** – index into the list referenced by outer

Returns

the raveled index

Return type

List[int]

class kvarray.LocLight(*parent*)

Bases: `object`

exception kvarray.NoSupportError

Bases: `RuntimeError`

class kvarray.RunningStats(*nan_policy='omit', check_weights=True, **kwargs*)

Bases: `NiceRepr`

Track mean, std, min, and max values over time with constant memory.

Dynamically records per-element array statistics and can summarize them per-element, across channels, or globally.

Todo:

- [] This may need a few API tweaks and good documentation
-

Example

```

>>> import kwarray
>>> run = kwarray.RunningStats()
>>> ch1 = np.array([[0, 1], [3, 4]])
>>> ch2 = np.zeros((2, 2))
>>> img = np.dstack([ch1, ch2])
>>> run.update(np.dstack([ch1, ch2]))
>>> run.update(np.dstack([ch1 + 1, ch2]))
>>> run.update(np.dstack([ch1 + 2, ch2]))
>>> # No marginalization
>>> print('current-ave = ' + ub.urepr(run.summarize(axis=ub.NoParam), nl=2,
↳precision=3))
>>> # Average over channels (keeps spatial dims separate)
>>> print('chann-ave(k=1) = ' + ub.urepr(run.summarize(axis=0), nl=2, precision=3))
>>> print('chann-ave(k=0) = ' + ub.urepr(run.summarize(axis=0, keepdims=0), nl=2,
↳precision=3))
>>> # Average over spatial dims (keeps channels separate)
>>> print('spatial-ave(k=1) = ' + ub.urepr(run.summarize(axis=(1, 2)), nl=2,
↳precision=3))
>>> print('spatial-ave(k=0) = ' + ub.urepr(run.summarize(axis=(1, 2), keepdims=0),
↳nl=2, precision=3))
>>> # Average over all dims
>>> print('alldim-ave(k=1) = ' + ub.urepr(run.summarize(axis=None), nl=2,
↳precision=3))
>>> print('alldim-ave(k=0) = ' + ub.urepr(run.summarize(axis=None, keepdims=0),
↳nl=2, precision=3))

```

Parameters

nan_policy (*str*) – indicates how we will handle nan values

- if “omit” - set weights of nan items to zero.
- if “propagate” - propagate nans.
- if “raise” - then raise a ValueError if nans are given.

check_weights (*bool*):

if True, we check the weights for zeros (which can also implicitly occur when data has nans). Disabling this check will result in faster computation, but it is your responsibility to ensure all data passed to update is valid.

property shape

_update_from_other (*other*)

Combine this runner with another one.

update_many (*data*, *weights=1*)

Assumes first data axis represents multiple observations

Example

```
>>> import kwarray
>>> rng = kwarray.ensure_rng(0)
>>> run = kwarray.RunningStats()
>>> data = rng.randn(1, 2, 3)
>>> run.update_many(data)
>>> print(run.current())
>>> data = rng.randn(2, 2, 3)
>>> run.update_many(data)
>>> print(run.current())
>>> data = rng.randn(3, 2, 3)
>>> run.update_many(data)
>>> print(run.current())
>>> run.update_many(1000)
>>> print(run.current())
>>> assert np.all(run.current()['n'] == 7)
```

Example

```
>>> import kwarray
>>> rng = kwarray.ensure_rng(0)
>>> run = kwarray.RunningStats()
>>> data = rng.randn(1, 2, 3)
>>> run.update_many(data.ravel())
>>> print(run.current())
>>> data = rng.randn(2, 2, 3)
>>> run.update_many(data.ravel())
>>> print(run.current())
>>> data = rng.randn(3, 2, 3)
>>> run.update_many(data.ravel())
>>> print(run.current())
>>> run.update_many(1000)
>>> print(run.current())
>>> assert np.all(run.current()['n'] == 37)
```

update(data, weights=1)

Updates statistics across all data dimensions on a per-element basis

Example

```
>>> import kwarray
>>> data = np.full((7, 5), fill_value=1.3)
>>> weights = np.ones((7, 5), dtype=np.float32)
>>> run = kwarray.RunningStats()
>>> run.update(data, weights=1)
>>> run.update(data, weights=weights)
>>> rng = np.random
>>> weights[rng.rand(*weights.shape) > 0.5] = 0
>>> run.update(data, weights=weights)
```


Example

```

>>> import kwarray
>>> run = kwarray.RunningStats()
>>> data = np.array([[1, np.nan, np.nan], [0, np.nan, 1.]])
>>> run.update(data)
>>> print('current = {}'.format(ub.urepr(run.current(), nl=1)))
>>> print('summary(axis=None) = {}'.format(ub.urepr(run.summarize(), nl=1)))
>>> print('summary(axis=1) = {}'.format(ub.urepr(run.summarize(axis=1), nl=1)))
>>> print('summary(axis=0) = {}'.format(ub.urepr(run.summarize(axis=0), nl=1)))
>>> data = np.array([[2, 0, 1], [0, 1, np.nan]])
>>> run.update(data)
>>> data = np.array([[3, 1, 1], [0, 1, np.nan]])
>>> run.update(data)
>>> data = np.array([[4, 1, 1], [0, 1, 1.]])
>>> run.update(data)
>>> print('----')
>>> print('current = {}'.format(ub.urepr(run.current(), nl=1)))
>>> print('summary(axis=None) = {}'.format(ub.urepr(run.summarize(), nl=1)))
>>> print('summary(axis=1) = {}'.format(ub.urepr(run.summarize(axis=1), nl=1)))
>>> print('summary(axis=0) = {}'.format(ub.urepr(run.summarize(axis=0), nl=1)))

```

_sumsq_std(total, squares, n)

Sum of squares method to compute standard deviation

summarize(axis=None, keepdims=True)

Compute summary statistics across a one or more dimension

Parameters

- **axis** (*int* | *List[int]* | *None* | *NoParamType*) – axis or axes to summarize over, if None, all axes are summarized. if *ub.NoParam*, no axes are summarized the current result is returned.
- **keepdims** (*bool*, *default=True*) – if False removes the dimensions that are summarized over

Returns

containing minimum, maximum, mean, std, etc..

Return type

Dict

Raises

NoSupportError – if update was never called with valid data

Example

```

>>> # Test to make sure summarize works across different shapes
>>> base = np.array([1, 1, 1, 1, 0, 0, 0, 1])
>>> run0 = RunningStats()
>>> for _ in range(3):
>>>     run0.update(base.reshape(8, 1))
>>> run1 = RunningStats()
>>> for _ in range(3):

```

(continues on next page)

(continued from previous page)

```

>>> run1.update(base.reshape(4, 2))
>>> run2 = RunningStats()
>>> for _ in range(3):
>>>     run2.update(base.reshape(2, 2, 2))
>>> #
>>> # Summarizing over everything should be exactly the same
>>> s0N = run0.summarize(axis=None, keepdims=0)
>>> s1N = run1.summarize(axis=None, keepdims=0)
>>> s2N = run2.summarize(axis=None, keepdims=0)
>>> #assert ub.util_indexable.indexable_allclose(s0N, s1N, rel_tol=0.0, abs_
→tol=0.0)
>>> #assert ub.util_indexable.indexable_allclose(s1N, s2N, rel_tol=0.0, abs_
→tol=0.0)
>>> assert s0N['mean'] == 0.625

```

current()

Returns current statistics on a per-element basis (not summarized over any axis)

class kwarrray.SlidingWindow(*shape*, *window*, *overlap=None*, *stride=None*, *keepbound=False*, *allow_overshoot=False*)

Bases: `NiceRepr`

Slide a window of a certain shape over an array with a larger shape.

This can be used for iterating over a grid of sub-regions of 2d-images, 3d-volumes, or any n-dimensional array.

Yields slices of shape *window* that can be used to index into an array with shape *shape* via numpy / torch fancy indexing. This allows for fast iteration over subregions of a larger image. Because we generate a grid-basis using only shapes, the larger image does not need to be in memory as long as its width/height/depth/etc...

Parameters

- **shape** (*Tuple[int, ...]*) – shape of source array to slide across.
- **window** (*Tuple[int, ...]*) – shape of window that will be slid over the larger image.
- **overlap** (*float, default=0*) – a number between 0 and 1 indicating the fraction of overlap that parts will have. Specifying this is mutually exclusive with *stride*. Must be $0 \leq \text{overlap} < 1$.
- **stride** (*int, default=None*) – the number of cells (pixels) moved on each step of the window. Mutually exclusive with *overlap*.
- **keepbound** (*bool, default=False*) – if True, a non-uniform stride will be taken to ensure that the right / bottom of the image is returned as a slice if needed. Such a slice will not obey the overlap constraints. (Defaults to False)
- **allow_overshoot** (*bool, default=False*) – if False, we will raise an error if the window doesn't slide perfectly over the input shape.

Variables

- **strides** (*basis_shape - shape of the grid corresponding to the number of*) – the sliding window will take.
- **dimension** (*basis_slices - slices that will be taken in every*) –

Yields

Tuple[slice, ...] –

slices used for numpy indexing, the number of slices
in the tuple

Note: For each dimension, we generate a basis (which defines a grid), and we slide over that basis.

Todo:

- `[]` have an option that is allowed to go outside of the window bounds on the right and bottom when the slider overshoots.
-

Example

```
>>> from kwarray.util_slider import * # NOQA
>>> shape = (10, 10)
>>> window = (5, 5)
>>> self = SlidingWindow(shape, window)
>>> for i, index in enumerate(self):
>>>     print('i={}, index={}'.format(i, index))
i=0, index=(slice(0, 5, None), slice(0, 5, None))
i=1, index=(slice(0, 5, None), slice(5, 10, None))
i=2, index=(slice(5, 10, None), slice(0, 5, None))
i=3, index=(slice(5, 10, None), slice(5, 10, None))
```

Example

```
>>> from kwarray.util_slider import * # NOQA
>>> shape = (16, 16)
>>> window = (4, 4)
>>> self = SlidingWindow(shape, window, overlap=(.5, .25))
>>> print('self.stride = {!r}'.format(self.stride))
self.stride = [2, 3]
>>> list(ub.chunks(self.grid, 5))
[[ (0, 0), (0, 1), (0, 2), (0, 3), (0, 4)],
  [(1, 0), (1, 1), (1, 2), (1, 3), (1, 4)],
  [(2, 0), (2, 1), (2, 2), (2, 3), (2, 4)],
  [(3, 0), (3, 1), (3, 2), (3, 3), (3, 4)],
  [(4, 0), (4, 1), (4, 2), (4, 3), (4, 4)],
  [(5, 0), (5, 1), (5, 2), (5, 3), (5, 4)],
  [(6, 0), (6, 1), (6, 2), (6, 3), (6, 4)]]
```

Example

```
>>> # Test shapes that dont fit
>>> # When the window is bigger than the shape, the left-aligned slices
>>> # are returned.
>>> self = SlidingWindow((3, 3), (12, 12), allow_overshoot=True, keepbound=True)
>>> print(list(self))
[(slice(0, 12, None), slice(0, 12, None))]
>>> print(list(SlidingWindow((3, 3), None, allow_overshoot=True, keepbound=True)))
[(slice(0, 3, None), slice(0, 3, None))]
>>> print(list(SlidingWindow((3, 3), (None, 2), allow_overshoot=True,
↳ keepbound=True)))
[(slice(0, 3, None), slice(0, 2, None)), (slice(0, 3, None), slice(1, 3, None))]
```

_compute_stride(overlap, stride, shape, window)

Ensures that stride has overlap the correct shape. If stride is not provided, compute stride from desired overlap.

_iter_basis_frac()

property grid

Generate indices into the “basis” slice for each dimension. This enumerates the nd indices of the grid.

Yields

`Tuple[int, ...]`

property slices

Generate slices for each window (equivalent to `iter(self)`)

Example

```
>>> shape = (220, 220)
>>> window = (10, 10)
>>> self = SlidingWindow(shape, window, stride=5)
>>> list(self)[41:45]
[(slice(0, 10, None), slice(205, 215, None)),
 (slice(0, 10, None), slice(210, 220, None)),
 (slice(5, 15, None), slice(0, 10, None)),
 (slice(5, 15, None), slice(5, 15, None))]
>>> print('self.overlap = {!r}'.format(self.overlap))
self.overlap = [0.5, 0.5]
```

property centers

Generate centers of each window

Yields

`Tuple[float, ...]` – the center coordinate of the slice

Example

```

>>> shape = (4, 4)
>>> window = (3, 3)
>>> self = SlidingWindow(shape, window, stride=1)
>>> list(zip(self.centers, self.slices))
[((1.0, 1.0), (slice(0, 3, None), slice(0, 3, None))),
 ((1.0, 2.0), (slice(0, 3, None), slice(1, 4, None))),
 ((2.0, 1.0), (slice(1, 4, None), slice(0, 3, None))),
 ((2.0, 2.0), (slice(1, 4, None), slice(1, 4, None)))]
>>> shape = (3, 3)
>>> window = (2, 2)
>>> self = SlidingWindow(shape, window, stride=1)
>>> list(zip(self.centers, self.slices))
[((0.5, 0.5), (slice(0, 2, None), slice(0, 2, None))),
 ((0.5, 1.5), (slice(0, 2, None), slice(1, 3, None))),
 ((1.5, 0.5), (slice(1, 3, None), slice(0, 2, None))),
 ((1.5, 1.5), (slice(1, 3, None), slice(1, 3, None)))]

```

class kwarray.**Stitcher**(shape, device='numpy', dtype='float32', nan_policy='propagate')

Bases: [NiceRepr](#)

Stitches multiple possibly overlapping slices into a larger array.

This is used to invert the SlidingWindow. For semenatic segmentation the patches are probability chips. Overlapping chips are averaged together.

SeeAlso:

[*kwarray.RunningStats*](#) - similarly performs running means, but can also track other statistics.

Example

```

>>> from kwarray.util_slider import * # NOQA
>>> import sys
>>> # Build a high resolution image and slice it into chips
>>> highres = np.random.rand(5, 200, 200).astype(np.float32)
>>> target_shape = (1, 50, 50)
>>> slider = SlidingWindow(highres.shape, target_shape, overlap=(0, .5, .5))
>>> # Show how Sticher can be used to reconstruct the original image
>>> sticher = Stitcher(slider.input_shape)
>>> for sl in list(slider):
...     chip = highres[sl]
...     sticher.add(sl, chip)
>>> assert sticher.weights.max() == 4, 'some parts should be processed 4 times'
>>> recon = sticher.finalize()

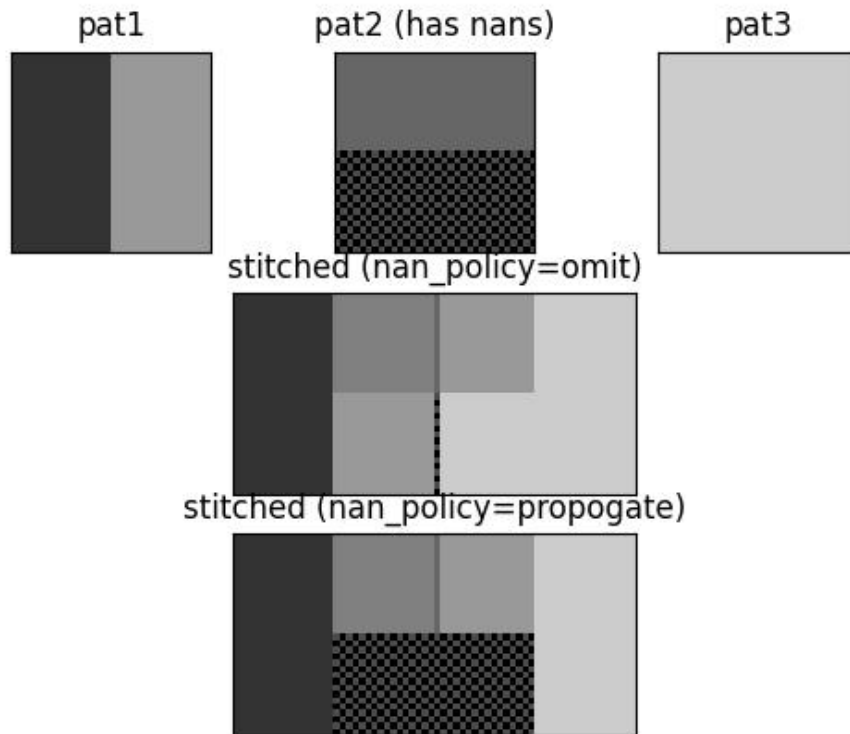
```

Example

```

>>> from kwarray.util_slider import * # NOQA
>>> import sys
>>> # Demo stitching 3 patterns where one has nans
>>> pat1 = np.full((32, 32), fill_value=0.2)
>>> pat2 = np.full((32, 32), fill_value=0.4)
>>> pat3 = np.full((32, 32), fill_value=0.8)
>>> pat1[:, 16:] = 0.6
>>> pat2[16:, :] = np.nan
>>> # Test with nan_policy=omit
>>> stitcher = Stitcher(shape=(32, 64), nan_policy='omit')
>>> stitcher[0:32, 0:32](pat1)
>>> stitcher[0:32, 16:48](pat2)
>>> stitcher[0:32, 33:64](pat3[:, 1:])
>>> final1 = stitcher.finalize()
>>> # Test without nan_policy=propagate
>>> stitcher = Stitcher(shape=(32, 64), nan_policy='propagate')
>>> stitcher[0:32, 0:32](pat1)
>>> stitcher[0:32, 16:48](pat2)
>>> stitcher[0:32, 33:64](pat3[:, 1:])
>>> final2 = stitcher.finalize()
>>> # Checks
>>> assert np.isnan(final1).sum() == 16, 'only should contain nan where no data was_
↳ stitched'
>>> assert np.isnan(final2).sum() == 512, 'should contain nan wherever a nan was_
↳ stitched'
>>> # xdoctest: +REQUIRES(--show)
>>> # xdoctest: +REQUIRES(module:kwplot)
>>> import kwplot
>>> import kwimage
>>> kwplot.autompl()
>>> kwplot.imshow(pat1, title='pat1', pnum=(3, 3, 1))
>>> kwplot.imshow(kwimage.nodata_checkerboard(pat2, square_shape=1), title='pat2_
↳ (has nans)', pnum=(3, 3, 2))
>>> kwplot.imshow(pat3, title='pat3', pnum=(3, 3, 3))
>>> kwplot.imshow(kwimage.nodata_checkerboard(final1, square_shape=1), title=
↳ 'stitched (nan_policy=omit)', pnum=(3, 1, 2))
>>> kwplot.imshow(kwimage.nodata_checkerboard(final2, square_shape=1), title=
↳ 'stitched (nan_policy=propagate)', pnum=(3, 1, 3))

```



Example

```
>>> # Example of weighted stitching
>>> # xdoctest: +REQUIRES(module:kwimage)
>>> from kwarray.util_slider import * # NOQA
>>> import kwimage
>>> import kwarray
>>> import sys
>>> data = kwimage.Mask.demo().data.astype(np.float32)
>>> data_dims = data.shape
>>> window_dims = (8, 8)
>>> # We are going to slide a window over the data, do some processing
>>> # and then stitch it all back together. There are a few ways we
>>> # can do it. Lets demo the params.
>>> basis = {
>>>     # Vary the overlap of the slider
>>>     'overlap': (0, 0.5, .9),
>>>     # Vary if we are using weighted stitching or not
>>>     'weighted': ['none', 'gauss'],
>>>     'keepbound': [True, False]
>>> }
>>> results = []
>>> gauss_weights = kwimage.gaussian_patch(window_dims)
```

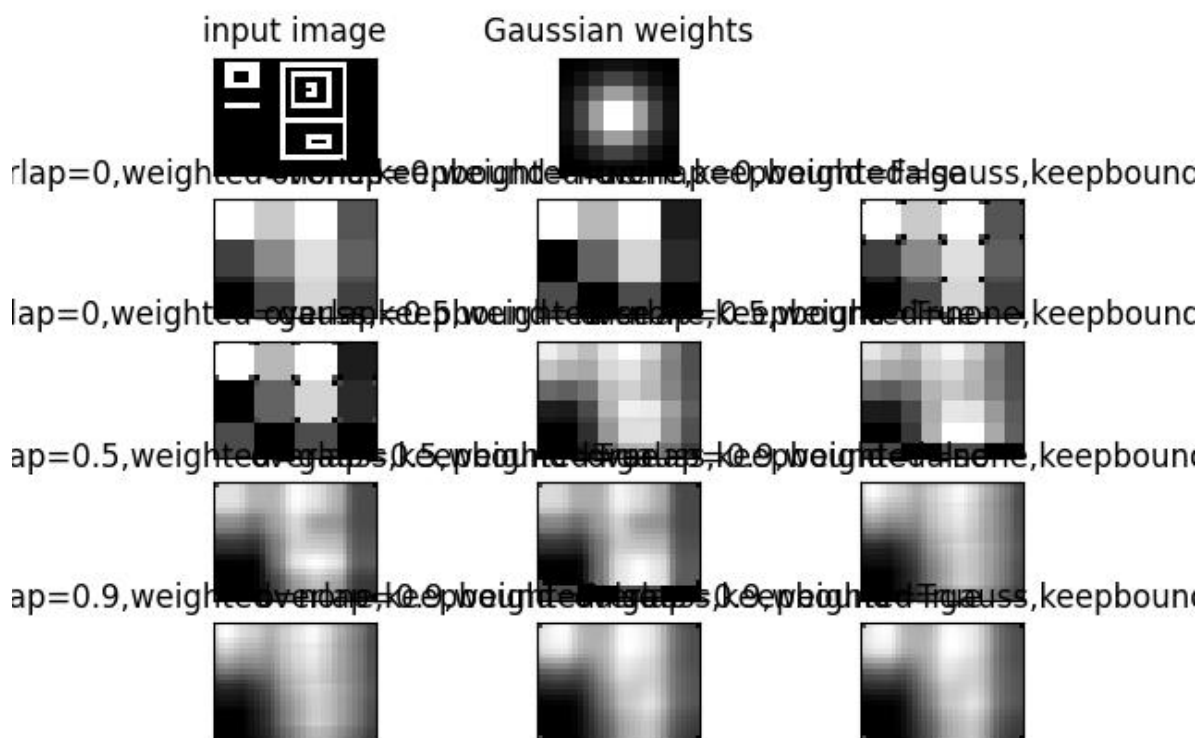
(continues on next page)

(continued from previous page)

```

>>> gauss_weights = kwimage.normalize(gauss_weights)
>>> for params in ub.named_product(basis):
>>>     if params['weighted'] == 'none':
>>>         weights = None
>>>     elif params['weighted'] == 'gauss':
>>>         weights = gauss_weights
>>>     # Build the slider and stitcher
>>>     slider = kwarray.SlidingWindow(
>>>         data_dims, window_dims, overlap=params['overlap'],
>>>         allow_overshoot=True,
>>>         keepbound=params['keepbound'])
>>>     stitcher = kwarray.Stitcher(data_dims)
>>>     # Loop over the regions
>>>     for sl in list(slider):
>>>         chip = data[sl]
>>>         # This is our dummy function for thie example.
>>>         predicted = np.ones_like(chip) * chip.sum() / chip.size
>>>         stitcher.add(sl, predicted, weight=weights)
>>>     final = stitcher.finalize()
>>>     results.append({
>>>         'final': final,
>>>         'params': params,
>>>     })
>>> # xdoctest: +REQUIRES(--show)
>>> # xdoctest: +REQUIRES(module:kwplot)
>>> import kwplot
>>> kwplot.autompl()
>>> pnum_ = kwplot.PlotNums(nCols=3, nSubplots=len(results) + 2)
>>> kwplot.imshow(data, pnum=pnum_(), title='input image')
>>> kwplot.imshow(gauss_weights, pnum=pnum_(), title='Gaussian weights')
>>> pnum_()
>>> for result in results:
>>>     param_key = ub.urepr(result['params'], compact=1)
>>>     final = result['final']
>>>     canvas = kwarray.normalize(final)
>>>     canvas = kwimage.fill_nans_with_checkers(canvas)
>>>     kwplot.imshow(canvas, pnum=pnum_(), title=param_key)

```

Parameters

- **shape** (*tuple*) – dimensions of the large image that will be created from the smaller pixels or patches.
- **device** (*str* | *int* | *torch.device*) – default is ‘numpy’, but if given as a torch device, then underlying operations will be done with torch tensors instead.
- **dtype** (*str*) – the datatype to use in the underlying accumulator.
- **nan_policy** (*str*) – if omit, check for nans and convert any to zero weight items in stitching.

add(*indices*, *patch*, *weight=None*)

Incorporate a new (possibly overlapping) patch or pixel using a weighted sum.

Parameters

- **indices** (*slice* | *tuple* | *None*) – typically a `Tuple[slice]` of pixels or a single pixel, but this can be any numpy fancy index.
- **patch** (*ndarray*) – data to patch into the bigger image.
- **weight** (*float* | *ndarray*) – weight of this patch (default to 1.0)

average()

Averages out contributions from overlapping adds using weighted average

Returns

out - the stitched image

Return type

ndarray

finalize(*indices=None*)

Averages out contributions from overlapping adds

Parameters

indices (*None* | *slice* | *tuple*) – if *None*, finalize the entire block, otherwise only finalize a subregion.

Returns

final - the stitched image

Return type

ndarray

kwarray.apply_embedded_slice(*data, data_slice, extra_padding, **padkw*)

Apply a precomputed embedded slice.

This is used as a subroutine in `padded_slice`.

Parameters

- **data** (*ndarray*) – data to slice
- **data_slice** (*Tuple[slice]*)
- **extra_padding** (*Tuple[slice]*)

Returns

ndarray

kwarray.apply_grouping(*items, groupxs, axis=0*)

Applies grouping from `group_indicies`.

Typically used in conjunction with `group_indicies()`.

Parameters

- **items** (*NDArray*) – items to group
- **groupxs** (*List[NDArray[None, Int]]*) – groups of indices
- **axis** (*None|int, default=0*)

Returns

grouped items

Return type

List[NDArray]

Example

```
>>> # xdoctest: +IGNORE_WHITESPACE
>>> idx_to_groupid = np.array([2, 1, 2, 1, 2, 1, 2, 3, 3, 3, 3])
>>> items          = np.array([1, 8, 5, 5, 8, 6, 7, 5, 3, 0, 9])
>>> (keys, groupxs) = group_indicies(idx_to_groupid)
>>> grouped_items = apply_grouping(items, groupxs)
>>> result = str(grouped_items)
>>> print(result)
[array([8, 5, 6]), array([1, 5, 8, 7]), array([5, 3, 0, 9])]
```

`kwarray.arglexmax(keys, multi=False)`

Find the index of the maximum element in a sequence of keys.

Parameters

- **keys** (*tuple*) – a k-tuple of k N-dimensional arrays. Like `np.lexsort` the last key in the sequence is used for the primary sort order, the second-to-last key for the secondary sort order, and so on.
- **multi** (*bool*) – if True, returns all indices that share the max value

Returns

either the index or list of indices

Return type

`int` | `NDArray[Any, Int]`

Example

```
>>> k, N = 100, 100
>>> rng = np.random.RandomState(0)
>>> keys = [(rng.rand(N) * N).astype(int) for _ in range(k)]
>>> multi_idx = arglexmax(keys, multi=True)
>>> idxs = np.lexsort(keys)
>>> assert sorted(idxs[::-1][:len(multi_idx)]) == sorted(multi_idx)
```

Benchark:

```
>>> import ubelt as ub
>>> k, N = 100, 100
>>> rng = np.random
>>> keys = [(rng.rand(N) * N).astype(int) for _ in range(k)]
>>> for timer in ub.Timerit(100, bestof=10, label='arglexmax'):
>>>     with timer:
>>>         arglexmax(keys)
>>> for timer in ub.Timerit(100, bestof=10, label='lexsort'):
>>>     with timer:
>>>         np.lexsort(keys)[-1]
```

`kwarray.argmaxima(arr, num, axis=None, ordered=True)`

Returns the top num maximum indicies.

This can be significantly faster than using `argsort`.

Parameters

- **arr** (*NDArray*) – input array
- **num** (*int*) – number of maximum indices to return
- **axis** (*int* | *None*) – axis to find maxima over. If None this is equivalent to using `arr.ravel()`.
- **ordered** (*bool*) – if False, returns the maximum elements in an arbitrary order, otherwise they are in decending order. (Setting this to false is a bit faster).

Todo:

- `[]` if `num` is `None`, return `arg` for all values equal to the maximum
-

Returns

NDArray

Example

```
>>> # Test cases with axis=None
>>> arr = (np.random.rand(100) * 100).astype(int)
>>> for num in range(0, len(arr) + 1):
>>>     idxs = argmaxima(arr, num)
>>>     idxs2 = argmaxima(arr, num, ordered=False)
>>>     assert np.all(arr[idxs] == np.array(sorted(arr)[::-1][:len(idxs)])),
↳ 'ordered=True must return in order'
>>>     assert sorted(idxs2) == sorted(idxs), 'ordered=False must return the right_
↳ idxs, but in any order'
```

Example

```
>>> # Test cases with axis
>>> arr = (np.random.rand(3, 5, 7) * 100).astype(int)
>>> for axis in range(len(arr.shape)):
>>>     for num in range(0, len(arr) + 1):
>>>         idxs = argmaxima(arr, num, axis=axis)
>>>         idxs2 = argmaxima(arr, num, ordered=False, axis=axis)
>>>         assert idxs.shape[axis] == num
>>>         assert idxs2.shape[axis] == num
```

`kwarray.argmaxinima(arr, num, axis=None, ordered=True)`

Returns the top `num` minimum indicies.

This can be significantly faster than using `argsort`.

Parameters

- **arr** (NDArray) – input array
- **num** (int) – number of minimum indices to return
- **axis** (int|None) – axis to find minima over. If `None` this is equivalent to using `arr.ravel()`.
- **ordered** (bool) – if `False`, returns the minimum elements in an arbitrary order, otherwise they are in ascending order. (Setting this to `false` is a bit faster).

Example

```
>>> arr = (np.random.rand(100) * 100).astype(int)
>>> for num in range(0, len(arr) + 1):
>>>     idxs = argminima(arr, num)
>>>     assert np.all(arr[idxs] == np.array(sorted(arr)[:len(idxs)])),
↳ 'ordered=True must return in order'
>>>     idxs2 = argminima(arr, num, ordered=False)
>>>     assert sorted(idxs2) == sorted(idxs), 'ordered=False must return the right_
↳ idxs, but in any order'
```

Example

```
>>> # Test cases with axis
>>> from kwarray.util_numpy import * # NOQA
>>> arr = (np.random.rand(3, 5, 7) * 100).astype(int)
>>> # make a unique array so we can check argmax consistency
>>> arr = np.arange(3 * 5 * 7)
>>> np.random.shuffle(arr)
>>> arr = arr.reshape(3, 5, 7)
>>> for axis in range(len(arr.shape)):
>>>     for num in range(0, len(arr) + 1):
>>>         idxs = argminima(arr, num, axis=axis)
>>>         idxs2 = argminima(arr, num, ordered=False, axis=axis)
>>>         print('idxs = {!r}'.format(idxs))
>>>         print('idxs2 = {!r}'.format(idxs2))
>>>         assert idxs.shape[axis] == num
>>>         assert idxs2.shape[axis] == num
>>>         # Check if argmin agrees with -argmax
>>>         idxs3 = argmaxima(-arr, num, axis=axis)
>>>         assert np.all(idxs3 == idxs)
```

Example

```
>>> arr = np.arange(20).reshape(4, 5) % 6
>>> argminima(arr, axis=1, num=2, ordered=False)
>>> argminima(arr, axis=1, num=2, ordered=True)
>>> argmaxima(-arr, axis=1, num=2, ordered=True)
>>> argmaxima(-arr, axis=1, num=2, ordered=False)
```

`kwarray.atleast_nd(arr, n, front=False)`

View inputs as arrays with at least n dimensions.

Parameters

- **arr** (*ArrayLike*) – An array-like object. Non-array inputs are converted to arrays. Arrays that already have n or more dimensions are preserved.
- **n** (*int*) – number of dimensions to ensure
- **front** (*bool*) – if True new dimensions are added to the front of the array. otherwise they are added to the back. Defaults to False.

Returns

An array with `a.ndim >= n`. Copies are avoided where possible, and views with three or more dimensions are returned. For example, a 1-D array of shape `(N,)` becomes a view of shape `(1, N, 1)`, and a 2-D array of shape `(M, N)` becomes a view of shape `(M, N, 1)`.

Return type

NDArray

See also:

`numpy.atleast_1d`, `numpy.atleast_2d`, `numpy.atleast_3d`

Example

```
>>> n = 2
>>> arr = np.array([1, 1, 1])
>>> arr_ = atleast_nd(arr, n)
>>> import ubelt as ub # NOQA
>>> result = ub.urepr(arr_.tolist(), nl=0)
>>> print(result)
[[1], [1], [1]]
```

Example

```
>>> n = 4
>>> arr1 = [1, 1, 1]
>>> arr2 = np.array(0)
>>> arr3 = np.array([[[[1]]]])
>>> arr1_ = atleast_nd(arr1, n)
>>> arr2_ = atleast_nd(arr2, n)
>>> arr3_ = atleast_nd(arr3, n)
>>> import ubelt as ub # NOQA
>>> result1 = ub.urepr(arr1_.tolist(), nl=0)
>>> result2 = ub.urepr(arr2_.tolist(), nl=0)
>>> result3 = ub.urepr(arr3_.tolist(), nl=0)
>>> result = '\n'.join([result1, result2, result3])
>>> print(result)
[[[1]], [[1]], [[1]]]
[[[0]]]
[[[1]]]
```

Note: Extensive benchmarks are in `kwarray/dev/bench_atleast_nd.py`

These demonstrate that this function is statistically faster than the numpy variants, although the difference is small. On average this function takes 480ns versus numpy which takes 790ns.

`kwarray.boolmask(indices, shape=None)`

Constructs an array of booleans where an item is True if its position is in `indices` otherwise it is False. This can be viewed as the inverse of `numpy.where()`.

Parameters

- **indices** (NDArray) – list of integer indices

- **shape** (*int* | *tuple*) – length of the returned list. If not specified the minimal possible shape to incorporate all the indices is used. In general, it is best practice to always specify this argument.

Returns

mask - mask[idx] is True if idx in indices

Return type

NDArray[Any, Int]

Example

```
>>> indices = [0, 1, 4]
>>> mask = boolmask(indices, shape=6)
>>> assert np.all(mask == [True, True, False, False, True, False])
>>> mask = boolmask(indices)
>>> assert np.all(mask == [True, True, False, False, True])
```

Example

```
>>> import kwarray
>>> import ubelt as ub # NOQA
>>> indices = np.array([(0, 0), (1, 1), (2, 1)])
>>> shape = (3, 3)
>>> mask = kwarray.boolmask(indices, shape)
>>> result = ub.urepr(mask, with_dtype=0)
>>> print(result)
np.array([[ True, False, False],
          [False,  True, False],
          [False,  True, False]])
```

kwarray.dtype_info(dtype)

Lookup datatype information

Parameters

dtype (*type*) – a numpy, torch, or python numeric data type

Returns

an iinfo of finfo structure depending on the input type.

Return type

numpy.iinfo | numpy.finfo | torch.iinfo | torch.finfo

References

..[DtypeNotes] https://higra.readthedocs.io/en/stable/_modules/higra/hg_utils.html#dtype_info

Example

```
>>> from kwarray.arrayapi import * # NOQA
>>> try:
>>>     import torch
>>> except ImportError:
>>>     torch = None
>>> results = []
>>> results += [dtype_info(float)]
>>> results += [dtype_info(int)]
>>> results += [dtype_info(complex)]
>>> results += [dtype_info(np.float32)]
>>> results += [dtype_info(np.int32)]
>>> results += [dtype_info(np.uint32)]
>>> if hasattr(np, 'complex256'):
>>>     results += [dtype_info(np.complex256)]
>>> if torch is not None:
>>>     results += [dtype_info(torch.float32)]
>>>     results += [dtype_info(torch.int64)]
>>>     results += [dtype_info(torch.complex64)]
>>> for info in results:
>>>     print('info = {!r}'.format(info))
>>> for info in results:
>>>     print('info.bits = {!r}'.format(info.bits))
```

`kwarray.embed_slice(slices, data_dims, pad=None)`

Embeds a “padded-slice” inside known data dimension.

Returns the valid data portion of the slice with extra padding for regions outside of the available dimension.

Given a slices for each dimension, image dimensions, and a padding get the corresponding slice from the image and any extra padding needed to achieve the requested window size.

Todo:

- [] Add the option to return the inverse slice
-

Parameters

- **slices** (*Tuple[slice, ...]*) – a tuple of slices for to apply to data data dimension.
- **data_dims** (*Tuple[int, ...]*) – n-dimension data sizes (e.g. 2d height, width)
- **pad** (*int | List[int] | Tuple[int, int]*) – extra pad applied to (start / end) / (both) sides of each slice dim

Returns

data_slice - **Tuple[slice]** a slice that can be applied to an array

with with shape *data_dims*. This slice will not correspond to the full window size if the requested slice is out of bounds.

extra_padding - **extra padding needed after slicing to achieve**

the requested window size.

Return type

Tuple

Example

```
>>> # Case where slice is inside the data dims on left edge
>>> import kwarray
>>> slices = (slice(0, 10), slice(0, 10))
>>> data_dims = [300, 300]
>>> pad = [10, 5]
>>> a, b = kwarray.embed_slice(slices, data_dims, pad)
>>> print('data_slice = {!r}'.format(a))
>>> print('extra_padding = {!r}'.format(b))
data_slice = (slice(0, 20, None), slice(0, 15, None))
extra_padding = [(10, 0), (5, 0)]
```

Example

```
>>> # Case where slice is bigger than the image
>>> import kwarray
>>> slices = (slice(-10, 400), slice(-10, 400))
>>> data_dims = [300, 300]
>>> pad = [10, 5]
>>> a, b = kwarray.embed_slice(slices, data_dims, pad)
>>> print('data_slice = {!r}'.format(a))
>>> print('extra_padding = {!r}'.format(b))
data_slice = (slice(0, 300, None), slice(0, 300, None))
extra_padding = [(20, 110), (15, 105)]
```

Example

```
>>> # Case where slice is inside than the image
>>> import kwarray
>>> slices = (slice(10, 40), slice(10, 40))
>>> data_dims = [300, 300]
>>> pad = None
>>> a, b = kwarray.embed_slice(slices, data_dims, pad)
>>> print('data_slice = {!r}'.format(a))
>>> print('extra_padding = {!r}'.format(b))
data_slice = (slice(10, 40, None), slice(10, 40, None))
extra_padding = [(0, 0), (0, 0)]
```

Example

```
>>> # Test error cases
>>> import kwarray
>>> import pytest
>>> slices = (slice(0, 40), slice(10, 40))
>>> data_dims = [300, 300]
>>> with pytest.raises(ValueError):
>>>     kwarray.embed_slice(slices, data_dims[0:1])
```

(continues on next page)

(continued from previous page)

```

>>> with pytest.raises(ValueError):
>>>     kwarray.embed_slice(slices[0:1], data_dims)
>>> with pytest.raises(ValueError):
>>>     kwarray.embed_slice(slices, data_dims, pad=[(1, 1)])
>>> with pytest.raises(ValueError):
>>>     kwarray.embed_slice(slices, data_dims, pad=[1])

```

`kwarray.ensure_rng(rng=None, api='numpy')`

Coerces input into a random number generator.

This function is useful for ensuring that your code uses a controlled internal random state that is independent of other modules.

If the input is `None`, then a global random state is returned.

If the input is a numeric value, then that is used as a seed to construct a random state.

If the input is a random number generator, then another random number generator with the same state is returned. Depending on the `api`, this random state is either return as-is, or used to construct an equivalent random state with the requested `api`.

Parameters

- **rng** (*int* | *float* | *None* | *numpy.random.RandomState* | *random.Random*) – if `None`, then defaults to the global rng. Otherwise this can be an integer or a `RandomState` class. Defaults to the global random.
- **api** (*str*) – specify the type of random number generator to use. This can either be ‘numpy’ for a `numpy.random.RandomState` object or ‘python’ for a `random.Random` object. Defaults to `numpy`.

Returns

rng - either a numpy or python random number generator, depending on the setting of `api`.

Return type

(*numpy.random.RandomState* | *random.Random*)

Example

```

>>> rng = ensure_rng(None)
>>> ensure_rng(0).randint(0, 1000)
684
>>> ensure_rng(np.random.RandomState(1)).randint(0, 1000)
37

```

Example

```

>>> num = 4
>>> print('--- Python as PYTHON ---')
>>> py_rng = random.Random(0)
>>> pp_nums = [py_rng.random() for _ in range(num)]
>>> print(pp_nums)
>>> print('--- Numpy as PYTHON ---')
>>> np_rng = ensure_rng(random.Random(0), api='numpy')

```

(continues on next page)

(continued from previous page)

```

>>> np_nums = [np_rng.rand() for _ in range(num)]
>>> print(np_nums)
>>> print('--- Numpy as NUMPY---')
>>> np_rng = np.random.RandomState(seed=0)
>>> nn_nums = [np_rng.rand() for _ in range(num)]
>>> print(nn_nums)
>>> print('--- Python as NUMPY---')
>>> py_rng = ensure_rng(np.random.RandomState(seed=0), api='python')
>>> pn_nums = [py_rng.random() for _ in range(num)]
>>> print(pn_nums)
>>> assert np_nums == pp_nums
>>> assert pn_nums == nn_nums

```

Example

```

>>> # Test that random modules can be coerced
>>> import random
>>> import numpy as np
>>> ensure_rng(random, api='python')
>>> ensure_rng(random, api='numpy')
>>> ensure_rng(np.random, api='python')
>>> ensure_rng(np.random, api='numpy')

```

`kwarray.equal_with_nan(a1, a2)`

Numpy has `array_equal` with `equal_nan=True`, but this is elementwise

Parameters

- **a1** (*ArrayLike*) – input array
- **a2** (*ArrayLike*) – input array

Example

```

>>> import kwarray
>>> a1 = np.array([
>>>     [np.nan, 0, np.nan],
>>>     [np.nan, 0, 0],
>>>     [np.nan, 1, 0],
>>>     [np.nan, 1, np.nan],
>>> ])
>>> a2 = np.array([np.nan, 0, np.nan])
>>> flags = kwarray.equal_with_nan(a1, a2)
>>> assert np.array_equal(flags, np.array([
>>>     [ True, False,  True],
>>>     [ True, False, False],
>>>     [ True,  True, False],
>>>     [ True,  True,  True]
>>> ]))

```

`kwarray.find_robust_normalizers(data, params='auto')`

Finds robust normalization statistics a set of scalar observations.

The idea is to estimate “fense” parameters: minimum and maximum values where anything under / above these values are likely outliers. For non-linear normalizaiton schemes we can also estimate an likely middle and extent of the data.

Parameters

- **data** (*ndarray*) – a 1D numpy array where invalid data has already been removed
- **params** (*str* | *dict*) – normalization params.

When passed as a dictionary valid params are:

scaling (str):

This is the “mode” that will be used in the final normalization. Currently has no impact on the Defaults to ‘linear’. Can also be ‘sigmoid’.

extrema (str):

The method for determening what the extrama are. Can be “quantile” for strict quantile clipping Can be “adaptive-quantile” for an IQR-like adjusted quantile method. Can be “tukey” or “IQR” for an exact IQR method.

low (float): This is the low quantile for likely inliers.

mid (float): This is the middle quantlie for likely inliers.

high (float): This is the high quantile for likely inliers.

Can be specified as a concise string.

The string “auto” defaults to:

```
dict(extrema='adaptive-quantile', scaling='linear', low=0.01,
mid=0.5, high=0.9).
```

The string “tukey” defaults to:

```
dict(extrema='tukey', scaling='linear').
```

Returns

normalization parameters that can be passed to `kwarray.normalize()` containing the keys:

type (str): which is always ‘normalize’

mode (str): the value of `params['scaling']`

min_val (float): the determined “robust” minimum inlier value.

max_val (float): the determined “robust” maximum inlier value.

beta (float): the determined “robust” middle value for use in non-linear normalizers.

alpha (float): the determined “robust” extent value for use in non-linear normalizers.

Return type

`Dict[str, str | float]`

Note: The defaults and methods of this function are subject to change.

Todo:

- [] No (or minimal) Magic Numbers! Use first principles to determine defaults.
- [] Probably a lot of literature on the subject.
- [] <https://arxiv.org/pdf/1707.09752.pdf>
- [] <https://www.tandfonline.com/doi/full/10.1080/02664763.2019.1671961>
- [] <https://www.rips-irsp.com/articles/10.5334/irsp.289/>
- [] **This function is not possible to get right in every case**
(probably can prove this with a NFL theroem), might be useful to allow the user to specify a “model” which is specific to some domain.

Example

```
>>> from kwarray.util_robust import * # NOQA
>>> data = np.random.rand(100)
>>> norm_params1 = find_robust_normalizers(data, params='auto')
>>> norm_params2 = find_robust_normalizers(data, params={'low': 0, 'high': 1.0})
>>> norm_params3 = find_robust_normalizers(np.empty(0), params='auto')
>>> print('norm_params1 = {}'.format(ub.urepr(norm_params1, nl=1)))
>>> print('norm_params2 = {}'.format(ub.urepr(norm_params2, nl=1)))
>>> print('norm_params3 = {}'.format(ub.urepr(norm_params3, nl=1)))
```

Example

```
>>> # xdoctest: +REQUIRES(module:scipy)
>>> from kwarray.util_robust import * # NOQA
>>> from kwarray.distributions import Mixture
>>> import ubelt as ub
>>> # A random mixture distribution for testing
>>> data = Mixture.random(6).sample(3000)
```

`kwarray.generalized_logistic(x, floor=0, capacity=1, C=1, y_intercept=None, Q=None, growth=1, v=1)`

A generalization of the logistic / sigmoid functions that allows for flexible specification of S-shaped curve.

This is also known as a “Richards curve” [WikiRichardsCurve].

Parameters

- **x** (*NDArray*) – input x coordinates
- **floor** (*float*) – the lower (left) asymptote. (Also called A in some texts). Defaults to 0.
- **capacity** (*float*) – the carrying capacity. When C=1, this is the upper (right) asymptote. (Also called K in some texts). Defaults to 1.
- **C** (*float*) – Has influence on the upper asymptote. Defaults to 1. This is typically not modified.
- **y_intercept** (*float | None*) – specify where the the y intercept is at x=0. Mutually exclusive with Q.
- **Q** (*float | None*) – related to the value of the function at x=0. Mutually exclusive with y_intercept. Defaults to 1.

- **growth** (*float*) – the growth rate (also called B in some texts). Defaults to 1.
- **v** (*float*) – Positive number that influences near which asymptote the growth occurs. Defaults to 1.

Returns

the values for each input

Return type

NDArray

References**Example**

```
>>> from kwarray.util_numpy import * # NOQA
>>> # xdoctest: +REQUIRES(module:pandas)
>>> import pandas as pd
>>> import ubelt as ub
>>> x = np.linspace(-3, 3, 30)
>>> basis = {
>>>     # 'y_intercept': [0.1, 0.5, 0.8, -1],
>>>     # 'y_intercept': [0.1, 0.5, 0.8],
>>>     'v': [0.5, 1.0, 2.0],
>>>     'growth': [-1, 0, 2],
>>> }
>>> grid = list(ub.named_product(basis))
>>> datas = []
>>> for params in grid:
>>>     y = generalized_logistic(x, **params)
>>>     data = pd.DataFrame({'x': x, 'y': y})
>>>     key = ub.urepr(params, compact=1)
>>>     data['key'] = key
>>>     for k, v in params.items():
>>>         data[k] = v
>>>     datas.append(data)
>>> all_data = pd.concat(datas).reset_index()
>>> # xdoctest: +REQUIRES(--show)
>>> # xdoctest: +REQUIRES(module:kwplot)
>>> import kwplot
>>> plt = kwplot.autoplt()
>>> sns = kwplot.autosns()
>>> plt.gca().cla()
>>> sns.lineplot(data=all_data, x='x', y='y', hue='growth', size='v')
```

`kwarray.group_consecutive(arr, offset=1)`

Returns lists of consecutive values. Implementation inspired by³.

Parameters

- **arr** (*NDArray*) – array of ordered values
- **offset** (*float*, *default=1*) – any two values separated by this offset are grouped. In the default case, when `offset=1`, this groups increasing values like: 0, 1, 2. When `offset` is 0

³ <http://stackoverflow.com/questions/7352684/groups-consecutive-elements>

it groups consecutive values that are the same, e.g.: 4, 4, 4.

Returns

a list of arrays that are the groups from the input

Return type

List[NDArray]

Note: This is equivalent (and faster) to using: `apply_grouping(data, group_consecutive_indices(data))`

References

Example

```
>>> arr = np.array([1, 2, 3, 5, 6, 7, 8, 9, 10, 15, 99, 100, 101])
>>> groups = group_consecutive(arr)
>>> print('groups = {}'.format(list(map(list, groups))))
groups = [[1, 2, 3], [5, 6, 7, 8, 9, 10], [15], [99, 100, 101]]
>>> arr = np.array([0, 0, 3, 0, 0, 7, 2, 3, 4, 4, 4, 1, 1])
>>> groups = group_consecutive(arr, offset=1)
>>> print('groups = {}'.format(list(map(list, groups))))
groups = [[0], [0], [3], [0], [0], [7], [2, 3, 4], [4], [4], [1], [1]]
>>> groups = group_consecutive(arr, offset=0)
>>> print('groups = {}'.format(list(map(list, groups))))
groups = [[0, 0], [3], [0, 0], [7], [2], [3], [4, 4, 4], [1, 1]]
```

`kwarray.group_consecutive_indices(arr, offset=1)`

Returns lists of indices pointing to consecutive values

Parameters

- **arr** (NDArray) – array of ordered values
- **offset** (float, default=1) – any two values separated by this offset are grouped.

Returns

groupxs: a list of indices

Return type

List[NDArray]

SeeAlso:

[`group\_consecutive\(\)`](#)

[`apply\_grouping\(\)`](#)

Example

```

>>> arr = np.array([1, 2, 3, 5, 6, 7, 8, 9, 10, 15, 99, 100, 101])
>>> groupxs = group_consecutive_indices(arr)
>>> print('groupxs = {}'.format(list(map(list, groupxs))))
groupxs = [[0, 1, 2], [3, 4, 5, 6, 7, 8], [9], [10, 11, 12]]
>>> assert all(np.array_equal(a, b) for a, b in zip(group_consecutive(arr, 1),
↳ apply_grouping(arr, groupxs)))
>>> arr = np.array([0, 0, 3, 0, 0, 7, 2, 3, 4, 4, 4, 1, 1])
>>> groupxs = group_consecutive_indices(arr, offset=1)
>>> print('groupxs = {}'.format(list(map(list, groupxs))))
groupxs = [[0], [1], [2], [3], [4], [5], [6, 7, 8], [9], [10], [11], [12]]
>>> assert all(np.array_equal(a, b) for a, b in zip(group_consecutive(arr, 1),
↳ apply_grouping(arr, groupxs)))
>>> groupxs = group_consecutive_indices(arr, offset=0)
>>> print('groupxs = {}'.format(list(map(list, groupxs))))
groupxs = [[0, 1], [2], [3, 4], [5], [6], [7], [8, 9, 10], [11, 12]]
>>> assert all(np.array_equal(a, b) for a, b in zip(group_consecutive(arr, 0),
↳ apply_grouping(arr, groupxs)))

```

`kwarray.group_indices(idx_to_groupid, assume_sorted=False)`

Find unique items and the indices at which they appear in an array.

A common use case of this function is when you have a list of objects (often numeric but sometimes not) and an array of “group-ids” corresponding to that list of objects.

Using this function will return a list of indices that can be used in conjunction with [apply_grouping\(\)](#) to group the elements. This is most useful when you have many lists (think column-major data) corresponding to the group-ids.

In cases where there is only one list of objects or knowing the indices doesn’t matter, then consider using `func:group_items` instead.

Parameters

- **idx_to_groupid** (*NDArray*) – The input array, where each item is interpreted as a group id. For the fastest runtime, the input array must be numeric (ideally with integer types). If the type is non-numeric then the less efficient `ubelt.group_items()` is used.
- **assume_sorted** (*bool*) – If the input array is sorted, then setting this to True will avoid an unnecessary sorting operation and improve efficiency. Defaults to False.

Returns

(**keys**, **groupxs**) -

keys (*NDArray*):

The unique elements of the input array in order

groupxs (*List[NDArray]*):

Corresponding list of indexes. The i-th item is an array indicating the indices where the item `key[i]` appeared in the input array.

Return type

Tuple[*NDArray*, List[*NDArray*]]

Example

```

>>> # xdoctest: +IGNORE_WHITESPACE
>>> import kwarray
>>> import ubelt as ub
>>> idx_to_groupid = np.array([2, 1, 2, 1, 2, 1, 2, 3, 3, 3, 3])
>>> (keys, groupxs) = kwarray.group_indices(idx_to_groupid)
>>> print('keys = ' + ub.urepr(keys, with_dtype=False))
>>> print('groupxs = ' + ub.urepr(groupxs, with_dtype=False))
keys = np.array([1, 2, 3])
groupxs = [
    np.array([1, 3, 5]),
    np.array([0, 2, 4, 6]),
    np.array([ 7,  8,  9, 10]),
]

```

Example

```

>>> # xdoctest: +IGNORE_WHITESPACE
>>> import kwarray
>>> import ubelt as ub
>>> # 2d arrays must be flattened before coming into this function so
>>> # information is on the last axis
>>> idx_to_groupid = np.array([[ 24], [ 129], [ 659], [ 659], [ 24],
...      [659], [ 659], [ 822], [ 659], [ 659], [24]]).T[0]
>>> (keys, groupxs) = kwarray.group_indices(idx_to_groupid)
>>> # Different versions of numpy may produce different orderings
>>> # so normalize these to make test output consistent
>>> # [gxs.sort() for gxs in groupxs]
>>> print('keys = ' + ub.urepr(keys, with_dtype=False))
>>> print('groupxs = ' + ub.urepr(groupxs, with_dtype=False))
keys = np.array([ 24, 129, 659, 822])
groupxs = [
    np.array([ 0,  4, 10]),
    np.array([1]),
    np.array([2, 3, 5, 6, 8, 9]),
    np.array([7]),
]

```

Example

```

>>> # xdoctest: +IGNORE_WHITESPACE
>>> import kwarray
>>> import ubelt as ub
>>> idx_to_groupid = np.array([True, True, False, True, False, False, True])
>>> (keys, groupxs) = kwarray.group_indices(idx_to_groupid)
>>> print(ub.urepr(keys, with_dtype=False))
>>> print(ub.urepr(groupxs, with_dtype=False))
np.array([False,  True])
[

```

(continues on next page)

(continued from previous page)

```
np.array([2, 4, 5]),
np.array([0, 1, 3, 6]),
]
```

Example

```
>>> # xdoctest: +IGNORE_WHITESPACE
>>> import ubelt as ub
>>> import kvarray
>>> idx_to_groupid = [('a', 'b'), ('d', 'b'), ('a', 'b'), ('a', 'b')]
>>> (keys, groupxs) = kvarray.group_indices(idx_to_groupid)
>>> print(ub.urepr(keys, with_dtype=False))
>>> print(ub.urepr(groupxs, with_dtype=False))
[
  ('a', 'b'),
  ('d', 'b'),
]
[
  np.array([0, 2, 3]),
  np.array([1]),
]
```

kvarray.group_items(*item_list*, *groupid_list*, *assume_sorted=False*, *axis=None*)

Groups a list of items by group id.

Works like `ubelt.group_items()`, but with numpy optimizations. This can be quite a bit faster than using `itertools.groupby()`¹.

In cases where there are many lists of items to group (think column-major data), consider using `group_indices()` and `apply_grouping()` instead.

Parameters

- **item_list** (*NDArray*) – The input array of items to group. Extended typing `NDArray[Any, VT]`
- **groupid_list** (*NDArray*) – Each item is an id corresponding to the item at the same position in *item_list*. For the fastest runtime, the input array must be numeric (ideally with integer types). This list must be 1-dimensional. Extended typing `NDArray[Any, KT]`
- **assume_sorted** (*bool*) – If the input array is sorted, then setting this to `True` will avoid an unnecessary sorting operation and improve efficiency. Defaults to `False`.
- **axis** (*int* | *None*) – Group along a particular axis in *items* if it is n-dimensional.

Returns

mapping from groupids to corresponding items. Extended typing `Dict[KT, NDArray[Any, VT]]`.

Return type

`Dict[Any, NDArray]`

¹ <http://stackoverflow.com/questions/4651683/>

² [numpy-grouping-using-itertools-groupby-performance](#)

References

Example

```
>>> from kvarray.util_groups import * # NOQA
>>> items = np.array([0, 1, 2, 3, 4, 5, 6, 7, 1, 1])
>>> keys = np.array([2, 2, 1, 1, 0, 1, 0, 1, 1, 1])
>>> grouped = group_items(items, keys)
>>> print('grouped = ' + ub.urepr(grouped, nl=1, with_dtype=False, sort=1))
grouped = {
    0: np.array([4, 6]),
    1: np.array([2, 3, 5, 7, 1, 1]),
    2: np.array([0, 1]),
}
```

`kvarray.isect_flags(arr, other)`

Check which items in an array intersect with another set of items

Parameters

- **arr** (*NDArray*) – items to check
- **other** (*Iterable*) – items to check if they exist in arr

Returns

booleans corresponding to arr indicating if any item in other
is also contained in other.

Return type

NDArray

Example

```
>>> arr = np.array([
>>>     [1, 2, 3, 4],
>>>     [5, 6, 3, 4],
>>>     [1, 1, 3, 4],
>>> ])
>>> other = np.array([1, 4, 6])
>>> mask = isect_flags(arr, other)
>>> print(mask)
[[ True False False  True]
 [False  True False  True]
 [ True  True False  True]]
```

`kvarray.iter_reduce_ufunc(ufunc, arrs, out=None, default=None)`

constant memory iteration and reduction

Applies ufunc from left to right over the input arrays

Parameters

- **ufunc** (*Callable*) – called on each pair of consecutive ndarrays
- **arrs** (*Iterator[NDArray]*) – iterator of ndarrays
- **default** (*object*) – return value when iterator is empty

Returns

if len(arrs) == 0, returns default if len(arrs) == 1, returns arrs[0], if len(arrs) >= 2, returns ufunc(...ufunc(ufunc(arrs[0], arrs[1]), arrs[2]),...arrs[n-1])

Return type

NDArray

Example

```
>>> arr_list = [
...     np.array([0, 1, 2, 3, 8, 9]),
...     np.array([4, 1, 2, 3, 4, 5]),
...     np.array([0, 5, 2, 3, 4, 5]),
...     np.array([1, 1, 6, 3, 4, 5]),
...     np.array([0, 1, 2, 7, 4, 5])
... ]
>>> memory = np.array([9, 9, 9, 9, 9, 9])
>>> gen_memory = memory.copy()
>>> def arr_gen(arr_list, gen_memory):
...     for arr in arr_list:
...         gen_memory[:] = arr
...         yield gen_memory
>>> print('memory = %r' % (memory,))
>>> print('gen_memory = %r' % (gen_memory,))
>>> ufunc = np.maximum
>>> res1 = iter_reduce_ufunc(ufunc, iter(arr_list), out=None)
>>> res2 = iter_reduce_ufunc(ufunc, iter(arr_list), out=memory)
>>> res3 = iter_reduce_ufunc(ufunc, arr_gen(arr_list, gen_memory), out=memory)
>>> print('res1      = %r' % (res1,))
>>> print('res2      = %r' % (res2,))
>>> print('res3      = %r' % (res3,))
>>> print('memory     = %r' % (memory,))
>>> print('gen_memory = %r' % (gen_memory,))
>>> assert np.all(res1 == res2)
>>> assert np.all(res2 == res3)
```

kwarray.maxvalue_assignment(value)

Finds the maximum value assignment based on a NxM value matrix. Any pair with a non-positive value will not be assigned.

Parameters

value (ndarray) – NxM matrix, value[i, j] is the value of matching i and j

Returns

tuple containing a list of assignment of rows
and columns, and the total value of the assignment.

Return type

Tuple[list, float]

CommandLine

```
xdoctest -m ~/code/kwarray/kwarray/algo_assignment.py maxvalue_assignment
```

Example

```
>>> # xdoctest: +REQUIRES(module:scipy)
>>> # Costs to match item i in set1 with item j in set2.
>>> value = np.array([
>>>     [9, 2, 1, 3],
>>>     [4, 1, 5, 5],
>>>     [9, 9, 2, 4],
>>>     [-1, -1, -1, -1],
>>> ])
>>> ret = maxvalue_assignment(value)
>>> # Note, depending on the scipy version the assignment might change
>>> # but the value should always be the same.
>>> print('Total value: {}'.format(ret[1]))
Total value: 23.0
>>> print('Assignment: {}'.format(ret[0])) # xdoc: +IGNORE_WANT
Assignment: [(0, 0), (1, 3), (2, 1)]
```

```
>>> ret = maxvalue_assignment(np.array([[np.inf]]))
>>> print('Assignment: {}'.format(ret[0]))
>>> print('Total value: {}'.format(ret[1]))
Assignment: [(0, 0)]
Total value: inf
```

```
>>> ret = maxvalue_assignment(np.array([[0]]))
>>> print('Assignment: {}'.format(ret[0]))
>>> print('Total value: {}'.format(ret[1]))
Assignment: []
Total value: 0
```

kwarray.mincost_assignment(cost)

Finds the minimum cost assignment based on a NxM cost matrix, subject to the constraint that each row can match at most one column and each column can match at most one row. Any pair with a cost of infinity will not be assigned.

Parameters

cost (*ndarray*) – NxM matrix, cost[i, j] is the cost to match i and j

Returns

tuple containing a list of assignment of rows
and columns, and the total cost of the assignment.

Return type

Tuple[list, float]

CommandLine

```
xdoctest -m ~/code/kwarray/kwarray/algo_assignment.py mincost_assignment
```

Example

```
>>> # xdoctest: +REQUIRES(module:scipy)
>>> # Costs to match item i in set1 with item j in set2.
>>> cost = np.array([
>>>     [9, 2, 1, 9],
>>>     [4, 1, 5, 5],
>>>     [9, 9, 2, 4],
>>> ])
>>> ret = mincost_assignment(cost)
>>> print('Assignment: {}'.format(ret[0]))
>>> print('Total cost: {}'.format(ret[1]))
Assignment: [(0, 2), (1, 1), (2, 3)]
Total cost: 6
```

Example

```
>>> # xdoctest: +REQUIRES(module:scipy)
>>> cost = np.array([
>>>     [0, 0, 0, 0],
>>>     [4, 1, 5, -np.inf],
>>>     [9, 9, np.inf, 4],
>>>     [9, -2, np.inf, 4],
>>> ])
>>> ret = mincost_assignment(cost)
>>> print('Assignment: {}'.format(ret[0]))
>>> print('Total cost: {}'.format(ret[1]))
Assignment: [(0, 2), (1, 3), (2, 0), (3, 1)]
Total cost: -inf
```

Example

```
>>> # xdoctest: +REQUIRES(module:scipy)
>>> cost = np.array([
>>>     [0, 0, 0, 0],
>>>     [4, 1, 5, -3],
>>>     [1, 9, np.inf, 0.1],
>>>     [np.inf, np.inf, np.inf, 100],
>>> ])
>>> ret = mincost_assignment(cost)
>>> print('Assignment: {}'.format(ret[0]))
>>> print('Total cost: {}'.format(ret[1]))
Assignment: [(0, 2), (1, 1), (2, 0), (3, 3)]
Total cost: 102.0
```

`kwarray.mindist_assignment(vecs1, vecs2, p=2)`

Finds minimum cost assignment between two sets of D dimensional vectors.

Parameters

- **vecs1** (*np.ndarray*) – NxD array of vectors representing items in vecs1
- **vecs2** (*np.ndarray*) – MxD array of vectors representing items in vecs2
- **p** (*float*) – L-p norm to use. Default is 2 (aka Euclidean)

Returns

tuple containing assignments of rows in vecs1 to
rows in vecs2, and the total distance between assigned pairs.

Return type

Tuple[list, float]

Note: Thin wrapper around `mincost_assignment`

CommandLine

```
xdoctest -m ~/code/kwarray/kwarray/algo_assignment.py mindist_assignment
```

CommandLine

```
xdoctest -m ~/code/kwarray/kwarray/algo_assignment.py mindist_assignment
```

Example

```
>>> # xdoctest: +REQUIRES(module:scipy)
>>> # Rows are detections in img1, cols are detections in img2
>>> rng = np.random.RandomState(43)
>>> vecs1 = rng.randint(0, 10, (5, 2))
>>> vecs2 = rng.randint(0, 10, (7, 2))
>>> ret = mindist_assignment(vecs1, vecs2)
>>> print('Total error: {:.4f}'.format(ret[1]))
Total error: 8.2361
>>> print('Assignment: {}'.format(ret[0])) # xdoc: +IGNORE_WANT
Assignment: [(0, 0), (1, 3), (2, 5), (3, 2), (4, 6)]
```

`kwarray.normalize(arr, mode='linear', alpha=None, beta=None, out=None, min_val=None, max_val=None)`

Normalizes input values based on a specified scheme.

The default behavior is a linear normalization between 0.0 and 1.0 based on the min/max values of the input. Parameters can be specified to achieve more general constrat stretching or signal rebalancing. Implements the linear and sigmoid normalization methods described in [WikiNorm].

Parameters

- **arr** (*NDAarray*) – array to normalize, usually an image

- **out** (*NDArray* | *None*) – output array. Note, that we will create an internal floating point copy for integer computations.
- **mode** (*str*) – either linear or sigmoid.
- **alpha** (*float*) – Only used if mode=sigmoid. Division factor (pre-sigmoid). If unspecified computed as: $\max(\text{abs}(\text{old_min} - \text{beta}), \text{abs}(\text{old_max} - \text{beta})) / 6.212606$. Note this parameter is sensitive to if the input is a float or uint8 image.
- **beta** (*float*) – subtractive factor (pre-sigmoid). This should be the intensity of the most interesting bits of the image, i.e. bring them to the center (0) of the distribution. Defaults to $(\text{max} - \text{min}) / 2$. Note this parameter is sensitive to if the input is a float or uint8 image.
- **min_val** – inputs lower than this minimum value are clipped
- **max_val** – inputs higher than this maximum value are clipped.

SeeAlso:

`find_robust_normalizers()` - determine robust parameters for normalize to mitigate the effect of outliers.

`robust_normalize()` - finds and applies robust normalization parameters

References

Example

```
>>> raw_f = np.random.rand(8, 8)
>>> norm_f = normalize(raw_f)
```

```
>>> raw_f = np.random.rand(8, 8) * 100
>>> norm_f = normalize(raw_f)
>>> assert isclose(norm_f.min(), 0)
>>> assert isclose(norm_f.max(), 1)
```

```
>>> raw_u = (np.random.rand(8, 8) * 255).astype(np.uint8)
>>> norm_u = normalize(raw_u)
```

Example

```
>>> # xdoctest: +REQUIRES(module:kwimage)
>>> import kwimage
>>> arr = kwimage.grab_test_image('lowcontrast')
>>> arr = kwimage.ensure_float01(arr)
>>> norms = {}
>>> norms['arr'] = arr.copy()
>>> norms['linear'] = normalize(arr, mode='linear')
>>> # xdoctest: +REQUIRES(module:scipy)
>>> norms['sigmoid'] = normalize(arr, mode='sigmoid')
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
```

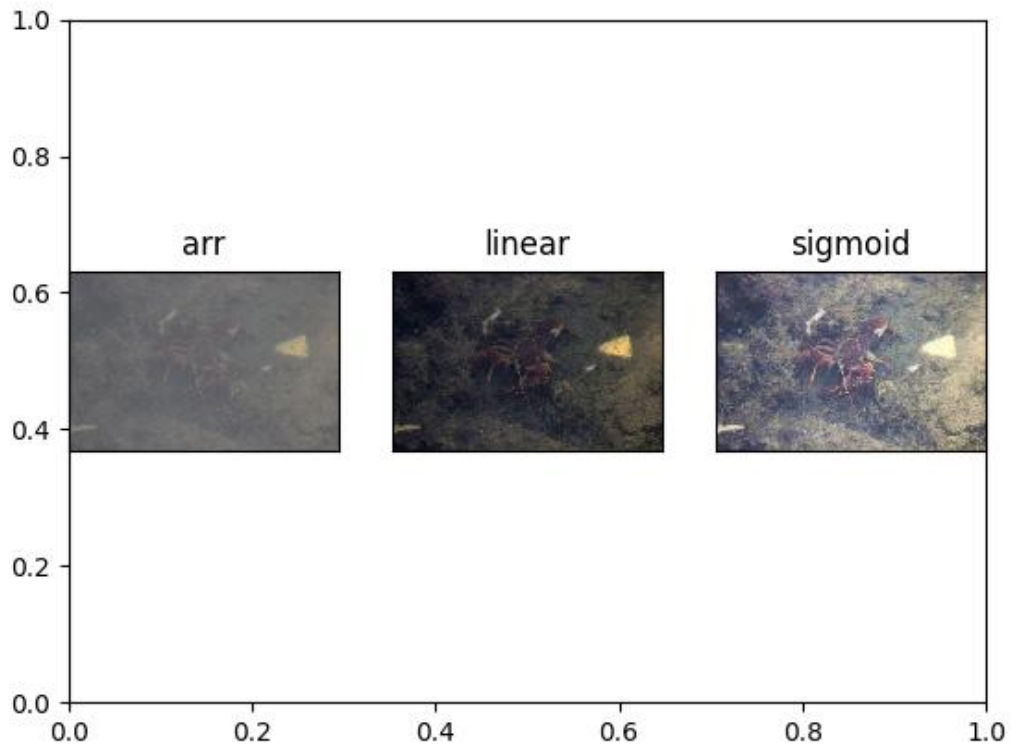
(continues on next page)

(continued from previous page)

```

>>> kwplot.autompl()
>>> kwplot.figure(fnum=1, doclf=True)
>>> pnum_ = kwplot.PlotNums(nSubplots=len(norms))
>>> for key, img in norms.items():
>>>     kwplot.imshow(img, pnum=pnum_(), title=key)

```



Example

```

>>> # xdoctest: +REQUIRES(module:kwimage)
>>> arr = np.array([np.inf])
>>> normalize(arr, mode='linear')
>>> # xdoctest: +REQUIRES(module:scipy)
>>> normalize(arr, mode='sigmoid')
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.figure(fnum=1, doclf=True)
>>> pnum_ = kwplot.PlotNums(nSubplots=len(norms))
>>> for key, img in norms.items():
>>>     kwplot.imshow(img, pnum=pnum_(), title=key)

```

Benchmark

```

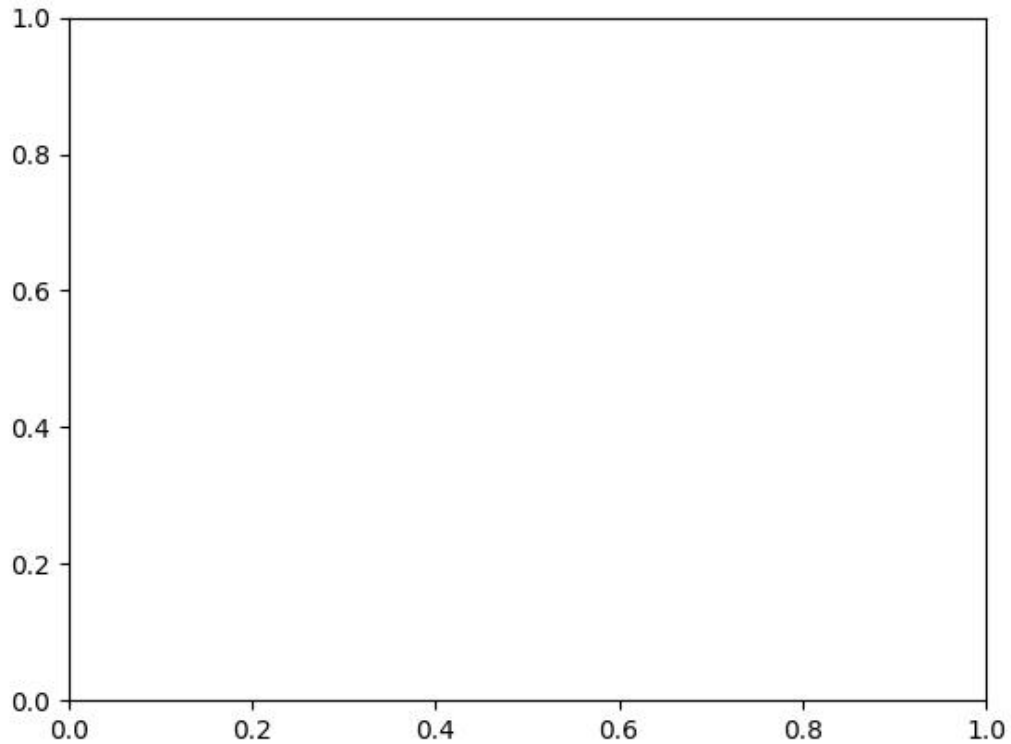
>>> # Our method is faster than standard in-line implementations for
>>> # uint8 and competitive with in-line float32, in addition to being
>>> # more concise and configurable. In 3.11 all inplace variants are
>>> # faster.
>>> # xdoctest: +REQUIRES(module:kwimage)
>>> import timerit
>>> import kwimage
>>> import kwarray
>>> ti = timerit.Timerit(1000, bestof=10, verbose=2, unit='ms')
>>> arr = kwimage.grab_test_image('lowcontrast', dsize=(512, 512))
>>> #
>>> arr = kwimage.ensure_float01(arr)
>>> out = arr.copy()
>>> for timer in ti.reset('inline_naive(float)':
>>>     with timer:
>>>         (arr - arr.min()) / (arr.max() - arr.min())
>>> #
>>> for timer in ti.reset('inline_faster(float)':
>>>     with timer:
>>>         max_ = arr.max()
>>>         min_ = arr.min()
>>>         result = (arr - min_) / (max_ - min_)
>>> #
>>> for timer in ti.reset('kwarray.normalize(float)':
>>>     with timer:
>>>         kwarray.normalize(arr)
>>> #
>>> for timer in ti.reset('kwarray.normalize(float, inplace)':
>>>     with timer:
>>>         kwarray.normalize(arr, out=out)
>>> #
>>> arr = kwimage.ensure_uint255(arr)
>>> out = arr.copy()
>>> for timer in ti.reset('inline_naive(uint8)':
>>>     with timer:
>>>         (arr - arr.min()) / (arr.max() - arr.min())
>>> #
>>> for timer in ti.reset('inline_faster(uint8)':
>>>     with timer:
>>>         max_ = arr.max()
>>>         min_ = arr.min()
>>>         result = (arr - min_) / (max_ - min_)
>>> #
>>> for timer in ti.reset('kwarray.normalize(uint8)':
>>>     with timer:
>>>         kwarray.normalize(arr)
>>> #
>>> for timer in ti.reset('kwarray.normalize(uint8, inplace)':
>>>     with timer:
>>>         kwarray.normalize(arr, out=out)
>>> print('ti.rankings = {}'.format(ub.urepr(

```

(continues on next page)

(continued from previous page)

```
>>> ti.rankings, nl=2, align=':', precision=5)))
```



`kwarray.one_hot_embedding(labels, num_classes, dim=1)`

Embedding labels to one-hot form.

Parameters

- **labels** – (LongTensor) class labels, sized [N,].
- **num_classes** – (int) number of classes.
- **dim** (*int*) – dimension which will be created, if negative

Returns

encoded labels, sized [N,#classes].

Return type

Tensor

References

<https://discuss.pytorch.org/t/convert-int-into-one-hot-format/507/4>

Example

```
>>> # each element in target has to have 0 <= value < C
>>> # xdoctest: +REQUIRES(module:torch)
>>> import torch
>>> labels = torch.LongTensor([0, 0, 1, 4, 2, 3])
>>> num_classes = max(labels) + 1
>>> t = one_hot_embedding(labels, num_classes)
>>> assert all(row[y] == 1 for row, y in zip(t.numpy(), labels.numpy()))
>>> import ubelt as ub
>>> print(ub.urepr(t.numpy().tolist()))
[
  [1.0, 0.0, 0.0, 0.0, 0.0],
  [1.0, 0.0, 0.0, 0.0, 0.0],
  [0.0, 1.0, 0.0, 0.0, 0.0],
  [0.0, 0.0, 0.0, 0.0, 1.0],
  [0.0, 0.0, 1.0, 0.0, 0.0],
  [0.0, 0.0, 0.0, 1.0, 0.0],
]
>>> t2 = one_hot_embedding(labels.numpy(), num_classes)
>>> assert np.all(t2 == t.numpy())
>>> from kwarray.util_torch import _torch_available_devices
>>> devices = _torch_available_devices()
>>> if devices:
>>>     device = devices[0]
>>>     try:
>>>         t3 = one_hot_embedding(labels.to(device), num_classes)
>>>     except RuntimeError:
>>>         pass
>>>     assert np.all(t3.cpu().numpy() == t.numpy())
```

Example

```
>>> # xdoctest: +REQUIRES(module:torch)
>>> import torch
>>> nC = num_classes = 3
>>> labels = (torch.rand(10, 11, 12) * nC).long()
>>> assert one_hot_embedding(labels, nC, dim=0).shape == (3, 10, 11, 12)
>>> assert one_hot_embedding(labels, nC, dim=1).shape == (10, 3, 11, 12)
>>> assert one_hot_embedding(labels, nC, dim=2).shape == (10, 11, 3, 12)
>>> assert one_hot_embedding(labels, nC, dim=3).shape == (10, 11, 12, 3)
>>> labels = (torch.rand(10, 11) * nC).long()
>>> assert one_hot_embedding(labels, nC, dim=0).shape == (3, 10, 11)
>>> assert one_hot_embedding(labels, nC, dim=1).shape == (10, 3, 11)
>>> labels = (torch.rand(10) * nC).long()
>>> assert one_hot_embedding(labels, nC, dim=0).shape == (3, 10)
>>> assert one_hot_embedding(labels, nC, dim=1).shape == (10, 3)
```

`kwarray.one_hot_lookup(data, indices)`

Return value of a particular column for each row in data.

Each item in labels corresponds to a row in data. Returns the index specified at each row.

Parameters

- **data** (*ArrayLike*) – N x C float array of values
- **indices** (*ArrayLike*) – N integer array between 0 and C. This is an column index for each row in data.

Returns

the selected probability for each row

Return type

ArrayLike

Note: This is functionally equivalent to `[row[c] for row, c in zip(data, indices)]` except that it works with pure matrix operations.

Todo:

- [] Allow the user to specify which dimension indices should be zipped over. By default it should be dim=0
 - [] Allow the user to specify which dimension indices should select from. By default it should be dim=1.
-

Example

```
>>> from kwarray.util_torch import * # NOQA
>>> data = np.array([
>>>     [0, 1, 2],
>>>     [3, 4, 5],
>>>     [6, 7, 8],
>>>     [9, 10, 11],
>>> ])
>>> indices = np.array([0, 1, 2, 1])
>>> res = one_hot_lookup(data, indices)
>>> print('res = {!r}'.format(res))
res = array([ 0,  4,  8, 10])
>>> alt = np.array([row[c] for row, c in zip(data, indices)])
>>> assert np.all(alt == res)
```

Example

```
>>> # xdoctest: +REQUIRES(module:torch)
>>> import torch
>>> data = torch.from_numpy(np.array([
>>>     [0, 1, 2],
>>>     [3, 4, 5],
>>>     [6, 7, 8],
>>>     [9, 10, 11],
>>> ]))
>>> indices = torch.from_numpy(np.array([0, 1, 2, 1])).long()
>>> res = one_hot_lookup(data, indices)
>>> print('res = {!r}'.format(res))
res = tensor([ 0,  4,  8, 10]...)
>>> alt = torch.LongTensor([row[c] for row, c in zip(data, indices)])
>>> assert torch.all(alt == res)
```

`kwarrray.padded_slice(data, slices, pad=None, padkw=None, return_info=False)`

Allows slices with out-of-bound coordinates. Any out of bounds coordinate will be sampled via padding.

Parameters

- **data** (*Sliceable*) – data to slice into. Any channels must be the last dimension.
- **slices** (*slice* | *Tuple[slice, ...]*) – slice for each dimensions
- **ndim** (*int*) – number of spatial dimensions
- **pad** (*List[int|Tuple]*) – additional padding of the slice
- **padkw** (*Dict*) – if unspecified defaults to `{'mode': 'constant'}`
- **return_info** (*bool*, *default=False*) – if True, return extra information about the transform.

Note: Negative slices have a different meaning here then they usually do. Normally, they indicate a wrap-around or a reversed stride, but here they index into out-of-bounds space (which depends on the pad mode). For example a slice of -2:1 literally samples two pixels to the left of the data and one pixel from the data, so you get two padded values and one data value.

SeeAlso:

`embed_slice` - finds the embedded slice and padding

Returns

data_sliced: subregion of the input data (possibly with padding,
depending on if the original slice went out of bounds)

Tuple[Sliceable, Dict] :

`data_sliced` : as above

`transform` : information on how to return to the original coordinates

Currently a dict containing:

st_dims: a list indicating the low and high space-time
coordinate values of the returned data slice.

The structure of this dictionary mach change in the future

Return type
Sliceable

Example

```
>>> import kwarray
>>> data = np.arange(5)
>>> slices = [slice(-2, 7)]
```

```
>>> data_sliced = kwarray.padded_slice(data, slices)
>>> print(ub.urepr(data_sliced, with_dtype=False))
np.array([0, 0, 0, 1, 2, 3, 4, 0, 0])
```

```
>>> data_sliced = kwarray.padded_slice(data, slices, pad=[(3, 3)])
>>> print(ub.urepr(data_sliced, with_dtype=False))
np.array([0, 0, 0, 0, 0, 0, 1, 2, 3, 4, 0, 0, 0, 0, 0])
```

```
>>> data_sliced = kwarray.padded_slice(data, slice(3, 4), pad=[(1, 0)])
>>> print(ub.urepr(data_sliced, with_dtype=False))
np.array([2, 3])
```

`kwarray.random_combinations(items, size, num=None, rng=None)`

Yields num combinations of length size from items in random order

Parameters

- **items** (*List*) – pool of items to choose from
- **size** (*int*) – Number of items in each combination
- **num** (*int* | *None*) – Number of combinations to generate. If *None*, generate them all.
- **rng** (*int* | *float* | *None* | *numpy.random.RandomState* | *random.Random*) – seed or random number generator. Defaults to the global state of the python random module.

Yields

Tuple – a random combination of items of length size.

Example

```
>>> # xdoctest: +REQUIRES(module:scipy)
>>> import ubelt as ub
>>> items = list(range(10))
>>> size = 3
>>> num = 5
>>> rng = 0
>>> # xdoctest: +IGNORE_WANT
>>> combos = list(random_combinations(items, size, num, rng))
>>> print('combos = {}'.format(ub.urepr(combos, nl=1)))
combos = [
    (0, 6, 9),
    (4, 7, 8),
    (4, 6, 7),
```

(continues on next page)

(continued from previous page)

```
(2, 3, 5),
(1, 2, 4),
]
```

Example

```
>>> # xdoctest: +REQUIRES(module:scipy)
>>> import ubelt as ub
>>> items = list(zip(range(10), range(10)))
>>> # xdoctest: +IGNORE_WANT
>>> combos = list(random_combinations(items, 3, num=5, rng=0))
>>> print('combos = {}'.format(ub.urepr(combos, nl=1)))
combos = [
    ((0, 0), (6, 6), (9, 9)),
    ((4, 4), (7, 7), (8, 8)),
    ((4, 4), (6, 6), (7, 7)),
    ((2, 2), (3, 3), (5, 5)),
    ((1, 1), (2, 2), (4, 4)),
]
```

`kwarray.random_product(items, num=None, rng=None)`

Yields num items from the cartesian product of items in a random order.

Parameters

- **items** (*List[Sequence]*) – items to get cartesian product of packed in a list or tuple. (note this deviates from api of `itertools.product()`)
- **num** (*int | None*) – maximum number of items to generate. If None generate them all
- **rng** (*int | float | None | numpy.random.RandomState | random.Random*) – Seed or random number generator. Defaults to the global state of the python random module.

Yields

Tuple – a random item in the cartesian product

Example

```
>>> import ubelt as ub
>>> items = [(1, 2, 3), (4, 5, 6, 7)]
>>> rng = 0
>>> # xdoctest: +IGNORE_WANT
>>> products = list(random_product(items, rng=0))
>>> print(ub.urepr(products, nl=0))
[(3, 4), (1, 7), (3, 6), (2, 7), ... (1, 6), (2, 5), (2, 4)]
>>> products = list(random_product(items, num=3, rng=0))
>>> print(ub.urepr(products, nl=0))
[(3, 4), (1, 7), (3, 6)]
```


Example

```
>>> # xdoctest: +REQUIRES(--profile)
>>> rng = ensure_rng(0)
>>> items = [np.array([15, 14]), np.array([27, 26]),
>>>          np.array([21, 22]), np.array([32, 31])]
>>> num = 2
>>> for _ in range(100):
>>>     list(random_product(items, num=num, rng=rng))
```

`kwarray.robust_normalize(imdata, return_info=False, nodata=None, axis=None, dtype=<class 'numpy.float32'>, params='auto', mask=None)`

Normalize data intensities using heuristics to help put sensor data with extremely high or low contrast into a visible range.

This function is designed with an emphasis on getting something that is reasonable for visualization.

Todo:

- [x] Move to kwarray and renamed to robust_normalize?
- [] Support for M-estimators?

Parameters

- **imdata** (*ndarray*) – raw intensity data
- **return_info** (*bool*) – if True, return information about the chosen normalization heuristic.
- **params** (*str* | *dict*) – Can contain keys, low, high, or mid, scaling, extrema e.g. {'low': 0.1, 'mid': 0.8, 'high': 0.9, 'scaling': 'sigmoid'} See documentation in [find_robust_normalizers\(\)](#).
- **axis** (*None* | *int*) – The axis to normalize over, if unspecified, normalize jointly
- **nodata** (*None* | *int*) – A value representing nodata to leave unchanged during normalization, for example 0
- **dtype** (*type*) – can be float32 or float64
- **mask** (*ndarray* | *None*) – A mask indicating what pixels are valid and what pixels should be considered nodata. Mutually exclusive with nodata argument. A mask value of 1 indicates a VALID pixel. A mask value of 0 indicates an INVALID pixel. Note this is the opposite of a masked array.

Returns

a floating point array with values between 0 and 1. if return_info is specified, also returns extra data

Return type

ndarray | *Tuple*[*ndarray*, *Any*]

Note: This is effectively a combination of [find_robust_normalizers\(\)](#) and [normalize\(\)](#).

Example

```
>>> # xdoctest: +REQUIRES(module:scipy)
>>> from kwarray.util_robust import * # NOQA
>>> from kwarray.distributions import Mixture
>>> import ubelt as ub
>>> # A random mixture distribution for testing
>>> data = Mixture.random(6).sample(3000)
>>> param_basis = {
>>>     'scaling': ['linear', 'sigmoid'],
>>>     'high': [0.6, 0.8, 0.9, 1.0],
>>> }
>>> param_grid = list(ub.named_product(param_basis))
>>> param_grid += ['auto']
>>> param_grid += ['tukey']
>>> rows = []
>>> rows.append({'key': 'orig', 'result': data})
>>> for params in param_grid:
>>>     key = ub.urepr(params, compact=1)
>>>     result, info = robust_normalize(data, return_info=True, params=params)
>>>     print('key = {}'.format(key))
>>>     print('info = {}'.format(ub.urepr(info, nl=1)))
>>>     rows.append({'key': key, 'info': info, 'result': result})
>>> # xdoctest: +REQUIRES(--show)
>>> import seaborn as sns
>>> import kwplot
>>> kwplot.autompl()
>>> pnum_ = kwplot.PlotNums(nSubplots=len(rows))
>>> for row in rows:
>>>     ax = kwplot.figure(fnum=1, pnum=pnum_()).gca()
>>>     sns.histplot(data=row['result'], kde=True, bins=128, ax=ax, stat='density')
>>>     ax.set_title(row['key'])
```

Example

```
>>> # xdoctest: +REQUIRES(module:kwimage)
>>> from kwarray.util_robust import * # NOQA
>>> import ubelt as ub
>>> import kwimage
>>> import kwarray
>>> s = 512
>>> bit_depth = 11
>>> dtype = np.uint16
>>> max_val = int(2 ** bit_depth)
>>> min_val = int(0)
>>> rng = kwarray.ensure_rng(0)
>>> background = np.random.randint(min_val, max_val, size=(s, s), dtype=dtype)
>>> poly1 = kwimage.Polygon.random(rng=rng).scale(s / 2)
>>> poly2 = kwimage.Polygon.random(rng=rng).scale(s / 2).translate(s / 2)
>>> foreground = np.zeros_like(background, dtype=np.uint8)
>>> foreground = poly1.fill(foreground, value=255)
```

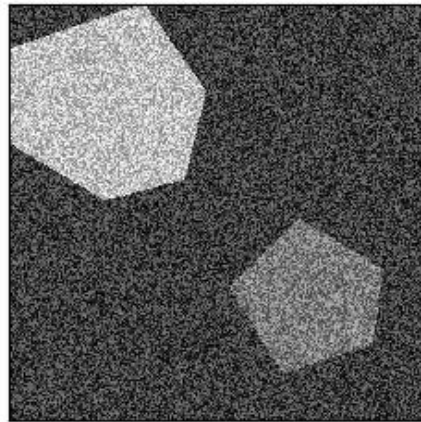
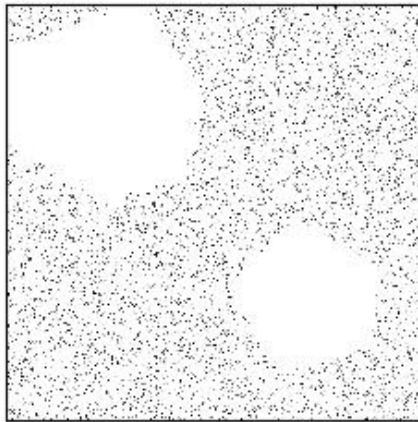
(continues on next page)

(continued from previous page)

```

>>> foreground = poly2.fill(forground, value=122)
>>> foreground = (kwimage.ensure_float01(forground) * max_val).astype(dtype)
>>> imdata = background + foreground
>>> normed, info = kwarray.robust_normalize(imdata, return_info=True)
>>> print('info = {}'.format(ub.urepr(info, nl=1)))
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(imdata, pnum=(1, 2, 1), fnum=1)
>>> kwplot.imshow(normed, pnum=(1, 2, 2), fnum=1)

```



`kwarray.seed_global(seed, offset=0)`

Seeds the python, numpy, and torch global random states

Parameters

- **seed** (*int*) – seed to use
- **offset** (*int*) – if specified, uses a different seed for each global random state separated by this offset. Defaults to 0.

`kwarray.setcover(candidate_sets_dict, items=None, set_weights=None, item_values=None, max_weight=None, algo='approx')`

Finds a feasible solution to the minimum weight maximum value set cover. The quality and runtime of the solution will depend on the backend algorithm selected.

Parameters

- **candidate_sets_dict** (*Dict[KT, List[VT]]*) – a dictionary where keys are the candidate set ids and each value is a candidate cover set.
- **items** (*Optional[VT]*) – the set of all items to be covered, if not specified, it is inferred from the candidate cover sets
- **set_weights** (*Optional[Dict[KT, float]]*) – maps candidate set ids to a cost for using this candidate cover in the solution. If not specified the weight of each candidate cover defaults to 1.
- **item_values** (*Optional[Dict[VT, float]]*) – maps each item to a value we get for returning this item in the solution. If not specified the value of each item defaults to 1.
- **max_weight** (*Optional[float]*) – if specified, the total cost of the returned cover is constrained to be less than this number.
- **algo** (*str*) – specifies which algorithm to use. Can either be ‘approx’ for the greedy solution or ‘exact’ for the globally optimal solution. Note the ‘exact’ algorithm solves an integer-linear-program, which can be very slow and requires the *pulp* package to be installed.

Returns

a subdict of candidate_sets_dict containing the chosen solution.

Return type

Dict

Example

```
>>> candidate_sets_dict = {
>>>     'a': [1, 2, 3, 8, 9, 0],
>>>     'b': [1, 2, 3, 4, 5],
>>>     'c': [4, 5, 7],
>>>     'd': [5, 6, 7],
>>>     'e': [6, 7, 8, 9, 0],
>>> }
>>> greedy_soln = setcover(candidate_sets_dict, algo='greedy')
>>> print('greedy_soln = {}'.format(ub.urepr(greedy_soln, nl=0)))
greedy_soln = {'a': [1, 2, 3, 8, 9, 0], 'c': [4, 5, 7], 'd': [5, 6, 7]}
>>> # xdoc: +REQUIRES(module:pulp)
>>> exact_soln = setcover(candidate_sets_dict, algo='exact')
>>> print('exact_soln = {}'.format(ub.urepr(exact_soln, nl=0)))
exact_soln = {'b': [1, 2, 3, 4, 5], 'e': [6, 7, 8, 9, 0]}
```

kwarray.**shuffle**(items, rng=None)

Shuffles a list inplace and then returns it for convinience

Parameters

- **items** (*list | ndarray*) – data to shuffle
- **rng** (*int | float | None | numpy.random.RandomState | random.Random*) – seed or random number gen

Returns

this is the input, but returned for convinience

Return type

list

Example

```
>>> list1 = [1, 2, 3, 4, 5, 6]
>>> list2 = shuffle(list(list1), rng=1)
>>> assert list1 != list2
>>> result = str(list2)
>>> print(result)
[3, 2, 5, 1, 4, 6]
```

`kwarray.standard_normal(size, mean=0, std=1, dtype=<class 'float'>, rng=<module 'numpy.random' from '/home/docs/checkouts/readthedocs.org/user_builds/kwarray/envs/latest/lib/python3.11/site-packages/numpy/random/__init__.py'>)`

Draw samples from a standard Normal distribution with a specified mean and standard deviation.

Parameters

- **size** (*int* | *Tuple[int, ...]*) – shape of the returned ndarray
- **mean** (*float*) – mean of the normal distribution. defaults to 0
- **std** (*float*) – standard deviation of the normal distribution. defaults to 1.
- **dtype** (*type*) – either `np.float32` (default) or `np.float64`
- **rng** (*numpy.random.RandomState*) – underlying random state

Returns

normally distributed random numbers with chosen size and dtype Extended typing `NDArray[Literal[size], Literal[dtype]]`

Return type

ndarray

Benchmark

```
>>> from timerit import Timerit
>>> import kwarray
>>> size = (300, 300, 3)
>>> for timer in Timerit(100, bestof=10, label='dtype=np.float32'):
>>>     rng = kwarray.ensure_rng(0)
>>>     with timer:
>>>         ours = standard_normal(size, rng=rng, dtype=np.float32)
>>> # Timed best=4.705 ms, mean=4.75 ± 0.085 ms for dtype=np.float32
>>> for timer in Timerit(100, bestof=10, label='dtype=np.float64'):
>>>     rng = kwarray.ensure_rng(0)
>>>     with timer:
>>>         theirs = standard_normal(size, rng=rng, dtype=np.float64)
>>> # Timed best=9.327 ms, mean=9.794 ± 0.4 ms for rng=np.float64
```

`kwarray.standard_normal32(size, mean=0, std=1, rng=<module 'numpy.random' from '/home/docs/checkouts/readthedocs.org/user_builds/kwarray/envs/latest/lib/python3.11/site-packages/numpy/random/__init__.py'>)`

Fast normally distributed random variables.

Uses the Box–Muller transform [[WikiBoxMuller](#)].

The difference between this function and `numpy.random.standard_normal()` is that we use float32 arrays in the backend instead of float64. Halving the amount of bits that need to be manipulated can significantly reduce the execution time, and 32-bit precision is often good enough.

Parameters

- **size** (*int* | *Tuple[int, ...]*) – shape of the returned ndarray
- **mean** (*float*, *default=0*) – mean of the normal distribution
- **std** (*float*, *default=1*) – standard deviation of the normal distribution
- **rng** (*numpy.random.RandomState*) – underlying random state

Returns

normally distributed random numbers with chosen size.

Return type

ndarray[Any, Float32]

References

SeeAlso:

- `standard_normal()`
- `standard_normal64()`

Example

```
>>> # xdoctest: +REQUIRES(module:scipy)
>>> import scipy
>>> import scipy.stats
>>> pts = 1000
>>> # Our numbers are normally distributed with high probability
>>> rng = np.random.RandomState(28041990)
>>> ours_a = standard_normal32(pts, rng=rng)
>>> ours_b = standard_normal32(pts, rng=rng) + 2
>>> ours = np.concatenate((ours_a, ours_b)) # numerical stability?
>>> p = scipy.stats.normaltest(ours)[1]
>>> print('Probability our data is non-normal is: {:.4g}'.format(p))
Probability our data is non-normal is: 1.573e-14
>>> rng = np.random.RandomState(28041990)
>>> theirs_a = rng.standard_normal(pts)
>>> theirs_b = rng.standard_normal(pts) + 2
>>> theirs = np.concatenate((theirs_a, theirs_b))
>>> p = scipy.stats.normaltest(theirs)[1]
>>> print('Probability their data is non-normal is: {:.4g}'.format(p))
Probability their data is non-normal is: 3.272e-11
```

Example

```
>>> pts = 1000
>>> rng = np.random.RandomState(28041990)
>>> ours = standard_normal32(pts, mean=10, std=3, rng=rng)
>>> assert np.abs(ours.std() - 3.0) < 0.1
>>> assert np.abs(ours.mean() - 10.0) < 0.1
```

Example

```
>>> # Test an even and odd numbers of points
>>> assert standard_normal32(3).shape == (3,)
>>> assert standard_normal32(2).shape == (2,)
>>> assert standard_normal32(1).shape == (1,)
>>> assert standard_normal32(0).shape == (0,)
>>> assert standard_normal32((3, 1)).shape == (3, 1)
>>> assert standard_normal32((3, 0)).shape == (3, 0)
```

`kwarray.standard_normal64(size, mean=0, std=1, rng=<module 'numpy.random' from
'/home/docs/checkouts/readthedocs.org/user_builds/kwarray/envs/latest/lib/python3.11/site-packages/numpy/random/__init__.py'>)`

Simple wrapper around `rng.standard_normal` to make an API compatible with `standard_normal32()`.

Parameters

- **size** (*int* | *Tuple[int, ...]*) – shape of the returned ndarray
- **mean** (*float*) – mean of the normal distribution. defaults to 0
- **std** (*float*) – standard deviation of the normal distribution. defaults to 1.
- **rng** (*numpy.random.RandomState*) – underlying random state

Returns

normally distributed random numbers with chosen size.

Return type

ndarray[Any, Float64]

SeeAlso:

- `standard_normal`
- `standard_normal32`

Example

```
>>> pts = 1000
>>> rng = np.random.RandomState(28041994)
>>> out = standard_normal64(pts, mean=10, std=3, rng=rng)
>>> assert np.abs(out.std() - 3.0) < 0.1
>>> assert np.abs(out.mean() - 10.0) < 0.1
```

`kwarrray.stats_dict(inputs, axis=None, nan=False, sum=False, extreme=True, n_extreme=False, median=False, shape=True, size=False, quantile='auto')`

Describe statistics about an input array

Parameters

- **inputs** (*ArrayLike*) – set of values to get statistics of
- **axis** (*int*) – if **inputs** is ndarray then this specifies the axis
- **nan** (*bool*) – report number of nan items (TODO: rename to skipna)
- **sum** (*bool*) – report sum of values
- **extreme** (*bool*) – report min and max values
- **n_extreme** (*bool*) – report extreme value frequencies
- **median** (*bool*) – report median
- **size** (*bool*) – report array size
- **shape** (*bool*) – report array shape
- **quantile** (*str | bool | List[float]*) – defaults to 'auto'. Can also be a list of quantiles to compute. if truthy computes quantiles.

Returns

dictionary of common numpy statistics (min, max, mean, std, nMin, nMax, shape)

Return type

`collections.OrderedDict`

SeeAlso:

`scipy.stats.describe()` `pandas.DataFrame.describe()`

Example

```
>>> # xdoctest: +IGNORE_WHITESPACE
>>> from kwarrray.util_averages import * # NOQA
>>> axis = 0
>>> rng = np.random.RandomState(0)
>>> inputs = rng.rand(10, 2).astype(np.float32)
>>> stats = stats_dict(inputs, axis=axis, nan=False, median=True)
>>> import ubelt as ub # NOQA
>>> result = str(ub.urepr(stats, nl=1, precision=4, with_dtype=True))
>>> print(result)
{
  'mean': np.array([[0.5206, 0.6425]], dtype=np.float32),
  'std': np.array([[0.2854, 0.2517]], dtype=np.float32),
  'min': np.array([[0.0202, 0.0871]], dtype=np.float32),
  'max': np.array([[0.9637, 0.9256]], dtype=np.float32),
  'q_0.25': np.array([0.4271, 0.5329], dtype=np.float64),
  'q_0.50': np.array([0.5584, 0.6805], dtype=np.float64),
  'q_0.75': np.array([0.7343, 0.8607], dtype=np.float64),
  'med': np.array([0.5584, 0.6805], dtype=np.float32),
  'shape': (10, 2),
}
```


Example

```
>>> # xdoctest: +IGNORE_WHITESPACE
>>> from kwarray.util_averages import * # NOQA
>>> axis = 0
>>> rng = np.random.RandomState(0)
>>> inputs = rng.randint(0, 42, size=100).astype(np.float32)
>>> inputs[4] = np.nan
>>> stats = stats_dict(inputs, axis=axis, nan=True, quantile='auto')
>>> import ubelt as ub # NOQA
>>> result = str(ub.urepr(stats, nl=0, precision=1, strkeys=True))
>>> print(result)
```

Example

```
>>> import kwarray
>>> import ubelt as ub
>>> rng = kwarray.ensure_rng(0)
>>> orig_inputs = rng.rand(1, 1, 2, 3)
>>> param_grid = ub.named_product({
>>>     # 'axis': (None, 0, (0, 1), -1),
>>>     'axis': [None],
>>>     'percent_nan': [0, 0.5, 1.0],
>>>     'nan': [True, False],
>>>     'sum': [1],
>>>     'extreme': [True],
>>>     'n_extreme': [True],
>>>     'median': [1],
>>>     'size': [1],
>>>     'shape': [1],
>>>     'quantile': ['auto'],
>>> })
>>> for params in param_grid:
>>>     kwargs = params.copy()
>>>     percent_nan = kwargs.pop('percent_nan', 0)
>>>     if percent_nan:
>>>         inputs = orig_inputs.copy()
>>>         inputs[rng.rand(*inputs.shape) < percent_nan] = np.nan
>>>     else:
>>>         inputs = orig_inputs
>>>     stats = kwarray.stats_dict(inputs, **kwargs)
>>>     print('---')
>>>     print('params = {}'.format(ub.urepr(params, nl=1)))
>>>     print('stats = {}'.format(ub.urepr(stats, nl=1)))
```

`kwarray.uniform`(*low*=0.0, *high*=1.0, *size*=None, *dtype*=<class 'numpy.float32'>, *rng*=<module 'numpy.random' from
 /home/docs/checkouts/readthedocs.org/user_builds/kwarray/envs/latest/lib/python3.11/site-packages/numpy/random/__init__.py'>)

Draws float32 samples from a uniform distribution.

Samples are uniformly distributed over the half-open interval [*low*, *high*) (includes *low*, but excludes *high*).

Parameters

- **low** (*float*) – Lower boundary of the output interval. All values generated will be greater than or equal to low. Defaults to 0.
- **high** (*float*) – Upper boundary of the output interval. All values generated will be less than high. Default to 1.
- **size** (*int* | *Tuple[int, ...]* | *None*) – Output shape. If the given shape is, e.g., (m, n, k), then $m * n * k$ samples are drawn. If size is None (default), a single value is returned if low and high are both scalars. Otherwise, `np.broadcast(low, high).size` samples are drawn.
- **dtype** (*type*) – either `np.float32` or `np.float64`. Defaults to `float32`
- **rng** (*numpy.random.RandomState*) – underlying random state

Returns

uniformly distributed random numbers with chosen size and dtype Extended typing
`NDArray[Literal[size], Literal[dtype]]`

Return type

`ndarray`

Benchmark

```
>>> from timerit import Timerit
>>> import kwarray
>>> size = (300, 300, 3)
>>> for timer in Timerit(100, bestof=10, label='dtype=np.float32'):
>>>     rng = kwarray.ensure_rng(0)
>>>     with timer:
>>>         ours = standard_normal(size, rng=rng, dtype=np.float32)
>>> # Timed best=4.705 ms, mean=4.75 ± 0.085 ms for dtype=np.float32
>>> for timer in Timerit(100, bestof=10, label='dtype=np.float64'):
>>>     rng = kwarray.ensure_rng(0)
>>>     with timer:
>>>         theirs = standard_normal(size, rng=rng, dtype=np.float64)
>>> # Timed best=9.327 ms, mean=9.794 ± 0.4 ms for rng=np.float64
```

`kwarray.uniform32(low=0.0, high=1.0, size=None, rng=<module 'numpy.random' from
'/home/docs/checkouts/readthedocs.org/user_builds/kwarray/envs/latest/lib/python3.11/site-
packages/numpy/random/__init__.py'>)`

Draws float32 samples from a uniform distribution.

Samples are uniformly distributed over the half-open interval [low, high) (includes low, but excludes high).

Parameters

- **low** (*float, default=0.0*) – Lower boundary of the output interval. All values generated will be greater than or equal to low.
- **high** (*float, default=1.0*) – Upper boundary of the output interval. All values generated will be less than high.
- **size** (*int* | *Tuple[int, ...]* | *None*) – Output shape. If the given shape is, e.g., (m, n, k), then $m * n * k$ samples are drawn. If size is None (default), a single value is returned if low and high are both scalars. Otherwise, `np.broadcast(low, high).size` samples are drawn.

Returns

uniformly distributed random numbers with chosen size.

Return type

ndarray[Any, Float32]

Example

```
>>> rng = np.random.RandomState(0)
>>> uniform32(low=0.0, high=1.0, size=None, rng=rng)
0.5488...
>>> uniform32(low=0.0, high=1.0, size=2000, rng=rng).sum()
1004.94...
>>> uniform32(low=-10, high=10.0, size=2000, rng=rng).sum()
202.44...
```

Benchmark

```
>>> from timerit import Timerit
>>> import kwarray
>>> size = 512 * 512
>>> for timer in Timerit(100, bestof=10, label='theirs: dtype=np.float64'):
>>>     rng = kwarray.ensure_rng(0)
>>>     with timer:
>>>         theirs = rng.uniform(size=size)
>>> for timer in Timerit(100, bestof=10, label='theirs: dtype=np.float32'):
>>>     rng = kwarray.ensure_rng(0)
>>>     with timer:
>>>         theirs = rng.rand(size).astype(np.float32)
>>> for timer in Timerit(100, bestof=10, label='ours: dtype=np.float32'):
>>>     rng = kwarray.ensure_rng(0)
>>>     with timer:
>>>         ours = uniform32(size=size)
```

`kwarray.unique_rows(arr, ordered=False, return_index=False)`

Like `unique`, but works on rows

Parameters

- **arr** (*NDArray*) – must be a contiguous C style array
- **ordered** (*bool*) – if true, keeps relative ordering

References

<https://stackoverflow.com/questions/16970982/find-unique-rows-in-numpy-array>

Example

```
>>> import kwarray
>>> from kwarray.util_numpy import * # NOQA
>>> rng = kwarray.ensure_rng(0)
>>> arr = rng.randint(0, 2, size=(22, 3))
>>> arr_unique = unique_rows(arr)
>>> print('arr_unique = {!r}'.format(arr_unique))
>>> arr_unique, idxs = unique_rows(arr, return_index=True, ordered=True)
>>> assert np.all(arr[idxs] == arr_unique)
>>> print('arr_unique = {!r}'.format(arr_unique))
>>> print('idxs = {!r}'.format(idxs))
>>> arr_unique, idxs = unique_rows(arr, return_index=True, ordered=False)
>>> assert np.all(arr[idxs] == arr_unique)
>>> print('arr_unique = {!r}'.format(arr_unique))
>>> print('idxs = {!r}'.format(idxs))
```

INDICES AND TABLES

- `genindex`
- `modindex`

BIBLIOGRAPHY

- [WikiMaxCov] https://en.wikipedia.org/wiki/Maximum_coverage_problem
- [DiscVsCont] <https://www.quora.com/Can-a-random-variable-be-both-discrete-and-continuous>
- [StephensMixture] https://stephens999.github.io/fiveMinuteStats/intro_to_mixture_models.html
- [GrosseMixture] https://www.cs.toronto.edu/~rgrosse/csc321/mixture_models.pdf
- [WikiNormal] https://en.wikipedia.org/wiki/Normal_distribution
- [WikiCLT] https://en.wikipedia.org/wiki/Central_limit_theorem
- [WikiBoxMuller] https://en.wikipedia.org/wiki/Box%E2%80%93Muller_transform
- [WikiRichardsCurve] https://en.wikipedia.org/wiki/Generalised_logistic_function
- [SO44313620] <https://stackoverflow.com/questions/44313620/convert-randomstate>
- [SO44313620] <https://stackoverflow.com/questions/44313620/convert-randomstate>
- [OxfordShapeSpread] <http://mathcenter.oxford.emory.edu/site/math117/shapeCenterAndSpread/>
- [YTFindOutliers] <https://www.youtube.com/watch?v=zY1WFMAA-ec>
- [WikiNorm] [https://en.wikipedia.org/wiki/Normalization_\(image_processing\)](https://en.wikipedia.org/wiki/Normalization_(image_processing))
- [WikiRichardsCurve] https://en.wikipedia.org/wiki/Generalised_logistic_function
- [WikiNorm] [https://en.wikipedia.org/wiki/Normalization_\(image_processing\)](https://en.wikipedia.org/wiki/Normalization_(image_processing))
- [WikiBoxMuller] https://en.wikipedia.org/wiki/Box%E2%80%93Muller_transform

PYTHON MODULE INDEX

k

- [kwarray](#), [127](#)
- [kwarray.__init__](#), [1](#)
- [kwarray.algo_assignment](#), [4](#)
- [kwarray.algo_setcover](#), [8](#)
- [kwarray.arrayapi](#), [11](#)
- [kwarray.dataframe_light](#), [42](#)
- [kwarray.distributions](#), [48](#)
- [kwarray.fast_rand](#), [68](#)
- [kwarray.util_averages](#), [73](#)
- [kwarray.util_groups](#), [82](#)
- [kwarray.util_misc](#), [87](#)
- [kwarray.util_numpy](#), [88](#)
- [kwarray.util_random](#), [97](#)
- [kwarray.util_robust](#), [103](#)
- [kwarray.util_slices](#), [111](#)
- [kwarray.util_slider](#), [115](#)
- [kwarray.util_torch](#), [124](#)

Symbols

- `_BinOpMixin` (class in `kwarray.distributions`), 51
- `_ImplRegistry` (class in `kwarray.arrayapi`), 12
- `_RBinOpMixin` (class in `kwarray.distributions`), 51
- `_apimethod()` (in module `kwarray.arrayapi`), 12
- `_apimethod()` (`kwarray.arrayapi._ImplRegistry` method), 12
- `_apply_padding()` (in module `kwarray.util_slices`), 113
- `_apply_robust_normalizer()` (in module `kwarray.util_robust`), 107
- `_body_str()` (`kwarray.distributions.Parameterized` method), 50
- `_coerce_pad()` (in module `kwarray.util_slices`), 115
- `_coerce_rng_type()` (in module `kwarray.util_random`), 101
- `_coerce_timedelta()` (in module `kwarray.distributions`), 54
- `_combine_mean_stds()` (in module `kwarray.util_averages`), 78
- `_compute_stride()` (`kwarray.SlidingWindow` method), 144
- `_compute_stride()` (`kwarray.util_slider.SlidingWindow` method), 117
- `_custom_quantile_extreme_estimator()` (in module `kwarray.util_robust`), 105
- `_demodata()` (`kwarray.DataFrameLight` class method), 134
- `_demodata()` (`kwarray.dataframe_light.DataFrameLight` class method), 45
- `_ensure_datamethods_names_are_registered()` (`kwarray.arrayapi._ImplRegistry` method), 12
- `_generate_on_a_time_budget()` (in module `kwarray.distributions`), 54
- `_getcol()` (`kwarray.DataFrameLight` method), 134
- `_getcol()` (`kwarray.dataframe_light.DataFrameLight` method), 45
- `_getcols()` (`kwarray.DataFrameLight` method), 134
- `_getcols()` (`kwarray.dataframe_light.DataFrameLight` method), 45
- `_getrow()` (`kwarray.DataFrameLight` method), 134
- `_getrow()` (`kwarray.dataframe_light.DataFrameLight` method), 45
- `_gmean()` (in module `kwarray.util_averages`), 75
- `_implmethod()` (`kwarray.arrayapi._ImplRegistry` method), 12
- `_is_in_onnx_export()` (in module `kwarray.util_torch`), 124
- `_isinstance2()` (in module `kwarray.distributions`), 49
- `_issubclass2()` (in module `kwarray.distributions`), 49
- `_iter_basis_frac()` (`kwarray.SlidingWindow` method), 144
- `_iter_basis_frac()` (`kwarray.util_slider.SlidingWindow` method), 117
- `_make_body()` (`kwarray.distributions.Parameterized` method), 50
- `_no_keeptdim_indexer()` (in module `kwarray.util_averages`), 82
- `_npstate_to_pystate()` (in module `kwarray.util_random`), 101
- `_numpy` (`kwarray.ArrayAPI` attribute), 128
- `_numpy` (`kwarray.arrayapi.ArrayAPI` attribute), 38
- `_numpymethod()` (in module `kwarray.arrayapi`), 12
- `_pandas()` (`kwarray.DataFrameLight` method), 134
- `_pandas()` (`kwarray.dataframe_light.DataFrameLight` method), 45
- `_postprocess_keeptdms()` (in module `kwarray.util_averages`), 82
- `_process_docstrings()` (in module `kwarray.distributions`), 67
- `_pystate_to_npstate()` (in module `kwarray.util_random`), 101
- `_quantile_extreme_estimator()` (in module `kwarray.util_robust`), 105
- `_register()` (`kwarray.arrayapi._ImplRegistry` method), 12
- `_setchild()` (`kwarray.distributions.Parameterized` method), 49
- `_setcover_greedy_new()` (in module `kwarray.algo_setcover`), 9
- `_setcover_greedy_old()` (in module `kwarray.algo_setcover`), 8
- `_setcover_ilp()` (in module `kwarray.algo_setcover`), 10
- `_setparam()` (`kwarray.distributions.Parameterized` method), 49

method), 49
 _setparam() (kwarray.distributions.ParameterizedList
 method), 51
 _show() (kwarray.distributions.Distribution method), 53
 _slices1d() (in module kwarray.util_slider), 123
 _sumsq_std() (kwarray.RunningStats method), 141
 _sumsq_std() (kwarray.util_averages.RunningStats
 method), 78
 _test_distributions() (in module kwar-
 ray.distributions), 67
 _torch (kwarray.ArrayAPI attribute), 128
 _torch (kwarray.arrayapi.ArrayAPI attribute), 38
 _torch_available_devices() (in module kwar-
 ray.util_torch), 126
 _torch_dtype_lut() (in module kwarray.arrayapi), 41
 _torchmethod() (in module kwarray.arrayapi), 12
 _trysample() (in module kwarray.distributions), 59
 _tukey_quantile_fence() (in module kwar-
 ray.util_robust), 104
 _update_from_other() (kwarray.RunningStats
 method), 139
 _update_from_other() (kwar-
 ray.util_averages.RunningStats method),
 76
 _update_internals() (kwar-
 ray.distributions.TruncNormal method),
 64

A

abs() (kwarray.distributions._BinOpMixin method), 51
 add() (kwarray.Stitcher method), 149
 add() (kwarray.util_slider.Stitcher method), 122
 all() (kwarray.ArrayAPI static method), 130
 all() (kwarray.arrayapi.ArrayAPI static method), 40
 all() (kwarray.arrayapi.NumpyImpls static method), 31
 all() (kwarray.arrayapi.TorchImpls static method), 21
 any() (kwarray.ArrayAPI static method), 130
 any() (kwarray.arrayapi.ArrayAPI static method), 40
 any() (kwarray.arrayapi.NumpyImpls static method), 29
 any() (kwarray.arrayapi.TorchImpls static method), 21
 append() (kwarray.distributions.ParameterizedList
 method), 51
 apply_embedded_slice() (in module kwarray), 150
 apply_embedded_slice() (in module kwar-
 ray.util_slices), 113
 apply_grouping() (in module kwarray), 150
 apply_grouping() (in module kwarray.util_groups), 85
 arglexmax() (in module kwarray), 150
 arglexmax() (in module kwarray.util_numpy), 94
 argmax() (kwarray.ArrayAPI static method), 129
 argmax() (kwarray.arrayapi.ArrayAPI static method),
 39
 argmax() (kwarray.arrayapi.NumpyImpls static
 method), 26

argmax() (kwarray.arrayapi.TorchImpls static method),
 16
 argmaxima() (in module kwarray), 151
 argmaxima() (in module kwarray.util_numpy), 91
 argmin() (kwarray.ArrayAPI static method), 129
 argmin() (kwarray.arrayapi.ArrayAPI static method),
 39
 argmin() (kwarray.arrayapi.NumpyImpls static
 method), 26
 argmin() (kwarray.arrayapi.TorchImpls static method),
 16
 argminima() (in module kwarray), 152
 argminima() (in module kwarray.util_numpy), 92
 argsort() (kwarray.ArrayAPI static method), 129
 argsort() (kwarray.arrayapi.ArrayAPI static method),
 39
 argsort() (kwarray.arrayapi.NumpyImpls static
 method), 26
 argsort() (kwarray.arrayapi.TorchImpls static method),
 16
 array_equal() (kwarray.ArrayAPI static method), 130
 array_equal() (kwarray.arrayapi.ArrayAPI static
 method), 40
 array_equal() (kwarray.arrayapi.NumpyImpls static
 method), 28
 array_equal() (kwarray.arrayapi.TorchImpls static
 method), 20
 ArrayAPI (class in kwarray), 127
 ArrayAPI (class in kwarray.arrayapi), 37
 asarray() (kwarray.ArrayAPI static method), 129
 asarray() (kwarray.arrayapi.ArrayAPI static method),
 40
 asarray() (kwarray.arrayapi.NumpyImpls static
 method), 35
 asarray() (kwarray.arrayapi.TorchImpls static method),
 21
 astype() (kwarray.ArrayAPI static method), 129
 astype() (kwarray.arrayapi.ArrayAPI static method),
 40
 astype() (kwarray.arrayapi.NumpyImpls static
 method), 35
 astype() (kwarray.arrayapi.TorchImpls static method),
 21
 atleast_nd() (in module kwarray), 153
 atleast_nd() (in module kwarray.util_numpy), 90
 atleast_nd() (kwarray.ArrayAPI static method), 129
 atleast_nd() (kwarray.arrayapi.ArrayAPI static
 method), 39
 atleast_nd() (kwarray.arrayapi.NumpyImpls static
 method), 25
 atleast_nd() (kwarray.arrayapi.TorchImpls static
 method), 13
 average() (kwarray.Stitcher method), 149
 average() (kwarray.util_slider.Stitcher method), 122

B

Bernoulli (class in kwarray.distributions), 64
 Binomial (class in kwarray.distributions), 65
 boolmask() (in module kwarray), 154
 boolmask() (in module kwarray.util_numpy), 88

C

cast() (kwarray.distributions.Distribution class method), 53
 CastError (in module kwarray.distributions), 60
 cat() (kwarray.ArrayAPI static method), 128
 cat() (kwarray.arrayapi.ArrayAPI static method), 39
 cat() (kwarray.arrayapi.NumpyImpls static method), 25
 cat() (kwarray.arrayapi.TorchImpls static method), 13
 Categorical (class in kwarray.distributions), 65
 CategoryUniform (class in kwarray.distributions), 66
 ceil() (kwarray.ArrayAPI static method), 130
 ceil() (kwarray.arrayapi.ArrayAPI static method), 40
 ceil() (kwarray.arrayapi.NumpyImpls static method), 35
 ceil() (kwarray.arrayapi.TorchImpls static method), 22
 centers (kwarray.SlidingWindow property), 144
 centers (kwarray.util_slider.SlidingWindow property), 117
 children() (kwarray.distributions.Parameterized method), 49
 clear() (kwarray.dataframe_light.DataFrameLight method), 46
 clear() (kwarray.DataFrameLight method), 134
 clip() (kwarray.ArrayAPI static method), 130
 clip() (kwarray.arrayapi.ArrayAPI static method), 40
 clip() (kwarray.arrayapi.NumpyImpls static method), 36
 clip() (kwarray.arrayapi.TorchImpls static method), 22
 clip() (kwarray.distributions._BinOpMixin method), 51
 coerce() (kwarray.ArrayAPI static method), 128
 coerce() (kwarray.arrayapi.ArrayAPI static method), 39
 coerce() (kwarray.distributions.Bernoulli class method), 65
 coerce() (kwarray.distributions.DiscreteUniform class method), 62
 coerce() (kwarray.distributions.Distribution class method), 53
 coerce() (kwarray.distributions.Uniform class method), 60
 CoerceError, 60
 columns (kwarray.dataframe_light.DataFrameLight property), 45
 columns (kwarray.DataFrameLight property), 134
 Composed (class in kwarray.distributions), 57
 compress() (kwarray.ArrayAPI static method), 129
 compress() (kwarray.arrayapi.ArrayAPI static method), 39

compress() (kwarray.arrayapi.NumpyImpls static method), 25
 compress() (kwarray.arrayapi.TorchImpls static method), 13
 compress() (kwarray.dataframe_light.DataFrameArray method), 48
 compress() (kwarray.dataframe_light.DataFrameLight method), 46
 compress() (kwarray.DataFrameArray method), 130
 compress() (kwarray.DataFrameLight method), 134
 concat() (kwarray.dataframe_light.DataFrameLight class method), 46
 concat() (kwarray.DataFrameLight class method), 135
 Constant (class in kwarray.distributions), 61
 contiguous() (kwarray.ArrayAPI static method), 130
 contiguous() (kwarray.arrayapi.ArrayAPI static method), 40
 contiguous() (kwarray.arrayapi.NumpyImpls static method), 35
 contiguous() (kwarray.arrayapi.TorchImpls static method), 21
 ContinuousDistribution (class in kwarray.distributions), 54
 copy() (kwarray.ArrayAPI static method), 130
 copy() (kwarray.arrayapi.ArrayAPI static method), 40
 copy() (kwarray.arrayapi.NumpyImpls static method), 32
 copy() (kwarray.arrayapi.TorchImpls static method), 21
 copy() (kwarray.dataframe_light.DataFrameLight method), 46
 copy() (kwarray.DataFrameLight method), 135
 current() (kwarray.RunningStats method), 142
 current() (kwarray.util_averages.RunningStats method), 78

D

DataFrameArray (class in kwarray), 130
 DataFrameArray (class in kwarray.dataframe_light), 48
 DataFrameLight (class in kwarray), 130
 DataFrameLight (class in kwarray.dataframe_light), 42
 DiscreteDistribution (class in kwarray.distributions), 54
 DiscreteUniform (class in kwarray.distributions), 62
 Distribution (class in kwarray.distributions), 51
 dtype_info() (in module kwarray), 155
 dtype_info() (in module kwarray.arrayapi), 41
 dtype_kind() (kwarray.ArrayAPI static method), 130
 dtype_kind() (kwarray.arrayapi.ArrayAPI static method), 40
 dtype_kind() (kwarray.arrayapi.NumpyImpls static method), 35
 dtype_kind() (kwarray.arrayapi.TorchImpls static method), 22

E

`embed_slice()` (in module `kwarray`), 156
`embed_slice()` (in module `kwarray.util_slices`), 113
`empty()` (`kwarray.arrayapi.NumpyImpls` static method), 26
`empty()` (`kwarray.arrayapi.TorchImpls` static method), 16
`empty_like()` (`kwarray.ArrayAPI` static method), 129
`empty_like()` (`kwarray.arrayapi.ArrayAPI` static method), 39
`empty_like()` (`kwarray.arrayapi.NumpyImpls` static method), 25
`empty_like()` (`kwarray.arrayapi.TorchImpls` static method), 16
`ensure()` (`kwarray.arrayapi.NumpyImpls` static method), 35
`ensure()` (`kwarray.arrayapi.TorchImpls` static method), 22
`ensure_rng()` (in module `kwarray`), 158
`ensure_rng()` (in module `kwarray.util_random`), 101
`equal_with_nan()` (in module `kwarray`), 159
`equal_with_nan()` (in module `kwarray.util_numpy`), 96
`exp()` (`kwarray.distributions._BinOpMixin` method), 51
`Exponential` (class in `kwarray.distributions`), 60
`extend()` (`kwarray.dataframe_light.DataFrameArray` method), 48
`extend()` (`kwarray.dataframe_light.DataFrameLight` method), 46
`extend()` (`kwarray.DataFrameArray` method), 130
`extend()` (`kwarray.DataFrameLight` method), 135

F

`finalize()` (`kwarray.Stitcher` method), 150
`finalize()` (`kwarray.util_slider.Stitcher` method), 123
`find_robust_normalizers()` (in module `kwarray`), 159
`find_robust_normalizers()` (in module `kwarray.util_robust`), 103
`FlatIndexer` (class in `kwarray`), 137
`FlatIndexer` (class in `kwarray.util_misc`), 87
`floor()` (`kwarray.ArrayAPI` static method), 130
`floor()` (`kwarray.arrayapi.ArrayAPI` static method), 40
`floor()` (`kwarray.arrayapi.NumpyImpls` static method), 35
`floor()` (`kwarray.arrayapi.TorchImpls` static method), 22
`from_dict()` (`kwarray.dataframe_light.DataFrameLight` class method), 47
`from_dict()` (`kwarray.DataFrameLight` class method), 135
`from_pandas()` (`kwarray.dataframe_light.DataFrameLight` class method), 47

`from_pandas()` (`kwarray.DataFrameLight` class method), 135
`fromlist()` (`kwarray.FlatIndexer` class method), 137
`fromlist()` (`kwarray.util_misc.FlatIndexer` class method), 87
`full()` (`kwarray.arrayapi.NumpyImpls` static method), 26
`full()` (`kwarray.arrayapi.TorchImpls` static method), 16
`full_like()` (`kwarray.ArrayAPI` static method), 129
`full_like()` (`kwarray.arrayapi.ArrayAPI` static method), 39
`full_like()` (`kwarray.arrayapi.NumpyImpls` static method), 25
`full_like()` (`kwarray.arrayapi.TorchImpls` static method), 16

G

`generalized_logistic()` (in module `kwarray`), 161
`generalized_logistic()` (in module `kwarray.util_numpy`), 95
`get()` (`kwarray.dataframe_light.DataFrameLight` method), 45
`get()` (`kwarray.DataFrameLight` method), 134
`grid` (`kwarray.SlidingWindow` property), 144
`grid` (`kwarray.util_slider.SlidingWindow` property), 117
`group_consecutive()` (in module `kwarray`), 162
`group_consecutive()` (in module `kwarray.util_groups`), 85
`group_consecutive_indices()` (in module `kwarray`), 163
`group_consecutive_indices()` (in module `kwarray.util_groups`), 86
`group_indices()` (in module `kwarray`), 164
`group_indices()` (in module `kwarray.util_groups`), 83
`group_items()` (in module `kwarray`), 166
`group_items()` (in module `kwarray.util_groups`), 82
`groupby()` (`kwarray.dataframe_light.DataFrameLight` method), 47
`groupby()` (`kwarray.DataFrameLight` method), 135

H

`hstack()` (`kwarray.ArrayAPI` static method), 128
`hstack()` (`kwarray.arrayapi.ArrayAPI` static method), 39
`hstack()` (`kwarray.arrayapi.NumpyImpls` static method), 22
`hstack()` (`kwarray.arrayapi.TorchImpls` static method), 13

I

`iceil()` (`kwarray.ArrayAPI` static method), 130
`iceil()` (`kwarray.arrayapi.ArrayAPI` static method), 40
`iceil()` (`kwarray.arrayapi.NumpyImpls` static method), 35

- `iceil()` (*kwarray.arrayapi.TorchImpls static method*), 22
 - `idstr()` (*kwarray.distributions.Parameterized method*), 50
 - `idstr()` (*kwarray.distributions.ParameterizedList method*), 51
 - `ifloor()` (*kwarray.ArrayAPI static method*), 130
 - `ifloor()` (*kwarray.arrayapi.ArrayAPI static method*), 40
 - `ifloor()` (*kwarray.arrayapi.NumpyImpls static method*), 35
 - `ifloor()` (*kwarray.arrayapi.TorchImpls static method*), 22
 - `iloc` (*kwarray.dataframe_light.DataFrameLight property*), 44
 - `iloc` (*kwarray.DataFrameLight property*), 133
 - `impl()` (*kwarray.ArrayAPI static method*), 128
 - `impl()` (*kwarray.arrayapi.ArrayAPI static method*), 38
 - `int()` (*kwarray.distributions._BinOpMixin method*), 51
 - `iround()` (*kwarray.ArrayAPI static method*), 130
 - `iround()` (*kwarray.arrayapi.ArrayAPI static method*), 40
 - `iround()` (*kwarray.arrayapi.NumpyImpls static method*), 35
 - `iround()` (*kwarray.arrayapi.TorchImpls static method*), 22
 - `is_numpy` (*kwarray.arrayapi.NumpyImpls attribute*), 22
 - `is_numpy` (*kwarray.arrayapi.TorchImpls attribute*), 13
 - `is_tensor` (*kwarray.arrayapi.NumpyImpls attribute*), 22
 - `is_tensor` (*kwarray.arrayapi.TorchImpls attribute*), 13
 - `isect_flags()` (*in module kwarray*), 167
 - `isect_flags()` (*in module kwarray.util_numpy*), 90
 - `iter_reduce_ufunc()` (*in module kwarray*), 167
 - `iter_reduce_ufunc()` (*in module kwarray.util_numpy*), 89
 - `iterrows()` (*kwarray.dataframe_light.DataFrameLight method*), 47
 - `iterrows()` (*kwarray.DataFrameLight method*), 136
- ## J
- `json_id()` (*kwarray.distributions.Parameterized method*), 50
- ## K
- `keys()` (*kwarray.dataframe_light.DataFrameLight method*), 45
 - `keys()` (*kwarray.DataFrameLight method*), 134
 - `kron()` (*kwarray.arrayapi.NumpyImpls static method*), 37
 - `kwarray`
 - module, 127
 - `kwarray.__init__`
 - module, 1
 - `kwarray.algo_assignment`
 - module, 4
 - `kwarray.algo_setcover`
 - module, 8
 - `kwarray.arrayapi`
 - module, 11
 - `kwarray.dataframe_light`
 - module, 42
 - `kwarray.distributions`
 - module, 48
 - `kwarray.fast_rand`
 - module, 68
 - `kwarray.util_averages`
 - module, 73
 - `kwarray.util_groups`
 - module, 82
 - `kwarray.util_misc`
 - module, 87
 - `kwarray.util_numpy`
 - module, 88
 - `kwarray.util_random`
 - module, 97
 - `kwarray.util_robust`
 - module, 103
 - `kwarray.util_slices`
 - module, 111
 - `kwarray.util_slider`
 - module, 115
 - `kwarray.util_torch`
 - module, 124
- ## L
- `loc` (*kwarray.dataframe_light.DataFrameLight property*), 44
 - `loc` (*kwarray.DataFrameLight property*), 133
 - `LocLight` (*class in kwarray*), 138
 - `LocLight` (*class in kwarray.dataframe_light*), 42
 - `log` (*kwarray.arrayapi.NumpyImpls attribute*), 29
 - `log()` (*kwarray.ArrayAPI static method*), 130
 - `log()` (*kwarray.arrayapi.ArrayAPI static method*), 40
 - `log()` (*kwarray.arrayapi.TorchImpls static method*), 21
 - `log()` (*kwarray.distributions._BinOpMixin method*), 51
 - `log10()` (*kwarray.distributions._BinOpMixin method*), 51
 - `log2` (*kwarray.arrayapi.NumpyImpls attribute*), 29
 - `log2()` (*kwarray.ArrayAPI static method*), 130
 - `log2()` (*kwarray.arrayapi.ArrayAPI static method*), 40
 - `log2()` (*kwarray.arrayapi.TorchImpls static method*), 21
- ## M
- `matmul` (*kwarray.arrayapi.NumpyImpls attribute*), 26
 - `matmul()` (*kwarray.ArrayAPI static method*), 129
 - `matmul()` (*kwarray.arrayapi.ArrayAPI static method*), 40

matmul() (*kwarray.arrayapi.TorchImpls static method*), 21

max() (*kwarray.ArrayAPI static method*), 129

max() (*kwarray.arrayapi.ArrayAPI static method*), 39

max() (*kwarray.arrayapi.NumpyImpls static method*), 26

max() (*kwarray.arrayapi.TorchImpls static method*), 17

max_argmax() (*kwarray.ArrayAPI static method*), 129

max_argmax() (*kwarray.arrayapi.ArrayAPI static method*), 40

max_argmax() (*kwarray.arrayapi.NumpyImpls static method*), 26

max_argmax() (*kwarray.arrayapi.TorchImpls static method*), 17

maximum() (*kwarray.ArrayAPI static method*), 129

maximum() (*kwarray.arrayapi.ArrayAPI static method*), 40

maximum() (*kwarray.arrayapi.NumpyImpls static method*), 26

maximum() (*kwarray.arrayapi.TorchImpls static method*), 19

maxvalue_assignment() (*in module kwarray*), 168

maxvalue_assignment() (*in module kwarray.algo_assignment*), 6

min() (*kwarray.ArrayAPI static method*), 129

min() (*kwarray.arrayapi.ArrayAPI static method*), 39

min() (*kwarray.arrayapi.NumpyImpls static method*), 26

min() (*kwarray.arrayapi.TorchImpls static method*), 17

min_argmin() (*kwarray.ArrayAPI static method*), 129

min_argmin() (*kwarray.arrayapi.ArrayAPI static method*), 40

min_argmin() (*kwarray.arrayapi.NumpyImpls static method*), 26

min_argmin() (*kwarray.arrayapi.TorchImpls static method*), 18

mincost_assignment() (*in module kwarray*), 169

mincost_assignment() (*in module kwarray.algo_assignment*), 5

mindist_assignment() (*in module kwarray*), 170

mindist_assignment() (*in module kwarray.algo_assignment*), 4

minimum() (*kwarray.ArrayAPI static method*), 129

minimum() (*kwarray.arrayapi.ArrayAPI static method*), 40

minimum() (*kwarray.arrayapi.NumpyImpls static method*), 26

minimum() (*kwarray.arrayapi.TorchImpls static method*), 19

MixedDistribution (*class in kwarray.distributions*), 54

Mixture (*class in kwarray.distributions*), 54

module

- kwarray, 127
- kwarray.__init__, 1
- kwarray.algo_assignment, 4
- kwarray.algo_setcover, 8

- kwarray.arrayapi, 11
- kwarray.dataframe_light, 42
- kwarray.distributions, 48
- kwarray.fast_rand, 68
- kwarray.util_averages, 73
- kwarray.util_groups, 82
- kwarray.util_misc, 87
- kwarray.util_numpy, 88
- kwarray.util_random, 97
- kwarray.util_robust, 103
- kwarray.util_slices, 111
- kwarray.util_slider, 115
- kwarray.util_torch, 124

N

nan_to_num() (*kwarray.ArrayAPI static method*), 129

nan_to_num() (*kwarray.arrayapi.ArrayAPI static method*), 40

nan_to_num() (*kwarray.arrayapi.NumpyImpls static method*), 26

nan_to_num() (*kwarray.arrayapi.TorchImpls static method*), 21

NonlinearUniform (*class in kwarray.distributions*), 65

nonzero() (*kwarray.ArrayAPI static method*), 129

nonzero() (*kwarray.arrayapi.ArrayAPI static method*), 40

nonzero() (*kwarray.arrayapi.NumpyImpls static method*), 33

nonzero() (*kwarray.arrayapi.TorchImpls static method*), 21

Normal (*class in kwarray.distributions*), 62

normalize() (*in module kwarray*), 171

normalize() (*in module kwarray.util_robust*), 107

NoSupportError, 75, 138

numel() (*kwarray.ArrayAPI static method*), 129

numel() (*kwarray.arrayapi.ArrayAPI static method*), 39

numel() (*kwarray.arrayapi.NumpyImpls static method*), 25

numel() (*kwarray.arrayapi.TorchImpls static method*), 16

numpy() (*kwarray.ArrayAPI static method*), 129

numpy() (*kwarray.arrayapi.ArrayAPI static method*), 40

numpy() (*kwarray.arrayapi.NumpyImpls static method*), 35

numpy() (*kwarray.arrayapi.TorchImpls static method*), 21

NumpyImpls (*class in kwarray.arrayapi*), 22

O

one_hot_embedding() (*in module kwarray*), 175

one_hot_embedding() (*in module kwarray.util_torch*), 124

one_hot_lookup() (*in module kwarray*), 176

one_hot_lookup() (*in module kwarray.util_torch*), 125

`ones()` (*kwarray.arrayapi.NumpyImpls static method*), 26
`ones()` (*kwarray.arrayapi.TorchImpls static method*), 16
`ones_like()` (*kwarray.ArrayAPI static method*), 129
`ones_like()` (*kwarray.arrayapi.ArrayAPI static method*), 39
`ones_like()` (*kwarray.arrayapi.NumpyImpls static method*), 26
`ones_like()` (*kwarray.arrayapi.TorchImpls static method*), 16

P

`pad()` (*kwarray.ArrayAPI static method*), 130
`pad()` (*kwarray.arrayapi.ArrayAPI static method*), 40
`pad()` (*kwarray.arrayapi.NumpyImpls static method*), 35
`pad()` (*kwarray.arrayapi.TorchImpls static method*), 21
`padded_slice()` (*in module kwarray*), 178
`padded_slice()` (*in module kwarray.util_slices*), 111
`pandas()` (*kwarray.dataframe_light.DataFrameLight method*), 45
`pandas()` (*kwarray.DataFrameLight method*), 133
`Parameterized` (*class in kwarray.distributions*), 49
`ParameterizedList` (*class in kwarray.distributions*), 51
`parameters()` (*kwarray.distributions.Parameterized method*), 50
`PDF` (*class in kwarray.distributions*), 66
`plot()` (*kwarray.distributions.Distribution method*), 53

R

`random()` (*kwarray.distributions.Distribution class method*), 52
`random()` (*kwarray.distributions.Mixture class method*), 56
`random()` (*kwarray.distributions.Normal class method*), 63
`random()` (*kwarray.distributions.TruncNormal class method*), 64
`random_combinations()` (*in module kwarray*), 179
`random_combinations()` (*in module kwarray.util_random*), 99
`random_product()` (*in module kwarray*), 180
`random_product()` (*in module kwarray.util_random*), 100
`ravel()` (*kwarray.FlatIndexer method*), 138
`ravel()` (*kwarray.util_misc.FlatIndexer method*), 88
`rename()` (*kwarray.dataframe_light.DataFrameLight method*), 47
`rename()` (*kwarray.DataFrameLight method*), 136
`repeat()` (*kwarray.ArrayAPI static method*), 129
`repeat()` (*kwarray.arrayapi.ArrayAPI static method*), 39
`repeat()` (*kwarray.arrayapi.NumpyImpls static method*), 25

`repeat()` (*kwarray.arrayapi.TorchImpls static method*), 15
`reset_index()` (*kwarray.dataframe_light.DataFrameLight method*), 47
`reset_index()` (*kwarray.DataFrameLight method*), 135
`result_type()` (*kwarray.ArrayAPI static method*), 128
`result_type()` (*kwarray.arrayapi.ArrayAPI static method*), 39
`result_type()` (*kwarray.arrayapi.NumpyImpls static method*), 25
`result_type()` (*kwarray.arrayapi.TorchImpls static method*), 13
`robust_normalize()` (*in module kwarray*), 181
`robust_normalize()` (*in module kwarray.util_robust*), 105
`round()` (*kwarray.ArrayAPI static method*), 130
`round()` (*kwarray.arrayapi.ArrayAPI static method*), 40
`round()` (*kwarray.arrayapi.NumpyImpls static method*), 35
`round()` (*kwarray.arrayapi.TorchImpls static method*), 22
`round()` (*kwarray.distributions._BinOpMixin method*), 51
`RunningStats` (*class in kwarray*), 138
`RunningStats` (*class in kwarray.util_averages*), 75

S

`sample()` (*kwarray.distributions.Bernoulli method*), 65
`sample()` (*kwarray.distributions.Binomial method*), 65
`sample()` (*kwarray.distributions.Categorical method*), 65
`sample()` (*kwarray.distributions.CategoryUniform method*), 66
`sample()` (*kwarray.distributions.Composed method*), 59
`sample()` (*kwarray.distributions.Constant method*), 62
`sample()` (*kwarray.distributions.DiscreteUniform method*), 62
`sample()` (*kwarray.distributions.Distribution method*), 52
`sample()` (*kwarray.distributions.Exponential method*), 61
`sample()` (*kwarray.distributions.Mixture method*), 56
`sample()` (*kwarray.distributions.NonlinearUniform method*), 66
`sample()` (*kwarray.distributions.Normal method*), 63
`sample()` (*kwarray.distributions.PDF method*), 67
`sample()` (*kwarray.distributions.TruncNormal method*), 64
`sample()` (*kwarray.distributions.Uniform method*), 60
`sample()` (*kwarray.distributions.Value method*), 49
`seed()` (*kwarray.distributions.Distribution method*), 52
`seed()` (*kwarray.distributions.Parameterized method*), 50

seed_global() (in module kwarray), 183
seed_global() (in module kwarray.util_random), 98
Seeded (class in kwarray.distributions), 67
seeded() (kwarray.distributions.Distribution class method), 53
setcover() (in module kwarray), 183
setcover() (in module kwarray.algo_setcover), 8
shape (kwarray.RunningStats property), 139
shape (kwarray.util_averages.RunningStats property), 76
shuffle() (in module kwarray), 184
shuffle() (in module kwarray.util_random), 98
slices (kwarray.SlidingWindow property), 144
slices (kwarray.util_slider.SlidingWindow property), 117
SlidingWindow (class in kwarray), 142
SlidingWindow (class in kwarray.util_slider), 115
softmax() (kwarray.ArrayAPI static method), 130
softmax() (kwarray.arrayapi.ArrayAPI static method), 40
softmax() (kwarray.arrayapi.NumpyImpls static method), 37
softmax() (kwarray.arrayapi.TorchImpls static method), 22
sort_values() (kwarray.dataframe_light.DataFrameLight method), 45
sort_values() (kwarray.DataFrameLight method), 134
sqrt() (kwarray.distributions._BinOpMixin method), 51
standard_normal() (in module kwarray), 185
standard_normal() (in module kwarray.fast_rand), 69
standard_normal32() (in module kwarray), 185
standard_normal32() (in module kwarray.fast_rand), 69
standard_normal64() (in module kwarray), 187
standard_normal64() (in module kwarray.fast_rand), 71
stats_dict() (in module kwarray), 187
stats_dict() (in module kwarray.util_averages), 73
Stitcher (class in kwarray), 145
Stitcher (class in kwarray.util_slider), 118
sum() (kwarray.ArrayAPI static method), 129
sum() (kwarray.arrayapi.ArrayAPI static method), 39
sum() (kwarray.arrayapi.NumpyImpls static method), 26
sum() (kwarray.arrayapi.TorchImpls static method), 21
summarize() (kwarray.RunningStats method), 141
summarize() (kwarray.util_averages.RunningStats method), 78

T

T() (kwarray.ArrayAPI static method), 129
T() (kwarray.arrayapi.ArrayAPI static method), 40
T() (kwarray.arrayapi.NumpyImpls static method), 25
T() (kwarray.arrayapi.TorchImpls static method), 15
take() (kwarray.ArrayAPI static method), 129

take() (kwarray.arrayapi.ArrayAPI static method), 39
take() (kwarray.arrayapi.NumpyImpls static method), 25
take() (kwarray.arrayapi.TorchImpls static method), 13
take() (kwarray.dataframe_light.DataFrameArray method), 48
take() (kwarray.dataframe_light.DataFrameLight method), 46
take() (kwarray.DataFrameArray method), 130
take() (kwarray.DataFrameLight method), 134
tensor() (kwarray.ArrayAPI static method), 129
tensor() (kwarray.arrayapi.ArrayAPI static method), 40
tensor() (kwarray.arrayapi.NumpyImpls static method), 35
tensor() (kwarray.arrayapi.TorchImpls static method), 21
tile() (kwarray.ArrayAPI static method), 129
tile() (kwarray.arrayapi.ArrayAPI static method), 39
tile() (kwarray.arrayapi.NumpyImpls static method), 25
tile() (kwarray.arrayapi.TorchImpls static method), 14
to_dict() (kwarray.dataframe_light.DataFrameLight method), 44
to_dict() (kwarray.DataFrameLight method), 133
to_string() (kwarray.dataframe_light.DataFrameLight method), 44
to_string() (kwarray.DataFrameLight method), 133
tolist() (kwarray.ArrayAPI static method), 129
tolist() (kwarray.arrayapi.ArrayAPI static method), 40
tolist() (kwarray.arrayapi.NumpyImpls static method), 35
tolist() (kwarray.arrayapi.TorchImpls static method), 21
TorchImpls (class in kwarray.arrayapi), 13
TorchNumpyCompat (in module kwarray.arrayapi), 41
transpose() (kwarray.ArrayAPI static method), 130
transpose() (kwarray.arrayapi.ArrayAPI static method), 40
transpose() (kwarray.arrayapi.NumpyImpls static method), 25
transpose() (kwarray.arrayapi.TorchImpls static method), 15
TruncNormal (class in kwarray.distributions), 63

U

Uniform (class in kwarray.distributions), 60
uniform() (in module kwarray), 189
uniform() (in module kwarray.fast_rand), 68
uniform32() (in module kwarray), 190
uniform32() (in module kwarray.fast_rand), 71
union() (kwarray.dataframe_light.DataFrameLight method), 46

[union\(\)](#) (*kvarray.DataFrameLight method*), [135](#)
[unique_rows\(\)](#) (*in module kvarray*), [191](#)
[unique_rows\(\)](#) (*in module kvarray.util_numpy*), [93](#)
[unravel\(\)](#) (*kvarray.FlatIndexer method*), [137](#)
[unravel\(\)](#) (*kvarray.util_misc.FlatIndexer method*), [87](#)
[update\(\)](#) (*kvarray.RunningStats method*), [140](#)
[update\(\)](#) (*kvarray.util_averages.RunningStats method*),
[77](#)
[update_many\(\)](#) (*kvarray.RunningStats method*), [139](#)
[update_many\(\)](#) (*kvarray.util_averages.RunningStats method*), [76](#)

V

[Value](#) (*class in kvarray.distributions*), [49](#)
[values](#) (*kvarray.dataframe_light.DataFrameLight property*), [44](#)
[values](#) (*kvarray.DataFrameLight property*), [133](#)
[view\(\)](#) (*kvarray.ArrayAPI static method*), [129](#)
[view\(\)](#) (*kvarray.arrayapi.ArrayAPI static method*), [39](#)
[view\(\)](#) (*kvarray.arrayapi.NumpyImpls static method*),
[25](#)
[view\(\)](#) (*kvarray.arrayapi.TorchImpls static method*), [13](#)
[vstack\(\)](#) (*kvarray.ArrayAPI static method*), [129](#)
[vstack\(\)](#) (*kvarray.arrayapi.ArrayAPI static method*),
[39](#)
[vstack\(\)](#) (*kvarray.arrayapi.NumpyImpls static method*), [24](#)
[vstack\(\)](#) (*kvarray.arrayapi.TorchImpls static method*),
[13](#)

Z

[zeros\(\)](#) (*kvarray.arrayapi.NumpyImpls static method*),
[26](#)
[zeros\(\)](#) (*kvarray.arrayapi.TorchImpls static method*),
[16](#)
[zeros_like\(\)](#) (*kvarray.ArrayAPI static method*), [129](#)
[zeros_like\(\)](#) (*kvarray.arrayapi.ArrayAPI static method*), [39](#)
[zeros_like\(\)](#) (*kvarray.arrayapi.NumpyImpls static method*), [25](#)
[zeros_like\(\)](#) (*kvarray.arrayapi.TorchImpls static method*), [16](#)