kwarray Documentation

Release 0.5.20

Jon Crall

Nov 05, 2021

CONTENTS

The `kwarray` module implements a small set of pure-python extensions to numpy and torch.

CHAPTER

ONE

FUNCTION USEFULNESS

Function name	Usefulness
kwarray.ArrayAPI()	219
kwarray.ensure_rng()	135
kwarray.boolmask()	23
kwarray.stats_dict()	21
kwarray.DataFrameArray()	21
kwarray.group_indices()	20
kwarray.shuffle()	20
kwarray.argmaxima()	16
kwarray.seed_global()	8
kwarray.maxvalue_assignment()	7
kwarray.one_hot_embedding()	7
kwarray.atleast_nd()	6
kwarray.iter_reduce_ufunc()	5
kwarray.isect_flags()	5
kwarray.group_items()	4
kwarray.mincost_assignment()	3
kwarray.standard_normal()	3
kwarray.arglexmax()	2
kwarray.apply_grouping()	1
kwarray.DataFrameLight()	1
kwarray.uniform()	1

1.1 API Reference

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

[1] Created with sphinx-autoapi

1.1.1 kwarray

The `kwarray` module implements a small set of pure-python extensions to numpy and torch.

Submodules

kwarray.algo_assignment

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

The core implementation of munkres in 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

Module Contents

Functions

mindist_assignment(vecs1, vecs2, p=2)	Finds minimum cost assignment between two sets of D dimensional vectors.
mincost_assignment(cost)	Finds the minimum cost assignment based on a NxM cost matrix, subject to
maxvalue_assignment(value)	Finds the maximum value assignment based on a NxM value matrix. Any pair

`kwarray.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 Eucliedean)

Returns

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

Return type Tuple[list, float]

Notes

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 Contents

Functions

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.
_setcover_greedy_old(candidate_sets_dict, items=None, set_weights=None, item_values=None, max_weight=None)	Benchmark:
_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 algorithms.
_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

`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[Hashable, List[Hashable]]*) – a dictionary where keys are the candidate set ids and each value is a candidate cover set.
- **items** (*Hashable, optional*) – the set of all items to be covered, if not specified, it is infered from the candidate cover sets
- **set_weights** (*Dict, optional*) – maps candidate set ids to a cost for using this candidate cover in the solution. If not specified the weight of each candiate cover defaults to 1.
- **item_values** (*Dict, optional*) – 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** (*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.repr2(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.repr2(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 algorithms.

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(len(items) + 1) approximation algorithm Weighted Maximum Cover: 1 - 1/e == .632 approximation algorithm Generalized maximum coverage is not implemented

References

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

Notes

pip install git+git://github.com/tangentlabs/django-oscar.git#egg=django-oscar. # TODO: wrap https://github.com/martin-steinegger/setcover/blob/master/SetCover.cpp # pip install SetCoverPy # This is actually much slower than my implementation from SetCoverPy import setcover g = setcover.SetCover(full_overlaps, cost=np.ones(len(full_overlaps))) g.greedy() keep = np.where(g.s)[0]

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.repr2(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.repr2(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.repr2(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

---

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

Example

```
>>> # xdoc: +REQUIRES(module:pulp)
>>> candidate_sets_dict = {}
>>> exact_soln = _setcover_ilp(candidate_sets_dict)
>>> print('exact_soln = {}'.format(ub.repr2(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

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 convinience. It is convinient 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.

Example

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

(continues on next page)

(continued from previous page)

```
>>> print('res2_np = {!r}'.format(res2_np))
>>> assert np.allclose(res1_np, res2_np)
>>> # Method 2: choose the impl on the fly
>>> result1 = ArrayAPI.sum(data1, axis=0)
>>> result2 = ArrayAPI.sum(data2, axis=0)
>>> res1_np = ArrayAPI.numpy(result1)
>>> res2_np = 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()
```

Module Contents

Classes

_ImplRegistry	
TorchImpls	Torch backend for the ArrayAPI API
NumpyImpls	Numpy backend for the ArrayAPI API
ArrayAPI	Compatability API between torch and numpy.

Functions

_get_funcname(func)

_torch_dtype_lut()

dtype_info(dtype)

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

Attributes

torch
_REGISTERY
_torchmethod
_numpymethod
_apimethod
TorchNumpyCompat

kwarray.arrayapi.**torch**

kwarray.arrayapi.**_get_funcname**(*func*)

class kwarray.arrayapi.**_ImplRegistry**

Bases: object

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

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

_apimethod(*self*, *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(*self*)

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

kwarray.arrayapi.**_REGISTERY**

kwarray.arrayapi.**_torchmethod**

kwarray.arrayapi.**_numpymethod**

kwarray.arrayapi.**_apimethod**

class kwarray.arrayapi.**TorchImpls**

Bases: object

Torch backend for the ArrayAPI API

is_tensor = True

is_numpy = False

ensure

cat(*datas*, *axis=- 1*)

hstack(*datas*)

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

Example

```
>>> # xdoctest: +REQUIRES(module: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)
```

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)
>>> 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)
```

atleast_nd(*arr*, *n*, *front=False*)

view(*data*, **shape*)

take(*data*, *indices*, *axis=None*)

compress(*data*, *flags*, *axis=None*)

Example

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

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)
```

abstract 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:
```

(continues on next page)

(continued from previous page)

```
>>>     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])

T(*data*)

transpose(*data*, *axes*)

Example

```
>>> # xdoctest: +REQUIRES(module: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)
```

numel(*data*)

full_like(*data*, *fill_value*, *dtype=None*)

empty_like(*data*, *dtype=None*)

zeros_like(*data*, *dtype=None*)

ones_like(*data*, *dtype=None*)

full(*shape*, *fill_value*, *dtype=float*)

empty(*shape*, *dtype=float*)

zeros(*shape*, *dtype=float*)

ones(*shape*, *dtype=float*)

argmax(*data*, *axis=None*)

argsort(*data*, *axis=- 1*, *descending=False*)

Example

```
>>> # xdoctest: +REQUIRES(module:torch)
>>> from kwarray.arrayapi import *  # NOQA
>>> 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)
>>> res2 = ArrayAPI.argsort(data2, axis=0, descending=True)
>>> assert np.all(res1.numpy() == res2)
```

max(*data*, *axis=None*)

Example

```
>>> # xdoctest: +REQUIRES(module:torch)
>>> 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)
```

max_argmax(*data*, *axis=None*)

Note: this isn't always gaurenteed to be compatibile with numpy if there are equal elements in data. See: >>> np.ones(10).argmax() # xdoctest: +IGNORE_WANT 0 >>> torch.ones(10).argmax() # xdoctest: +IGNORE_WANT tensor(9)

maximum(*data1*, *data2*, *out=None*)

Example

```
>>> # xdoctest: +REQUIRES(module: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)
```

minimum(*data1*, *data2*, *out=None*)

Example

```
>>> # xdoctest: +REQUIRES(module: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)
```

sum(*data*, *axis=None*)

nan_to_num(*x*, *copy=True*)

copy(*data*)

nonzero(*data*)

astype(*data*, *dtype*, *copy=True*)

tensor(*data*, *device=ub.NoParam*)

numpy(*data*)

tolist(*data*)

contiguous(*data*)

pad(*data*, *pad_width*, *mode='constant'*)

asarray(*data*, *dtype=None*)
Cast data into a tensor representation

Example

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

dtype_kind(*data*)
returns the numpy code for the data type kind

floor(*data*, *out=None*)

ceil(*data*, *out=None*)

ifloor(*data*, *out=None*)

iceil(*data*, *out=None*)

round(*data*, *decimals=0*, *out=None*)

Example

```
>>> # xdoctest: +REQUIRES(module:torch)
>>> import kwarray
>>> 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())
```

iround(*data*, *out=None*, *dtype=int*)

clip(*data*, *a_min=None*, *a_max=None*, *out=None*)

softmax(*data*, *axis=None*)

class kwarray.arrayapi.**NumpyImpls**

Bases: object

Numpy backend for the ArrayAPI API

is_tensor = False

is_numpy = True

hstack

vstack

matmul

nan_to_num

log

log2

any

all

copy

nonzero

ensure

clip

cat(*datas*, *axis=- 1*)

atleast_nd(*arr*, *n*, *front=False*)

view(*data*, **shape*)

take(*data*, *indices*, *axis=None*)

compress(*data*, *flags*, *axis=None*)

repeat(*data*, *repeats*, *axis=None*)

tile(*data*, *reps*)

T(*data*)

transpose(*data*, *axes*)

numel(*data*)

empty_like(*data*, *dtype=None*)

full_like(*data*, *fill_value*, *dtype=None*)

zeros_like(*data*, *dtype=None*)

ones_like(*data*, *dtype=None*)

full(*shape*, *fill_value*, *dtype=float*)

empty(*shape*, *dtype=float*)

zeros(*shape*, *dtype=float*)

ones(*shape*, *dtype=float*)

argmax(*data*, *axis=None*)

argsort(*data*, *axis=- 1*, *descending=False*)

max(*data*, *axis=None*)

max_argmax(*data*, *axis=None*)

sum(*data*, *axis=None*)

maximum(*data1*, *data2*, *out=None*)

minimum(*data1*, *data2*, *out=None*)

astype(*data*, *dtype*, *copy=True*)

tensor(*data*, *device=ub.NoParam*)

numpy(*data*)

tolist(*data*)

contiguous(*data*)

pad(*data*, *pad_width*, *mode='constant'*)

asarray(*data*, *dtype=None*)

Cast data into a numpy representation

dtype_kind(*data*)

floor(*data*, *out=None*)

ceil(*data*, *out=None*)

ifloor(*data*, *out=None*)

iceil(*data*, *out=None*)

round(*data*, *decimals=0*, *out=None*)

iround(*data*, *out=None*, *dtype=int*)

softmax(*data*, *axis=None*)

class kwarray.arrayapi.**ArrayAPI**

Bases: object

Compatability 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)
>>> take = 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)
>>> compress = 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
>>> 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

_numpy

take

compress

repeat

tile

view

numel

atleast_nd

full_like

ones_like

zeros_like

empty_like

sum

argmax

argsort

max

maximum

minimum

matmul

astype

nonzero

nan_to_num

tensor

numpy

tolist

asarray

asarray

T

transpose

contiguous

pad

dtype_kind

max_argmax

any

all

log2

log

copy

iceil

ifloor

floor

ceil

round

iround

clip

softmax

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).

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

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

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

kwarray.arrayapi.**TorchNumpyCompat**

kwarray.arrayapi.**_torch_dtype_lut**()

kwarray.arrayapi.**dtype_info**(*dtype*)

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

Returns an iinfo of finfo structure depending on the input type.

Return type struct

References

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

Example

```
>>> from kwarray.arrayapi import *  # NOQA
>>> 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

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.

Module Contents

Classes

LocLight	
DataFrameLight	Implements a subset of the pandas.DataFrame API
DataFrameArray	DataFrameLight assumes the backend is a Dict[list]

Attributes

pd
__version__

kwarray.dataframe_light.**pd**

kwarray.dataframe_light.**__version__ = 0.0.1**

class kwarray.dataframe_light.**LocLight**(*parent*)

Bases: object

__getitem__(*self*, *index*)

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

Bases: ubelt.NiceRepr

Implements a subset of the pandas.DataFrame API

The API is restricted to facilitate speed tradeoffs

Notes

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.

Notes

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 =====')
```

(continues on next page)

(continued from previous page)

```
>>> _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)
>>> 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()}
```

(continues on next page)

(continued from previous page)

```
>>> 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 =======
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(*self*)

property values(*self*)

property loc(*self*)

__eq__(*self*, *other*)

Example

```
>>> # xdoctest: +REQUIRES(module:pandas)
>>> self = DataFrameLight._demodata(num=7)
>>> other = self.pandas()
>>> assert np.all(self == other)
```

to_string(*self*, **args*, ***kwargs*)

to_dict(*self*, *orient='dict'*, *into=dict*)

Convert the data frame into a dictionary.

Parameters

- **orient** (*str*) – Currently naitively suports 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(*self*)

Convert back to pandas if you need the full API

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(*self*)

Deprecated, use self.pandas instead

classmethod _demodata(*cls*, *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
```

__nice__(*self*)

__len__(*self*)

__contains__(*self*, *item*)

__normalize__(*self*)

Try to convert input data to Dict[List]

property columns(*self*)

sort_values(*self*, *key*, *inplace=False*, *ascending=True*)

keys(*self*)

_getrow(*self*, *index*)

_getcol(*self*, *key*)

_getcols(*self*, *keys*)

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

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

clear(*self*)

Removes all rows inplace

__getitem__(*self*, *key*)

Note: only handles the case where key is a single column name.

Example

```
>>> df_light = DataFrameLight._demodata(num=7)
>>> sub1 = df_light['bar']
>>> # xdoctest: +REQUIRES(module:pandas)
>>> df_heavy = df_light.pandas()
>>> sub2 = df_heavy['bar']
>>> assert np.all(sub1 == sub2)
```

__setitem__(*self*, *key*, *value*)

Note: only handles the case where key is a single column name. and value is an array of all the values to set.

Example

```
>>> df_light = DataFrameLight._demodata(num=7)
>>> value = [2] * len(df_light)
>>> df_light['bar'] = value
>>> # xdoctest: +REQUIRES(module:pandas)
>>> df_heavy = df_light.pandas()
>>> df_heavy['bar'] = value
>>> assert np.all(df_light == df_heavy)
```

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

NOTE: NOT A PART OF THE PANDAS API

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

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

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

Notes

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(*self*)

extend(*self*, *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(*self*, **others*)

Note: Note part of the pandas API

classmethod concat(*cls*, *others*)

classmethod from_pandas(*cls*, *df*)

classmethod from_dict(*cls*, *records*)

reset_index(*self*, *drop=False*)

noop for compatability, the light version doesnt store an index

groupby(*self*, *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)])
```

Ignore:

```
>>> self = DataFrameLight._demodata(num=1000)
>>> args = ['cx']
>>> self['cx'] = (np.random.rand(len(self)) * 10).astype(np.int)
>>> # As expected, our custom restricted implementation is faster
>>> # than pandas
>>> ub.Timerit(100).call(lambda: dict(list(self.pandas().groupby('cx')))).
↪print()
>>> ub.Timerit(100).call(lambda: dict(self.groupby('cx'))).print()
```

rename(*self*, *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(*self*)

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.repr2(list(self.iterrows())))
[
    (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

__normalize__(*self*)

Try to convert input data to Dict[ndarray]

extend(*self*, *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
```

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

NOTE: NOT A PART OF THE PANDAS API

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

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

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

Notes

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)
```

kwarray.distributions

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?

Module Contents

Classes

Uniform	Defaults to a uniform distribution over floats between 0 and 1
Exponential	**Example**
Constant	**Example**
DiscreteUniform	Uniform distribution over integers.
Normal	`>>> self = Normal(mean=100, rng=0)`
Bernoulli	self = Normal()
Binomial	self = Normal()
Categorical	**Example**
TruncNormal	A truncated normal distribution.

class kwarray.distributions.**Uniform**(*low=0*, *high=1*, *rng=None*)

Bases: `Distribution`

Defaults to a uniform distribution over floats between 0 and 1

Example

```
>>> 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)
```

sample(*self*, **shape*)

classmethod coerce(*cls*, *arg*)

class kwarray.distributions.**Exponential**(*scale=1*, *rng=None*)

Bases: Distribution

Example

```
>>> self = Exponential(rng=0)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.figure(fnum=1, doclf=True)
>>> self._show(500, bins=25)
```

sample(*self*, **shape*)

class kwarray.distributions.**Constant**(*value=0*, *rng=None*)

Bases: Distribution

Example

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

sample(*self*, **shape*)

class kwarray.distributions.**DiscreteUniform**(*min=0*, *max=1*, *rng=None*)

Bases: Distribution

Uniform distribution over integers.

Parameters

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

Example

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

sample(*self*, **shape*)

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

class kwarray.distributions.**Normal**(*mean=0*, *std=1*, *rng=None*)

Bases: Distribution

```
>>> self = Normal(mean=100, rng=0)
>>> self.sample()
>>> self.sample(100)
```

sample(*self*, **shape*)

class kwarray.distributions.**Bernoulli**(*p=0.5*, *rng=None*)

Bases: Distribution

self = Normal() self.sample() self.sample(1)

sample(*self*, **shape*)

classmethod coerce(*cls*, *arg*)

class kwarray.distributions.**Binomial**(*n=1*, *p=0.5*, *rng=None*)

Bases: Distribution

self = Normal() self.sample() self.sample(1)

sample(*self*, **shape*)

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

Bases: Distribution

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]])
```

sample(*self*, **shape*)

class kwarray.distributions.**TruncNormal**(*mean=0*, *std=1*, *low=- np.inf*, *high=np.inf*, *rng=None*)

Bases: Distribution

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*)

Example

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

Example

```
>>> 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.repr2(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(*self*)

sample(*self*, **shape*)

kwarray.fast_rand

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

Module Contents

Functions

uniform(low=0.0, high=1.0, size=None, dtype=np.float32, rng=np.random)	Draws float32 samples from a uniform distribution.
standard_normal(size, mean=0, std=1, dtype=float, rng=np.random)	Draw samples from a standard Normal distribution with a specified mean and
standard_normal32(size, mean=0, std=1, rng=np.random)	Fast normally distributed random variables using the Box–Muller transform
standard_normal64(size, mean=0, std=1, rng=np.random)	Simple wrapper around rng.standard_normal to make an API compatible with
uniform32(low=0.0, high=1.0, size=None, rng=np.random)	Draws float32 samples from a uniform distribution.

`kwarray.fast_rand.`**`uniform`**(*low=0.0*, *high=1.0*, *size=None*, *dtype=np.float32*, *rng=np.random*)

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], default=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
- **rng** (*numpy.random.RandomState*) – underlying random state

Returns normally distributed random numbers with chosen dtype

Return type ndarray[dtype]

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=float*, *rng=np.random*)

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

Parameters

- **size** (*int | Tuple[int, *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
- **dtype** (*type*) – either np.float32 or np.float64
- **rng** (*numpy.random.RandomState*) – underlying random state

Returns normally distributed random numbers with chosen dtype

Return type ndarray[dtype]

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=np.random*)

Fast normally distributed random variables using the Box–Muller transform

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, *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

Return type ndarray[float32]

References

https://en.wikipedia.org/wiki/Box%E2%80%93Muller_transform

SeeAlso:

- standard_normal
- standard_normal64

Example

```
>>> 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=np.random*)

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

Parameters

- **size** (*int | Tuple[int, *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

Return type ndarray[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=np.random*)

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], default=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.

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

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.

Module Contents

Classes

RunningStats	Dynamically records per-element array statistics and can summarized them

Functions

stats_dict(inputs, axis=None, nan=False, sum=False, extreme=True, n_extreme=False, median=False, shape=True, size=False)	Describe statistics about an input array
_gmean(a, axis=0, dtype=None, clobber=False)	Compute the geometric mean along the specified axis.

Attributes

torch

kwarray.util_averages.**torch**

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

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
- **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

Returns

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

Return type collections.OrderedDict

SeeAlso: scipy.stats.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.repr2(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),
    'med': np.array([0.5584, 0.6805], dtype=np.float32),
    'shape': (10, 2),
}
```

Example

```
>>> # xdoctest: +IGNORE_WHITESPACE
>>> 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)
>>> import ubelt as ub  # NOQA
```

(continues on next page)

(continued from previous page)

```
>>> result = str(ub.repr2(stats, nl=0, precision=1, strkeys=True))
>>> print(result)
{mean: 20.0, std: 13.2, min: 0.0, max: 41.0, num_nan: 1, shape: (100,)}
```

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)]
```

class kwarray.util_averages.**RunningStats**(*run*)

Bases: ubelt.NiceRepr

Dynamically records per-element array statistics and can summarized 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.repr2(run.summarize(axis=ub.NoParam), nl=2,␣
↪precision=3))
>>> # Average over channels (keeps spatial dims separate)
>>> print('chann-ave(k=1) = ' + ub.repr2(run.summarize(axis=0), nl=2, precision=3))
>>> print('chann-ave(k=0) = ' + ub.repr2(run.summarize(axis=0, keepdims=0), nl=2,␣
↪precision=3))
>>> # Average over spatial dims (keeps channels separate)
>>> print('spatial-ave(k=1) = ' + ub.repr2(run.summarize(axis=(1, 2)), nl=2,␣
↪precision=3))
>>> print('spatial-ave(k=0) = ' + ub.repr2(run.summarize(axis=(1, 2), keepdims=0),␣
↪nl=2, precision=3))
>>> # Average over all dims
```

(continues on next page)

(continued from previous page)

```
>>> print('alldim-ave(k=1) = ' + ub.repr2(run.summarize(axis=None), nl=2,␣
↪precision=3))
>>> print('alldim-ave(k=0) = ' + ub.repr2(run.summarize(axis=None, keepdims=0),␣
↪nl=2, precision=3))
```

__nice__(*self*)

property shape(*run*)

update(*run*, *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)
```

_sumsq_std(*run*, *total*, *squares*, *n*)

Sum of squares method to compute standard deviation

summarize(*run*, *axis=None*, *keepdims=True*)

Compute summary statistics across a one or more dimension

Parameters

- **axis** (*int | List[int] | None | ub.NoParam*) – 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

current(*run*)

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

Todo:

- **[X] I want this method and summarize to be unified somehow.** I don't know how to paramatarize it because axis=None usually means summarize over everything, and I need to way to encode, summarize over nothing but the "sequence" dimension (which was given incrementally by the update function), which is what this function does.

---

kwarray.util_groups

Functions for partitioning numpy arrays into groups.

Module Contents

Functions

group_items(item_list, groupid_list, assume_sorted=False, axis=None)	Groups a list of items by group id.
group_indices(idx_to_groupid, assume_sorted=False)	Find unique items and the indices at which they appear in an array.
apply_grouping(items, groupxs, axis=0)	Applies grouping from group_indicies.
group_consecutive(arr, offset=1)	Returns lists of consecutive values. Implementation inspired by[3].
group_consecutive_indices(arr, offset=1)	Returns lists of indices pointing to consecutive values

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()`[1,2].

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[T1]*) – The input array of items to group.
- **groupid_list** (*ndarray[T2]*) – 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.
- **assume_sorted** (*bool, default=False*) – If the input array is sorted, then setting this to True will avoid an unnecessary sorting operation and improve efficiency.
- **axis** (*int | None*) – group along a particular axis in `items` if it is n-dimensional

Returns mapping from groupids to corresponding items

Return type Dict[T2, ndarray[T1]]

[3] http://stackoverflow.com/questions/7352684/groups-consecutive-elements
[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])
>>> keys = np.array( [2, 2, 1, 1, 0, 1, 0, 1])
>>> grouped = group_items(items, keys)
>>> print(ub.repr2(grouped, nl=1, with_dtype=False))
{
    0: np.array([4, 6]),
    1: np.array([2, 3, 5, 7]),
    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, default=False*) – If the input array is sorted, then setting this to True will avoid an unnecessary sorting operation and improve efficiency.

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[ndarrays]]

Example

```
>>> # xdoctest: +IGNORE_WHITESPACE
>>> import ubelt as ub
>>> idx_to_groupid = np.array([2, 1, 2, 1, 2, 1, 2, 3, 3, 3, 3])
>>> (keys, groupxs) = group_indices(idx_to_groupid)
>>> print(ub.repr2(keys, with_dtype=False))
>>> print(ub.repr2(groupxs, with_dtype=False))
np.array([1, 2, 3])
```

(continues on next page)

(continued from previous page)

```
[
    np.array([1, 3, 5]),
    np.array([0, 2, 4, 6]),
    np.array([ 7,  8,  9, 10]),
]
```

Example

```
>>> # xdoctest: +IGNORE_WHITESPACE
>>> import ubelt as ub
>>> idx_to_groupid = np.array([[  24], [ 129], [ 659], [ 659], [ 24],
...       [659], [ 659], [ 822], [ 659], [ 659], [24]])
>>> # 2d arrays must be flattened before coming into this function so
>>> # information is on the last axis
>>> (keys, groupxs) = group_indices(idx_to_groupid.T[0])
>>> print(ub.repr2(keys, with_dtype=False))
>>> print(ub.repr2(groupxs, with_dtype=False))
np.array([ 24, 129, 659, 822])
[
    np.array([ 0,  4, 10]),
    np.array([1]),
    np.array([2, 3, 5, 6, 8, 9]),
    np.array([7]),
]
```

Example

```
>>> # xdoctest: +IGNORE_WHITESPACE
>>> import ubelt as ub
>>> idx_to_groupid = np.array([True, True, False, True, False, False, True])
>>> (keys, groupxs) = group_indices(idx_to_groupid)
>>> print(ub.repr2(keys, with_dtype=False))
>>> print(ub.repr2(groupxs, with_dtype=False))
np.array([False,  True])
[
    np.array([2, 4, 5]),
    np.array([0, 1, 3, 6]),
]
```

Example

```
>>> # xdoctest: +IGNORE_WHITESPACE
>>> import ubelt as ub
>>> idx_to_groupid = [('a', 'b'),  ('d', 'b'), ('a', 'b'), ('a', 'b')]
>>> (keys, groupxs) = group_indices(idx_to_groupid)
>>> print(ub.repr2(keys, with_dtype=False))
>>> print(ub.repr2(groupxs, with_dtype=False))
[
    ('a', 'b'),
    ('d', 'b'),
]
[
    np.array([0, 2, 3]),
    np.array([1]),
]
```

kwarray.util_groups.**apply_grouping**(*items*, *groupxs*, *axis=0*)

Applies grouping from group_indicies.

Typically used in conjunction with `group_indices()`.

Parameters

- **items** (*ndarray*) – items to group
- **groupxs** (*List[ndarrays[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_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 by[?].

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 thta are the same, e.g.: 4, 4, 4.

Returns a list of arrays that are the groups from the input

Return type List[ndarray]

Notes

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()

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))))
```

(continues on next page)

(continued from previous page)

```
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 Contents

Classes

FlatIndexer	Creates a flat "view" of a jagged nested indexable object.

class kwarray.util_misc.**FlatIndexer**(*lens*)

Bases: ubelt.NiceRepr

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

Parameters **lens** (*list*) – 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**(*cls*, *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

__len__(*self*)

unravel(*self*, *index*)

Parameters **index** – 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(*self*, *outer*, *inner*)

Parameters

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

Returns the raveled index

Return type index

kwarray.util_numpy

Numpy specific extensions

Module Contents

Functions

boolmask(indices, shape=None)	Constructs an array of booleans where an item is True if its position is in
iter_reduce_ufunc(ufunc, arrs, out=None, default=None)	constant memory iteration and reduction
isect_flags(arr, other)	Check which items in an array intersect with another set of items
atleast_nd(arr, n, front=False)	View inputs as arrays with at least n dimensions.
argmaxima(arr, num, axis=None, ordered=True)	Returns the top num maximum indicies.
argminima(arr, num, axis=None, ordered=True)	Returns the top num minimum indicies.
unique_rows(arr, ordered=False)	Like unique, but works on rows
arglexmax(keys, multi=False)	Find the index of the maximum element in a sequence of keys.
normalize(arr, mode='linear', alpha=None, beta=None, out=None)	Rebalance signal values via contrast stretching.

`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 incoporate 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[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

```
>>> indices = np.array([(0, 0), (1, 1), (2, 1)])
>>> shape = (3, 3)
>>> mask = boolmask(indices, shape)
>>> import ubelt as ub  # NOQA
>>> result = ub.repr2(mask)
>>> print(result)
np.array([[ True, False, False],
          [False,  True, False],
          [False,  True, False]], dtype=np.bool)
```

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

constant memory iteration and reduction

applys 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.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 that item 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]
```

(continues on next page)

(continued from previous page)

```
 [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** (*array_like*) – 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, default=False*) – if True new dimensions are added to the front of the array. otherwise they are added to the back.

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.repr2(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.repr2(arr1_.tolist(), nl=0)
>>> result2 = ub.repr2(arr2_.tolist(), nl=0)
>>> result3 = ub.repr2(arr3_.tolist(), nl=0)
>>> result = '\n'.join([result1, result2, result3])
>>> print(result)
[[[[1]]], [[[1]]], [[[1]]]]
[[[[0]]]]
[[[[[1]]]]]
```

Notes

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.argminima(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 argrees 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*)

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=(12, 3))
>>> arr_unique = unique_rows(arr)
>>> print('arr_unique = {!r}'.format(arr_unique))
```

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[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.util_numpy.**normalize**(*arr*, *mode='linear'*, *alpha=None*, *beta=None*, *out=None*)

Rebalance signal values via contrast stretching.

By default linearly stretches array values to minimum and maximum values.

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(abs(old_min - beta), abs(old_max - 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 `(max - min) / 2`. Note this parameter is sensitive to if the input is a float or uint8 image.

References

https://en.wikipedia.org/wiki/Normalization_(image_processing)

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')
>>> 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)
```

Benchmark: # Our method is faster than standard in-line implementations.

import timerit ti = timerit.Timerit(100, bestof=10, verbose=2, unit='ms') arr = kwimage.grab_test_image('lowcontrast', dsize=(512, 512))

print('— uint8 —') arr = ensure_float01(arr) out = arr.copy() for timer in ti.reset('naive1-float'):

with timer: (arr - arr.min()) / (arr.max() - arr.min())

import timerit for timer in ti.reset('simple-float'):

with timer: **max_** = arr.max() **min_** = arr.min() result = (arr - **min_**) / (**max_** - **min_**)

for timer in ti.reset('normalize-float'):

with timer: normalize(arr)

for timer in ti.reset('normalize-float-inplace'):

with timer: normalize(arr, out=out)

print('— float —') arr = ensure_uint255(arr) out = arr.copy() for timer in ti.reset('naive1-uint8'):

with timer: (arr - arr.min()) / (arr.max() - arr.min())

import timerit for timer in ti.reset('simple-uint8'):

with timer: **max_** = arr.max() **min_** = arr.min() result = (arr - **min_**) / (**max_** - **min_**)

for timer in ti.reset('normalize-uint8'):

with timer: normalize(arr)

for timer in ti.reset('normalize-uint8-inplace'):

with timer: normalize(arr, out=out)

Ignore: globals().update(xdev.get_func_kwargs(normalize))

kwarray.util_random

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

Module Contents

Functions

seed_global(seed, offset=0)	Seeds the python, numpy, and torch global random states
shuffle(items, rng=None)	Shuffles a list inplace and then returns it for convinience
random_combinations(items, size, num=None, rng=None)	Yields `num` combinations of length `size` from items in random order
random_product(items, num=None, rng=None)	Yields `num` items from the cartesian product of items in a random order.
_npstate_to_pystate(npstate)	Convert state of a NumPy RandomState object to a state
_pystate_to_npstate(pystate)	Convert state of a Python Random object to state usable
_coerce_rng_type(rng)	Internal method that transforms input seeds into an integer form.
ensure_rng(rng, api='numpy')	Coerces input into a random number generator.

Attributes

_SEED_MAX

kwarray.util_random.**_SEED_MAX**

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, optional*) – if specified, uses a different seed for each global random state separated by this offset.

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

Shuffles a list inplace and then returns it for convinience

Parameters

- **items** (*list or ndarray*) – list to shuffle
- **rng** (*RandomState or int*) – 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** (*None, default=None*) – number of combinations to generate
- **rng** (*int | RandomState, default=None*) – seed or random number generator

Yields *Tuple* – a random combination of `items` of length `size`.

Example

```
>>> 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.repr2(combos, nl=1)))
combos = [
    (0, 6, 9),
    (4, 7, 8),
    (4, 6, 7),
    (2, 3, 5),
    (1, 2, 4),
]
```

Example

```
>>> 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.repr2(combos, nl=1)))
combos = [
    ((0, 0), (6, 6), (9, 9)),
    ((4, 4), (7, 7), (8, 8)),
    ((4, 4), (6, 6), (7, 7)),
```

(continues on next page)

(continued from previous page)

```
    ((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 caresian product of packed in a list or tuple. (note this deviates from api of `itertools.product()`)
- **num** (*int, default=None*) – maximum number of items to generate. If None, all
- **rng** (*random.Random | np.random.RandomState | int*) – random number generator

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.repr2(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.repr2(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[1].

[1] https://stackoverflow.com/questions/44313620/convert-randomstate

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[2].

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*, *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 | numpy.random.RandomState | random.Random | None*) – if None, then defaults to the global rng. Otherwise this can be an integer or a RandomState class
- **api** (*str, default='numpy'*) – 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.

Returns

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

Return type (numpy.random.RandomState | random.Random)

[2] https://stackoverflow.com/questions/44313620/convert-randomstate

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')
```

Ignore:

```
>>> np.random.seed(0)
>>> np.random.randint(0, 10000)
2732
>>> np.random.seed(0)
>>> np.random.mtrand._rand.randint(0, 10000)
2732
>>> np.random.seed(0)
>>> ensure_rng(None).randint(0, 10000)
```

(continues on next page)

(continued from previous page)

```
2732
>>> np.random.randint(0, 10000)
9845
>>> ensure_rng(None).randint(0, 10000)
3264
```

kwarray.util_slices

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.

Module Contents

Functions

padded_slice(data, slices, pad=None, padkw=None, return_info=False)	Allows slices with out-of-bound coordinates. Any out of bounds coordinate
apply_embedded_slice(data, data_slice, extra_padding, **padkw)	Apply a precomputed embedded slice.
_apply_padding(array, pad_width, **padkw)	Alternative to numpy pad with different short-cut semantics for
embed_slice(slices, data_dims, pad=None)	Embeds a "padded-slice" inside known data dimension.

Attributes

__TODO__

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[T]*) – 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

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

```
>>> data_sliced = padded_slice(data, slices)
>>> print(ub.repr2(data_sliced, with_dtype=False))
np.array([0, 0, 0, 1, 2, 3, 4, 0, 0])
```

```
>>> data_sliced = padded_slice(data, slices, pad=(3, 3))
>>> print(ub.repr2(data_sliced, with_dtype=False))
np.array([0, 0, 0, 0, 0, 0, 1, 2, 3, 4, 0, 0, 0, 0, 0])
```

```
>>> data_sliced = padded_slice(data, slice(3, 4), pad=[(1, 0)])
>>> print(ub.repr2(data_sliced, with_dtype=False))
np.array([2, 3])
```

`kwarray.util_slices.__TODO__ = Multiline-String`

```
    - [ ] Could have a kwarray function to expose this inverse slice
          functionality. Also having a top-level call to apply an embedded␣
↪slice
```

(continues on next page)

(continued from previous page)

```
        would be good.

    chip_index = tuple([slice(tl_y, br_y), slice(tl_x, br_x)])
    data_slice, padding = kwarray.embed_slice(chip_index, imdata.shape)
    # 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
    inverse_slice = (
        slice(padding[0][0], imdata.shape[0] - padding[0][1]),
        slice(padding[1][0], imdata.shape[1] - padding[1][1]),
    )
    chip = kwarray.padded_slice(imdata, chip_index)
    chip = imdata[chip_index]

    fgdata = function(chip)

    # Apply just the data part back to the original
    imdata[tl_y:br_y, tl_x:br_x, :] = fgdata[inverse_slice]
```

`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** (*List[int|Tuple]*) – extra pad applied to (left and right) / (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
>>> from kwarray.util_slices import *  # NOQA
>>> slices = (slice(0, 10), slice(0, 10))
>>> data_dims  = [300, 300]
>>> pad        = [10, 5]
>>> a, b = 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
>>> slices = (slice(-10, 400), slice(-10, 400))
>>> data_dims  = [300, 300]
>>> pad        = [10, 5]
>>> a, b = 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
>>> slices = (slice(10, 40), slice(10, 40))
>>> data_dims  = [300, 300]
>>> pad        = None
>>> a, b = 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)]
```

kwarray.util_slider

Module Contents

Classes

SlidingWindow	Slide a window of a certain shape over an array with a larger shape.
Stitcher	Stitches multiple possibly overlapping slices into a larger array.

Functions

_slices1d(margin, stop, step=None, start=0, keepbound=False, check=True)	Helper to generates slices in a single dimension.

Attributes

torch

kwarray.util_slider.**torch**

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

Bases: ubelt.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 <= 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

Notes

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

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 returend.
>>> 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))]
```

__nice__(*self*)

_compute_stride(*self*, *overlap*, *stride*, *shape*, *window*)

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

__len__(*self*)

_iter_basis_frac(*self*)

__iter__(*self*)

__getitem__(*self*, *index*)

Get a specific item by its flat (raveled) index

Example

```
>>> from kwarray.util_slider import *  # NOQA
>>> window = (10, 10)
>>> shape = (20, 20)
>>> self = SlidingWindow(shape, window, stride=5)
>>> itered_items = list(self)
>>> assert len(itered_items) == len(self)
>>> indexed_items = [self[i] for i in range(len(self))]
>>> assert itered_items[0] == self[0]
>>> assert itered_items[-1] == self[-1]
>>> assert itered_items == indexed_items
```

property grid(*self*)

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

Yields Tuple[int, ...]

property slices(*self*)

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(*self*)

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**(*stitcher*, *shape*, *device='numpy'*)

Bases: ubelt.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.

Parameters **shape** (*tuple*) – dimensions of the large image that will be created from the smaller pixels or patches.

Todo:

- **[ ] Look at the old "add_fast" code in the netharn version and see if** it is worth porting. This code is kept in the dev folder in ../dev/_dev_slider.py

---

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
>>> stitcher = Stitcher(slider.input_shape)
>>> for sl in list(slider):
...     chip = highres[sl]
...     stitcher.add(sl, chip)
>>> assert stitcher.weights.max() == 4, 'some parts should be processed 4 times'
>>> recon = stitcher.finalize()
```

__nice__(*stitcher*)

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

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

Parameters

- **indices** (*slice or tuple*) – 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 or ndarray*) – weight of this patch (default to 1.0)

average(*stitcher*)

Averages out contributions from overlapping adds using weighted average

Returns ndarray: the stitched image

Return type out

finalize(*stitcher*, *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 ndarray: the stitched image

Return type final

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))
```

kwarray.util_torch

Torch specific extensions

Module Contents

Functions

_is_in_onnx_export()	

one_hot_embedding(labels, num_classes, dim=1)	Embedding labels to one-hot form.
one_hot_lookup(data, indices)	Return value of a particular column for each row in data.

Attributes

torch

kwarray.util_torch.**torch**

kwarray.util_torch.**_is_in_onnx_export**()

kwarray.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)
>>> 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.repr2(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())
>>> if torch.cuda.is_available():
>>>     t3 = one_hot_embedding(labels.to(0), num_classes)
>>>     assert np.all(t3.cpu().numpy() == t.numpy())
```

Example

```
>>> # xdoctest: +REQUIRES(module: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 corresonds 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

Notes

This is functionally equivalent to `[row[c] for row, c in zip(data, indices)]` except that it is 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],
>>> ])
```

(continues on next page)

(continued from previous page)

```
>>> 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)
```

Ignore:

```
>>> # xdoctest: +REQUIRES(module:torch, module:onnx, module:onnx_tf)
>>> # Test if this converts to ONNX
>>> from kwarray.util_torch import *  # NOQA
>>> import torch.onnx
>>> import io
>>> import onnx
>>> import onnx_tf.backend
>>> import numpy as np
>>> 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()
>>> class TFConvertWrapper(torch.nn.Module):
>>>     def forward(self, data, indices):
>>>         return one_hot_lookup(data, indices)
>>> ###
>>> # Test the ONNX export
>>> wrapped = TFConvertWrapper()
>>> onnx_file = io.BytesIO()
>>> torch.onnx.export(
>>>     wrapped, tuple([data, indices]),
```

(continues on next page)

(continued from previous page)

```
>>>     input_names=['data', 'indices'],
>>>     output_names=['out'],
>>>     f=onnx_file,
>>>     opset_version=11,
>>>     verbose=1,
>>> )
>>> onnx_file.seek(0)
>>> onnx_model = onnx.load(onnx_file)
>>> onnx_tf_model = onnx_tf.backend.prepare(onnx_model)
>>> # Test that the resulting graph tensors are concretely sized.
>>> import tensorflow as tf
>>> onnx_gd = onnx_tf_model.graph.as_graph_def()
>>> output_tensors = tf.import_graph_def(
>>>     onnx_gd,
>>>     input_map={},
>>>     return_elements=[onnx_tf_model.tensor_dict[ol].name for ol in onnx_tf_
↪model.outputs]
>>> )
>>> assert all(isinstance(d.value, int) for t in output_tensors for d in t.
↪shape)
>>> tf_outputs = onnx_tf_model.run([data, indices])
>>> pt_outputs = wrapped(data, indices)
>>> print('tf_outputs = {!r}'.format(tf_outputs))
>>> print('pt_outputs = {!r}'.format(pt_outputs))
>>> ###
>>> # Test if data is more than 2D
>>> shape = (4, 3, 8)
>>> data = torch.arange(int(np.prod(shape))).view(*shape).float()
>>> indices = torch.from_numpy(np.array([0, 1, 2, 1])).long()
>>> onnx_file = io.BytesIO()
>>> torch.onnx.export(
>>>     wrapped, tuple([data, indices]),
>>>     input_names=['data', 'indices'],
>>>     output_names=['out'],
>>>     f=onnx_file,
>>>     opset_version=11,
>>>     verbose=1,
>>> )
>>> onnx_file.seek(0)
>>> onnx_model = onnx.load(onnx_file)
>>> onnx_tf_model = onnx_tf.backend.prepare(onnx_model)
>>> # Test that the resulting graph tensors are concretely sized.
>>> import tensorflow as tf
>>> onnx_gd = onnx_tf_model.graph.as_graph_def()
>>> output_tensors = tf.import_graph_def(
>>>     onnx_gd,
>>>     input_map={},
>>>     return_elements=[onnx_tf_model.tensor_dict[ol].name for ol in onnx_tf_
↪model.outputs]
>>> )
>>> assert all(isinstance(d.value, int) for t in output_tensors for d in t.
↪shape)
```

(continues on next page)

(continued from previous page)

```
>>> tf_outputs = onnx_tf_model.run([data, indices])
>>> pt_outputs = wrapped(data, indices)
>>> print('tf_outputs = {!r}'.format(tf_outputs))
>>> print('pt_outputs = {!r}'.format(pt_outputs))
```

Package Contents

Classes

ArrayAPI	Compatability API between torch and numpy.
DataFrameArray	DataFrameLight assumes the backend is a Dict[list]
DataFrameLight	Implements a subset of the pandas.DataFrame API
LocLight	
RunningStats	Dynamically records per-element array statistics and can summarized them
FlatIndexer	Creates a flat "view" of a jagged nested indexable object.
SlidingWindow	Slide a window of a certain shape over an array with a larger shape.
Stitcher	Stitches multiple possibly overlapping slices into a larger array.

Functions

dtype_info(dtype)	**Parameters dtype** (*type*) -- a numpy, torch, or python numeric data type
maxvalue_assignment(value)	Finds the maximum value assignment based on a NxM value matrix. Any pair
mincost_assignment(cost)	Finds the minimum cost assignment based on a NxM cost matrix, subject to
mindist_assignment(vecs1, vecs2, p=2)	Finds minimum cost assignment between two sets of D dimensional vectors.
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.
standard_normal(size, mean=0, std=1, dtype=float, rng=np.random)	Draw samples from a standard Normal distribution with a specified mean and
standard_normal32(size, mean=0, std=1, rng=np.random)	Fast normally distributed random variables using the Box–Muller transform
standard_normal64(size, mean=0, std=1, rng=np.random)	Simple wrapper around rng.standard_normal to make an API compatible with
uniform(low=0.0, high=1.0, size=None, dtype=np.float32, rng=np.random)	Draws float32 samples from a uniform distribution.
uniform32(low=0.0, high=1.0, size=None, rng=np.random)	Draws float32 samples from a uniform distribution.

continues on next page

Table 26 – continued from previous page

stats_dict(inputs, axis=None, nan=False, sum=False, extreme=True, n_extreme=False, median=False, shape=True, size=False)	Describe statistics about an input array
apply_grouping(items, groupxs, axis=0)	Applies grouping from group_indicies.
group_consecutive(arr, offset=1)	Returns lists of consecutive values. Implementation inspired by[3].
group_consecutive_indices(arr, offset=1)	Returns lists of indices pointing to consecutive values
group_indices(idx_to_groupid, assume_sorted=False)	Find unique items and the indices at which they appear in an array.
group_items(item_list, groupid_list, assume_sorted=False, axis=None)	Groups a list of items by group id.
arglexmax(keys, multi=False)	Find the index of the maximum element in a sequence of keys.
argmaxima(arr, num, axis=None, ordered=True)	Returns the top num maximum indicies.
argminima(arr, num, axis=None, ordered=True)	Returns the top num minimum indicies.
atleast_nd(arr, n, front=False)	View inputs as arrays with at least n dimensions.
boolmask(indices, shape=None)	Constructs an array of booleans where an item is True if its position is in
isect_flags(arr, other)	Check which items in an array intersect with another set of items
iter_reduce_ufunc(ufunc, arrs, out=None, default=None)	constant memory iteration and reduction
normalize(arr, mode='linear', alpha=None, beta=None, out=None)	Rebalance signal values via contrast stretching.
ensure_rng(rng, api='numpy')	Coerces input into a random number generator.
random_combinations(items, size, num=None, rng=None)	Yields num combinations of length size from items in random order
random_product(items, num=None, rng=None)	Yields num items from the cartesian product of items in a random order.
seed_global(seed, offset=0)	Seeds the python, numpy, and torch global random states
shuffle(items, rng=None)	Shuffles a list inplace and then returns it for convinience
embed_slice(slices, data_dims, pad=None)	Embeds a "padded-slice" inside known data dimension.
padded_slice(data, slices, pad=None, padkw=None, return_info=False)	Allows slices with out-of-bound coordinates. Any out of bounds coordinate
one_hot_embedding(labels, num_classes, dim=1)	Embedding labels to one-hot form.
one_hot_lookup(data, indices)	Return value of a particular column for each row in data.

class kwarray.**ArrayAPI**

Bases: object

Compatability 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.

[3] http://stackoverflow.com/questions/7352684/groups-consecutive-elements

Example

```
>>> # Use the easy-to-use, but inefficient array api
>>> # xdoctest: +REQUIRES(module:torch)
>>> take = 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)
>>> compress = 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
>>> 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

_numpy

take

compress

repeat

tile

view

numel

atleast_nd

full_like

ones_like

zeros_like

empty_like

sum

argmax

argsort

max

maximum

minimum

matmul

astype

nonzero

nan_to_num

tensor

numpy

tolist

asarray

asarray

T

transpose

contiguous

pad

dtype_kind

max_argmax

any

all

log2

log

copy

iceil

ifloor

floor

ceil

round

iround

clip

softmax

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).

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

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

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

kwarray.**dtype_info**(*dtype*)

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

Returns an iinfo of finfo structure depending on the input type.

Return type struct

References

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

Example

```
>>> from kwarray.arrayapi import *  # NOQA
>>> 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)]
```

(continues on next page)

(continued from previous page)

```
>>>     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.**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 Eucliedean)

Returns

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

Return type Tuple[list, float]

Notes

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.**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[Hashable, List[Hashable]]*) – a dictionary where keys are the candidate set ids and each value is a candidate cover set.
- **items** (*Hashable, optional*) – the set of all items to be covered, if not specified, it is infered from the candidate cover sets
- **set_weights** (*Dict, optional*) – maps candidate set ids to a cost for using this candidate cover in the solution. If not specified the weight of each candiate cover defaults to 1.
- **item_values** (*Dict, optional*) – 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** (*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.repr2(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.repr2(exact_soln, nl=0)))
exact_soln = {'b': [1, 2, 3, 4, 5], 'e': [6, 7, 8, 9, 0]}
```

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

__normalize__(*self*)

Try to convert input data to Dict[ndarray]

extend(*self*, *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
```

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

NOTE: NOT A PART OF THE PANDAS API

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

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

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

Notes

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)
```

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

Bases: `ubelt.NiceRepr`

Implements a subset of the pandas.DataFrame API

The API is restricted to facilitate speed tradeoffs

Notes

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.

Notes

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)
>>> df_heavy = df_light.pandas()
```

(continues on next page)

(continued from previous page)

```
>>> 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(*self*)

property values(*self*)

property loc(*self*)

__eq__(*self*, *other*)

Example

```
>>> # xdoctest: +REQUIRES(module:pandas)
>>> self = DataFrameLight._demodata(num=7)
>>> other = self.pandas()
>>> assert np.all(self == other)
```

to_string(*self*, **args*, ***kwargs*)

to_dict(*self*, *orient='dict'*, *into=dict*)

Convert the data frame into a dictionary.

Parameters

- **orient** (*str*) – Currently naitively suports 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(*self*)

Convert back to pandas if you need the full API

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(*self*)

Deprecated, use self.pandas instead

classmethod _demodata(*cls*, *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
```

__nice__(*self*)

__len__(*self*)

__contains__(*self*, *item*)

__normalize__(*self*)

Try to convert input data to Dict[List]

property columns(*self*)

sort_values(*self*, *key*, *inplace=False*, *ascending=True*)

keys(*self*)

_getrow(*self*, *index*)

_getcol(*self*, *key*)

_getcols(*self*, *keys*)

get(*self*, *key*, *default=None*)
Get item for given key. Returns default value if not found.

clear(*self*)
Removes all rows inplace

__getitem__(*self*, *key*)

Note: only handles the case where key is a single column name.

Example

```
>>> df_light = DataFrameLight._demodata(num=7)
>>> sub1 = df_light['bar']
>>> # xdoctest: +REQUIRES(module:pandas)
>>> df_heavy = df_light.pandas()
>>> sub2 = df_heavy['bar']
>>> assert np.all(sub1 == sub2)
```

__setitem__(*self*, *key*, *value*)

Note: only handles the case where key is a single column name. and value is an array of all the values to set.

Example

```
>>> df_light = DataFrameLight._demodata(num=7)
>>> value = [2] * len(df_light)
>>> df_light['bar'] = value
>>> # xdoctest: +REQUIRES(module:pandas)
>>> df_heavy = df_light.pandas()
>>> df_heavy['bar'] = value
>>> assert np.all(df_light == df_heavy)
```

compress(*self*, *flags*, *inplace=False*)
NOTE: NOT A PART OF THE PANDAS API

take(*self*, *indices*, *inplace=False*)
Return the elements in the given *positional* indices along an axis.

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

Notes

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(*self*)

extend(*self*, *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(*self*, **others*)

Note: Note part of the pandas API

classmethod concat(*cls*, *others*)

classmethod from_pandas(*cls*, *df*)

classmethod from_dict(*cls*, *records*)

reset_index(*self*, *drop=False*)

noop for compatability, the light version doesnt store an index

groupby(*self*, *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)])
```

Ignore:

```
>>> self = DataFrameLight._demodata(num=1000)
>>> args = ['cx']
>>> self['cx'] = (np.random.rand(len(self)) * 10).astype(np.int)
>>> # As expected, our custom restricted implementation is faster
>>> # than pandas
>>> ub.Timerit(100).call(lambda: dict(list(self.pandas().groupby('cx')))).
↪print()
>>> ub.Timerit(100).call(lambda: dict(self.groupby('cx'))).print()
```

rename(*self*, *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(*self*)

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.repr2(list(self.iterrows())))
[
    (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.**LocLight**(*parent*)

Bases: object

__getitem__(*self*, *index*)

kwarray.**standard_normal**(*size*, *mean=0*, *std=1*, *dtype=float*, *rng=np.random*)

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

Parameters

- **size** (*int | Tuple[int, *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
- **dtype** (*type*) – either np.float32 or np.float64
- **rng** (*numpy.random.RandomState*) – underlying random state

Returns normally distributed random numbers with chosen dtype

Return type ndarray[dtype]

Benchmark:

```
>>> from timerit import Timerit
>>> import kwarray
>>> size = (300, 300, 3)
>>> for timer in Timerit(100, bestof=10, label='dtype=np.float32'):
```

(continues on next page)

(continued from previous page)

```
>>>     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=np.random*)

Fast normally distributed random variables using the Box–Muller transform

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, *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

Return type ndarray[float32]

References

https://en.wikipedia.org/wiki/Box%E2%80%93Muller_transform

SeeAlso:

- standard_normal
- standard_normal64

Example

```
>>> 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)
```

(continues on next page)

(continued from previous page)

```
>>> 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=np.random*)

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

Parameters

- **size** (*int | Tuple[int, *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

Return type ndarray[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.**uniform**(*low=0.0*, *high=1.0*, *size=None*, *dtype=np.float32*, *rng=np.random*)

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], default=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
- **rng** (*numpy.random.RandomState*) – underlying random state

Returns normally distributed random numbers with chosen dtype

Return type ndarray[dtype]

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=np.random*)

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], default=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.

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)
```

class kwarray.**RunningStats**(*run*)

Bases: `ubelt.NiceRepr`

Dynamically records per-element array statistics and can summarized 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.repr2(run.summarize(axis=ub.NoParam), nl=2,␣
↪precision=3))
>>> # Average over channels (keeps spatial dims separate)
>>> print('chann-ave(k=1) = ' + ub.repr2(run.summarize(axis=0), nl=2, precision=3))
>>> print('chann-ave(k=0) = ' + ub.repr2(run.summarize(axis=0, keepdims=0), nl=2,␣
↪precision=3))
>>> # Average over spatial dims (keeps channels separate)
>>> print('spatial-ave(k=1) = ' + ub.repr2(run.summarize(axis=(1, 2)), nl=2,␣
↪precision=3))
>>> print('spatial-ave(k=0) = ' + ub.repr2(run.summarize(axis=(1, 2), keepdims=0),␣
↪nl=2, precision=3))
>>> # Average over all dims
>>> print('alldim-ave(k=1) = ' + ub.repr2(run.summarize(axis=None), nl=2,␣
↪precision=3))
>>> print('alldim-ave(k=0) = ' + ub.repr2(run.summarize(axis=None, keepdims=0),␣
↪nl=2, precision=3))
```

__nice__(*self*)

property shape(*run*)

update(*run*, *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)
```

_sumsq_std(*run*, *total*, *squares*, *n*)

Sum of squares method to compute standard deviation

summarize(*run*, *axis=None*, *keepdims=True*)

Compute summary statistics across a one or more dimension

Parameters

- **axis** (*int | List[int] | None | ub.NoParam*) – 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

`current`(*run*)

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

Todo:

- **[X] I want this method and summarize to be unified somehow.** I don't know how to paramatarize it because axis=None usually means summarize over everything, and I need to way to encode, summarize over nothing but the "sequence" dimension (which was given incrementally by the update function), which is what this function does.

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

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
- **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

Returns

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

Return type collections.OrderedDict

SeeAlso: scipy.stats.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.repr2(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),
    'med': np.array([0.5584, 0.6805], dtype=np.float32),
    'shape': (10, 2),
}
```

Example

```
>>> # xdoctest: +IGNORE_WHITESPACE
>>> 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)
>>> import ubelt as ub  # NOQA
>>> result = str(ub.repr2(stats, nl=0, precision=1, strkeys=True))
>>> print(result)
{mean: 20.0, std: 13.2, min: 0.0, max: 41.0, num_nan: 1, shape: (100,)}
```

`kwarray.`**`apply_grouping`**(*items*, *groupxs*, *axis=0*)

Applies grouping from group_indicies.

Typically used in conjunction with `group_indices()`.

Parameters

- **items** (*ndarray*) – items to group
- **groupxs** (*List[ndarrays[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_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.**group_consecutive**(*arr*, *offset=1*)

Returns lists of consecutive values. Implementation inspired by[Page 81, 3].

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 thta are the same, e.g.: 4, 4, 4.

Returns a list of arrays that are the groups from the input

Return type List[ndarray]

Notes

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, default=False*) – If the input array is sorted, then setting this to True will avoid an unnecessary sorting operation and improve efficiency.

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[ndarrays]]

Example

```
>>> # xdoctest: +IGNORE_WHITESPACE
>>> import ubelt as ub
>>> idx_to_groupid = np.array([2, 1, 2, 1, 2, 1, 2, 3, 3, 3, 3])
>>> (keys, groupxs) = group_indices(idx_to_groupid)
>>> print(ub.repr2(keys, with_dtype=False))
>>> print(ub.repr2(groupxs, with_dtype=False))
np.array([1, 2, 3])
[
    np.array([1, 3, 5]),
    np.array([0, 2, 4, 6]),
    np.array([ 7,  8,  9, 10]),
]
```

Example

```
>>> # xdoctest: +IGNORE_WHITESPACE
>>> import ubelt as ub
>>> idx_to_groupid = np.array([[  24], [ 129], [ 659], [ 659], [ 24],
...       [659], [ 659], [ 822], [ 659], [ 659], [24]])
>>> # 2d arrays must be flattened before coming into this function so
>>> # information is on the last axis
>>> (keys, groupxs) = group_indices(idx_to_groupid.T[0])
>>> print(ub.repr2(keys, with_dtype=False))
>>> print(ub.repr2(groupxs, with_dtype=False))
np.array([ 24, 129, 659, 822])
[
    np.array([ 0,  4, 10]),
    np.array([1]),
    np.array([2, 3, 5, 6, 8, 9]),
    np.array([7]),
]
```

Example

```
>>> # xdoctest: +IGNORE_WHITESPACE
>>> import ubelt as ub
>>> idx_to_groupid = np.array([True, True, False, True, False, False, True])
>>> (keys, groupxs) = group_indices(idx_to_groupid)
>>> print(ub.repr2(keys, with_dtype=False))
>>> print(ub.repr2(groupxs, with_dtype=False))
np.array([False,  True])
[
    np.array([2, 4, 5]),
    np.array([0, 1, 3, 6]),
]
```

Example

```
>>> # xdoctest: +IGNORE_WHITESPACE
>>> import ubelt as ub
>>> idx_to_groupid = [('a', 'b'),  ('d', 'b'), ('a', 'b'), ('a', 'b')]
>>> (keys, groupxs) = group_indices(idx_to_groupid)
>>> print(ub.repr2(keys, with_dtype=False))
>>> print(ub.repr2(groupxs, with_dtype=False))
[
    ('a', 'b'),
    ('d', 'b'),
]
[
    np.array([0, 2, 3]),
    np.array([1]),
]
```

kwarray.**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][2].

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[T1]*) – The input array of items to group.
- **groupid_list** (*ndarray[T2]*) – 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.
- **assume_sorted** (*bool, default=False*) – If the input array is sorted, then setting this to True will avoid an unnecessary sorting operation and improve efficiency.
- **axis** (*int | None*) – group along a particular axis in `items` if it is n-dimensional

Returns mapping from groupids to corresponding items

Return type Dict[T2, ndarray[T1]]

References

Example

```
>>> from kwarray.util_groups import *  # NOQA
>>> items = np.array([0, 1, 2, 3, 4, 5, 6, 7])
>>> keys = np.array( [2, 2, 1, 1, 0, 1, 0, 1])
>>> grouped = group_items(items, keys)
>>> print(ub.repr2(grouped, nl=1, with_dtype=False))
{
    0: np.array([4, 6]),
```

(continues on next page)

[1] http://stackoverflow.com/questions/4651683/
[2] numpy-grouping-using-itertools-groupby-performance

(continued from previous page)

```
    1: np.array([2, 3, 5, 7]),
    2: np.array([0, 1]),
}
```

class kwarray.**FlatIndexer**(*lens*)

Bases: ubelt.NiceRepr

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

Parameters **lens** (*list*) – 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**(*cls*, *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

__len__(*self*)

unravel(*self*, *index*)

Parameters **index** – 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(*self*, *outer*, *inner*)

Parameters

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

Returns the raveled index

Return type index

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[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.argminima(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 argrees 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** (*array_like*) – 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, default=False*) – if True new dimensions are added to the front of the array. otherwise they are added to the back.

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.repr2(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.repr2(arr1_.tolist(), nl=0)
>>> result2 = ub.repr2(arr2_.tolist(), nl=0)
>>> result3 = ub.repr2(arr3_.tolist(), nl=0)
>>> result = '\n'.join([result1, result2, result3])
>>> print(result)
[[[[1]]], [[[1]]], [[[1]]]]
[[[[0]]]]
[[[[[1]]]]]
```

Notes

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 incoporate 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[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

```
>>> indices = np.array([(0, 0), (1, 1), (2, 1)])
>>> shape = (3, 3)
>>> mask = boolmask(indices, shape)
>>> import ubelt as ub  # NOQA
>>> result = ub.repr2(mask)
>>> print(result)
np.array([[ True, False, False],
          [False,  True, False],
          [False,  True, False]], dtype=np.bool)
```

kwarray.**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 that item 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]
```

(continues on next page)

(continued from previous page)

```
[False  True False  True]
[ True  True False  True]]
```

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

constant memory iteration and reduction

applys 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.**normalize**(*arr*, *mode='linear'*, *alpha=None*, *beta=None*, *out=None*)

Rebalance signal values via contrast stretching.

By default linearly stretches array values to minimum and maximum values.

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(abs(old_min - beta), abs(old_max - 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 `(max - min) / 2`. Note this parameter is sensitive to if the input is a float or uint8 image.

References

https://en.wikipedia.org/wiki/Normalization_(image_processing)

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')
>>> 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)
```

Benchmark: # Our method is faster than standard in-line implementations.

import timerit ti = timerit.Timerit(100, bestof=10, verbose=2, unit='ms') arr = kwimage.grab_test_image('lowcontrast', dsize=(512, 512))

print('— uint8 —') arr = ensure_float01(arr) out = arr.copy() for timer in ti.reset('naive1-float'):

with timer: (arr - arr.min()) / (arr.max() - arr.min())

import timerit for timer in ti.reset('simple-float'):

with timer: **max_** = arr.max() **min_** = arr.min() result = (arr - **min_**) / (**max_** - **min_**)

for timer in ti.reset('normalize-float'):

with timer: normalize(arr)

for timer in ti.reset('normalize-float-inplace'):

with timer: normalize(arr, out=out)

print('— float —') arr = ensure_uint255(arr) out = arr.copy() for timer in ti.reset('naive1-uint8'):

with timer: (arr - arr.min()) / (arr.max() - arr.min())

import timerit for timer in ti.reset('simple-uint8'):

with timer: **max_** = arr.max() **min_** = arr.min() result = (arr - **min_**) / (**max_** - **min_**)

for timer in ti.reset('normalize-uint8'):

with timer: normalize(arr)

for timer in ti.reset('normalize-uint8-inplace'):

with timer: normalize(arr, out=out)

Ignore: globals().update(xdev.get_func_kwargs(normalize))

`kwarray.ensure_rng(rng, 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 | numpy.random.RandomState | random.Random | None*) – if None, then defaults to the global rng. Otherwise this can be an integer or a RandomState class
- **api** (*str, default='numpy'*) – 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.

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')
```

Ignore:

```
>>> np.random.seed(0)
>>> np.random.randint(0, 10000)
2732
>>> np.random.seed(0)
>>> np.random.mtrand._rand.randint(0, 10000)
```

(continues on next page)

(continued from previous page)

```
2732
>>> np.random.seed(0)
>>> ensure_rng(None).randint(0, 10000)
2732
>>> np.random.randint(0, 10000)
9845
>>> ensure_rng(None).randint(0, 10000)
3264
```

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** (*None, default=None*) – number of combinations to generate
- **rng** (*int | RandomState, default=None*) – seed or random number generator

Yields *Tuple* – a random combination of items of length size.

Example

```
>>> 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.repr2(combos, nl=1)))
combos = [
    (0, 6, 9),
    (4, 7, 8),
    (4, 6, 7),
    (2, 3, 5),
    (1, 2, 4),
]
```

Example

```
>>> 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.repr2(combos, nl=1)))
combos = [
    ((0, 0), (6, 6), (9, 9)),
    ((4, 4), (7, 7), (8, 8)),
```

(continues on next page)

(continued from previous page)

```
    ((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 caresian product of packed in a list or tuple. (note this deviates from api of `itertools.product()`)
- **num** (*int, default=None*) – maximum number of items to generate. If None, all
- **rng** (*random.Random | np.random.RandomState | int*) – random number generator

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.repr2(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.repr2(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.**seed_global**(*seed*, *offset=0*)

Seeds the python, numpy, and torch global random states

Parameters

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

kwarray.**shuffle**(*items*, *rng=None*)

Shuffles a list inplace and then returns it for convinience

Parameters

- **items** (*list or ndarray*) – list to shuffle
- **rng** (*RandomState or int*) – 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.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** (*List[int|Tuple]*) – extra pad applied to (left and right) / (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
>>> from kwarray.util_slices import *  # NOQA
>>> slices = (slice(0, 10), slice(0, 10))
>>> data_dims  = [300, 300]
>>> pad        = [10, 5]
>>> a, b = 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
>>> slices = (slice(-10, 400), slice(-10, 400))
>>> data_dims  = [300, 300]
>>> pad        = [10, 5]
>>> a, b = 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
>>> slices = (slice(10, 40), slice(10, 40))
>>> data_dims  = [300, 300]
>>> pad        = None
>>> a, b = 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)]
```

`kwarray.`**`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[T]*) – 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

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

```
>>> data_sliced = padded_slice(data, slices)
>>> print(ub.repr2(data_sliced, with_dtype=False))
np.array([0, 0, 0, 1, 2, 3, 4, 0, 0])
```

```
>>> data_sliced = padded_slice(data, slices, pad=(3, 3))
>>> print(ub.repr2(data_sliced, with_dtype=False))
np.array([0, 0, 0, 0, 0, 0, 1, 2, 3, 4, 0, 0, 0, 0, 0])
```

```
>>> data_sliced = padded_slice(data, slice(3, 4), pad=[(1, 0)])
>>> print(ub.repr2(data_sliced, with_dtype=False))
np.array([2, 3])
```

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

Bases: `ubelt.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 <= 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

Notes

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

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))
```

(continues on next page)

(continued from previous page)

```
[[(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 returend.
>>> 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))]
```

__nice__(*self*)

_compute_stride(*self*, *overlap*, *stride*, *shape*, *window*)

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

__len__(*self*)

_iter_basis_frac(*self*)

__iter__(*self*)

__getitem__(*self*, *index*)

Get a specific item by its flat (raveled) index

Example

```
>>> from kwarray.util_slider import *  # NOQA
>>> window = (10, 10)
>>> shape = (20, 20)
>>> self = SlidingWindow(shape, window, stride=5)
>>> itered_items = list(self)
>>> assert len(itered_items) == len(self)
>>> indexed_items = [self[i] for i in range(len(self))]
>>> assert itered_items[0] == self[0]
>>> assert itered_items[-1] == self[-1]
>>> assert itered_items == indexed_items
```

property grid(*self*)

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

Yields Tuple[int, ...]

property slices(*self*)

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(*self*)

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**(*stitcher*, *shape*, *device='numpy'*)

Bases: `ubelt.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.

Parameters **shape** (*tuple*) – dimensions of the large image that will be created from the smaller pixels or patches.

Todo:

- **[ ] Look at the old "add_fast" code in the netharn version and see if** it is worth porting. This code is kept in the dev folder in ../dev/_dev_slider.py

---

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
>>> stitcher = Stitcher(slider.input_shape)
>>> for sl in list(slider):
...     chip = highres[sl]
...     stitcher.add(sl, chip)
>>> assert stitcher.weights.max() == 4, 'some parts should be processed 4 times'
>>> recon = stitcher.finalize()
```

__nice__(*stitcher*)

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

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

Parameters

- **indices** (*slice or tuple*) – 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 or ndarray*) – weight of this patch (default to 1.0)

average(*stitcher*)

Averages out contributions from overlapping adds using weighted average

Returns ndarray: the stitched image

Return type out

finalize(*stitcher*, *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 ndarray: the stitched image

Return type final

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)
>>> 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.repr2(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())
>>> if torch.cuda.is_available():
>>>     t3 = one_hot_embedding(labels.to(0), num_classes)
>>>     assert np.all(t3.cpu().numpy() == t.numpy())
```

Example

```
>>> # xdoctest: +REQUIRES(module: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 corresonds 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

Notes

This is functionally equivalent to `[row[c] for row, c in zip(data, indices)]` except that it is 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()
```

(continues on next page)

(continued from previous page)

```
>>> 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)
```

Ignore:

```
>>> # xdoctest: +REQUIRES(module:torch, module:onnx, module:onnx_tf)
>>> # Test if this converts to ONNX
>>> from kwarray.util_torch import *  # NOQA
>>> import torch.onnx
>>> import io
>>> import onnx
>>> import onnx_tf.backend
>>> import numpy as np
>>> 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()
>>> class TFConvertWrapper(torch.nn.Module):
>>>     def forward(self, data, indices):
>>>         return one_hot_lookup(data, indices)
>>> ###
>>> # Test the ONNX export
>>> wrapped = TFConvertWrapper()
>>> onnx_file = io.BytesIO()
>>> torch.onnx.export(
>>>     wrapped, tuple([data, indices]),
>>>     input_names=['data', 'indices'],
>>>     output_names=['out'],
>>>     f=onnx_file,
>>>     opset_version=11,
>>>     verbose=1,
>>> )
>>> onnx_file.seek(0)
>>> onnx_model = onnx.load(onnx_file)
>>> onnx_tf_model = onnx_tf.backend.prepare(onnx_model)
>>> # Test that the resulting graph tensors are concretely sized.
>>> import tensorflow as tf
>>> onnx_gd = onnx_tf_model.graph.as_graph_def()
>>> output_tensors = tf.import_graph_def(
>>>     onnx_gd,
>>>     input_map={},
>>>     return_elements=[onnx_tf_model.tensor_dict[ol].name for ol in onnx_tf_
↪model.outputs]
>>> )
>>> assert all(isinstance(d.value, int) for t in output_tensors for d in t.
↪shape)
```

(continues on next page)

(continued from previous page)

```
>>> tf_outputs = onnx_tf_model.run([data, indices])
>>> pt_outputs = wrapped(data, indices)
>>> print('tf_outputs = {!r}'.format(tf_outputs))
>>> print('pt_outputs = {!r}'.format(pt_outputs))
>>> ###
>>> # Test if data is more than 2D
>>> shape = (4, 3, 8)
>>> data = torch.arange(int(np.prod(shape))).view(*shape).float()
>>> indices = torch.from_numpy(np.array([0, 1, 2, 1])).long()
>>> onnx_file = io.BytesIO()
>>> torch.onnx.export(
>>>     wrapped, tuple([data, indices]),
>>>     input_names=['data', 'indices'],
>>>     output_names=['out'],
>>>     f=onnx_file,
>>>     opset_version=11,
>>>     verbose=1,
>>> )
>>> onnx_file.seek(0)
>>> onnx_model = onnx.load(onnx_file)
>>> onnx_tf_model = onnx_tf.backend.prepare(onnx_model)
>>> # Test that the resulting graph tensors are concretely sized.
>>> import tensorflow as tf
>>> onnx_gd = onnx_tf_model.graph.as_graph_def()
>>> output_tensors = tf.import_graph_def(
>>>     onnx_gd,
>>>     input_map={},
>>>     return_elements=[onnx_tf_model.tensor_dict[ol].name for ol in onnx_tf_
↪model.outputs]
>>> )
>>> assert all(isinstance(d.value, int) for t in output_tensors for d in t.
↪shape)
>>> tf_outputs = onnx_tf_model.run([data, indices])
>>> pt_outputs = wrapped(data, indices)
>>> print('tf_outputs = {!r}'.format(tf_outputs))
>>> print('pt_outputs = {!r}'.format(pt_outputs))
```

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

k

INDEX

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

Z