kwimage Documentation

Release 0.5.7

Jon Crall

Jan 23, 2020

Contents:

CHAPTER 1

Function Usefulness

Function name	Usefulness
kwimage.Boxes()	303
kwimage.Detections()	184
kwimage.Color()	134
kwimage.Polygon()	114
kwimage.grab_test_image()	81
kwimage.imread()	81
kwimage.imwrite()	71
kwimage.MultiPolygon()	71
kwimage.ensure_float01()	70
kwimage.Heatmap()	70
kwimage.imresize()	65
kwimage.ensure_alpha_channel()	61
kwimage.Mask()	59
kwimage.Points()	58
kwimage.ensure_uint255()	56
kwimage.Coords()	53
kwimage.convert_colorspace()	53
kwimage.atleast_3channels()	47
kwimage.draw_text_on_image()	46
kwimage.overlay_alpha_images()	43
kwimage.overlay_alpha_layers()	34
kwimage.PolygonList()	32
kwimage.warp_tensor()	26
kwimage.grab_test_image_fpath()	25
kwimage.stack_images()	24
kwimage.PointsList()	22
kwimage.stack_images_grid()	20
kwimage.gaussian_patch()	17

Continued on next page

Table 1 – continued from previous page

Function name	Usefulness
kwimage.imscale()	17
kwimage.encode_run_length()	14
kwimage.non_max_supression()	12
kwimage.MaskList()	10
kwimage.daq_spatial_nms()	9
kwimage.warp_points()	8
kwimage.make_vector_field()	8
kwimage.draw_clf_on_image()	8
kwimage.subpixel_accum()	7
kwimage.decode_run_length()	7
kwimage.rle_translate()	6
kwimage.subpixel_slice()	4
kwimage.subpixel_setvalue()	4
kwimage.subpixel_maximum()	4
kwimage.subpixel_translate()	4
kwimage.num_channels()	4
kwimage.subpixel_getvalue()	2

1.1 API Reference

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

1.1.1 kwimage

mkinit ~/code/kwimage/kwimage/algo/__init__.py –relative -w –nomod mkinit ~/code/kwimage/kwimage/structs/__init__.py –relative -w –nomod mkinit ~/code/kwimage/kwimage/__init__.py –relative -w –nomod

Subpackages

kwimage.algo

mkinit ~/code/kwimage/kwimage/algo/__init__.py -w –relative

Subpackages

kwimage.algo._nms_backend

Submodules

kwimage.algo._nms_backend.py_nms

Fast R-CNN Copyright (c) 2015 Microsoft Licensed under The MIT License [see LICENSE for details] Written by Ross Girshick

[1] Created with sphinx-autoapi

Module Contents

kwimage.algo._nms_backend.py_nms.**py_nms**(*np_tlbr*, *np_scores*, *thresh*, *bias=1*)
Pure Python NMS baseline.

References

https://github.com/rbgirshick/fast-rcnn/blob/master/lib/utils/nms.py

Example

```
>>> np_tlbr = np.array([
>>>     [0, 0, 100, 100],
>>>     [0, 0, 100, 100],
>>>     [100, 100, 10, 10],
>>>     [10, 10, 100, 100],
>>>     [50, 50, 100, 100],
>>>     [100, 100, 150, 101],
>>>     [120, 100, 180, 101],
>>>     [150, 100, 200, 101],
>>> ], dtype=np.float32)
>>> np_scores = np.linspace(0, 1, len(np_tlbr))
>>> thresh = 0.1
>>> bias = 0.0
>>> keep = sorted(py_nms(np_tlbr, np_scores, thresh, bias))
>>> print('keep = {!r}'.format(keep))
keep = [2, 4, 5, 7]
```

Example

```
>>> from kwimage.algo._nms_backend.py_nms import *  # NOQA
>>> np_tlbr = np.array([
>>>     [0, 0, 100, 100],
>>>     [100, 100, 10, 10],
>>>     [10, 10, 100, 100],
>>>     [50, 50, 100, 100],
>>> ], dtype=np.float32)
>>> np_scores = np.array([.1, .5, .9, .1])
>>> keep = list(py_nms(np_tlbr, np_scores, thresh=0.0, bias=1.0))
>>> print('keep@0.0 = {!r}'.format(keep))
>>> keep = list(py_nms(np_tlbr, np_scores, thresh=0.2, bias=1.0))
>>> print('keep@0.2 = {!r}'.format(keep))
>>> keep = list(py_nms(np_tlbr, np_scores, thresh=0.5, bias=1.0))
>>> print('keep@0.5 = {!r}'.format(keep))
>>> keep = list(py_nms(np_tlbr, np_scores, thresh=1.0, bias=1.0))
>>> print('keep@1.0 = {!r}'.format(keep))
keep@0.0 = [2, 1]
keep@0.2 = [2, 1]
keep@0.5 = [2, 1, 3]
keep@1.0 = [2, 1, 3, 0]
```

kwimage.algo._nms_backend.torch_nms

Module Contents

kwimage.algo._nms_backend.torch_nms.**_TORCH_HAS_BOOL_COMP**

kwimage.algo._nms_backend.torch_nms.**torch_nms**(*tlbr*, *scores*, *classes=None*, *thresh=0.5*, *bias=0*, *fast=False*)

Non maximum suppression implemented with pytorch tensors

CURRENTLY NOT WORKING

Parameters

- **tlbr** (*Tensor*) – Bounding boxes of one image in the format (tlbr)
- **scores** (*Tensor*) – Scores of each box
- **classes** (*Tensor, optional*) – the classes of each box. If specified nms is applied to each class separately.
- **thresh** (*float*) – iou threshold

Returns keep: boolean array indicating which boxes were not pruned.

Return type ByteTensor

Example

```
>>> # DISABLE_DOCTEST
>>> import torch
>>> import numpy as np
>>> tlbr = torch.FloatTensor(np.array([
>>>     [0, 0, 100, 100],
>>>     [100, 100, 10, 10],
>>>     [10, 10, 100, 100],
>>>     [50, 50, 100, 100],
>>>     [100, 100, 130, 130],
>>>     [100, 100, 130, 130],
>>>     [100, 100, 130, 130],
>>> ], dtype=np.float32))
>>> scores = torch.FloatTensor(np.array([.1, .5, .9, .1, .3, .5, .4]))
>>> classes = torch.LongTensor(np.array([0, 0, 0, 0, 0, 0, 0]))
>>> thresh = .5
>>> flags = torch_nms(tlbr, scores, classes, thresh)
>>> keep = np.nonzero(flags).view(-1)
>>> tlbr[flags]
>>> tlbr[keep]
```

Example

```
>>> # DISABLE_DOCTEST
>>> import torch
>>> import numpy as np
>>> # Test to check that conflicts are correctly resolved
>>> tlbr = torch.FloatTensor(np.array([
>>>     [100, 100, 150, 101],
```

(continues on next page)

(continued from previous page)

```
>>>     [120, 100, 180, 101],
>>>     [150, 100, 200, 101],
>>> ], dtype=np.float32))
>>> scores = torch.FloatTensor(np.linspace(.8, .9, len(tlbr)))
>>> classes = None
>>> thresh = .3
>>> keep = torch_nms(tlbr, scores, classes, thresh, fast=False)
>>> bboxes[keep]
```

kwimage.algo._nms_backend.torch_nms.**test_class_torch**()

Submodules

kwimage.algo.algo_nms

Generic Non-Maximum Suppression API with efficient backend implementations

Module Contents

kwimage.algo.algo_nms.**daq_spatial_nms**(*tlbr*, *scores*, *diameter*, *thresh*, *max_depth=6*, *stop_size=2048*, *recsize=2048*, *impl='auto'*, *device_id=None*)

Divide and conquor speedup non-max-supression algorithm for when bboxes have a known max size

Parameters

- **tlbr** (*ndarray*) – boxes in (tlx, tly, brx, bry) format
- **scores** (*ndarray*) – scores of each box
- **diameter** (*int or Tuple[int, int]*) – Distance from split point to consider rectification. If specified as an integer, then number is used for both height and width. If specified as a tuple, then dims are assumed to be in [height, width] format.
- **thresh** (*float*) – iou threshold. Boxes are removed if they overlap greater than this threshold. 0 is the most strict, resulting in the fewest boxes, and 1 is the most permissive resulting in the most.
- **max_depth** (*int*) – maximum number of times we can divide and conquor
- **stop_size** (*int*) – number of boxes that triggers full NMS computation
- **recsize** (*int*) – number of boxes that triggers full NMS recombination
- **impl** (*str*) – algorithm to use

LookInfo: # Didn't read yet but it seems similar http://www.cyberneum.de/fileadmin/user_upload/files/publications/CVPR2010-Lampert_{[}0{]}.pdf

https://www.researchgate.net/publication/220929789_Efficient_Non-Maximum_Suppression

This seems very similar https://projet.liris.cnrs.fr/m2disco/pub/Congres/2006-ICPR/DATA/C03_0406.PDF

Example

```
>>> import kwimage
>>> # Make a bunch of boxes with the same width and height
>>> #boxes = kwimage.Boxes.random(230397, scale=1000, format='cxywh')
>>> boxes = kwimage.Boxes.random(237, scale=1000, format='cxywh')
>>> boxes.data.T[2] = 10
>>> boxes.data.T[3] = 10
>>> #
>>> tlbr = boxes.to_tlbr().data.astype(np.float32)
>>> scores = np.arange(0, len(tlbr)).astype(np.float32)
>>> #
>>> n_megabytes = (tlbr.size * tlbr.dtype.itemsize) / (2 ** 20)
>>> print('n_megabytes = {!r}'.format(n_megabytes))
>>> #
>>> thresh = iou_thresh = 0.01
>>> impl = 'auto'
>>> max_depth = 20
>>> diameter = 10
>>> stop_size = 2000
>>> recsize = 500
>>> #
>>> import ubelt as ub
>>> #
>>> with ub.Timer(label='daq'):
>>>     keep1 = daq_spatial_nms(tlbr, scores,
>>>         diameter=diameter, thresh=thresh, max_depth=max_depth,
>>>         stop_size=stop_size, recsize=recsize, impl=impl)
>>> #
>>> with ub.Timer(label='full'):
>>>     keep2 = non_max_supression(tlbr, scores,
>>>         thresh=thresh, impl=impl)
>>> #
>>> # Due to the greedy nature of the algorithm, there will be slight
>>> # differences in results, but they will be mostly similar.
>>> similarity = len(set(keep1) & set(keep2)) / len(set(keep1) | set(keep2))
>>> print('similarity = {!r}'.format(similarity))
```

kwimage.algo.algo_nms.**_impls**

class kwimage.algo.algo_nms.**_NMS_Impls**

_lazy_init(*self*)

kwimage.algo.algo_nms.**_impls**

kwimage.algo.algo_nms.**available_nms_impls**()

List available values for the *impl* kwarg of *non_max_supression*

CommandLine: xdoctest -m kwimage.algo.algo_nms available_nms_impls

Example

```
>>> impls = available_nms_impls()
>>> assert 'numpy' in impls
>>> print('impls = {!r}'.format(impls))
```

kwimage.algo.algo_nms.**_heuristic_auto_nms_impl**(*code*, *num*, *valid=None*)
Defined with help from ~/code/kwimage/dev/bench_nms.py

Parameters

- **code** (*str*) – text that indicates which type of data you have tensor0 is a tensor on a cuda device, tensor is on the cpu, and numpy is a ndarray.
- **num** (*int*) – number of boxes you have to supress.
- **valid** (*List[str]*) – the list of valid implementations, an error will be raised if heuristic preferences do not intersect with this list.

Ignore: _impls._funcs valid_pref = ub.oset(preference) & set(_impls._funcs.keys()) python ~/code/kwimage/dev/bench_nms.py –show –small-boxes –thresh=0.6

kwimage.algo.algo_nms.**non_max_supression**(*tlbr*, *scores*, *thresh*, *bias=0.0*, *classes=None*, *impl='auto'*, *device_id=None*)
Non-Maximum Suppression - remove redundant bounding boxes

Parameters

- **tlbr** (*ndarray[float32]*) – Nx4 boxes in tlbr format
- **scores** (*ndarray[float32]*) – score for each bbox
- **thresh** (*float*) – iou threshold. Boxes are removed if they overlap greater than this threshold (i.e. Boxes are removed if iou > threshold). Thresh = 0 is the most strict, resulting in the fewest boxes, and 1 is the most permissive resulting in the most.
- **bias** (*float*) – bias for iou computation either 0 or 1
- **classes** (*ndarray[int64] or None*) – integer classes. If specified NMS is done on a perclass basis.
- **impl** (*str*) – implementation can be auto, python, cython_cpu, or gpu
- **device_id** (*int*) – used if impl is gpu, device id to work on. If not specified *torch.cuda.current_device()* is used.

Notes

Using impl='cython_gpu' may result in an CUDA memory error that is not exposed to the python processes. In other words your program will hard crash if impl='cython_gpu', and you feed it too many bounding boxes. Ideally this will be fixed in the future.

References

https://github.com/facebookresearch/Detectron/blob/master/detectron/utils/cython_nms.pyx https://www.pyimagesearch.com/2015/02/16/faster-non-maximum-suppression-python/ https://github.com/bharatsingh430/soft-nms/blob/master/lib/nms/cpu_nms.pyx <- TODO

CommandLine: xdoctest -m ~/code/kwimage/kwimage/algo/algo_nms.py non_max_supression

Example

```
>>> from kwimage.algo.algo_nms import *
>>> from kwimage.algo.algo_nms import _impls
>>> tlbr = np.array([
>>>     [0, 0, 100, 100],
>>>     [100, 100, 10, 10],
>>>     [10, 10, 100, 100],
>>>     [50, 50, 100, 100],
>>> ], dtype=np.float32)
>>> scores = np.array([.1, .5, .9, .1])
>>> keep = non_max_supression(tlbr, scores, thresh=0.5, impl='numpy')
>>> print('keep = {!r}'.format(keep))
>>> assert keep == [2, 1, 3]
>>> thresh = 0.0
>>> non_max_supression(tlbr, scores, thresh, impl='numpy')
>>> if 'numpy' in available_nms_impls():
>>>     keep = non_max_supression(tlbr, scores, thresh, impl='numpy')
>>>     assert list(keep) == [2, 1]
>>> if 'cython_cpu' in available_nms_impls():
>>>     keep = non_max_supression(tlbr, scores, thresh, impl='cython_cpu')
>>>     assert list(keep) == [2, 1]
>>> if 'cython_gpu' in available_nms_impls():
>>>     keep = non_max_supression(tlbr, scores, thresh, impl='cython_gpu')
>>>     assert list(keep) == [2, 1]
>>> if 'torch' in available_nms_impls():
>>>     keep = non_max_supression(tlbr, scores, thresh, impl='torch')
>>>     assert set(keep.tolist()) == {2, 1}
>>> if 'torchvision' in available_nms_impls():
>>>     keep = non_max_supression(tlbr, scores, thresh, impl='torchvision')  #␣
↪note torchvision has no bias
>>>     assert list(keep) == [2]
>>> thresh = 1.0
>>> if 'numpy' in available_nms_impls():
>>>     keep = non_max_supression(tlbr, scores, thresh, impl='numpy')
>>>     assert list(keep) == [2, 1, 3, 0]
>>> if 'cython_cpu' in available_nms_impls():
>>>     keep = non_max_supression(tlbr, scores, thresh, impl='cython_cpu')
>>>     assert list(keep) == [2, 1, 3, 0]
>>> if 'cython_gpu' in available_nms_impls():
>>>     keep = non_max_supression(tlbr, scores, thresh, impl='cython_gpu')
>>>     assert list(keep) == [2, 1, 3, 0]
>>> if 'torch' in available_nms_impls():
>>>     keep = non_max_supression(tlbr, scores, thresh, impl='torch')
>>>     assert set(keep.tolist()) == {2, 1, 3, 0}
>>> if 'torchvision' in available_nms_impls():
>>>     keep = non_max_supression(tlbr, scores, thresh, impl='torchvision')  #␣
↪note torchvision has no bias
>>>     assert set(kwarray.ArrayAPI.tolist(keep)) == {2, 1, 3, 0}
```

Example

```
>>> import ubelt as ub
>>> tlbr = np.array([
>>>     [0, 0, 100, 100],
>>>     [100, 100, 10, 10],
>>>     [10, 10, 100, 100],
>>>     [50, 50, 100, 100],
```

(continues on next page)

(continued from previous page)

```
>>>     [100, 100, 150, 101],
>>>     [120, 100, 180, 101],
>>>     [150, 100, 200, 101],
>>> ], dtype=np.float32)
>>> scores = np.linspace(0, 1, len(tlbr))
>>> thresh = .2
>>> solutions = {}
>>> if not _impls._funcs:
>>>     _impls._lazy_init()
>>> for impl in _impls._funcs:
>>>     keep = non_max_supression(tlbr, scores, thresh, impl=impl)
>>>     solutions[impl] = sorted(keep)
>>> assert 'numpy' in solutions
>>> print('solutions = {}'.format(ub.repr2(solutions, nl=1)))
>>> assert ub.allsame(solutions.values())
```

CommandLine: xdoctest -m ~/code/kwimage/kwimage/algo/algo_nms.py non_max_supression

Example

```
>>> import ubelt as ub
>>> # Check that zero-area boxes are ok
>>> tlbr = np.array([
>>>     [0, 0, 0, 0],
>>>     [0, 0, 0, 0],
>>>     [10, 10, 10, 10],
>>> ], dtype=np.float32)
>>> scores = np.array([1, 2, 3], dtype=np.float32)
>>> thresh = .2
>>> solutions = {}
>>> if not _impls._funcs:
>>>     _impls._lazy_init()
>>> for impl in _impls._funcs:
>>>     keep = non_max_supression(tlbr, scores, thresh, impl=impl)
>>>     solutions[impl] = sorted(keep)
>>> assert 'numpy' in solutions
>>> print('solutions = {}'.format(ub.repr2(solutions, nl=1)))
>>> assert ub.allsame(solutions.values())
```

Package Contents

kwimage.algo.**available_nms_impls**()

List available values for the *impl* kwarg of *non_max_supression*

CommandLine: xdoctest -m kwimage.algo.algo_nms available_nms_impls

Example

```
>>> impls = available_nms_impls()
>>> assert 'numpy' in impls
>>> print('impls = {!r}'.format(impls))
```

kwimage.algo.**daq_spatial_nms**(*tlbr*, *scores*, *diameter*, *thresh*, *max_depth=6*, *stop_size=2048*, *recsize=2048*, *impl='auto'*, *device_id=None*)

Divide and conquor speedup non-max-supression algorithm for when bboxes have a known max size

Parameters

- **tlbr** (*ndarray*) – boxes in (tlx, tly, brx, bry) format
- **scores** (*ndarray*) – scores of each box
- **diameter** (*int or Tuple[int, int]*) – Distance from split point to consider rectification. If specified as an integer, then number is used for both height and width. If specified as a tuple, then dims are assumed to be in [height, width] format.
- **thresh** (*float*) – iou threshold. Boxes are removed if they overlap greater than this threshold. 0 is the most strict, resulting in the fewest boxes, and 1 is the most permissive resulting in the most.
- **max_depth** (*int*) – maximum number of times we can divide and conquor
- **stop_size** (*int*) – number of boxes that triggers full NMS computation
- **recsize** (*int*) – number of boxes that triggers full NMS recombination
- **impl** (*str*) – algorithm to use

LookInfo: # Didn't read yet but it seems similar http://www.cyberneum.de/fileadmin/user_upload/files/publications/CVPR2010-Lampert_{[}0{]}.pdf

https://www.researchgate.net/publication/220929789_Efficient_Non-Maximum_Suppression

This seems very similar https://projet.liris.cnrs.fr/m2disco/pub/Congres/2006-ICPR/DATA/C03_0406.PDF

Example

```
>>> import kwimage
>>> # Make a bunch of boxes with the same width and height
>>> #boxes = kwimage.Boxes.random(230397, scale=1000, format='cxywh')
>>> boxes = kwimage.Boxes.random(237, scale=1000, format='cxywh')
>>> boxes.data.T[2] = 10
>>> boxes.data.T[3] = 10
>>> #
>>> tlbr = boxes.to_tlbr().data.astype(np.float32)
>>> scores = np.arange(0, len(tlbr)).astype(np.float32)
>>> #
>>> n_megabytes = (tlbr.size * tlbr.dtype.itemsize) / (2 ** 20)
>>> print('n_megabytes = {!r}'.format(n_megabytes))
>>> #
>>> thresh = iou_thresh = 0.01
>>> impl = 'auto'
>>> max_depth = 20
>>> diameter = 10
>>> stop_size = 2000
>>> recsize = 500
>>> #
>>> import ubelt as ub
>>> #
>>> with ub.Timer(label='daq'):
>>>     keep1 = daq_spatial_nms(tlbr, scores,
```

(continues on next page)

(continued from previous page)

```
>>>         diameter=diameter, thresh=thresh, max_depth=max_depth,
>>>         stop_size=stop_size, recsize=recsize, impl=impl)
>>> #
>>> with ub.Timer(label='full'):
>>>     keep2 = non_max_supression(tlbr, scores,
>>>         thresh=thresh, impl=impl)
>>> #
>>> # Due to the greedy nature of the algorithm, there will be slight
>>> # differences in results, but they will be mostly similar.
>>> similarity = len(set(keep1) & set(keep2)) / len(set(keep1) | set(keep2))
>>> print('similarity = {!r}'.format(similarity))
```

kwimage.algo.**non_max_supression**(*tlbr*, *scores*, *thresh*, *bias=0.0*, *classes=None*, *impl='auto'*, *device_id=None*)

Non-Maximum Suppression - remove redundant bounding boxes

Parameters

- **tlbr** (*ndarray[float32]*) – Nx4 boxes in tlbr format
- **scores** (*ndarray[float32]*) – score for each bbox
- **thresh** (*float*) – iou threshold. Boxes are removed if they overlap greater than this threshold (i.e. Boxes are removed if iou > threshold). Thresh = 0 is the most strict, resulting in the fewest boxes, and 1 is the most permissive resulting in the most.
- **bias** (*float*) – bias for iou computation either 0 or 1
- **classes** (*ndarray[int64] or None*) – integer classes. If specified NMS is done on a perclass basis.
- **impl** (*str*) – implementation can be auto, python, cython_cpu, or gpu
- **device_id** (*int*) – used if impl is gpu, device id to work on. If not specified *torch.cuda.current_device()* is used.

Notes

Using impl='cython_gpu' may result in an CUDA memory error that is not exposed to the python processes. In other words your program will hard crash if impl='cython_gpu', and you feed it too many bounding boxes. Ideally this will be fixed in the future.

References

https://github.com/facebookresearch/Detectron/blob/master/detectron/utils/cython_nms.pyx https://www.pyimagesearch.com/2015/02/16/faster-non-maximum-suppression-python/ https://github.com/bharatsingh430/soft-nms/blob/master/lib/nms/cpu_nms.pyx <- TODO

CommandLine: xdoctest -m ~/code/kwimage/kwimage/algo/algo_nms.py non_max_supression

Example

```
>>> from kwimage.algo.algo_nms import *
>>> from kwimage.algo.algo_nms import _impls
>>> tlbr = np.array([
```

(continues on next page)

(continued from previous page)

```
>>>     [0, 0, 100, 100],
>>>     [100, 100, 10, 10],
>>>     [10, 10, 100, 100],
>>>     [50, 50, 100, 100],
>>> ], dtype=np.float32)
>>> scores = np.array([.1, .5, .9, .1])
>>> keep = non_max_supression(tlbr, scores, thresh=0.5, impl='numpy')
>>> print('keep = {!r}'.format(keep))
>>> assert keep == [2, 1, 3]
>>> thresh = 0.0
>>> non_max_supression(tlbr, scores, thresh, impl='numpy')
>>> if 'numpy' in available_nms_impls():
>>>     keep = non_max_supression(tlbr, scores, thresh, impl='numpy')
>>>     assert list(keep) == [2, 1]
>>> if 'cython_cpu' in available_nms_impls():
>>>     keep = non_max_supression(tlbr, scores, thresh, impl='cython_cpu')
>>>     assert list(keep) == [2, 1]
>>> if 'cython_gpu' in available_nms_impls():
>>>     keep = non_max_supression(tlbr, scores, thresh, impl='cython_gpu')
>>>     assert list(keep) == [2, 1]
>>> if 'torch' in available_nms_impls():
>>>     keep = non_max_supression(tlbr, scores, thresh, impl='torch')
>>>     assert set(keep.tolist()) == {2, 1}
>>> if 'torchvision' in available_nms_impls():
>>>     keep = non_max_supression(tlbr, scores, thresh, impl='torchvision')  #␣
↪note torchvision has no bias
>>>     assert list(keep) == [2]
>>> thresh = 1.0
>>> if 'numpy' in available_nms_impls():
>>>     keep = non_max_supression(tlbr, scores, thresh, impl='numpy')
>>>     assert list(keep) == [2, 1, 3, 0]
>>> if 'cython_cpu' in available_nms_impls():
>>>     keep = non_max_supression(tlbr, scores, thresh, impl='cython_cpu')
>>>     assert list(keep) == [2, 1, 3, 0]
>>> if 'cython_gpu' in available_nms_impls():
>>>     keep = non_max_supression(tlbr, scores, thresh, impl='cython_gpu')
>>>     assert list(keep) == [2, 1, 3, 0]
>>> if 'torch' in available_nms_impls():
>>>     keep = non_max_supression(tlbr, scores, thresh, impl='torch')
>>>     assert set(keep.tolist()) == {2, 1, 3, 0}
>>> if 'torchvision' in available_nms_impls():
>>>     keep = non_max_supression(tlbr, scores, thresh, impl='torchvision')  #␣
↪note torchvision has no bias
>>>     assert set(kwarray.ArrayAPI.tolist(keep)) == {2, 1, 3, 0}
```

Example

```
>>> import ubelt as ub
>>> tlbr = np.array([
>>>     [0, 0, 100, 100],
>>>     [100, 100, 10, 10],
>>>     [10, 10, 100, 100],
>>>     [50, 50, 100, 100],
>>>     [100, 100, 150, 101],
>>>     [120, 100, 180, 101],
```

(continues on next page)

(continued from previous page)

```
>>>     [150, 100, 200, 101],
>>> ], dtype=np.float32)
>>> scores = np.linspace(0, 1, len(tlbr))
>>> thresh = .2
>>> solutions = {}
>>> if not _impls._funcs:
>>>     _impls._lazy_init()
>>> for impl in _impls._funcs:
>>>     keep = non_max_supression(tlbr, scores, thresh, impl=impl)
>>>     solutions[impl] = sorted(keep)
>>> assert 'numpy' in solutions
>>> print('solutions = {}'.format(ub.repr2(solutions, nl=1)))
>>> assert ub.allsame(solutions.values())
```

CommandLine: xdoctest -m ~/code/kwimage/kwimage/algo/algo_nms.py non_max_supression

Example

```
>>> import ubelt as ub
>>> # Check that zero-area boxes are ok
>>> tlbr = np.array([
>>>     [0, 0, 0, 0],
>>>     [0, 0, 0, 0],
>>>     [10, 10, 10, 10],
>>> ], dtype=np.float32)
>>> scores = np.array([1, 2, 3], dtype=np.float32)
>>> thresh = .2
>>> solutions = {}
>>> if not _impls._funcs:
>>>     _impls._lazy_init()
>>> for impl in _impls._funcs:
>>>     keep = non_max_supression(tlbr, scores, thresh, impl=impl)
>>>     solutions[impl] = sorted(keep)
>>> assert 'numpy' in solutions
>>> print('solutions = {}'.format(ub.repr2(solutions, nl=1)))
>>> assert ub.allsame(solutions.values())
```

kwimage.structs

mkinit ~/code/kwimage/kwimage/structs/__init__.py -w –relative –nomod

A common thread in many kwimage.structs / kwannot objects is that they attempt to store multiple data elements using a single data structure when possible e.g. the classes are Boxes, Points, Detections, Coords, and not Box, Detection, Coord. The exceptions are Polygon, Heatmap, and Mask, where it made more sense to have one object-per item because each individual item is a reasonably sized chuck of data.

Another commonality is that objects have only two main attributes: .data and .meta. These allow the underlying representation of the object to vary as needed.

Currently Boxes and Mask do not have a .meta attribute. They instead have a .format attribute which is a text-code indicating the underlying layout of the data.

The *data* and *meta* instance attributes in the Points, Detections, and Heatmaps classes are dictionaries. These classes also have a *__datakeys__* and *__metakeys__* class attribute, which are lists of strings. These lists specify

which keys are expected in each dictionary. For instance, *Points.__datakeys__ = ['xy', 'class_idxs', 'visible']* and *Points.__metakeys__ = ['classes']*. All objects in the data dictionary are expected to be aligned, whereas the meta dictionary is for auxillay data. For example in Points, the xy position data['xy'][i] is expected to have the class index data['class_idxs'][i]. By convention, a class index indexes into the list of category names stored in *meta['classes']*.

The Heatmap.data behaves slighly different than Points. Its *data* dictionary stores different per-pixel attributes like class probability scores, or offset vectors. The *meta* dictionary stores data like the originaly image dimensions (heatmaps are usually downsampled wrt the image that they correspond to) and the transformation matrices would warp the "data" space back onto the original image space.

Note that the developer can add any extra data or meta keys that they like, but they should keep in mind that all items in *data* should be aligned, whereas *meta* can contain arbitrary information.

Subpackages

kwimage.structs._boxes_backend

kwimage.structs._mask_backend

Submodules

kwimage.structs._generic

Module Contents

class kwimage.structs._generic.**Spatial**
Bases: ubelt.NiceRepr

Abstract base class defining the spatial annotation API

class kwimage.structs._generic.**ObjectList**(*data*, *meta=None*)
Bases: *kwimage.structs._generic.Spatial*

Stores a list of potentially heterogenous structures, each item usually corresponds to a different object.

shape

dtype

__len__(*self*)

__nice__(*self*)

__getitem__(*self*, *index*)

__iter__(*self*)

translate(*self*, *offset*, *output_dims=None*, *inplace=False*)

scale(*self*, *factor*, *output_dims=None*, *inplace=False*)

warp(*self*, *transform*, *input_dims=None*, *output_dims=None*, *inplace=False*)

apply(*self*, *func*)

to_coco(*self*, *style='orig'*)

compress(*self*, *flags*, *axis=0*)

take(*self*, *indices*, *axis=0*)

draw(*self*, ***kwargs*)

draw_on(*self*, *image*, ***kwargs*)

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

numpy(*self*)

classmethod concatenate(*cls*, *items*, *axis=0*)

Parameters

- **items** (*Sequence[ObjectList]*) – multiple object lists of the same type
- **axis** (*int | None*) – unused, always implied to be axis 0

Returns combined object list

Return type *ObjectList*

Example

```
>>> import kwimage
>>> cls = kwimage.MaskList
>>> sub_cls = kwimage.Mask
>>> item1 = cls([sub_cls.random(), sub_cls.random()])
>>> item2 = cls([sub_cls.random()])
>>> items = [item1, item2]
>>> new = cls.concatenate(items)
>>> assert len(new) == 3
```

is_tensor(*cls*)

is_numpy(*cls*)

classmethod random(*cls*)

kwimage.structs._generic.**_consistent_dtype_fixer**(*data*)
helper for ensuring out.dtype == in.dtype

kwimage.structs._generic.**_safe_take**(*data*, *indices*, *axis*)

kwimage.structs._generic.**_safe_compress**(*data*, *flags*, *axis*)

kwimage.structs._generic.**_issubclass2**(*child*, *parent*)
Uses string comparisons to avoid ipython reload errors. Much less robust though.

kwimage.structs._generic.**_isinstance2**(*obj*, *cls*)
Uses string comparisons to avoid ipython reload errors. Much less robust though.

Example

import kwimage from kwimage.structs import _generic cls = kwimage.structs._generic.ObjectList obj = kwimage.MaskList([]) _generic._isinstance2(obj, cls)

_generic._isinstance2(kwimage.MaskList([]), _generic.ObjectList)

dets = kwimage.Detections(boxes=kwimage.Boxes.random(3).numpy(), class_idxs=[0, 1, 1], segmentations=kwimage.MaskList([None] * 3)

)

kwimage.structs.boxes

Vectorized Bounding Boxes

kwimage.Boxes is a tool for efficiently transporting a set of bounding boxes within python as well as methods for operating on bounding boxes. It is a VERY thin wrapper around a pure numpy/torch array/tensor representation, and thus it is very fast.

Raw bounding boxes come in lots of different formats. There are lots of ways to parameterize two points! Because of this THE USER MUST ALWAYS BE EXPLICIT ABOUT THE BOX FORMAT.

There are 3 main bounding box formats: xywh: top left xy-coordinates and width height offsets cxywh: center xy-coordinates and width height offsets tlbr: top left and bottom right xy coordinates

Here is some example usage

Example

```
>>> from kwimage.structs.boxes import Boxes
>>> data = np.array([[ 0,  0, 10, 10],
>>>                  [ 5,  5, 50, 50],
>>>                  [10,  0, 20, 10],
>>>                  [20,  0, 30, 10]])
>>> # Note that the format of raw data is ambiguous, so you must specify
>>> boxes = Boxes(data, 'tlbr')
>>> print('boxes = {!r}'.format(boxes))
boxes = <Boxes(tlbr,
    array([[ 0,  0, 10, 10],
           [ 5,  5, 50, 50],
           [10,  0, 20, 10],
           [20,  0, 30, 10]]))>
```

```
>>> # Now you can operate on those boxes easily
>>> print(boxes.translate((10, 10)))
<Boxes(tlbr,
    array([[10., 10., 20., 20.],
           [15., 15., 60., 60.],
           [20., 10., 30., 20.],
           [30., 10., 40., 20.]]))>
>>> print(boxes.to_cxywh())
<Boxes(cxywh,
    array([[ 5. ,  5. , 10. , 10. ],
           [27.5, 27.5, 45. , 45. ],
           [15. ,  5. , 10. , 10. ],
           [25. ,  5. , 10. , 10. ]]))>
>>> print(ub.repr2(boxes.ious(boxes), precision=2, with_dtype=False))
np.array([[1.  , 0.01, 0.  , 0.  ],
          [0.01, 1.  , 0.02, 0.02],
          [0.  , 0.02, 1.  , 0.  ],
          [0.  , 0.02, 0.  , 1.  ]])
```

Module Contents

class kwimage.structs.boxes.**Boxes**(*data*, *format=None*, *check=True*)

Bases: kwimage.structs.boxes._BoxConversionMixins, kwimage.structs.boxes.

_BoxPropertyMixins, kwimage.structs.boxes._BoxTransformMixins, kwimage.structs.boxes._BoxDrawMixins, ubelt.NiceRepr

Converts boxes between different formats as long as the last dimension contains 4 coordinates and the format is specified.

This is a convinience class, and should not not store the data for very long. The general idiom should be create class, convert data, and then get the raw data and let the class be garbage collected. This will help ensure that your code is portable and understandable if this class is not available.

Example

```
>>> # xdoctest: +IGNORE_WHITESPACE
>>> Boxes([25, 30, 15, 10], 'xywh')
<Boxes(xywh, array([25, 30, 15, 10]))>
>>> Boxes([25, 30, 15, 10], 'xywh').to_xywh()
<Boxes(xywh, array([25, 30, 15, 10]))>
>>> Boxes([25, 30, 15, 10], 'xywh').to_cxywh()
<Boxes(cxywh, array([32.5, 35. , 15. , 10. ]))>
>>> Boxes([25, 30, 15, 10], 'xywh').to_tlbr()
<Boxes(tlbr, array([25, 30, 40, 40]))>
>>> Boxes([25, 30, 15, 10], 'xywh').scale(2).to_tlbr()
<Boxes(tlbr, array([50., 60., 80., 80.]))>
>>> Boxes(torch.FloatTensor([[25, 30, 15, 20]]), 'xywh').scale(.1).to_tlbr()
<Boxes(tlbr, tensor([[ 2.5000,  3.0000,  4.0000,  5.0000]]))>
```

Example

```
>>> datas = [
>>>     [1, 2, 3, 4],
>>>     [[1, 2, 3, 4], [4, 5, 6, 7]],
>>>     [[[1, 2, 3, 4], [4, 5, 6, 7]]],
>>> ]
>>> formats = BoxFormat.cannonical
>>> for format1 in formats:
>>>     for data in datas:
>>>         self = box1 = Boxes(data, format1)
>>>         for format2 in formats:
>>>             box2 = box1.toformat(format2)
>>>             back = box2.toformat(format1)
>>>             assert box1 == back
```

device
If the backend is torch returns the data device, otherwise None

__getitem__(*self*, *index*)

__eq__(*self*, *other*)
Tests equality of two Boxes objects

Example

```
>>> box0 = box1 = Boxes([[1, 2, 3, 4]], 'xywh')
>>> box2 = Boxes(box0.data, 'tlbr')
>>> box3 = Boxes([[0, 2, 3, 4]], box0.format)
>>> box4 = Boxes(box0.data, box2.format)
>>> assert box0 == box1
>>> assert not box0 == box2
>>> assert not box2 == box3
>>> assert box2 == box4
```

__len__(*self*)

__nice__(*self*)

__repr__(*self*)

classmethod random(*Boxes*, *num=1*, *scale=1.0*, *format=BoxFormat.XYWH*, *anchors=None*, *anchor_std=1.0 / 6*, *tensor=False*, *rng=None*)

Makes random boxes; typically for testing purposes

Parameters

- **num** (*int*) – number of boxes to generate
- **scale** (*float | Tuple[float, float]*) – size of imgdims
- **format** (*str*) – format of boxes to be created (e.g. tlbr, xywh)
- **anchors** (*ndarray*) – normalized width / heights of anchor boxes to perterb and randomly place. (must be in range 0-1)
- **anchor_std** (*float*) – magnitude of noise applied to anchor shapes
- **tensor** (*bool*) – if True, returns boxes in tensor format
- **rng** (*None | int | RandomState*) – initial random seed

Example

```
>>> # xdoctest: +IGNORE_WHITESPACE
>>> Boxes.random(3, rng=0, scale=100)
<Boxes(xywh,
    array([[54, 54,  6, 17],
           [42, 64,  1, 25],
           [79, 38, 17, 14]]))>
>>> Boxes.random(3, rng=0, scale=100).tensor()
<Boxes(xywh,
    tensor([[ 54,  54,   6,  17],
            [ 42,  64,   1,  25],
            [ 79,  38,  17,  14]]))>
>>> anchors = np.array([[.5, .5], [.3, .3]])
>>> Boxes.random(3, rng=0, scale=100, anchors=anchors)
<Boxes(xywh,
    array([[ 2, 13, 51, 51],
           [32, 51, 32, 36],
           [36, 28, 23, 26]]))>
```

Example

```
>>> # Boxes position/shape within 0-1 space should be uniform.
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> fig = kwplot.figure(fnum=1, doclf=True)
>>> fig.gca().set_xlim(0, 128)
>>> fig.gca().set_ylim(0, 128)
>>> import kwimage
>>> kwimage.Boxes.random(num=10).scale(128).draw()
```

copy (*self*)

classmethod concatenate (*cls*, *boxes*, *axis=0*)

Concatenates multiple boxes together

Parameters

- **boxes** (*Sequence[Boxes]*) – list of boxes to concatenate
- **axis** (*int, default=0*) – axis to stack on

Returns stacked boxes

Return type *Boxes*

Example

```
>>> boxes = [Boxes.random(3) for _ in range(3)]
>>> new = Boxes.concatenate(boxes)
>>> assert len(new) == 9
>>> assert np.all(new.data[3:6] == boxes[1].data)
```

Example

```
>>> boxes = [Boxes.random(3) for _ in range(3)]
>>> boxes[0].data = boxes[0].data[0]
>>> boxes[1].data = boxes[0].data[0:0]
>>> new = Boxes.concatenate(boxes)
>>> assert len(new) == 4
>>> new = Boxes.concatenate([b.tensor() for b in boxes])
>>> assert len(new) == 4
```

compress (*self*, *flags*, *axis=0*, *inplace=False*)

Filters boxes based on a boolean criterion

Parameters

- **flags** (*ArrayLike[bool]*) – true for items to be kept
- **axis** (*int*) – you usually want this to be 0
- **inplace** (*bool*) – if True, modifies this object

Example

```
>>> self = Boxes([[25, 30, 15, 10]], 'tlbr')
>>> self.compress([True])
<Boxes(tlbr, array([[25, 30, 15, 10]]))>
>>> self.compress([False])
<Boxes(tlbr, array([], shape=(0, 4), dtype=int64))>
```

take(*self*, *idxs*, *axis=0*, *inplace=False*)
Takes a subset of items at specific indices

Parameters

- **indices** (*ArrayLike[int]*) – indexes of items to take
- **axis** (*int*) – you usually want this to be 0
- **inplace** (*bool*) – if True, modifies this object

Example

```
>>> self = Boxes([[25, 30, 15, 10]], 'tlbr')
>>> self.take([0])
<Boxes(tlbr, array([[25, 30, 15, 10]]))>
>>> self.take([])
<Boxes(tlbr, array([], shape=(0, 4), dtype=int64))>
```

is_tensor(*self*)
is the backend fueled by torch?

is_numpy(*self*)
is the backend fueled by numpy?

_impl(*self*)
returns the kwarray.ArrayAPI implementation for the data

Example

```
>>> assert Boxes.random().numpy()._impl.is_numpy
>>> assert Boxes.random().tensor()._impl.is_tensor
```

astype(*self*, *dtype*)
Changes the type of the internal array used to represent the boxes

Notes

this operation is not inplace

Example

```
>>> # xdoctest: +IGNORE_WHITESPACE
>>> Boxes.random(3, 100, rng=0).tensor().astype('int32')
<Boxes(xywh,
```

(continues on next page)

(continued from previous page)

```
    tensor([[54, 54,  6, 17],
            [42, 64,  1, 25],
            [79, 38, 17, 14]], dtype=torch.int32))>
>>> Boxes.random(3, 100, rng=0).numpy().astype('int32')
<Boxes(xywh,
    array([[54, 54,  6, 17],
           [42, 64,  1, 25],
           [79, 38, 17, 14]], dtype=int32))>
>>> Boxes.random(3, 100, rng=0).tensor().astype('float32')
>>> Boxes.random(3, 100, rng=0).numpy().astype('float32')
```

numpy (*self*)

Converts tensors to numpy. Does not change memory if possible.

Example

```
>>> self = Boxes.random(3).tensor()
>>> newself = self.numpy()
>>> self.data[0, 0] = 0
>>> assert newself.data[0, 0] == 0
>>> self.data[0, 0] = 1
>>> assert self.data[0, 0] == 1
```

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

Converts numpy to tensors. Does not change memory if possible.

Example

```
>>> self = Boxes.random(3)
>>> newself = self.tensor()
>>> self.data[0, 0] = 0
>>> assert newself.data[0, 0] == 0
>>> self.data[0, 0] = 1
>>> assert self.data[0, 0] == 1
```

ious (*self*, *other*, *bias=0*, *impl='auto'*, *mode=None*)

Compute IOUs (intersection area over union area) between these boxes and another set of boxes.

Parameters

- **other** (*Boxes*) – boxes to compare IoUs against
- **bias** (*int, default=0*) – either 0 or 1, does TL=BR have area of 0 or 1?
- **impl** (*str, default='auto'*) – code to specify implementation used to ious. Can be either torch, py, c, or auto. Efficiency and the exact result will vary by implementation, but they will always be close. Some implementations only accept certain data types (e.g. impl='c', only accepts float32 numpy arrays). See ~/code/kwimage/dev/bench_bbox.py for benchmark details. On my system the torch impl was fastest (when the data was on the GPU).
- **mode** – depricated, use impl

Examples

```
>>> self = Boxes(np.array([[ 0,  0, 10, 10],
>>>                        [10,  0, 20, 10],
>>>                        [20,  0, 30, 10]]), 'tlbr')
>>> other = Boxes(np.array([6, 2, 20, 10]), 'tlbr')
>>> overlaps = self.ious(other, bias=1).round(2)
>>> assert np.all(np.isclose(overlaps, [0.21, 0.63, 0.04])), repr(overlaps)
```

Examples

```
>>> # xdoctest: +IGNORE_WHITESPACE
>>> Boxes(np.empty(0), 'xywh').ious(Boxes(np.empty(4), 'xywh')).shape
(0,)
>>> #Boxes(np.empty(4), 'xywh').ious(Boxes(np.empty(0), 'xywh')).shape
>>> Boxes(np.empty((0, 4)), 'xywh').ious(Boxes(np.empty((0, 4)), 'xywh')).
↪shape
(0, 0)
>>> Boxes(np.empty((1, 4)), 'xywh').ious(Boxes(np.empty((0, 4)), 'xywh')).
↪shape
(1, 0)
>>> Boxes(np.empty((0, 4)), 'xywh').ious(Boxes(np.empty((1, 4)), 'xywh')).
↪shape
(0, 1)
```

Examples

```
>>> formats = BoxFormat.cannonical
>>> istensors = [False, True]
>>> results = {}
>>> for format in formats:
>>>     for tensor in istensors:
>>>         boxes1 = Boxes.random(5, scale=10.0, rng=0, format=format,␣
↪tensor=tensor)
>>>         boxes2 = Boxes.random(7, scale=10.0, rng=1, format=format,␣
↪tensor=tensor)
>>>         ious = boxes1.ious(boxes2)
>>>         results[(format, tensor)] = ious
>>> results = {k: v.numpy() if torch.is_tensor(v) else v for k, v in results.
↪items() }
>>> results = {k: v.tolist() for k, v in results.items()}
>>> print(ub.repr2(results, sk=True, precision=3, nl=2))
>>> from functools import partial
>>> assert ub.allsame(results.values(), partial(np.allclose, atol=1e-07))
```

isect_area (*self*, *other*, *bias=0*)

Intersection part of intersection over union computation

Examples

```
>>> # xdoctest: +IGNORE_WHITESPACE
>>> self = Boxes.random(5, scale=10.0, rng=0, format='tlbr')
```

(continues on next page)

(continued from previous page)

```
>>> other = Boxes.random(3, scale=10.0, rng=1, format='tlbr')
>>> isect = self.isect_area(other, bias=0)
>>> ious_v1 = isect / ((self.area + other.area.T) - isect)
>>> ious_v2 = self.ious(other, bias=0)
>>> assert np.allclose(ious_v1, ious_v2)
```

intersection(*self*, *other*)

Pairwise intersection between two sets of Boxes

Returns intersected boxes

Return type *Boxes*

Examples

```
>>> # xdoctest: +IGNORE_WHITESPACE
>>> from kwimage.structs.boxes import *  # NOQA
>>> self = Boxes.random(5, rng=0).scale(10.)
>>> other = self.translate(1)
>>> new = self.intersection(other)
>>> new_area = np.nan_to_num(new.area).ravel()
>>> alt_area = np.diag(self.isect_area(other))
>>> close = np.isclose(new_area, alt_area)
>>> assert np.all(close)
```

view(*self*, **shape*)

Passthrough method to view or reshape

Example

```
>>> self = Boxes.random(6, scale=10.0, rng=0, format='xywh').tensor()
>>> assert list(self.view(3, 2, 4).data.shape) == [3, 2, 4]
>>> self = Boxes.random(6, scale=10.0, rng=0, format='tlbr').tensor()
>>> assert list(self.view(3, 2, 4).data.shape) == [3, 2, 4]
```

kwimage.structs.coords

Coordinates the fundamental "point" datatype. They do not contain metadata, only geometry. See the *Points* data type for a structure that maintains metadata on top of coordinate data.

Module Contents

kwimage.structs.coords.**_HAS_IMGAUG_FLIP_BUG**

class kwimage.structs.coords.**Coords**(*data=None*, *meta=None*)

Bases: *kwimage.structs._generic.Spatial*, ubelt.NiceRepr

This stores arbitrary sparse n-dimensional coordinate geometry.

You can specify data, but you don't have to. We dont care what it is, we just warp it.

Note: This class was designed to hold coordinates in r/c format, but in general this class is anostic to dimension ordering as long as you are consistent. However, there are two places where this matters:

(1) drawing and (2) gdal/imgaug-warping. In these places we will assume x/y for legacy reasons. This may change in the future.

CommandLine: xdoctest -m kwimage.structs.coords Coords

Example

```
>>> from kwimage.structs.coords import *  # NOQA
>>> import kwarray
>>> rng = kwarray.ensure_rng(0)
>>> self = Coords.random(num=4, dim=3, rng=rng)
>>> matrix = rng.rand(4, 4)
>>> self.warp(matrix)
>>> self.translate(3, inplace=True)
>>> self.translate(3, inplace=True)
>>> self.scale(2)
>>> self.tensor()
>>> # self.tensor(device=0)
>>> self.tensor().tensor().numpy().numpy()
>>> self.numpy()
>>> #self.draw_on()
```

__repr__

dtype

dim

shape

device
If the backend is torch returns the data device, otherwise None

_impl
Returns the internal tensor/numpy ArrayAPI implementation

__nice__(*self*)

__len__(*self*)

copy(*self*)

classmethod random(*Coords*, *num=1*, *dim=2*, *rng=None*, *meta=None*)
Makes random coordinates; typically for testing purposes

is_numpy(*self*)

is_tensor(*self*)

compress(*self*, *flags*, *axis=0*, *inplace=False*)
Filters items based on a boolean criterion

Parameters

- **flags** (*ArrayLike[bool]*) – true for items to be kept
- **axis** (*int*) – you usually want this to be 0

- **inplace** (*bool, default=False*) – if True, modifies this object

Returns filtered coords

Return type *Coords*

Example

```
>>> from kwimage.structs.coords import *  # NOQA
>>> self = Coords.random(10, rng=0)
>>> self.compress([True] * len(self))
>>> self.compress([False] * len(self))
<Coords(data=array([], shape=(0, 2), dtype=float64))>
>>> self = self.tensor()
>>> self.compress([True] * len(self))
>>> self.compress([False] * len(self))
```

take(*self*, *indices*, *axis=0*, *inplace=False*)

Takes a subset of items at specific indices

Parameters

- **indices** (*ArrayLike[int]*) – indexes of items to take
- **axis** (*int*) – you usually want this to be 0
- **inplace** (*bool, default=False*) – if True, modifies this object

Returns filtered coords

Return type *Coords*

Example

```
>>> self = Coords(np.array([[25, 30, 15, 10]]))
>>> self.take([0])
<Coords(data=array([[25, 30, 15, 10]]))>
>>> self.take([])
<Coords(data=array([], shape=(0, 4), dtype=int64))>
```

astype(*self*, *dtype*, *inplace=False*)

Changes the data type

Parameters

- **dtype** – new type
- **inplace** (*bool, default=False*) – if True, modifies this object

view(*self*, **shape*)

Passthrough method to view or reshape

Parameters ***shape** – new shape of the data

Example

```
>>> self = Coords.random(6, dim=4).tensor()
>>> assert list(self.view(3, 2, 4).data.shape) == [3, 2, 4]
>>> self = Coords.random(6, dim=4).numpy()
>>> assert list(self.view(3, 2, 4).data.shape) == [3, 2, 4]
```

classmethod concatenate(*cls*, *coords*, *axis=0*)

Concatenates lists of coordinates together

Parameters

- **coords** (*Sequence[Coords]*) – list of coords to concatenate
- **axis** (*int, default=0*) – axis to stack on

Returns stacked coords

Return type *Coords*

CommandLine: xdoctest -m kwimage.structs.coords Coords.concatenate

Example

```
>>> coords = [Coords.random(3) for _ in range(3)]
>>> new = Coords.concatenate(coords)
>>> assert len(new) == 9
>>> assert np.all(new.data[3:6] == coords[1].data)
```

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

Converts numpy to tensors. Does not change memory if possible.

Example

```
>>> self = Coords.random(3).numpy()
>>> newself = self.tensor()
>>> self.data[0, 0] = 0
>>> assert newself.data[0, 0] == 0
>>> self.data[0, 0] = 1
>>> assert self.data[0, 0] == 1
```

numpy(*self*)

Converts tensors to numpy. Does not change memory if possible.

Example

```
>>> self = Coords.random(3).tensor()
>>> newself = self.numpy()
>>> self.data[0, 0] = 0
>>> assert newself.data[0, 0] == 0
>>> self.data[0, 0] = 1
>>> assert self.data[0, 0] == 1
```

warp(*self*, *transform*, *input_dims=None*, *output_dims=None*, *inplace=False*)

Generalized coordinate transform.

Parameters

- **transform** (*GeometricTransform | ArrayLike | Augmenter*) – scikit-image tranform, a transformation matrix, or an imgaug Augmenter.
- **input_dims** (*Tuple*) – shape of the image these objects correspond to (only needed / used when transform is an imgaug augmenter)
- **output_dims** (*Tuple*) – unused in non-raster structures, only exists for compatibility.
- **inplace** (*bool, default=False*) – if True, modifies data inplace

Notes

Let D = self.dims

transformation matrices can be either:

- (D + 1) x (D + 1) # for homog
- D x D # for scale / rotate
- D x (D + 1) # for affine

Example

```
>>> from kwimage.structs.coords import *  # NOQA
>>> self = Coords.random(10, rng=0)
>>> transform = skimage.transform.AffineTransform(scale=(2, 2))
>>> new = self.warp(transform)
>>> assert np.all(new.data == self.scale(2).data)
```

Doctest:

```
>>> self = Coords.random(10, rng=0)
>>> assert np.all(self.warp(np.eye(3)).data == self.data)
>>> assert np.all(self.warp(np.eye(2)).data == self.data)
```

Ignore:

```
>>> # xdoctest: +SKIP
>>> # xdoctest: +REQUIRES(module:osr)
>>> wgs84_crs = osr.SpatialReference()
>>> wgs84_crs.ImportFromEPSG(4326)
>>> transform = osr.CoordinateTransformation(wgs84_crs, wgs84_crs)
>>> self = Coords.random(10, rng=0)
>>> new = self.warp(transform)
>>> assert np.all(new.data == self.data)
```

_warp_imgaug (*self*, *augmenter*, *input_dims*, *inplace=False*)

Warps by applying an augmenter from the imgaug library

Note: We are assuming you are using X/Y coordinates here.

Parameters

- **augmenter** (*imgaug.augmenters.Augmenter*)

- **input_dims** (*Tuple*) – h/w of the input image
- **inplace** (*bool, default=False*) – if True, modifies data inplace

CommandLine: xdoctest -m ~/code/kwimage/kwimage/structs/coords.py Coords._warp_imgaug

Example

```
>>> # xdoctest: +REQUIRES(module:imgaug)
>>> from kwimage.structs.coords import *  # NOQA
>>> import imgaug
>>> input_dims = (10, 10)
>>> self = Coords.random(10).scale(input_dims)
>>> augmenter = imgaug.augmenters.Fliplr(p=1)
>>> new = self._warp_imgaug(augmenter, input_dims)
>>> # y coordinate should not change
>>> assert np.allclose(self.data[:, 1], new.data[:, 1])
>>> assert np.allclose(input_dims[0] - self.data[:, 0], new.data[:, 0])
```

```
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.figure(fnum=1, doclf=True)
>>> from matplotlib import pyplot as pl
>>> ax = plt.gca()
>>> ax.set_xlim(0, input_dims[0])
>>> ax.set_ylim(0, input_dims[1])
>>> self.draw(color='red', alpha=.4, radius=0.1)
>>> new.draw(color='blue', alpha=.4, radius=0.1)
```

Example

```
>>> # xdoctest: +REQUIRES(module:imgaug)
>>> from kwimage.structs.coords import *  # NOQA
>>> import imgaug
>>> input_dims = (32, 32)
>>> inplace = 0
>>> self = Coords.random(1000, rng=142).scale(input_dims).scale(.8)
>>> self.data = self.data.astype(np.int32).astype(np.float32)
>>> augmenter = imgaug.augmenters.CropAndPad(px=(-4, 4), keep_size=1).to_
↪deterministic()
>>> new = self._warp_imgaug(augmenter, input_dims)
>>> # Change should be linear
>>> norm1 = (self.data - self.data.min(axis=0)) / (self.data.max(axis=0) -␣
↪self.data.min(axis=0))
>>> norm2 = (new.data - new.data.min(axis=0)) / (new.data.max(axis=0) - new.
↪data.min(axis=0))
>>> diff = norm1 - norm2
>>> assert np.allclose(diff, 0, atol=1e-6, rtol=1e-4)
>>> #assert np.allclose(self.data[:, 1], new.data[:, 1])
>>> #assert np.allclose(input_dims[0] - self.data[:, 0], new.data[:, 0])
>>> # xdoc: +REQUIRES(--show)
>>> import kwimage
>>> im = kwimage.imresize(kwimage.grab_test_image(), dsize=input_dims[::-1])
```

(continues on next page)

(continued from previous page)

```
>>> new_im = augmenter.augment_image(im)
>>> import kwplot
>>> plt = kwplot.autoplt()
>>> kwplot.figure(fnum=1, doclf=True)
>>> kwplot.imshow(im, pnum=(1, 2, 1), fnum=1)
>>> self.draw(color='red', alpha=.8, radius=0.5)
>>> kwplot.imshow(new_im, pnum=(1, 2, 2), fnum=1)
>>> new.draw(color='blue', alpha=.8, radius=0.5, coord_axes=[1, 0])
```

to_imgaug(*self*, *input_dims*)

Example

```
>>> # xdoctest: +REQUIRES(module:imgaug)
>>> from kwimage.structs.coords import *  # NOQA
>>> self = Coords.random(10)
>>> input_dims = (10, 10)
>>> kpoi = self.to_imgaug(input_dims)
>>> new = Coords.from_imgaug(kpoi)
>>> assert np.allclose(new.data, self.data)
```

classmethod from_imgaug(*cls*, *kpoi*)

scale(*self*, *factor*, *output_dims=None*, *inplace=False*)

Scale coordinates by a factor

Parameters

- **factor** (*float or Tuple[float, float]*) – scale factor as either a scalar or per-dimension tuple.
- **output_dims** (*Tuple*) – unused in non-raster spatial structures

Example

```
>>> from kwimage.structs.coords import *  # NOQA
>>> self = Coords.random(10, rng=0)
>>> new = self.scale(10)
>>> assert new.data.max() <= 10
```

```
>>> self = Coords.random(10, rng=0)
>>> self.data = (self.data * 10).astype(np.int)
>>> new = self.scale(10)
>>> assert new.data.dtype.kind == 'i'
>>> new = self.scale(10.0)
>>> assert new.data.dtype.kind == 'f'
```

translate(*self*, *offset*, *output_dims=None*, *inplace=False*)

Shift the coordinates

Parameters

- **offset** (*float or Tuple[float]*) – transation offset as either a scalar or a per-dimension tuple.
- **output_dims** (*Tuple*) – unused in non-raster spatial structures

Example

```
>>> from kwimage.structs.coords import *  # NOQA
>>> self = Coords.random(10, dim=3, rng=0)
>>> new = self.translate(10)
>>> assert new.data.min() >= 10
>>> assert new.data.max() <= 11
>>> Coords.random(3, dim=3, rng=0)
>>> Coords.random(3, dim=3, rng=0).translate((1, 2, 3))
```

fill (*self*, *image*, *value*, *coord_axes=None*, *interp='bilinear'*)
Sets sub-coordinate locations in a grid to a particular value

Parameters **coord_axes** (*Tuple*) – specify which image axes each coordinate dim corresponds to. For 2D images, if you are storing r/c data, set to [0,1], if you are storing x/y data, set to [1,0].

draw_on (*self, image=None, fill_value=1, coord_axes=[1, 0], interp='bilinear'*)

Note: unlike other methods, the defaults assume x/y internal data

---

Parameters **coord_axes** (*Tuple*) – specify which image axes each coordinate dim corresponds to. For 2D images, if you are storing r/c data, set to [0,1], if you are storing x/y data, set to [1,0].

In other words the i-th entry in coord_axes specifies which row-major spatial dimension the i-th column of a coordinate corresponds to. The index is the coordinate dimension and the value is the axes dimension.

Example

```
>>> # xdoc: +REQUIRES(module:kwplot)
>>> from kwimage.structs.coords import *  # NOQA
>>> s = 256
>>> self = Coords.random(10, meta={'shape': (s, s)}).scale(s)
>>> self.data[0] = [10, 10]
>>> self.data[1] = [20, 40]
>>> image = np.zeros((s, s))
>>> fill_value = 1
>>> image = self.draw_on(image, fill_value, coord_axes=[1, 0], interp=
↪'bilinear')
>>> # image = self.draw_on(image, fill_value, coord_axes=[0, 1], interp=
↪'nearest')
>>> # image = self.draw_on(image, fill_value, coord_axes=[1, 0], interp=
↪'bilinear')
>>> # image = self.draw_on(image, fill_value, coord_axes=[1, 0], interp=
↪'nearest')
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.figure(fnum=1, doclf=True)
>>> kwplot.autompl()
>>> kwplot.imshow(image)
>>> self.draw(radius=3, alpha=.5, coord_axes=[1, 0])
```

draw(*self, color='blue', ax=None, alpha=None, coord_axes=[1, 0], radius=1*)

Note: unlike other methods, the defaults assume x/y internal data

Parameters **coord_axes** (*Tuple*) – specify which image axes each coordinate dim corresponds to. For 2D images,

if you are storing r/c data, set to [0,1], if you are storing x/y data, set to [1,0].

Example

```
>>> # xdoc: +REQUIRES(module:kwplot)
>>> from kwimage.structs.coords import *  # NOQA
>>> self = Coords.random(10)
>>> # xdoc: +REQUIRES(--show)
>>> self.draw(radius=3.0)
>>> import kwplot
>>> kwplot.autompl()
>>> self.draw(radius=3.0)
```

kwimage.structs.detections

Structure for efficient access and modification of bounding boxes with associated scores and class labels. Builds on top of the *kwimage.Boxes* structure.

Also can optionally incorporate *kwimage.PolygonList* for segmentation masks and *kwimage.PointsList* for keypoints.

Module Contents

kwimage.structs.detections.**_TORCH_HAS_BOOL_COMP**

class kwimage.structs.detections.**_DetDrawMixin**

Non critical methods for visualizing detections

draw(*self, color='blue', alpha=None, labels=True, centers=False, lw=2, fill=False, ax=None, radius=5, kpts=True, sseg=True, setlim=False, boxes=True*)

Draws boxes using matplotlib

Example

```
>>> # xdoc: +REQUIRES(module:kwplot)
>>> self = Detections.random(num=10, scale=512.0, rng=0, classes=['a', 'b', 'c
↪'])
>>> self.boxes.translate((-128, -128), inplace=True)
>>> image = (np.random.rand(256, 256) * 255).astype(np.uint8)
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> fig = kwplot.figure(fnum=1, doclf=True)
>>> kwplot.imshow(image)
```

(continues on next page)

(continued from previous page)

```
>>> # xdoc: +REQUIRES(--show)
>>> self.draw(color='blue', alpha=None)
>>> # xdoc: +REQUIRES(--show)
>>> for o in fig.findobj():  # http://matplotlib.1069221.n5.nabble.com/How-to-
↪turn-off-all-clipping-td1813.html
>>>     o.set_clip_on(False)
>>> kwplot.show_if_requested()
```

draw_on(*self*, *image*, *color='blue'*, *alpha=None*, *labels=True*, *radius=5*, *kpts=True*, *sseg=True*, *boxes=True*, *ssegkw=None*)

Draws boxes directly on the image using OpenCV

Parameters **image** (*ndarray[uint8]*) – must be in uint8 format

Returns image with labeled boxes drawn on it

Return type ndarray[uint8]

CommandLine: xdoctest -m kwimage.structs.detections _DetDrawMixin.draw_on:1 –profile –show

Example

```
>>> # xdoc: +REQUIRES(module:kwplot)
>>> import kwplot
>>> self = Detections.random(num=10, scale=512, rng=0)
>>> image = (np.random.rand(512, 512) * 255).astype(np.uint8)
>>> image2 = self.draw_on(image, color='blue')
>>> # xdoc: +REQUIRES(--show)
>>> kwplot.figure(fnum=2000, doclf=True)
>>> kwplot.autompl()
>>> kwplot.imshow(image2)
>>> kwplot.show_if_requested()
```

Example

```
>>> # xdoc: +REQUIRES(module:kwplot)
>>> # xdoc: +REQUIRES(--profile)
>>> import kwplot
>>> self = Detections.random(num=100, scale=512, rng=0, keypoints=True,␣
↪segmentations=True)
>>> image = (np.random.rand(512, 512) * 255).astype(np.uint8)
>>> image2 = self.draw_on(image, color='blue')
>>> # xdoc: +REQUIRES(--show)
>>> kwplot.figure(fnum=2000, doclf=True)
>>> kwplot.autompl()
>>> kwplot.imshow(image2)
>>> kwplot.show_if_requested()
```

Ignore: import xdev globals().update(xdev.get_func_kwargs(kwimage.Detections.draw_on))

_make_alpha(*self*, *alpha*)

Either passes through user specified alpha or chooses a sensible default

_make_labels(*self*, *labels*)
Either passes through user specified labels or chooses a sensible default

class kwimage.structs.detections.**_DetAlgoMixin**
Non critical methods for algorithmic manipulation of detections

non_max_supression(*self*, *thresh=0.0*, *perclass=False*, *impl='auto'*, *daq=False*, *device_id=None*)
Find high scoring minimally overlapping detections

Parameters

- **thresh** (*float*) – iou threshold between 0 and 1. A box is removed if it overlaps with a previously chosen box by more than this threshold. Higher values are are more permissive (more boxes are returned). A value of 0 means that returned boxes will have no overlap.
- **perclass** (*bool*) – if True, works on a per-class basis
- **impl** (*str*) – nms implementation to use
- **daq** (*Bool | Dict*) – if False, uses reqgular nms, otherwise uses divide and conquor algorithm. If *daq* is a Dict, then it is used as the kwargs to *kwimage.daq_spatial_nms*
- **device_id** – try not to use. only used if impl is gpu

Returns indices of boxes to keep

Return type ndarray[int]

non_max_supress(*self*, *thresh=0.0*, *perclass=False*, *impl='auto'*, *daq=False*)
Convinience method. Like *non_max_supression*, but returns to supressed boxes instead of the indices to keep.

rasterize(*self*, *bg_size*, *input_dims*, *soften=1*, *tf_data_to_img=None*, *img_dims=None*, *exclude=[]*)
Ambiguous conversion from a Heatmap to a Detections object.

SeeAlso: Heatmap.detect

Returns raster-space detections.

Return type *kwimage.Heatmap*

Example

```
>>> # xdoctest: +REQUIRES(module:ndsampler)
>>> from kwimage.structs.detections import *  # NOQA
>>> self, iminfo, sampler = Detections.demo()
>>> image = iminfo['imdata']
>>> input_dims = iminfo['imdata'].shape[0:2]
>>> bg_size = [100, 100]
>>> heatmap = self.rasterize(bg_size, input_dims)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.figure(fnum=1, pnum=(2, 2, 1))
>>> heatmap.draw(invert=True)
>>> kwplot.figure(fnum=1, pnum=(2, 2, 2))
>>> kwplot.imshow(heatmap.draw_on(image))
>>> kwplot.figure(fnum=1, pnum=(2, 1, 2))
>>> kwplot.imshow(heatmap.draw_stacked())
```

class kwimage.structs.detections.**Detections**(*data=None*, *meta=None*, *datakeys=None*, *metakeys=None*, *checks=True*, ***kwargs*)

Bases: ubelt.NiceRepr, *kwimage.structs.detections._DetAlgoMixin*, *kwimage.structs.detections._DetDrawMixin*

Container for holding and manipulating multiple detections.

Variables

- **data** (*Dict*) – dictionary containing corresponding lists. The length of each list is the number of detections. This contains the bounding boxes, confidence scores, and class indices. Details of the most common keys and types are as follows:

  boxes (kwimage.Boxes[ArrayLike]): multiple bounding boxes scores (ArrayLike): associated scores class_idxs (ArrayLike): associated class indices

  Additional custom keys may be specified as long as (a) the values are array-like and the first axis corresponds to the standard data values and (b) are custom keys are listed in the *datakeys* kwargs when constructing the Detections.

- **meta** (*Dict*) – This contains contextual information about the detections. This includes the class names, which can be indexed into via the class indexes.

Example

```
>>> self = Detections.random(10)
>>> other = Detections(self)
>>> assert other.data == self.data
>>> assert other.data is self.data, 'try not to copy unless necessary'
```

__datakeys__ = ['boxes', 'scores', 'class_idxs', 'probs', 'weights', 'keypoints', 'segm

__metakeys__ = ['classes']

boxes

class_idxs

scores
typically only populated for predicted detections

probs
typically only populated for predicted detections

weights
typically only populated for groundtruth detections

classes

device
If the backend is torch returns the data device, otherwise None

dtype

__nice__(*self*)

__len__(*self*)

copy(*self*)
Returns a deep copy of this Detections object

classmethod from_coco_annots(*cls*, *anns*, *cats=None*, *classes=None*, *kp_classes=None*, *shape=None*, *dset=None*)

Create a Detections object from a list of coco-like annotations.

Parameters

- **anns** (*List[Dict]*) – list of coco-like annotation objects
- **dset** (*CocoDataset*) – if specified, cats, classes, and kp_classes can are ignored.
- **cats** (*List[Dict]*) – coco-format category information. Used only if *dset* is not specified.
- **classes** (*ndsampler.CategoryTree*) – category tree with coco class info. Used only if *dset* is not specified.
- **kp_classes** (*ndsampler.CategoryTree*) – keypoint category tree with coco keypoint class info. Used only if *dset* is not specified.
- **shape** (*tuple*) – shape of parent image

Returns a detections object

Return type *Detections*

Example

```
>>> from kwimage.structs.detections import *  # NOQA
>>> # xdoctest: +REQUIRES(--module:ndsampler)
>>> anns = [{
>>>     'id': 0,
>>>     'image_id': 1,
>>>     'category_id': 2,
>>>     'bbox': [2, 3, 10, 10],
>>>     'keypoints': [4.5, 4.5, 2],
>>>     'segmentation': {
>>>         'counts': '_11a04M2O0O20N101N3L_5',
>>>         'size': [20, 20],
>>>     },
>>> }]
>>> dataset = {
>>>     'images': [],
>>>     'annotations': [],
>>>     'categories': [
>>>         {'id': 0, 'name': 'background'},
>>>         {'id': 2, 'name': 'class1', 'keypoints': ['spot']}
>>>     ]
>>> }
>>> #import ndsampler
>>> #dset = ndsampler.CocoDataset(dataset)
>>> cats = dataset['categories']
>>> dets = Detections.from_coco_annots(anns, cats)
```

Example

```
>>> import kwimage
>>> # xdoctest: +REQUIRES(--module:ndsampler)
>>> import ndsampler
>>> sampler = ndsampler.CocoSampler.demo('photos')
```

(continues on next page)

(continued from previous page)

```
>>> iminfo, anns = sampler.load_image_with_annots(1)
>>> shape = iminfo['imdata'].shape[0:2]
>>> kp_classes = sampler.dset.keypoint_categories()
>>> dets = kwimage.Detections.from_coco_annots(
>>>     anns, sampler.dset.dataset['categories'], sampler.catgraph,
>>>     kp_classes, shape=shape)
```

to_coco(*self*, *cname_to_cat=None*, *style='orig'*)
Converts this set of detections into coco-like annotation dictionaries.

Notes

Not all aspects of the MS-COCO format can be accurately represented, so some liberties are taken. The MS-COCO standard defines that annotations should specifiy a category_id field, but in some cases this information is not available so we will populate a 'category_name' field if possible and in the worst case fall back to 'category_index'.

Additionally, detections may contain additional information beyond the MS-COCO standard, and this information (e.g. weight, prob, score) is added as forign fields.

Parameters

- **cname_to_cat** – currently ignored.
- **style** (*str*) – either orig (for the original coco format) or new for the more general ndsampler-style coco format.

Yields *dict* – coco-like annotation structures

Example

```
>>> # xdoctest: +REQUIRES(module:ndsampler)
>>> from kwimage.structs.detections import *
>>> self = Detections.demo()[0]
>>> cname_to_cat = None
>>> list(self.to_coco())
```

num_boxes(*self*)

warp(*self*, *transform*, *input_dims=None*, *output_dims=None*, *inplace=False*)
Spatially warp the detections.

Example

```
>>> import skimage
>>> transform = skimage.transform.AffineTransform(scale=(2, 3),␣
↪translation=(4, 5))
>>> self = Detections.random(2)
>>> new = self.warp(transform)
>>> assert new.boxes == self.boxes.warp(transform)
>>> assert new != self
```

scale(*self*, *factor*, *output_dims=None*, *inplace=False*)
Spatially warp the detections.

Example

```
>>> import skimage
>>> transform = skimage.transform.AffineTransform(scale=(2, 3),␣
↪translation=(4, 5))
>>> self = Detections.random(2)
>>> new = self.warp(transform)
>>> assert new.boxes == self.boxes.warp(transform)
>>> assert new != self
```

translate (*self*, *offset*, *output_dims=None*, *inplace=False*)

Spatially warp the detections.

Example

```
>>> import skimage
>>> self = Detections.random(2)
>>> new = self.translate(10)
```

classmethod concatenate (*cls*, *dets*)

Parameters **boxes** (*Sequence[Detections]*) – list of detections to concatenate

Returns stacked detections

Return type *Detections*

Example

```
>>> self = Detections.random(2)
>>> other = Detections.random(3)
>>> dets = [self, other]
>>> new = Detections.concatenate(dets)
>>> assert new.num_boxes() == 5
```

```
>>> self = Detections.random(2, segmentations=True)
>>> other = Detections.random(3, segmentations=True)
>>> dets = [self, other]
>>> new = Detections.concatenate(dets)
>>> assert new.num_boxes() == 5
```

argsort (*self*, *reverse=True*)

Sorts detection indices by descending (or ascending) scores

Returns sorted indices

Return type ndarray[int]

sort (*self*, *reverse=True*)

Sorts detections by descending (or ascending) scores

Returns sorted copy of self

Return type *kwimage.structs.Detections*

compress (*self*, *flags*, *axis=0*)

Returns a subset where corresponding locations are True.

Parameters **flags** (*ndarray[bool]*) – mask marking selected items

Returns subset of self

Return type *kwimage.structs.Detections*

CommandLine: xdoctest -m kwimage.structs.detections Detections.compress

Example

```
>>> import kwimage
>>> dets = kwimage.Detections.random(keypoints='dense')
>>> flags = np.random.rand(len(dets)) > 0.5
>>> subset = dets.compress(flags)
>>> assert len(subset) == flags.sum()
>>> subset = dets.tensor().compress(flags)
>>> assert len(subset) == flags.sum()
```

z = dets.tensor().data['keypoints'].data['xy'] z.compress(flags) ub.map_vals(lambda x: x.shape, dets.data) ub.map_vals(lambda x: x.shape, subset.data)

take(*self*, *indices*, *axis=0*)

Returns a subset specified by indices

Parameters **indices** (*ndarray[int]*) – indices to select

Returns subset of self

Return type *kwimage.structs.Detections*

Example

```
>>> import kwimage
>>> dets = kwimage.Detections(boxes=kwimage.Boxes.random(10))
>>> subset = dets.take([2, 3, 5, 7])
>>> assert len(subset) == 4
>>> subset = dets.tensor().take([2, 3, 5, 7])
>>> assert len(subset) == 4
```

__getitem__(*self*, *index*)

Fancy slicing / subset / indexing.

Note: scalar indices are always coerced into index lists of length 1.

Example

```
>>> import kwimage
>>> import kwarray
>>> dets = kwimage.Detections(boxes=kwimage.Boxes.random(10))
>>> indices = [2, 3, 5, 7]
>>> flags = kwarray.boolmask(indices, len(dets))
>>> assert dets[flags].data == dets[indices].data
```

is_tensor(*self*)

is the backend fueled by torch?

is_numpy(*self*)
is the backend fueled by numpy?

numpy(*self*)
Converts tensors to numpy. Does not change memory if possible.

Example

```
>>> self = Detections.random(3).tensor()
>>> newself = self.numpy()
>>> self.scores[0] = 0
>>> assert newself.scores[0] == 0
>>> self.scores[0] = 1
>>> assert self.scores[0] == 1
>>> self.numpy().numpy()
```

tensor(*self*, *device=ub.NoParam*)
Converts numpy to tensors. Does not change memory if possible.

Example

```
>>> from kwimage.structs.detections import *
>>> self = Detections.random(3)
>>> newself = self.tensor()
>>> self.scores[0] = 0
>>> assert newself.scores[0] == 0
>>> self.scores[0] = 1
>>> assert self.scores[0] == 1
>>> self.tensor().tensor()
```

classmethod demo(*Detections*)

classmethod random(*cls*, *num=10*, *scale=1.0*, *rng=None*, *classes=3*, *keypoints=False*, *tensor=False*, *segmentations=False*)
Creates dummy data, suitable for use in tests and benchmarks

Parameters

- **num** (*int*) – number of boxes
- **scale** (*float | tuple, default=1.0*) – bounding image size
- **classes** (*int | Sequence*) – list of class labels or number of classes
- **tensor** (*bool, default=False*) – determines backend
- **rng** (*np.random.RandomState*) – random state

Example

```
>>> import kwimage
>>> dets = kwimage.Detections.random(keypoints='jagged')
>>> dets.data['keypoints'].data[0].data
>>> dets.data['keypoints'].meta
>>> dets = kwimage.Detections.random(keypoints='dense')
>>> dets = kwimage.Detections.random(keypoints='dense', segmentations=True).
↪scale(1000)
```

(continues on next page)

(continued from previous page)

```
>>> # xdoctest:+REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> dets.draw(setlim=True)
```

Example

```
>>> # Boxes position/shape within 0-1 space should be uniform.
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> fig = kwplot.figure(fnum=1, doclf=True)
>>> fig.gca().set_xlim(0, 128)
>>> fig.gca().set_ylim(0, 128)
>>> import kwimage
>>> kwimage.Detections.random(num=10, segmentations=True).scale(128).draw()
```

kwimage.structs.detections.**_dets_to_fcmaps**(*dets*, *bg_size*, *input_dims*, *bg_idx=0*, *pmin=0.6*, *pmax=1.0*, *soft=True*, *exclude=[]*)

Construct semantic segmentation detection targets from annotations in dictionary format.

Rasterize detections.

Parameters

- **dets** (*kwimage.Detections*)
- **bg_size** (*tuple*) – size (W, H) to predict for backgrounds
- **input_dims** (*tuple*) – window H, W

Returns

with keys size : 2D ndarray containing the W,H of the object dxdy : 2D ndarray containing the x,y offset of the object cidx : 2D ndarray containing the class index of the object

Return type dict

Ignore: import xdev globals().update(xdev.get_func_kwargs(_dets_to_fcmaps))

Example

```
>>> # xdoctest: +REQUIRES(module:ndsampler)
>>> from kwimage.structs.detections import *  # NOQA
>>> from kwimage.structs.detections import _dets_to_fcmaps
>>> import kwimage
>>> import ndsampler
>>> sampler = ndsampler.CocoSampler.demo('photos')
>>> iminfo, anns = sampler.load_image_with_annots(1)
>>> image = iminfo['imdata']
>>> input_dims = image.shape[0:2]
>>> kp_classes = sampler.dset.keypoint_categories()
>>> dets = kwimage.Detections.from_coco_annots(
>>>     anns, sampler.dset.dataset['categories'],
>>>     sampler.catgraph, kp_classes, shape=input_dims)
>>> bg_size = [100, 100]
```

(continues on next page)

(continued from previous page)

```
>>> bg_idxs = sampler.catgraph.index('background')
>>> fcn_target = _dets_to_fcmaps(dets, bg_size, input_dims, bg_idxs)
>>> fcn_target.keys()
>>> print('fcn_target: ' + ub.repr2(ub.map_vals(lambda x: x.shape, fcn_target),␣
↪nl=1))
fcn_target: {
    'cidx': (512, 512),
    'class_probs': (10, 512, 512),
    'dxdy': (2, 512, 512),
    'kpts': (2, 7, 512, 512),
    'kpts_ignore': (7, 512, 512),
    'size': (2, 512, 512),
}
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> size_mask = fcn_target['size']
>>> dxdy_mask = fcn_target['dxdy']
>>> cidx_mask = fcn_target['cidx']
>>> kpts_mask = fcn_target['kpts']
>>> def _vizmask(dxdy_mask):
>>>     dx, dy = dxdy_mask
>>>     mag = np.sqrt(dx ** 2 + dy ** 2)
>>>     mag /= (mag.max() + 1e-9)
>>>     mask = (cidx_mask != 0).astype(np.float32)
>>>     angle = np.arctan2(dy, dx)
>>>     orimask = kwplot.make_orimask(angle, mask, alpha=mag)
>>>     vecmask = kwplot.make_vector_field(
>>>         dx, dy, stride=4, scale=0.1, thickness=1, tipLength=.2,
>>>         line_type=16)
>>>     return [vecmask, orimask]
>>> vecmask, orimask = _vizmask(dxdy_mask)
>>> raster = kwimage.overlay_alpha_layers(
>>>     [vecmask, orimask, image], keepalpha=False)
>>> raster = dets.draw_on((raster * 255).astype(np.uint8),
>>>                       labels=True, alpha=None)
>>> kwplot.imshow(raster)
>>> kwplot.show_if_requested()
```

raster = (kwimage.overlay_alpha_layers(_vizmask(kpts_mask[:, 5]) + [image], keepalpha=False) * 255).astype(np.uint8) kwplot.imshow(raster, pnum=(1, 3, 2), fnum=1) raster = (kwimage.overlay_alpha_layers(_vizmask(kpts_mask[:, 6]) + [image], keepalpha=False) * 255).astype(np.uint8) kwplot.imshow(raster, pnum=(1, 3, 3), fnum=1) raster = (kwimage.overlay_alpha_layers(_vizmask(dxdy_mask) + [image], keepalpha=False) * 255).astype(np.uint8) raster = dets.draw_on(raster, labels=True, alpha=None) kwplot.imshow(raster, pnum=(1, 3, 1), fnum=1) raster = kwimage.overlay_alpha_layers(

[vecmask, orimask, image], keepalpha=False)

raster = dets.draw_on((raster * 255).astype(np.uint8), labels=True, alpha=None)

kwplot.imshow(raster) kwplot.show_if_requested()

kwimage.structs.heatmap

Todo:

- [ ] Remove doctest dependency on ndsampler?

CommandLine: xdoctest -m ~/code/kwimage/kwimage/structs/heatmap.py __doc__

Example

```
>>> # xdoctest: +REQUIRES(module:ndsampler)
>>> # xdoctest: +REQUIRES(--mask)
>>> from kwimage.structs.heatmap import *  # NOQA
>>> import kwimage
>>> import ndsampler
>>> sampler = ndsampler.CocoSampler.demo('shapes')
>>> iminfo, anns = sampler.load_image_with_annots(1)
>>> image = iminfo['imdata']
>>> input_dims = image.shape[0:2]
>>> kp_classes = sampler.dset.keypoint_categories()
>>> dets = kwimage.Detections.from_coco_annots(
>>>     anns, sampler.dset.dataset['categories'],
>>>     sampler.catgraph, kp_classes, shape=input_dims)
>>> bg_size = [100, 100]
>>> heatmap = dets.rasterize(bg_size, input_dims, soften=2)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.figure(fnum=1, doclf=True)
>>> kwplot.imshow(image)
>>> heatmap.draw(invert=True, kpts=[0, 1, 2, 3, 4])
```

Example

```
>>> # xdoctest: +REQUIRES(module:ndsampler)
>>> # xdoctest: +REQUIRES(--mask)
>>> from kwimage.structs.heatmap import *  # NOQA
>>> from kwimage.structs.detections import _dets_to_fcmaps
>>> import kwimage
>>> import ndsampler
>>> sampler = ndsampler.CocoSampler.demo('shapes')
>>> iminfo, anns = sampler.load_image_with_annots(1)
>>> image = iminfo['imdata']
>>> input_dims = image.shape[0:2]
>>> kp_classes = sampler.dset.keypoint_categories()
>>> dets = kwimage.Detections.from_coco_annots(
>>>     anns, sampler.dset.dataset['categories'],
>>>     sampler.catgraph, kp_classes, shape=input_dims)
>>> bg_size = [100, 100]
>>> bg_idxs = sampler.catgraph.index('background')
>>> fcn_target = _dets_to_fcmaps(dets, bg_size, input_dims, bg_idxs)
>>> fcn_target.keys()
>>> print('fcn_target: ' + ub.repr2(ub.map_vals(lambda x: x.shape, fcn_target), nl=1))
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
```

(continues on next page)

(continued from previous page)

```
>>> kwplot.autompl()
>>> size_mask = fcn_target['size']
>>> dxdy_mask = fcn_target['dxdy']
>>> cidx_mask = fcn_target['cidx']
>>> kpts_mask = fcn_target['kpts']
>>> def _vizmask(dxdy_mask):
>>>     dx, dy = dxdy_mask
>>>     mag = np.sqrt(dx ** 2 + dy ** 2)
>>>     mag /= (mag.max() + 1e-9)
>>>     mask = (cidx_mask != 0).astype(np.float32)
>>>     angle = np.arctan2(dy, dx)
>>>     orimask = kwplot.make_orimask(angle, mask, alpha=mag)
>>>     vecmask = kwplot.make_vector_field(
>>>         dx, dy, stride=4, scale=0.1, thickness=1, tipLength=.2,
>>>         line_type=16)
>>>     return [vecmask, orimask]
>>> vecmask, orimask = _vizmask(dxdy_mask)
>>> raster = kwimage.overlay_alpha_layers(
>>>     [vecmask, orimask, image], keepalpha=False)
>>> raster = dets.draw_on((raster * 255).astype(np.uint8),
>>>                       labels=True, alpha=None)
>>> kwplot.imshow(raster)
>>> kwplot.show_if_requested()
```

Module Contents

class kwimage.structs.heatmap.**_HeatmapDrawMixin**

Bases: object

mixin methods for drawing heatmap details

_colorize_class_idx(*self*)

colorize(*self*, *channel=None*, *invert=False*, *with_alpha=1.0*, *interpolation='linear'*, *imgspace=False*, *cmap=None*)

Creates a colorized version of a heatmap channel suitable for visualization

Parameters

- **channel** (*int | str*) – index of category to visualize, or a special code indicating how to visualize multiple classes.
- **imgspace** (*bool, default=False*) – colorize the image after warping into the image space.

CommandLine: xdoctest -m ~/code/kwimage/kwimage/structs/heatmap.py _HeatmapDrawMixin.colorize –show

Example

```
>>> # xdoctest: +REQUIRES(module:kwplot)
>>> self = Heatmap.random(rng=0, dims=(32, 32))
>>> colormask1 = self.colorize(0, imgspace=False)
>>> colormask2 = self.colorize(0, imgspace=True)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
```

(continues on next page)

(continued from previous page)

```
>>> kwplot.autompl()
>>> kwplot.imshow(colormask1, pnum=(1, 2, 1), fnum=1, title='output space')
>>> kwplot.imshow(colormask2, pnum=(1, 2, 2), fnum=1, title='image space')
>>> kwplot.show_if_requested()
```

Example

```
>>> # xdoctest: +REQUIRES(module:kwplot)
>>> self = Heatmap.random(rng=0, dims=(32, 32))
>>> colormask1 = self.colorize('diameter', imgspace=False)
>>> colormask2 = self.colorize('diameter', imgspace=True)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(colormask1, pnum=(1, 2, 1), fnum=1, title='output space')
>>> kwplot.imshow(colormask2, pnum=(1, 2, 2), fnum=1, title='image space')
>>> kwplot.show_if_requested()
```

Ignore:

```
>>> # xdoctest: +REQUIRES(module:kwplot)
>>> self = Heatmap.random(rng=0, dims=(32, 32))
>>> self.data['class_energy'] = (self.data['class_probs'] - .5) * 10
>>> colormask1 = self.colorize('class_energy_color', imgspace=False)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(colormask1, fnum=1, title='output space')
>>> kwplot.show_if_requested()
```

draw_stacked(*self*, *image=None*, *dsize=(224*, *224)*, *ignore_class_idxs={}*, *top=None*, *chosen_cxs=None*)

Draws per-class probabilities and stacks them into a single image

Example

```
>>> # xdoctest: +REQUIRES(module:kwplot)
>>> self = Heatmap.random(rng=0, dims=(32, 32))
>>> stacked = self.draw_stacked()
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(stacked)
```

draw(*self*, *channel=None*, *image=None*, *imgspace=None*, ***kwargs*)

Accepts same args as draw_on, but uses maplotlib

Parameters **channel** (*int | str*) – category index to visualize, or special key

draw_on(*self*, *image*, *channel=None*, *invert=False*, *with_alpha=1.0*, *interpolation='linear'*, *vecs=False*, *kpts=None*, *imgspace=None*)

Overlays a heatmap channel on top of an image

Parameters

- **image** (*ndarray*) – image to draw on
- **channel** (*int | str*) – category index to visualize, or special key. special keys are: class_idx, class_probs, class_idx
- **imgspace** (*bool, default=False*) – colorize the image after warping into the image space.

Todo:

- **[ ] Find a way to visualize offset, diameter, and class_probs** either individually or all at the same time

---

Example

```
>>> # xdoctest: +REQUIRES(module:kwplot)
>>> import kwarray
>>> import kwimage
>>> image = kwimage.grab_test_image('astro')
>>> probs = kwimage.gaussian_patch(image.shape[0:2])[None, :]
>>> probs = probs / probs.max()
>>> class_probs = kwarray.ArrayAPI.cat([probs, 1 - probs], axis=0)
>>> self = kwimage.Heatmap(class_probs=class_probs, offset=5 * np.random.
↪randn(2, *probs.shape[1:]))
>>> toshow = self.draw_on(image, 0, vecs=True, with_alpha=0.85)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(toshow)
```

Example

```
>>> # xdoctest: +REQUIRES(module:kwplot)
>>> # xdoctest: +REQUIRES(module:ndsampler)
>>> import kwimage
>>> self = kwimage.Heatmap.random(dims=(200, 200), dets='coco',␣
↪keypoints=True)
>>> image = kwimage.grab_test_image('astro')
>>> toshow = self.draw_on(image, 0, vecs=False, with_alpha=0.85)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(toshow)
```

Example

```
>>> # xdoctest: +REQUIRES(module:kwplot)
>>> # xdoctest: +REQUIRES(module:ndsampler)
>>> import kwimage
>>> self = kwimage.Heatmap.random(dims=(200, 200), dets='coco',␣
↪keypoints=True)
>>> kpts = [6]
>>> self = self.warp(self.tf_data_to_img.params)
```

(continues on next page)

(continued from previous page)

```
>>> image = kwimage.grab_test_image('astro')
>>> image = kwimage.ensure_alpha_channel(image)
>>> toshow = self.draw_on(image, 0, with_alpha=0.85, kpts=kpts)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(toshow)
```

class kwimage.structs.heatmap.**_HeatmapWarpMixin**

Bases: object

mixin method having to do with warping and aligning heatmaps

_align_other(*self*, *other*)

Warp another Heatmap (with the same underlying imgdims) into the same space as this heatmap. This lets us perform elementwise operations on the two heatmaps (like geometric mean).

Parameters **other** (*Heatmap*) – the heatmap to align with *self*

Returns warped version of *other* that aligns with *self*.

Return type *Heatmap*

Example

```
>>> self = Heatmap.random((120, 130), img_dims=(200, 210), classes=2,␣
↪nblips=10, rng=0)
>>> other = Heatmap.random((60, 70), img_dims=(200, 210), classes=2,␣
↪nblips=10, rng=1)
>>> other2 = self._align_other(other)
>>> assert self.shape != other.shape
>>> assert self.shape == other2.shape
>>> # xdoctest: +REQUIRES(--show)
>>> kwplot.autompl()
>>> kwplot.imshow(self.colorize(0, imgspace=False), fnum=1, pnum=(3, 2, 1))
>>> kwplot.imshow(self.colorize(1, imgspace=False), fnum=1, pnum=(3, 2, 2))
>>> kwplot.imshow(other.colorize(0, imgspace=False), fnum=1, pnum=(3, 2, 3))
>>> kwplot.imshow(other.colorize(1, imgspace=False), fnum=1, pnum=(3, 2, 4))
```

_align(*self*, *mask*, *interpolation='linear'*)

Align a linear combination of heatmap channels with the original image

DEPRICATE

_warp_imgspace(*self*, *chw*, *interpolation='linear'*)

upscale(*self*, *channel=None*, *interpolation='linear'*)

Warp the heatmap with the image dimensions

Example

```
>>> self = Heatmap.random(rng=0, dims=(32, 32))
>>> colormask = self.upscale()
```

warp(*self*, *mat=None*, *input_dims=None*, *output_dims=None*, *interpolation='linear'*, *modify_spatial_coords=True*, *int_interpolation='nearest'*, *mat_is_xy=True*, *version=None*)

Warp all spatial maps. If the map contains spatial data, that data is also warped (ignoring the translation component).

Parameters

- **mat** (*ArrayLike*) – transformation matrix
- **input_dims** (*tuple*) – unused, only exists for compatibility
- **output_dims** (*tuple*) – size of the output heatmap
- **interpolation** (*str*) – see *kwimage.warp_tensor*
- **int_interpolation** (*str*) – interpolation used for interger types (should be nearest)
- **mat_is_xy** (*bool, default=True*) – set to false if the matrix is in yx space instead of xy space

Returns this heatmap warped into a new spatial dimension

Return type *Heatmap*

Ignore: # Verify swapping rows 0 and 1 and then swapping columns 0 and 1 # Produces a matrix that works with permuted coordinates # It does. import sympy a, b, c, d, e, f, g, h, i, x, y, z = sympy.symbols('a, b, c, d, e, f, g, h, i, x, y, z') M1 = sympy.Matrix([[a, b, c], [d, e, f], [g, h, i]]) M2 = sympy.Matrix([[e, d, f], [b, a, c], [h, g, i]]) xy = sympy.Matrix([[x], [y], [z]]) yx = sympy.Matrix([[y], [x], [z]])

R1 = M1.multiply(xy) R2 = M2.multiply(yx) R3 = sympy.Matrix([[R1[1]], [R1[0]], [R1[2]],]) assert R2 == R3

Example

```
>>> from kwimage.structs.heatmap import *  # NOQA
>>> self = Heatmap.random(rng=0, keypoints=True)
>>> S = 3.0
>>> mat = np.eye(3) * S
>>> mat[-1, -1] = 1
>>> newself = self.warp(mat, np.array(self.dims) * S).numpy()
>>> assert newself.offset.shape[0] == 2
>>> assert newself.diameter.shape[0] == 2
>>> f1 = newself.offset.max() / self.offset.max()
>>> assert f1 == S
>>> f2 = newself.diameter.max() / self.diameter.max()
>>> assert f2 == S
```

Example

```
>>> import kwimage
>>> # xdoctest: +REQUIRES(module:ndsampler)
>>> self = kwimage.Heatmap.random(dims=(100, 100), dets='coco',␣
↪keypoints=True)
>>> image = np.zeros(self.img_dims)
>>> toshow = self.draw_on(image, 1, vecs=True, with_alpha=0.85)
>>> # xdoctest: +REQUIRES(--show)
```

(continues on next page)

(continued from previous page)

```
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.figure(fnum=1, doclf=True)
>>> kwplot.imshow(toshow)
```

scale (*self*, *factor*, *output_dims=None*, *interpolation='linear'*)
Scale the heatmap

translate (*self*, *offset*, *output_dims=None*, *interpolation='linear'*)

class kwimage.structs.heatmap.**_HeatmapAlgoMixin**
Bases: object

Algorithmic operations on heatmaps

classmethod combine (*cls*, *heatmaps*, *root_index=None*, *dtype=np.float32*)
Combine multiple heatmaps into a single heatmap.

Parameters

- **heatmaps** (*Sequence[Heatmap]*) – multiple heatmaps to combine into one
- **root_index** (*int*) – which heatmap in the sequence to align other heatmaps with

Returns the combined heatmap

Return type *Heatmap*

Example

```
>>> from kwimage.structs.heatmap import *  # NOQA
>>> a = Heatmap.random((120, 130), img_dims=(200, 210), classes=2, nblips=10,␣
↪rng=0)
>>> b = Heatmap.random((60, 70), img_dims=(200, 210), classes=2, nblips=10,␣
↪rng=1)
>>> c = Heatmap.random((40, 30), img_dims=(200, 210), classes=2, nblips=10,␣
↪rng=1)
>>> heatmaps = [a, b, c]
>>> newself = Heatmap.combine(heatmaps, root_index=2)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(a.colorize(0, imgspace=1), fnum=1, pnum=(4, 2, 1))
>>> kwplot.imshow(a.colorize(1, imgspace=1), fnum=1, pnum=(4, 2, 2))
>>> kwplot.imshow(b.colorize(0, imgspace=1), fnum=1, pnum=(4, 2, 3))
>>> kwplot.imshow(b.colorize(1, imgspace=1), fnum=1, pnum=(4, 2, 4))
>>> kwplot.imshow(c.colorize(0, imgspace=1), fnum=1, pnum=(4, 2, 5))
>>> kwplot.imshow(c.colorize(1, imgspace=1), fnum=1, pnum=(4, 2, 6))
>>> kwplot.imshow(newself.colorize(0, imgspace=1), fnum=1, pnum=(4, 2, 7))
>>> kwplot.imshow(newself.colorize(1, imgspace=1), fnum=1, pnum=(4, 2, 8))
>>> # xdoctest: +REQUIRES(--show)
>>> kwplot.imshow(a.colorize('offset', imgspace=1), fnum=2, pnum=(4, 1, 1))
>>> kwplot.imshow(b.colorize('offset', imgspace=1), fnum=2, pnum=(4, 1, 2))
>>> kwplot.imshow(c.colorize('offset', imgspace=1), fnum=2, pnum=(4, 1, 3))
>>> kwplot.imshow(newself.colorize('offset', imgspace=1), fnum=2, pnum=(4, 1,␣
↪4))
>>> # xdoctest: +REQUIRES(--show)
>>> kwplot.imshow(a.colorize('diameter', imgspace=1), fnum=3, pnum=(4, 1, 1))
```

(continues on next page)

(continued from previous page)

```
>>> kwplot.imshow(b.colorize('diameter', imgspace=1), fnum=3, pnum=(4, 1, 2))
>>> kwplot.imshow(c.colorize('diameter', imgspace=1), fnum=3, pnum=(4, 1, 3))
>>> kwplot.imshow(newself.colorize('diameter', imgspace=1), fnum=3, pnum=(4,␣
↪1, 4))
```

detect (*self*, *channel*, *invert=False*, *min_score=0.01*, *num_min=10*, *max_dims=None*, *min_dims=None*, *dim_thresh_space='image'*)

Lossy conversion from a Heatmap to a Detections object.

For efficiency, the detections are returned in the same space as the heatmap, which usually some downsampled version of the image space. This is because it is more efficient to transform the detections into image-space after non-max supression is applied.

Parameters

- **channel** (*int | ArrayLike[*DIMS]*) – class index to detect objects in. Alternatively, channel can be a custom probability map as long as its dimension agree with the heatmap.
- **invert** (*bool, default=False*) – if True, inverts the probabilities in the chosen channel. (Useful if you have a background channel but want to detect foreground objects).
- **min_score** (*float, default=0.1*) – probability threshold required for a pixel to be converted into a detection.
- **num_min** (*int, default=10*) – always return at least *nmin* of the highest scoring detections even if they aren't above the *min_score* threshold.
- **max_dims** (*Tuple[int, int]*) – maximum height / width of detections By default these are expected to be in image-space.
- **min_dims** (*Tuple[int, int]*) – minimum height / width of detections By default these are expected to be in image-space.
- **dim_thresh_space** (*str, default='image'*) – When dim_thresh_space=='native', dimension thresholds (e.g. min_dims and max_dims) are specified in the native heatmap space (i.e. usually a downsampled space). If dim_thresh_space=='image', then dimension thresholds are interpreted in the original image space.

Returns

raw detections.

Note that these detections will not have class_idx populated

It is the users responsbility to run non-max suppression on these results to remove duplicate detections.

Return type *kwimage.Detections*

SeeAlso: Detections.rasterize

Example

```
>>> # xdoctest: +REQUIRES(module:ndsampler)
>>> from kwimage.structs.heatmap import *  # NOQA
>>> import ndsampler
>>> self = Heatmap.random(rng=2, dims=(32, 32))
>>> dets = self.detect(channel=0, max_dims=7, num_min=None)
>>> img_dets = dets.warp(self.tf_data_to_img)
```

(continues on next page)

(continued from previous page)

```
>>> assert img_dets.boxes.to_xywh().width.max() <= 7
>>> assert img_dets.boxes.to_xywh().height.max() <= 7
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> dets1 = dets.sort().take(range(30))
>>> colormask1 = self.colorize(0, imgspace=False)
>>> kwplot.imshow(colormask1, pnum=(1, 2, 1), fnum=1, title='output space')
>>> dets1.draw()
>>> # Transform heatmap and detections into image space.
>>> dets2 = dets1.warp(self.tf_data_to_img)
>>> colormask2 = self.colorize(0, imgspace=True)
>>> kwplot.imshow(colormask2, pnum=(1, 2, 2), fnum=1, title='image space')
>>> dets2.draw()
```

Example

```
>>> # xdoctest: +REQUIRES(module:ndsampler)
>>> from kwimage.structs.heatmap import *  # NOQA
>>> import ndsampler
>>> catgraph = ndsampler.CategoryTree.demo()
>>> class_energy = torch.rand(len(catgraph), 32, 32)
>>> class_probs = catgraph.hierarchical_softmax(class_energy, dim=0)
>>> self = Heatmap.random(rng=0, dims=(32, 32), classes=catgraph,␣
↪keypoints=True)
>>> print(ub.repr2(ub.map_vals(lambda x: x.shape, self.data), nl=1))
>>> self.data['class_probs'] = class_probs.numpy()
>>> channel = catgraph.index('background')
>>> dets = self.detect(channel, invert=True)
>>> class_idx, scores = catgraph.decision(dets.probs, dim=1)
>>> dets.data['class_idx'] = class_idx
>>> dets.data['scores'] = scores
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> dets1 = dets.sort().take(range(10))
>>> colormask1 = self.colorize(0, imgspace=False)
>>> kwplot.imshow(colormask1, pnum=(1, 2, 1), fnum=1, title='output space')
>>> dets1.draw(radius=1.0)
>>> # Transform heatmap and detections into image space.
>>> colormask2 = self.colorize(0, imgspace=True)
>>> dets2 = dets1.warp(self.tf_data_to_img)
>>> kwplot.imshow(colormask2, pnum=(1, 2, 2), fnum=1, title='image space')
>>> dets2.draw(radius=1.0)
```

class kwimage.structs.heatmap.**Heatmap**(*data=None*, *meta=None*, ***kwargs*)

Bases: *kwimage.structs._generic.Spatial*, *kwimage.structs.heatmap._HeatmapDrawMixin*, *kwimage.structs.heatmap._HeatmapWarpMixin*, *kwimage.structs.heatmap._HeatmapAlgoMixin*

Keeps track of a downscaled heatmap and how to transform it to overlay the original input image. Heatmaps generally are used to estimate class probabilites at each pixel. This data struction additionally contains logic to augment pixel with offset (dydx) and scale (diamter) information.

Variables

- **data** (*Dict[str, object]*) – dictionary containing spatially aligned heatmap data. Valid keys are as follows.

  class_probs (ArrayLike[C, H, W] | ArrayLike[C, D, H, W]): A probability map for each class. C is the number of classes.

  offset (ArrayLike[2, H, W] | ArrayLike[3, D, H, W], optional): object center position offset in y,x / t,y,x coordinates

  diamter (ArrayLike[2, H, W] | ArrayLike[3, D, H, W], optional): object bounding box sizes in h,w / d,h,w coordinates

  keypoints (ArrayLike[2, K, H, W] | ArrayLike[3, K, D, H, W], optional): y/x offsets for K different keypoint classes

- **data** – dictionary containing miscellanious metadata about the heatmap data. Valid keys are as follows.

  img_dims (Tuple[H, W] | Tuple[D, H, W]): original image dimension

  tf_data_to_image (skimage.transform._geometric.GeometricTransform): transformation matrix (typically similarity or affine) that projects the given1.8719898042840075, heatmap onto the image dimensions such that the image and heatmap are spatially aligned.

  classes (List[str] | ndsampler.CategoryTree): information about which index in *data['class_probs']* corresponds to which semantic class.

- ****kwargs** – any key that is accepted by the *data* or *meta* dictionaries can be specified as a keyword argument to this class and it will be properly placed in the appropriate internal dictionary.

CommandLine: xdoctest -m ~/code/kwimage/kwimage/structs/heatmap.py Heatmap –show

Example

```
>>> import kwimage
>>> class_probs = kwimage.grab_test_image(dsize=(32, 32), space='gray')[None, ] /␣
↪255.0
>>> img_dims = (220, 220)
>>> tf_data_to_img = skimage.transform.AffineTransform(translation=(-18, -18),␣
↪scale=(8, 8))
>>> self = Heatmap(class_probs=class_probs, img_dims=img_dims,
>>>                tf_data_to_img=tf_data_to_img)
>>> aligned = self.upscale()
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(aligned[0])
>>> kwplot.show_if_requested()
```

__datakeys__ = ['class_probs', 'offset', 'diameter', 'keypoints', 'class_idx', 'class_

__metakeys__ = ['img_dims', 'tf_data_to_img', 'classes', 'kp_classes']

__spatialkeys__ = ['offset', 'diameter', 'keypoints']

shape

bounds

dims
space-time dimensions of this heatmap

_impl
Returns the internal tensor/numpy ArrayAPI implementation

Returns kwarray.ArrayAPI

class_probs

offset

diameter

img_dims

tf_data_to_img

classes

__nice__(*self*)

__getitem__(*self*, *index*)

__len__(*self*)

is_numpy(*self*)

is_tensor(*self*)

classmethod random(*cls*, *dims=(10, 10)*, *classes=3*, *diameter=True*, *offset=True*, *keypoints=False*, *img_dims=None*, *dets=None*, *nblips=10*, *noise=0.0*, *rng=None*)
Creates dummy data, suitable for use in tests and benchmarks

Parameters

- **dims** (*Tuple*) – dimensions of the heatmap
- **img_dims** (*Tuple*) – dimensions of the image the heatmap corresponds to

Example

```
>>> from kwimage.structs.heatmap import *  # NOQA
>>> self = Heatmap.random((128, 128), img_dims=(200, 200),
>>>     classes=3, nblips=10, rng=0, noise=0.1)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(self.colorize(0, imgspace=0), fnum=1, pnum=(1, 4, 1),␣
↪doclf=1)
>>> kwplot.imshow(self.colorize(1, imgspace=0), fnum=1, pnum=(1, 4, 2))
>>> kwplot.imshow(self.colorize(2, imgspace=0), fnum=1, pnum=(1, 4, 3))
>>> kwplot.imshow(self.colorize(3, imgspace=0), fnum=1, pnum=(1, 4, 4))
```

Ignore: self.detect(0).sort().non_max_supress()[-np.arange(1, 4)].draw() from kwimage.structs.heatmap import * # NOQA import xdev globals().update(xdev.get_func_kwargs(Heatmap.random))

Example

```
>>> # xdoctest: +REQUIRES(module:ndsampler)
>>> import kwimage
>>> self = kwimage.Heatmap.random(dims=(50, 200), dets='coco',
>>>                               keypoints=True)
>>> image = np.zeros(self.img_dims)
>>> toshow = self.draw_on(image, 1, vecs=True, kpts=0, with_alpha=0.85)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.figure(fnum=1, doclf=True)
>>> kwplot.imshow(toshow)
```

Ignore:

```
>>> kwplot.figure(fnum=1, doclf=True)
>>> kwplot.imshow(image)
>>> dets.draw()
>>> dets.data['keypoints'].draw(radius=6)
>>> dets.data['segmentations'].draw()
```

```
>>> self.draw()
```

numpy(*self*)
Converts underlying data to numpy arrays

tensor(*self*, *device=ub.NoParam*)
Converts underlying data to torch tensors

kwimage.structs.heatmap.**_prob_to_dets**(*probs*, *diameter=None*, *offset=None*, *class_probs=None*, *keypoints=None*, *min_score=0.01*, *num_min=10*, *max_dims=None*, *min_dims=None*)

Directly convert a one-channel probability map into a Detections object.

Helper for Heatmap.detect

It does this by converting each pixel above a threshold in a probability map to a detection with a specified diameter.

Parameters

- **probs** (*ArrayLike[H, W]*) – liklihood that each particular pixel should be detected as an object.
- **diameter** (*ArrayLike[2, H, W] | Tuple*) – H, W sizes for the bounding box at each pixel location. If passed as a tuple, then all boxes receive that diameter.
- **offset** (*Tuple | ArrayLike[2, H, W], default=0*) – Y, X offsets from the pixel location to the bounding box center. If passed as a tuple, then all boxes receive that offset.
- **class_probs** (*ArrayLike[C, H, W], optional*) – probabilities for each class at each pixel location. If specified, this will populate the *probs* attribute of the returned Detections object.
- **keypoints** (*ArrayLike[2, K, H, W], optional*) – Keypoint predictions for all keypoint classes
- **min_score** (*float, default=0.1*) – probability threshold required for a pixel to be converted into a detection.
- **num_min** (*int, default=10*) – always return at least *nmin* of the highest scoring detections even if they aren't above the *min_score* threshold.

Returns

raw detections. It is the users responsbility to run non-max suppression on these results to remove duplicate detections.

Return type *kwimage.Detections*

Example

```
>>> rng = np.random.RandomState(0)
>>> probs = rng.rand(3, 3).astype(np.float32)
>>> min_score = .5
>>> diameter = [10, 10]
>>> dets = _prob_to_dets(probs, diameter, min_score=min_score)
>>> assert dets.boxes.data.dtype.kind == 'f'
>>> assert len(dets) == 9
>>> dets = _prob_to_dets(torch.FloatTensor(probs), diameter, min_score=min_score)
>>> assert dets.boxes.data.dtype.is_floating_point
>>> assert len(dets) == 9
```

Example

```
>>> import kwimage
>>> from kwimage.structs.heatmap import *
>>> from kwimage.structs.heatmap import _prob_to_dets
>>> heatmap = kwimage.Heatmap.random(rng=0, dims=(3, 3), keypoints=True)
>>> # Try with numpy
>>> min_score = .5
>>> dets = _prob_to_dets(heatmap.class_probs[0], heatmap.diameter,
>>>                          heatmap.offset, heatmap.class_probs,
>>>                          heatmap.data['keypoints'],
>>>                          min_score)
>>> assert dets.boxes.data.dtype.kind == 'f'
>>> assert 'keypoints' in dets.data
>>> dets_np = dets
>>> # Try with torch
>>> heatmap = heatmap.tensor()
>>> dets = _prob_to_dets(heatmap.class_probs[0], heatmap.diameter,
>>>                          heatmap.offset, heatmap.class_probs,
>>>                          heatmap.data['keypoints'],
>>>                          min_score)
>>> assert dets.boxes.data.dtype.is_floating_point
>>> assert len(dets) == len(dets_np)
>>> dets_torch = dets
>>> assert np.all(dets_torch.numpy().boxes.data == dets_np.boxes.data)
```

Ignore: import kwil kwil.autompl() dets.draw(setlim=True, radius=.1)

Example

```
>>> heatmap = Heatmap.random(rng=0, dims=(3, 3), diameter=1)
>>> probs = heatmap.class_probs[0]
>>> diameter = heatmap.diameter
```

(continues on next page)

(continued from previous page)

```
>>> offset = heatmap.offset
>>> class_probs = heatmap.class_probs
>>> min_score = 0.5
>>> dets = _prob_to_dets(probs, diameter, offset, class_probs, None, min_score)
```

kwimage.structs.heatmap.**smooth_prob**(*prob*, *k=3*, *inplace=False*, *eps=1e-09*)
Smooths the probability map, but preserves the magnitude of the peaks.

Notes

even if inplace is true, we still need to make a copy of the input array, however, we do ensure that it is cleaned up before we leave the function scope.

sigma=0.8 @ k=3, sigma=1.1 @ k=5, sigma=1.4 @ k=7

kwimage.structs.heatmap.**_remove_translation**(*tf*)
Removes the translation component of a transform

Todo:

- [ ] Is this possible in more general cases? E.g. projective transforms?

---

kwimage.structs.heatmap.**_gmean**(*a*, *axis=0*, *clobber=False*)
Compute the geometric mean along the specified axis.

Modification of the scipy.mstats method to be more memory efficient

Example

```
>>> rng = np.random.RandomState(0)
>>> C, H, W = 8, 32, 32
>>> axis = 0
>>> a = rng.rand(2, C, H, W)
>>> _gmean(a)
```

kwimage.structs.mask

Data structure for Binary Masks

Structure for efficient encoding of per-annotation segmentation masks Based on efficient cython/C code in the cocoapi [1].

References

Goals: The goal of this file is to create a datastructure that lets the developer seemlessly convert between:

(1) raw binary uint8 masks

(2) memory-efficient comprsssed run-length-encodings of binary segmentation masks. (3) convex polygons (4) convex hull polygons (5) bounding box

It is not there yet, and the API is subject to change in order to better accomplish these goals.

Notes

IN THIS FILE ONLY: size corresponds to a h/w tuple to be compatible with the coco semantics. Everywhere else in this repo, size uses opencv semantics which are w/h.

Module Contents

class kwimage.structs.mask.**Mask**(*data=None*, *format=None*)

Bases: ubelt.NiceRepr, kwimage.structs.mask._MaskConversionMixin, kwimage.structs.mask._MaskConstructorMixin, kwimage.structs.mask._MaskTransformMixin, kwimage.structs.mask._MaskDrawMixin

Manages a single segmentation mask and can convert to and from multiple formats including:

- bytes_rle - byte encoded run length encoding
- array_rle - raw run length encoding
- c_mask - c-style binary mask
- f_mask - fortran-style binary mask

Example

```
>>> # xdoc: +REQUIRES(--mask)
>>> # a ms-coco style compressed bytes rle segmentation
>>> segmentation = {'size': [5, 9], 'counts': ';?1B10O30O4'}
>>> mask = Mask(segmentation, 'bytes_rle')
>>> # convert to binary numpy representation
>>> binary_mask = mask.to_c_mask().data
>>> print(ub.repr2(binary_mask.tolist(), nl=1, nobr=1))
[0, 0, 0, 1, 1, 1, 1, 1, 0],
[0, 0, 1, 1, 1, 0, 0, 0, 0],
[0, 0, 1, 1, 1, 1, 1, 1, 0],
[0, 0, 1, 1, 1, 0, 1, 1, 0],
[0, 0, 1, 1, 1, 0, 1, 1, 0],
```

dtype

shape

area

Returns the number of non-zero pixels

Example

```
>>> # xdoc: +REQUIRES(--mask)
>>> self = Mask.demo()
>>> self.area
150
```

__nice__(*self*)

classmethod random(*Mask*, *rng=None*, *shape=(32, 32)*)

Example

Mask.random(rng=0).draw()

copy (*self*)
Performs a deep copy of the mask data

Example

```
>>> self = Mask.random(shape=(8, 8), rng=0)
>>> other = self.copy()
>>> assert other.data is not self.data
```

union (*self*, **others*)
This can be used as a staticmethod or an instancemethod

Example

```
>>> # xdoc: +REQUIRES(--mask)
>>> from kwimage.structs.mask import *  # NOQA
>>> masks = [Mask.random(shape=(8, 8), rng=i) for i in range(2)]
>>> mask = Mask.union(*masks)
>>> print(mask.area)
>>> masks = [m.to_c_mask() for m in masks]
>>> mask = Mask.union(*masks)
>>> print(mask.area)
```

```
>>> masks = [m.to_bytes_rle() for m in masks]
>>> mask = Mask.union(*masks)
>>> print(mask.area)
```

Benchmark: import ubelt as ub ti = ub.Timerit(100, bestof=10, verbose=2)

masks = [Mask.random(shape=(172, 172), rng=i) for i in range(2)]

for timer in ti.reset('native rle union'): masks = [m.to_bytes_rle() for m in masks] with timer:

mask = Mask.union(*masks)

for timer in ti.reset('native cmask union'): masks = [m.to_c_mask() for m in masks] with timer:

mask = Mask.union(*masks)

for timer in ti.reset('cmask->rle union'): masks = [m.to_c_mask() for m in masks] with timer:

mask = Mask.union(*[m.to_bytes_rle() for m in masks])

intersection (*self*, **others*)
This can be used as a staticmethod or an instancemethod

Example

```
>>> # xdoc: +REQUIRES(--mask)
>>> masks = [Mask.random(shape=(8, 8), rng=i) for i in range(2)]
>>> mask = Mask.intersection(*masks)
>>> print(mask.area)
```

get_patch(*self*)

Extract the patch with non-zero data

Example

```
>>> # xdoc: +REQUIRES(--mask)
>>> from kwimage.structs.mask import *  # NOQA
>>> self = Mask.random(shape=(8, 8), rng=0)
>>> self.get_patch()
```

get_xywh(*self*)

Gets the bounding xywh box coordinates of this mask

Returns

x, y, w, h: Note we dont use a Boxes object because a general singular version does not yet exist.

Return type ndarray

Example

```
>>> # xdoc: +REQUIRES(--mask)
>>> self = Mask.random(shape=(8, 8), rng=0)
>>> self.get_xywh().tolist()
>>> self = Mask.random(rng=0).translate((10, 10))
>>> self.get_xywh().tolist()
```

get_polygon(*self*)

Returns a list of (x,y)-coordinate lists. The length of the list is equal to the number of disjoint regions in the mask.

Returns

polygon around each connected component of the mask. Each ndarray is an Nx2 array of xy points.

Return type List[ndarray]

Note: The returned polygon may not surround points that are only one pixel thick.

Example

```
>>> # xdoc: +REQUIRES(--mask)
>>> from kwimage.structs.mask import *  # NOQA
>>> self = Mask.random(shape=(8, 8), rng=0)
>>> polygons = self.get_polygon()
>>> print('polygons = ' + ub.repr2(polygons))
```

(continues on next page)

(continued from previous page)

```
>>> polygons = self.get_polygon()
>>> self = self.to_bytes_rle()
>>> other = Mask.from_polygons(polygons, self.shape)
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> image = np.ones(self.shape)
>>> image = self.draw_on(image, color='blue')
>>> image = other.draw_on(image, color='red')
>>> kwplot.imshow(image)
```

polygons = [np.array([[6, 4],[7, 4]], dtype=np.int32), np.array([[0, 1],[0, 3],[2, 3],[2, 1]], dtype=np.int32),

]

to_mask (*self*, *dims=None*)

to_boxes (*self*)

Returns the bounding box of the mask.

classmethod demo (*cls*)

Demo mask with holes and disjoint shapes

to_multi_polygon (*self*)

Returns a MultiPolygon object fit around this raster including disjoint pieces and holes.

Returns vectorized representation

Return type *MultiPolygon*

Example

```
>>> # xdoc: +REQUIRES(--mask)
>>> from kwimage.structs.mask import *  # NOQA
>>> self = Mask.demo()
>>> self = self.scale(5)
>>> multi_poly = self.to_multi_polygon()
>>> # xdoc: +REQUIRES(module:kwplot)
>>> # xdoc: +REQUIRES(--show)
>>> self.draw(color='red')
>>> multi_poly.scale(1.1).draw(color='blue')
```

```
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> image = np.ones(self.shape)
>>> image = self.draw_on(image, color='blue')
>>> #image = other.draw_on(image, color='red')
>>> kwplot.imshow(image)
>>> multi_poly.draw()
```

get_convex_hull (*self*)

Returns a list of xy points around the convex hull of this mask

Note: The returned polygon may not surround points that are only one pixel thick.

Example

```
>>> # xdoc: +REQUIRES(--mask)
>>> self = Mask.random(shape=(8, 8), rng=0)
>>> polygons = self.get_convex_hull()
>>> print('polygons = ' + ub.repr2(polygons))
>>> other = Mask.from_polygons(polygons, self.shape)
```

iou (*self*, *other*)

The area of intersection over the area of union

Todo:

- **[ ] Write plural Masks version of this class, which should** be able to perform this operation more efficiently.

CommandLine: xdoctest -m kwimage.structs.mask Mask.iou

Example

```
>>> # xdoc: +REQUIRES(--mask)
>>> self = Mask.demo()
>>> other = self.translate(1)
>>> iou = self.iou(other)
>>> print('iou = {:.4f}'.format(iou))
iou = 0.0830
```

classmethod coerce (*Mask*, *data*, *dims=None*)

Attempts to auto-inspect the format of the data and conver to Mask

Parameters

- **data** – the data to coerce
- **dims** (*Tuple*) – required for certain formats like polygons height / width of the source image

Returns Mask

Example

```
>>> # xdoc: +REQUIRES(--mask)
>>> segmentation = {'size': [5, 9], 'counts': ';?1B10O30O4'}
>>> polygon = [
>>>     [np.array([[3, 0],[2, 1],[2, 4],[4, 4],[4, 3],[7, 0]])],
>>>     [np.array([[2, 1],[2, 2],[4, 2],[4, 1]])],
>>> ]
>>> dims = (9, 5)
>>> mask = (np.random.rand(32, 32) > .5).astype(np.uint8)
```

(continues on next page)

(continued from previous page)

```
>>> Mask.coerce(polygon, dims).to_bytes_rle()
>>> Mask.coerce(segmentation).to_bytes_rle()
>>> Mask.coerce(mask).to_bytes_rle()
```

_to_coco(*self*)

Example

```
>>> # xdoc: +REQUIRES(--mask)
>>> from kwimage.structs.mask import *  # NOQA
>>> self = Mask.demo()
>>> data = self._to_coco()
>>> print(ub.repr2(data, nl=1))
```

to_coco(*self*, *style='orig'*)

Example

```
>>> # xdoc: +REQUIRES(--mask)
>>> from kwimage.structs.mask import *  # NOQA
>>> self = Mask.demo()
>>> data = self.to_coco()
>>> print(ub.repr2(data, nl=1))
```

class kwimage.structs.mask.**MaskList**

Bases: *kwimage.structs._generic.ObjectList*

Store and manipulate multiple masks, usually within the same image

to_polygon_list(*self*)

Converts all mask objects to polygon objects

kwimage.structs.points

Module Contents

class kwimage.structs.points.**_PointsWarpMixin**

dtype

_warp_imgaug(*self*, *augmenter*, *input_dims*, *inplace=False*)

Warps by applying an augmenter from the imgaug library

Parameters

- **augmenter** (*imgaug.augmenters.Augmenter*)
- **input_dims** (*Tuple*) – h/w of the input image
- **inplace** (*bool, default=False*) – if True, modifies data inplace

Example

```
>>> # xdoctest: +REQUIRES(module:imgaug)
>>> from kwimage.structs.points import *  # NOQA
>>> import imgaug
>>> input_dims = (10, 10)
>>> self = Points.random(10).scale(input_dims)
>>> augmenter = imgaug.augmenters.Fliplr(p=1)
>>> new = self._warp_imgaug(augmenter, input_dims)
```

```
>>> self = Points(xy=(np.random.rand(10, 2) * 10).astype(np.int))
>>> augmenter = imgaug.augmenters.Fliplr(p=1)
>>> new = self._warp_imgaug(augmenter, input_dims)
```

```
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> plt = kwplot.autoplt()
>>> kwplot.figure(fnum=1, doclf=True)
>>> ax = plt.gca()
>>> ax.set_xlim(0, 10)
>>> ax.set_ylim(0, 10)
>>> self.draw(color='red', alpha=.4, radius=0.1)
>>> new.draw(color='blue', alpha=.4, radius=0.1)
```

to_imgaug(*self*, *input_dims*)

Example

```
>>> # xdoctest: +REQUIRES(module:imgaug)
>>> from kwimage.structs.points import *  # NOQA
>>> pts = Points.random(10)
>>> input_dims = (10, 10)
>>> kpoi = pts.to_imgaug(input_dims)
```

classmethod from_imgaug(*cls*, *kpoi*)

warp(*self*, *transform*, *input_dims=None*, *output_dims=None*, *inplace=False*)

Generalized coordinate transform.

Parameters

- **transform** (*GeometricTransform | ArrayLike | Augmenter*) – scikit-image tranform, a 3x3 transformation matrix, or an imgaug Augmenter.
- **input_dims** (*Tuple*) – shape of the image these objects correspond to (only needed / used when transform is an imgaug augmenter)
- **output_dims** (*Tuple*) – unused, only exists for compatibility
- **inplace** (*bool, default=False*) – if True, modifies data inplace

Example

```
>>> from kwimage.structs.points import *  # NOQA
>>> self = Points.random(10, rng=0)
```

(continues on next page)

(continued from previous page)

```
>>> transform = skimage.transform.AffineTransform(scale=(2, 2))
>>> new = self.warp(transform)
>>> assert np.all(new.xy == self.scale(2).xy)
```

Doctest:

```
>>> self = Points.random(10, rng=0)
>>> assert np.all(self.warp(np.eye(3)).xy == self.xy)
>>> assert np.all(self.warp(np.eye(2)).xy == self.xy)
```

scale (*self*, *factor*, *output_dims=None*, *inplace=False*)

Scale a points by a factor

Parameters

- **factor** (*float or Tuple[float, float]*) – scale factor as either a scalar or a (sf_x, sf_y) tuple.
- **output_dims** (*Tuple*) – unused in non-raster spatial structures

Example

```
>>> from kwimage.structs.points import *  # NOQA
>>> self = Points.random(10, rng=0)
>>> new = self.scale(10)
>>> assert new.xy.max() <= 10
```

translate (*self*, *offset*, *output_dims=None*, *inplace=False*)

Shift the points

Parameters

- **factor** (*float or Tuple[float]*) – transation amount as either a scalar or a (t_x, t_y) tuple.
- **output_dims** (*Tuple*) – unused in non-raster spatial structures

Example

```
>>> from kwimage.structs.points import *  # NOQA
>>> self = Points.random(10, rng=0)
>>> new = self.translate(10)
>>> assert new.xy.min() >= 10
>>> assert new.xy.max() <= 11
```

class kwimage.structs.points.**Points**(*data=None*, *meta=None*, *datakeys=None*, *metakeys=None*, ***kwargs*)

Bases: *kwimage.structs._generic.Spatial*, *kwimage.structs.points._PointsWarpMixin*

Stores multiple keypoints for a single object.

This stores both the geometry and the class metadata if available

Ignore:

meta = { “names” = [‘head’, ‘nose’, ‘tail’], “skeleton” = [(0, 1), (0, 2)],

}

Example

```
>>> from kwimage.structs.points import *  # NOQA
>>> xy = np.random.rand(10, 2)
>>> pts = Points(xy=xy)
>>> print('pts = {!r}'.format(pts))
```

__datakeys__ = ['xy', 'class_idxs', 'visible']

__metakeys__ = ['classes']

__repr__

shape

xy

__nice__(*self*)

__len__(*self*)

classmethod random(*Points*, *num=1*, *classes=None*, *rng=None*)

Makes random points; typically for testing purposes

Example

```
>>> import kwimage
>>> self = kwimage.Points.random(classes=[1, 2, 3])
>>> self.data
>>> print('self.data = {!r}'.format(self.data))
```

is_numpy(*self*)

is_tensor(*self*)

_impl(*self*)

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

Example

```
>>> from kwimage.structs.points import *  # NOQA
>>> self = Points.random(10)
>>> self.tensor()
```

numpy(*self*)

Example

```
>>> from kwimage.structs.points import *  # NOQA
>>> self = Points.random(10)
>>> self.tensor().numpy().tensor().numpy()
```

draw_on(*self*, *image*, *color='white'*, *radius=None*, *copy=False*)

CommandLine: xdoctest -m ~/code/kwimage/kwimage/structs/points.py Points.draw_on –show

Example

```
>>> # xdoc: +REQUIRES(module:kwplot)
>>> from kwimage.structs.points import *  # NOQA
>>> s = 128
>>> image = np.zeros((s, s))
>>> self = Points.random(10).scale(s)
>>> image = self.draw_on(image)
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.figure(fnum=1, doclf=True)
>>> kwplot.autompl()
>>> kwplot.imshow(image)
>>> self.draw(radius=3, alpha=.5)
>>> kwplot.show_if_requested()
```

Example

```
>>> # xdoc: +REQUIRES(module:kwplot)
>>> from kwimage.structs.points import *  # NOQA
>>> s = 128
>>> image = np.zeros((s, s))
>>> self = Points.random(10).scale(s)
>>> image = self.draw_on(image, radius=3, color='distinct')
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.figure(fnum=1, doclf=True)
>>> kwplot.autompl()
>>> kwplot.imshow(image)
>>> self.draw(radius=3, alpha=.5, color='classes')
>>> kwplot.show_if_requested()
```

Example

```
>>> import kwimage
>>> s = 32
>>> self = kwimage.Points.random(10).scale(s)
>>> color = 'blue'
>>> # Test drawong on all channel + dtype combinations
>>> im3 = np.zeros((s, s, 3), dtype=np.float32)
>>> im_chans = {
>>>     'im3': im3,
>>>     'im1': kwimage.convert_colorspace(im3, 'rgb', 'gray'),
>>>     'im4': kwimage.convert_colorspace(im3, 'rgb', 'rgba'),
>>> }
>>> inputs = {}
>>> for k, im in im_chans.items():
>>>     inputs[k + '_01'] = (kwimage.ensure_float01(im.copy()), {'radius':␣
↪None})
>>>     inputs[k + '_255'] = (kwimage.ensure_uint255(im.copy()), {'radius':␣
↪None})
>>> outputs = {}
>>> for k, v in inputs.items():
>>>     im, kw = v
```

(continues on next page)

(continued from previous page)

```
>>>     outputs[k] = self.draw_on(im, color=color, **kw)
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.figure(fnum=2, doclf=True)
>>> kwplot.autompl()
>>> pnum_ = kwplot.PlotNums(nCols=2, nRows=len(inputs))
>>> for k in inputs.keys():
>>>     kwplot.imshow(inputs[k][0], fnum=2, pnum=pnum_(), title=k)
>>>     kwplot.imshow(outputs[k], fnum=2, pnum=pnum_(), title=k)
>>> kwplot.show_if_requested()
```

draw(*self*, *color='blue'*, *ax=None*, *alpha=None*, *radius=1*, ***kwargs*)

Example

```
>>> # xdoc: +REQUIRES(module:kwplot)
>>> from kwimage.structs.points import *  # NOQA
>>> pts = Points.random(10)
>>> # xdoc: +REQUIRES(--show)
>>> pts.draw(radius=0.01)
```

```
>>> from kwimage.structs.points import *  # NOQA
>>> self = Points.random(10, classes=['a', 'b', 'c'])
>>> self.draw(radius=0.01, color='classes')
```

compress(*self*, *flags*, *axis=0*, *inplace=False*)

Filters items based on a boolean criterion

Example

```
>>> from kwimage.structs.points import *  # NOQA
>>> self = Points.random(4)
>>> flags = [1, 0, 1, 1]
>>> other = self.compress(flags)
>>> assert len(self) == 4
>>> assert len(other) == 3
```

```
>>> other = self.tensor().compress(flags)
>>> assert len(other) == 3
```

take(*self*, *indices*, *axis=0*, *inplace=False*)

Takes a subset of items at specific indices

Example

```
>>> from kwimage.structs.points import *  # NOQA
>>> self = Points.random(4)
>>> indices = [1, 3]
>>> other = self.take(indices)
>>> assert len(self) == 4
>>> assert len(other) == 2
```

```
>>> other = self.tensor().take(indices)
>>> assert len(other) == 2
```

classmethod concatenate(*cls*, *points*, *axis=0*)

to_coco(*self*, *style='orig'*)
Converts to an mscoco-like representation

Note: items that are usually id-references to other objects may need to be rectified.

Parameters **style** (*str*) – either orig, new, new-id, or new-name

Returns mscoco-like representation

Return type Dict

Example

```
>>> from kwimage.structs.points import *  # NOQA
>>> self = Points.random(4, classes=['a', 'b'])
>>> orig = self._to_coco(style='orig')
>>> print('orig = {!r}'.format(orig))
>>> new_name = self._to_coco(style='new-name')
>>> print('new_name = {}'.format(ub.repr2(new_name, nl=-1)))
>>> # xdoctest: +REQUIRES(module:ndsampler)
>>> import ndsampler
>>> self.meta['classes'] = ndsampler.CategoryTree.coerce(self.meta['classes'])
>>> new_id = self._to_coco(style='new-id')
>>> print('new_id = {}'.format(ub.repr2(new_id, nl=-1)))
```

_to_coco(*self*, *style='orig'*)
See to_coco

classmethod _from_coco(*cls*, *coco_kpts*, *class_idxs=None*, *classes=None*)

classmethod from_coco(*cls*, *coco_kpts*, *class_idxs=None*, *classes=None*)

Parameters

- **coco_kpts** (*list | dict*) – either the original list keypoint encoding or the new dict keypoint encoding.
- **class_idxs** (*list*) – only needed if using old style
- **classes** (*list | CategoryTree*) – list of all keypoint category names

Example

```
>>> ##
>>> classes = ['mouth', 'left-hand', 'right-hand']
>>> coco_kpts = [
>>>     {'xy': (0, 0), 'visible': 2, 'keypoint_category': 'left-hand'},
>>>     {'xy': (1, 2), 'visible': 2, 'keypoint_category': 'mouth'},
>>> ]
>>> Points.from_coco(coco_kpts, classes=classes)
```

(continues on next page)

(continued from previous page)

```
>>> # Test without classes
>>> Points.from_coco(coco_kpts)
>>> # Test without any category info
>>> coco_kpts2 = [ub.dict_diff(d, {'keypoint_category'}) for d in coco_kpts]
>>> Points.from_coco(coco_kpts2)
>>> # Test without category instead of keypoint_category
>>> coco_kpts3 = [ub.map_keys(lambda x: x.replace('keypoint_', ''), d) for d␣
↪in coco_kpts]
>>> Points.from_coco(coco_kpts3)
>>> #
>>> # Old style
>>> coco_kpts = [0, 0, 2, 0, 1, 2]
>>> Points.from_coco(coco_kpts)
>>> # Fail case
>>> coco_kpts4 = [{'xy': [4686.5, 1341.5], 'category': 'dot'}]
>>> Points.from_coco(coco_kpts4, classes=[])
```

Example

```
>>> # xdoctest: +REQUIRES(module:ndsampler)
>>> import ndsampler
>>> classes = ndsampler.CategoryTree.from_coco([
>>>     {'name': 'mouth', 'id': 2}, {'name': 'left-hand', 'id': 3}, {'name':
↪'right-hand', 'id': 5}
>>> ])
>>> coco_kpts = [
>>>     {'xy': (0, 0), 'visible': 2, 'keypoint_category_id': 5},
>>>     {'xy': (1, 2), 'visible': 2, 'keypoint_category_id': 2},
>>> ]
>>> pts = Points.from_coco(coco_kpts, classes=classes)
>>> assert pts.data['class_idxs'].tolist() == [2, 0]
```

class kwimage.structs.points.**PointsList**

Bases: *kwimage.structs._generic.ObjectList*

Stores a list of Points, each item usually corresponds to a different object.

Notes

TODO: when the data is homogenous we can use a more efficient # representation, otherwise we have to use heterogenous storage.

kwimage.structs.polygon

Module Contents

class kwimage.structs.polygon.**_PolyArrayBackend**

is_numpy(*self*)

is_tensor(*self*)

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

Example

```
>>> from kwimage.structs.polygon import *
>>> self = Polygon.random()
>>> self.tensor()
```

numpy(*self*)

Example

```
>>> from kwimage.structs.polygon import *
>>> self = Polygon.random()
>>> self.tensor().numpy().tensor().numpy()
```

class kwimage.structs.polygon.**_PolyWarpMixin**

_warp_imgaug(*self*, *augmenter*, *input_dims*, *inplace=False*)
Warps by applying an augmenter from the imgaug library

Parameters

- **augmenter** (*imgaug.augmenters.Augmenter*)
- **input_dims** (*Tuple*) – h/w of the input image
- **inplace** (*bool, default=False*) – if True, modifies data inplace

Example

```
>>> # xdoctest: +REQUIRES(module:imgaug)
>>> from kwimage.structs.polygon import *  # NOQA
>>> import imgaug
>>> input_dims = np.array((10, 10))
>>> self = Polygon.random(10, n_holes=1, rng=0).scale(input_dims)
>>> augmenter = imgaug.augmenters.Fliplr(p=1)
>>> new = self._warp_imgaug(augmenter, input_dims)
>>> assert np.allclose(self.data['exterior'].data[:, 1], new.data['exterior'].
↪data[:, 1])
>>> assert np.allclose(input_dims[0] - self.data['exterior'].data[:, 0], new.
↪data['exterior'].data[:, 0])
```

```
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.figure(fnum=1, doclf=True)
>>> from matplotlib import pyplot as pl
>>> ax = plt.gca()
>>> ax.set_xlim(0, 10)
>>> ax.set_ylim(0, 10)
>>> self.draw(color='red', alpha=.4)
>>> new.draw(color='blue', alpha=.4)
```

to_imgaug(*self*, *shape*)

warp(*self*, *transform*, *input_dims=None*, *output_dims=None*, *inplace=False*)
Generalized coordinate transform.

Parameters

- **transform** (*GeometricTransform | ArrayLike | Augmenter*) – scikit-image tranform, a 3x3 transformation matrix, or an imgaug Augmenter.
- **input_dims** (*Tuple*) – shape of the image these objects correspond to (only needed / used when transform is an imgaug augmenter)
- **output_dims** (*Tuple*) – unused, only exists for compatibility
- **inplace** (*bool, default=False*) – if True, modifies data inplace

Example

```
>>> from kwimage.structs.polygon import *  # NOQA
>>> self = Polygon.random()
>>> transform = skimage.transform.AffineTransform(scale=(2, 2))
>>> new = self.warp(transform)
```

Doctest:

```
>>> # xdoctest: +REQUIRES(module:imgaug)
>>> self = Polygon.random()
>>> import imgaug
>>> augmenter = imgaug.augmenters.Fliplr(p=1)
>>> new = self.warp(augmenter, input_dims=(1, 1))
>>> print('new = {!r}'.format(new.data))
>>> print('self = {!r}'.format(self.data))
>>> #assert np.all(self.warp(np.eye(3)).exterior == self.exterior)
>>> #assert np.all(self.warp(np.eye(2)).exterior == self.exterior)
```

scale (*self*, *factor*, *output_dims=None*, *inplace=False*)

Scale a polygon by a factor

Parameters

- **factor** (*float or Tuple[float, float]*) – scale factor as either a scalar or a (sf_x, sf_y) tuple.
- **output_dims** (*Tuple*) – unused in non-raster spatial structures

Example

```
>>> from kwimage.structs.polygon import *  # NOQA
>>> self = Polygon.random(10, rng=0)
>>> new = self.scale(10)
```

translate (*self*, *offset*, *output_dims=None*, *inplace=False*)

Shift the polygon up/down left/right

Parameters

- **factor** (*float or Tuple[float]*) – transation amount as either a scalar or a (t_x, t_y) tuple.
- **output_dims** (*Tuple*) – unused in non-raster spatial structures

Example

```
>>> from kwimage.structs.polygon import *  # NOQA
>>> self = Polygon.random(10, rng=0)
>>> new = self.translate(10)
```

class kwimage.structs.polygon.**Polygon**(*data=None*, *meta=None*, *datakeys=None*, *metakeys=None*, ***kwargs*)

Bases: *kwimage.structs._generic.Spatial*, *kwimage.structs.polygon._PolyArrayBackend*, *kwimage.structs.polygon._PolyWarpMixin*, ubelt.NiceRepr

Represents a single polygon as set of exterior boundary points and a list of internal polygons representing holes.

By convention exterior boundaries should be counterclockwise and interior holes should be clockwise.

Example

```
>>> data = {
>>>     'exterior': np.array([[13,  1], [13, 19], [25, 19], [25,  1]]),
>>>     'interiors': [
>>>         np.array([[13, 13], [14, 12], [24, 12], [25, 13], [25, 18], [24, 19],␣
↪[14, 19], [13, 18]]),
>>>         np.array([[13,  2], [14,  1], [24,  1], [25, 2], [25, 11], [24, 12],␣
↪[14, 12], [13, 11]])]
>>> }
>>> self = Polygon(**data)
```

__datakeys__ = ['exterior', 'interiors']

__metakeys__ = ['classes']

__nice__(*self*)

classmethod circle(*cls*, *xy*, *r*, *resolution=64*)

Create a circular polygon

Example

```
>>> xy = (0.5, 0.5)
>>> r = .3
>>> poly = Polygon.circle(xy, r)
```

classmethod random(*cls*, *n=6*, *n_holes=0*, *convex=True*, *tight=False*, *rng=None*)

Parameters

- **n** (*int*) – number of points in the polygon (must be 3 or more)
- **n_holes** (*int*) – number of holes
- **tight** (*bool, default=False*) – fits the minimum and maximum points between 0 and 1
- **convex** (*bool, default=True*) – force resulting polygon will be convex (may remove exterior points)

CommandLine: xdoctest -m kwimage.structs.polygon Polygon.random

Example

```
>>> rng = None
>>> n = 4
>>> n_holes = 1
>>> cls = Polygon
>>> self = Polygon.random(n=n, rng=rng, n_holes=n_holes, convex=1)
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.figure(fnum=1, doclf=True)
>>> kwplot.autompl()
>>> self.draw()
```

References

https://gis.stackexchange.com/questions/207731/random-multipolygon https://stackoverflow.com/questions/8997099/random-polygon https://stackoverflow.com/questions/27548363/from-voronoi-tessellation-to-shapely-polygons https://stackoverflow.com/questions/8997099/algorithm-to-generate-random-2d-polygon

_impl(*self*)

to_mask(*self*, *dims=None*)

Convert this polygon to a mask

Todo:

- [ ] currently not efficient

Example

```
>>> from kwimage.structs.polygon import *  # NOQA
>>> self = Polygon.random(n_holes=1).scale(128)
>>> mask = self.to_mask((128, 128))
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.figure(fnum=1, doclf=True)
>>> mask.draw(color='blue')
>>> mask.to_multi_polygon().draw(color='red', alpha=.5)
```

fill(*self*, *image*, *value=1*)

Inplace fill in an image based on this polyon.

_to_cv_countours(*self*)

OpenCV polygon representation, which is a list of points. Holes are implicitly represented. When another polygon is drawn over an existing polyon via cv2.fillPoly

Returns

where each ndarray is of shape [N, 1, 2], where N is the number of points on the boundary, the middle dimension is always 1, and the trailing dimension represents x and y coordinates respectively.

Return type List[ndarray]

to_shapely(*self*)

Example

```
>>> # xdoc: +REQUIRES(module:kwplot)
>>> # xdoc: +REQUIRES(module:shapely)
>>> from kwimage.structs.polygon import *  # NOQA
>>> self = Polygon.random(n_holes=1)
>>> self = self.scale(100)
>>> geom = self.to_shapely()
>>> print('geom = {!r}'.format(geom))
```

classmethod from_shapely(*Polygon*, *geom*)
Convert a shapely polygon to a kwimage.Polygon

Parameters **geom** (*shapely.geometry.polygon.Polygon*) – a shapely polygon

classmethod from_wkt(*Polygon*, *data*)
Convert a WKT string to a kwimage.Polygon

Parameters **data** (*str*) – a WKT polygon string

Example

data = kwimage.Polygon.random().to_shapely().to_wkt() data = 'POLYGON ((0.11 0.61, 0.07 0.588, 0.015 0.50, 0.11 0.61))' self = Polygon.from_wkt(data)

classmethod from_geojson(*MultiPolygon*, *data_geojson*)
Convert a geojson polygon to a kwimage.Polygon

Parameters **data_geojson** (*dict*) – geojson data

Example

```
>>> self = Polygon.random(n_holes=2)
>>> data_geojson = self.to_geojson()
>>> new = Polygon.from_geojson(data_geojson)
```

classmethod from_coco(*cls*, *data*, *dims=None*)
Accepts either new-style or old-style coco polygons

draw_on(*self*, *image*, *color='blue'*, *fill=True*, *border=False*, *alpha=1.0*, *copy=False*)
Rasterizes a polygon on an image. See *draw* for a vectorized matplotlib version.

Parameters

- **image** (*ndarray*) – image to raster polygon on.
- **color** (*str | tuple*) – data coercable to a color
- **fill** (*bool, default=True*) – draw the center mass of the polygon
- **border** (*bool, default=False*) – draw the border of the polygon
- **alpha** (*float, default=1.0*) – polygon transparency (setting alpha < 1 makes this function much slower).
- **copy** (*bool, default=False*) – if False only copies if necessary

Example

```
>>> # xdoc: +REQUIRES(module:kwplot)
>>> from kwimage.structs.polygon import *  # NOQA
>>> self = Polygon.random(n_holes=1).scale(128)
>>> image = np.zeros((128, 128), dtype=np.float32)
>>> image = self.draw_on(image)
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(image, fnum=1)
```

Example

```
>>> import kwimage
>>> color = 'blue'
>>> self = kwimage.Polygon.random(n_holes=1).scale(128)
>>> image = np.zeros((128, 128), dtype=np.float32)
>>> # Test drawong on all channel + dtype combinations
>>> im3 = np.random.rand(128, 128, 3)
>>> im_chans = {
>>>     'im3': im3,
>>>     'im1': kwimage.convert_colorspace(im3, 'rgb', 'gray'),
>>>     'im4': kwimage.convert_colorspace(im3, 'rgb', 'rgba'),
>>> }
>>> inputs = {}
>>> for k, im in im_chans.items():
>>>     inputs[k + '_01'] = (kwimage.ensure_float01(im.copy()), {'alpha':␣
↪None})
>>>     inputs[k + '_255'] = (kwimage.ensure_uint255(im.copy()), {'alpha':␣
↪None})
>>>     inputs[k + '_01_a'] = (kwimage.ensure_float01(im.copy()), {'alpha': 0.
↪5})
>>>     inputs[k + '_255_a'] = (kwimage.ensure_uint255(im.copy()), {'alpha':␣
↪0.5})
>>> outputs = {}
>>> for k, v in inputs.items():
>>>     im, kw = v
>>>     outputs[k] = self.draw_on(im, color=color, **kw)
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.figure(fnum=2, doclf=True)
>>> kwplot.autompl()
>>> pnum_ = kwplot.PlotNums(nCols=2, nRows=len(inputs))
>>> for k in inputs.keys():
>>>     kwplot.imshow(inputs[k][0], fnum=2, pnum=pnum_(), title=k)
>>>     kwplot.imshow(outputs[k], fnum=2, pnum=pnum_(), title=k)
>>> kwplot.show_if_requested()
```

draw(*self*, *color='blue'*, *ax=None*, *alpha=1.0*, *radius=1*, *setlim=False*, *border=False*, *linewidth=2*)
Draws polygon in a matplotlib axes. See *draw_on* for in-memory image modification.

Example

```
>>> # xdoc: +REQUIRES(module:kwplot)
>>> from kwimage.structs.polygon import *  # NOQA
>>> self = Polygon.random(n_holes=1)
>>> self = self.scale(100)
>>> # xdoc: +REQUIRES(--show)
>>> self.draw()
>>> import kwplot
>>> kwplot.autompl()
>>> from matplotlib import pyplot as plt
>>> kwplot.figure(fnum=2)
>>> self.draw(setlim=True)
```

_to_coco(*self*, *style='orig'*)

to_coco(*self*, *style='orig'*)

to_geojson(*self*)
Converts polygon to a geojson structure

to_multi_polygon(*self*)

to_boxes(*self*)

copy(*self*)

clip(*self*, *x_min*, *y_min*, *x_max*, *y_max*, *inplace=False*)
Clip polygon to image boundaries.

Example

```
>>> from kwimage.structs.polygon import *
>>> self = Polygon.random().scale(10).translate(-1)
>>> self2 = self.clip(1, 1, 3, 3)
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> self2.draw(setlim=True)
```

kwimage.structs.polygon.**_order_vertices**(*verts*)

References

https://stackoverflow.com/questions/1709283/how-can-i-sort-a-coordinate-list-for-a-rectangle-counterclockwise

class kwimage.structs.polygon.**MultiPolygon**
Bases: *kwimage.structs._generic.ObjectList*

classmethod random(*self*, *n=3*, *rng=None*, *tight=False*)

to_multi_polygon(*self*)

to_mask(*self*, *dims=None*)
Returns a mask object indication regions occupied by this multipolygon

Example

```
>>> from kwimage.structs.polygon import *  # NOQA
>>> s = 100
>>> self = MultiPolygon.random(rng=0).scale(s)
>>> dims = (s, s)
>>> mask = self.to_mask(dims)
```

```
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.figure(fnum=1, doclf=True)
>>> from matplotlib import pyplot as pl
>>> ax = plt.gca()
>>> ax.set_xlim(0, s)
>>> ax.set_ylim(0, s)
>>> self.draw(color='red', alpha=.4)
>>> mask.draw(color='blue', alpha=.4)
```

classmethod coerce(*cls*, *data*, *dims=None*)
See Mask.coerce

to_shapely(*self*)

Example

```
>>> # xdoc: +REQUIRES(module:kwplot)
>>> # xdoc: +REQUIRES(module:shapely)
>>> from kwimage.structs.polygon import *  # NOQA
>>> self = MultiPolygon.random(rng=0)
>>> geom = self.to_shapely()
>>> print('geom = {!r}'.format(geom))
```

classmethod from_shapely(*MultiPolygon*, *geom*)
Convert a shapely polygon or multipolygon to a kwimage.MultiPolygon

classmethod from_geojson(*MultiPolygon*, *data_geojson*)
Convert a geojson polygon or multipolygon to a kwimage.MultiPolygon

Example

```
>>> import kwimage
>>> orig = kwimage.MultiPolygon.random()
>>> data_geojson = orig.to_geojson()
>>> self = kwimage.MultiPolygon.from_geojson(data_geojson)
```

to_geojson(*self*)
Converts polygon to a geojson structure

classmethod from_coco(*cls*, *data*, *dims=None*)
Accepts either new-style or old-style coco multi-polygons

_to_coco(*self*, *style='orig'*)

to_coco(*self*, *style='orig'*)

Example

```
>>> from kwimage.structs.polygon import *  # NOQA
>>> self = MultiPolygon.random(1, rng=0)
>>> self.to_coco()
```

class kwimage.structs.polygon.**PolygonList**

Bases: *kwimage.structs._generic.ObjectList*

to_polygon_list(*self*)

Package Contents

class kwimage.structs.**Boxes**(*data*, *format=None*, *check=True*)

Bases: kwimage.structs.boxes._BoxConversionMixins, kwimage.structs.boxes._BoxPropertyMixins, kwimage.structs.boxes._BoxTransformMixins, kwimage.structs.boxes._BoxDrawMixins, ubelt.NiceRepr

Converts boxes between different formats as long as the last dimension contains 4 coordinates and the format is specified.

This is a convinience class, and should not not store the data for very long. The general idiom should be create class, convert data, and then get the raw data and let the class be garbage collected. This will help ensure that your code is portable and understandable if this class is not available.

Example

```
>>> # xdoctest: +IGNORE_WHITESPACE
>>> Boxes([25, 30, 15, 10], 'xywh')
<Boxes(xywh, array([25, 30, 15, 10]))>
>>> Boxes([25, 30, 15, 10], 'xywh').to_xywh()
<Boxes(xywh, array([25, 30, 15, 10]))>
>>> Boxes([25, 30, 15, 10], 'xywh').to_cxywh()
<Boxes(cxywh, array([32.5, 35. , 15. , 10. ]))>
>>> Boxes([25, 30, 15, 10], 'xywh').to_tlbr()
<Boxes(tlbr, array([25, 30, 40, 40]))>
>>> Boxes([25, 30, 15, 10], 'xywh').scale(2).to_tlbr()
<Boxes(tlbr, array([50., 60., 80., 80.]))>
>>> Boxes(torch.FloatTensor([[25, 30, 15, 20]]), 'xywh').scale(.1).to_tlbr()
<Boxes(tlbr, tensor([[ 2.5000,  3.0000,  4.0000,  5.0000]]))>
```

Example

```
>>> datas = [
>>>     [1, 2, 3, 4],
>>>     [[1, 2, 3, 4], [4, 5, 6, 7]],
>>>     [[[1, 2, 3, 4], [4, 5, 6, 7]]],
>>> ]
>>> formats = BoxFormat.cannonical
>>> for format1 in formats:
>>>     for data in datas:
>>>         self = box1 = Boxes(data, format1)
>>>         for format2 in formats:
```

(continues on next page)

(continued from previous page)

```
>>>             box2 = box1.toformat(format2)
>>>             back = box2.toformat(format1)
>>>             assert box1 == back
```

device
If the backend is torch returns the data device, otherwise None

__getitem__(*self*, *index*)

__eq__(*self*, *other*)
Tests equality of two Boxes objects

Example

```
>>> box0 = box1 = Boxes([[1, 2, 3, 4]], 'xywh')
>>> box2 = Boxes(box0.data, 'tlbr')
>>> box3 = Boxes([[0, 2, 3, 4]], box0.format)
>>> box4 = Boxes(box0.data, box2.format)
>>> assert box0 == box1
>>> assert not box0 == box2
>>> assert not box2 == box3
>>> assert box2 == box4
```

__len__(*self*)

__nice__(*self*)

__repr__(*self*)

classmethod random(*Boxes*, *num=1*, *scale=1.0*, *format=BoxFormat.XYWH*, *anchors=None*, *anchor_std=1.0 / 6*, *tensor=False*, *rng=None*)
Makes random boxes; typically for testing purposes

Parameters

- **num** (*int*) – number of boxes to generate
- **scale** (*float | Tuple[float, float]*) – size of imgdims
- **format** (*str*) – format of boxes to be created (e.g. tlbr, xywh)
- **anchors** (*ndarray*) – normalized width / heights of anchor boxes to perterb and randomly place. (must be in range 0-1)
- **anchor_std** (*float*) – magnitude of noise applied to anchor shapes
- **tensor** (*bool*) – if True, returns boxes in tensor format
- **rng** (*None | int | RandomState*) – initial random seed

Example

```
>>> # xdoctest: +IGNORE_WHITESPACE
>>> Boxes.random(3, rng=0, scale=100)
<Boxes(xywh,
    array([[54, 54,  6, 17],
           [42, 64,  1, 25],
           [79, 38, 17, 14]]))>
```

(continues on next page)

(continued from previous page)

```
>>> Boxes.random(3, rng=0, scale=100).tensor()
<Boxes(xywh,
    tensor([[ 54,  54,   6,  17],
            [ 42,  64,   1,  25],
            [ 79,  38,  17,  14]]))>
>>> anchors = np.array([[.5, .5], [.3, .3]])
>>> Boxes.random(3, rng=0, scale=100, anchors=anchors)
<Boxes(xywh,
    array([[ 2, 13, 51, 51],
           [32, 51, 32, 36],
           [36, 28, 23, 26]]))>
```

Example

```
>>> # Boxes position/shape within 0-1 space should be uniform.
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> fig = kwplot.figure(fnum=1, doclf=True)
>>> fig.gca().set_xlim(0, 128)
>>> fig.gca().set_ylim(0, 128)
>>> import kwimage
>>> kwimage.Boxes.random(num=10).scale(128).draw()
```

copy (*self*)

classmethod concatenate (*cls*, *boxes*, *axis=0*)

Concatenates multiple boxes together

Parameters

- **boxes** (*Sequence[Boxes]*) – list of boxes to concatenate
- **axis** (*int, default=0*) – axis to stack on

Returns stacked boxes

Return type *Boxes*

Example

```
>>> boxes = [Boxes.random(3) for _ in range(3)]
>>> new = Boxes.concatenate(boxes)
>>> assert len(new) == 9
>>> assert np.all(new.data[3:6] == boxes[1].data)
```

Example

```
>>> boxes = [Boxes.random(3) for _ in range(3)]
>>> boxes[0].data = boxes[0].data[0]
>>> boxes[1].data = boxes[0].data[0:0]
>>> new = Boxes.concatenate(boxes)
>>> assert len(new) == 4
```

(continues on next page)

(continued from previous page)

```
>>> new = Boxes.concatenate([b.tensor() for b in boxes])
>>> assert len(new) == 4
```

compress (*self*, *flags*, *axis=0*, *inplace=False*)

Filters boxes based on a boolean criterion

Parameters

- **flags** (*ArrayLike[bool]*) – true for items to be kept
- **axis** (*int*) – you usually want this to be 0
- **inplace** (*bool*) – if True, modifies this object

Example

```
>>> self = Boxes([[25, 30, 15, 10]], 'tlbr')
>>> self.compress([True])
<Boxes(tlbr, array([[25, 30, 15, 10]]))>
>>> self.compress([False])
<Boxes(tlbr, array([], shape=(0, 4), dtype=int64))>
```

take (*self*, *idxs*, *axis=0*, *inplace=False*)

Takes a subset of items at specific indices

Parameters

- **indices** (*ArrayLike[int]*) – indexes of items to take
- **axis** (*int*) – you usually want this to be 0
- **inplace** (*bool*) – if True, modifies this object

Example

```
>>> self = Boxes([[25, 30, 15, 10]], 'tlbr')
>>> self.take([0])
<Boxes(tlbr, array([[25, 30, 15, 10]]))>
>>> self.take([])
<Boxes(tlbr, array([], shape=(0, 4), dtype=int64))>
```

is_tensor (*self*)

is the backend fueled by torch?

is_numpy (*self*)

is the backend fueled by numpy?

_impl (*self*)

returns the kwarray.ArrayAPI implementation for the data

Example

```
>>> assert Boxes.random().numpy()._impl.is_numpy
>>> assert Boxes.random().tensor()._impl.is_tensor
```

astype (*self*, *dtype*)
Changes the type of the internal array used to represent the boxes

Notes

this operation is not inplace

Example

```
>>> # xdoctest: +IGNORE_WHITESPACE
>>> Boxes.random(3, 100, rng=0).tensor().astype('int32')
<Boxes(xywh,
    tensor([[54, 54,  6, 17],
            [42, 64,  1, 25],
            [79, 38, 17, 14]], dtype=torch.int32))>
>>> Boxes.random(3, 100, rng=0).numpy().astype('int32')
<Boxes(xywh,
    array([[54, 54,  6, 17],
           [42, 64,  1, 25],
           [79, 38, 17, 14]], dtype=int32))>
>>> Boxes.random(3, 100, rng=0).tensor().astype('float32')
>>> Boxes.random(3, 100, rng=0).numpy().astype('float32')
```

numpy (*self*)
Converts tensors to numpy. Does not change memory if possible.

Example

```
>>> self = Boxes.random(3).tensor()
>>> newself = self.numpy()
>>> self.data[0, 0] = 0
>>> assert newself.data[0, 0] == 0
>>> self.data[0, 0] = 1
>>> assert self.data[0, 0] == 1
```

tensor (*self*, *device=ub.NoParam*)
Converts numpy to tensors. Does not change memory if possible.

Example

```
>>> self = Boxes.random(3)
>>> newself = self.tensor()
>>> self.data[0, 0] = 0
>>> assert newself.data[0, 0] == 0
>>> self.data[0, 0] = 1
>>> assert self.data[0, 0] == 1
```

ious (*self*, *other*, *bias=0*, *impl='auto'*, *mode=None*)
Compute IOUs (intersection area over union area) between these boxes and another set of boxes.

Parameters

- **other** (*Boxes*) – boxes to compare IoUs against

- **bias** (*int, default=0*) – either 0 or 1, does TL=BR have area of 0 or 1?
- **impl** (*str, default='auto'*) – code to specify implementation used to ious. Can be either torch, py, c, or auto. Efficiency and the exact result will vary by implementation, but they will always be close. Some implementations only accept certain data types (e.g. impl='c', only accepts float32 numpy arrays). See ~/code/kwimage/dev/bench_bbox.py for benchmark details. On my system the torch impl was fastest (when the data was on the GPU).
- **mode** – depricated, use impl

Examples

```
>>> self = Boxes(np.array([[ 0,  0, 10, 10],
>>>                        [10,  0, 20, 10],
>>>                        [20,  0, 30, 10]]), 'tlbr')
>>> other = Boxes(np.array([6, 2, 20, 10]), 'tlbr')
>>> overlaps = self.ious(other, bias=1).round(2)
>>> assert np.all(np.isclose(overlaps, [0.21, 0.63, 0.04])), repr(overlaps)
```

Examples

```
>>> # xdoctest: +IGNORE_WHITESPACE
>>> Boxes(np.empty(0), 'xywh').ious(Boxes(np.empty(4), 'xywh')).shape
(0,)
>>> #Boxes(np.empty(4), 'xywh').ious(Boxes(np.empty(0), 'xywh')).shape
>>> Boxes(np.empty((0, 4)), 'xywh').ious(Boxes(np.empty((0, 4)), 'xywh')).
↪shape
(0, 0)
>>> Boxes(np.empty((1, 4)), 'xywh').ious(Boxes(np.empty((0, 4)), 'xywh')).
↪shape
(1, 0)
>>> Boxes(np.empty((0, 4)), 'xywh').ious(Boxes(np.empty((1, 4)), 'xywh')).
↪shape
(0, 1)
```

Examples

```
>>> formats = BoxFormat.cannonical
>>> istensors = [False, True]
>>> results = {}
>>> for format in formats:
>>>     for tensor in istensors:
>>>         boxes1 = Boxes.random(5, scale=10.0, rng=0, format=format,␣
↪tensor=tensor)
>>>         boxes2 = Boxes.random(7, scale=10.0, rng=1, format=format,␣
↪tensor=tensor)
>>>         ious = boxes1.ious(boxes2)
>>>         results[(format, tensor)] = ious
>>> results = {k: v.numpy() if torch.is_tensor(v) else v for k, v in results.
↪items() }
>>> results = {k: v.tolist() for k, v in results.items()}
>>> print(ub.repr2(results, sk=True, precision=3, nl=2))
```

(continues on next page)

(continued from previous page)

```
>>> from functools import partial
>>> assert ub.allsame(results.values(), partial(np.allclose, atol=1e-07))
```

isect_area(*self*, *other*, *bias=0*)

Intersection part of intersection over union computation

Examples

```
>>> # xdoctest: +IGNORE_WHITESPACE
>>> self = Boxes.random(5, scale=10.0, rng=0, format='tlbr')
>>> other = Boxes.random(3, scale=10.0, rng=1, format='tlbr')
>>> isect = self.isect_area(other, bias=0)
>>> ious_v1 = isect / ((self.area + other.area.T) - isect)
>>> ious_v2 = self.ious(other, bias=0)
>>> assert np.allclose(ious_v1, ious_v2)
```

intersection(*self*, *other*)

Pairwise intersection between two sets of Boxes

Returns intersected boxes

Return type *Boxes*

Examples

```
>>> # xdoctest: +IGNORE_WHITESPACE
>>> from kwimage.structs.boxes import *  # NOQA
>>> self = Boxes.random(5, rng=0).scale(10.)
>>> other = self.translate(1)
>>> new = self.intersection(other)
>>> new_area = np.nan_to_num(new.area).ravel()
>>> alt_area = np.diag(self.isect_area(other))
>>> close = np.isclose(new_area, alt_area)
>>> assert np.all(close)
```

view(*self*, **shape*)

Passthrough method to view or reshape

Example

```
>>> self = Boxes.random(6, scale=10.0, rng=0, format='xywh').tensor()
>>> assert list(self.view(3, 2, 4).data.shape) == [3, 2, 4]
>>> self = Boxes.random(6, scale=10.0, rng=0, format='tlbr').tensor()
>>> assert list(self.view(3, 2, 4).data.shape) == [3, 2, 4]
```

class kwimage.structs.**Coords**(*data=None*, *meta=None*)

Bases: *kwimage.structs._generic.Spatial*, ubelt.NiceRepr

This stores arbitrary sparse n-dimensional coordinate geometry.

You can specify data, but you don't have to. We dont care what it is, we just warp it.

Note: This class was designed to hold coordinates in r/c format, but in general this class is anostic to dimension ordering as long as you are consistent. However, there are two places where this matters:

(1) drawing and (2) gdal/imgaug-warping. In these places we will assume x/y for legacy reasons. This may change in the future.

CommandLine: xdoctest -m kwimage.structs.coords Coords

Example

```
>>> from kwimage.structs.coords import *  # NOQA
>>> import kwarray
>>> rng = kwarray.ensure_rng(0)
>>> self = Coords.random(num=4, dim=3, rng=rng)
>>> matrix = rng.rand(4, 4)
>>> self.warp(matrix)
>>> self.translate(3, inplace=True)
>>> self.translate(3, inplace=True)
>>> self.scale(2)
>>> self.tensor()
>>> # self.tensor(device=0)
>>> self.tensor().tensor().numpy().numpy()
>>> self.numpy()
>>> #self.draw_on()
```

__repr__

dtype

dim

shape

device
If the backend is torch returns the data device, otherwise None

_impl
Returns the internal tensor/numpy ArrayAPI implementation

__nice__(*self*)

__len__(*self*)

copy(*self*)

classmethod random(*Coords*, *num=1*, *dim=2*, *rng=None*, *meta=None*)
Makes random coordinates; typically for testing purposes

is_numpy(*self*)

is_tensor(*self*)

compress(*self*, *flags*, *axis=0*, *inplace=False*)
Filters items based on a boolean criterion

Parameters

- **flags** (*ArrayLike[bool]*) – true for items to be kept
- **axis** (*int*) – you usually want this to be 0

- **inplace** (*bool, default=False*) – if True, modifies this object

Returns filtered coords

Return type *Coords*

Example

```
>>> from kwimage.structs.coords import *  # NOQA
>>> self = Coords.random(10, rng=0)
>>> self.compress([True] * len(self))
>>> self.compress([False] * len(self))
<Coords(data=array([], shape=(0, 2), dtype=float64))>
>>> self = self.tensor()
>>> self.compress([True] * len(self))
>>> self.compress([False] * len(self))
```

take(*self*, *indices*, *axis=0*, *inplace=False*)

Takes a subset of items at specific indices

Parameters

- **indices** (*ArrayLike[int]*) – indexes of items to take
- **axis** (*int*) – you usually want this to be 0
- **inplace** (*bool, default=False*) – if True, modifies this object

Returns filtered coords

Return type *Coords*

Example

```
>>> self = Coords(np.array([[25, 30, 15, 10]]))
>>> self.take([0])
<Coords(data=array([[25, 30, 15, 10]]))>
>>> self.take([])
<Coords(data=array([], shape=(0, 4), dtype=int64))>
```

astype(*self*, *dtype*, *inplace=False*)

Changes the data type

Parameters

- **dtype** – new type
- **inplace** (*bool, default=False*) – if True, modifies this object

view(*self*, **shape*)

Passthrough method to view or reshape

Parameters ***shape** – new shape of the data

Example

```
>>> self = Coords.random(6, dim=4).tensor()
>>> assert list(self.view(3, 2, 4).data.shape) == [3, 2, 4]
>>> self = Coords.random(6, dim=4).numpy()
>>> assert list(self.view(3, 2, 4).data.shape) == [3, 2, 4]
```

classmethod concatenate(*cls*, *coords*, *axis=0*)

Concatenates lists of coordinates together

Parameters

- **coords** (*Sequence[Coords]*) – list of coords to concatenate
- **axis** (*int, default=0*) – axis to stack on

Returns stacked coords

Return type *Coords*

CommandLine: xdoctest -m kwimage.structs.coords Coords.concatenate

Example

```
>>> coords = [Coords.random(3) for _ in range(3)]
>>> new = Coords.concatenate(coords)
>>> assert len(new) == 9
>>> assert np.all(new.data[3:6] == coords[1].data)
```

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

Converts numpy to tensors. Does not change memory if possible.

Example

```
>>> self = Coords.random(3).numpy()
>>> newself = self.tensor()
>>> self.data[0, 0] = 0
>>> assert newself.data[0, 0] == 0
>>> self.data[0, 0] = 1
>>> assert self.data[0, 0] == 1
```

numpy(*self*)

Converts tensors to numpy. Does not change memory if possible.

Example

```
>>> self = Coords.random(3).tensor()
>>> newself = self.numpy()
>>> self.data[0, 0] = 0
>>> assert newself.data[0, 0] == 0
>>> self.data[0, 0] = 1
>>> assert self.data[0, 0] == 1
```

warp(*self*, *transform*, *input_dims=None*, *output_dims=None*, *inplace=False*)

Generalized coordinate transform.

Parameters

- **transform** (*GeometricTransform | ArrayLike | Augmenter*) – scikit-image tranform, a transformation matrix, or an imgaug Augmenter.
- **input_dims** (*Tuple*) – shape of the image these objects correspond to (only needed / used when transform is an imgaug augmenter)
- **output_dims** (*Tuple*) – unused in non-raster structures, only exists for compatibility.
- **inplace** (*bool, default=False*) – if True, modifies data inplace

Notes

Let D = self.dims

transformation matrices can be either:

- (D + 1) x (D + 1) # for homog
- D x D # for scale / rotate
- D x (D + 1) # for affine

Example

```
>>> from kwimage.structs.coords import *  # NOQA
>>> self = Coords.random(10, rng=0)
>>> transform = skimage.transform.AffineTransform(scale=(2, 2))
>>> new = self.warp(transform)
>>> assert np.all(new.data == self.scale(2).data)
```

Doctest:

```
>>> self = Coords.random(10, rng=0)
>>> assert np.all(self.warp(np.eye(3)).data == self.data)
>>> assert np.all(self.warp(np.eye(2)).data == self.data)
```

Ignore:

```
>>> # xdoctest: +SKIP
>>> # xdoctest: +REQUIRES(module:osr)
>>> wgs84_crs = osr.SpatialReference()
>>> wgs84_crs.ImportFromEPSG(4326)
>>> transform = osr.CoordinateTransformation(wgs84_crs, wgs84_crs)
>>> self = Coords.random(10, rng=0)
>>> new = self.warp(transform)
>>> assert np.all(new.data == self.data)
```

_warp_imgaug (*self*, *augmenter*, *input_dims*, *inplace=False*)

Warps by applying an augmenter from the imgaug library

Note: We are assuming you are using X/Y coordinates here.

Parameters

- **augmenter** (*imgaug.augmenters.Augmenter*)

- **input_dims** (*Tuple*) – h/w of the input image
- **inplace** (*bool, default=False*) – if True, modifies data inplace

CommandLine: xdoctest -m ~/code/kwimage/kwimage/structs/coords.py Coords._warp_imgaug

Example

```
>>> # xdoctest: +REQUIRES(module:imgaug)
>>> from kwimage.structs.coords import *  # NOQA
>>> import imgaug
>>> input_dims = (10, 10)
>>> self = Coords.random(10).scale(input_dims)
>>> augmenter = imgaug.augmenters.Fliplr(p=1)
>>> new = self._warp_imgaug(augmenter, input_dims)
>>> # y coordinate should not change
>>> assert np.allclose(self.data[:, 1], new.data[:, 1])
>>> assert np.allclose(input_dims[0] - self.data[:, 0], new.data[:, 0])
```

```
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.figure(fnum=1, doclf=True)
>>> from matplotlib import pyplot as pl
>>> ax = plt.gca()
>>> ax.set_xlim(0, input_dims[0])
>>> ax.set_ylim(0, input_dims[1])
>>> self.draw(color='red', alpha=.4, radius=0.1)
>>> new.draw(color='blue', alpha=.4, radius=0.1)
```

Example

```
>>> # xdoctest: +REQUIRES(module:imgaug)
>>> from kwimage.structs.coords import *  # NOQA
>>> import imgaug
>>> input_dims = (32, 32)
>>> inplace = 0
>>> self = Coords.random(1000, rng=142).scale(input_dims).scale(.8)
>>> self.data = self.data.astype(np.int32).astype(np.float32)
>>> augmenter = imgaug.augmenters.CropAndPad(px=(-4, 4), keep_size=1).to_
↪deterministic()
>>> new = self._warp_imgaug(augmenter, input_dims)
>>> # Change should be linear
>>> norm1 = (self.data - self.data.min(axis=0)) / (self.data.max(axis=0) -␣
↪self.data.min(axis=0))
>>> norm2 = (new.data - new.data.min(axis=0)) / (new.data.max(axis=0) - new.
↪data.min(axis=0))
>>> diff = norm1 - norm2
>>> assert np.allclose(diff, 0, atol=1e-6, rtol=1e-4)
>>> #assert np.allclose(self.data[:, 1], new.data[:, 1])
>>> #assert np.allclose(input_dims[0] - self.data[:, 0], new.data[:, 0])
>>> # xdoc: +REQUIRES(--show)
>>> import kwimage
>>> im = kwimage.imresize(kwimage.grab_test_image(), dsize=input_dims[::-1])
```

(continues on next page)

(continued from previous page)

```
>>> new_im = augmenter.augment_image(im)
>>> import kwplot
>>> plt = kwplot.autoplt()
>>> kwplot.figure(fnum=1, doclf=True)
>>> kwplot.imshow(im, pnum=(1, 2, 1), fnum=1)
>>> self.draw(color='red', alpha=.8, radius=0.5)
>>> kwplot.imshow(new_im, pnum=(1, 2, 2), fnum=1)
>>> new.draw(color='blue', alpha=.8, radius=0.5, coord_axes=[1, 0])
```

to_imgaug(*self*, *input_dims*)

Example

```
>>> # xdoctest: +REQUIRES(module:imgaug)
>>> from kwimage.structs.coords import *  # NOQA
>>> self = Coords.random(10)
>>> input_dims = (10, 10)
>>> kpoi = self.to_imgaug(input_dims)
>>> new = Coords.from_imgaug(kpoi)
>>> assert np.allclose(new.data, self.data)
```

classmethod from_imgaug(*cls*, *kpoi*)

scale(*self*, *factor*, *output_dims=None*, *inplace=False*)

Scale coordinates by a factor

Parameters

- **factor** (*float or Tuple[float, float]*) – scale factor as either a scalar or per-dimension tuple.
- **output_dims** (*Tuple*) – unused in non-raster spatial structures

Example

```
>>> from kwimage.structs.coords import *  # NOQA
>>> self = Coords.random(10, rng=0)
>>> new = self.scale(10)
>>> assert new.data.max() <= 10
```

```
>>> self = Coords.random(10, rng=0)
>>> self.data = (self.data * 10).astype(np.int)
>>> new = self.scale(10)
>>> assert new.data.dtype.kind == 'i'
>>> new = self.scale(10.0)
>>> assert new.data.dtype.kind == 'f'
```

translate(*self*, *offset*, *output_dims=None*, *inplace=False*)

Shift the coordinates

Parameters

- **offset** (*float or Tuple[float]*) – transation offset as either a scalar or a per-dimension tuple.
- **output_dims** (*Tuple*) – unused in non-raster spatial structures

Example

```
>>> from kwimage.structs.coords import *  # NOQA
>>> self = Coords.random(10, dim=3, rng=0)
>>> new = self.translate(10)
>>> assert new.data.min() >= 10
>>> assert new.data.max() <= 11
>>> Coords.random(3, dim=3, rng=0)
>>> Coords.random(3, dim=3, rng=0).translate((1, 2, 3))
```

fill (*self*, *image*, *value*, *coord_axes=None*, *interp='bilinear'*)
Sets sub-coordinate locations in a grid to a particular value

Parameters **coord_axes** (*Tuple*) – specify which image axes each coordinate dim corresponds to. For 2D images, if you are storing r/c data, set to [0,1], if you are storing x/y data, set to [1,0].

draw_on (*self, image=None, fill_value=1, coord_axes=[1, 0], interp='bilinear'*)

Note: unlike other methods, the defaults assume x/y internal data

---

Parameters **coord_axes** (*Tuple*) – specify which image axes each coordinate dim corresponds to. For 2D images, if you are storing r/c data, set to [0,1], if you are storing x/y data, set to [1,0].

In other words the i-th entry in coord_axes specifies which row-major spatial dimension the i-th column of a coordinate corresponds to. The index is the coordinate dimension and the value is the axes dimension.

Example

```
>>> # xdoc: +REQUIRES(module:kwplot)
>>> from kwimage.structs.coords import *  # NOQA
>>> s = 256
>>> self = Coords.random(10, meta={'shape': (s, s)}).scale(s)
>>> self.data[0] = [10, 10]
>>> self.data[1] = [20, 40]
>>> image = np.zeros((s, s))
>>> fill_value = 1
>>> image = self.draw_on(image, fill_value, coord_axes=[1, 0], interp=
↪'bilinear')
>>> # image = self.draw_on(image, fill_value, coord_axes=[0, 1], interp=
↪'nearest')
>>> # image = self.draw_on(image, fill_value, coord_axes=[1, 0], interp=
↪'bilinear')
>>> # image = self.draw_on(image, fill_value, coord_axes=[1, 0], interp=
↪'nearest')
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.figure(fnum=1, doclf=True)
>>> kwplot.autompl()
>>> kwplot.imshow(image)
>>> self.draw(radius=3, alpha=.5, coord_axes=[1, 0])
```

draw(*self, color='blue', ax=None, alpha=None, coord_axes=[1, 0], radius=1*)

Note: unlike other methods, the defaults assume x/y internal data

Parameters **coord_axes** (*Tuple*) – specify which image axes each coordinate dim corresponds to. For 2D images,

if you are storing r/c data, set to [0,1], if you are storing x/y data, set to [1,0].

Example

```
>>> # xdoc: +REQUIRES(module:kwplot)
>>> from kwimage.structs.coords import *  # NOQA
>>> self = Coords.random(10)
>>> # xdoc: +REQUIRES(--show)
>>> self.draw(radius=3.0)
>>> import kwplot
>>> kwplot.autompl()
>>> self.draw(radius=3.0)
```

class kwimage.structs.**Detections**(*data=None, meta=None, datakeys=None, metakeys=None, checks=True, **kwargs*)

Bases: ubelt.NiceRepr, *kwimage.structs.detections._DetAlgoMixin*, *kwimage.structs.detections._DetDrawMixin*

Container for holding and manipulating multiple detections.

Variables

- **data** (*Dict*) – dictionary containing corresponding lists. The length of each list is the number of detections. This contains the bounding boxes, confidence scores, and class indices. Details of the most common keys and types are as follows:

  boxes (kwimage.Boxes[ArrayLike]): multiple bounding boxes scores (ArrayLike): associated scores class_idxs (ArrayLike): associated class indices

  Additional custom keys may be specified as long as (a) the values are array-like and the first axis corresponds to the standard data values and (b) are custom keys are listed in the *datakeys* kwargs when constructing the Detections.

- **meta** (*Dict*) – This contains contextual information about the detections. This includes the class names, which can be indexed into via the class indexes.

Example

```
>>> self = Detections.random(10)
>>> other = Detections(self)
>>> assert other.data == self.data
>>> assert other.data is self.data, 'try not to copy unless necessary'
```

__datakeys__ = ['boxes', 'scores', 'class_idxs', 'probs', 'weights', 'keypoints', 'segm

__metakeys__ = ['classes']

boxes

class_idxs

scores
typically only populated for predicted detections

probs
typically only populated for predicted detections

weights
typically only populated for groundtruth detections

classes

device
If the backend is torch returns the data device, otherwise None

dtype

__nice__(*self*)

__len__(*self*)

copy(*self*)
Returns a deep copy of this Detections object

classmethod from_coco_annots(*cls*, *anns*, *cats=None*, *classes=None*, *kp_classes=None*, *shape=None*, *dset=None*)
Create a Detections object from a list of coco-like annotations.

Parameters

- **anns** (*List[Dict]*) – list of coco-like annotation objects
- **dset** (*CocoDataset*) – if specified, cats, classes, and kp_classes can are ignored.
- **cats** (*List[Dict]*) – coco-format category information. Used only if *dset* is not specified.
- **classes** (*ndsampler.CategoryTree*) – category tree with coco class info. Used only if *dset* is not specified.
- **kp_classes** (*ndsampler.CategoryTree*) – keypoint category tree with coco keypoint class info. Used only if *dset* is not specified.
- **shape** (*tuple*) – shape of parent image

Returns a detections object

Return type *Detections*

Example

```
>>> from kwimage.structs.detections import *  # NOQA
>>> # xdoctest: +REQUIRES(--module:ndsampler)
>>> anns = [{
>>>     'id': 0,
>>>     'image_id': 1,
>>>     'category_id': 2,
>>>     'bbox': [2, 3, 10, 10],
>>>     'keypoints': [4.5, 4.5, 2],
>>>     'segmentation': {
>>>         'counts': '_11a04M2O0O20N101N3L_5',
>>>         'size': [20, 20],
>>>     },
```

(continues on next page)

(continued from previous page)

```
>>> }]
>>> dataset = {
>>>     'images': [],
>>>     'annotations': [],
>>>     'categories': [
>>>         {'id': 0, 'name': 'background'},
>>>         {'id': 2, 'name': 'class1', 'keypoints': ['spot']}
>>>     ]
>>> }
>>> #import ndsampler
>>> #dset = ndsampler.CocoDataset(dataset)
>>> cats = dataset['categories']
>>> dets = Detections.from_coco_annots(anns, cats)
```

Example

```
>>> import kwimage
>>> # xdoctest: +REQUIRES(--module:ndsampler)
>>> import ndsampler
>>> sampler = ndsampler.CocoSampler.demo('photos')
>>> iminfo, anns = sampler.load_image_with_annots(1)
>>> shape = iminfo['imdata'].shape[0:2]
>>> kp_classes = sampler.dset.keypoint_categories()
>>> dets = kwimage.Detections.from_coco_annots(
>>>     anns, sampler.dset.dataset['categories'], sampler.catgraph,
>>>     kp_classes, shape=shape)
```

to_coco (*self*, *cname_to_cat=None*, *style='orig'*)

Converts this set of detections into coco-like annotation dictionaries.

Notes

Not all aspects of the MS-COCO format can be accurately represented, so some liberties are taken. The MS-COCO standard defines that annotations should specifiy a category_id field, but in some cases this information is not available so we will populate a 'category_name' field if possible and in the worst case fall back to 'category_index'.

Additionally, detections may contain additional information beyond the MS-COCO standard, and this information (e.g. weight, prob, score) is added as forign fields.

Parameters

- **cname_to_cat** – currently ignored.
- **style** (*str*) – either orig (for the original coco format) or new for the more general ndsampler-style coco format.

Yields *dict* – coco-like annotation structures

Example

```
>>> # xdoctest: +REQUIRES(module:ndsampler)
>>> from kwimage.structs.detections import *
```

(continues on next page)

(continued from previous page)

```
>>> self = Detections.demo()[0]
>>> cname_to_cat = None
>>> list(self.to_coco())
```

num_boxes (*self*)

warp (*self*, *transform*, *input_dims=None*, *output_dims=None*, *inplace=False*)

Spatially warp the detections.

Example

```
>>> import skimage
>>> transform = skimage.transform.AffineTransform(scale=(2, 3),␣
↪translation=(4, 5))
>>> self = Detections.random(2)
>>> new = self.warp(transform)
>>> assert new.boxes == self.boxes.warp(transform)
>>> assert new != self
```

scale (*self*, *factor*, *output_dims=None*, *inplace=False*)

Spatially warp the detections.

Example

```
>>> import skimage
>>> transform = skimage.transform.AffineTransform(scale=(2, 3),␣
↪translation=(4, 5))
>>> self = Detections.random(2)
>>> new = self.warp(transform)
>>> assert new.boxes == self.boxes.warp(transform)
>>> assert new != self
```

translate (*self*, *offset*, *output_dims=None*, *inplace=False*)

Spatially warp the detections.

Example

```
>>> import skimage
>>> self = Detections.random(2)
>>> new = self.translate(10)
```

classmethod concatenate (*cls*, *dets*)

Parameters **boxes** (*Sequence[Detections]*) – list of detections to concatenate

Returns stacked detections

Return type *Detections*

Example

```
>>> self = Detections.random(2)
>>> other = Detections.random(3)
>>> dets = [self, other]
>>> new = Detections.concatenate(dets)
>>> assert new.num_boxes() == 5
```

```
>>> self = Detections.random(2, segmentations=True)
>>> other = Detections.random(3, segmentations=True)
>>> dets = [self, other]
>>> new = Detections.concatenate(dets)
>>> assert new.num_boxes() == 5
```

argsort (*self*, *reverse=True*)

Sorts detection indices by descending (or ascending) scores

Returns sorted indices

Return type ndarray[int]

sort (*self*, *reverse=True*)

Sorts detections by descending (or ascending) scores

Returns sorted copy of self

Return type *kwimage.structs.Detections*

compress (*self*, *flags*, *axis=0*)

Returns a subset where corresponding locations are True.

Parameters **flags** (*ndarray[bool]*) – mask marking selected items

Returns subset of self

Return type *kwimage.structs.Detections*

CommandLine: xdoctest -m kwimage.structs.detections Detections.compress

Example

```
>>> import kwimage
>>> dets = kwimage.Detections.random(keypoints='dense')
>>> flags = np.random.rand(len(dets)) > 0.5
>>> subset = dets.compress(flags)
>>> assert len(subset) == flags.sum()
>>> subset = dets.tensor().compress(flags)
>>> assert len(subset) == flags.sum()
```

z = dets.tensor().data[‘keypoints’].data[‘xy’] z.compress(flags) ub.map_vals(lambda x: x.shape, dets.data) ub.map_vals(lambda x: x.shape, subset.data)

take (*self*, *indices*, *axis=0*)

Returns a subset specified by indices

Parameters **indices** (*ndarray[int]*) – indices to select

Returns subset of self

Return type *kwimage.structs.Detections*

Example

```
>>> import kwimage
>>> dets = kwimage.Detections(boxes=kwimage.Boxes.random(10))
>>> subset = dets.take([2, 3, 5, 7])
>>> assert len(subset) == 4
>>> subset = dets.tensor().take([2, 3, 5, 7])
>>> assert len(subset) == 4
```

__getitem__(*self*, *index*)
Fancy slicing / subset / indexing.

Note: scalar indices are always coerced into index lists of length 1.

Example

```
>>> import kwimage
>>> import kwarray
>>> dets = kwimage.Detections(boxes=kwimage.Boxes.random(10))
>>> indices = [2, 3, 5, 7]
>>> flags = kwarray.boolmask(indices, len(dets))
>>> assert dets[flags].data == dets[indices].data
```

is_tensor(*self*)
is the backend fueled by torch?

is_numpy(*self*)
is the backend fueled by numpy?

numpy(*self*)
Converts tensors to numpy. Does not change memory if possible.

Example

```
>>> self = Detections.random(3).tensor()
>>> newself = self.numpy()
>>> self.scores[0] = 0
>>> assert newself.scores[0] == 0
>>> self.scores[0] = 1
>>> assert self.scores[0] == 1
>>> self.numpy().numpy()
```

tensor(*self*, *device=ub.NoParam*)
Converts numpy to tensors. Does not change memory if possible.

Example

```
>>> from kwimage.structs.detections import *
>>> self = Detections.random(3)
>>> newself = self.tensor()
>>> self.scores[0] = 0
>>> assert newself.scores[0] == 0
>>> self.scores[0] = 1
```

(continues on next page)

(continued from previous page)

```
>>> assert self.scores[0] == 1
>>> self.tensor().tensor()
```

classmethod demo(*Detections*)

classmethod random(*cls*, *num=10*, *scale=1.0*, *rng=None*, *classes=3*, *keypoints=False*, *tensor=False*, *segmentations=False*)

Creates dummy data, suitable for use in tests and benchmarks

Parameters

- **num** (*int*) – number of boxes
- **scale** (*float | tuple, default=1.0*) – bounding image size
- **classes** (*int | Sequence*) – list of class labels or number of classes
- **tensor** (*bool, default=False*) – determines backend
- **rng** (*np.random.RandomState*) – random state

Example

```
>>> import kwimage
>>> dets = kwimage.Detections.random(keypoints='jagged')
>>> dets.data['keypoints'].data[0].data
>>> dets.data['keypoints'].meta
>>> dets = kwimage.Detections.random(keypoints='dense')
>>> dets = kwimage.Detections.random(keypoints='dense', segmentations=True).
↪scale(1000)
>>> # xdoctest:+REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> dets.draw(setlim=True)
```

Example

```
>>> # Boxes position/shape within 0-1 space should be uniform.
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> fig = kwplot.figure(fnum=1, doclf=True)
>>> fig.gca().set_xlim(0, 128)
>>> fig.gca().set_ylim(0, 128)
>>> import kwimage
>>> kwimage.Detections.random(num=10, segmentations=True).scale(128).draw()
```

class kwimage.structs.**Heatmap**(*data=None*, *meta=None*, ***kwargs*)

Bases: *kwimage.structs._generic.Spatial*, *kwimage.structs.heatmap._HeatmapDrawMixin*, *kwimage.structs.heatmap._HeatmapWarpMixin*, *kwimage.structs.heatmap._HeatmapAlgoMixin*

Keeps track of a downscaled heatmap and how to transform it to overlay the original input image. Heatmaps generally are used to estimate class probabilites at each pixel. This data struction additionally contains logic to augment pixel with offset (dydx) and scale (diamter) information.

Variables

- **data** (*Dict[str, object]*) – dictionary containing spatially aligned heatmap data. Valid keys are as follows.

  class_probs (ArrayLike[C, H, W] | ArrayLike[C, D, H, W]): A probability map for each class. C is the number of classes.

  offset (ArrayLike[2, H, W] | ArrayLike[3, D, H, W], optional): object center position offset in y,x / t,y,x coordinates

  diamter (ArrayLike[2, H, W] | ArrayLike[3, D, H, W], optional): object bounding box sizes in h,w / d,h,w coordinates

  keypoints (ArrayLike[2, K, H, W] | ArrayLike[3, K, D, H, W], optional): y/x offsets for K different keypoint classes

- **data** – dictionary containing miscellanious metadata about the heatmap data. Valid keys are as follows.

  img_dims (Tuple[H, W] | Tuple[D, H, W]): original image dimension

  tf_data_to_image (skimage.transform._geometric.GeometricTransform): transformation matrix (typically similarity or affine) that projects the given1.8719898042840075, heatmap onto the image dimensions such that the image and heatmap are spatially aligned.

  classes (List[str] | ndsampler.CategoryTree): information about which index in *data['class_probs']* corresponds to which semantic class.

- ****kwargs** – any key that is accepted by the *data* or *meta* dictionaries can be specified as a keyword argument to this class and it will be properly placed in the appropriate internal dictionary.

CommandLine: xdoctest -m ~/code/kwimage/kwimage/structs/heatmap.py Heatmap –show

Example

```
>>> import kwimage
>>> class_probs = kwimage.grab_test_image(dsize=(32, 32), space='gray')[None, ] /␣
↪255.0
>>> img_dims = (220, 220)
>>> tf_data_to_img = skimage.transform.AffineTransform(translation=(-18, -18),␣
↪scale=(8, 8))
>>> self = Heatmap(class_probs=class_probs, img_dims=img_dims,
>>>                tf_data_to_img=tf_data_to_img)
>>> aligned = self.upscale()
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(aligned[0])
>>> kwplot.show_if_requested()
```

__datakeys__ = ['class_probs', 'offset', 'diameter', 'keypoints', 'class_idx', 'class_

__metakeys__ = ['img_dims', 'tf_data_to_img', 'classes', 'kp_classes']

__spatialkeys__ = ['offset', 'diameter', 'keypoints']

shape

bounds

dims
space-time dimensions of this heatmap

_impl
Returns the internal tensor/numpy ArrayAPI implementation

Returns kwarray.ArrayAPI

class_probs

offset

diameter

img_dims

tf_data_to_img

classes

__nice__(*self*)

__getitem__(*self*, *index*)

__len__(*self*)

is_numpy(*self*)

is_tensor(*self*)

classmethod random(*cls*, *dims=(10, 10)*, *classes=3*, *diameter=True*, *offset=True*, *keypoints=False*, *img_dims=None*, *dets=None*, *nblips=10*, *noise=0.0*, *rng=None*)
Creates dummy data, suitable for use in tests and benchmarks

Parameters

- **dims** (*Tuple*) – dimensions of the heatmap
- **img_dims** (*Tuple*) – dimensions of the image the heatmap corresponds to

Example

```
>>> from kwimage.structs.heatmap import *  # NOQA
>>> self = Heatmap.random((128, 128), img_dims=(200, 200),
>>>     classes=3, nblips=10, rng=0, noise=0.1)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(self.colorize(0, imgspace=0), fnum=1, pnum=(1, 4, 1),␣
↪doclf=1)
>>> kwplot.imshow(self.colorize(1, imgspace=0), fnum=1, pnum=(1, 4, 2))
>>> kwplot.imshow(self.colorize(2, imgspace=0), fnum=1, pnum=(1, 4, 3))
>>> kwplot.imshow(self.colorize(3, imgspace=0), fnum=1, pnum=(1, 4, 4))
```

Ignore: self.detect(0).sort().non_max_supress()[-np.arange(1, 4)].draw() from kwimage.structs.heatmap import * # NOQA import xdev globals().update(xdev.get_func_kwargs(Heatmap.random))

Example

```
>>> # xdoctest: +REQUIRES(module:ndsampler)
>>> import kwimage
>>> self = kwimage.Heatmap.random(dims=(50, 200), dets='coco',
>>>                               keypoints=True)
>>> image = np.zeros(self.img_dims)
>>> toshow = self.draw_on(image, 1, vecs=True, kpts=0, with_alpha=0.85)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.figure(fnum=1, doclf=True)
>>> kwplot.imshow(toshow)
```

Ignore:

```
>>> kwplot.figure(fnum=1, doclf=True)
>>> kwplot.imshow(image)
>>> dets.draw()
>>> dets.data['keypoints'].draw(radius=6)
>>> dets.data['segmentations'].draw()
```

```
>>> self.draw()
```

numpy(*self*)
Converts underlying data to numpy arrays

tensor(*self*, *device=ub.NoParam*)
Converts underlying data to torch tensors

kwimage.structs.**smooth_prob**(*prob*, *k=3*, *inplace=False*, *eps=1e-09*)
Smooths the probability map, but preserves the magnitude of the peaks.

Notes

even if inplace is true, we still need to make a copy of the input array, however, we do ensure that it is cleaned up before we leave the function scope.

sigma=0.8 @ k=3, sigma=1.1 @ k=5, sigma=1.4 @ k=7

class kwimage.structs.**Mask**(*data=None*, *format=None*)
Bases: ubelt.NiceRepr, kwimage.structs.mask._MaskConversionMixin, kwimage.structs.mask._MaskConstructorMixin, kwimage.structs.mask._MaskTransformMixin, kwimage.structs.mask._MaskDrawMixin

Manages a single segmentation mask and can convert to and from multiple formats including:

- bytes_rle - byte encoded run length encoding
- array_rle - raw run length encoding
- c_mask - c-style binary mask
- f_mask - fortran-style binary mask

Example

```
>>> # xdoc: +REQUIRES(--mask)
>>> # a ms-coco style compressed bytes rle segmentation
>>> segmentation = {'size': [5, 9], 'counts': ';?1B10O30O4'}
>>> mask = Mask(segmentation, 'bytes_rle')
>>> # convert to binary numpy representation
>>> binary_mask = mask.to_c_mask().data
>>> print(ub.repr2(binary_mask.tolist(), nl=1, nobr=1))
[0, 0, 0, 1, 1, 1, 1, 1, 0],
[0, 0, 1, 1, 1, 0, 0, 0, 0],
[0, 0, 1, 1, 1, 1, 1, 1, 0],
[0, 0, 1, 1, 1, 0, 1, 1, 0],
[0, 0, 1, 1, 1, 0, 1, 1, 0],
```

dtype

shape

area
Returns the number of non-zero pixels

Example

```
>>> # xdoc: +REQUIRES(--mask)
>>> self = Mask.demo()
>>> self.area
150
```

__nice__(*self*)

classmethod random(*Mask*, *rng=None*, *shape=(32, 32)*)

Example

Mask.random(rng=0).draw()

copy(*self*)
Performs a deep copy of the mask data

Example

```
>>> self = Mask.random(shape=(8, 8), rng=0)
>>> other = self.copy()
>>> assert other.data is not self.data
```

union(*self*, **others*)
This can be used as a staticmethod or an instancemethod

Example

```
>>> # xdoc: +REQUIRES(--mask)
>>> from kwimage.structs.mask import *  # NOQA
>>> masks = [Mask.random(shape=(8, 8), rng=i) for i in range(2)]
>>> mask = Mask.union(*masks)
>>> print(mask.area)
>>> masks = [m.to_c_mask() for m in masks]
>>> mask = Mask.union(*masks)
>>> print(mask.area)
```

```
>>> masks = [m.to_bytes_rle() for m in masks]
>>> mask = Mask.union(*masks)
>>> print(mask.area)
```

Benchmark: import ubelt as ub ti = ub.Timerit(100, bestof=10, verbose=2)

masks = [Mask.random(shape=(172, 172), rng=i) for i in range(2)]

for timer in ti.reset('native rle union'): masks = [m.to_bytes_rle() for m in masks] with timer:

mask = Mask.union(*masks)

for timer in ti.reset('native cmask union'): masks = [m.to_c_mask() for m in masks] with timer:

mask = Mask.union(*masks)

for timer in ti.reset('cmask->rle union'): masks = [m.to_c_mask() for m in masks] with timer:

mask = Mask.union(*[m.to_bytes_rle() for m in masks])

intersection(*self*, **others*)

This can be used as a staticmethod or an instancemethod

Example

```
>>> # xdoc: +REQUIRES(--mask)
>>> masks = [Mask.random(shape=(8, 8), rng=i) for i in range(2)]
>>> mask = Mask.intersection(*masks)
>>> print(mask.area)
```

get_patch(*self*)

Extract the patch with non-zero data

Example

```
>>> # xdoc: +REQUIRES(--mask)
>>> from kwimage.structs.mask import *  # NOQA
>>> self = Mask.random(shape=(8, 8), rng=0)
>>> self.get_patch()
```

get_xywh(*self*)

Gets the bounding xywh box coordinates of this mask

Returns

x, y, w, h: Note we dont use a Boxes object because a general singular version does not yet exist.

Return type ndarray

Example

```
>>> # xdoc: +REQUIRES(--mask)
>>> self = Mask.random(shape=(8, 8), rng=0)
>>> self.get_xywh().tolist()
>>> self = Mask.random(rng=0).translate((10, 10))
>>> self.get_xywh().tolist()
```

get_polygon(*self*)

Returns a list of (x,y)-coordinate lists. The length of the list is equal to the number of disjoint regions in the mask.

Returns

polygon around each connected component of the mask. Each ndarray is an Nx2 array of xy points.

Return type List[ndarray]

Note: The returned polygon may not surround points that are only one pixel thick.

Example

```
>>> # xdoc: +REQUIRES(--mask)
>>> from kwimage.structs.mask import *  # NOQA
>>> self = Mask.random(shape=(8, 8), rng=0)
>>> polygons = self.get_polygon()
>>> print('polygons = ' + ub.repr2(polygons))
>>> polygons = self.get_polygon()
>>> self = self.to_bytes_rle()
>>> other = Mask.from_polygons(polygons, self.shape)
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> image = np.ones(self.shape)
>>> image = self.draw_on(image, color='blue')
>>> image = other.draw_on(image, color='red')
>>> kwplot.imshow(image)
```

polygons = [np.array([[6, 4],[7, 4]], dtype=np.int32), np.array([[0, 1],[0, 3],[2, 3],[2, 1]], dtype=np.int32),

]

to_mask(*self*, *dims=None*)

to_boxes(*self*)

Returns the bounding box of the mask.

classmethod demo(*cls*)

Demo mask with holes and disjoint shapes

to_multi_polygon(*self*)
Returns a MultiPolygon object fit around this raster including disjoint pieces and holes.

Returns vectorized representation

Return type *MultiPolygon*

Example

```
>>> # xdoc: +REQUIRES(--mask)
>>> from kwimage.structs.mask import *  # NOQA
>>> self = Mask.demo()
>>> self = self.scale(5)
>>> multi_poly = self.to_multi_polygon()
>>> # xdoc: +REQUIRES(module:kwplot)
>>> # xdoc: +REQUIRES(--show)
>>> self.draw(color='red')
>>> multi_poly.scale(1.1).draw(color='blue')
```

```
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> image = np.ones(self.shape)
>>> image = self.draw_on(image, color='blue')
>>> #image = other.draw_on(image, color='red')
>>> kwplot.imshow(image)
>>> multi_poly.draw()
```

get_convex_hull(*self*)
Returns a list of xy points around the convex hull of this mask

Note: The returned polygon may not surround points that are only one pixel thick.

Example

```
>>> # xdoc: +REQUIRES(--mask)
>>> self = Mask.random(shape=(8, 8), rng=0)
>>> polygons = self.get_convex_hull()
>>> print('polygons = ' + ub.repr2(polygons))
>>> other = Mask.from_polygons(polygons, self.shape)
```

iou(*self*, *other*)
The area of intersection over the area of union

Todo:

- **[ ] Write plural Masks version of this class, which should** be able to perform this operation more efficiently.

CommandLine: xdoctest -m kwimage.structs.mask Mask.iou

Example

```
>>> # xdoc: +REQUIRES(--mask)
>>> self = Mask.demo()
>>> other = self.translate(1)
>>> iou = self.iou(other)
>>> print('iou = {:.4f}'.format(iou))
iou = 0.0830
```

classmethod coerce(*Mask*, *data*, *dims=None*)

Attempts to auto-inspect the format of the data and conver to Mask

Parameters

- **data** – the data to coerce
- **dims** (*Tuple*) – required for certain formats like polygons height / width of the source image

Returns Mask

Example

```
>>> # xdoc: +REQUIRES(--mask)
>>> segmentation = {'size': [5, 9], 'counts': ';?1B10O30O4'}
>>> polygon = [
>>>     [np.array([[3, 0],[2, 1],[2, 4],[4, 4],[4, 3],[7, 0]])],
>>>     [np.array([[2, 1],[2, 2],[4, 2],[4, 1]])],
>>> ]
>>> dims = (9, 5)
>>> mask = (np.random.rand(32, 32) > .5).astype(np.uint8)
>>> Mask.coerce(polygon, dims).to_bytes_rle()
>>> Mask.coerce(segmentation).to_bytes_rle()
>>> Mask.coerce(mask).to_bytes_rle()
```

_to_coco(*self*)

Example

```
>>> # xdoc: +REQUIRES(--mask)
>>> from kwimage.structs.mask import *  # NOQA
>>> self = Mask.demo()
>>> data = self._to_coco()
>>> print(ub.repr2(data, nl=1))
```

to_coco(*self*, *style='orig'*)

Example

```
>>> # xdoc: +REQUIRES(--mask)
>>> from kwimage.structs.mask import *  # NOQA
>>> self = Mask.demo()
>>> data = self.to_coco()
>>> print(ub.repr2(data, nl=1))
```

class kwimage.structs.**MaskList**
Bases: *kwimage.structs._generic.ObjectList*

Store and manipulate multiple masks, usually within the same image

to_polygon_list(*self*)
Converts all mask objects to polygon objects

class kwimage.structs.**Points**(*data=None*, *meta=None*, *datakeys=None*, *metakeys=None*, ***kwargs*)
Bases: *kwimage.structs._generic.Spatial*, *kwimage.structs.points._PointsWarpMixin*

Stores multiple keypoints for a single object.

This stores both the geometry and the class metadata if available

Ignore:

meta = { “names” = [‘head’, ‘nose’, ‘tail’], “skeleton” = [(0, 1), (0, 2)],

}

Example

```
>>> from kwimage.structs.points import *  # NOQA
>>> xy = np.random.rand(10, 2)
>>> pts = Points(xy=xy)
>>> print('pts = {!r}'.format(pts))
```

__datakeys__ = ['xy', 'class_idxs', 'visible']

__metakeys__ = ['classes']

__repr__

shape

xy

__nice__(*self*)

__len__(*self*)

classmethod random(*Points*, *num=1*, *classes=None*, *rng=None*)
Makes random points; typically for testing purposes

Example

```
>>> import kwimage
>>> self = kwimage.Points.random(classes=[1, 2, 3])
>>> self.data
>>> print('self.data = {!r}'.format(self.data))
```

is_numpy(*self*)

is_tensor(*self*)

_impl(*self*)

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

Example

```
>>> from kwimage.structs.points import *  # NOQA
>>> self = Points.random(10)
>>> self.tensor()
```

numpy(*self*)

Example

```
>>> from kwimage.structs.points import *  # NOQA
>>> self = Points.random(10)
>>> self.tensor().numpy().tensor().numpy()
```

draw_on(*self*, *image*, *color='white'*, *radius=None*, *copy=False*)

CommandLine: xdoctest -m ~/code/kwimage/kwimage/structs/points.py Points.draw_on –show

Example

```
>>> # xdoc: +REQUIRES(module:kwplot)
>>> from kwimage.structs.points import *  # NOQA
>>> s = 128
>>> image = np.zeros((s, s))
>>> self = Points.random(10).scale(s)
>>> image = self.draw_on(image)
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.figure(fnum=1, doclf=True)
>>> kwplot.autompl()
>>> kwplot.imshow(image)
>>> self.draw(radius=3, alpha=.5)
>>> kwplot.show_if_requested()
```

Example

```
>>> # xdoc: +REQUIRES(module:kwplot)
>>> from kwimage.structs.points import *  # NOQA
>>> s = 128
>>> image = np.zeros((s, s))
>>> self = Points.random(10).scale(s)
>>> image = self.draw_on(image, radius=3, color='distinct')
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.figure(fnum=1, doclf=True)
>>> kwplot.autompl()
>>> kwplot.imshow(image)
>>> self.draw(radius=3, alpha=.5, color='classes')
>>> kwplot.show_if_requested()
```

Example

```
>>> import kwimage
>>> s = 32
>>> self = kwimage.Points.random(10).scale(s)
>>> color = 'blue'
>>> # Test drawong on all channel + dtype combinations
>>> im3 = np.zeros((s, s, 3), dtype=np.float32)
>>> im_chans = {
>>>     'im3': im3,
>>>     'im1': kwimage.convert_colorspace(im3, 'rgb', 'gray'),
>>>     'im4': kwimage.convert_colorspace(im3, 'rgb', 'rgba'),
>>> }
>>> inputs = {}
>>> for k, im in im_chans.items():
>>>     inputs[k + '_01'] = (kwimage.ensure_float01(im.copy()), {'radius':␣
↪None})
>>>     inputs[k + '_255'] = (kwimage.ensure_uint255(im.copy()), {'radius':␣
↪None})
>>> outputs = {}
>>> for k, v in inputs.items():
>>>     im, kw = v
>>>     outputs[k] = self.draw_on(im, color=color, **kw)
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.figure(fnum=2, doclf=True)
>>> kwplot.autompl()
>>> pnum_ = kwplot.PlotNums(nCols=2, nRows=len(inputs))
>>> for k in inputs.keys():
>>>     kwplot.imshow(inputs[k][0], fnum=2, pnum=pnum_(), title=k)
>>>     kwplot.imshow(outputs[k], fnum=2, pnum=pnum_(), title=k)
>>> kwplot.show_if_requested()
```

draw(*self*, *color='blue'*, *ax=None*, *alpha=None*, *radius=1*, ***kwargs*)

Example

```
>>> # xdoc: +REQUIRES(module:kwplot)
>>> from kwimage.structs.points import *  # NOQA
>>> pts = Points.random(10)
>>> # xdoc: +REQUIRES(--show)
>>> pts.draw(radius=0.01)
```

```
>>> from kwimage.structs.points import *  # NOQA
>>> self = Points.random(10, classes=['a', 'b', 'c'])
>>> self.draw(radius=0.01, color='classes')
```

compress(*self*, *flags*, *axis=0*, *inplace=False*)

Filters items based on a boolean criterion

Example

```
>>> from kwimage.structs.points import *  # NOQA
>>> self = Points.random(4)
```

(continues on next page)

(continued from previous page)

```
>>> flags = [1, 0, 1, 1]
>>> other = self.compress(flags)
>>> assert len(self) == 4
>>> assert len(other) == 3
```

```
>>> other = self.tensor().compress(flags)
>>> assert len(other) == 3
```

take (*self*, *indices*, *axis=0*, *inplace=False*)
Takes a subset of items at specific indices

Example

```
>>> from kwimage.structs.points import *  # NOQA
>>> self = Points.random(4)
>>> indices = [1, 3]
>>> other = self.take(indices)
>>> assert len(self) == 4
>>> assert len(other) == 2
```

```
>>> other = self.tensor().take(indices)
>>> assert len(other) == 2
```

classmethod concatenate (*cls*, *points*, *axis=0*)

to_coco (*self*, *style='orig'*)
Converts to an mscoco-like representation

Note: items that are usually id-references to other objects may need to be rectified.

Parameters **style** (*str*) – either orig, new, new-id, or new-name

Returns mscoco-like representation

Return type Dict

Example

```
>>> from kwimage.structs.points import *  # NOQA
>>> self = Points.random(4, classes=['a', 'b'])
>>> orig = self._to_coco(style='orig')
>>> print('orig = {!r}'.format(orig))
>>> new_name = self._to_coco(style='new-name')
>>> print('new_name = {}'.format(ub.repr2(new_name, nl=-1)))
>>> # xdoctest: +REQUIRES(module:ndsampler)
>>> import ndsampler
>>> self.meta['classes'] = ndsampler.CategoryTree.coerce(self.meta['classes'])
>>> new_id = self._to_coco(style='new-id')
>>> print('new_id = {}'.format(ub.repr2(new_id, nl=-1)))
```

_to_coco (*self*, *style='orig'*)
See to_coco

classmethod _from_coco(*cls*, *coco_kpts*, *class_idxs=None*, *classes=None*)

classmethod from_coco(*cls*, *coco_kpts*, *class_idxs=None*, *classes=None*)

Parameters

- **coco_kpts** (*list | dict*) – either the original list keypoint encoding or the new dict keypoint encoding.
- **class_idxs** (*list*) – only needed if using old style
- **classes** (*list | CategoryTree*) – list of all keypoint category names

Example

```
>>> ##
>>> classes = ['mouth', 'left-hand', 'right-hand']
>>> coco_kpts = [
>>>     {'xy': (0, 0), 'visible': 2, 'keypoint_category': 'left-hand'},
>>>     {'xy': (1, 2), 'visible': 2, 'keypoint_category': 'mouth'},
>>> ]
>>> Points.from_coco(coco_kpts, classes=classes)
>>> # Test without classes
>>> Points.from_coco(coco_kpts)
>>> # Test without any category info
>>> coco_kpts2 = [ub.dict_diff(d, {'keypoint_category'}) for d in coco_kpts]
>>> Points.from_coco(coco_kpts2)
>>> # Test without category instead of keypoint_category
>>> coco_kpts3 = [ub.map_keys(lambda x: x.replace('keypoint_', ''), d) for d␣
↪in coco_kpts]
>>> Points.from_coco(coco_kpts3)
>>> #
>>> # Old style
>>> coco_kpts = [0, 0, 2, 0, 1, 2]
>>> Points.from_coco(coco_kpts)
>>> # Fail case
>>> coco_kpts4 = [{'xy': [4686.5, 1341.5], 'category': 'dot'}]
>>> Points.from_coco(coco_kpts4, classes=[])
```

Example

```
>>> # xdoctest: +REQUIRES(module:ndsampler)
>>> import ndsampler
>>> classes = ndsampler.CategoryTree.from_coco([
>>>     {'name': 'mouth', 'id': 2}, {'name': 'left-hand', 'id': 3}, {'name':
↪'right-hand', 'id': 5}
>>> ])
>>> coco_kpts = [
>>>     {'xy': (0, 0), 'visible': 2, 'keypoint_category_id': 5},
>>>     {'xy': (1, 2), 'visible': 2, 'keypoint_category_id': 2},
>>> ]
>>> pts = Points.from_coco(coco_kpts, classes=classes)
>>> assert pts.data['class_idxs'].tolist() == [2, 0]
```

class kwimage.structs.**PointsList**

Bases: *kwimage.structs._generic.ObjectList*

Stores a list of Points, each item usually corresponds to a different object.

Notes

TODO: when the data is homogenous we can use a more efficient # representation, otherwise we have to use heterogenous storage.

class kwimage.structs.**MultiPolygon**

Bases: *kwimage.structs._generic.ObjectList*

classmethod random(*self*, *n=3*, *rng=None*, *tight=False*)

to_multi_polygon(*self*)

to_mask(*self*, *dims=None*)

Returns a mask object indication regions occupied by this multipolygon

Example

```
>>> from kwimage.structs.polygon import *  # NOQA
>>> s = 100
>>> self = MultiPolygon.random(rng=0).scale(s)
>>> dims = (s, s)
>>> mask = self.to_mask(dims)
```

```
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.figure(fnum=1, doclf=True)
>>> from matplotlib import pyplot as pl
>>> ax = plt.gca()
>>> ax.set_xlim(0, s)
>>> ax.set_ylim(0, s)
>>> self.draw(color='red', alpha=.4)
>>> mask.draw(color='blue', alpha=.4)
```

classmethod coerce(*cls*, *data*, *dims=None*)

See Mask.coerce

to_shapely(*self*)

Example

```
>>> # xdoc: +REQUIRES(module:kwplot)
>>> # xdoc: +REQUIRES(module:shapely)
>>> from kwimage.structs.polygon import *  # NOQA
>>> self = MultiPolygon.random(rng=0)
>>> geom = self.to_shapely()
>>> print('geom = {!r}'.format(geom))
```

classmethod from_shapely(*MultiPolygon*, *geom*)

Convert a shapely polygon or multipolygon to a kwimage.MultiPolygon

classmethod from_geojson(*MultiPolygon*, *data_geojson*)

Convert a geojson polygon or multipolygon to a kwimage.MultiPolygon

Example

```
>>> import kwimage
>>> orig = kwimage.MultiPolygon.random()
>>> data_geojson = orig.to_geojson()
>>> self = kwimage.MultiPolygon.from_geojson(data_geojson)
```

to_geojson(*self*)
Converts polygon to a geojson structure

classmethod from_coco(*cls*, *data*, *dims=None*)
Accepts either new-style or old-style coco multi-polygons

_to_coco(*self*, *style='orig'*)

to_coco(*self*, *style='orig'*)

Example

```
>>> from kwimage.structs.polygon import *  # NOQA
>>> self = MultiPolygon.random(1, rng=0)
>>> self.to_coco()
```

class kwimage.structs.**Polygon**(*data=None*, *meta=None*, *datakeys=None*, *metakeys=None*, ***kwargs*)
Bases: *kwimage.structs._generic.Spatial*, *kwimage.structs.polygon._PolyArrayBackend*, *kwimage.structs.polygon._PolyWarpMixin*, ubelt.NiceRepr

Represents a single polygon as set of exterior boundary points and a list of internal polygons representing holes.

By convention exterior boundaries should be counterclockwise and interior holes should be clockwise.

Example

```
>>> data = {
>>>     'exterior': np.array([[13,  1], [13, 19], [25, 19], [25,  1]]),
>>>     'interiors': [
>>>         np.array([[13, 13], [14, 12], [24, 12], [25, 13], [25, 18], [24, 19],␣
↪[14, 19], [13, 18]]),
>>>         np.array([[13,  2], [14,  1], [24,  1], [25, 2], [25, 11], [24, 12],␣
↪[14, 12], [13, 11]])]
>>> }
>>> self = Polygon(**data)
```

__datakeys__ = ['exterior', 'interiors']

__metakeys__ = ['classes']

__nice__(*self*)

classmethod circle(*cls*, *xy*, *r*, *resolution=64*)
Create a circular polygon

Example

```
>>> xy = (0.5, 0.5)
>>> r = .3
>>> poly = Polygon.circle(xy, r)
```

classmethod random(*cls*, *n=6*, *n_holes=0*, *convex=True*, *tight=False*, *rng=None*)

Parameters

- **n** (*int*) – number of points in the polygon (must be 3 or more)
- **n_holes** (*int*) – number of holes
- **tight** (*bool, default=False*) – fits the minimum and maximum points between 0 and 1
- **convex** (*bool, default=True*) – force resulting polygon will be convex (may remove exterior points)

CommandLine: xdoctest -m kwimage.structs.polygon Polygon.random

Example

```
>>> rng = None
>>> n = 4
>>> n_holes = 1
>>> cls = Polygon
>>> self = Polygon.random(n=n, rng=rng, n_holes=n_holes, convex=1)
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.figure(fnum=1, doclf=True)
>>> kwplot.autompl()
>>> self.draw()
```

References

https://gis.stackexchange.com/questions/207731/random-multipolygon https://stackoverflow.com/questions/8997099/random-polygon https://stackoverflow.com/questions/27548363/from-voronoi-tessellation-to-shapely-polygons https://stackoverflow.com/questions/8997099/algorithm-to-generate-random-2d-polygon

_impl(*self*)

to_mask(*self*, *dims=None*)

Convert this polygon to a mask

Todo:

- [ ] currently not efficient

Example

```
>>> from kwimage.structs.polygon import *  # NOQA
>>> self = Polygon.random(n_holes=1).scale(128)
>>> mask = self.to_mask((128, 128))
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.figure(fnum=1, doclf=True)
>>> mask.draw(color='blue')
>>> mask.to_multi_polygon().draw(color='red', alpha=.5)
```

fill(*self*, *image*, *value=1*)
Inplace fill in an image based on this polyon.

_to_cv_countours(*self*)
OpenCV polygon representation, which is a list of points. Holes are implicitly represented. When another polygon is drawn over an existing polyon via cv2.fillPoly

Returns

where each ndarray is of shape [N, 1, 2], where N is the number of points on the boundary, the middle dimension is always 1, and the trailing dimension represents x and y coordinates respectively.

Return type List[ndarray]

to_shapely(*self*)

Example

```
>>> # xdoc: +REQUIRES(module:kwplot)
>>> # xdoc: +REQUIRES(module:shapely)
>>> from kwimage.structs.polygon import *  # NOQA
>>> self = Polygon.random(n_holes=1)
>>> self = self.scale(100)
>>> geom = self.to_shapely()
>>> print('geom = {!r}'.format(geom))
```

classmethod from_shapely(*Polygon*, *geom*)
Convert a shapely polygon to a kwimage.Polygon

Parameters **geom** (*shapely.geometry.polygon.Polygon*) – a shapely polygon

classmethod from_wkt(*Polygon*, *data*)
Convert a WKT string to a kwimage.Polygon

Parameters **data** (*str*) – a WKT polygon string

Example

data = kwimage.Polygon.random().to_shapely().to_wkt() data = 'POLYGON ((0.11 0.61, 0.07 0.588, 0.015 0.50, 0.11 0.61))' self = Polygon.from_wkt(data)

classmethod from_geojson(*MultiPolygon*, *data_geojson*)
Convert a geojson polygon to a kwimage.Polygon

Parameters **data_geojson** (*dict*) – geojson data

Example

```
>>> self = Polygon.random(n_holes=2)
>>> data_geojson = self.to_geojson()
>>> new = Polygon.from_geojson(data_geojson)
```

classmethod from_coco(*cls*, *data*, *dims=None*)
Accepts either new-style or old-style coco polygons

draw_on(*self*, *image*, *color='blue'*, *fill=True*, *border=False*, *alpha=1.0*, *copy=False*)
Rasterizes a polygon on an image. See *draw* for a vectorized matplotlib version.

Parameters

- **image** (*ndarray*) – image to raster polygon on.
- **color** (*str | tuple*) – data coercable to a color
- **fill** (*bool, default=True*) – draw the center mass of the polygon
- **border** (*bool, default=False*) – draw the border of the polygon
- **alpha** (*float, default=1.0*) – polygon transparency (setting alpha < 1 makes this function much slower).
- **copy** (*bool, default=False*) – if False only copies if necessary

Example

```
>>> # xdoc: +REQUIRES(module:kwplot)
>>> from kwimage.structs.polygon import *  # NOQA
>>> self = Polygon.random(n_holes=1).scale(128)
>>> image = np.zeros((128, 128), dtype=np.float32)
>>> image = self.draw_on(image)
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(image, fnum=1)
```

Example

```
>>> import kwimage
>>> color = 'blue'
>>> self = kwimage.Polygon.random(n_holes=1).scale(128)
>>> image = np.zeros((128, 128), dtype=np.float32)
>>> # Test drawong on all channel + dtype combinations
>>> im3 = np.random.rand(128, 128, 3)
>>> im_chans = {
>>>     'im3': im3,
>>>     'im1': kwimage.convert_colorspace(im3, 'rgb', 'gray'),
>>>     'im4': kwimage.convert_colorspace(im3, 'rgb', 'rgba'),
>>> }
>>> inputs = {}
>>> for k, im in im_chans.items():
>>>     inputs[k + '_01'] = (kwimage.ensure_float01(im.copy()), {'alpha':␣
↪None})
>>>     inputs[k + '_255'] = (kwimage.ensure_uint255(im.copy()), {'alpha':␣
↪None})
```

(continues on next page)

(continued from previous page)

```
>>>     inputs[k + '_01_a'] = (kwimage.ensure_float01(im.copy()), {'alpha': 0.
↪5})
>>>     inputs[k + '_255_a'] = (kwimage.ensure_uint255(im.copy()), {'alpha':␣
↪0.5})
>>> outputs = {}
>>> for k, v in inputs.items():
>>>     im, kw = v
>>>     outputs[k] = self.draw_on(im, color=color, **kw)
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.figure(fnum=2, doclf=True)
>>> kwplot.autompl()
>>> pnum_ = kwplot.PlotNums(nCols=2, nRows=len(inputs))
>>> for k in inputs.keys():
>>>     kwplot.imshow(inputs[k][0], fnum=2, pnum=pnum_(), title=k)
>>>     kwplot.imshow(outputs[k], fnum=2, pnum=pnum_(), title=k)
>>> kwplot.show_if_requested()
```

draw (*self*, *color='blue'*, *ax=None*, *alpha=1.0*, *radius=1*, *setlim=False*, *border=False*, *linewidth=2*)
Draws polygon in a matplotlib axes. See *draw_on* for in-memory image modification.

Example

```
>>> # xdoc: +REQUIRES(module:kwplot)
>>> from kwimage.structs.polygon import *  # NOQA
>>> self = Polygon.random(n_holes=1)
>>> self = self.scale(100)
>>> # xdoc: +REQUIRES(--show)
>>> self.draw()
>>> import kwplot
>>> kwplot.autompl()
>>> from matplotlib import pyplot as plt
>>> kwplot.figure(fnum=2)
>>> self.draw(setlim=True)
```

_to_coco (*self*, *style='orig'*)

to_coco (*self*, *style='orig'*)

to_geojson (*self*)
Converts polygon to a geojson structure

to_multi_polygon (*self*)

to_boxes (*self*)

copy (*self*)

clip (*self*, *x_min*, *y_min*, *x_max*, *y_max*, *inplace=False*)
Clip polygon to image boundaries.

Example

```
>>> from kwimage.structs.polygon import *
>>> self = Polygon.random().scale(10).translate(-1)
```

(continues on next page)

(continued from previous page)

```
>>> self2 = self.clip(1, 1, 3, 3)
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> self2.draw(setlim=True)
```

class kwimage.structs.**PolygonList**

Bases: *kwimage.structs._generic.ObjectList*

to_polygon_list(*self*)

Submodules

kwimage.im_alphablend

Module Contents

kwimage.im_alphablend.**overlay_alpha_layers**(*layers*, *keepalpha=True*, *dtype=np.float32*)

Stacks a sequences of layers on top of one another. The first item is the topmost layer and the last item is the bottommost layer.

Parameters

- **layers** (*Sequence[ndarray]*) – stack of images
- **keepalpha** (*bool*) – if False, the alpha channel is removed after blending
- **dtype** (*np.dtype*) – format for blending computation (defaults to float32)

Returns raster: the blended images

Return type ndarray

References

http://stackoverflow.com/questions/25182421/overlay-numpy-alpha https://en.wikipedia.org/wiki/Alpha_compositing#Alpha_blending

Example

```
>>> import kwimage
>>> keys = ['astro', 'carl', 'stars']
>>> layers = [kwimage.grab_test_image(k, dsize=(100, 100)) for k in keys]
>>> layers = [kwimage.ensure_alpha_channel(g, alpha=.5) for g in layers]
>>> stacked = overlay_alpha_layers(layers)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(stacked)
>>> kwplot.show_if_requested()
```

kwimage.im_alphablend.**overlay_alpha_images**(*img1*, *img2*, *keepalpha=True*, *dtype=np.float32*, *impl='inplace'*)

Places img1 on top of img2 respecting alpha channels. Works like the Photoshop layers with opacity.

Parameters

- **img1** (*ndarray*) – top image to overlay over img2
- **img2** (*ndarray*) – base image to superimpose on
- **keepalpha** (*bool*) – if False, the alpha channel is removed after blending
- **dtype** (*np.dtype*) – format for blending computation (defaults to float32)
- **impl** (*str, default=inplace*) – code specifying the backend implementation

Returns raster: the blended images

Return type ndarray

Todo:

- [ ] Make fast C++ version of this function

References

http://stackoverflow.com/questions/25182421/overlay-numpy-alpha https://en.wikipedia.org/wiki/Alpha_compositing#Alpha_blending

Example

```
>>> import kwimage
>>> img1 = kwimage.grab_test_image('astro', dsize=(100, 100))
>>> img2 = kwimage.grab_test_image('carl', dsize=(100, 100))
>>> img1 = kwimage.ensure_alpha_channel(img1, alpha=.5)
>>> img3 = overlay_alpha_images(img1, img2)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(img3)
>>> kwplot.show_if_requested()
```

kwimage.im_alphablend.**_prep_rgb_alpha**(*img*, *dtype=np.float32*)

kwimage.im_alphablend.**_alpha_blend_simple**(*rgb1*, *alpha1*, *rgb2*, *alpha2*)

Core alpha blending algorithm

SeeAlso: _alpha_blend_inplace - alternative implementation

kwimage.im_alphablend.**_alpha_blend_inplace**(*rgb1*, *alpha1*, *rgb2*, *alpha2*)

Uglier but faster(? maybe not) version of the core alpha blending algorithm using preallocation and in-place computation where possible.

SeeAlso: _alpha_blend_simple - alternative implementation

Example

```
>>> rng = np.random.RandomState(0)
>>> rgb1, rgb2 = rng.rand(10, 10, 3), rng.rand(10, 10, 3)
>>> alpha1, alpha2 = rng.rand(10, 10), rng.rand(10, 10)
>>> f1, f2 = _alpha_blend_inplace(rgb1, alpha1, rgb2, alpha2)
>>> s1, s2 = _alpha_blend_simple(rgb1, alpha1, rgb2, alpha2)
>>> assert np.all(f1 == s1) and np.all(f2 == s2)
>>> alpha1, alpha2 = np.zeros((10, 10)), np.zeros((10, 10))
>>> f1, f2 = _alpha_blend_inplace(rgb1, alpha1, rgb2, alpha2)
>>> s1, s2 = _alpha_blend_simple(rgb1, alpha1, rgb2, alpha2)
>>> assert np.all(f1 == s1) and np.all(f2 == s2)
```

kwimage.im_alphablend.**_alpha_blend_numexpr1**(*rgb1*, *alpha1*, *rgb2*, *alpha2*)
Alternative. Not well optimized

kwimage.im_alphablend.**_alpha_blend_numexpr2**(*rgb1*, *alpha1*, *rgb2*, *alpha2*)
Alternative. Not well optimized

kwimage.im_alphablend.**ensure_alpha_channel**(*img*, *alpha=1.0*, *dtype=np.float32*, *copy=False*)
Returns the input image with 4 channels.

Parameters

- **img** (*ndarray*) – an image with shape [H, W], [H, W, 1], [H, W, 3], or [H, W, 4].
- **alpha** (*float, default=1.0*) – default value for missing alpha channel
- **dtype** (*type, default=np.float32*) – a numpy floating type
- **copy** (*bool, default=False*) – always copy if True, else copy if needed.

Returns an image with specified dtype with shape [H, W, 4].

Raises *ValueError - if the input image does not have 1, 3, or 4 input channels* – or if the image cannot be converted into a float01 representation

kwimage.im_color

Module Contents

kwimage.im_color.**_lookup_colorspace_object**(*space*)

kwimage.im_color.**_colormath_convert**(*src_color*, *src_space*, *dst_space*)
Uses colormath to convert colors

Example

```
>>> # xdoctest: +REQUIRES(module:colormath)
>>> import kwimage
>>> src_color = kwimage.Color('turquoise').as01()
>>> print('src_color = {}'.format(ub.repr2(src_color, nl=0, precision=2)))
>>> src_space = 'rgb'
>>> dst_space = 'lab'
>>> lab_color = _colormath_convert(src_color, src_space, dst_space)
>>> print('lab_color = {}'.format(ub.repr2(lab_color, nl=0, precision=2)))
lab_color = (78.11, -70.09, -9.33)
>>> rgb_color = _colormath_convert(lab_color, 'lab', 'rgb')
```

(continues on next page)

(continued from previous page)

```
>>> print('rgb_color = {}'.format(ub.repr2(rgb_color, nl=0, precision=2)))
rgb_color = (0.29, 0.88, 0.81)
>>> hsv_color = _colormath_convert(lab_color, 'lab', 'hsv')
>>> print('hsv_color = {}'.format(ub.repr2(hsv_color, nl=0, precision=2)))
hsv_color = (175.39, 1.00, 0.88)
```

class kwimage.im_color.**Color**(*color*, *alpha=None*, *space=None*)

Bases: ubelt.NiceRepr

Used for converting a single color between spaces and encodings. This should only be used when handling small numbers of colors(e.g. 1), don't use this to represent an image.

move to colorutil?

Parameters **space** (*str*) – colorspace of wrapped color. Assume RGB if not specified and it cannot be inferred

CommandLine: xdoctest -m ~/code/kwimage/kwimage/im_color.py Color

Example

```
>>> print(Color('g'))
>>> print(Color('orangered'))
>>> print(Color('#AAAAAA').as255())
>>> print(Color([0, 255, 0]))
>>> print(Color([1, 1, 1.]))
>>> print(Color([1, 1, 1]))
>>> print(Color(Color([1, 1, 1])).as255())
>>> print(Color(Color([1., 0, 1, 0])).ashex())
>>> print(Color([1, 1, 1], alpha=255))
>>> print(Color([1, 1, 1], alpha=255, space='lab'))
```

__nice__(*self*)

_forimage(*self*, *image*, *space='rgb'*)

Experimental function.

Create a numeric color tuple that agrees with the format of the input image (i.e. float or int, with 3 or 4 channels).

Parameters

- **image** (*ndarray*) – image to return color for
- **space** (*str, default=rgb*) – colorspace of the input image.

Example

```
>>> img_f3 = np.zeros([8, 8, 3], dtype=np.float32)
>>> img_u3 = np.zeros([8, 8, 3], dtype=np.uint8)
>>> img_f4 = np.zeros([8, 8, 4], dtype=np.float32)
>>> img_u4 = np.zeros([8, 8, 4], dtype=np.uint8)
>>> Color('red')._forimage(img_f3)
(1.0, 0.0, 0.0)
>>> Color('red')._forimage(img_f4)
(1.0, 0.0, 0.0, 1.0)
```

(continues on next page)

(continued from previous page)

```
>>> Color('red')._forimage(img_u3)
(255, 0, 0)
>>> Color('red')._forimage(img_u4)
(255, 0, 0, 255)
>>> Color('red', alpha=0.5)._forimage(img_f4)
(1.0, 0.0, 0.0, 0.5)
>>> Color('red', alpha=0.5)._forimage(img_u4)
(255, 0, 0, 127)
```

ashex(*self*, *space=None*)

as255(*self*, *space=None*)

as01(*self*, *space=None*)
self = mplutil.Color('red') mplutil.Color('green').as01('rgba')

classmethod _is_base01(*channels*)
check if a color is in base 01

classmethod _is_base255(*Color*, *channels*)
there is a one corner case where all pixels are 1 or less

classmethod _hex_to_01(*Color*, *hex_color*)
hex_color = '#6A5AFFAF'

_ensure_color01(*Color*, *color*)
Infer what type color is and normalize to 01

classmethod _255_to_01(*Color*, *color255*)
converts base 255 color to base 01 color

classmethod _string_to_01(*Color*, *color*)
mplutil.Color._string_to_01('green') mplutil.Color._string_to_01('red')

classmethod named_colors(*cls*)

Returns names of colors that Color accepts

Return type List[str]

classmethod distinct(*Color*, *num*, *space='rgb'*)
Make multiple distinct colors

classmethod random(*Color*, *pool='named'*)

kwimage.im_color.**BASE_COLORS**

kwimage.im_color.**TABLEAU_COLORS = [['blue', '#1f77b4'], ['orange', '#ff7f0e'], ['green', '#**

kwimage.im_color.**TABLEAU_COLORS**

kwimage.im_color.**XKCD_COLORS**

kwimage.im_color.**XKCD_COLORS**

kwimage.im_color.**CSS4_COLORS**

kwimage.im_core

Not sure how to best classify these functions

Module Contents

kwimage.im_core.**num_channels**(*img*)

Returns the number of color channels in an image

Parameters **img** (*ndarray*) – an image with 2 or 3 dimensions.

Returns the number of color channels (1, 3, or 4)

Return type int

Example

```
>>> H = W = 3
>>> assert num_channels(np.empty((W, H))) == 1
>>> assert num_channels(np.empty((W, H, 1))) == 1
>>> assert num_channels(np.empty((W, H, 3))) == 3
>>> assert num_channels(np.empty((W, H, 4))) == 4
>>> # xdoctest: +REQUIRES(module:pytest)
>>> import pytest
>>> with pytest.raises(ValueError):
...     num_channels(np.empty((W, H, 2)))
```

kwimage.im_core.**ensure_float01**(*img*, *dtype=np.float32*, *copy=True*)

Ensure that an image is encoded using a float32 properly

Parameters

- **img** (*ndarray*) – an image in uint255 or float01 format. Other formats will raise errors.
- **dtype** (*type, default=np.float32*) – a numpy floating type
- **copy** (*bool, default=False*) – always copy if True, else copy if needed.

Returns an array of floats in the range 0-1

Return type ndarray

Raises ValueError – if the image type is integer and not in [0-255]

Example

```
>>> ensure_float01(np.array([[0, .5, 1.0]]))
array([[0. , 0.5, 1. ]], dtype=float32)
>>> ensure_float01(np.array([[0, 1, 200]]))
array([[0..., 0.0039..., 0.784...]], dtype=float32)
```

kwimage.im_core.**ensure_uint255**(*img*, *copy=True*)

Ensure that an image is encoded using a uint8 properly. Either

Parameters

- **img** (*ndarray*) – an image in uint255 or float01 format. Other formats will raise errors.
- **copy** (*bool, default=False*) – always copy if True, else copy if needed.

Returns an array of bytes in the range 0-255

Return type ndarray

Raises

- `ValueError` – if the image type is float and not in [0-1]
- `ValueError` – if the image type is integer and not in [0-255]

Example

```
>>> ensure_uint255(np.array([[0, .5, 1.0]]))
array([[  0, 127, 255]], dtype=uint8)
>>> ensure_uint255(np.array([[0, 1, 200]]))
array([[  0,   1, 200]], dtype=uint8)
```

kwimage.im_core.**make_channels_comparable**(*img1*, *img2*, *atleast3d=False*)

Broadcasts image arrays so they can have elementwise operations applied

Parameters

- **img1** (*ndarray*) – first image
- **img2** (*ndarray*) – second image
- **atleast3d** (*bool, default=False*) – if true we ensure that the channel dimension exists (only relevant for 1-channel images)

Example

```
>>> import itertools as it
>>> wh_basis = [(5, 5), (3, 5), (5, 3), (1, 1), (1, 3), (3, 1)]
>>> for w, h in wh_basis:
>>>     shape_basis = [(w, h), (w, h, 1), (w, h, 3)]
>>>     # Test all permutations of shap inputs
>>>     for shape1, shape2 in it.product(shape_basis, shape_basis):
>>>         print('*    input shapes: %r, %r' % (shape1, shape2))
>>>         img1 = np.empty(shape1)
>>>         img2 = np.empty(shape2)
>>>         img1, img2 = make_channels_comparable(img1, img2)
>>>         print('... output shapes: %r, %r' % (img1.shape, img2.shape))
>>>         elem = (img1 + img2)
>>>         print('... elem(+) shape: %r' % (elem.shape,))
>>>         assert elem.size == img1.size, 'outputs should have same size'
>>>         assert img1.size == img2.size, 'new imgs should have same size'
>>>         print('--------')
```

kwimage.im_core.**_alpha_fill_for**(*img*)

helper for make_channels_comparable

kwimage.im_core.**atleast_3channels**(*arr*, *copy=True*)

Ensures that there are 3 channels in the image

Parameters

- **arr** (*ndarray[N, M, ...]*) – the image
- **copy** (*bool*) – Always copies if True, if False, then copies only when the size of the array must change.

Returns with shape (N, M, C), where C in {3, 4}

Return type ndarray

Doctest:

```
>>> assert atleast_3channels(np.zeros((10, 10))).shape[-1] == 3
>>> assert atleast_3channels(np.zeros((10, 10, 1))).shape[-1] == 3
>>> assert atleast_3channels(np.zeros((10, 10, 3))).shape[-1] == 3
>>> assert atleast_3channels(np.zeros((10, 10, 4))).shape[-1] == 4
```

kwimage.im_cv2

Wrappers around cv2 functions

Note: all functions in kwimage work with RGB input by default instead of BGR.

Module Contents

kwimage.im_cv2.**_CV2_INTERPOLATION_TYPES**

kwimage.im_cv2.**_rectify_interpolation**(*interpolation*, *default=cv2.INTER_LANCZOS4*, *grow_default=cv2.INTER_LANCZOS4*, *shrink_default=cv2.INTER_AREA*, *scale=None*)

Converts interpolation into flags suitable cv2 functions

Parameters

- **interpolation** (*int or str*) – string or cv2-style interpolation type
- **default** (*int*) – cv2 flag to use if *interpolation* is None and scale is None.
- **grow_default** (*int*) – cv2 flag to use if *interpolation* is None and scale is greater than or equal to 1.
- **shrink_default** (*int*) – cv2 flag to use if *interpolation* is None and scale is less than 1.
- **scale** (*float*) – indicate if the interpolation will be used to scale the image.

Returns

flag specifying interpolation type that can be passed to functions like cv2.resize, cv2.warpAffine, etc...

Return type int

kwimage.im_cv2.**imscale**(*img*, *scale*, *interpolation=None*, *return_scale=False*)

Resizes an image by a scale factor.

Because the result image must have an integer number of pixels, the scale factor is rounded, and the rounded scale factor is optionaly returned.

Parameters

- **img** (*ndarray*) – image to resize
- **scale** (*float or Tuple[float, float]*) – desired floating point scale factor. If a tuple, the dimension ordering is x,y.
- **interpolation** (*str | int*) – interpolation key or code (e.g. linear lanczos)
- **return_scale** (*bool, default=False*) – if True returns both the new image and the actual scale factor used to achive the new integer image size.

SeeAlso: imresize

Example

```
>>> import kwimage
>>> import numpy as np
>>> img = np.zeros((10, 10, 3), dtype=np.uint8)
>>> new_img, new_scale = kwimage.imscale(img, scale=.85,
>>>                                      interpolation='nearest',
>>>                                      return_scale=True)
>>> assert new_scale == (.8, .8)
>>> assert new_img.shape == (8, 8, 3)
```

kwimage.im_cv2.**imresize**(*img*, *scale=None*, *dsize=None*, *max_dim=None*, *min_dim=None*, *interpolation=None*, *letterbox=False*, *return_info=False*)

Resize an image based on a scale factor, final size, or size and aspect ratio.

Slightly more general than cv2.resize and kwimage.imscale, allows for specification of either a scale factor, a final size, or the final size for a particular dimension.

Parameters

- **img** (*ndarray*) – image to resize
- **scale** (*float or Tuple[float, float]*) – desired floating point scale factor. If a tuple, the dimension ordering is x,y. Mutually exclusive with dsize, max_dim, and min_dim.
- **dsize** (*Tuple[None | int, None | int]*) – the desired with and height of the new image. If a dimension is None, then it is automatically computed to preserve aspect ratio. Mutually exclusive with size, max_dim, and min_dim.
- **max_dim** (*int*) – new size of the maximum dimension, the other dimension is scaled to maintain aspect ratio. Mutually exclusive with size, dsize, and min_dim.
- **min_dim** (*int*) – new size of the minimum dimension, the other dimension is scaled to maintain aspect ratio.Mutually exclusive with size, dsize, and max_dim.
- **interpolation** (*str | int*) – interpolation key or code (e.g. linear lanczos)
- **letterbox** (*bool, default=False*) – if used in conjunction with dsize, then the image is scaled and translated to fit in the center of the new image while maintaining aspect ratio. Black padding is added if necessary.
- **return_info** (*bool, default=False*) – if True returns information about the final transformation in a dictionary.

Returns the new image and optionally an info dictionary

Return type ndarray | Tuple[ndarray, Dict]

Example

```
>>> import kwimage
>>> import numpy as np
>>> # Test scale
>>> img = np.zeros((16, 10, 3), dtype=np.uint8)
>>> new_img, info = kwimage.imresize(img, scale=.85,
>>>                                  interpolation='area',
>>>                                  return_info=True)
>>> print('info = {!r}'.format(info))
>>> assert info['scale'].tolist() == [.8, 0.875]
```

(continues on next page)

(continued from previous page)

```
>>> # Test dsize without None
>>> new_img, info = kwimage.imresize(img, dsize=(5, 12),
>>>                                  interpolation='area',
>>>                                  return_info=True)
>>> print('info = {!r}'.format(info))
>>> assert info['scale'].tolist() == [0.5 , 0.75]
>>> # Test dsize with None
>>> new_img, info = kwimage.imresize(img, dsize=(6, None),
>>>                                  interpolation='area',
>>>                                  return_info=True)
>>> print('info = {!r}'.format(info))
>>> assert info['scale'].tolist() == [0.6, 0.625]
>>> # Test max_dim
>>> new_img, info = kwimage.imresize(img, max_dim=6,
>>>                                  interpolation='area',
>>>                                  return_info=True)
>>> print('info = {!r}'.format(info))
>>> assert info['scale'].tolist() == [0.4  , 0.375]
>>> # Test min_dim
>>> new_img, info = kwimage.imresize(img, min_dim=6,
>>>                                  interpolation='area',
>>>                                  return_info=True)
>>> print('info = {!r}'.format(info))
>>> assert info['scale'].tolist() == [0.6  , 0.625]
```

Example

```
>>> import kwimage
>>> import numpy as np
>>> # Test letterbox resize
>>> img = np.ones((5, 10, 3), dtype=np.float32)
>>> new_img, info = kwimage.imresize(img, dsize=(19, 19),
>>>                                  letterbox=True,
>>>                                  return_info=True)
>>> print('info = {!r}'.format(info))
>>> assert info['offset'].tolist() == [0, 4]
>>> img = np.ones((10, 5, 3), dtype=np.float32)
>>> new_img, info = kwimage.imresize(img, dsize=(19, 19),
>>>                                  letterbox=True,
>>>                                  return_info=True)
>>> print('info = {!r}'.format(info))
>>> assert info['offset'].tolist() == [4, 0]
```

```
>>> import kwimage
>>> import numpy as np
>>> # Test letterbox resize
>>> img = np.random.rand(100, 200)
>>> new_img, info = kwimage.imresize(img, dsize=(300, 300), letterbox=True,␣
↪return_info=True)
```

kwimage.im_cv2.**convert_colorspace**(*img*, *src_space*, *dst_space*, *copy=False*, *implicit=False*, *dst=None*)

Converts colorspace of img. Convinience function around cv2.cvtColor

Parameters

- **img** (*ndarray*) – image data with float32 or uint8 precision
- **src_space** (*str*) – input image colorspace. (e.g. BGR, GRAY)
- **dst_space** (*str*) – desired output colorspace. (e.g. RGB, HSV, LAB)
- **implicit** (*bool*) –

  if False, the user must correctly specify if the input/output colorspaces contain alpha channels.

  If True and the input image has an alpha channel, we modify src_space and dst_space to ensure they both end with "A".
- **dst** (*ndarray[uint8_t, ndim=2], optional*) – inplace-output array.

Returns img - image data

Return type ndarray

Note: Note the LAB and HSV colorspaces in float do not go into the 0-1 range.

For HSV the floating point range is: 0:360, 0:1, 0:1

For LAB the floating point range is: 0:100, -86.1875:98.234375, -107.859375:94.46875 (Note, that some extreme combinations of a and b are not valid)

Example

```
>>> import numpy as np
>>> convert_colorspace(np.array([[[0, 0, 1]]], dtype=np.float32), 'RGB', 'LAB')
>>> convert_colorspace(np.array([[[0, 1, 0]]], dtype=np.float32), 'RGB', 'LAB')
>>> convert_colorspace(np.array([[[1, 0, 0]]], dtype=np.float32), 'RGB', 'LAB')
>>> convert_colorspace(np.array([[[1, 1, 1]]], dtype=np.float32), 'RGB', 'LAB')
>>> convert_colorspace(np.array([[[0, 0, 1]]], dtype=np.float32), 'RGB', 'HSV')
```

Ignore: # Check LAB output ranges import itertools as it s = 1 _iter = it.product(range(0, 256, s), range(0, 256, s), range(0, 256, s)) minvals = np.full(3, np.inf) maxvals = np.full(3, -np.inf) for r, g, b in ub.ProgIter(_iter, total=(256 // s) ** 3):

img255 = np.array([[[r, g, b]]], dtype=np.uint8) img01 = (img255 / 255.0).astype(np.float32) lab = convert_colorspace(img01, 'rgb', 'lab') np.minimum(lab[0, 0], minvals, out=minvals) np.maximum(lab[0, 0], maxvals, out=maxvals)

print('minvals = {}'.format(ub.repr2(minvals, nl=0))) print('maxvals = {}'.format(ub.repr2(maxvals, nl=0)))

kwimage.im_cv2.**_lookup_cv2_colorspace_conversion_code**(*src_space*, *dst_space*)

kwimage.im_cv2.**gaussian_patch**(*shape=(7, 7)*, *sigma=None*)

Creates a 2D gaussian patch with a specific size and sigma

Parameters

- **shape** (*Tuple[int, int]*) – patch height and width
- **sigma** (*float | Tuple[float, float]*) – gaussian standard deviation

References

http://docs.opencv.org/modules/imgproc/doc/filtering.html#getgaussiankernel

---

Todo:

- [ ] Look into this C-implementation

https://kwgitlab.kitware.com/computer-vision/heatmap/blob/master/heatmap/heatmap.c

---

CommandLine: xdoctest -m kwimage.im_cv2 gaussian_patch –show

Example

```
>>> import numpy as np
>>> shape = (88, 24)
>>> sigma = None  # 1.0
>>> gausspatch = gaussian_patch(shape, sigma)
>>> sum_ = gausspatch.sum()
>>> assert np.all(np.isclose(sum_, 1.0))
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> norm = (gausspatch - gausspatch.min()) / (gausspatch.max() - gausspatch.min())
>>> kwplot.imshow(norm)
>>> kwplot.show_if_requested()
```

Example

```
>>> import numpy as np
>>> shape = (24, 24)
>>> sigma = 3.0
>>> gausspatch = gaussian_patch(shape, sigma)
>>> sum_ = gausspatch.sum()
>>> assert np.all(np.isclose(sum_, 1.0))
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> norm = (gausspatch - gausspatch.min()) / (gausspatch.max() - gausspatch.min())
>>> kwplot.imshow(norm)
>>> kwplot.show_if_requested()
```

kwimage.im_demodata

Module Contents

kwimage.im_demodata.**_TEST_IMAGES**

kwimage.im_demodata.**grab_test_image**(*key='astro'*, *space='rgb'*, *dsize=None*, *interpolation='lanczos'*)

Ensures that the test image exists (this might use the network), reads it and returns the the image pixels.

Parameters

- **key** (*str*) – which test image to grab. Valid choices are: astro - an astronaught carl - Carl Sagan paraview - ParaView logo stars - picture of stars in the sky airport - SkySat image of Beijing Capital International Airport on 18 February 2018
- **space** (*str, default='rgb'*) – which colorspace to return in
- **dsize** (*Tuple[int, int], default=None*) – if specified resizes image to this size

Returns the requested image

Return type ndarray

CommandLine: xdoctest -m ~/code/kwimage/kwimage/im_demodata.py grab_test_image

Example

```
>>> for key in grab_test_image.keys():
...     grab_test_image(key)
>>> grab_test_image('astro', dsize=(255, 255)).shape
(255, 255, 3)
```

kwimage.im_demodata.**grab_test_image_fpath**(*key='astro'*)

Ensures that the test image exists (this might use the network) and returns the cached filepath to the requested image.

Parameters **key** (*str*) – which test image to grab. Valid choices are: astro - an astronaught carl - Carl Sagan paraview - ParaView logo stars - picture of stars in the sky

Returns path to the requested image

Return type str

Example

```
>>> for key in grab_test_image.keys():
...     grab_test_image_fpath(key)
```

kwimage.im_demodata.**keys**

kwimage.im_demodata.**keys**

kwimage.im_draw

Module Contents

kwimage.im_draw.**draw_text_on_image**(*img*, *text*, *org*, ***kwargs*)

Draws multiline text on an image using opencv

Note: This function also exists in kwplot

The image is modified inplace. If the image is non-contiguous then this returns a UMat instead of a ndarray, so be carefull with that.

Parameters

- **img** (*ndarray*) – image to draw on (inplace)
- **text** (*str*) – text to draw
- **org** (*tuple*) – x, y location of the text string in the image. if bottomLeftOrigin=True this is the bottom-left corner of the text otherwise it is the top-left corner (default).
- ****kwargs** – color (tuple): default blue thickneess (int): defaults to 2 fontFace (int): defaults to cv2.FONT_HERSHEY_SIMPLEX fontScale (float): defaults to 1.0 valign (str, default=bottom): either top, center, or bottom

References

https://stackoverflow.com/questions/27647424/

Example

```
>>> import kwimage
>>> img = kwimage.grab_test_image(space='rgb')
>>> img2 = kwimage.draw_text_on_image(img.copy(), 'FOOBAR', org=(0, 0), valign=
↪'top')
>>> assert img2.shape == img.shape
>>> assert np.any(img2 != img)
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(img2, fontScale=10)
>>> kwplot.show_if_requested()
```

Example

```
>>> import kwimage
>>> img = kwimage.grab_test_image(space='rgb')
>>> img2 = kwimage.draw_text_on_image(img, 'FOOBAR\nbazbiz\nspam', org=(0, 0),␣
↪valign='top', border=2)
>>> img2 = kwimage.draw_text_on_image(img, 'FOOBAR\nbazbiz\nspam', org=(150, 0),␣
↪valign='center', border=2)
>>> img2 = kwimage.draw_text_on_image(img, 'FOOBAR\nbazbiz\nspam', org=(300, 0),␣
↪valign='bottom', border=2)
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(img2, fontScale=10)
>>> kwplot.show_if_requested()
```

Example

```
>>> # Ensure the function works with float01 or uint255 images
>>> import kwimage
>>> img = kwimage.grab_test_image(space='rgb')
```

(continues on next page)

(continued from previous page)

```
>>> img = kwimage.ensure_float01(img)
>>> img2 = kwimage.draw_text_on_image(img, 'FOOBAR\nbazbiz\nspam', org=(0, 0),␣
↪valign='top', border=2)
```

kwimage.im_draw.**draw_clf_on_image**(*im*, *classes*, *tcx*, *probs=None*, *pcx=None*, *border=1*)

Draws classification label on an image

Parameters

- **im** (*ndarray*) – the image
- **classes** (*Sequence | CategoryTree*) – list of class names
- **tcx** (*int*) – true class index
- **probs** (*ndarray*) – predicted class probs for each class
- **pcx** (*int*) – predicted class index. (if none uses argmax of probs)

Example

```
>>> import torch
>>> import kwarray
>>> import kwimage
>>> rng = kwarray.ensure_rng(0)
>>> im = (rng.rand(300, 300) * 255).astype(np.uint8)
>>> classes = ['cls_a', 'cls_b', 'cls_c']
>>> tcx = 1
>>> probs = rng.rand(len(classes))
>>> probs[tcx] = 0
>>> probs = torch.FloatTensor(probs).softmax(dim=0).numpy()
>>> im1_ = kwimage.draw_clf_on_image(im, classes, tcx, probs)
>>> probs[tcx] = .9
>>> probs = torch.FloatTensor(probs).softmax(dim=0).numpy()
>>> im2_ = kwimage.draw_clf_on_image(im, classes, tcx, probs)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(im1_, colorspace='rgb', pnum=(1, 2, 1), fnum=1, doclf=True)
>>> kwplot.imshow(im2_, colorspace='rgb', pnum=(1, 2, 2), fnum=1)
>>> kwplot.show_if_requested()
```

kwimage.im_draw.**draw_boxes_on_image**(*img*, *boxes*, *color='blue'*, *thickness=1*, *box_format=None*, *colorspace='rgb'*)

Draws boxes on an image.

Parameters

- **img** (*ndarray*) – image to copy and draw on
- **boxes** (*nh.util.Boxes*) – boxes to draw
- **colorspace** (*str*) – string code of the input image colorspace

Example

```
>>> import kwimage
>>> import numpy as np
>>> img = np.zeros((10, 10, 3), dtype=np.uint8)
>>> color = 'dodgerblue'
>>> thickness = 1
>>> boxes = kwimage.Boxes([[1, 1, 8, 8]], 'tlbr')
>>> img2 = draw_boxes_on_image(img, boxes, color, thickness)
>>> assert tuple(img2[1, 1]) == (30, 144, 255)
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()  # xdoc: +SKIP
>>> kwplot.figure(doclf=True, fnum=1)
>>> kwplot.imshow(img2)
```

kwimage.im_draw.**make_heatmask**(*probs*, *cmap='plasma'*, *with_alpha=1.0*, *space='rgb'*, *dsize=None*)

Colorizes a single-channel intensity mask (with an alpha channel)

Parameters

- **probs** (*ndarray*) – 2D probability map with values between 0 and 1
- **cmap** (*str*) – mpl colormap
- **with_alpha** (*float*) – between 0 and 1, uses probs as the alpha multipled by this number.
- **space** (*str*) – output colorspace
- **dsize** (*tuple*) – if not None, then output is resized to W,H=dsize

SeeAlso: kwimage.overlay_alpha_images

Example

```
>>> # xdoc: +REQUIRES(module:matplotlib)
>>> probs = np.tile(np.linspace(0, 1, 10), (10, 1))
>>> heatmask = make_heatmask(probs, with_alpha=0.8, dsize=(100, 100))
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.imshow(heatmask, fnum=1, doclf=True, colorspace='rgb')
>>> kwplot.show_if_requested()
```

kwimage.im_draw.**make_orimask**(*radians*, *mag=None*, *alpha=1.0*)

Makes a colormap in HSV space where the orientation changes color and mag changes the saturation/value.

Parameters

- **radians** (*ndarray*) – orientation in radians
- **mag** (*ndarray*) – magnitude (must be normalized between 0 and 1)
- **alpha** (*float | ndarray*) – if False or None, then the image is returned without alpha if a float, then mag is scaled by this and used as the alpha channel if an ndarray, then this is explicilty set as the alpha channel

Returns an rgb / rgba image in 01 space

Return type ndarray[float32]

SeeAlso: kwimage.overlay_alpha_images

Example

```
>>> # xdoc: +REQUIRES(module:matplotlib)
>>> x, y = np.meshgrid(np.arange(64), np.arange(64))
>>> dx, dy = x - 32, y - 32
>>> radians = np.arctan2(dx, dy)
>>> mag = np.sqrt(dx ** 2 + dy ** 2)
>>> orimask = make_orimask(radians, mag)
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.imshow(orimask, fnum=1, doclf=True, colorspace='rgb')
>>> kwplot.show_if_requested()
```

kwimage.im_draw.**make_vector_field**(*dx*, *dy*, *stride=1*, *thresh=0.0*, *scale=1.0*, *alpha=1.0*, *color='red'*, *thickness=1*, *tipLength=0.1*, *line_type='aa'*)

Create an image representing a 2D vector field.

Parameters

- **dx** (*ndarray*) – grid of vector x components
- **dy** (*ndarray*) – grid of vector y components
- **stride** (*int*) – sparsity of vectors
- **thresh** (*float*) – only plot vectors with magnitude greater than thres
- **scale** (*float*) – multiply magnitude for easier visualization
- **alpha** (*float*) – alpha value for vectors. Non-vector regions receive 0 alpha (if False, no alpha channel is used)
- **color** (*str | tuple | kwimage.Color*) – RGB color of the vectors
- **thickness** (*int, default=1*) – thickness of arrows
- **tipLength** (*float, default=0.1*) – fraction of line length
- **line_type** (*int*) – either cv2.LINE_4, cv2.LINE_8, or cv2.LINE_AA

Returns vec_img: an rgb/rgba image in 0-1 space

Return type ndarray[float32]

SeeAlso: kwimage.overlay_alpha_images

Example

```
>>> x, y = np.meshgrid(np.arange(512), np.arange(512))
>>> dx, dy = x - 256.01, y - 256.01
>>> radians = np.arctan2(dx, dy)
>>> mag = np.sqrt(dx ** 2 + dy ** 2)
>>> dx, dy = dx / mag, dy / mag
>>> img = make_vector_field(dx, dy, stride=10, scale=10, alpha=False)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(img)
>>> kwplot.show_if_requested()
```

kwimage.im_io

Module Contents

kwimage.im_io.**imread**(*fpath*, *space='auto'*, *backend='auto'*)

Reads image data in a specified format using some backend implementation.

Parameters

- **fpath** (*str*) – path to the file to be read
- **space** (*str, default='auto'*) – the desired colorspace of the image. Can by any colorspace accepted by *convert_colorspace*, or it can be 'auto', in which case the colorspace of the image is unmodified (except in the case where a color image is read by opencv, in which case we convert BGR to RGB by default). If None, then no modification is made to whaveter backend is used to read the image.
- **backend** (*str, default='auto'*) – which backend reader to use. By default the file extension is used to determine this, but it can be manually overridden. Valid backends are gdal, skimage, and cv2.

Note: if space is something non-standard like HSV or LAB, then the file must be a normal 8-bit color image, otherwise an error will occur.

Raises

- IOError - If the image cannot be read
- ImportError - If trying to read a nitf without gdal
- NotImplementedError - if trying to read a corner-case image

Example

```
>>> # xdoctest: +REQUIRES(--network)
>>> from kwimage.im_io import *  # NOQA
>>> import tempfile
>>> from os.path import splitext  # NOQA
>>> # Test a non-standard image, which encodes a depth map
>>> fpath = ub.grabdata('http://www.topcoder.com/contest/problem/UrbanMapper3D/
↪JAX_Tile_043_DTM.tif')
>>> img1 = imread(fpath)
>>> # Check that write + read preserves data
>>> tmp = tempfile.NamedTemporaryFile(suffix=splitext(fpath)[1])
>>> imwrite(tmp.name, img1)
>>> img2 = imread(tmp.name)
>>> assert np.all(img2 == img1)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(img1, pnum=(1, 2, 1), fnum=1)
>>> kwplot.imshow(img2, pnum=(1, 2, 2), fnum=1)
```

Example

```
>>> # xdoctest: +REQUIRES(--network)
>>> import tempfile
>>> img1 = imread(ub.grabdata('http://i.imgur.com/iXNf4Me.png', fname='ada.png'))
>>> tmp_tif = tempfile.NamedTemporaryFile(suffix='.tif')
>>> tmp_png = tempfile.NamedTemporaryFile(suffix='.png')
>>> imwrite(tmp_tif.name, img1)
>>> imwrite(tmp_png.name, img1)
>>> tif_im = imread(tmp_tif.name)
>>> png_im = imread(tmp_png.name)
>>> assert np.all(tif_im == png_im)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(png_im, pnum=(1, 2, 1), fnum=1)
>>> kwplot.imshow(tif_im, pnum=(1, 2, 2), fnum=1)
```

Example

```
>>> # xdoctest: +REQUIRES(--network)
>>> import tempfile
>>> tif_fpath = ub.grabdata('https://ghostscript.com/doc/tiff/test/images/rgb-3c-
↪16b.tiff', fname='pepper.tif'),
>>> img1 = imread(tif_fpath)
>>> tmp_tif = tempfile.NamedTemporaryFile(suffix='.tif')
>>> tmp_png = tempfile.NamedTemporaryFile(suffix='.png')
>>> imwrite(tmp_tif.name, img1)
>>> imwrite(tmp_png.name, img1)
>>> tif_im = imread(tmp_tif.name)
>>> png_im = imread(tmp_png.name)
>>> assert np.all(tif_im == png_im)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(png_im / 2 ** 16, pnum=(1, 2, 1), fnum=1)
>>> kwplot.imshow(tif_im / 2 ** 16, pnum=(1, 2, 2), fnum=1)
```

kwimage.im_io.**_imread_skimage**(*fpath*)

kwimage.im_io.**_imread_cv2**(*fpath*)

kwimage.im_io.**_imread_gdal**(*fpath*)
gdal imread backend

kwimage.im_io.**imwrite**(*fpath*, *image*, *space='auto'*)
Writes image data to disk.

Parameters

- **fpath** (*PathLike*) – location to save the imaeg
- **image** (*ndarray*) – image data
- **space** (*str*) – the colorspace of the image to save. Can by any colorspace accepted by *convert_colorspace*, or it can be 'auto', in which case we assume the input image is either RGB, RGBA or grayscale. If None, then absolutely no color modification is made and whatever backend is used writes the image as-is.

Notes

The image may be modified to preserve its colorspace depending on which backend is used to write the image.

Raises `Exception` – if the image cannot be written

Doctest:

```
>>> # xdoctest: +REQUIRES(--network)
>>> # This should be moved to a unit test
>>> import tempfile
>>> test_image_paths = [
>>>     ub.grabdata('https://ghostscript.com/doc/tiff/test/images/rgb-3c-16b.
↪tiff', fname='pepper.tif'),
>>>     ub.grabdata('http://i.imgur.com/iXNf4Me.png', fname='ada.png'),
>>>     #ub.grabdata('http://www.topcoder.com/contest/problem/UrbanMapper3D/
↪JAX_Tile_043_DTM.tif'),
>>>     ub.grabdata('https://upload.wikimedia.org/wikipedia/commons/f/fa/
↪Grayscale_8bits_palette_sample_image.png', fname='parrot.png')
>>> ]
>>> for fpath in test_image_paths:
>>>     for space in ['auto', 'rgb', 'bgr', 'gray', 'rgba']:
>>>         img1 = imread(fpath, space=space)
>>>         print('Test im-io consistency of fpath = {!r} in {} space, shape=
↪{}'.format(fpath, space, img1.shape))
>>>         # Write the image in TIF and PNG format
>>>         tmp_tif = tempfile.NamedTemporaryFile(suffix='.tif')
>>>         tmp_png = tempfile.NamedTemporaryFile(suffix='.png')
>>>         imwrite(tmp_tif.name, img1, space=space)
>>>         imwrite(tmp_png.name, img1, space=space)
>>>         tif_im = imread(tmp_tif.name, space=space)
>>>         png_im = imread(tmp_png.name, space=space)
>>>         assert np.all(tif_im == png_im), 'im-read/write inconsistency'
>>>         if space == 'gray':
>>>             assert tif_im.ndim == 2
>>>             assert png_im.ndim == 2
>>>         elif space in ['rgb', 'bgr']:
>>>             assert tif_im.shape[2] == 3
>>>             assert png_im.shape[2] == 3
>>>         elif space in ['rgba', 'bgra']:
>>>             assert tif_im.shape[2] == 4
>>>             assert png_im.shape[2] == 4
```

kwimage.im_io.**_have_gdal**()

kwimage.im_runlen

Logic pertaining to run-length encodings

SeeAlso:

kwimage.structs.mask - stores binary segmentation masks, using RLEs as a backend representation. Also contains cython logic for handling the coco-rle format.

Module Contents

kwimage.im_runlen.**encode_run_length**(*img*, *binary=False*, *order='C'*)

Construct the run length encoding (RLE) of an image.

Parameters

- **img** (*ndarray*) – 2D image
- **binary** (*bool, default=True*) – set to True for compatibility with COCO
- **order** (*{'C', 'F'}, default='C'*) – row-major (C) or column-major (F)

Returns

encoding: dictionary items are: counts (ndarray): the run length encoding shape (Tuple): the original image shape binary (bool): if the counts encoding is binary or multiple values are ok order ({'C', 'F'}, default='C'): encoding order

Return type Dict[str, object]

SeeAlso:

- kwimage.Mask - a cython-backed data structure to handle coco-style RLEs

Example

```
>>> import ubelt as ub
>>> lines = ub.codeblock(
>>>     '''
>>>     ..........
>>>     ......111.
>>>     ..2...111.
>>>     .222..111.
>>>     22222.....
>>>     .222......
>>>     ..2.......
>>>     ''').replace('.', '0').splitlines()
>>> img = np.array([list(map(int, line)) for line in lines])
>>> encoding = encode_run_length(img)
>>> target = np.array([0,16,1,3,0,3,2,1,0,3,1,3,0,2,2,3,0,2,1,3,0,1,2,5,0,6,2,3,0,
↪8,2,1,0,7])
>>> assert np.all(target == encoding['counts'])
```

Example

```
>>> binary = True
>>> img = np.array([[1, 0, 1, 1, 1, 0, 0, 1, 0]])
>>> encoding = encode_run_length(img, binary=True)
>>> assert encoding['counts'].tolist() == [0, 1, 1, 3, 2, 1, 1]
```

kwimage.im_runlen.**decode_run_length**(*counts*, *shape*, *binary=False*, *dtype=np.uint8*, *order='C'*)

Decode run length encoding back into an image.

Parameters

- **counts** (*ndarray*) – the run-length encoding

- **shape** (*Tuple[int, int]*)
- **binary** (*bool*) – if the RLU is binary or non-binary. Set to True for compatibility with COCO.
- **dtype** (*dtype, default=np.uint8*) – data type for decoded image
- **order** (*{'C', 'F'}, default='C'*) – row-major (C) or column-major (F)

Returns the reconstructed image

Return type ndarray

Example

```
>>> from kwimage.im_runlen import *  # NOQA
>>> img = np.array([[1, 0, 1, 1, 1, 0, 0, 1, 0]])
>>> encoded = encode_run_length(img, binary=True)
>>> recon = decode_run_length(**encoded)
>>> assert np.all(recon == img)
```

```
>>> import ubelt as ub
>>> lines = ub.codeblock(
>>>     '''
>>>     .........
>>>     ......111.
>>>     ..2...111.
>>>     .222..111.
>>>     22222.....
>>>     .222......
>>>     ..2.......
>>>     ''').replace('.', '0').splitlines()
>>> img = np.array([list(map(int, line)) for line in lines])
>>> encoded = encode_run_length(img)
>>> recon = decode_run_length(**encoded)
>>> assert np.all(recon == img)
```

kwimage.im_runlen.**rle_translate**(*rle*, *offset*, *output_shape=None*)

Translates a run-length encoded image in RLE-space.

Parameters

- **rle** (*dict*) – an enconding dict returned by *encode_run_length*
- **offset** (*Tuple*) – x,y offset, CAREFUL, this can only accept integers
- **output_shape** (*Tuple, optional*) – h,w of transformed mask. If unspecified the input rle shape is used.

SeeAlso: # ITK has some RLE code that looks like it can perform translations https://github.com/KitwareMedical/ITKRLEImage/blob/master/include/itkRLERegionOfInterestImageFilter.h

Doctest:

```
>>> # test that translate works on all zero images
>>> img = np.zeros((7, 8), dtype=np.uint8)
>>> rle = encode_run_length(img, binary=True, order='F')
>>> new_rle = rle_translate(rle, (1, 2), (6, 9))
>>> assert np.all(new_rle['counts'] == [54])
```

Example

```
>>> from kwimage.im_runlen import *  # NOQA
>>> img = np.array([
>>>     [1, 1, 1, 1],
>>>     [0, 1, 0, 0],
>>>     [0, 1, 0, 1],
>>>     [1, 1, 1, 1],], dtype=np.uint8)
>>> rle = encode_run_length(img, binary=True, order='C')
>>> offset = (1, -1)
>>> output_shape = (3, 5)
>>> new_rle = rle_translate(rle, offset, output_shape)
>>> decoded = decode_run_length(**new_rle)
>>> print(decoded)
[[0 0 1 0 0]
 [0 0 1 0 1]
 [0 1 1 1 1]]
```

Example

```
>>> from kwimage.im_runlen import *  # NOQA
>>> img = np.array([
>>>     [0, 0, 0],
>>>     [0, 1, 0],
>>>     [0, 0, 0]], dtype=np.uint8)
>>> rle = encode_run_length(img, binary=True, order='C')
>>> new_rle = rle_translate(rle, (1, 0))
>>> decoded = decode_run_length(**new_rle)
>>> print(decoded)
[[0 0 0]
 [0 0 1]
 [0 0 0]]
>>> new_rle = rle_translate(rle, (0, 1))
>>> decoded = decode_run_length(**new_rle)
>>> print(decoded)
[[0 0 0]
 [0 0 0]
 [0 1 0]]
```

kwimage.im_runlen.**_rle_bytes_to_array**(*s*, *impl='auto'*)

Uncompresses a coco-bytes RLE into an array representation.

Parameters

- **s** (*bytes*) – compressed coco bytes rle
- **impl** (*str*) – which implementation to use (defaults to cython is possible)

CommandLine: xdoctest -m ~/code/kwimage/kwimage/im_runlen.py _rle_bytes_to_array

Benchmark:

```
>>> import ubelt as ub
>>> from kwimage.im_runlen import _rle_bytes_to_array
>>> s = b';?1B10O30O4'
>>> ti = ub.Timerit(1000, bestof=50, verbose=2)
>>> # --- time python impl ---
>>> for timer in ti.reset('python'):
```

(continues on next page)

(continued from previous page)

```
>>>     with timer:
>>>         _rle_bytes_to_array(s, impl='python')
>>> # --- time cython impl ---
>>> # xdoctest: +REQUIRES(--mask)
>>> for timer in ti.reset('cython'):
>>>     with timer:
>>>         _rle_bytes_to_array(s, impl='cython')
```

kwimage.im_runlen.**_rle_array_to_bytes**(*counts*, *impl='auto'*)

Compresses an array RLE into a coco-bytes RLE.

Parameters

- **counts** (*ndarray*) – uncompressed array rle
- **impl** (*str*) – which implementation to use (defaults to cython is possible)

Example

```
>>> # xdoctest: +REQUIRES(--mask)
>>> from kwimage.im_runlen import _rle_array_to_bytes
>>> from kwimage.im_runlen import _rle_bytes_to_array
>>> arr_counts = np.array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1])
>>> str_counts = _rle_array_to_bytes(arr_counts)
>>> arr_counts2 = _rle_bytes_to_array(str_counts)
>>> assert np.all(arr_counts2 == arr_counts)
```

Benchmark:

```
>>> # xdoctest: +REQUIRES(--mask)
>>> import ubelt as ub
>>> from kwimage.im_runlen import _rle_array_to_bytes
>>> from kwimage.im_runlen import _rle_bytes_to_array
>>> counts = np.array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1])
>>> ti = ub.Timerit(1000, bestof=50, verbose=2)
>>> # --- time python impl ---
>>> #for timer in ti.reset('python'):
>>> #    with timer:
>>> #        _rle_array_to_bytes(s, impl='python')
>>> # --- time cython impl ---
>>> for timer in ti.reset('cython'):
>>>     with timer:
>>>         _rle_array_to_bytes(s, impl='cython')
```

kwimage.im_stack

Stack images

Module Contents

kwimage.im_stack.**stack_images**(*images*, *axis=0*, *resize=None*, *interpolation=None*, *overlap=0*, *return_info=False*, *bg_value=None*)

Make a new image with the input images side-by-side

Parameters

- **images** (*Iterable[ndarray[ndim=2]]*) – image data
- **axis** (*int*) – axis to stack on (either 0 or 1)
- **resize** (*int, str, or None*) – if None image sizes are not modified, otherwise resize resize can be either 0 or 1. We resize the *resize*-th image to match the *1 - resize*-th image. Can also be strings "larger" or "smaller".
- **interpolation** (*int or str*) – string or cv2-style interpolation type. only used if resize or overlap > 0
- **overlap** (*int*) – number of pixels to overlap. Using a negative number results in a border.
- **return_info** (*bool*) – if True, returns transforms (scales and translations) to map from original image to its new location.

Returns

an image of stacked images side by side

OR

Tuple[ndarray, List]: where the first item is the aformentioned stacked image and the second item is a list of transformations for each input image mapping it to its location in the returned image.

Return type ndarray

Example

```
>>> import kwimage
>>> img1 = kwimage.grab_test_image('carl', space='rgb')
>>> img2 = kwimage.grab_test_image('astro', space='rgb')
>>> images = [img1, img2]
>>> imgB, transforms = stack_images(images, axis=0, resize='larger',
>>>                                 overlap=-10, return_info=True)
>>> print(imgB.shape)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> import kwimage
>>> kwplot.autompl()
>>> kwplot.imshow(imgB, colorspace='rgb')
>>> wh1 = np.multiply(img1.shape[0:2][::-1], transforms[0].scale)
>>> wh2 = np.multiply(img2.shape[0:2][::-1], transforms[1].scale)
>>> xoff1, yoff1 = transforms[0].translation
>>> xoff2, yoff2 = transforms[1].translation
>>> xywh1 = (xoff1, yoff1, wh1[0], wh1[1])
>>> xywh2 = (xoff2, yoff2, wh2[0], wh2[1])
>>> kwplot.draw_boxes(kwimage.Boxes([xywh1], 'xywh'), color=(1.0, 0, 0))
>>> kwplot.draw_boxes(kwimage.Boxes([xywh2], 'xywh'), color=(1.0, 0, 0))
>>> kwplot.show_if_requested()
((662, 512, 3), (0.0, 0.0), (0, 150))
```

kwimage.im_stack.**stack_images_grid**(*images*, *chunksize=None*, *axis=0*, *overlap=0*, *return_info=False*, *bg_value=None*)

Stacks images in a grid. Optionally return transforms of original image positions in the output image.

Parameters

- **images** (*Iterable[ndarray[ndim=2]]*) – image data
- **chunksize** (*int, default=None*) – number of rows per column or columns per row depending on the value of *axis*. If unspecified, computes this as *int(sqrt(len(images)))*.
- **axis** (*int, default=0*) – If 0, chunksize is columns per row. If 1, chunksize is rows per column.
- **overlap** (*int*) – number of pixels to overlap. Using a negative number results in a border.
- **return_info** (*bool*) – if True, returns transforms (scales and translations) to map from original image to its new location.

Returns

an image of stacked images in a grid pattern

OR

Tuple[ndarray, List]: where the first item is the aformentioned stacked image and the second item is a list of transformations for each input image mapping it to its location in the returned image.

Return type ndarray

kwimage.im_stack.**_stack_two_images**(*img1*, *img2*, *axis=0*, *resize=None*, *interpolation=None*, *overlap=0*, *bg_value=None*)

Returns imgB, offset_tup, sf_tup

Return type Tuple[ndarray, Tuple, Tuple]

Ignore: import xinspect globals().update(xinspect.get_func_kwargs(_stack_two_images)) resize = 1 overlap = -10

kwimage.util_warp

Module Contents

kwimage.util_warp.**TORCH_GRID_SAMPLE_HAS_ALIGN**

kwimage.util_warp.**_coordinate_grid**(*dims*, *align_corners=False*)

Creates a homogenous coordinate system.

Parameters

- **dims** (*Tuple[int]**) – height / width or depth / height / width
- **align_corners** (*bool*) – returns a grid where the left and right corners assigned to the extreme values and intermediate values are interpolated.

Returns Tensor[shape=(3, *DIMS)]

References

https://github.com/ClementPinard/SfmLearner-Pytorch/blob/master/inverse_warp.py

Example

```
>>> # xdoctest: +IGNORE_WHITESPACE
>>> _coordinate_grid((2, 2))
tensor([[[0., 1.],
         [0., 1.]],
        [[0., 0.],
         [1., 1.]],
        [[1., 1.],
         [1., 1.]]])
>>> _coordinate_grid((2, 2, 2))
>>> _coordinate_grid((2, 2), align_corners=True)
tensor([[[0., 2.],
         [0., 2.]],
        [[0., 0.],
         [2., 2.]],
        [[1., 1.],
         [1., 1.]]])
```

kwimage.util_warp.**warp_tensor**(*inputs*, *mat*, *output_dims*, *mode='bilinear'*, *padding_mode='zeros'*, *isinv=False*, *ishomog=None*, *align_corners=False*, *new_mode=False*)

A pytorch implementation of warp affine that works similarly to cv2.warpAffine / cv2.warpPerspective.

It is possible to use 3x3 transforms to warp 2D image data. It is also possible to use 4x4 transforms to warp 3D volumetric data.

Parameters

- **inputs** (*Tensor[..., *DIMS]*) – tensor to warp. Up to 3 (determined by output_dims) of the trailing space-time dimensions are warped. Best practice is to use inputs with the shape in [B, C, *DIMS].
- **mat** (*Tensor*) – either a 3x3 / 4x4 single transformation matrix to apply to all inputs or Bx3x3 or Bx4x4 tensor that specifies a transformation matrix for each batch item.
- **output_dims** (*Tuple[int]**) –

  The output space-time dimensions. This can either be in the form (W,), (H, W), or (D, H, W).
- **mode** (*str*) – Can be bilinear or nearest. See *torch.nn.functional.grid_sample*
- **padding_mode** (*str*) – Can be zeros, border, or reflection. See *torch.nn.functional.grid_sample*.
- **isinv** (*bool, default=False*) – Set to true if *mat* is the inverse transform
- **ishomog** (*bool, default=None*) – Set to True if the matrix is non-affine
- **align_corners** (*bool, default=False*) – Note the default of False does not work correctly with grid_sample in torch <= 1.2, but using align_corners=True isnt typically what you want either. We will be stuck with buggy functionality until torch 1.3 is released.

  However, using align_corners=0 does seem to reasonably correspond with opencv behavior.

Notes

Also, it may be possible to speed up the code with *F.affine_grid*

KNOWN ISSUE: There appears to some difference with cv2.warpAffine when rotation or shear are non-zero. I'm not sure what the cause is. It may just be floating point issues, but Im' not sure.

Todo:

- [ ] FIXME: see example in Mask.scale where this algo breaks when

the matrix is *2x3* - [ ] Make this algo work when matrix ix 2x2

References

https://discuss.pytorch.org/t/affine-transformation-matrix-paramters-conversion/19522 https://github.com/pytorch/pytorch/issues/15386

Example

```
>>> # Create a relatively simple affine matrix
>>> import skimage
>>> mat = torch.FloatTensor(skimage.transform.AffineTransform(
>>>     translation=[1, -1], scale=[.532, 2],
>>>     rotation=0, shear=0,
>>> ).params)
>>> # Create inputs and an output dimension
>>> input_shape = [1, 1, 4, 5]
>>> inputs = torch.arange(int(np.prod(input_shape))).reshape(*input_shape).float()
>>> output_dims = (11, 7)
>>> # Warp with our code
>>> result1 = warp_tensor(inputs, mat, output_dims=output_dims, align_corners=0)
>>> print('result1 =\n{}'.format(ub.repr2(result1.cpu().numpy()[0, 0],␣
↪precision=2)))
>>> # Warp with opencv
>>> import cv2
>>> cv2_M = mat.cpu().numpy()[0:2]
>>> src = inputs[0, 0].cpu().numpy()
>>> dsize = tuple(output_dims[::-1])
>>> result2 = cv2.warpAffine(src, cv2_M, dsize=dsize, flags=cv2.INTER_LINEAR)
>>> print('result2 =\n{}'.format(ub.repr2(result2, precision=2)))
>>> # Ensure the results are the same (up to floating point errors)
>>> assert np.all(np.isclose(result1[0, 0].cpu().numpy(), result2, atol=1e-2,␣
↪rtol=1e-2))
```

Example

```
>>> # Create a relatively simple affine matrix
>>> import skimage
>>> mat = torch.FloatTensor(skimage.transform.AffineTransform(
>>>     rotation=0.01, shear=0.1).params)
>>> # Create inputs and an output dimension
>>> input_shape = [1, 1, 4, 5]
>>> inputs = torch.arange(int(np.prod(input_shape))).reshape(*input_shape).float()
>>> output_dims = (11, 7)
>>> # Warp with our code
```

(continues on next page)

(continued from previous page)

```
>>> result1 = warp_tensor(inputs, mat, output_dims=output_dims)
>>> print('result1 =\n{}'.format(ub.repr2(result1.cpu().numpy()[0, 0],␣
↪precision=2, supress_small=True)))
>>> print('result1.shape = {}'.format(result1.shape))
>>> # Warp with opencv
>>> import cv2
>>> cv2_M = mat.cpu().numpy()[0:2]
>>> src = inputs[0, 0].cpu().numpy()
>>> dsize = tuple(output_dims[::-1])
>>> result2 = cv2.warpAffine(src, cv2_M, dsize=dsize, flags=cv2.INTER_LINEAR)
>>> print('result2 =\n{}'.format(ub.repr2(result2, precision=2)))
>>> print('result2.shape = {}'.format(result2.shape))
>>> # Ensure the results are the same (up to floating point errors)
>>> # NOTE: The floating point errors seem to be significant for rotation / shear
>>> assert np.all(np.isclose(result1[0, 0].cpu().numpy(), result2, atol=1,␣
↪rtol=1e-2))
```

Example

```
>>> # Create a random affine matrix
>>> import skimage
>>> rng = np.random.RandomState(0)
>>> mat = torch.FloatTensor(skimage.transform.AffineTransform(
>>>     translation=rng.randn(2), scale=1 + rng.randn(2),
>>>     rotation=rng.randn() / 10., shear=rng.randn() / 10.,
>>> ).params)
>>> # Create inputs and an output dimension
>>> input_shape = [1, 1, 5, 7]
>>> inputs = torch.arange(int(np.prod(input_shape))).reshape(*input_shape).float()
>>> output_dims = (3, 11)
>>> # Warp with our code
>>> result1 = warp_tensor(inputs, mat, output_dims=output_dims, align_corners=0)
>>> print('result1 =\n{}'.format(ub.repr2(result1.cpu().numpy()[0, 0],␣
↪precision=2)))
>>> # Warp with opencv
>>> import cv2
>>> cv2_M = mat.cpu().numpy()[0:2]
>>> src = inputs[0, 0].cpu().numpy()
>>> dsize = tuple(output_dims[::-1])
>>> result2 = cv2.warpAffine(src, cv2_M, dsize=dsize, flags=cv2.INTER_LINEAR)
>>> print('result2 =\n{}'.format(ub.repr2(result2, precision=2)))
>>> # Ensure the results are the same (up to floating point errors)
>>> # NOTE: The errors seem to be significant for rotation / shear
>>> assert np.all(np.isclose(result1[0, 0].cpu().numpy(), result2, atol=1,␣
↪rtol=1e-2))
```

Example

```
>>> # Test 3D warping with identity
>>> mat = torch.eye(4)
>>> input_dims = [2, 3, 3]
>>> output_dims = (2, 3, 3)
>>> input_shape = [1, 1] + input_dims
```

(continues on next page)

(continued from previous page)

```
>>> inputs = torch.arange(int(np.prod(input_shape))).reshape(*input_shape).float()
>>> result = warp_tensor(inputs, mat, output_dims=output_dims)
>>> print('result =\n{}'.format(ub.repr2(result.cpu().numpy()[0, 0],␣
↪precision=2)))
>>> assert torch.all(inputs == result)
```

Example

```
>>> # Test 3D warping with scaling
>>> mat = torch.FloatTensor([
>>>     [0.8,   0,   0, 0],
>>>     [  0, 1.0,   0, 0],
>>>     [  0,   0, 1.2, 0],
>>>     [  0,   0,   0, 1],
>>> ])
>>> input_dims = [2, 3, 3]
>>> output_dims = (2, 3, 3)
>>> input_shape = [1, 1] + input_dims
>>> inputs = torch.arange(int(np.prod(input_shape))).reshape(*input_shape).float()
>>> result = warp_tensor(inputs, mat, output_dims=output_dims, align_corners=0)
>>> print('result =\n{}'.format(ub.repr2(result.cpu().numpy()[0, 0],␣
↪precision=2)))
result =
np.array([[[ 0.  ,  1.25,  1.  ],
           [ 3.  ,  4.25,  2.5 ],
           [ 6.  ,  7.25,  4.  ]],
          ...
          [[ 7.5 ,  8.75,  4.75],
           [10.5 , 11.75,  6.25],
           [13.5 , 14.75,  7.75]]], dtype=np.float32)
```

Example

```
>>> mat = torch.eye(3)
>>> input_dims = [5, 7]
>>> output_dims = (11, 7)
>>> for n_prefix_dims in [0, 1, 2, 3, 4, 5]:
>>>     input_shape = [2] * n_prefix_dims + input_dims
>>>     inputs = torch.arange(int(np.prod(input_shape))).reshape(*input_shape).
↪float()
>>>     result = warp_tensor(inputs, mat, output_dims=output_dims)
>>>     #print('result =\n{}'.format(ub.repr2(result.cpu().numpy(),␣
↪precision=2)))
>>>     print(result.shape)
```

Example

```
>>> mat = torch.eye(4)
>>> input_dims = [5, 5, 5]
>>> output_dims = (6, 6, 6)
>>> for n_prefix_dims in [0, 1, 2, 3, 4, 5]:
```

(continues on next page)

(continued from previous page)

```
>>>      input_shape = [2] * n_prefix_dims + input_dims
>>>      inputs = torch.arange(int(np.prod(input_shape))).reshape(*input_shape).
↪float()
>>>      result = warp_tensor(inputs, mat, output_dims=output_dims)
>>>      #print('result =\n{}'.format(ub.repr2(result.cpu().numpy(),␣
↪precision=2)))
>>>      print(result.shape)
```

Ignore: import xdev globals().update(xdev.get_func_kwargs(warp_tensor)) >>> import cv2 >>> inputs = torch.arange(9).view(1, 1, 3, 3).float() + 2 >>> input_dims = inputs.shape[2:] >>> #output_dims = (6, 6) >>> def fmt(a): >>> return ub.repr2(a.numpy(), precision=2) >>> s = 2.5 >>> output_dims = tuple(np.round((np.array(input_dims) * s)).astype(np.int).tolist()) >>> mat = torch.FloatTensor([[s, 0, 0], [0, s, 0], [0, 0, 1]]) >>> inv = mat.inverse() >>> warp_tensor(inputs, mat, output_dims) >>> print('## INPUTS') >>> print(fmt(inputs)) >>> print('nalign_corners=True') >>> print('---') >>> print('## warp_tensor, align_corners=True') >>> print(fmt(warp_tensor(inputs, inv, output_dims, isinv=True, align_corners=True))) >>> print('## interpolate, align_corners=True') >>> print(fmt(F.interpolate(inputs, output_dims, mode='bilinear', align_corners=True))) >>> print('nalign_corners=False') >>> print('---') >>> print('## warp_tensor, align_corners=False, new_mode=False') >>> print(fmt(warp_tensor(inputs, inv, output_dims, isinv=True, align_corners=False))) >>> print('## warp_tensor, align_corners=False, new_mode=True') >>> print(fmt(warp_tensor(inputs, inv, output_dims, isinv=True, align_corners=False, new_mode=True))) >>> print('## interpolate, align_corners=False') >>> print(fmt(F.interpolate(inputs, output_dims, mode='bilinear', align_corners=False))) >>> print('## interpolate (scale), align_corners=False') >>> print(ub.repr2(F.interpolate(inputs, scale_factor=s, mode='bilinear', align_corners=False).numpy(), precision=2)) >>> cv2_M = mat.cpu().numpy()[0:2] >>> src = inputs[0, 0].cpu().numpy() >>> dsize = tuple(output_dims[::-1]) >>> print('nOpen CV warp Result') >>> result2 = (cv2.warpAffine(src, cv2_M, dsize=dsize, flags=cv2.INTER_LINEAR)) >>> print('result2 =n{}'.format(ub.repr2(result2, precision=2)))

kwimage.util_warp.**subpixel_align**(*dst*, *src*, *index*, *interp_axes=None*)

Returns an aligned version of the source tensor and destination index.

Used as the backend to implement other subpixel functions like: subpixel_accum, subpixel_maximum.

kwimage.util_warp.**subpixel_set**(*dst*, *src*, *index*, *interp_axes=None*)

Add the source values array into the destination array at a particular subpixel index.

Parameters

- **dst** (*ArrayLike*) – destination accumulation array
- **src** (*ArrayLike*) – source array containing values to add
- **index** (*Tuple[slice]*) – subpixel slice into dst that corresponds with src
- **interp_axes** (*tuple*) – specify which axes should be spatially interpolated

---

Todo:

- [ ]: allow index to be a sequence indices

Example

```
>>> import kwimage
>>> dst = np.zeros(5) + .1
>>> src = np.ones(2)
>>> index = [slice(1.5, 3.5)]
>>> kwimage.util_warp.subpixel_set(dst, src, index)
>>> print(ub.repr2(dst, precision=2, with_dtype=0))
np.array([0.1, 0.5, 1. , 0.5, 0.1])
```

kwimage.util_warp.**subpixel_accum**(*dst*, *src*, *index*, *interp_axes=None*)

Add the source values array into the destination array at a particular subpixel index.

Parameters

- **dst** (*ArrayLike*) – destination accumulation array
- **src** (*ArrayLike*) – source array containing values to add
- **index** (*Tuple[slice]*) – subpixel slice into dst that corresponds with src
- **interp_axes** (*tuple*) – specify which axes should be spatially interpolated

Notes

Inputs:

+—+—+—+—+—+—+ **dst.shape = (5,)** +—+—+ src.shape = (2,) |=======| index = 1.5:3.5

Subpixel shift the source by -0.5. When the index is non-integral, pad the aligned src with an extra value to ensure all dst pixels that would be influenced by the smaller subpixel shape are influenced by the aligned src. Note that we are not scaling.

+—+—+—+—+ aligned_src.shape = (3,) |==========| aligned_index = 1:4

Example

```
>>> dst = np.zeros(5)
>>> src = np.ones(2)
>>> index = [slice(1.5, 3.5)]
>>> subpixel_accum(dst, src, index)
>>> print(ub.repr2(dst, precision=2, with_dtype=0))
np.array([0. , 0.5, 1. , 0.5, 0. ])
```

Example

```
>>> dst = np.zeros((6, 6))
>>> src = np.ones((3, 3))
>>> index = (slice(1.5, 4.5), slice(1, 4))
>>> subpixel_accum(dst, src, index)
>>> print(ub.repr2(dst, precision=2, with_dtype=0))
np.array([[0. , 0. , 0. , 0. , 0. , 0. ],
          [0. , 0.5, 0.5, 0.5, 0. , 0. ],
          [0. , 1. , 1. , 1. , 0. , 0. ],
          [0. , 1. , 1. , 1. , 0. , 0. ],
```

(continues on next page)

(continued from previous page)

```
          [0. , 0.5, 0.5, 0.5, 0. , 0. ],
          [0. , 0. , 0. , 0. , 0. , 0. ]])
>>> dst = torch.zeros((1, 3, 6, 6))
>>> src = torch.ones((1, 3, 3, 3))
>>> index = (slice(None), slice(None), slice(1.5, 4.5), slice(1.25, 4.25))
>>> subpixel_accum(dst, src, index)
>>> print(ub.repr2(dst.numpy()[0, 0], precision=2, with_dtype=0))
np.array([[0.  , 0.  , 0.  , 0.  , 0.  , 0.  ],
          [0.  , 0.38, 0.5 , 0.5 , 0.12, 0.  ],
          [0.  , 0.75, 1.  , 1.  , 0.25, 0.  ],
          [0.  , 0.75, 1.  , 1.  , 0.25, 0.  ],
          [0.  , 0.38, 0.5 , 0.5 , 0.12, 0.  ],
          [0.  , 0.  , 0.  , 0.  , 0.  , 0.  ]])
```

Doctest:

```
>>> # TODO: move to a unit test file
>>> subpixel_accum(np.zeros(5), np.ones(2), [slice(1.5, 3.5)]).tolist()
[0.0, 0.5, 1.0, 0.5, 0.0]
>>> subpixel_accum(np.zeros(5), np.ones(2), [slice(0, 2)]).tolist()
[1.0, 1.0, 0.0, 0.0, 0.0]
>>> subpixel_accum(np.zeros(5), np.ones(3), [slice(.5, 3.5)]).tolist()
[0.5, 1.0, 1.0, 0.5, 0.0]
>>> subpixel_accum(np.zeros(5), np.ones(3), [slice(-1, 2)]).tolist()
[1.0, 1.0, 0.0, 0.0, 0.0]
>>> subpixel_accum(np.zeros(5), np.ones(3), [slice(-1.5, 1.5)]).tolist()
[1.0, 0.5, 0.0, 0.0, 0.0]
>>> subpixel_accum(np.zeros(5), np.ones(3), [slice(10, 13)]).tolist()
[0.0, 0.0, 0.0, 0.0, 0.0]
>>> subpixel_accum(np.zeros(5), np.ones(3), [slice(3.25, 6.25)]).tolist()
[0.0, 0.0, 0.0, 0.75, 1.0]
>>> subpixel_accum(np.zeros(5), np.ones(3), [slice(4.9, 7.9)]).tolist()
[0.0, 0.0, 0.0, 0.0, 0.099...]
>>> subpixel_accum(np.zeros(5), np.ones(9), [slice(-1.5, 7.5)]).tolist()
[1.0, 1.0, 1.0, 1.0, 1.0]
>>> subpixel_accum(np.zeros(5), np.ones(9), [slice(2.625, 11.625)]).tolist()
[0.0, 0.0, 0.375, 1.0, 1.0]
>>> subpixel_accum(np.zeros(5), 1, [slice(2.625, 11.625)]).tolist()
[0.0, 0.0, 0.375, 1.0, 1.0]
```

kwimage.util_warp.**subpixel_maximum**(*dst*, *src*, *index*, *interp_axes=None*)

Take the max of the source values array into and the destination array at a particular subpixel index. Modifies the destination array.

Parameters

- **dst** (*ArrayLike*) – destination array to index into
- **src** (*ArrayLike*) – source array that agrees with the index
- **index** (*Tuple[slice]*) – subpixel slice into dst that corresponds with src
- **interp_axes** (*tuple*) – specify which axes should be spatially interpolated

Example

```
>>> dst = np.array([0, 1.0, 1.0, 1.0, 0])
>>> src = np.array([2.0, 2.0])
>>> index = [slice(1.6, 3.6)]
>>> subpixel_maximum(dst, src, index)
>>> print(ub.repr2(dst, precision=2, with_dtype=0))
np.array([0. , 1. , 2. , 1.2, 0. ])
```

Example

```
>>> dst = torch.zeros((1, 3, 5, 5)) + .5
>>> src = torch.ones((1, 3, 3, 3))
>>> index = (slice(None), slice(None), slice(1.4, 4.4), slice(1.25, 4.25))
>>> subpixel_maximum(dst, src, index)
>>> print(ub.repr2(dst.numpy()[0, 0], precision=2, with_dtype=0))
np.array([[0.5 , 0.5 , 0.5 , 0.5 , 0.5 ],
          [0.5 , 0.5 , 0.6 , 0.6 , 0.5 ],
          [0.5 , 0.75, 1.  , 1.  , 0.5 ],
          [0.5 , 0.75, 1.  , 1.  , 0.5 ],
          [0.5 , 0.5 , 0.5 , 0.5 , 0.5 ]])
```

kwimage.util_warp.**subpixel_minimum**(*dst*, *src*, *index*, *interp_axes=None*)

Take the min of the source values array into and the destination array at a particular subpixel index. Modifies the destination array.

Parameters

- **dst** (*ArrayLike*) – destination array to index into
- **src** (*ArrayLike*) – source array that agrees with the index
- **index** (*Tuple[slice]*) – subpixel slice into dst that corresponds with src
- **interp_axes** (*tuple*) – specify which axes should be spatially interpolated

Example

```
>>> dst = np.array([0, 1.0, 1.0, 1.0, 0])
>>> src = np.array([2.0, 2.0])
>>> index = [slice(1.6, 3.6)]
>>> subpixel_minimum(dst, src, index)
>>> print(ub.repr2(dst, precision=2, with_dtype=0))
np.array([0. , 0.8, 1. , 1. , 0. ])
```

Example

```
>>> dst = torch.zeros((1, 3, 5, 5)) + .5
>>> src = torch.ones((1, 3, 3, 3))
>>> index = (slice(None), slice(None), slice(1.4, 4.4), slice(1.25, 4.25))
>>> subpixel_minimum(dst, src, index)
>>> print(ub.repr2(dst.numpy()[0, 0], precision=2, with_dtype=0))
np.array([[0.5 , 0.5 , 0.5 , 0.5 , 0.5 ],
          [0.5 , 0.45, 0.5 , 0.5 , 0.15],
```

(continues on next page)

(continued from previous page)

```
          [0.5 , 0.5 , 0.5 , 0.5 , 0.25],
          [0.5 , 0.5 , 0.5 , 0.5 , 0.25],
          [0.5 , 0.3 , 0.4 , 0.4 , 0.1 ]])
```

kwimage.util_warp.**subpixel_slice**(*inputs*, *index*)

Take a subpixel slice from a larger image. The returned output is left-aligned with the requested slice.

Parameters

- **inputs** (*ArrayLike*) – data
- **index** (*Tuple[slice]*) – a slice to subpixel accuracy

Example

```
>>> inputs = np.arange(5 * 5 * 3).reshape(5, 5, 3)
>>> index = [slice(0, 3), slice(0, 3)]
>>> outputs = subpixel_slice(inputs, index)
>>> index = [slice(0.5, 3.5), slice(-0.5, 2.5)]
>>> outputs = subpixel_slice(inputs, index)
```

```
>>> inputs = np.arange(5 * 5).reshape(1, 5, 5).astype(np.float)
>>> index = [slice(None), slice(3, 6), slice(3, 6)]
>>> outputs = subpixel_slice(inputs, index)
>>> print(outputs)
[[[18. 19.  0.]
  [23. 24.  0.]
  [ 0.  0.  0.]]]
>>> index = [slice(None), slice(3.5, 6.5), slice(2.5, 5.5)]
>>> outputs = subpixel_slice(inputs, index)
>>> print(outputs)
[[[20.   21.   10.75]
  [11.25 11.75  6.  ]
  [ 0.    0.    0.  ]]]
```

kwimage.util_warp.**subpixel_translate**(*inputs*, *shift*, *interp_axes=None*, *output_shape=None*)

Translates an image by a subpixel shift value using bilinear interpolation

Parameters

- **inputs** (*ArrayLike*) – data to translate
- **shift** (*Sequence*) – amount to translate each dimension specified by *interp_axes*. Note: if inputs contains more than one "image" then all "images" are translated by the same amount. This function contains no mechanism for translating each image differently. Note that by default this is a y,x shift for 2 dimensions.
- **interp_axes** (*Sequence, default=None*) – axes to perform interpolation on, if not specified the final *n* axes are interpolated, where *n=len(shift)*
- **output_shape** (*tuple, default=None*) – if specified the output is returned with this shape, otherwise

Notes

This function powers most other functions in this file. Speedups here can go a long way.

Example

```
>>> inputs = np.arange(5) + 1
>>> print(inputs.tolist())
[1, 2, 3, 4, 5]
>>> outputs = subpixel_translate(inputs, 1.5)
>>> print(outputs.tolist())
[0.0, 0.5, 1.5, 2.5, 3.5]
```

Example

```
>>> inputs = torch.arange(9).view(1, 1, 3, 3).float()
>>> print(inputs.long())
tensor([[[[0, 1, 2],
          [3, 4, 5],
          [6, 7, 8]]]])
>>> outputs = subpixel_translate(inputs, (-.4, .5), output_shape=(1, 1, 2, 5))
>>> print(outputs)
tensor([[[[0.6000, 1.7000, 2.7000, 1.6000, 0.0000],
          [2.1000, 4.7000, 5.7000, 3.1000, 0.0000]]]])
```

Ignore:

```
>>> inputs = np.arange(5)
>>> shift = -.6
>>> interp_axes = None
>>> subpixel_translate(inputs, -.6)
>>> subpixel_translate(inputs[None, None, None, :], -.6)
>>> inputs = np.arange(25).reshape(5, 5)
>>> shift = (-1.6, 2.3)
>>> interp_axes = (0, 1)
>>> subpixel_translate(inputs, shift, interp_axes, output_shape=(9, 9))
>>> subpixel_translate(inputs, shift, interp_axes, output_shape=(3, 4))
```

kwimage.util_warp.**_padded_slice**(*data*, *in_slice*, *ndim=None*, *pad_slice=None*, *pad_mode='constant'*, ***padkw*)

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

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.

Parameters

- **data** (*Sliceable[T]*) – data to slice into. Any channels must be the last dimension.
- **in_slice** (*Tuple[slice, ...]*) – slice for each dimensions
- **ndim** (*int*) – number of spatial dimensions
- **pad_slice** (*List[int|Tuple]*) – additional padding of the slice

Returns

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

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

Return type Tuple[Sliceable, List]

Example

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

```
>>> data_sliced, st_dims = _padded_slice(data, in_slice)
>>> print(ub.repr2(data_sliced, with_dtype=False))
>>> print(st_dims)
np.array([0, 0, 0, 1, 2, 3, 4, 0, 0])
[(-2, 7)]
```

```
>>> data_sliced, st_dims = _padded_slice(data, in_slice, pad_slice=(3, 3))
>>> print(ub.repr2(data_sliced, with_dtype=False))
>>> print(st_dims)
np.array([0, 0, 0, 0, 0, 0, 1, 2, 3, 4, 0, 0, 0, 0, 0])
[(-5, 10)]
```

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

kwimage.util_warp.**_ensure_arraylike**(*data*, *n=None*)

kwimage.util_warp.**_rectify_slice**(*data_dims*, *low_dims*, *high_dims*, *pad_slice=None*)

Given image dimensions, bounding box dimensions, and a padding get the corresponding slice from the image and any extra padding needed to achieve the requested window size.

Parameters

- **data_dims** (*tuple*) – n-dimension data sizes (e.g. 2d height, width)
- **low_dims** (*tuple*) – bounding box low values (e.g. 2d ymin, xmin)
- **high_dims** (*tuple*) – bounding box high values (e.g. 2d ymax, xmax)
- **pad_slice** (*List[int|Tuple]*) – pad applied to (left and right) / (both) sides of each slice dim

Returns

data_slice - low and high values of a fancy slice corresponding to the image with shape *data_dims*. This slice may not correspond to the full window size if the requested bounding box goes out of bounds.

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

Return type Tuple

Example

```
>>> # Case where 2D-bbox is inside the data dims on left edge
>>> # Comprehensive 1D-cases are in the unit-test file
>>> data_dims  = [300, 300]
>>> low_dims   = [0, 0]
>>> high_dims  = [10, 10]
>>> pad_slice  = [(10, 10), (5, 5)]
>>> a, b = _rectify_slice(data_dims, low_dims, high_dims, pad_slice)
>>> print('data_slice = {!r}'.format(a))
>>> print('extra_padding = {!r}'.format(b))
data_slice = [(0, 20), (0, 15)]
extra_padding = [(10, 0), (5, 0)]
```

kwimage.util_warp.**_warp_tensor_cv2**(*inputs*, *mat*, *output_dims*, *mode='linear'*, *ishomog=None*)

implementation with cv2.warpAffine for speed / correctness comparison

On GPU: torch is faster in both modes On CPU: torch is faster for homog, but cv2 is faster for affine

Benchmark:

```
>>> from kwimage.util.util_warp import *
>>> from kwimage.util.util_warp import _warp_tensor_cv2
>>> from kwimage.util.util_warp import warp_tensor
>>> import numpy as np
>>> ti = ub.Timerit(10, bestof=3, verbose=2, unit='ms')
>>> mode = 'linear'
>>> rng = np.random.RandomState(0)
>>> inputs = torch.Tensor(rng.rand(16, 10, 32, 32)).to('cpu')
>>> mat = torch.FloatTensor([[2.5, 0, 10.5], [0, 3, 0], [0, 0, 1]])
>>> mat[2, 0] = .009
>>> mat[2, 2] = 2
>>> output_dims = (64, 64)
>>> results = ub.odict()
>>> # -------------
>>> for timer in ti.reset('warp_tensor(torch)'):
>>>     with timer:
>>>         outputs = warp_tensor(inputs, mat, output_dims, mode=mode)
>>>         torch.cuda.synchronize()
>>> results[ti.label] = outputs
>>> # -------------
>>> inputs = inputs.cpu().numpy()
>>> mat = mat.cpu().numpy()
>>> for timer in ti.reset('warp_tensor(cv2)'):
>>>     with timer:
>>>         outputs = _warp_tensor_cv2(inputs, mat, output_dims, mode=mode)
>>> results[ti.label] = outputs
>>> import itertools as it
>>> for k1, k2 in it.combinations(results, 2):
>>>     a = kwarray.ArrayAPI.numpy(results[k1])
>>>     b = kwarray.ArrayAPI.numpy(results[k2])
>>>     diff = np.abs(a - b)
>>>     diff_stats = kwarray.stats_dict(diff, n_extreme=1, extreme=1)
>>>     print('{} - {}: {}'.format(k1, k2, ub.repr2(diff_stats, nl=0,␣
↪precision=4)))
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
```

(continues on next page)

(continued from previous page)

```
>>> kwplot.imshow(results['warp_tensor(torch)'][0, 0], fnum=1, pnum=(1, 2, 1),
↪ title='torch')
>>> kwplot.imshow(results['warp_tensor(cv2)'][0, 0], fnum=1, pnum=(1, 2, 2),␣
↪title='cv2')
```

kwimage.util_warp.**warp_points**(*matrix*, *pts*)

Warp ND points / coordinates using a transformation matrix.

Homogoenous coordinates are added on the fly if needed. Works with both numpy and torch.

Parameters

- **matrix** (*ArrayLike*) – [D1 x D2] transformation matrix. if using homogenous coordinates D2=D + 1, otherwise D2=D. if using homogenous coordinates and the matrix represents an Affine transformation, then either D1=D or D1=D2, i.e. the last row of zeros and a one is optional.
- **pts** (*ArrayLike*) – [N1 x ... x D] points (usually x, y). If points are already in homogenous space, then the output will be returned in homogenous space. D is the dimensionality of the points. The leading axis may take any shape, but usually, shape will be [N x D] where N is the number of points.

Retrns: new_pts (ArrayLike): the points after being transformed by the matrix

Example

```
>>> from kwimage.util_warp import *  # NOQA
>>> # --- with numpy
>>> rng = np.random.RandomState(0)
>>> pts = rng.rand(10, 2)
>>> matrix = rng.rand(2, 2)
>>> warp_points(matrix, pts)
>>> # --- with torch
>>> pts = torch.Tensor(pts)
>>> matrix = torch.Tensor(matrix)
>>> warp_points(matrix, pts)
```

Example

```
>>> from kwimage.util_warp import *  # NOQA
>>> # --- with numpy
>>> pts = np.ones((10, 2))
>>> matrix = np.diag([2, 3, 1])
>>> ra = warp_points(matrix, pts)
>>> rb = warp_points(torch.Tensor(matrix), torch.Tensor(pts))
>>> assert np.allclose(ra, rb.numpy())
```

Example

```
>>> from kwimage.util_warp import *  # NOQA
>>> # test different cases
>>> rng = np.random.RandomState(0)
```

(continues on next page)

(continued from previous page)

```
>>> # Test 3x3 style projective matrices
>>> pts = rng.rand(1000, 2)
>>> matrix = rng.rand(3, 3)
>>> ra33 = warp_points(matrix, pts)
>>> rb33 = warp_points(torch.Tensor(matrix), torch.Tensor(pts))
>>> assert np.allclose(ra33, rb33.numpy())
>>> # Test opencv style affine matrices
>>> pts = rng.rand(10, 2)
>>> matrix = rng.rand(2, 3)
>>> ra23 = warp_points(matrix, pts)
>>> rb23 = warp_points(torch.Tensor(matrix), torch.Tensor(pts))
>>> assert np.allclose(ra33, rb33.numpy())
```

kwimage.util_warp.**subpixel_getvalue**(*img*, *pts*, *coord_axes=None*, *interp='bilinear'*, *bordermode='edge'*)

Get values at subpixel locations

Parameters

- **img** (*ArrayLike*) – image to sample from
- **pts** (*ArrayLike*) – subpixel rc-coordinates to sample
- **coord_axes** (*Sequence, default=None*) – axes to perform interpolation on, if not specified the first *d* axes are interpolated, where *d=pts.shape[-1]*. IE: this indicates which axes each coordinate dimension corresponds to.
- **interp** (*str*) – interpolation mode
- **bordermode** (*str*) – how locations outside the image are handled

Example

```
>>> from kwimage.util_warp import *  # NOQA
>>> img = np.arange(3 * 3).reshape(3, 3)
>>> pts = np.array([[1, 1], [1.5, 1.5], [1.9, 1.1]])
>>> subpixel_getvalue(img, pts)
array([4. , 6. , 6.8])
>>> subpixel_getvalue(img, pts, coord_axes=(1, 0))
array([4. , 6. , 5.2])
>>> img = torch.Tensor(img)
>>> pts = torch.Tensor(pts)
>>> subpixel_getvalue(img, pts)
tensor([4.0000, 6.0000, 6.8000])
>>> subpixel_getvalue(img.numpy(), pts.numpy(), interp='nearest')
array([4., 8., 7.], dtype=float32)
>>> subpixel_getvalue(img.numpy(), pts.numpy(), interp='nearest', coord_axes=[1,␣
↪0])
array([4., 8., 5.], dtype=float32)
>>> subpixel_getvalue(img, pts, interp='nearest')
tensor([4., 8., 7.])
```

References

stackoverflow.com/uestions/12729228/simple-binlin-interp-images-numpy

SeeAlso: cv2.getRectSubPix(image, patchSize, center[, patch[, patchType]])

kwimage.util_warp.**subpixel_setvalue**(*img*, *pts*, *value*, *coord_axes=None*, *interp='bilinear'*, *bordermode='edge'*)

Set values at subpixel locations

Parameters

- **img** (*ArrayLike*) – image to set values in
- **pts** (*ArrayLike*) – subpixel rc-coordinates to set
- **value** (*ArrayLike*) – value to place in the image
- **coord_axes** (*Sequence, default=None*) – axes to perform interpolation on, if not specified the first *d* axes are interpolated, where *d=pts.shape[-1]*. IE: this indicates which axes each coordinate dimension corresponds to.
- **interp** (*str*) – interpolation mode
- **bordermode** (*str*) – how locations outside the image are handled

Example

```
>>> from kwimage.util_warp import *  # NOQA
>>> img = np.arange(3 * 3).reshape(3, 3).astype(np.float)
>>> pts = np.array([[1, 1], [1.5, 1.5], [1.9, 1.1]])
>>> interp = 'bilinear'
>>> value = 0
>>> print('img = {!r}'.format(img))
>>> pts = np.array([[1.5, 1.5]])
>>> img2 = subpixel_setvalue(img.copy(), pts, value)
>>> print('img2 = {!r}'.format(img2))
>>> pts = np.array([[1.0, 1.0]])
>>> img2 = subpixel_setvalue(img.copy(), pts, value)
>>> print('img2 = {!r}'.format(img2))
>>> pts = np.array([[1.1, 1.9]])
>>> img2 = subpixel_setvalue(img.copy(), pts, value)
>>> print('img2 = {!r}'.format(img2))
>>> img2 = subpixel_setvalue(img.copy(), pts, value, coord_axes=[1, 0])
>>> print('img2 = {!r}'.format(img2))
```

kwimage.util_warp.**_bilinear_coords**(*ptsT*, *impl*, *img*, *coord_axes*)

Package Contents

kwimage.**available_nms_impls**()

List available values for the *impl* kwarg of *non_max_supression*

CommandLine: xdoctest -m kwimage.algo.algo_nms available_nms_impls

Example

```
>>> impls = available_nms_impls()
>>> assert 'numpy' in impls
>>> print('impls = {!r}'.format(impls))
```

kwimage.**daq_spatial_nms**(*tlbr*, *scores*, *diameter*, *thresh*, *max_depth=6*, *stop_size=2048*, *recsize=2048*, *impl='auto'*, *device_id=None*)

Divide and conquor speedup non-max-supression algorithm for when bboxes have a known max size

Parameters

- **tlbr** (*ndarray*) – boxes in (tlx, tly, brx, bry) format
- **scores** (*ndarray*) – scores of each box
- **diameter** (*int or Tuple[int, int]*) – Distance from split point to consider rectification. If specified as an integer, then number is used for both height and width. If specified as a tuple, then dims are assumed to be in [height, width] format.
- **thresh** (*float*) – iou threshold. Boxes are removed if they overlap greater than this threshold. 0 is the most strict, resulting in the fewest boxes, and 1 is the most permissive resulting in the most.
- **max_depth** (*int*) – maximum number of times we can divide and conquor
- **stop_size** (*int*) – number of boxes that triggers full NMS computation
- **recsize** (*int*) – number of boxes that triggers full NMS recombination
- **impl** (*str*) – algorithm to use

LookInfo: # Didn't read yet but it seems similar http://www.cyberneum.de/fileadmin/user_upload/files/publications/CVPR2010-Lampert_{[}0{]}.pdf

https://www.researchgate.net/publication/220929789_Efficient_Non-Maximum_Suppression

This seems very similar https://projet.liris.cnrs.fr/m2disco/pub/Congres/2006-ICPR/DATA/C03_0406.PDF

Example

```
>>> import kwimage
>>> # Make a bunch of boxes with the same width and height
>>> #boxes = kwimage.Boxes.random(230397, scale=1000, format='cxywh')
>>> boxes = kwimage.Boxes.random(237, scale=1000, format='cxywh')
>>> boxes.data.T[2] = 10
>>> boxes.data.T[3] = 10
>>> #
>>> tlbr = boxes.to_tlbr().data.astype(np.float32)
>>> scores = np.arange(0, len(tlbr)).astype(np.float32)
>>> #
>>> n_megabytes = (tlbr.size * tlbr.dtype.itemsize) / (2 ** 20)
>>> print('n_megabytes = {!r}'.format(n_megabytes))
>>> #
>>> thresh = iou_thresh = 0.01
>>> impl = 'auto'
>>> max_depth = 20
>>> diameter = 10
>>> stop_size = 2000
>>> recsize = 500
>>> #
>>> import ubelt as ub
>>> #
>>> with ub.Timer(label='daq'):
>>>     keep1 = daq_spatial_nms(tlbr, scores,
```

(continues on next page)

(continued from previous page)

```
>>>         diameter=diameter, thresh=thresh, max_depth=max_depth,
>>>         stop_size=stop_size, recsize=recsize, impl=impl)
>>> #
>>> with ub.Timer(label='full'):
>>>     keep2 = non_max_supression(tlbr, scores,
>>>         thresh=thresh, impl=impl)
>>> #
>>> # Due to the greedy nature of the algorithm, there will be slight
>>> # differences in results, but they will be mostly similar.
>>> similarity = len(set(keep1) & set(keep2)) / len(set(keep1) | set(keep2))
>>> print('similarity = {!r}'.format(similarity))
```

kwimage.**non_max_supression**(*tlbr*, *scores*, *thresh*, *bias=0.0*, *classes=None*, *impl='auto'*, *device_id=None*)

Non-Maximum Suppression - remove redundant bounding boxes

Parameters

- **tlbr** (*ndarray[float32]*) – Nx4 boxes in tlbr format
- **scores** (*ndarray[float32]*) – score for each bbox
- **thresh** (*float*) – iou threshold. Boxes are removed if they overlap greater than this threshold (i.e. Boxes are removed if iou > threshold). Thresh = 0 is the most strict, resulting in the fewest boxes, and 1 is the most permissive resulting in the most.
- **bias** (*float*) – bias for iou computation either 0 or 1
- **classes** (*ndarray[int64] or None*) – integer classes. If specified NMS is done on a perclass basis.
- **impl** (*str*) – implementation can be auto, python, cython_cpu, or gpu
- **device_id** (*int*) – used if impl is gpu, device id to work on. If not specified *torch.cuda.current_device()* is used.

Notes

Using impl='cython_gpu' may result in an CUDA memory error that is not exposed to the python processes. In other words your program will hard crash if impl='cython_gpu', and you feed it too many bounding boxes. Ideally this will be fixed in the future.

References

https://github.com/facebookresearch/Detectron/blob/master/detectron/utils/cython_nms.pyx https://www.pyimagesearch.com/2015/02/16/faster-non-maximum-suppression-python/ https://github.com/bharatsingh430/soft-nms/blob/master/lib/nms/cpu_nms.pyx <- TODO

CommandLine: xdoctest -m ~/code/kwimage/kwimage/algo/algo_nms.py non_max_supression

Example

```
>>> from kwimage.algo.algo_nms import *
>>> from kwimage.algo.algo_nms import _impls
>>> tlbr = np.array([
```

(continues on next page)

(continued from previous page)

```
>>>     [0, 0, 100, 100],
>>>     [100, 100, 10, 10],
>>>     [10, 10, 100, 100],
>>>     [50, 50, 100, 100],
>>> ], dtype=np.float32)
>>> scores = np.array([.1, .5, .9, .1])
>>> keep = non_max_supression(tlbr, scores, thresh=0.5, impl='numpy')
>>> print('keep = {!r}'.format(keep))
>>> assert keep == [2, 1, 3]
>>> thresh = 0.0
>>> non_max_supression(tlbr, scores, thresh, impl='numpy')
>>> if 'numpy' in available_nms_impls():
>>>     keep = non_max_supression(tlbr, scores, thresh, impl='numpy')
>>>     assert list(keep) == [2, 1]
>>> if 'cython_cpu' in available_nms_impls():
>>>     keep = non_max_supression(tlbr, scores, thresh, impl='cython_cpu')
>>>     assert list(keep) == [2, 1]
>>> if 'cython_gpu' in available_nms_impls():
>>>     keep = non_max_supression(tlbr, scores, thresh, impl='cython_gpu')
>>>     assert list(keep) == [2, 1]
>>> if 'torch' in available_nms_impls():
>>>     keep = non_max_supression(tlbr, scores, thresh, impl='torch')
>>>     assert set(keep.tolist()) == {2, 1}
>>> if 'torchvision' in available_nms_impls():
>>>     keep = non_max_supression(tlbr, scores, thresh, impl='torchvision')  #␣
↪note torchvision has no bias
>>>     assert list(keep) == [2]
>>> thresh = 1.0
>>> if 'numpy' in available_nms_impls():
>>>     keep = non_max_supression(tlbr, scores, thresh, impl='numpy')
>>>     assert list(keep) == [2, 1, 3, 0]
>>> if 'cython_cpu' in available_nms_impls():
>>>     keep = non_max_supression(tlbr, scores, thresh, impl='cython_cpu')
>>>     assert list(keep) == [2, 1, 3, 0]
>>> if 'cython_gpu' in available_nms_impls():
>>>     keep = non_max_supression(tlbr, scores, thresh, impl='cython_gpu')
>>>     assert list(keep) == [2, 1, 3, 0]
>>> if 'torch' in available_nms_impls():
>>>     keep = non_max_supression(tlbr, scores, thresh, impl='torch')
>>>     assert set(keep.tolist()) == {2, 1, 3, 0}
>>> if 'torchvision' in available_nms_impls():
>>>     keep = non_max_supression(tlbr, scores, thresh, impl='torchvision')  #␣
↪note torchvision has no bias
>>>     assert set(kwarray.ArrayAPI.tolist(keep)) == {2, 1, 3, 0}
```

Example

```
>>> import ubelt as ub
>>> tlbr = np.array([
>>>     [0, 0, 100, 100],
>>>     [100, 100, 10, 10],
>>>     [10, 10, 100, 100],
>>>     [50, 50, 100, 100],
>>>     [100, 100, 150, 101],
>>>     [120, 100, 180, 101],
```

(continues on next page)

(continued from previous page)

```
>>>     [150, 100, 200, 101],
>>> ], dtype=np.float32)
>>> scores = np.linspace(0, 1, len(tlbr))
>>> thresh = .2
>>> solutions = {}
>>> if not _impls._funcs:
>>>     _impls._lazy_init()
>>> for impl in _impls._funcs:
>>>     keep = non_max_supression(tlbr, scores, thresh, impl=impl)
>>>     solutions[impl] = sorted(keep)
>>> assert 'numpy' in solutions
>>> print('solutions = {}'.format(ub.repr2(solutions, nl=1)))
>>> assert ub.allsame(solutions.values())
```

CommandLine: xdoctest -m ~/code/kwimage/kwimage/algo/algo_nms.py non_max_supression

Example

```
>>> import ubelt as ub
>>> # Check that zero-area boxes are ok
>>> tlbr = np.array([
>>>     [0, 0, 0, 0],
>>>     [0, 0, 0, 0],
>>>     [10, 10, 10, 10],
>>> ], dtype=np.float32)
>>> scores = np.array([1, 2, 3], dtype=np.float32)
>>> thresh = .2
>>> solutions = {}
>>> if not _impls._funcs:
>>>     _impls._lazy_init()
>>> for impl in _impls._funcs:
>>>     keep = non_max_supression(tlbr, scores, thresh, impl=impl)
>>>     solutions[impl] = sorted(keep)
>>> assert 'numpy' in solutions
>>> print('solutions = {}'.format(ub.repr2(solutions, nl=1)))
>>> assert ub.allsame(solutions.values())
```

kwimage.**ensure_alpha_channel**(*img*, *alpha=1.0*, *dtype=np.float32*, *copy=False*)

Returns the input image with 4 channels.

Parameters

- **img** (*ndarray*) – an image with shape [H, W], [H, W, 1], [H, W, 3], or [H, W, 4].
- **alpha** (*float, default=1.0*) – default value for missing alpha channel
- **dtype** (*type, default=np.float32*) – a numpy floating type
- **copy** (*bool, default=False*) – always copy if True, else copy if needed.

Returns an image with specified dtype with shape [H, W, 4].

Raises *ValueError - if the input image does not have 1, 3, or 4 input channels* – or if the image cannot be converted into a float01 representation

kwimage.**overlay_alpha_images**(*img1*, *img2*, *keepalpha=True*, *dtype=np.float32*, *impl='inplace'*)

Places img1 on top of img2 respecting alpha channels. Works like the Photoshop layers with opacity.

Parameters

- **img1** (*ndarray*) – top image to overlay over img2
- **img2** (*ndarray*) – base image to superimpose on
- **keepalpha** (*bool*) – if False, the alpha channel is removed after blending
- **dtype** (*np.dtype*) – format for blending computation (defaults to float32)
- **impl** (*str, default=inplace*) – code specifying the backend implementation

Returns raster: the blended images

Return type ndarray

Todo:

- [ ] Make fast C++ version of this function

References

http://stackoverflow.com/questions/25182421/overlay-numpy-alpha https://en.wikipedia.org/wiki/Alpha_compositing#Alpha_blending

Example

```
>>> import kwimage
>>> img1 = kwimage.grab_test_image('astro', dsize=(100, 100))
>>> img2 = kwimage.grab_test_image('carl', dsize=(100, 100))
>>> img1 = kwimage.ensure_alpha_channel(img1, alpha=.5)
>>> img3 = overlay_alpha_images(img1, img2)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(img3)
>>> kwplot.show_if_requested()
```

kwimage.**overlay_alpha_layers**(*layers*, *keepalpha=True*, *dtype=np.float32*)

Stacks a sequences of layers on top of one another. The first item is the topmost layer and the last item is the bottommost layer.

Parameters

- **layers** (*Sequence[ndarray]*) – stack of images
- **keepalpha** (*bool*) – if False, the alpha channel is removed after blending
- **dtype** (*np.dtype*) – format for blending computation (defaults to float32)

Returns raster: the blended images

Return type ndarray

References

http://stackoverflow.com/questions/25182421/overlay-numpy-alpha https://en.wikipedia.org/wiki/Alpha_compositing#Alpha_blending

Example

```
>>> import kwimage
>>> keys = ['astro', 'carl', 'stars']
>>> layers = [kwimage.grab_test_image(k, dsize=(100, 100)) for k in keys]
>>> layers = [kwimage.ensure_alpha_channel(g, alpha=.5) for g in layers]
>>> stacked = overlay_alpha_layers(layers)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(stacked)
>>> kwplot.show_if_requested()
```

kwimage.**BASE_COLORS**

kwimage.**CSS4_COLORS**

class kwimage.**Color**(*color*, *alpha=None*, *space=None*)

Bases: ubelt.NiceRepr

Used for converting a single color between spaces and encodings. This should only be used when handling small numbers of colors(e.g. 1), don't use this to represent an image.

move to colorutil?

Parameters **space** (*str*) – colorspace of wrapped color. Assume RGB if not specified and it cannot be inferred

CommandLine: xdoctest -m ~/code/kwimage/kwimage/im_color.py Color

Example

```
>>> print(Color('g'))
>>> print(Color('orangered'))
>>> print(Color('#AAAAAA').as255())
>>> print(Color([0, 255, 0]))
>>> print(Color([1, 1, 1.]))
>>> print(Color([1, 1, 1]))
>>> print(Color(Color([1, 1, 1])).as255())
>>> print(Color(Color([1., 0, 1, 0])).ashex())
>>> print(Color([1, 1, 1], alpha=255))
>>> print(Color([1, 1, 1], alpha=255, space='lab'))
```

__nice__(*self*)

_forimage(*self*, *image*, *space='rgb'*)

Experimental function.

Create a numeric color tuple that agrees with the format of the input image (i.e. float or int, with 3 or 4 channels).

Parameters

- **image** (*ndarray*) – image to return color for
- **space** (*str, default=rgb*) – colorspace of the input image.

Example

```
>>> img_f3 = np.zeros([8, 8, 3], dtype=np.float32)
>>> img_u3 = np.zeros([8, 8, 3], dtype=np.uint8)
>>> img_f4 = np.zeros([8, 8, 4], dtype=np.float32)
>>> img_u4 = np.zeros([8, 8, 4], dtype=np.uint8)
>>> Color('red')._forimage(img_f3)
(1.0, 0.0, 0.0)
>>> Color('red')._forimage(img_f4)
(1.0, 0.0, 0.0, 1.0)
>>> Color('red')._forimage(img_u3)
(255, 0, 0)
>>> Color('red')._forimage(img_u4)
(255, 0, 0, 255)
>>> Color('red', alpha=0.5)._forimage(img_f4)
(1.0, 0.0, 0.0, 0.5)
>>> Color('red', alpha=0.5)._forimage(img_u4)
(255, 0, 0, 127)
```

ashex(*self*, *space=None*)

as255(*self*, *space=None*)

as01(*self*, *space=None*)
self = mplutil.Color('red') mplutil.Color('green').as01('rgba')

classmethod _is_base01(*channels*)
check if a color is in base 01

classmethod _is_base255(*Color*, *channels*)
there is a one corner case where all pixels are 1 or less

classmethod _hex_to_01(*Color*, *hex_color*)
hex_color = '#6A5AFFAF'

_ensure_color01(*Color*, *color*)
Infer what type color is and normalize to 01

classmethod _255_to_01(*Color*, *color255*)
converts base 255 color to base 01 color

classmethod _string_to_01(*Color*, *color*)
mplutil.Color._string_to_01('green') mplutil.Color._string_to_01('red')

classmethod named_colors(*cls*)

Returns names of colors that Color accepts

Return type List[str]

classmethod distinct(*Color*, *num*, *space='rgb'*)
Make multiple distinct colors

classmethod random(*Color*, *pool='named'*)

kwimage.**TABLEAU_COLORS**

kwimage.**XKCD_COLORS**

kwimage.**atleast_3channels**(*arr*, *copy=True*)
Ensures that there are 3 channels in the image

Parameters

- **arr** (*ndarray[N, M, ...]*) – the image
- **copy** (*bool*) – Always copies if True, if False, then copies only when the size of the array must change.

Returns with shape (N, M, C), where C in {3, 4}

Return type ndarray

Doctest:

```
>>> assert atleast_3channels(np.zeros((10, 10))).shape[-1] == 3
>>> assert atleast_3channels(np.zeros((10, 10, 1))).shape[-1] == 3
>>> assert atleast_3channels(np.zeros((10, 10, 3))).shape[-1] == 3
>>> assert atleast_3channels(np.zeros((10, 10, 4))).shape[-1] == 4
```

kwimage.**ensure_float01**(*img*, *dtype=np.float32*, *copy=True*)

Ensure that an image is encoded using a float32 properly

Parameters

- **img** (*ndarray*) – an image in uint255 or float01 format. Other formats will raise errors.
- **dtype** (*type, default=np.float32*) – a numpy floating type
- **copy** (*bool, default=False*) – always copy if True, else copy if needed.

Returns an array of floats in the range 0-1

Return type ndarray

Raises `ValueError` – if the image type is integer and not in [0-255]

Example

```
>>> ensure_float01(np.array([[0, .5, 1.0]]))
array([[0. , 0.5, 1. ]], dtype=float32)
>>> ensure_float01(np.array([[0, 1, 200]]))
array([[0..., 0.0039..., 0.784...]], dtype=float32)
```

kwimage.**ensure_uint255**(*img*, *copy=True*)

Ensure that an image is encoded using a uint8 properly. Either

Parameters

- **img** (*ndarray*) – an image in uint255 or float01 format. Other formats will raise errors.
- **copy** (*bool, default=False*) – always copy if True, else copy if needed.

Returns an array of bytes in the range 0-255

Return type ndarray

Raises

- `ValueError` – if the image type is float and not in [0-1]
- `ValueError` – if the image type is integer and not in [0-255]

Example

```
>>> ensure_uint255(np.array([[0, .5, 1.0]]))
array([[  0, 127, 255]], dtype=uint8)
>>> ensure_uint255(np.array([[0, 1, 200]]))
array([[  0,   1, 200]], dtype=uint8)
```

kwimage.**make_channels_comparable**(*img1*, *img2*, *atleast3d=False*)

Broadcasts image arrays so they can have elementwise operations applied

Parameters

- **img1** (*ndarray*) – first image
- **img2** (*ndarray*) – second image
- **atleast3d** (*bool, default=False*) – if true we ensure that the channel dimension exists (only relevant for 1-channel images)

Example

```
>>> import itertools as it
>>> wh_basis = [(5, 5), (3, 5), (5, 3), (1, 1), (1, 3), (3, 1)]
>>> for w, h in wh_basis:
>>>     shape_basis = [(w, h), (w, h, 1), (w, h, 3)]
>>>     # Test all permutations of shap inputs
>>>     for shape1, shape2 in it.product(shape_basis, shape_basis):
>>>         print('*    input shapes: %r, %r' % (shape1, shape2))
>>>         img1 = np.empty(shape1)
>>>         img2 = np.empty(shape2)
>>>         img1, img2 = make_channels_comparable(img1, img2)
>>>         print('... output shapes: %r, %r' % (img1.shape, img2.shape))
>>>         elem = (img1 + img2)
>>>         print('... elem(+) shape: %r' % (elem.shape,))
>>>         assert elem.size == img1.size, 'outputs should have same size'
>>>         assert img1.size == img2.size, 'new imgs should have same size'
>>>         print('--------')
```

kwimage.**num_channels**(*img*)

Returns the number of color channels in an image

Parameters **img** (*ndarray*) – an image with 2 or 3 dimensions.

Returns the number of color channels (1, 3, or 4)

Return type int

Example

```
>>> H = W = 3
>>> assert num_channels(np.empty((W, H))) == 1
>>> assert num_channels(np.empty((W, H, 1))) == 1
>>> assert num_channels(np.empty((W, H, 3))) == 3
>>> assert num_channels(np.empty((W, H, 4))) == 4
>>> # xdoctest: +REQUIRES(module:pytest)
>>> import pytest
```

(continues on next page)

(continued from previous page)

```
>>> with pytest.raises(ValueError):
...     num_channels(np.empty((W, H, 2)))
```

kwimage.**convert_colorspace**(*img*, *src_space*, *dst_space*, *copy=False*, *implicit=False*, *dst=None*)

Converts colorspace of img. Convinience function around cv2.cvtColor

Parameters

- **img** (*ndarray*) – image data with float32 or uint8 precision
- **src_space** (*str*) – input image colorspace. (e.g. BGR, GRAY)
- **dst_space** (*str*) – desired output colorspace. (e.g. RGB, HSV, LAB)
- **implicit** (*bool*) –

  if False, the user must correctly specify if the input/output colorspaces contain alpha channels.

  If True and the input image has an alpha channel, we modify src_space and dst_space to ensure they both end with "A".
- **dst** (*ndarray[uint8_t, ndim=2], optional*) – inplace-output array.

Returns img - image data

Return type ndarray

Note: Note the LAB and HSV colorspaces in float do not go into the 0-1 range.

For HSV the floating point range is: 0:360, 0:1, 0:1

For LAB the floating point range is: 0:100, -86.1875:98.234375, -107.859375:94.46875 (Note, that some extreme combinations of a and b are not valid)

Example

```
>>> import numpy as np
>>> convert_colorspace(np.array([[[0, 0, 1]]], dtype=np.float32), 'RGB', 'LAB')
>>> convert_colorspace(np.array([[[0, 1, 0]]], dtype=np.float32), 'RGB', 'LAB')
>>> convert_colorspace(np.array([[[1, 0, 0]]], dtype=np.float32), 'RGB', 'LAB')
>>> convert_colorspace(np.array([[[1, 1, 1]]], dtype=np.float32), 'RGB', 'LAB')
>>> convert_colorspace(np.array([[[0, 0, 1]]], dtype=np.float32), 'RGB', 'HSV')
```

Ignore: # Check LAB output ranges import itertools as it s = 1 _iter = it.product(range(0, 256, s), range(0, 256, s), range(0, 256, s)) minvals = np.full(3, np.inf) maxvals = np.full(3, -np.inf) for r, g, b in ub.ProgIter(_iter, total=(256 // s) ** 3):

    img255 = np.array([[[r, g, b]]], dtype=np.uint8) img01 = (img255 / 255.0).astype(np.float32) lab = convert_colorspace(img01, 'rgb', 'lab') np.minimum(lab[0, 0], minvals, out=minvals) np.maximum(lab[0, 0], maxvals, out=maxvals)

print('minvals = {}'.format(ub.repr2(minvals, nl=0))) print('maxvals = {}'.format(ub.repr2(maxvals, nl=0)))

kwimage.**gaussian_patch**(*shape=(7, 7)*, *sigma=None*)

Creates a 2D gaussian patch with a specific size and sigma

Parameters

- **shape** (*Tuple[int, int]*) – patch height and width
- **sigma** (*float | Tuple[float, float]*) – gaussian standard deviation

References

http://docs.opencv.org/modules/imgproc/doc/filtering.html#getgaussiankernel

---

Todo:

- [ ] Look into this C-implementation

https://kwgitlab.kitware.com/computer-vision/heatmap/blob/master/heatmap/heatmap.c

---

CommandLine: xdoctest -m kwimage.im_cv2 gaussian_patch –show

Example

```
>>> import numpy as np
>>> shape = (88, 24)
>>> sigma = None  # 1.0
>>> gausspatch = gaussian_patch(shape, sigma)
>>> sum_ = gausspatch.sum()
>>> assert np.all(np.isclose(sum_, 1.0))
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> norm = (gausspatch - gausspatch.min()) / (gausspatch.max() - gausspatch.min())
>>> kwplot.imshow(norm)
>>> kwplot.show_if_requested()
```

Example

```
>>> import numpy as np
>>> shape = (24, 24)
>>> sigma = 3.0
>>> gausspatch = gaussian_patch(shape, sigma)
>>> sum_ = gausspatch.sum()
>>> assert np.all(np.isclose(sum_, 1.0))
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> norm = (gausspatch - gausspatch.min()) / (gausspatch.max() - gausspatch.min())
>>> kwplot.imshow(norm)
>>> kwplot.show_if_requested()
```

kwimage.**imresize**(*img*, *scale=None*, *dsize=None*, *max_dim=None*, *min_dim=None*, *interpolation=None*, *letterbox=False*, *return_info=False*)

Resize an image based on a scale factor, final size, or size and aspect ratio.

Slightly more general than cv2.resize and kwimage.imscale, allows for specification of either a scale factor, a final size, or the final size for a particular dimension.

Parameters

- **img** (*ndarray*) – image to resize
- **scale** (*float or Tuple[float, float]*) – desired floating point scale factor. If a tuple, the dimension ordering is x,y. Mutually exclusive with dsize, max_dim, and min_dim.
- **dsize** (*Tuple[None | int, None | int]*) – the desired with and height of the new image. If a dimension is None, then it is automatically computed to preserve aspect ratio. Mutually exclusive with size, max_dim, and min_dim.
- **max_dim** (*int*) – new size of the maximum dimension, the other dimension is scaled to maintain aspect ratio. Mutually exclusive with size, dsize, and min_dim.
- **min_dim** (*int*) – new size of the minimum dimension, the other dimension is scaled to maintain aspect ratio.Mutually exclusive with size, dsize, and max_dim.
- **interpolation** (*str | int*) – interpolation key or code (e.g. linear lanczos)
- **letterbox** (*bool, default=False*) – if used in conjunction with dsize, then the image is scaled and translated to fit in the center of the new image while maintaining aspect ratio. Black padding is added if necessary.
- **return_info** (*bool, default=False*) – if True returns information about the final transformation in a dictionary.

Returns the new image and optionally an info dictionary

Return type ndarray | Tuple[ndarray, Dict]

Example

```
>>> import kwimage
>>> import numpy as np
>>> # Test scale
>>> img = np.zeros((16, 10, 3), dtype=np.uint8)
>>> new_img, info = kwimage.imresize(img, scale=.85,
>>>                                  interpolation='area',
>>>                                  return_info=True)
>>> print('info = {!r}'.format(info))
>>> assert info['scale'].tolist() == [.8, 0.875]
>>> # Test dsize without None
>>> new_img, info = kwimage.imresize(img, dsize=(5, 12),
>>>                                  interpolation='area',
>>>                                  return_info=True)
>>> print('info = {!r}'.format(info))
>>> assert info['scale'].tolist() == [0.5 , 0.75]
>>> # Test dsize with None
>>> new_img, info = kwimage.imresize(img, dsize=(6, None),
>>>                                  interpolation='area',
>>>                                  return_info=True)
>>> print('info = {!r}'.format(info))
>>> assert info['scale'].tolist() == [0.6, 0.625]
>>> # Test max_dim
>>> new_img, info = kwimage.imresize(img, max_dim=6,
>>>                                  interpolation='area',
>>>                                  return_info=True)
>>> print('info = {!r}'.format(info))
>>> assert info['scale'].tolist() == [0.4  , 0.375]
```

(continues on next page)

(continued from previous page)

```
>>> # Test min_dim
>>> new_img, info = kwimage.imresize(img, min_dim=6,
>>>                                  interpolation='area',
>>>                                  return_info=True)
>>> print('info = {!r}'.format(info))
>>> assert info['scale'].tolist() == [0.6  , 0.625]
```

Example

```
>>> import kwimage
>>> import numpy as np
>>> # Test letterbox resize
>>> img = np.ones((5, 10, 3), dtype=np.float32)
>>> new_img, info = kwimage.imresize(img, dsize=(19, 19),
>>>                                  letterbox=True,
>>>                                  return_info=True)
>>> print('info = {!r}'.format(info))
>>> assert info['offset'].tolist() == [0, 4]
>>> img = np.ones((10, 5, 3), dtype=np.float32)
>>> new_img, info = kwimage.imresize(img, dsize=(19, 19),
>>>                                  letterbox=True,
>>>                                  return_info=True)
>>> print('info = {!r}'.format(info))
>>> assert info['offset'].tolist() == [4, 0]
```

```
>>> import kwimage
>>> import numpy as np
>>> # Test letterbox resize
>>> img = np.random.rand(100, 200)
>>> new_img, info = kwimage.imresize(img, dsize=(300, 300), letterbox=True,␣
↪return_info=True)
```

kwimage.**imscale**(*img*, *scale*, *interpolation=None*, *return_scale=False*)

Resizes an image by a scale factor.

Because the result image must have an integer number of pixels, the scale factor is rounded, and the rounded scale factor is optionaly returned.

Parameters

- **img** (*ndarray*) – image to resize
- **scale** (*float or Tuple[float, float]*) – desired floating point scale factor. If a tuple, the dimension ordering is x,y.
- **interpolation** (*str | int*) – interpolation key or code (e.g. linear lanczos)
- **return_scale** (*bool, default=False*) – if True returns both the new image and the actual scale factor used to achive the new integer image size.

SeeAlso: imresize

Example

```
>>> import kwimage
>>> import numpy as np
>>> img = np.zeros((10, 10, 3), dtype=np.uint8)
>>> new_img, new_scale = kwimage.imscale(img, scale=.85,
>>>                                      interpolation='nearest',
>>>                                      return_scale=True)
>>> assert new_scale == (.8, .8)
>>> assert new_img.shape == (8, 8, 3)
```

kwimage.**grab_test_image**(*key='astro'*, *space='rgb'*, *dsize=None*, *interpolation='lanczos'*)

Ensures that the test image exists (this might use the network), reads it and returns the the image pixels.

Parameters

- **key** (*str*) – which test image to grab. Valid choices are: astro - an astronaught carl - Carl Sagan paraview - ParaView logo stars - picture of stars in the sky airport - SkySat image of Beijing Capital International Airport on 18 February 2018
- **space** (*str, default='rgb'*) – which colorspace to return in
- **dsize** (*Tuple[int, int], default=None*) – if specified resizes image to this size

Returns the requested image

Return type ndarray

CommandLine: xdoctest -m ~/code/kwimage/kwimage/im_demodata.py grab_test_image

Example

```
>>> for key in grab_test_image.keys():
...     grab_test_image(key)
>>> grab_test_image('astro', dsize=(255, 255)).shape
(255, 255, 3)
```

kwimage.**grab_test_image_fpath**(*key='astro'*)

Ensures that the test image exists (this might use the network) and returns the cached filepath to the requested image.

Parameters **key** (*str*) – which test image to grab. Valid choices are: astro - an astronaught carl - Carl Sagan paraview - ParaView logo stars - picture of stars in the sky

Returns path to the requested image

Return type str

Example

```
>>> for key in grab_test_image.keys():
...     grab_test_image_fpath(key)
```

kwimage.**draw_boxes_on_image**(*img*, *boxes*, *color='blue'*, *thickness=1*, *box_format=None*, *colorspace='rgb'*)

Draws boxes on an image.

Parameters

- **img** (*ndarray*) – image to copy and draw on
- **boxes** (*nh.util.Boxes*) – boxes to draw
- **colorspace** (*str*) – string code of the input image colorspace

Example

```
>>> import kwimage
>>> import numpy as np
>>> img = np.zeros((10, 10, 3), dtype=np.uint8)
>>> color = 'dodgerblue'
>>> thickness = 1
>>> boxes = kwimage.Boxes([[1, 1, 8, 8]], 'tlbr')
>>> img2 = draw_boxes_on_image(img, boxes, color, thickness)
>>> assert tuple(img2[1, 1]) == (30, 144, 255)
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()  # xdoc: +SKIP
>>> kwplot.figure(doclf=True, fnum=1)
>>> kwplot.imshow(img2)
```

kwimage.**draw_clf_on_image**(*im*, *classes*, *tcx*, *probs=None*, *pcx=None*, *border=1*)

Draws classification label on an image

Parameters

- **im** (*ndarray*) – the image
- **classes** (*Sequence | CategoryTree*) – list of class names
- **tcx** (*int*) – true class index
- **probs** (*ndarray*) – predicted class probs for each class
- **pcx** (*int*) – predicted class index. (if none uses argmax of probs)

Example

```
>>> import torch
>>> import kwarray
>>> import kwimage
>>> rng = kwarray.ensure_rng(0)
>>> im = (rng.rand(300, 300) * 255).astype(np.uint8)
>>> classes = ['cls_a', 'cls_b', 'cls_c']
>>> tcx = 1
>>> probs = rng.rand(len(classes))
>>> probs[tcx] = 0
>>> probs = torch.FloatTensor(probs).softmax(dim=0).numpy()
>>> im1_ = kwimage.draw_clf_on_image(im, classes, tcx, probs)
>>> probs[tcx] = .9
>>> probs = torch.FloatTensor(probs).softmax(dim=0).numpy()
>>> im2_ = kwimage.draw_clf_on_image(im, classes, tcx, probs)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(im1_, colorspace='rgb', pnum=(1, 2, 1), fnum=1, doclf=True)
```

(continues on next page)

(continued from previous page)

```
>>> kwplot.imshow(im2_, colorspace='rgb', pnum=(1, 2, 2), fnum=1)
>>> kwplot.show_if_requested()
```

kwimage.**draw_text_on_image**(*img*, *text*, *org*, ***kwargs*)

Draws multiline text on an image using opencv

Note: This function also exists in kwplot

The image is modified inplace. If the image is non-contiguous then this returns a UMat instead of a ndarray, so be carefull with that.

Parameters

- **img** (*ndarray*) – image to draw on (inplace)
- **text** (*str*) – text to draw
- **org** (*tuple*) – x, y location of the text string in the image. if bottomLeftOrigin=True this is the bottom-left corner of the text otherwise it is the top-left corner (default).
- ****kwargs** – color (tuple): default blue thickneess (int): defaults to 2 fontFace (int): defaults to cv2.FONT_HERSHEY_SIMPLEX fontScale (float): defaults to 1.0 valign (str, default=bottom): either top, center, or bottom

References

https://stackoverflow.com/questions/27647424/

Example

```
>>> import kwimage
>>> img = kwimage.grab_test_image(space='rgb')
>>> img2 = kwimage.draw_text_on_image(img.copy(), 'FOOBAR', org=(0, 0), valign=
↪'top')
>>> assert img2.shape == img.shape
>>> assert np.any(img2 != img)
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(img2, fontScale=10)
>>> kwplot.show_if_requested()
```

Example

```
>>> import kwimage
>>> img = kwimage.grab_test_image(space='rgb')
>>> img2 = kwimage.draw_text_on_image(img, 'FOOBAR\nbazbiz\nspam', org=(0, 0),␣
↪valign='top', border=2)
>>> img2 = kwimage.draw_text_on_image(img, 'FOOBAR\nbazbiz\nspam', org=(150, 0),␣
↪valign='center', border=2)
>>> img2 = kwimage.draw_text_on_image(img, 'FOOBAR\nbazbiz\nspam', org=(300, 0),␣
↪valign='bottom', border=2)
```

(continues on next page)

(continued from previous page)

```
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(img2, fontScale=10)
>>> kwplot.show_if_requested()
```

Example

```
>>> # Ensure the function works with float01 or uint255 images
>>> import kwimage
>>> img = kwimage.grab_test_image(space='rgb')
>>> img = kwimage.ensure_float01(img)
>>> img2 = kwimage.draw_text_on_image(img, 'FOOBAR\nbazbiz\nspam', org=(0, 0),␣
↪valign='top', border=2)
```

kwimage.**make_heatmask**(*probs*, *cmap='plasma'*, *with_alpha=1.0*, *space='rgb'*, *dsize=None*)

Colorizes a single-channel intensity mask (with an alpha channel)

Parameters

- **probs** (*ndarray*) – 2D probability map with values between 0 and 1
- **cmap** (*str*) – mpl colormap
- **with_alpha** (*float*) – between 0 and 1, uses probs as the alpha multipled by this number.
- **space** (*str*) – output colorspace
- **dsize** (*tuple*) – if not None, then output is resized to W,H=dsize

SeeAlso: kwimage.overlay_alpha_images

Example

```
>>> # xdoc: +REQUIRES(module:matplotlib)
>>> probs = np.tile(np.linspace(0, 1, 10), (10, 1))
>>> heatmask = make_heatmask(probs, with_alpha=0.8, dsize=(100, 100))
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.imshow(heatmask, fnum=1, doclf=True, colorspace='rgb')
>>> kwplot.show_if_requested()
```

kwimage.**make_orimask**(*radians*, *mag=None*, *alpha=1.0*)

Makes a colormap in HSV space where the orientation changes color and mag changes the saturation/value.

Parameters

- **radians** (*ndarray*) – orientation in radians
- **mag** (*ndarray*) – magnitude (must be normalized between 0 and 1)
- **alpha** (*float | ndarray*) – if False or None, then the image is returned without alpha if a float, then mag is scaled by this and used as the alpha channel if an ndarray, then this is explicilty set as the alpha channel

Returns an rgb / rgba image in 01 space

Return type ndarray[float32]

SeeAlso: kwimage.overlay_alpha_images

Example

```
>>> # xdoc: +REQUIRES(module:matplotlib)
>>> x, y = np.meshgrid(np.arange(64), np.arange(64))
>>> dx, dy = x - 32, y - 32
>>> radians = np.arctan2(dx, dy)
>>> mag = np.sqrt(dx ** 2 + dy ** 2)
>>> orimask = make_orimask(radians, mag)
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.imshow(orimask, fnum=1, doclf=True, colorspace='rgb')
>>> kwplot.show_if_requested()
```

kwimage.**make_vector_field**(*dx*, *dy*, *stride=1*, *thresh=0.0*, *scale=1.0*, *alpha=1.0*, *color='red'*, *thickness=1*, *tipLength=0.1*, *line_type='aa'*)

Create an image representing a 2D vector field.

Parameters

- **dx** (*ndarray*) – grid of vector x components
- **dy** (*ndarray*) – grid of vector y components
- **stride** (*int*) – sparsity of vectors
- **thresh** (*float*) – only plot vectors with magnitude greater than thres
- **scale** (*float*) – multiply magnitude for easier visualization
- **alpha** (*float*) – alpha value for vectors. Non-vector regions receive 0 alpha (if False, no alpha channel is used)
- **color** (*str | tuple | kwimage.Color*) – RGB color of the vectors
- **thickness** (*int, default=1*) – thickness of arrows
- **tipLength** (*float, default=0.1*) – fraction of line length
- **line_type** (*int*) – either cv2.LINE_4, cv2.LINE_8, or cv2.LINE_AA

Returns vec_img: an rgb/rgba image in 0-1 space

Return type ndarray[float32]

SeeAlso: kwimage.overlay_alpha_images

Example

```
>>> x, y = np.meshgrid(np.arange(512), np.arange(512))
>>> dx, dy = x - 256.01, y - 256.01
>>> radians = np.arctan2(dx, dy)
>>> mag = np.sqrt(dx ** 2 + dy ** 2)
>>> dx, dy = dx / mag, dy / mag
>>> img = make_vector_field(dx, dy, stride=10, scale=10, alpha=False)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
```

(continues on next page)

(continued from previous page)

```
>>> kwplot.imshow(img)
>>> kwplot.show_if_requested()
```

kwimage.**imread**(*fpath*, *space='auto'*, *backend='auto'*)

Reads image data in a specified format using some backend implementation.

Parameters

- **fpath** (*str*) – path to the file to be read
- **space** (*str, default='auto'*) – the desired colorspace of the image. Can by any colorspace accepted by *convert_colorspace*, or it can be 'auto', in which case the colorspace of the image is unmodified (except in the case where a color image is read by opencv, in which case we convert BGR to RGB by default). If None, then no modification is made to whaveter backend is used to read the image.
- **backend** (*str, default='auto'*) – which backend reader to use. By default the file extension is used to determine this, but it can be manually overridden. Valid backends are gdal, skimage, and cv2.

Note: if space is something non-standard like HSV or LAB, then the file must be a normal 8-bit color image, otherwise an error will occur.

Raises

- IOError - If the image cannot be read
- ImportError - If trying to read a nitf without gdal
- NotImplementedError - if trying to read a corner-case image

Example

```
>>> # xdoctest: +REQUIRES(--network)
>>> from kwimage.im_io import *  # NOQA
>>> import tempfile
>>> from os.path import splitext  # NOQA
>>> # Test a non-standard image, which encodes a depth map
>>> fpath = ub.grabdata('http://www.topcoder.com/contest/problem/UrbanMapper3D/
↪JAX_Tile_043_DTM.tif')
>>> img1 = imread(fpath)
>>> # Check that write + read preserves data
>>> tmp = tempfile.NamedTemporaryFile(suffix=splitext(fpath)[1])
>>> imwrite(tmp.name, img1)
>>> img2 = imread(tmp.name)
>>> assert np.all(img2 == img1)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(img1, pnum=(1, 2, 1), fnum=1)
>>> kwplot.imshow(img2, pnum=(1, 2, 2), fnum=1)
```

Example

```
>>> # xdoctest: +REQUIRES(--network)
>>> import tempfile
>>> img1 = imread(ub.grabdata('http://i.imgur.com/iXNf4Me.png', fname='ada.png'))
>>> tmp_tif = tempfile.NamedTemporaryFile(suffix='.tif')
>>> tmp_png = tempfile.NamedTemporaryFile(suffix='.png')
>>> imwrite(tmp_tif.name, img1)
>>> imwrite(tmp_png.name, img1)
>>> tif_im = imread(tmp_tif.name)
>>> png_im = imread(tmp_png.name)
>>> assert np.all(tif_im == png_im)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(png_im, pnum=(1, 2, 1), fnum=1)
>>> kwplot.imshow(tif_im, pnum=(1, 2, 2), fnum=1)
```

Example

```
>>> # xdoctest: +REQUIRES(--network)
>>> import tempfile
>>> tif_fpath = ub.grabdata('https://ghostscript.com/doc/tiff/test/images/rgb-3c-
↪16b.tiff', fname='pepper.tif'),
>>> img1 = imread(tif_fpath)
>>> tmp_tif = tempfile.NamedTemporaryFile(suffix='.tif')
>>> tmp_png = tempfile.NamedTemporaryFile(suffix='.png')
>>> imwrite(tmp_tif.name, img1)
>>> imwrite(tmp_png.name, img1)
>>> tif_im = imread(tmp_tif.name)
>>> png_im = imread(tmp_png.name)
>>> assert np.all(tif_im == png_im)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(png_im / 2 ** 16, pnum=(1, 2, 1), fnum=1)
>>> kwplot.imshow(tif_im / 2 ** 16, pnum=(1, 2, 2), fnum=1)
```

kwimage.**imwrite**(*fpath*, *image*, *space='auto'*)

Writes image data to disk.

Parameters

- **fpath** (*PathLike*) – location to save the imaeg
- **image** (*ndarray*) – image data
- **space** (*str*) – the colorspace of the image to save. Can by any colorspace accepted by *convert_colorspace*, or it can be 'auto', in which case we assume the input image is either RGB, RGBA or grayscale. If None, then absolutely no color modification is made and whatever backend is used writes the image as-is.

Notes

The image may be modified to preserve its colorspace depending on which backend is used to write the image.

Raises `Exception` – if the image cannot be written

Doctest:

```
>>> # xdoctest: +REQUIRES(--network)
>>> # This should be moved to a unit test
>>> import tempfile
>>> test_image_paths = [
>>>     ub.grabdata('https://ghostscript.com/doc/tiff/test/images/rgb-3c-16b.
↪tiff', fname='pepper.tif'),
>>>     ub.grabdata('http://i.imgur.com/iXNf4Me.png', fname='ada.png'),
>>>     #ub.grabdata('http://www.topcoder.com/contest/problem/UrbanMapper3D/
↪JAX_Tile_043_DTM.tif'),
>>>     ub.grabdata('https://upload.wikimedia.org/wikipedia/commons/f/fa/
↪Grayscale_8bits_palette_sample_image.png', fname='parrot.png')
>>> ]
>>> for fpath in test_image_paths:
>>>     for space in ['auto', 'rgb', 'bgr', 'gray', 'rgba']:
>>>         img1 = imread(fpath, space=space)
>>>         print('Test im-io consistency of fpath = {!r} in {} space, shape=
↪{}'.format(fpath, space, img1.shape))
>>>         # Write the image in TIF and PNG format
>>>         tmp_tif = tempfile.NamedTemporaryFile(suffix='.tif')
>>>         tmp_png = tempfile.NamedTemporaryFile(suffix='.png')
>>>         imwrite(tmp_tif.name, img1, space=space)
>>>         imwrite(tmp_png.name, img1, space=space)
>>>         tif_im = imread(tmp_tif.name, space=space)
>>>         png_im = imread(tmp_png.name, space=space)
>>>         assert np.all(tif_im == png_im), 'im-read/write inconsistency'
>>>         if space == 'gray':
>>>             assert tif_im.ndim == 2
>>>             assert png_im.ndim == 2
>>>         elif space in ['rgb', 'bgr']:
>>>             assert tif_im.shape[2] == 3
>>>             assert png_im.shape[2] == 3
>>>         elif space in ['rgba', 'bgra']:
>>>             assert tif_im.shape[2] == 4
>>>             assert png_im.shape[2] == 4
```

kwimage.**decode_run_length**(*counts*, *shape*, *binary=False*, *dtype=np.uint8*, *order='C'*)

Decode run length encoding back into an image.

Parameters

- **counts** (*ndarray*) – the run-length encoding
- **shape** (*Tuple[int, int]*)
- **binary** (*bool*) – if the RLU is binary or non-binary. Set to True for compatibility with COCO.
- **dtype** (*dtype, default=np.uint8*) – data type for decoded image
- **order** (*{'C', 'F'}, default='C'*) – row-major (C) or column-major (F)

Returns the reconstructed image

Return type ndarray

Example

```
>>> from kwimage.im_runlen import *  # NOQA
>>> img = np.array([[1, 0, 1, 1, 1, 0, 0, 1, 0]])
>>> encoded = encode_run_length(img, binary=True)
>>> recon = decode_run_length(**encoded)
>>> assert np.all(recon == img)
```

```
>>> import ubelt as ub
>>> lines = ub.codeblock(
>>>     '''
>>>     .........
>>>     ......111.
>>>     ..2...111.
>>>     .222..111.
>>>     22222.....
>>>     .222......
>>>     ..2.......
>>>     ''').replace('.', '0').splitlines()
>>> img = np.array([list(map(int, line)) for line in lines])
>>> encoded = encode_run_length(img)
>>> recon = decode_run_length(**encoded)
>>> assert np.all(recon == img)
```

kwimage.**encode_run_length**(*img*, *binary=False*, *order='C'*)

Construct the run length encoding (RLE) of an image.

Parameters

- **img** (*ndarray*) – 2D image
- **binary** (*bool, default=True*) – set to True for compatibility with COCO
- **order** (*{'C', 'F'}, default='C'*) – row-major (C) or column-major (F)

Returns

encoding: dictionary items are: counts (ndarray): the run length encoding shape (Tuple): the original image shape binary (bool): if the counts encoding is binary or multiple values are ok order ({'C', 'F'}, default='C'): encoding order

Return type Dict[str, object]

SeeAlso:

- kwimage.Mask - a cython-backed data structure to handle coco-style RLEs

Example

```
>>> import ubelt as ub
>>> lines = ub.codeblock(
>>>     '''
>>>     .........
>>>     ......111.
>>>     ..2...111.
>>>     .222..111.
>>>     22222.....
>>>     .222......
```

(continues on next page)

(continued from previous page)

```
>>>     ..2.......
>>>     ''').replace('.', '0').splitlines()
>>> img = np.array([list(map(int, line)) for line in lines])
>>> encoding = encode_run_length(img)
>>> target = np.array([0,16,1,3,0,3,2,1,0,3,1,3,0,2,2,3,0,2,1,3,0,1,2,5,0,6,2,3,0,
↪8,2,1,0,7])
>>> assert np.all(target == encoding['counts'])
```

Example

```
>>> binary = True
>>> img = np.array([[1, 0, 1, 1, 1, 0, 0, 1, 0]])
>>> encoding = encode_run_length(img, binary=True)
>>> assert encoding['counts'].tolist() == [0, 1, 1, 3, 2, 1, 1]
```

kwimage.**rle_translate**(*rle*, *offset*, *output_shape=None*)

Translates a run-length encoded image in RLE-space.

Parameters

- **rle** (*dict*) – an enconding dict returned by *encode_run_length*
- **offset** (*Tuple*) – x,y offset, CAREFUL, this can only accept integers
- **output_shape** (*Tuple, optional*) – h,w of transformed mask. If unspecified the input rle shape is used.

SeeAlso: # ITK has some RLE code that looks like it can perform translations https://github.com/KitwareMedical/ITKRLEImage/blob/master/include/itkRLERegionOfInterestImageFilter.h

Doctest:

```
>>> # test that translate works on all zero images
>>> img = np.zeros((7, 8), dtype=np.uint8)
>>> rle = encode_run_length(img, binary=True, order='F')
>>> new_rle = rle_translate(rle, (1, 2), (6, 9))
>>> assert np.all(new_rle['counts'] == [54])
```

Example

```
>>> from kwimage.im_runlen import *  # NOQA
>>> img = np.array([
>>>     [1, 1, 1, 1],
>>>     [0, 1, 0, 0],
>>>     [0, 1, 0, 1],
>>>     [1, 1, 1, 1],], dtype=np.uint8)
>>> rle = encode_run_length(img, binary=True, order='C')
>>> offset = (1, -1)
>>> output_shape = (3, 5)
>>> new_rle = rle_translate(rle, offset, output_shape)
>>> decoded = decode_run_length(**new_rle)
>>> print(decoded)
[[0 0 1 0 0]
 [0 0 1 0 1]
 [0 1 1 1 1]]
```

Example

```
>>> from kwimage.im_runlen import *  # NOQA
>>> img = np.array([
>>>     [0, 0, 0],
>>>     [0, 1, 0],
>>>     [0, 0, 0]], dtype=np.uint8)
>>> rle = encode_run_length(img, binary=True, order='C')
>>> new_rle = rle_translate(rle, (1, 0))
>>> decoded = decode_run_length(**new_rle)
>>> print(decoded)
[[0 0 0]
 [0 0 1]
 [0 0 0]]
>>> new_rle = rle_translate(rle, (0, 1))
>>> decoded = decode_run_length(**new_rle)
>>> print(decoded)
[[0 0 0]
 [0 0 0]
 [0 1 0]]
```

kwimage.**stack_images**(*images*, *axis=0*, *resize=None*, *interpolation=None*, *overlap=0*, *return_info=False*, *bg_value=None*)

Make a new image with the input images side-by-side

Parameters

- **images** (*Iterable[ndarray[ndim=2]]*) – image data
- **axis** (*int*) – axis to stack on (either 0 or 1)
- **resize** (*int, str, or None*) – if None image sizes are not modified, otherwise resize resize can be either 0 or 1. We resize the *resize*-th image to match the *1 - resize*-th image. Can also be strings "larger" or "smaller".
- **interpolation** (*int or str*) – string or cv2-style interpolation type. only used if resize or overlap > 0
- **overlap** (*int*) – number of pixels to overlap. Using a negative number results in a border.
- **return_info** (*bool*) – if True, returns transforms (scales and translations) to map from original image to its new location.

Returns

an image of stacked images side by side

OR

Tuple[ndarray, List]: where the first item is the aformentioned stacked image and the second item is a list of transformations for each input image mapping it to its location in the returned image.

Return type ndarray

Example

```
>>> import kwimage
>>> img1 = kwimage.grab_test_image('carl', space='rgb')
>>> img2 = kwimage.grab_test_image('astro', space='rgb')
```

(continues on next page)

(continued from previous page)

```
>>> images = [img1, img2]
>>> imgB, transforms = stack_images(images, axis=0, resize='larger',
>>>                                 overlap=-10, return_info=True)
>>> print(imgB.shape)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> import kwimage
>>> kwplot.autompl()
>>> kwplot.imshow(imgB, colorspace='rgb')
>>> wh1 = np.multiply(img1.shape[0:2][::-1], transforms[0].scale)
>>> wh2 = np.multiply(img2.shape[0:2][::-1], transforms[1].scale)
>>> xoff1, yoff1 = transforms[0].translation
>>> xoff2, yoff2 = transforms[1].translation
>>> xywh1 = (xoff1, yoff1, wh1[0], wh1[1])
>>> xywh2 = (xoff2, yoff2, wh2[0], wh2[1])
>>> kwplot.draw_boxes(kwimage.Boxes([xywh1], 'xywh'), color=(1.0, 0, 0))
>>> kwplot.draw_boxes(kwimage.Boxes([xywh2], 'xywh'), color=(1.0, 0, 0))
>>> kwplot.show_if_requested()
((662, 512, 3), (0.0, 0.0), (0, 150))
```

kwimage.**stack_images_grid**(*images*, *chunksize=None*, *axis=0*, *overlap=0*, *return_info=False*, *bg_value=None*)

Stacks images in a grid. Optionally return transforms of original image positions in the output image.

Parameters

- **images** (*Iterable[ndarray[ndim=2]]*) – image data
- **chunksize** (*int, default=None*) – number of rows per column or columns per row depending on the value of *axis*. If unspecified, computes this as *int(sqrt(len(images)))*.
- **axis** (*int, default=0*) – If 0, chunksize is columns per row. If 1, chunksize is rows per column.
- **overlap** (*int*) – number of pixels to overlap. Using a negative number results in a border.
- **return_info** (*bool*) – if True, returns transforms (scales and translations) to map from original image to its new location.

Returns

an image of stacked images in a grid pattern

OR

Tuple[ndarray, List]: where the first item is the aformentioned stacked image and the second item is a list of transformations for each input image mapping it to its location in the returned image.

Return type ndarray

class kwimage.**Boxes**(*data*, *format=None*, *check=True*)

Bases: kwimage.structs.boxes._BoxConversionMixins, kwimage.structs.boxes._BoxPropertyMixins, kwimage.structs.boxes._BoxTransformMixins, kwimage.structs.boxes._BoxDrawMixins, ubelt.NiceRepr

Converts boxes between different formats as long as the last dimension contains 4 coordinates and the format is specified.

This is a convinience class, and should not not store the data for very long. The general idiom should be create class, convert data, and then get the raw data and let the class be garbage collected. This will help ensure that

your code is portable and understandable if this class is not available.

Example

```
>>> # xdoctest: +IGNORE_WHITESPACE
>>> Boxes([25, 30, 15, 10], 'xywh')
<Boxes(xywh, array([25, 30, 15, 10]))>
>>> Boxes([25, 30, 15, 10], 'xywh').to_xywh()
<Boxes(xywh, array([25, 30, 15, 10]))>
>>> Boxes([25, 30, 15, 10], 'xywh').to_cxywh()
<Boxes(cxywh, array([32.5, 35. , 15. , 10. ]))>
>>> Boxes([25, 30, 15, 10], 'xywh').to_tlbr()
<Boxes(tlbr, array([25, 30, 40, 40]))>
>>> Boxes([25, 30, 15, 10], 'xywh').scale(2).to_tlbr()
<Boxes(tlbr, array([50., 60., 80., 80.]))>
>>> Boxes(torch.FloatTensor([[25, 30, 15, 20]]), 'xywh').scale(.1).to_tlbr()
<Boxes(tlbr, tensor([[ 2.5000,  3.0000,  4.0000,  5.0000]]))>
```

Example

```
>>> datas = [
>>>     [1, 2, 3, 4],
>>>     [[1, 2, 3, 4], [4, 5, 6, 7]],
>>>     [[[1, 2, 3, 4], [4, 5, 6, 7]]],
>>> ]
>>> formats = BoxFormat.cannonical
>>> for format1 in formats:
>>>     for data in datas:
>>>         self = box1 = Boxes(data, format1)
>>>         for format2 in formats:
>>>             box2 = box1.toformat(format2)
>>>             back = box2.toformat(format1)
>>>             assert box1 == back
```

device
If the backend is torch returns the data device, otherwise None

__getitem__(*self*, *index*)

__eq__(*self*, *other*)
Tests equality of two Boxes objects

Example

```
>>> box0 = box1 = Boxes([[1, 2, 3, 4]], 'xywh')
>>> box2 = Boxes(box0.data, 'tlbr')
>>> box3 = Boxes([[0, 2, 3, 4]], box0.format)
>>> box4 = Boxes(box0.data, box2.format)
>>> assert box0 == box1
>>> assert not box0 == box2
>>> assert not box2 == box3
>>> assert box2 == box4
```

__len__(*self*)

__nice__(*self*)

__repr__(*self*)

classmethod random(*Boxes*, *num=1*, *scale=1.0*, *format=BoxFormat.XYWH*, *anchors=None*, *anchor_std=1.0 / 6*, *tensor=False*, *rng=None*)

Makes random boxes; typically for testing purposes

Parameters

- **num** (*int*) – number of boxes to generate
- **scale** (*float | Tuple[float, float]*) – size of imgdims
- **format** (*str*) – format of boxes to be created (e.g. tlbr, xywh)
- **anchors** (*ndarray*) – normalized width / heights of anchor boxes to perterb and randomly place. (must be in range 0-1)
- **anchor_std** (*float*) – magnitude of noise applied to anchor shapes
- **tensor** (*bool*) – if True, returns boxes in tensor format
- **rng** (*None | int | RandomState*) – initial random seed

Example

```
>>> # xdoctest: +IGNORE_WHITESPACE
>>> Boxes.random(3, rng=0, scale=100)
<Boxes(xywh,
    array([[54, 54,  6, 17],
           [42, 64,  1, 25],
           [79, 38, 17, 14]]))>
>>> Boxes.random(3, rng=0, scale=100).tensor()
<Boxes(xywh,
    tensor([[ 54,  54,   6,  17],
            [ 42,  64,   1,  25],
            [ 79,  38,  17,  14]]))>
>>> anchors = np.array([[.5, .5], [.3, .3]])
>>> Boxes.random(3, rng=0, scale=100, anchors=anchors)
<Boxes(xywh,
    array([[ 2, 13, 51, 51],
           [32, 51, 32, 36],
           [36, 28, 23, 26]]))>
```

Example

```
>>> # Boxes position/shape within 0-1 space should be uniform.
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> fig = kwplot.figure(fnum=1, doclf=True)
>>> fig.gca().set_xlim(0, 128)
>>> fig.gca().set_ylim(0, 128)
>>> import kwimage
>>> kwimage.Boxes.random(num=10).scale(128).draw()
```

copy(*self*)

classmethod concatenate(*cls*, *boxes*, *axis=0*)
Concatenates multiple boxes together

Parameters

- **boxes** (*Sequence[Boxes]*) – list of boxes to concatenate
- **axis** (*int, default=0*) – axis to stack on

Returns stacked boxes

Return type *Boxes*

Example

```
>>> boxes = [Boxes.random(3) for _ in range(3)]
>>> new = Boxes.concatenate(boxes)
>>> assert len(new) == 9
>>> assert np.all(new.data[3:6] == boxes[1].data)
```

Example

```
>>> boxes = [Boxes.random(3) for _ in range(3)]
>>> boxes[0].data = boxes[0].data[0]
>>> boxes[1].data = boxes[0].data[0:0]
>>> new = Boxes.concatenate(boxes)
>>> assert len(new) == 4
>>> new = Boxes.concatenate([b.tensor() for b in boxes])
>>> assert len(new) == 4
```

compress(*self*, *flags*, *axis=0*, *inplace=False*)
Filters boxes based on a boolean criterion

Parameters

- **flags** (*ArrayLike[bool]*) – true for items to be kept
- **axis** (*int*) – you usually want this to be 0
- **inplace** (*bool*) – if True, modifies this object

Example

```
>>> self = Boxes([[25, 30, 15, 10]], 'tlbr')
>>> self.compress([True])
<Boxes(tlbr, array([[25, 30, 15, 10]]))>
>>> self.compress([False])
<Boxes(tlbr, array([], shape=(0, 4), dtype=int64))>
```

take(*self*, *idxs*, *axis=0*, *inplace=False*)
Takes a subset of items at specific indices

Parameters

- **indices** (*ArrayLike[int]*) – indexes of items to take
- **axis** (*int*) – you usually want this to be 0

- **inplace** (*bool*) – if True, modifies this object

Example

```
>>> self = Boxes([[25, 30, 15, 10]], 'tlbr')
>>> self.take([0])
<Boxes(tlbr, array([[25, 30, 15, 10]]))>
>>> self.take([])
<Boxes(tlbr, array([], shape=(0, 4), dtype=int64))>
```

is_tensor(*self*)
is the backend fueled by torch?

is_numpy(*self*)
is the backend fueled by numpy?

_impl(*self*)
returns the kwarray.ArrayAPI implementation for the data

Example

```
>>> assert Boxes.random().numpy()._impl.is_numpy
>>> assert Boxes.random().tensor()._impl.is_tensor
```

astype(*self*, *dtype*)
Changes the type of the internal array used to represent the boxes

Notes

this operation is not inplace

Example

```
>>> # xdoctest: +IGNORE_WHITESPACE
>>> Boxes.random(3, 100, rng=0).tensor().astype('int32')
<Boxes(xywh,
    tensor([[54, 54,  6, 17],
            [42, 64,  1, 25],
            [79, 38, 17, 14]], dtype=torch.int32))>
>>> Boxes.random(3, 100, rng=0).numpy().astype('int32')
<Boxes(xywh,
    array([[54, 54,  6, 17],
           [42, 64,  1, 25],
           [79, 38, 17, 14]], dtype=int32))>
>>> Boxes.random(3, 100, rng=0).tensor().astype('float32')
>>> Boxes.random(3, 100, rng=0).numpy().astype('float32')
```

numpy(*self*)
Converts tensors to numpy. Does not change memory if possible.

Example

```
>>> self = Boxes.random(3).tensor()
>>> newself = self.numpy()
>>> self.data[0, 0] = 0
>>> assert newself.data[0, 0] == 0
>>> self.data[0, 0] = 1
>>> assert self.data[0, 0] == 1
```

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

Converts numpy to tensors. Does not change memory if possible.

Example

```
>>> self = Boxes.random(3)
>>> newself = self.tensor()
>>> self.data[0, 0] = 0
>>> assert newself.data[0, 0] == 0
>>> self.data[0, 0] = 1
>>> assert self.data[0, 0] == 1
```

ious (*self*, *other*, *bias=0*, *impl='auto'*, *mode=None*)

Compute IOUs (intersection area over union area) between these boxes and another set of boxes.

Parameters

- **other** (*Boxes*) – boxes to compare IoUs against
- **bias** (*int, default=0*) – either 0 or 1, does TL=BR have area of 0 or 1?
- **impl** (*str, default='auto'*) – code to specify implementation used to ious. Can be either torch, py, c, or auto. Efficiency and the exact result will vary by implementation, but they will always be close. Some implementations only accept certain data types (e.g. impl='c', only accepts float32 numpy arrays). See ~/code/kwimage/dev/bench_bbox.py for benchmark details. On my system the torch impl was fastest (when the data was on the GPU).
- **mode** – depricated, use impl

Examples

```
>>> self = Boxes(np.array([[ 0,  0, 10, 10],
>>>                        [10,  0, 20, 10],
>>>                        [20,  0, 30, 10]]), 'tlbr')
>>> other = Boxes(np.array([6, 2, 20, 10]), 'tlbr')
>>> overlaps = self.ious(other, bias=1).round(2)
>>> assert np.all(np.isclose(overlaps, [0.21, 0.63, 0.04])), repr(overlaps)
```

Examples

```
>>> # xdoctest: +IGNORE_WHITESPACE
>>> Boxes(np.empty(0), 'xywh').ious(Boxes(np.empty(4), 'xywh')).shape
(0,)
>>> #Boxes(np.empty(4), 'xywh').ious(Boxes(np.empty(0), 'xywh')).shape
```

(continues on next page)

(continued from previous page)

```
>>> Boxes(np.empty((0, 4)), 'xywh').ious(Boxes(np.empty((0, 4)), 'xywh')).
↪shape
(0, 0)
>>> Boxes(np.empty((1, 4)), 'xywh').ious(Boxes(np.empty((0, 4)), 'xywh')).
↪shape
(1, 0)
>>> Boxes(np.empty((0, 4)), 'xywh').ious(Boxes(np.empty((1, 4)), 'xywh')).
↪shape
(0, 1)
```

Examples

```
>>> formats = BoxFormat.cannonical
>>> istensors = [False, True]
>>> results = {}
>>> for format in formats:
>>>     for tensor in istensors:
>>>         boxes1 = Boxes.random(5, scale=10.0, rng=0, format=format,␣
↪tensor=tensor)
>>>         boxes2 = Boxes.random(7, scale=10.0, rng=1, format=format,␣
↪tensor=tensor)
>>>         ious = boxes1.ious(boxes2)
>>>         results[(format, tensor)] = ious
>>> results = {k: v.numpy() if torch.is_tensor(v) else v for k, v in results.
↪items() }
>>> results = {k: v.tolist() for k, v in results.items()}
>>> print(ub.repr2(results, sk=True, precision=3, nl=2))
>>> from functools import partial
>>> assert ub.allsame(results.values(), partial(np.allclose, atol=1e-07))
```

isect_area(*self*, *other*, *bias=0*)

Intersection part of intersection over union computation

Examples

```
>>> # xdoctest: +IGNORE_WHITESPACE
>>> self = Boxes.random(5, scale=10.0, rng=0, format='tlbr')
>>> other = Boxes.random(3, scale=10.0, rng=1, format='tlbr')
>>> isect = self.isect_area(other, bias=0)
>>> ious_v1 = isect / ((self.area + other.area.T) - isect)
>>> ious_v2 = self.ious(other, bias=0)
>>> assert np.allclose(ious_v1, ious_v2)
```

intersection(*self*, *other*)

Pairwise intersection between two sets of Boxes

Returns intersected boxes

Return type *Boxes*

Examples

```
>>> # xdoctest: +IGNORE_WHITESPACE
>>> from kwimage.structs.boxes import *  # NOQA
>>> self = Boxes.random(5, rng=0).scale(10.)
>>> other = self.translate(1)
>>> new = self.intersection(other)
>>> new_area = np.nan_to_num(new.area).ravel()
>>> alt_area = np.diag(self.isect_area(other))
>>> close = np.isclose(new_area, alt_area)
>>> assert np.all(close)
```

view(*self*, **shape*)
Passthrough method to view or reshape

Example

```
>>> self = Boxes.random(6, scale=10.0, rng=0, format='xywh').tensor()
>>> assert list(self.view(3, 2, 4).data.shape) == [3, 2, 4]
>>> self = Boxes.random(6, scale=10.0, rng=0, format='tlbr').tensor()
>>> assert list(self.view(3, 2, 4).data.shape) == [3, 2, 4]
```

class kwimage.**Coords**(*data=None*, *meta=None*)
Bases: *kwimage.structs._generic.Spatial*, ubelt.NiceRepr

This stores arbitrary sparse n-dimensional coordinate geometry.

You can specify data, but you don't have to. We dont care what it is, we just warp it.

Note: This class was designed to hold coordinates in r/c format, but in general this class is anostic to dimension ordering as long as you are consistent. However, there are two places where this matters:

(1) drawing and (2) gdal/imgaug-warping. In these places we will assume x/y for legacy reasons. This may change in the future.

CommandLine: xdoctest -m kwimage.structs.coords Coords

Example

```
>>> from kwimage.structs.coords import *  # NOQA
>>> import kwarray
>>> rng = kwarray.ensure_rng(0)
>>> self = Coords.random(num=4, dim=3, rng=rng)
>>> matrix = rng.rand(4, 4)
>>> self.warp(matrix)
>>> self.translate(3, inplace=True)
>>> self.translate(3, inplace=True)
>>> self.scale(2)
>>> self.tensor()
>>> # self.tensor(device=0)
>>> self.tensor().tensor().numpy().numpy()
>>> self.numpy()
>>> #self.draw_on()
```

__repr__

dtype

dim

shape

device
If the backend is torch returns the data device, otherwise None

_impl
Returns the internal tensor/numpy ArrayAPI implementation

__nice__(*self*)

__len__(*self*)

copy(*self*)

classmethod random(*Coords*, *num=1*, *dim=2*, *rng=None*, *meta=None*)
Makes random coordinates; typically for testing purposes

is_numpy(*self*)

is_tensor(*self*)

compress(*self*, *flags*, *axis=0*, *inplace=False*)
Filters items based on a boolean criterion

Parameters

- **flags** (*ArrayLike[bool]*) – true for items to be kept
- **axis** (*int*) – you usually want this to be 0
- **inplace** (*bool, default=False*) – if True, modifies this object

Returns filtered coords

Return type *Coords*

Example

```
>>> from kwimage.structs.coords import *  # NOQA
>>> self = Coords.random(10, rng=0)
>>> self.compress([True] * len(self))
>>> self.compress([False] * len(self))
<Coords(data=array([], shape=(0, 2), dtype=float64))>
>>> self = self.tensor()
>>> self.compress([True] * len(self))
>>> self.compress([False] * len(self))
```

take(*self*, *indices*, *axis=0*, *inplace=False*)
Takes a subset of items at specific indices

Parameters

- **indices** (*ArrayLike[int]*) – indexes of items to take
- **axis** (*int*) – you usually want this to be 0
- **inplace** (*bool, default=False*) – if True, modifies this object

Returns filtered coords

Return type *Coords*

Example

```
>>> self = Coords(np.array([[25, 30, 15, 10]]))
>>> self.take([0])
<Coords(data=array([[25, 30, 15, 10]]))>
>>> self.take([])
<Coords(data=array([], shape=(0, 4), dtype=int64))>
```

astype (*self*, *dtype*, *inplace=False*)
Changes the data type

Parameters

- **dtype** – new type
- **inplace** (*bool, default=False*) – if True, modifies this object

view (*self*, **shape*)
Passthrough method to view or reshape

Parameters ***shape** – new shape of the data

Example

```
>>> self = Coords.random(6, dim=4).tensor()
>>> assert list(self.view(3, 2, 4).data.shape) == [3, 2, 4]
>>> self = Coords.random(6, dim=4).numpy()
>>> assert list(self.view(3, 2, 4).data.shape) == [3, 2, 4]
```

classmethod concatenate (*cls*, *coords*, *axis=0*)
Concatenates lists of coordinates together

Parameters

- **coords** (*Sequence[Coords]*) – list of coords to concatenate
- **axis** (*int, default=0*) – axis to stack on

Returns stacked coords

Return type *Coords*

CommandLine: xdoctest -m kwimage.structs.coords Coords.concatenate

Example

```
>>> coords = [Coords.random(3) for _ in range(3)]
>>> new = Coords.concatenate(coords)
>>> assert len(new) == 9
>>> assert np.all(new.data[3:6] == coords[1].data)
```

tensor (*self*, *device=ub.NoParam*)
Converts numpy to tensors. Does not change memory if possible.

Example

```
>>> self = Coords.random(3).numpy()
>>> newself = self.tensor()
>>> self.data[0, 0] = 0
>>> assert newself.data[0, 0] == 0
>>> self.data[0, 0] = 1
>>> assert self.data[0, 0] == 1
```

numpy(*self*)

Converts tensors to numpy. Does not change memory if possible.

Example

```
>>> self = Coords.random(3).tensor()
>>> newself = self.numpy()
>>> self.data[0, 0] = 0
>>> assert newself.data[0, 0] == 0
>>> self.data[0, 0] = 1
>>> assert self.data[0, 0] == 1
```

warp(*self*, *transform*, *input_dims=None*, *output_dims=None*, *inplace=False*)

Generalized coordinate transform.

Parameters

- **transform** (*GeometricTransform | ArrayLike | Augmenter*) – scikit-image tranform, a transformation matrix, or an imgaug Augmenter.
- **input_dims** (*Tuple*) – shape of the image these objects correspond to (only needed / used when transform is an imgaug augmenter)
- **output_dims** (*Tuple*) – unused in non-raster structures, only exists for compatibility.
- **inplace** (*bool, default=False*) – if True, modifies data inplace

Notes

Let D = self.dims

transformation matrices can be either:

- (D + 1) x (D + 1) # for homog
- D x D # for scale / rotate
- D x (D + 1) # for affine

Example

```
>>> from kwimage.structs.coords import *  # NOQA
>>> self = Coords.random(10, rng=0)
>>> transform = skimage.transform.AffineTransform(scale=(2, 2))
>>> new = self.warp(transform)
>>> assert np.all(new.data == self.scale(2).data)
```

Doctest:

```
>>> self = Coords.random(10, rng=0)
>>> assert np.all(self.warp(np.eye(3)).data == self.data)
>>> assert np.all(self.warp(np.eye(2)).data == self.data)
```

Ignore:

```
>>> # xdoctest: +SKIP
>>> # xdoctest: +REQUIRES(module:osr)
>>> wgs84_crs = osr.SpatialReference()
>>> wgs84_crs.ImportFromEPSG(4326)
>>> transform = osr.CoordinateTransformation(wgs84_crs, wgs84_crs)
>>> self = Coords.random(10, rng=0)
>>> new = self.warp(transform)
>>> assert np.all(new.data == self.data)
```

_warp_imgaug(*self*, *augmenter*, *input_dims*, *inplace=False*)

Warps by applying an augmenter from the imgaug library

Note: We are assuming you are using X/Y coordinates here.

Parameters

- **augmenter** (*imgaug.augmenters.Augmenter*)
- **input_dims** (*Tuple*) – h/w of the input image
- **inplace** (*bool, default=False*) – if True, modifies data inplace

CommandLine: xdoctest -m ~/code/kwimage/kwimage/structs/coords.py Coords._warp_imgaug

Example

```
>>> # xdoctest: +REQUIRES(module:imgaug)
>>> from kwimage.structs.coords import *  # NOQA
>>> import imgaug
>>> input_dims = (10, 10)
>>> self = Coords.random(10).scale(input_dims)
>>> augmenter = imgaug.augmenters.Fliplr(p=1)
>>> new = self._warp_imgaug(augmenter, input_dims)
>>> # y coordinate should not change
>>> assert np.allclose(self.data[:, 1], new.data[:, 1])
>>> assert np.allclose(input_dims[0] - self.data[:, 0], new.data[:, 0])
```

```
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.figure(fnum=1, doclf=True)
>>> from matplotlib import pyplot as pl
>>> ax = plt.gca()
>>> ax.set_xlim(0, input_dims[0])
>>> ax.set_ylim(0, input_dims[1])
>>> self.draw(color='red', alpha=.4, radius=0.1)
>>> new.draw(color='blue', alpha=.4, radius=0.1)
```

Example

```
>>> # xdoctest: +REQUIRES(module:imgaug)
>>> from kwimage.structs.coords import *  # NOQA
>>> import imgaug
>>> input_dims = (32, 32)
>>> inplace = 0
>>> self = Coords.random(1000, rng=142).scale(input_dims).scale(.8)
>>> self.data = self.data.astype(np.int32).astype(np.float32)
>>> augmenter = imgaug.augmenters.CropAndPad(px=(-4, 4), keep_size=1).to_
↪deterministic()
>>> new = self._warp_imgaug(augmenter, input_dims)
>>> # Change should be linear
>>> norm1 = (self.data - self.data.min(axis=0)) / (self.data.max(axis=0) -␣
↪self.data.min(axis=0))
>>> norm2 = (new.data - new.data.min(axis=0)) / (new.data.max(axis=0) - new.
↪data.min(axis=0))
>>> diff = norm1 - norm2
>>> assert np.allclose(diff, 0, atol=1e-6, rtol=1e-4)
>>> #assert np.allclose(self.data[:, 1], new.data[:, 1])
>>> #assert np.allclose(input_dims[0] - self.data[:, 0], new.data[:, 0])
>>> # xdoc: +REQUIRES(--show)
>>> import kwimage
>>> im = kwimage.imresize(kwimage.grab_test_image(), dsize=input_dims[::-1])
>>> new_im = augmenter.augment_image(im)
>>> import kwplot
>>> plt = kwplot.autoplt()
>>> kwplot.figure(fnum=1, doclf=True)
>>> kwplot.imshow(im, pnum=(1, 2, 1), fnum=1)
>>> self.draw(color='red', alpha=.8, radius=0.5)
>>> kwplot.imshow(new_im, pnum=(1, 2, 2), fnum=1)
>>> new.draw(color='blue', alpha=.8, radius=0.5, coord_axes=[1, 0])
```

to_imgaug(*self*, *input_dims*)

Example

```
>>> # xdoctest: +REQUIRES(module:imgaug)
>>> from kwimage.structs.coords import *  # NOQA
>>> self = Coords.random(10)
>>> input_dims = (10, 10)
>>> kpoi = self.to_imgaug(input_dims)
>>> new = Coords.from_imgaug(kpoi)
>>> assert np.allclose(new.data, self.data)
```

classmethod from_imgaug(*cls*, *kpoi*)

scale(*self*, *factor*, *output_dims=None*, *inplace=False*)
Scale coordinates by a factor

Parameters

- **factor** (*float or Tuple[float, float]*) – scale factor as either a scalar or per-dimension tuple.
- **output_dims** (*Tuple*) – unused in non-raster spatial structures

Example

```
>>> from kwimage.structs.coords import *  # NOQA
>>> self = Coords.random(10, rng=0)
>>> new = self.scale(10)
>>> assert new.data.max() <= 10
```

```
>>> self = Coords.random(10, rng=0)
>>> self.data = (self.data * 10).astype(np.int)
>>> new = self.scale(10)
>>> assert new.data.dtype.kind == 'i'
>>> new = self.scale(10.0)
>>> assert new.data.dtype.kind == 'f'
```

translate (*self*, *offset*, *output_dims=None*, *inplace=False*)
Shift the coordinates

Parameters

- **offset** (*float or Tuple[float]*) – transation offset as either a scalar or a per-dimension tuple.
- **output_dims** (*Tuple*) – unused in non-raster spatial structures

Example

```
>>> from kwimage.structs.coords import *  # NOQA
>>> self = Coords.random(10, dim=3, rng=0)
>>> new = self.translate(10)
>>> assert new.data.min() >= 10
>>> assert new.data.max() <= 11
>>> Coords.random(3, dim=3, rng=0)
>>> Coords.random(3, dim=3, rng=0).translate((1, 2, 3))
```

fill (*self*, *image*, *value*, *coord_axes=None*, *interp='bilinear'*)
Sets sub-coordinate locations in a grid to a particular value

Parameters **coord_axes** (*Tuple*) – specify which image axes each coordinate dim corresponds to. For 2D images, if you are storing r/c data, set to [0,1], if you are storing x/y data, set to [1,0].

draw_on (*self, image=None, fill_value=1, coord_axes=[1, 0], interp='bilinear'*)

Note: unlike other methods, the defaults assume x/y internal data

---

Parameters **coord_axes** (*Tuple*) – specify which image axes each coordinate dim corresponds to. For 2D images, if you are storing r/c data, set to [0,1], if you are storing x/y data, set to [1,0].

In other words the i-th entry in coord_axes specifies which row-major spatial dimension the i-th column of a coordinate corresponds to. The index is the coordinate dimension and the value is the axes dimension.

Example

```
>>> # xdoc: +REQUIRES(module:kwplot)
>>> from kwimage.structs.coords import *  # NOQA
>>> s = 256
>>> self = Coords.random(10, meta={'shape': (s, s)}).scale(s)
>>> self.data[0] = [10, 10]
>>> self.data[1] = [20, 40]
>>> image = np.zeros((s, s))
>>> fill_value = 1
>>> image = self.draw_on(image, fill_value, coord_axes=[1, 0], interp=
↪'bilinear')
>>> # image = self.draw_on(image, fill_value, coord_axes=[0, 1], interp=
↪'nearest')
>>> # image = self.draw_on(image, fill_value, coord_axes=[1, 0], interp=
↪'bilinear')
>>> # image = self.draw_on(image, fill_value, coord_axes=[1, 0], interp=
↪'nearest')
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.figure(fnum=1, doclf=True)
>>> kwplot.autompl()
>>> kwplot.imshow(image)
>>> self.draw(radius=3, alpha=.5, coord_axes=[1, 0])
```

draw(*self, color='blue', ax=None, alpha=None, coord_axes=[1, 0], radius=1*)

Note: unlike other methods, the defaults assume x/y internal data

Parameters **coord_axes** (*Tuple*) – specify which image axes each coordinate dim corresponds to. For 2D images,

if you are storing r/c data, set to [0,1], if you are storing x/y data, set to [1,0].

Example

```
>>> # xdoc: +REQUIRES(module:kwplot)
>>> from kwimage.structs.coords import *  # NOQA
>>> self = Coords.random(10)
>>> # xdoc: +REQUIRES(--show)
>>> self.draw(radius=3.0)
>>> import kwplot
>>> kwplot.autompl()
>>> self.draw(radius=3.0)
```

class kwimage.**Detections**(*data=None, meta=None, datakeys=None, metakeys=None, checks=True, **kwargs*)

Bases: ubelt.NiceRepr, *kwimage.structs.detections._DetAlgoMixin*, *kwimage.structs.detections._DetDrawMixin*

Container for holding and manipulating multiple detections.

Variables

- **data** (*Dict*) – dictionary containing corresponding lists. The length of each list is the number of detections. This contains the bounding boxes, confidence scores, and class indices. Details of the most common keys and types are as follows:

  boxes (kwimage.Boxes[ArrayLike]): multiple bounding boxes scores (ArrayLike): associated scores class_idxs (ArrayLike): associated class indices

  Additional custom keys may be specified as long as (a) the values are array-like and the first axis corresponds to the standard data values and (b) are custom keys are listed in the *datakeys* kwargs when constructing the Detections.

- **meta** (*Dict*) – This contains contextual information about the detections. This includes the class names, which can be indexed into via the class indexes.

Example

```
>>> self = Detections.random(10)
>>> other = Detections(self)
>>> assert other.data == self.data
>>> assert other.data is self.data, 'try not to copy unless necessary'
```

__datakeys__ = ['boxes', 'scores', 'class_idxs', 'probs', 'weights', 'keypoints', 'seg

__metakeys__ = ['classes']

boxes

class_idxs

scores
typically only populated for predicted detections

probs
typically only populated for predicted detections

weights
typically only populated for groundtruth detections

classes

device
If the backend is torch returns the data device, otherwise None

dtype

__nice__(*self*)

__len__(*self*)

copy(*self*)
Returns a deep copy of this Detections object

classmethod from_coco_annots(*cls*, *anns*, *cats=None*, *classes=None*, *kp_classes=None*, *shape=None*, *dset=None*)
Create a Detections object from a list of coco-like annotations.

Parameters

- **anns** (*List[Dict]*) – list of coco-like annotation objects
- **dset** (*CocoDataset*) – if specified, cats, classes, and kp_classes can are ignored.
- **cats** (*List[Dict]*) – coco-format category information. Used only if *dset* is not specified.

- **classes** (*ndsampler.CategoryTree*) – category tree with coco class info. Used only if *dset* is not specified.
- **kp_classes** (*ndsampler.CategoryTree*) – keypoint category tree with coco keypoint class info. Used only if *dset* is not specified.
- **shape** (*tuple*) – shape of parent image

Returns a detections object

Return type *Detections*

Example

```
>>> from kwimage.structs.detections import *  # NOQA
>>> # xdoctest: +REQUIRES(--module:ndsampler)
>>> anns = [{
>>>     'id': 0,
>>>     'image_id': 1,
>>>     'category_id': 2,
>>>     'bbox': [2, 3, 10, 10],
>>>     'keypoints': [4.5, 4.5, 2],
>>>     'segmentation': {
>>>         'counts': '_11a04M2O0O20N101N3L_5',
>>>         'size': [20, 20],
>>>     },
>>> }]
>>> dataset = {
>>>     'images': [],
>>>     'annotations': [],
>>>     'categories': [
>>>         {'id': 0, 'name': 'background'},
>>>         {'id': 2, 'name': 'class1', 'keypoints': ['spot']}
>>>     ]
>>> }
>>> #import ndsampler
>>> #dset = ndsampler.CocoDataset(dataset)
>>> cats = dataset['categories']
>>> dets = Detections.from_coco_annots(anns, cats)
```

Example

```
>>> import kwimage
>>> # xdoctest: +REQUIRES(--module:ndsampler)
>>> import ndsampler
>>> sampler = ndsampler.CocoSampler.demo('photos')
>>> iminfo, anns = sampler.load_image_with_annots(1)
>>> shape = iminfo['imdata'].shape[0:2]
>>> kp_classes = sampler.dset.keypoint_categories()
>>> dets = kwimage.Detections.from_coco_annots(
>>>     anns, sampler.dset.dataset['categories'], sampler.catgraph,
>>>     kp_classes, shape=shape)
```

to_coco (*self*, *cname_to_cat=None*, *style='orig'*)

Converts this set of detections into coco-like annotation dictionaries.

Notes

Not all aspects of the MS-COCO format can be accurately represented, so some liberties are taken. The MS-COCO standard defines that annotations should specifiy a category_id field, but in some cases this information is not available so we will populate a 'category_name' field if possible and in the worst case fall back to 'category_index'.

Additionally, detections may contain additional information beyond the MS-COCO standard, and this information (e.g. weight, prob, score) is added as forign fields.

Parameters

- **cname_to_cat** – currently ignored.
- **style** (*str*) – either orig (for the original coco format) or new for the more general ndsampler-style coco format.

Yields *dict* – coco-like annotation structures

Example

```
>>> # xdoctest: +REQUIRES(module:ndsampler)
>>> from kwimage.structs.detections import *
>>> self = Detections.demo()[0]
>>> cname_to_cat = None
>>> list(self.to_coco())
```

num_boxes(*self*)

warp(*self*, *transform*, *input_dims=None*, *output_dims=None*, *inplace=False*)
Spatially warp the detections.

Example

```
>>> import skimage
>>> transform = skimage.transform.AffineTransform(scale=(2, 3),␣
↪translation=(4, 5))
>>> self = Detections.random(2)
>>> new = self.warp(transform)
>>> assert new.boxes == self.boxes.warp(transform)
>>> assert new != self
```

scale(*self*, *factor*, *output_dims=None*, *inplace=False*)
Spatially warp the detections.

Example

```
>>> import skimage
>>> transform = skimage.transform.AffineTransform(scale=(2, 3),␣
↪translation=(4, 5))
>>> self = Detections.random(2)
>>> new = self.warp(transform)
>>> assert new.boxes == self.boxes.warp(transform)
>>> assert new != self
```

translate (*self*, *offset*, *output_dims=None*, *inplace=False*)
Spatially warp the detections.

Example

```
>>> import skimage
>>> self = Detections.random(2)
>>> new = self.translate(10)
```

classmethod concatenate (*cls*, *dets*)

Parameters **boxes** (*Sequence[Detections]*) – list of detections to concatenate

Returns stacked detections

Return type *Detections*

Example

```
>>> self = Detections.random(2)
>>> other = Detections.random(3)
>>> dets = [self, other]
>>> new = Detections.concatenate(dets)
>>> assert new.num_boxes() == 5
```

```
>>> self = Detections.random(2, segmentations=True)
>>> other = Detections.random(3, segmentations=True)
>>> dets = [self, other]
>>> new = Detections.concatenate(dets)
>>> assert new.num_boxes() == 5
```

argsort (*self*, *reverse=True*)
Sorts detection indices by descending (or ascending) scores

Returns sorted indices

Return type ndarray[int]

sort (*self*, *reverse=True*)
Sorts detections by descending (or ascending) scores

Returns sorted copy of self

Return type *kwimage.structs.Detections*

compress (*self*, *flags*, *axis=0*)
Returns a subset where corresponding locations are True.

Parameters **flags** (*ndarray[bool]*) – mask marking selected items

Returns subset of self

Return type *kwimage.structs.Detections*

CommandLine: xdoctest -m kwimage.structs.detections Detections.compress

Example

```
>>> import kwimage
>>> dets = kwimage.Detections.random(keypoints='dense')
>>> flags = np.random.rand(len(dets)) > 0.5
>>> subset = dets.compress(flags)
>>> assert len(subset) == flags.sum()
>>> subset = dets.tensor().compress(flags)
>>> assert len(subset) == flags.sum()
```

z = dets.tensor().data['keypoints'].data['xy'] z.compress(flags) ub.map_vals(lambda x: x.shape, dets.data) ub.map_vals(lambda x: x.shape, subset.data)

take(*self*, *indices*, *axis=0*)

Returns a subset specified by indices

Parameters **indices** (*ndarray[int]*) – indices to select

Returns subset of self

Return type *kwimage.structs.Detections*

Example

```
>>> import kwimage
>>> dets = kwimage.Detections(boxes=kwimage.Boxes.random(10))
>>> subset = dets.take([2, 3, 5, 7])
>>> assert len(subset) == 4
>>> subset = dets.tensor().take([2, 3, 5, 7])
>>> assert len(subset) == 4
```

__getitem__(*self*, *index*)

Fancy slicing / subset / indexing.

Note: scalar indices are always coerced into index lists of length 1.

Example

```
>>> import kwimage
>>> import kwarray
>>> dets = kwimage.Detections(boxes=kwimage.Boxes.random(10))
>>> indices = [2, 3, 5, 7]
>>> flags = kwarray.boolmask(indices, len(dets))
>>> assert dets[flags].data == dets[indices].data
```

is_tensor(*self*)

is the backend fueled by torch?

is_numpy(*self*)

is the backend fueled by numpy?

numpy(*self*)

Converts tensors to numpy. Does not change memory if possible.

Example

```
>>> self = Detections.random(3).tensor()
>>> newself = self.numpy()
>>> self.scores[0] = 0
>>> assert newself.scores[0] == 0
>>> self.scores[0] = 1
>>> assert self.scores[0] == 1
>>> self.numpy().numpy()
```

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

Converts numpy to tensors. Does not change memory if possible.

Example

```
>>> from kwimage.structs.detections import *
>>> self = Detections.random(3)
>>> newself = self.tensor()
>>> self.scores[0] = 0
>>> assert newself.scores[0] == 0
>>> self.scores[0] = 1
>>> assert self.scores[0] == 1
>>> self.tensor().tensor()
```

classmethod demo (*Detections*)

classmethod random (*cls*, *num=10*, *scale=1.0*, *rng=None*, *classes=3*, *keypoints=False*, *tensor=False*, *segmentations=False*)

Creates dummy data, suitable for use in tests and benchmarks

Parameters

- **num** (*int*) – number of boxes
- **scale** (*float | tuple, default=1.0*) – bounding image size
- **classes** (*int | Sequence*) – list of class labels or number of classes
- **tensor** (*bool, default=False*) – determines backend
- **rng** (*np.random.RandomState*) – random state

Example

```
>>> import kwimage
>>> dets = kwimage.Detections.random(keypoints='jagged')
>>> dets.data['keypoints'].data[0].data
>>> dets.data['keypoints'].meta
>>> dets = kwimage.Detections.random(keypoints='dense')
>>> dets = kwimage.Detections.random(keypoints='dense', segmentations=True).
↪scale(1000)
>>> # xdoctest:+REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> dets.draw(setlim=True)
```

Example

```
>>> # Boxes position/shape within 0-1 space should be uniform.
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> fig = kwplot.figure(fnum=1, doclf=True)
>>> fig.gca().set_xlim(0, 128)
>>> fig.gca().set_ylim(0, 128)
>>> import kwimage
>>> kwimage.Detections.random(num=10, segmentations=True).scale(128).draw()
```

class kwimage.**Heatmap**(*data=None*, *meta=None*, ***kwargs*)

Bases: *kwimage.structs._generic.Spatial*, *kwimage.structs.heatmap._HeatmapDrawMixin*, *kwimage.structs.heatmap._HeatmapWarpMixin*, *kwimage.structs.heatmap._HeatmapAlgoMixin*

Keeps track of a downscaled heatmap and how to transform it to overlay the original input image. Heatmaps generally are used to estimate class probabilites at each pixel. This data struction additionally contains logic to augment pixel with offset (dydx) and scale (diamter) information.

Variables

- **data** (*Dict[str, object]*) – dictionary containing spatially aligned heatmap data. Valid keys are as follows.

  class_probs (ArrayLike[C, H, W] | ArrayLike[C, D, H, W]): A probability map for each class. C is the number of classes.

  offset (ArrayLike[2, H, W] | ArrayLike[3, D, H, W], optional): object center position offset in y,x / t,y,x coordinates

  diamter (ArrayLike[2, H, W] | ArrayLike[3, D, H, W], optional): object bounding box sizes in h,w / d,h,w coordinates

  keypoints (ArrayLike[2, K, H, W] | ArrayLike[3, K, D, H, W], optional): y/x offsets for K different keypoint classes

- **data** – dictionary containing miscellanious metadata about the heatmap data. Valid keys are as follows.

  img_dims (Tuple[H, W] | Tuple[D, H, W]): original image dimension

  tf_data_to_image (skimage.transform._geometric.GeometricTransform): transformation matrix (typically similarity or affine) that projects the given1.8719898042840075, heatmap onto the image dimensions such that the image and heatmap are spatially aligned.

  classes (List[str] | ndsampler.CategoryTree): information about which index in *data['class_probs']* corresponds to which semantic class.

- ****kwargs** – any key that is accepted by the *data* or *meta* dictionaries can be specified as a keyword argument to this class and it will be properly placed in the appropriate internal dictionary.

CommandLine: xdoctest -m ~/code/kwimage/kwimage/structs/heatmap.py Heatmap –show

Example

```
>>> import kwimage
>>> class_probs = kwimage.grab_test_image(dsize=(32, 32), space='gray')[None, ] /␣
↪255.0
>>> img_dims = (220, 220)
>>> tf_data_to_img = skimage.transform.AffineTransform(translation=(-18, -18),␣
↪scale=(8, 8))
>>> self = Heatmap(class_probs=class_probs, img_dims=img_dims,
>>>                tf_data_to_img=tf_data_to_img)
>>> aligned = self.upscale()
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(aligned[0])
>>> kwplot.show_if_requested()
```

__datakeys__ = ['class_probs', 'offset', 'diameter', 'keypoints', 'class_idx', 'class_

__metakeys__ = ['img_dims', 'tf_data_to_img', 'classes', 'kp_classes']

__spatialkeys__ = ['offset', 'diameter', 'keypoints']

shape

bounds

dims
space-time dimensions of this heatmap

_impl
Returns the internal tensor/numpy ArrayAPI implementation

Returns kwarray.ArrayAPI

class_probs

offset

diameter

img_dims

tf_data_to_img

classes

__nice__(*self*)

__getitem__(*self*, *index*)

__len__(*self*)

is_numpy(*self*)

is_tensor(*self*)

classmethod random(*cls*, *dims=(10, 10)*, *classes=3*, *diameter=True*, *offset=True*, *keypoints=False*, *img_dims=None*, *dets=None*, *nblips=10*, *noise=0.0*, *rng=None*)
Creates dummy data, suitable for use in tests and benchmarks

Parameters

- **dims** (*Tuple*) – dimensions of the heatmap
- **img_dims** (*Tuple*) – dimensions of the image the heatmap corresponds to

Example

```
>>> from kwimage.structs.heatmap import *  # NOQA
>>> self = Heatmap.random((128, 128), img_dims=(200, 200),
>>>     classes=3, nblips=10, rng=0, noise=0.1)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(self.colorize(0, imgspace=0), fnum=1, pnum=(1, 4, 1),␣
↪doclf=1)
>>> kwplot.imshow(self.colorize(1, imgspace=0), fnum=1, pnum=(1, 4, 2))
>>> kwplot.imshow(self.colorize(2, imgspace=0), fnum=1, pnum=(1, 4, 3))
>>> kwplot.imshow(self.colorize(3, imgspace=0), fnum=1, pnum=(1, 4, 4))
```

Ignore: self.detect(0).sort().non_max_supress()[-np.arange(1, 4)].draw() from kwimage.structs.heatmap import * # NOQA import xdev globals().update(xdev.get_func_kwargs(Heatmap.random))

Example

```
>>> # xdoctest: +REQUIRES(module:ndsampler)
>>> import kwimage
>>> self = kwimage.Heatmap.random(dims=(50, 200), dets='coco',
>>>                               keypoints=True)
>>> image = np.zeros(self.img_dims)
>>> toshow = self.draw_on(image, 1, vecs=True, kpts=0, with_alpha=0.85)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.figure(fnum=1, doclf=True)
>>> kwplot.imshow(toshow)
```

Ignore:

```
>>> kwplot.figure(fnum=1, doclf=True)
>>> kwplot.imshow(image)
>>> dets.draw()
>>> dets.data['keypoints'].draw(radius=6)
>>> dets.data['segmentations'].draw()
```

```
>>> self.draw()
```

numpy(*self*)
Converts underlying data to numpy arrays

tensor(*self*, *device=ub.NoParam*)
Converts underlying data to torch tensors

class kwimage.**Mask**(*data=None*, *format=None*)
Bases: ubelt.NiceRepr, kwimage.structs.mask._MaskConversionMixin, kwimage.structs.mask._MaskConstructorMixin, kwimage.structs.mask._MaskTransformMixin, kwimage.structs.mask._MaskDrawMixin

Manages a single segmentation mask and can convert to and from multiple formats including:

- bytes_rle - byte encoded run length encoding

- array_rle - raw run length encoding
- c_mask - c-style binary mask
- f_mask - fortran-style binary mask

Example

```
>>> # xdoc: +REQUIRES(--mask)
>>> # a ms-coco style compressed bytes rle segmentation
>>> segmentation = {'size': [5, 9], 'counts': ';?1B10O30O4'}
>>> mask = Mask(segmentation, 'bytes_rle')
>>> # convert to binary numpy representation
>>> binary_mask = mask.to_c_mask().data
>>> print(ub.repr2(binary_mask.tolist(), nl=1, nobr=1))
[0, 0, 0, 1, 1, 1, 1, 1, 0],
[0, 0, 1, 1, 1, 0, 0, 0, 0],
[0, 0, 1, 1, 1, 1, 1, 1, 0],
[0, 0, 1, 1, 1, 0, 1, 1, 0],
[0, 0, 1, 1, 1, 0, 1, 1, 0],
```

dtype

shape

area
Returns the number of non-zero pixels

Example

```
>>> # xdoc: +REQUIRES(--mask)
>>> self = Mask.demo()
>>> self.area
150
```

__nice__(*self*)

classmethod random(*Mask*, *rng=None*, *shape=(32, 32)*)

Example

Mask.random(rng=0).draw()

copy(*self*)
Performs a deep copy of the mask data

Example

```
>>> self = Mask.random(shape=(8, 8), rng=0)
>>> other = self.copy()
>>> assert other.data is not self.data
```

union(*self*, **others*)
This can be used as a staticmethod or an instancemethod

Example

```
>>> # xdoc: +REQUIRES(--mask)
>>> from kwimage.structs.mask import *  # NOQA
>>> masks = [Mask.random(shape=(8, 8), rng=i) for i in range(2)]
>>> mask = Mask.union(*masks)
>>> print(mask.area)
>>> masks = [m.to_c_mask() for m in masks]
>>> mask = Mask.union(*masks)
>>> print(mask.area)
```

```
>>> masks = [m.to_bytes_rle() for m in masks]
>>> mask = Mask.union(*masks)
>>> print(mask.area)
```

Benchmark: import ubelt as ub ti = ub.Timerit(100, bestof=10, verbose=2)

masks = [Mask.random(shape=(172, 172), rng=i) for i in range(2)]

for timer in ti.reset('native rle union'): masks = [m.to_bytes_rle() for m in masks] with timer:

mask = Mask.union(*masks)

for timer in ti.reset('native cmask union'): masks = [m.to_c_mask() for m in masks] with timer:

mask = Mask.union(*masks)

for timer in ti.reset('cmask->rle union'): masks = [m.to_c_mask() for m in masks] with timer:

mask = Mask.union(*[m.to_bytes_rle() for m in masks])

intersection(*self*, **others*)

This can be used as a staticmethod or an instancemethod

Example

```
>>> # xdoc: +REQUIRES(--mask)
>>> masks = [Mask.random(shape=(8, 8), rng=i) for i in range(2)]
>>> mask = Mask.intersection(*masks)
>>> print(mask.area)
```

get_patch(*self*)

Extract the patch with non-zero data

Example

```
>>> # xdoc: +REQUIRES(--mask)
>>> from kwimage.structs.mask import *  # NOQA
>>> self = Mask.random(shape=(8, 8), rng=0)
>>> self.get_patch()
```

get_xywh(*self*)

Gets the bounding xywh box coordinates of this mask

Returns

x, y, w, h: Note we dont use a Boxes object because a general singular version does not yet exist.

Return type ndarray

Example

```
>>> # xdoc: +REQUIRES(--mask)
>>> self = Mask.random(shape=(8, 8), rng=0)
>>> self.get_xywh().tolist()
>>> self = Mask.random(rng=0).translate((10, 10))
>>> self.get_xywh().tolist()
```

get_polygon(*self*)

Returns a list of (x,y)-coordinate lists. The length of the list is equal to the number of disjoint regions in the mask.

Returns

polygon around each connected component of the mask. Each ndarray is an Nx2 array of xy points.

Return type List[ndarray]

Note: The returned polygon may not surround points that are only one pixel thick.

Example

```
>>> # xdoc: +REQUIRES(--mask)
>>> from kwimage.structs.mask import *  # NOQA
>>> self = Mask.random(shape=(8, 8), rng=0)
>>> polygons = self.get_polygon()
>>> print('polygons = ' + ub.repr2(polygons))
>>> polygons = self.get_polygon()
>>> self = self.to_bytes_rle()
>>> other = Mask.from_polygons(polygons, self.shape)
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> image = np.ones(self.shape)
>>> image = self.draw_on(image, color='blue')
>>> image = other.draw_on(image, color='red')
>>> kwplot.imshow(image)
```

polygons = [np.array([[6, 4],[7, 4]], dtype=np.int32), np.array([[0, 1],[0, 3],[2, 3],[2, 1]], dtype=np.int32),

]

to_mask(*self*, *dims=None*)

to_boxes(*self*)

Returns the bounding box of the mask.

classmethod demo(*cls*)
Demo mask with holes and disjoint shapes

to_multi_polygon(*self*)
Returns a MultiPolygon object fit around this raster including disjoint pieces and holes.

Returns vectorized representation

Return type *MultiPolygon*

Example

```
>>> # xdoc: +REQUIRES(--mask)
>>> from kwimage.structs.mask import *  # NOQA
>>> self = Mask.demo()
>>> self = self.scale(5)
>>> multi_poly = self.to_multi_polygon()
>>> # xdoc: +REQUIRES(module:kwplot)
>>> # xdoc: +REQUIRES(--show)
>>> self.draw(color='red')
>>> multi_poly.scale(1.1).draw(color='blue')
```

```
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> image = np.ones(self.shape)
>>> image = self.draw_on(image, color='blue')
>>> #image = other.draw_on(image, color='red')
>>> kwplot.imshow(image)
>>> multi_poly.draw()
```

get_convex_hull(*self*)
Returns a list of xy points around the convex hull of this mask

Note: The returned polygon may not surround points that are only one pixel thick.

Example

```
>>> # xdoc: +REQUIRES(--mask)
>>> self = Mask.random(shape=(8, 8), rng=0)
>>> polygons = self.get_convex_hull()
>>> print('polygons = ' + ub.repr2(polygons))
>>> other = Mask.from_polygons(polygons, self.shape)
```

iou(*self*, *other*)
The area of intersection over the area of union

Todo:

- **[ ] Write plural Masks version of this class, which should** be able to perform this operation more efficiently.

CommandLine: xdoctest -m kwimage.structs.mask Mask.iou

Example

```
>>> # xdoc: +REQUIRES(--mask)
>>> self = Mask.demo()
>>> other = self.translate(1)
>>> iou = self.iou(other)
>>> print('iou = {:.4f}'.format(iou))
iou = 0.0830
```

classmethod coerce(*Mask*, *data*, *dims=None*)

Attempts to auto-inspect the format of the data and conver to Mask

Parameters

- **data** – the data to coerce
- **dims** (*Tuple*) – required for certain formats like polygons height / width of the source image

Returns Mask

Example

```
>>> # xdoc: +REQUIRES(--mask)
>>> segmentation = {'size': [5, 9], 'counts': ';?1B10O30O4'}
>>> polygon = [
>>>     [np.array([[3, 0],[2, 1],[2, 4],[4, 4],[4, 3],[7, 0]])],
>>>     [np.array([[2, 1],[2, 2],[4, 2],[4, 1]])],
>>> ]
>>> dims = (9, 5)
>>> mask = (np.random.rand(32, 32) > .5).astype(np.uint8)
>>> Mask.coerce(polygon, dims).to_bytes_rle()
>>> Mask.coerce(segmentation).to_bytes_rle()
>>> Mask.coerce(mask).to_bytes_rle()
```

_to_coco(*self*)

Example

```
>>> # xdoc: +REQUIRES(--mask)
>>> from kwimage.structs.mask import *  # NOQA
>>> self = Mask.demo()
>>> data = self._to_coco()
>>> print(ub.repr2(data, nl=1))
```

to_coco(*self*, *style='orig'*)

Example

```
>>> # xdoc: +REQUIRES(--mask)
>>> from kwimage.structs.mask import *  # NOQA
>>> self = Mask.demo()
>>> data = self.to_coco()
>>> print(ub.repr2(data, nl=1))
```

class kwimage.**MaskList**

Bases: *kwimage.structs._generic.ObjectList*

Store and manipulate multiple masks, usually within the same image

to_polygon_list(*self*)

Converts all mask objects to polygon objects

class kwimage.**MultiPolygon**

Bases: *kwimage.structs._generic.ObjectList*

classmethod random(*self*, *n=3*, *rng=None*, *tight=False*)

to_multi_polygon(*self*)

to_mask(*self*, *dims=None*)

Returns a mask object indication regions occupied by this multipolygon

Example

```
>>> from kwimage.structs.polygon import *  # NOQA
>>> s = 100
>>> self = MultiPolygon.random(rng=0).scale(s)
>>> dims = (s, s)
>>> mask = self.to_mask(dims)
```

```
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.figure(fnum=1, doclf=True)
>>> from matplotlib import pyplot as pl
>>> ax = plt.gca()
>>> ax.set_xlim(0, s)
>>> ax.set_ylim(0, s)
>>> self.draw(color='red', alpha=.4)
>>> mask.draw(color='blue', alpha=.4)
```

classmethod coerce(*cls*, *data*, *dims=None*)

See Mask.coerce

to_shapely(*self*)

Example

```
>>> # xdoc: +REQUIRES(module:kwplot)
>>> # xdoc: +REQUIRES(module:shapely)
>>> from kwimage.structs.polygon import *  # NOQA
>>> self = MultiPolygon.random(rng=0)
>>> geom = self.to_shapely()
>>> print('geom = {!r}'.format(geom))
```

classmethod from_shapely(*MultiPolygon*, *geom*)

Convert a shapely polygon or multipolygon to a kwimage.MultiPolygon

classmethod from_geojson(*MultiPolygon*, *data_geojson*)

Convert a geojson polygon or multipolygon to a kwimage.MultiPolygon

Example

```
>>> import kwimage
>>> orig = kwimage.MultiPolygon.random()
>>> data_geojson = orig.to_geojson()
>>> self = kwimage.MultiPolygon.from_geojson(data_geojson)
```

to_geojson(*self*)
Converts polygon to a geojson structure

classmethod from_coco(*cls*, *data*, *dims=None*)
Accepts either new-style or old-style coco multi-polygons

_to_coco(*self*, *style='orig'*)

to_coco(*self*, *style='orig'*)

Example

```
>>> from kwimage.structs.polygon import *  # NOQA
>>> self = MultiPolygon.random(1, rng=0)
>>> self.to_coco()
```

class kwimage.**Points**(*data=None*, *meta=None*, *datakeys=None*, *metakeys=None*, ***kwargs*)
Bases: *kwimage.structs._generic.Spatial*, *kwimage.structs.points._PointsWarpMixin*

Stores multiple keypoints for a single object.

This stores both the geometry and the class metadata if available

Ignore:

meta = { "names" = ['head', 'nose', 'tail'], "skeleton" = [(0, 1), (0, 2)],

}

Example

```
>>> from kwimage.structs.points import *  # NOQA
>>> xy = np.random.rand(10, 2)
>>> pts = Points(xy=xy)
>>> print('pts = {!r}'.format(pts))
```

__datakeys__ = ['xy', 'class_idxs', 'visible']

__metakeys__ = ['classes']

__repr__

shape

xy

__nice__(*self*)

__len__(*self*)

classmethod random(*Points*, *num=1*, *classes=None*, *rng=None*)
Makes random points; typically for testing purposes

Example

```
>>> import kwimage
>>> self = kwimage.Points.random(classes=[1, 2, 3])
>>> self.data
>>> print('self.data = {!r}'.format(self.data))
```

is_numpy(*self*)

is_tensor(*self*)

_impl(*self*)

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

Example

```
>>> from kwimage.structs.points import *  # NOQA
>>> self = Points.random(10)
>>> self.tensor()
```

numpy(*self*)

Example

```
>>> from kwimage.structs.points import *  # NOQA
>>> self = Points.random(10)
>>> self.tensor().numpy().tensor().numpy()
```

draw_on(*self*, *image*, *color='white'*, *radius=None*, *copy=False*)

CommandLine: xdoctest -m ~/code/kwimage/kwimage/structs/points.py Points.draw_on –show

Example

```
>>> # xdoc: +REQUIRES(module:kwplot)
>>> from kwimage.structs.points import *  # NOQA
>>> s = 128
>>> image = np.zeros((s, s))
>>> self = Points.random(10).scale(s)
>>> image = self.draw_on(image)
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.figure(fnum=1, doclf=True)
>>> kwplot.autompl()
>>> kwplot.imshow(image)
>>> self.draw(radius=3, alpha=.5)
>>> kwplot.show_if_requested()
```

Example

```
>>> # xdoc: +REQUIRES(module:kwplot)
>>> from kwimage.structs.points import *  # NOQA
>>> s = 128
>>> image = np.zeros((s, s))
>>> self = Points.random(10).scale(s)
>>> image = self.draw_on(image, radius=3, color='distinct')
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.figure(fnum=1, doclf=True)
>>> kwplot.autompl()
>>> kwplot.imshow(image)
>>> self.draw(radius=3, alpha=.5, color='classes')
>>> kwplot.show_if_requested()
```

Example

```
>>> import kwimage
>>> s = 32
>>> self = kwimage.Points.random(10).scale(s)
>>> color = 'blue'
>>> # Test drawong on all channel + dtype combinations
>>> im3 = np.zeros((s, s, 3), dtype=np.float32)
>>> im_chans = {
>>>     'im3': im3,
>>>     'im1': kwimage.convert_colorspace(im3, 'rgb', 'gray'),
>>>     'im4': kwimage.convert_colorspace(im3, 'rgb', 'rgba'),
>>> }
>>> inputs = {}
>>> for k, im in im_chans.items():
>>>     inputs[k + '_01'] = (kwimage.ensure_float01(im.copy()), {'radius':␣
↪None})
>>>     inputs[k + '_255'] = (kwimage.ensure_uint255(im.copy()), {'radius':␣
↪None})
>>> outputs = {}
>>> for k, v in inputs.items():
>>>     im, kw = v
>>>     outputs[k] = self.draw_on(im, color=color, **kw)
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.figure(fnum=2, doclf=True)
>>> kwplot.autompl()
>>> pnum_ = kwplot.PlotNums(nCols=2, nRows=len(inputs))
>>> for k in inputs.keys():
>>>     kwplot.imshow(inputs[k][0], fnum=2, pnum=pnum_(), title=k)
>>>     kwplot.imshow(outputs[k], fnum=2, pnum=pnum_(), title=k)
>>> kwplot.show_if_requested()
```

draw(*self*, *color='blue'*, *ax=None*, *alpha=None*, *radius=1*, ***kwargs*)

Example

```
>>> # xdoc: +REQUIRES(module:kwplot)
>>> from kwimage.structs.points import *  # NOQA
>>> pts = Points.random(10)
```

(continues on next page)

(continued from previous page)

```
>>> # xdoc: +REQUIRES(--show)
>>> pts.draw(radius=0.01)
```

```
>>> from kwimage.structs.points import *  # NOQA
>>> self = Points.random(10, classes=['a', 'b', 'c'])
>>> self.draw(radius=0.01, color='classes')
```

compress (*self*, *flags*, *axis=0*, *inplace=False*)

Filters items based on a boolean criterion

Example

```
>>> from kwimage.structs.points import *  # NOQA
>>> self = Points.random(4)
>>> flags = [1, 0, 1, 1]
>>> other = self.compress(flags)
>>> assert len(self) == 4
>>> assert len(other) == 3
```

```
>>> other = self.tensor().compress(flags)
>>> assert len(other) == 3
```

take (*self*, *indices*, *axis=0*, *inplace=False*)

Takes a subset of items at specific indices

Example

```
>>> from kwimage.structs.points import *  # NOQA
>>> self = Points.random(4)
>>> indices = [1, 3]
>>> other = self.take(indices)
>>> assert len(self) == 4
>>> assert len(other) == 2
```

```
>>> other = self.tensor().take(indices)
>>> assert len(other) == 2
```

classmethod concatenate (*cls*, *points*, *axis=0*)

to_coco (*self*, *style='orig'*)

Converts to an mscoco-like representation

Note: items that are usually id-references to other objects may need to be rectified.

Parameters **style** (*str*) – either orig, new, new-id, or new-name

Returns mscoco-like representation

Return type Dict

Example

```
>>> from kwimage.structs.points import *  # NOQA
>>> self = Points.random(4, classes=['a', 'b'])
>>> orig = self._to_coco(style='orig')
>>> print('orig = {!r}'.format(orig))
>>> new_name = self._to_coco(style='new-name')
>>> print('new_name = {}'.format(ub.repr2(new_name, nl=-1)))
>>> # xdoctest: +REQUIRES(module:ndsampler)
>>> import ndsampler
>>> self.meta['classes'] = ndsampler.CategoryTree.coerce(self.meta['classes'])
>>> new_id = self._to_coco(style='new-id')
>>> print('new_id = {}'.format(ub.repr2(new_id, nl=-1)))
```

_to_coco(*self*, *style='orig'*)
See to_coco

classmethod _from_coco(*cls*, *coco_kpts*, *class_idxs=None*, *classes=None*)

classmethod from_coco(*cls*, *coco_kpts*, *class_idxs=None*, *classes=None*)

Parameters

- **coco_kpts** (*list | dict*) – either the original list keypoint encoding or the new dict keypoint encoding.
- **class_idxs** (*list*) – only needed if using old style
- **classes** (*list | CategoryTree*) – list of all keypoint category names

Example

```
>>> ##
>>> classes = ['mouth', 'left-hand', 'right-hand']
>>> coco_kpts = [
>>>     {'xy': (0, 0), 'visible': 2, 'keypoint_category': 'left-hand'},
>>>     {'xy': (1, 2), 'visible': 2, 'keypoint_category': 'mouth'},
>>> ]
>>> Points.from_coco(coco_kpts, classes=classes)
>>> # Test without classes
>>> Points.from_coco(coco_kpts)
>>> # Test without any category info
>>> coco_kpts2 = [ub.dict_diff(d, {'keypoint_category'}) for d in coco_kpts]
>>> Points.from_coco(coco_kpts2)
>>> # Test without category instead of keypoint_category
>>> coco_kpts3 = [ub.map_keys(lambda x: x.replace('keypoint_', ''), d) for d␣
↪in coco_kpts]
>>> Points.from_coco(coco_kpts3)
>>> #
>>> # Old style
>>> coco_kpts = [0, 0, 2, 0, 1, 2]
>>> Points.from_coco(coco_kpts)
>>> # Fail case
>>> coco_kpts4 = [{'xy': [4686.5, 1341.5], 'category': 'dot'}]
>>> Points.from_coco(coco_kpts4, classes=[])
```

Example

```
>>> # xdoctest: +REQUIRES(module:ndsampler)
>>> import ndsampler
>>> classes = ndsampler.CategoryTree.from_coco([
>>>     {'name': 'mouth', 'id': 2}, {'name': 'left-hand', 'id': 3}, {'name':
↪'right-hand', 'id': 5}
>>> ])
>>> coco_kpts = [
>>>     {'xy': (0, 0), 'visible': 2, 'keypoint_category_id': 5},
>>>     {'xy': (1, 2), 'visible': 2, 'keypoint_category_id': 2},
>>> ]
>>> pts = Points.from_coco(coco_kpts, classes=classes)
>>> assert pts.data['class_idxs'].tolist() == [2, 0]
```

class kwimage.**PointsList**

Bases: *kwimage.structs._generic.ObjectList*

Stores a list of Points, each item usually corresponds to a different object.

Notes

TODO: when the data is homogenous we can use a more efficient # representation, otherwise we have to use heterogenous storage.

class kwimage.**Polygon**(*data=None*, *meta=None*, *datakeys=None*, *metakeys=None*, ***kwargs*)

Bases: *kwimage.structs._generic.Spatial*, *kwimage.structs.polygon._PolyArrayBackend*, *kwimage.structs.polygon._PolyWarpMixin*, ubelt.NiceRepr

Represents a single polygon as set of exterior boundary points and a list of internal polygons representing holes.

By convention exterior boundaries should be counterclockwise and interior holes should be clockwise.

Example

```
>>> data = {
>>>     'exterior': np.array([[13,  1], [13, 19], [25, 19], [25,  1]]),
>>>     'interiors': [
>>>         np.array([[13, 13], [14, 12], [24, 12], [25, 13], [25, 18], [24, 19],␣
↪[14, 19], [13, 18]]),
>>>         np.array([[13,  2], [14,  1], [24,  1], [25, 2], [25, 11], [24, 12],␣
↪[14, 12], [13, 11]])]
>>> }
>>> self = Polygon(**data)
```

__datakeys__ = ['exterior', 'interiors']

__metakeys__ = ['classes']

__nice__(*self*)

classmethod circle(*cls*, *xy*, *r*, *resolution=64*)

Create a circular polygon

Example

```
>>> xy = (0.5, 0.5)
>>> r = .3
>>> poly = Polygon.circle(xy, r)
```

classmethod random(*cls*, *n=6*, *n_holes=0*, *convex=True*, *tight=False*, *rng=None*)

Parameters

- **n** (*int*) – number of points in the polygon (must be 3 or more)
- **n_holes** (*int*) – number of holes
- **tight** (*bool, default=False*) – fits the minimum and maximum points between 0 and 1
- **convex** (*bool, default=True*) – force resulting polygon will be convex (may remove exterior points)

CommandLine: xdoctest -m kwimage.structs.polygon Polygon.random

Example

```
>>> rng = None
>>> n = 4
>>> n_holes = 1
>>> cls = Polygon
>>> self = Polygon.random(n=n, rng=rng, n_holes=n_holes, convex=1)
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.figure(fnum=1, doclf=True)
>>> kwplot.autompl()
>>> self.draw()
```

References

https://gis.stackexchange.com/questions/207731/random-multipolygon https://stackoverflow.com/questions/8997099/random-polygon https://stackoverflow.com/questions/27548363/from-voronoi-tessellation-to-shapely-polygons https://stackoverflow.com/questions/8997099/algorithm-to-generate-random-2d-polygon

_impl(*self*)

to_mask(*self*, *dims=None*)

Convert this polygon to a mask

Todo:

- [ ] currently not efficient

Example

```
>>> from kwimage.structs.polygon import *  # NOQA
>>> self = Polygon.random(n_holes=1).scale(128)
>>> mask = self.to_mask((128, 128))
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.figure(fnum=1, doclf=True)
>>> mask.draw(color='blue')
>>> mask.to_multi_polygon().draw(color='red', alpha=.5)
```

fill(*self*, *image*, *value=1*)
Inplace fill in an image based on this polyon.

_to_cv_countours(*self*)
OpenCV polygon representation, which is a list of points. Holes are implicitly represented. When another polygon is drawn over an existing polyon via cv2.fillPoly

Returns

where each ndarray is of shape [N, 1, 2], where N is the number of points on the boundary, the middle dimension is always 1, and the trailing dimension represents x and y coordinates respectively.

Return type List[ndarray]

to_shapely(*self*)

Example

```
>>> # xdoc: +REQUIRES(module:kwplot)
>>> # xdoc: +REQUIRES(module:shapely)
>>> from kwimage.structs.polygon import *  # NOQA
>>> self = Polygon.random(n_holes=1)
>>> self = self.scale(100)
>>> geom = self.to_shapely()
>>> print('geom = {!r}'.format(geom))
```

classmethod from_shapely(*Polygon*, *geom*)
Convert a shapely polygon to a kwimage.Polygon

Parameters **geom** (*shapely.geometry.polygon.Polygon*) – a shapely polygon

classmethod from_wkt(*Polygon*, *data*)
Convert a WKT string to a kwimage.Polygon

Parameters **data** (*str*) – a WKT polygon string

Example

data = kwimage.Polygon.random().to_shapely().to_wkt() data = 'POLYGON ((0.11 0.61, 0.07 0.588, 0.015 0.50, 0.11 0.61))' self = Polygon.from_wkt(data)

classmethod from_geojson(*MultiPolygon*, *data_geojson*)
Convert a geojson polygon to a kwimage.Polygon

Parameters **data_geojson** (*dict*) – geojson data

Example

```
>>> self = Polygon.random(n_holes=2)
>>> data_geojson = self.to_geojson()
>>> new = Polygon.from_geojson(data_geojson)
```

classmethod from_coco(*cls*, *data*, *dims=None*)
Accepts either new-style or old-style coco polygons

draw_on(*self*, *image*, *color='blue'*, *fill=True*, *border=False*, *alpha=1.0*, *copy=False*)
Rasterizes a polygon on an image. See *draw* for a vectorized matplotlib version.

Parameters

- **image** (*ndarray*) – image to raster polygon on.
- **color** (*str | tuple*) – data coercable to a color
- **fill** (*bool, default=True*) – draw the center mass of the polygon
- **border** (*bool, default=False*) – draw the border of the polygon
- **alpha** (*float, default=1.0*) – polygon transparency (setting alpha < 1 makes this function much slower).
- **copy** (*bool, default=False*) – if False only copies if necessary

Example

```
>>> # xdoc: +REQUIRES(module:kwplot)
>>> from kwimage.structs.polygon import *  # NOQA
>>> self = Polygon.random(n_holes=1).scale(128)
>>> image = np.zeros((128, 128), dtype=np.float32)
>>> image = self.draw_on(image)
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(image, fnum=1)
```

Example

```
>>> import kwimage
>>> color = 'blue'
>>> self = kwimage.Polygon.random(n_holes=1).scale(128)
>>> image = np.zeros((128, 128), dtype=np.float32)
>>> # Test drawong on all channel + dtype combinations
>>> im3 = np.random.rand(128, 128, 3)
>>> im_chans = {
>>>     'im3': im3,
>>>     'im1': kwimage.convert_colorspace(im3, 'rgb', 'gray'),
>>>     'im4': kwimage.convert_colorspace(im3, 'rgb', 'rgba'),
>>> }
>>> inputs = {}
>>> for k, im in im_chans.items():
>>>     inputs[k + '_01'] = (kwimage.ensure_float01(im.copy()), {'alpha':␣
↪None})
>>>     inputs[k + '_255'] = (kwimage.ensure_uint255(im.copy()), {'alpha':␣
↪None})
```

(continues on next page)

(continued from previous page)

```
>>>     inputs[k + '_01_a'] = (kwimage.ensure_float01(im.copy()), {'alpha': 0.
↪5})
>>>     inputs[k + '_255_a'] = (kwimage.ensure_uint255(im.copy()), {'alpha':␣
↪0.5})
>>> outputs = {}
>>> for k, v in inputs.items():
>>>     im, kw = v
>>>     outputs[k] = self.draw_on(im, color=color, **kw)
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.figure(fnum=2, doclf=True)
>>> kwplot.autompl()
>>> pnum_ = kwplot.PlotNums(nCols=2, nRows=len(inputs))
>>> for k in inputs.keys():
>>>     kwplot.imshow(inputs[k][0], fnum=2, pnum=pnum_(), title=k)
>>>     kwplot.imshow(outputs[k], fnum=2, pnum=pnum_(), title=k)
>>> kwplot.show_if_requested()
```

draw(*self*, *color='blue'*, *ax=None*, *alpha=1.0*, *radius=1*, *setlim=False*, *border=False*, *linewidth=2*)
Draws polygon in a matplotlib axes. See *draw_on* for in-memory image modification.

Example

```
>>> # xdoc: +REQUIRES(module:kwplot)
>>> from kwimage.structs.polygon import *  # NOQA
>>> self = Polygon.random(n_holes=1)
>>> self = self.scale(100)
>>> # xdoc: +REQUIRES(--show)
>>> self.draw()
>>> import kwplot
>>> kwplot.autompl()
>>> from matplotlib import pyplot as plt
>>> kwplot.figure(fnum=2)
>>> self.draw(setlim=True)
```

_to_coco(*self*, *style='orig'*)

to_coco(*self*, *style='orig'*)

to_geojson(*self*)
Converts polygon to a geojson structure

to_multi_polygon(*self*)

to_boxes(*self*)

copy(*self*)

clip(*self*, *x_min*, *y_min*, *x_max*, *y_max*, *inplace=False*)
Clip polygon to image boundaries.

Example

```
>>> from kwimage.structs.polygon import *
>>> self = Polygon.random().scale(10).translate(-1)
```

(continues on next page)

(continued from previous page)

```
>>> self2 = self.clip(1, 1, 3, 3)
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> self2.draw(setlim=True)
```

class kwimage.**PolygonList**

Bases: *kwimage.structs._generic.ObjectList*

to_polygon_list(*self*)

kwimage.**smooth_prob**(*prob*, *k=3*, *inplace=False*, *eps=1e-09*)

Smooths the probability map, but preserves the magnitude of the peaks.

Notes

even if inplace is true, we still need to make a copy of the input array, however, we do ensure that it is cleaned up before we leave the function scope.

sigma=0.8 @ k=3, sigma=1.1 @ k=5, sigma=1.4 @ k=7

kwimage.**TORCH_GRID_SAMPLE_HAS_ALIGN**

kwimage.**subpixel_accum**(*dst*, *src*, *index*, *interp_axes=None*)

Add the source values array into the destination array at a particular subpixel index.

Parameters

- **dst** (*ArrayLike*) – destination accumulation array
- **src** (*ArrayLike*) – source array containing values to add
- **index** (*Tuple[slice]*) – subpixel slice into dst that corresponds with src
- **interp_axes** (*tuple*) – specify which axes should be spatially interpolated

Notes

Inputs:

+—+—+—+—+—+ **dst.shape = (5,)** +—+—+ src.shape = (2,) |=======| index = 1.5:3.5

Subpixel shift the source by -0.5. When the index is non-integral, pad the aligned src with an extra value to ensure all dst pixels that would be influenced by the smaller subpixel shape are influenced by the aligned src. Note that we are not scaling.

+—+—+—+—+ aligned_src.shape = (3,) |==========| aligned_index = 1:4

Example

```
>>> dst = np.zeros(5)
>>> src = np.ones(2)
>>> index = [slice(1.5, 3.5)]
>>> subpixel_accum(dst, src, index)
>>> print(ub.repr2(dst, precision=2, with_dtype=0))
np.array([0. , 0.5, 1. , 0.5, 0. ])
```

Example

```
>>> dst = np.zeros((6, 6))
>>> src = np.ones((3, 3))
>>> index = (slice(1.5, 4.5), slice(1, 4))
>>> subpixel_accum(dst, src, index)
>>> print(ub.repr2(dst, precision=2, with_dtype=0))
np.array([[0. , 0. , 0. , 0. , 0. , 0. ],
          [0. , 0.5, 0.5, 0.5, 0. , 0. ],
          [0. , 1. , 1. , 1. , 0. , 0. ],
          [0. , 1. , 1. , 1. , 0. , 0. ],
          [0. , 0.5, 0.5, 0.5, 0. , 0. ],
          [0. , 0. , 0. , 0. , 0. , 0. ]])
>>> dst = torch.zeros((1, 3, 6, 6))
>>> src = torch.ones((1, 3, 3, 3))
>>> index = (slice(None), slice(None), slice(1.5, 4.5), slice(1.25, 4.25))
>>> subpixel_accum(dst, src, index)
>>> print(ub.repr2(dst.numpy()[0, 0], precision=2, with_dtype=0))
np.array([[0.  , 0.  , 0.  , 0.  , 0.  , 0.  ],
          [0.  , 0.38, 0.5 , 0.5 , 0.12, 0.  ],
          [0.  , 0.75, 1.  , 1.  , 0.25, 0.  ],
          [0.  , 0.75, 1.  , 1.  , 0.25, 0.  ],
          [0.  , 0.38, 0.5 , 0.5 , 0.12, 0.  ],
          [0.  , 0.  , 0.  , 0.  , 0.  , 0.  ]])
```

Doctest:

```
>>> # TODO: move to a unit test file
>>> subpixel_accum(np.zeros(5), np.ones(2), [slice(1.5, 3.5)]).tolist()
[0.0, 0.5, 1.0, 0.5, 0.0]
>>> subpixel_accum(np.zeros(5), np.ones(2), [slice(0, 2)]).tolist()
[1.0, 1.0, 0.0, 0.0, 0.0]
>>> subpixel_accum(np.zeros(5), np.ones(3), [slice(.5, 3.5)]).tolist()
[0.5, 1.0, 1.0, 0.5, 0.0]
>>> subpixel_accum(np.zeros(5), np.ones(3), [slice(-1, 2)]).tolist()
[1.0, 1.0, 0.0, 0.0, 0.0]
>>> subpixel_accum(np.zeros(5), np.ones(3), [slice(-1.5, 1.5)]).tolist()
[1.0, 0.5, 0.0, 0.0, 0.0]
>>> subpixel_accum(np.zeros(5), np.ones(3), [slice(10, 13)]).tolist()
[0.0, 0.0, 0.0, 0.0, 0.0]
>>> subpixel_accum(np.zeros(5), np.ones(3), [slice(3.25, 6.25)]).tolist()
[0.0, 0.0, 0.0, 0.75, 1.0]
>>> subpixel_accum(np.zeros(5), np.ones(3), [slice(4.9, 7.9)]).tolist()
[0.0, 0.0, 0.0, 0.0, 0.099...]
>>> subpixel_accum(np.zeros(5), np.ones(9), [slice(-1.5, 7.5)]).tolist()
[1.0, 1.0, 1.0, 1.0, 1.0]
>>> subpixel_accum(np.zeros(5), np.ones(9), [slice(2.625, 11.625)]).tolist()
[0.0, 0.0, 0.375, 1.0, 1.0]
>>> subpixel_accum(np.zeros(5), 1, [slice(2.625, 11.625)]).tolist()
[0.0, 0.0, 0.375, 1.0, 1.0]
```

kwimage.**subpixel_align**(*dst*, *src*, *index*, *interp_axes=None*)

Returns an aligned version of the source tensor and destination index.

Used as the backend to implement other subpixel functions like: subpixel_accum, subpixel_maximum.

kwimage.**subpixel_getvalue**(*img*, *pts*, *coord_axes=None*, *interp='bilinear'*, *bordermode='edge'*)

Get values at subpixel locations

Parameters

- **img** (*ArrayLike*) – image to sample from
- **pts** (*ArrayLike*) – subpixel rc-coordinates to sample
- **coord_axes** (*Sequence, default=None*) – axes to perform interpolation on, if not specified the first *d* axes are interpolated, where *d=pts.shape[-1]*. IE: this indicates which axes each coordinate dimension corresponds to.
- **interp** (*str*) – interpolation mode
- **bordermode** (*str*) – how locations outside the image are handled

Example

```
>>> from kwimage.util_warp import *  # NOQA
>>> img = np.arange(3 * 3).reshape(3, 3)
>>> pts = np.array([[1, 1], [1.5, 1.5], [1.9, 1.1]])
>>> subpixel_getvalue(img, pts)
array([4. , 6. , 6.8])
>>> subpixel_getvalue(img, pts, coord_axes=(1, 0))
array([4. , 6. , 5.2])
>>> img = torch.Tensor(img)
>>> pts = torch.Tensor(pts)
>>> subpixel_getvalue(img, pts)
tensor([4.0000, 6.0000, 6.8000])
>>> subpixel_getvalue(img.numpy(), pts.numpy(), interp='nearest')
array([4., 8., 7.], dtype=float32)
>>> subpixel_getvalue(img.numpy(), pts.numpy(), interp='nearest', coord_axes=[1,␣
↪0])
array([4., 8., 5.], dtype=float32)
>>> subpixel_getvalue(img, pts, interp='nearest')
tensor([4., 8., 7.])
```

References

stackoverflow.com/uestions/12729228/simple-binlin-interp-images-numpy

SeeAlso: cv2.getRectSubPix(image, patchSize, center[, patch[, patchType]])

kwimage.**subpixel_maximum**(*dst*, *src*, *index*, *interp_axes=None*)

Take the max of the source values array into and the destination array at a particular subpixel index. Modifies the destination array.

Parameters

- **dst** (*ArrayLike*) – destination array to index into
- **src** (*ArrayLike*) – source array that agrees with the index
- **index** (*Tuple[slice]*) – subpixel slice into dst that corresponds with src
- **interp_axes** (*tuple*) – specify which axes should be spatially interpolated

Example

```
>>> dst = np.array([0, 1.0, 1.0, 1.0, 0])
>>> src = np.array([2.0, 2.0])
>>> index = [slice(1.6, 3.6)]
>>> subpixel_maximum(dst, src, index)
>>> print(ub.repr2(dst, precision=2, with_dtype=0))
np.array([0. , 1. , 2. , 1.2, 0. ])
```

Example

```
>>> dst = torch.zeros((1, 3, 5, 5)) + .5
>>> src = torch.ones((1, 3, 3, 3))
>>> index = (slice(None), slice(None), slice(1.4, 4.4), slice(1.25, 4.25))
>>> subpixel_maximum(dst, src, index)
>>> print(ub.repr2(dst.numpy()[0, 0], precision=2, with_dtype=0))
np.array([[0.5 , 0.5 , 0.5 , 0.5 , 0.5 ],
          [0.5 , 0.5 , 0.6 , 0.6 , 0.5 ],
          [0.5 , 0.75, 1.  , 1.  , 0.5 ],
          [0.5 , 0.75, 1.  , 1.  , 0.5 ],
          [0.5 , 0.5 , 0.5 , 0.5 , 0.5 ]])
```

kwimage.**subpixel_minimum**(*dst*, *src*, *index*, *interp_axes=None*)

Take the min of the source values array into and the destination array at a particular subpixel index. Modifies the destination array.

Parameters

- **dst** (*ArrayLike*) – destination array to index into
- **src** (*ArrayLike*) – source array that agrees with the index
- **index** (*Tuple[slice]*) – subpixel slice into dst that corresponds with src
- **interp_axes** (*tuple*) – specify which axes should be spatially interpolated

Example

```
>>> dst = np.array([0, 1.0, 1.0, 1.0, 0])
>>> src = np.array([2.0, 2.0])
>>> index = [slice(1.6, 3.6)]
>>> subpixel_minimum(dst, src, index)
>>> print(ub.repr2(dst, precision=2, with_dtype=0))
np.array([0. , 0.8, 1. , 1. , 0. ])
```

Example

```
>>> dst = torch.zeros((1, 3, 5, 5)) + .5
>>> src = torch.ones((1, 3, 3, 3))
>>> index = (slice(None), slice(None), slice(1.4, 4.4), slice(1.25, 4.25))
>>> subpixel_minimum(dst, src, index)
>>> print(ub.repr2(dst.numpy()[0, 0], precision=2, with_dtype=0))
np.array([[0.5 , 0.5 , 0.5 , 0.5 , 0.5 ],
          [0.5 , 0.45, 0.5 , 0.5 , 0.15],
```

(continues on next page)

(continued from previous page)

```
            [0.5 , 0.5 , 0.5 , 0.5 , 0.25],
            [0.5 , 0.5 , 0.5 , 0.5 , 0.25],
            [0.5 , 0.3 , 0.4 , 0.4 , 0.1 ]])
```

kwimage.**subpixel_set**(*dst*, *src*, *index*, *interp_axes=None*)

Add the source values array into the destination array at a particular subpixel index.

Parameters

- **dst** (*ArrayLike*) – destination accumulation array
- **src** (*ArrayLike*) – source array containing values to add
- **index** (*Tuple[slice]*) – subpixel slice into dst that corresponds with src
- **interp_axes** (*tuple*) – specify which axes should be spatially interpolated

---

Todo:

- [ ]: allow index to be a sequence indices

Example

```
>>> import kwimage
>>> dst = np.zeros(5) + .1
>>> src = np.ones(2)
>>> index = [slice(1.5, 3.5)]
>>> kwimage.util_warp.subpixel_set(dst, src, index)
>>> print(ub.repr2(dst, precision=2, with_dtype=0))
np.array([0.1, 0.5, 1. , 0.5, 0.1])
```

kwimage.**subpixel_setvalue**(*img*, *pts*, *value*, *coord_axes=None*, *interp='bilinear'*, *bordermode='edge'*)

Set values at subpixel locations

Parameters

- **img** (*ArrayLike*) – image to set values in
- **pts** (*ArrayLike*) – subpixel rc-coordinates to set
- **value** (*ArrayLike*) – value to place in the image
- **coord_axes** (*Sequence, default=None*) – axes to perform interpolation on, if not specified the first *d* axes are interpolated, where *d=pts.shape[-1]*. IE: this indicates which axes each coordinate dimension corresponds to.
- **interp** (*str*) – interpolation mode
- **bordermode** (*str*) – how locations outside the image are handled

Example

```
>>> from kwimage.util_warp import *  # NOQA
>>> img = np.arange(3 * 3).reshape(3, 3).astype(np.float)
>>> pts = np.array([[1, 1], [1.5, 1.5], [1.9, 1.1]])
```

(continues on next page)

(continued from previous page)

```
>>> interp = 'bilinear'
>>> value = 0
>>> print('img = {!r}'.format(img))
>>> pts = np.array([[1.5, 1.5]])
>>> img2 = subpixel_setvalue(img.copy(), pts, value)
>>> print('img2 = {!r}'.format(img2))
>>> pts = np.array([[1.0, 1.0]])
>>> img2 = subpixel_setvalue(img.copy(), pts, value)
>>> print('img2 = {!r}'.format(img2))
>>> pts = np.array([[1.1, 1.9]])
>>> img2 = subpixel_setvalue(img.copy(), pts, value)
>>> print('img2 = {!r}'.format(img2))
>>> img2 = subpixel_setvalue(img.copy(), pts, value, coord_axes=[1, 0])
>>> print('img2 = {!r}'.format(img2))
```

kwimage.**subpixel_slice**(*inputs*, *index*)

Take a subpixel slice from a larger image. The returned output is left-aligned with the requested slice.

Parameters

- **inputs** (*ArrayLike*) – data
- **index** (*Tuple[slice]*) – a slice to subpixel accuracy

Example

```
>>> inputs = np.arange(5 * 5 * 3).reshape(5, 5, 3)
>>> index = [slice(0, 3), slice(0, 3)]
>>> outputs = subpixel_slice(inputs, index)
>>> index = [slice(0.5, 3.5), slice(-0.5, 2.5)]
>>> outputs = subpixel_slice(inputs, index)
```

```
>>> inputs = np.arange(5 * 5).reshape(1, 5, 5).astype(np.float)
>>> index = [slice(None), slice(3, 6), slice(3, 6)]
>>> outputs = subpixel_slice(inputs, index)
>>> print(outputs)
[[[18. 19.  0.]
  [23. 24.  0.]
  [ 0.  0.  0.]]]
>>> index = [slice(None), slice(3.5, 6.5), slice(2.5, 5.5)]
>>> outputs = subpixel_slice(inputs, index)
>>> print(outputs)
[[[20.   21.   10.75]
  [11.25 11.75  6.  ]
  [ 0.    0.    0.  ]]]
```

kwimage.**subpixel_translate**(*inputs*, *shift*, *interp_axes=None*, *output_shape=None*)

Translates an image by a subpixel shift value using bilinear interpolation

Parameters

- **inputs** (*ArrayLike*) – data to translate
- **shift** (*Sequence*) – amount to translate each dimension specified by *interp_axes*. Note: if inputs contains more than one "image" then all "images" are translated by the same amount. This function contains no mechanism for translating each image differently. Note that by default this is a y,x shift for 2 dimensions.

- **interp_axes** (*Sequence, default=None*) – axes to perform interpolation on, if not specified the final *n* axes are interpolated, where *n=len(shift)*
- **output_shape** (*tuple, default=None*) – if specified the output is returned with this shape, otherwise

Notes

This function powers most other functions in this file. Speedups here can go a long way.

Example

```
>>> inputs = np.arange(5) + 1
>>> print(inputs.tolist())
[1, 2, 3, 4, 5]
>>> outputs = subpixel_translate(inputs, 1.5)
>>> print(outputs.tolist())
[0.0, 0.5, 1.5, 2.5, 3.5]
```

Example

```
>>> inputs = torch.arange(9).view(1, 1, 3, 3).float()
>>> print(inputs.long())
tensor([[[[0, 1, 2],
          [3, 4, 5],
          [6, 7, 8]]]])
>>> outputs = subpixel_translate(inputs, (-.4, .5), output_shape=(1, 1, 2, 5))
>>> print(outputs)
tensor([[[[0.6000, 1.7000, 2.7000, 1.6000, 0.0000],
          [2.1000, 4.7000, 5.7000, 3.1000, 0.0000]]]])
```

Ignore:

```
>>> inputs = np.arange(5)
>>> shift = -.6
>>> interp_axes = None
>>> subpixel_translate(inputs, -.6)
>>> subpixel_translate(inputs[None, None, None, :], -.6)
>>> inputs = np.arange(25).reshape(5, 5)
>>> shift = (-1.6, 2.3)
>>> interp_axes = (0, 1)
>>> subpixel_translate(inputs, shift, interp_axes, output_shape=(9, 9))
>>> subpixel_translate(inputs, shift, interp_axes, output_shape=(3, 4))
```

kwimage.**warp_points**(*matrix*, *pts*)

Warp ND points / coordinates using a transformation matrix.

Homogoenous coordinates are added on the fly if needed. Works with both numpy and torch.

Parameters

- **matrix** (*ArrayLike*) – [D1 x D2] transformation matrix. if using homogenous coordinates D2=D + 1, otherwise D2=D. if using homogenous coordinates and the matrix represents an

Affine transformation, then either D1=D or D1=D2, i.e. the last row of zeros and a one is optional.

- **pts** (*ArrayLike*) – [N1 x ... x D] points (usually x, y). If points are already in homogenous space, then the output will be returned in homogenous space. D is the dimensionality of the points. The leading axis may take any shape, but usually, shape will be [N x D] where N is the number of points.

Retrns: new_pts (ArrayLike): the points after being transformed by the matrix

Example

```
>>> from kwimage.util_warp import *  # NOQA
>>> # --- with numpy
>>> rng = np.random.RandomState(0)
>>> pts = rng.rand(10, 2)
>>> matrix = rng.rand(2, 2)
>>> warp_points(matrix, pts)
>>> # --- with torch
>>> pts = torch.Tensor(pts)
>>> matrix = torch.Tensor(matrix)
>>> warp_points(matrix, pts)
```

Example

```
>>> from kwimage.util_warp import *  # NOQA
>>> # --- with numpy
>>> pts = np.ones((10, 2))
>>> matrix = np.diag([2, 3, 1])
>>> ra = warp_points(matrix, pts)
>>> rb = warp_points(torch.Tensor(matrix), torch.Tensor(pts))
>>> assert np.allclose(ra, rb.numpy())
```

Example

```
>>> from kwimage.util_warp import *  # NOQA
>>> # test different cases
>>> rng = np.random.RandomState(0)
>>> # Test 3x3 style projective matrices
>>> pts = rng.rand(1000, 2)
>>> matrix = rng.rand(3, 3)
>>> ra33 = warp_points(matrix, pts)
>>> rb33 = warp_points(torch.Tensor(matrix), torch.Tensor(pts))
>>> assert np.allclose(ra33, rb33.numpy())
>>> # Test opencv style affine matrices
>>> pts = rng.rand(10, 2)
>>> matrix = rng.rand(2, 3)
>>> ra23 = warp_points(matrix, pts)
>>> rb23 = warp_points(torch.Tensor(matrix), torch.Tensor(pts))
>>> assert np.allclose(ra33, rb33.numpy())
```

kwimage.**warp_tensor**(*inputs*, *mat*, *output_dims*, *mode='bilinear'*, *padding_mode='zeros'*, *isinv=False*, *ishomog=None*, *align_corners=False*, *new_mode=False*)

A pytorch implementation of warp affine that works similarly to cv2.warpAffine / cv2.warpPerspective.

It is possible to use 3x3 transforms to warp 2D image data. It is also possible to use 4x4 transforms to warp 3D volumetric data.

Parameters

- **inputs** (*Tensor[..., *DIMS]*) – tensor to warp. Up to 3 (determined by output_dims) of the trailing space-time dimensions are warped. Best practice is to use inputs with the shape in [B, C, *DIMS].
- **mat** (*Tensor*) – either a 3x3 / 4x4 single transformation matrix to apply to all inputs or Bx3x3 or Bx4x4 tensor that specifies a transformation matrix for each batch item.
- **output_dims** (*Tuple[int]*) –

  The output space-time dimensions. This can either be in the form (W,), (H, W), or (D, H, W).
- **mode** (*str*) – Can be bilinear or nearest. See *torch.nn.functional.grid_sample*
- **padding_mode** (*str*) – Can be zeros, border, or reflection. See *torch.nn.functional.grid_sample*.
- **isinv** (*bool, default=False*) – Set to true if *mat* is the inverse transform
- **ishomog** (*bool, default=None*) – Set to True if the matrix is non-affine
- **align_corners** (*bool, default=False*) – Note the default of False does not work correctly with grid_sample in torch <= 1.2, but using align_corners=True isnt typically what you want either. We will be stuck with buggy functionality until torch 1.3 is released.

  However, using align_corners=0 does seem to reasonably correspond with opencv behavior.

Notes

Also, it may be possible to speed up the code with *F.affine_grid*

KNOWN ISSUE: There appears to some difference with cv2.warpAffine when rotation or shear are non-zero. I'm not sure what the cause is. It may just be floating point issues, but Im' not sure.

Todo:

- [ ] FIXME: see example in Mask.scale where this algo breaks when

the matrix is *2x3* - [ ] Make this algo work when matrix ix 2x2

References

https://discuss.pytorch.org/t/affine-transformation-matrix-paramters-conversion/19522 https://github.com/pytorch/pytorch/issues/15386

Example

```
>>> # Create a relatively simple affine matrix
>>> import skimage
>>> mat = torch.FloatTensor(skimage.transform.AffineTransform(
>>>     translation=[1, -1], scale=[.532, 2],
>>>     rotation=0, shear=0,
```

(continues on next page)

(continued from previous page)

```
>>> ).params)
>>> # Create inputs and an output dimension
>>> input_shape = [1, 1, 4, 5]
>>> inputs = torch.arange(int(np.prod(input_shape))).reshape(*input_shape).float()
>>> output_dims = (11, 7)
>>> # Warp with our code
>>> result1 = warp_tensor(inputs, mat, output_dims=output_dims, align_corners=0)
>>> print('result1 =\n{}'.format(ub.repr2(result1.cpu().numpy()[0, 0],␣
↪precision=2)))
>>> # Warp with opencv
>>> import cv2
>>> cv2_M = mat.cpu().numpy()[0:2]
>>> src = inputs[0, 0].cpu().numpy()
>>> dsize = tuple(output_dims[::-1])
>>> result2 = cv2.warpAffine(src, cv2_M, dsize=dsize, flags=cv2.INTER_LINEAR)
>>> print('result2 =\n{}'.format(ub.repr2(result2, precision=2)))
>>> # Ensure the results are the same (up to floating point errors)
>>> assert np.all(np.isclose(result1[0, 0].cpu().numpy(), result2, atol=1e-2,␣
↪rtol=1e-2))
```

Example

```
>>> # Create a relatively simple affine matrix
>>> import skimage
>>> mat = torch.FloatTensor(skimage.transform.AffineTransform(
>>>     rotation=0.01, shear=0.1).params)
>>> # Create inputs and an output dimension
>>> input_shape = [1, 1, 4, 5]
>>> inputs = torch.arange(int(np.prod(input_shape))).reshape(*input_shape).float()
>>> output_dims = (11, 7)
>>> # Warp with our code
>>> result1 = warp_tensor(inputs, mat, output_dims=output_dims)
>>> print('result1 =\n{}'.format(ub.repr2(result1.cpu().numpy()[0, 0],␣
↪precision=2, supress_small=True)))
>>> print('result1.shape = {}'.format(result1.shape))
>>> # Warp with opencv
>>> import cv2
>>> cv2_M = mat.cpu().numpy()[0:2]
>>> src = inputs[0, 0].cpu().numpy()
>>> dsize = tuple(output_dims[::-1])
>>> result2 = cv2.warpAffine(src, cv2_M, dsize=dsize, flags=cv2.INTER_LINEAR)
>>> print('result2 =\n{}'.format(ub.repr2(result2, precision=2)))
>>> print('result2.shape = {}'.format(result2.shape))
>>> # Ensure the results are the same (up to floating point errors)
>>> # NOTE: The floating point errors seem to be significant for rotation / shear
>>> assert np.all(np.isclose(result1[0, 0].cpu().numpy(), result2, atol=1,␣
↪rtol=1e-2))
```

Example

```
>>> # Create a random affine matrix
>>> import skimage
>>> rng = np.random.RandomState(0)
```

(continues on next page)

(continued from previous page)

```
>>> mat = torch.FloatTensor(skimage.transform.AffineTransform(
>>>     translation=rng.randn(2), scale=1 + rng.randn(2),
>>>     rotation=rng.randn() / 10., shear=rng.randn() / 10.,
>>> ).params)
>>> # Create inputs and an output dimension
>>> input_shape = [1, 1, 5, 7]
>>> inputs = torch.arange(int(np.prod(input_shape))).reshape(*input_shape).float()
>>> output_dims = (3, 11)
>>> # Warp with our code
>>> result1 = warp_tensor(inputs, mat, output_dims=output_dims, align_corners=0)
>>> print('result1 =\n{}'.format(ub.repr2(result1.cpu().numpy()[0, 0],␣
↪precision=2)))
>>> # Warp with opencv
>>> import cv2
>>> cv2_M = mat.cpu().numpy()[0:2]
>>> src = inputs[0, 0].cpu().numpy()
>>> dsize = tuple(output_dims[::-1])
>>> result2 = cv2.warpAffine(src, cv2_M, dsize=dsize, flags=cv2.INTER_LINEAR)
>>> print('result2 =\n{}'.format(ub.repr2(result2, precision=2)))
>>> # Ensure the results are the same (up to floating point errors)
>>> # NOTE: The errors seem to be significant for rotation / shear
>>> assert np.all(np.isclose(result1[0, 0].cpu().numpy(), result2, atol=1,␣
↪rtol=1e-2))
```

Example

```
>>> # Test 3D warping with identity
>>> mat = torch.eye(4)
>>> input_dims = [2, 3, 3]
>>> output_dims = (2, 3, 3)
>>> input_shape = [1, 1] + input_dims
>>> inputs = torch.arange(int(np.prod(input_shape))).reshape(*input_shape).float()
>>> result = warp_tensor(inputs, mat, output_dims=output_dims)
>>> print('result =\n{}'.format(ub.repr2(result.cpu().numpy()[0, 0],␣
↪precision=2)))
>>> assert torch.all(inputs == result)
```

Example

```
>>> # Test 3D warping with scaling
>>> mat = torch.FloatTensor([
>>>     [0.8,   0,   0, 0],
>>>     [  0, 1.0,   0, 0],
>>>     [  0,   0, 1.2, 0],
>>>     [  0,   0,   0, 1],
>>> ])
>>> input_dims = [2, 3, 3]
>>> output_dims = (2, 3, 3)
>>> input_shape = [1, 1] + input_dims
>>> inputs = torch.arange(int(np.prod(input_shape))).reshape(*input_shape).float()
>>> result = warp_tensor(inputs, mat, output_dims=output_dims, align_corners=0)
>>> print('result =\n{}'.format(ub.repr2(result.cpu().numpy()[0, 0],␣
↪precision=2)))
```

(continues on next page)

(continued from previous page)

```
result =
np.array([[[ 0.  ,  1.25,  1.  ],
           [ 3.  ,  4.25,  2.5 ],
           [ 6.  ,  7.25,  4.  ]],
          ...
          [[ 7.5 ,  8.75,  4.75],
           [10.5 , 11.75,  6.25],
           [13.5 , 14.75,  7.75]]], dtype=np.float32)
```

Example

```
>>> mat = torch.eye(3)
>>> input_dims = [5, 7]
>>> output_dims = (11, 7)
>>> for n_prefix_dims in [0, 1, 2, 3, 4, 5]:
>>>     input_shape = [2] * n_prefix_dims + input_dims
>>>     inputs = torch.arange(int(np.prod(input_shape))).reshape(*input_shape).
↪float()
>>>     result = warp_tensor(inputs, mat, output_dims=output_dims)
>>>     #print('result =\n{}'.format(ub.repr2(result.cpu().numpy(),␣
↪precision=2)))
>>>     print(result.shape)
```

Example

```
>>> mat = torch.eye(4)
>>> input_dims = [5, 5, 5]
>>> output_dims = (6, 6, 6)
>>> for n_prefix_dims in [0, 1, 2, 3, 4, 5]:
>>>     input_shape = [2] * n_prefix_dims + input_dims
>>>     inputs = torch.arange(int(np.prod(input_shape))).reshape(*input_shape).
↪float()
>>>     result = warp_tensor(inputs, mat, output_dims=output_dims)
>>>     #print('result =\n{}'.format(ub.repr2(result.cpu().numpy(),␣
↪precision=2)))
>>>     print(result.shape)
```

Ignore: import xdev globals().update(xdev.get_func_kwargs(warp_tensor)) >>> import cv2 >>> inputs = torch.arange(9).view(1, 1, 3, 3).float() + 2 >>> input_dims = inputs.shape[2:] >>> #output_dims = (6, 6) >>> def fmt(a): >>> return ub.repr2(a.numpy(), precision=2) >>> s = 2.5 >>> output_dims = tuple(np.round((np.array(input_dims) * s)).astype(np.int).tolist()) >>> mat = torch.FloatTensor([[s, 0, 0], [0, s, 0], [0, 0, 1]]) >>> inv = mat.inverse() >>> warp_tensor(inputs, mat, output_dims) >>> print('## INPUTS') >>> print(fmt(inputs)) >>> print('nalign_corners=True') >>> print('---') >>> print('## warp_tensor, align_corners=True') >>> print(fmt(warp_tensor(inputs, inv, output_dims, isinv=True, align_corners=True))) >>> print('## interpolate, align_corners=True') >>> print(fmt(F.interpolate(inputs, output_dims, mode='bilinear', align_corners=True))) >>> print('nalign_corners=False') >>> print('---') >>> print('## warp_tensor, align_corners=False, new_mode=False') >>> print(fmt(warp_tensor(inputs, inv, output_dims, isinv=True, align_corners=False))) >>> print('## warp_tensor, align_corners=False, new_mode=True') >>> print(fmt(warp_tensor(inputs, inv, output_dims, isinv=True, align_corners=False, new_mode=True))) >>> print('## interpolate, align_corners=False') >>> print(fmt(F.interpolate(inputs, output_dims, mode='bilinear', align_corners=False))) >>> print('## interpolate (scale), align_corners=False') >>>

print(ub.repr2(F.interpolate(inputs, scale_factor=s, mode='bilinear', align_corners=False).numpy(), precision=2)) >>> cv2_M = mat.cpu().numpy()[0:2] >>> src = inputs[0, 0].cpu().numpy() >>> dsize = tuple(output_dims[::-1]) >>> print('nOpen CV warp Result') >>> result2 = (cv2.warpAffine(src, cv2_M, dsize=dsize, flags=cv2.INTER_LINEAR)) >>> print('result2 =n{}'.format(ub.repr2(result2, precision=2)))

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

k

Index

Symbols

A

B

C

D

E

F

G

H

I

K

M

N

O

P

R

S

T

U

V

W

X