kwcoco Documentation

Release 0.4.2

Jon Crall

Sep 06, 2022

PACKAGE LAYOUT

If you are new, please see our getting started document: getting_started

Please also see information in the repo README, which contains similar but complementary information.

For notes about warping and spaces see warping_and_spaces. The Kitware COCO module defines a variant of the Microsoft COCO format, originally developed for the "collected images in context" object detection challenge. We are backwards compatible with the original module, but we also have improved implementations in several places, including segmentations, keypoints, annotation tracks, multi-spectral images, and videos (which represents a generic sequence of images).

A kwcoco file is a "manifest" that serves as a single reference that points to all images, categories, and annotations in a computer vision dataset. Thus, when applying an algorithm to a dataset, it is sufficient to have the algorithm take one dataset parameter: the path to the kwcoco file. Generally a kwcoco file will live in a "bundle" directory along with the data that it references, and paths in the kwcoco file will be relative to the location of the kwcoco file itself.

The main data structure in this model is largely based on the implementation in https://github.com/cocodataset/cocoapi It uses the same efficient core indexing data structures, but in our implementation the indexing can be optionally turned off, functions are silent by default (with the exception of long running processes, which optionally show progress by default). We support helper functions that add and remove images, categories, and annotations.

The *kwcoco.CocoDataset* class is capable of dynamic addition and removal of categories, images, and annotations. Has better support for keypoints and segmentation formats than the original COCO format. Despite being written in Python, this data structure is reasonably efficient.

```
>>> import kwcoco
>>> import json
>>> # Create demo data
>>> demo = kwcoco.CocoDataset.demo()
>>> # Reroot can switch between absolute / relative-paths
>>> demo.reroot(absolute=True)
>>> # could also use demo.dump / demo.dumps, but this is more explicit
>>> text = json.dumps(demo.dataset)
>>> with open('demo.json', 'w') as file:
>>>     file.write(text)

>>> # Read from disk
>>> self = kwcoco.CocoDataset('demo.json')

>>> # Add data
>>> cid = self.add_category('Cat')
>>> gid = self.add_image('new-img.jpg')
>>> aid = self.add_annotation(image_id=gid, category_id=cid, bbox=[0, 0, 100, 100])

>>> # Remove data
>>> self.remove_annotations([aid])
>>> self.remove_images([gid])
>>> self.remove_categories([cid])

>>> # Look at data
>>> import ubelt as ub
>>> print(ub.repr2(self.basic_stats(), nl=1))
>>> print(ub.repr2(self.extended_stats(), nl=2))
>>> print(ub.repr2(self.boxsize_stats(), nl=3))
>>> print(ub.repr2(self.category_annotation_frequency()))
```

(continues on next page)

(continued from previous page)

```
>>> # Inspect data
>>> # xdoctest: +REQUIRES(module:kwplot)
>>> import kwplot
>>> kwplot.autompl()
>>> self.show_image(gid=1)

>>> # Access single-item data via imgs, cats, anns
>>> cid = 1
>>> self.cats[cid]
{'id': 1, 'name': 'astronaut', 'supercategory': 'human'}

>>> gid = 1
>>> self.imgs[gid]
{'id': 1, 'file_name': '...astro.png', 'url': 'https://i.imgur.com/KXhKM72.png'}

>>> aid = 3
>>> self.anns[aid]
{'id': 3, 'image_id': 1, 'category_id': 3, 'line': [326, 369, 500, 500]}

>>> # Access multi-item data via the annots and images helper objects
>>> aids = self.index.gid_to_aids[2]
>>> annots = self.annots(aids)

>>> print('annots = {}'.format(ub.repr2(annots, nl=1, sv=1)))
annots = <Annots(num=2)>

>>> annots.lookup('category_id')
[6, 4]

>>> annots.lookup('bbox')
[[37, 6, 230, 240], [124, 96, 45, 18]]

>>> # built in conversions to efficient kwimage array DataStructures
>>> print(ub.repr2(annots.detections.data, sv=1))
{
    'boxes': <Boxes(xywh,
                array([[ 37.,   6., 230., 240.],
                       [124.,  96.,  45.,  18.]], dtype=float32))>,
    'class_idxs': [5, 3],
    'keypoints': <PointsList(n=2)>,
    'segmentations': <PolygonList(n=2)>,
}

>>> gids = list(self.imgs.keys())
>>> images = self.images(gids)
>>> print('images = {}'.format(ub.repr2(images, nl=1, sv=1)))
images = <Images(num=3)>

>>> images.lookup('file_name')
['...astro.png', '...carl.png', '...stars.png']

>>> print('images.annots = {}'.format(images.annots))
```

(continues on next page)

(continued from previous page)

```
images.annots = <AnnotGroups(n=3, m=3.7, s=3.9)>

>>> print('images.annots.cids = {!r}'.format(images.annots.cids))
images.annots.cids = [[1, 2, 3, 4, 5, 5, 5, 5, 5], [6, 4], []]
```

CHAPTER

ONE

COCODATASET API

The following is a logical grouping of the public kwcoco.CocoDataset API attributes and methods. See the in-code documentation for further details.

1.1 CocoDataset classmethods (via MixinCocoExtras)

- *kwcoco.CocoDataset.coerce* - Attempt to transform the input into the intended CocoDataset.
- *kwcoco.CocoDataset.demo* - Create a toy coco dataset for testing and demo puposes
- *kwcoco.CocoDataset.random* - Creates a random CocoDataset according to distribution parameters

1.2 CocoDataset classmethods (via CocoDataset)

- *kwcoco.CocoDataset.from_coco_paths* - Constructor from multiple coco file paths.
- *kwcoco.CocoDataset.from_data* - Constructor from a json dictionary
- *kwcoco.CocoDataset.from_image_paths* - Constructor from a list of images paths.

1.3 CocoDataset slots

- `kwcoco.CocoDataset.index` - an efficient lookup index into the coco data structure. The index defines its own attributes like `anns`, `cats`, `imgs`, `gid_to_aids`, `file_name_to_img`, etc. See `CocoIndex` for more details on which attributes are available.
- `kwcoco.CocoDataset.hashid` - If computed, this will be a hash uniquely identifing the dataset. To ensure this is computed see `kwcoco.coco_dataset.MixinCocoExtras._build_hashid()`.
- `kwcoco.CocoDataset.hashid_parts` -
- `kwcoco.CocoDataset.tag` - A tag indicating the name of the dataset.
- `kwcoco.CocoDataset.dataset` - raw json data structure. This is the base dictionary that contains {'annotations': List, 'images': List, 'categories': List}
- `kwcoco.CocoDataset.bundle_dpath` - If known, this is the root path that all image file names are relative to. This can also be manually overwritten by the user.
- `kwcoco.CocoDataset.assets_dpath` -
- `kwcoco.CocoDataset.cache_dpath` -

1.4 CocoDataset properties

- kwcoco.CocoDataset.anns -
- kwcoco.CocoDataset.cats -
- kwcoco.CocoDataset.cid_to_aids -
- kwcoco.CocoDataset.data_fpath -
- kwcoco.CocoDataset.data_root -
- *kwcoco.CocoDataset.fpath* - if known, this stores the filepath the dataset was loaded from
- kwcoco.CocoDataset.gid_to_aids -
- kwcoco.CocoDataset.img_root -
- kwcoco.CocoDataset.imgs -
- kwcoco.CocoDataset.n_annots -
- kwcoco.CocoDataset.n_cats -
- kwcoco.CocoDataset.n_images -
- kwcoco.CocoDataset.n_videos -
- kwcoco.CocoDataset.name_to_cat -

1.5 CocoDataset methods (via MixinCocoAddRemove)

- *kwcoco.CocoDataset.add_annotation* - Add an annotation to the dataset (dynamically updates the index)
- *kwcoco.CocoDataset.add_annotations* - Faster less-safe multi-item alternative to add_annotation.
- *kwcoco.CocoDataset.add_category* - Adds a category
- *kwcoco.CocoDataset.add_image* - Add an image to the dataset (dynamically updates the index)
- *kwcoco.CocoDataset.add_images* - Faster less-safe multi-item alternative
- *kwcoco.CocoDataset.add_video* - Add a video to the dataset (dynamically updates the index)
- *kwcoco.CocoDataset.clear_annotations* - Removes all annotations (but not images and categories)
- *kwcoco.CocoDataset.clear_images* - Removes all images and annotations (but not categories)
- *kwcoco.CocoDataset.ensure_category* - Like `add_category()`, but returns the existing category id if it already exists instead of failing. In this case all metadata is ignored.
- *kwcoco.CocoDataset.ensure_image* - Like `add_image()`,, but returns the existing image id if it already exists instead of failing. In this case all metadata is ignored.
- *kwcoco.CocoDataset.remove_annotation* - Remove a single annotation from the dataset
- *kwcoco.CocoDataset.remove_annotation_keypoints* - Removes all keypoints with a particular category
- *kwcoco.CocoDataset.remove_annotations* - Remove multiple annotations from the dataset.
- *kwcoco.CocoDataset.remove_categories* - Remove categories and all annotations in those categories. Currently does not change any hierarchy information
- *kwcoco.CocoDataset.remove_images* - Remove images and any annotations contained by them

- *kwcoco.CocoDataset.remove_keypoint_categories* - Removes all keypoints of a particular category as well as all annotation keypoints with those ids.
- *kwcoco.CocoDataset.remove_videos* - Remove videos and any images / annotations contained by them
- *kwcoco.CocoDataset.set_annotation_category* - Sets the category of a single annotation

1.6 CocoDataset methods (via MixinCocoObjects)

- *kwcoco.CocoDataset.annots* - Return vectorized annotation objects
- *kwcoco.CocoDataset.categories* - Return vectorized category objects
- *kwcoco.CocoDataset.images* - Return vectorized image objects
- *kwcoco.CocoDataset.videos* - Return vectorized video objects

1.7 CocoDataset methods (via MixinCocoStats)

- *kwcoco.CocoDataset.basic_stats* - Reports number of images, annotations, and categories.
- *kwcoco.CocoDataset.boxsize_stats* - Compute statistics about bounding box sizes.
- *kwcoco.CocoDataset.category_annotation_frequency* - Reports the number of annotations of each category
- *kwcoco.CocoDataset.category_annotation_type_frequency* - Reports the number of annotations of each type for each category
- *kwcoco.CocoDataset.conform* - Make the COCO file conform a stricter spec, infers attibutes where possible.
- *kwcoco.CocoDataset.extended_stats* - Reports number of images, annotations, and categories.
- *kwcoco.CocoDataset.find_representative_images* - Find images that have a wide array of categories. Attempt to find the fewest images that cover all categories using images that contain both a large and small number of annotations.
- *kwcoco.CocoDataset.keypoint_annotation_frequency* -
- *kwcoco.CocoDataset.stats* - This function corresponds to *kwcoco.cli.coco_stats*.
- *kwcoco.CocoDataset.validate* - Performs checks on this coco dataset.

1.8 CocoDataset methods (via MixinCocoAccessors)

- *kwcoco.CocoDataset.category_graph* - Construct a networkx category hierarchy
- *kwcoco.CocoDataset.delayed_load* - Experimental method
- *kwcoco.CocoDataset.get_auxiliary_fpath* - Returns the full path to auxiliary data for an image
- *kwcoco.CocoDataset.get_image_fpath* - Returns the full path to the image
- *kwcoco.CocoDataset.keypoint_categories* - Construct a consistent CategoryTree representation of keypoint classes
- *kwcoco.CocoDataset.load_annot_sample* - Reads the chip of an annotation. Note this is much less efficient than using a sampler, but it doesn't require disk cache.

- *kwcoco.CocoDataset.load_image* - Reads an image from disk and
- *kwcoco.CocoDataset.object_categories* - Construct a consistent CategoryTree representation of object classes

1.9 CocoDataset methods (via CocoDataset)

- *kwcoco.CocoDataset.copy* - Deep copies this object
- *kwcoco.CocoDataset.dump* - Writes the dataset out to the json format
- *kwcoco.CocoDataset.dumps* - Writes the dataset out to the json format
- *kwcoco.CocoDataset.subset* - Return a subset of the larger coco dataset by specifying which images to port. All annotations in those images will be taken.
- *kwcoco.CocoDataset.union* - Merges multiple `CocoDataset` items into one. Names and associations are retained, but ids may be different.
- *kwcoco.CocoDataset.view_sql* - Create a cached SQL interface to this dataset suitable for large scale multiprocessing use cases.

1.10 CocoDataset methods (via MixinCocoExtras)

- *kwcoco.CocoDataset.corrupted_images* - Check for images that don't exist or can't be opened
- *kwcoco.CocoDataset.missing_images* - Check for images that don't exist
- *kwcoco.CocoDataset.rename_categories* - Rename categories with a potentially coarser categorization.
- *kwcoco.CocoDataset.reroot* - Rebase image/data paths onto a new image/data root.

1.11 CocoDataset methods (via MixinCocoDraw)

- *kwcoco.CocoDataset.draw_image* - Use kwimage to draw all annotations on an image and return the pixels as a numpy array.
- *kwcoco.CocoDataset.imread* - Loads a particular image
- *kwcoco.CocoDataset.show_image* - Use matplotlib to show an image with annotations overlaid

CHAPTER

TWO

KWCOCO

2.1 kwcoco package

2.1.1 Subpackages

2.1.1.1 kwcoco.cli package

2.1.1.1.1 Submodules

2.1.1.1.1.1 kwcoco.cli.coco_conform module

class kwcoco.cli.coco_conform.**CocoConformCLI**

Bases: object

name = 'conform'

class **CLIConfig**(*data=None*, *default=None*, *cmdline=False*)

Bases: Config

Make the COCO file conform to the spec.

Populates inferable information such as image size, annotation area, etc.

epilog = '\n Example Usage:\n kwcoco conform --help\n kwcoco conform --src=special:shapes8 --dst conformed.json\n '

default = {'dst': <Value(None: None)>, 'ensure_imgsize': <Value(None: True)>, 'legacy': <Value(None: False)>, 'pycocotools_info': <Value(None: True)>, 'src': <Value(None: None)>, 'workers': <Value(None: 8)>}

classmethod **main**(*cmdline=True*, ***kw*)

Example

```
>>> # xdoctest: +SKIP
>>> kw = {'src': 'special:shapes8'}
>>> cmdline = False
>>> cls = CocoConformCLI
>>> cls.main(cmdline, **kw)
```

2.1.1.1.1.2 kwcoco.cli.coco_eval module

Wraps the logic in kwcoco/coco_evaluator.py with a command line script

class kwcoco.cli.coco_eval.**CocoEvalCLIConfig**(*data=None*, *default=None*, *cmdline=False*)

Bases: Config

Evaluate and score predicted versus truth detections / classifications in a COCO dataset

default = {'ap_method': <Value(None: 'pycocotools')>, 'area_range': <Value(None: ['all'])>, 'assign_workers': <Value(None: 8)>, 'classes_of_interest': <Value(<class 'list'>: None)>, 'compat': <Value(None: 'mutex')>, 'draw': <Value(None: True)>, 'expt_title': <Value(<class 'str'>: '')>, 'force_pycocoutils': <Value(None: False)>, 'fp_cutoff': <Value(None: inf)>, 'ignore_classes': <Value(<class 'list'>: None)>, 'implicit_ignore_classes': <Value(None: ['ignore'])>, 'implicit_negative_classes': <Value(None: ['background'])>, 'iou_bias': <Value(None: 1)>, 'iou_thresh': <Value(None: 0.5)>, 'load_workers': <Value(None: 0)>, 'max_dets': <Value(None: inf)>, 'monotonic_ppv': <Value(None: True)>, 'out_dpath': <Value(<class 'str'>: './coco_metrics')>, 'ovthresh': <Value(None: None)>, 'pred_dataset': <Value(<class 'str'>: None)>, 'true_dataset': <Value(<class 'str'>: None)>, 'use_area_attr': <Value(None: 'try')>, 'use_image_names': <Value(None: False)>}

class kwcoco.cli.coco_eval.**CocoEvalCLI**

Bases: object

name = 'eval'

CLIConfig

alias of *CocoEvalCLIConfig*

classmethod main(*cmdline=True*, ***kw*)

Example

```
>>> # xdoctest: +REQUIRES(module:kwplot)
>>> from kwcoco.cli.coco_eval import *  # NOQA
>>> import ubelt as ub
>>> from kwcoco.cli.coco_eval import *  # NOQA
>>> from os.path import join
>>> import kwcoco
>>> dpath = ub.ensure_app_cache_dir('kwcoco/tests/eval')
>>> true_dset = kwcoco.CocoDataset.demo('shapes8')
>>> from kwcoco.demo.perterb import perterb_coco
```

(continues on next page)

(continued from previous page)

```
>>> kwargs = {
>>>     'box_noise': 0.5,
>>>     'n_fp': (0, 10),
>>>     'n_fn': (0, 10),
>>> }
>>> pred_dset = perterb_coco(true_dset, **kwargs)
>>> true_dset.fpath = join(dpath, 'true.mscoco.json')
>>> pred_dset.fpath = join(dpath, 'pred.mscoco.json')
>>> true_dset.dump(true_dset.fpath)
>>> pred_dset.dump(pred_dset.fpath)
>>> draw = False  # set to false for faster tests
>>> CocoEvalCLI.main(
>>>     true_dataset=true_dset.fpath,
>>>     pred_dataset=pred_dset.fpath,
>>>     draw=draw, out_dpath=dpath)
```

kwcoco.cli.coco_eval.**main**(*cmdline=True*, ***kw*)

Todo:

- [X] should live in kwcoco.cli.coco_eval

CommandLine

```
# Generate test data
xdoctest -m kwcoco.cli.coco_eval CocoEvalCLI.main

kwcoco eval \
    --true_dataset=$HOME/.cache/kwcoco/tests/eval/true.mscoco.json \
    --pred_dataset=$HOME/.cache/kwcoco/tests/eval/pred.mscoco.json \
    --out_dpath=$HOME/.cache/kwcoco/tests/eval/out \
    --force_pycocoutils=False \
    --area_range=all,0-4096,4096-inf

nautilus $HOME/.cache/kwcoco/tests/eval/out
```

2.1.1.1.1.3 kwcoco.cli.coco_grab module

class kwcoco.cli.coco_grab.**CocoGrabCLI**

Bases: object

name = 'grab'

class CLIConfig(*data=None*, *default=None*, *cmdline=False*)

Bases: Config

Grab standard datasets.

Example

kwcoco grab cifar10 camvid

```
default = {'dpath':  <Path(<class 'str'>:
Path('/home/docs/.cache/kwcoco/data'))>, 'names':  <Value(None:  [])>}
```

classmethod main(*cmdline=True*, ***kw*)

2.1.1.1.1.4 kwcoco.cli.coco_modify_categories module

class kwcoco.cli.coco_modify_categories.**CocoModifyCatsCLI**

Bases: object

Remove, rename, or coarsen categories.

name = 'modify_categories'

class CLIConfig(*data=None*, *default=None*, *cmdline=False*)

Bases: Config

Rename or remove categories

```
epilog = '\n Example Usage:\n kwcoco modify_categories --help\n kwcoco
modify_categories --src=special:shapes8 --dst modcats.json\n kwcoco
modify_categories --src=special:shapes8 --dst modcats.json --rename
eff:F,star:sun\n kwcoco modify_categories --src=special:shapes8 --dst
modcats.json --remove eff,star\n kwcoco modify_categories --src=special:shapes8
--dst modcats.json --keep eff,\n\n kwcoco modify_categories
--src=special:shapes8 --dst modcats.json --keep=[] --keep_annots=True\n '

default = {'dst':  <Value(None:  None)>, 'keep':  <Value(None:  None)>,
'keep_annots':  <Value(None:  False)>, 'remove':  <Value(None:  None)>,
'rename':  <Value(<class 'str'>:  None)>, 'src':  <Value(None:  None)>}
```

classmethod main(*cmdline=True*, ***kw*)

Example

```
>>> # xdoctest: +SKIP
>>> kw = {'src': 'special:shapes8'}
>>> cmdline = False
>>> cls = CocoModifyCatsCLI
>>> cls.main(cmdline, **kw)
```

2.1.1.1.1.5 kwcoco.cli.coco_reroot module

class kwcoco.cli.coco_reroot.**CocoRerootCLI**

Bases: object

name = 'reroot'

class **CLIConfig**(*data=None*, *default=None*, *cmdline=False*)

Bases: Config

Reroot image paths onto a new image root.

Modify the root of a coco dataset such to either make paths relative to a new root or make paths absolute.

Todo:

- [ ] Evaluate that all tests cases work

---

epilog = '\n\n Example Usage:\n kwcoco reroot --help\n kwcoco reroot --src=special:shapes8 --dst rerooted.json\n kwcoco reroot --src=special:shapes8 --new_prefix=foo --check=True --dst rerooted.json\n '

default = {'absolute': <Value(None: True)>, 'check': <Value(None: True)>, 'dst': <Value(None: None)>, 'new_prefix': <Value(None: None)>, 'old_prefix': <Value(None: None)>, 'src': <Value(None: None)>}

classmethod **main**(*cmdline=True*, ***kw*)

Example

```
>>> # xdoctest: +SKIP
>>> kw = {'src': 'special:shapes8'}
>>> cmdline = False
>>> cls = CocoRerootCLI
>>> cls.main(cmdline, **kw)
```

2.1.1.1.1.6 kwcoco.cli.coco_show module

class kwcoco.cli.coco_show.**CocoShowCLI**

Bases: object

name = 'show'

class **CLIConfig**(*data=None*, *default=None*, *cmdline=False*)

Bases: Config

Visualize a COCO image using matplotlib or opencv, optionally writing it to disk

epilog = '\n Example Usage:\n kwcoco show --help\n kwcoco show --src=special:shapes8 --gid=1\n kwcoco show --src=special:shapes8 --gid=1 --dst out.png\n '

default = {'aid': <Value(None: None)>, 'channels': <Value(<class 'str'>: None)>, 'dst': <Value(None: None)>, 'gid': <Value(None: None)>, 'mode': <Value(None: 'matplotlib')>, 'show_annots': <Value(None: True)>, 'show_labels': <Value(None: False)>, 'src': <Value(None: None)>}

classmethod main(*cmdline=True*, ***kw*)

Todo:

- [ ] Visualize auxiliary data

Example

```
>>> # xdoctest: +SKIP
>>> kw = {'src': 'special:shapes8'}
>>> cmdline = False
>>> cls = CocoShowCLI
>>> cls.main(cmdline, **kw)
```

2.1.1.1.1.7 kwcoco.cli.coco_split module

class kwcoco.cli.coco_split.CocoSplitCLI

Bases: object

name = 'split'

class CLIConfig(*data=None*, *default=None*, *cmdline=False*)

Bases: Config

Split a single COCO dataset into two sub-datasets.

default = {'dst1': <Value(None: 'split1.mscoco.json')>, 'dst2': <Value(None: 'split2.mscoco.json')>, 'factor': <Value(None: 3)>, 'rng': <Value(None: None)>, 'src': <Value(None: None)>}

epilog = '\n Example Usage:\n kwcoco split --src special:shapes8 --dst1=learn.mscoco.json --dst2=test.mscoco.json --factor=3 --rng=42\n '

classmethod main(*cmdline=True*, ***kw*)

Example

```
>>> from kwcoco.cli.coco_split import *  # NOQA
>>> import ubelt as ub
>>> dpath = ub.Path.appdir('kwcoco/tests/cli/split').ensuredir()
>>> kw = {'src': 'special:shapes8',
>>>       'dst1': dpath / 'train.json',
>>>       'dst2': dpath / 'test.json'}
>>> cmdline = False
>>> cls = CocoSplitCLI
>>> cls.main(cmdline, **kw)
```

2.1.1.1.1.8 kwcoco.cli.coco_stats module

class kwcoco.cli.coco_stats.**CocoStatsCLI**

Bases: object

name = 'stats'

class **CLIConfig**(*data=None*, *default=None*, *cmdline=False*)

Bases: Config

Compute summary statistics about a COCO dataset

default = {'annot_attrs': <Value(None: False)>, 'basic': <Value(None: True)>, 'boxes': <Value(None: False)>, 'catfreq': <Value(None: True)>, 'embed': <Value(None: False)>, 'extended': <Value(None: True)>, 'image_attrs': <Value(None: False)>, 'image_size': <Value(None: False)>, 'src': <Value(None: ['special:shapes8'])>, 'video_attrs': <Value(None: False)>}

epilog = '\n Example Usage:\n kwcoco stats --src=special:shapes8\n kwcoco stats --src=special:shapes8 --boxes=True\n '

classmethod **main**(*cmdline=True*, ***kw*)

Example

```
>>> kw = {'src': 'special:shapes8'}
>>> cmdline = False
>>> cls = CocoStatsCLI
>>> cls.main(cmdline, **kw)
```

2.1.1.1.1.9 kwcoco.cli.coco_subset module

class kwcoco.cli.coco_subset.**CocoSubsetCLI**

Bases: object

name = 'subset'

class **CLIConfig**(*data=None*, *default=None*, *cmdline=False*)

Bases: Config

Take a subset of this dataset and write it to a new file

default = {'absolute': <Value(None: 'auto')>, 'channels': <Value(None: None)>, 'copy_assets': <Value(None: False)>, 'dst': <Value(None: None)>, 'gids': <Value(None: None)>, 'include_categories': <Value(<class 'str'>: None)>, 'select_images': <Value(<class 'str'>: None)>, 'select_videos': <Value(None: None)>, 'src': <Value(None: None)>}

epilog = '\n Example Usage:\n kwcoco subset --src special:shapes8 --dst=foo.kwcoco.json\n\n # Take only the even image-ids\n kwcoco subset --src special:shapes8 --dst=foo-even.kwcoco.json --select_images \'.id % 2 == 0\'\n\n # Take only the videos where the name ends with 2\n kwcoco subset --src special:vidshapes8 --dst=vidsub.kwcoco.json --select_videos \'.name | endswith("2")\'\n '

classmethod main(*cmdline=True*, ***kw*)

Example

```
>>> from kwcoco.cli.coco_subset import *  # NOQA
>>> import ubelt as ub
>>> dpath = ub.Path.appdir('kwcoco/tests/cli/union').ensuredir()
>>> kw = {'src': 'special:shapes8',
>>>       'dst': dpath / 'subset.json',
>>>       'include_categories': 'superstar'}
>>> cmdline = False
>>> cls = CocoSubsetCLI
>>> cls.main(cmdline, **kw)
```

kwcoco.cli.coco_subset.**query_subset**(*dset*, *config*)

Example

```
>>> # xdoctest: +REQUIRES(module:jq)
>>> from kwcoco.cli.coco_subset import *  # NOQA
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo()
>>> assert dset.n_images == 3
>>> #
>>> config = CocoSubsetCLI.CLIConfig({'select_images': '.id < 3'})
>>> new_dset = query_subset(dset, config)
>>> assert new_dset.n_images == 2
>>> #
>>> config = CocoSubsetCLI.CLIConfig({'select_images': '.file_name | test(".*.png")
↪'})
>>> new_dset = query_subset(dset, config)
>>> assert all(n.endswith('.png') for n in new_dset.images().lookup('file_name'))
>>> assert new_dset.n_images == 2
>>> #
>>> config = CocoSubsetCLI.CLIConfig({'select_images': '.file_name | test(".*.png")␣
↪| not'})
>>> new_dset = query_subset(dset, config)
>>> assert not any(n.endswith('.png') for n in new_dset.images().lookup('file_name
↪'))
>>> assert new_dset.n_images == 1
>>> #
>>> config = CocoSubsetCLI.CLIConfig({'select_images': '.id < 3 and (.file_name |␣
↪test(".*.png"))'})
>>> new_dset = query_subset(dset, config)
>>> assert new_dset.n_images == 1
>>> #
>>> config = CocoSubsetCLI.CLIConfig({'select_images': '.id < 3 or (.file_name |␣
↪test(".*.png"))'})
>>> new_dset = query_subset(dset, config)
>>> assert new_dset.n_images == 3
```

Example

```
>>> # xdoctest: +REQUIRES(module:jq)
>>> from kwcoco.cli.coco_subset import *  # NOQA
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('vidshapes8')
>>> assert dset.n_videos == 8
>>> assert dset.n_images == 16
>>> config = CocoSubsetCLI.CLIConfig({'select_videos': '.name == "toy_video_3"'})
>>> new_dset = query_subset(dset, config)
>>> assert new_dset.n_images == 2
>>> assert new_dset.n_videos == 1
```

2.1.1.1.1.10 kwcoco.cli.coco_toydata module

class kwcoco.cli.coco_toydata.**CocoToyDataCLI**

Bases: object

name = 'toydata'

class CLIConfig(*data=None*, *default=None*, *cmdline=False*)

Bases: Config

Create COCO toydata for demo and testing purposes.

default = {'bundle_dpath': <Value(None: None)>, 'dst': <Value(None: None)>, 'key': <Value(None: 'shapes8')>, 'use_cache': <Value(None: True)>, 'verbose': <Value(None: False)>}

epilog = '\n Example Usage:\n kwcoco toydata --key=shapes8 --dst=toydata.kwcoco.json\n\n kwcoco toydata --key=shapes8 --bundle_dpath=my_test_bundle_v1\n kwcoco toydata --key=shapes8 --bundle_dpath=my_test_bundle_v1\n\n kwcoco toydata \\\n --key=vidshapes1-frames32 \\\n --dst=./mytoybundle/dataset.kwcoco.json\n\n TODO:\n - [ ] allow specification of images directory\n '

classmethod main(*cmdline=True*, ***kw*)

Example

```
>>> from kwcoco.cli.coco_toydata import *  # NOQA
>>> import ubelt as ub
>>> dpath = ub.Path.appdir('kwcoco/tests/cli/demo').ensuredir()
>>> kw = {'key': 'shapes8', 'dst': dpath / 'test.json'}
>>> cmdline = False
>>> cls = CocoToyDataCLI
>>> cls.main(cmdline, **kw)
```

2.1.1.1.1.11 kwcoco.cli.coco_union module

class kwcoco.cli.coco_union.**CocoUnionCLI**

Bases: object

name = 'union'

class CLIConfig(*data=None*, *default=None*, *cmdline=False*)

Bases: Config

Combine multiple COCO datasets into a single merged dataset.

default = {'absolute': <Value(None: False)>, 'dst': <Value(None: 'combo.kwcoco.json')>, 'src': <Value(None: [])>}

epilog = '\n Example Usage:\n kwcoco union --src special:shapes8 special:shapes1 --dst=combo.kwcoco.json\n '

classmethod main(*cmdline=True*, ***kw*)

Example

```
>>> from kwcoco.cli.coco_union import *  # NOQA
>>> import ubelt as ub
>>> dpath = ub.Path.appdir('kwcoco/tests/cli/union').ensuredir()
>>> dst_fpath = dpath / 'combo.kwcoco.json'
>>> kw = {
>>>     'src': ['special:shapes8', 'special:shapes1'],
>>>     'dst': dst_fpath
>>> }
>>> cmdline = False
>>> cls = CocoUnionCLI
>>> cls.main(cmdline, **kw)
```

2.1.1.1.1.12 kwcoco.cli.coco_validate module

class kwcoco.cli.coco_validate.**CocoValidateCLI**

Bases: object

name = 'validate'

class CLIConfig(*data=None*, *default=None*, *cmdline=False*)

Bases: Config

Validate that a coco file conforms to the json schema, that assets exist, and potentially fix corrupted assets by removing them.

default = {'channels': <Value(None: True)>, 'corrupted': <Value(None: False)>, 'dst': <Value(None: None)>, 'fastfail': <Value(None: False)>, 'fix': <Value(None: None)>, 'img_attrs': <Value(None: 'warn')>, 'missing': <Value(None: True)>, 'require_relative': <Value(None: False)>, 'schema': <Value(None: True)>, 'src': <Value(None: ['special:shapes8'])>, 'unique': <Value(None: True)>, 'verbose': <Value(None: 1)>}

```
epilog = '\n Example Usage:\n kwcoco toydata --dst foo.json
--key=special:shapes8\n kwcoco validate --src=foo.json --corrupted=True\n '
```

classmethod main(*cmdline=True*, ***kw*)

Example

```
>>> from kwcoco.cli.coco_validate import *  # NOQA
>>> kw = {'src': 'special:shapes8'}
>>> cmdline = False
>>> cls = CocoValidateCLI
>>> cls.main(cmdline, **kw)
```

2.1.1.1.2 Module contents

2.1.1.2 kwcoco.data package

2.1.1.2.1 Submodules

2.1.1.2.1.1 kwcoco.data.grab_camvid module

Downloads the CamVid data if necessary, and converts it to COCO.

kwcoco.data.grab_camvid.**grab_camvid_train_test_val_splits**(*coco_dset*, *mode='segnet'*)

kwcoco.data.grab_camvid.**grab_camvid_sampler**()

Grab a kwcoco.CocoSampler object for the CamVid dataset.

Returns
sampler

Return type
kwcoco.CocoSampler

Example

```
>>> # xdoctest: +REQUIRES(--download)
>>> sampler = grab_camvid_sampler()
>>> print('sampler = {!r}'.format(sampler))
>>> # sampler.load_sample()
>>> for gid in ub.ProgIter(sampler.image_ids, desc='load image'):
>>>     img = sampler.load_image(gid)
```

kwcoco.data.grab_camvid.**grab_coco_camvid**()

Example

```
>>> # xdoctest: +REQUIRES(--download)
>>> dset = grab_coco_camvid()
>>> print('dset = {!r}'.format(dset))
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> plt = kwplot.autoplt()
>>> plt.clf()
>>> dset.show_image(gid=1)
```

kwcoco.data.grab_camvid.**grab_raw_camvid**()

Grab the raw camvid data.

kwcoco.data.grab_camvid.**rgb_to_cid**(*r*, *g*, *b*)

kwcoco.data.grab_camvid.**cid_to_rgb**(*cid*)

kwcoco.data.grab_camvid.**convert_camvid_raw_to_coco**(*camvid_raw_info*)

Converts the raw camvid format to an MSCOCO based format, (which lets use use kwcoco's COCO backend).

Example

```
>>> # xdoctest: +REQUIRES(--download)
>>> camvid_raw_info = grab_raw_camvid()
>>> # test with a reduced set of data
>>> del camvid_raw_info['img_paths'][2:]
>>> del camvid_raw_info['mask_paths'][2:]
>>> dset = convert_camvid_raw_to_coco(camvid_raw_info)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> plt = kwplot.autoplt()
>>> kwplot.figure(fnum=1, pnum=(1, 2, 1))
>>> dset.show_image(gid=1)
>>> kwplot.figure(fnum=1, pnum=(1, 2, 2))
>>> dset.show_image(gid=2)
```

kwcoco.data.grab_camvid.**main**()

Dump the paths to the coco file to stdout

By default these will go to in the path:

~/.cache/kwcoco/camvid/camvid-master

The four files will be:

~/.cache/kwcoco/camvid/camvid-master/camvid-full.mscoco.json ~/.cache/kwcoco/camvid/camvid-master/camvid-train.mscoco.json ~/.cache/kwcoco/camvid/camvid-master/camvid-vali.mscoco.json ~/.cache/kwcoco/camvid/camvid-master/camvid-test.mscoco.json

2.1.1.2.1.2 kwcoco.data.grab_cifar module

2.1.1.2.1.3 kwcoco.data.grab_datasets module

Todo:

- [ ] UCF101 - Action Recognition Data Set - https://www.crcv.ucf.edu/data/UCF101.php
- [ ] HMDB: a large human motion database - https://serre-lab.clps.brown.edu/resource/hmdb-a-large-human-motion-database/
- [ ] https://paperswithcode.com/dataset/imagenet
- [ ] https://paperswithcode.com/dataset/coco
- [ ] https://paperswithcode.com/dataset/fashion-mnist
- [ ] https://paperswithcode.com/dataset/visual-question-answering
- [ ] https://paperswithcode.com/dataset/lfw
- [ ] https://paperswithcode.com/dataset/lsun
- [ ] https://paperswithcode.com/dataset/ava
- [ ] https://paperswithcode.com/dataset/activitynet
- [ ] https://paperswithcode.com/dataset/clevr

2.1.1.2.1.4 kwcoco.data.grab_domainnet module

References

http://ai.bu.edu/M3SDA/#dataset

kwcoco.data.grab_domainnet.**grab_domain_net**()

Todo:

- [ ] Allow the user to specify the download directory, generalize this

pattern across the data grab scripts.

2.1.1.2.1.5 kwcoco.data.grab_spacenet module

References

https://medium.com/the-downlinq/the-spacenet-7-multi-temporal-urban-development-challenge-algorithmic-baseline-4515ec9bd9fe https://arxiv.org/pdf/2102.11958.pdf https://spacenet.ai/sn7-challenge/

kwcoco.data.grab_spacenet.**grab_spacenet7**(*data_dpath*)

References

https://spacenet.ai/sn7-challenge/

Requires:
awscli

kwcoco.data.grab_spacenet.**convert_spacenet_to_kwcoco**(*extract_dpath*, *coco_fpath*)

Converts the raw SpaceNet7 dataset to kwcoco

Note:

- The "train" directory contains 60 "videos" representing a region over time.
- **Each "video" directory contains :**
  - images - unmasked images
  - images_masked - images with masks applied
  - labels - geojson polys in wgs84?
  - labels_match - geojson polys in wgs84 with track ids?
  - labels_match_pix - geojson polys in pixels with track ids?
  - UDM_masks - unusable data masks (binary data corresponding with an image, may not exist)

File names appear like:
"global_monthly_2018_01_mosaic_L15-1538E-1163N_6154_3539_13"

---

kwcoco.data.grab_spacenet.**main**()

2.1.1.2.1.6 kwcoco.data.grab_voc module

kwcoco.data.grab_voc.**convert_voc_to_coco**(*dpath=None*)

kwcoco.data.grab_voc.**ensure_voc_data**(*dpath=None*, *force=False*, *years=[2007, 2012]*)

Download the Pascal VOC data if it does not already exist.

Note:

- [ ] These URLS seem to be dead

Example

```
>>> # xdoctest: +REQUIRES(--download)
>>> devkit_dpath = ensure_voc_data()
```

kwcoco.data.grab_voc.**ensure_voc_coco**(*dpath=None*)

Download the Pascal VOC data and convert it to coco, if it does exit.

Parameters
dpath (*str*) – download directory. Defaults to "~/data/VOC".

Returns

mapping from dataset tags to coco file paths.
The original datasets have keys prefixed with underscores. The standard splits keys are train, vali, and test.

Return type
Dict[str, str]

kwcoco.data.grab_voc.**main**()

2.1.1.2.2 Module contents

2.1.1.3 kwcoco.demo package

2.1.1.3.1 Submodules

2.1.1.3.1.1 kwcoco.demo.boids module

class kwcoco.demo.boids.**Boids**(*num*, *dims=2*, *rng=None*, ***kwargs*)

Bases: NiceRepr

Efficient numpy based backend for generating boid positions.

BOID = bird-oid object

References

https://www.youtube.com/watch?v=mhjuuHl6qHM https://medium.com/better-programming/boids-simulating-birds-flock-behavior-in-python-9fff99375118 https://en.wikipedia.org/wiki/Boids

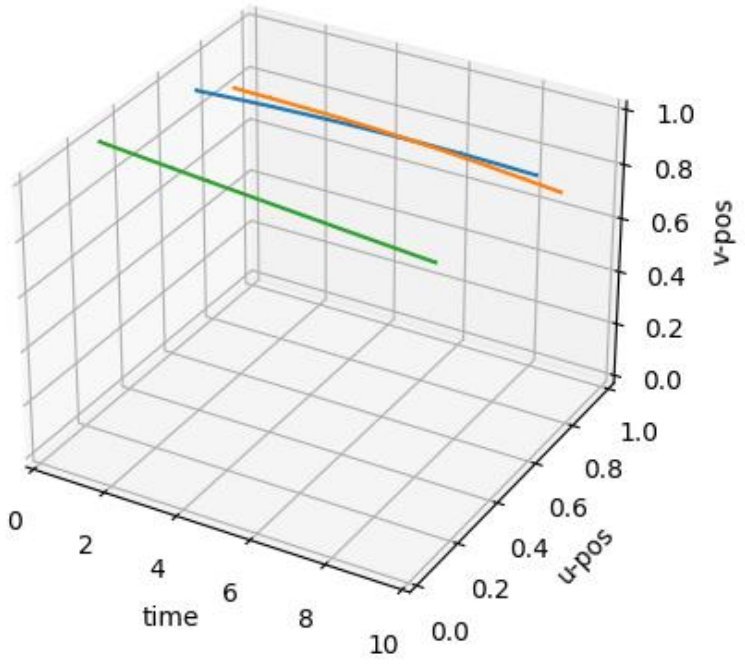
Example

```
>>> from kwcoco.demo.boids import *  # NOQA
>>> num_frames = 10
>>> num_objects = 3
>>> rng = None
>>> self = Boids(num=num_objects, rng=rng).initialize()
>>> paths = self.paths(num_frames)
>>> #
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> plt = kwplot.autoplt()
>>> from mpl_toolkits.mplot3d import Axes3D  # NOQA
>>> ax = plt.gca(projection='3d')
>>> ax.cla()
>>> #
>>> for path in paths:
>>>     time = np.arange(len(path))
>>>     ax.plot(time, path.T[0] * 1, path.T[1] * 1, ',-')
>>> ax.set_xlim(0, num_frames)
>>> ax.set_ylim(-.01, 1.01)
```

(continues on next page)

(continued from previous page)

```
>>> ax.set_zlim(-.01, 1.01)
>>> ax.set_xlabel('time')
>>> ax.set_ylabel('u-pos')
>>> ax.set_zlabel('v-pos')
>>> kwplot.show_if_requested()
```



import xdev _ = xdev.profile_now(self.compute_forces)() _ = xdev.profile_now(self.update_neighbors)()

Example

```
>>> # Test determenism
>>> from kwcoco.demo.boids import *  # NOQA
>>> num_frames = 2
>>> num_objects = 1
>>> rng = 4532
>>> self = Boids(num=num_objects, rng=rng).initialize()
>>> #print(ub.hash_data(self.pos))
>>> #print(ub.hash_data(self.vel))
>>> #print(ub.hash_data(self.acc))
>>> tocheck = []
>>> for i in range(100):
>>>     self = Boids(num=num_objects, rng=rng).initialize()
>>>     self.step()
```

(continues on next page)

(continued from previous page)

```
>>>     self.step()
>>>     self.step()
>>>     tocheck.append(self.pos.copy())
>>> assert ub.allsame(list(map(ub.hash_data, tocheck)))
```

initialize()

update_neighbors()

compute_forces()

boundary_conditions()

step()

Update positions, velocities, and accelerations

paths(*num_steps*)

kwcoco.demo.boids.**clamp_mag**(*vec*, *mag*, *axis=None*)

vec = np.random.rand(10, 2) mag = 1.0 axis = 1 new_vec = clamp_mag(vec, mag, axis) np.linalg.norm(new_vec, axis=axis)

kwcoco.demo.boids.**triu_condense_multi_index**(*multi_index*, *dims*, *symetric=False*)

Like np.ravel_multi_index but returns positions in an upper triangular condensed square matrix

Examples

multi_index (Tuple[ArrayLike]):
indexes for each dimension into the square matrix

dims (Tuple[int]):
shape of each dimension in the square matrix (should all be the same)

symetric (bool):
if True, converts lower triangular indices to their upper triangular location. This may cause a copy to occur.

References

https://stackoverflow.com/a/36867493/887074 https://numpy.org/doc/stable/reference/generated/numpy.ravel_multi_index.html#numpy.ravel_multi_index

Examples

```
>>> dims = (3, 3)
>>> symetric = True
>>> multi_index = (np.array([0, 0, 1]), np.array([1, 2, 2]))
>>> condensed_idxs = triu_condense_multi_index(multi_index, dims, symetric=symetric)
>>> assert condensed_idxs.tolist() == [0, 1, 2]
```

```
>>> n = 7
>>> symetric = True
>>> multi_index = np.triu_indices(n=n, k=1)
>>> condensed_idxs = triu_condense_multi_index(multi_index, [n] * 2,␣
↪symetric=symetric)
>>> assert condensed_idxs.tolist() == list(range(n * (n - 1) // 2))
>>> from scipy.spatial.distance import pdist, squareform
>>> square_mat = np.zeros((n, n))
>>> conden_mat = squareform(square_mat)
>>> conden_mat[condensed_idxs] = np.arange(len(condensed_idxs)) + 1
>>> square_mat = squareform(conden_mat)
>>> print('square_mat =\n{}'.format(ub.repr2(square_mat, nl=1)))
```

```
>>> n = 7
>>> symetric = True
>>> multi_index = np.tril_indices(n=n, k=-1)
>>> condensed_idxs = triu_condense_multi_index(multi_index, [n] * 2,␣
↪symetric=symetric)
>>> assert sorted(condensed_idxs.tolist()) == list(range(n * (n - 1) // 2))
>>> from scipy.spatial.distance import pdist, squareform
>>> square_mat = np.zeros((n, n))
>>> conden_mat = squareform(square_mat, checks=False)
>>> conden_mat[condensed_idxs] = np.arange(len(condensed_idxs)) + 1
>>> square_mat = squareform(conden_mat)
>>> print('square_mat =\n{}'.format(ub.repr2(square_mat, nl=1)))
```

kwcoco.demo.boids.**closest_point_on_line_segment**(*pts*, *e1*, *e2*)

Finds the closet point from p on line segment (e1, e2)

Parameters

- **pts** (*ndarray*) – xy points [Nx2]
- **e1** (*ndarray*) – the first xy endpoint of the segment
- **e2** (*ndarray*) – the second xy endpoint of the segment

Returns

pt_on_seg - the closest xy point on (e1, e2) from ptp

Return type

ndarray

References

http://en.wikipedia.org/wiki/Distance_from_a_point_to_a_line http://stackoverflow.com/questions/849211/shortest-distance-between-a-point-and-a-line-segment

Example

```
>>> # ENABLE_DOCTEST
>>> from kwcoco.demo.boids import *  # NOQA
>>> verts = np.array([[ 21.83012702,  13.16987298],
>>>                   [ 16.83012702,  21.83012702],
>>>                   [  8.16987298,  16.83012702],
>>>                   [ 13.16987298,   8.16987298],
>>>                   [ 21.83012702,  13.16987298]])
>>> rng = np.random.RandomState(0)
>>> pts = rng.rand(64, 2) * 20 + 5
>>> e1, e2 = verts[0:2]
>>> closest_point_on_line_segment(pts, e1, e2)
```

2.1.1.3.1.2 kwcoco.demo.perterb module

kwcoco.demo.perterb.**perterb_coco**(*coco_dset*, ***kwargs*)

Perterbs a coco dataset

Parameters

- **rng** (*int, default=0*)
- **box_noise** (*int, default=0*)
- **cls_noise** (*int, default=0*)
- **null_pred** (*bool, default=False*)
- **with_probs** (*bool, default=False*)
- **score_noise** (*float, default=0.2*)
- **hacked** (*int, default=1*)

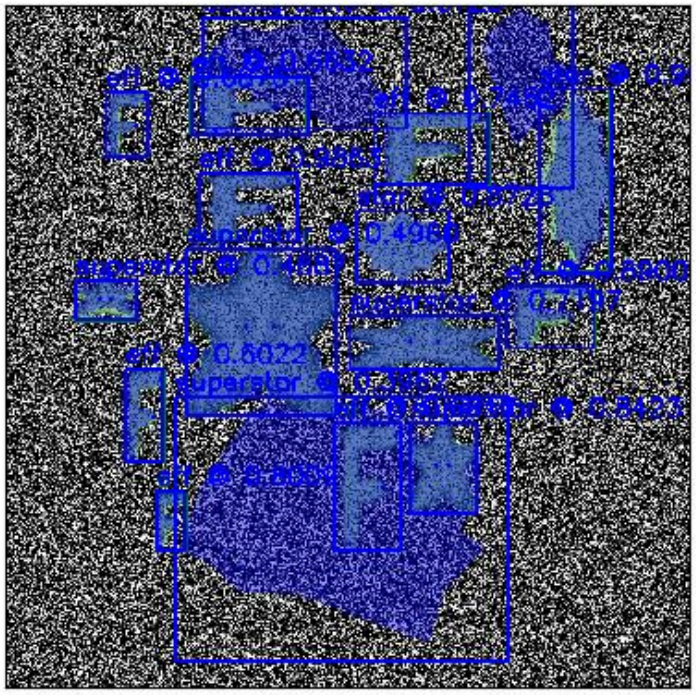
Example

```
>>> from kwcoco.demo.perterb import *  # NOQA
>>> from kwcoco.demo.perterb import _demo_construct_probs
>>> import kwcoco
>>> coco_dset = true_dset = kwcoco.CocoDataset.demo('shapes2')
>>> kwargs = {
>>>     'box_noise': 0.5,
>>>     'n_fp': 3,
>>>     'with_probs': 1,
>>>     'with_heatmaps': 1,
>>> }
>>> pred_dset = perterb_coco(true_dset, **kwargs)
>>> pred_dset._check_json_serializable()
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> gid = 1
>>> canvas = true_dset.delayed_load(gid).finalize()
```

(continues on next page)

(continued from previous page)

```
>>> canvas = true_dset.annots(gid=gid).detections.draw_on(canvas, color='green')
>>> canvas = pred_dset.annots(gid=gid).detections.draw_on(canvas, color='blue')
>>> kwplot.imshow(canvas)
```



2.1.1.3.1.3 kwcoco.demo.toydata module

Generates "toydata" for demo and testing purposes.

Note: The implementation of *demodata_toy_img* and *demodata_toy_dset* should be redone using the tools built for *random_video_dset*, which have more extensible implementations.

kwcoco.demo.toydata.**demodata_toy_dset**(*image_size=(600, 600)*, *n_imgs=5*, *verbose=3*, *rng=0*, *newstyle=True*, *dpath=None*, *fpath=None*, *bundle_dpath=None*, *aux=None*, *use_cache=True*, ***kwargs*)

Create a toy detection problem

Parameters

- **image_size** (*Tuple[int, int]*) – The width and height of the generated images
- **n_imgs** (*int*) – number of images to generate
- **rng** (*int | RandomState, default=0*) – random number generator or seed

- **newstyle** (*bool, default=True*) – create newstyle kwcoco data
- **dpath** (*str*) – path to the directory that will contain the bundle, (defaults to a kwcoco cache dir). Ignored if *bundle_dpath* is given.
- **fpath** (*str*) – path to the kwcoco file. The parent will be the bundle if it is not specified. Should be a descendant of the dpath if specified.
- **bundle_dpath** (*str*) – path to the directory that will store images. If specified, dpath is ignored. If unspecified, a bundle will be written inside *dpath*.
- **aux** (*bool*) – if True generates dummy auxiliary channels
- **verbose** (*int, default=3*) – verbosity mode
- **use_cache** (*bool, default=True*) – if True caches the generated json in the *dpath*.
- ****kwargs** – used for old backwards compatible argument names gsize - alias for image_size

Return type
kwcoco.CocoDataset

SeeAlso:
random_video_dset

CommandLine

```
xdoctest -m kwcoco.demo.toydata_image demodata_toy_dset --show
```

Todo:

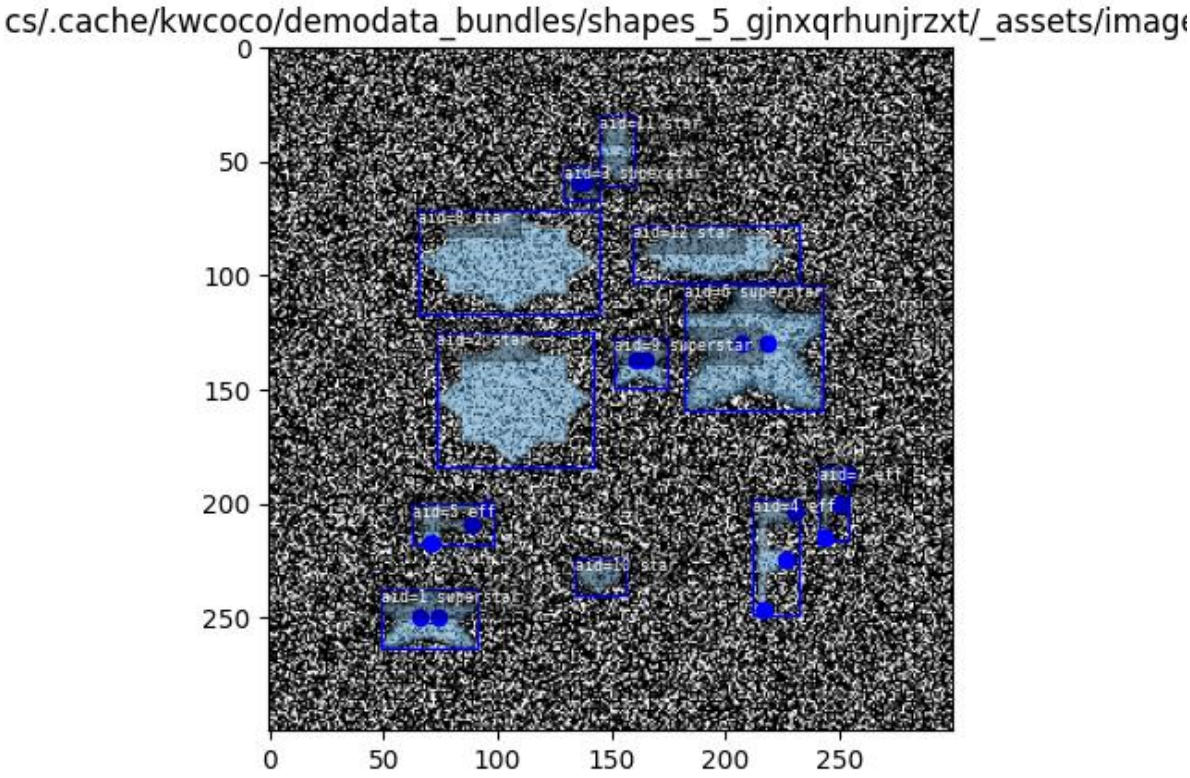
- [ ] Non-homogeneous images sizes

Example

```
>>> from kwcoco.demo.toydata_image import *
>>> import kwcoco
>>> dset = demodata_toy_dset(image_size=(300, 300), aux=True, use_cache=False)
>>> # xdoctest: +REQUIRES(--show)
>>> print(ub.repr2(dset.dataset, nl=2))
>>> import kwplot
>>> kwplot.autompl()
>>> dset.show_image(gid=1)
>>> ub.startfile(dset.bundle_dpath)
```

dset._tree()

```
>>> from kwcoco.demo.toydata_image import *
>>> import kwcoco
```



dset = demodata_toy_dset(image_size=(300, 300), aux=True, use_cache=False) print(dset.imgs[1]) dset._tree()

dset = demodata_toy_dset(image_size=(300, 300), aux=True, use_cache=False,
bundle_dpath='test_bundle')

print(dset.imgs[1]) dset._tree()

dset = demodata_toy_dset(
image_size=(300, 300), aux=True, use_cache=False, dpath='test_cache_dpath')

kwcoco.demo.toydata.**random_single_video_dset**(*image_size=(600, 600)*, *num_frames=5*, *num_tracks=3*, *tid_start=1*, *gid_start=1*, *video_id=1*, *anchors=None*, *rng=None*, *render=False*, *dpath=None*, *autobuild=True*, *verbose=3*, *aux=None*, *multispectral=False*, *max_speed=0.01*, *channels=None*, *multisensor=False*, ***kwargs*)

Create the video scene layout of object positions.

Note: Does not render the data unless specified.

Parameters

- **image_size** (*Tuple[int, int]*) – size of the images
- **num_frames** (*int*) – number of frames in this video
- **num_tracks** (*int*) – number of tracks in this video

- **tid_start** (*int, default=1*) – track-id start index
- **gid_start** (*int, default=1*) – image-id start index
- **video_id** (*int, default=1*) – video-id of this video
- **anchors** (*ndarray | None*) – base anchor sizes of the object boxes we will generate.
- **rng** (*RandomState*) – random state / seed
- **render** (*bool | dict*) – if truthy, does the rendering according to provided params in the case of dict input.
- **autobuild** (*bool, default=True*) – prebuild coco lookup indexes
- **verbose** (*int*) – verbosity level
- **aux** (*bool | List[str]*) – if specified generates auxiliary channels
- **multispectral** (*bool*) – if specified simulates multispectral imagry This is similar to aux, but has no "main" file.
- **max_speed** (*float*) – max speed of movers
- **channels** (*str | None | kwcoco.ChannelSpec*) – if specified generates multispectral images with dummy channels
- **multisensor** (*bool*) –

  if True, generates demodata from "multiple sensors", in
  other words, observations may have different "bands".
- ****kwargs** – used for old backwards compatible argument names gsize - alias for image_size

---

Todo:

- [ ] Need maximum allowed object overlap measure
- [ ] Need better parameterized path generation

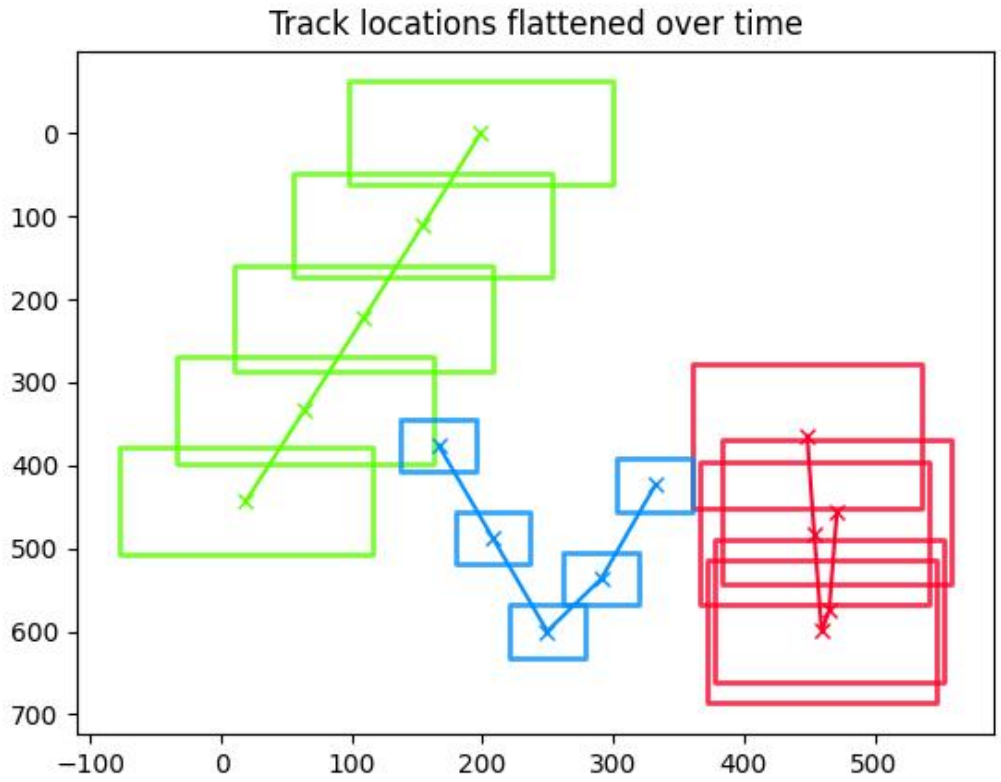
Example

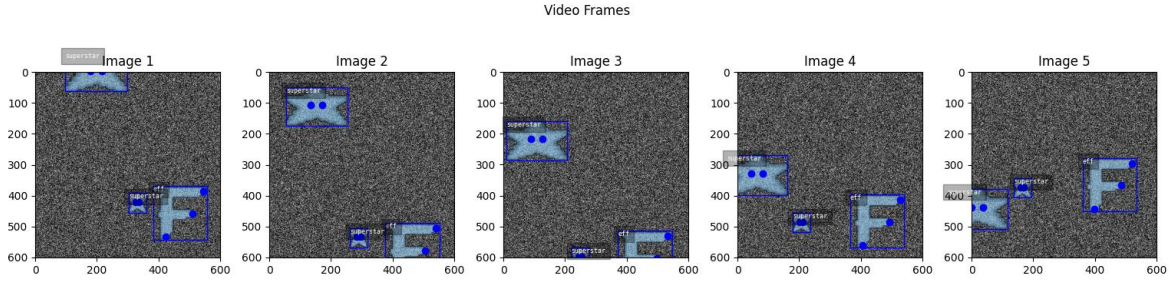
```
>>> import numpy as np
>>> from kwcoco.demo.toydata_video import random_single_video_dset
>>> anchors = np.array([ [0.3, 0.3],  [0.1, 0.1]])
>>> dset = random_single_video_dset(render=True, num_frames=5,
>>>                                 num_tracks=3, anchors=anchors,
>>>                                 max_speed=0.2, rng=91237446)
>>> # xdoctest: +REQUIRES(--show)
>>> # Show the tracks in a single image
>>> import kwplot
>>> import kwimage
>>> #kwplot.autosns()
>>> kwplot.autoplt()
>>> # Group track boxes and centroid locations
>>> paths = []
>>> track_boxes = []
>>> for tid, aids in dset.index.trackid_to_aids.items():
```

(continues on next page)

(continued from previous page)

```
>>>     boxes = dset.annots(aids).boxes.to_cxywh()
>>>     path = boxes.data[:, 0:2]
>>>     paths.append(path)
>>>     track_boxes.append(boxes)
>>> # Plot the tracks over time
>>> ax = kwplot.figure(fnum=1, doclf=1).gca()
>>> colors = kwimage.Color.distinct(len(track_boxes))
>>> for i, boxes in enumerate(track_boxes):
>>>     color = colors[i]
>>>     path = boxes.data[:, 0:2]
>>>     boxes.draw(color=color, centers={'radius': 0.01}, alpha=0.8)
>>>     ax.plot(path.T[0], path.T[1], 'x-', color=color)
>>> ax.invert_yaxis()
>>> ax.set_title('Track locations flattened over time')
>>> # Plot the image sequence
>>> fig = kwplot.figure(fnum=2, doclf=1)
>>> gids = list(dset.imgs.keys())
>>> pnums = kwplot.PlotNums(nRows=1, nSubplots=len(gids))
>>> for gid in gids:
>>>     dset.show_image(gid, pnum=pnums(), fnum=2, title=f'Image {gid}', show_aid=0,
↪ setlim='image')
>>> fig.suptitle('Video Frames')
>>> fig.set_size_inches(15.4, 4.0)
>>> fig.tight_layout()
>>> kwplot.show_if_requested()
```





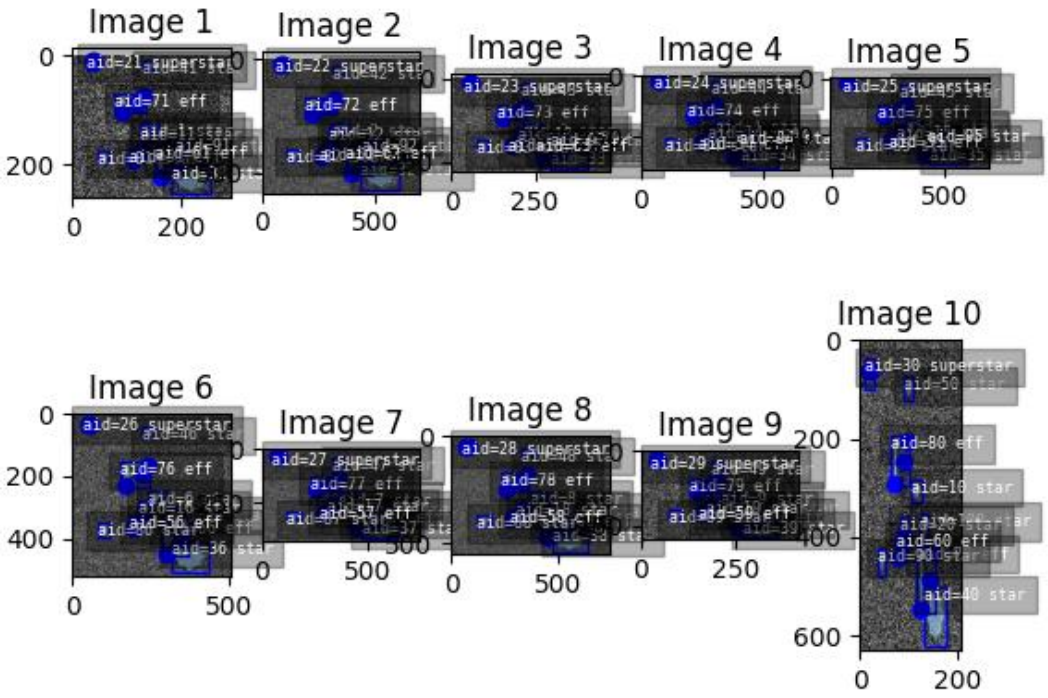
Example

```
>>> from kwcoco.demo.toydata_video import *  # NOQA
>>> anchors = np.array([ [0.2, 0.2],  [0.1, 0.1]])
>>> gsize = np.array([(600, 600)])
>>> print(anchors * gsize)
>>> dset = random_single_video_dset(render=True, num_frames=10,
>>>                                 anchors=anchors, num_tracks=10,
>>>                                 image_size='random')
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> plt = kwplot.autoplt()
>>> plt.clf()
```

(continues on next page)

(continued from previous page)

```
>>> gids = list(dset.imgs.keys())
>>> pnums = kwplot.PlotNums(nSubplots=len(gids))
>>> for gid in gids:
>>>     dset.show_image(gid, pnum=pnums(), fnum=1, title=f'Image {gid}')
>>> kwplot.show_if_requested()
```



Example

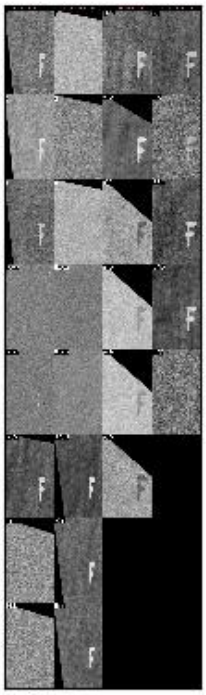
```
>>> from kwcoco.demo.toydata_video import *  # NOQA
>>> dset = random_single_video_dset(num_frames=10, num_tracks=10, aux=True)
>>> assert 'auxiliary' in dset.imgs[1]
>>> assert dset.imgs[1]['auxiliary'][0]['channels']
>>> assert dset.imgs[1]['auxiliary'][1]['channels']
```

Example

```
>>> from kwcoco.demo.toydata_video import *  # NOQA
>>> multispectral = True
>>> dset = random_single_video_dset(num_frames=1, num_tracks=1, multispectral=True)
>>> dset._check_json_serializable()
>>> dset.dataset['images']
>>> assert dset.imgs[1]['auxiliary'][1]['channels']
>>> # test that we can render
>>> render_toy_dataset(dset, rng=0, dpath=None, renderkw={})
```

Example

```
>>> from kwcoco.demo.toydata_video import *  # NOQA
>>> dset = random_single_video_dset(num_frames=4, num_tracks=1, multispectral=True,␣
↪multisensor=True, image_size='random', rng=2338)
>>> dset._check_json_serializable()
>>> assert dset.imgs[1]['auxiliary'][1]['channels']
>>> # Print before and after render
>>> #print('multisensor-images = {}'.format(ub.repr2(dset.dataset['images'], nl=-2)))
>>> #print('multisensor-images = {}'.format(ub.repr2(dset.dataset, nl=-2)))
>>> print(ub.hash_data(dset.dataset))
>>> # test that we can render
>>> render_toy_dataset(dset, rng=0, dpath=None, renderkw={})
>>> #print('multisensor-images = {}'.format(ub.repr2(dset.dataset['images'], nl=-2)))
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> from kwcoco.demo.toydata_video import _draw_video_sequence  # NOQA
>>> gids = [1, 2, 3, 4]
>>> final = _draw_video_sequence(dset, gids)
>>> print('dset.fpath = {!r}'.format(dset.fpath))
>>> kwplot.imshow(final)
```



kwcoco.demo.toydata.**random_video_dset**(*num_videos=1*, *num_frames=2*, *num_tracks=2*, *anchors=None*, *image_size=(600, 600)*, *verbose=3*, *render=False*, *aux=None*, *multispectral=False*, *multisensor=False*, *rng=None*, *dpath=None*, *max_speed=0.01*, *channels=None*, ***kwargs*)

Create a toy Coco Video Dataset

Parameters

- **num_videos** (*int*) – number of videos
- **num_frames** (*int*) – number of images per video
- **num_tracks** (*int*) – number of tracks per video
- **image_size** (*Tuple[int, int]*) – The width and height of the generated images
- **render** (*bool | dict*) – if truthy the toy annotations are synthetically rendered. See `render_toy_image()` for details.
- **rng** (*int | None | RandomState*) – random seed / state
- **dpath** (*str*) – only used if render is truthy, place to write rendered images.
- **verbose** (*int, default=3*) – verbosity mode
- **aux** (*bool*) – if True generates dummy auxiliary channels
- **multispectral** (*bool*) – similar to aux, but does not have the concept of a "main" image.
- **max_speed** (*float*) – max speed of movers
- **channels** (*str*) – experimental new way to get MSI with specific band distributions.

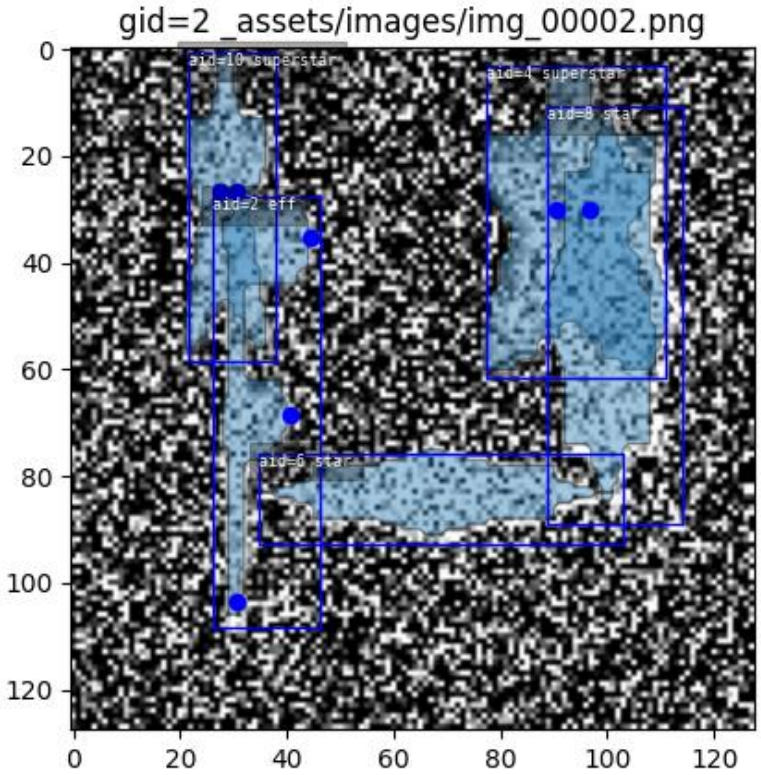
- ****kwargs** – used for old backwards compatible argument names gsize - alias for image_size

SeeAlso:
random_single_video_dset

Example

```
>>> from kwcoco.demo.toydata_video import *  # NOQA
>>> dset = random_video_dset(render=True, num_videos=3, num_frames=2,
>>>                          num_tracks=5, image_size=(128, 128))
>>> # xdoctest: +REQUIRES(--show)
>>> dset.show_image(1, doclf=True)
>>> dset.show_image(2, doclf=True)
```

```
>>> from kwcoco.demo.toydata_video import *  # NOQA
dset = random_video_dset(render=False, num_videos=3, num_frames=2,
    num_tracks=10)
dset._tree()
dset.imgs[1]
```



kwcoco.demo.toydata.**demodata_toy_img**(*anchors=None*, *image_size=(104, 104)*, *categories=None*, *n_annots=(0, 50)*, *fg_scale=0.5*, *bg_scale=0.8*, *bg_intensity=0.1*, *fg_intensity=0.9*, *gray=True*, *centerobj=None*, *exact=False*, *newstyle=True*, *rng=None*, *aux=None*, ***kwargs*)

Generate a single image with non-overlapping toy objects of available categories.

Todo:

DEPRECATE IN FAVOR OF
random_single_video_dset + render_toy_image

Parameters

- **anchors** (*ndarray*) – Nx2 base width / height of boxes
- **gsize** (*Tuple[int, int]*) – width / height of the image
- **categories** (*List[str]*) – list of category names
- **n_annots** (*Tuple | int*) – controls how many annotations are in the image. if it is a tuple, then it is interpreted as uniform random bounds
- **fg_scale** (*float*) – standard deviation of foreground intensity
- **bg_scale** (*float*) – standard deviation of background intensity
- **bg_intensity** (*float*) – mean of background intensity
- **fg_intensity** (*float*) – mean of foreground intensity
- **centerobj** (*bool*) – if 'pos', then the first annotation will be in the center of the image, if 'neg', then no annotations will be in the center.
- **exact** (*bool*) – if True, ensures that exactly the number of specified annots are generated.
- **newstyle** (*bool*) – use new-sytle kwcoco format
- **rng** (*RandomState*) – the random state used to seed the process
- **aux** – if specified builds auxiliary channels
- ****kwargs** – used for old backwards compatible argument names. gsize - alias for image_size

CommandLine

```
xdoctest -m kwcoco.demo.toydata_image demodata_toy_img:0 --profile
xdoctest -m kwcoco.demo.toydata_image demodata_toy_img:1 --show
```

Example

```
>>> from kwcoco.demo.toydata_image import *  # NOQA
>>> img, anns = demodata_toy_img(image_size=(32, 32), anchors=[[.3, .3]], rng=0)
>>> img['imdata'] = '<ndarray shape={}>'.format(img['imdata'].shape)
>>> print('img = {}'.format(ub.repr2(img)))
>>> print('anns = {}'.format(ub.repr2(anns, nl=2, cbr=True)))
>>> # xdoctest: +IGNORE_WANT
img = {
    'height': 32,
    'imdata': '<ndarray shape=(32, 32, 3)>',
    'width': 32,
```

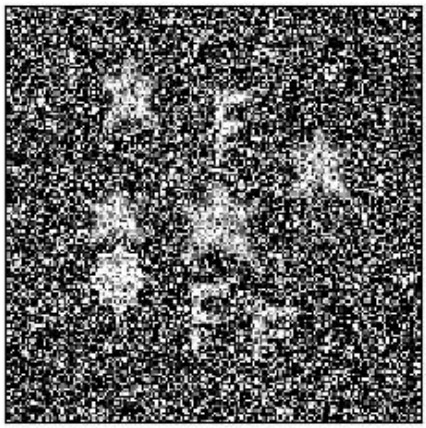
(continues on next page)

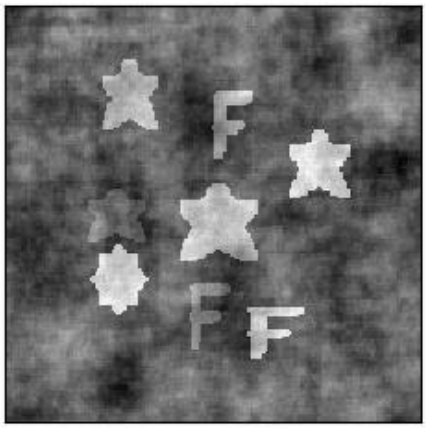
(continued from previous page)

```
}
anns = [{'bbox': [15, 10, 9, 8],
  'category_name': 'star',
  'keypoints': [],
  'segmentation': {'counts': '[`06j0000O20N1000e8', 'size': [32, 32]},},
 {'bbox': [11, 20, 7, 7],
  'category_name': 'star',
  'keypoints': [],
  'segmentation': {'counts': 'g;1m04N0O20N102L[=', 'size': [32, 32]},},
 {'bbox': [4, 4, 8, 6],
  'category_name': 'superstar',
  'keypoints': [{'keypoint_category': 'left_eye', 'xy': [7.25, 6.8125]}, {'keypoint_
↪category': 'right_eye', 'xy': [8.75, 6.8125]}],
  'segmentation': {'counts': 'U4210j0300O01010O00MVO0ed0', 'size': [32, 32]},},
 {'bbox': [3, 20, 6, 7],
  'category_name': 'star',
  'keypoints': [],
  'segmentation': {'counts': 'g31m04N000002L[f0', 'size': [32, 32]},},]
```

Example

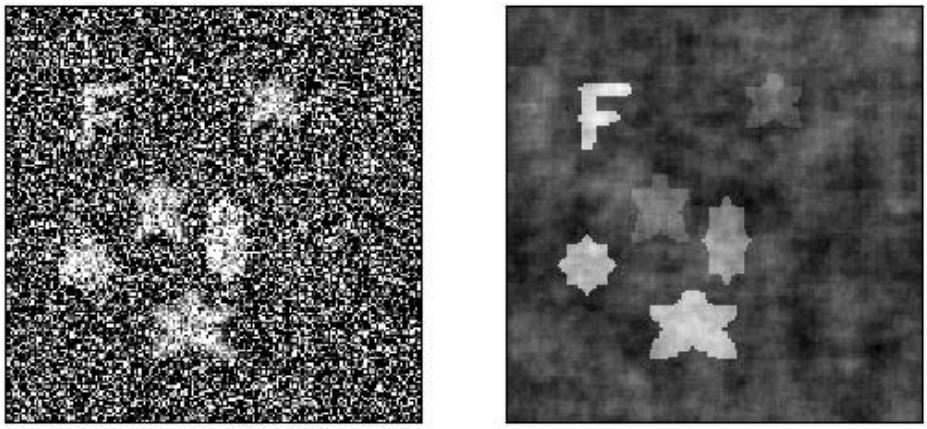
```
>>> # xdoctest: +REQUIRES(--show)
>>> img, anns = demodata_toy_img(image_size=(172, 172), rng=None, aux=True)
>>> print('anns = {}'.format(ub.repr2(anns, nl=1)))
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(img['imdata'], pnum=(1, 2, 1), fnum=1)
>>> auxdata = img['auxiliary'][0]['imdata']
>>> kwplot.imshow(auxdata, pnum=(1, 2, 2), fnum=1)
>>> kwplot.show_if_requested()
```





Example

```
>>> # xdoctest: +REQUIRES(--show)
>>> img, anns = demodata_toy_img(image_size=(172, 172), rng=None, aux=True)
>>> print('anns = {}'.format(ub.repr2(anns, nl=1)))
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(img['imdata'], pnum=(1, 2, 1), fnum=1)
>>> auxdata = img['auxiliary'][0]['imdata']
>>> kwplot.imshow(auxdata, pnum=(1, 2, 2), fnum=1)
>>> kwplot.show_if_requested()
```



2.1.1.3.1.4 kwcoco.demo.toydata_image module

Generates “toydata” for demo and testing purposes.

Loose image version of the toydata generators.

Note: The implementation of *demodata_toy_img* and *demodata_toy_dset* should be redone using the tools built for *random_video_dset*, which have more extensible implementations.

---

kwcoco.demo.toydata_image.**demodata_toy_dset**(*image_size=(600, 600)*, *n_imgs=5*, *verbose=3*, *rng=0*, *newstyle=True*, *dpath=None*, *fpath=None*, *bundle_dpath=None*, *aux=None*, *use_cache=True*, ***kwargs*)

Create a toy detection problem

Parameters

- **image_size** (*Tuple[int, int]*) – The width and height of the generated images
- **n_imgs** (*int*) – number of images to generate
- **rng** (*int | RandomState, default=0*) – random number generator or seed
- **newstyle** (*bool, default=True*) – create newstyle kwcoco data

- **dpath** (*str*) – path to the directory that will contain the bundle, (defaults to a kwcoco cache dir). Ignored if *bundle_dpath* is given.
- **fpath** (*str*) – path to the kwcoco file. The parent will be the bundle if it is not specified. Should be a descendant of the dpath if specified.
- **bundle_dpath** (*str*) – path to the directory that will store images. If specified, dpath is ignored. If unspecified, a bundle will be written inside *dpath*.
- **aux** (*bool*) – if True generates dummy auxiliary channels
- **verbose** (*int, default=3*) – verbosity mode
- **use_cache** (*bool, default=True*) – if True caches the generated json in the *dpath*.
- ****kwargs** – used for old backwards compatible argument names gsize - alias for image_size

Return type
kwcoco.CocoDataset

SeeAlso:
random_video_dset

CommandLine

```
xdoctest -m kwcoco.demo.toydata_image demodata_toy_dset --show
```

Todo:

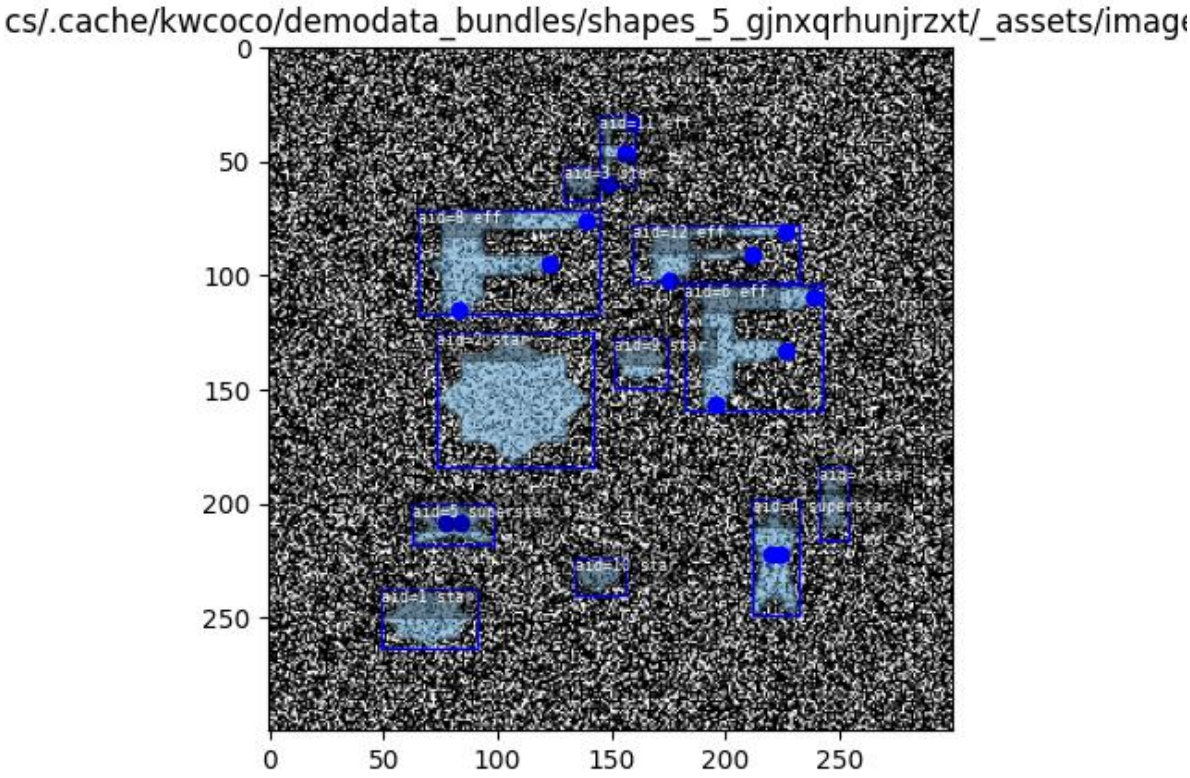
- [ ] Non-homogeneous images sizes

Example

```
>>> from kwcoco.demo.toydata_image import *
>>> import kwcoco
>>> dset = demodata_toy_dset(image_size=(300, 300), aux=True, use_cache=False)
>>> # xdoctest: +REQUIRES(--show)
>>> print(ub.repr2(dset.dataset, nl=2))
>>> import kwplot
>>> kwplot.autompl()
>>> dset.show_image(gid=1)
>>> ub.startfile(dset.bundle_dpath)
```

dset._tree()

```
>>> from kwcoco.demo.toydata_image import *
>>> import kwcoco
```



dset = demodata_toy_dset(image_size=(300, 300), aux=True, use_cache=False) print(dset.imgs[1]) dset._tree()

dset = demodata_toy_dset(image_size=(300, 300), aux=True, use_cache=False,
bundle_dpath='test_bundle')

print(dset.imgs[1]) dset._tree()

dset = demodata_toy_dset(
image_size=(300, 300), aux=True, use_cache=False, dpath='test_cache_dpath')

kwcoco.demo.toydata_image.**demodata_toy_img**(*anchors=None*, *image_size=(104, 104)*, *categories=None*, *n_annots=(0, 50)*, *fg_scale=0.5*, *bg_scale=0.8*, *bg_intensity=0.1*, *fg_intensity=0.9*, *gray=True*, *centerobj=None*, *exact=False*, *newstyle=True*, *rng=None*, *aux=None*, ***kwargs*)

Generate a single image with non-overlapping toy objects of available categories.

Todo:

DEPRECATE IN FAVOR OF
random_single_video_dset + render_toy_image

Parameters

- **anchors** (*ndarray*) – Nx2 base width / height of boxes
- **gsize** (*Tuple[int, int]*) – width / height of the image

- **categories** (*List[str]*) – list of category names
- **n_annots** (*Tuple | int*) – controls how many annotations are in the image. if it is a tuple, then it is interpreted as uniform random bounds
- **fg_scale** (*float*) – standard deviation of foreground intensity
- **bg_scale** (*float*) – standard deviation of background intensity
- **bg_intensity** (*float*) – mean of background intensity
- **fg_intensity** (*float*) – mean of foreground intensity
- **centerobj** (*bool*) – if 'pos', then the first annotation will be in the center of the image, if 'neg', then no annotations will be in the center.
- **exact** (*bool*) – if True, ensures that exactly the number of specified annots are generated.
- **newstyle** (*bool*) – use new-sytle kwcoco format
- **rng** (*RandomState*) – the random state used to seed the process
- **aux** – if specified builds auxiliary channels
- ****kwargs** – used for old backwards compatible argument names. gsize - alias for image_size

CommandLine

```
xdoctest -m kwcoco.demo.toydata_image demodata_toy_img:0 --profile
xdoctest -m kwcoco.demo.toydata_image demodata_toy_img:1 --show
```

Example

```
>>> from kwcoco.demo.toydata_image import *  # NOQA
>>> img, anns = demodata_toy_img(image_size=(32, 32), anchors=[[.3, .3]], rng=0)
>>> img['imdata'] = '<ndarray shape={}>'.format(img['imdata'].shape)
>>> print('img = {}'.format(ub.repr2(img)))
>>> print('anns = {}'.format(ub.repr2(anns, nl=2, cbr=True)))
>>> # xdoctest: +IGNORE_WANT
img = {
    'height': 32,
    'imdata': '<ndarray shape=(32, 32, 3)>',
    'width': 32,
}
anns = [{'bbox': [15, 10, 9, 8],
  'category_name': 'star',
  'keypoints': [],
  'segmentation': {'counts': '[`06j0000O20N1000e8', 'size': [32, 32]},},
 {'bbox': [11, 20, 7, 7],
  'category_name': 'star',
  'keypoints': [],
  'segmentation': {'counts': 'g;1m04N0O20N102L[=', 'size': [32, 32]},},
 {'bbox': [4, 4, 8, 6],
  'category_name': 'superstar',
  'keypoints': [{'keypoint_category': 'left_eye', 'xy': [7.25, 6.8125]}, {'keypoint_
↪category': 'right_eye', 'xy': [8.75, 6.8125]}],
```

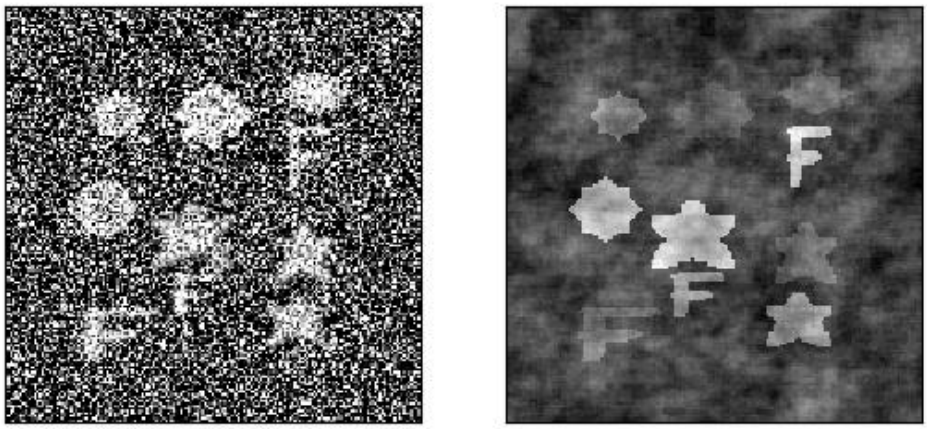
(continues on next page)

(continued from previous page)

```
  'segmentation': {'counts': 'U4210j0300O01010O00MVO0ed0', 'size': [32, 32]},},
 {'bbox': [3, 20, 6, 7],
  'category_name': 'star',
  'keypoints': [],
  'segmentation': {'counts': 'g31m04N000002L[f0', 'size': [32, 32]},},]
```

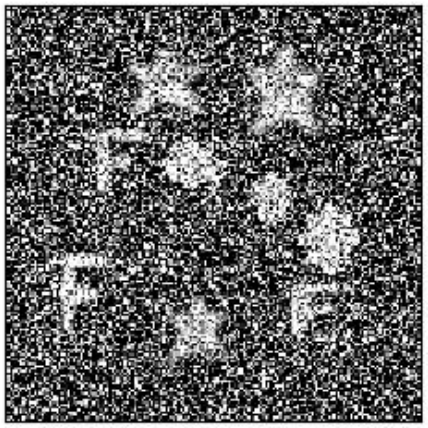
Example

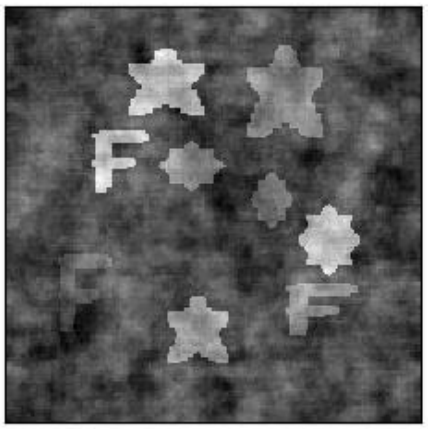
```
>>> # xdoctest: +REQUIRES(--show)
>>> img, anns = demodata_toy_img(image_size=(172, 172), rng=None, aux=True)
>>> print('anns = {}'.format(ub.repr2(anns, nl=1)))
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(img['imdata'], pnum=(1, 2, 1), fnum=1)
>>> auxdata = img['auxiliary'][0]['imdata']
>>> kwplot.imshow(auxdata, pnum=(1, 2, 2), fnum=1)
>>> kwplot.show_if_requested()
```



Example

```
>>> # xdoctest: +REQUIRES(--show)
>>> img, anns = demodata_toy_img(image_size=(172, 172), rng=None, aux=True)
>>> print('anns = {}'.format(ub.repr2(anns, nl=1)))
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(img['imdata'], pnum=(1, 2, 1), fnum=1)
>>> auxdata = img['auxiliary'][0]['imdata']
>>> kwplot.imshow(auxdata, pnum=(1, 2, 2), fnum=1)
>>> kwplot.show_if_requested()
```





2.1.1.3.1.5 kwcoco.demo.toydata_video module

Generates "toydata" for demo and testing purposes.

This is the video version of the toydata generator and should be prefered to the loose image version in toydata_image.

kwcoco.demo.toydata_video.**random_video_dset**(*num_videos=1*, *num_frames=2*, *num_tracks=2*, *anchors=None*, *image_size=(600, 600)*, *verbose=3*, *render=False*, *aux=None*, *multispectral=False*, *multisensor=False*, *rng=None*, *dpath=None*, *max_speed=0.01*, *channels=None*, ***kwargs*)

Create a toy Coco Video Dataset

Parameters

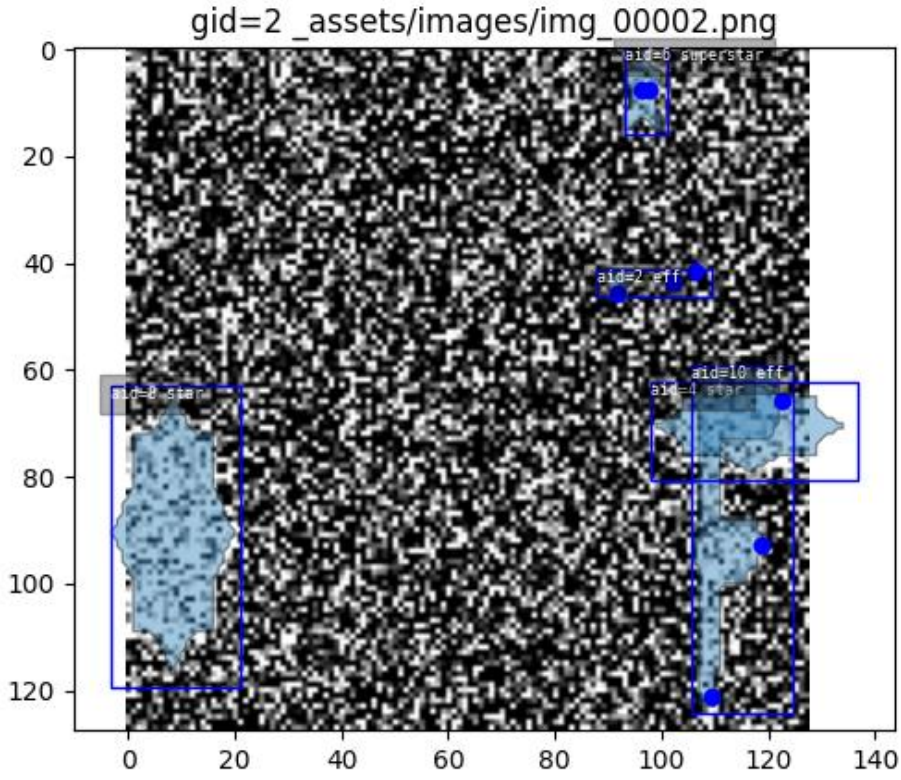
- **num_videos** (*int*) – number of videos
- **num_frames** (*int*) – number of images per video
- **num_tracks** (*int*) – number of tracks per video
- **image_size** (*Tuple[int, int]*) – The width and height of the generated images
- **render** (*bool | dict*) – if truthy the toy annotations are synthetically rendered. See *render_toy_image()* for details.
- **rng** (*int | None | RandomState*) – random seed / state
- **dpath** (*str*) – only used if render is truthy, place to write rendered images.
- **verbose** (*int, default=3*) – verbosity mode
- **aux** (*bool*) – if True generates dummy auxiliary channels
- **multispectral** (*bool*) – similar to aux, but does not have the concept of a "main" image.
- **max_speed** (*float*) – max speed of movers
- **channels** (*str*) – experimental new way to get MSI with specific band distributions.
- ****kwargs** – used for old backwards compatible argument names gsize - alias for image_size

SeeAlso:
random_single_video_dset

Example

```
>>> from kwcoco.demo.toydata_video import *  # NOQA
>>> dset = random_video_dset(render=True, num_videos=3, num_frames=2,
>>>                          num_tracks=5, image_size=(128, 128))
>>> # xdoctest: +REQUIRES(--show)
>>> dset.show_image(1, doclf=True)
>>> dset.show_image(2, doclf=True)
```

```
>>> from kwcoco.demo.toydata_video import *  # NOQA
dset = random_video_dset(render=False, num_videos=3, num_frames=2,
    num_tracks=10)
dset._tree()
dset.imgs[1]
```



kwcoco.demo.toydata_video.**random_single_video_dset**(*image_size=(600, 600)*, *num_frames=5*, *num_tracks=3*, *tid_start=1*, *gid_start=1*, *video_id=1*, *anchors=None*, *rng=None*, *render=False*, *dpath=None*, *autobuild=True*, *verbose=3*, *aux=None*, *multispectral=False*, *max_speed=0.01*, *channels=None*, *multisensor=False*, ***kwargs*)

Create the video scene layout of object positions.

Note: Does not render the data unless specified.

Parameters

- **image_size** (*Tuple[int, int]*) – size of the images
- **num_frames** (*int*) – number of frames in this video
- **num_tracks** (*int*) – number of tracks in this video
- **tid_start** (*int, default=1*) – track-id start index
- **gid_start** (*int, default=1*) – image-id start index
- **video_id** (*int, default=1*) – video-id of this video
- **anchors** (*ndarray | None*) – base anchor sizes of the object boxes we will generate.
- **rng** (*RandomState*) – random state / seed

- **render** (*bool | dict*) – if truthy, does the rendering according to provided params in the case of dict input.
- **autobuild** (*bool, default=True*) – prebuild coco lookup indexes
- **verbose** (*int*) – verbosity level
- **aux** (*bool | List[str]*) – if specified generates auxiliary channels
- **multispectral** (*bool*) – if specified simulates multispectral imagry This is similar to aux, but has no "main" file.
- **max_speed** (*float*) – max speed of movers
- **channels** (*str | None | kwcoco.ChannelSpec*) – if specified generates multispectral images with dummy channels
- **multisensor** (*bool*) –

  if True, generates demodata from "multiple sensors", in
  other words, observations may have different "bands".
- ****kwargs** – used for old backwards compatible argument names gsize - alias for image_size

---

Todo:

- [ ] Need maximum allowed object overlap measure
- [ ] Need better parameterized path generation

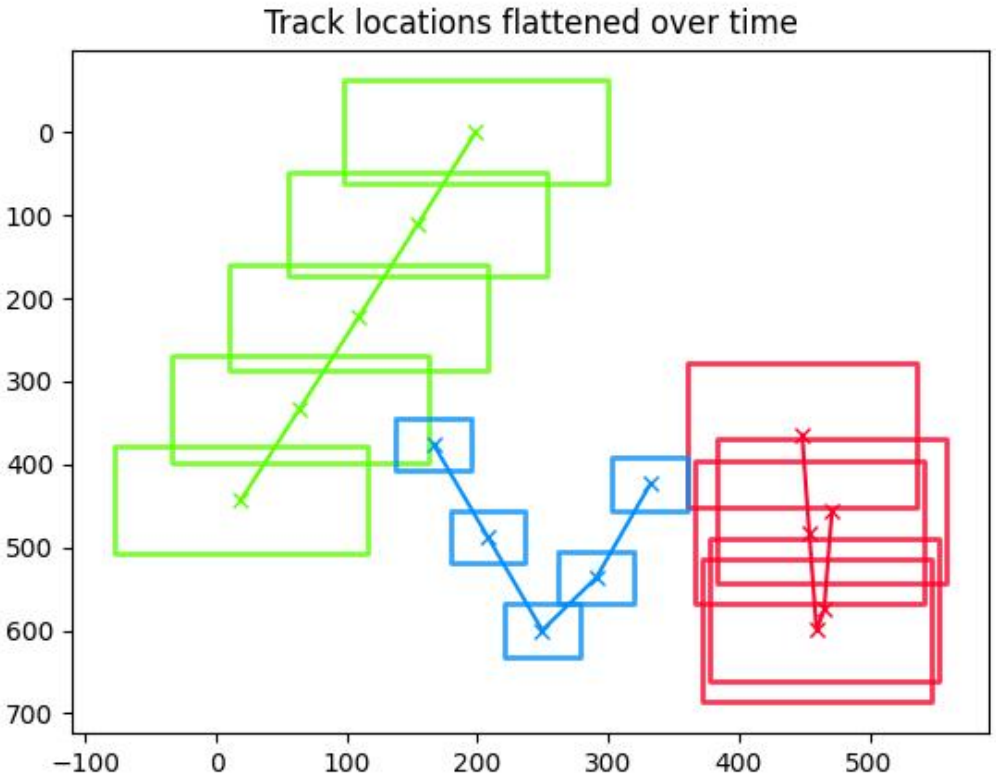
Example

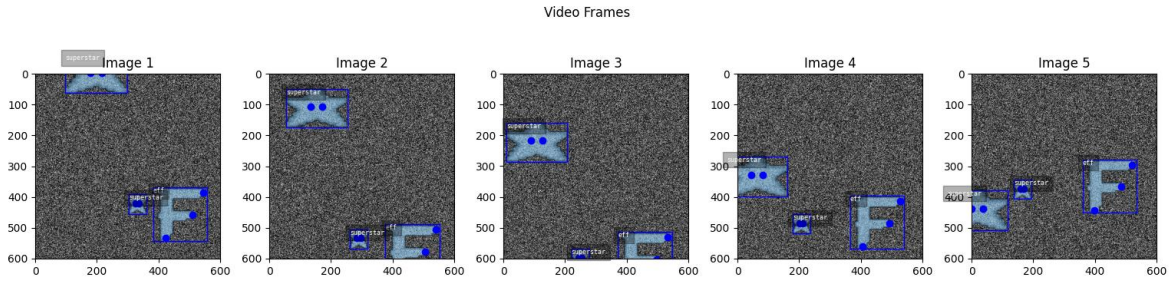
```
>>> import numpy as np
>>> from kwcoco.demo.toydata_video import random_single_video_dset
>>> anchors = np.array([ [0.3, 0.3],  [0.1, 0.1]])
>>> dset = random_single_video_dset(render=True, num_frames=5,
>>>                                 num_tracks=3, anchors=anchors,
>>>                                 max_speed=0.2, rng=91237446)
>>> # xdoctest: +REQUIRES(--show)
>>> # Show the tracks in a single image
>>> import kwplot
>>> import kwimage
>>> #kwplot.autosns()
>>> kwplot.autoplt()
>>> # Group track boxes and centroid locations
>>> paths = []
>>> track_boxes = []
>>> for tid, aids in dset.index.trackid_to_aids.items():
>>>     boxes = dset.annots(aids).boxes.to_cxywh()
>>>     path = boxes.data[:, 0:2]
>>>     paths.append(path)
>>>     track_boxes.append(boxes)
>>> # Plot the tracks over time
>>> ax = kwplot.figure(fnum=1, doclf=1).gca()
>>> colors = kwimage.Color.distinct(len(track_boxes))
>>> for i, boxes in enumerate(track_boxes):
```

(continues on next page)

(continued from previous page)

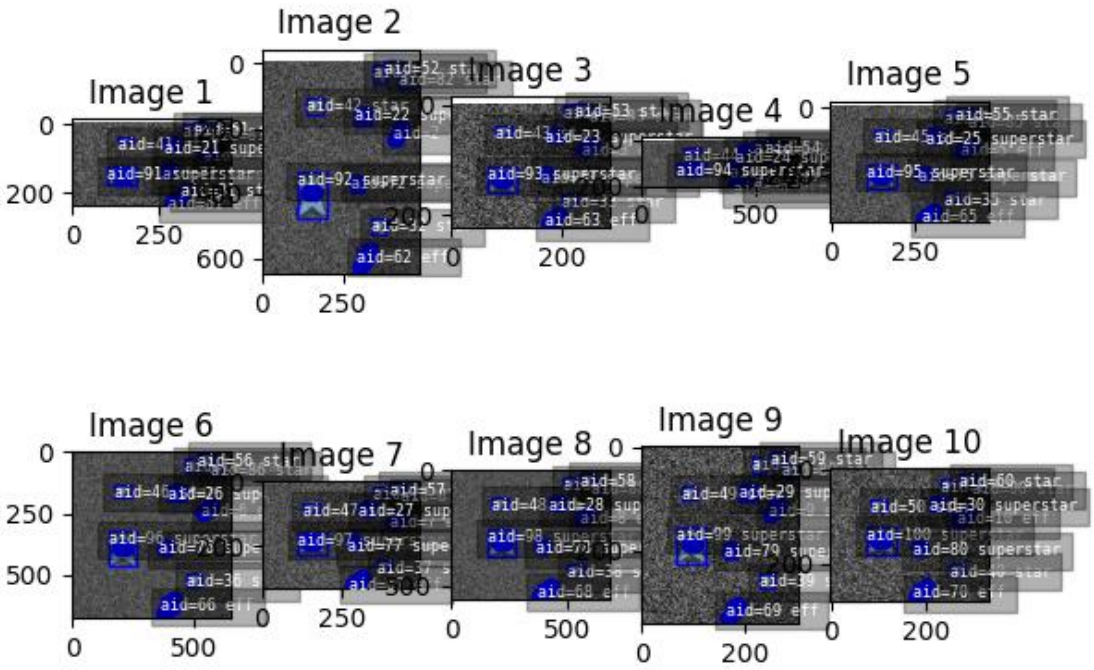
```
>>>     color = colors[i]
>>>     path = boxes.data[:, 0:2]
>>>     boxes.draw(color=color, centers={'radius': 0.01}, alpha=0.8)
>>>     ax.plot(path.T[0], path.T[1], 'x-', color=color)
>>> ax.invert_yaxis()
>>> ax.set_title('Track locations flattened over time')
>>> # Plot the image sequence
>>> fig = kwplot.figure(fnum=2, doclf=1)
>>> gids = list(dset.imgs.keys())
>>> pnums = kwplot.PlotNums(nRows=1, nSubplots=len(gids))
>>> for gid in gids:
>>>     dset.show_image(gid, pnum=pnums(), fnum=2, title=f'Image {gid}', show_aid=0,
↪ setlim='image')
>>> fig.suptitle('Video Frames')
>>> fig.set_size_inches(15.4, 4.0)
>>> fig.tight_layout()
>>> kwplot.show_if_requested()
```





Example

```
>>> from kwcoco.demo.toydata_video import *  # NOQA
>>> anchors = np.array([ [0.2, 0.2],  [0.1, 0.1]])
>>> gsize = np.array([(600, 600)])
>>> print(anchors * gsize)
>>> dset = random_single_video_dset(render=True, num_frames=10,
>>>                                 anchors=anchors, num_tracks=10,
>>>                                 image_size='random')
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> plt = kwplot.autoplt()
>>> plt.clf()
>>> gids = list(dset.imgs.keys())
>>> pnums = kwplot.PlotNums(nSubplots=len(gids))
>>> for gid in gids:
>>>     dset.show_image(gid, pnum=pnums(), fnum=1, title=f'Image {gid}')
>>> kwplot.show_if_requested()
```



Example

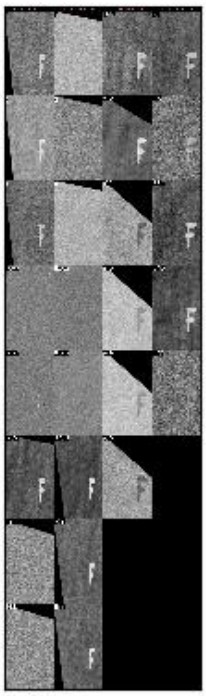
```
>>> from kwcoco.demo.toydata_video import *  # NOQA
>>> dset = random_single_video_dset(num_frames=10, num_tracks=10, aux=True)
>>> assert 'auxiliary' in dset.imgs[1]
>>> assert dset.imgs[1]['auxiliary'][0]['channels']
>>> assert dset.imgs[1]['auxiliary'][1]['channels']
```

Example

```
>>> from kwcoco.demo.toydata_video import *  # NOQA
>>> multispectral = True
>>> dset = random_single_video_dset(num_frames=1, num_tracks=1, multispectral=True)
>>> dset._check_json_serializable()
>>> dset.dataset['images']
>>> assert dset.imgs[1]['auxiliary'][1]['channels']
>>> # test that we can render
>>> render_toy_dataset(dset, rng=0, dpath=None, renderkw={})
```

Example

```
>>> from kwcoco.demo.toydata_video import *  # NOQA
>>> dset = random_single_video_dset(num_frames=4, num_tracks=1, multispectral=True,␣
↪multisensor=True, image_size='random', rng=2338)
>>> dset._check_json_serializable()
>>> assert dset.imgs[1]['auxiliary'][1]['channels']
>>> # Print before and after render
>>> #print('multisensor-images = {}'.format(ub.repr2(dset.dataset['images'], nl=-2)))
>>> #print('multisensor-images = {}'.format(ub.repr2(dset.dataset, nl=-2)))
>>> print(ub.hash_data(dset.dataset))
>>> # test that we can render
>>> render_toy_dataset(dset, rng=0, dpath=None, renderkw={})
>>> #print('multisensor-images = {}'.format(ub.repr2(dset.dataset['images'], nl=-2)))
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> from kwcoco.demo.toydata_video import _draw_video_sequence  # NOQA
>>> gids = [1, 2, 3, 4]
>>> final = _draw_video_sequence(dset, gids)
>>> print('dset.fpath = {!r}'.format(dset.fpath))
>>> kwplot.imshow(final)
```



kwcoco.demo.toydata_video.**render_toy_dataset**(*dset*, *rng*, *dpath=None*, *renderkw=None*, *verbose=0*)

Create toydata_video renderings for a preconstructed coco dataset.

Parameters

- **dset** (*kwcoco.CocoDataset*) – A dataset that contains special "renderable" annotations. (e.g. the demo shapes). Each image can contain special fields that influence how an image will be rendered.

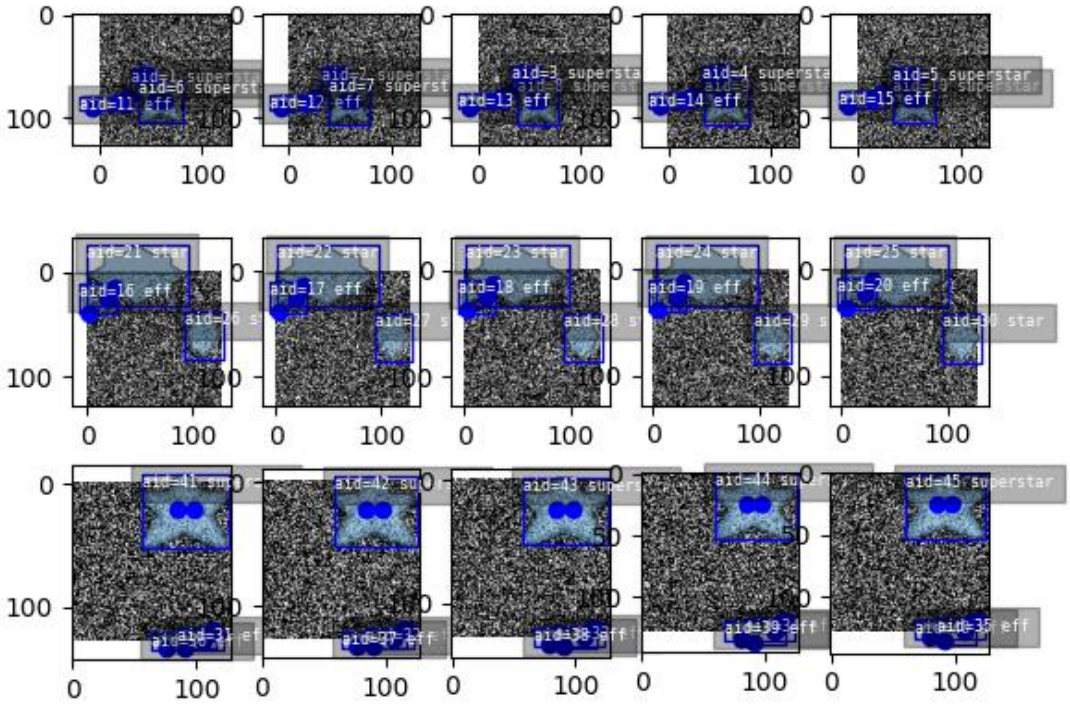
  Currently this process is simple, it just creates a noisy image with the shapes superimposed over where they should exist as indicated by the annotations. In the future this may become more sophisticated.

  Each item in *dset.dataset['images']* will be modified to add the "file_name" field indicating where the rendered data is writen.

- **rng** (*int | None | RandomState*) – random state
- **dpath** (*str*) – The location to write the images to. If unspecified, it is written to the rendered folder inside the kwcoco cache directory.
- **renderkw** (*dict*) – See *render_toy_image()* for details. Also takes imwrite keywords args only handled in this function. TODO better docs.

Example

```
>>> from kwcoco.demo.toydata_video import *  # NOQA
>>> import kwarray
>>> rng = None
>>> rng = kwarray.ensure_rng(rng)
>>> num_tracks = 3
>>> dset = random_video_dset(rng=rng, num_videos=3, num_frames=5,
>>>                          num_tracks=num_tracks, image_size=(128, 128))
>>> dset = render_toy_dataset(dset, rng)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> plt = kwplot.autoplt()
>>> plt.clf()
>>> gids = list(dset.imgs.keys())
>>> pnums = kwplot.PlotNums(nSubplots=len(gids), nRows=num_tracks)
>>> for gid in gids:
>>>     dset.show_image(gid, pnum=pnums(), fnum=1, title=False)
>>> pnums = kwplot.PlotNums(nSubplots=len(gids))
```



kwcoco.demo.toydata_video.**render_toy_image**(*dset*, *gid*, *rng=None*, *renderkw=None*)

Modifies dataset inplace, rendering synthetic annotations.

This does not write to disk. Instead this writes to placeholder values in the image dictionary.

Parameters

- **dset** (*kwcoco.CocoDataset*) – coco dataset with renderable anotations / images
- **gid** (*int*) – image to render
- **rng** (*int | None | RandomState*) – random state
- **renderkw** (*dict*) – rendering config gray (boo): gray or color images fg_scale (float): foreground noisyness (gauss std) bg_scale (float): background noisyness (gauss std) fg_intensity (float): foreground brightness (gauss mean) bg_intensity (float): background brightness (gauss mean) newstyle (bool): use new kwcoco datastructure formats with_kpts (bool): include keypoint info with_sseg (bool): include segmentation info

Returns

the inplace-modified image dictionary

Return type

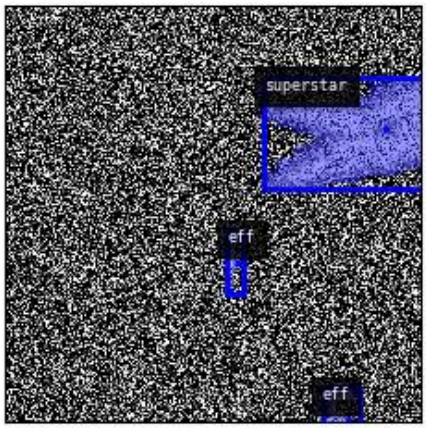
Dict

Example

```
>>> from kwcoco.demo.toydata_video import *  # NOQA
>>> image_size=(600, 600)
>>> num_frames=5
>>> verbose=3
>>> rng = None
>>> import kwarray
>>> rng = kwarray.ensure_rng(rng)
>>> aux = 'mx'
>>> dset = random_single_video_dset(
>>>     image_size=image_size, num_frames=num_frames, verbose=verbose, aux=aux,␣
↪rng=rng)
>>> print('dset.dataset = {}'.format(ub.repr2(dset.dataset, nl=2)))
>>> gid = 1
>>> renderkw = {}
>>> render_toy_image(dset, gid, rng, renderkw=renderkw)
>>> img = dset.imgs[gid]
>>> canvas = img['imdata']
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(canvas, doclf=True, pnum=(1, 2, 1))
>>> dets = dset.annots(gid=gid).detections
>>> dets.draw()
```

```
>>> auxdata = img['auxiliary'][0]['imdata']
>>> aux_canvas = false_color(auxdata)
>>> kwplot.imshow(aux_canvas, pnum=(1, 2, 2))
>>> _ = dets.draw()
```

```
>>> # xdoctest: +REQUIRES(--show)
>>> img, anns = demodata_toy_img(image_size=(172, 172), rng=None, aux=True)
>>> print('anns = {}'.format(ub.repr2(anns, nl=1)))
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(img['imdata'], pnum=(1, 2, 1), fnum=1)
>>> auxdata = img['auxiliary'][0]['imdata']
>>> kwplot.imshow(auxdata, pnum=(1, 2, 2), fnum=1)
>>> kwplot.show_if_requested()
```



Example

```
>>> from kwcoco.demo.toydata_video import *  # NOQA
>>> multispectral = True
>>> dset = random_single_video_dset(num_frames=1, num_tracks=1, multispectral=True)
>>> gid = 1
>>> dset.imgs[gid]
>>> rng = kwarray.ensure_rng(0)
>>> renderkw = {'with_sseg': True}
>>> img = render_toy_image(dset, gid, rng=rng, renderkw=renderkw)
```

kwcoco.demo.toydata_video.**render_foreground**(*imdata*, *chan_to_auxinfo*, *dset*, *annots*, *catpats*, *with_sseg*, *with_kpts*, *dims*, *newstyle*, *gray*, *rng*)

Renders demo annoations on top of a demo background

kwcoco.demo.toydata_video.**render_background**(*img*, *rng*, *gray*, *bg_intensity*, *bg_scale*)

kwcoco.demo.toydata_video.**false_color**(*twochan*)

TODO: the function ensure_false_color will eventually be ported to kwimage use that instead.

kwcoco.demo.toydata_video.**random_multi_object_path**(*num_objects*, *num_frames*, *rng=None*, *max_speed=0.01*)

kwcoco.demo.toydata_video.**random_path**(*num*, *degree=1*, *dimension=2*, *rng=None*, *mode='boid'*)

Create a random path using a somem ethod curve.

Parameters

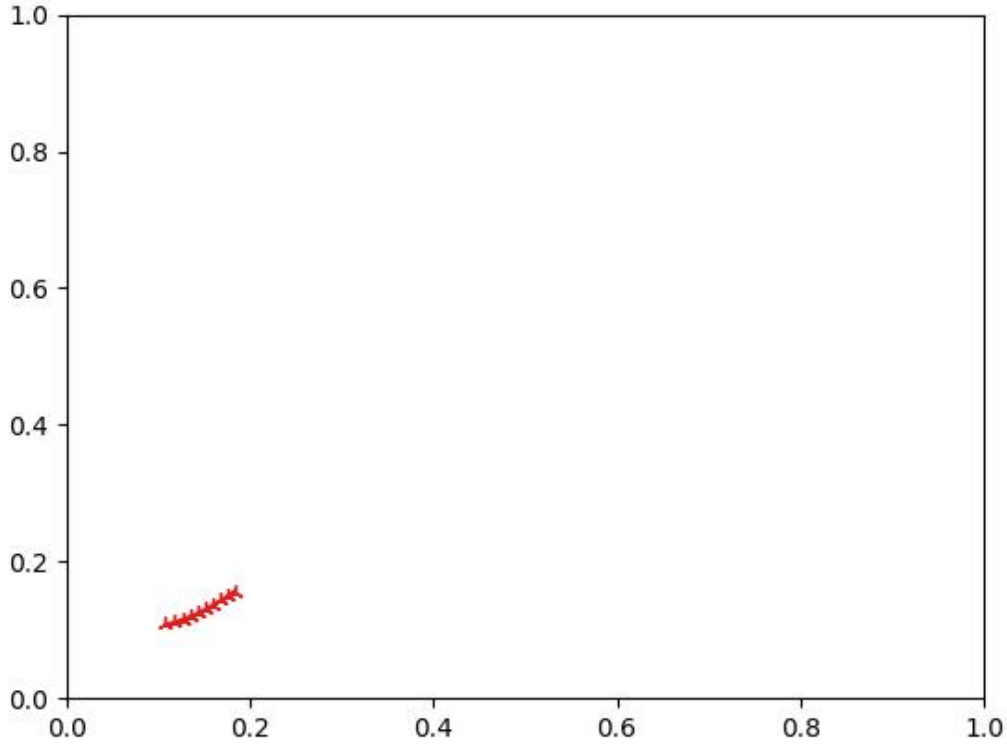
- **num** (*int*) – number of points in the path
- **degree** (*int, default=1*) – degree of curvieness of the path
- **dimension** (*int, default=2*) – number of spatial dimensions
- **mode** (*str*) – can be boid, walk, or bezier
- **rng** (*RandomState, default=None*) – seed

References

https://github.com/dhermes/bezier

Example

```
>>> from kwcoco.demo.toydata_video import *  # NOQA
>>> num = 10
>>> dimension = 2
>>> degree = 3
>>> rng = None
>>> path = random_path(num, degree, dimension, rng, mode='boid')
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> plt = kwplot.autoplt()
>>> kwplot.multi_plot(xdata=path[:, 0], ydata=path[:, 1], fnum=1, doclf=1, xlim=(0,␣
↪1), ylim=(0, 1))
>>> kwplot.show_if_requested()
```



Example

```
>>> # xdoctest: +REQUIRES(--3d)
>>> # xdoctest: +REQUIRES(module:bezier)
>>> import kwarray
>>> import kwplot
>>> plt = kwplot.autoplt()
>>> #
>>> num= num_frames = 100
>>> rng = kwarray.ensure_rng(0)
>>> #
>>> from kwcoco.demo.toydata_video import *  # NOQA
>>> paths = []
>>> paths.append(random_path(num, degree=3, dimension=3, mode='bezier'))
>>> paths.append(random_path(num, degree=2, dimension=3, mode='bezier'))
>>> paths.append(random_path(num, degree=4, dimension=3, mode='bezier'))
>>> #
>>> from mpl_toolkits.mplot3d import Axes3D  # NOQA
>>> ax = plt.gca(projection='3d')
>>> ax.cla()
>>> #
>>> for path in paths:
>>>     time = np.arange(len(path))
```

(continues on next page)

(continued from previous page)

```
>>>     ax.plot(time, path.T[0] * 1, path.T[1] * 1, 'o-')
>>> ax.set_xlim(0, num_frames)
>>> ax.set_ylim(-.01, 1.01)
>>> ax.set_zlim(-.01, 1.01)
>>> ax.set_xlabel('x')
>>> ax.set_ylabel('y')
>>> ax.set_zlabel('z')
```

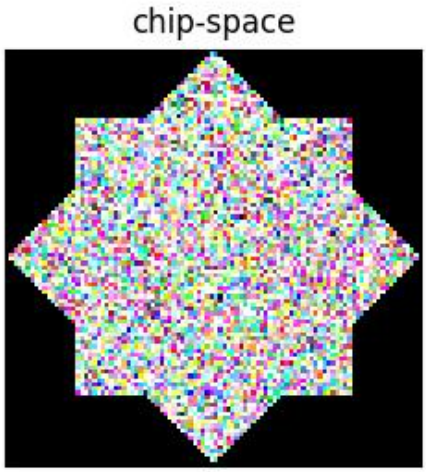
2.1.1.3.1.6 kwcoco.demo.toypatterns module

class kwcoco.demo.toypatterns.**CategoryPatterns**(*categories=None*, *fg_scale=0.5*, *fg_intensity=0.9*, *rng=None*)

Bases: object

Example

```
>>> from kwcoco.demo.toypatterns import *  # NOQA
>>> self = CategoryPatterns.coerce()
>>> chip = np.zeros((100, 100, 3))
>>> offset = (20, 10)
>>> dims = (160, 140)
>>> info = self.random_category(chip, offset, dims)
>>> print('info = {}'.format(ub.repr2(info, nl=1)))
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(info['data'], pnum=(1, 2, 1), fnum=1, title='chip-space')
>>> kpts = kwimage.Points._from_coco(info['keypoints'])
>>> kpts.translate(-np.array(offset)).draw(radius=3)
>>> #####
>>> mask = kwimage.Mask.coerce(info['segmentation'])
>>> kwplot.imshow(mask.to_c_mask().data, pnum=(1, 2, 2), fnum=1, title='img-space')
>>> kpts.draw(radius=3)
>>> kwplot.show_if_requested()
```



classmethod coerce(*data=None*, ***kwargs*)

Construct category patterns from either defaults or only with specific categories. Can accept either an existig category pattern object, a list of known catnames, or mscoco category dictionaries.

Example

```
>>> data = ['superstar']
>>> self = CategoryPatterns.coerce(data)
```

index(*name*)

get(*index*, *default=NoParam*)

random_category(*chip*, *xy_offset=None*, *dims=None*, *newstyle=True*, *size=None*)

Example

```
>>> from kwcoco.demo.toypatterns import *  # NOQA
>>> self = CategoryPatterns.coerce(['superstar'])
>>> chip = np.random.rand(64, 64)
>>> info = self.random_category(chip)
```

render_category(*cname*, *chip*, *xy_offset=None*, *dims=None*, *newstyle=True*, *size=None*)

Example

```
>>> from kwcoco.demo.toypatterns import *  # NOQA
>>> self = CategoryPatterns.coerce(['superstar'])
>>> chip = np.random.rand(64, 64)
>>> info = self.render_category('superstar', chip, newstyle=True)
>>> print('info = {}'.format(ub.repr2(info, nl=-1)))
>>> info = self.render_category('superstar', chip, newstyle=False)
>>> print('info = {}'.format(ub.repr2(info, nl=-1)))
```

Example

```
>>> from kwcoco.demo.toypatterns import *  # NOQA
>>> self = CategoryPatterns.coerce(['superstar'])
>>> chip = None
>>> dims = (64, 64)
>>> info = self.render_category('superstar', chip, newstyle=True, dims=dims,␣
↪size=dims)
>>> print('info = {}'.format(ub.repr2(info, nl=-1)))
```

kwcoco.demo.toypatterns.**star**(*a*, *dtype=<class 'numpy.uint8'>*)

Generates a star shaped structuring element.

Much faster than skimage.morphology version

class kwcoco.demo.toypatterns.**Rasters**

Bases: object

static superstar()

test data patch

static eff()

test data patch

2.1.1.3.2 Module contents

2.1.1.4 kwcoco.examples package

2.1.1.4.1 Submodules

2.1.1.4.1.1 kwcoco.examples.bench_large_hyperspectral module

2.1.1.4.1.2 kwcoco.examples.draw_gt_and_predicted_boxes module

kwcoco.examples.draw_gt_and_predicted_boxes.**draw_true_and_pred_boxes**(*true_fpath*, *pred_fpath*, *gid*, *viz_fpath*)

How do you generally visualize gt and predicted bounding boxes together?

Example

```
>>> import kwcoco
>>> import ubelt as ub
>>> from os.path import join
>>> from kwcoco.demo.perterb import perterb_coco
>>> # Create a working directory
>>> dpath = ub.ensure_app_cache_dir('kwcoco/examples/draw_true_and_pred_boxes')
>>> # Lets setup some dummy true data
>>> true_dset = kwcoco.CocoDataset.demo('shapes2')
>>> true_dset.fpath = join(dpath, 'true_dset.kwcoco.json')
>>> true_dset.dump(true_dset.fpath, newlines=True)
>>> # Lets setup some dummy predicted data
>>> pred_dset = perterb_coco(true_dset, box_noise=100, rng=421)
>>> pred_dset.fpath = join(dpath, 'pred_dset.kwcoco.json')
>>> pred_dset.dump(pred_dset.fpath, newlines=True)
>>> #
>>> # We now have our true and predicted data, lets visualize
>>> true_fpath = true_dset.fpath
>>> pred_fpath = pred_dset.fpath
>>> print('dpath = {!r}'.format(dpath))
>>> print('true_fpath = {!r}'.format(true_fpath))
>>> print('pred_fpath = {!r}'.format(pred_fpath))
>>> # Lets choose an image id to visualize and a path to write to
>>> gid = 1
>>> viz_fpath = join(dpath, 'viz_{}.jpg'.format(gid))
>>> # The answer to the question is in the logic of this function
>>> draw_true_and_pred_boxes(true_fpath, pred_fpath, gid, viz_fpath)
```

2.1.1.4.1.3 kwcoco.examples.faq module

These are answers to the questions: How do I?

kwcoco.examples.faq.**get_images_with_videoid**()

Q: How would you recommend querying a kwcoco file to get all of the images associated with a video id?

kwcoco.examples.faq.**get_all_channels_in_dataset**()

Q. After I load a kwcoco.json into a kwcoco_dset, is there a nice way to query what channels are available for the input imagery? It looks like I can iterate over .imgs and build my own set, but maybe theres a built in way

A. The better way is to use the CocoImage API.

kwcoco.examples.faq.**whats_the_difference_between_Images_and_CocoImage**()

Q. What is the difference between *kwcoco.Images* and *kwcoco.CocoImage*.

It's a little weird because it grew organically, but the "vectorized API" calls like *.images*, *.annots*, *.videos* are methods for handling multiple dictionaries at once. E.g. *dset.images().lookup('width')* returns a list of the width attribute for each dictionary that particular *Images* object is indexing (which by default is all of them, although you can filter).

In contrast the *kwcoco.CocoImage* object is for working with exactly one image. The important thing to note is if you have a CocoImage *coco_img = dset.coco_image(1)* The *coco_img.img* attribute is exactly the underlying dictionary. So you are never too far away from it.

Similarly for the *Images* objects: *dset.images().objs* returns a list of all of the image dictionaries in that set.

2.1.1.4.1.4 kwcoco.examples.getting_started_existing_dataset module

kwcoco.examples.getting_started_existing_dataset.**getting_started_existing_dataset**()

If you want to start using the Python API. Just open IPython and try:

kwcoco.examples.getting_started_existing_dataset.**the_core_dataset_backend**()

kwcoco.examples.getting_started_existing_dataset.**demo_vectorize_interface**()

```
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('vidshapes2')
>>> #
>>> aids = [1, 2, 3, 4]
>>> annots = dset.annots(aids)
...
>>> print('annots = {!r}'.format(annots))
annots = <Annots(num=4) at ...>
```

```
>>> annots.lookup('bbox')
[[346.5, 335.2, 33.2, 99.4],
 [344.5, 327.7, 48.8, 111.1],
 [548.0, 154.4, 57.2, 62.1],
 [548.7, 151.0, 59.4, 80.5]]
```

```
>>> gids = annots.lookup('image_id')
>>> print('gids = {!r}'.format(gids))
gids = [1, 2, 1, 2]
```

```
>>> images = dset.images(gids)
>>> list(zip(images.lookup('width'), images.lookup('height')))
[(600, 600), (600, 600), (600, 600), (600, 600)]
```

2.1.1.4.1.5 kwcoco.examples.loading_multispectral_data module

kwcoco.examples.loading_multispectral_data.**demo_load_msi_data**()

2.1.1.4.1.6 kwcoco.examples.modification_example module

kwcoco.examples.modification_example.**dataset_modification_example_via_copy**()

Say you are given a dataset as input and you need to add your own annotation "predictions" to it. You could copy the existing dataset, remove all the annotations, and then add your new annotations.

kwcoco.examples.modification_example.**dataset_modification_example_via_construction**()

Alternatively you can make a new dataset and copy over categories / images as needed

2.1.1.4.1.7 kwcoco.examples.simple_kwcoco_torch_dataset module

This example demonstrates how to use kwcoco to write a very simple torch dataset. This assumes the dataset will be single-image RGB inputs. This file is intended to talk the reader through what we are doing and why.

This example aims for clairity over being concise. There are APIs exposed by kwcoco (and its sister module ndsampler) that can perform the same tasks more efficiently and with fewer lines of code.

If you run the doctest, it will produce a visualization that shows the images with boxes drawn on it, running it multiple times will let you see the augmentations. This can be done with the following command:

xdoctest -m kwcoco.examples.simple_kwcoco_torch_dataset KWCocoSimpleTorchDataset –show

Or just copy the doctest into IPython and run it.

class kwcoco.examples.simple_kwcoco_torch_dataset.**KWCocoSimpleTorchDataset**(*coco_dset*, *input_dims=None*, *antialias=False*, *rng=None*)

Bases: object

A simple torch dataloader where each image is considered a single item.

Parameters

- **coco_dset** (*kwcoco.CocoDataset | str*) – something coercable to a kwcoco dataset, this could either be a *kwcoco.CocoDataset* object, a path to a kwcoco manifest on disk, or a special toydata code. See kwcoco.CocoDataset.coerce() for more details.
- **input_dims** (*Tuple[int, int]*) – These are the (height, width) dimensions that the image will be resized to.
- **antialias** (*bool, default=False*) – If true, we will antialias before downsampling.
- **rng** (*RandomState | int | None*) – an existing random number generator or a random seed to produce deterministic augmentations.

Example

```
>>> # xdoctest: +REQUIRES(module:torch)
>>> from kwcoco.examples.simple_kwcoco_torch_dataset import *  # NOQA
>>> import kwcoco
>>> coco_dset = kwcoco.CocoDataset.demo('shapes8')
>>> input_dims = (384, 384)
>>> self = torch_dset = KWCocoSimpleTorchDataset(coco_dset, input_dims=input_dims)
>>> index = len(self) // 2
>>> item = self[index]
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.figure(doclf=True, fnum=1)
>>> kwplot.autompl()
>>> canvas = item['inputs']['rgb'].numpy().transpose(1, 2, 0)
>>> # Construct kwimage objects for batch item visualization
>>> dets = kwimage.Detections(
>>>     boxes=kwimage.Boxes(item['labels']['cxywh'], 'cxywh'),
>>>     class_idxs=item['labels']['class_idxs'],
>>>     classes=self.classes,
>>> ).numpy()
>>> # Overlay annotations on the image
>>> canvas = dets.draw_on(canvas)
>>> kwplot.imshow(canvas)
>>> kwplot.show_if_requested()
```

2.1.1.4.1.8 kwcoco.examples.vectorized_interface module

kwcoco.examples.vectorized_interface.**demo_vectorized_interface**()

This demonstrates how to use the kwcoco vectorized interface for images / categories / annotations.

2.1.1.4.2 Module contents

2.1.1.5 kwcoco.metrics package

2.1.1.5.1 Submodules

2.1.1.5.1.1 kwcoco.metrics.assignment module

Todo:

- **[ ] _fast_pdist_priority: Look at absolute difference in sibling entropy**
  when deciding whether to go up or down in the tree.
- **[ ] medschool applications true-pred matching (applicant proposing) fast**
  algorithm.
- **[ ] Maybe looping over truth rather than pred is faster? but it makes you**
  have to combine pred score / ious, which is weird.
- **[x] preallocate ndarray and use hstack to build confusion vectors?**

  - doesn't help

- **[ ] relevant classes / classes / classes-of-interest we care about needs**
  to be a first class member of detection metrics.
- **[ ] Add parameter that allows one prediction to "match" to more than one**
  truth object. (example: we have a duck detector problem and all the ducks in a row are annotated as separate object, and we only care about getting the group)

2.1.1.5.1.2 kwcoco.metrics.clf_report module

kwcoco.metrics.clf_report.**classification_report**(*y_true*, *y_pred*, *target_names=None*, *sample_weight=None*, *verbose=False*, *remove_unsupported=False*, *log=None*, *ascii_only=False*)

Computes a classification report which is a collection of various metrics commonly used to evaulate classification quality. This can handle binary and multiclass settings.

Note that this function does not accept probabilities or scores and must instead act on final decisions. See ovr_classification_report for a probability based report function using a one-vs-rest strategy.

This emulates the bm(cm) Matlab script [MatlabBM] written by David Powers that is used for computing bookmaker, markedness, and various other scores and is based on the paper [PowersMetrics].

References

Parameters

- **y_true** (*array*) – true labels for each item
- **y_pred** (*array*) – predicted labels for each item
- **target_names** (*List*) – mapping from label to category name
- **sample_weight** (*ndarray*) – weight for each item
- **verbose** (*False*) – print if True
- **log** (*callable*) – print or logging function
- **remove_unsupported** (*bool, default=False*) – removes categories that have no support.
- **ascii_only** (*bool, default=False*) – if True dont use unicode characters. if the environ ASCII_ONLY is present this is forced to True and cannot be undone.

Example

```
>>> # xdoctest: +IGNORE_WANT
>>> # xdoctest: +REQUIRES(module:sklearn)
>>> # xdoctest: +REQUIRES(module:pandas)
>>> y_true = [1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3]
>>> y_pred = [1, 2, 1, 3, 1, 2, 2, 3, 2, 2, 3, 3, 2, 3, 3, 3, 1, 3]
>>> target_names = None
>>> sample_weight = None
>>> report = classification_report(y_true, y_pred, verbose=0, ascii_only=1)
```

(continues on next page)

(continued from previous page)

```
>>> print(report['confusion'])
pred  1  2  3  r
real
1     3  1  1   5
2     0  4  1   5
3     1  1  6   8
p     4  6  8  18
>>> print(report['metrics'])
metric    precision  recall    fpr  markedness  bookmaker    mcc  support
class
1            0.7500  0.6000 0.0769      0.6071     0.5231 0.5635        5
2            0.6667  0.8000 0.1538      0.5833     0.6462 0.6139        5
3            0.7500  0.7500 0.2000      0.5500     0.5500 0.5500        8
combined     0.7269  0.7222 0.1530      0.5751     0.5761 0.5758       18
```

Example

```
>>> # xdoctest: +IGNORE_WANT
>>> # xdoctest: +REQUIRES(module:sklearn)
>>> # xdoctest: +REQUIRES(module:pandas)
>>> from kwcoco.metrics.clf_report import *  # NOQA
>>> y_true = [1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3]
>>> y_pred = [1, 2, 1, 3, 1, 2, 2, 3, 2, 2, 3, 3, 2, 3, 3, 3, 1, 3]
>>> target_names = None
>>> sample_weight = None
>>> logs = []
>>> report = classification_report(y_true, y_pred, verbose=1, ascii_only=True,␣
↪log=logs.append)
>>> print('\n'.join(logs))
```

kwcoco.metrics.clf_report.**ovr_classification_report**(*mc_y_true*, *mc_probs*, *target_names=None*, *sample_weight=None*, *metrics=None*, *verbose=0*, *remove_unsupported=False*, *log=None*)

One-vs-rest classification report

Parameters

- **mc_y_true** (*ndarray[Any, Int]*) – multiclass truth labels (integer label format). Shape [N].
- **mc_probs** (*ndarray*) – multiclass probabilities for each class. Shape [N x C].
- **target_names** (*Dict[int, str]*) – mapping from int label to string name
- **sample_weight** (*ndarray*) – weight for each item. Shape [N].
- **metrics** (*List[str]*) – names of metrics to compute

Example

```
>>> # xdoctest: +IGNORE_WANT
>>> # xdoctest: +REQUIRES(module:sklearn)
>>> # xdoctest: +REQUIRES(module:pandas)
>>> from kwcoco.metrics.clf_report import *  # NOQA
>>> y_true = [1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 0, 0, 0, 0, 0, 0, 0, 0]
>>> y_probs = np.random.rand(len(y_true), max(y_true) + 1)
>>> target_names = None
>>> sample_weight = None
>>> verbose = True
>>> report = ovr_classification_report(y_true, y_probs)
>>> print(report['ave'])
auc      0.6541
ap       0.6824
kappa    0.0963
mcc      0.1002
brier    0.2214
dtype: float64
>>> print(report['ovr'])
     auc     ap  kappa    mcc  brier  support  weight
0 0.6062 0.6161 0.0526 0.0598 0.2608        8  0.4444
1 0.5846 0.6014 0.0000 0.0000 0.2195        5  0.2778
2 0.8000 0.8693 0.2623 0.2652 0.1602        5  0.2778
```

2.1.1.5.1.3 kwcoco.metrics.confusion_measures module

Classes that store accumulated confusion measures (usually derived from confusion vectors).

For each chosen threshold value:

- thresholds[i] - the i-th threshold value

The primary data we manipulate are arrays of "confusion" counts, i.e.

- tp_count[i] - true positives at the i-th threshold
- fp_count[i] - false positives at the i-th threshold
- fn_count[i] - false negatives at the i-th threshold
- tn_count[i] - true negatives at the i-th threshold

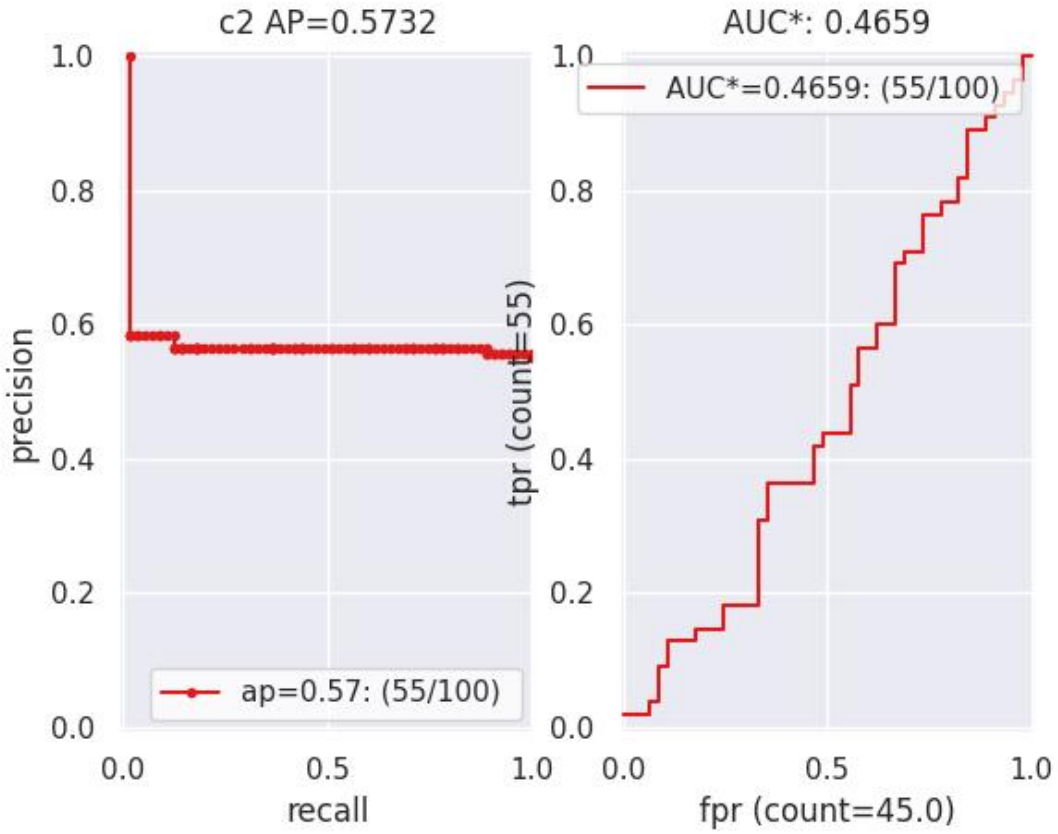
class kwcoco.metrics.confusion_measures.**Measures**(*info*)

Bases: NiceRepr, *DictProxy*

Holds accumulated confusion counts, and derived measures

Example

```
>>> from kwcoco.metrics.confusion_vectors import BinaryConfusionVectors  # NOQA
>>> binvecs = BinaryConfusionVectors.demo(n=100, p_error=0.5)
>>> self = binvecs.measures()
>>> print('self = {!r}'.format(self))
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> self.draw(doclf=True)
>>> self.draw(key='pr',  pnum=(1, 2, 1))
>>> self.draw(key='roc', pnum=(1, 2, 2))
>>> kwplot.show_if_requested()
```



property catname

reconstruct()

classmethod from_json(*state*)

summary()

maximized_thresholds()

Returns thresholds that maximize metrics.

counts()

draw(*key=None*, *prefix=''*, ***kw*)

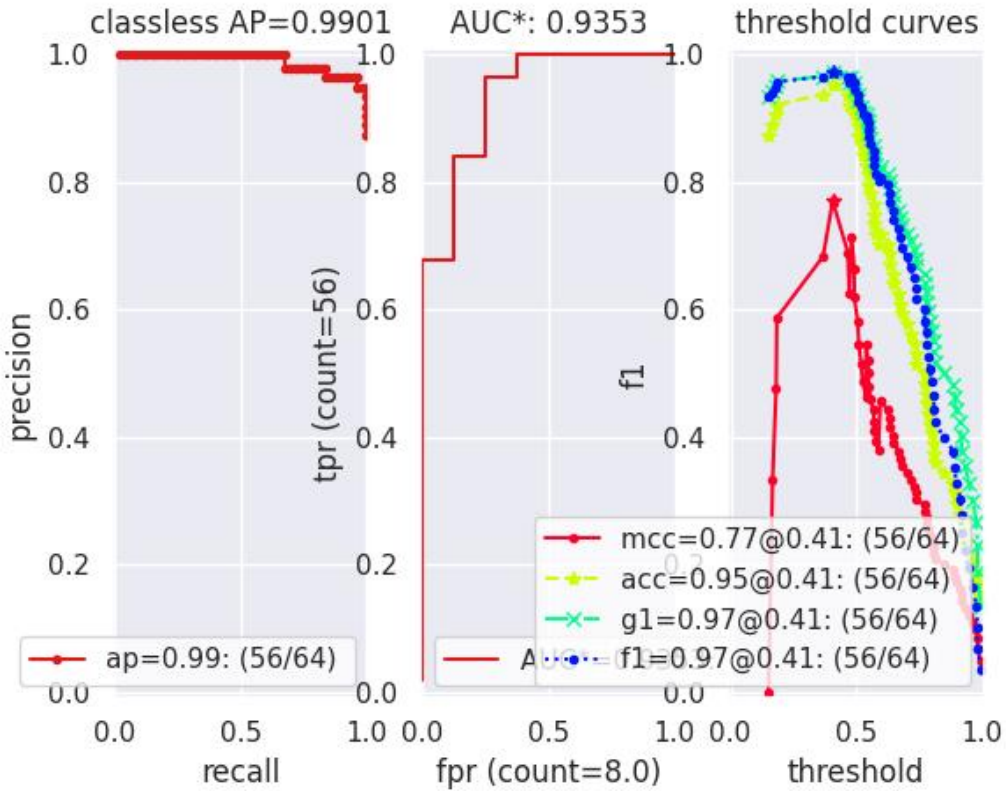
Example

```
>>> # xdoctest: +REQUIRES(module:kwplot)
>>> # xdoctest: +REQUIRES(module:pandas)
>>> from kwcoco.metrics.confusion_vectors import ConfusionVectors  # NOQA
>>> cfsn_vecs = ConfusionVectors.demo()
>>> ovr_cfsn = cfsn_vecs.binarize_ovr(keyby='name')
>>> self = ovr_cfsn.measures()['perclass']
>>> self.draw('mcc', doclf=True, fnum=1)
>>> self.draw('pr', doclf=1, fnum=2)
>>> self.draw('roc', doclf=1, fnum=3)
```

summary_plot(*fnum=1*, *title=''*, *subplots='auto'*)

Example

```
>>> from kwcoco.metrics.confusion_measures import *  # NOQA
>>> from kwcoco.metrics.confusion_vectors import ConfusionVectors  # NOQA
>>> cfsn_vecs = ConfusionVectors.demo(n=3, p_error=0.5)
>>> binvecs = cfsn_vecs.binarize_classless()
>>> self = binvecs.measures()
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> self.summary_plot()
>>> kwplot.show_if_requested()
```



classmethod demo(***kwargs*)

Create a demo Measures object for testing / demos

Parameters

****kwargs** – passed to `BinaryConfusionVectors.demo()`. some valid keys are: n, rng, p_rue, p_error, p_miss.

classmethod combine(*tocombine*, *precision=None*, *growth=None*, *thresh_bins=None*)

Combine binary confusion metrics

Parameters

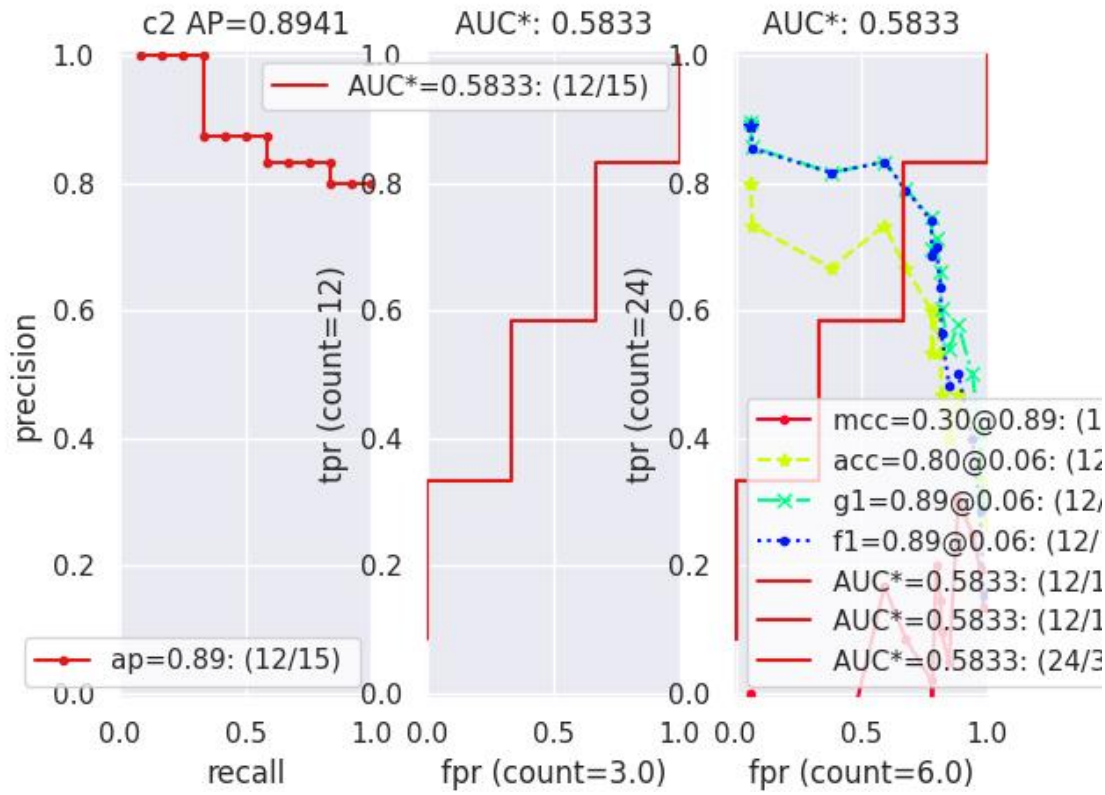
- **tocombine** (*List[Measures]*) – a list of measures to combine into one
- **precision** (*int | None*) – If specified rounds thresholds to this precision which can prevent a RAM explosion when combining a large number of measures. However, this is a lossy operation and will impact the underlying scores. NOTE: use `growth` instead.
- **growth** (*int | None*) – if specified this limits how much the resulting measures are allowed to grow by. If None, growth is unlimited. Otherwise, if growth is 'max', the growth is limited to the maximum length of an input. We might make this more numerical in the future.
- **thresh_bins** (*int*) – Force this many threshold bins.

Returns

kwcoco.metrics.confusion_measures.Measures

Example

```
>>> from kwcoco.metrics.confusion_measures import *  # NOQA
>>> measures1 = Measures.demo(n=15)
>>> measures2 = measures1
>>> tocombine = [measures1, measures2]
>>> new_measures = Measures.combine(tocombine)
>>> new_measures.reconstruct()
>>> print('new_measures = {!r}'.format(new_measures))
>>> print('measures1 = {!r}'.format(measures1))
>>> print('measures2 = {!r}'.format(measures2))
>>> print(ub.repr2(measures1.__json__(), nl=1, sort=0))
>>> print(ub.repr2(measures2.__json__(), nl=1, sort=0))
>>> print(ub.repr2(new_measures.__json__(), nl=1, sort=0))
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.figure(fnum=1)
>>> new_measures.summary_plot()
>>> measures1.summary_plot()
>>> measures1.draw('roc')
>>> measures2.draw('roc')
>>> new_measures.draw('roc')
```



Example

```
>>> # Demonstrate issues that can arrise from choosing a precision
>>> # that is too low  when combining metrics. Breakpoints
>>> # between different metrics can get muddled, but choosing a
>>> # precision that is too high can overwhelm memory.
>>> from kwcoco.metrics.confusion_measures import *  # NOQA
>>> base = ub.map_vals(np.asarray, {
>>>     'tp_count':   [ 1,  1,  2,  2,  2,  2,  3],
>>>     'fp_count':   [ 0,  1,  1,  2,  3,  4,  5],
>>>     'fn_count':   [ 1,  1,  0,  0,  0,  0,  0],
>>>     'tn_count':   [ 5,  4,  4,  3,  2,  1,  0],
>>>     'thresholds': [.0, .0, .0, .0, .0, .0, .0],
>>> })
>>> # Make tiny offsets to thresholds
>>> rng = kwarray.ensure_rng(0)
>>> n = len(base['thresholds'])
>>> offsets = [
>>>     sorted(rng.rand(n) * 10 ** -rng.randint(4, 7))[::-1]
>>>     for _ in range(20)
>>> ]
>>> tocombine = []
>>> for offset in offsets:
>>>     base_n = base.copy()
>>>     base_n['thresholds'] += offset
>>>     measures_n = Measures(base_n).reconstruct()
>>>     tocombine.append(measures_n)
>>> for precision in [6, 5, 2]:
>>>     combo = Measures.combine(tocombine, precision=precision).reconstruct()
>>>     print('precision = {!r}'.format(precision))
>>>     print('combo = {}'.format(ub.repr2(combo, nl=1)))
>>>     print('num_thresholds = {}'.format(len(combo['thresholds'])))
>>> for growth in [None, 'max', 'log', 'root', 'half']:
>>>     combo = Measures.combine(tocombine, growth=growth).reconstruct()
>>>     print('growth = {!r}'.format(growth))
>>>     print('combo = {}'.format(ub.repr2(combo, nl=1)))
>>>     print('num_thresholds = {}'.format(len(combo['thresholds'])))
>>>     #print(combo.counts().pandas())
```

Example

```
>>> # Test case: combining a single measures should leave it unchanged
>>> from kwcoco.metrics.confusion_measures import *  # NOQA
>>> measures = Measures.demo(n=40, p_true=0.2, p_error=0.4, p_miss=0.6)
>>> df1 = measures.counts().pandas().fillna(0)
>>> print(df1)
>>> tocombine = [measures]
>>> combo = Measures.combine(tocombine)
>>> df2 = combo.counts().pandas().fillna(0)
>>> print(df2)
>>> assert np.allclose(df1, df2)
```

```
>>> combo = Measures.combine(tocombine, thresh_bins=2)
>>> df3 = combo.counts().pandas().fillna(0)
>>> print(df3)
```

```
>>> # I am NOT sure if this is correct or not
>>> thresh_bins = 20
>>> combo = Measures.combine(tocombine, thresh_bins=thresh_bins)
>>> df4 = combo.counts().pandas().fillna(0)
>>> print(df4)
```

```
>>> combo = Measures.combine(tocombine, thresh_bins=np.linspace(0, 1, 20))
>>> df4 = combo.counts().pandas().fillna(0)
>>> print(df4)
```

assert np.allclose(combo[‘thresholds’], measures[‘thresholds’]) assert np.allclose(combo[‘fp_count’], measures[‘fp_count’]) assert np.allclose(combo[‘tp_count’], measures[‘tp_count’]) assert np.allclose(combo[‘tp_count’], measures[‘tp_count’])

globals().update(xdev.get_func_kwargs(Measures.combine))

Example

```
>>> # Test degenerate case
>>> from kwcoco.metrics.confusion_measures import *  # NOQA
>>> tocombine = [
>>>     {'fn_count': [0.0], 'fp_count': [359980.0], 'thresholds': [0.0], 'tn_
↪count': [0.0], 'tp_count': [7747.0]},
>>>     {'fn_count': [0.0], 'fp_count': [360849.0], 'thresholds': [0.0], 'tn_
↪count': [0.0], 'tp_count': [424.0]},
>>>     {'fn_count': [0.0], 'fp_count': [367003.0], 'thresholds': [0.0], 'tn_
↪count': [0.0], 'tp_count': [991.0]},
>>>     {'fn_count': [0.0], 'fp_count': [367976.0], 'thresholds': [0.0], 'tn_
↪count': [0.0], 'tp_count': [1017.0]},
>>>     {'fn_count': [0.0], 'fp_count': [676338.0], 'thresholds': [0.0], 'tn_
↪count': [0.0], 'tp_count': [7067.0]},
>>>     {'fn_count': [0.0], 'fp_count': [676348.0], 'thresholds': [0.0], 'tn_
↪count': [0.0], 'tp_count': [7406.0]},
>>>     {'fn_count': [0.0], 'fp_count': [676626.0], 'thresholds': [0.0], 'tn_
↪count': [0.0], 'tp_count': [7858.0]},
>>>     {'fn_count': [0.0], 'fp_count': [676693.0], 'thresholds': [0.0], 'tn_
↪count': [0.0], 'tp_count': [10969.0]},
>>>     {'fn_count': [0.0], 'fp_count': [677269.0], 'thresholds': [0.0], 'tn_
↪count': [0.0], 'tp_count': [11188.0]},
>>>     {'fn_count': [0.0], 'fp_count': [677331.0], 'thresholds': [0.0], 'tn_
↪count': [0.0], 'tp_count': [11734.0]},
>>>     {'fn_count': [0.0], 'fp_count': [677395.0], 'thresholds': [0.0], 'tn_
↪count': [0.0], 'tp_count': [11556.0]},
>>>     {'fn_count': [0.0], 'fp_count': [677418.0], 'thresholds': [0.0], 'tn_
↪count': [0.0], 'tp_count': [11621.0]},
>>>     {'fn_count': [0.0], 'fp_count': [677422.0], 'thresholds': [0.0], 'tn_
↪count': [0.0], 'tp_count': [11424.0]},
>>>     {'fn_count': [0.0], 'fp_count': [677648.0], 'thresholds': [0.0], 'tn_
```

(continues on next page)

(continued from previous page)

```
↪count': [0.0], 'tp_count': [9804.0]},
>>>     {'fn_count': [0.0], 'fp_count': [677826.0], 'thresholds': [0.0], 'tn_
↪count': [0.0], 'tp_count': [2470.0]},
>>>     {'fn_count': [0.0], 'fp_count': [677834.0], 'thresholds': [0.0], 'tn_
↪count': [0.0], 'tp_count': [2470.0]},
>>>     {'fn_count': [0.0], 'fp_count': [677835.0], 'thresholds': [0.0], 'tn_
↪count': [0.0], 'tp_count': [2470.0]},
>>>     {'fn_count': [11123.0, 0.0], 'fp_count': [0.0, 676754.0], 'thresholds':␣
↪[0.0002442002442002442, 0.0], 'tn_count': [676754.0, 0.0], 'tp_count': [2.0,␣
↪11125.0]},
>>>     {'fn_count': [7738.0, 0.0], 'fp_count': [0.0, 676466.0], 'thresholds':␣
↪[0.0002442002442002442, 0.0], 'tn_count': [676466.0, 0.0], 'tp_count': [0.0,␣
↪7738.0]},
>>>     {'fn_count': [8653.0, 0.0], 'fp_count': [0.0, 676341.0], 'thresholds':␣
↪[0.0002442002442002442, 0.0], 'tn_count': [676341.0, 0.0], 'tp_count': [0.0,␣
↪8653.0]},
>>> ]
>>> thresh_bins = np.linspace(0, 1, 4)
>>> combo = Measures.combine(tocombine, thresh_bins=thresh_bins).reconstruct()
>>> print('tocombine = {}'.format(ub.repr2(tocombine, nl=2)))
>>> print('thresh_bins = {!r}'.format(thresh_bins))
>>> print(ub.repr2(combo.__json__(), nl=1))
>>> for thresh_bins in [4096, 1]:
>>>     combo = Measures.combine(tocombine, thresh_bins=thresh_bins).
↪reconstruct()
>>>     print('thresh_bins = {!r}'.format(thresh_bins))
>>>     print('combo = {}'.format(ub.repr2(combo, nl=1)))
>>>     print('num_thresholds = {}'.format(len(combo['thresholds'])))
>>> for precision in [6, 5, 2]:
>>>     combo = Measures.combine(tocombine, precision=precision).reconstruct()
>>>     print('precision = {!r}'.format(precision))
>>>     print('combo = {}'.format(ub.repr2(combo, nl=1)))
>>>     print('num_thresholds = {}'.format(len(combo['thresholds'])))
>>> for growth in [None, 'max', 'log', 'root', 'half']:
>>>     combo = Measures.combine(tocombine, growth=growth).reconstruct()
>>>     print('growth = {!r}'.format(growth))
>>>     print('combo = {}'.format(ub.repr2(combo, nl=1)))
>>>     print('num_thresholds = {}'.format(len(combo['thresholds'])))
```

kwcoco.metrics.confusion_measures.**reversable_diff**(*arr*, *assume_sorted=1*, *reverse=False*)

Does a reversable array difference operation.

This will be used to find positions where accumulation happened in confusion count array.

class kwcoco.metrics.confusion_measures.**PerClass_Measures**(*cx_to_info*)

Bases: NiceRepr, *DictProxy*

summary()

classmethod from_json(*state*)

draw(*key='mcc'*, *prefix=''*, ***kw*)

Example

```
>>> # xdoctest: +REQUIRES(module:kwplot)
>>> from kwcoco.metrics.confusion_vectors import ConfusionVectors  # NOQA
>>> cfsn_vecs = ConfusionVectors.demo()
>>> ovr_cfsn = cfsn_vecs.binarize_ovr(keyby='name')
>>> self = ovr_cfsn.measures()['perclass']
>>> self.draw('mcc', doclf=True, fnum=1)
>>> self.draw('pr', doclf=1, fnum=2)
>>> self.draw('roc', doclf=1, fnum=3)
```

draw_roc(*prefix=''*, ***kw*)

draw_pr(*prefix=''*, ***kw*)

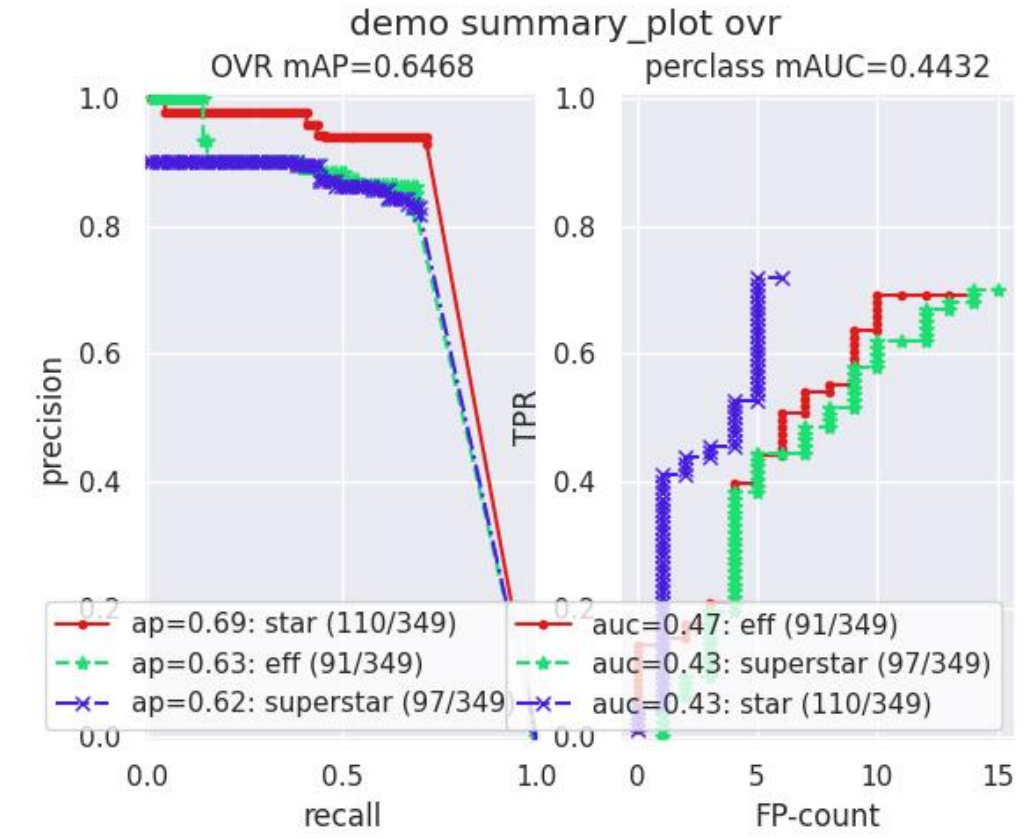
summary_plot(*fnum=1*, *title=''*, *subplots='auto'*)

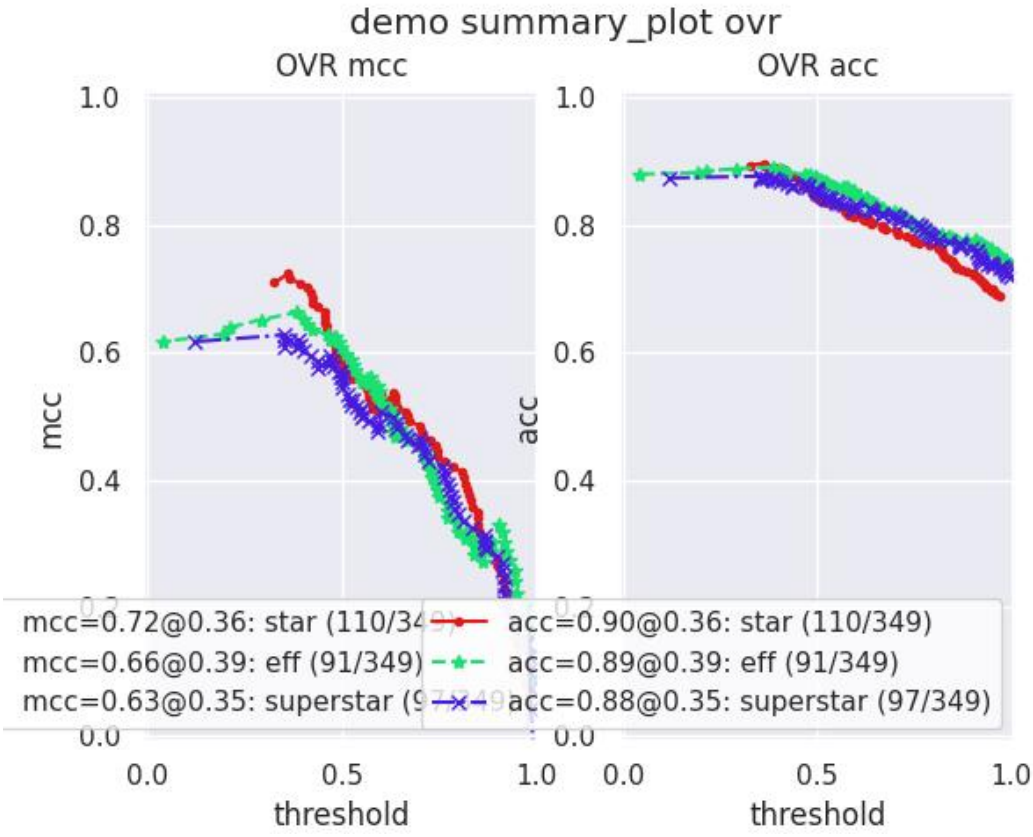
CommandLine

```
python ~/code/kwcoco/kwcoco/metrics/confusion_measures.py PerClass_Measures.
↪summary_plot --show
```

Example

```
>>> from kwcoco.metrics.confusion_measures import *  # NOQA
>>> from kwcoco.metrics.detect_metrics import DetectionMetrics
>>> dmet = DetectionMetrics.demo(
>>>     n_fp=(0, 1), n_fn=(0, 3), nimgs=32, nboxes=(0, 32),
>>>     classes=3, rng=0, newstyle=1, box_noise=0.7, cls_noise=0.2, score_
↪noise=0.3, with_probs=False)
>>> cfsn_vecs = dmet.confusion_vectors()
>>> ovr_cfsn = cfsn_vecs.binarize_ovr(keyby='name', ignore_classes=['vector',
↪'raster'])
>>> self = ovr_cfsn.measures()['perclass']
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> import seaborn as sns
>>> sns.set()
>>> self.summary_plot(title='demo summary_plot ovr', subplots=['pr', 'roc'])
>>> kwplot.show_if_requested()
>>> self.summary_plot(title='demo summary_plot ovr', subplots=['mcc', 'acc'],␣
↪fnum=2)
```


demo summary_plot ovr
OVR mAP=0.6468
perclass mAUC=0.4432
precision
recall
TPR
FP-count
ap=0.69: star (110/349)
ap=0.63: eff (91/349)
ap=0.62: superstar (97/349
auc=0.47: eff (91/349)
auc=0.43: superstar (97/349)
auc=0.43: star (110/349)


demo summary_plot ovr
OVR mcc
OVR acc
mcc
acc
threshold
threshold
mcc=0.72@0.36: star (110/34
mcc=0.66@0.39: eff (91/349)
mcc=0.63@0.35: superstar (9
acc=0.90@0.36: star (110/349)
acc=0.89@0.39: eff (91/349)
acc=0.88@0.35: superstar (97/349)

class kwcoco.metrics.confusion_measures.**MeasureCombiner**(*precision=None*, *growth=None*, *thresh_bins=None*)

Bases: object

Helper to iteravely combine binary measures generated by some process

Example

```
>>> from kwcoco.metrics.confusion_measures import *  # NOQA
>>> from kwcoco.metrics.confusion_vectors import BinaryConfusionVectors
>>> rng = kwarray.ensure_rng(0)
>>> bin_combiner = MeasureCombiner(growth='max')
>>> for _ in range(80):
>>>     bin_cfsn_vecs = BinaryConfusionVectors.demo(n=rng.randint(40, 50), rng=rng,␣
↪p_true=0.2, p_error=0.4, p_miss=0.6)
>>>     bin_measures = bin_cfsn_vecs.measures()
>>>     bin_combiner.submit(bin_measures)
>>> combined = bin_combiner.finalize()
>>> print('combined = {!r}'.format(combined))
```

property queue_size

submit(*other*)

combine()

finalize()

class kwcoco.metrics.confusion_measures.**OneVersusRestMeasureCombiner**(*precision=None*, *growth=None*, *thresh_bins=None*)

Bases: object

Helper to iteravely combine ovr measures generated by some process

Example

```
>>> from kwcoco.metrics.confusion_measures import *  # NOQA
>>> from kwcoco.metrics.confusion_vectors import OneVsRestConfusionVectors
>>> rng = kwarray.ensure_rng(0)
>>> ovr_combiner = OneVersusRestMeasureCombiner(growth='max')
>>> for _ in range(80):
>>>     ovr_cfsn_vecs = OneVsRestConfusionVectors.demo()
>>>     ovr_measures = ovr_cfsn_vecs.measures()
>>>     ovr_combiner.submit(ovr_measures)
>>> combined = ovr_combiner.finalize()
>>> print('combined = {!r}'.format(combined))
```

submit(*other*)

combine()

finalize()

kwcoco.metrics.confusion_measures.**populate_info**(*info*)

Given raw accumulated confusion counts, populated secondary measures like AP, AUC, F1, MCC, etc..

2.1.1.5.1.4 kwcoco.metrics.confusion_vectors module

Classes that store raw confusion vectors, which can be accumulated into confusion measures.

class kwcoco.metrics.confusion_vectors.**ConfusionVectors**(*data*, *classes*, *probs=None*)

Bases: NiceRepr

Stores information used to construct a confusion matrix. This includes corresponding vectors of predicted labels, true labels, sample weights, etc...

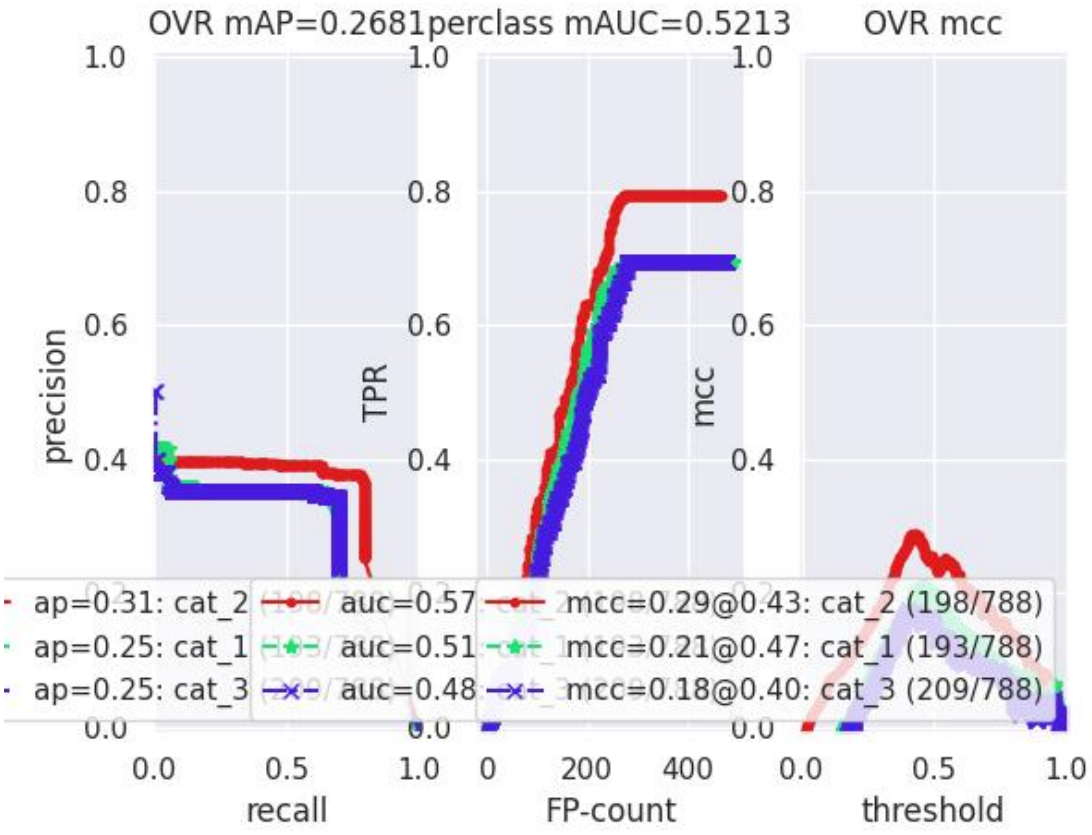
Variables

- **data** (*kwarray.DataFrameArray*) – should at least have keys true, pred, weight
- **classes** (*Sequence | CategoryTree*) – list of category names or category graph
- **probs** (*ndarray, optional*) – probabilities for each class

Example

```
>>> # xdoctest: IGNORE_WANT
>>> # xdoctest: +REQUIRES(module:pandas)
>>> from kwcoco.metrics import DetectionMetrics
>>> dmet = DetectionMetrics.demo(
>>>     nimgs=10, nboxes=(0, 10), n_fp=(0, 1), classes=3)
>>> cfsn_vecs = dmet.confusion_vectors()
>>> print(cfsn_vecs.data._pandas())
     pred  true   score  weight     iou  txs  pxs  gid
0       2     2 10.0000  1.0000  1.0000    0    4    0
1       2     2  7.5025  1.0000  1.0000    1    3    0
2       1     1  5.0050  1.0000  1.0000    2    2    0
3       3    -1  2.5075  1.0000 -1.0000   -1    1    0
4       2    -1  0.0100  1.0000 -1.0000   -1    0    0
5      -1     2  0.0000  1.0000 -1.0000    3   -1    0
6      -1     2  0.0000  1.0000 -1.0000    4   -1    0
7       2     2 10.0000  1.0000  1.0000    0    5    1
8       2     2  8.0020  1.0000  1.0000    1    4    1
9       1     1  6.0040  1.0000  1.0000    2    3    1
..    ...   ...     ...     ...     ...  ...  ...  ...
62     -1     2  0.0000  1.0000 -1.0000    7   -1    7
63     -1     3  0.0000  1.0000 -1.0000    8   -1    7
64     -1     1  0.0000  1.0000 -1.0000    9   -1    7
65      1    -1 10.0000  1.0000 -1.0000   -1    0    8
66      1     1  0.0100  1.0000  1.0000    0    1    8
67      3    -1 10.0000  1.0000 -1.0000   -1    3    9
68      2     2  6.6700  1.0000  1.0000    0    2    9
69      2     2  3.3400  1.0000  1.0000    1    1    9
70      3    -1  0.0100  1.0000 -1.0000   -1    0    9
71     -1     2  0.0000  1.0000 -1.0000    2   -1    9
```

```
>>> # xdoctest: +REQUIRES(--show)
>>> # xdoctest: +REQUIRES(module:pandas)
>>> import kwplot
>>> kwplot.autompl()
>>> from kwcoco.metrics.confusion_vectors import ConfusionVectors
>>> cfsn_vecs = ConfusionVectors.demo(
>>>     nimgs=128, nboxes=(0, 10), n_fp=(0, 3), n_fn=(0, 3), classes=3)
>>> cx_to_binvecs = cfsn_vecs.binarize_ovr()
>>> measures = cx_to_binvecs.measures()['perclass']
>>> print('measures = {!r}'.format(measures))
measures = <PerClass_Measures({
    'cat_1': <Measures({'ap': 0.227, 'auc': 0.507, 'catname': cat_1, 'max_f1': f1=0.
↪45@0.47, 'nsupport': 788.000})>,
    'cat_2': <Measures({'ap': 0.288, 'auc': 0.572, 'catname': cat_2, 'max_f1': f1=0.
↪51@0.43, 'nsupport': 788.000})>,
    'cat_3': <Measures({'ap': 0.225, 'auc': 0.484, 'catname': cat_3, 'max_f1': f1=0.
↪46@0.40, 'nsupport': 788.000})>,
}) at 0x7facf77bdfd0>
>>> kwplot.figure(fnum=1, doclf=True)
>>> measures.draw(key='pr', fnum=1, pnum=(1, 3, 1))
>>> measures.draw(key='roc', fnum=1, pnum=(1, 3, 2))
>>> measures.draw(key='mcc', fnum=1, pnum=(1, 3, 3))
...
```



classmethod from_json(*state*)

classmethod demo(***kw*)

Parameters
****kwargs** – See *kwcoco.metrics.DetectionMetrics.demo()*

Returns
ConfusionVectors

Example

```
>>> cfsn_vecs = ConfusionVectors.demo()
>>> print('cfsn_vecs = {!r}'.format(cfsn_vecs))
>>> cx_to_binvecs = cfsn_vecs.binarize_ovr()
>>> print('cx_to_binvecs = {!r}'.format(cx_to_binvecs))
```

classmethod from_arrays(*true*, *pred=None*, *score=None*, *weight=None*, *probs=None*, *classes=None*)

Construct confusion vector data structure from component arrays

Example

```
>>> # xdoctest: +REQUIRES(module:pandas)
>>> import kwarray
>>> classes = ['person', 'vehicle', 'object']
>>> rng = kwarray.ensure_rng(0)
>>> true = (rng.rand(10) * len(classes)).astype(int)
>>> probs = rng.rand(len(true), len(classes))
>>> cfsn_vecs = ConfusionVectors.from_arrays(true=true, probs=probs,␣
↪classes=classes)
>>> cfsn_vecs.confusion_matrix()
pred     person  vehicle  object
real
person        0        0       0
vehicle       2        4       1
object        2        1       0
```

confusion_matrix(*compress=False*)

Builds a confusion matrix from the confusion vectors.

Parameters
compress (*bool, default=False*) – if True removes rows / columns with no entries

Returns

cm
[the labeled confusion matrix]

(Note: we should write a efficient replacement for
this use case. #remove_pandas)

Return type
pd.DataFrame

CommandLine

```
xdoctest -m kwcoco.metrics.confusion_vectors ConfusionVectors.confusion_matrix
```

Example

```
>>> # xdoctest: +REQUIRES(module:pandas)
>>> from kwcoco.metrics import DetectionMetrics
>>> dmet = DetectionMetrics.demo(
>>>     nimgs=10, nboxes=(0, 10), n_fp=(0, 1), n_fn=(0, 1), classes=3, cls_
↪noise=.2)
>>> cfsn_vecs = dmet.confusion_vectors()
>>> cm = cfsn_vecs.confusion_matrix()
...
>>> print(cm.to_string(float_format=lambda x: '%.2f' % x))
pred        background  cat_1  cat_2  cat_3
real
background        0.00   1.00   2.00   3.00
cat_1             3.00  12.00   0.00   0.00
cat_2             3.00   0.00  14.00   0.00
cat_3             2.00   0.00   0.00  17.00
```

coarsen(*cxs*)

Creates a coarsened set of vectors

Returns
ConfusionVectors

binarize_classless(*negative_classes=None*)

Creates a binary representation useful for measuring the performance of detectors. It is assumed that scores of "positive" classes should be high and "negative" clases should be low.

Parameters
negative_classes (*List[str | int]*) – list of negative class names or idxs, by default chooses any class with a true class index of -1. These classes should ideally have low scores.

Returns
BinaryConfusionVectors

Note: The "classlessness" of this depends on the compat="all" argument being used when constructing confusion vectors, otherwise it becomes something like a macro-average because the class information was used in deciding which true and predicted boxes were allowed to match.

Example

```
>>> from kwcoco.metrics import DetectionMetrics
>>> dmet = DetectionMetrics.demo(
>>>     nimgs=10, nboxes=(0, 10), n_fp=(0, 1), n_fn=(0, 1), classes=3)
>>> cfsn_vecs = dmet.confusion_vectors()
>>> class_idxs = list(dmet.classes.node_to_idx.values())
>>> binvecs = cfsn_vecs.binarize_classless()
```

binarize_ovr(*mode=1*, *keyby='name'*, *ignore_classes={'ignore'}*, *approx=False*)

Transforms cfsn_vecs into one-vs-rest BinaryConfusionVectors for each category.

Parameters

- **mode** (*int, default=1*) – 0 for heirarchy aware or 1 for voc like. MODE 0 IS PROBABLY BROKEN
- **keyby** (*int | str*) – can be cx or name
- **ignore_classes** (*Set[str]*) – category names to ignore
- **approx** (*bool, default=0*) – if True try and approximate missing scores otherwise assume they are irrecoverable and use -inf

Returns

which behaves like
Dict[int, BinaryConfusionVectors]: cx_to_binvecs

Return type
OneVsRestConfusionVectors

Example

```
>>> from kwcoco.metrics.confusion_vectors import *  # NOQA
>>> cfsn_vecs = ConfusionVectors.demo()
>>> print('cfsn_vecs = {!r}'.format(cfsn_vecs))
>>> catname_to_binvecs = cfsn_vecs.binarize_ovr(keyby='name')
>>> print('catname_to_binvecs = {!r}'.format(catname_to_binvecs))
```

cfsn_vecs.data.pandas() catname_to_binvecs.cx_to_binvecs[‘class_1’].data.pandas()

Note:

---

classification_report(*verbose=0*)

Build a classification report with various metrics.

Example

```
>>> # xdoctest: +REQUIRES(module:pandas)
>>> from kwcoco.metrics.confusion_vectors import *  # NOQA
>>> cfsn_vecs = ConfusionVectors.demo()
>>> report = cfsn_vecs.classification_report(verbose=1)
```

class kwcoco.metrics.confusion_vectors.**OneVsRestConfusionVectors**(*cx_to_binvecs*, *classes*)

Bases: NiceRepr

Container for multiple one-vs-rest binary confusion vectors

Variables

- **cx_to_binvecs** –
- **classes** –

Example

```
>>> from kwcoco.metrics import DetectionMetrics
>>> dmet = DetectionMetrics.demo(
>>>     nimgs=10, nboxes=(0, 10), n_fp=(0, 1), classes=3)
>>> cfsn_vecs = dmet.confusion_vectors()
>>> self = cfsn_vecs.binarize_ovr(keyby='name')
>>> print('self = {!r}'.format(self))
```

classmethod demo()

Parameters
****kwargs** – See *kwcoco.metrics.DetectionMetrics.demo()*

Returns
ConfusionVectors

keys()

measures(*stabalize_thresh=7*, *fp_cutoff=None*, *monotonic_ppv=True*, *ap_method='pycocotools'*)

Creates binary confusion measures for every one-versus-rest category.

Parameters

- **stabalize_thresh** (*int, default=7*) – if fewer than this many data points inserts dummy stabilization data so curves can still be drawn.
- **fp_cutoff** (*int, default=None*) – maximum number of false positives in the truncated roc curves. None is equivalent to `float('inf')`
- **monotonic_ppv** (*bool, default=True*) – if True ensures that precision is always increasing as recall decreases. This is done in pycocotools scoring, but I'm not sure its a good idea.

SeeAlso:
BinaryConfusionVectors.measures()

Example

```
>>> self = OneVsRestConfusionVectors.demo()
>>> thresh_result = self.measures()['perclass']
```

ovr_classification_report()

class kwcoco.metrics.confusion_vectors.**BinaryConfusionVectors**(*data*, *cx=None*, *classes=None*)

Bases: NiceRepr

Stores information about a binary classification problem. This is always with respect to a specific class, which is given by *cx* and *classes*.

The *data* DataFrameArray must contain

is_true - if the row is an instance of class *classes[cx]* *pred_score* - the predicted probability of class *classes[cx]*, and *weight* - sample weight of the example

Example

```
>>> from kwcoco.metrics.confusion_vectors import *  # NOQA
>>> self = BinaryConfusionVectors.demo(n=10)
>>> print('self = {!r}'.format(self))
>>> print('measures = {}'.format(ub.repr2(self.measures())))
```

```
>>> self = BinaryConfusionVectors.demo(n=0)
>>> print('measures = {}'.format(ub.repr2(self.measures())))
```

```
>>> self = BinaryConfusionVectors.demo(n=1)
>>> print('measures = {}'.format(ub.repr2(self.measures())))
```

```
>>> self = BinaryConfusionVectors.demo(n=2)
>>> print('measures = {}'.format(ub.repr2(self.measures())))
```

classmethod demo(*n=10*, *p_true=0.5*, *p_error=0.2*, *p_miss=0.0*, *rng=None*)

Create random data for tests

Parameters

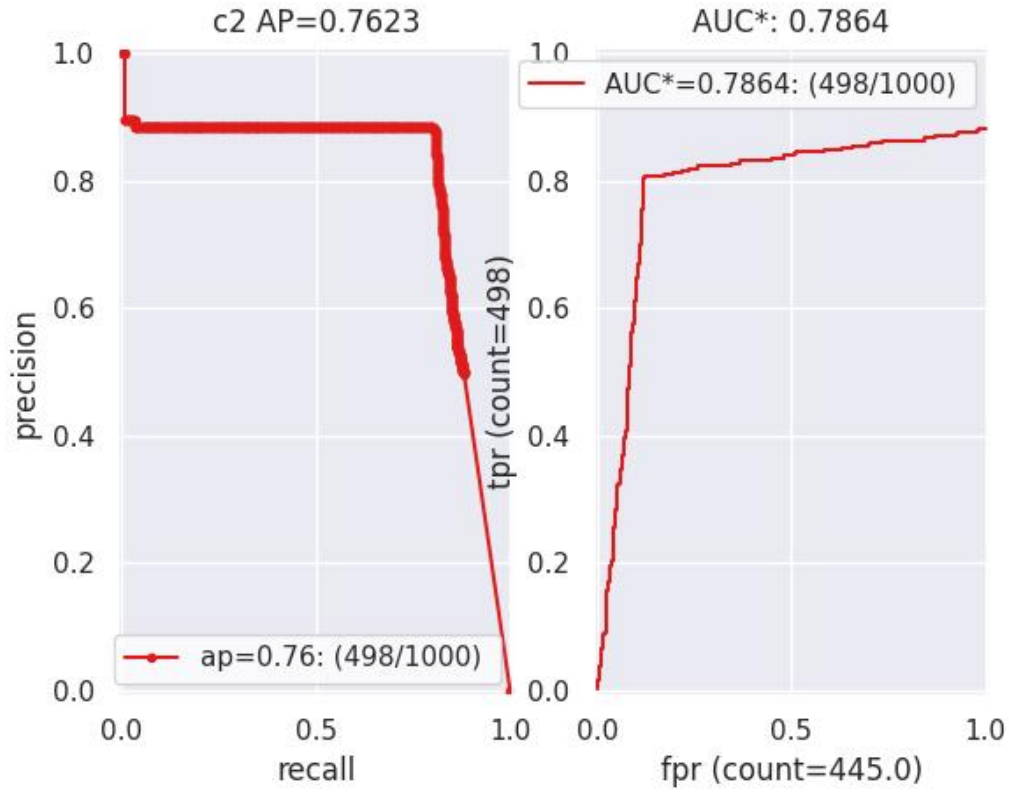
- **n** (*int*) – number of rows
- **p_true** (*float*) – fraction of real positive cases
- **p_error** (*float*) – probability of making a recoverable mistake
- **p_miss** (*float*) – probability of making a unrecoverable mistake
- **rng** (*int | RandomState*) – random seed / state

Returns

BinaryConfusionVectors

Example

```
>>> from kwcoco.metrics.confusion_vectors import *  # NOQA
>>> cfsn = BinaryConfusionVectors.demo(n=1000, p_error=0.1, p_miss=0.1)
>>> measures = cfsn.measures()
>>> print('measures = {}'.format(ub.repr2(measures, nl=1)))
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.figure(fnum=1, pnum=(1, 2, 1))
>>> measures.draw('pr')
>>> kwplot.figure(fnum=1, pnum=(1, 2, 2))
>>> measures.draw('roc')
```



property catname

measures(*stabalize_thresh=7*, *fp_cutoff=None*, *monotonic_ppv=True*, *ap_method='pycocotools'*)

Get statistics (F1, G1, MCC) versus thresholds

Parameters

- **stabalize_thresh** (*int, default=7*) – if fewer than this many data points inserts dummy stabalization data so curves can still be drawn.
- **fp_cutoff** (*int, default=None*) – maximum number of false positives in the truncated roc curves. `None` is equivalent to `float('inf')`
- **monotonic_ppv** (*bool, default=True*) – if True ensures that precision is always increasing as recall decreases. This is done in pycocotools scoring, but I'm not sure its a good idea.

Example

```
>>> from kwcoco.metrics.confusion_vectors import *  # NOQA
>>> self = BinaryConfusionVectors.demo(n=0)
>>> print('measures = {}'.format(ub.repr2(self.measures())))
>>> self = BinaryConfusionVectors.demo(n=1, p_true=0.5, p_error=0.5)
>>> print('measures = {}'.format(ub.repr2(self.measures())))
>>> self = BinaryConfusionVectors.demo(n=3, p_true=0.5, p_error=0.5)
>>> print('measures = {}'.format(ub.repr2(self.measures())))
```

```
>>> self = BinaryConfusionVectors.demo(n=100, p_true=0.5, p_error=0.5, p_miss=0.
↪3)
>>> print('measures = {}'.format(ub.repr2(self.measures())))
>>> print('measures = {}'.format(ub.repr2(ub.odict(self.measures()))))
```

References

https://en.wikipedia.org/wiki/Confusion_matrix https://en.wikipedia.org/wiki/Precision_and_recall https://en.wikipedia.org/wiki/Matthews_correlation_coefficient

draw_distribution()

2.1.1.5.1.5 kwcoco.metrics.detect_metrics module

class kwcoco.metrics.detect_metrics.**DetectionMetrics**(*classes=None*)

Bases: NiceRepr

Object that computes associations between detections and can convert them into sklearn-compatible representations for scoring.

Variables

- **gid_to_true_dets** (*Dict*) – maps image ids to truth
- **gid_to_pred_dets** (*Dict*) – maps image ids to predictions
- **classes** (CategoryTree) – category coder

Example

```
>>> dmet = DetectionMetrics.demo(
>>>     nimgs=100, nboxes=(0, 3), n_fp=(0, 1), classes=8, score_noise=0.9,␣
↪hacked=False)
>>> print(dmet.score_kwcoco(bias=0, compat='mutex', prioritize='iou')['mAP'])
...
>>> # NOTE: IN GENERAL NETHARN AND VOC ARE NOT THE SAME
>>> print(dmet.score_voc(bias=0)['mAP'])
0.8582...
>>> #print(dmet.score_coco()['mAP'])
```

clear()

classmethod from_coco(*true_coco*, *pred_coco*, *gids=None*, *verbose=0*)

Create detection metrics from two coco files representing the truth and predictions.

Parameters

- **true_coco** (*kwcoco.CocoDataset*)
- **pred_coco** (*kwcoco.CocoDataset*)

Example

```
>>> import kwcoco
>>> from kwcoco.demo.perterb import perterb_coco
>>> true_coco = kwcoco.CocoDataset.demo('shapes')
>>> perterbkw = dict(box_noise=0.5, cls_noise=0.5, score_noise=0.5)
>>> pred_coco = perterb_coco(true_coco, **perterbkw)
>>> self = DetectionMetrics.from_coco(true_coco, pred_coco)
>>> self.score_voc()
```

add_predictions(*pred_dets*, *imgname=None*, *gid=None*)

Register/Add predicted detections for an image

Parameters

- **pred_dets** (*kwimage.Detections*) – predicted detections
- **imgname** (*str*) – a unique string to identify the image
- **gid** (*int | None*) – the integer image id if known

add_truth(*true_dets*, *imgname=None*, *gid=None*)

Register/Add groundtruth detections for an image

Parameters

- **true_dets** (*kwimage.Detections*) – groundtruth
- **imgname** (*str*) – a unique string to identify the image
- **gid** (*int | None*) – the integer image id if known

true_detections(*gid*)

gets Detections representation for groundtruth in an image

pred_detections(*gid*)

gets Detections representation for predictions in an image

confusion_vectors(*iou_thresh=0.5*, *bias=0*, *gids=None*, *compat='mutex'*, *prioritize='iou'*, *ignore_classes='ignore'*, *background_class=NoParam*, *verbose='auto'*, *workers=0*, *track_probs='try'*, *max_dets=None*)

Assigns predicted boxes to the true boxes so we can transform the detection problem into a classification problem for scoring.

Parameters

- **iou_thresh** (*float | List[float], default=0.5*) – bounding box overlap iou threshold required for assignment if a list, then return type is a dict
- **bias** (*float, default=0.0*) – for computing bounding box overlap, either 1 or 0

- **gids** (*List[int], default=None*) – which subset of images ids to compute confusion metrics on. If not specified all images are used.
- **compat** (*str, default='all'*) – can be ('ancestors' | 'mutex' | 'all'). determines which pred boxes are allowed to match which true boxes. If 'mutex', then pred boxes can only match true boxes of the same class. If 'ancestors', then pred boxes can match true boxes that match or have a coarser label. If 'all', then any pred can match any true, regardless of its category label.
- **prioritize** (*str, default='iou'*) – can be ('iou' | 'class' | 'correct') determines which box to assign to if mutiple true boxes overlap a predicted box. if prioritize is iou, then the true box with maximum iou (above iou_thresh) will be chosen. If prioritize is class, then it will prefer matching a compatible class above a higher iou. If prioritize is correct, then ancestors of the true class are preferred over descendents of the true class, over unreleated classes.
- **ignore_classes** (*set | str, default={'ignore'}*) – class names indicating ignore regions
- **background_class** (*str, default=ub.NoParam*) – Name of the background class. If unspecified we try to determine it with heuristics. A value of None means there is no background class.
- **verbose** (*int | str, default='auto'*) – verbosity flag. In auto mode, verbose=1 if len(gids) > 1000.
- **workers** (*int, default=0*) – number of parallel assignment processes
- **track_probs** (*str, default='try'*) – can be 'try', 'force', or False. if truthy, we assume probabilities for multiple classes are available.

Returns

kwcoco.metrics.confusion_vectors.ConfusionVectors | Dict[float, kwcoco.metrics.confusion_vectors.ConfusionVectors]

Example

```
>>> dmet = DetectionMetrics.demo(nimgs=30, classes=3,
>>>                              nboxes=10, n_fp=3, box_noise=10,
>>>                              with_probs=False)
>>> iou_to_cfsn = dmet.confusion_vectors(iou_thresh=[0.3, 0.5, 0.9])
>>> for t, cfsn in iou_to_cfsn.items():
>>>     print('t = {!r}'.format(t))
...     print(cfsn.binarize_ovr().measures())
...     print(cfsn.binarize_classless().measures())
```

score_kwant(*iou_thresh=0.5*)

Scores the detections using kwant

score_kwcoco(*iou_thresh=0.5*, *bias=0*, *gids=None*, *compat='all'*, *prioritize='iou'*)

our scoring method

score_voc(*iou_thresh=0.5*, *bias=1*, *method='voc2012'*, *gids=None*, *ignore_classes='ignore'*)

score using voc method

Example

```
>>> dmet = DetectionMetrics.demo(
>>>     nimgs=100, nboxes=(0, 3), n_fp=(0, 1), classes=8,
>>>     score_noise=.5)
>>> print(dmet.score_voc()['mAP'])
0.9399...
```

score_pycocotools(*with_evaler=False*, *with_confusion=False*, *verbose=0*, *iou_thresholds=None*)

score using ms-coco method

Returns
dictionary with pct info

Return type
Dict

Example

```
>>> # xdoctest: +REQUIRES(module:pycocotools)
>>> from kwcoco.metrics.detect_metrics import *
>>> dmet = DetectionMetrics.demo(
>>>     nimgs=10, nboxes=(0, 3), n_fn=(0, 1), n_fp=(0, 1), classes=8, with_
↪probs=False)
>>> pct_info = dmet.score_pycocotools(verbose=1,
>>>                                   with_evaler=True,
>>>                                   with_confusion=True,
>>>                                   iou_thresholds=[0.5, 0.9])
>>> evaler = pct_info['evaler']
>>> iou_to_cfsn_vecs = pct_info['iou_to_cfsn_vecs']
>>> for iou_thresh in iou_to_cfsn_vecs.keys():
>>>     print('iou_thresh = {!r}'.format(iou_thresh))
>>>     cfsn_vecs = iou_to_cfsn_vecs[iou_thresh]
>>>     ovr_measures = cfsn_vecs.binarize_ovr().measures()
>>>     print('ovr_measures = {}'.format(ub.repr2(ovr_measures, nl=1,␣
↪precision=4)))
```

Note: by default pycocotools computes average precision as the literal average of computed precisions at 101 uniformly spaced recall thresholds.

pycocoutils seems to only allow predictions with the same category as the truth to match those truth objects. This should be the same as calling dmet.confusion_vectors with compat = mutex

pycocoutils does not take into account the fact that each box often has a score for each category.

pycocoutils will be incorrect if any annotation has an id of 0

a major difference in the way kwcoco scores versus pycocoutils is the calculation of AP. The assignment between truth and predicted detections produces similar enough results. Given our confusion vectors we use the scikit-learn definition of AP, whereas pycocoutils seems to compute precision and recall — more or less correctly — but then it resamples the precision at various specified recall thresholds (in the *accumulate* function, specifically how *pr* is resampled into the *q* array). This can lead to a large difference in reported scores.

pycocoutils also smooths out the precision such that it is monotonic decreasing, which might not be the best idea.

pycocotools area ranges are inclusive on both ends, that means the "small" and "medium" truth selections do overlap somewhat.

---

score_coco(*with_evaler=False*, *with_confusion=False*, *verbose=0*, *iou_thresholds=None*)

score using ms-coco method

Returns
dictionary with pct info

Return type
Dict

Example

```
>>> # xdoctest: +REQUIRES(module:pycocotools)
>>> from kwcoco.metrics.detect_metrics import *
>>> dmet = DetectionMetrics.demo(
>>>     nimgs=10, nboxes=(0, 3), n_fn=(0, 1), n_fp=(0, 1), classes=8, with_
↪probs=False)
>>> pct_info = dmet.score_pycocotools(verbose=1,
>>>                                   with_evaler=True,
>>>                                   with_confusion=True,
>>>                                   iou_thresholds=[0.5, 0.9])
>>> evaler = pct_info['evaler']
>>> iou_to_cfsn_vecs = pct_info['iou_to_cfsn_vecs']
>>> for iou_thresh in iou_to_cfsn_vecs.keys():
>>>     print('iou_thresh = {!r}'.format(iou_thresh))
>>>     cfsn_vecs = iou_to_cfsn_vecs[iou_thresh]
>>>     ovr_measures = cfsn_vecs.binarize_ovr().measures()
>>>     print('ovr_measures = {}'.format(ub.repr2(ovr_measures, nl=1,␣
↪precision=4)))
```

Note: by default pycocotools computes average precision as the literal average of computed precisions at 101 uniformly spaced recall thresholds.

pycocoutils seems to only allow predictions with the same category as the truth to match those truth objects. This should be the same as calling dmet.confusion_vectors with compat = mutex

pycocoutils does not take into account the fact that each box often has a score for each category.

pycocoutils will be incorrect if any annotation has an id of 0

a major difference in the way kwcoco scores versus pycocoutils is the calculation of AP. The assignment between truth and predicted detections produces similar enough results. Given our confusion vectors we use the scikit-learn definition of AP, whereas pycocoutils seems to compute precision and recall — more or less correctly — but then it resamples the precision at various specified recall thresholds (in the *accumulate* function, specifically how *pr* is resampled into the *q* array). This can lead to a large difference in reported scores.

pycocoutils also smooths out the precision such that it is monotonic decreasing, which might not be the best idea.

pycocotools area ranges are inclusive on both ends, that means the “small” and “medium” truth selections do overlap somewhat.

---

classmethod demo(***kwargs*)

Creates random true boxes and predicted boxes that have some noisy offset from the truth.

Kwargs:

classes (int):
class list or the number of foreground classes. Defaults to 1.

nimgs (int): number of images in the coco datasts. Defaults to 1.

nboxes (int): boxes per image. Defaults to 1.

n_fp (int): number of false positives. Defaults to 0.

n_fn (int):
number of false negatives. Defaults to 0.

box_noise (float):
std of a normal distribution used to perterb both box location and box size. Defaults to 0.

cls_noise (float):
probability that a class label will change. Must be within 0 and 1. Defaults to 0.

anchors (ndarray):
used to create random boxes. Defaults to None.

null_pred (bool):
if True, predicted classes are returned as null, which means only localization scoring is suitable. Defaults to 0.

with_probs (bool):
if True, includes per-class probabilities with predictions Defaults to 1.

CommandLine

```
xdoctest -m kwcoco.metrics.detect_metrics DetectionMetrics.demo:2 --show
```

Example

```
>>> kwargs = {}
>>> # Seed the RNG
>>> kwargs['rng'] = 0
>>> # Size parameters determine how big the data is
>>> kwargs['nimgs'] = 5
>>> kwargs['nboxes'] = 7
>>> kwargs['classes'] = 11
>>> # Noise parameters perterb predictions further from the truth
>>> kwargs['n_fp'] = 3
>>> kwargs['box_noise'] = 0.1
>>> kwargs['cls_noise'] = 0.5
>>> dmet = DetectionMetrics.demo(**kwargs)
>>> print('dmet.classes = {}'.format(dmet.classes))
```

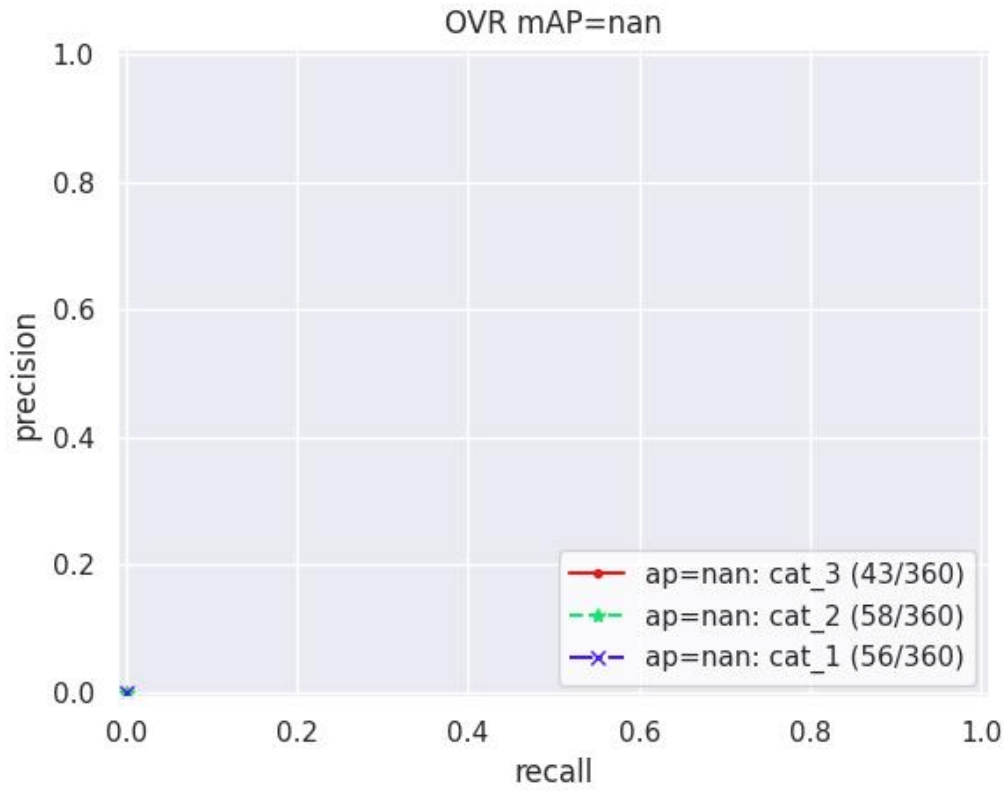
(continues on next page)

(continued from previous page)

```
dmet.classes = <CategoryTree(nNodes=12, maxDepth=3, maxBreadth=4...)>
>>> # Can grab kwimage.Detection object for any image
>>> print(dmet.true_detections(gid=0))
<Detections(4)>
>>> print(dmet.pred_detections(gid=0))
<Detections(7)>
```

Example

```
>>> # Test case with null predicted categories
>>> dmet = DetectionMetrics.demo(nimgs=30, null_pred=1, classes=3,
>>>                              nboxes=10, n_fp=3, box_noise=0.1,
>>>                              with_probs=False)
>>> dmet.gid_to_pred_dets[0].data
>>> dmet.gid_to_true_dets[0].data
>>> cfsn_vecs = dmet.confusion_vectors()
>>> binvecs_ovr = cfsn_vecs.binarize_ovr()
>>> binvecs_per = cfsn_vecs.binarize_classless()
>>> measures_per = binvecs_per.measures()
>>> measures_ovr = binvecs_ovr.measures()
>>> print('measures_per = {!r}'.format(measures_per))
>>> print('measures_ovr = {!r}'.format(measures_ovr))
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> measures_ovr['perclass'].draw(key='pr', fnum=2)
```



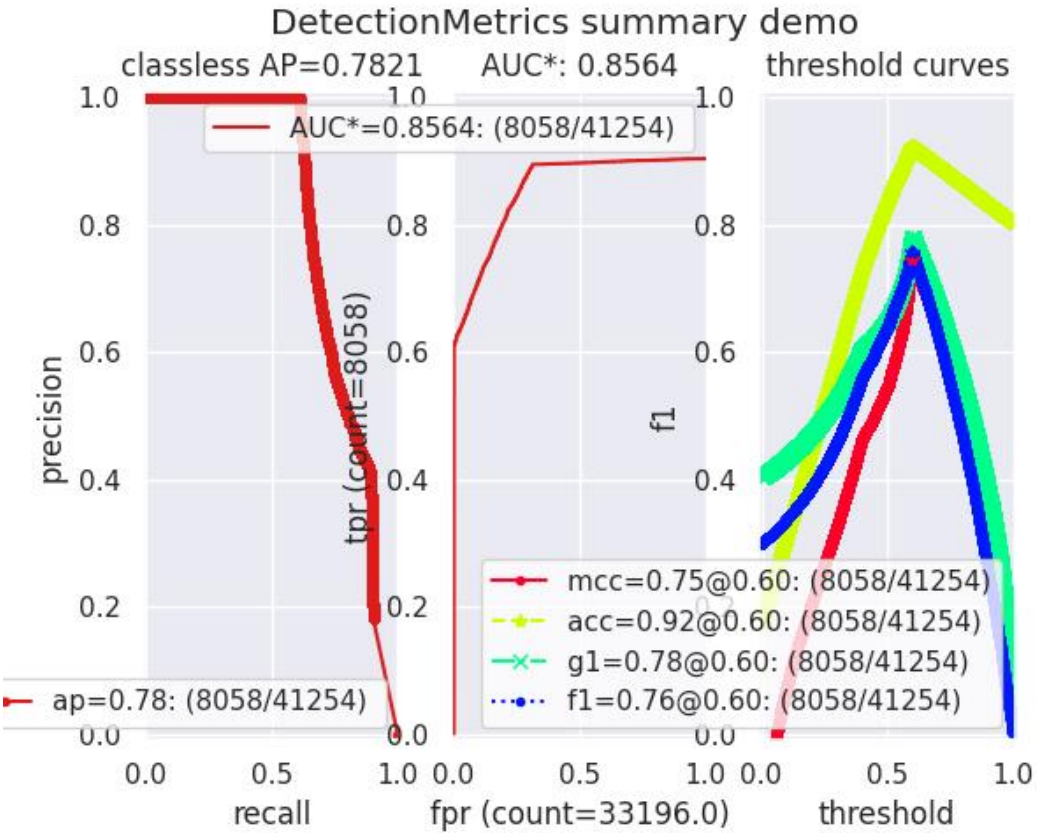
Example

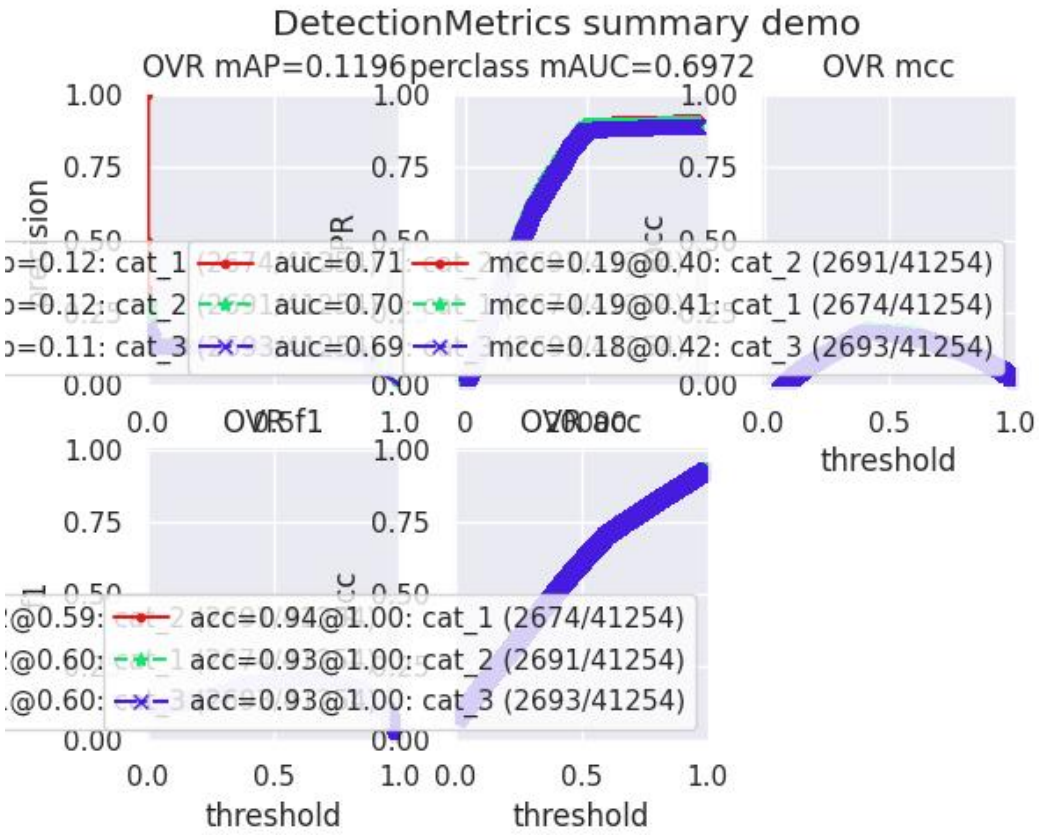
```
>>> from kwcoco.metrics.confusion_vectors import *  # NOQA
>>> from kwcoco.metrics.detect_metrics import DetectionMetrics
>>> dmet = DetectionMetrics.demo(
>>>     n_fp=(0, 1), n_fn=(0, 1), nimgs=32, nboxes=(0, 16),
>>>     classes=3, rng=0, newstyle=1, box_noise=0.5, cls_noise=0.0, score_
↪noise=0.3, with_probs=False)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> summary = dmet.summarize(plot=True, title='DetectionMetrics summary demo',␣
↪with_ovr=True, with_bin=False)
>>> summary['bin_measures']
>>> kwplot.show_if_requested()
```

summarize(*out_dpath=None*, *plot=False*, *title=''*, *with_bin='auto'*, *with_ovr='auto'*)

Example

```
>>> from kwcoco.metrics.confusion_vectors import *  # NOQA
>>> from kwcoco.metrics.detect_metrics import DetectionMetrics
>>> dmet = DetectionMetrics.demo(
>>>     n_fp=(0, 128), n_fn=(0, 4), nimgs=512, nboxes=(0, 32),
>>>     classes=3, rng=0)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> dmet.summarize(plot=True, title='DetectionMetrics summary demo')
>>> kwplot.show_if_requested()
```





kwcoco.metrics.detect_metrics.**pycocotools_confusion_vectors**(*dmet*, *evaler*, *iou_thresh=0.5*, *verbose=0*)

Example

```
>>> # xdoctest: +REQUIRES(module:pycocotools)
>>> from kwcoco.metrics.detect_metrics import *
>>> dmet = DetectionMetrics.demo(
>>>     nimgs=10, nboxes=(0, 3), n_fn=(0, 1), n_fp=(0, 1), classes=8, with_
↪probs=False)
>>> coco_scores = dmet.score_pycocotools(with_evaler=True)
>>> evaler = coco_scores['evaler']
>>> cfsn_vecs = pycocotools_confusion_vectors(dmet, evaler, verbose=1)
```

kwcoco.metrics.detect_metrics.**eval_detections_cli**(***kw*)

DEPRECATED USE *kwcoco eval* instead

CommandLine

```
xdoctest -m ~/code/kwcoco/kwcoco/metrics/detect_metrics.py eval_detections_cli
```

kwcoco.metrics.detect_metrics.**pct_summarize2**(*self*)

2.1.1.5.1.6 kwcoco.metrics.drawing module

kwcoco.metrics.drawing.**draw_perclass_roc**(*cx_to_info*, *classes=None*, *prefix=''*, *fnum=1*, *fp_axis='count'*, ***kw*)

Parameters

- **cx_to_info** (*kwcoco.metrics.confusion_measures.PerClass_Measures | Dict*)
- **fp_axis** (*str*) – can be count or rate

kwcoco.metrics.drawing.**demo_format_options**()

kwcoco.metrics.drawing.**concice_si_display**(*val*, *eps=1e-08*, *precision=2*, *si_thresh=4*)

Display numbers in scientific notation if above a threshold

Parameters

- **eps** (*float*) – threshold to be formated as an integer if other integer conditions hold.
- **precision** (*int*) – maximum significant digits (might print less)
- **si_thresh** (*int*) – If the number is less than 10^{si_thresh}, then it will be printed as an integer if it is within eps of an integer.

References

https://docs.python.org/2/library/stdtypes.html#string-formatting-operations

Example

```
>>> grid = {
>>>     'sign': [1, -1],
>>>     'exp': [1, -1],
>>>     'big_part': [0, 32132e3, 4000000032],
>>>     'med_part': [0, 0.5, 0.9432, 0.000043, 0.01, 1, 2],
>>>     'small_part': [0, 1321e-3, 43242e-11],
>>> }
>>> for kw in ub.named_product(grid):
>>>     sign = kw.pop('sign')
>>>     exp = kw.pop('exp')
>>>     raw = (sum(map(float, kw.values())))
>>>     val = sign * raw ** exp if raw != 0 else sign * raw
>>>     print('{:>20} - {}'.format(concice_si_display(val), val))
>>> from kwcoco.metrics.drawing import *  # NOQA
>>> print(concice_si_display(40000000432432))
>>> print(concice_si_display(473243280432890))
>>> print(concice_si_display(473243284289))
```

(continues on next page)

(continued from previous page)

```
>>> print(concice_si_display(473243289))
>>> print(concice_si_display(4739))
>>> print(concice_si_display(473))
>>> print(concice_si_display(0.432432))
>>> print(concice_si_display(0.132432))
>>> print(concice_si_display(1.0000043))
>>> print(concice_si_display(01.000000000000000000000000000043))
```

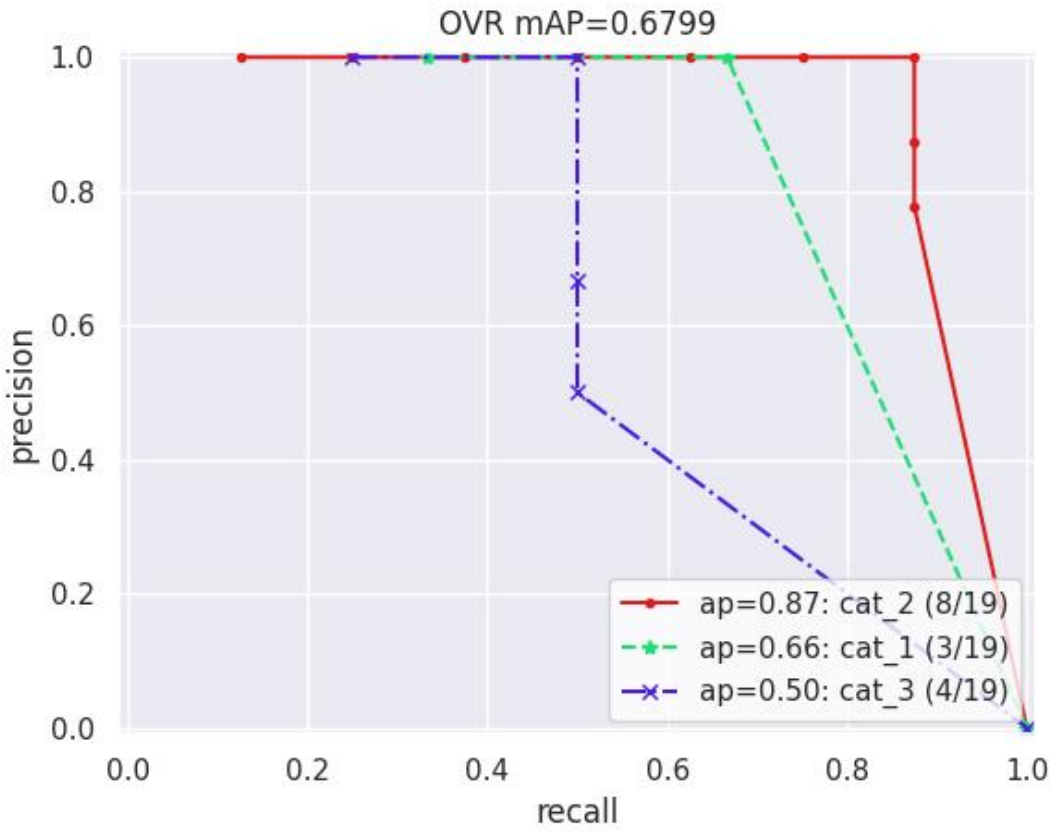
kwcoco.metrics.drawing.**draw_perclass_prcurve**(*cx_to_info*, *classes=None*, *prefix=''*, *fnum=1*, ***kw*)

Parameters

cx_to_info (*kwcoco.metrics.confusion_measures.PerClass_Measures | Dict*)

Example

```
>>> # xdoctest: +REQUIRES(module:kwplot)
>>> from kwcoco.metrics.drawing import *  # NOQA
>>> from kwcoco.metrics import DetectionMetrics
>>> dmet = DetectionMetrics.demo(
>>>     nimgs=3, nboxes=(0, 10), n_fp=(0, 3), n_fn=(0, 2), classes=3, score_noise=0.
↪1, box_noise=0.1, with_probs=False)
>>> cfsn_vecs = dmet.confusion_vectors()
>>> print(cfsn_vecs.data.pandas())
>>> classes = cfsn_vecs.classes
>>> cx_to_info = cfsn_vecs.binarize_ovr().measures()['perclass']
>>> print('cx_to_info = {}'.format(ub.repr2(cx_to_info, nl=1)))
>>> import kwplot
>>> kwplot.autompl()
>>> draw_perclass_prcurve(cx_to_info, classes)
>>> # xdoctest: +REQUIRES(--show)
>>> kwplot.show_if_requested()
```



kwcoco.metrics.drawing.**draw_perclass_thresholds**(*cx_to_info*, *key='mcc'*, *classes=None*, *prefix=''*, *fnum=1*, ***kw*)

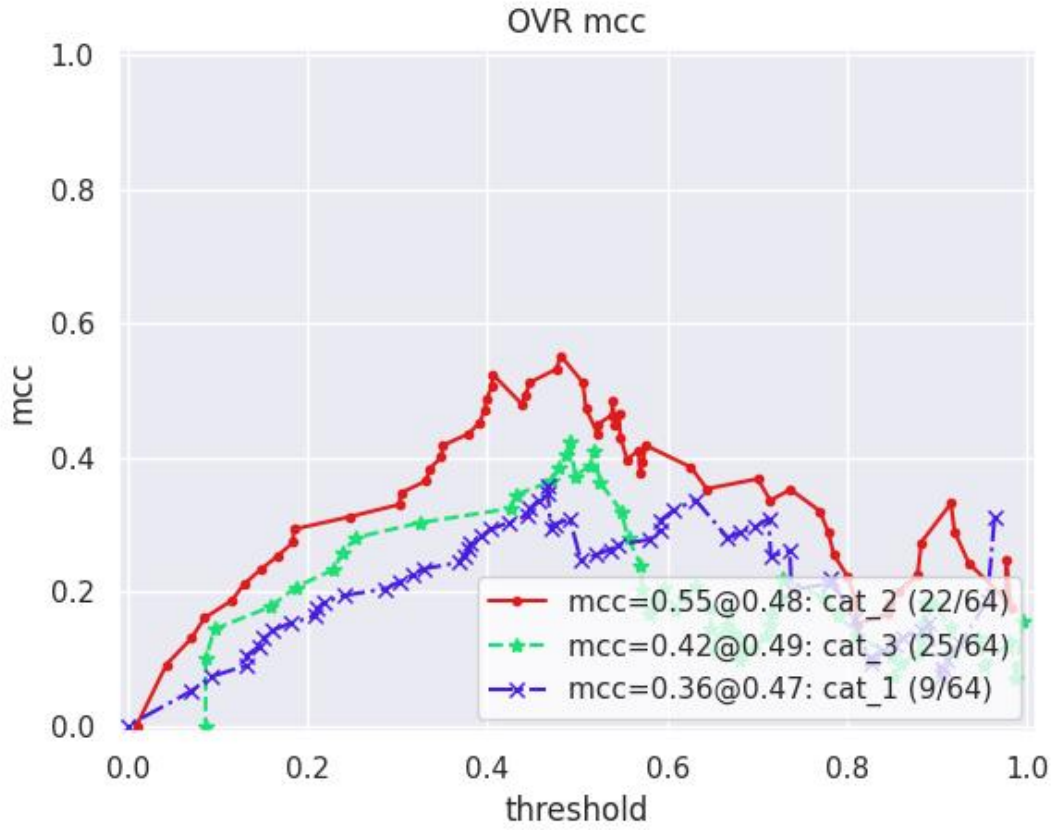
Parameters

cx_to_info (*kwcoco.metrics.confusion_measures.PerClass_Measures | Dict*)

Note: Each category is inspected independently of one another, there is no notion of confusion.

Example

```
>>> # xdoctest: +REQUIRES(module:kwplot)
>>> from kwcoco.metrics.drawing import *  # NOQA
>>> from kwcoco.metrics import ConfusionVectors
>>> cfsn_vecs = ConfusionVectors.demo()
>>> classes = cfsn_vecs.classes
>>> ovr_cfsn = cfsn_vecs.binarize_ovr(keyby='name')
>>> cx_to_info = ovr_cfsn.measures()['perclass']
>>> import kwplot
>>> kwplot.autompl()
>>> key = 'mcc'
>>> draw_perclass_thresholds(cx_to_info, key, classes)
>>> # xdoctest: +REQUIRES(--show)
>>> kwplot.show_if_requested()
```



kwcoco.metrics.drawing.**draw_roc**(*info*, *prefix=''*, *fnum=1*, ***kw*)

Parameters

info (*Measures | Dict*)

Note: There needs to be enough negative examples for using ROC to make any sense!

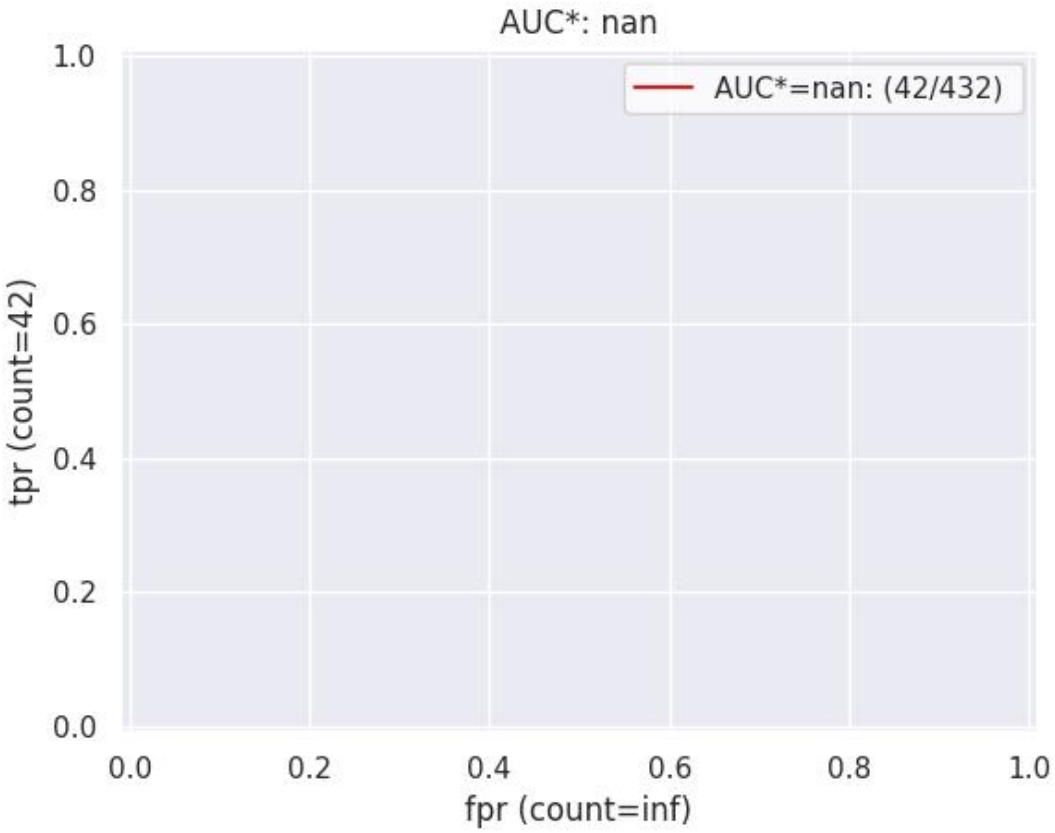
Example

```
>>> # xdoctest: +REQUIRES(module:kwplot, module:seaborn)
>>> from kwcoco.metrics.drawing import *  # NOQA
>>> from kwcoco.metrics import DetectionMetrics
>>> dmet = DetectionMetrics.demo(nimgs=30, null_pred=1, classes=3,
>>>                              nboxes=10, n_fp=10, box_noise=0.3,
>>>                              with_probs=False)
>>> dmet.true_detections(0).data
>>> cfsn_vecs = dmet.confusion_vectors(compat='mutex', prioritize='iou', bias=0)
>>> print(cfsn_vecs.data._pandas().sort_values('score'))
>>> classes = cfsn_vecs.classes
>>> info = ub.peek(cfsn_vecs.binarize_ovr().measures()['perclass'].values())
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
```

(continues on next page)

(continued from previous page)

```
>>> kwplot.autompl()
>>> draw_roc(info)
>>> kwplot.show_if_requested()
```



kwcoco.metrics.drawing.**draw_prcurve**(*info*, *prefix=''*, *fnum=1*, ***kw*)

Draws a single pr curve.

Parameters

info (*Measures | Dict*)

Example

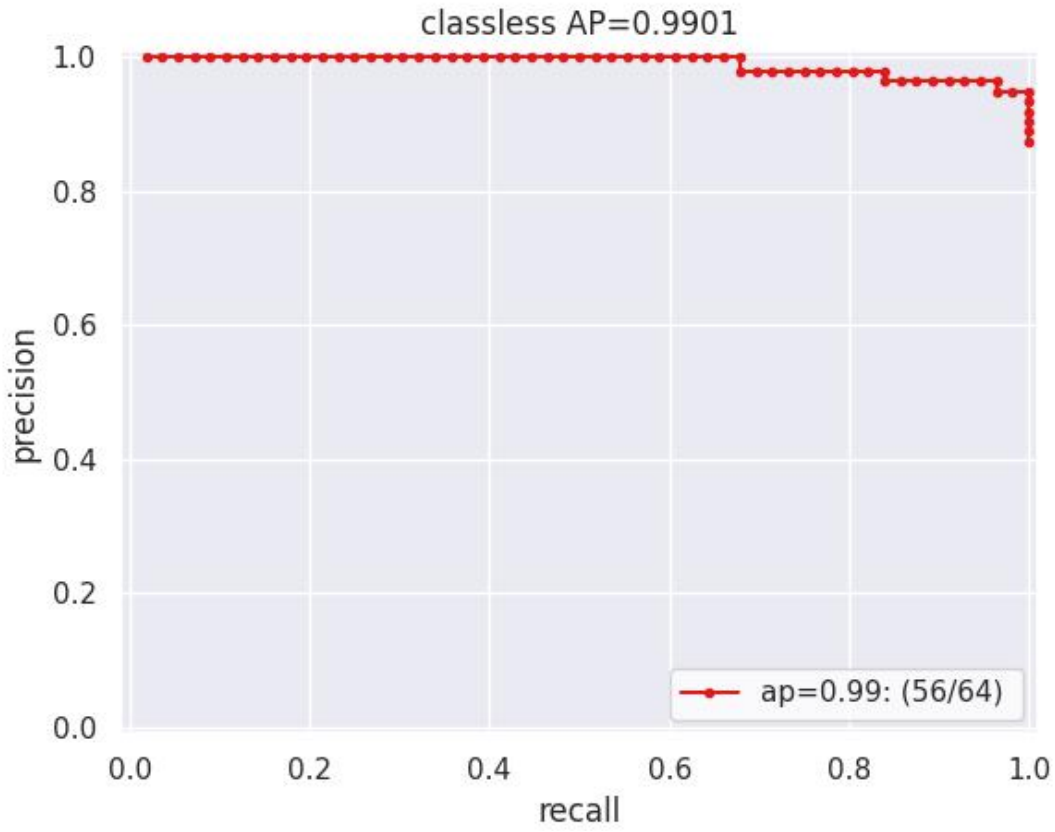
```
>>> # xdoctest: +REQUIRES(module:kwplot)
>>> from kwcoco.metrics import DetectionMetrics
>>> dmet = DetectionMetrics.demo(
>>>     nimgs=10, nboxes=(0, 10), n_fp=(0, 1), classes=3)
>>> cfsn_vecs = dmet.confusion_vectors()
```

```
>>> classes = cfsn_vecs.classes
>>> info = cfsn_vecs.binarize_classless().measures()
>>> import kwplot
>>> kwplot.autompl()
>>> draw_prcurve(info)
```

(continues on next page)

(continued from previous page)

```
>>> # xdoctest: +REQUIRES(--show)
>>> kwplot.show_if_requested()
```



kwcoco.metrics.drawing.**draw_threshold_curves**(*info*, *keys=None*, *prefix=''*, *fnum=1*, ***kw*)

Parameters

info (*Measures | Dict*)

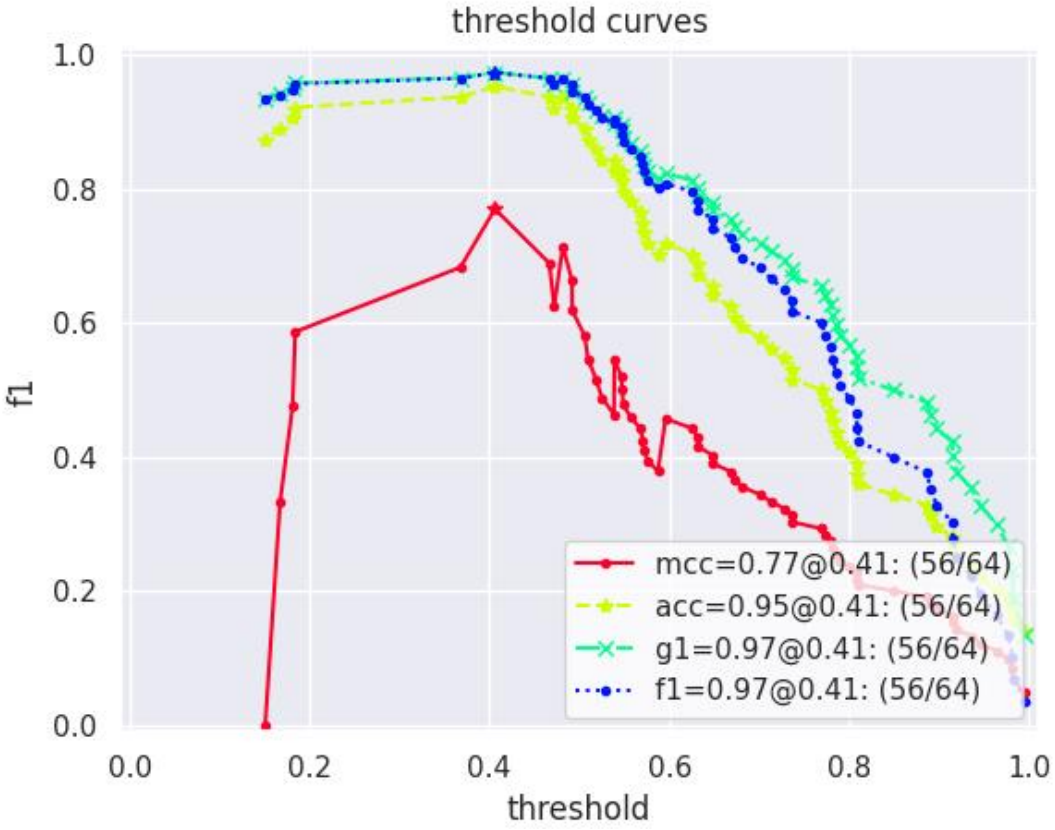
Example

```
>>> # xdoctest: +REQUIRES(module:kwplot)
>>> import sys, ubelt
>>> sys.path.append(ubelt.expandpath('~/code/kwcoco'))
>>> from kwcoco.metrics.drawing import *  # NOQA
>>> from kwcoco.metrics import DetectionMetrics
>>> dmet = DetectionMetrics.demo(
>>>     nimgs=10, nboxes=(0, 10), n_fp=(0, 1), classes=3)
>>> cfsn_vecs = dmet.confusion_vectors()
>>> info = cfsn_vecs.binarize_classless().measures()
>>> keys = None
>>> import kwplot
>>> kwplot.autompl()
>>> draw_threshold_curves(info, keys)
```

(continues on next page)

(continued from previous page)

```
>>> # xdoctest: +REQUIRES(--show)
>>> kwplot.show_if_requested()
```



2.1.1.5.1.7 kwcoco.metrics.functional module

kwcoco.metrics.functional.**fast_confusion_matrix**(*y_true*, *y_pred*, *n_labels*, *sample_weight=None*)

faster version of sklearn confusion matrix that avoids the expensive checks and label rectification

Parameters

- **y_true** (*ndarray[Any, Int]*) – ground truth class label for each sample
- **y_pred** (*ndarray[Any, Int]*) – predicted class label for each sample
- **n_labels** (*int*) – number of labels
- **sample_weight** (*ndarray*) – weight of each sample Extended typing `ndarray[Any, Int | Float]`

Returns

matrix where rows represent real and cols represent pred and the value at each cell is the total amount of weight Extended typing `ndarray[Shape['*, *'], Int64 | Float64]`

Return type

ndarray

Example

```
>>> y_true = np.array([0, 0, 0, 0, 1, 1, 1, 0,  0, 1])
>>> y_pred = np.array([0, 0, 0, 0, 0, 0, 0, 1,  1, 1])
>>> fast_confusion_matrix(y_true, y_pred, 2)
array([[4, 2],
       [3, 1]])
>>> fast_confusion_matrix(y_true, y_pred, 2).ravel()
array([4, 2, 3, 1])
```

2.1.1.5.1.8 kwcoco.metrics.sklearn_alts module

Faster pure-python versions of sklearn functions that avoid expensive checks and label rectifications. It is assumed that all labels are consecutive non-negative integers.

kwcoco.metrics.sklearn_alts.**confusion_matrix**(*y_true*, *y_pred*, *n_labels=None*, *labels=None*, *sample_weight=None*)

faster version of sklearn confusion matrix that avoids the expensive checks and label rectification

Runs in about 0.7ms

Returns
matrix where rows represent real and cols represent pred

Return type
ndarray

Example

```
>>> y_true = np.array([0, 0, 0, 0, 1, 1, 1, 0,  0, 1])
>>> y_pred = np.array([0, 0, 0, 0, 0, 0, 0, 1,  1, 1])
>>> confusion_matrix(y_true, y_pred, 2)
array([[4, 2],
       [3, 1]])
>>> confusion_matrix(y_true, y_pred, 2).ravel()
array([4, 2, 3, 1])
```

Benchmark

```
>>> # xdoctest: +SKIP
>>> import ubelt as ub
>>> y_true = np.random.randint(0, 2, 10000)
>>> y_pred = np.random.randint(0, 2, 10000)
>>> n = 1000
>>> for timer in ub.Timerit(n, bestof=10, label='py-time'):
>>>     sample_weight = [1] * len(y_true)
>>>     confusion_matrix(y_true, y_pred, 2, sample_weight=sample_weight)
>>> for timer in ub.Timerit(n, bestof=10, label='np-time'):
>>>     sample_weight = np.ones(len(y_true), dtype=int)
>>>     confusion_matrix(y_true, y_pred, 2, sample_weight=sample_weight)
```

kwcoco.metrics.sklearn_alts.**global_accuracy_from_confusion**(*cfsn*)

kwcoco.metrics.sklearn_alts.**class_accuracy_from_confusion**(*cfsn*)

2.1.1.5.1.9 kwcoco.metrics.util module

class kwcoco.metrics.util.**DictProxy**

Bases: DictLike

Allows an object to proxy the behavior of a dict attribute

keys()

2.1.1.5.1.10 kwcoco.metrics.voc_metrics module

class kwcoco.metrics.voc_metrics.**VOC_Metrics**(*classes=None*)

Bases: NiceRepr

API to compute object detection scores using Pascal VOC evaluation method.

To use, add true and predicted detections for each image and then run the *VOC_Metrics.score()* function.

Variables

- **recs** (*Dict[int, List[dict]]*) – true boxes for each image. maps image ids to a list of records within that image. Each record is a tlbr bbox, a difficult flag, and a class name.
- **cx_to_lines** (*Dict[int, List]*) – VOC formatted prediction preditions. mapping from class index to all predictions for that category. Each "line" is a list of [[<imgid>, <score>, <tl_x>, <tl_y>, <br_x>, <br_y>]].

add_truth(*true_dets*, *gid*)

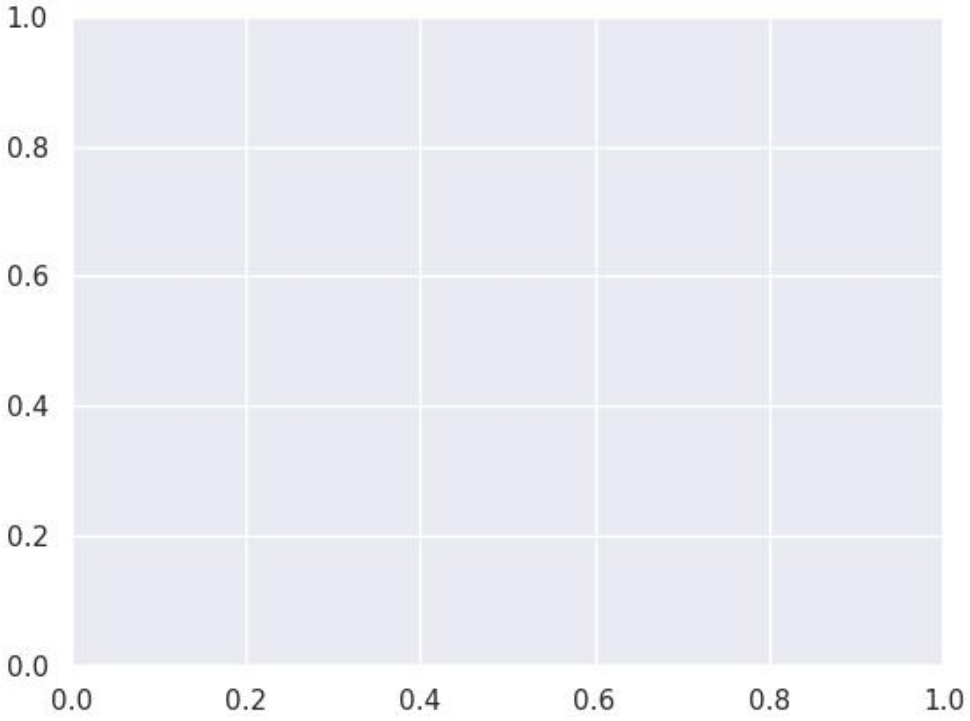
add_predictions(*pred_dets*, *gid*)

score(*iou_thresh=0.5*, *bias=1*, *method='voc2012'*)

Compute VOC scores for every category

Example

```
>>> from kwcoco.metrics.detect_metrics import DetectionMetrics
>>> from kwcoco.metrics.voc_metrics import *  # NOQA
>>> dmet = DetectionMetrics.demo(
>>>     nimgs=1, nboxes=(0, 100), n_fp=(0, 30), n_fn=(0, 30), classes=2, score_
↪noise=0.9)
>>> self = VOC_Metrics(classes=dmet.classes)
>>> self.add_truth(dmet.true_detections(0), 0)
>>> self.add_predictions(dmet.pred_detections(0), 0)
>>> voc_scores = self.score()
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.figure(fnum=1, doclf=True)
>>> voc_scores['perclass'].draw()
```



kwplot.figure(fnum=2) dmet.true_detections(0).draw(color='green', labels=None) dmet.pred_detections(0).draw(color='blue', labels=None) kwplot.autoplt().gca().set_xlim(0, 100) kwplot.autoplt().gca().set_ylim(0, 100)

2.1.1.5.2 Module contents

mkinit kwcoco.metrics -w –relative

class kwcoco.metrics.**BinaryConfusionVectors**(*data*, *cx=None*, *classes=None*)

Bases: NiceRepr

Stores information about a binary classification problem. This is always with respect to a specific class, which is given by *cx* and *classes*.

The *data* DataFrameArray must contain

is_true - if the row is an instance of class *classes[cx] pred_score* - the predicted probability of class *classes[cx]*, and *weight* - sample weight of the example

Example

```
>>> from kwcoco.metrics.confusion_vectors import *  # NOQA
>>> self = BinaryConfusionVectors.demo(n=10)
>>> print('self = {!r}'.format(self))
>>> print('measures = {}'.format(ub.repr2(self.measures())))
```

```
>>> self = BinaryConfusionVectors.demo(n=0)
>>> print('measures = {}'.format(ub.repr2(self.measures())))
```

```
>>> self = BinaryConfusionVectors.demo(n=1)
>>> print('measures = {}'.format(ub.repr2(self.measures())))
```

```
>>> self = BinaryConfusionVectors.demo(n=2)
>>> print('measures = {}'.format(ub.repr2(self.measures())))
```

classmethod demo(*n=10*, *p_true=0.5*, *p_error=0.2*, *p_miss=0.0*, *rng=None*)

Create random data for tests

Parameters

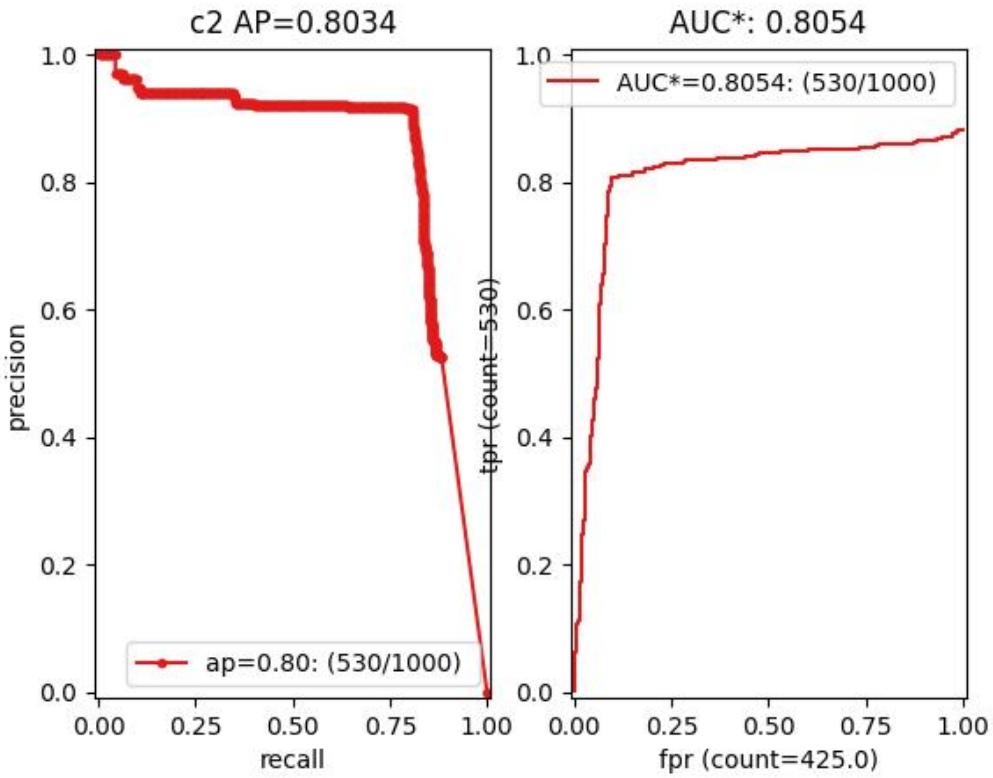
- **n** (*int*) – number of rows
- **p_true** (*float*) – fraction of real positive cases
- **p_error** (*float*) – probability of making a recoverable mistake
- **p_miss** (*float*) – probability of making a unrecoverable mistake
- **rng** (*int | RandomState*) – random seed / state

Returns

BinaryConfusionVectors

Example

```
>>> from kwcoco.metrics.confusion_vectors import *  # NOQA
>>> cfsn = BinaryConfusionVectors.demo(n=1000, p_error=0.1, p_miss=0.1)
>>> measures = cfsn.measures()
>>> print('measures = {}'.format(ub.repr2(measures, nl=1)))
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.figure(fnum=1, pnum=(1, 2, 1))
>>> measures.draw('pr')
>>> kwplot.figure(fnum=1, pnum=(1, 2, 2))
>>> measures.draw('roc')
```



property catname

measures(*stabalize_thresh=7*, *fp_cutoff=None*, *monotonic_ppv=True*, *ap_method='pycocotools'*)

Get statistics (F1, G1, MCC) versus thresholds

Parameters

- **stabalize_thresh** (*int, default=7*) – if fewer than this many data points inserts dummy stabalization data so curves can still be drawn.
- **fp_cutoff** (*int, default=None*) – maximum number of false positives in the truncated roc curves. `None` is equivalent to `float('inf')`
- **monotonic_ppv** (*bool, default=True*) – if True ensures that precision is always increasing as recall decreases. This is done in pycocotools scoring, but I'm not sure its a good idea.

Example

```
>>> from kwcoco.metrics.confusion_vectors import *  # NOQA
>>> self = BinaryConfusionVectors.demo(n=0)
>>> print('measures = {}'.format(ub.repr2(self.measures())))
>>> self = BinaryConfusionVectors.demo(n=1, p_true=0.5, p_error=0.5)
>>> print('measures = {}'.format(ub.repr2(self.measures())))
>>> self = BinaryConfusionVectors.demo(n=3, p_true=0.5, p_error=0.5)
>>> print('measures = {}'.format(ub.repr2(self.measures())))
```

```
>>> self = BinaryConfusionVectors.demo(n=100, p_true=0.5, p_error=0.5, p_miss=0.
↪3)
>>> print('measures = {}'.format(ub.repr2(self.measures())))
>>> print('measures = {}'.format(ub.repr2(ub.odict(self.measures()))))
```

References

https://en.wikipedia.org/wiki/Confusion_matrix https://en.wikipedia.org/wiki/Precision_and_recall https://en.wikipedia.org/wiki/Matthews_correlation_coefficient

draw_distribution()

class kwcoco.metrics.**ConfusionVectors**(*data*, *classes*, *probs=None*)

Bases: NiceRepr

Stores information used to construct a confusion matrix. This includes corresponding vectors of predicted labels, true labels, sample weights, etc…

Variables

- **data** (*kwarray.DataFrameArray*) – should at least have keys true, pred, weight
- **classes** (*Sequence | CategoryTree*) – list of category names or category graph
- **probs** (*ndarray, optional*) – probabilities for each class

Example

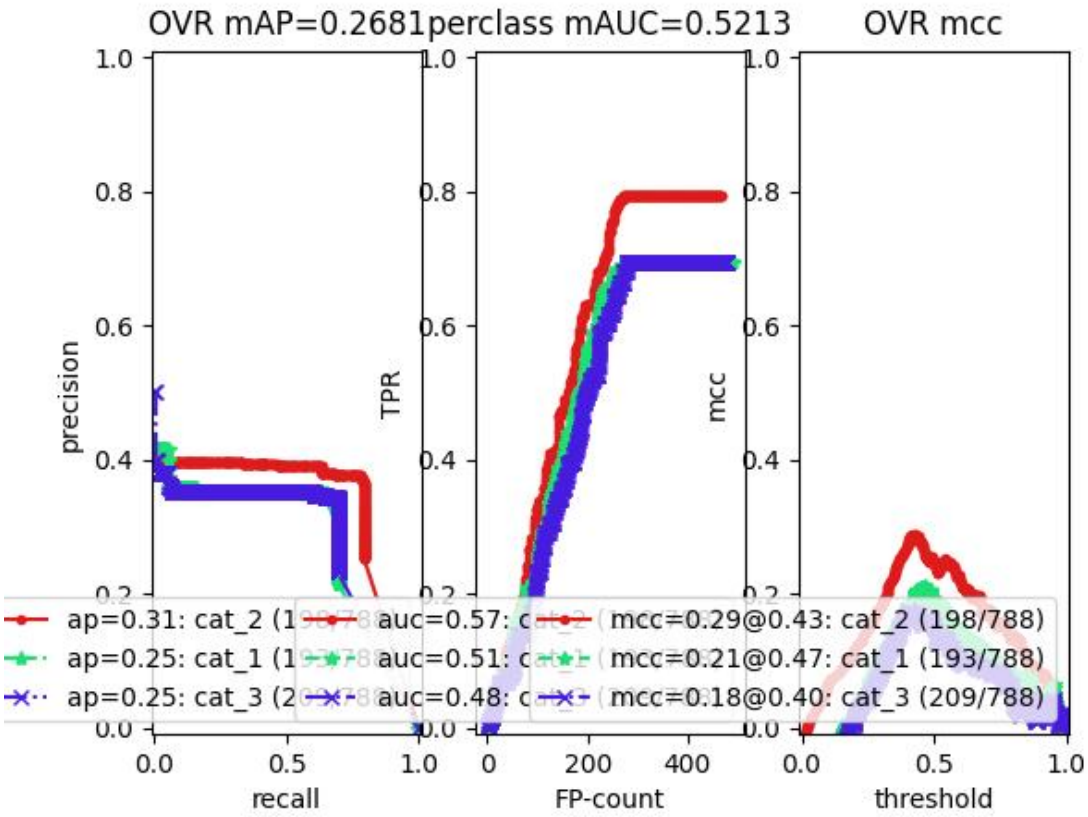
```
>>> # xdoctest: IGNORE_WANT
>>> # xdoctest: +REQUIRES(module:pandas)
>>> from kwcoco.metrics import DetectionMetrics
>>> dmet = DetectionMetrics.demo(
>>>     nimgs=10, nboxes=(0, 10), n_fp=(0, 1), classes=3)
>>> cfsn_vecs = dmet.confusion_vectors()
>>> print(cfsn_vecs.data._pandas())
     pred  true   score  weight     iou  txs  pxs  gid
0       2     2 10.0000  1.0000  1.0000    0    4    0
1       2     2  7.5025  1.0000  1.0000    1    3    0
2       1     1  5.0050  1.0000  1.0000    2    2    0
3       3    -1  2.5075  1.0000 -1.0000   -1    1    0
4       2    -1  0.0100  1.0000 -1.0000   -1    0    0
5      -1     2  0.0000  1.0000 -1.0000    3   -1    0
6      -1     2  0.0000  1.0000 -1.0000    4   -1    0
7       2     2 10.0000  1.0000  1.0000    0    5    1
8       2     2  8.0020  1.0000  1.0000    1    4    1
9       1     1  6.0040  1.0000  1.0000    2    3    1
..    ...   ...     ...     ...     ...  ...  ...  ...
62     -1     2  0.0000  1.0000 -1.0000    7   -1    7
63     -1     3  0.0000  1.0000 -1.0000    8   -1    7
64     -1     1  0.0000  1.0000 -1.0000    9   -1    7
65      1    -1 10.0000  1.0000 -1.0000   -1    0    8
66      1     1  0.0100  1.0000  1.0000    0    1    8
67      3    -1 10.0000  1.0000 -1.0000   -1    3    9
```

(continues on next page)

(continued from previous page)

```
68      2     2  6.6700  1.0000  1.0000    0    2    9
69      2     2  3.3400  1.0000  1.0000    1    1    9
70      3    -1  0.0100  1.0000 -1.0000   -1    0    9
71     -1     2  0.0000  1.0000 -1.0000    2   -1    9
```

```
>>> # xdoctest: +REQUIRES(--show)
>>> # xdoctest: +REQUIRES(module:pandas)
>>> import kwplot
>>> kwplot.autompl()
>>> from kwcoco.metrics.confusion_vectors import ConfusionVectors
>>> cfsn_vecs = ConfusionVectors.demo(
>>>     nimgs=128, nboxes=(0, 10), n_fp=(0, 3), n_fn=(0, 3), classes=3)
>>> cx_to_binvecs = cfsn_vecs.binarize_ovr()
>>> measures = cx_to_binvecs.measures()['perclass']
>>> print('measures = {!r}'.format(measures))
measures = <PerClass_Measures({
    'cat_1': <Measures({'ap': 0.227, 'auc': 0.507, 'catname': cat_1, 'max_f1': f1=0.
↪45@0.47, 'nsupport': 788.000})>,
    'cat_2': <Measures({'ap': 0.288, 'auc': 0.572, 'catname': cat_2, 'max_f1': f1=0.
↪51@0.43, 'nsupport': 788.000})>,
    'cat_3': <Measures({'ap': 0.225, 'auc': 0.484, 'catname': cat_3, 'max_f1': f1=0.
↪46@0.40, 'nsupport': 788.000})>,
}) at 0x7facf77bdfd0>
>>> kwplot.figure(fnum=1, doclf=True)
>>> measures.draw(key='pr', fnum=1, pnum=(1, 3, 1))
>>> measures.draw(key='roc', fnum=1, pnum=(1, 3, 2))
>>> measures.draw(key='mcc', fnum=1, pnum=(1, 3, 3))
...
```



classmethod from_json(*state*)

classmethod demo(***kw*)

Parameters
****kwargs** – See *kwcoco.metrics.DetectionMetrics.demo()*

Returns
ConfusionVectors

Example

```
>>> cfsn_vecs = ConfusionVectors.demo()
>>> print('cfsn_vecs = {!r}'.format(cfsn_vecs))
>>> cx_to_binvecs = cfsn_vecs.binarize_ovr()
>>> print('cx_to_binvecs = {!r}'.format(cx_to_binvecs))
```

classmethod from_arrays(*true*, *pred=None*, *score=None*, *weight=None*, *probs=None*, *classes=None*)
Construct confusion vector data structure from component arrays

Example

```
>>> # xdoctest: +REQUIRES(module:pandas)
>>> import kwarray
>>> classes = ['person', 'vehicle', 'object']
>>> rng = kwarray.ensure_rng(0)
>>> true = (rng.rand(10) * len(classes)).astype(int)
>>> probs = rng.rand(len(true), len(classes))
>>> cfsn_vecs = ConfusionVectors.from_arrays(true=true, probs=probs,␣
↪classes=classes)
>>> cfsn_vecs.confusion_matrix()
pred     person  vehicle  object
real
person        0        0       0
vehicle       2        4       1
object        2        1       0
```

confusion_matrix(*compress=False*)

Builds a confusion matrix from the confusion vectors.

Parameters

compress (*bool, default=False*) – if True removes rows / columns with no entries

Returns

cm

[the labeled confusion matrix]

(Note: we should write a efficient replacement for

this use case. #remove_pandas)

Return type

pd.DataFrame

CommandLine

```
xdoctest -m kwcoco.metrics.confusion_vectors ConfusionVectors.confusion_matrix
```

Example

```
>>> # xdoctest: +REQUIRES(module:pandas)
>>> from kwcoco.metrics import DetectionMetrics
>>> dmet = DetectionMetrics.demo(
>>>     nimgs=10, nboxes=(0, 10), n_fp=(0, 1), n_fn=(0, 1), classes=3, cls_
↪noise=.2)
>>> cfsn_vecs = dmet.confusion_vectors()
>>> cm = cfsn_vecs.confusion_matrix()
...
>>> print(cm.to_string(float_format=lambda x: '%.2f' % x))
pred        background  cat_1  cat_2  cat_3
real
background        0.00   1.00   2.00   3.00
cat_1             3.00  12.00   0.00   0.00
```

(continues on next page)

(continued from previous page)

```
cat_2              3.00   0.00  14.00   0.00
cat_3              2.00   0.00   0.00  17.00
```

coarsen(*cxs*)

Creates a coarsened set of vectors

Returns
ConfusionVectors

binarize_classless(*negative_classes=None*)

Creates a binary representation useful for measuring the performance of detectors. It is assumed that scores of "positive" classes should be high and "negative" clases should be low.

Parameters
negative_classes (*List[str | int]*) – list of negative class names or idxs, by default chooses any class with a true class index of -1. These classes should ideally have low scores.

Returns
BinaryConfusionVectors

Note: The "classlessness" of this depends on the compat="all" argument being used when constructing confusion vectors, otherwise it becomes something like a macro-average because the class information was used in deciding which true and predicted boxes were allowed to match.

Example

```
>>> from kwcoco.metrics import DetectionMetrics
>>> dmet = DetectionMetrics.demo(
>>>     nimgs=10, nboxes=(0, 10), n_fp=(0, 1), n_fn=(0, 1), classes=3)
>>> cfsn_vecs = dmet.confusion_vectors()
>>> class_idxs = list(dmet.classes.node_to_idx.values())
>>> binvecs = cfsn_vecs.binarize_classless()
```

binarize_ovr(*mode=1*, *keyby='name'*, *ignore_classes={'ignore'}*, *approx=False*)

Transforms cfsn_vecs into one-vs-rest BinaryConfusionVectors for each category.

Parameters

- **mode** (*int, default=1*) – 0 for heirarchy aware or 1 for voc like. MODE 0 IS PROBABLY BROKEN
- **keyby** (*int | str*) – can be cx or name
- **ignore_classes** (*Set[str]*) – category names to ignore
- **approx** (*bool, default=0*) – if True try and approximate missing scores otherwise assume they are irrecoverable and use -inf

Returns

which behaves like
Dict[int, BinaryConfusionVectors]: cx_to_binvecs

Return type
OneVsRestConfusionVectors

Example

```
>>> from kwcoco.metrics.confusion_vectors import *  # NOQA
>>> cfsn_vecs = ConfusionVectors.demo()
>>> print('cfsn_vecs = {!r}'.format(cfsn_vecs))
>>> catname_to_binvecs = cfsn_vecs.binarize_ovr(keyby='name')
>>> print('catname_to_binvecs = {!r}'.format(catname_to_binvecs))
```

cfsn_vecs.data.pandas() catname_to_binvecs.cx_to_binvecs[‘class_1’].data.pandas()

Note:

classification_report(*verbose=0*)

Build a classification report with various metrics.

Example

```
>>> # xdoctest: +REQUIRES(module:pandas)
>>> from kwcoco.metrics.confusion_vectors import *  # NOQA
>>> cfsn_vecs = ConfusionVectors.demo()
>>> report = cfsn_vecs.classification_report(verbose=1)
```

class kwcoco.metrics.**DetectionMetrics**(*classes=None*)

Bases: NiceRepr

Object that computes associations between detections and can convert them into sklearn-compatible representations for scoring.

Variables

- **gid_to_true_dets** (*Dict*) – maps image ids to truth
- **gid_to_pred_dets** (*Dict*) – maps image ids to predictions
- **classes** (CategoryTree) – category coder

Example

```
>>> dmet = DetectionMetrics.demo(
>>>     nimgs=100, nboxes=(0, 3), n_fp=(0, 1), classes=8, score_noise=0.9,␣
↪hacked=False)
>>> print(dmet.score_kwcoco(bias=0, compat='mutex', prioritize='iou')['mAP'])
...
>>> # NOTE: IN GENERAL NETHARN AND VOC ARE NOT THE SAME
>>> print(dmet.score_voc(bias=0)['mAP'])
0.8582...
>>> #print(dmet.score_coco()['mAP'])
```

clear()

classmethod from_coco(*true_coco*, *pred_coco*, *gids=None*, *verbose=0*)

Create detection metrics from two coco files representing the truth and predictions.

Parameters

- **true_coco** (*kwcoco.CocoDataset*)
- **pred_coco** (*kwcoco.CocoDataset*)

Example

```
>>> import kwcoco
>>> from kwcoco.demo.perterb import perterb_coco
>>> true_coco = kwcoco.CocoDataset.demo('shapes')
>>> perterbkw = dict(box_noise=0.5, cls_noise=0.5, score_noise=0.5)
>>> pred_coco = perterb_coco(true_coco, **perterbkw)
>>> self = DetectionMetrics.from_coco(true_coco, pred_coco)
>>> self.score_voc()
```

add_predictions(*pred_dets*, *imgname=None*, *gid=None*)

Register/Add predicted detections for an image

Parameters

- **pred_dets** (*kwimage.Detections*) – predicted detections
- **imgname** (*str*) – a unique string to identify the image
- **gid** (*int | None*) – the integer image id if known

add_truth(*true_dets*, *imgname=None*, *gid=None*)

Register/Add groundtruth detections for an image

Parameters

- **true_dets** (*kwimage.Detections*) – groundtruth
- **imgname** (*str*) – a unique string to identify the image
- **gid** (*int | None*) – the integer image id if known

true_detections(*gid*)

gets Detections representation for groundtruth in an image

pred_detections(*gid*)

gets Detections representation for predictions in an image

confusion_vectors(*iou_thresh=0.5*, *bias=0*, *gids=None*, *compat='mutex'*, *prioritize='iou'*, *ignore_classes='ignore'*, *background_class=NoParam*, *verbose='auto'*, *workers=0*, *track_probs='try'*, *max_dets=None*)

Assigns predicted boxes to the true boxes so we can transform the detection problem into a classification problem for scoring.

Parameters

- **iou_thresh** (*float | List[float], default=0.5*) – bounding box overlap iou threshold required for assignment if a list, then return type is a dict
- **bias** (*float, default=0.0*) – for computing bounding box overlap, either 1 or 0
- **gids** (*List[int], default=None*) – which subset of images ids to compute confusion metrics on. If not specified all images are used.

- **compat** (*str, default='all'*) – can be ('ancestors' | 'mutex' | 'all'). determines which pred boxes are allowed to match which true boxes. If 'mutex', then pred boxes can only match true boxes of the same class. If 'ancestors', then pred boxes can match true boxes that match or have a coarser label. If 'all', then any pred can match any true, regardless of its category label.
- **prioritize** (*str, default='iou'*) – can be ('iou' | 'class' | 'correct') determines which box to assign to if mutiple true boxes overlap a predicted box. if prioritize is iou, then the true box with maximum iou (above iou_thresh) will be chosen. If prioritize is class, then it will prefer matching a compatible class above a higher iou. If prioritize is correct, then ancestors of the true class are preferred over descendents of the true class, over unreleated classes.
- **ignore_classes** (*set | str, default={'ignore'}*) – class names indicating ignore regions
- **background_class** (*str, default=ub.NoParam*) – Name of the background class. If unspecified we try to determine it with heuristics. A value of None means there is no background class.
- **verbose** (*int | str, default='auto'*) – verbosity flag. In auto mode, verbose=1 if len(gids) > 1000.
- **workers** (*int, default=0*) – number of parallel assignment processes
- **track_probs** (*str, default='try'*) – can be 'try', 'force', or False. if truthy, we assume probabilities for multiple classes are available.

Returns

kwcoco.metrics.confusion_vectors.ConfusionVectors | Dict[float, kwcoco.metrics.confusion_vectors.ConfusionVectors]

Example

```
>>> dmet = DetectionMetrics.demo(nimgs=30, classes=3,
>>>                              nboxes=10, n_fp=3, box_noise=10,
>>>                              with_probs=False)
>>> iou_to_cfsn = dmet.confusion_vectors(iou_thresh=[0.3, 0.5, 0.9])
>>> for t, cfsn in iou_to_cfsn.items():
>>>     print('t = {!r}'.format(t))
...     print(cfsn.binarize_ovr().measures())
...     print(cfsn.binarize_classless().measures())
```

score_kwant(*iou_thresh=0.5*)

Scores the detections using kwant

score_kwcoco(*iou_thresh=0.5*, *bias=0*, *gids=None*, *compat='all'*, *prioritize='iou'*)

our scoring method

score_voc(*iou_thresh=0.5*, *bias=1*, *method='voc2012'*, *gids=None*, *ignore_classes='ignore'*)

score using voc method

Example

```
>>> dmet = DetectionMetrics.demo(
>>>     nimgs=100, nboxes=(0, 3), n_fp=(0, 1), classes=8,
>>>     score_noise=.5)
>>> print(dmet.score_voc()['mAP'])
0.9399...
```

score_pycocotools(*with_evaler=False*, *with_confusion=False*, *verbose=0*, *iou_thresholds=None*)

score using ms-coco method

Returns
dictionary with pct info

Return type
Dict

Example

```
>>> # xdoctest: +REQUIRES(module:pycocotools)
>>> from kwcoco.metrics.detect_metrics import *
>>> dmet = DetectionMetrics.demo(
>>>     nimgs=10, nboxes=(0, 3), n_fn=(0, 1), n_fp=(0, 1), classes=8, with_
↪probs=False)
>>> pct_info = dmet.score_pycocotools(verbose=1,
>>>                                   with_evaler=True,
>>>                                   with_confusion=True,
>>>                                   iou_thresholds=[0.5, 0.9])
>>> evaler = pct_info['evaler']
>>> iou_to_cfsn_vecs = pct_info['iou_to_cfsn_vecs']
>>> for iou_thresh in iou_to_cfsn_vecs.keys():
>>>     print('iou_thresh = {!r}'.format(iou_thresh))
>>>     cfsn_vecs = iou_to_cfsn_vecs[iou_thresh]
>>>     ovr_measures = cfsn_vecs.binarize_ovr().measures()
>>>     print('ovr_measures = {}'.format(ub.repr2(ovr_measures, nl=1,␣
↪precision=4)))
```

Note: by default pycocotools computes average precision as the literal average of computed precisions at 101 uniformly spaced recall thresholds.

pycocoutils seems to only allow predictions with the same category as the truth to match those truth objects. This should be the same as calling dmet.confusion_vectors with compat = mutex

pycocoutils does not take into account the fact that each box often has a score for each category.

pycocoutils will be incorrect if any annotation has an id of 0

a major difference in the way kwcoco scores versus pycocoutils is the calculation of AP. The assignment between truth and predicted detections produces similar enough results. Given our confusion vectors we use the scikit-learn definition of AP, whereas pycocoutils seems to compute precision and recall — more or less correctly — but then it resamples the precision at various specified recall thresholds (in the *accumulate* function, specifically how *pr* is resampled into the *q* array). This can lead to a large difference in reported scores.

pycocoutils also smooths out the precision such that it is monotonic decreasing, which might not be the best idea.

pycocotools area ranges are inclusive on both ends, that means the "small" and "medium" truth selections do overlap somewhat.

---

score_coco(*with_evaler=False*, *with_confusion=False*, *verbose=0*, *iou_thresholds=None*)

score using ms-coco method

Returns
dictionary with pct info

Return type
Dict

Example

```
>>> # xdoctest: +REQUIRES(module:pycocotools)
>>> from kwcoco.metrics.detect_metrics import *
>>> dmet = DetectionMetrics.demo(
>>>     nimgs=10, nboxes=(0, 3), n_fn=(0, 1), n_fp=(0, 1), classes=8, with_
↪probs=False)
>>> pct_info = dmet.score_pycocotools(verbose=1,
>>>                                   with_evaler=True,
>>>                                   with_confusion=True,
>>>                                   iou_thresholds=[0.5, 0.9])
>>> evaler = pct_info['evaler']
>>> iou_to_cfsn_vecs = pct_info['iou_to_cfsn_vecs']
>>> for iou_thresh in iou_to_cfsn_vecs.keys():
>>>     print('iou_thresh = {!r}'.format(iou_thresh))
>>>     cfsn_vecs = iou_to_cfsn_vecs[iou_thresh]
>>>     ovr_measures = cfsn_vecs.binarize_ovr().measures()
>>>     print('ovr_measures = {}'.format(ub.repr2(ovr_measures, nl=1,␣
↪precision=4)))
```

Note: by default pycocotools computes average precision as the literal average of computed precisions at 101 uniformly spaced recall thresholds.

pycocoutils seems to only allow predictions with the same category as the truth to match those truth objects. This should be the same as calling dmet.confusion_vectors with compat = mutex

pycocoutils does not take into account the fact that each box often has a score for each category.

pycocoutils will be incorrect if any annotation has an id of 0

a major difference in the way kwcoco scores versus pycocoutils is the calculation of AP. The assignment between truth and predicted detections produces similar enough results. Given our confusion vectors we use the scikit-learn definition of AP, whereas pycocoutils seems to compute precision and recall — more or less correctly — but then it resamples the precision at various specified recall thresholds (in the *accumulate* function, specifically how *pr* is resampled into the *q* array). This can lead to a large difference in reported scores.

pycocoutils also smooths out the precision such that it is monotonic decreasing, which might not be the best idea.

pycocotools area ranges are inclusive on both ends, that means the “small” and “medium” truth selections do overlap somewhat.

---

classmethod demo(***kwargs*)

Creates random true boxes and predicted boxes that have some noisy offset from the truth.

Kwargs:

classes (int):
class list or the number of foreground classes. Defaults to 1.

nimgs (int): number of images in the coco datasts. Defaults to 1.

nboxes (int): boxes per image. Defaults to 1.

n_fp (int): number of false positives. Defaults to 0.

n_fn (int):
number of false negatives. Defaults to 0.

box_noise (float):
std of a normal distribution used to perterb both box location and box size. Defaults to 0.

cls_noise (float):
probability that a class label will change. Must be within 0 and 1. Defaults to 0.

anchors (ndarray):
used to create random boxes. Defaults to None.

null_pred (bool):
if True, predicted classes are returned as null, which means only localization scoring is suitable. Defaults to 0.

with_probs (bool):
if True, includes per-class probabilities with predictions Defaults to 1.

CommandLine

```
xdoctest -m kwcoco.metrics.detect_metrics DetectionMetrics.demo:2 --show
```

Example

```
>>> kwargs = {}
>>> # Seed the RNG
>>> kwargs['rng'] = 0
>>> # Size parameters determine how big the data is
>>> kwargs['nimgs'] = 5
>>> kwargs['nboxes'] = 7
>>> kwargs['classes'] = 11
>>> # Noise parameters perterb predictions further from the truth
>>> kwargs['n_fp'] = 3
>>> kwargs['box_noise'] = 0.1
>>> kwargs['cls_noise'] = 0.5
>>> dmet = DetectionMetrics.demo(**kwargs)
>>> print('dmet.classes = {}'.format(dmet.classes))
```

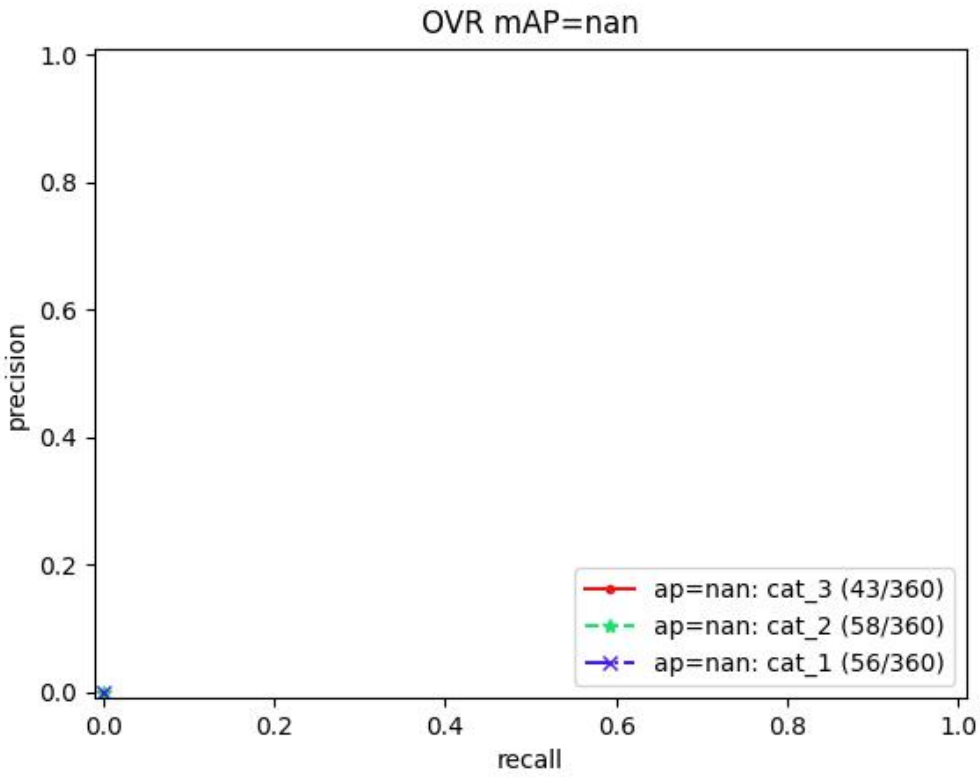
(continues on next page)

(continued from previous page)

```
dmet.classes = <CategoryTree(nNodes=12, maxDepth=3, maxBreadth=4...)>
>>> # Can grab kwimage.Detection object for any image
>>> print(dmet.true_detections(gid=0))
<Detections(4)>
>>> print(dmet.pred_detections(gid=0))
<Detections(7)>
```

Example

```
>>> # Test case with null predicted categories
>>> dmet = DetectionMetrics.demo(nimgs=30, null_pred=1, classes=3,
>>>                              nboxes=10, n_fp=3, box_noise=0.1,
>>>                              with_probs=False)
>>> dmet.gid_to_pred_dets[0].data
>>> dmet.gid_to_true_dets[0].data
>>> cfsn_vecs = dmet.confusion_vectors()
>>> binvecs_ovr = cfsn_vecs.binarize_ovr()
>>> binvecs_per = cfsn_vecs.binarize_classless()
>>> measures_per = binvecs_per.measures()
>>> measures_ovr = binvecs_ovr.measures()
>>> print('measures_per = {!r}'.format(measures_per))
>>> print('measures_ovr = {!r}'.format(measures_ovr))
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> measures_ovr['perclass'].draw(key='pr', fnum=2)
```



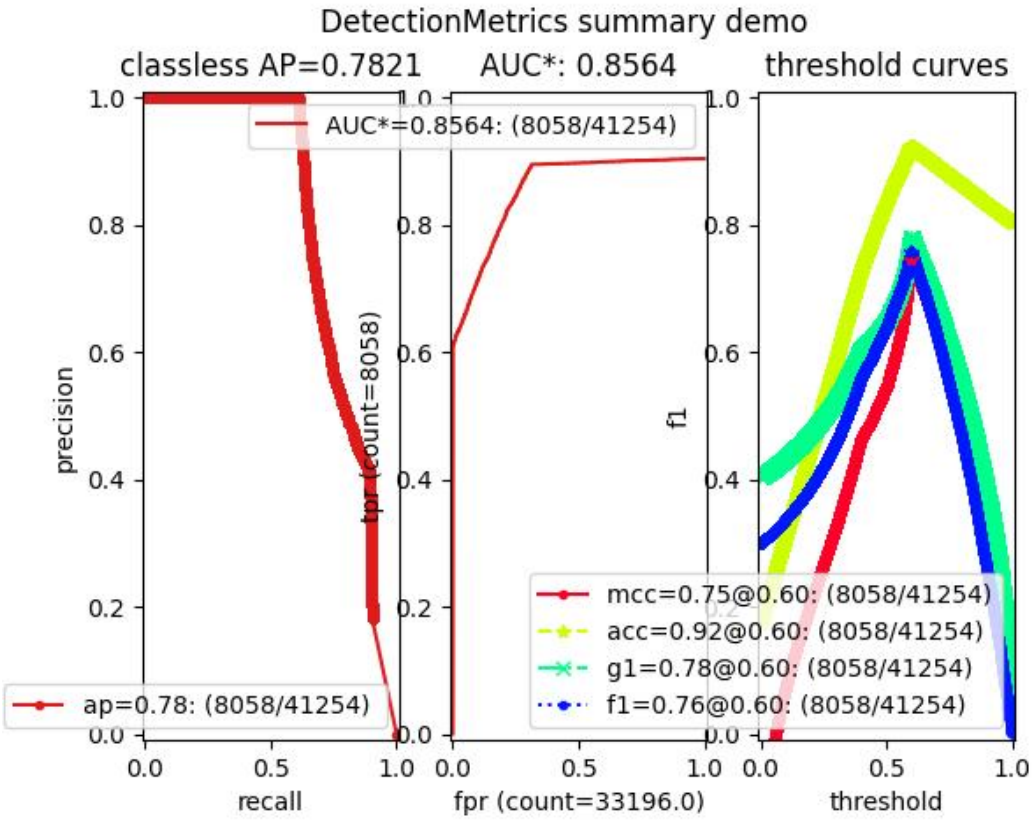
Example

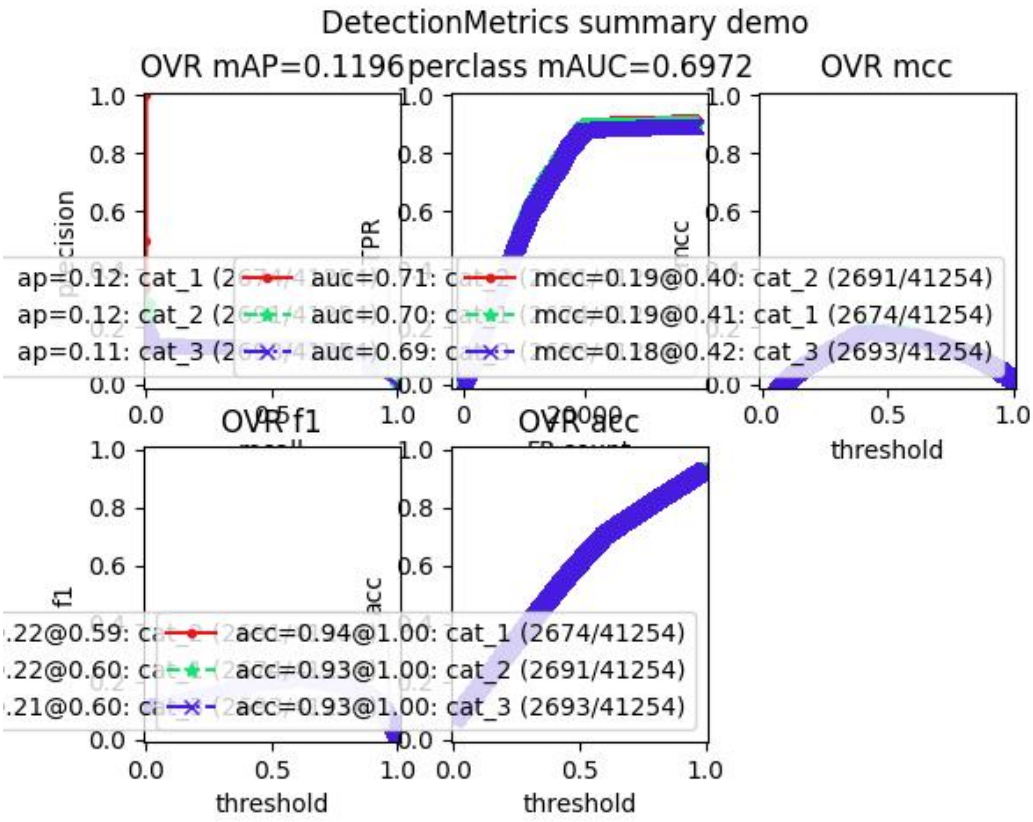
```
>>> from kwcoco.metrics.confusion_vectors import *  # NOQA
>>> from kwcoco.metrics.detect_metrics import DetectionMetrics
>>> dmet = DetectionMetrics.demo(
>>>     n_fp=(0, 1), n_fn=(0, 1), nimgs=32, nboxes=(0, 16),
>>>     classes=3, rng=0, newstyle=1, box_noise=0.5, cls_noise=0.0, score_
↪noise=0.3, with_probs=False)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> summary = dmet.summarize(plot=True, title='DetectionMetrics summary demo',␣
↪with_ovr=True, with_bin=False)
>>> summary['bin_measures']
>>> kwplot.show_if_requested()
```

summarize(*out_dpath=None*, *plot=False*, *title=''*, *with_bin='auto'*, *with_ovr='auto'*)

Example

```
>>> from kwcoco.metrics.confusion_vectors import *  # NOQA
>>> from kwcoco.metrics.detect_metrics import DetectionMetrics
>>> dmet = DetectionMetrics.demo(
>>>     n_fp=(0, 128), n_fn=(0, 4), nimgs=512, nboxes=(0, 32),
>>>     classes=3, rng=0)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> dmet.summarize(plot=True, title='DetectionMetrics summary demo')
>>> kwplot.show_if_requested()
```





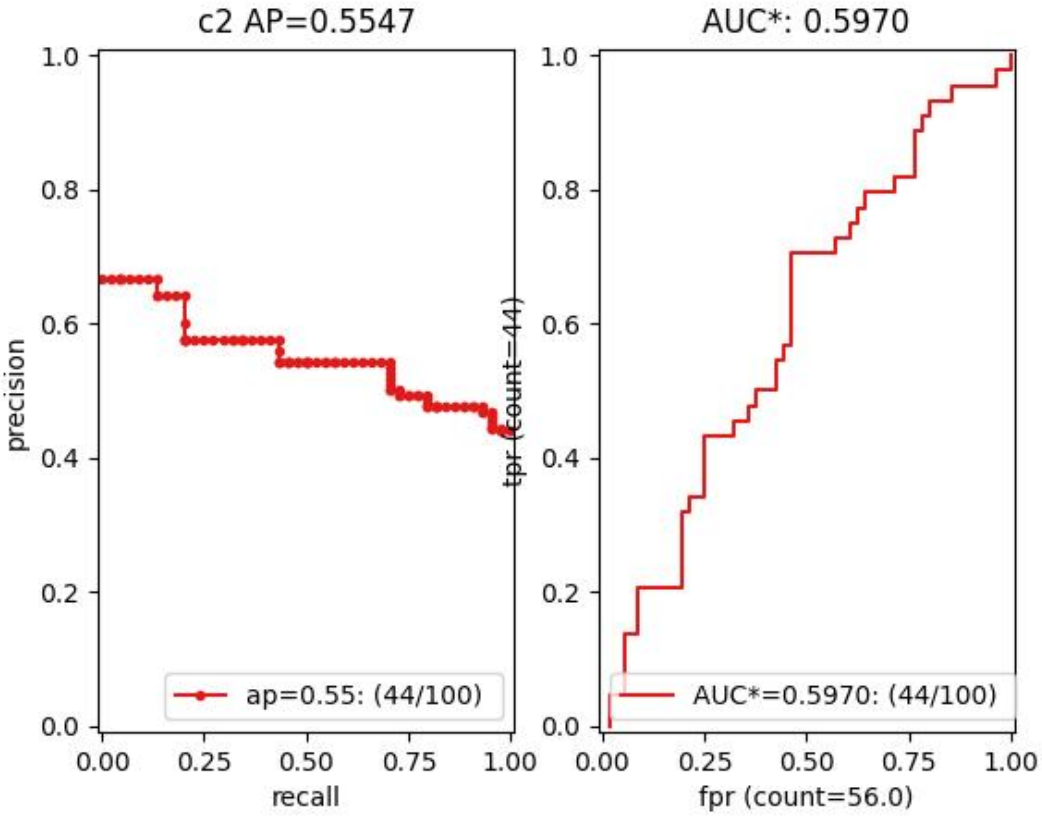
class kwcoco.metrics.**Measures**(*info*)

Bases: NiceRepr, *DictProxy*

Holds accumulated confusion counts, and derived measures

Example

```
>>> from kwcoco.metrics.confusion_vectors import BinaryConfusionVectors  # NOQA
>>> binvecs = BinaryConfusionVectors.demo(n=100, p_error=0.5)
>>> self = binvecs.measures()
>>> print('self = {!r}'.format(self))
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> self.draw(doclf=True)
>>> self.draw(key='pr',  pnum=(1, 2, 1))
>>> self.draw(key='roc', pnum=(1, 2, 2))
>>> kwplot.show_if_requested()
```



property catname

reconstruct()

classmethod from_json(*state*)

summary()

maximized_thresholds()

Returns thresholds that maximize metrics.

counts()

draw(*key=None*, *prefix=''*, ***kw*)

Example

```
>>> # xdoctest: +REQUIRES(module:kwplot)
>>> # xdoctest: +REQUIRES(module:pandas)
>>> from kwcoco.metrics.confusion_vectors import ConfusionVectors  # NOQA
>>> cfsn_vecs = ConfusionVectors.demo()
>>> ovr_cfsn = cfsn_vecs.binarize_ovr(keyby='name')
>>> self = ovr_cfsn.measures()['perclass']
>>> self.draw('mcc', doclf=True, fnum=1)
```

(continues on next page)

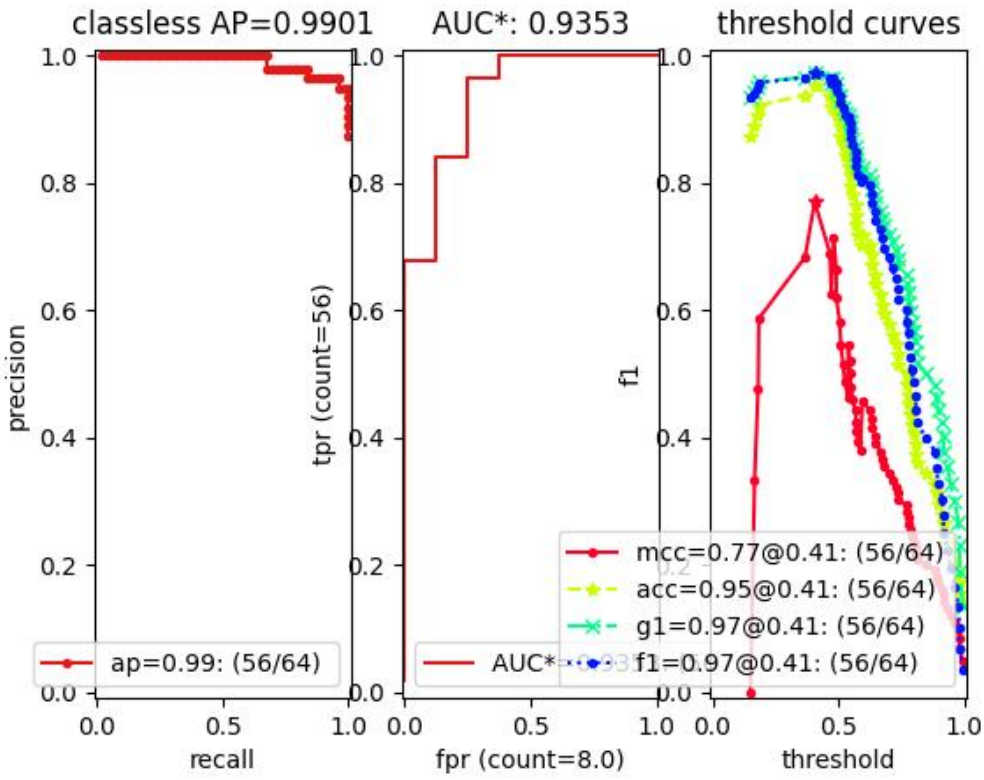
(continued from previous page)

```
>>> self.draw('pr', doclf=1, fnum=2)
>>> self.draw('roc', doclf=1, fnum=3)
```

summary_plot(*fnum=1*, *title=''*, *subplots='auto'*)

Example

```
>>> from kwcoco.metrics.confusion_measures import *  # NOQA
>>> from kwcoco.metrics.confusion_vectors import ConfusionVectors  # NOQA
>>> cfsn_vecs = ConfusionVectors.demo(n=3, p_error=0.5)
>>> binvecs = cfsn_vecs.binarize_classless()
>>> self = binvecs.measures()
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> self.summary_plot()
>>> kwplot.show_if_requested()
```



classmethod demo(***kwargs*)

Create a demo Measures object for testing / demos

Parameters

****kwargs** – passed to *BinaryConfusionVectors.demo()*. some valid keys are: n, rng, p_rue, p_error, p_miss.

classmethod combine(*tocombine*, *precision=None*, *growth=None*, *thresh_bins=None*)

Combine binary confusion metrics

Parameters

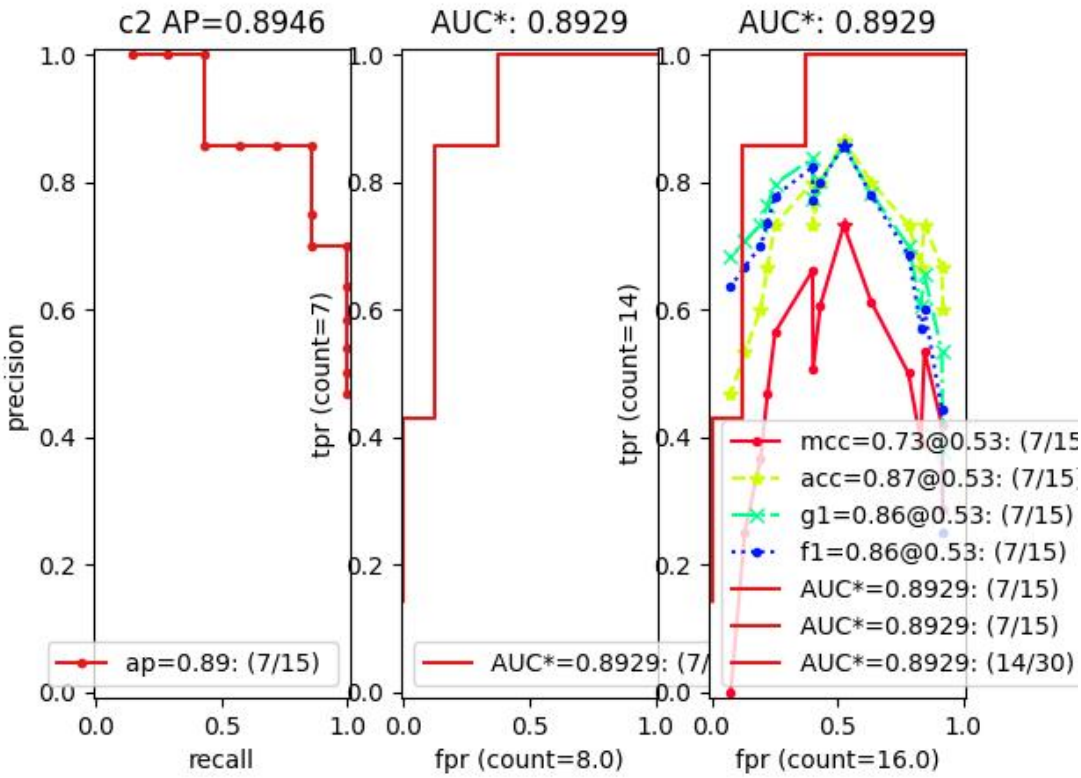
- **tocombine** (*List[Measures]*) – a list of measures to combine into one
- **precision** (*int | None*) – If specified rounds thresholds to this precision which can prevent a RAM explosion when combining a large number of measures. However, this is a lossy operation and will impact the underlying scores. NOTE: use `growth` instead.
- **growth** (*int | None*) – if specified this limits how much the resulting measures are allowed to grow by. If None, growth is unlimited. Otherwise, if growth is 'max', the growth is limited to the maximum length of an input. We might make this more numerical in the future.
- **thresh_bins** (*int*) – Force this many threshold bins.

Returns

kwcoco.metrics.confusion_measures.Measures

Example

```
>>> from kwcoco.metrics.confusion_measures import *  # NOQA
>>> measures1 = Measures.demo(n=15)
>>> measures2 = measures1
>>> tocombine = [measures1, measures2]
>>> new_measures = Measures.combine(tocombine)
>>> new_measures.reconstruct()
>>> print('new_measures = {!r}'.format(new_measures))
>>> print('measures1 = {!r}'.format(measures1))
>>> print('measures2 = {!r}'.format(measures2))
>>> print(ub.repr2(measures1.__json__(), nl=1, sort=0))
>>> print(ub.repr2(measures2.__json__(), nl=1, sort=0))
>>> print(ub.repr2(new_measures.__json__(), nl=1, sort=0))
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.figure(fnum=1)
>>> new_measures.summary_plot()
>>> measures1.summary_plot()
>>> measures1.draw('roc')
>>> measures2.draw('roc')
>>> new_measures.draw('roc')
```



Example

```
>>> # Demonstrate issues that can arrise from choosing a precision
>>> # that is too low  when combining metrics. Breakpoints
>>> # between different metrics can get muddled, but choosing a
>>> # precision that is too high can overwhelm memory.
>>> from kwcoco.metrics.confusion_measures import *  # NOQA
>>> base = ub.map_vals(np.asarray, {
>>>     'tp_count':   [ 1,  1,  2,  2,  2,  2,  3],
>>>     'fp_count':   [ 0,  1,  1,  2,  3,  4,  5],
>>>     'fn_count':   [ 1,  1,  0,  0,  0,  0,  0],
>>>     'tn_count':   [ 5,  4,  4,  3,  2,  1,  0],
>>>     'thresholds': [.0, .0, .0, .0, .0, .0, .0],
>>> })
>>> # Make tiny offsets to thresholds
>>> rng = kwarray.ensure_rng(0)
>>> n = len(base['thresholds'])
>>> offsets = [
>>>     sorted(rng.rand(n) * 10 ** -rng.randint(4, 7))[::-1]
>>>     for _ in range(20)
>>> ]
>>> tocombine = []
>>> for offset in offsets:
>>>     base_n = base.copy()
>>>     base_n['thresholds'] += offset
```

(continues on next page)

(continued from previous page)

```
>>>     measures_n = Measures(base_n).reconstruct()
>>>     tocombine.append(measures_n)
>>> for precision in [6, 5, 2]:
>>>     combo = Measures.combine(tocombine, precision=precision).reconstruct()
>>>     print('precision = {!r}'.format(precision))
>>>     print('combo = {}'.format(ub.repr2(combo, nl=1)))
>>>     print('num_thresholds = {}'.format(len(combo['thresholds'])))
>>> for growth in [None, 'max', 'log', 'root', 'half']:
>>>     combo = Measures.combine(tocombine, growth=growth).reconstruct()
>>>     print('growth = {!r}'.format(growth))
>>>     print('combo = {}'.format(ub.repr2(combo, nl=1)))
>>>     print('num_thresholds = {}'.format(len(combo['thresholds'])))
>>>     #print(combo.counts().pandas())
```

Example

```
>>> # Test case: combining a single measures should leave it unchanged
>>> from kwcoco.metrics.confusion_measures import *  # NOQA
>>> measures = Measures.demo(n=40, p_true=0.2, p_error=0.4, p_miss=0.6)
>>> df1 = measures.counts().pandas().fillna(0)
>>> print(df1)
>>> tocombine = [measures]
>>> combo = Measures.combine(tocombine)
>>> df2 = combo.counts().pandas().fillna(0)
>>> print(df2)
>>> assert np.allclose(df1, df2)
```

```
>>> combo = Measures.combine(tocombine, thresh_bins=2)
>>> df3 = combo.counts().pandas().fillna(0)
>>> print(df3)
```

```
>>> # I am NOT sure if this is correct or not
>>> thresh_bins = 20
>>> combo = Measures.combine(tocombine, thresh_bins=thresh_bins)
>>> df4 = combo.counts().pandas().fillna(0)
>>> print(df4)
```

```
>>> combo = Measures.combine(tocombine, thresh_bins=np.linspace(0, 1, 20))
>>> df4 = combo.counts().pandas().fillna(0)
>>> print(df4)
```

assert np.allclose(combo[‘thresholds’], measures[‘thresholds’]) assert np.allclose(combo[‘fp_count’], measures[‘fp_count’]) assert np.allclose(combo[‘tp_count’], measures[‘tp_count’]) assert np.allclose(combo[‘tp_count’], measures[‘tp_count’])

globals().update(xdev.get_func_kwargs(Measures.combine))

Example

```
>>> # Test degenerate case
>>> from kwcoco.metrics.confusion_measures import *  # NOQA
>>> tocombine = [
>>>     {'fn_count': [0.0], 'fp_count': [359980.0], 'thresholds': [0.0], 'tn_
↪count': [0.0], 'tp_count': [7747.0]},
>>>     {'fn_count': [0.0], 'fp_count': [360849.0], 'thresholds': [0.0], 'tn_
↪count': [0.0], 'tp_count': [424.0]},
>>>     {'fn_count': [0.0], 'fp_count': [367003.0], 'thresholds': [0.0], 'tn_
↪count': [0.0], 'tp_count': [991.0]},
>>>     {'fn_count': [0.0], 'fp_count': [367976.0], 'thresholds': [0.0], 'tn_
↪count': [0.0], 'tp_count': [1017.0]},
>>>     {'fn_count': [0.0], 'fp_count': [676338.0], 'thresholds': [0.0], 'tn_
↪count': [0.0], 'tp_count': [7067.0]},
>>>     {'fn_count': [0.0], 'fp_count': [676348.0], 'thresholds': [0.0], 'tn_
↪count': [0.0], 'tp_count': [7406.0]},
>>>     {'fn_count': [0.0], 'fp_count': [676626.0], 'thresholds': [0.0], 'tn_
↪count': [0.0], 'tp_count': [7858.0]},
>>>     {'fn_count': [0.0], 'fp_count': [676693.0], 'thresholds': [0.0], 'tn_
↪count': [0.0], 'tp_count': [10969.0]},
>>>     {'fn_count': [0.0], 'fp_count': [677269.0], 'thresholds': [0.0], 'tn_
↪count': [0.0], 'tp_count': [11188.0]},
>>>     {'fn_count': [0.0], 'fp_count': [677331.0], 'thresholds': [0.0], 'tn_
↪count': [0.0], 'tp_count': [11734.0]},
>>>     {'fn_count': [0.0], 'fp_count': [677395.0], 'thresholds': [0.0], 'tn_
↪count': [0.0], 'tp_count': [11556.0]},
>>>     {'fn_count': [0.0], 'fp_count': [677418.0], 'thresholds': [0.0], 'tn_
↪count': [0.0], 'tp_count': [11621.0]},
>>>     {'fn_count': [0.0], 'fp_count': [677422.0], 'thresholds': [0.0], 'tn_
↪count': [0.0], 'tp_count': [11424.0]},
>>>     {'fn_count': [0.0], 'fp_count': [677648.0], 'thresholds': [0.0], 'tn_
↪count': [0.0], 'tp_count': [9804.0]},
>>>     {'fn_count': [0.0], 'fp_count': [677826.0], 'thresholds': [0.0], 'tn_
↪count': [0.0], 'tp_count': [2470.0]},
>>>     {'fn_count': [0.0], 'fp_count': [677834.0], 'thresholds': [0.0], 'tn_
↪count': [0.0], 'tp_count': [2470.0]},
>>>     {'fn_count': [0.0], 'fp_count': [677835.0], 'thresholds': [0.0], 'tn_
↪count': [0.0], 'tp_count': [2470.0]},
>>>     {'fn_count': [11123.0, 0.0], 'fp_count': [0.0, 676754.0], 'thresholds':␣
↪[0.0002442002442002442, 0.0], 'tn_count': [676754.0, 0.0], 'tp_count': [2.0,␣
↪11125.0]},
>>>     {'fn_count': [7738.0, 0.0], 'fp_count': [0.0, 676466.0], 'thresholds':␣
↪[0.0002442002442002442, 0.0], 'tn_count': [676466.0, 0.0], 'tp_count': [0.0,␣
↪7738.0]},
>>>     {'fn_count': [8653.0, 0.0], 'fp_count': [0.0, 676341.0], 'thresholds':␣
↪[0.0002442002442002442, 0.0], 'tn_count': [676341.0, 0.0], 'tp_count': [0.0,␣
↪8653.0]},
>>> ]
>>> thresh_bins = np.linspace(0, 1, 4)
>>> combo = Measures.combine(tocombine, thresh_bins=thresh_bins).reconstruct()
>>> print('tocombine = {}'.format(ub.repr2(tocombine, nl=2)))
>>> print('thresh_bins = {!r}'.format(thresh_bins))
```

(continues on next page)

(continued from previous page)

```
>>> print(ub.repr2(combo.__json__(), nl=1))
>>> for thresh_bins in [4096, 1]:
>>>     combo = Measures.combine(tocombine, thresh_bins=thresh_bins).
↪reconstruct()
>>>     print('thresh_bins = {!r}'.format(thresh_bins))
>>>     print('combo = {}'.format(ub.repr2(combo, nl=1)))
>>>     print('num_thresholds = {}'.format(len(combo['thresholds'])))
>>> for precision in [6, 5, 2]:
>>>     combo = Measures.combine(tocombine, precision=precision).reconstruct()
>>>     print('precision = {!r}'.format(precision))
>>>     print('combo = {}'.format(ub.repr2(combo, nl=1)))
>>>     print('num_thresholds = {}'.format(len(combo['thresholds'])))
>>> for growth in [None, 'max', 'log', 'root', 'half']:
>>>     combo = Measures.combine(tocombine, growth=growth).reconstruct()
>>>     print('growth = {!r}'.format(growth))
>>>     print('combo = {}'.format(ub.repr2(combo, nl=1)))
>>>     print('num_thresholds = {}'.format(len(combo['thresholds'])))
```

class kwcoco.metrics.**OneVsRestConfusionVectors**(*cx_to_binvecs*, *classes*)

Bases: NiceRepr

Container for multiple one-vs-rest binary confusion vectors

Variables

- **cx_to_binvecs** –
- **classes** –

Example

```
>>> from kwcoco.metrics import DetectionMetrics
>>> dmet = DetectionMetrics.demo(
>>>     nimgs=10, nboxes=(0, 10), n_fp=(0, 1), classes=3)
>>> cfsn_vecs = dmet.confusion_vectors()
>>> self = cfsn_vecs.binarize_ovr(keyby='name')
>>> print('self = {!r}'.format(self))
```

classmethod demo()

Parameters

****kwargs** – See *kwcoco.metrics.DetectionMetrics.demo()*

Returns

ConfusionVectors

keys()

measures(*stabalize_thresh=7*, *fp_cutoff=None*, *monotonic_ppv=True*, *ap_method='pycocotools'*)

Creates binary confusion measures for every one-versus-rest category.

Parameters

- **stabalize_thresh** (*int, default=7*) – if fewer than this many data points inserts dummy stabilization data so curves can still be drawn.

- **fp_cutoff** (*int, default=None*) – maximum number of false positives in the truncated roc curves. None is equivalent to `float('inf')`
- **monotonic_ppv** (*bool, default=True*) – if True ensures that precision is always increasing as recall decreases. This is done in pycocotools scoring, but I'm not sure its a good idea.

SeeAlso:
BinaryConfusionVectors.measures()

Example

```
>>> self = OneVsRestConfusionVectors.demo()
>>> thresh_result = self.measures()['perclass']
```

ovr_classification_report()

class kwcoco.metrics.**PerClass_Measures**(*cx_to_info*)

Bases: NiceRepr, *DictProxy*

summary()

classmethod from_json(*state*)

draw(*key='mcc'*, *prefix=''*, ***kw*)

Example

```
>>> # xdoctest: +REQUIRES(module:kwplot)
>>> from kwcoco.metrics.confusion_vectors import ConfusionVectors  # NOQA
>>> cfsn_vecs = ConfusionVectors.demo()
>>> ovr_cfsn = cfsn_vecs.binarize_ovr(keyby='name')
>>> self = ovr_cfsn.measures()['perclass']
>>> self.draw('mcc', doclf=True, fnum=1)
>>> self.draw('pr', doclf=1, fnum=2)
>>> self.draw('roc', doclf=1, fnum=3)
```

draw_roc(*prefix=''*, ***kw*)

draw_pr(*prefix=''*, ***kw*)

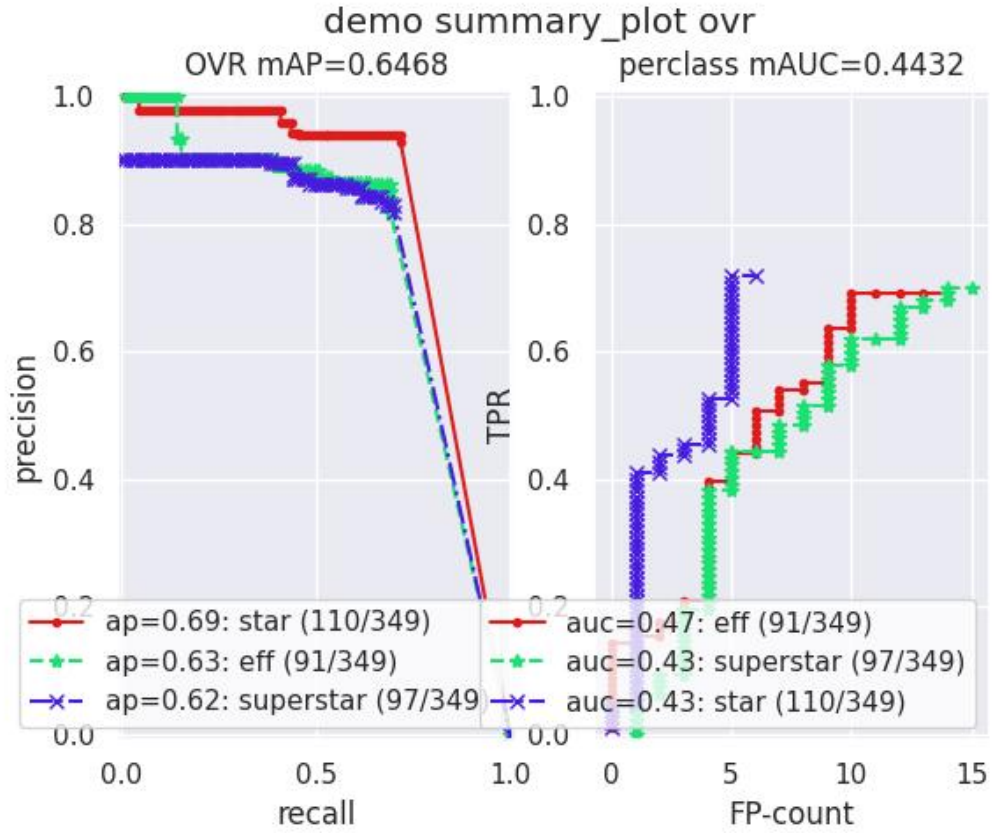
summary_plot(*fnum=1*, *title=''*, *subplots='auto'*)

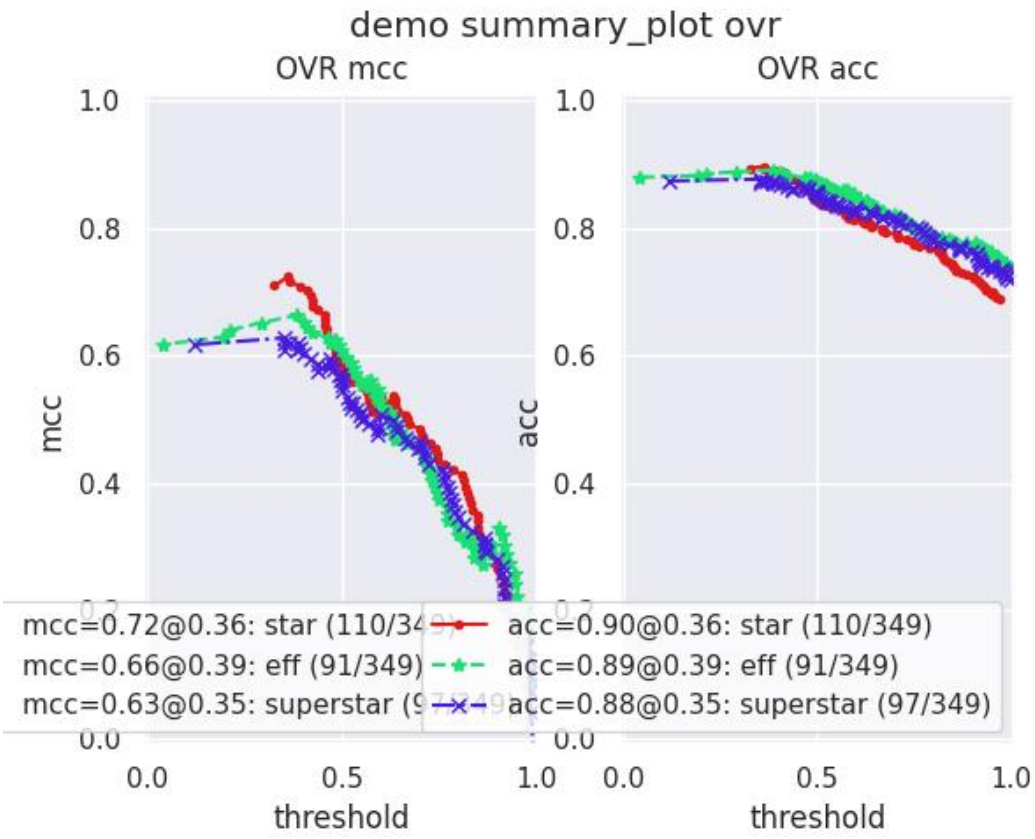
CommandLine

```
python ~/code/kwcoco/kwcoco/metrics/confusion_measures.py PerClass_Measures.
↪summary_plot --show
```

Example

```
>>> from kwcoco.metrics.confusion_measures import *  # NOQA
>>> from kwcoco.metrics.detect_metrics import DetectionMetrics
>>> dmet = DetectionMetrics.demo(
>>>     n_fp=(0, 1), n_fn=(0, 3), nimgs=32, nboxes=(0, 32),
>>>     classes=3, rng=0, newstyle=1, box_noise=0.7, cls_noise=0.2, score_
↪noise=0.3, with_probs=False)
>>> cfsn_vecs = dmet.confusion_vectors()
>>> ovr_cfsn = cfsn_vecs.binarize_ovr(keyby='name', ignore_classes=['vector',
↪'raster'])
>>> self = ovr_cfsn.measures()['perclass']
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> import seaborn as sns
>>> sns.set()
>>> self.summary_plot(title='demo summary_plot ovr', subplots=['pr', 'roc'])
>>> kwplot.show_if_requested()
>>> self.summary_plot(title='demo summary_plot ovr', subplots=['mcc', 'acc'],␣
↪fnum=2)
```





kwcoco.metrics.**eval_detections_cli**(**kw)

DEPRECATED USE *kwcoco eval* instead

CommandLine

```
xdoctest -m ~/code/kwcoco/kwcoco/metrics/detect_metrics.py eval_detections_cli
```

2.1.1.6 kwcoco.util package

2.1.1.6.1 Subpackages

2.1.1.6.1.1 kwcoco.util.delayed_ops package

2.1.1.6.1.2 Submodules

2.1.1.6.1.3 kwcoco.util.delayed_ops.delayed_base module

Abstract nodes

class kwcoco.util.delayed_ops.delayed_base.**DelayedOperation**

Bases: NiceRepr

nesting()

Returns
Dict[str, dict]

as_graph()

Returns
networkx.DiGraph

write_network_text(*with_labels=True*)

property shape

Returns: None | Tuple[int | None, ...]

children()

Yields
Any

prepare()

If metadata is missing, perform minimal IO operations in order to prepopulate metadata that could help us better optimize the operation tree.

Returns
DelayedOperation2

finalize(*prepare=True*, *optimize=True*, ***kwargs*)

Evaluate the operation tree in full.

Parameters

- **prepare** (*bool*) – ensure prepare is called to ensure metadata exists if possible before optimizing. Defaults to True.
- **optimize** (*bool*) – ensure the graph is optimized before loading. Default to True.
- ****kwargs** – for backwards compatibility, these will allow for in-place modification of select nested parameters. In general these should not be used, and may be deprecated.

Returns
ArrayLike

Notes

Do not overload this method. Overload `DelayedOperation2._finalize()` instead.

optimize()

Returns
DelayedOperation2

class kwcoco.util.delayed_ops.delayed_base.**DelayedNaryOperation**(*parts*)

Bases: *DelayedOperation*

For operations that have multiple input arrays

children()

Yields
Any

class kwcoco.util.delayed_ops.delayed_base.**DelayedUnaryOperation**(*subdata*)

Bases: *DelayedOperation*

For operations that have a single input array

children()

Yields
Any

kwcoco.util.delayed_ops.delayed_base.**DelayedOperation2**

alias of *DelayedOperation*

kwcoco.util.delayed_ops.delayed_base.**DelayedUnaryOperation2**

alias of *DelayedUnaryOperation*

kwcoco.util.delayed_ops.delayed_base.**DelayedNaryOperation2**

alias of *DelayedNaryOperation*

2.1.1.6.1.4 kwcoco.util.delayed_ops.delayed_leafs module

Terminal nodes

class kwcoco.util.delayed_ops.delayed_leafs.**DelayedImageLeaf**(*subdata=None*, *dsize=None*, *channels=None*)

Bases: *DelayedImage*

get_transform_from_leaf()

Returns the transformation that would align data with the leaf

Returns
kwimage.Affine

optimize()

class kwcoco.util.delayed_ops.delayed_leafs.**DelayedLoad**(*fpath*, *channels=None*, *dsize=None*, *nodata_method=None*)

Bases: *DelayedImageLeaf*

Reads an image from disk.

If a gdal backend is available, and the underlying image is in the appropriate formate (e.g. COG) this will return a lazy reference that enables fast overviews and crops.

Example

```
>>> from kwcoco.util.delayed_ops import *  # NOQA
>>> self = DelayedLoad.demo(dsize=(16, 16)).prepare()
>>> data1 = self.finalize()
```

Example

```
>>> # xdoctest: +REQUIRES(module:osgeo)
>>> # Demo code to develop support for overviews
>>> from kwcoco.util.delayed_ops import *  # NOQA
>>> import kwimage
>>> import ubelt as ub
>>> fpath = kwimage.grab_test_image_fpath(overviews=3)
>>> self = DelayedLoad(fpath, channels='r|g|b').prepare()
>>> print(f'self={self}')
>>> print('self.meta = {}'.format(ub.repr2(self.meta, nl=1)))
>>> quantization = {
>>>     'quant_max': 255,
>>>     'nodata': 0,
>>> }
>>> node0 = self
>>> node1 = node0.get_overview(2)
>>> node2 = node1[13:900, 11:700]
>>> node3 = node2.dequantize(quantization)
>>> node4 = node3.warp({'scale': 0.05})
>>> #
>>> data0 = node0._validate().finalize()
>>> data1 = node1._validate().finalize()
>>> data2 = node2._validate().finalize()
>>> data3 = node3._validate().finalize()
>>> data4 = node4._validate().finalize()
>>> node4.write_network_text()
```

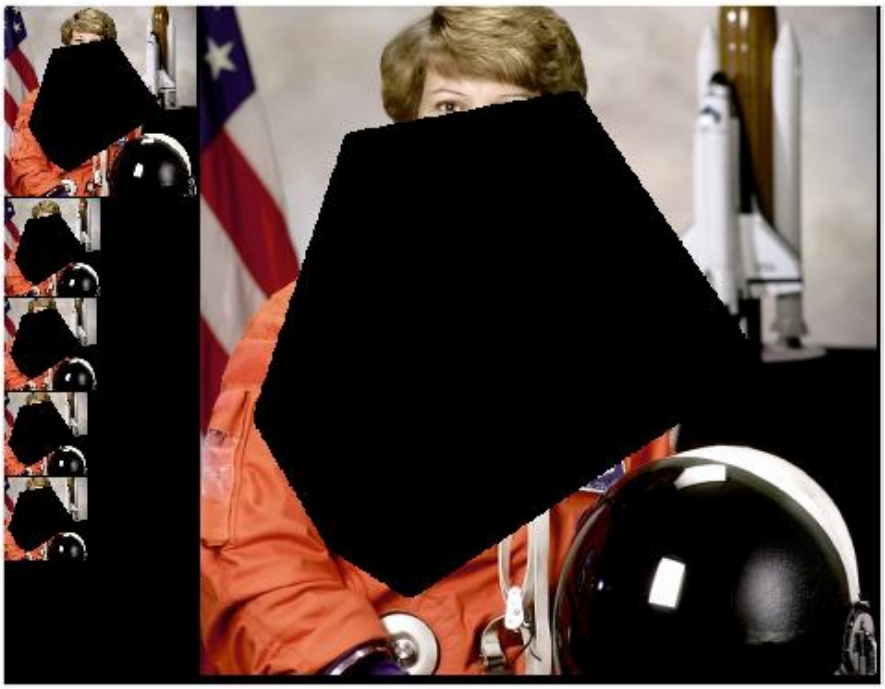
Example

```
>>> # xdoctest: +REQUIRES(module:osgeo)
>>> # Test delayed ops with int16 and nodata values
>>> from kwcoco.util.delayed_ops import *  # NOQA
>>> import kwimage
>>> from kwcoco.util.delayed_ops.helpers import quantize_float01
>>> import ubelt as ub
>>> dpath = ub.Path.appdir('kwcoco/tests/test_delay_nodata').ensuredir()
>>> fpath = dpath / 'data.tif'
>>> data = kwimage.ensure_float01(kwimage.grab_test_image())
>>> poly = kwimage.Polygon.random(rng=321032).scale(data.shape[0])
>>> poly.fill(data, np.nan)
>>> data_uint16, quantization = quantize_float01(data)
>>> nodata = quantization['nodata']
>>> kwimage.imwrite(fpath, data_uint16, nodata=nodata, backend='gdal', overviews=3)
>>> # Test loading the data
>>> self = DelayedLoad(fpath, channels='r|g|b', nodata_method='float').prepare()
>>> node0 = self
>>> node1 = node0.dequantize(quantization)
>>> node2 = node1.warp({'scale': 0.51}, interpolation='lanczos')
>>> node3 = node2[13:900, 11:700]
>>> node4 = node3.warp({'scale': 0.9}, interpolation='lanczos')
```

(continues on next page)

(continued from previous page)

```
>>> node4.write_network_text()
>>> node5 = node4.optimize()
>>> node5.write_network_text()
>>> node6 = node5.warp({'scale': 8}, interpolation='lanczos').optimize()
>>> node6.write_network_text()
>>> #
>>> data0 = node0._validate().finalize()
>>> data1 = node1._validate().finalize()
>>> data2 = node2._validate().finalize()
>>> data3 = node3._validate().finalize()
>>> data4 = node4._validate().finalize()
>>> data5 = node5._validate().finalize()
>>> data6 = node6._validate().finalize()
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> stack1 = kwimage.stack_images([data1, data2, data3, data4, data5])
>>> stack2 = kwimage.stack_images([stack1, data6], axis=1)
>>> kwplot.imshow(stack2)
```



property fpath

classmethod demo(*key='astro'*, *dsize=None*, *channels=None*)

prepare()

If metadata is missing, perform minimal IO operations in order to prepopulate metadata that could help us better optimize the operation tree.

Returns
DelayedLoad

class kwcoco.util.delayed_ops.delayed_leafs.**DelayedNans**(*dsize=None*, *channels=None*)

Bases: *DelayedImageLeaf*

Constructs nan channels as needed

Example

self = DelayedNans((10, 10), channel_spec.FusedChannelSpec.coerce('rgb')) region_slices = (slice(5, 10), slice(1, 12)) delayed = self.crop(region_slices)

Example

```
>>> from kwcoco.util.delayed_ops import *  # NOQA
>>> import kwcoco
>>> dsize = (307, 311)
>>> c1 = DelayedNans(dsize=dsize, channels='foo')
>>> c2 = DelayedLoad.demo('astro', dsize=dsize, channels='R|G|B').prepare()
>>> cat = DelayedChannelConcat([c1, c2])
>>> warped_cat = cat.warp({'scale': 1.07}, dsize=(328, 332))._validate()
>>> warped_cat._validate().optimize().finalize()
```

class kwcoco.util.delayed_ops.delayed_leafs.**DelayedIdentity**(*data*, *channels=None*, *dsize=None*)

Bases: *DelayedImageLeaf*

Returns an ndarray as-is

Example

self = DelayedNans((10, 10), channel_spec.FusedChannelSpec.coerce('rgb')) region_slices = (slice(5, 10), slice(1, 12)) delayed = self.crop(region_slices)

Example

```
>>> from kwcoco.util.delayed_ops import *  # NOQA
>>> import kwcoco
>>> arr = kwimage.checkerboard()
>>> self = DelayedIdentity(arr, channels='gray')
>>> warp = self.warp({'scale': 1.07})
>>> warp.optimize().finalize()
```

kwcoco.util.delayed_ops.delayed_leafs.**DelayedIdentity2**

alias of *DelayedIdentity*

kwcoco.util.delayed_ops.delayed_leafs.**DelayedNans2**

alias of *DelayedNans*

kwcoco.util.delayed_ops.delayed_leafs.**DelayedLoad2**

alias of *DelayedLoad*

kwcoco.util.delayed_ops.delayed_leafs.**DelayedImageLeaf2**

alias of *DelayedImageLeaf*

2.1.1.6.1.5 kwcoco.util.delayed_ops.delayed_nodes module

Intermediate operations

class kwcoco.util.delayed_ops.delayed_nodes.**DelayedStack**(*parts*, *axis*)

Bases: *DelayedNaryOperation*

Stacks multiple arrays together.

property shape

Returns: None | Tuple[int | None, ...]

class kwcoco.util.delayed_ops.delayed_nodes.**DelayedConcat**(*parts*, *axis*)

Bases: *DelayedNaryOperation*

Stacks multiple arrays together.

property shape

Returns: None | Tuple[int | None, ...]

class kwcoco.util.delayed_ops.delayed_nodes.**DelayedFrameStack**(*parts*)

Bases: *DelayedStack*

Stacks multiple arrays together.

class kwcoco.util.delayed_ops.delayed_nodes.**ImageOpsMixin**

Bases: object

crop(*space_slice=None*, *chan_idxs=None*, *clip=True*, *wrap=True*, *pad=0*)

Crops an image along integer pixel coordinates.

Parameters

- **space_slice** (*Tuple[slice, slice]*) – y-slice and x-slice.
- **chan_idxs** (*List[int]*) – indexes of bands to take
- **clip** (*bool*) – if True, the slice is interpreted normally, where it won't go past the image extent, otherwise slicing into negative regions or past the image bounds will result in padding. Defaults to True.
- **wrap** (*bool*) – if True, negative indexes "wrap around", otherwise they are treated as is. Defaults to True.
- **pad** (*int | List[Tuple[int, int]]*) – if specified, applies extra padding

Returns

DelayedImage

Example

```
>>> from kwcoco.util.delayed_ops import DelayedLoad
>>> import kwimage
>>> self = DelayedLoad.demo().prepare()
>>> self = self.dequantize({'quant_max': 255})
>>> self = self.warp({'scale': 1 / 2})
>>> pad = 0
>>> h, w = space_dims = self.dsize[::-1]
>>> grid = list(ub.named_product({
>>>     'left': [0, -64], 'right': [0, 64],
>>>     'top': [0, -64], 'bot': [0, 64],}))
>>> grid += [
>>>     {'left': 64, 'right': -64, 'top': 0, 'bot': 0},
>>>     {'left': 64, 'right': 64, 'top': 0, 'bot': 0},
>>>     {'left': 0, 'right': 0, 'top': 64, 'bot': -64},
>>>     {'left': 64, 'right': -64, 'top': 64, 'bot': -64},
>>> ]
>>> crops = []
>>> for pads in grid:
>>>     space_slice = (slice(pads['top'], h + pads['bot']),
>>>                    slice(pads['left'], w + pads['right']))
>>>     delayed = self.crop(space_slice)
>>>     crop = delayed.finalize()
>>>     yyxx = kwimage.Boxes.from_slice(space_slice, wrap=False, clip=0).
↪toformat('_yyxx').data[0]
>>>     title = '[{}:{}, {}:{}]'.format(*yyxx)
>>>     crop_canvas = kwimage.draw_header_text(crop, title, fit=True, bg_color=
↪'kw_darkgray')
>>>     crops.append(crop_canvas)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> canvas = kwimage.stack_images_grid(crops, pad=16, bg_value='kw_darkgreen')
>>> canvas = kwimage.fill_nans_with_checkers(canvas)
>>> kwplot.imshow(canvas, title='Normal Slicing: Cropped Images With␣
↪Wrap+Clipped Slices', doclf=1, fnum=1)
>>> kwplot.show_if_requested()
```

Example

```
>>> # Demo the case with pads / no-clips / no-wraps
>>> from kwcoco.util.delayed_ops import DelayedLoad
>>> import kwimage
>>> self = DelayedLoad.demo().prepare()
>>> self = self.dequantize({'quant_max': 255})
>>> self = self.warp({'scale': 1 / 2})
>>> pad = [(64, 128), (32, 96)]
>>> pad = [(0, 20), (0, 0)]
>>> pad = 0
>>> pad = 8
>>> h, w = space_dims = self.dsize[::-1]
>>> grid = list(ub.named_product({
>>>     'left': [0, -64], 'right': [0, 64],
>>>     'top': [0, -64], 'bot': [0, 64],}))
>>> grid += [
>>>     {'left': 64, 'right': -64, 'top': 0, 'bot': 0},
>>>     {'left': 64, 'right': 64, 'top': 0, 'bot': 0},
>>>     {'left': 0, 'right': 0, 'top': 64, 'bot': -64},
>>>     {'left': 64, 'right': -64, 'top': 64, 'bot': -64},
>>> ]
>>> crops = []
>>> for pads in grid:
>>>     space_slice = (slice(pads['top'], h + pads['bot']),
```

(continues on next page)

(continued from previous page)

```
>>>                     slice(pads['left'], w + pads['right']))
>>>     delayed = self._padded_crop(space_slice, pad=pad)
>>>     crop = delayed.finalize(optimize=1)
>>>     yyxx = kwimage.Boxes.from_slice(space_slice, wrap=False, clip=0).
↪toformat('_yyxx').data[0]
>>>     title = '[{}:{}, {}:{}]'.format(*yyxx)
>>>     if pad:
>>>         title += f'{chr(10)}pad={pad}'
>>>     crop_canvas = kwimage.draw_header_text(crop, title, fit=True, bg_color=
↪'kw_darkgray')
>>>     crops.append(crop_canvas)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> canvas = kwimage.stack_images_grid(crops, pad=16, bg_value='kw_darkgreen',␣
↪resize='smaller')
>>> canvas = kwimage.fill_nans_with_checkers(canvas)
>>> kwplot.imshow(canvas, title='Negative Slicing: Cropped Images With␣
↪clip=False wrap=False', doclf=1, fnum=2)
>>> kwplot.show_if_requested()
```

warp(*transform*, *dsize='auto'*, *antialias=True*, *interpolation='linear'*, *border_value='auto'*)

Applys an affine transformation to the image

Parameters

- **transform** (*ndarray | dict | kwimage.Affine*) – a coercable affine matrix. See `kwimage.Affine` for details on what can be coerced.
- **dsize** (*Tuple[int, int] | str*) – The width / height of the output canvas. If 'auto', dsize is computed such that the positive coordinates of the warped image will fit in the new canvas. In this case, any pixel that maps to a negative coordinate will be clipped. This has the property that the input transformation is not modified.
- **antialias** (*bool*) – if True determines if the transform is downsampling and applies antialiasing via gaussian a blur. Defaults to False
- **interpolation** (*str*) – interpolation code or cv2 integer. Interpolation codes are linear, nearest, cubic, lancsoz, and area. Defaults to "linear".
- **border_value** (*int | float | str*) – if auto will be nan for float and 0 for int.

Returns
DelayedImage

scale(*scale*, *dsize='auto'*, *antialias=True*, *interpolation='linear'*, *border_value='auto'*)

An alias for self.warp({"scale": scale}, ...)

dequantize(*quantization*)

Rescales image intensities from int to floats.

Parameters
quantization (*Dict[str, Any]*) – see *kwcoco.util.delayed_ops.helpers.dequantize()*

Returns
DelayedDequantize

get_overview(*overview*)

Downsamples an image by a factor of two.

Parameters
overview (*int*) – the overview to use (assuming it exists)

Returns
DelayedOverview

as_xarray()

Returns
DelayedAsXarray

class kwcoco.util.delayed_ops.delayed_nodes.**DelayedChannelConcat**(*parts*, *dsize=None*)

Bases: *ImageOpsMixin*, *DelayedConcat*

Stacks multiple arrays together.

CommandLine

```
xdoctest -m /home/joncrall/code/kwcoco/kwcoco/util/delayed_ops/delayed_nodes.py␣
↪DelayedChannelConcat:1
```

Example

```
>>> from kwcoco.util.delayed_ops import *  # NOQA
>>> from kwcoco.util.delayed_ops.delayed_leafs import DelayedLoad
>>> import kwcoco
>>> dsize = (307, 311)
>>> c1 = DelayedNans(dsize=dsize, channels='foo')
>>> c2 = DelayedLoad.demo('astro', dsize=dsize, channels='R|G|B').prepare()
>>> cat = DelayedChannelConcat([c1, c2])
>>> warped_cat = cat.warp({'scale': 1.07}, dsize=(328, 332))
>>> warped_cat._validate()
>>> warped_cat.finalize()
```

Example

```
>>> # Test case that failed in initial implementation
>>> # Due to incorrectly pushing channel selection under the concat
>>> from kwcoco.util.delayed_ops import *  # NOQA
>>> import kwimage
>>> fpath = kwimage.grab_test_image_fpath()
>>> base1 = DelayedLoad(fpath, channels='r|g|b').prepare()
>>> base2 = DelayedLoad(fpath, channels='x|y|z').prepare().scale(2)
>>> base3 = DelayedLoad(fpath, channels='i|j|k').prepare().scale(2)
>>> bands = [base2, base1[:, :, 0].scale(2).evaluate(),
>>>          base1[:, :, 1].evaluate().scale(2),
>>>          base1[:, :, 2].evaluate().scale(2), base3]
>>> delayed = DelayedChannelConcat(bands)
>>> delayed = delayed.warp({'scale': 2})
>>> delayed = delayed[0:100, 0:55, [0, 2, 4]]
>>> delayed.write_network_text()
>>> delayed.optimize()
```

property channels

Returns: None | kwcoco.FusedChannelSpec

property shape

Returns: Tuple[int | None, int | None, int | None]

optimize()

Returns
DelayedImage

take_channels(*channels*)

This method returns a subset of the vision data with only the specified bands / channels.

Parameters

channels (*List[int] | slice | channel_spec.FusedChannelSpec*) – List of integers indexes, a slice, or a channel spec, which is typically a pipe (|) delimited list of channel codes. See *kwcoco.ChannelSpec* for more detials.

Returns

a delayed vision operation that only operates on the following channels.

Return type

DelayedArray

Example

```
>>> from kwcoco.util.delayed_ops.delayed_nodes import *  # NOQA
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('vidshapes8-multispectral')
>>> self = delayed = dset.coco_image(1).delay(mode=1)
>>> channels = 'B11|B8|B1|B10'
>>> new = self.take_channels(channels)
```

Example

```
>>> # Complex case
>>> import kwcoco
>>> from kwcoco.util.delayed_ops.delayed_nodes import *  # NOQA
>>> from kwcoco.util.delayed_ops.delayed_leafs import DelayedLoad
>>> dset = kwcoco.CocoDataset.demo('vidshapes8-multispectral')
>>> delayed = dset.coco_image(1).delay(mode=1)
>>> astro = DelayedLoad.demo('astro', channels='r|g|b').prepare()
>>> aligned = astro.warp(kwimage.Affine.scale(600 / 512), dsize='auto')
>>> self = combo = DelayedChannelConcat(delayed.parts + [aligned])
>>> channels = 'B1|r|B8|g'
>>> new = self.take_channels(channels)
>>> new_cropped = new.crop((slice(10, 200), slice(12, 350)))
>>> new_opt = new_cropped.optimize()
>>> datas = new_opt.finalize()
>>> if 1:
>>>     new_cropped.write_network_text(with_labels='name')
>>>     new_opt.write_network_text(with_labels='name')
>>> vizable = kwimage.normalize_intensity(datas, axis=2)
>>> self._validate()
>>> new._validate()
>>> new_cropped._validate()
>>> new_opt._validate()
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> stacked = kwimage.stack_images(vizable.transpose(2, 0, 1))
>>> kwplot.imshow(stacked)
```

Example

```
>>> # Test case where requested channel does not exist
>>> import kwcoco
>>> from kwcoco.util.delayed_ops.delayed_nodes import *  # NOQA
>>> dset = kwcoco.CocoDataset.demo('vidshapes8-multispectral', use_cache=1,␣
↪verbose=100)
>>> self = delayed = dset.coco_image(1).delay(mode=1)
>>> channels = 'B1|foobar|bazbiz|B8'
>>> new = self.take_channels(channels)
>>> new_cropped = new.crop((slice(10, 200), slice(12, 350)))
>>> fused = new_cropped.finalize()
>>> assert fused.shape == (190, 338, 4)
>>> assert np.all(np.isnan(fused[..., 1:3]))
>>> assert not np.any(np.isnan(fused[..., 0]))
>>> assert not np.any(np.isnan(fused[..., 3]))
```

property num_overviews

Returns: int

as_xarray()

Returns

DelayedAsXarray

undo_warps(*remove=None*, *retain=None*, *squash_nans=False*, *return_warps=False*)

Attempts to "undo" warping for each concatenated channel and returns a list of delayed operations that are cropped to the right regions.

Typically you will retrain offset, theta, and shear to remove scale. This ensures the data is spatially aligned up to a scale factor.

Parameters

- **remove** (*List[str]*) – if specified, list components of the warping to remove. Can include: "offset", "scale", "shearx", "theta". Typically set this to ["scale"].
- **retain** (*List[str]*) – if specified, list components of the warping to retain. Can include: "offset", "scale", "shearx", "theta". Mutually exclusive with "remove". If neither remove or retain is specified, retain is set to `[]`.
- **squash_nans** (*bool*) – if True, pure nan channels are squashed into a 1x1 array as they do not correspond to a real source.
- **return_warps** (*bool*) – if True, return the transforms we applied. This is useful when you need to warp objects in the original space into the jagged space.

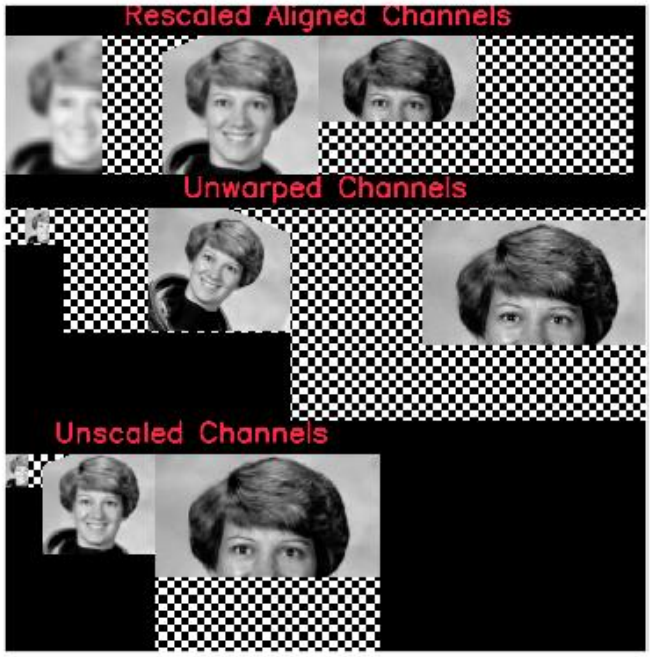
Example

```
>>> from kwcoco.util.delayed_ops.delayed_nodes import *  # NOQA
>>> from kwcoco.util.delayed_ops.delayed_leafs import DelayedLoad
>>> from kwcoco.util.delayed_ops.delayed_leafs import DelayedNans
>>> import ubelt as ub
>>> import kwimage
>>> import kwarray
>>> import numpy as np
>>> # Demo case where we have different channels at different resolutions
>>> base = DelayedLoad.demo(channels='r|g|b').prepare().dequantize({'quant_max
↪': 255})
>>> bandR = base[:, :, 0].scale(100 / 512)[:, :-50].evaluate()
>>> bandG = base[:, :, 1].scale(300 / 512).warp({'theta': np.pi / 8, 'about':␣
↪(150, 150)}).evaluate()
>>> bandB = base[:, :, 2].scale(600 / 512)[:150, :].evaluate()
>>> bandN = DelayedNans((600, 600), channels='N')
>>> # Make a concatenation of images of different underlying native resolutions
>>> delayed_vidspace = DelayedChannelConcat([
>>>     bandR.scale(6, dsize=(600, 600)).optimize(),
>>>     bandG.warp({'theta': -np.pi / 8, 'about': (150, 150)}).scale(2,␣
↪dsize=(600, 600)).optimize(),
>>>     bandB.scale(1, dsize=(600, 600)).optimize(),
>>>     bandN,
>>> ]).warp({'scale': 0.7}).optimize()
>>> vidspace_box = kwimage.Boxes([[100, 10, 270, 160]], 'ltrb')
>>> vidspace_poly = vidspace_box.to_polygons()[0]
>>> vidspace_slice = vidspace_box.to_slices()[0]
>>> self = delayed_vidspace[vidspace_slice].optimize()
>>> print('--- Aligned --- ')
>>> self.write_network_text()
>>> squash_nans = True
>>> undone_all_parts, tfs1 = self.undo_warps(squash_nans=squash_nans, return_
```

(continues on next page)

(continued from previous page)

```
↪warps=True)
>>> undone_scale_parts, tfs2 = self.undo_warps(remove=['scale'], squash_
↪nans=squash_nans, return_warps=True)
>>> stackable_aligned = self.finalize().transpose(2, 0, 1)
>>> stackable_undone_all = []
>>> stackable_undone_scale = []
>>> print('--- Undone All --- ')
>>> for undone in undone_all_parts:
...     undone.write_network_text()
...     stackable_undone_all.append(undone.finalize())
>>> print('--- Undone Scale --- ')
>>> for undone in undone_scale_parts:
...     undone.write_network_text()
...     stackable_undone_scale.append(undone.finalize())
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> canvas0 = kwimage.stack_images(stackable_aligned, axis=1)
>>> canvas1 = kwimage.stack_images(stackable_undone_all, axis=1)
>>> canvas2 = kwimage.stack_images(stackable_undone_scale, axis=1)
>>> canvas0 = kwimage.draw_header_text(canvas0, 'Rescaled Aligned Channels')
>>> canvas1 = kwimage.draw_header_text(canvas1, 'Unwarped Channels')
>>> canvas2 = kwimage.draw_header_text(canvas2, 'Unscaled Channels')
>>> canvas = kwimage.stack_images([canvas0, canvas1, canvas2], axis=0)
>>> canvas = kwimage.fill_nans_with_checkers(canvas)
>>> kwplot.imshow(canvas)
```



class kwcoco.util.delayed_ops.delayed_nodes.**DelayedArray**(*subdata=None*)

Bases: *DelayedUnaryOperation*

A generic NDArray.

property shape

Returns: None | Tuple[int | None, ...]

class kwcoco.util.delayed_ops.delayed_nodes.**DelayedImage**(*subdata=None*, *dsize=None*, *channels=None*)

Bases: *ImageOpsMixin*, *DelayedArray*

For the case where an array represents a 2D image with multiple channels

property shape

Returns: None | Tuple[int | None, int | None, int | None]

property num_channels

Returns: None | int

property dsize

Returns: None | Tuple[int | None, int | None]

property channels

Returns: None | kwcoco.FusedChannelSpec

property num_overviews

Returns: int

take_channels(*channels*)

This method returns a subset of the vision data with only the specified bands / channels.

Parameters

channels (*List[int] | slice | channel_spec.FusedChannelSpec*) – List of integers indexes, a slice, or a channel spec, which is typically a pipe (|) delimited list of channel codes. See kwcoco.ChannelSpec for more detials.

Returns

a new delayed load with a fused take channel operation

Return type

DelayedCrop

Note: The channel subset must exist here or it will raise an error. A better implementation (via pymbolic) might be able to do better

---

Example

```
>>> #
>>> # Test Channel Select Via Code
>>> from kwcoco.util.delayed_ops.delayed_nodes import *  # NOQA
>>> from kwcoco.util.delayed_ops import DelayedLoad
>>> self = DelayedLoad.demo(dsize=(16, 16), channels='r|g|b').prepare()
>>> channels = 'r|b'
>>> new = self.take_channels(channels)._validate()
>>> new2 = new[:, :, [1, 0]]._validate()
>>> new3 = new2[:, :, [1]]._validate()
```

Example

```
>>> from kwcoco.util.delayed_ops.delayed_nodes import *  # NOQA
>>> from kwcoco.util.delayed_ops import DelayedLoad
>>> import kwcoco
>>> self = DelayedLoad.demo('astro').prepare()
>>> channels = [2, 0]
>>> new = self.take_channels(channels)
>>> new3 = new.take_channels([1, 0])
>>> new._validate()
>>> new3._validate()
```

```
>>> final1 = self.finalize()
>>> final2 = new.finalize()
>>> final3 = new3.finalize()
>>> assert np.all(final1[..., 2] == final2[..., 0])
>>> assert np.all(final1[..., 0] == final2[..., 1])
>>> assert final2.shape[2] == 2
```

```
>>> assert np.all(final1[..., 2] == final3[..., 1])
>>> assert np.all(final1[..., 0] == final3[..., 0])
>>> assert final3.shape[2] == 2
```

get_transform_from_leaf()

Returns the transformation that would align data with the leaf

evaluate()

Evaluate this node and return the data as an identity.

Returns

DelayedIdentity

undo_warp(*remove=None*, *retain=None*, *squash_nans=False*, *return_warp=False*)

Attempts to "undo" warping for each concatenated channel and returns a list of delayed operations that are cropped to the right regions.

Typically you will retrain offset, theta, and shear to remove scale. This ensures the data is spatially aligned up to a scale factor.

Parameters

- **remove** (*List[str]*) – if specified, list components of the warping to remove. Can include: "offset", "scale", "shearx", "theta". Typically set this to ["scale"].
- **retain** (*List[str]*) – if specified, list components of the warping to retain. Can include: "offset", "scale", "shearx", "theta". Mutually exclusive with "remove". If neither remove or retain is specified, retain is set to [].
- **squash_nans** (*bool*) – if True, pure nan channels are squashed into a 1x1 array as they do not correspond to a real source.
- **return_warp** (*bool*) – if True, return the transform we applied. This is useful when you need to warp objects in the original space into the jagged space.

SeeAlso:

DelayedChannelConcat.undo_warps

Example

```
>>> # Test similar to undo_warps, but on each channel separately
>>> from kwcoco.util.delayed_ops.delayed_nodes import *  # NOQA
>>> from kwcoco.util.delayed_ops.delayed_leafs import DelayedLoad
>>> from kwcoco.util.delayed_ops.delayed_leafs import DelayedNans
>>> import ubelt as ub
>>> import kwimage
>>> import kwarray
>>> import numpy as np
>>> # Demo case where we have different channels at different resolutions
>>> base = DelayedLoad.demo(channels='r|g|b').prepare().dequantize({'quant_max
↪': 255})
>>> bandR = base[:, :, 0].scale(100 / 512)[:, :-50].evaluate()
>>> bandG = base[:, :, 1].scale(300 / 512).warp({'theta': np.pi / 8, 'about':␣
↪(150, 150)}).evaluate()
>>> bandB = base[:, :, 2].scale(600 / 512)[:150, :].evaluate()
```

(continues on next page)

(continued from previous page)

```
>>> bandN = DelayedNans((600, 600), channels='N')
>>> B0 = bandR.scale(6, dsize=(600, 600)).optimize()
>>> B1 = bandG.warp({'theta': -np.pi / 8, 'about': (150, 150)}).scale(2,␣
↪dsize=(600, 600)).optimize()
>>> B2 = bandB.scale(1, dsize=(600, 600)).optimize()
>>> vidspace_box = kwimage.Boxes([[-10, -10, 270, 160]], 'ltrb').scale(1 / .7).
↪quantize()
>>> vidspace_poly = vidspace_box.to_polygons()[0]
>>> vidspace_slice = vidspace_box.to_slices()[0]
>>> # Test with the padded crop
>>> self0 = B0.crop(vidspace_slice, wrap=0, clip=0, pad=10).optimize()
>>> self1 = B1.crop(vidspace_slice, wrap=0, clip=0, pad=10).optimize()
>>> self2 = B2.crop(vidspace_slice, wrap=0, clip=0, pad=10).optimize()
>>> parts = [self0, self1, self2]
>>> # Run the undo on each channel
>>> undone_scale_parts = [d.undo_warp(remove=['scale']) for d in parts]
>>> print('--- Aligned --- ')
>>> stackable_aligned = []
>>> for d in parts:
>>>     d.write_network_text()
>>>     stackable_aligned.append(d.finalize())
>>> print('--- Undone Scale --- ')
>>> stackable_undone_scale = []
>>> for undone in undone_scale_parts:
...     undone.write_network_text()
...     stackable_undone_scale.append(undone.finalize())
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> canvas0 = kwimage.stack_images(stackable_aligned, axis=1, pad=5, bg_value=
↪'kw_darkgray')
>>> canvas2 = kwimage.stack_images(stackable_undone_scale, axis=1, pad=5, bg_
↪value='kw_darkgray')
>>> canvas0 = kwimage.draw_header_text(canvas0, 'Rescaled Channels')
>>> canvas2 = kwimage.draw_header_text(canvas2, 'Native Scale Channels')
>>> canvas = kwimage.stack_images([canvas0, canvas2], axis=0, bg_value='kw_
↪darkgray')
>>> canvas = kwimage.fill_nans_with_checkers(canvas)
>>> kwplot.imshow(canvas)
```

class kwcoco.util.delayed_ops.delayed_nodes.**DelayedAsXarray**(*subdata=None*, *dsize=None*, *channels=None*)

Bases: *DelayedImage*

Casts the data to an xarray object in the finalize step

Example;

```
>>> from kwcoco.util.delayed_ops.delayed_nodes import *  # NOQA
>>> from kwcoco.util.delayed_ops import DelayedLoad
>>> # without channels
>>> base = DelayedLoad.demo(dsize=(16, 16)).prepare()
>>> self = base.as_xarray()
>>> final = self._validate().finalize()
>>> assert len(final.coords) == 0
>>> assert final.dims == ('y', 'x', 'c')
>>> # with channels
>>> base = DelayedLoad.demo(dsize=(16, 16), channels='r|g|b').prepare()
>>> self = base.as_xarray()
>>> final = self._validate().finalize()
>>> assert final.coords.indexes['c'].tolist() == ['r', 'g', 'b']
>>> assert final.dims == ('y', 'x', 'c')
```

optimize()

Returns
DelayedImage

class kwcoco.util.delayed_ops.delayed_nodes.**DelayedWarp**(*subdata*, *transform*, *dsize='auto'*, *antialias=True*, *interpolation='linear'*, *border_value='auto'*)

Bases: *DelayedImage*

Applies an affine transform to an image.

Example

```
>>> from kwcoco.util.delayed_ops.delayed_nodes import *  # NOQA
>>> from kwcoco.util.delayed_ops import DelayedLoad
>>> self = DelayedLoad.demo(dsize=(16, 16)).prepare()
>>> warp1 = self.warp({'scale': 3})
>>> warp2 = warp1.warp({'theta': 0.1})
>>> warp3 = warp2._opt_fuse_warps()
>>> warp3._validate()
>>> print(ub.repr2(warp2.nesting(), nl=-1, sort=0))
>>> print(ub.repr2(warp3.nesting(), nl=-1, sort=0))
```

property transform

Returns: kwimage.Affine

optimize()

Returns
DelayedImage

Example

```
>>> # Demo optimization that removes a noop warp
>>> from kwcoco.util.delayed_ops import DelayedLoad
>>> import kwimage
>>> base = DelayedLoad.demo(channels='r|g|b').prepare()
>>> self = base.warp(kwimage.Affine.eye())
>>> new = self.optimize()
>>> assert len(self.as_graph().nodes) == 2
>>> assert len(new.as_graph().nodes) == 1
```

Example

```
>>> # Test optimize nans
>>> from kwcoco.util.delayed_ops import DelayedNans
>>> import kwimage
>>> base = DelayedNans(dsize=(100, 100), channels='a|b|c')
>>> self = base.warp(kwimage.Affine.scale(0.1))
>>> # Should simply return a new nan generator
>>> new = self.optimize()
>>> assert len(new.as_graph().nodes) == 1
```

class kwcoco.util.delayed_ops.delayed_nodes.**DelayedDequantize**(*subdata*, *quantization*)

Bases: *DelayedImage*

Rescales image intensities from int to floats.

The output is usually between 0 and 1. This also handles transforming nodata into nan values.

optimize()

Returns
DelayedImage

Example

```
>>> # Test a case that caused an error in development
>>> from kwcoco.util.delayed_ops.delayed_nodes import *  # NOQA
>>> from kwcoco.util.delayed_ops import DelayedLoad
>>> fpath = kwimage.grab_test_image_fpath()
>>> base = DelayedLoad(fpath, channels='r|g|b').prepare()
>>> quantization = {'quant_max': 255, 'nodata': 0}
>>> self = base.get_overview(1).dequantize(quantization)
>>> self.write_network_text()
>>> opt = self.optimize()
```

class kwcoco.util.delayed_ops.delayed_nodes.**DelayedCrop**(*subdata*, *space_slice=None*, *chan_idxs=None*)

Bases: *DelayedImage*

Crops an image along integer pixel coordinates.

Example

```
>>> from kwcoco.util.delayed_ops.delayed_nodes import *  # NOQA
>>> from kwcoco.util.delayed_ops import DelayedLoad
>>> base = DelayedLoad.demo(dsize=(16, 16)).prepare()
>>> # Test Fuse Crops Space Only
>>> crop1 = base[4:12, 0:16]
>>> self = crop1[2:6, 0:8]
>>> opt = self._opt_fuse_crops()
>>> self.write_network_text()
>>> opt.write_network_text()
>>> #
>>> # Test Channel Select Via Index
>>> self = base[:, :, [0]]
>>> self.write_network_text()
>>> final = self._finalize()
>>> assert final.shape == (16, 16, 1)
>>> assert base[:, :, [0, 1]].finalize().shape == (16, 16, 2)
>>> assert base[:, :, [2, 0, 1]].finalize().shape == (16, 16, 3)
```

optimize()

Returns
DelayedImage

Example

```
>>> # Test optimize nans
>>> from kwcoco.util.delayed_ops import DelayedNans
>>> import kwimage
>>> base = DelayedNans(dsize=(100, 100), channels='a|b|c')
>>> self = base[0:10, 0:5]
>>> # Should simply return a new nan generator
>>> new = self.optimize()
>>> self.write_network_text()
>>> new.write_network_text()
>>> assert len(new.as_graph().nodes) == 1
```

class kwcoco.util.delayed_ops.delayed_nodes.**DelayedOverview**(*subdata*, *overview*)

Bases: *DelayedImage*

Downsamples an image by a factor of two.

If the underlying image being loaded has precomputed overviews it simply loads these instead of downsampling the original image, which is more efficient.

Example

```
>>> # xdoctest: +REQUIRES(module:osgeo)
>>> # Make a complex chain of operations and optimize it
>>> from kwcoco.util.delayed_ops import *  # NOQA
>>> import kwimage
>>> fpath = kwimage.grab_test_image_fpath(overviews=3)
>>> dimg = DelayedLoad(fpath, channels='r|g|b').prepare()
>>> dimg = dimg.get_overview(1)
>>> dimg = dimg.get_overview(1)
>>> dimg = dimg.get_overview(1)
>>> dopt = dimg.optimize()
>>> if 1:
>>>     import networkx as nx
>>>     dimg.write_network_text()
>>>     dopt.write_network_text()
>>> print(ub.repr2(dopt.nesting(), nl=-1, sort=0))
>>> final0 = dimg._finalize()[:]
>>> final1 = dopt._finalize()[:]
>>> assert final0.shape == final1.shape
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(final0, pnum=(1, 2, 1), fnum=1, title='raw')
>>> kwplot.imshow(final1, pnum=(1, 2, 2), fnum=1, title='optimized')
```

property num_overviews

Returns: int

optimize()

Returns

DelayedImage

kwcoco.util.delayed_ops.delayed_nodes.**DelayedOverview2**

alias of *DelayedOverview*

kwcoco.util.delayed_ops.delayed_nodes.**DelayedCrop2**

alias of *DelayedCrop*

kwcoco.util.delayed_ops.delayed_nodes.**DelayedDequantize2**

alias of *DelayedDequantize*

kwcoco.util.delayed_ops.delayed_nodes.**DelayedWarp2**

alias of *DelayedWarp*

kwcoco.util.delayed_ops.delayed_nodes.**DelayedAsXarray2**

alias of *DelayedAsXarray*

kwcoco.util.delayed_ops.delayed_nodes.**DelayedImage2**

alias of *DelayedImage*

kwcoco.util.delayed_ops.delayed_nodes.**DelayedArray2**

alias of *DelayedArray*

kwcoco.util.delayed_ops.delayed_nodes.**DelayedChannelConcat2**

alias of *DelayedChannelConcat*

kwcoco.util.delayed_ops.delayed_nodes.**DelayedFrameStack2**

alias of *DelayedFrameStack*

kwcoco.util.delayed_ops.delayed_nodes.**DelayedConcat2**

alias of *DelayedConcat*

kwcoco.util.delayed_ops.delayed_nodes.**DelayedStack2**

alias of *DelayedStack*

2.1.1.6.1.6 kwcoco.util.delayed_ops.helpers module

kwcoco.util.delayed_ops.helpers.**dequantize**(*quant_data*, *quantization*)

Helper for dequantization

Parameters

- **quant_data** (*ndarray*) – data to dequantize
- **quantization** (*Dict[str, Any]*) – quantization information dictionary to undo. Expected keys are: orig_type (str) orig_min (float) orig_max (float) quant_min (float) quant_max (float) nodata (None | int)

Returns

dequantized data

Return type

ndarray

Example

```
>>> quant_data = (np.random.rand(4, 4) * 256).astype(np.uint8)
>>> quantization = {
>>>     'orig_dtype': 'float32',
>>>     'orig_min': 0,
>>>     'orig_max': 1,
>>>     'quant_min': 0,
>>>     'quant_max': 255,
>>>     'nodata': None,
>>> }
>>> dequantize(quant_data, quantization)
```

Example

```
>>> quant_data = np.ones((4, 4), dtype=np.uint8)
>>> quantization = {
>>>     'orig_dtype': 'float32',
>>>     'orig_min': 0,
>>>     'orig_max': 1,
>>>     'quant_min': 1,
>>>     'quant_max': 1,
>>>     'nodata': None,
>>> }
>>> dequantize(quant_data, quantization)
```

kwcoco.util.delayed_ops.helpers.**quantize_float01**(*imdata*, *old_min=0*, *old_max=1*, *quantize_dtype=<class 'numpy.int16'>*)

Note: Setting old_min / old_max indicates the possible extend of the input data (and it will be clipped to it). It does not mean that the input data has to have those min and max values, but it should be between them.

Example

```
>>> from kwcoco.util.delayed_ops.helpers import *  # NOQA
>>> # Test error when input is not nicely between 0 and 1
>>> imdata = (np.random.randn(32, 32, 3) - 1.) * 2.5
>>> quant1, quantization1 = quantize_float01(imdata, old_min=0, old_max=1)
>>> recon1 = dequantize(quant1, quantization1)
>>> error1 = np.abs((recon1 - imdata)).sum()
>>> print('error1 = {!r}'.format(error1))
>>> #
>>> for i in range(1, 20):
>>>     print('i = {!r}'.format(i))
>>>     quant2, quantization2 = quantize_float01(imdata, old_min=-i, old_max=i)
>>>     recon2 = dequantize(quant2, quantization2)
>>>     error2 = np.abs((recon2 - imdata)).sum()
>>>     print('error2 = {!r}'.format(error2))
```

Example

```
>>> # Test dequantize with uint8
>>> from kwcoco.util.util_delayed_poc import dequantize
>>> imdata = np.random.randn(32, 32, 3)
>>> quant1, quantization1 = quantize_float01(imdata, old_min=0, old_max=1, quantize_
↪dtype=np.uint8)
>>> recon1 = dequantize(quant1, quantization1)
>>> error1 = np.abs((recon1 - imdata)).sum()
>>> print('error1 = {!r}'.format(error1))
```

Example

```
>>> # Test quantization with different signed / unsigned combos
>>> print(quantize_float01(None, 0, 1, np.int16))
>>> print(quantize_float01(None, 0, 1, np.int8))
>>> print(quantize_float01(None, 0, 1, np.uint8))
>>> print(quantize_float01(None, 0, 1, np.uint16))
```

class kwcoco.util.delayed_ops.helpers.**AsciiDirectedGlyphs**

Bases: _AsciiBaseGlyphs

last = 'L-> '

mid = '|-> '

backedge = '<-'

class kwcoco.util.delayed_ops.helpers.**AsciiUndirectedGlyphs**

Bases: _AsciiBaseGlyphs

last = 'L-- '

mid = '|-- '

backedge = '-'

class kwcoco.util.delayed_ops.helpers.**UtfDirectedGlyphs**

Bases: _UtfBaseGlyphs

last = '└─ '

mid = '├─ '

backedge = ''

class kwcoco.util.delayed_ops.helpers.**UtfUndirectedGlyphs**

Bases: _UtfBaseGlyphs

last = '└── '

mid = '├── '

backedge = '─'

kwcoco.util.delayed_ops.helpers.**generate_network_text**(*graph*, *with_labels=True*, *sources=None*, *max_depth=None*, *ascii_only=False*)

Generate lines in the "network text" format

This works via a depth-first traversal of the graph and writing a line for each unique node encountered. Non-tree edges are written to the right of each node, and connection to a non-tree edge is indicated with an ellipsis. This representation works best when the input graph is a forest, but any graph can be represented.

This notation is original to networkx, although it is simple enough that it may be known in existing literature. See #5602 for details. The procedure is summarized as follows:

1. Given a set of source nodes (which can be specified, or automatically discovered via finding the (strongly) connected components and choosing one node with minimum degree from each), we traverse the graph in depth first order.

2. Each reachable node will be printed exactly once on it's own line.

3. Edges are indicated in one of three ways:

    a. a parent "L-style" connection on the upper left. This corresponds to a traversal in the directed DFS tree.

    b. a backref "<-style" connection shown directly on the right. For directed graphs, these are drawn for any incoming edges to a node that is not a parent edge. For undirected graphs, these are drawn for only the non-parent edges that have already been represented (The edges that have not been represented will be handled in the recursive case).

    c. a child "L-style" connection on the lower right. Drawing of the children are handled recursively.

4. The children of each node (wrt the directed DFS tree) are drawn underneath and to the right of it. In the case that a child node has already been drawn the connection is replaced with an ellipsis ("...") to indicate that there is one or more connections represented elsewhere.

5. If a maximum depth is specified, an edge to nodes past this maximum depth will be represented by an ellipsis.

Parameters

- **graph** (*nx.DiGraph | nx.Graph*) – Graph to represent
- **with_labels** (*bool | str*) – If True will use the "label" attribute of a node to display if it exists otherwise it will use the node value itself. If given as a string, then that attribte name will be used instead of "label". Defaults to True.
- **sources** (*List*) – Specifies which nodes to start traversal from. Note: nodes that are not reachable from one of these sources may not be shown. If unspecified, the minimal set of nodes needed to reach all others will be used.
- **max_depth** (*int | None*) – The maximum depth to traverse before stopping. Defaults to None.
- **ascii_only** (*Boolean*) – If True only ASCII characters are used to construct the visualization

Yields

str (*a line of generated text*)

kwcoco.util.delayed_ops.helpers.**write_network_text**(*graph*, *path=None*, *with_labels=True*, *sources=None*, *max_depth=None*, *ascii_only=False*, *end='\n'*)

Creates a nice text representation of a graph

This works via a depth-first traversal of the graph and writing a line for each unique node encountered. Non-tree edges are written to the right of each node, and connection to a non-tree edge is indicated with an ellipsis. This representation works best when the input graph is a forest, but any graph can be represented.

Parameters

- **graph** (*nx.DiGraph | nx.Graph*) – Graph to represent
- **path** (*string or file or callable or None*) – Filename or file handle for data output. if a function, then it will be called for each generated line. if None, this will default to "sys.stdout.write"
- **with_labels** (*bool | str*) – If True will use the "label" attribute of a node to display if it exists otherwise it will use the node value itself. If given as a string, then that attribte name will be used instead of "label". Defaults to True.
- **sources** (*List*) – Specifies which nodes to start traversal from. Note: nodes that are not reachable from one of these sources may not be shown. If unspecified, the minimal set of nodes needed to reach all others will be used.
- **max_depth** (*int | None*) – The maximum depth to traverse before stopping. Defaults to None.
- **ascii_only** (*Boolean*) – If True only ASCII characters are used to construct the visualization

- **end** (*string*) – The line ending characater

Example

```
>>> import networkx as nx
>>> graph = nx.balanced_tree(r=2, h=2, create_using=nx.DiGraph)
>>> write_network_text(graph)
╙── 0
    ├─╼ 1
    │   ├─╼ 3
    │   └─╼ 4
    └─╼ 2
        ├─╼ 5
        └─╼ 6
```

```
>>> # A near tree with one non-tree edge
>>> graph.add_edge(5, 1)
>>> write_network_text(graph)
╙── 0
    ├─╼ 1  5
    │   ├─╼ 3
    │   └─╼ 4
    └─╼ 2
        ├─╼ 5
        │   └─  ...
        └─╼ 6
```

```
>>> graph = nx.cycle_graph(5)
>>> write_network_text(graph)
╙── 0
    ├── 1
    │   └── 2
    │       └── 3
    │           └── 4 ─ 0
    └──  ...
```

```
>>> graph = nx.generators.barbell_graph(4, 2)
>>> write_network_text(graph)
╙── 4
    ├── 5
    │   └── 6
    │       ├── 7
    │       │   ├── 8 ─ 6
    │       │   │   └── 9 ─ 6, 7
    │       │   └──  ...
    │       └──  ...
    └── 3
        ├── 0
        │   ├── 1 ─ 3
        │   │   └── 2 ─ 0, 3
        │   └──  ...
        └──  ...
```

```
>>> graph = nx.complete_graph(5, create_using=nx.Graph)
>>> write_network_text(graph)
╙── 0
    ├── 1
    │   ├── 2 ─ 0
    │   │   ├── 3 ─ 0, 1
    │   │   │   └── 4 ─ 0, 1, 2
    │   │   └──  ...
    │   └──  ...
    └──  ...
```

```
>>> graph = nx.complete_graph(3, create_using=nx.DiGraph)
>>> write_network_text(graph)
╙── 0  1, 2
    ├─ 1  2
    │   ├─ 2  0
    │   │   └─  ...
    │   └─  ...
    └─  ...
```

kwcoco.util.delayed_ops.helpers.**graph_str**(*graph*, *with_labels=True*, *sources=None*, *write=None*, *ascii_only=False*)

Creates a nice utf8 representation of a forest

This function has been superseded by `nx.readwrite.text.generate_network_text()`, which should be used instead.

Parameters

- **graph** (*nx.DiGraph | nx.Graph*) – Graph to represent (must be a tree, forest, or the empty graph)
- **with_labels** (*bool*) – If True will use the "label" attribute of a node to display if it exists otherwise it will use the node value itself. Defaults to True.
- **sources** (*List*) – Mainly relevant for undirected forests, specifies which nodes to list first. If unspecified the root nodes of each tree will be used for directed forests; for undirected forests this defaults to the nodes with the smallest degree.
- **write** (*callable*) – Function to use to write to, if None new lines are appended to a list and returned. If set to the *print* function, lines will be written to stdout as they are generated. If specified, this function will return None. Defaults to None.
- **ascii_only** (*Boolean*) – If True only ASCII characters are used to construct the visualization

Returns

utf8 representation of the tree / forest

Return type

str | None

Example

```
>>> import networkx as nx
>>> graph = nx.balanced_tree(r=2, h=3, create_using=nx.DiGraph)
>>> print(graph_str(graph))
╙── 0
    ├─╼ 1
    │   ├─╼ 3
    │   │   ├─╼ 7
    │   │   └─╼ 8
    │   └─╼ 4
    │       ├─╼ 9
    │       └─╼ 10
    └─╼ 2
        ├─╼ 5
        │   ├─╼ 11
        │   └─╼ 12
        └─╼ 6
            ├─╼ 13
            └─╼ 14
```

```
>>> graph = nx.balanced_tree(r=1, h=2, create_using=nx.Graph)
>>> print(graph_str(graph))
╙── 0
    └─╼ 1
        └─╼ 2
```

```
>>> print(graph_str(graph, ascii_only=True))
+-- 0
    L-- 1
        L-- 2
```

2.1.1.6.1.7 Module contents

A rewrite of the delayed operations

Note: The classes in this submodule will have their names changed when the old POC delayed operations are deprecated.

Todo: The optimize logic could likley be better expressed as some sort of AST transformer.

Example

```
>>> # xdoctest: +REQUIRES(module:osgeo)
>>> from kwcoco.util.delayed_ops import *  # NOQA
>>> import kwimage
>>> fpath = kwimage.grab_test_image_fpath(overviews=3)
>>> dimg = DelayedLoad(fpath, channels='r|g|b').prepare()
>>> quantization = {'quant_max': 255, 'nodata': 0}
>>> #
>>> # Make a complex chain of operations
>>> dimg = dimg.dequantize(quantization)
>>> dimg = dimg.warp({'scale': 1.1})
>>> dimg = dimg.warp({'scale': 1.1})
>>> dimg = dimg[0:400, 1:400]
>>> dimg = dimg.warp({'scale': 0.5})
>>> dimg = dimg[0:800, 1:800]
>>> dimg = dimg.warp({'scale': 0.5})
>>> dimg = dimg[0:800, 1:800]
>>> dimg = dimg.warp({'scale': 0.5})
>>> dimg = dimg.warp({'scale': 1.1})
>>> dimg = dimg.warp({'scale': 1.1})
>>> dimg = dimg.warp({'scale': 2.1})
>>> dimg = dimg[0:200, 1:200]
>>> dimg = dimg[1:200, 2:200]
>>> dimg.write_network_text()
╙── Crop dsize=(128,130),space_slice=(slice(1,131,None),slice(2,130,None))
    └─ Crop dsize=(130,131),space_slice=(slice(0,131,None),slice(1,131,None))
        └─ Warp dsize=(131,131),transform={scale=2.1000}
            └─ Warp dsize=(62,62),transform={scale=1.1000}
                └─ Warp dsize=(56,56),transform={scale=1.1000}
                    └─ Warp dsize=(50,50),transform={scale=0.5000}
                        └─ Crop dsize=(99,100),space_slice=(slice(0,100,None),slice(1,
↪100,None))
                            └─ Warp dsize=(100,100),transform={scale=0.5000}
                                └─ Crop dsize=(199,200),space_slice=(slice(0,200,None),
↪slice(1,200,None))
                                    └─ Warp dsize=(200,200),transform={scale=0.5000}
                                        └─ Crop dsize=(399,400),space_slice=(slice(0,400,
↪None),slice(1,400,None))
                                            └─ Warp dsize=(621,621),transform={scale=1.
↪1000}
                                                └─ Warp dsize=(564,564),transform=
↪{scale=1.1000}
                                                    └─ Dequantize dsize=(512,512),
↪quantization={quant_max=255,nodata=0}
                                                        └─ Load channels=r|g|b,
↪dsize=(512,512),num_overviews=3,fname=astro_overviews=3.tif
```

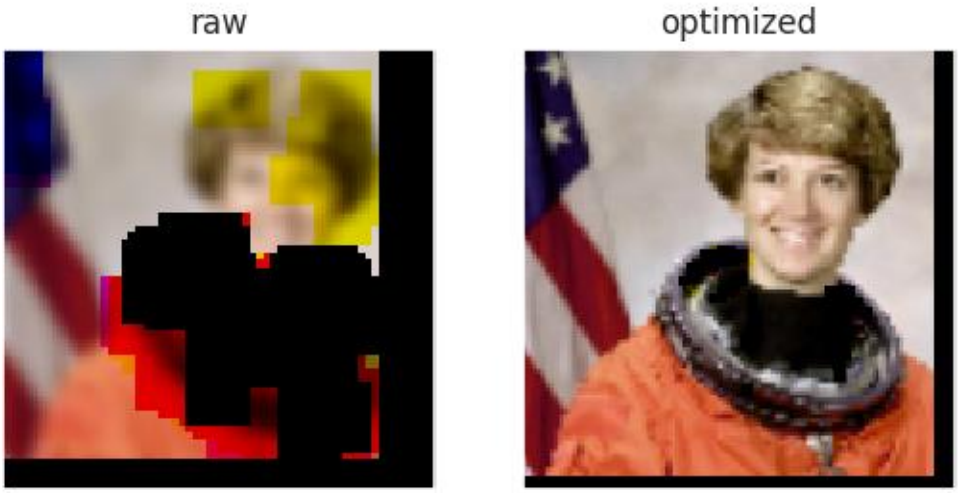
```
>>> # Optimize the chain
>>> dopt = dimg.optimize()
>>> dopt.write_network_text()
╙── Warp dsize=(128,130),transform={offset=(-0.6...,-1.0...),scale=1.5373}
    └─ Dequantize dsize=(80,83),quantization={quant_max=255,nodata=0}
```

(continues on next page)

(continued from previous page)

```
        └─ Crop dsize=(80,83),space_slice=(slice(0,83,None),slice(3,83,None))
            └─ Overview dsize=(128,128),overview=2
                └─ Load channels=r|g|b,dsize=(512,512),num_overviews=3,fname=astro_
↪overviews=3.tif
```

```
>>> final0 = dimg.finalize(optimize=False)
>>> final1 = dopt.finalize()
>>> assert final0.shape == final1.shape
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(final0, pnum=(1, 2, 1), fnum=1, title='raw')
>>> kwplot.imshow(final1, pnum=(1, 2, 2), fnum=1, title='optimized')
```



Example

```
>>> # xdoctest: +REQUIRES(module:osgeo)
>>> from kwcoco.util.delayed_ops import *  # NOQA
>>> import ubelt as ub
>>> import kwimage
>>> # Sometimes we want to manipulate data in a space, but then remove all
>>> # warps in order to get a sample without any data artifacts.  This is
>>> # handled by adding a new transform that inverts everything and optimizing
>>> # it, which results in all warps canceling each other out.
>>> fpath = kwimage.grab_test_image_fpath()
>>> base = DelayedLoad(fpath, channels='r|g|b').prepare()
>>> warp = kwimage.Affine.random(rng=321, offset=0)
>>> warp = kwimage.Affine.scale(0.5)
>>> orig = base.get_overview(1).warp(warp)[16:96, 24:128]
>>> delayed = orig.optimize()
>>> print('Orig')
>>> orig.write_network_text()
>>> print('Delayed')
>>> delayed.write_network_text()
>>> # Get the transform that would bring us back to the leaf
>>> tf_root_from_leaf = delayed.get_transform_from_leaf()
>>> print('tf_root_from_leaf =\n{}'.format(ub.repr2(tf_root_from_leaf, nl=1)))
>>> undo_all = tf_root_from_leaf.inv()
>>> print('undo_all =\n{}'.format(ub.repr2(undo_all, nl=1)))
>>> undo_scale = kwimage.Affine.coerce(ub.dict_diff(undo_all.concise(), ['offset']))
>>> print('undo_scale =\n{}'.format(ub.repr2(undo_scale, nl=1)))
>>> print('Undone All')
>>> undone_all = delayed.warp(undo_all).optimize()
>>> undone_all.write_network_text()
>>> # Discard translation components
>>> print('Undone Scale')
>>> undone_scale = delayed.warp(undo_scale).optimize()
>>> undone_scale.write_network_text()
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> to_stack = []
>>> to_stack.append(base.finalize(optimize=False))
>>> to_stack.append(orig.finalize(optimize=False))
>>> to_stack.append(delayed.finalize(optimize=False))
>>> to_stack.append(undone_all.finalize(optimize=False))
>>> to_stack.append(undone_scale.finalize(optimize=False))
>>> kwplot.autompl()
>>> stack = kwimage.stack_images(to_stack, axis=1, bg_value=(5, 100, 10), pad=10)
>>> kwplot.imshow(stack)
```

CommandLine

```
xdoctest -m /home/joncrall/code/kwcoco/kwcoco/util/delayed_ops/__init__.py __doc__:2
```

Example

```
>>> # xdoctest: +REQUIRES(module:osgeo)
>>> from kwcoco.util.delayed_ops import *  # NOQA
>>> import ubelt as ub
>>> import kwimage
>>> import kwarray
>>> import numpy as np
>>> # Demo case where we have different channels at different resolutions
>>> base = DelayedLoad.demo(channels='r|g|b').prepare().dequantize({'quant_max': 255})
>>> bandR = base[:, :, 0].scale(100 / 512)[:, :-50].evaluate()
>>> bandG = base[:, :, 1].scale(300 / 512).warp({'theta': np.pi / 8, 'about': (150, 150)}
↪).evaluate()
>>> bandB = base[:, :, 2].scale(600 / 512)[:150, :].evaluate()
>>> # Align the bands in "video" space
>>> delayed_vidspace = DelayedChannelConcat([
>>>     bandR.scale(6, dsize=(600, 600)).optimize(),
```

(continues on next page)

(continued from previous page)

```
>>>     bandG.warp({'theta': -np.pi / 8, 'about': (150, 150)}).scale(2, dsize=(600,␣
↪600)).optimize(),
>>>     bandB.scale(1, dsize=(600, 600)).optimize(),
>>> ]).warp(
>>>   #{'scale': 0.35, 'theta': 0.3, 'about': (30, 50), 'offset': (-10, -80)}
>>>   {'scale': 0.7}
>>> )
>>> #delayed_vidspace._set_nested_params(border_value=0)
>>> vidspace_box = kwimage.Boxes([[100, 10, 270, 160]], 'ltrb')
>>> vidspace_poly = vidspace_box.to_polygons()[0]
>>> vidspace_slice = vidspace_box.to_slices()[0]
>>> crop_vidspace = delayed_vidspace[vidspace_slice]
>>> crop_vidspace._set_nested_params(interpolation='lanczos')
>>> # Note: this only works because the graph is lazilly optimized
>>> crop_vidspace_box = vidspace_box.warp(crop_vidspace._transform_from_subdata())
>>> crop_vidspace_poly = vidspace_poly.warp(crop_vidspace._transform_from_subdata())
>>> opt_crop_vidspace = crop_vidspace.optimize()
>>> print('Original: Video Space')
>>> delayed_vidspace.write_network_text()
>>> print('Original Crop: Video Space')
>>> crop_vidspace.write_network_text()
>>> print('Optimized Crop: Video Space')
>>> opt_crop_vidspace.write_network_text()
>>> tostack_grid = []
>>> # Drop boxes in asset space
>>> tostack_grid.append([]); row = tostack_grid[-1]
>>> row.append(kwimage.draw_text_on_image(None, text='Underlying asset bands (imagine␣
↪these are on disk)'))
>>> tostack_grid.append([]); row = tostack_grid[-1]
>>> delayed_vidspace_opt = delayed_vidspace.optimize()
>>> tf_vidspace_to_rband = delayed_vidspace_opt.parts[0].get_transform_from_leaf().inv()
>>> tf_vidspace_to_gband = delayed_vidspace_opt.parts[1].get_transform_from_leaf().inv()
>>> tf_vidspace_to_bband = delayed_vidspace_opt.parts[2].get_transform_from_leaf().inv()
>>> rband_box = vidspace_box.warp(tf_vidspace_to_rband)
>>> gband_box = vidspace_box.warp(tf_vidspace_to_gband)
>>> bband_box = vidspace_box.warp(tf_vidspace_to_bband)
>>> rband_poly = vidspace_poly.warp(tf_vidspace_to_rband)
>>> gband_poly = vidspace_poly.warp(tf_vidspace_to_gband)
>>> bband_poly = vidspace_poly.warp(tf_vidspace_to_bband)
>>> row.append(kwimage.draw_header_text(rband_poly.draw_on(rband_box.draw_on(bandR.
↪finalize()), edgecolor='b', fill=0), 'R'))
>>> row.append(kwimage.draw_header_text(gband_poly.draw_on(gband_box.draw_on(bandG.
↪finalize()), edgecolor='b', fill=0), 'asset G-band'))
>>> row.append(kwimage.draw_header_text(bband_poly.draw_on(bband_box.draw_on(bandB.
↪finalize()), edgecolor='b', fill=0), 'asset B-band'))
>>> # Draw the box in image space
>>> tostack_grid.append([]); row = tostack_grid[-1]
>>> row.append(kwimage.draw_text_on_image(None, text='A Box in Virtual Video Space (This␣
↪space is conceptually easy to work in)'))
>>> tostack_grid.append([]); row = tostack_grid[-1]
>>> def _tocanvas(img):
...     if img.dtype.kind == 'u':
```

(continues on next page)

(continued from previous page)

```
...         return img
...     return kwimage.ensure_uint255(kwimage.fill_nans_with_checkers(img))
>>> row.append(kwimage.draw_header_text(vidspace_box.draw_on(_tocanvas(delayed_vidspace.
↪finalize())), 'vidspace'))
>>> # Draw finalized aligned crops
>>> tostack_grid.append([]); row = tostack_grid[-1]
>>> row.append(kwimage.draw_text_on_image(None, text='Finalized delayed warp/crop. Left-
↪to-Right: Original, Optimized, Difference'))
>>> tostack_grid.append([]); row = tostack_grid[-1]
>>> crop_opt_final = opt_crop_vidspace.finalize()
>>> crop_raw_final = crop_vidspace.finalize(optimize=False)
>>> row.append(crop_raw_final)
>>> row.append(crop_opt_final)
>>> row.append(kwimage.ensure_uint255(kwarray.normalize(np.linalg.norm(kwimage.ensure_
↪float01(crop_opt_final) - kwimage.ensure_float01(crop_raw_final), axis=2))))
>>> # Get the transform that would bring us back to the leaf
>>> tostack_grid.append([]); row = tostack_grid[-1]
>>> row.append(kwimage.draw_text_on_image(None, text='The "Unwarped" / "Unscaled"␣
↪cropped regions'))
>>> tostack_grid.append([]); row = tostack_grid[-1]
>>> for chosen_band in opt_crop_vidspace.parts:
>>>     spec = chosen_band.channels.spec
>>>     lut = {c[0]: c for c in ['red', 'green', 'blue']}
>>>     color = lut[spec]
>>>     print(ub.color_text('============', color))
>>>     print(ub.color_text(spec, color))
>>>     print(ub.color_text('============', color))
>>>     chosen_band.write_network_text()
>>>     tf_root_from_leaf = chosen_band.get_transform_from_leaf()
>>>     tf_leaf_from_root = tf_root_from_leaf.inv()
>>>     undo_all = tf_leaf_from_root
>>>     undo_scale = kwimage.Affine.coerce(ub.dict_diff(undo_all.concise(), ['offset',
↪'theta']))
>>>     print('tf_root_from_leaf = {}'.format(ub.repr2(tf_root_from_leaf.concise(),␣
↪nl=1)))
>>>     print('undo_all = {}'.format(ub.repr2(undo_all.concise(), nl=1)))
>>>     print('undo_scale = {}'.format(ub.repr2(undo_scale.concise(), nl=1)))
>>>     print('Undone All')
>>>     undone_all = chosen_band.warp(undo_all, interpolation='lanczos').optimize()
>>>     undone_all.write_network_text()
>>>     # Discard translation components
>>>     print('Undone Scale')
>>>     undone_scale = chosen_band.warp(undo_scale).optimize()
>>>     undone_scale.write_network_text()
>>>     undone_all_canvas = undone_all.finalize()
>>>     undone_scale_canvas = undone_scale.finalize()
>>>     undone_all_canvas = crop_vidspace_box.warp(undo_all).draw_on(undone_all_canvas)
>>>     undone_scale_canvas = crop_vidspace_box.warp(undo_scale).draw_on(undone_scale_
↪canvas)
>>>     undone_all_canvas = crop_vidspace_poly.warp(undo_all).draw_on(undone_all_canvas,␣
↪edgecolor='b', fill=0)
>>>     undone_scale_canvas = crop_vidspace_poly.warp(undo_scale).draw_on(undone_scale_
```

(continues on next page)

(continued from previous page)

```
↪canvas, edgecolor='b', fill=0)
>>>     #row.append(kwimage.stack_images([undone_all_canvas, undone_scale_canvas],␣
↪axis=0, bg_value=(5, 100, 10), pad=10))
>>>     row.append(undone_all_canvas)
>>>     row.append(undone_scale_canvas)
>>>     print(ub.color_text('============', color))
>>> #
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> tostack_grid = [[_tocanvas(c) for c in cols] for cols in tostack_grid]
>>> tostack_rows  = [kwimage.stack_images(cols, axis=1, bg_value=(5, 100, 10), pad=10)␣
↪for cols in tostack_grid if cols]
>>> stack = kwimage.stack_images(tostack_rows, axis=0, bg_value=(5, 100, 10), pad=10)
>>> kwplot.imshow(stack, title='notice how the "undone all" crops are shifted to the␣
↪right such that they align with the original image')
>>> kwplot.show_if_requested()
```

class kwcoco.util.delayed_ops.**DelayedArray**(*subdata=None*)

Bases: *DelayedUnaryOperation*

A generic NDArray.

property shape

Returns: None | Tuple[int | None, ...]

kwcoco.util.delayed_ops.**DelayedArray2**

alias of *DelayedArray*

class kwcoco.util.delayed_ops.**DelayedAsXarray**(*subdata=None*, *dsize=None*, *channels=None*)

Bases: *DelayedImage*

Casts the data to an xarray object in the finalize step

Example;

```
>>> from kwcoco.util.delayed_ops.delayed_nodes import *  # NOQA
>>> from kwcoco.util.delayed_ops import DelayedLoad
>>> # without channels
>>> base = DelayedLoad.demo(dsize=(16, 16)).prepare()
>>> self = base.as_xarray()
>>> final = self._validate().finalize()
>>> assert len(final.coords) == 0
>>> assert final.dims == ('y', 'x', 'c')
>>> # with channels
>>> base = DelayedLoad.demo(dsize=(16, 16), channels='r|g|b').prepare()
>>> self = base.as_xarray()
>>> final = self._validate().finalize()
>>> assert final.coords.indexes['c'].tolist() == ['r', 'g', 'b']
>>> assert final.dims == ('y', 'x', 'c')
```

optimize()

Returns
DelayedImage

kwcoco.util.delayed_ops.**DelayedAsXarray2**

alias of *DelayedAsXarray*

class kwcoco.util.delayed_ops.**DelayedChannelConcat**(*parts*, *dsize=None*)

Bases: *ImageOpsMixin*, *DelayedConcat*

Stacks multiple arrays together.

CommandLine

```
xdoctest -m /home/joncrall/code/kwcoco/kwcoco/util/delayed_ops/delayed_nodes.py␣
↪DelayedChannelConcat:1
```

Example

```
>>> from kwcoco.util.delayed_ops import *  # NOQA
>>> from kwcoco.util.delayed_ops.delayed_leafs import DelayedLoad
>>> import kwcoco
>>> dsize = (307, 311)
>>> c1 = DelayedNans(dsize=dsize, channels='foo')
>>> c2 = DelayedLoad.demo('astro', dsize=dsize, channels='R|G|B').prepare()
>>> cat = DelayedChannelConcat([c1, c2])
>>> warped_cat = cat.warp({'scale': 1.07}, dsize=(328, 332))
>>> warped_cat._validate()
>>> warped_cat.finalize()
```

Example

```
>>> # Test case that failed in initial implementation
>>> # Due to incorrectly pushing channel selection under the concat
>>> from kwcoco.util.delayed_ops import *  # NOQA
>>> import kwimage
>>> fpath = kwimage.grab_test_image_fpath()
>>> base1 = DelayedLoad(fpath, channels='r|g|b').prepare()
>>> base2 = DelayedLoad(fpath, channels='x|y|z').prepare().scale(2)
>>> base3 = DelayedLoad(fpath, channels='i|j|k').prepare().scale(2)
>>> bands = [base2, base1[:, :, 0].scale(2).evaluate(),
>>>          base1[:, :, 1].evaluate().scale(2),
>>>          base1[:, :, 2].evaluate().scale(2), base3]
>>> delayed = DelayedChannelConcat(bands)
>>> delayed = delayed.warp({'scale': 2})
>>> delayed = delayed[0:100, 0:55, [0, 2, 4]]
>>> delayed.write_network_text()
>>> delayed.optimize()
```

property channels

Returns: None | kwcoco.FusedChannelSpec

property shape

Returns: Tuple[int | None, int | None, int | None]

optimize()

Returns
DelayedImage

take_channels(*channels*)

This method returns a subset of the vision data with only the specified bands / channels.

Parameters
channels (*List[int] | slice | channel_spec.FusedChannelSpec*) – List of integers indexes, a slice, or a channel spec, which is typically a pipe (|) delimited list of channel codes. See *kwcoco.ChannelSpec* for more detials.

Returns
a delayed vision operation that only operates on the following channels.

Return type
DelayedArray

Example

```
>>> from kwcoco.util.delayed_ops.delayed_nodes import *  # NOQA
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('vidshapes8-multispectral')
>>> self = delayed = dset.coco_image(1).delay(mode=1)
>>> channels = 'B11|B8|B1|B10'
>>> new = self.take_channels(channels)
```

Example

```
>>> # Complex case
>>> import kwcoco
>>> from kwcoco.util.delayed_ops.delayed_nodes import *  # NOQA
>>> from kwcoco.util.delayed_ops.delayed_leafs import DelayedLoad
>>> dset = kwcoco.CocoDataset.demo('vidshapes8-multispectral')
>>> delayed = dset.coco_image(1).delay(mode=1)
>>> astro = DelayedLoad.demo('astro', channels='r|g|b').prepare()
>>> aligned = astro.warp(kwimage.Affine.scale(600 / 512), dsize='auto')
>>> self = combo = DelayedChannelConcat(delayed.parts + [aligned])
>>> channels = 'B1|r|B8|g'
>>> new = self.take_channels(channels)
>>> new_cropped = new.crop((slice(10, 200), slice(12, 350)))
>>> new_opt = new_cropped.optimize()
>>> datas = new_opt.finalize()
>>> if 1:
>>>     new_cropped.write_network_text(with_labels='name')
>>>     new_opt.write_network_text(with_labels='name')
>>> vizable = kwimage.normalize_intensity(datas, axis=2)
>>> self._validate()
>>> new._validate()
>>> new_cropped._validate()
>>> new_opt._validate()
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> stacked = kwimage.stack_images(vizable.transpose(2, 0, 1))
>>> kwplot.imshow(stacked)
```

Example

```
>>> # Test case where requested channel does not exist
>>> import kwcoco
>>> from kwcoco.util.delayed_ops.delayed_nodes import *  # NOQA
>>> dset = kwcoco.CocoDataset.demo('vidshapes8-multispectral', use_cache=1,␣
↪verbose=100)
>>> self = delayed = dset.coco_image(1).delay(mode=1)
>>> channels = 'B1|foobar|bazbiz|B8'
>>> new = self.take_channels(channels)
>>> new_cropped = new.crop((slice(10, 200), slice(12, 350)))
>>> fused = new_cropped.finalize()
>>> assert fused.shape == (190, 338, 4)
>>> assert np.all(np.isnan(fused[..., 1:3]))
>>> assert not np.any(np.isnan(fused[..., 0]))
>>> assert not np.any(np.isnan(fused[..., 3]))
```

property num_overviews

Returns: int

as_xarray()

Returns

DelayedAsXarray

undo_warps(*remove=None*, *retain=None*, *squash_nans=False*, *return_warps=False*)

Attempts to "undo" warping for each concatenated channel and returns a list of delayed operations that are cropped to the right regions.

Typically you will retrain offset, theta, and shear to remove scale. This ensures the data is spatially aligned up to a scale factor.

Parameters

- **remove** (*List[str]*) – if specified, list components of the warping to remove. Can include: "offset", "scale", "shearx", "theta". Typically set this to ["scale"].
- **retain** (*List[str]*) – if specified, list components of the warping to retain. Can include: "offset", "scale", "shearx", "theta". Mutually exclusive with "remove". If neither remove or retain is specified, retain is set to `[]`.
- **squash_nans** (*bool*) – if True, pure nan channels are squashed into a 1x1 array as they do not correspond to a real source.
- **return_warps** (*bool*) – if True, return the transforms we applied. This is useful when you need to warp objects in the original space into the jagged space.

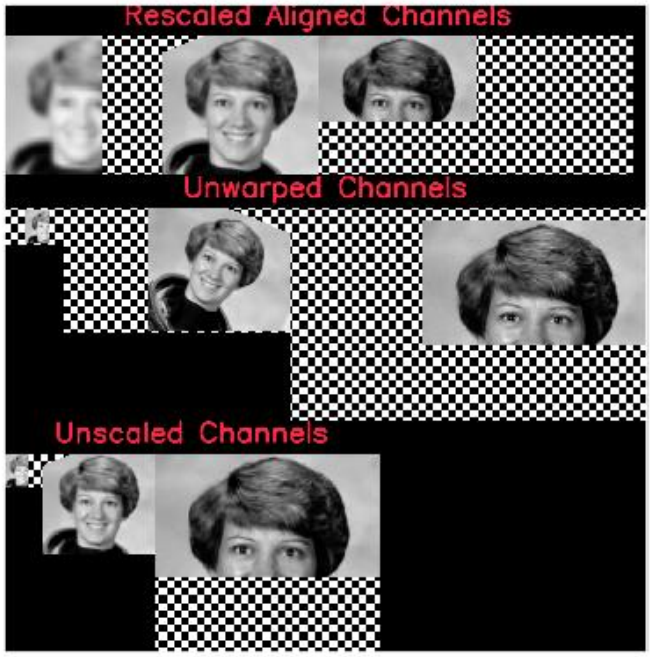
Example

```
>>> from kwcoco.util.delayed_ops.delayed_nodes import *  # NOQA
>>> from kwcoco.util.delayed_ops.delayed_leafs import DelayedLoad
>>> from kwcoco.util.delayed_ops.delayed_leafs import DelayedNans
>>> import ubelt as ub
>>> import kwimage
>>> import kwarray
>>> import numpy as np
>>> # Demo case where we have different channels at different resolutions
>>> base = DelayedLoad.demo(channels='r|g|b').prepare().dequantize({'quant_max
↪': 255})
>>> bandR = base[:, :, 0].scale(100 / 512)[:, :-50].evaluate()
>>> bandG = base[:, :, 1].scale(300 / 512).warp({'theta': np.pi / 8, 'about':␣
↪(150, 150)}).evaluate()
>>> bandB = base[:, :, 2].scale(600 / 512)[:150, :].evaluate()
>>> bandN = DelayedNans((600, 600), channels='N')
>>> # Make a concatenation of images of different underlying native resolutions
>>> delayed_vidspace = DelayedChannelConcat([
>>>     bandR.scale(6, dsize=(600, 600)).optimize(),
>>>     bandG.warp({'theta': -np.pi / 8, 'about': (150, 150)}).scale(2,␣
↪dsize=(600, 600)).optimize(),
>>>     bandB.scale(1, dsize=(600, 600)).optimize(),
>>>     bandN,
>>> ]).warp({'scale': 0.7}).optimize()
>>> vidspace_box = kwimage.Boxes([[100, 10, 270, 160]], 'ltrb')
>>> vidspace_poly = vidspace_box.to_polygons()[0]
>>> vidspace_slice = vidspace_box.to_slices()[0]
>>> self = delayed_vidspace[vidspace_slice].optimize()
>>> print('--- Aligned --- ')
>>> self.write_network_text()
>>> squash_nans = True
>>> undone_all_parts, tfs1 = self.undo_warps(squash_nans=squash_nans, return_
```

(continues on next page)

(continued from previous page)

```
↪warps=True)
>>> undone_scale_parts, tfs2 = self.undo_warps(remove=['scale'], squash_
↪nans=squash_nans, return_warps=True)
>>> stackable_aligned = self.finalize().transpose(2, 0, 1)
>>> stackable_undone_all = []
>>> stackable_undone_scale = []
>>> print('--- Undone All --- ')
>>> for undone in undone_all_parts:
...     undone.write_network_text()
...     stackable_undone_all.append(undone.finalize())
>>> print('--- Undone Scale --- ')
>>> for undone in undone_scale_parts:
...     undone.write_network_text()
...     stackable_undone_scale.append(undone.finalize())
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> canvas0 = kwimage.stack_images(stackable_aligned, axis=1)
>>> canvas1 = kwimage.stack_images(stackable_undone_all, axis=1)
>>> canvas2 = kwimage.stack_images(stackable_undone_scale, axis=1)
>>> canvas0 = kwimage.draw_header_text(canvas0, 'Rescaled Aligned Channels')
>>> canvas1 = kwimage.draw_header_text(canvas1, 'Unwarped Channels')
>>> canvas2 = kwimage.draw_header_text(canvas2, 'Unscaled Channels')
>>> canvas = kwimage.stack_images([canvas0, canvas1, canvas2], axis=0)
>>> canvas = kwimage.fill_nans_with_checkers(canvas)
>>> kwplot.imshow(canvas)
```



kwcoco.util.delayed_ops.**DelayedChannelConcat2**

alias of *DelayedChannelConcat*

class kwcoco.util.delayed_ops.**DelayedConcat**(*parts*, *axis*)

Bases: *DelayedNaryOperation*

Stacks multiple arrays together.

property shape

Returns: None | Tuple[int | None, ...]

kwcoco.util.delayed_ops.**DelayedConcat2**

alias of *DelayedConcat*

class kwcoco.util.delayed_ops.**DelayedCrop**(*subdata*, *space_slice=None*, *chan_idxs=None*)

Bases: *DelayedImage*

Crops an image along integer pixel coordinates.

Example

```
>>> from kwcoco.util.delayed_ops.delayed_nodes import *  # NOQA
>>> from kwcoco.util.delayed_ops import DelayedLoad
>>> base = DelayedLoad.demo(dsize=(16, 16)).prepare()
>>> # Test Fuse Crops Space Only
>>> crop1 = base[4:12, 0:16]
>>> self = crop1[2:6, 0:8]
>>> opt = self._opt_fuse_crops()
>>> self.write_network_text()
>>> opt.write_network_text()
>>> #
>>> # Test Channel Select Via Index
>>> self = base[:, :, [0]]
>>> self.write_network_text()
>>> final = self._finalize()
>>> assert final.shape == (16, 16, 1)
>>> assert base[:, :, [0, 1]].finalize().shape == (16, 16, 2)
>>> assert base[:, :, [2, 0, 1]].finalize().shape == (16, 16, 3)
```

optimize()

Returns
DelayedImage

Example

```
>>> # Test optimize nans
>>> from kwcoco.util.delayed_ops import DelayedNans
>>> import kwimage
>>> base = DelayedNans(dsize=(100, 100), channels='a|b|c')
>>> self = base[0:10, 0:5]
>>> # Should simply return a new nan generator
>>> new = self.optimize()
>>> self.write_network_text()
>>> new.write_network_text()
>>> assert len(new.as_graph().nodes) == 1
```

kwcoco.util.delayed_ops.**DelayedCrop2**

alias of *DelayedCrop*

class kwcoco.util.delayed_ops.**DelayedDequantize**(*subdata*, *quantization*)

Bases: *DelayedImage*

Rescales image intensities from int to floats.

The output is usually between 0 and 1. This also handles transforming nodata into nan values.

optimize()

Returns
DelayedImage

Example

```
>>> # Test a case that caused an error in development
>>> from kwcoco.util.delayed_ops.delayed_nodes import *  # NOQA
>>> from kwcoco.util.delayed_ops import DelayedLoad
>>> fpath = kwimage.grab_test_image_fpath()
>>> base = DelayedLoad(fpath, channels='r|g|b').prepare()
>>> quantization = {'quant_max': 255, 'nodata': 0}
>>> self = base.get_overview(1).dequantize(quantization)
>>> self.write_network_text()
>>> opt = self.optimize()
```

kwcoco.util.delayed_ops.**DelayedDequantize2**

alias of *DelayedDequantize*

class kwcoco.util.delayed_ops.**DelayedFrameStack**(*parts*)

Bases: *DelayedStack*

Stacks multiple arrays together.

kwcoco.util.delayed_ops.**DelayedFrameStack2**

alias of *DelayedFrameStack*

class kwcoco.util.delayed_ops.**DelayedIdentity**(*data*, *channels=None*, *dsize=None*)

Bases: *DelayedImageLeaf*

Returns an ndarray as-is

Example

self = DelayedNans((10, 10), channel_spec.FusedChannelSpec.coerce('rgb')) region_slices = (slice(5, 10), slice(1, 12)) delayed = self.crop(region_slices)

Example

```
>>> from kwcoco.util.delayed_ops import *  # NOQA
>>> import kwcoco
>>> arr = kwimage.checkerboard()
>>> self = DelayedIdentity(arr, channels='gray')
>>> warp = self.warp({'scale': 1.07})
>>> warp.optimize().finalize()
```

kwcoco.util.delayed_ops.**DelayedIdentity2**

alias of *DelayedIdentity*

class kwcoco.util.delayed_ops.**DelayedImage**(*subdata=None*, *dsize=None*, *channels=None*)

Bases: *ImageOpsMixin*, *DelayedArray*

For the case where an array represents a 2D image with multiple channels

property shape

Returns: None | Tuple[int | None, int | None, int | None]

property num_channels

Returns: None | int

property dsize

Returns: None | Tuple[int | None, int | None]

property channels

Returns: None | kwcoco.FusedChannelSpec

property num_overviews

Returns: int

take_channels(*channels*)

This method returns a subset of the vision data with only the specified bands / channels.

Parameters

channels (*List[int] | slice | channel_spec.FusedChannelSpec*) – List of integers indexes, a slice, or a channel spec, which is typically a pipe (|) delimited list of channel codes. See kwcoco.ChannelSpec for more detials.

Returns

a new delayed load with a fused take channel operation

Return type

DelayedCrop

Note: The channel subset must exist here or it will raise an error. A better implementation (via pymbolic) might be able to do better

---

Example

```
>>> #
>>> # Test Channel Select Via Code
>>> from kwcoco.util.delayed_ops.delayed_nodes import *  # NOQA
>>> from kwcoco.util.delayed_ops import DelayedLoad
>>> self = DelayedLoad.demo(dsize=(16, 16), channels='r|g|b').prepare()
>>> channels = 'r|b'
>>> new = self.take_channels(channels)._validate()
>>> new2 = new[:, :, [1, 0]]._validate()
>>> new3 = new2[:, :, [1]]._validate()
```

Example

```
>>> from kwcoco.util.delayed_ops.delayed_nodes import *  # NOQA
>>> from kwcoco.util.delayed_ops import DelayedLoad
>>> import kwcoco
>>> self = DelayedLoad.demo('astro').prepare()
>>> channels = [2, 0]
>>> new = self.take_channels(channels)
>>> new3 = new.take_channels([1, 0])
>>> new._validate()
>>> new3._validate()
```

```
>>> final1 = self.finalize()
>>> final2 = new.finalize()
>>> final3 = new3.finalize()
>>> assert np.all(final1[..., 2] == final2[..., 0])
>>> assert np.all(final1[..., 0] == final2[..., 1])
>>> assert final2.shape[2] == 2
```

```
>>> assert np.all(final1[..., 2] == final3[..., 1])
>>> assert np.all(final1[..., 0] == final3[..., 0])
>>> assert final3.shape[2] == 2
```

get_transform_from_leaf()

Returns the transformation that would align data with the leaf

evaluate()

Evaluate this node and return the data as an identity.

Returns

DelayedIdentity

undo_warp(*remove=None*, *retain=None*, *squash_nans=False*, *return_warp=False*)

Attempts to "undo" warping for each concatenated channel and returns a list of delayed operations that are cropped to the right regions.

Typically you will retrain offset, theta, and shear to remove scale. This ensures the data is spatially aligned up to a scale factor.

Parameters

- **remove** (*List[str]*) – if specified, list components of the warping to remove. Can include: "offset", "scale", "shearx", "theta". Typically set this to ["scale"].
- **retain** (*List[str]*) – if specified, list components of the warping to retain. Can include: "offset", "scale", "shearx", "theta". Mutually exclusive with "remove". If neither remove or retain is specified, retain is set to `[]`.
- **squash_nans** (*bool*) – if True, pure nan channels are squashed into a 1x1 array as they do not correspond to a real source.
- **return_warp** (*bool*) – if True, return the transform we applied. This is useful when you need to warp objects in the original space into the jagged space.

SeeAlso:

DelayedChannelConcat.undo_warps

Example

```
>>> # Test similar to undo_warps, but on each channel separately
>>> from kwcoco.util.delayed_ops.delayed_nodes import *  # NOQA
>>> from kwcoco.util.delayed_ops.delayed_leafs import DelayedLoad
>>> from kwcoco.util.delayed_ops.delayed_leafs import DelayedNans
>>> import ubelt as ub
>>> import kwimage
>>> import kwarray
>>> import numpy as np
```

(continues on next page)

(continued from previous page)

```
>>> # Demo case where we have different channels at different resolutions
>>> base = DelayedLoad.demo(channels='r|g|b').prepare().dequantize({'quant_max
↪': 255})
>>> bandR = base[:, :, 0].scale(100 / 512)[:, :-50].evaluate()
>>> bandG = base[:, :, 1].scale(300 / 512).warp({'theta': np.pi / 8, 'about':␣
↪(150, 150)}).evaluate()
>>> bandB = base[:, :, 2].scale(600 / 512)[:150, :].evaluate()
>>> bandN = DelayedNans((600, 600), channels='N')
>>> B0 = bandR.scale(6, dsize=(600, 600)).optimize()
>>> B1 = bandG.warp({'theta': -np.pi / 8, 'about': (150, 150)}).scale(2,␣
↪dsize=(600, 600)).optimize()
>>> B2 = bandB.scale(1, dsize=(600, 600)).optimize()
>>> vidspace_box = kwimage.Boxes([[-10, -10, 270, 160]], 'ltrb').scale(1 / .7).
↪quantize()
>>> vidspace_poly = vidspace_box.to_polygons()[0]
>>> vidspace_slice = vidspace_box.to_slices()[0]
>>> # Test with the padded crop
>>> self0 = B0.crop(vidspace_slice, wrap=0, clip=0, pad=10).optimize()
>>> self1 = B1.crop(vidspace_slice, wrap=0, clip=0, pad=10).optimize()
>>> self2 = B2.crop(vidspace_slice, wrap=0, clip=0, pad=10).optimize()
>>> parts = [self0, self1, self2]
>>> # Run the undo on each channel
>>> undone_scale_parts = [d.undo_warp(remove=['scale']) for d in parts]
>>> print('--- Aligned --- ')
>>> stackable_aligned = []
>>> for d in parts:
>>>     d.write_network_text()
>>>     stackable_aligned.append(d.finalize())
>>> print('--- Undone Scale --- ')
>>> stackable_undone_scale = []
>>> for undone in undone_scale_parts:
...     undone.write_network_text()
...     stackable_undone_scale.append(undone.finalize())
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> canvas0 = kwimage.stack_images(stackable_aligned, axis=1, pad=5, bg_value=
↪'kw_darkgray')
>>> canvas2 = kwimage.stack_images(stackable_undone_scale, axis=1, pad=5, bg_
↪value='kw_darkgray')
>>> canvas0 = kwimage.draw_header_text(canvas0, 'Rescaled Channels')
>>> canvas2 = kwimage.draw_header_text(canvas2, 'Native Scale Channels')
>>> canvas = kwimage.stack_images([canvas0, canvas2], axis=0, bg_value='kw_
↪darkgray')
>>> canvas = kwimage.fill_nans_with_checkers(canvas)
>>> kwplot.imshow(canvas)
```

kwcoco.util.delayed_ops.**DelayedImage2**

alias of *DelayedImage*

class kwcoco.util.delayed_ops.**DelayedImageLeaf**(*subdata=None*, *dsize=None*, *channels=None*)

Bases: *DelayedImage*

get_transform_from_leaf()

Returns the transformation that would align data with the leaf

Returns

kwimage.Affine

optimize()

kwcoco.util.delayed_ops.**DelayedImageLeaf2**

alias of *DelayedImageLeaf*

class kwcoco.util.delayed_ops.**DelayedLoad**(*fpath*, *channels=None*, *dsize=None*, *nodata_method=None*)

Bases: *DelayedImageLeaf*

Reads an image from disk.

If a gdal backend is available, and the underlying image is in the appropriate formate (e.g. COG) this will return a lazy reference that enables fast overviews and crops.

Example

```
>>> from kwcoco.util.delayed_ops import *  # NOQA
>>> self = DelayedLoad.demo(dsize=(16, 16)).prepare()
>>> data1 = self.finalize()
```

Example

```
>>> # xdoctest: +REQUIRES(module:osgeo)
>>> # Demo code to develop support for overviews
>>> from kwcoco.util.delayed_ops import *  # NOQA
>>> import kwimage
>>> import ubelt as ub
>>> fpath = kwimage.grab_test_image_fpath(overviews=3)
>>> self = DelayedLoad(fpath, channels='r|g|b').prepare()
>>> print(f'self={self}')
>>> print('self.meta = {}'.format(ub.repr2(self.meta, nl=1)))
>>> quantization = {
>>>     'quant_max': 255,
>>>     'nodata': 0,
>>> }
>>> node0 = self
>>> node1 = node0.get_overview(2)
>>> node2 = node1[13:900, 11:700]
>>> node3 = node2.dequantize(quantization)
>>> node4 = node3.warp({'scale': 0.05})
>>> #
>>> data0 = node0._validate().finalize()
>>> data1 = node1._validate().finalize()
>>> data2 = node2._validate().finalize()
>>> data3 = node3._validate().finalize()
>>> data4 = node4._validate().finalize()
>>> node4.write_network_text()
```

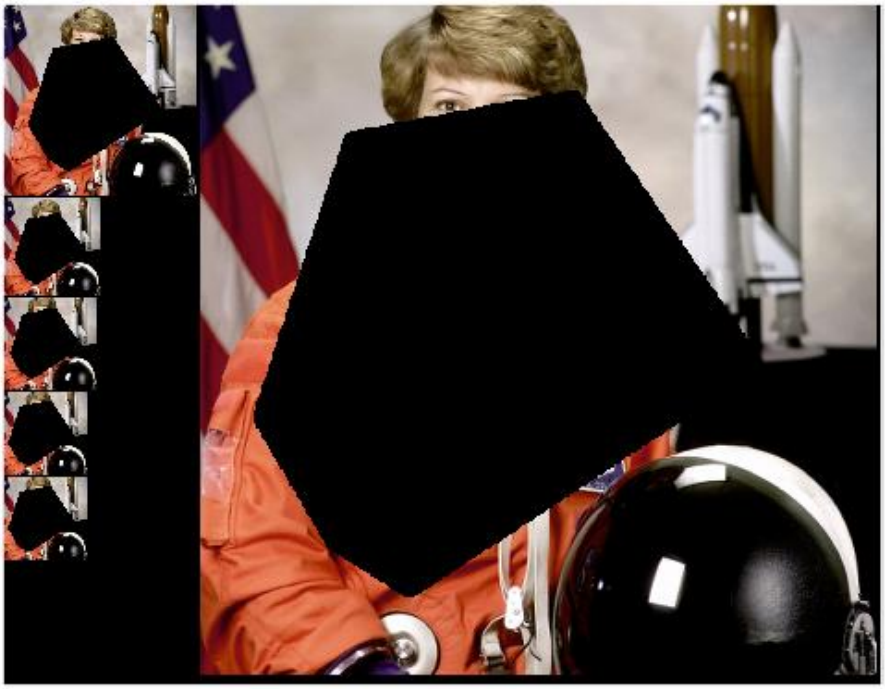
Example

```
>>> # xdoctest: +REQUIRES(module:osgeo)
>>> # Test delayed ops with int16 and nodata values
>>> from kwcoco.util.delayed_ops import *  # NOQA
>>> import kwimage
>>> from kwcoco.util.delayed_ops.helpers import quantize_float01
>>> import ubelt as ub
>>> dpath = ub.Path.appdir('kwcoco/tests/test_delay_nodata').ensuredir()
>>> fpath = dpath / 'data.tif'
>>> data = kwimage.ensure_float01(kwimage.grab_test_image())
>>> poly = kwimage.Polygon.random(rng=321032).scale(data.shape[0])
>>> poly.fill(data, np.nan)
>>> data_uint16, quantization = quantize_float01(data)
>>> nodata = quantization['nodata']
>>> kwimage.imwrite(fpath, data_uint16, nodata=nodata, backend='gdal', overviews=3)
```

(continues on next page)

(continued from previous page)

```
>>> # Test loading the data
>>> self = DelayedLoad(fpath, channels='r|g|b', nodata_method='float').prepare()
>>> node0 = self
>>> node1 = node0.dequantize(quantization)
>>> node2 = node1.warp({'scale': 0.51}, interpolation='lanczos')
>>> node3 = node2[13:900, 11:700]
>>> node4 = node3.warp({'scale': 0.9}, interpolation='lanczos')
>>> node4.write_network_text()
>>> node5 = node4.optimize()
>>> node5.write_network_text()
>>> node6 = node5.warp({'scale': 8}, interpolation='lanczos').optimize()
>>> node6.write_network_text()
>>> #
>>> data0 = node0._validate().finalize()
>>> data1 = node1._validate().finalize()
>>> data2 = node2._validate().finalize()
>>> data3 = node3._validate().finalize()
>>> data4 = node4._validate().finalize()
>>> data5 = node5._validate().finalize()
>>> data6 = node6._validate().finalize()
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> stack1 = kwimage.stack_images([data1, data2, data3, data4, data5])
>>> stack2 = kwimage.stack_images([stack1, data6], axis=1)
>>> kwplot.imshow(stack2)
```



property fpath

classmethod demo(*key='astro'*, *dsize=None*, *channels=None*)

prepare()

If metadata is missing, perform minimal IO operations in order to prepopulate metadata that could help us better optimize the operation tree.

Returns
DelayedLoad

kwcoco.util.delayed_ops.**DelayedLoad2**

alias of *DelayedLoad*

class kwcoco.util.delayed_ops.**DelayedNans**(*dsize=None*, *channels=None*)

Bases: *DelayedImageLeaf*

Constructs nan channels as needed

Example

self = DelayedNans((10, 10), channel_spec.FusedChannelSpec.coerce('rgb')) region_slices = (slice(5, 10), slice(1, 12)) delayed = self.crop(region_slices)

Example

```
>>> from kwcoco.util.delayed_ops import *  # NOQA
>>> import kwcoco
>>> dsize = (307, 311)
>>> c1 = DelayedNans(dsize=dsize, channels='foo')
>>> c2 = DelayedLoad.demo('astro', dsize=dsize, channels='R|G|B').prepare()
>>> cat = DelayedChannelConcat([c1, c2])
>>> warped_cat = cat.warp({'scale': 1.07}, dsize=(328, 332))._validate()
>>> warped_cat._validate().optimize().finalize()
```

kwcoco.util.delayed_ops.**DelayedNans2**

alias of *DelayedNans*

class kwcoco.util.delayed_ops.**DelayedNaryOperation**(*parts*)

Bases: *DelayedOperation*

For operations that have multiple input arrays

children()

Yields
Any

kwcoco.util.delayed_ops.**DelayedNaryOperation2**

alias of *DelayedNaryOperation*

class kwcoco.util.delayed_ops.**DelayedOperation**

Bases: NiceRepr

nesting()

Returns
Dict[str, dict]

as_graph()

Returns
networkx.DiGraph

write_network_text(*with_labels=True*)

property shape

Returns: None | Tuple[int | None, ...]

children()

Yields
Any

prepare()

If metadata is missing, perform minimal IO operations in order to prepopulate metadata that could help us better optimize the operation tree.

Returns
DelayedOperation2

finalize(*prepare=True*, *optimize=True*, ***kwargs*)

Evaluate the operation tree in full.

Parameters

- **prepare** (*bool*) – ensure prepare is called to ensure metadata exists if possible before optimizing. Defaults to True.
- **optimize** (*bool*) – ensure the graph is optimized before loading. Default to True.
- ****kwargs** – for backwards compatibility, these will allow for in-place modification of select nested parameters. In general these should not be used, and may be deprecated.

Returns
ArrayLike

Notes

Do not overload this method. Overload `DelayedOperation2._finalize()` instead.

optimize()

Returns
DelayedOperation2

kwcoco.util.delayed_ops.**DelayedOperation2**

alias of *DelayedOperation*

class kwcoco.util.delayed_ops.**DelayedOverview**(*subdata*, *overview*)

Bases: *DelayedImage*

Downsamples an image by a factor of two.

If the underlying image being loaded has precomputed overviews it simply loads these instead of downsampling the original image, which is more efficient.

Example

```
>>> # xdoctest: +REQUIRES(module:osgeo)
>>> # Make a complex chain of operations and optimize it
>>> from kwcoco.util.delayed_ops import *  # NOQA
>>> import kwimage
>>> fpath = kwimage.grab_test_image_fpath(overviews=3)
>>> dimg = DelayedLoad(fpath, channels='r|g|b').prepare()
>>> dimg = dimg.get_overview(1)
>>> dimg = dimg.get_overview(1)
>>> dimg = dimg.get_overview(1)
>>> dopt = dimg.optimize()
>>> if 1:
>>>     import networkx as nx
```

(continues on next page)

(continued from previous page)

```
>>>     dimg.write_network_text()
>>>     dopt.write_network_text()
>>> print(ub.repr2(dopt.nesting(), nl=-1, sort=0))
>>> final0 = dimg._finalize()[:]
>>> final1 = dopt._finalize()[:]
>>> assert final0.shape == final1.shape
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(final0, pnum=(1, 2, 1), fnum=1, title='raw')
>>> kwplot.imshow(final1, pnum=(1, 2, 2), fnum=1, title='optimized')
```

property num_overviews

Returns: int

optimize()

Returns

DelayedImage

kwcoco.util.delayed_ops.**DelayedOverview2**

alias of *DelayedOverview*

class kwcoco.util.delayed_ops.**DelayedStack**(*parts*, *axis*)

Bases: *DelayedNaryOperation*

Stacks multiple arrays together.

property shape

Returns: None | Tuple[int | None, ...]

kwcoco.util.delayed_ops.**DelayedStack2**

alias of *DelayedStack*

class kwcoco.util.delayed_ops.**DelayedUnaryOperation**(*subdata*)

Bases: *DelayedOperation*

For operations that have a single input array

children()

Yields
Any

kwcoco.util.delayed_ops.**DelayedUnaryOperation2**

alias of *DelayedUnaryOperation*

class kwcoco.util.delayed_ops.**DelayedWarp**(*subdata*, *transform*, *dsize='auto'*, *antialias=True*, *interpolation='linear'*, *border_value='auto'*)

Bases: *DelayedImage*

Applies an affine transform to an image.

Example

```
>>> from kwcoco.util.delayed_ops.delayed_nodes import *  # NOQA
>>> from kwcoco.util.delayed_ops import DelayedLoad
>>> self = DelayedLoad.demo(dsize=(16, 16)).prepare()
>>> warp1 = self.warp({'scale': 3})
>>> warp2 = warp1.warp({'theta': 0.1})
>>> warp3 = warp2._opt_fuse_warps()
>>> warp3._validate()
>>> print(ub.repr2(warp2.nesting(), nl=-1, sort=0))
>>> print(ub.repr2(warp3.nesting(), nl=-1, sort=0))
```

property transform

Returns: kwimage.Affine

optimize()

Returns
DelayedImage

Example

```
>>> # Demo optimization that removes a noop warp
>>> from kwcoco.util.delayed_ops import DelayedLoad
>>> import kwimage
>>> base = DelayedLoad.demo(channels='r|g|b').prepare()
>>> self = base.warp(kwimage.Affine.eye())
>>> new = self.optimize()
>>> assert len(self.as_graph().nodes) == 2
>>> assert len(new.as_graph().nodes) == 1
```

Example

```
>>> # Test optimize nans
>>> from kwcoco.util.delayed_ops import DelayedNans
>>> import kwimage
>>> base = DelayedNans(dsize=(100, 100), channels='a|b|c')
>>> self = base.warp(kwimage.Affine.scale(0.1))
>>> # Should simply return a new nan generator
>>> new = self.optimize()
>>> assert len(new.as_graph().nodes) == 1
```

kwcoco.util.delayed_ops.**DelayedWarp2**

alias of *DelayedWarp*

class kwcoco.util.delayed_ops.**ImageOpsMixin**

Bases: object

crop(*space_slice=None*, *chan_idxs=None*, *clip=True*, *wrap=True*, *pad=0*)

Crops an image along integer pixel coordinates.

Parameters

- **space_slice** (*Tuple[slice, slice]*) – y-slice and x-slice.
- **chan_idxs** (*List[int]*) – indexes of bands to take
- **clip** (*bool*) – if True, the slice is interpreted normally, where it won't go past the image extent, otherwise slicing into negative regions or past the image bounds will result in padding. Defaults to True.
- **wrap** (*bool*) – if True, negative indexes "wrap around", otherwise they are treated as is. Defaults to True.
- **pad** (*int | List[Tuple[int, int]]*) – if specified, applies extra padding

Returns

DelayedImage

Example

```
>>> from kwcoco.util.delayed_ops import DelayedLoad
>>> import kwimage
>>> self = DelayedLoad.demo().prepare()
>>> self = self.dequantize({'quant_max': 255})
>>> self = self.warp({'scale': 1 / 2})
>>> pad = 0
>>> h, w = space_dims = self.dsize[::-1]
>>> grid = list(ub.named_product({
>>>     'left': [0, -64], 'right': [0, 64],
>>>     'top': [0, -64], 'bot': [0, 64],}))
>>> grid += [
>>>     {'left': 64, 'right': -64, 'top': 0, 'bot': 0},
>>>     {'left': 64, 'right': 64, 'top': 0, 'bot': 0},
>>>     {'left': 0, 'right': 0, 'top': 64, 'bot': -64},
>>>     {'left': 64, 'right': -64, 'top': 64, 'bot': -64},
>>> ]
>>> crops = []
>>> for pads in grid:
>>>     space_slice = (slice(pads['top'], h + pads['bot']),
>>>                    slice(pads['left'], w + pads['right']))
>>>     delayed = self.crop(space_slice)
>>>     crop = delayed.finalize()
>>>     yyxx = kwimage.Boxes.from_slice(space_slice, wrap=False, clip=0).
↪toformat('_yyxx').data[0]
>>>     title = '[{}:{}, {}:{}]'.format(*yyxx)
>>>     crop_canvas = kwimage.draw_header_text(crop, title, fit=True, bg_color=
↪'kw_darkgray')
>>>     crops.append(crop_canvas)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> canvas = kwimage.stack_images_grid(crops, pad=16, bg_value='kw_darkgreen')
>>> canvas = kwimage.fill_nans_with_checkers(canvas)
>>> kwplot.imshow(canvas, title='Normal Slicing: Cropped Images With␣
↪Wrap+Clipped Slices', doclf=1, fnum=1)
>>> kwplot.show_if_requested()
```

Example

```
>>> # Demo the case with pads / no-clips / no-wraps
>>> from kwcoco.util.delayed_ops import DelayedLoad
>>> import kwimage
>>> self = DelayedLoad.demo().prepare()
>>> self = self.dequantize({'quant_max': 255})
>>> self = self.warp({'scale': 1 / 2})
>>> pad = [(64, 128), (32, 96)]
>>> pad = [(0, 20), (0, 0)]
>>> pad = 0
>>> pad = 8
>>> h, w = space_dims = self.dsize[::-1]
>>> grid = list(ub.named_product({
>>>     'left': [0, -64], 'right': [0, 64],
>>>     'top': [0, -64], 'bot': [0, 64],}))
>>> grid += [
>>>     {'left': 64, 'right': -64, 'top': 0, 'bot': 0},
>>>     {'left': 64, 'right': 64, 'top': 0, 'bot': 0},
>>>     {'left': 0, 'right': 0, 'top': 64, 'bot': -64},
>>>     {'left': 64, 'right': -64, 'top': 64, 'bot': -64},
>>> ]
>>> crops = []
>>> for pads in grid:
>>>     space_slice = (slice(pads['top'], h + pads['bot']),
```

(continues on next page)

(continued from previous page)

```
>>>                     slice(pads['left'], w + pads['right']))
>>>     delayed = self._padded_crop(space_slice, pad=pad)
>>>     crop = delayed.finalize(optimize=1)
>>>     yyxx = kwimage.Boxes.from_slice(space_slice, wrap=False, clip=0).
↪toformat('_yyxx').data[0]
>>>     title = '[{}:{}, {}:{}]'.format(*yyxx)
>>>     if pad:
>>>         title += f'{chr(10)}pad={pad}'
>>>     crop_canvas = kwimage.draw_header_text(crop, title, fit=True, bg_color=
↪'kw_darkgray')
>>>     crops.append(crop_canvas)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> canvas = kwimage.stack_images_grid(crops, pad=16, bg_value='kw_darkgreen',␣
↪resize='smaller')
>>> canvas = kwimage.fill_nans_with_checkers(canvas)
>>> kwplot.imshow(canvas, title='Negative Slicing: Cropped Images With␣
↪clip=False wrap=False', doclf=1, fnum=2)
>>> kwplot.show_if_requested()
```

warp(*transform*, *dsize='auto'*, *antialias=True*, *interpolation='linear'*, *border_value='auto'*)

Applys an affine transformation to the image

Parameters

- **transform** (*ndarray | dict | kwimage.Affine*) – a coercable affine matrix. See `kwimage.Affine` for details on what can be coerced.
- **dsize** (*Tuple[int, int] | str*) – The width / height of the output canvas. If 'auto', dsize is computed such that the positive coordinates of the warped image will fit in the new canvas. In this case, any pixel that maps to a negative coordinate will be clipped. This has the property that the input transformation is not modified.
- **antialias** (*bool*) – if True determines if the transform is downsampling and applies antialiasing via gaussian a blur. Defaults to False
- **interpolation** (*str*) – interpolation code or cv2 integer. Interpolation codes are linear, nearest, cubic, lancsoz, and area. Defaults to "linear".
- **border_value** (*int | float | str*) – if auto will be nan for float and 0 for int.

Returns
DelayedImage

`scale`(*scale*, *dsize='auto'*, *antialias=True*, *interpolation='linear'*, *border_value='auto'*)

An alias for self.warp({"scale": scale}, ...)

`dequantize`(*quantization*)

Rescales image intensities from int to floats.

Parameters
quantization (*Dict[str, Any]*) – see *kwcoco.util.delayed_ops.helpers.dequantize()*

Returns
DelayedDequantize

`get_overview`(*overview*)

Downsamples an image by a factor of two.

Parameters
overview (*int*) – the overview to use (assuming it exists)

Returns
DelayedOverview

`as_xarray`()

Returns
DelayedAsXarray

2.1.1.6.2 Submodules

2.1.1.6.2.1 kwcoco.util.dict_like module

`class` kwcoco.util.dict_like.`DictLike`

Bases: `NiceRepr`

An inherited class must specify the `getitem`, `setitem`, and `keys` methods.

A class is dictionary like if it has:

`__iter__`, `__len__`, `__contains__`, `__getitem__`, `items`, `keys`, `values`, `get`,

and if it should be writable it should have: `__delitem__`, `__setitem__`, `update`,

And perhaps: `copy`,

`__iter__`, `__len__`, `__contains__`, `__getitem__`, `items`, `keys`, `values`, `get`,

and if it should be writable it should have: `__delitem__`, `__setitem__`, `update`,

And perhaps: `copy`,

getitem(*key*)

Parameters
key (*Any*) – a key

Returns
a value

Return type
Any

setitem(*key*, *value*)

Parameters

- **key** (*Any*)
- **value** (*Any*)

delitem(*key*)

Parameters
key (*Any*)

keys()

Yields
Any – a key

items()

Yields
Tuple[Any, Any] – a key value pair

values()

Yields
Any – a value

copy()

Return type
Dict

to_dict()

Return type
Dict

asdict()

Return type
Dict

update(*other*)

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

Parameters

- **key** (*Any*)
- **default** (*Any*)

Return type
Any

2.1.1.6.2.2 kwcoco.util.jsonschema_elements module

Functional interface into defining jsonschema structures.

See mixin classes for details.

Example

```
>>> from kwcoco.util.jsonschema_elements import *  # NOQA
>>> elem = SchemaElements()
>>> for base in SchemaElements.__bases__:
>>>     print('\n\n====\nbase = {!r}'.format(base))
>>>     attrs = [key for key in dir(base) if not key.startswith('_')]
>>>     for key in attrs:
>>>         value = getattr(elem, key)
>>>         print('{} = {}'.format(key, value))
```

class kwcoco.util.jsonschema_elements.**Element**(*base*, *options={}*, *_magic=None*)

Bases: dict

A dictionary used to define an element of a JSON Schema.

The exact keys/values for the element will depend on the type of element being described. The *SchemaElements* defines exactly what these are for the core elements. (e.g. OBJECT, INTEGER, NULL, ARRAY, ANYOF)

Example

```
>>> from kwcoco.coco_schema import *  # NOQA
>>> self = Element(base={'type': 'demo'}, options={'opt1', 'opt2'})
>>> new = self(opt1=3)
>>> print('self = {}'.format(ub.repr2(self, nl=1, sort=1)))
>>> print('new = {}'.format(ub.repr2(new, nl=1, sort=1)))
>>> print('new2 = {}'.format(ub.repr2(new(), nl=1, sort=1)))
>>> print('new3 = {}'.format(ub.repr2(new(title='myvar'), nl=1, sort=1)))
>>> print('new4 = {}'.format(ub.repr2(new(title='myvar')(examples=['']), nl=1,␣
↪sort=1)))
>>> print('new5 = {}'.format(ub.repr2(new(badattr=True), nl=1, sort=1)))
self = {
    'type': 'demo',
}
```

(continues on next page)

(continued from previous page)

```
new = {
    'opt1': 3,
    'type': 'demo',
}
new2 = {
    'opt1': 3,
    'type': 'demo',
}
new3 = {
    'opt1': 3,
    'title': 'myvar',
    'type': 'demo',
}
new4 = {
    'examples': [''],
    'opt1': 3,
    'title': 'myvar',
    'type': 'demo',
}
new5 = {
    'opt1': 3,
    'type': 'demo',
}
```

validate(*instance=NoParam*)

If `instance` is given, validates that that dictionary conforms to this schema. Otherwise validates that this is a valid schema element.

Parameters
instance (*dict*) – a dictionary to validate

class kwcoco.util.jsonschema_elements.**ScalarElements**

Bases: `object`

Single-valued elements

property NULL

//json-schema.org/understanding-json-schema/reference/null.html

Type
https

property BOOLEAN

//json-schema.org/understanding-json-schema/reference/null.html

Type
https

property STRING

//json-schema.org/understanding-json-schema/reference/string.html

Type
https

property NUMBER

//json-schema.org/understanding-json-schema/reference/numeric.html#number

Type
https

property INTEGER

//json-schema.org/understanding-json-schema/reference/numeric.html#integer

Type
https

class kwcoco.util.jsonschema_elements.**QuantifierElements**

Bases: object

Quantifier types

https://json-schema.org/understanding-json-schema/reference/combining.html#allof

Example

```
>>> from kwcoco.util.jsonschema_elements import *  # NOQA
>>> elem.ANYOF(elem.STRING, elem.NUMBER).validate()
>>> elem.ONEOF(elem.STRING, elem.NUMBER).validate()
>>> elem.NOT(elem.NULL).validate()
>>> elem.NOT(elem.ANY).validate()
>>> elem.ANY.validate()
```

property ANY

ALLOF(**TYPES*)

ANYOF(**TYPES*)

ONEOF(**TYPES*)

NOT(*TYPE*)

class kwcoco.util.jsonschema_elements.**ContainerElements**

Bases: object

Types that contain other types

Example

```
>>> from kwcoco.util.jsonschema_elements import *  # NOQA
>>> print(elem.ARRAY().validate())
>>> print(elem.OBJECT().validate())
>>> print(elem.OBJECT().validate())
{'type': 'array', 'items': {}}
{'type': 'object', 'properties': {}}
{'type': 'object', 'properties': {}}
```

ARRAY(*TYPE={}*, ***kw*)

https://json-schema.org/understanding-json-schema/reference/array.html

Example

```
>>> from kwcoco.util.jsonschema_elements import *  # NOQA
>>> ARRAY(numItems=3)
>>> schema = ARRAY(minItems=3)
>>> schema.validate()
{'type': 'array', 'items': {}, 'minItems': 3}
```

OBJECT(*PROPERTIES={}*, ***kw*)

https://json-schema.org/understanding-json-schema/reference/object.html

Example

```
>>> import jsonschema
>>> schema = elem.OBJECT()
>>> jsonschema.validate({}, schema)
>>> #
>>> import jsonschema
>>> schema = elem.OBJECT({
>>>     'key1': elem.ANY(),
>>>     'key2': elem.ANY(),
>>> }, required=['key1'])
>>> jsonschema.validate({'key1': None}, schema)
>>> #
>>> import jsonschema
>>> schema = elem.OBJECT({
>>>     'key1': elem.OBJECT({'arr': elem.ARRAY()}),
>>>     'key2': elem.ANY(),
>>> }, required=['key1'], title='a title')
>>> schema.validate()
>>> print('schema = {}'.format(ub.repr2(schema, sort=1, nl=-1)))
>>> jsonschema.validate({'key1': {'arr': []}}, schema)
schema = {
    'properties': {
        'key1': {
            'properties': {
                'arr': {'items': {}, 'type': 'array'}
            },
            'type': 'object'
        },
        'key2': {}
    },
    'required': ['key1'],
    'title': 'a title',
    'type': 'object'
}
```

class kwcoco.util.jsonschema_elements.**SchemaElements**

Bases: *ScalarElements*, *QuantifierElements*, *ContainerElements*

Functional interface into defining jsonschema structures.

See mixin classes for details.

References

https://json-schema.org/understanding-json-schema/

Todo:

- [ ] Generics: title, description, default, examples

CommandLine

```
xdoctest -m /home/joncrall/code/kwcoco/kwcoco/util/jsonschema_elements.py␣
↪SchemaElements
```

Example

```
>>> from kwcoco.util.jsonschema_elements import *  # NOQA
>>> elem = SchemaElements()
>>> elem.ARRAY(elem.ANY())
>>> schema = OBJECT({
>>>     'prop1': ARRAY(INTEGER, minItems=3),
>>>     'prop2': ARRAY(STRING, numItems=2),
>>>     'prop3': ARRAY(OBJECT({
>>>         'subprob1': NUMBER,
>>>         'subprob2': NUMBER,
>>>     }))
>>> })
>>> print('schema = {}'.format(ub.repr2(schema, nl=2, sort=1)))
schema = {
    'properties': {
        'prop1': {'items': {'type': 'integer'}, 'minItems': 3, 'type': 'array'},
        'prop2': {'items': {'type': 'string'}, 'maxItems': 2, 'minItems': 2, 'type
↪': 'array'},
        'prop3': {'items': {'properties': {'subprob1': {'type': 'number'}, 'subprob2
↪': {'type': 'number'}}, 'type': 'object'}, 'type': 'array'},
    },
    'type': 'object',
}
```

```
>>> TYPE = elem.OBJECT({
>>>     'p1': ANY,
>>>     'p2': ANY,
>>> }, required=['p1'])
>>> import jsonschema
>>> inst = {'p1': None}
>>> jsonschema.validate(inst, schema=TYPE)
>>> #jsonschema.validate({'p2': None}, schema=TYPE)
```

kwcoco.util.jsonschema_elements.**ALLOF**(**TYPES*)

kwcoco.util.jsonschema_elements.**ANYOF**(**TYPES*)

kwcoco.util.jsonschema_elements.**ARRAY**(*TYPE={}*, ***kw*)

https://json-schema.org/understanding-json-schema/reference/array.html

Example

```
>>> from kwcoco.util.jsonschema_elements import *  # NOQA
>>> ARRAY(numItems=3)
>>> schema = ARRAY(minItems=3)
>>> schema.validate()
{'type': 'array', 'items': {}, 'minItems': 3}
```

kwcoco.util.jsonschema_elements.**NOT**(*TYPE*)

kwcoco.util.jsonschema_elements.**OBJECT**(*PROPERTIES={}*, ***kw*)

https://json-schema.org/understanding-json-schema/reference/object.html

Example

```
>>> import jsonschema
>>> schema = elem.OBJECT()
>>> jsonschema.validate({}, schema)
>>> #
>>> import jsonschema
>>> schema = elem.OBJECT({
>>>     'key1': elem.ANY(),
>>>     'key2': elem.ANY(),
>>> }, required=['key1'])
>>> jsonschema.validate({'key1': None}, schema)
>>> #
>>> import jsonschema
>>> schema = elem.OBJECT({
>>>     'key1': elem.OBJECT({'arr': elem.ARRAY()}),
>>>     'key2': elem.ANY(),
>>> }, required=['key1'], title='a title')
>>> schema.validate()
>>> print('schema = {}'.format(ub.repr2(schema, sort=1, nl=-1)))
>>> jsonschema.validate({'key1': {'arr': []}}, schema)
schema = {
    'properties': {
        'key1': {
            'properties': {
                'arr': {'items': {}, 'type': 'array'}
            },
            'type': 'object'
        },
        'key2': {}
    },
    'required': ['key1'],
    'title': 'a title',
    'type': 'object'
}
```

kwcoco.util.jsonschema_elements.**ONEOF**(**TYPES*)

2.1.1.6.2.3 kwcoco.util.lazy_frame_backends module

Ducktyped interfaces for loading subregions of images with standard slice syntax

class kwcoco.util.lazy_frame_backends.**CacheDict**(**args*, *cache_len: int = 10*, ***kwargs*)

Bases: OrderedDict

Dict with a limited length, ejecting LRUs as needed.

Example

```
>>> c = CacheDict(cache_len=2)
>>> c[1] = 1
>>> c[2] = 2
>>> c[3] = 3
>>> c
CacheDict([(2, 2), (3, 3)])
>>> c[2]
2
>>> c[4] = 4
>>> c
CacheDict([(2, 2), (4, 4)])
>>>
```

References

https://gist.github.com/davesteele/44793cd0348f59f8fadd49d7799bd306

class kwcoco.util.lazy_frame_backends.**LazySpectralFrameFile**(*fpath*)

Bases: NiceRepr

Potentially faster than GDAL for HDR formats.

classmethod available()

Returns True if this backend is available

property ndim

property shape

property dtype

class kwcoco.util.lazy_frame_backends.**LazyRasterIOFrameFile**(*fpath*)

Bases: NiceRepr

fpath = '/home/joncrall/.cache/kwcoco/demo/large_hyperspectral/big_img_128.bsq' lazy_rio = LazyRasterIOFrameFile(fpath) ds = lazy_rio._ds

classmethod available()

Returns True if this backend is available

property ndim

property shape

property dtype

class kwcoco.util.lazy_frame_backends.**LazyGDalFrameFile**(*fpath*, *nodata_method=None*, *overview=None*)

Bases: NiceRepr

Todo:

- [ ] Move to its own backend module
- **[ ] When used with COCO, allow the image metadata to populate the** height, width, and channels if possible.

Example

```
>>> # xdoctest: +REQUIRES(module:osgeo)
>>> self = LazyGDalFrameFile.demo()
>>> print('self = {!r}'.format(self))
>>> self[0:3, 0:3]
>>> self[:, :, 0]
>>> self[0]
>>> self[0, 3]
```

```
>>> # import kwplot
>>> # kwplot.imshow(self[:])
```

Parameters

- **fpath** (*str*) – the path to the file to load
- **nodata_method** (*None* | *int* | *str*) – how to handle nodata
- **overview** (*int*) – The overview level to load (zero is no overview)

Example

```
>>> # See if we can reproduce the INTERLEAVE bug
```

data = np.random.rand(128, 128, 64) import kwimage import ubelt as ub from os.path import join dpath = ub.ensure_app_cache_dir('kwcoco/tests/reader') fpath = join(dpath, 'foo.tiff') kwimage.imwrite(fpath, data, backend='skimage') recon1 = kwimage.imread(fpath) recon1.shape

self = LazyGDalFrameFile(fpath) self.shape self[:]

classmethod available()

Returns True if this backend is available

get_overview(*overview*)

Returns the overview relative to this one.

get_absolute_overview(*overview*)

Returns the overview relative to the base

classmethod demo(*key='astro'*, *dsize=None*)

property ndim

property num_overviews

property num_absolute_overviews

property shape

property dtype

2.1.1.6.2.4 kwcoco.util.util_archive module

class kwcoco.util.util_archive.**Archive**(*fpath=None*, *mode='r'*, *backend=None*, *file=None*)

Bases: object

Abstraction over zipfile and tarfile

Todo: see if we can use one of these other tools instead

SeeAlso:

https://github.com/RKrahl/archive-tools https://pypi.org/project/arlib/

Example

```
>>> from kwcoco.util.util_archive import Archive
>>> import ubelt as ub
>>> dpath = ub.Path.appdir('kwcoco', 'tests', 'util', 'archive')
>>> dpath.delete().ensuredir()
>>> # Test write mode
>>> mode = 'w'
>>> arc_zip = Archive(str(dpath / 'demo.zip'), mode=mode)
>>> arc_tar = Archive(str(dpath / 'demo.tar.gz'), mode=mode)
>>> open(dpath / 'data_1only.txt', 'w').write('bazbzzz')
>>> open(dpath / 'data_2only.txt', 'w').write('buzzz')
>>> open(dpath / 'data_both.txt', 'w').write('foobar')
>>> #
>>> arc_zip.add(dpath / 'data_both.txt')
>>> arc_zip.add(dpath / 'data_1only.txt')
>>> #
>>> arc_tar.add(dpath / 'data_both.txt')
>>> arc_tar.add(dpath / 'data_2only.txt')
>>> #
>>> arc_zip.close()
>>> arc_tar.close()
>>> #
>>> # Test read mode
```

(continues on next page)

(continued from previous page)

```
>>> arc_zip = Archive(str(dpath / 'demo.zip'), mode='r')
>>> arc_tar = Archive(str(dpath / 'demo.tar.gz'), mode='r')
>>> # Test names
>>> name = 'data_both.txt'
>>> assert name in arc_zip.names()
>>> assert name in arc_tar.names()
>>> # Test read
>>> assert arc_zip.read(name, mode='r') == 'foobar'
>>> assert arc_tar.read(name, mode='r') == 'foobar'
>>> #
>>> # Test extractall
>>> extract_dpath = ub.ensuredir(str(dpath / 'extracted'))
>>> extracted1 = arc_zip.extractall(extract_dpath)
>>> extracted2 = arc_tar.extractall(extract_dpath)
>>> for fpath in extracted2:
>>>     print(open(fpath, 'r').read())
>>> for fpath in extracted1:
>>>     print(open(fpath, 'r').read())
```

names()

read(*name*, *mode='rb'*)

Read data directly out of the archive.

Parameters

- **name** (*str*) – the name of the archive member to read
- **mode** (*str*) – This is a conceptual parameter that emulates the usual open mode. Defaults to “rb”, which returns data as raw bytes. If “r” will decode the bytes into utf8-text.

classmethod coerce(*data*)

Either open an archive file path or coerce an existing ZipFile or tarfile structure into this wrapper class

add(*fpath*, *arcname=None*)

close()

extractall(*output_dpath='.'*, *verbose=1*, *overwrite=True*)

kwcoco.util.util_archive.**unarchive_file**(*archive_fpath*, *output_dpath='.'*, *verbose=1*, *overwrite=True*)

2.1.1.6.2.5 kwcoco.util.util_delayed_poc module

THIS WILL BE DEPRECATED in favor of delayed_ops

This module is ported from ndsampler, and will likely eventually move to kwimage and be refactored using pymbolic

The classes in this file represent a tree of delayed operations.

Proof of concept for delayed chainable transforms in Python.

There are several optimizations that could be applied.

This is similar to GDAL’s virtual raster table, but it works in memory and I think it is easier to chain operations.

SeeAlso:

../../dev/symbolic_delayed.py

Warning: As the name implies this is a proof of concept, and the actual implementation was hacked together too quickly. Serious refactoring will be necessary.

Concepts:

Each class should be a layer that adds a new transformation on top of underlying nested layers. Adding new layers should be quick, and there should always be the option to "finalize" a stack of layers, chaining the transforms / operations and then applying one final efficient transform at the end.

Todo:

- **[x] Need to handle masks / nodata values when warping. Might need to**
  rely more on gdal / rasterio for this.

Conventions:

- dsize = (always in width / height), no channels are present
- shape for images is always (height, width, channels)
- channels are always the last dimension of each image, if no channel dim is specified, finalize will add it.
- **Videos must be the last process in the stack, and add a leading**
  time dimension to the shape. dsize is still width, height, but shape is now: (time, height, width, chan)

Example

```
>>> # Example demonstrating the modivating use case
>>> # We have multiple aligned frames for a video, but each of
>>> # those frames is in a different resolution. Furthermore,
>>> # each of the frames consists of channels in different resolutions.
>>> # Create raw channels in some "native" resolution for frame 1
>>> from kwcoco.util.util_delayed_poc import *  # NOQA
>>> f1_chan1 = DelayedIdentity.demo('astro', chan=0, dsize=(300, 300))
>>> f1_chan2 = DelayedIdentity.demo('astro', chan=1, dsize=(200, 200))
>>> f1_chan3 = DelayedIdentity.demo('astro', chan=2, dsize=(10, 10))
>>> # Create raw channels in some "native" resolution for frame 2
>>> f2_chan1 = DelayedIdentity.demo('carl', dsize=(64, 64), chan=0)
>>> f2_chan2 = DelayedIdentity.demo('carl', dsize=(260, 260), chan=1)
>>> f2_chan3 = DelayedIdentity.demo('carl', dsize=(10, 10), chan=2)
>>> #
>>> # Delayed warp each channel into its "image" space
>>> # Note: the images never actually enter this space we transform through it
>>> f1_dsize = np.array((3, 3))
>>> f2_dsize = np.array((2, 2))
>>> f1_img = DelayedChannelConcat([
>>>     f1_chan1.delayed_warp(kwimage.Affine.scale(f1_dsize / f1_chan1.dsize), dsize=f1_
↪dsize),
>>>     f1_chan2.delayed_warp(kwimage.Affine.scale(f1_dsize / f1_chan2.dsize), dsize=f1_
↪dsize),
>>>     f1_chan3.delayed_warp(kwimage.Affine.scale(f1_dsize / f1_chan3.dsize), dsize=f1_
↪dsize),
>>> ])
```

(continues on next page)

(continued from previous page)

```
>>> f2_img = DelayedChannelConcat([
>>>     f2_chan1.delayed_warp(kwimage.Affine.scale(f2_dsize / f2_chan1.dsize), dsize=f2_
↪dsize),
>>>     f2_chan2.delayed_warp(kwimage.Affine.scale(f2_dsize / f2_chan2.dsize), dsize=f2_
↪dsize),
>>>     f2_chan3.delayed_warp(kwimage.Affine.scale(f2_dsize / f2_chan3.dsize), dsize=f2_
↪dsize),
>>> ])
>>> # Combine frames into a video
>>> vid_dsize = np.array((280, 280))
>>> vid = DelayedFrameConcat([
>>>     f1_img.delayed_warp(kwimage.Affine.scale(vid_dsize / f1_img.dsize), dsize=vid_
↪dsize),
>>>     f2_img.delayed_warp(kwimage.Affine.scale(vid_dsize / f2_img.dsize), dsize=vid_
↪dsize),
>>> ])
>>> vid.nesting
>>> print('vid.nesting = {}'.format(ub.repr2(vid.__json__(), nl=-2)))
>>> final = vid.finalize(interpolation='nearest')
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(final[0], pnum=(1, 2, 1), fnum=1)
>>> kwplot.imshow(final[1], pnum=(1, 2, 2), fnum=1)
```

Example

```
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('vidshapes8-multispectral')
>>> delayed = dset.coco_image(1).delay(mode=0)
>>> from kwcoco.util.util_delayed_poc import *  # NOQA
>>> astro = DelayedLoad.demo('astro')
>>> print('MSI = ' + ub.repr2(delayed.__json__(), nl=-3, sort=0))
>>> print('ASTRO = ' + ub.repr2(astro.__json__(), nl=2, sort=0))
```

```
>>> subchan = delayed.take_channels('B1|B8')
>>> subcrop = subchan.delayed_crop((slice(10, 80), slice(30, 50)))
>>> #
>>> subcrop.nesting()
>>> subchan.nesting()
>>> subchan.finalize()
>>> subcrop.finalize()
>>> #
>>> msi_crop = delayed.delayed_crop((slice(10, 80), slice(30, 50)))
>>> msi_warp = msi_crop.delayed_warp(kwimage.Affine.scale(3), dsize='auto')
>>> subdata = msi_warp.take_channels('B11|B1')
>>> final = subdata.finalize()
```

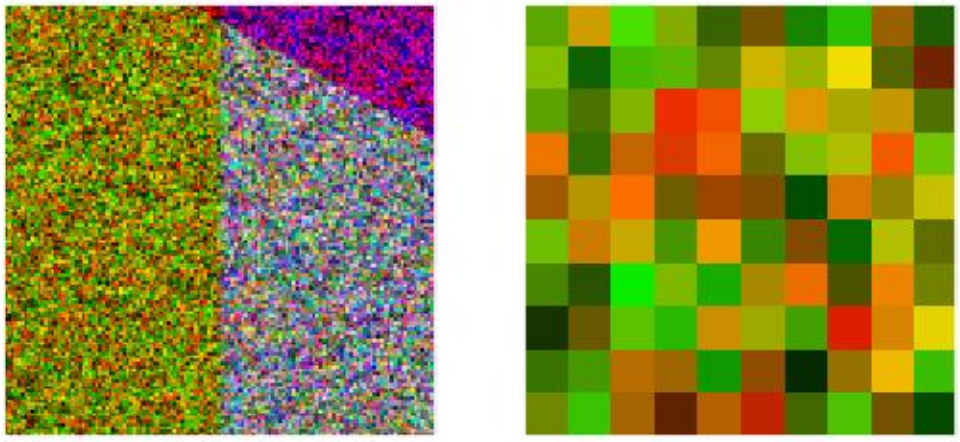
(continues on next page)

(continued from previous page)

```
>>> assert final.shape == (210, 60, 2)
```

Example

```
>>> # test case where an auxiliary image does not map entirely on the image.
>>> from kwcoco.util.util_delayed_poc import *  # NOQA
>>> import kwimage
>>> from os.path import join
>>> dpath = ub.ensure_app_cache_dir('kwcoco/tests/delayed_ops')
>>> chan1_fpath = join(dpath, 'chan1.tiff')
>>> chan2_fpath = join(dpath, 'chan2.tiff')
>>> chan3_fpath = join(dpath, 'chan2.tiff')
>>> chan1_raw = np.random.rand(128, 128, 1)
>>> chan2_raw = np.random.rand(64, 64, 1)
>>> chan3_raw = np.random.rand(256, 256, 1)
>>> kwimage.imwrite(chan1_fpath, chan1_raw)
>>> kwimage.imwrite(chan2_fpath, chan2_raw)
>>> kwimage.imwrite(chan3_fpath, chan3_raw)
>>> #
>>> c1 = channel_spec.FusedChannelSpec.coerce('c1')
>>> c2 = channel_spec.FusedChannelSpec.coerce('c2')
>>> c3 = channel_spec.FusedChannelSpec.coerce('c2')
>>> aux1 = DelayedLoad(chan1_fpath, dsize=chan1_raw.shape[0:2][::-1], channels=c1, num_
↪bands=1)
>>> aux2 = DelayedLoad(chan2_fpath, dsize=chan2_raw.shape[0:2][::-1], channels=c2, num_
↪bands=1)
>>> aux3 = DelayedLoad(chan3_fpath, dsize=chan3_raw.shape[0:2][::-1], channels=c3, num_
↪bands=1)
>>> #
>>> img_dsize = (128, 128)
>>> transform1 = kwimage.Affine.coerce(scale=0.5)
>>> transform2 = kwimage.Affine.coerce(theta=0.5, shearx=0.01, offset=(-20, -40))
>>> transform3 = kwimage.Affine.coerce(offset=(64, 0)) @ kwimage.Affine.random(rng=10)
>>> part1 = aux1.delayed_warp(np.eye(3), dsize=img_dsize)
>>> part2 = aux2.delayed_warp(transform2, dsize=img_dsize)
>>> part3 = aux3.delayed_warp(transform3, dsize=img_dsize)
>>> delayed = DelayedChannelConcat([part1, part2, part3])
>>> #
>>> delayed_crop = delayed.crop((slice(0, 10), slice(0, 10)))
>>> delayed_final = delayed_crop.finalize()
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> final = delayed.finalize()
>>> kwplot.imshow(final, fnum=1, pnum=(1, 2, 1))
>>> kwplot.imshow(delayed_final, fnum=1, pnum=(1, 2, 2))
```



comp = delayed_crop.components[2]

comp.sub_data.finalize()

data = np.array([[0]]).astype(np.float32) kwimage.warp_affine(data, np.eye(3), dsize=(32, 32)) kwimage.warp_affine(data, np.eye(3))

kwimage.warp_affine(data[0:0], np.eye(3))

transform = kwimage.Affine.coerce(scale=0.1) data = np.array([[0]]).astype(np.float32)

data = np.array([[]]).astype(np.float32) kwimage.warp_affine(data, transform, dsize=(0, 2), antialias=True)

data = np.array([[]]).astype(np.float32) kwimage.warp_affine(data, transform, dsize=(10, 10))

data = np.array([[0]]).astype(np.float32) kwimage.warp_affine(data, transform, dsize=(0, 2), antialias=True)

data = np.array([[0]]).astype(np.float32) kwimage.warp_affine(data, transform, dsize=(10, 10))

cv2.warpAffine(
kwimage.grab_test_image(dsize=(1, 1)), kwimage.Affine.coerce(scale=0.1).matrix[0:2], dsize=(0, 1),

)

class kwcoco.util.util_delayed_poc.**DelayedVisionOperation**

Bases: NiceRepr

Base class for nodes in a tree of delayed computer-vision operations

finalize(***kwargs*)

children()

Abstract method, which should generate all of the direct children of a node in the operation tree.

Yields
Any

nesting()

warp(**args*, ***kwargs*)

alias for delayed_warp, might change to this API in the future

crop(**args*, ***kwargs*)

alias for delayed_crop, might change to this API in the future

class kwcoco.util.util_delayed_poc.**DelayedVideoOperation**

Bases: *DelayedVisionOperation*

class kwcoco.util.util_delayed_poc.**DelayedImageOperation**

Bases: *DelayedVisionOperation*

Operations that pertain only to images

delayed_crop(*region_slices*)

Create a new delayed image that performs a crop in the transformed "self" space.

Parameters
region_slices (*Tuple[slice, slice]*) – y-slice and x-slice.

Note: Returns a heuristically "simplified" tree. In the current implementation there are only 3 operations, cat, warp, and crop. All cats go at the top, all crops go at the bottom, all warps are in the middle.

Returns
lazy executed delayed transform

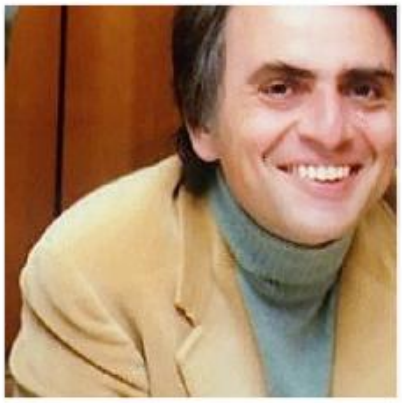
Return type
DelayedImageOperation

Example

```
>>> dsize = (100, 100)
>>> tf2 = kwimage.Affine.affine(scale=3).matrix
>>> self = DelayedWarp(np.random.rand(33, 33), tf2, dsize)
>>> region_slices = (slice(5, 10), slice(1, 12))
>>> delayed_crop = self.delayed_crop(region_slices)
>>> print(ub.repr2(delayed_crop.nesting(), nl=-1, sort=0))
>>> delayed_crop.finalize()
```

Example

```
>>> chan1 = DelayedLoad.demo('astro')
>>> chan2 = DelayedLoad.demo('carl')
>>> warped1a = chan1.delayed_warp(kwimage.Affine.scale(1.2).matrix)
>>> warped2a = chan2.delayed_warp(kwimage.Affine.scale(1.5))
>>> warped1b = warped1a.delayed_warp(kwimage.Affine.scale(1.2).matrix)
>>> warped2b = warped2a.delayed_warp(kwimage.Affine.scale(1.5))
>>> #
>>> region_slices = (slice(97, 677), slice(5, 691))
>>> self = warped2b
>>> #
>>> crop1 = warped1b.delayed_crop(region_slices)
>>> crop2 = warped2b.delayed_crop(region_slices)
>>> print(ub.repr2(warped1b.nesting(), nl=-1, sort=0))
>>> print(ub.repr2(warped2b.nesting(), nl=-1, sort=0))
>>> # Notice how the crop merges the two nesting layers
>>> # (via the hueristic optimize step)
>>> print(ub.repr2(crop1.nesting(), nl=-1, sort=0))
>>> print(ub.repr2(crop2.nesting(), nl=-1, sort=0))
>>> frame1 = crop1.finalize(dsize=(500, 500))
>>> frame2 = crop2.finalize(dsize=(500, 500))
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(frame1, pnum=(1, 2, 1), fnum=1)
>>> kwplot.imshow(frame2, pnum=(1, 2, 2), fnum=1)
```

delayed_warp(*transform*, *dsize=None*)

Delayed transform the underlying data.

Note: this deviates from kwimage warp functions because instead of "output_dims" (specified in c-style shape) we specify dsize (w, h).

Returns
new delayed transform a chained transform

Return type
DelayedImageOperation

take_channels(*channels*)

Returns
DelayedVisionOperation

class kwcoco.util.util_delayed_poc.**DelayedIdentity**(*sub_data*, *dsize=None*, *channels=None*, *quantization=None*)

Bases: *DelayedImageOperation*

Noop leaf that does nothing. Can be used to hold raw data.

Typically used to just hold raw data.

DelayedIdentity.demo('astro', chan=0, dsize=(32, 32))

Example

```
>>> from kwcoco.util.util_delayed_poc import *  # NOQA
>>> sub_data = np.random.rand(31, 37, 3)
>>> self = DelayedIdentity(sub_data)
>>> self = DelayedIdentity(sub_data, channels='L|a|b')
```

```
>>> # test with quantization
>>> rng = kwarray.ensure_rng(32)
>>> sub_data_quant = (rng.rand(31, 37, 3) * 1000).astype(np.int16)
>>> sub_data_quant[0, 0] = -9999
>>> self = DelayedIdentity(sub_data_quant, channels='L|a|b', quantization={
>>>     'orig_min': 0.,
>>>     'orig_max': 1.,
>>>     'quant_min': 0,
>>>     'quant_max': 1000,
>>>     'nodata': -9999,
>>> })
>>> final1 = self.finalize(dequantize=True)
>>> final2 = self.finalize(dequantize=False)
>>> assert np.all(np.isnan(final1[0, 0]))
>>> scale = final2 / final1
>>> scales = scale[scale > 0]
>>> assert np.all(np.isclose(scales, 1000))
>>> # check that take channels works
>>> new_subdata = self.take_channels('a')
>>> sub_final1 = new_subdata.finalize(dequantize=True)
>>> sub_final2 = new_subdata.finalize(dequantize=False)
>>> assert sub_final1.dtype.kind == 'f'
>>> assert sub_final2.dtype.kind == 'i'
```

classmethod demo(*key='astro'*, *chan=None*, *dsize=None*)

children()

Yields
Any

finalize(***kwargs*)

take_channels(*channels*)

Returns
DelayedIdentity

kwcoco.util.util_delayed_poc.**dequantize**(*quant_data*, *quantization*)

Helper for dequantization

class kwcoco.util.util_delayed_poc.**DelayedNans**(*dsize=None*, *channels=None*)

Bases: *DelayedImageOperation*

Constructs nan channels as needed

Example

self = DelayedNans((10, 10), channel_spec.FusedChannelSpec.coerce('rgb')) region_slices = (slice(5, 10), slice(1, 12)) delayed = self.delayed_crop(region_slices)

Example

```
>>> from kwcoco.util.util_delayed_poc import *  # NOQA
>>> dsize = (307, 311)
>>> c1 = DelayedNans(dsize=dsize, channels=channel_spec.FusedChannelSpec.coerce('foo
↪'))
>>> c2 = DelayedLoad.demo('astro', dsize=dsize).load_shape(True)
>>> cat = DelayedChannelConcat([c1, c2])
>>> warped_cat = cat.delayed_warp(kwimage.Affine.scale(1.07), dsize=(328, 332))
>>> warped_cat.finalize()
```

#>>> cropped = warped_cat.delayed_crop((slice(0, 300), slice(0, 100))) #>>> cropped.finalize().shape

property shape

property num_bands

property dsize

property channels

children()

Yields
Any

finalize(***kwargs*)

delayed_crop(*region_slices*)

Return type
DelayedNans

delayed_warp(*transform*, *dsize=None*)

Return type
DelayedNans

class kwcoco.util.util_delayed_poc.**DelayedLoad**(*fpath*, *channels=None*, *dsize=None*, *num_bands=None*, *immediate_crop=None*, *immediate_chan_idxs=None*, *immediate_dsize=None*, *quantization=None*)

Bases: *DelayedImageOperation*

A load operation for a specific sub-region and sub-bands in a specified image.

Note: This class contains support for fusing certain lazy operations into this layer, namely cropping, scaling, and channel selection.

For now these are named `immediates`

Example

```
>>> fpath = kwimage.grab_test_image_fpath()
>>> self = DelayedLoad(fpath)
>>> print('self = {!r}'.format(self))
>>> self.load_shape()
>>> print('self = {!r}'.format(self))
>>> self.finalize()
```

```
>>> f1_img = DelayedLoad.demo('astro', dsize=(300, 300))
>>> f2_img = DelayedLoad.demo('carl', dsize=(256, 320))
>>> print('f1_img = {!r}'.format(f1_img))
>>> print('f2_img = {!r}'.format(f2_img))
>>> print(f2_img.finalize().shape)
>>> print(f1_img.finalize().shape)
```

```
>>> fpath = kwimage.grab_test_image_fpath()
>>> channels = channel_spec.FusedChannelSpec.coerce('rgb')
>>> self = DelayedLoad(fpath, channels=channels)
```

Example

```
>>> # Test with quantization
>>> fpath = kwimage.grab_test_image_fpath()
>>> channels = channel_spec.FusedChannelSpec.coerce('rgb')
>>> self = DelayedLoad(fpath, channels=channels, quantization={
>>>     'orig_min': 0.,
>>>     'orig_max': 1.,
>>>     'quant_min': 0,
>>>     'quant_max': 256,
>>>     'nodata': None,
>>> })
>>> final1 = self.finalize(dequantize=False)
>>> final2 = self.finalize(dequantize=True)
>>> assert final1.dtype.kind == 'u'
>>> assert final2.dtype.kind == 'f'
>>> assert final2.max() <= 1
```

classmethod demo(*key='astro'*, *dsize=None*)

classmethod coerce(*data*)

children()

Yields
Any

nesting()

load_shape(*use_channel_heuristic=False*)

property shape

property num_bands

property dsize

property channels

property fpath

finalize(***kwargs*)

Todo:

- [ ] Load from overviews if a scale will be necessary

Parameters

****kwargs** –

nodata_method
[if specified this data item is treated as nodata, the] data is then converted to floats and the nodata value is replaced with nan.

delayed_crop(*region_slices*)

Parameters
region_slices (*Tuple[slice, slice]*) – y-slice and x-slice.

Returns
a new delayed load object with a fused crop operation

Return type
DelayedLoad

Example

```
>>> # Test chained crop operations
>>> from kwcoco.util.util_delayed_poc import *  # NOQA
>>> self = orig = DelayedLoad.demo('astro').load_shape()
>>> region_slices = slices1 = (slice(0, 90), slice(30, 60))
>>> self = crop1 = orig.delayed_crop(slices1)
>>> region_slices = slices2 = (slice(10, 21), slice(10, 22))
>>> self = crop2 = crop1.delayed_crop(slices2)
>>> region_slices = slices3 = (slice(3, 20), slice(5, 20))
>>> crop3 = crop2.delayed_crop(slices3)
>>> # Spot check internals
>>> print('orig = {}'.format(ub.repr2(orig.__json__(), nl=2)))
>>> print('crop1 = {}'.format(ub.repr2(crop1.__json__(), nl=2)))
>>> print('crop2 = {}'.format(ub.repr2(crop2.__json__(), nl=2)))
>>> print('crop3 = {}'.format(ub.repr2(crop3.__json__(), nl=2)))
>>> # Test internals
>>> assert crop3._immediates['crop'][0].start == 13
>>> assert crop3._immediates['crop'][0].stop == 21
>>> # Test shapes work out correctly
>>> assert crop3.finalize().shape == (8, 7, 3)
>>> assert crop2.finalize().shape == (11, 12, 3)
```

(continues on next page)

(continued from previous page)

```
>>> assert crop1.take_channels([1, 2]).finalize().shape == (90, 30, 2)
>>> assert orig.finalize().shape == (512, 512, 3)
```

Note:

```
This chart gives an intuition on how new absolute slice coords
are computed from existing absolute coords ane relative coords.

     5 7    <- new
     3 5    <- rel
   --------
   01234567  <- relative coordinates
   --------
   2      9  <- curr
 ----------
 0123456789  <- absolute coordinates
 ----------
```

`take_channels`(*channels*)

This method returns a subset of the vision data with only the specified bands / channels.

Parameters

channels (*List[int] | slice | channel_spec.FusedChannelSpec*) – List of integers indexes, a slice, or a channel spec, which is typically a pipe (|) delimited list of channel codes. See kwcoco.ChannelSpec for more detials.

Returns

a new delayed load with a fused take channel operation

Return type

DelayedLoad

Note: The channel subset must exist here or it will raise an error. A better implementation (via pymbolic) might be able to do better

Example

```
>>> from kwcoco.util.util_delayed_poc import *  # NOQA
>>> import kwcoco
>>> self = DelayedLoad.demo('astro').load_shape()
>>> channels = [2, 0]
>>> new = self.take_channels(channels)
>>> new3 = new.take_channels([1, 0])
```

```
>>> final1 = self.finalize()
>>> final2 = new.finalize()
>>> final3 = new3.finalize()
>>> assert np.all(final1[..., 2] == final2[..., 0])
```

(continues on next page)

(continued from previous page)

```
>>> assert np.all(final1[..., 0] == final2[..., 1])
>>> assert final2.shape[2] == 2
```

```
>>> assert np.all(final1[..., 2] == final3[..., 1])
>>> assert np.all(final1[..., 0] == final3[..., 0])
>>> assert final3.shape[2] == 2
```

class kwcoco.util.util_delayed_poc.**DelayedFrameConcat**(*frames*, *dsize=None*)

Bases: *DelayedVideoOperation*

Represents multiple frames in a video

Note:

```
Video[0]:
    Frame[0]:
        Chan[0]: (32) +--------------------------------+
        Chan[1]: (16) +----------------+
        Chan[2]: ( 8) +--------+
    Frame[1]:
        Chan[0]: (30) +------------------------------+
        Chan[1]: (14) +--------------+
        Chan[2]: ( 6) +------+
```

Todo:

- [ ] Support computing the transforms when none of the data is loaded

---

Example

```
>>> # Simpler case with fewer nesting levels
>>> rng = kwarray.ensure_rng(None)
>>> # Delayed warp each channel into its "image" space
>>> # Note: the images never enter the space we transform through
>>> f1_img = DelayedLoad.demo('astro', (300, 300))
>>> f2_img = DelayedLoad.demo('carl', (256, 256))
>>> # Combine frames into a video
>>> vid_dsize = np.array((100, 100))
>>> self = vid = DelayedFrameConcat([
>>>     f1_img.delayed_warp(kwimage.Affine.scale(vid_dsize / f1_img.dsize)),
>>>     f2_img.delayed_warp(kwimage.Affine.scale(vid_dsize / f2_img.dsize)),
>>> ], dsize=vid_dsize)
>>> print(ub.repr2(vid.nesting(), nl=-1, sort=0))
>>> final = vid.finalize(interpolation='nearest', dsize=(32, 32))
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(final[0], pnum=(1, 2, 1), fnum=1)
```

(continues on next page)

(continued from previous page)

```
>>> kwplot.imshow(final[1], pnum=(1, 2, 2), fnum=1)
>>> region_slices = (slice(0, 90), slice(30, 60))
```

children()

Yields
Any

property channels

property shape

finalize(***kwargs*)
Execute the final transform

delayed_crop(*region_slices*)

Returns
DelayedFrameConcat

Example

```
>>> from kwcoco.util.util_delayed_poc import *  # NOQA
>>> # Create raw channels in some "native" resolution for frame 1
>>> f1_chan1 = DelayedIdentity.demo('astro', chan=(1, 0), dsize=(300, 300))
>>> f1_chan2 = DelayedIdentity.demo('astro', chan=2, dsize=(10, 10))
>>> # Create raw channels in some "native" resolution for frame 2
>>> f2_chan1 = DelayedIdentity.demo('carl', dsize=(64, 64), chan=(1, 0))
>>> f2_chan2 = DelayedIdentity.demo('carl', dsize=(10, 10), chan=2)
>>> #
>>> f1_dsize = np.array(f1_chan1.dsize)
>>> f2_dsize = np.array(f2_chan1.dsize)
>>> f1_img = DelayedChannelConcat([
>>>     f1_chan1.delayed_warp(kwimage.Affine.scale(f1_dsize / f1_chan1.dsize),␣
↪dsize=f1_dsize),
>>>     f1_chan2.delayed_warp(kwimage.Affine.scale(f1_dsize / f1_chan2.dsize),␣
↪dsize=f1_dsize),
>>> ])
>>> f2_img = DelayedChannelConcat([
>>>     f2_chan1.delayed_warp(kwimage.Affine.scale(f2_dsize / f2_chan1.dsize),␣
↪dsize=f2_dsize),
>>>     f2_chan2.delayed_warp(kwimage.Affine.scale(f2_dsize / f2_chan2.dsize),␣
↪dsize=f2_dsize),
>>> ])
>>> vid_dsize = np.array((280, 280))
>>> full_vid = DelayedFrameConcat([
>>>     f1_img.delayed_warp(kwimage.Affine.scale(vid_dsize / f1_img.dsize),␣
↪dsize=vid_dsize),
>>>     f2_img.delayed_warp(kwimage.Affine.scale(vid_dsize / f2_img.dsize),␣
↪dsize=vid_dsize),
>>> ])
>>> region_slices = (slice(80, 200), slice(80, 200))
>>> print(ub.repr2(full_vid.nesting(), nl=-1, sort=0))
>>> crop_vid = full_vid.delayed_crop(region_slices)
>>> final_full = full_vid.finalize(interpolation='nearest')
>>> final_crop = crop_vid.finalize(interpolation='nearest')
>>> import pytest
>>> with pytest.raises(ValueError):
>>>     # should not be able to crop a crop yet
>>>     crop_vid.delayed_crop(region_slices)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(final_full[0], pnum=(2, 2, 1), fnum=1)
>>> kwplot.imshow(final_full[1], pnum=(2, 2, 2), fnum=1)
>>> kwplot.imshow(final_crop[0], pnum=(2, 2, 3), fnum=1)
>>> kwplot.imshow(final_crop[1], pnum=(2, 2, 4), fnum=1)
```

delayed_warp(*transform*, *dsize=None*)

Delayed transform the underlying data.

Note: this deviates from kwimage warp functions because instead of "output_dims" (specified in c-style shape) we specify dsize (w, h).

Returns
new delayed transform a chained transform

Return type
DelayedWarp

class kwcoco.util.util_delayed_poc.**JaggedArray**(*parts*, *axis*)

Bases: NiceRepr

The result of an unaligned concatenate

property shape

class kwcoco.util.util_delayed_poc.**DelayedChannelConcat**(*components*, *dsize=None*, *jagged=False*)

Bases: *DelayedImageOperation*

Represents multiple channels in an image that could be concatenated

Variables
components (*List[DelayedWarp]*) – a list of stackable channels. Each component may be comprised of multiple channels.

Todo:

- [ ] can this be generalized into a delayed concat and combined with DelayedFrameConcat?
- [ ] can all concats be delayed until the very end?

Example

```
>>> from kwcoco.util.util_delayed_poc import *  # NOQA
>>> # Create 3 delayed operations to concatenate
>>> comp1 = DelayedWarp(np.random.rand(11, 7))
>>> comp2 = DelayedWarp(np.random.rand(11, 7, 3))
>>> comp3 = DelayedWarp(
>>>     np.random.rand(3, 5, 2),
>>>     transform=kwimage.Affine.affine(scale=(7/5, 11/3)).matrix,
>>>     dsize=(7, 11)
>>> )
>>> components = [comp1, comp2, comp3]
>>> chans = DelayedChannelConcat(components)
>>> final = chans.finalize()
>>> assert final.shape == chans.shape
>>> assert final.shape == (11, 7, 6)
```

```
>>> # We should be able to nest DelayedChannelConcat inside virutal images
>>> frame1 = DelayedWarp(
>>>     chans, transform=kwimage.Affine.affine(scale=2.2).matrix,
>>>     dsize=(20, 26))
>>> frame2 = DelayedWarp(
>>>     np.random.rand(3, 3, 6), dsize=(20, 26))
>>> frame3 = DelayedWarp(
>>>     np.random.rand(3, 3, 6), dsize=(20, 26))
```

```
>>> print(ub.repr2(frame1.nesting(), nl=-1, sort=False))
>>> frame1.finalize()
>>> vid = DelayedFrameConcat([frame1, frame2, frame3])
>>> print(ub.repr2(vid.nesting(), nl=-1, sort=False))
```

Example

```
>>> from kwcoco.util.util_delayed_poc import *  # NOQA
>>> # If requested, we can return arrays of the different sizes.
>>> # but usually this will raise an error.
>>> comp1 = DelayedWarp.random(dsize=(32, 32), nesting=(1, 5), channels=1)
>>> comp2 = DelayedWarp.random(dsize=(8, 8), nesting=(1, 5), channels=1)
>>> comp3 = DelayedWarp.random(dsize=(64, 64), nesting=(1, 5), channels=1)
>>> components = [comp1, comp2, comp3]
>>> self = DelayedChannelConcat(components, jagged=True)
>>> final = self.finalize()
>>> print('final = {!r}'.format(final))
```

children()

Yields
Any

classmethod random(*num_parts=3*, *rng=None*)

Example

```
>>> self = DelayedChannelConcat.random()
>>> print('self = {!r}'.format(self))
>>> print(ub.repr2(self.nesting(), nl=-1, sort=0))
```

property channels

property shape

finalize(***kwargs*)

Execute the final transform

delayed_warp(*transform*, *dsize=None*)

Delayed transform the underlying data.

Note: this deviates from kwimage warp functions because instead of "output_dims" (specified in c-style shape) we specify dsize (w, h).

Returns
new delayed transform a chained transform

Return type
DelayedWarp

take_channels(*channels*)

This method returns a subset of the vision data with only the specified bands / channels.

Parameters
channels (*List[int] | slice | channel_spec.FusedChannelSpec*) – List of integers indexes, a slice, or a channel spec, which is typically a pipe (|) delimited list of channel codes. See kwcoco.ChannelSpec for more detials.

Returns: # DelayedVisionOperation: # a delayed vision operation that only operates on the following # channels.

Example

```
>>> from kwcoco.util.util_delayed_poc import *  # NOQA
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('vidshapes8-multispectral')
>>> self = delayed = dset.coco_image(1).delay(mode=0)
>>> channels = 'B11|B8|B1|B10'
>>> new = self.take_channels(channels)
```

Example

```
>>> # Complex case
>>> import kwcoco
>>> from kwcoco.util.util_delayed_poc import *  # NOQA
>>> dset = kwcoco.CocoDataset.demo('vidshapes8-multispectral')
>>> delayed = dset.coco_image(1).delay(mode=0)
>>> astro = DelayedLoad.demo('astro').load_shape(use_channel_heuristic=True)
>>> aligned = astro.warp(kwimage.Affine.scale(600 / 512), dsize='auto')
>>> self = combo = DelayedChannelConcat(delayed.components + [aligned])
>>> channels = 'B1|r|B8|g'
>>> new = self.take_channels(channels)
>>> new_cropped = new.crop((slice(10, 200), slice(12, 350)))
>>> datas = new_cropped.finalize()
>>> vizable = kwimage.normalize_intensity(datas, axis=2)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> stacked = kwimage.stack_images(vizable.transpose(2, 0, 1))
>>> kwplot.imshow(stacked)
```

CommandLine

```
xdoctest -m /home/joncrall/code/kwcoco/kwcoco/util/util_delayed_poc.py␣
↪DelayedChannelConcat.take_channels:2 --profile
```

Example

```
>>> # Test case where requested channel does not exist
>>> import kwcoco
>>> from kwcoco.util.util_delayed_poc import *  # NOQA
>>> dset = kwcoco.CocoDataset.demo('vidshapes8-multispectral', use_cache=1,␣
↪verbose=100)
>>> self = dset.coco_image(1).delay(mode=0)
>>> channels = 'B1|foobar|bazbiz|B8'
>>> new = self.take_channels(channels)
>>> new_cropped = new.crop((slice(10, 200), slice(12, 350)))
>>> fused = new_cropped.finalize()
>>> assert fused.shape == (190, 338, 4)
>>> assert np.all(np.isnan(fused[..., 1:3]))
>>> assert not np.any(np.isnan(fused[..., 0]))
>>> assert not np.any(np.isnan(fused[..., 3]))
```

class kwcoco.util.util_delayed_poc.**DelayedWarp**(*sub_data*, *transform=None*, *dsize=None*)

Bases: *DelayedImageOperation*

POC for chainable transforms

Note: "sub" is used to refer to the underlying data in its native coordinates and resolution.

"self" is used to refer to the data in the transformed coordinates that are exposed by this class.

Variables

- **sub_data** (DelayedWarp | *ArrayLike*) – array-like image data at a native resolution
- **transform** (*kwimage.Transform*) – transforms data from native "sub"-image-space to "self"-image-space.

Example

```
>>> dsize = (12, 12)
>>> tf1 = np.array([[2, 0, 0], [0, 2, 0], [0, 0, 1]])
>>> tf2 = np.array([[3, 0, 0], [0, 3, 0], [0, 0, 1]])
>>> tf3 = np.array([[4, 0, 0], [0, 4, 0], [0, 0, 1]])
>>> band1 = DelayedWarp(np.random.rand(6, 6), tf1, dsize)
>>> band2 = DelayedWarp(np.random.rand(4, 4), tf2, dsize)
>>> band3 = DelayedWarp(np.random.rand(3, 3), tf3, dsize)
>>> #
>>> # Execute a crop in a one-level transformed space
>>> region_slices = (slice(5, 10), slice(0, 12))
>>> delayed_crop = band2.delayed_crop(region_slices)
```

(continues on next page)

(continued from previous page)

```
>>> final_crop = delayed_crop.finalize()
>>> #
>>> # Execute a crop in a nested transformed space
>>> tf4 = np.array([[1.5, 0, 0], [0, 1.5, 0], [0, 0, 1]])
>>> chained = DelayedWarp(band2, tf4, (18, 18))
>>> delayed_crop = chained.delayed_crop(region_slices)
>>> final_crop = delayed_crop.finalize()
>>> #
>>> tf4 = np.array([[.5, 0, 0], [0, .5, 0], [0, 0, 1]])
>>> chained = DelayedWarp(band2, tf4, (6, 6))
>>> delayed_crop = chained.delayed_crop(region_slices)
>>> final_crop = delayed_crop.finalize()
>>> #
>>> region_slices = (slice(1, 5), slice(2, 4))
>>> delayed_crop = chained.delayed_crop(region_slices)
>>> final_crop = delayed_crop.finalize()
```

Example

```
>>> dsize = (17, 12)
>>> tf = np.array([[5.2, 0, 1.1], [0, 3.1, 2.2], [0, 0, 1]])
>>> self = DelayedWarp(np.random.rand(3, 5, 13), tf, dsize=dsize)
>>> self.finalize().shape
```

classmethod random(*dsize=None*, *raw_width=(8, 64)*, *raw_height=(8, 64)*, *channels=(1, 5)*, *nesting=(2, 5)*, *rng=None*)

Create a random delayed warp operation for testing / demo

Parameters

- **dsize** (*Tuple[int, int] | None*) – The width and height of the finalized data. If unspecified, it will be a function of the random warps.
- **raw_width** (*int | Tuple[int, int]*) – The exact or min / max width of the random raw data
- **raw_height** (*int | Tuple[int, int]*) – The exact or min / max height of the random raw data
- **nesting** (*Tuple[int, int]*) – The exact or min / max random depth of warp nestings
- **channels** (*int | Tuple[int, int]*) – The exact or min / max number of random channels.

Returns

DelayedWarp

Example

```
>>> self = DelayedWarp.random(nesting=(4, 7))
>>> print('self = {!r}'.format(self))
>>> print(ub.repr2(self.nesting(), nl=-1, sort=0))
```

property channels

children()

Yields
Any

property dsize

property num_bands

property shape

finalize(*transform=None*, *dsize=None*, *interpolation='linear'*, ***kwargs*)

Execute the final transform

Can pass a parent transform to augment this underlying transform.

Parameters

- **transform** (*kwimage.Transform*) – an additional transform to perform
- **dsize** (*Tuple[int, int]*) – overrides destination canvas size

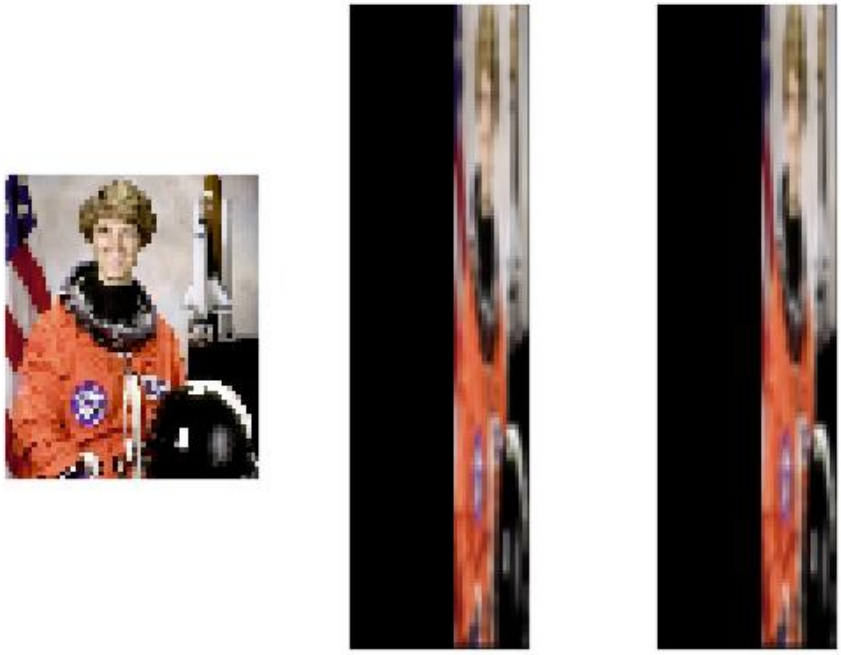
Example

```
>>> tf = np.array([[0.9, 0, 3.9], [0, 1.1, -.5], [0, 0, 1]])
>>> raw = kwimage.grab_test_image(dsize=(54, 65))
>>> raw = kwimage.ensure_float01(raw)
>>> # Test nested finalize
>>> layer1 = raw
>>> num = 10
>>> for _ in range(num):
...     layer1  = DelayedWarp(layer1, tf, dsize='auto')
>>> final1 = layer1.finalize()
>>> # Test non-nested finalize
>>> layer2 = list(layer1._optimize_paths())[0]
>>> final2 = layer2.finalize()
>>> #
>>> print(ub.repr2(layer1.nesting(), nl=-1, sort=0))
>>> print(ub.repr2(layer2.nesting(), nl=-1, sort=0))
>>> print('final1 = {!r}'.format(final1))
>>> print('final2 = {!r}'.format(final2))
>>> print('final1.shape = {!r}'.format(final1.shape))
>>> print('final2.shape = {!r}'.format(final2.shape))
>>> assert np.allclose(final1, final2)
>>> #
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
```

(continues on next page)

(continued from previous page)

```
>>> kwplot.autompl()
>>> kwplot.imshow(raw, pnum=(1, 3, 1), fnum=1)
>>> kwplot.imshow(final1, pnum=(1, 3, 2), fnum=1)
>>> kwplot.imshow(final2, pnum=(1, 3, 3), fnum=1)
>>> kwplot.show_if_requested()
```



Example

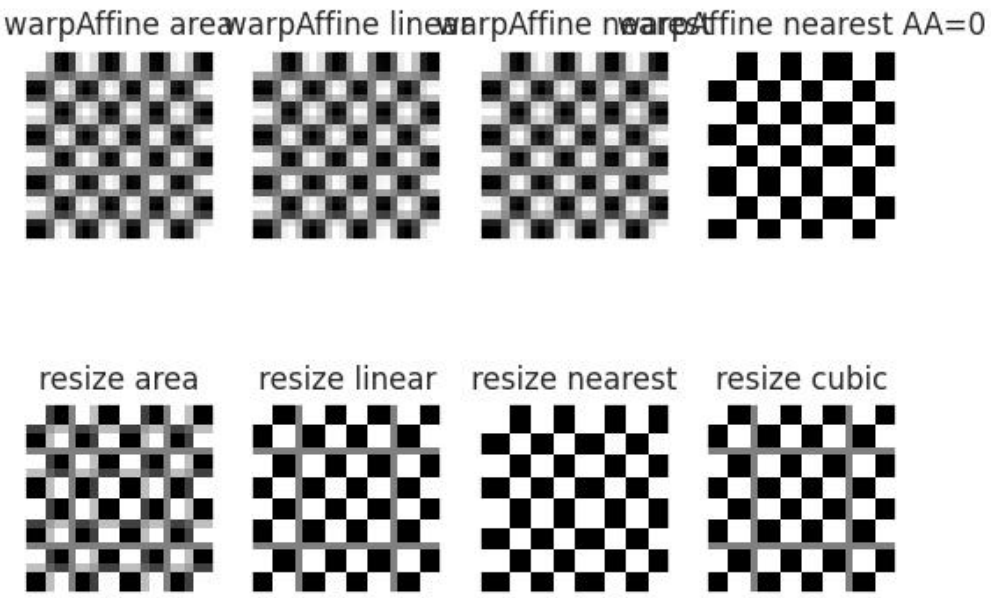
```
>>> # Test aliasing
>>> s = DelayedIdentity.demo()
>>> s = DelayedIdentity.demo('checkerboard')
>>> a = s.delayed_warp(kwimage.Affine.scale(0.05), dsize='auto')
>>> b = s.delayed_warp(kwimage.Affine.scale(3), dsize='auto')
```

```
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> # It looks like downsampling linear and area is the same
>>> # Does warpAffine have no alias handling?
>>> pnum_ = kwplot.PlotNums(nRows=2, nCols=4)
>>> kwplot.imshow(a.finalize(interpolation='area'), pnum=pnum_(), title=
↪'warpAffine area')
>>> kwplot.imshow(a.finalize(interpolation='linear'), pnum=pnum_(), title=
```

(continues on next page)

(continued from previous page)

```
↪'warpAffine linear')
>>> kwplot.imshow(a.finalize(interpolation='nearest'), pnum=pnum_(), title=
↪'warpAffine nearest')
>>> kwplot.imshow(a.finalize(interpolation='nearest', antialias=False),␣
↪pnum=pnum_(), title='warpAffine nearest AA=0')
>>> kwplot.imshow(kwimage.imresize(s.finalize(), dsize=a.dsize, interpolation=
↪'area'), pnum=pnum_(), title='resize area')
>>> kwplot.imshow(kwimage.imresize(s.finalize(), dsize=a.dsize, interpolation=
↪'linear'), pnum=pnum_(), title='resize linear')
>>> kwplot.imshow(kwimage.imresize(s.finalize(), dsize=a.dsize, interpolation=
↪'nearest'), pnum=pnum_(), title='resize nearest')
>>> kwplot.imshow(kwimage.imresize(s.finalize(), dsize=a.dsize, interpolation=
↪'cubic'), pnum=pnum_(), title='resize cubic')
```



Example

```
>>> # xdoctest: +REQUIRES(--ipfs)
>>> # Demo code to develop support for overviews
>>> import kwimage
>>> from kwcoco.util.util_delayed_poc import *  # NOQA
>>> fpath = ub.grabdata('https://ipfs.io/ipfs/
↪QmaFcb565HM9FV8f41jrfCZcu1CXsZZMXEosjmbgeBhFQr', fname='PXL_20210411_
↪150641385.jpg')
```

(continues on next page)

(continued from previous page)

```
>>> data0 = kwimage.imread(fpath, overview=0, backend='gdal')
>>> data1 = kwimage.imread(fpath, overview=1, backend='gdal')
>>> delayed_load = DelayedLoad(fpath=fpath)
>>> delayed_load._ensure_dsize()
>>> self = delayed_load.warp(kwimage.Affine.affine(scale=0.1), dsize='auto')
>>> transform = kwimage.Affine.affine(scale=2.0)
>>> imdata = self.finalize(transform=transform)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(imdata)
```

take_channels(*channels*)

class kwcoco.util.util_delayed_poc.**DelayedCrop**(*sub_data*, *sub_slices*)

Bases: *DelayedImageOperation*

Represent a delayed crop operation

Example

```
>>> sub_data = DelayedLoad.demo()
>>> sub_slices = (slice(5, 10), slice(1, 12))
>>> self = DelayedCrop(sub_data, sub_slices)
>>> print(ub.repr2(self.nesting(), nl=-1, sort=0))
>>> final = self.finalize()
>>> print('final.shape = {!r}'.format(final.shape))
```

Example

```
>>> sub_data = DelayedLoad.demo()
>>> sub_slices = (slice(5, 10), slice(1, 12))
>>> crop1 = DelayedCrop(sub_data, sub_slices)
>>> import pytest
>>> # Should only error while huristics are in use.
>>> with pytest.raises(ValueError):
>>>     crop2 = DelayedCrop(crop1, sub_slices)
```

property channels

children()

Yields
Any

finalize(***kwargs*)

2.1.1.6.2.6 kwcoco.util.util_futures module

Deprecated and functionality moved to ubelt

class kwcoco.util.util_futures.**Executor**(*mode='thread'*, *max_workers=0*)

Bases: object

Wrapper around a specific executor.

Abstracts Serial, Thread, and Process Executor via arguments.

Parameters

- **mode** (*str, default='thread'*) – either thread, serial, or process
- **max_workers** (*int, default=0*) – number of workers. If 0, serial is forced.

Example

```
>>> import platform
>>> import sys
>>> # The process backend breaks pyp3 when using coverage
>>> if 'pypy' in platform.python_implementation().lower():
...     import pytest
...     pytest.skip('not testing process on pypy')
>>> if sys.platform.startswith('win32'):
...     import pytest
...     pytest.skip('not running this test on win32 for now')
>>> import ubelt as ub
>>> # Fork before threading!
>>> # https://pybay.com/site_media/slides/raymond2017-keynote/combo.html
>>> self1 = ub.Executor(mode='serial', max_workers=0)
>>> self1.__enter__()
>>> self2 = ub.Executor(mode='process', max_workers=2)
>>> self2.__enter__()
>>> self3 = ub.Executor(mode='thread', max_workers=2)
>>> self3.__enter__()
>>> jobs = []
>>> jobs.append(self1.submit(sum, [1, 2, 3]))
>>> jobs.append(self1.submit(sum, [1, 2, 3]))
>>> jobs.append(self2.submit(sum, [10, 20, 30]))
>>> jobs.append(self2.submit(sum, [10, 20, 30]))
>>> jobs.append(self3.submit(sum, [4, 5, 5]))
>>> jobs.append(self3.submit(sum, [4, 5, 5]))
>>> for job in jobs:
>>>     result = job.result()
>>>     print('result = {!r}'.format(result))
>>> self1.__exit__(None, None, None)
>>> self2.__exit__(None, None, None)
>>> self3.__exit__(None, None, None)
```

Example

```
>>> import ubelt as ub
>>> self1 = ub.Executor(mode='serial', max_workers=0)
>>> with self1:
>>>     jobs = []
>>>     for i in range(10):
>>>         jobs.append(self1.submit(sum, [i + 1, i]))
>>>     for job in jobs:
>>>         job.add_done_callback(lambda x: print('done callback got x = {}'.
↪format(x)))
>>>         result = job.result()
>>>         print('result = {!r}'.format(result))
```

submit(*func*, **args*, ***kw*)

Calls the submit function of the underlying backend.

Returns
a future representing the job

Return type
concurrent.futures.Future

shutdown()

Calls the shutdown function of the underlying backend.

map(*fn*, **iterables*, ***kwargs*)

Calls the map function of the underlying backend.

CommandLine

```
xdoctest -m ubelt.util_futures Executor.map
```

Example

```
>>> import ubelt as ub
>>> import concurrent.futures
>>> import string
>>> with ub.Executor(mode='serial') as executor:
...     result_iter = executor.map(int, string.digits)
...     results = list(result_iter)
>>> print('results = {!r}'.format(results))
results = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> with ub.Executor(mode='thread', max_workers=2) as executor:
...     result_iter = executor.map(int, string.digits)
...     results = list(result_iter)
>>> # xdoctest: +IGNORE_WANT
>>> print('results = {!r}'.format(results))
results = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

class kwcoco.util.util_futures.**JobPool**(*mode='thread'*, *max_workers=0*)

Bases: object

Abstracts away boilerplate of submitting and collecting jobs

This is a basic wrapper around *ubelt.util_futures.Executor* that simplifies the most basic case.

Example

```
>>> import ubelt as ub
>>> def worker(data):
>>>     return data + 1
>>> pool = ub.JobPool('thread', max_workers=16)
>>> for data in ub.ProgIter(range(10), desc='submit jobs'):
>>>     pool.submit(worker, data)
>>> final = []
>>> for job in pool.as_completed(desc='collect jobs'):
>>>     info = job.result()
>>>     final.append(info)
>>> print('final = {!r}'.format(final))
```

submit(*func*, **args*, ***kwargs*)

Submit a job managed by the pool

Parameters

- **func** (*Callable[..., Any]*) – A callable that will take as many arguments as there are passed iterables.
- ***args** – positional arguments to pass to the function
- ****kwargs** – keyword arguments to pass to the function

Returns
a future representing the job

Return type
concurrent.futures.Future

shutdown()

as_completed(*timeout=None*, *desc=None*, *progkw=None*)

Generates completed jobs in an arbitrary order

Parameters

- **timeout** (*float | None*) – Specify the the maximum number of seconds to wait for a job.
- **desc** (*str | None*) – if specified, reports progress with a ubelt.progiter.ProgIter object.
- **progkw** (*dict | None*) – extra keyword arguments to ubelt.progiter.ProgIter.

Yields
concurrent.futures.Future – The completed future object containing the results of a job.

CommandLine

```
xdoctest -m ubelt.util_futures JobPool.as_completed
```

Example

```
>>> import ubelt as ub
>>> pool = ub.JobPool('thread', max_workers=8)
>>> text = ub.paragraph(
...     '''
...     UDP is a cool protocol, check out the wiki:
...
...     UDP-based Data Transfer Protocol (UDT), is a high-performance
...     data transfer protocol designed for transferring large
...     volumetric datasets over high-speed wide area networks. Such
...     settings are typically disadvantageous for the more common TCP
...     protocol.
...     ''')
>>> for word in text.split(' '):
...     pool.submit(print, word)
>>> for _ in pool.as_completed():
...     pass
>>> pool.shutdown()
```

join(***kwargs*)

Like *JobPool.as_completed()*, but executes the *result* method of each future and returns only after all processes are complete. This allows for lower-boilerplate prototyping.

Parameters

****kwargs** – passed to *JobPool.as_completed()*

Returns

list of results

Return type

List[Any]

Example

```
>>> import ubelt as ub
>>> # We just want to try replacing our simple iterative algorithm
>>> # with the embarassingly parallel version
>>> arglist = list(zip(range(1000), range(1000)))
>>> func = ub.identity
>>> #
>>> # Original version
>>> for args in arglist:
>>>     func(*args)
>>> #
>>> # Potentially parallel version
>>> jobs = ub.JobPool(max_workers=0)
>>> for args in arglist:
```

(continues on next page)

(continued from previous page)

```
>>>     jobs.submit(func, *args)
>>> _ = jobs.join(desc='running')
```

2.1.1.6.2.7 kwcoco.util.util_json module

kwcoco.util.util_json.**ensure_json_serializable**(*dict_*, *normalize_containers=False*, *verbose=0*)

Attempt to convert common types (e.g. numpy) into something json complient

Convert numpy and tuples into lists

Parameters

normalize_containers (*bool*) – if True, normalizes dict containers to be standard python structures. Defaults to False.

Example

```
>>> data = ub.ddict(lambda: int)
>>> data['foo'] = ub.ddict(lambda: int)
>>> data['bar'] = np.array([1, 2, 3])
>>> data['foo']['a'] = 1
>>> data['foo']['b'] = (1, np.array([1, 2, 3]), {3: np.int32(3), 4: np.float16(1.0)}
↪)
>>> dict_ = data
>>> print(ub.repr2(data, nl=-1))
>>> assert list(find_json_unserializable(data))
>>> result = ensure_json_serializable(data, normalize_containers=True)
>>> print(ub.repr2(result, nl=-1))
>>> assert not list(find_json_unserializable(result))
>>> assert type(result) is dict
```

kwcoco.util.util_json.**find_json_unserializable**(*data*, *quickcheck=False*)

Recurse through json datastructure and find any component that causes a serialization error. Record the location of these errors in the datastructure as we recurse through the call tree.

Parameters

- **data** (*object*) – data that should be json serializable
- **quickcheck** (*bool*) – if True, check the entire datastructure assuming its ok before doing the python-based recursive logic.

Returns

list of “bad part” dictionaries containing items

’value’ - the value that caused the serialization error

’loc’ - which contains a list of key/indexes that can be used to lookup the location of the unserializable value. If the “loc” is a list, then it indicates a rare case where a key in a dictionary is causing the serialization error.

Return type

List[Dict]

Example

```
>>> from kwcoco.util.util_json import *  # NOQA
>>> part = ub.ddict(lambda: int)
>>> part['foo'] = ub.ddict(lambda: int)
>>> part['bar'] = np.array([1, 2, 3])
>>> part['foo']['a'] = 1
>>> # Create a dictionary with two unserializable parts
>>> data = [1, 2, {'nest1': [2, part]}, {frozenset({'badkey'}): 3, 2: 4}]
>>> parts = list(find_json_unserializable(data))
>>> print('parts = {}'.format(ub.repr2(parts, nl=1)))
>>> # Check expected structure of bad parts
>>> assert len(parts) == 2
>>> part = parts[1]
>>> assert list(part['loc']) == [2, 'nest1', 1, 'bar']
>>> # We can use the "loc" to find the bad value
>>> for part in parts:
>>>     # "loc" is a list of directions containing which keys/indexes
>>>     # to traverse at each descent into the data structure.
>>>     directions = part['loc']
>>>     curr = data
>>>     special_flag = False
>>>     for key in directions:
>>>         if isinstance(key, list):
>>>             # special case for bad keys
>>>             special_flag = True
>>>             break
>>>         else:
>>>             # normal case for bad values
>>>             curr = curr[key]
>>>     if special_flag:
>>>         assert part['data'] in curr.keys()
>>>         assert part['data'] is key[1]
>>>     else:
>>>         assert part['data'] is curr
```

kwcoco.util.util_json.**indexable_allclose**(*dct1*, *dct2*, *return_info=False*)

Walks through two nested data structures and ensures that everything is roughly the same.

Parameters

- **dct1** – a nested indexable item
- **dct2** – a nested indexable item

Example

```
>>> from kwcoco.util.util_json import indexable_allclose
>>> dct1 = {
>>>     'foo': [1.222222, 1.333],
>>>     'bar': 1,
>>>     'baz': [],
>>> }
>>> dct2 = {
>>>     'foo': [1.22222, 1.333],
>>>     'bar': 1,
>>>     'baz': [],
>>> }
>>> assert indexable_allclose(dct1, dct2)
```

2.1.1.6.2.8 kwcoco.util.util_monkey module

class kwcoco.util.util_monkey.**SupressPrint**(**mods*, ***kw*)

Bases: object

Temporarily replace the print function in a module with a noop

Parameters

- ***mods** – the modules to disable print in
- ****kw** – only accepts "enabled" enabled (bool, default=True): enables or disables this context

class kwcoco.util.util_monkey.**Reloadable**

Bases: type

This is a metaclass that overrides the behavior of isinstance and issubclass when invoked on classes derived from this such that they only check that the module and class names agree, which are preserved through module reloads, whereas class instances are not.

This is useful for interactive develoment, but should be removed in production.

Example

```
>>> from kwcoco.util.util_monkey import *  # NOQA
>>> # Illustrate what happens with a reload when using this utility
>>> # versus without it.
>>> class Base1:
>>>     ...
>>> class Derived1(Base1):
>>>     ...
>>> @Reloadable.add_metaclass
>>> class Base2:
>>>     ...
>>> class Derived2(Base2):
>>>     ...
>>> inst1 = Derived1()
>>> inst2 = Derived2()
```

(continues on next page)

(continued from previous page)

```
>>> assert isinstance(inst1, Derived1)
>>> assert isinstance(inst2, Derived2)
>>> # Simulate reload
>>> class Base1:
>>>     ...
>>> class Derived1(Base1):
>>>     ...
>>> @Reloadable.add_metaclass
>>> class Base2:
>>>     ...
>>> class Derived2(Base2):
>>>     ...
>>> assert not isinstance(inst1, Derived1)
>>> assert isinstance(inst2, Derived2)
```

classmethod add_metaclass(*cls*)

Class decorator for creating a class with this as a metaclass

classmethod developing(*cls*)

Like add_metaclass, but warns the user that they are developing. This helps remind them to remove this in production

2.1.1.6.2.9 kwcoco.util.util_reroot module

Rerooting is harder than you would think

kwcoco.util.util_reroot.**special_reroot_single**(*dset*, *verbose=0*)

kwcoco.util.util_reroot.**resolve_relative_to**(*path*, *dpath*, *strict=False*)

Given a path, try to resolve its symlinks such that it is relative to the given dpath.

Example

```
>>> from kwcoco.util.util_reroot import *  # NOQA
>>> import os
>>> def _symlink(self, target, verbose=0):
>>>     return ub.Path(ub.symlink(target, self, verbose=verbose))
>>> ub.Path._symlink = _symlink
>>> #
>>> # TODO: try to enumerate all basic cases
>>> #
>>> base = ub.Path.appdir('kwcoco/tests/reroot')
>>> base.delete().ensuredir()
>>> #
>>> drive1 = (base / 'drive1').ensuredir()
>>> drive2 = (base / 'drive2').ensuredir()
>>> #
>>> data_repo1 = (drive1 / 'data_repo1').ensuredir()
>>> cache = (data_repo1 / '.cache').ensuredir()
>>> real_file1 = (cache / 'real_file1').touch()
```

(continues on next page)

(continued from previous page)

```
>>> #
>>> real_bundle = (data_repo1 / 'real_bundle').ensuredir()
>>> real_assets = (real_bundle / 'assets').ensuredir()
>>> #
>>> # Symlink file outside of the bundle
>>> link_file1 = (real_assets / 'link_file1')._symlink(real_file1)
>>> real_file2 = (real_assets / 'real_file2').touch()
>>> link_file2 = (real_assets / 'link_file2')._symlink(real_file2)
>>> #
>>> #
>>> # A symlink to the data repo
>>> data_repo2 = (drive1 / 'data_repo2')._symlink(data_repo1)
>>> data_repo3 = (drive2 / 'data_repo3')._symlink(data_repo1)
>>> data_repo4 = (drive2 / 'data_repo4')._symlink(data_repo2)
>>> #
>>> # A prediction repo TODO
>>> pred_repo5 = (drive2 / 'pred_repo5').ensuredir()
>>> #
>>> # _ = ub.cmd(f'tree -a {base}', verbose=3)
>>> #
>>> fpaths = []
>>> for r, ds, fs in os.walk(base, followlinks=True):
>>>     for f in fs:
>>>         if 'file' in f:
>>>             fpath = ub.Path(r) / f
>>>             fpaths.append(fpath)
>>> #
>>> #
>>> dpath = real_bundle.resolve()
>>> #
>>> for path in fpaths:
>>>     # print(f'{path}')
>>>     # print(f'{path.resolve()=}')
>>>     resolved_rel = resolve_relative_to(path, dpath)
>>>     print('resolved_rel = {!r}'.format(resolved_rel))
```

kwcoco.util.util_reroot.**resolve_directory_symlinks**(*path*)

Only resolve symlinks of directories, not the base file

2.1.1.6.2.10 kwcoco.util.util_sklearn module

Extensions to sklearn constructs

class kwcoco.util.util_sklearn.**StratifiedGroupKFold**(*n_splits=3*, *shuffle=False*, *random_state=None*)

Bases: _BaseKFold

Stratified K-Folds cross-validator with Grouping

Provides train/test indices to split data in train/test sets.

This cross-validation object is a variation of GroupKFold that returns stratified folds. The folds are made by preserving the percentage of samples for each class.

This is an old interface and should likely be refactored and modernized.

Parameters
n_splits (*int, default=3*) – Number of folds. Must be at least 2.

`split`(*X*, *y*, *groups=None*)

Generate indices to split data into training and test set.

2.1.1.6.2.11 kwcoco.util.util_truncate module

Truncate utility based on python-slugify.

https://pypi.org/project/python-slugify/1.2.2/

kwcoco.util.util_truncate.**smart_truncate**(*string*, *max_length=0*, *separator=' '*, *trunc_loc=0.5*)

Truncate a string. :param string (str): string for modification :param max_length (int): output string length :param word_boundary (bool): :param save_order (bool): if True then word order of output string is like input string :param separator (str): separator between words :param trunc_loc (float): fraction of location where to remove the text :return:

2.1.1.6.3 Module contents

mkinit ~/code/kwcoco/kwcoco/util/__init__.py -w mkinit ~/code/kwcoco/kwcoco/util/__init__.py –lazy

kwcoco.util.**ALLOF**(**TYPES*)

kwcoco.util.**ANYOF**(**TYPES*)

kwcoco.util.**ARRAY**(*TYPE={}*, ***kw*)

https://json-schema.org/understanding-json-schema/reference/array.html

Example

```
>>> from kwcoco.util.jsonschema_elements import *  # NOQA
>>> ARRAY(numItems=3)
>>> schema = ARRAY(minItems=3)
>>> schema.validate()
{'type': 'array', 'items': {}, 'minItems': 3}
```

class kwcoco.util.**Archive**(*fpath=None*, *mode='r'*, *backend=None*, *file=None*)

Bases: `object`

Abstraction over zipfile and tarfile

Todo: see if we can use one of these other tools instead

SeeAlso:
https://github.com/RKrahl/archive-tools https://pypi.org/project/arlib/

Example

```
>>> from kwcoco.util.util_archive import Archive
>>> import ubelt as ub
>>> dpath = ub.Path.appdir('kwcoco', 'tests', 'util', 'archive')
>>> dpath.delete().ensuredir()
>>> # Test write mode
>>> mode = 'w'
>>> arc_zip = Archive(str(dpath / 'demo.zip'), mode=mode)
>>> arc_tar = Archive(str(dpath / 'demo.tar.gz'), mode=mode)
>>> open(dpath / 'data_1only.txt', 'w').write('bazbzzz')
>>> open(dpath / 'data_2only.txt', 'w').write('buzzz')
>>> open(dpath / 'data_both.txt', 'w').write('foobar')
>>> #
>>> arc_zip.add(dpath / 'data_both.txt')
>>> arc_zip.add(dpath / 'data_1only.txt')
>>> #
>>> arc_tar.add(dpath / 'data_both.txt')
>>> arc_tar.add(dpath / 'data_2only.txt')
>>> #
>>> arc_zip.close()
>>> arc_tar.close()
>>> #
>>> # Test read mode
>>> arc_zip = Archive(str(dpath / 'demo.zip'), mode='r')
>>> arc_tar = Archive(str(dpath / 'demo.tar.gz'), mode='r')
>>> # Test names
>>> name = 'data_both.txt'
>>> assert name in arc_zip.names()
>>> assert name in arc_tar.names()
>>> # Test read
>>> assert arc_zip.read(name, mode='r') == 'foobar'
>>> assert arc_tar.read(name, mode='r') == 'foobar'
>>> #
>>> # Test extractall
>>> extract_dpath = ub.ensuredir(str(dpath / 'extracted'))
>>> extracted1 = arc_zip.extractall(extract_dpath)
>>> extracted2 = arc_tar.extractall(extract_dpath)
>>> for fpath in extracted2:
>>>     print(open(fpath, 'r').read())
>>> for fpath in extracted1:
>>>     print(open(fpath, 'r').read())
```

names()

read(*name*, *mode='rb'*)

Read data directly out of the archive.

Parameters

- **name** (*str*) – the name of the archive member to read
- **mode** (*str*) – This is a conceptual parameter that emulates the usual open mode. Defaults to “rb”, which returns data as raw bytes. If “r” will decode the bytes into utf8-text.

classmethod coerce(*data*)

Either open an archive file path or coerce an existing ZipFile or tarfile structure into this wrapper class

add(*fpath*, *arcname=None*)

close()

extractall(*output_dpath='.'*, *verbose=1*, *overwrite=True*)

class kwcoco.util.**ContainerElements**

Bases: object

Types that contain other types

Example

```
>>> from kwcoco.util.jsonschema_elements import *  # NOQA
>>> print(elem.ARRAY().validate())
>>> print(elem.OBJECT().validate())
>>> print(elem.OBJECT().validate())
{'type': 'array', 'items': {}}
{'type': 'object', 'properties': {}}
{'type': 'object', 'properties': {}}
```

ARRAY(*TYPE={}*, ***kw*)

https://json-schema.org/understanding-json-schema/reference/array.html

Example

```
>>> from kwcoco.util.jsonschema_elements import *  # NOQA
>>> ARRAY(numItems=3)
>>> schema = ARRAY(minItems=3)
>>> schema.validate()
{'type': 'array', 'items': {}, 'minItems': 3}
```

OBJECT(*PROPERTIES={}*, ***kw*)

https://json-schema.org/understanding-json-schema/reference/object.html

Example

```
>>> import jsonschema
>>> schema = elem.OBJECT()
>>> jsonschema.validate({}, schema)
>>> #
>>> import jsonschema
>>> schema = elem.OBJECT({
>>>     'key1': elem.ANY(),
>>>     'key2': elem.ANY(),
>>> }, required=['key1'])
>>> jsonschema.validate({'key1': None}, schema)
>>> #
```

(continues on next page)

(continued from previous page)

```
>>> import jsonschema
>>> schema = elem.OBJECT({
>>>     'key1': elem.OBJECT({'arr': elem.ARRAY()}),
>>>     'key2': elem.ANY(),
>>> }, required=['key1'], title='a title')
>>> schema.validate()
>>> print('schema = {}'.format(ub.repr2(schema, sort=1, nl=-1)))
>>> jsonschema.validate({'key1': {'arr': []}}, schema)
schema = {
    'properties': {
        'key1': {
            'properties': {
                'arr': {'items': {}, 'type': 'array'}
            },
            'type': 'object'
        },
        'key2': {}
    },
    'required': ['key1'],
    'title': 'a title',
    'type': 'object'
}
```

class kwcoco.util.**DictLike**

Bases: NiceRepr

An inherited class must specify the `getitem`, `setitem`, and `keys` methods.

A class is dictionary like if it has:

`__iter__`, `__len__`, `__contains__`, `__getitem__`, `items`, `keys`, `values`, `get`,

and if it should be writable it should have: `__delitem__`, `__setitem__`, `update`,

And perhaps: `copy`,

`__iter__`, `__len__`, `__contains__`, `__getitem__`, `items`, `keys`, `values`, `get`,

and if it should be writable it should have: `__delitem__`, `__setitem__`, `update`,

And perhaps: `copy`,

getitem(*key*)

Parameters
key (*Any*) – a key

Returns
a value

Return type
Any

setitem(*key*, *value*)

Parameters

- **key** (*Any*)

- **value** (*Any*)

`delitem`(*key*)

Parameters
key (*Any*)

`keys`()

Yields
Any – a key

`items`()

Yields
Tuple[Any, Any] – a key value pair

`values`()

Yields
Any – a value

`copy`()

Return type
Dict

`to_dict`()

Return type
Dict

`asdict`()

Return type
Dict

`update`(*other*)

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

Parameters

- **key** (*Any*)
- **default** (*Any*)

Return type
Any

`class kwcoco.util.Element`(*base*, *options={}*, *_magic=None*)

Bases: `dict`

A dictionary used to define an element of a JSON Schema.

The exact keys/values for the element will depend on the type of element being described. The *SchemaElements* defines exactly what these are for the core elements. (e.g. OBJECT, INTEGER, NULL, ARRAY, ANYOF)

Example

```
>>> from kwcoco.coco_schema import *  # NOQA
>>> self = Element(base={'type': 'demo'}, options={'opt1', 'opt2'})
>>> new = self(opt1=3)
>>> print('self = {}'.format(ub.repr2(self, nl=1, sort=1)))
>>> print('new = {}'.format(ub.repr2(new, nl=1, sort=1)))
>>> print('new2 = {}'.format(ub.repr2(new(), nl=1, sort=1)))
>>> print('new3 = {}'.format(ub.repr2(new(title='myvar'), nl=1, sort=1)))
>>> print('new4 = {}'.format(ub.repr2(new(title='myvar')(examples=['']), nl=1,␣
↪sort=1)))
>>> print('new5 = {}'.format(ub.repr2(new(badattr=True), nl=1, sort=1)))
self = {
    'type': 'demo',
}
new = {
    'opt1': 3,
    'type': 'demo',
}
new2 = {
    'opt1': 3,
    'type': 'demo',
}
new3 = {
    'opt1': 3,
    'title': 'myvar',
    'type': 'demo',
}
new4 = {
    'examples': [''],
    'opt1': 3,
    'title': 'myvar',
    'type': 'demo',
}
new5 = {
    'opt1': 3,
    'type': 'demo',
}
```

validate(*instance=NoParam*)

If `instance` is given, validates that that dictionary conforms to this schema. Otherwise validates that this is a valid schema element.

Parameters

instance (*dict*) – a dictionary to validate

class kwcoco.util.**IndexableWalker**(*data*, *dict_cls=(<class 'dict'>,)*, *list_cls=(<class 'list'>, <class 'tuple'>)*)

Bases: Generator

Traverses through a nested tree-liked indexable structure.

Generates a path and value to each node in the structure. The path is a list of indexes which if applied in order will reach the value.

The `__setitem__` method can be used to modify a nested value based on the path returned by the generator.

When generating values, you can use "send" to prevent traversal of a particular branch.

RelatedWork:

- **https://pypi.org/project/python-benedict/ - implements a dictionary** subclass with similar nested indexing abilities.

Example

```
>>> # Given Nested Data
>>> data = {
>>>     'foo': {'bar': 1},
>>>     'baz': [{'biz': 3}, {'buz': [4, 5, 6]}],
>>> }
>>> # Create an IndexableWalker
>>> walker = IndexableWalker(data)
>>> # We iterate over the data as if it was flat
>>> # ignore the <want> string due to order issues on older Pythons
>>> # xdoctest: +IGNORE_WANT
>>> for path, val in walker:
>>>     print(path)
['foo']
['baz']
['baz', 0]
['baz', 1]
['baz', 1, 'buz']
['baz', 1, 'buz', 0]
['baz', 1, 'buz', 1]
['baz', 1, 'buz', 2]
['baz', 0, 'biz']
['foo', 'bar']
>>> # We can use "paths" as keys to getitem into the walker
>>> path = ['baz', 1, 'buz', 2]
>>> val = walker[path]
>>> assert val == 6
>>> # We can use "paths" as keys to setitem into the walker
>>> assert data['baz'][1]['buz'][2] == 6
>>> walker[path] = 7
>>> assert data['baz'][1]['buz'][2] == 7
>>> # We can use "paths" as keys to delitem into the walker
>>> assert data['baz'][1]['buz'][1] == 5
>>> del walker[['baz', 1, 'buz', 1]]
>>> assert data['baz'][1]['buz'][1] == 7
```

Example

```
>>> # Create nested data
>>> # xdoctest: +REQUIRES(module:numpy)
>>> import numpy as np
>>> import ubelt as ub
>>> data = ub.ddict(lambda: int)
>>> data['foo'] = ub.ddict(lambda: int)
>>> data['bar'] = np.array([1, 2, 3])
>>> data['foo']['a'] = 1
>>> data['foo']['b'] = np.array([1, 2, 3])
>>> data['foo']['c'] = [1, 2, 3]
>>> data['baz'] = 3
>>> print('data = {}'.format(ub.repr2(data, nl=True)))
>>> # We can walk through every node in the nested tree
>>> walker = IndexableWalker(data)
>>> for path, value in walker:
>>>     print('walk path = {}'.format(ub.repr2(path, nl=0)))
>>>     if path[-1] == 'c':
>>>         # Use send to prevent traversing this branch
>>>         got = walker.send(False)
>>>         # We can modify the value based on the returned path
>>>         walker[path] = 'changed the value of c'
>>> print('data = {}'.format(ub.repr2(data, nl=True)))
>>> assert data['foo']['c'] == 'changed the value of c'
```

Example

```
>>> # Test sending false for every data item
>>> # xdoctest: +REQUIRES(CPython)
>>> # xdoctest: +REQUIRES(module:numpy)
>>> import ubelt as ub
>>> import numpy as np
>>> data = {1: 1}
>>> walker = IndexableWalker(data)
>>> for path, value in walker:
>>>     print('walk path = {}'.format(ub.repr2(path, nl=0)))
>>>     walker.send(False)
>>> data = {}
>>> walker = IndexableWalker(data)
>>> for path, value in walker:
>>>     walker.send(False)
```

send(*arg*) → send 'arg' into generator,
return next yielded value or raise StopIteration.

throw(*typ*[, *val*[, *tb*]]) → raise exception in generator,
return next yielded value or raise StopIteration.

kwcoco.util.**NOT**(*TYPE*)

kwcoco.util.**OBJECT**(*PROPERTIES={}*, ***kw*)

https://json-schema.org/understanding-json-schema/reference/object.html

Example

```
>>> import jsonschema
>>> schema = elem.OBJECT()
>>> jsonschema.validate({}, schema)
>>> #
>>> import jsonschema
>>> schema = elem.OBJECT({
>>>     'key1': elem.ANY(),
>>>     'key2': elem.ANY(),
>>> }, required=['key1'])
>>> jsonschema.validate({'key1': None}, schema)
>>> #
>>> import jsonschema
>>> schema = elem.OBJECT({
>>>     'key1': elem.OBJECT({'arr': elem.ARRAY()}),
>>>     'key2': elem.ANY(),
>>> }, required=['key1'], title='a title')
>>> schema.validate()
>>> print('schema = {}'.format(ub.repr2(schema, sort=1, nl=-1)))
>>> jsonschema.validate({'key1': {'arr': []}}, schema)
schema = {
    'properties': {
        'key1': {
            'properties': {
                'arr': {'items': {}, 'type': 'array'}
            },
            'type': 'object'
        },
        'key2': {}
    },
    'required': ['key1'],
    'title': 'a title',
    'type': 'object'
}
```

kwcoco.util.**ONEOF**(**TYPES*)

class kwcoco.util.**QuantifierElements**

Bases: object

Quantifier types

https://json-schema.org/understanding-json-schema/reference/combining.html#allof

Example

```
>>> from kwcoco.util.jsonschema_elements import *  # NOQA
>>> elem.ANYOF(elem.STRING, elem.NUMBER).validate()
>>> elem.ONEOF(elem.STRING, elem.NUMBER).validate()
>>> elem.NOT(elem.NULL).validate()
>>> elem.NOT(elem.ANY).validate()
>>> elem.ANY.validate()
```

property ANY

ALLOF(**TYPES*)

ANYOF(**TYPES*)

ONEOF(**TYPES*)

NOT(*TYPE*)

class kwcoco.util.**ScalarElements**

Bases: object

Single-valued elements

property NULL

//json-schema.org/understanding-json-schema/reference/null.html

Type
https

property BOOLEAN

//json-schema.org/understanding-json-schema/reference/null.html

Type
https

property STRING

//json-schema.org/understanding-json-schema/reference/string.html

Type
https

property NUMBER

//json-schema.org/understanding-json-schema/reference/numeric.html#number

Type
https

property INTEGER

//json-schema.org/understanding-json-schema/reference/numeric.html#integer

Type
https

class kwcoco.util.**SchemaElements**

Bases: *ScalarElements*, *QuantifierElements*, *ContainerElements*

Functional interface into defining jsonschema structures.

See mixin classes for details.

References

https://json-schema.org/understanding-json-schema/

Todo:

- [ ] Generics: title, description, default, examples

CommandLine

```
xdoctest -m /home/joncrall/code/kwcoco/kwcoco/util/jsonschema_elements.py␣
↪SchemaElements
```

Example

```
>>> from kwcoco.util.jsonschema_elements import *  # NOQA
>>> elem = SchemaElements()
>>> elem.ARRAY(elem.ANY())
>>> schema = OBJECT({
>>>     'prop1': ARRAY(INTEGER, minItems=3),
>>>     'prop2': ARRAY(STRING, numItems=2),
>>>     'prop3': ARRAY(OBJECT({
>>>         'subprob1': NUMBER,
>>>         'subprob2': NUMBER,
>>>     }))
>>> })
>>> print('schema = {}'.format(ub.repr2(schema, nl=2, sort=1)))
schema = {
    'properties': {
        'prop1': {'items': {'type': 'integer'}, 'minItems': 3, 'type': 'array'},
        'prop2': {'items': {'type': 'string'}, 'maxItems': 2, 'minItems': 2, 'type
↪': 'array'},
        'prop3': {'items': {'properties': {'subprob1': {'type': 'number'}, 'subprob2
↪': {'type': 'number'}}, 'type': 'object'}, 'type': 'array'},
    },
    'type': 'object',
}
```

```
>>> TYPE = elem.OBJECT({
>>>     'p1': ANY,
>>>     'p2': ANY,
>>> }, required=['p1'])
>>> import jsonschema
>>> inst = {'p1': None}
>>> jsonschema.validate(inst, schema=TYPE)
>>> #jsonschema.validate({'p2': None}, schema=TYPE)
```

class kwcoco.util.**StratifiedGroupKFold**(*n_splits=3*, *shuffle=False*, *random_state=None*)

Bases: _BaseKFold

Stratified K-Folds cross-validator with Grouping

Provides train/test indices to split data in train/test sets.

This cross-validation object is a variation of GroupKFold that returns stratified folds. The folds are made by preserving the percentage of samples for each class.

This is an old interface and should likely be refactored and modernized.

Parameters

n_splits (*int, default=3*) – Number of folds. Must be at least 2.

split(*X*, *y*, *groups=None*)

Generate indices to split data into training and test set.

kwcoco.util.**ensure_json_serializable**(*dict_*, *normalize_containers=False*, *verbose=0*)

Attempt to convert common types (e.g. numpy) into something json complient

Convert numpy and tuples into lists

Parameters

normalize_containers (*bool*) – if True, normalizes dict containers to be standard python structures. Defaults to False.

Example

```
>>> data = ub.ddict(lambda: int)
>>> data['foo'] = ub.ddict(lambda: int)
>>> data['bar'] = np.array([1, 2, 3])
>>> data['foo']['a'] = 1
>>> data['foo']['b'] = (1, np.array([1, 2, 3]), {3: np.int32(3), 4: np.float16(1.0)}
↪)
>>> dict_ = data
>>> print(ub.repr2(data, nl=-1))
>>> assert list(find_json_unserializable(data))
>>> result = ensure_json_serializable(data, normalize_containers=True)
>>> print(ub.repr2(result, nl=-1))
>>> assert not list(find_json_unserializable(result))
>>> assert type(result) is dict
```

kwcoco.util.**find_json_unserializable**(*data*, *quickcheck=False*)

Recurse through json datastructure and find any component that causes a serialization error. Record the location of these errors in the datastructure as we recurse through the call tree.

Parameters

- **data** (*object*) – data that should be json serializable
- **quickcheck** (*bool*) – if True, check the entire datastructure assuming its ok before doing the python-based recursive logic.

Returns

list of “bad part” dictionaries containing items

’value’ - the value that caused the serialization error

’loc’ - which contains a list of key/indexes that can be used to lookup the location of the unserializable value. If the “loc” is a list, then it indicates a rare case where a key in a dictionary is causing the serialization error.

Return type
List[Dict]

Example

```
>>> from kwcoco.util.util_json import *  # NOQA
>>> part = ub.ddict(lambda: int)
>>> part['foo'] = ub.ddict(lambda: int)
>>> part['bar'] = np.array([1, 2, 3])
>>> part['foo']['a'] = 1
>>> # Create a dictionary with two unserializable parts
>>> data = [1, 2, {'nest1': [2, part]}, {frozenset({'badkey'}): 3, 2: 4}]
>>> parts = list(find_json_unserializable(data))
>>> print('parts = {}'.format(ub.repr2(parts, nl=1)))
>>> # Check expected structure of bad parts
>>> assert len(parts) == 2
>>> part = parts[1]
>>> assert list(part['loc']) == [2, 'nest1', 1, 'bar']
>>> # We can use the "loc" to find the bad value
>>> for part in parts:
>>>     # "loc" is a list of directions containing which keys/indexes
>>>     # to traverse at each descent into the data structure.
>>>     directions = part['loc']
>>>     curr = data
>>>     special_flag = False
>>>     for key in directions:
>>>         if isinstance(key, list):
>>>             # special case for bad keys
>>>             special_flag = True
>>>             break
>>>         else:
>>>             # normal case for bad values
>>>             curr = curr[key]
>>>     if special_flag:
>>>         assert part['data'] in curr.keys()
>>>         assert part['data'] is key[1]
>>>     else:
>>>         assert part['data'] is curr
```

kwcoco.util.**indexable_allclose**(*dct1*, *dct2*, *return_info=False*)

Walks through two nested data structures and ensures that everything is roughly the same.

Parameters

- **dct1** – a nested indexable item
- **dct2** – a nested indexable item

Example

```
>>> from kwcoco.util.util_json import indexable_allclose
>>> dct1 = {
>>>     'foo': [1.222222, 1.333],
>>>     'bar': 1,
>>>     'baz': [],
>>> }
>>> dct2 = {
>>>     'foo': [1.22222, 1.333],
>>>     'bar': 1,
>>>     'baz': [],
>>> }
>>> assert indexable_allclose(dct1, dct2)
```

kwcoco.util.**resolve_directory_symlinks**(*path*)

Only resolve symlinks of directories, not the base file

kwcoco.util.**resolve_relative_to**(*path*, *dpath*, *strict=False*)

Given a path, try to resolve its symlinks such that it is relative to the given dpath.

Example

```
>>> from kwcoco.util.util_reroot import *  # NOQA
>>> import os
>>> def _symlink(self, target, verbose=0):
>>>     return ub.Path(ub.symlink(target, self, verbose=verbose))
>>> ub.Path._symlink = _symlink
>>> #
>>> # TODO: try to enumerate all basic cases
>>> #
>>> base = ub.Path.appdir('kwcoco/tests/reroot')
>>> base.delete().ensuredir()
>>> #
>>> drive1 = (base / 'drive1').ensuredir()
>>> drive2 = (base / 'drive2').ensuredir()
>>> #
>>> data_repo1 = (drive1 / 'data_repo1').ensuredir()
>>> cache = (data_repo1 / '.cache').ensuredir()
>>> real_file1 = (cache / 'real_file1').touch()
>>> #
>>> real_bundle = (data_repo1 / 'real_bundle').ensuredir()
>>> real_assets = (real_bundle / 'assets').ensuredir()
>>> #
>>> # Symlink file outside of the bundle
>>> link_file1 = (real_assets / 'link_file1')._symlink(real_file1)
>>> real_file2 = (real_assets / 'real_file2').touch()
>>> link_file2 = (real_assets / 'link_file2')._symlink(real_file2)
>>> #
>>> #
>>> # A symlink to the data repo
>>> data_repo2 = (drive1 / 'data_repo2')._symlink(data_repo1)
```

(continues on next page)

(continued from previous page)

```
>>> data_repo3 = (drive2 / 'data_repo3')._symlink(data_repo1)
>>> data_repo4 = (drive2 / 'data_repo4')._symlink(data_repo2)
>>> #
>>> # A prediction repo TODO
>>> pred_repo5 = (drive2 / 'pred_repo5').ensuredir()
>>> #
>>> # _ = ub.cmd(f'tree -a {base}', verbose=3)
>>> #
>>> fpaths = []
>>> for r, ds, fs in os.walk(base, followlinks=True):
>>>     for f in fs:
>>>         if 'file' in f:
>>>             fpath = ub.Path(r) / f
>>>             fpaths.append(fpath)
>>> #
>>> #
>>> dpath = real_bundle.resolve()
>>> #
>>> for path in fpaths:
>>>     # print(f'{path}')
>>>     # print(f'{path.resolve()=}')
>>>     resolved_rel = resolve_relative_to(path, dpath)
>>>     print('resolved_rel = {!r}'.format(resolved_rel))
```

kwcoco.util.**smart_truncate**(*string*, *max_length=0*, *separator=' '*, *trunc_loc=0.5*)

Truncate a string. :param string (str): string for modification :param max_length (int): output string length :param word_boundary (bool): :param save_order (bool): if True then word order of output string is like input string :param separator (str): separator between words :param trunc_loc (float): fraction of location where to remove the text :return:

kwcoco.util.**special_reroot_single**(*dset*, *verbose=0*)

kwcoco.util.**unarchive_file**(*archive_fpath*, *output_dpath='.'*, *verbose=1*, *overwrite=True*)

2.1.2 Submodules

2.1.2.1 kwcoco.abstract_coco_dataset module

class kwcoco.abstract_coco_dataset.**AbstractCocoDataset**

Bases: ABC

This is a common base for all variants of the Coco Dataset

At the time of writing there is kwcoco.CocoDataset (which is the dictionary-based backend), and the kwcoco.coco_sql_dataset.CocoSqlDataset, which is experimental.

2.1.2.2 kwcoco.category_tree module

The `category_tree` module defines the *CategoryTree* class, which is used for maintaining flat or hierarchical category information. The kwcoco version of this class only contains the datastructure and does not contain any torch operations. See the ndsampler version for the extension with torch operations.

class kwcoco.category_tree.**CategoryTree**(*graph=None*, *checks=True*)

Bases: NiceRepr

Wrapper that maintains flat or hierarchical category information.

Helps compute softmaxes and probabilities for tree-based categories where a directed edge (A, B) represents that A is a superclass of B.

Note: There are three basic properties that this object maintains:

```
node:
    Alphanumeric string names that should be generally descriptive.
    Using spaces and special characters in these names is
    discouraged, but can be done.  This is the COCO category "name"
    attribute.  For categories this may be denoted as (name, node,
    cname, catname).

id:
    The integer id of a category should ideally remain consistent.
    These are often given by a dataset (e.g. a COCO dataset).  This
    is the COCO category "id" attribute. For categories this is
    often denoted as (id, cid).

index:
    Contigous zero-based indices that indexes the list of
    categories.  These should be used for the fastest access in
    backend computation tasks. Typically corresponds to the
    ordering of the channels in the final linear layer in an
    associated model.  For categories this is often denoted as
    (index, cidx, idx, or cx).
```

Variables

- **idx_to_node** (*List[str]*) – a list of class names. Implicitly maps from index to category name.
- **id_to_node** (*Dict[int, str]*) – maps integer ids to category names
- **node_to_id** (*Dict[str, int]*) – maps category names to ids
- **node_to_idx** (*Dict[str, int]*) – maps category names to indexes
- **graph** (*networkx.Graph*) – a Graph that stores any hierarchy information. For standard mutually exclusive classes, this graph is edgeless. Nodes in this graph can maintain category attributes / properties.
- **idx_groups** (*List[List[int]]*) – groups of category indices that share the same parent category.

Example

```
>>> from kwcoco.category_tree import *
>>> graph = nx.from_dict_of_lists({
>>>     'background': [],
>>>     'foreground': ['animal'],
>>>     'animal': ['mammal', 'fish', 'insect', 'reptile'],
>>>     'mammal': ['dog', 'cat', 'human', 'zebra'],
>>>     'zebra': ['grevys', 'plains'],
>>>     'grevys': ['fred'],
>>>     'dog': ['boxer', 'beagle', 'golden'],
>>>     'cat': ['maine coon', 'persian', 'sphynx'],
>>>     'reptile': ['bearded dragon', 't-rex'],
>>> }, nx.DiGraph)
>>> self = CategoryTree(graph)
>>> print(self)
<CategoryTree(nNodes=22, maxDepth=6, maxBreadth=4...)>
```

Example

```
>>> # The coerce classmethod is the easiest way to create an instance
>>> import kwcoco
>>> kwcoco.CategoryTree.coerce(['a', 'b', 'c'])
<CategoryTree...nNodes=3, nodes=...'a', 'b', 'c'...
>>> kwcoco.CategoryTree.coerce(4)
<CategoryTree...nNodes=4, nodes=...'class_1', 'class_2', 'class_3', ...
>>> kwcoco.CategoryTree.coerce(4)
```

copy()

classmethod from_mutex(*nodes*, *bg_hack=True*)

Parameters
nodes (*List[str]*) – or a list of class names (in which case they will all be assumed to be mutually exclusive)

Example

```
>>> print(CategoryTree.from_mutex(['a', 'b', 'c']))
<CategoryTree(nNodes=3, ...)>
```

classmethod from_json(*state*)

Parameters
state (*Dict*) – see __getstate__ / __json__ for details

classmethod from_coco(*categories*)

Create a CategoryTree object from coco categories

Parameters
List[Dict] – list of coco-style categories

classmethod coerce(*data*, ***kw*)

Attempt to coerce data as a CategoryTree object.

This is primarily useful for when the software stack depends on categories being represent

This will work if the input data is a specially formatted json dict, a list of mutually exclusive classes, or if it is already a CategoryTree. Otherwise an error will be thrown.

Parameters

- **data** (*object*) – a known representation of a category tree.
- ****kwargs** – input type specific arguments

Returns
self

Return type
CategoryTree

Raises

- **TypeError - if the input format is unknown** –
- **ValueError - if kwargs are not compatible with the input format** –

Example

```
>>> import kwcoco
>>> classes1 = kwcoco.CategoryTree.coerce(3)  # integer
>>> classes2 = kwcoco.CategoryTree.coerce(classes1.__json__())  # graph dict
>>> classes3 = kwcoco.CategoryTree.coerce(['class_1', 'class_2', 'class_3'])  #␣
↪mutex list
>>> classes4 = kwcoco.CategoryTree.coerce(classes1.graph)  # nx Graph
>>> classes5 = kwcoco.CategoryTree.coerce(classes1)  # cls
>>> # xdoctest: +REQUIRES(module:ndsampler)
>>> import ndsampler
>>> classes6 = ndsampler.CategoryTree.coerce(3)
>>> classes7 = ndsampler.CategoryTree.coerce(classes1)
>>> classes8 = kwcoco.CategoryTree.coerce(classes6)
```

classmethod demo(*key='coco'*, ***kwargs*)

Parameters
key (*str*) – specify which demo dataset to use. Can be ‘coco’ (which uses the default coco demo data). Can be ‘btree’ which creates a binary tree and accepts kwargs ‘r’ and ‘h’ for branching-factor and height. Can be ‘btree2’, which is the same as btree but returns strings

CommandLine

```
xdoctest -m ~/code/kwcoco/kwcoco/category_tree.py CategoryTree.demo
```

Example

```
>>> from kwcoco.category_tree import *
>>> self = CategoryTree.demo()
>>> print('self = {}'.format(self))
self = <CategoryTree(nNodes=10, maxDepth=2, maxBreadth=4...)>
```

to_coco()

Converts to a coco-style data structure

Yields
Dict – coco category dictionaries

property id_to_idx

Example:

```
>>> import kwcoco
>>> self = kwcoco.CategoryTree.demo()
>>> self.id_to_idx[1]
```

property idx_to_id

Example:

```
>>> import kwcoco
>>> self = kwcoco.CategoryTree.demo()
>>> self.idx_to_id[0]
```

idx_to_ancestor_idxs(*include_self=True*)

Mapping from a class index to its ancestors

Parameters
include_self (*bool, default=True*) – if True includes each node as its own ancestor.

idx_to_descendants_idxs(*include_self=False*)

Mapping from a class index to its descendants (including itself)

Parameters
include_self (*bool, default=False*) – if True includes each node as its own descendant.

idx_pairwise_distance()

Get a matrix encoding the distance from one class to another.

Distances

- from parents to children are positive (descendants),
- from children to parents are negative (ancestors),
- between unreachable nodes (wrt to forward and reverse graph) are nan.

is_mutex()

Returns True if all categories are mutually exclusive (i.e. flat)

If true, then the classes may be represented as a simple list of class names without any loss of information, otherwise the underlying category graph is necessary to preserve all knowledge.

Todo:

- [ ] what happens when we have a dummy root?

---

property num_classes

property class_names

property category_names

property cats

Returns a mapping from category names to category attributes.

If this category tree was constructed from a coco-dataset, then this will contain the coco category attributes.

Returns
Dict[str, Dict[str, object]]

Example

```
>>> from kwcoco.category_tree import *
>>> self = CategoryTree.demo()
>>> print('self.cats = {!r}'.format(self.cats))
```

index(*node*)

Return the index that corresponds to the category name

show()

forest_str()

normalize()

Applies a normalization scheme to the categories.

Note: this may break other tasks that depend on exact category names.

Returns
CategoryTree

Example

```
>>> from kwcoco.category_tree import *  # NOQA
>>> import kwcoco
>>> orig = kwcoco.CategoryTree.demo('animals_v1')
>>> self = kwcoco.CategoryTree(nx.relabel_nodes(orig.graph, str.upper))
>>> norm = self.normalize()
```

2.1.2.3 kwcoco.channel_spec module

This module defines the KWCOCO Channel Specification and API.

The KWCOCO Channel specification is a way to semantically express how a combination of image channels are grouped. This can specify how these channels (somtimes called bands or features) are arranged on disk or input to an algorithm. The core idea reduces to a `Set[List[str]]` — or a unordered set of ordered sequences of strings corresponding to channel "names". The way these are specified is with a "," to separate lists in an unordered set and with a "|" to separate the channel names. Other syntax exists for convinience, but a strict normalized channel spec only contains these core symbols.

Another way to think of a kwcoco channel spec is that splitting the spec by "," gives groups of channels that should be processed together and "late-fused". Within each group the "|" operator "early-fuses" the channels.

For instance, say we had a network and we wanted to process 3-channel rgb images in one stream and 1-channel infrared images in a second stream and then fuse them together. The kwcoco channel specification for channels labled as 'red', 'green', 'blue', and 'infrared' would be:

```
infrared,red|green|blue
```

Note, it is up to an algorithm to do any early-late fusion. KWCoco simply provides the specification as a tool to quickly access a particular combination of channels from disk.

The ChannelSpec has these simple rules:

```
* each 1D channel is a alphanumeric string.

* The pipe ('|') separates aligned early fused stremas (non-communative)

* The comma (',') separates late-fused streams, (happens after pipe operations, and is␣
↪communative)

* Certain common sets of early fused channels have codenames, for example:

    rgb = r|g|b
    rgba = r|g|b|a
    dxdy = dy|dy

* Multiple channels can be specified via a "slice" notation. For example:

    mychan.0:4

    represents 4 channels:
        mychan.0, mychan.1, mychan.2, and mychan.3

    slices after the "." work like python slices
```

The detailed grammar for the spec is

```
?start: stream

// An identifier can contain spaces
IDEN: ("_"|LETTER) ("_"|" "|LETTER|DIGIT)*

chan_single : IDEN
chan_getitem : IDEN "." INT
```

(continues on next page)

(continued from previous page)

```
chan_getslice_0b : IDEN ":" INT
chan_getslice_ab : IDEN "." INT ":" INT

// A channel code can just be an ID, or it can have a getitem
// style syntax with a scalar or slice as an argument
chan_code : chan_single | chan_getslice_0b | chan_getslice_ab | chan_getitem

// Fused channels are an ordered sequence of channel codes (without sensors)
fused : chan_code ("|" chan_code)*

// Channels can be specified in a sequence but must contain parens
fused_seq : "(" fused ("," fused)* ")"

channel_rhs : fused | fused_seq

stream : channel_rhs ("," channel_rhs)*

%import common.DIGIT
%import common.LETTER
%import common.INT
```

Note that a stream refers to a the full ChannelSpec and fused refers to FusedChannelSpec.

For single arrays, the spec is always an early fused spec.

Todo:

- [X] : normalize representations? e.g: rgb = r|g|b? - OPTIONAL
- [X] : rename to BandsSpec or SensorSpec? - REJECTED
- [ ] : allow bands to be coerced, i.e. rgb -> gray, or gray->rgb

---

Todo:

- [x]: Use FusedChannelSpec as a member of ChannelSpec
- [x]: Handle special slice suffix for length calculations

SeeAlso:
:module:kwcoco.sensorchan_spec - The generalized sensor / channel specification

---

Note:

- do not specify the same channel in FusedChannelSpec twice

Example

```
>>> import kwcoco
>>> spec = kwcoco.ChannelSpec('b1|b2|b3,m.0:4|x1|x2,x.3|x.4|x.5')
>>> print(spec)
<ChannelSpec(b1|b2|b3,m.0:4|x1|x2,x.3|x.4|x.5)>
>>> for stream in spec.streams():
>>>     print(stream)
<FusedChannelSpec(b1|b2|b3)>
<FusedChannelSpec(m.0:4|x1|x2)>
<FusedChannelSpec(x.3|x.4|x.5)>
>>> # Normalization
>>> normalized = spec.normalize()
>>> print(normalized)
<ChannelSpec(b1|b2|b3,m.0|m.1|m.2|m.3|x1|x2,x.3|x.4|x.5)>
>>> print(normalized.fuse().spec)
b1|b2|b3|m.0|m.1|m.2|m.3|x1|x2|x.3|x.4|x.5
>>> print(normalized.fuse().concise().spec)
b1|b2|b3|m:4|x1|x2|x.3:6
```

class kwcoco.channel_spec.**BaseChannelSpec**

Bases: NiceRepr

Common code API between *FusedChannelSpec* and *ChannelSpec*

Todo:

- **[ ] Keep working on this base spec and ensure the inheriting classes**
  conform to it.

---

abstract property spec

The string encodeing of this spec

Returns
str

abstract classmethod coerce(*data*)

Try and interpret the input data as some sort of spec

Parameters
data (*str | int | list | dict | BaseChannelSpec*) – any input data that is known to represent a spec

Returns
BaseChannelSpec

abstract streams()

Breakup this spec into individual early-fused components

Returns
List[FusedChannelSpec]

abstract normalize()

Expand all channel codes into their normalized long-form

Returns
BaseChannelSpec

abstract intersection(*other*)

abstract union(*other*)

abstract difference()

abstract issubset(*other*)

abstract issuperset(*other*)

late_fuse(*other*)

Example

```
>>> import kwcoco
>>> a = kwcoco.ChannelSpec.coerce('A|B|C,edf')
>>> b = kwcoco.ChannelSpec.coerce('A12')
>>> c = kwcoco.ChannelSpec.coerce('')
>>> d = kwcoco.ChannelSpec.coerce('rgb')
>>> print(a.late_fuse(b).spec)
>>> print((a + b).spec)
>>> print((b + a).spec)
>>> print((a + b + c).spec)
>>> print(sum([a, b, c, d]).spec)
A|B|C,edf,A12
A|B|C,edf,A12
A12,A|B|C,edf
A|B|C,edf,A12
A|B|C,edf,A12,rgb
```

path_sanitize(*maxlen=None*)

Clean up the channel spec so it can be used in a pathname.

Parameters

maxlen (*int*) – if specified, and the name is longer than this length, it is shortened. Must be 8 or greater.

Returns

path suitable for usage in a filename

Return type

str

Note: This mapping is not invertible and should not be relied on to reconstruct the path spec. This is only a convenience.

Example

```
>>> import kwcoco
>>> print(kwcoco.FusedChannelSpec.coerce('a chan with space|bar|baz').path_
↪sanitize())
a chan with space_bar_baz
>>> print(kwcoco.ChannelSpec.coerce('foo|bar|baz,biz').path_sanitize())
foo_bar_baz,biz
```

Example

```
>>> import kwcoco
>>> print(kwcoco.ChannelSpec.coerce('foo.0:3').normalize().path_sanitize(24))
foo.0_foo.1_foo.2
>>> print(kwcoco.ChannelSpec.coerce('foo.0:256').normalize().path_sanitize(24))
tuuxtfnrsvdhezkdndysxo_256
```

class kwcoco.channel_spec.**FusedChannelSpec**(*parsed*, *_is_normalized=False*)

Bases: *BaseChannelSpec*

A specific type of channel spec with only one early fused stream.

The channels in this stream are non-communative

Behaves like a list of atomic-channel codes (which may represent more than 1 channel), normalized codes always represent exactly 1 channel.

Note: This class name and API is in flux and subject to change.

Todo: A special code indicating a name and some number of bands that that names contains, this would primarilly be used for large numbers of channels produced by a network. Like:

resnet_d35d060_L5:512

or

resnet_d35d060_L5[:512]

might refer to a very specific (hashed) set of resnet parameters with 512 bands

maybe we can do something slicly like:

resnet_d35d060_L5[A:B] resnet_d35d060_L5:A:B

Do we want to "just store the code" and allow for parsing later?

Or do we want to ensure the serialization is parsed before we construct the data structure?

Example

```
>>> from kwcoco.channel_spec import *  # NOQA
>>> import pickle
>>> self = FusedChannelSpec.coerce(3)
>>> recon = pickle.loads(pickle.dumps(self))
>>> self = ChannelSpec.coerce('a|b,c|d')
>>> recon = pickle.loads(pickle.dumps(self))
```

classmethod concat(*items*)

property spec

unique()

classmethod parse(*spec*)

classmethod coerce(*data*)

Example

```
>>> from kwcoco.channel_spec import *  # NOQA
>>> FusedChannelSpec.coerce(['a', 'b', 'c'])
>>> FusedChannelSpec.coerce('a|b|c')
>>> FusedChannelSpec.coerce(3)
>>> FusedChannelSpec.coerce(FusedChannelSpec(['a']))
>>> assert FusedChannelSpec.coerce('').numel() == 0
```

concise()

Shorted the channel spec by de-normaliz slice syntax

Returns
concise spec

Return type
FusedChannelSpec

Example

```
>>> from kwcoco.channel_spec import *  # NOQA
>>> self = FusedChannelSpec.coerce(
>>>      'b|a|a.0|a.1|a.2|a.5|c|a.8|a.9|b.0:3|c.0')
>>> short = self.concise()
>>> long = short.normalize()
>>> numels = [c.numel() for c in [self, short, long]]
>>> print('self.spec  = {!r}'.format(self.spec))
>>> print('short.spec = {!r}'.format(short.spec))
>>> print('long.spec  = {!r}'.format(long.spec))
>>> print('numels = {!r}'.format(numels))
self.spec  = 'b|a|a.0|a.1|a.2|a.5|c|a.8|a.9|b.0:3|c.0'
short.spec = 'b|a|a:3|a.5|c|a.8:10|b:3|c.0'
long.spec  = 'b|a|a.0|a.1|a.2|a.5|c|a.8|a.9|b.0|b.1|b.2|c.0'
```

(continues on next page)

(continued from previous page)

```
numels = [13, 13, 13]
>>> assert long.concise().spec == short.spec
```

normalize()

Replace aliases with explicit single-band-per-code specs

Returns
normalize spec

Return type
FusedChannelSpec

Example

```
>>> from kwcoco.channel_spec import *  # NOQA
>>> self = FusedChannelSpec.coerce('b1|b2|b3|rgb')
>>> normed = self.normalize()
>>> print('self = {}'.format(self))
>>> print('normed = {}'.format(normed))
self = <FusedChannelSpec(b1|b2|b3|rgb)>
normed = <FusedChannelSpec(b1|b2|b3|r|g|b)>
>>> self = FusedChannelSpec.coerce('B:1:11')
>>> normed = self.normalize()
>>> print('self = {}'.format(self))
>>> print('normed = {}'.format(normed))
self = <FusedChannelSpec(B:1:11)>
normed = <FusedChannelSpec(B.1|B.2|B.3|B.4|B.5|B.6|B.7|B.8|B.9|B.10)>
>>> self = FusedChannelSpec.coerce('B.1:11')
>>> normed = self.normalize()
>>> print('self = {}'.format(self))
>>> print('normed = {}'.format(normed))
self = <FusedChannelSpec(B.1:11)>
normed = <FusedChannelSpec(B.1|B.2|B.3|B.4|B.5|B.6|B.7|B.8|B.9|B.10)>
```

numel()

Total number of channels in this spec

sizes()

Returns a list indicating the size of each atomic code

Returns
List[int]

Example

```
>>> from kwcoco.channel_spec import *  # NOQA
>>> self = FusedChannelSpec.coerce('b1|Z:3|b2|b3|rgb')
>>> self.sizes()
[1, 3, 1, 1, 3]
>>> assert(FusedChannelSpec.parse('a.0').numel()) == 1
>>> assert(FusedChannelSpec.parse('a:0').numel()) == 0
>>> assert(FusedChannelSpec.parse('a:1').numel()) == 1
```

code_list()

Return the expanded code list

as_list()

as_oset()

as_set()

to_set()

to_oset()

to_list()

as_path()

Returns a string suitable for use in a path.

Note, this may no longer be a valid channel spec

difference(*other*)

Set difference

Example

```
>>> FCS = FusedChannelSpec.coerce
>>> self = FCS('rgb|disparity|flowx|flowy')
>>> other = FCS('r|b')
>>> self.difference(other)
>>> other = FCS('flowx')
>>> self.difference(other)
>>> FCS = FusedChannelSpec.coerce
>>> assert len((FCS('a') - {'a'}).parsed) == 0
>>> assert len((FCS('a.0:3') - {'a.0'}).parsed) == 2
```

intersection(*other*)

Example

```
>>> FCS = FusedChannelSpec.coerce
>>> self = FCS('rgb|disparity|flowx|flowy')
>>> other = FCS('r|b|XX')
>>> self.intersection(other)
```

union(*other*)

Example

```
>>> from kwcoco.channel_spec import *  # NOQA
>>> FCS = FusedChannelSpec.coerce
>>> self = FCS('rgb|disparity|flowx|flowy')
>>> other = FCS('r|b|XX')
>>> self.union(other)
```

issubset(*other*)

issuperset(*other*)

component_indices(*axis=2*)

Look up component indices within this stream

Example

```
>>> FCS = FusedChannelSpec.coerce
>>> self = FCS('disparity|rgb|flowx|flowy')
>>> component_indices = self.component_indices()
>>> print('component_indices = {}'.format(ub.repr2(component_indices, nl=1)))
component_indices = {
    'disparity': (slice(...), slice(...), slice(0, 1, None)),
    'flowx': (slice(...), slice(...), slice(4, 5, None)),
    'flowy': (slice(...), slice(...), slice(5, 6, None)),
    'rgb': (slice(...), slice(...), slice(1, 4, None)),
}
```

streams()

Idempotence with *ChannelSpec.streams()*

fuse()

Idempotence with *ChannelSpec.streams()*

class kwcoco.channel_spec.**ChannelSpec**(*spec*, *parsed=None*)

Bases: *BaseChannelSpec*

Parse and extract information about network input channel specs for early or late fusion networks.

Behaves like a dictionary of FusedChannelSpec objects

Todo:

- **[ ] Rename to something that indicates this is a collection of** FusedChannelSpec? MultiChannelSpec?

Note: This class name and API is in flux and subject to change.

Note: The pipe ('|') character represents an early-fused input stream, and order matters (it is non-communative).

The comma (',') character separates different inputs streams/branches for a multi-stream/branch network which will be lated fused. Order does not matter

---

Example

```
>>> from kwcoco.channel_spec import *  # NOQA
>>> # Integer spec
>>> ChannelSpec.coerce(3)
<ChannelSpec(u0|u1|u2) ...>
```

```
>>> # single mode spec
>>> ChannelSpec.coerce('rgb')
<ChannelSpec(rgb) ...>
```

```
>>> # early fused input spec
>>> ChannelSpec.coerce('rgb|disprity')
<ChannelSpec(rgb|disprity) ...>
```

```
>>> # late fused input spec
>>> ChannelSpec.coerce('rgb,disprity')
<ChannelSpec(rgb,disprity) ...>
```

```
>>> # early and late fused input spec
>>> ChannelSpec.coerce('rgb|ir,disprity')
<ChannelSpec(rgb|ir,disprity) ...>
```

Example

```
>>> self = ChannelSpec('gray')
>>> print('self.info = {}'.format(ub.repr2(self.info, nl=1)))
>>> self = ChannelSpec('rgb')
>>> print('self.info = {}'.format(ub.repr2(self.info, nl=1)))
>>> self = ChannelSpec('rgb|disparity')
>>> print('self.info = {}'.format(ub.repr2(self.info, nl=1)))
>>> self = ChannelSpec('rgb|disparity,disparity')
>>> print('self.info = {}'.format(ub.repr2(self.info, nl=1)))
>>> self = ChannelSpec('rgb,disparity,flowx|flowy')
>>> print('self.info = {}'.format(ub.repr2(self.info, nl=1)))
```

Example

```
>>> specs = [
>>>     'rgb',              # and rgb input
>>>     'rgb|disprity',     # rgb early fused with disparity
>>>     'rgb,disprity',     # rgb early late with disparity
>>>     'rgb|ir,disprity',  # rgb early fused with ir and late fused with disparity
>>>     3,                  # 3 unknown channels
>>> ]
>>> for spec in specs:
>>>     print('======================')
>>>     print('spec = {!r}'.format(spec))
>>>     #
>>>     self = ChannelSpec.coerce(spec)
>>>     print('self = {!r}'.format(self))
>>>     sizes = self.sizes()
>>>     print('sizes = {!r}'.format(sizes))
>>>     print('self.info = {}'.format(ub.repr2(self.info, nl=1)))
>>>     #
>>>     item = self._demo_item((1, 1), rng=0)
>>>     inputs = self.encode(item)
>>>     components = self.decode(inputs)
>>>     input_shapes = ub.map_vals(lambda x: x.shape, inputs)
>>>     component_shapes = ub.map_vals(lambda x: x.shape, components)
>>>     print('item = {}'.format(ub.repr2(item, precision=1)))
>>>     print('inputs = {}'.format(ub.repr2(inputs, precision=1)))
>>>     print('input_shapes = {}'.format(ub.repr2(input_shapes)))
>>>     print('components = {}'.format(ub.repr2(components, precision=1)))
>>>     print('component_shapes = {}'.format(ub.repr2(component_shapes, nl=1)))
```

property spec

property info

classmethod coerce(*data*)

Attempt to interpret the data as a channel specification

Returns
ChannelSpec

Example

```
>>> from kwcoco.channel_spec import *  # NOQA
>>> data = FusedChannelSpec.coerce(3)
>>> assert ChannelSpec.coerce(data).spec == 'u0|u1|u2'
>>> data = ChannelSpec.coerce(3)
>>> assert data.spec == 'u0|u1|u2'
>>> assert ChannelSpec.coerce(data).spec == 'u0|u1|u2'
>>> data = ChannelSpec.coerce('u:3')
>>> assert data.normalize().spec == 'u.0|u.1|u.2'
```

parse()

Build internal representation

Example

```
>>> from kwcoco.channel_spec import *  # NOQA
>>> self = ChannelSpec('b1|b2|b3|rgb,B:3')
>>> print(self.parse())
>>> print(self.normalize().parse())
>>> ChannelSpec('').parse()
```

Example

```
>>> base = ChannelSpec('rgb|disparity,flowx|r|flowy')
>>> other = ChannelSpec('rgb')
>>> self = base.intersection(other)
>>> assert self.numel() == 4
```

concise()

Example

```
>>> self = ChannelSpec('b1|b2,b3|rgb|B.0,B.1|B.2')
>>> print(self.concise().spec)
b1|b2,b3|r|g|b|B.0,B.1:3
```

normalize()

Replace aliases with explicit single-band-per-code specs

Returns
normalized spec

Return type
ChannelSpec

Example

```
>>> self = ChannelSpec('b1|b2,b3|rgb,B:3')
>>> normed = self.normalize()
>>> print('self   = {}'.format(self))
>>> print('normed = {}'.format(normed))
self   = <ChannelSpec(b1|b2,b3|rgb,B:3)>
normed = <ChannelSpec(b1|b2,b3|r|g|b,B.0|B.1|B.2)>
```

keys()

values()

items()

fuse()

Fuse all parts into an early fused channel spec

Returns
FusedChannelSpec

Example

```
>>> from kwcoco.channel_spec import *  # NOQA
>>> self = ChannelSpec.coerce('b1|b2,b3|rgb,B:3')
>>> fused = self.fuse()
>>> print('self  = {}'.format(self))
>>> print('fused = {}'.format(fused))
self  = <ChannelSpec(b1|b2,b3|rgb,B:3)>
fused = <FusedChannelSpec(b1|b2|b3|rgb|B:3)>
```

streams()

Breaks this spec up into one spec for each early-fused input stream

Example

self = ChannelSpec.coerce('r|g,B1|B2,fx|fy') list(map(len, self.streams()))

code_list()

as_path()

Returns a string suitable for use in a path.

Note, this may no longer be a valid channel spec

difference(*other*)

Set difference. Remove all instances of other channels from this set of channels.

Example

```
>>> from kwcoco.channel_spec import *
>>> self = ChannelSpec('rgb|disparity,flowx|r|flowy')
>>> other = ChannelSpec('rgb')
>>> print(self.difference(other))
>>> other = ChannelSpec('flowx')
>>> print(self.difference(other))
<ChannelSpec(disparity,flowx|flowy)>
<ChannelSpec(r|g|b|disparity,r|flowy)>
```

Example

```
>>> from kwcoco.channel_spec import *
>>> self = ChannelSpec('a|b,c|d')
>>> new = self - {'a', 'b'}
>>> len(new.sizes()) == 1
>>> empty = new - 'c|d'
>>> assert empty.numel() == 0
```

intersection(*other*)

Set difference. Remove all instances of other channels from this set of channels.

Example

```
>>> from kwcoco.channel_spec import *
>>> self = ChannelSpec('rgb|disparity,flowx|r|flowy')
>>> other = ChannelSpec('rgb')
>>> new = self.intersection(other)
>>> print(new)
>>> print(new.numel())
>>> other = ChannelSpec('flowx')
>>> new = self.intersection(other)
>>> print(new)
>>> print(new.numel())
<ChannelSpec(r|g|b,r)>
4
<ChannelSpec(flowx)>
1
```

union(*other*)

Union simply tags on a second channel spec onto this one. Duplicates are maintained.

Example

```
>>> from kwcoco.channel_spec import *
>>> self = ChannelSpec('rgb|disparity,flowx|r|flowy')
>>> other = ChannelSpec('rgb')
>>> new = self.union(other)
>>> print(new)
>>> print(new.numel())
>>> other = ChannelSpec('flowx')
>>> new = self.union(other)
>>> print(new)
>>> print(new.numel())
<ChannelSpec(r|g|b|disparity,flowx|r|flowy,r|g|b)>
10
<ChannelSpec(r|g|b|disparity,flowx|r|flowy,flowx)>
8
```

issubset(*other*)

issuperset(*other*)

numel()

Total number of channels in this spec

sizes()

Number of dimensions for each fused stream channel

IE: The EARLY-FUSED channel sizes

Example

```
>>> self = ChannelSpec('rgb|disparity,flowx|flowy,B:10')
>>> self.normalize().concise()
>>> self.sizes()
```

unique(*normalize=False*)

Returns the unique channels that will need to be given or loaded

encode(*item*, *axis=0*, *mode=1*)

Given a dictionary containing preloaded components of the network inputs, build a concatenated (fused) network representations of each input stream.

Parameters

- **item** (*Dict[str, Tensor]*) – a batch item containing unfused parts. each key should be a single-stream (optionally early fused) channel key.
- **axis** (*int, default=0*) – concatenation dimension

Returns

mapping between input stream and its early fused tensor input.

Return type

Dict[str, Tensor]

Example

```
>>> from kwcoco.channel_spec import *  # NOQA
>>> import numpy as np
>>> dims = (4, 4)
>>> item = {
>>>     'rgb': np.random.rand(3, *dims),
>>>     'disparity': np.random.rand(1, *dims),
>>>     'flowx': np.random.rand(1, *dims),
>>>     'flowy': np.random.rand(1, *dims),
>>> }
>>> # Complex Case
>>> self = ChannelSpec('rgb,disparity,rgb|disparity|flowx|flowy,flowx|flowy')
>>> fused = self.encode(item)
>>> input_shapes = ub.map_vals(lambda x: x.shape, fused)
>>> print('input_shapes = {}'.format(ub.repr2(input_shapes, nl=1)))
>>> # Simpler case
>>> self = ChannelSpec('rgb|disparity')
>>> fused = self.encode(item)
>>> input_shapes = ub.map_vals(lambda x: x.shape, fused)
>>> print('input_shapes = {}'.format(ub.repr2(input_shapes, nl=1)))
```

Example

```
>>> # Case where we have to break up early fused data
>>> import numpy as np
>>> dims = (40, 40)
>>> item = {
>>>     'rgb|disparity': np.random.rand(4, *dims),
>>>     'flowx': np.random.rand(1, *dims),
>>>     'flowy': np.random.rand(1, *dims),
>>> }
>>> # Complex Case
>>> self = ChannelSpec('rgb,disparity,rgb|disparity,rgb|disparity|flowx|flowy,
↪flowx|flowy,flowx,disparity')
>>> inputs = self.encode(item)
>>> input_shapes = ub.map_vals(lambda x: x.shape, inputs)
>>> print('input_shapes = {}'.format(ub.repr2(input_shapes, nl=1)))
```

```
>>> # xdoctest: +REQUIRES(--bench)
>>> #self = ChannelSpec('rgb|disparity,flowx|flowy')
>>> import timerit
>>> ti = timerit.Timerit(100, bestof=10, verbose=2)
>>> for timer in ti.reset('mode=simple'):
>>>     with timer:
>>>         inputs = self.encode(item, mode=0)
>>> for timer in ti.reset('mode=minimize-concat'):
>>>     with timer:
>>>         inputs = self.encode(item, mode=1)
```

decode(*inputs*, *axis=1*)

break an early fused item into its components

Parameters

- **inputs** (*Dict[str, Tensor]*) – dictionary of components
- **axis** (*int, default=1*) – channel dimension

Example

```
>>> from kwcoco.channel_spec import *  # NOQA
>>> import numpy as np
>>> dims = (4, 4)
>>> item_components = {
>>>     'rgb': np.random.rand(3, *dims),
>>>     'ir': np.random.rand(1, *dims),
>>> }
>>> self = ChannelSpec('rgb|ir')
>>> item_encoded = self.encode(item_components)
>>> batch = {k: np.concatenate([v[None, :], v[None, :]], axis=0)
...          for k, v in item_encoded.items()}
>>> components = self.decode(batch)
```

Example

```
>>> # xdoctest: +REQUIRES(module:netharn, module:torch)
>>> import torch
>>> import numpy as np
>>> dims = (4, 4)
>>> components = {
>>>     'rgb': np.random.rand(3, *dims),
>>>     'ir': np.random.rand(1, *dims),
>>> }
>>> components = ub.map_vals(torch.from_numpy, components)
>>> self = ChannelSpec('rgb|ir')
>>> encoded = self.encode(components)
>>> from netharn.data import data_containers
>>> item = {k: data_containers.ItemContainer(v, stack=True)
>>>         for k, v in encoded.items()}
>>> batch = data_containers.container_collate([item, item])
>>> components = self.decode(batch)
```

component_indices(*axis=2*)

Look up component indices within fused streams

Example

```
>>> dims = (4, 4)
>>> inputs = ['flowx', 'flowy', 'disparity']
>>> self = ChannelSpec('disparity,flowx|flowy')
>>> component_indices = self.component_indices()
>>> print('component_indices = {}'.format(ub.repr2(component_indices, nl=1)))
component_indices = {
    'disparity': ('disparity', (slice(None, None, None), slice(None, None,␣
↪None), slice(0, 1, None))),
    'flowx': ('flowx|flowy', (slice(None, None, None), slice(None, None, None),␣
↪slice(0, 1, None))),
    'flowy': ('flowx|flowy', (slice(None, None, None), slice(None, None, None),␣
↪slice(1, 2, None))),
}
```

kwcoco.channel_spec.**subsequence_index**(*oset1*, *oset2*)

Returns a slice into the first items indicating the position of the second items if they exist.

This is a variant of the substring problem.

Returns

None | slice

Example

```
>>> oset1 = ub.oset([1, 2, 3, 4, 5, 6])
>>> oset2 = ub.oset([2, 3, 4])
>>> index = subsequence_index(oset1, oset2)
>>> assert index
```

```
>>> oset1 = ub.oset([1, 2, 3, 4, 5, 6])
>>> oset2 = ub.oset([2, 4, 3])
>>> index = subsequence_index(oset1, oset2)
>>> assert not index
```

kwcoco.channel_spec.**oset_insert**(*self*, *index*, *obj*)

kwcoco.channel_spec.**oset_delitem**(*self*, *index*)

for ubelt oset, todo contribute back to luminosoinsight

```
>>> self = ub.oset([1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> index = slice(3, 5)
>>> oset_delitem(self, index)
```

2.1.2.4 kwcoco.coco_dataset module

An implementation and extension of the original MS-COCO API [CocoFormat].

Extends the format to also include line annotations.

The following describes psuedo-code for the high level spec (some of which may not be have full support in the Python API). A formal json-schema is defined in *kwcoco.coco_schema*.

An informal spec is as follows:

```
# All object categories are defined here.
category = {
    'id': int,
    'name': str,  # unique name of the category
    'supercategory': str,   # parent category name
}

# Videos are used to manage collections or sequences of images.
# Frames do not necesarilly have to be aligned or uniform time steps
video = {
    'id': int,
    'name': str,  # a unique name for this video.

    'width': int  # the base width of this video (all associated images must have this␣
↪width)
    'height': int  # the base height of this video (all associated images must have this␣
↪height)

    # In the future this may be extended to allow pointing to video files
}
```

(continues on next page)

(continued from previous page)

```
# Specifies how to find sensor data of a particular scene at a particular
# time. This is usually paths to rgb images, but auxiliary information
# can be used to specify multiple bands / etc...

# NOTE: in the future we will transition from calling these auxiliary items
# to calling these asset items. As such the key will change from
# "auxiliary" to "asset". The API will be updated to maintain backwards
# compatibility while this transition occurs.

image = {
    'id': int,

    'name': str,  # an encouraged but optional unique name (ideally not larger than 256␣
↪characters)
    'file_name': str,  # relative path to the "base" image data (optional if auxiliary␣
↪items are specified)

    'width': int,   # pixel width of "base" image
    'height': int,  # pixel height of "base" image

    'channels': <ChannelSpec>,   # a string encoding of the channels in the main image␣
↪(optional if auxiliary items are specified)

    'auxiliary': [  # information about any auxiliary channels / bands
        {
            'file_name': str,     # relative path to associated file
            'channels': <ChannelSpec>,   # a string encoding
            'width':     <int>    # pixel width of image asset
            'height':    <int>    # pixel height of image asset
            'warp_aux_to_img': <TransformSpec>,  # tranform from "base" image space to␣
↪auxiliary/asset space. (identity if unspecified)
            'quantization': <QuantizationSpec>,  # indicates that the underlying data␣
↪was quantized
        }, ...
    ]

    'video_id': str  # if this image is a frame in a video sequence, this id is shared␣
↪by all frames in that sequence.
    'timestamp': str | int  # a iso-string timestamp or an integer in flicks.
    'frame_index': int  # ordinal frame index which can be used if timestamp is unknown.
    'warp_img_to_vid': <TransformSpec>  # a transform image space to video space␣
↪(identity if unspecified), can be used for sensor alignment or video stabilization
}

TransformSpec:
    The spec can be anything coercable to a kwimage.Affine object.
    This can be an explicit affine transform matrix like:
        {'type': 'affine': 'matrix': <a-3x3 matrix>},

    But it can also be a concise dict containing one or more of these keys
        {
            'scale': <float|Tuple[float, float]>,
```

(continues on next page)

(continued from previous page)

```
            'offset': <float|Tuple[float, float]>,
            'skew': <float>,
            'theta': <float>,  # radians counter-clock-wise
        }

ChannelSpec:
    This is a string that describes the channel composition of an image.
    For the purposes of kwcoco, separate different channel names with a
    pipe ('|'). If the spec is not specified, methods may fall back on
    grayscale or rgb processing. There are special string. For instance
    'rgb' will expand into 'r|g|b'. In other applications you can "late
    fuse" inputs by separating them with a "," and "early fuse" by
    separating with a "|". Early fusion returns a solid array/tensor, late
    fusion returns separated arrays/tensors.

QuantizationSpec:
    This is a dictionary of the form:
        {
            'orig_min': <float>, # min original intensity
            'orig_max': <float>, # min original intensity
            'quant_min': <int>, # min quantized intensity
            'quant_max': <int>, # max quantized intensity
            'nodata': <int|None>,  # integer value to interpret as nan
        }

# Ground truth is specified as annotations, each belongs to a spatial
# region in an image. This must reference a subregion of the image in pixel
# coordinates. Additional non-schma properties can be specified to track
# location in other coordinate systems. Annotations can be linked over time
# by specifying track-ids.
annotation = {
    'id': int,
    'image_id': int,
    'category_id': int,

    'track_id': <int | str | uuid>  # indicates association between annotations across␣
↪images

    'bbox': [tl_x, tl_y, w, h],  # xywh format)
    'score' : float,
    'prob' : List[float],
    'weight' : float,

    'caption': str,  # a text caption for this annotation
    'keypoints' : <Keypoints | List[int] > # an accepted keypoint format
    'segmentation': <RunLengthEncoding | Polygon | MaskPath | WKT >,  # an accepted␣
↪segmentation format
}

# A dataset bundles a manifest of all aformentioned data into one structure.
dataset = {
    'categories': [category, ...],
```

(continues on next page)

(continued from previous page)

```
    'videos': [video, ...]
    'images': [image, ...]
    'annotations': [annotation, ...]
    'licenses': [],
    'info': [],
}

Polygon:
    A flattened list of xy coordinates.
    [x1, y1, x2, y2, ..., xn, yn]

    or a list of flattened list of xy coordinates if the CCs are disjoint
    [[x1, y1, x2, y2, ..., xn, yn], [x1, y1, ..., xm, ym],]

    Note: the original coco spec does not allow for holes in polygons.

    We also allow a non-standard dictionary encoding of polygons
        {'exterior': [(x1, y1)...],
         'interiors': [[(x1, y1), ...], ...]}

    TODO: Support WTK

RunLengthEncoding:
    The RLE can be in a special bytes encoding or in a binary array
    encoding. We reuse the original C functions are in [PyCocoToolsMask]_
    in ``kwimage.structs.Mask`` to provide a convinient way to abstract
    this rather esoteric bytes encoding.

    For pure python implementations see kwimage:
        Converting from an image to RLE can be done via kwimage.run_length_encoding
        Converting from RLE back to an image can be done via:
            kwimage.decode_run_length

        For compatibility with the COCO specs ensure the binary flags
        for these functions are set to true.

Keypoints:
    Annotation keypoints may also be specified in this non-standard (but
    ultimately more general) way:

    'annotations': [
        {
            'keypoints': [
                {
                    'xy': <x1, y1>,
                    'visible': <0 or 1 or 2>,
                    'keypoint_category_id': <kp_cid>,
                    'keypoint_category': <kp_name, optional>,  # this can be specified␣
↪instead of an id
                }, ...
            ]
        }, ...
```

(continues on next page)

(continued from previous page)

```
    ],
    'keypoint_categories': [{
        'name': <str>,
        'id': <int>,  # an id for this keypoint category
        'supercategory': <kp_name>  # name of coarser parent keypoint class (for␣
↪hierarchical keypoints)
        'reflection_id': <kp_cid>  # specify only if the keypoint id would be swapped␣
↪with another keypoint type
    },...
    ]

    In this scheme the "keypoints" property of each annotation (which used
    to be a list of floats) is now specified as a list of dictionaries that
    specify each keypoints location, id, and visibility explicitly. This
    allows for things like non-unique keypoints, partial keypoint
    annotations. This also removes the ordering requirement, which makes it
    simpler to keep track of each keypoints class type.

    We also have a new top-level dictionary to specify all the possible
    keypoint categories.

    TODO: Support WTK

Auxiliary Channels / Image Assets:
    For multimodal or multispectral images it is possible to specify
    auxiliary channels in an image dictionary as follows:

    {
        'id': int,
        'file_name': str,    # path to the "base" image (may be None)
        'name': str,         # a unique name for the image (must be given if file_name␣
↪is None)
        'channels': <ChannelSpec>,  # a spec code that indicates the layout of the "base
↪" image channels.
        'auxiliary': [  # information about auxiliary channels
            {
                'file_name': str,
                'channels': <ChannelSpec>
            }, ... # can have many auxiliary channels with unique specs
        ]
    }

    Note that specifing a filename / channels for the base image is not
    necessary, and mainly useful for augmenting an existing single-image
    dataset with multimodal information. Typically if an image consists of
    more than one file, all file information should be stored in the
    "auxiliary" or "assets" list.

    NEW DOCS:
        In an MSI use case you should think of the "auxiliary" list as a
        list of single file assets that are composed to make the entire
        image. Your assets might include sensed bands, computed features,
```

(continues on next page)

(continued from previous page)

```
        or quality information. For instance a list of auxiliary items may
        look like this:

        image = {
            "name": "my_msi_image",
            "width": 400,
            "height": 400,

            "video_id": 2,
            "timestamp": "2020-01-1",
            "frame_index": 5,
            "warp_img_to_vid": {"type": "affine", "scale", 1.4},

            "auxiliary": [
               {"channels": "red|green|blue": "file_name": "rgb.tif", "warp_aux_to_img":
↪{"scale": 1.0}, "height": 400, "width": 400, ...},
               ...
               {"channels": "cloudmask": "file_name": "cloudmask.tif", "warp_aux_to_img
↪": {"scale": 4.0}, "height": 100, "width": 100, ...},
               {"channels": "nir": "file_name": "nir.tif", "warp_aux_to_img": {"scale":␣
↪2.0}, "height": 200, "width": 200, ...},
               {"channels": "swir": "file_name": "swir.tif", "warp_aux_to_img": {"scale
↪": 2.0}, "height": 200, "width": 200, ...},
               {"channels": "model1_predictions:0.6": "file_name": "model1_preds.tif",
↪"warp_aux_to_img": {"scale": 8.0}, "height": 50, "width": 50, ...},
               {"channels": "model2_predictions:0.3": "file_name": "model2_preds.tif",
↪"warp_aux_to_img": {"scale": 8.0}, "height": 50, "width": 50, ...},
            ]
        }

        Note that there is no file_name or channels parameter in the image
        object itself. This pattern indicates that image is composed of
        multiple assets. One could indicate that an asset is primary by
        giving its information to the parent image, but for better STAC
        compatibility, all assets for MSI images should simply be listed
        as "auxiliary" items.


Video Sequences:
    For video sequences, we add the following video level index:

    'videos': [
        { 'id': <int>, 'name': <video_name:str> },
    ]

    Note that the videos might be given as encoded mp4/avi/etc.. files (in
    which case the name should correspond to a path) or as a series of
    frames in which case the images should be used to index the extracted
    frames and information in them.

    Then image dictionaries are augmented as follows:
```

(continues on next page)

(continued from previous page)

```
    {
        'video_id': str  # optional, if this image is a frame in a video sequence, this␣
↪id is shared by all frames in that sequence.
        'timestamp': int  # optional, timestamp (ideally in flicks), used to identify␣
↪the timestamp of the frame. Only applicable video inputs.
        'frame_index': int  # optional, ordinal frame index which can be used if␣
↪timestamp is unknown.
    }

    And annotations are augmented as follows:

    {
        'track_id': <int | str | uuid>  # optional, indicates association between␣
↪annotations across frames
    }
```

Note: The main object in this file is *CocoDataset*, which is composed of several mixin classes. See the class and method documentation for more details.

Todo:

- [ ] Use ijson to lazilly load pieces of the dataset in the background or on demand. This will give us faster access to categories / images, whereas we will always have to wait for annotations etc…
- [X] Should img_root be changed to bundle_dpath?
- [ ] Read video data, return numpy arrays (requires API for images)
- [ ] Spec for video URI, and convert to frames @ framerate function.
- [ ] Document channel spec
- [ ] Document sensor-channel spec
- [X] Add remove videos method
- **[ ] Efficiency: Make video annotations more efficient by only tracking** keyframes, provide an API to obtain a dense or interpolated annotation on an intermediate frame.
- **[ ] Efficiency: Allow each section of the kwcoco file to be written as a** separate json file. Perhaps allow genric pointer support? Might get messy.
- [ ] Reroot needs to be redesigned very carefully.

References

class kwcoco.coco_dataset.**MixinCocoDepricate**

Bases: object

These functions are marked for deprication and may be removed at any time

class kwcoco.coco_dataset.**MixinCocoAccessors**

Bases: object

TODO: better name

delayed_load(*gid*, *channels=None*, *space='image'*)

Experimental method

Parameters

- **gid** (*int*) – image id to load
- **channels** (*kwcoco.FusedChannelSpec*) – specific channels to load. if unspecified, all channels are loaded.
- **space** (*str*) – can either be "image" for loading in image space, or "video" for loading in video space.

Todo:

- **[X] Currently can only take all or none of the channels from each** base-image / auxiliary dict. For instance if the main image is r|g|b you can't just select g|b at the moment.
- **[X] The order of the channels in the delayed load should** match the requested channel order.
- [X] TODO: add nans to bands that don't exist or throw an error

---

Example

```
>>> import kwcoco
>>> gid = 1
>>> #
>>> self = kwcoco.CocoDataset.demo('vidshapes8-multispectral')
>>> delayed = self.delayed_load(gid)
>>> print('delayed = {!r}'.format(delayed))
>>> print('delayed.finalize() = {!r}'.format(delayed.finalize()))
>>> #
>>> self = kwcoco.CocoDataset.demo('shapes8')
>>> delayed = self.delayed_load(gid)
>>> print('delayed = {!r}'.format(delayed))
>>> print('delayed.finalize() = {!r}'.format(delayed.finalize()))
```

```
>>> crop = delayed.crop((slice(0, 3), slice(0, 3)))
>>> crop.finalize()
```

```
>>> # TODO: should only select the "red" channel
>>> self = kwcoco.CocoDataset.demo('shapes8')
>>> delayed = self.delayed_load(gid, channels='r')
```

```
>>> import kwcoco
>>> gid = 1
>>> #
>>> self = kwcoco.CocoDataset.demo('vidshapes8-multispectral')
>>> delayed = self.delayed_load(gid, channels='B1|B2', space='image')
>>> print('delayed = {!r}'.format(delayed))
>>> delayed = self.delayed_load(gid, channels='B1|B2|B11', space='image')
>>> print('delayed = {!r}'.format(delayed))
>>> delayed = self.delayed_load(gid, channels='B8|B1', space='video')
>>> print('delayed = {!r}'.format(delayed))
```

```
>>> delayed = self.delayed_load(gid, channels='B8|foo|bar|B1', space='video')
>>> print('delayed = {!r}'.format(delayed))
```

load_image(*gid_or_img*, *channels=None*)

Reads an image from disk and

Parameters

- **gid_or_img** (*int | dict*) – image id or image dict
- **channels** (*str | None*) – if specified, load data from auxiliary channels instead

Returns
the image

Return type
np.ndarray

Todo:

- **[ ] allow specification of multiple channels - use delayed image**
  for this.

---

get_image_fpath(*gid_or_img*, *channels=None*)

Returns the full path to the image

Parameters

- **gid_or_img** (*int | dict*) – image id or image dict
- **channels** (*str, default=None*) – if specified, return a path to data containing auxiliary channels instead

Returns
full path to the image

Return type
PathLike

get_auxiliary_fpath(*gid_or_img*, *channels*)

Returns the full path to auxiliary data for an image

Parameters

- **gid_or_img** (*int | dict*) – an image or its id
- **channels** (*str*) – the auxiliary channel to load (e.g. disparity)

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo('shapes8', aux=True)
>>> self.get_auxiliary_fpath(1, 'disparity')
```

load_annot_sample(*aid_or_ann*, *image=None*, *pad=None*)

Reads the chip of an annotation. Note this is much less efficient than using a sampler, but it doesn't require disk cache.

Maybe depricate?

Parameters

- **aid_or_int** (*int | dict*) – annot id or dict
- **image** (*ArrayLike, default=None*) – preloaded image (note: this process is inefficient unless image is specified)

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> sample = self.load_annot_sample(2, pad=100)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(sample['im'])
>>> kwplot.show_if_requested()
```

category_graph()

Construct a networkx category hierarchy

Returns

graph: a directed graph where category names are the nodes, supercategories define edges, and items in each category dict (e.g. category id) are added as node properties.

Return type

networkx.DiGraph

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> graph = self.category_graph()
>>> assert 'astronaut' in graph.nodes()
>>> assert 'keypoints' in graph.nodes['human']
```

object_categories()

Construct a consistent CategoryTree representation of object classes

Returns

category data structure

Return type

kwcoco.CategoryTree

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> classes = self.object_categories()
>>> print('classes = {}'.format(classes))
```

keypoint_categories()

Construct a consistent CategoryTree representation of keypoint classes

Returns
category data structure

Return type
kwcoco.CategoryTree

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> classes = self.keypoint_categories()
>>> print('classes = {}'.format(classes))
```

coco_image(*gid*)

Parameters
gid (*int*) – image id

Returns
kwcoco.coco_image.CocoImage

class kwcoco.coco_dataset.**MixinCocoExtras**

Bases: object

Misc functions for coco

classmethod coerce(*key*, *sqlview=False*, ***kw*)

Attempt to transform the input into the intended CocoDataset.

Parameters

- **key** – this can either be an instance of a CocoDataset, a string URI pointing to an on-disk dataset, or a special key for creating demodata.
- **sqlview** (*bool*) – If True, will return the dataset as a cached sql view, which can be quicker to load and use in some instances. Defaults to False.
- ****kw** – passed to whatever constructor is chosen (if any)

Returns
AbstractCocoDataset | kwcoco.CocoDataset | kwcoco.CocoSqlDatabase

Example

```
>>> # test coerce for various input methods
>>> import kwcoco
>>> from kwcoco.coco_sql_dataset import assert_dsets_allclose
>>> dct_dset = kwcoco.CocoDataset.coerce('special:shapes8')
>>> copy1 = kwcoco.CocoDataset.coerce(dct_dset)
>>> copy2 = kwcoco.CocoDataset.coerce(dct_dset.fpath)
>>> assert assert_dsets_allclose(dct_dset, copy1)
>>> assert assert_dsets_allclose(dct_dset, copy2)
>>> # xdoctest: +REQUIRES(module:sqlalchemy)
>>> sql_dset = dct_dset.view_sql()
>>> copy3 = kwcoco.CocoDataset.coerce(sql_dset)
>>> copy4 = kwcoco.CocoDataset.coerce(sql_dset.fpath)
>>> assert assert_dsets_allclose(dct_dset, sql_dset)
>>> assert assert_dsets_allclose(dct_dset, copy3)
>>> assert assert_dsets_allclose(dct_dset, copy4)
```

classmethod demo(*key='photos'*, ***kwargs*)

Create a toy coco dataset for testing and demo puposes

Parameters

- **key** (*str, default=photos*) – Either ‘photos’, ‘shapes’, or ‘vidshapes’. There are also special sufixes that can control behavior.

  Basic options that define which flavor of demodata to generate are: *photos*, *shapes*, and *vidshapes*. A numeric suffix e.g. *vidshapes8* can be specified to indicate the size of the generated demo dataset. There are other special suffixes that are available. See the code in this function for explicit details on what is allowed.

  TODO: better documentation for these demo datasets.

  As a quick summary: the vidshapes key is the most robust and mature demodata set, and here are several useful variants of the vidshapes key.

  (1) vidshapes8 - the 8 suffix is the number of videos in this case.

  (2) vidshapes8-multispectral - generate 8 multispectral videos.

  (3) vidshapes8-msi - msi is an alias for multispectral.

  (4) vidshapes8-frames5 - generate 8 videos with 5 frames each. (4) vidshapes2-speed0.1-frames7 - generate 2 videos with 7 frames where the objects move with with a speed of 0.1.

- ****kwargs** – if key is shapes, these arguments are passed to toydata generation. The Kwargs section of this docstring documents a subset of the available options. For full details, see `demodata_toy_dset()` and `random_video_dset()`.

Kwargs:

image_size (Tuple[int, int]): width / height size of the images

dpath (str): path to the output image directory, defaults to using
kwcoco cache dir.

aux (bool): if True generates dummy auxiliary channels

rng (int | RandomState, default=0):
random number generator or seed

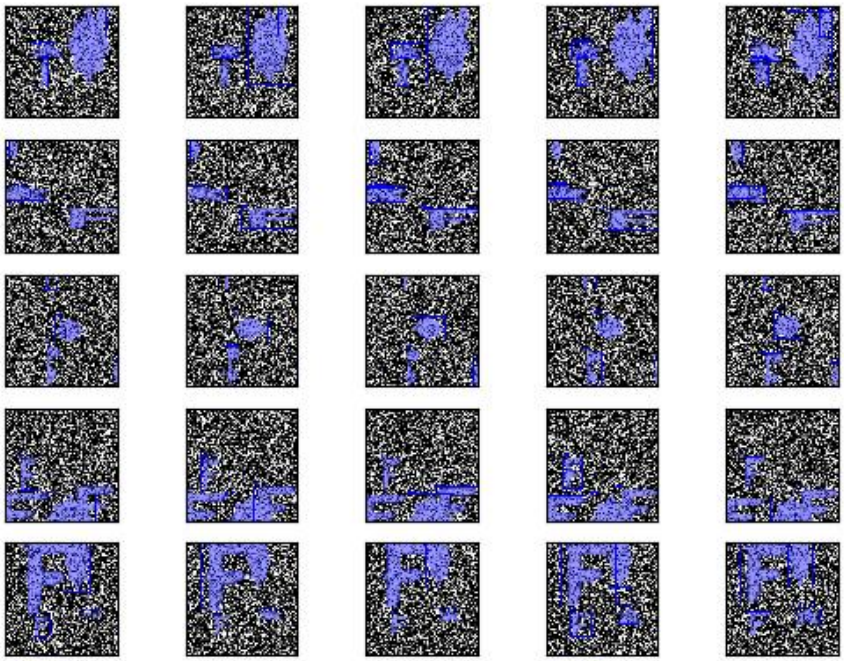
verbose (int, default=3): verbosity mode

Example

```
>>> # Basic demodata keys
>>> print(CocoDataset.demo('photos', verbose=1))
>>> print(CocoDataset.demo('shapes', verbose=1))
>>> print(CocoDataset.demo('vidshapes', verbose=1))
>>> # Varaints of demodata keys
>>> print(CocoDataset.demo('shapes8', verbose=0))
>>> print(CocoDataset.demo('shapes8-msi', verbose=0))
>>> print(CocoDataset.demo('shapes8-frames1-speed0.2-msi', verbose=0))
```

Example

```
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('vidshapes5', num_frames=5,
>>>                                verbose=0, rng=None)
>>> dset = kwcoco.CocoDataset.demo('vidshapes5', num_frames=5,
>>>                                num_tracks=4, verbose=0, rng=44)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> pnums = kwplot.PlotNums(nSubplots=len(dset.index.imgs))
>>> fnum = 1
>>> for gx, gid in enumerate(dset.index.imgs.keys()):
>>>     canvas = dset.draw_image(gid=gid)
>>>     kwplot.imshow(canvas, pnum=pnums[gx], fnum=fnum)
>>>     #dset.show_image(gid=gid, pnum=pnums[gx])
>>> kwplot.show_if_requested()
```



Example

```
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('vidshapes5-aux', num_frames=1,
>>>                                verbose=0, rng=None)
```

Example

```
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('vidshapes1-multispectral', num_frames=5,
>>>                                verbose=0, rng=None)
>>> # This is the first use-case of image names
>>> assert len(dset.index.file_name_to_img) == 0, (
>>>     'the multispectral demo case has no "base" image')
>>> assert len(dset.index.name_to_img) == len(dset.index.imgs) == 5
>>> dset.remove_images([1])
>>> assert len(dset.index.name_to_img) == len(dset.index.imgs) == 4
>>> dset.remove_videos([1])
>>> assert len(dset.index.name_to_img) == len(dset.index.imgs) == 0
```

classmethod random(*rng=None*)

Creates a random CocoDataset according to distribution parameters

Todo:

- [ ] parametarize

missing_images(*check_aux=False*, *verbose=0*)

Check for images that don't exist

Parameters

- **check_aux** (*bool, default=False*) – if specified also checks auxiliary images
- **verbose** (*int*) – verbosity level

Returns

bad indexes and paths and ids

Return type

List[Tuple[int, str, int]]

corrupted_images(*check_aux=False*, *verbose=0*)

Check for images that don't exist or can't be opened

Parameters

- **check_aux** (*bool, default=False*) – if specified also checks auxiliary images
- **verbose** (*int*) – verbosity level

Returns

bad indexes and paths and ids

Return type

List[Tuple[int, str, int]]

rename_categories(*mapper*, *rebuild=True*, *merge_policy='ignore'*)

Rename categories with a potentially coarser categorization.

Parameters

- **mapper** (*dict | Callable*) – maps old names to new names. If multiple names are mapped to the same category, those categories will be merged.
- **merge_policy** (*str*) – How to handle multiple categories that map to the same name. Can be update or ignore.

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> self.rename_categories({'astronomer': 'person',
>>>                         'astronaut': 'person',
>>>                         'mouth': 'person',
>>>                         'helmet': 'hat'})
>>> assert 'hat' in self.name_to_cat
>>> assert 'helmet' not in self.name_to_cat
>>> # Test merge case
>>> self = kwcoco.CocoDataset.demo()
>>> mapper = {
```

(continues on next page)

(continued from previous page)

```
>>>     'helmet': 'rocket',
>>>     'astronomer': 'rocket',
>>>     'human': 'rocket',
>>>     'mouth': 'helmet',
>>>     'star': 'gas'
>>> }
>>> self.rename_categories(mapper)
```

reroot(*new_root=None*, *old_prefix=None*, *new_prefix=None*, *absolute=False*, *check=True*, *safe=True*, *verbose=0*)

Modify the prefix of the image/data paths onto a new image/data root.

Parameters

- **new_root** (*str | None*) – New image root. If unspecified the current `self.bundle_dpath` is used. If old_prefix and new_prefix are unspecified, they will attempt to be determined based on the current root (which assumes the file paths exist at that root) and this new root. Defaults to None.
- **old_prefix** (*str | None*) – If specified, removes this prefix from file names. This also prevents any inferences that might be made via "new_root". Defaults to None.
- **new_prefix** (*str | None*) – If specified, adds this prefix to the file names. This also prevents any inferences that might be made via "new_root". Defaults to None.
- **absolute** (*bool*) – if True, file names are stored as absolute paths, otherwise they are relative to the new image root. Defaults to False.
- **check** (*bool*) – if True, checks that the images all exist. Defaults to True.
- **safe** (*bool*) – if True, does not overwrite values until all checks pass. Defaults to True.
- **verbose** (*int*) – verbosity level, default=0.

CommandLine

```
xdoctest -m kwcoco.coco_dataset MixinCocoExtras.reroot
```

Todo:

- [ ] Incorporate maximum ordered subtree embedding?

Example

```
>>> import kwcoco
>>> def report(dset, name):
>>>     gid = 1
>>>     abs_fpath = dset.get_image_fpath(gid)
>>>     rel_fpath = dset.index.imgs[gid]['file_name']
>>>     color = 'green' if exists(abs_fpath) else 'red'
>>>     print('strategy_name = {!r}'.format(name))
>>>     print(ub.color_text('abs_fpath = {!r}'.format(abs_fpath), color))
```

(continues on next page)

(continued from previous page)

```
>>>     print('rel_fpath = {!r}'.format(rel_fpath))
>>> dset = self = kwcoco.CocoDataset.demo()
>>> # Change base relative directory
>>> bundle_dpath = ub.expandpath('~')
>>> print('ORIG self.imgs = {!r}'.format(self.imgs))
>>> print('ORIG dset.bundle_dpath = {!r}'.format(dset.bundle_dpath))
>>> print('NEW bundle_dpath       = {!r}'.format(bundle_dpath))
>>> self.reroot(bundle_dpath)
>>> report(self, 'self')
>>> print('NEW self.imgs = {!r}'.format(self.imgs))
>>> assert self.imgs[1]['file_name'].startswith('.cache')
```

```
>>> # Use absolute paths
>>> self.reroot(absolute=True)
>>> assert self.imgs[1]['file_name'].startswith(bundle_dpath)
```

```
>>> # Switch back to relative paths
>>> self.reroot()
>>> assert self.imgs[1]['file_name'].startswith('.cache')
```

Example

```
>>> # demo with auxiliary data
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo('shapes8', aux=True)
>>> bundle_dpath = ub.expandpath('~')
>>> print(self.imgs[1]['file_name'])
>>> print(self.imgs[1]['auxiliary'][0]['file_name'])
>>> self.reroot(new_root=bundle_dpath)
>>> print(self.imgs[1]['file_name'])
>>> print(self.imgs[1]['auxiliary'][0]['file_name'])
>>> assert self.imgs[1]['file_name'].startswith('.cache')
>>> assert self.imgs[1]['auxiliary'][0]['file_name'].startswith('.cache')
```

property data_root

In the future we will deprecate data_root for bundle_dpath

property img_root

In the future we will deprecate img_root for bundle_dpath

property data_fpath

data_fpath is an alias of fpath

class kwcoco.coco_dataset.**MixinCocoObjects**

Bases: object

Expose methods to construct object lists / groups.

This is an alternative vectorized ORM-like interface to the coco dataset

annots(*aids=None*, *gid=None*, *trackid=None*)

Return vectorized annotation objects

Parameters

- **aids** (*List[int]*) – annotation ids to reference, if unspecified all annotations are returned.
- **gid** (*int*) – return all annotations that belong to this image id. mutually exclusive with other arguments.
- **trackid** (*int*) – return all annotations that belong to this track. mutually exclusive with other arguments.

Returns
vectorized annotation object

Return type
kwcoco.coco_objects1d.Annots

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> annots = self.annots()
>>> print(annots)
<Annots(num=11)>
>>> sub_annots = annots.take([1, 2, 3])
>>> print(sub_annots)
<Annots(num=3)>
>>> print(ub.repr2(sub_annots.get('bbox', None)))
[
    [350, 5, 130, 290],
    None,
    None,
]
```

images(*gids=None*, *video_id=None*, *names=None*, *vidid=None*)

Return vectorized image objects

Parameters

- **gids** (*List[int]*) – image ids to reference, if unspecified all images are returned.
- **video_id** (*int*) – returns all images that belong to this video id. mutually exclusive with *gids* arg.
- **names** (*List[str]*) – lookup images by their names.

Returns
vectorized image object

Return type
kwcoco.coco_objects1d.Images

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> images = self.images()
>>> print(images)
<Images(num=3)>
```

```
>>> self = kwcoco.CocoDataset.demo('vidshapes2')
>>> video_id = 1
>>> images = self.images(video_id=video_id)
>>> assert all(v == video_id for v in images.lookup('video_id'))
>>> print(images)
<Images(num=2)>
```

categories(*cids=None*)

Return vectorized category objects

Parameters

cids (*List[int]*) – category ids to reference, if unspecified all categories are returned.

Returns

vectorized category object

Return type

kwcoco.coco_objects1d.Categories

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> categories = self.categories()
>>> print(categories)
<Categories(num=8)>
```

videos(*vidids=None*, *names=None*)

Return vectorized video objects

Parameters

- **vidids** (*List[int]*) – video ids to reference, if unspecified all videos are returned.
- **names** (*List[str]*) – lookup videos by their name.

Returns

vectorized video object

Return type

kwcoco.coco_objects1d.Videos

Todo:

- **[ ] This conflicts with what should be the property that**
  should redirect to `index.videos`, we should resolve this somehow. E.g. all other main members of the index (anns, imgs, cats) have a toplevel dataset property, we don't have one for videos because the name we would pick conflicts with this.

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo('vidshapes2')
>>> videos = self.videos()
>>> print(videos)
>>> videos.lookup('name')
>>> videos.lookup('id')
>>> print('videos.objs = {}'.format(ub.repr2(videos.objs[0:2], nl=1)))
```

class kwcoco.coco_dataset.**MixinCocoStats**

Bases: object

Methods for getting stats about the dataset

property n_annots

The number of annotations in the dataset

property n_images

The number of images in the dataset

property n_cats

The number of categories in the dataset

property n_videos

The number of videos in the dataset

keypoint_annotation_frequency()

DEPRECATED

Example

```
>>> import kwcoco
>>> import ubelt as ub
>>> self = kwcoco.CocoDataset.demo('shapes', rng=0)
>>> hist = self.keypoint_annotation_frequency()
>>> hist = ub.odict(sorted(hist.items()))
>>> # FIXME: for whatever reason demodata generation is not determenistic when␣
↪seeded
>>> print(ub.repr2(hist))  # xdoc: +IGNORE_WANT
{
    'bot_tip': 6,
    'left_eye': 14,
    'mid_tip': 6,
    'right_eye': 14,
    'top_tip': 6,
}
```

category_annotation_frequency()

Reports the number of annotations of each category

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> hist = self.category_annotation_frequency()
>>> print(ub.repr2(hist))
{
    'astroturf': 0,
    'human': 0,
    'astronaut': 1,
    'astronomer': 1,
    'helmet': 1,
    'rocket': 1,
    'mouth': 2,
    'star': 5,
}
```

category_annotation_type_frequency()

DEPRECATED

Reports the number of annotations of each type for each category

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> hist = self.category_annotation_frequency()
>>> print(ub.repr2(hist))
```

conform(***config*)

Make the COCO file conform a stricter spec, infers attibutes where possible.

Corresponds to the `kwcoco conform` CLI tool.

KWArgs:

**config :

pycocotools_info (default=True): returns info required by pycocotools

ensure_imgsize (default=True): ensure image size is populated

legacy (default=False): if true tries to convert data structures to items compatible with the original pycocotools spec

workers (int): number of parallel jobs for IO tasks

Example

```
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('shapes8')
>>> dset.index.imgs[1].pop('width')
>>> dset.conform(legacy=True)
>>> assert 'width' in dset.index.imgs[1]
>>> assert 'area' in dset.index.anns[1]
```

validate(***config*)

Performs checks on this coco dataset.

Corresponds to the `kwcoco validate` CLI tool.

Parameters

****config** – schema (default=True): if True, validate the json-schema

unique (default=True): if True, validate unique secondary keys

missing (default=True): if True, validate registered files exist

corrupted (default=False): if True, validate data in registered files

channels (default=True): if True, validate that channels in auxiliary/asset items are all unique.

require_relative (default=False): if True, causes validation to fail if paths are non-portable, i.e. all paths must be relative to the bundle directory. if>0, paths must be relative to bundle root. if>1, paths must be inside bundle root.

img_attrs (default='warn'): if truthy, check that image attributes contain width and height entries. If 'warn', then warn if they do not exist. If 'error', then fail.

verbose (default=1): verbosity flag

fastfail (default=False): if True raise errors immediately

Returns

result containing keys -
status (bool): False if any errors occurred errors (List[str]): list of all error messages missing (List): List of any missing images corrupted (List): List of any corrupted images

Return type
dict

SeeAlso:
`_check_integrity()` - performs internal checks

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> import pytest
>>> with pytest.warns(UserWarning):
>>>     result = self.validate()
>>> assert not result['errors']
>>> assert result['warnings']
```

stats(***kwargs*)

Compute summary statistics to describe the dataset at a high level

This function corresponds to *kwcoco.cli.coco_stats*.

KWargs:

basic(bool, default=True): return basic stats' extended(bool, default=True): return extended stats' catfreq(bool, default=True): return category frequency stats' boxes(bool, default=False): return bounding box stats'

annot_attrs(bool, default=True): return annotation attribute information' image_attrs(bool, default=True): return image attribute information'

Returns

info

Return type

dict

basic_stats()

Reports number of images, annotations, and categories.

SeeAlso:

kwcoco.coco_dataset.MixinCocoStats.basic_stats() *kwcoco.coco_dataset.MixinCocoStats.extended_stats()*

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> print(ub.repr2(self.basic_stats()))
{
    'n_anns': 11,
    'n_imgs': 3,
    'n_videos': 0,
    'n_cats': 8,
}
```

```
>>> from kwcoco.demo.toydata_video import random_video_dset
>>> dset = random_video_dset(render=True, num_frames=2, num_tracks=10, rng=0)
>>> print(ub.repr2(dset.basic_stats()))
{
    'n_anns': 20,
    'n_imgs': 2,
    'n_videos': 1,
    'n_cats': 3,
}
```

extended_stats()

Reports number of images, annotations, and categories.

SeeAlso:

kwcoco.coco_dataset.MixinCocoStats.basic_stats() *kwcoco.coco_dataset.MixinCocoStats.extended_stats()*

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> print(ub.repr2(self.extended_stats()))
```

boxsize_stats(*anchors=None*, *perclass=True*, *gids=None*, *aids=None*, *verbose=0*, *clusterkw={}*, *statskw={}*)

Compute statistics about bounding box sizes.

Also computes anchor boxes using kmeans if `anchors` is specified.

Parameters

- **anchors** (*int*) – if specified also computes box anchors via KMeans clustering
- **perclass** (*bool*) – if True also computes stats for each category
- **gids** (*List[int], default=None*) – if specified only compute stats for these image ids.
- **aids** (*List[int], default=None*) – if specified only compute stats for these annotation ids.
- **verbose** (*int*) – verbosity level
- **clusterkw** (*dict*) – kwargs for `sklearn.cluster.KMeans` used if computing anchors.
- **statskw** (*dict*) – kwargs for `kwarray.stats_dict()`

Returns
Stats are returned in width-height format.

Return type
Dict[str, Dict[str, Dict | ndarray]]

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo('shapes32')
>>> infos = self.boxsize_stats(anchors=4, perclass=False)
>>> print(ub.repr2(infos, nl=-1, precision=2))
```

```
>>> infos = self.boxsize_stats(gids=[1], statskw=dict(median=True))
>>> print(ub.repr2(infos, nl=-1, precision=2))
```

find_representative_images(*gids=None*)

Find images that have a wide array of categories.

Attempt to find the fewest images that cover all categories using images that contain both a large and small number of annotations.

Parameters
gids (*None | List*) – Subset of image ids to consider when finding representative images. Uses all images if unspecified.

Returns
list of image ids determined to be representative

Return type
List

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> gids = self.find_representative_images()
>>> print('gids = {!r}'.format(gids))
>>> gids = self.find_representative_images([3])
>>> print('gids = {!r}'.format(gids))
```

```
>>> self = kwcoco.CocoDataset.demo('shapes8')
>>> gids = self.find_representative_images()
>>> print('gids = {!r}'.format(gids))
>>> valid = {7, 1}
>>> gids = self.find_representative_images(valid)
>>> assert valid.issuperset(gids)
>>> print('gids = {!r}'.format(gids))
```

class kwcoco.coco_dataset.**MixinCocoDraw**

Bases: object

Matplotlib / display functionality

imread(*gid*)

DEPRECATED: use load_image or delayed_image

Loads a particular image

draw_image(*gid*, *channels=None*)

Use kwimage to draw all annotations on an image and return the pixels as a numpy array.

Parameters

- **gid** (*int*) – image id to draw
- **channels** (*kwcoco.ChannelSpec*) – the channel to draw on

Returns
canvas

Return type
ndarray

SeeAlso
kwcoco.coco_dataset.MixinCocoDraw.draw_image() *kwcoco.coco_dataset.MixinCocoDraw.show_image()*

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo('shapes8')
>>> self.draw_image(1)
>>> # Now you can dump the annotated image to disk / whatever
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(canvas)
```

show_image(*gid=None*, *aids=None*, *aid=None*, *channels=None*, *setlim=None*, ***kwargs*)

Use matplotlib to show an image with annotations overlaid

Parameters

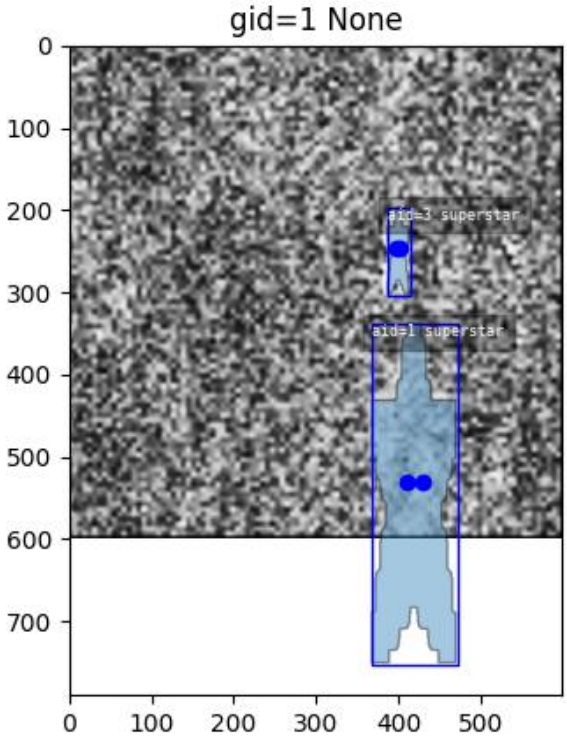
- **gid** (*int*) – image to show
- **aids** (*list*) – aids to highlight within the image
- **aid** (*int*) – a specific aid to focus on. If gid is not give, look up gid based on this aid.
- **setlim** (*None | str*) – if 'image' sets the limit to the image extent
- ****kwargs** – show_annots, show_aid, show_catname, show_kpname, show_segmentation, title, show_gid, show_filename, show_boxes,

SeeAlso

kwcoco.coco_dataset.MixinCocoDraw.draw_image() *kwcoco.coco_dataset.MixinCocoDraw.show_image()*

Example

```
>>> # xdoctest: +REQUIRES(module:kwplot)
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('vidshapes8-msi')
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> # xdoctest: -REQUIRES(--show)
>>> dset.show_image(gid=1, channels='B8')
>>> # xdoctest: +REQUIRES(--show)
>>> kwplot.show_if_requested()
```



class kwcoco.coco_dataset.**MixinCocoAddRemove**

Bases: object

Mixin functions to dynamically add / remove annotations images and categories while maintaining lookup indexes.

add_video(*name*, *id=None*, ***kw*)

Register a new video with the dataset

Parameters

- **name** (*str*) – Unique name for this video.
- **id** (*None | int*) – ADVANCED. Force using this image id.
- ****kw** – stores arbitrary key/value pairs in this new video

Returns

the video id assigned to the new video

Return type

int

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset()
>>> print('self.index.videos = {}'.format(ub.repr2(self.index.videos, nl=1)))
>>> print('self.index.imgs = {}'.format(ub.repr2(self.index.imgs, nl=1)))
>>> print('self.index.vidid_to_gids = {!r}'.format(self.index.vidid_to_gids))
```

```
>>> vidid1 = self.add_video('foo', id=3)
>>> vidid2 = self.add_video('bar')
>>> vidid3 = self.add_video('baz')
>>> print('self.index.videos = {}'.format(ub.repr2(self.index.videos, nl=1)))
>>> print('self.index.imgs = {}'.format(ub.repr2(self.index.imgs, nl=1)))
>>> print('self.index.vidid_to_gids = {!r}'.format(self.index.vidid_to_gids))
```

```
>>> gid1 = self.add_image('foo1.jpg', video_id=vidid1, frame_index=0)
>>> gid2 = self.add_image('foo2.jpg', video_id=vidid1, frame_index=1)
>>> gid3 = self.add_image('foo3.jpg', video_id=vidid1, frame_index=2)
>>> gid4 = self.add_image('bar1.jpg', video_id=vidid2, frame_index=0)
>>> print('self.index.videos = {}'.format(ub.repr2(self.index.videos, nl=1)))
>>> print('self.index.imgs = {}'.format(ub.repr2(self.index.imgs, nl=1)))
>>> print('self.index.vidid_to_gids = {!r}'.format(self.index.vidid_to_gids))
```

```
>>> self.remove_images([gid2])
>>> print('self.index.vidid_to_gids = {!r}'.format(self.index.vidid_to_gids))
```

add_image(*file_name=None*, *id=None*, ***kw*)

Register a new image with the dataset

Parameters

- **file_name** (*str | None*) – relative or absolute path to image. if not given, then "name" must be specified and we will exepect that "auxiliary" assets are eventually added.
- **id** (*None | int*) – ADVANCED. Force using this image id.
- **name** (*str*) – a unique key to identify this image
- **width** (*int*) – base width of the image
- **height** (*int*) – base height of the image
- **channels** (*ChannelSpec*) – specification of base channels. Only relevant if file_name is given.
- **auxiliary** (*List[Dict]*) – specification of auxiliary assets. See CocoImage.add_auxiliary_item for details
- **video_id** (*int*) – id of parent video, if applicable
- **frame_index** (*int*) – frame index in parent video
- **timestamp** (*number | str*) – timestamp of frame index
- ****kw** – stores arbitrary key/value pairs in this new image

Returns

the image id assigned to the new image

Return type
int

SeeAlso:
add_image() add_images() ensure_image()

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> import kwimage
>>> gname = kwimage.grab_test_image_fpath('paraview')
>>> gid = self.add_image(gname)
>>> assert self.imgs[gid]['file_name'] == gname
```

add_auxiliary_item(*gid*, *file_name=None*, *channels=None*, ***kwargs*)

Adds an auxiliary / asset item to the image dictionary.

Parameters

- **gid** (*int*) – The image id to add the auxiliary/asset item to.
- **file_name** (*str | None*) – The name of the file relative to the bundle directory. If unspecified, imdata must be given.
- **channels** (*str | kwcoco.FusedChannelSpec*) – The channel code indicating what each of the bands represents. These channels should be disjoint wrt to the existing data in this image (this is not checked).
- ****kwargs** – See `CocoImage.add_auxiliary_item()` for more details

Example

```
>>> import kwcoco
>>> dset = kwcoco.CocoDataset()
>>> gid = dset.add_image(name='my_image_name', width=200, height=200)
>>> dset.add_auxiliary_item(gid, 'path/fake_B0.tif', channels='B0',
>>>                         width=200, height=200,
>>>                         warp_aux_to_img={'scale': 1.0})
```

add_annotation(*image_id*, *category_id=None*, *bbox=NoParam*, *segmentation=NoParam*, *keypoints=NoParam*, *id=None*, ***kw*)

Register a new annotation with the dataset

Parameters

- **image_id** (*int*) – image_id the annoatation is added to.
- **category_id** (*int | None*) – category_id for the new annotaiton
- **bbox** (*list | kwimage.Boxes*) – bounding box in xywh format
- **segmentation** (*Dict | List | Any*) – keypoints in some accepted format, see `kwimage.Mask.to_coco()` and `kwimage.MultiPolygon.to_coco()`. Extended types: *MaskLike | MultiPolygonLike*.

- **keypoints** (*Any*) – keypoints in some accepted format, see `kwimage.Keypoints.to_coco()`. Extended types: *KeypointsLike*.
- **id** (*None | int*) – Force using this annotation id. Typically you should NOT specify this. A new unused id will be chosen and returned.
- ****kw** – stores arbitrary key/value pairs in this new image, Common respected key/values include but are not limited to the following: track_id (int | str): some value used to associate annotations that belong to the same "track". score : float prob : List[float] weight (float): a weight, usually used to indicate if a ground truth annotation is difficult / important. This generalizes standard "is_hard" or "ignore" attributes in other formats. caption (str): a text caption for this annotation

Returns
the annotation id assigned to the new annotation

Return type
int

SeeAlso:
kwcoco.coco_dataset.MixinCocoAddRemove.add_annotation() *kwcoco.coco_dataset.MixinCocoAddRemove.add_annotations()*

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> image_id = 1
>>> cid = 1
>>> bbox = [10, 10, 20, 20]
>>> aid = self.add_annotation(image_id, cid, bbox)
>>> assert self.anns[aid]['bbox'] == bbox
```

Example

```
>>> import kwimage
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> new_det = kwimage.Detections.random(1, segmentations=True, keypoints=True)
>>> # kwimage datastructures have methods to convert to coco recognized formats
>>> new_ann_data = list(new_det.to_coco(style='new'))[0]
>>> image_id = 1
>>> aid = self.add_annotation(image_id, **new_ann_data)
>>> # Lookup the annotation we just added
>>> ann = self.index.anns[aid]
>>> print('ann = {}'.format(ub.repr2(ann, nl=-2)))
```

Example

```
>>> # Attempt to add annot without a category or bbox
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> image_id = 1
>>> aid = self.add_annotation(image_id)
>>> assert None in self.index.cid_to_aids
```

Example

```
>>> # Attempt to add annot using various styles of kwimage structures
>>> import kwcoco
>>> import kwimage
>>> self = kwcoco.CocoDataset.demo()
>>> image_id = 1
>>> #--
>>> kw = {}
>>> kw['segmentation'] = kwimage.Polygon.random()
>>> kw['keypoints'] = kwimage.Points.random()
>>> aid = self.add_annotation(image_id, **kw)
>>> ann = self.index.anns[aid]
>>> print('ann = {}'.format(ub.repr2(ann, nl=2)))
>>> #--
>>> kw = {}
>>> kw['segmentation'] = kwimage.Mask.random()
>>> aid = self.add_annotation(image_id, **kw)
>>> ann = self.index.anns[aid]
>>> assert ann.get('segmentation', None) is not None
>>> print('ann = {}'.format(ub.repr2(ann, nl=2)))
>>> #--
>>> kw = {}
>>> kw['segmentation'] = kwimage.Mask.random().to_array_rle()
>>> aid = self.add_annotation(image_id, **kw)
>>> ann = self.index.anns[aid]
>>> assert ann.get('segmentation', None) is not None
>>> print('ann = {}'.format(ub.repr2(ann, nl=2)))
>>> #--
>>> kw = {}
>>> kw['segmentation'] = kwimage.Polygon.random().to_coco()
>>> kw['keypoints'] = kwimage.Points.random().to_coco()
>>> aid = self.add_annotation(image_id, **kw)
>>> ann = self.index.anns[aid]
>>> assert ann.get('segmentation', None) is not None
>>> assert ann.get('keypoints', None) is not None
>>> print('ann = {}'.format(ub.repr2(ann, nl=2)))
```

add_category(*name*, *supercategory=None*, *id=None*, ***kw*)

Register a new category with the dataset

Parameters

- **name** (*str*) – name of the new category

- **supercategory** (*str | None*) – parent of this category
- **id** (*int | None*) – use this category id, if it was not taken
- ****kw** – stores arbitrary key/value pairs in this new image

Returns
the category id assigned to the new category

Return type
int

SeeAlso:
kwcoco.coco_dataset.MixinCocoAddRemove.add_category() *kwcoco.coco_dataset.MixinCocoAddRemove.ensure_category()*

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> prev_n_cats = self.n_cats
>>> cid = self.add_category('dog', supercategory='object')
>>> assert self.cats[cid]['name'] == 'dog'
>>> assert self.n_cats == prev_n_cats + 1
>>> import pytest
>>> with pytest.raises(ValueError):
>>>     self.add_category('dog', supercategory='object')
```

ensure_image(*file_name*, *id=None*, ***kw*)

Register an image if it is new or returns an existing id.

Like *kwcoco.coco_dataset.MixinCocoAddRemove.add_image()*, but returns the existing image id if it already exists instead of failing. In this case all metadata is ignored.

Parameters

- **file_name** (*str*) – relative or absolute path to image
- **id** (*None | int*) – ADVANCED. Force using this image id.
- ****kw** – stores arbitrary key/value pairs in this new image

Returns
the existing or new image id

Return type
int

SeeAlso:
kwcoco.coco_dataset.MixinCocoAddRemove.add_image() *kwcoco.coco_dataset.MixinCocoAddRemove.add_images()* *kwcoco.coco_dataset.MixinCocoAddRemove.ensure_image()*

ensure_category(*name*, *supercategory=None*, *id=None*, ***kw*)

Register a category if it is new or returns an existing id.

Like *kwcoco.coco_dataset.MixinCocoAddRemove.add_category()*, but returns the existing category id if it already exists instead of failing. In this case all metadata is ignored.

Returns
the existing or new category id

Return type
int

SeeAlso:
kwcoco.coco_dataset.MixinCocoAddRemove.add_category() *kwcoco.coco_dataset.MixinCocoAddRemove.ensure_category()*

add_annotations(*anns*)

Faster less-safe multi-item alternative to add_annotation.

We assume the annotations are well formatted in kwcoco compliant dictionaries, including the "id" field. No validation checks are made when calling this function.

Parameters
anns (*List[Dict]*) – list of annotation dictionaries

SeeAlso:
add_annotation() *add_annotations()*

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> anns = [self.anns[aid] for aid in [2, 3, 5, 7]]
>>> self.remove_annotations(anns)
>>> assert self.n_annots == 7 and self._check_index()
>>> self.add_annotations(anns)
>>> assert self.n_annots == 11 and self._check_index()
```

add_images(*imgs*)

Faster less-safe multi-item alternative

We assume the images are well formatted in kwcoco compliant dictionaries, including the "id" field. No validation checks are made when calling this function.

Note: THIS FUNCTION WAS DESIGNED FOR SPEED, AS SUCH IT DOES NOT CHECK IF THE IMAGE-IDs or FILE_NAMES ARE DUPLICATED AND WILL BLINDLY ADD DATA EVEN IF IT IS BAD. THE SINGLE IMAGE VERSION IS SLOWER BUT SAFER.

Parameters
imgs (*List[Dict]*) – list of image dictionaries

SeeAlso:
kwcoco.coco_dataset.MixinCocoAddRemove.add_image() *kwcoco.coco_dataset.MixinCocoAddRemove.add_images()* *kwcoco.coco_dataset.MixinCocoAddRemove.ensure_image()*

Example

```
>>> import kwcoco
>>> imgs = kwcoco.CocoDataset.demo().dataset['images']
>>> self = kwcoco.CocoDataset()
>>> self.add_images(imgs)
>>> assert self.n_images == 3 and self._check_index()
```

clear_images()

Removes all images and annotations (but not categories)

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> self.clear_images()
>>> print(ub.repr2(self.basic_stats(), nobr=1, nl=0, si=1))
n_anns: 0, n_imgs: 0, n_videos: 0, n_cats: 8
```

clear_annotations()

Removes all annotations (but not images and categories)

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> self.clear_annotations()
>>> print(ub.repr2(self.basic_stats(), nobr=1, nl=0, si=1))
n_anns: 0, n_imgs: 3, n_videos: 0, n_cats: 8
```

remove_annotation(*aid_or_ann*)

Remove a single annotation from the dataset

If you have multiple annotations to remove its more efficient to remove them in batch with *kwcoco.coco_dataset.MixinCocoAddRemove.remove_annotations()*

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> aids_or_anns = [self.anns[2], 3, 4, self.anns[1]]
>>> self.remove_annotations(aids_or_anns)
>>> assert len(self.dataset['annotations']) == 7
>>> self._check_index()
```

remove_annotations(*aids_or_anns*, *verbose=0*, *safe=True*)

Remove multiple annotations from the dataset.

Parameters

- **anns_or_aids** (*List*) – list of annotation dicts or ids

- **safe** (*bool, default=True*) – if True, we perform checks to remove duplicates and non-existing identifiers.

Returns
num_removed: information on the number of items removed

Return type
Dict

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> prev_n_annots = self.n_annots
>>> aids_or_anns = [self.anns[2], 3, 4, self.anns[1]]
>>> self.remove_annotations(aids_or_anns)  # xdoc: +IGNORE_WANT
{'annotations': 4}
>>> assert len(self.dataset['annotations']) == prev_n_annots - 4
>>> self._check_index()
```

remove_categories(*cat_identifiers*, *keep_annots=False*, *verbose=0*, *safe=True*)

Remove categories and all annotations in those categories.

Currently does not change any hierarchy information

Parameters

- **cat_identifiers** (*List*) – list of category dicts, names, or ids
- **keep_annots** (*bool, default=False*) – if True, keeps annotations, but removes category labels.
- **safe** (*bool, default=True*) – if True, we perform checks to remove duplicates and non-existing identifiers.

Returns
num_removed: information on the number of items removed

Return type
Dict

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> cat_identifiers = [self.cats[1], 'rocket', 3]
>>> self.remove_categories(cat_identifiers)
>>> assert len(self.dataset['categories']) == 5
>>> self._check_index()
```

remove_images(*gids_or_imgs*, *verbose=0*, *safe=True*)

Remove images and any annotations contained by them

Parameters

- **gids_or_imgs** (*List*) – list of image dicts, names, or ids

- **safe** (*bool, default=True*) – if True, we perform checks to remove duplicates and non-existing identifiers.

Returns
num_removed: information on the number of items removed

Return type
Dict

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> assert len(self.dataset['images']) == 3
>>> gids_or_imgs = [self.imgs[2], 'astro.png']
>>> self.remove_images(gids_or_imgs)  # xdoc: +IGNORE_WANT
{'annotations': 11, 'images': 2}
>>> assert len(self.dataset['images']) == 1
>>> self._check_index()
>>> gids_or_imgs = [3]
>>> self.remove_images(gids_or_imgs)
>>> assert len(self.dataset['images']) == 0
>>> self._check_index()
```

remove_videos(*vidids_or_videos*, *verbose=0*, *safe=True*)

Remove videos and any images / annotations contained by them

Parameters

- **vidids_or_videos** (*List*) – list of video dicts, names, or ids
- **safe** (*bool, default=True*) – if True, we perform checks to remove duplicates and non-existing identifiers.

Returns
num_removed: information on the number of items removed

Return type
Dict

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo('vidshapes8')
>>> assert len(self.dataset['videos']) == 8
>>> vidids_or_videos = [self.dataset['videos'][0]['id']]
>>> self.remove_videos(vidids_or_videos)  # xdoc: +IGNORE_WANT
{'annotations': 4, 'images': 2, 'videos': 1}
>>> assert len(self.dataset['videos']) == 7
>>> self._check_index()
```

remove_annotation_keypoints(*kp_identifiers*)

Removes all keypoints with a particular category

Parameters
kp_identifiers (*List*) – list of keypoint category dicts, names, or ids

Returns
num_removed: information on the number of items removed

Return type
Dict

remove_keypoint_categories(*kp_identifiers*)

Removes all keypoints of a particular category as well as all annotation keypoints with those ids.

Parameters
kp_identifiers (*List*) – list of keypoint category dicts, names, or ids

Returns
num_removed: information on the number of items removed

Return type
Dict

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo('shapes', rng=0)
>>> kp_identifiers = ['left_eye', 'mid_tip']
>>> remove_info = self.remove_keypoint_categories(kp_identifiers)
>>> print('remove_info = {!r}'.format(remove_info))
>>> # FIXME: for whatever reason demodata generation is not determenistic when␣
↪seeded
>>> # assert remove_info == {'keypoint_categories': 2, 'annotation_keypoints': 16,
↪'reflection_ids': 1}
>>> assert self._resolve_to_kpcat('right_eye')['reflection_id'] is None
```

set_annotation_category(*aid_or_ann*, *cid_or_cat*)

Sets the category of a single annotation

Parameters

- **aid_or_ann** (*dict | int*) – annotation dict or id
- **cid_or_cat** (*dict | int*) – category dict or id

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> old_freq = self.category_annotation_frequency()
>>> aid_or_ann = aid = 2
>>> cid_or_cat = new_cid = self.ensure_category('kitten')
>>> self.set_annotation_category(aid, new_cid)
>>> new_freq = self.category_annotation_frequency()
>>> print('new_freq = {}'.format(ub.repr2(new_freq, nl=1)))
>>> print('old_freq = {}'.format(ub.repr2(old_freq, nl=1)))
>>> assert sum(new_freq.values()) == sum(old_freq.values())
>>> assert new_freq['kitten'] == 1
```

class kwcoco.coco_dataset.**CocoIndex**

Bases: object

Fast lookup index for the COCO dataset with dynamic modification

Variables

- **imgs** (*Dict[int, dict]*) – mapping between image ids and the image dictionaries
- **anns** (*Dict[int, dict]*) – mapping between annotation ids and the annotation dictionaries
- **cats** (*Dict[int, dict]*) – mapping between category ids and the category dictionaries
- **kpcats** (*Dict[int, dict]*) – mapping between keypoint category ids and keypoint category dictionaries
- **gid_to_aids** (*Dict[int, List[int]]*) – mapping between an image-id and annotation-ids that belong to it
- **cid_to_aids** (*Dict[int, List[int]]*) – mapping between an category-id and annotation-ids that belong to it
- **cid_to_gids** (*Dict[int, List[int]]*) – mapping between an category-id and image-ids that contain at least one annotation with this cateogry id.
- **trackid_to_aids** (*Dict[int, List[int]]*) – mapping between a track-id and annotation-ids that belong to it
- **vidid_to_gids** (*Dict[int, List[int]]*) – mapping between an video-id and images-ids that belong to it
- **name_to_video** (*Dict[str, dict]*) – mapping between a video name and the video dictionary.
- **name_to_cat** (*Dict[str, dict]*) – mapping between a category name and the category dictionary.
- **name_to_img** (*Dict[str, dict]*) – mapping between a image name and the image dictionary.
- **file_name_to_img** (*Dict[str, dict]*) – mapping between a image file_name and the image dictionary.

property **cid_to_gids**

Example:

```
>>> import kwcoco
>>> self = dset = kwcoco.CocoDataset()
>>> self.index.cid_to_gids
```

clear()

build(*parent*)

Build all id-to-obj reverse indexes from scratch.

Parameters

parent (*kwcoco.CocoDataset*) – the dataset to index

Notation:

aid - Annotation ID gid - imaGe ID cid - Category ID vidid - Video ID

Example

```
>>> import kwcoco
>>> parent = kwcoco.CocoDataset.demo('vidshapes1', num_frames=4, rng=1)
>>> index = parent.index
>>> index.build(parent)
```

class kwcoco.coco_dataset.**MixinCocoIndex**

Bases: object

Give the dataset top level access to index attributes

property anns

property imgs

property cats

property gid_to_aids

property cid_to_aids

property name_to_cat

class kwcoco.coco_dataset.**CocoDataset**(*data=None*, *tag=None*, *bundle_dpath=None*, *img_root=None*, *fname=None*, *autobuild=True*)

Bases: *AbstractCocoDataset*, *MixinCocoAddRemove*, *MixinCocoStats*, *MixinCocoObjects*, *MixinCocoDraw*, *MixinCocoAccessors*, *MixinCocoExtras*, *MixinCocoIndex*, *MixinCocoDepricate*, NiceRepr

The main coco dataset class with a json dataset backend.

Variables

- **dataset** (*Dict*) – raw json data structure. This is the base dictionary that contains {'annotations': List, 'images': List, 'categories': List}
- **index** (CocoIndex) – an efficient lookup index into the coco data structure. The index defines its own attributes like anns, cats, imgs, gid_to_aids, file_name_to_img, etc. See *CocoIndex* for more details on which attributes are available.
- **fpath** (*PathLike | None*) – if known, this stores the filepath the dataset was loaded from
- **tag** (*str*) – A tag indicating the name of the dataset.
- **bundle_dpath** (*PathLike | None*) – If known, this is the root path that all image file names are relative to. This can also be manually overwritten by the user.
- **hashid** (*str | None*) – If computed, this will be a hash uniquely identifing the dataset. To ensure this is computed see kwcoco.coco_dataset.MixinCocoExtras._build_hashid().

References

http://cocodataset.org/#format http://cocodataset.org/#download

CommandLine

```
python -m kwcoco.coco_dataset CocoDataset --show
```

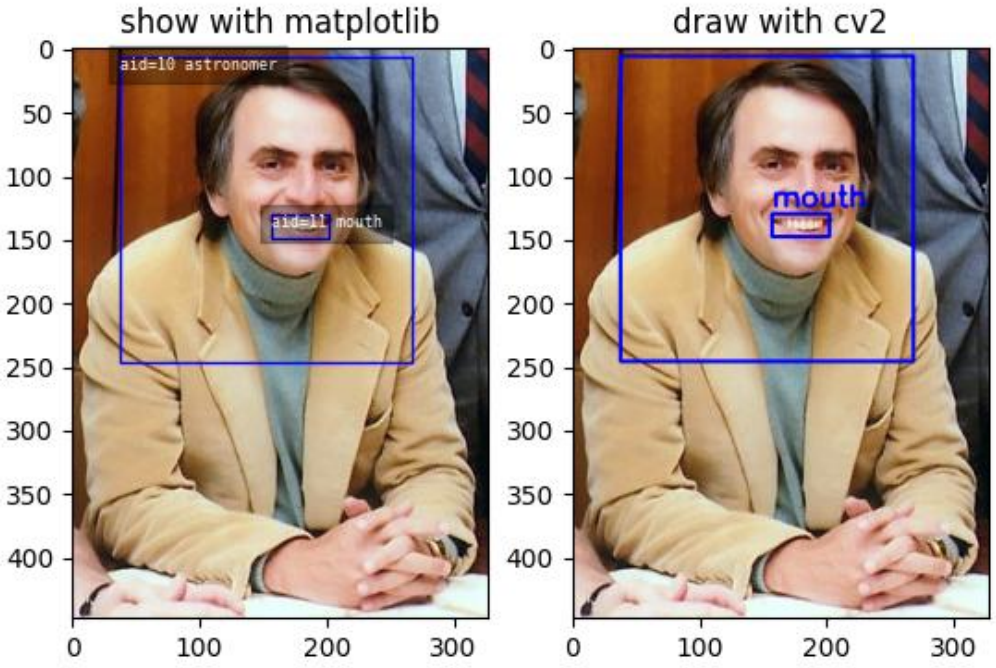
Example

```
>>> from kwcoco.coco_dataset import demo_coco_data
>>> import kwcoco
>>> import ubelt as ub
>>> # Returns a coco json structure
>>> dataset = demo_coco_data()
>>> # Pass the coco json structure to the API
>>> self = kwcoco.CocoDataset(dataset, tag='demo')
>>> # Now you can access the data using the index and helper methods
>>> #
>>> # Start by looking up an image by it's COCO id.
>>> image_id = 1
>>> img = self.index.imgs[image_id]
>>> print(ub.repr2(img, nl=1, sort=1))
{
    'file_name': 'astro.png',
    'id': 1,
    'url': 'https://i.imgur.com/KXhKM72.png',
}
>>> #
>>> # Use the (gid_to_aids) index to lookup annotations in the iamge
>>> annotation_id = sorted(self.index.gid_to_aids[image_id])[0]
>>> ann = self.index.anns[annotation_id]
>>> print(ub.repr2(ub.dict_diff(ann, {'segmentation'}), nl=1))
{
    'bbox': [10, 10, 360, 490],
    'category_id': 1,
    'id': 1,
    'image_id': 1,
    'keypoints': [247, 101, 2, 202, 100, 2],
}
>>> #
>>> # Use annotation category id to look up that information
>>> category_id = ann['category_id']
>>> cat = self.index.cats[category_id]
>>> print('cat = {}'.format(ub.repr2(cat, nl=1, sort=1)))
cat = {
    'id': 1,
    'name': 'astronaut',
    'supercategory': 'human',
}
>>> #
```

(continues on next page)

(continued from previous page)

```
>>> # Now play with some helper functions, like extended statistics
>>> extended_stats = self.extended_stats()
>>> # xdoctest: +IGNORE_WANT
>>> print('extended_stats = {}'.format(ub.repr2(extended_stats, nl=1, precision=2,␣
↪sort=1)))
extended_stats = {
    'annots_per_img': {'mean': 3.67, 'std': 3.86, 'min': 0.00, 'max': 9.00, 'nMin':␣
↪1, 'nMax': 1, 'shape': (3,)},
    'imgs_per_cat': {'mean': 0.88, 'std': 0.60, 'min': 0.00, 'max': 2.00, 'nMin': 2,
↪ 'nMax': 1, 'shape': (8,)},
    'cats_per_img': {'mean': 2.33, 'std': 2.05, 'min': 0.00, 'max': 5.00, 'nMin': 1,
↪ 'nMax': 1, 'shape': (3,)},
    'annots_per_cat': {'mean': 1.38, 'std': 1.49, 'min': 0.00, 'max': 5.00, 'nMin':␣
↪2, 'nMax': 1, 'shape': (8,)},
    'imgs_per_video': {'empty_list': True},
}
>>> # You can "draw" a raster of the annotated image with cv2
>>> canvas = self.draw_image(2)
>>> # Or if you have matplotlib you can "show" the image with mpl objects
>>> # xdoctest: +REQUIRES(--show)
>>> from matplotlib import pyplot as plt
>>> fig = plt.figure()
>>> ax1 = fig.add_subplot(1, 2, 1)
>>> self.show_image(gid=2)
>>> ax2 = fig.add_subplot(1, 2, 2)
>>> ax2.imshow(canvas)
>>> ax1.set_title('show with matplotlib')
>>> ax2.set_title('draw with cv2')
>>> plt.show()
```



property fpath

In the future we will deprecate img_root for bundle_dpath

classmethod from_data(*data*, *bundle_dpath=None*, *img_root=None*)

Constructor from a json dictionary

classmethod from_image_paths(*gpaths*, *bundle_dpath=None*, *img_root=None*)

Constructor from a list of images paths.

This is a convinience method.

Parameters

gpaths (*List[str]*) – list of image paths

Example

```
>>> import kwcoco
>>> coco_dset = kwcoco.CocoDataset.from_image_paths(['a.png', 'b.png'])
>>> assert coco_dset.n_images == 2
```

classmethod from_coco_paths(*fpaths*, *max_workers=0*, *verbose=1*, *mode='thread'*, *union='try'*)

Constructor from multiple coco file paths.

Loads multiple coco datasets and unions the result

Note: if the union operation fails, the list of individually loaded files is returned instead.

Parameters

- **fpaths** (*List[str]*) – list of paths to multiple coco files to be loaded and unioned.
- **max_workers** (*int, default=0*) – number of worker threads / processes
- **verbose** (*int*) – verbosity level
- **mode** (*str*) – thread, process, or serial
- **union** (*str | bool, default='try'*) – If True, unions the result datasets after loading. If False, just returns the result list. If 'try', then try to preform the union, but return the result list if it fails.

copy()

Deep copies this object

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> new = self.copy()
>>> assert new.imgs[1] is new.dataset['images'][0]
>>> assert new.imgs[1] == self.dataset['images'][0]
>>> assert new.imgs[1] is not self.dataset['images'][0]
```

dumps(*indent=None*, *newlines=False*)

Writes the dataset out to the json format

Parameters

newlines (*bool*) – if True, each annotation, image, category gets its own line

Note:

Using newlines=True is similar to:
print(ub.repr2(dset.dataset, nl=2, trailsep=False)) However, the above may not output valid json if it contains ndarrays.

Example

```
>>> import kwcoco
>>> import json
>>> self = kwcoco.CocoDataset.demo()
>>> text = self.dumps(newlines=True)
>>> print(text)
>>> self2 = kwcoco.CocoDataset(json.loads(text), tag='demo2')
>>> assert self2.dataset == self.dataset
>>> assert self2.dataset is not self.dataset
```

```
>>> text = self.dumps(newlines=True)
>>> print(text)
>>> self2 = kwcoco.CocoDataset(json.loads(text), tag='demo2')
>>> assert self2.dataset == self.dataset
>>> assert self2.dataset is not self.dataset
```

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.coerce('vidshapes1-msi-multisensor', verbose=3)
>>> self.remove_annotations(self.annots())
>>> text = self.dumps(newlines=True, indent='  ')
>>> print(text)
```

dump(*file*, *indent=None*, *newlines=False*, *temp_file=True*)

Writes the dataset out to the json format

Parameters

- **file** (*PathLike | IO*) – Where to write the data. Can either be a path to a file or an open file pointer / stream.
- **newlines** (*bool*) – if True, each annotation, image, category gets its own line.
- **temp_file** (*bool | str, default=True*) – Argument to `safer.open()`. Ignored if `file` is not a PathLike object.

Example

```
>>> import tempfile
>>> import json
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> file = tempfile.NamedTemporaryFile('w')
>>> self.dump(file)
>>> file.seek(0)
>>> text = open(file.name, 'r').read()
>>> print(text)
>>> file.seek(0)
>>> dataset = json.load(open(file.name, 'r'))
>>> self2 = kwcoco.CocoDataset(dataset, tag='demo2')
>>> assert self2.dataset == self.dataset
>>> assert self2.dataset is not self.dataset
```

```
>>> file = tempfile.NamedTemporaryFile('w')
>>> self.dump(file, newlines=True)
>>> file.seek(0)
>>> text = open(file.name, 'r').read()
>>> print(text)
>>> file.seek(0)
>>> dataset = json.load(open(file.name, 'r'))
>>> self2 = kwcoco.CocoDataset(dataset, tag='demo2')
```

(continues on next page)

(continued from previous page)

```
>>> assert self2.dataset == self.dataset
>>> assert self2.dataset is not self.dataset
```

union(*, *disjoint_tracks=True*, ***kwargs*)

Merges multiple *CocoDataset* items into one. Names and associations are retained, but ids may be different.

Parameters

- ***others** – a series of CocoDatasets that we will merge. Note, if called as an instance method, the "self" instance will be the first item in the "others" list. But if called like a classmethod, "others" will be empty by default.
- **disjoint_tracks** (*bool, default=True*) – if True, we will assume track-ids are disjoint and if two datasets share the same track-id, we will disambiguate them. Otherwise they will be copied over as-is.
- ****kwargs** – constructor options for the new merged CocoDataset

Returns

a new merged coco dataset

Return type

kwcoco.CocoDataset

CommandLine

```
xdoctest -m kwcoco.coco_dataset CocoDataset.union
```

Example

```
>>> import kwcoco
>>> # Test union works with different keypoint categories
>>> dset1 = kwcoco.CocoDataset.demo('shapes1')
>>> dset2 = kwcoco.CocoDataset.demo('shapes2')
>>> dset1.remove_keypoint_categories(['bot_tip', 'mid_tip', 'right_eye'])
>>> dset2.remove_keypoint_categories(['top_tip', 'left_eye'])
>>> dset_12a = kwcoco.CocoDataset.union(dset1, dset2)
>>> dset_12b = dset1.union(dset2)
>>> dset_21 = dset2.union(dset1)
>>> def add_hist(h1, h2):
>>>     return {k: h1.get(k, 0) + h2.get(k, 0) for k in set(h1) | set(h2)}
>>> kpfreq1 = dset1.keypoint_annotation_frequency()
>>> kpfreq2 = dset2.keypoint_annotation_frequency()
>>> kpfreq_want = add_hist(kpfreq1, kpfreq2)
>>> kpfreq_got1 = dset_12a.keypoint_annotation_frequency()
>>> kpfreq_got2 = dset_12b.keypoint_annotation_frequency()
>>> assert kpfreq_want == kpfreq_got1
>>> assert kpfreq_want == kpfreq_got2
```

```
>>> # Test disjoint gid datasets
>>> dset1 = kwcoco.CocoDataset.demo('shapes3')
```

(continues on next page)

(continued from previous page)

```
>>> for new_gid, img in enumerate(dset1.dataset['images'], start=10):
>>>     for aid in dset1.gid_to_aids[img['id']]:
>>>         dset1.anns[aid]['image_id'] = new_gid
>>>     img['id'] = new_gid
>>> dset1.index.clear()
>>> dset1._build_index()
>>> # ------
>>> dset2 = kwcoco.CocoDataset.demo('shapes2')
>>> for new_gid, img in enumerate(dset2.dataset['images'], start=100):
>>>     for aid in dset2.gid_to_aids[img['id']]:
>>>         dset2.anns[aid]['image_id'] = new_gid
>>>     img['id'] = new_gid
>>> dset1.index.clear()
>>> dset2._build_index()
>>> others = [dset1, dset2]
>>> merged = kwcoco.CocoDataset.union(*others)
>>> print('merged = {!r}'.format(merged))
>>> print('merged.imgs = {}'.format(ub.repr2(merged.imgs, nl=1)))
>>> assert set(merged.imgs) & set([10, 11, 12, 100, 101]) == set(merged.imgs)
```

```
>>> # Test data is not preserved
>>> dset2 = kwcoco.CocoDataset.demo('shapes2')
>>> dset1 = kwcoco.CocoDataset.demo('shapes3')
>>> others = (dset1, dset2)
>>> cls = self = kwcoco.CocoDataset
>>> merged = cls.union(*others)
>>> print('merged = {!r}'.format(merged))
>>> print('merged.imgs = {}'.format(ub.repr2(merged.imgs, nl=1)))
>>> assert set(merged.imgs) & set([1, 2, 3, 4, 5]) == set(merged.imgs)
```

```
>>> # Test track-ids are mapped correctly
>>> dset1 = kwcoco.CocoDataset.demo('vidshapes1')
>>> dset2 = kwcoco.CocoDataset.demo('vidshapes2')
>>> dset3 = kwcoco.CocoDataset.demo('vidshapes3')
>>> others = (dset1, dset2, dset3)
>>> for dset in others:
>>>     [a.pop('segmentation', None) for a in dset.index.anns.values()]
>>>     [a.pop('keypoints', None) for a in dset.index.anns.values()]
>>> cls = self = kwcoco.CocoDataset
>>> merged = cls.union(*others, disjoint_tracks=1)
>>> print('dset1.anns = {}'.format(ub.repr2(dset1.anns, nl=1)))
>>> print('dset2.anns = {}'.format(ub.repr2(dset2.anns, nl=1)))
>>> print('dset3.anns = {}'.format(ub.repr2(dset3.anns, nl=1)))
>>> print('merged.anns = {}'.format(ub.repr2(merged.anns, nl=1)))
```

Example

```
>>> import kwcoco
>>> # Test empty union
>>> empty_union = kwcoco.CocoDataset.union()
>>> assert len(empty_union.index.imgs) == 0
```

Todo:

- [ ] are supercategories broken?
- [ ] reuse image ids where possible
- [ ] reuse annotation / category ids where possible
- [X] handle case where no inputs are given
- [x] disambiguate track-ids
- [x] disambiguate video-ids

---

subset(*gids*, *copy=False*, *autobuild=True*)

Return a subset of the larger coco dataset by specifying which images to port. All annotations in those images will be taken.

Parameters

- **gids** (*List[int]*) – image-ids to copy into a new dataset
- **copy** (*bool, default=False*) – if True, makes a deep copy of all nested attributes, otherwise makes a shallow copy.
- **autobuild** (*bool, default=True*) – if True will automatically build the fast lookup index.

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> gids = [1, 3]
>>> sub_dset = self.subset(gids)
>>> assert len(self.index.gid_to_aids) == 3
>>> assert len(sub_dset.gid_to_aids) == 2
```

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo('vidshapes2')
>>> gids = [1, 2]
>>> sub_dset = self.subset(gids, copy=True)
>>> assert len(sub_dset.index.videos) == 1
>>> assert len(self.index.videos) == 2
```

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> sub1 = self.subset([1])
>>> sub2 = self.subset([2])
>>> sub3 = self.subset([3])
>>> others = [sub1, sub2, sub3]
>>> rejoined = kwcoco.CocoDataset.union(*others)
>>> assert len(sub1.anns) == 9
>>> assert len(sub2.anns) == 2
>>> assert len(sub3.anns) == 0
>>> assert rejoined.basic_stats() == self.basic_stats()
```

view_sql(*force_rewrite=False*, *memory=False*)

Create a cached SQL interface to this dataset suitable for large scale multiprocessing use cases.

Parameters

- **force_rewrite** (*bool, default=False*) – if True, forces an update to any existing cache file on disk
- **memory** (*bool, default=False*) – if True, the database is constructed in memory.

Note: This view cache is experimental and currently depends on the timestamp of the file pointed to by `self.fpath`. In other words dont use this on in-memory datasets.

---

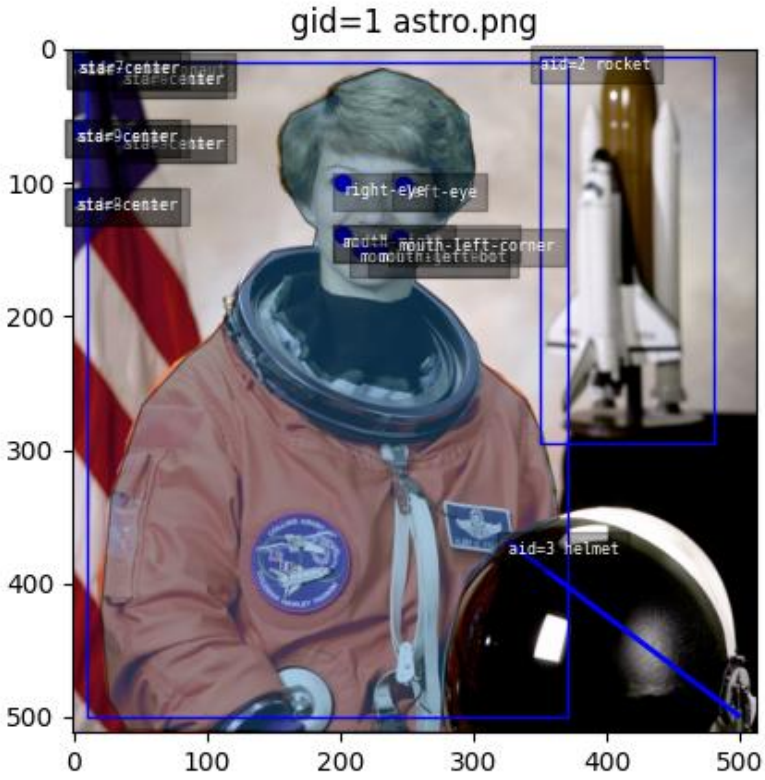
kwcoco.coco_dataset.**demo_coco_data**()

Simple data for testing.

This contains several non-standard fields, which help ensure robustness of functions tested with this data. For more compliant demodata see the `kwcoco.demodata` submodule

Example

```
>>> # xdoctest: +REQUIRES(--show)
>>> import kwcoco
>>> from kwcoco.coco_dataset import demo_coco_data
>>> dataset = demo_coco_data()
>>> self = kwcoco.CocoDataset(dataset, tag='demo')
>>> import kwplot
>>> kwplot.autompl()
>>> self.show_image(gid=1)
>>> kwplot.show_if_requested()
```



2.1.2.5 kwcoco.coco_evaluator module

Evaluates a predicted coco dataset against a truth coco dataset.

The components in this module work programatically or as a command line script.

Todo:

- **[ ] does evaluate return one result or multiple results**
  based on different configurations?
- [ ] max_dets - TODO: in original pycocoutils but not here
- [ ] Flag that allows for polygon instead of bounding box overlap
- **[ ] How do we note what iou_thresh and area-range were in**
  the result plots?

CommandLine

```
xdoctest -m kwcoco.coco_evaluator __doc__:0 --vd --slow
```

Example

```
>>> from kwcoco.coco_evaluator import *  # NOQA
>>> from kwcoco.coco_evaluator import CocoEvaluator
>>> import kwcoco
>>> # note: increase the number of images for better looking metrics
>>> true_dset = kwcoco.CocoDataset.demo('shapes8')
>>> from kwcoco.demo.perterb import perterb_coco
>>> kwargs = {
>>>     'box_noise': 0.5,
>>>     'n_fp': (0, 10),
>>>     'n_fn': (0, 10),
>>>     'with_probs': True,
>>> }
>>> pred_dset = perterb_coco(true_dset, **kwargs)
>>> print('true_dset = {!r}'.format(true_dset))
>>> print('pred_dset = {!r}'.format(pred_dset))
>>> config = {
>>>     'true_dataset': true_dset,
>>>     'pred_dataset': pred_dset,
>>>     'area_range': ['all', 'small'],
>>>     'iou_thresh': [0.3, 0.95],
>>> }
>>> coco_eval = CocoEvaluator(config)
>>> results = coco_eval.evaluate()
>>> # Now we can draw / serialize the results as we please
>>> dpath = ub.ensure_app_cache_dir('kwcoco/tests/test_out_dpath')
>>> results_fpath = join(dpath, 'metrics.json')
>>> print('results_fpath = {!r}'.format(results_fpath))
>>> results.dump(results_fpath, indent='    ')
>>> measures = results['area_range=all,iou_thresh=0.3'].nocls_measures
>>> import pandas as pd
>>> print(pd.DataFrame(ub.dict_isect(
>>>     measures, ['f1', 'g1', 'mcc', 'thresholds',
>>>                'ppv', 'tpr', 'tnr', 'npv', 'fpr',
>>>                'tp_count', 'fp_count',
>>>                'tn_count', 'fn_count'])).iloc[::100])
>>> # xdoctest: +REQUIRES(module:kwplot)
>>> # xdoctest: +REQUIRES(--slow)
>>> results.dump_figures(dpath)
>>> print('dpath = {!r}'.format(dpath))
>>> # xdoctest: +REQUIRES(--vd)
>>> if ub.argflag('--vd') or 1:
>>>     import xdev
>>>     xdev.view_directory(dpath)
```

class kwcoco.coco_evaluator.**CocoEvalConfig**(*data=None*, *default=None*, *cmdline=False*)

Bases: Config

Evaluate and score predicted versus truth detections / classifications in a COCO dataset

default = {'ap_method': <Value(None: 'pycocotools')>, 'area_range': <Value(None: ['all'])>, 'assign_workers': <Value(None: 8)>, 'classes_of_interest': <Value(<class 'list'>: None)>, 'compat': <Value(None: 'mutex')>, 'force_pycocoutils': <Value(None: False)>, 'fp_cutoff': <Value(None: inf)>, 'ignore_classes': <Value(<class 'list'>: None)>, 'implicit_ignore_classes': <Value(None: ['ignore'])>, 'implicit_negative_classes': <Value(None: ['background'])>, 'iou_bias': <Value(None: 1)>, 'iou_thresh': <Value(None: 0.5)>, 'load_workers': <Value(None: 0)>, 'max_dets': <Value(None: inf)>, 'monotonic_ppv': <Value(None: True)>, 'ovthresh': <Value(None: None)>, 'pred_dataset': <Value(<class 'str'>: None)>, 'true_dataset': <Value(<class 'str'>: None)>, 'use_area_attr': <Value(None: 'try')>, 'use_image_names': <Value(None: False)>}

normalize()

class kwcoco.coco_evaluator.**CocoEvaluator**(*config*)

Bases: object

Abstracts the evaluation process to execute on two coco datasets.

This can be run as a standalone script where the user specifies the paths to the true and predited dataset explicitly, or this can be used by a higher level script that produces the predictions and then sends them to this evaluator.

Example

```
>>> from kwcoco.coco_evaluator import CocoEvaluator
>>> from kwcoco.demo.perterb import perterb_coco
>>> import kwcoco
>>> true_dset = kwcoco.CocoDataset.demo('shapes8')
>>> kwargs = {
>>>     'box_noise': 0.5,
>>>     'n_fp': (0, 10),
>>>     'n_fn': (0, 10),
>>>     'with_probs': True,
>>> }
>>> pred_dset = perterb_coco(true_dset, **kwargs)
>>> config = {
>>>     'true_dataset': true_dset,
>>>     'pred_dataset': pred_dset,
>>>     'classes_of_interest': [],
>>> }
>>> coco_eval = CocoEvaluator(config)
>>> results = coco_eval.evaluate()
```

Config

alias of *CocoEvalConfig*

log(*msg*, *level='INFO'*)

evaluate()

Executes the main evaluation logic. Performs assignments between detections to make DetectionMetrics object, then creates per-item and ovr confusion vectors, and performs various threshold-vs-confusion analyses.

Returns

container storing (and capable of drawing / serializing) results

Return type
CocoResults

kwcoco.coco_evaluator.**dmet_area_weights**(*dmet*, *orig_weights*, *cfsn_vecs*, *area_ranges*, *coco_eval*, *use_area_attr=False*)

Hacky function to compute confusion vector ignore weights for different area thresholds. Needs to be slightly refactored.

class kwcoco.coco_evaluator.**CocoResults**(*resdata=None*)

Bases: NiceRepr, *DictProxy*

CommandLine

```
xdoctest -m /home/joncrall/code/kwcoco/kwcoco/coco_evaluator.py CocoResults --
↪profile
```

Example

```
>>> from kwcoco.coco_evaluator import *  # NOQA
>>> from kwcoco.coco_evaluator import CocoEvaluator
>>> import kwcoco
>>> true_dset = kwcoco.CocoDataset.demo('shapes2')
>>> from kwcoco.demo.perterb import perterb_coco
>>> kwargs = {
>>>     'box_noise': 0.5,
>>>     'n_fp': (0, 10),
>>>     'n_fn': (0, 10),
>>> }
>>> pred_dset = perterb_coco(true_dset, **kwargs)
>>> print('true_dset = {!r}'.format(true_dset))
>>> print('pred_dset = {!r}'.format(pred_dset))
>>> config = {
>>>     'true_dataset': true_dset,
>>>     'pred_dataset': pred_dset,
>>>     'area_range': ['small'],
>>>     'iou_thresh': [0.3],
>>> }
>>> coco_eval = CocoEvaluator(config)
>>> results = coco_eval.evaluate()
>>> # Now we can draw / serialize the results as we please
>>> dpath = ub.ensure_app_cache_dir('kwcoco/tests/test_out_dpath')
>>> #
>>> # test deserialization works
>>> state = results.__json__()
>>> self2 = CocoResults.from_json(state)
>>> #
>>> # xdoctest: +REQUIRES(module:kwplot)
```

(continues on next page)

(continued from previous page)

```
>>> results.dump_figures(dpath, figsize=(3, 2), tight=False)  # make this go faster
>>> results.dump(join(dpath, 'metrics.json'), indent='    ')
```

dump_figures(*out_dpath*, *expt_title=None*, *figsize='auto'*, *tight=True*)

classmethod from_json(*state*)

dump(*file*, *indent=' '*)

Serialize to json file

class kwcoco.coco_evaluator.**CocoSingleResult**(*nocls_measures*, *ovr_measures*, *cfsn_vecs*, *meta=None*)

Bases: NiceRepr

Container class to store, draw, summarize, and serialize results from CocoEvaluator.

Example

```
>>> # xdoctest: +REQUIRES(--slow)
>>> from kwcoco.coco_evaluator import *  # NOQA
>>> from kwcoco.coco_evaluator import CocoEvaluator
>>> import kwcoco
>>> true_dset = kwcoco.CocoDataset.demo('shapes8')
>>> from kwcoco.demo.perterb import perterb_coco
>>> kwargs = {
>>>     'box_noise': 0.2,
>>>     'n_fp': (0, 3),
>>>     'n_fn': (0, 3),
>>>     'with_probs': False,
>>> }
>>> pred_dset = perterb_coco(true_dset, **kwargs)
>>> print('true_dset = {!r}'.format(true_dset))
>>> print('pred_dset = {!r}'.format(pred_dset))
>>> config = {
>>>     'true_dataset': true_dset,
>>>     'pred_dataset': pred_dset,
>>>     'area_range': [(0, 32 ** 2), (32 ** 2, 96 ** 2)],
>>>     'iou_thresh': [0.3, 0.5, 0.95],
>>> }
>>> coco_eval = CocoEvaluator(config)
>>> results = coco_eval.evaluate()
>>> result = ub.peek(results.values())
>>> state = result.__json__()
>>> print('state = {}'.format(ub.repr2(state, nl=-1)))
>>> recon = CocoSingleResult.from_json(state)
>>> state = recon.__json__()
>>> print('state = {}'.format(ub.repr2(state, nl=-1)))
```

classmethod from_json(*state*)

dump(*file*, *indent=' '*)

Serialize to json file

dump_figures(*out_dpath*, *expt_title=None*, *figsize='auto'*, *tight=True*, *verbose=1*)

2.1.2.6 kwcoco.coco_image module

class kwcoco.coco_image.**CocoImage**(*img*, *dset=None*)

Bases: NiceRepr

An object-oriented representation of a coco image.

It provides helper methods that are specific to a single image.

This operates directly on a single coco image dictionary, but it can optionally be connected to a parent dataset, which allows it to use CocoDataset methods to query about relationships and resolve pointers.

This is different than the Images class in coco_object1d, which is just a vectorized interface to multiple objects.

Example

```
>>> import kwcoco
>>> dset1 = kwcoco.CocoDataset.demo('shapes8')
>>> dset2 = kwcoco.CocoDataset.demo('vidshapes8-multispectral')
```

```
>>> self = CocoImage(dset1.imgs[1], dset1)
>>> print('self = {!r}'.format(self))
>>> print('self.channels = {}'.format(ub.repr2(self.channels, nl=1)))
```

```
>>> self = CocoImage(dset2.imgs[1], dset2)
>>> print('self.channels = {}'.format(ub.repr2(self.channels, nl=1)))
>>> self.primary_asset()
```

classmethod from_gid(*dset*, *gid*)

property bundle_dpath

property video

Helper to grab the video for this image if it exists

detach()

Removes references to the underlying coco dataset, but keeps special information such that it wont be needed.

stats()

keys()

Proxy getter attribute for underlying *self.img* dictionary

get(*key*, *default=NoParam*)

Proxy getter attribute for underlying *self.img* dictionary

Example

```
>>> import pytest
>>> # without extra populated
>>> import kwcoco
>>> self = kwcoco.CocoImage({'foo': 1})
>>> assert self.get('foo') == 1
>>> assert self.get('foo', None) == 1
>>> # with extra populated
>>> self = kwcoco.CocoImage({'extra': {'foo': 1}})
>>> assert self.get('foo') == 1
>>> assert self.get('foo', None) == 1
>>> # without extra empty
>>> self = kwcoco.CocoImage({})
>>> with pytest.raises(KeyError):
>>>     self.get('foo')
>>> assert self.get('foo', None) is None
>>> # with extra empty
>>> self = kwcoco.CocoImage({'extra': {'bar': 1}})
>>> with pytest.raises(KeyError):
>>>     self.get('foo')
>>> assert self.get('foo', None) is None
```

property channels

property num_channels

property dsize

primary_image_filepath(*requires=None*)

primary_asset(*requires=None*)

Compute a “main” image asset.

Notes

Uses a heuristic.

- First, try to find the auxiliary image that has with the smallest

distortion to the base image (if known via warp_aux_to_img)

- Second, break ties by using the largest image if w / h is known
- Last, if previous information not available use the first auxiliary image.

Parameters

requires (*List[str]*) – list of attribute that must be non-None to consider an object as the primary one.

Returns

the asset dict or None if it is not found

Return type

None | dict

Todo:

- [ ] Add in primary heuristics

Example

```
>>> import kwarray
>>> from kwcoco.coco_image import *  # NOQA
>>> rng = kwarray.ensure_rng(0)
>>> def random_auxiliary(name, w=None, h=None):
>>>     return {'file_name': name, 'width': w, 'height': h}
>>> self = CocoImage({
>>>     'auxiliary': [
>>>         random_auxiliary('1'),
>>>         random_auxiliary('2'),
>>>         random_auxiliary('3'),
>>>     ]
>>> })
>>> assert self.primary_asset()['file_name'] == '1'
>>> self = CocoImage({
>>>     'auxiliary': [
>>>         random_auxiliary('1'),
>>>         random_auxiliary('2', 3, 3),
>>>         random_auxiliary('3'),
>>>     ]
>>> })
>>> assert self.primary_asset()['file_name'] == '2'
```

iter_image_filepaths(*with_bundle=True*)

Could rename to iter_asset_filepaths

Parameters

with_bundle (*bool*) – If True, prepends the bundle dpath to fully specify the path. Otherwise, just returns the registered string in the file_name attribute of each asset. Defaults to True.

iter_asset_objs()

Iterate through base + auxiliary dicts that have file paths

Yields

dict – an image or auxiliary dictionary

find_asset_obj(*channels*)

Find the asset dictionary with the specified channels

Example

```
>>> import kwcoco
>>> coco_img = kwcoco.CocoImage({'width': 128, 'height': 128})
>>> coco_img.add_auxiliary_item(
>>>     'rgb.png', channels='red|green|blue', width=32, height=32)
>>> assert coco_img.find_asset_obj('red') is not None
>>> assert coco_img.find_asset_obj('green') is not None
>>> assert coco_img.find_asset_obj('blue') is not None
>>> assert coco_img.find_asset_obj('red|blue') is not None
>>> assert coco_img.find_asset_obj('red|green|blue') is not None
>>> assert coco_img.find_asset_obj('red|green|blue') is not None
>>> assert coco_img.find_asset_obj('black') is None
>>> assert coco_img.find_asset_obj('r') is None
```

Example

```
>>> # Test with concise channel code
>>> import kwcoco
>>> coco_img = kwcoco.CocoImage({'width': 128, 'height': 128})
>>> coco_img.add_auxiliary_item(
>>>     'msi.png', channels='foo.0:128', width=32, height=32)
>>> assert coco_img.find_asset_obj('foo') is None
>>> assert coco_img.find_asset_obj('foo.3') is not None
>>> assert coco_img.find_asset_obj('foo.3:5') is not None
>>> assert coco_img.find_asset_obj('foo.3000') is None
```

add_auxiliary_item(*file_name=None*, *channels=None*, *imdata=None*, *warp_aux_to_img=None*, *width=None*, *height=None*, *imwrite=False*)

Adds an auxiliary / asset item to the image dictionary.

This operation can be done purely in-memory (the default), or the image data can be written to a file on disk (via the imwrite=True flag).

Parameters

- **file_name** (*str | None*) – The name of the file relative to the bundle directory. If unspecified, imdata must be given.
- **channels** (*str | kwcoco.FusedChannelSpec*) – The channel code indicating what each of the bands represents. These channels should be disjoint wrt to the existing data in this image (this is not checked).
- **imdata** (*ndarray | None*) – The underlying image data this auxiliary item represents. If unspecified, it is assumed file_name points to a path on disk that will eventually exist. If imdata, file_name, and the special imwrite=True flag are specified, this function will write the data to disk.
- **warp_aux_to_img** (*kwimage.Affine*) – The transformation from this auxiliary space to image space. If unspecified, assumes this item is related to image space by only a scale factor.
- **width** (*int*) – Width of the data in auxiliary space (inferred if unspecified)
- **height** (*int*) – Height of the data in auxiliary space (inferred if unspecified)

- **imwrite** (*bool*) – If specified, both imdata and file_name must be specified, and this will write the data to disk. Note: it it recommended that you simply call imwrite yourself before or after calling this function. This lets you better control imwrite parameters.

Todo:

- [ ] Allow imwrite to specify an executor that is used to

return a Future so the imwrite call does not block.

Example

```
>>> from kwcoco.coco_image import *  # NOQA
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('vidshapes8-multispectral')
>>> coco_img = dset.coco_image(1)
>>> imdata = np.random.rand(32, 32, 5)
>>> channels = kwcoco.FusedChannelSpec.coerce('Aux:5')
>>> coco_img.add_auxiliary_item(imdata=imdata, channels=channels)
```

Example

```
>>> import kwcoco
>>> dset = kwcoco.CocoDataset()
>>> gid = dset.add_image(name='my_image_name', width=200, height=200)
>>> coco_img = dset.coco_image(gid)
>>> coco_img.add_auxiliary_item('path/img1_B0.tif', channels='B0', width=200,␣
↪height=200)
>>> coco_img.add_auxiliary_item('path/img1_B1.tif', channels='B1', width=200,␣
↪height=200)
>>> coco_img.add_auxiliary_item('path/img1_B2.tif', channels='B2', width=200,␣
↪height=200)
>>> coco_img.add_auxiliary_item('path/img1_TCI.tif', channels='r|g|b',␣
↪width=200, height=200)
```

add_asset(*file_name=None*, *channels=None*, *imdata=None*, *warp_aux_to_img=None*, *width=None*, *height=None*, *imwrite=False*)

Adds an auxiliary / asset item to the image dictionary.

This operation can be done purely in-memory (the default), or the image data can be written to a file on disk (via the imwrite=True flag).

Parameters

- **file_name** (*str | None*) – The name of the file relative to the bundle directory. If unspecified, imdata must be given.
- **channels** (*str | kwcoco.FusedChannelSpec*) – The channel code indicating what each of the bands represents. These channels should be disjoint wrt to the existing data in this image (this is not checked).
- **imdata** (*ndarray | None*) – The underlying image data this auxiliary item represents. If unspecified, it is assumed file_name points to a path on disk that will eventually exist. If

imdata, file_name, and the special imwrite=True flag are specified, this function will write the data to disk.

- **warp_aux_to_img** (*kwimage.Affine*) – The transformation from this auxiliary space to image space. If unspecified, assumes this item is related to image space by only a scale factor.
- **width** (*int*) – Width of the data in auxiliary space (inferred if unspecified)
- **height** (*int*) – Height of the data in auxiliary space (inferred if unspecified)
- **imwrite** (*bool*) – If specified, both imdata and file_name must be specified, and this will write the data to disk. Note: it it recommended that you simply call imwrite yourself before or after calling this function. This lets you better control imwrite parameters.

Todo:

- [ ] Allow imwrite to specify an executor that is used to

return a Future so the imwrite call does not block.

Example

```
>>> from kwcoco.coco_image import *  # NOQA
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('vidshapes8-multispectral')
>>> coco_img = dset.coco_image(1)
>>> imdata = np.random.rand(32, 32, 5)
>>> channels = kwcoco.FusedChannelSpec.coerce('Aux:5')
>>> coco_img.add_auxiliary_item(imdata=imdata, channels=channels)
```

Example

```
>>> import kwcoco
>>> dset = kwcoco.CocoDataset()
>>> gid = dset.add_image(name='my_image_name', width=200, height=200)
>>> coco_img = dset.coco_image(gid)
>>> coco_img.add_auxiliary_item('path/img1_B0.tif', channels='B0', width=200,␣
↪height=200)
>>> coco_img.add_auxiliary_item('path/img1_B1.tif', channels='B1', width=200,␣
↪height=200)
>>> coco_img.add_auxiliary_item('path/img1_B2.tif', channels='B2', width=200,␣
↪height=200)
>>> coco_img.add_auxiliary_item('path/img1_TCI.tif', channels='r|g|b',␣
↪width=200, height=200)
```

delay(*channels=None*, *space='image'*, *bundle_dpath=None*, *interpolation='linear'*, *antialias=True*, *nodata_method=None*, *mode=1*)

Perform a delayed load on the data in this image.

The delayed load can load a subset of channels, and perform lazy warping operations. If the underlying data is in a tiled format this can reduce the amount of disk IO needed to read the data if only a small crop or lower resolution view of the data is needed.

Note: This method is experimental and relies on the delayed load proof-of-concept.

Parameters

- **gid** (*int*) – image id to load
- **channels** (*kwcoco.FusedChannelSpec*) – specific channels to load. if unspecified, all channels are loaded.
- **space** (*str*) – can either be "image" for loading in image space, or "video" for loading in video space.

Todo:

- **[X] Currently can only take all or none of the channels from each**
  base-image / auxiliary dict. For instance if the main image is r|g|b you can't just select g|b at the moment.
- **[X] The order of the channels in the delayed load should**
  match the requested channel order.
- [X] TODO: add nans to bands that don't exist or throw an error
- **[ ] This function could stand to have a better name. Maybe imread**
  with a delayed=True flag? Or maybe just delayed_load?

Example

```
>>> from kwcoco.coco_image import *  # NOQA
>>> import kwcoco
>>> gid = 1
>>> #
>>> dset = kwcoco.CocoDataset.demo('vidshapes8-multispectral')
>>> self = CocoImage(dset.imgs[gid], dset)
>>> delayed = self.delay()
>>> print('delayed = {!r}'.format(delayed))
>>> print('delayed.finalize() = {!r}'.format(delayed.finalize()))
>>> print('delayed.finalize() = {!r}'.format(delayed.finalize()))
>>> #
>>> dset = kwcoco.CocoDataset.demo('shapes8')
>>> delayed = dset.coco_image(gid).delay()
>>> print('delayed = {!r}'.format(delayed))
>>> print('delayed.finalize() = {!r}'.format(delayed.finalize()))
>>> print('delayed.finalize() = {!r}'.format(delayed.finalize()))
```

```
>>> crop = delayed.crop((slice(0, 3), slice(0, 3)))
>>> crop.finalize()
```

```
>>> # TODO: should only select the "red" channel
>>> dset = kwcoco.CocoDataset.demo('shapes8')
>>> delayed = CocoImage(dset.imgs[gid], dset).delay(channels='r')
```

```
>>> import kwcoco
>>> gid = 1
>>> #
>>> dset = kwcoco.CocoDataset.demo('vidshapes8-multispectral')
>>> delayed = dset.coco_image(gid).delay(channels='B1|B2', space='image')
>>> print('delayed = {!r}'.format(delayed))
>>> print('delayed.finalize() = {!r}'.format(delayed.finalize()))
>>> delayed = dset.coco_image(gid).delay(channels='B1|B2|B11', space='image')
>>> print('delayed = {!r}'.format(delayed))
>>> print('delayed.finalize() = {!r}'.format(delayed.finalize()))
>>> delayed = dset.coco_image(gid).delay(channels='B8|B1', space='video')
>>> print('delayed = {!r}'.format(delayed))
>>> print('delayed.finalize() = {!r}'.format(delayed.finalize()))
```

```
>>> delayed = dset.coco_image(gid).delay(channels='B8|foo|bar|B1', space='video
↪')
>>> print('delayed = {!r}'.format(delayed))
>>> print('delayed.finalize() = {!r}'.format(delayed.finalize()))
```

Example

```
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo()
>>> coco_img = dset.coco_image(1)
>>> # Test case where nothing is registered in the dataset
>>> delayed = coco_img.delay()
>>> final = delayed.finalize()
>>> assert final.shape == (512, 512, 3)
```

```
>>> delayed = coco_img.delay(mode=1)
>>> final = delayed.finalize()
>>> print('final.shape = {}'.format(ub.repr2(final.shape, nl=1)))
>>> assert final.shape == (512, 512, 3)
```

Example

```
>>> # Test that delay works when imdata is stored in the image
>>> # dictionary itself.
>>> from kwcoco.coco_image import *  # NOQA
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('vidshapes8-multispectral')
>>> coco_img = dset.coco_image(1)
>>> imdata = np.random.rand(6, 6, 5)
>>> imdata[:] = np.arange(5)[None, None, :]
>>> channels = kwcoco.FusedChannelSpec.coerce('Aux:5')
>>> coco_img.add_auxiliary_item(imdata=imdata, channels=channels)
>>> delayed = coco_img.delay(channels='B1|Aux:2:4', mode=1)
>>> final = delayed.finalize()
```

Example

```
>>> # Test delay when loading in asset space
>>> from kwcoco.coco_image import *  # NOQA
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('vidshapes8-msi-multisensor')
>>> coco_img = dset.coco_image(1)
>>> stream1 = coco_img.channels.streams()[0]
>>> stream2 = coco_img.channels.streams()[1]
>>> aux_delayed = coco_img.delay(stream1, space='asset')
>>> img_delayed = coco_img.delay(stream1, space='image')
>>> vid_delayed = coco_img.delay(stream1, space='video')
>>> #
>>> aux_imdata = aux_delayed.as_xarray().finalize()
>>> img_imdata = img_delayed.as_xarray().finalize()
>>> assert aux_imdata.shape != img_imdata.shape
>>> # Cannot load multiple asset items at the same time in
>>> # asset space
>>> import pytest
>>> fused_channels = stream1 | stream2
>>> with pytest.raises(kwcoco.exceptions.CoordinateCompatibilityError):
>>>     aux_delayed2 = coco_img.delay(fused_channels, space='asset')
```

valid_region(*space='image'*)

If this image has a valid polygon, return it in image, or video space

property warp_vid_from_img

property warp_img_from_vid

class kwcoco.coco_image.**CocoAsset**

Bases: object

A Coco Asset / Auxiliary Item

Represents one 2D image file relative to a parent img.

Could be a single asset, or an image with sub-assets, but sub-assets are ignored here.

Initially we called these "auxiliary" items, but I think we should change their name to "assets", which better maps with STAC terminology.

keys()

Proxy getter attribute for underlying *self.obj* dictionary

get(*key*, *default=NoParam*)

Proxy getter attribute for underlying *self.obj* dictionary

2.1.2.7 kwcoco.coco_objects1d module

Vectorized ORM-like objects used in conjunction with coco_dataset

class kwcoco.coco_objects1d.**ObjectList1D**(*ids*, *dset*, *key*)

Bases: NiceRepr

Vectorized access to lists of dictionary objects

Lightweight reference to a set of object (e.g. annotations, images) that allows for convenient property access.

Parameters

- **ids** (*List[int]*) – list of ids
- **dset** (*CocoDataset*) – parent dataset
- **key** (*str*) – main object name (e.g. 'images', 'annotations')

Types:
ObjT = Ann | Img | Cat # can be one of these types ObjectList1D gives us access to a List[ObjT]

Example

```
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo()
>>> # Both annots and images are object lists
>>> self = dset.annots()
>>> self = dset.images()
>>> # can call with a list of ids or not, for everything
>>> self = dset.annots([1, 2, 11])
>>> self = dset.images([1, 2, 3])
>>> self.lookup('id')
>>> self.lookup(['id'])
```

unique()

Removes any duplicates entries in this object

Returns
ObjectList1D

property objs

Get the underlying object dictionary for each object.

Returns
all object dictionaries

Return type
List[ObjT]

take(*idxs*)

Take a subset by index

Returns
ObjectList1D

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo().annots()
>>> assert len(self.take([0, 2, 3])) == 3
```

compress(*flags*)

Take a subset by flags

Returns
ObjectList1D

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo().images()
>>> assert len(self.compress([True, False, True])) == 2
```

peek()

Return the first object dictionary

Returns
object dictionary

Return type
ObjT

Example

```
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo()
>>> self = dset.images()
>>> assert self.peek()['id'] == 1
>>> # Check that subsets return correct items
>>> sub0 = self.compress([i % 2 == 0 for i in range(len(self))])
>>> sub1 = self.compress([i % 2 == 1 for i in range(len(self))])
>>> assert sub0.peek()['id'] == 1
>>> assert sub1.peek()['id'] == 2
```

lookup(*key*, *default=NoParam*, *keepid=False*)

Lookup a list of object attributes

Parameters

- **key** (*str | Iterable*) – name of the property you want to lookup can also be a list of names, in which case we return a dict
- **default** – if specified, uses this value if it doesn't exist in an ObjT.
- **keepid** – if True, return a mapping from ids to the property

Returns
a list of whatever type the object is Dict[str, ObjT]

Return type
List[ObjT]

Example

```
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo()
>>> self = dset.annots()
>>> self.lookup('id')
>>> key = ['id']
>>> default = None
>>> self.lookup(key=['id', 'image_id'])
>>> self.lookup(key=['id', 'image_id'])
>>> self.lookup(key='foo', default=None, keepid=True)
>>> self.lookup(key=['foo'], default=None, keepid=True)
>>> self.lookup(key=['id', 'image_id'], keepid=True)
```

get(*key*, *default=NoParam*, *keepid=False*)

Lookup a list of object attributes

Parameters

- **key** (*str*) – name of the property you want to lookup
- **default** – if specified, uses this value if it doesn't exist in an ObjT.
- **keepid** – if True, return a mapping from ids to the property

Returns
a list of whatever type the object is Dict[str, ObjT]

Return type
List[ObjT]

Example

```
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo()
>>> self = dset.annots()
>>> self.get('id')
>>> self.get(key='foo', default=None, keepid=True)
```

set(*key*, *values*)

Assign a value to each annotation

Parameters

- **key** (*str*) – the annotation property to modify
- **values** (*Iterable | Any*) – an iterable of values to set for each annot in the dataset. If the item is not iterable, it is assigned to all objects.

Example

```
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo()
>>> self = dset.annots()
>>> self.set('my-key1', 'my-scalar-value')
>>> self.set('my-key2', np.random.rand(len(self)))
>>> print('dset.imgs = {}'.format(ub.repr2(dset.imgs, nl=1)))
>>> self.get('my-key2')
```

attribute_frequency()

Compute the number of times each key is used in a dictionary

Returns
Dict[str, int]

Example

```
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo()
>>> self = dset.annots()
>>> attrs = self.attribute_frequency()
>>> print('attrs = {}'.format(ub.repr2(attrs, nl=1)))
```

class kwcoco.coco_objects1d.**ObjectGroups**(*groups*, *dset*)

Bases: NiceRepr

An object for holding a groups of *ObjectList1D* objects

lookup(*key*, *default=NoParam*)

class kwcoco.coco_objects1d.**Categories**(*ids*, *dset*)

Bases: *ObjectList1D*

Vectorized access to category attributes

SeeAlso:
kwcoco.coco_dataset.MixinCocoObjects.categories()

Example

```
>>> from kwcoco.coco_objects1d import Categories  # NOQA
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo()
>>> ids = list(dset.cats.keys())
>>> self = Categories(ids, dset)
>>> print('self.name = {!r}'.format(self.name))
>>> print('self.supercategory = {!r}'.format(self.supercategory))
```

property cids

property name

property supercategory

class kwcoco.coco_objects1d.Videos(*ids*, *dset*)

Bases: *ObjectList1D*

Vectorized access to video attributes

SeeAlso:

kwcoco.coco_dataset.MixinCocoObjects.videos()

Example

```
>>> from kwcoco.coco_objects1d import Videos  # NOQA
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('vidshapes5')
>>> ids = list(dset.index.videos.keys())
>>> self = Videos(ids, dset)
>>> print('self = {!r}'.format(self))
self = <Videos(num=5) at ...>
```

property images

Example:

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo('vidshapes8').videos()
>>> print(self.images)
<ImageGroups(n=8, m=2.0, s=0.0)>
```

class kwcoco.coco_objects1d.Images(*ids*, *dset*)

Bases: *ObjectList1D*

Vectorized access to image attributes

Example

```
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('photos')
>>> images = dset.images()
>>> print('images = {}'.format(images))
images = <Images(num=3)...>
>>> print('images.gname = {}'.format(images.gname))
images.gname = ['astro.png', 'carl.jpg', 'stars.png']
```

SeeAlso:

kwcoco.coco_dataset.MixinCocoObjects.images()

property coco_images

property gids

property gname

property gpath

property width

property height

property size

Example:

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo().images()
>>> self._dset._ensure_imgsize()
...
>>> print(self.size)
[(512, 512), (328, 448), (256, 256)]
```

property area

Example:

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo().images()
>>> self._dset._ensure_imgsize()
...
>>> print(self.area)
[262144, 146944, 65536]
```

property n_annots

Example:

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo().images()
>>> print(ub.repr2(self.n_annots, nl=0))
[9, 2, 0]
```

property aids

Example:

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo().images()
>>> print(ub.repr2(list(map(list, self.aids)), nl=0))
[[1, 2, 3, 4, 5, 6, 7, 8, 9], [10, 11], []]
```

property annots

Example:

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo().images()
>>> print(self.annots)
<AnnotGroups(n=3, m=3.7, s=3.9)>
```

class kwcoco.coco_objects1d.**Annots**(*ids*, *dset*)

Bases: *ObjectList1D*

Vectorized access to annotation attributes

SeeAlso:

kwcoco.coco_dataset.MixinCocoObjects.annots()

Example

```
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('photos')
>>> annots = dset.annots()
>>> print('annots = {}'.format(annots))
annots = <Annots(num=11)>
>>> image_ids = annots.lookup('image_id')
>>> print('image_ids = {}'.format(image_ids))
image_ids = [1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2]
```

property aids

The annotation ids of this column of annotations

property images

Get the column of images

Returns
Images

property image_id

property category_id

property gids

Get the column of image-ids

Returns
list of image ids

Return type
List[int]

property cids

Get the column of category-ids

Returns
List[int]

property cnames

Get the column of category names

Returns
List[int]

property detections

Get the kwimage-style detection objects

Returns
kwimage.Detections

Example

```
>>> # xdoctest: +REQUIRES(module:kwimage)
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo('shapes32').annots([1, 2, 11])
>>> dets = self.detections
>>> print('dets.data = {!r}'.format(dets.data))
>>> print('dets.meta = {!r}'.format(dets.meta))
```

property boxes

Get the column of kwimage-style bounding boxes

Returns
kwimage.Boxes

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo().annots([1, 2, 11])
>>> print(self.boxes)
<Boxes(xywh,
    array([[ 10,  10, 360, 490],
           [350,   5, 130, 290],
           [156, 130,  45,  18]]))>
```

property xywh

Returns raw boxes

DEPRECATED.

Returns
raw boxes in xywh format

Return type
List[List[int]]

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo().annots([1, 2, 11])
>>> print(self.xywh)
```

class kwcoco.coco_objects1d.**AnnotGroups**(*groups*, *dset*)

Bases: *ObjectGroups*

Annotation groups are vectorized lists of lists.

Each item represents a set of annotations that corresopnds with something (i.e. belongs to a particular image).

Example

```
>>> from kwcoco.coco_objects1d import ImageGroups
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('photos')
>>> images = dset.images()
>>> # Requesting the "annots" property from a Images object
>>> # will return an AnnotGroups object
>>> group: AnnotGroups = images.annots
>>> # Printing the group gives info on the mean/std of the number
>>> # of items per group.
>>> print(group)
<AnnotGroups(n=3, m=3.7, s=3.9)...>
>>> # Groups are fairly restrictive, they dont provide property level
>>> # access in many cases, but the lookup method is available
>>> print(group.lookup('id'))
[[1, 2, 3, 4, 5, 6, 7, 8, 9], [10, 11], []]
>>> print(group.lookup('image_id'))
[[1, 1, 1, 1, 1, 1, 1, 1, 1], [2, 2], []]
>>> print(group.lookup('category_id'))
[[1, 2, 3, 4, 5, 5, 5, 5, 5], [6, 4], []]
```

property cids

Get the grouped category ids for annotations in this group

Return type
List[List[id]]

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo('photos').images().annots
>>> print('self.cids = {}'.format(ub.repr2(self.cids, nl=0)))
self.cids = [[1, 2, 3, 4, 5, 5, 5, 5, 5], [6, 4], []]
```

property cnames

Get the grouped category names for annotations in this group

Return type
List[List[str]]

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo('photos').images().annots
>>> print('self.cnames = {}'.format(ub.repr2(self.cnames, nl=0)))
self.cnames = [['astronaut', 'rocket', 'helmet', 'mouth', 'star', 'star', 'star
↪', 'star', 'star'], ['astronomer', 'mouth'], []]
```

class kwcoco.coco_objects1d.**ImageGroups**(*groups*, *dset*)

Bases: *ObjectGroups*

Image groups are vectorized lists of other Image objects.

Each item represents a set of images that corresopnds with something (i.e. belongs to a particular video).

Example

```
>>> from kwcoco.coco_objects1d import ImageGroups
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('vidshapes8')
>>> videos = dset.videos()
>>> # Requesting the "images" property from a Videos object
>>> # will return an ImageGroups object
>>> group: ImageGroups = videos.images
>>> # Printing the group gives info on the mean/std of the number
>>> # of items per group.
>>> print(group)
<ImageGroups(n=8, m=2.0, s=0.0)...>
>>> # Groups are fairly restrictive, they dont provide property level
>>> # access in many cases, but the lookup method is available
>>> print(group.lookup('id'))
[[1, 2], [3, 4], [5, 6], [7, 8], [9, 10], [11, 12], [13, 14], [15, 16]]
>>> print(group.lookup('video_id'))
[[1, 1], [2, 2], [3, 3], [4, 4], [5, 5], [6, 6], [7, 7], [8, 8]]
>>> print(group.lookup('frame_index'))
[[0, 1], [0, 1], [0, 1], [0, 1], [0, 1], [0, 1], [0, 1], [0, 1]]
```

2.1.2.8 kwcoco.coco_schema module

CommandLine

```
python -m kwcoco.coco_schema
xdoctest -m kwcoco.coco_schema __doc__
```

Example

```
>>> import kwcoco
>>> from kwcoco.coco_schema import COCO_SCHEMA
>>> import jsonschema
>>> dset = kwcoco.CocoDataset.demo('shapes1')
>>> # print('dset.dataset = {}'.format(ub.repr2(dset.dataset, nl=2)))
>>> COCO_SCHEMA.validate(dset.dataset)
```

```
>>> try:
>>>     jsonschema.validate(dset.dataset, schema=COCO_SCHEMA)
>>> except jsonschema.exceptions.ValidationError as ex:
>>>     vali_ex = ex
>>>     print('ex = {!r}'.format(ex))
>>>     raise
>>> except jsonschema.exceptions.SchemaError as ex:
>>>     print('ex = {!r}'.format(ex))
>>>     schema_ex = ex
```

(continues on next page)

(continued from previous page)

```
>>>     print('schema_ex.instance = {}'.format(ub.repr2(schema_ex.instance, nl=-1)))
>>>     raise
```

```
>>> # Test the multispectral image defintino
>>> import copy
>>> dataset = dset.copy().dataset
>>> img = dataset['images'][0]
>>> img.pop('file_name')
>>> import pytest
>>> with pytest.raises(jsonschema.ValidationError):
>>>     COCO_SCHEMA.validate(dataset)
>>> import pytest
>>> img['auxiliary'] = [{'file_name': 'foobar'}]
>>> with pytest.raises(jsonschema.ValidationError):
>>>     COCO_SCHEMA.validate(dataset)
>>> img['name'] = 'aux-only images must have a name'
>>> COCO_SCHEMA.validate(dataset)
```

kwcoco.coco_schema.**deprecated**(**args*)

kwcoco.coco_schema.**TUPLE**(**args*, ***kw*)

2.1.2.9 kwcoco.coco_sql_dataset module

Todo:

- **[ ] We get better speeds with raw SQL over alchemy. Can we mitigate the** speed difference so we can take advantage of alchemy's expressiveness?

Finally got a baseline implementation of an SQLite backend for COCO datasets. This mostly plugs into my existing tools (as long as only read operations are used; haven't impelmented writing yet) by duck-typing the dict API.

This solves the issue of forking and then accessing nested dictionaries in the JSON-style COCO objects. (When you access the dictionary Python will increment a reference count which triggers copy-on-write for whatever memory page that data happened to live in. Non-contiguous access had the effect of excessive memory copies).

For "medium sized" datasets its quite a bit slower. Running through a torch DataLoader with 4 workers for 10,000 images executes at a rate of 100Hz but takes 850MB of RAM. Using the duck-typed SQL backend only uses 500MB (which includes the cost of caching), but runs at 45Hz (which includes the benefit of caching).

However, once I scale up to 100,000 images I start seeing benefits. The in-memory dictionary interface chugs at 1.05HZ, and is taking more than 4GB of memory at the time I killed the process (eta was over an hour). The SQL backend ran at 45Hz and took about 3 minutes and used about 2.45GB of memory.

Without a cache, SQL runs at 30HZ and takes 400MB for 10,000 images, and for 100,000 images it gets 30Hz with 1.1GB. There is also a much larger startup time. I'm not exactly sure what it is yet, but its probably some preprocessing I'm doing.

Using a LRU cache we get 45Hz and 1.05GB of memory, so that's a clear win. We do need to be sure to disable the cache if we ever implement write mode.

I'd like to be a bit faster on the medium sized datasets (I'd really like to avoid caching rows, which is why the speed is currently semi-reasonable), but I don't think I can do any better than this because single-row lookup time is *O(log(N))* for sqlite, whereas its *O(1)* for dictionaries. (I wish sqlite had an option to create a hash-table index for a table, but I

dont think it does). I optimized as many of the dictionary operations as possible (for instance, iterating through keys, values, and items should be O(N) instead of *O(N log(N))*), but the majority of the runtime cost is in the single-row lookup time.

There are a few questions I still have if anyone has insight:

- Say I want to select a subset of K rows from a table with N entries, and I have a list of all of the rowids that I want. Is there any way to do this better than `O(K log(N))`? I tried using a `SELECT col FROM table WHERE id IN (?, ?, ?, ?, ...)` filling in enough *?* as there are rows in my subset. I'm not sure what the complexity of using a query like this is. I'm not sure what the *IN* implementation looks like. Can this be done more efficiently by with a temporary table and a `JOIN`?
- There really is no way to do `O(1)` row lookup in sqlite right? Is there a way in PostgreSQL or some other backend sqlalchemy supports?

I found that PostgreSQL does support hash indexes: https://www.postgresql.org/docs/13/indexes-types.html I'm really not interested in setting up a global service though . I also found a 10-year old thread with a hash-index feature request for SQLite, which I unabashedly resurrected http://sqlite.1065341.n5.nabble.com/Feature-request-hash-index-td23367.html

class kwcoco.coco_sql_dataset.**Category**(***kwargs*)

Bases: `Base`

id

unique internal id

name

unique external name or identifier

alias

list of alter egos

supercategory

coarser category name

class kwcoco.coco_sql_dataset.**KeypointCategory**(***kwargs*)

Bases: `Base`

id

unique internal id

name

unique external name or identifier

alias

list of alter egos

supercategory

coarser category name

reflection_id

if augmentation reflects the image, change keypoint id to this

class kwcoco.coco_sql_dataset.**Video**(***kwargs*)

Bases: `Base`

id

unique internal id

name

caption

width

height

class kwcoco.coco_sql_dataset.**Image**(***kwargs*)

Bases: Base

id

unique internal id

name

file_name

width

height

video_id

timestamp

frame_index

channels

See ChannelSpec

warp_img_to_vid

See TransformSpec

auxiliary

class kwcoco.coco_sql_dataset.**Annotation**(***kwargs*)

Bases: Base

id

image_id

category_id

track_id

segmentation

keypoints

bbox

score

weight

prob

iscrowd

caption

kwcoco.coco_sql_dataset.**cls**

alias of *Image*

kwcoco.coco_sql_dataset.**orm_to_dict**(*obj*)

kwcoco.coco_sql_dataset.**dict_restructure**(*item*)

Removes the unstructured field so the API is transparent to the user.

class kwcoco.coco_sql_dataset.**SqlListProxy**(*session*, *cls*)

Bases: NiceRepr

A view of an SQL table that behaves like a Python list

class kwcoco.coco_sql_dataset.**SqlDictProxy**(*session*, *cls*, *keyattr=None*, *ignore_null=False*)

Bases: *DictLike*

Duck-types an SQL table as a dictionary of dictionaries.

The key is specified by an indexed column (by default it is the *id* column). The values are dictionaries containing all data for that row.

Note: With SQLite indexes are B-Trees so lookup is O(log(N)) and not O(1) as will regular dictionaries. Iteration should still be O(N), but databases have much more overhead than Python dictionaries.

Parameters

- **session** (*Session*) – the sqlalchemy session
- **cls** (*Type*) – the declarative sqlalchemy table class
- **keyattr** – the indexed column to use as the keys
- **ignore_null** (*bool*) – if True, ignores any keys set to NULL, otherwise NULL keys are allowed.

Example

```
>>> # xdoctest: +REQUIRES(module:sqlalchemy)
>>> from kwcoco.coco_sql_dataset import *  # NOQA
>>> import pytest
>>> sql_dset, dct_dset = demo(num=10)
>>> proxy = sql_dset.index.anns
```

```
>>> keys = list(proxy.keys())
>>> values = list(proxy.values())
>>> items = list(proxy.items())
>>> item_keys = [t[0] for t in items]
>>> item_vals = [t[1] for t in items]
>>> lut_vals = [proxy[key] for key in keys]
>>> assert item_vals == lut_vals == values
>>> assert item_keys == keys
>>> assert len(proxy) == len(keys)
```

```
>>> goodkey1 = keys[1]
>>> badkey1 = -100000000000
>>> badkey2 = 'foobarbazbiz'
>>> badkey3 = object()
>>> assert goodkey1 in proxy
>>> assert badkey1 not in proxy
>>> assert badkey2 not in proxy
>>> assert badkey3 not in proxy
>>> with pytest.raises(KeyError):
>>>     proxy[badkey1]
>>> with pytest.raises(KeyError):
>>>     proxy[badkey2]
>>> with pytest.raises(KeyError):
>>>     proxy[badkey3]
```

```
>>> # xdoctest: +SKIP
>>> from kwcoco.coco_sql_dataset import _benchmark_dict_proxy_ops
>>> ti = _benchmark_dict_proxy_ops(proxy)
>>> print('ti.measures = {}'.format(ub.repr2(ti.measures, nl=2, align=':',␣
↪precision=6)))
```

Example

```
>>> # xdoctest: +REQUIRES(module:sqlalchemy)
>>> from kwcoco.coco_sql_dataset import *  # NOQA
>>> import kwcoco
>>> # Test the variant of the SqlDictProxy where we ignore None keys
>>> # This is the case for name_to_img and file_name_to_img
>>> dct_dset = kwcoco.CocoDataset.demo('shapes1')
>>> dct_dset.add_image(name='no_file_image1')
>>> dct_dset.add_image(name='no_file_image2')
>>> dct_dset.add_image(name='no_file_image3')
>>> sql_dset = dct_dset.view_sql(memory=True)
>>> assert len(dct_dset.index.imgs) == 4
>>> assert len(dct_dset.index.file_name_to_img) == 1
>>> assert len(dct_dset.index.name_to_img) == 3
>>> assert len(sql_dset.index.imgs) == 4
>>> assert len(sql_dset.index.file_name_to_img) == 1
>>> assert len(sql_dset.index.name_to_img) == 3
```

```
>>> proxy = sql_dset.index.file_name_to_img
>>> assert len(list(proxy.keys())) == 1
>>> assert len(list(proxy.values())) == 1
```

```
>>> proxy = sql_dset.index.name_to_img
>>> assert len(list(proxy.keys())) == 3
>>> assert len(list(proxy.values())) == 3
```

```
>>> proxy = sql_dset.index.imgs
>>> assert len(list(proxy.keys())) == 4
```

(continues on next page)

(continued from previous page)

```
>>> assert len(list(proxy.values())) == 4
```

keys()

values()

items()

class kwcoco.coco_sql_dataset.**SqlIdGroupDictProxy**(*session*, *valattr*, *keyattr*, *parent_keyattr*, *group_order_attr=None*)

Bases: *DictLike*

Similar to *SqlDictProxy*, but maps ids to groups of other ids.

Simulates a dictionary that maps ids of a parent table to all ids of another table corresponding to rows where a specific column has that parent id.

The items in the group can be sorted by the `group_order_attr` if specified.

For example, imagine two tables: images with one column (id) and annotations with two columns (id, image_id). This class can help provide a mpaping from each *image.id* to a *Set[annotation.id]* where those annotation rows have *annotation.image_id* = *image.id*.

Example

```
>>> # xdoctest: +REQUIRES(module:sqlalchemy)
>>> from kwcoco.coco_sql_dataset import *  # NOQA
>>> sql_dset, dct_dset = demo(num=10)
>>> proxy = sql_dset.index.gid_to_aids
```

```
>>> keys = list(proxy.keys())
>>> values = list(proxy.values())
>>> items = list(proxy.items())
>>> item_keys = [t[0] for t in items]
>>> item_vals = [t[1] for t in items]
>>> lut_vals = [proxy[key] for key in keys]
>>> assert item_vals == lut_vals == values
>>> assert item_keys == keys
>>> assert len(proxy) == len(keys)
```

```
>>> # xdoctest: +SKIP
>>> from kwcoco.coco_sql_dataset import _benchmark_dict_proxy_ops
>>> ti = _benchmark_dict_proxy_ops(proxy)
>>> print('ti.measures = {}'.format(ub.repr2(ti.measures, nl=2, align=':',␣
↪precision=6)))
```

Example

```
>>> # xdoctest: +REQUIRES(module:sqlalchemy)
>>> from kwcoco.coco_sql_dataset import *  # NOQA
>>> import kwcoco
>>> # Test the group sorted variant of this by using vidid_to_gids
>>> # where the "gids" must be sorted by the image frame indexes
>>> dct_dset = kwcoco.CocoDataset.demo('vidshapes1')
>>> dct_dset.add_image(name='frame-index-order-demo1', frame_index=-30, video_id=1)
>>> dct_dset.add_image(name='frame-index-order-demo2', frame_index=10, video_id=1)
>>> dct_dset.add_image(name='frame-index-order-demo3', frame_index=3, video_id=1)
>>> dct_dset.add_video(name='empty-video1')
>>> dct_dset.add_video(name='empty-video2')
>>> dct_dset.add_video(name='empty-video3')
>>> sql_dset = dct_dset.view_sql(memory=True)
>>> orig = dct_dset.index.vidid_to_gids
>>> proxy = sql_dset.index.vidid_to_gids
>>> from kwcoco.util.util_json import indexable_allclose
>>> assert indexable_allclose(orig, dict(proxy))
>>> items = list(proxy.items())
>>> vals = list(proxy.values())
>>> keys = list(proxy.keys())
>>> assert len(keys) == len(vals)
>>> assert dict(zip(keys, vals)) == dict(items)
```

keys()

items()

values()

class kwcoco.coco_sql_dataset.**CocoSqlIndex**

Bases: object

Simulates the dictionary provided by *kwcoco.coco_dataset.CocoIndex*

build(*parent*)

class kwcoco.coco_sql_dataset.**CocoSqlDatabase**(*uri=None*, *tag=None*, *img_root=None*)

Bases: *AbstractCocoDataset*, *MixinCocoAccessors*, *MixinCocoObjects*, *MixinCocoStats*, *MixinCocoDraw*, NiceRepr

Provides an API nearly identical to kwcoco.CocoDatabase, but uses an SQL backend data store. This makes it robust to copy-on-write memory issues that arise when forking, as discussed in[1].

Note: By default constructing an instance of the CocoSqlDatabase does not create a connection to the databse. Use the *connect()* method to open a connection.

---

[1] https://github.com/pytorch/pytorch/issues/13246

References

Example

```
>>> # xdoctest: +REQUIRES(module:sqlalchemy)
>>> from kwcoco.coco_sql_dataset import *  # NOQA
>>> sql_dset, dct_dset = demo()
>>> dset1, dset2 = sql_dset, dct_dset
>>> tag1, tag2 = 'dset1', 'dset2'
>>> assert_dsets_allclose(sql_dset, dct_dset)
```

MEMORY_URI = 'sqlite:///:memory:'

classmethod coerce(*data*)

Create an SQL CocoDataset from the input pointer.

Example

import kwcoco dset = kwcoco.CocoDataset.demo('shapes8') data = dset.fpath self = CocoSqlDatabase.coerce(data)

from kwcoco.coco_sql_dataset import CocoSqlDatabase import kwcoco dset = kwcoco.CocoDataset.coerce('spacenet7.kwcoco.json')

self = CocoSqlDatabase.coerce(dset)

from kwcoco.coco_sql_dataset import CocoSqlDatabase sql_dset = CocoSqlDatabase.coerce('spacenet7.kwcoco.json')

from kwcoco.coco_sql_dataset import CocoSqlDatabase import kwcoco sql_dset = kwcoco.CocoDataset.coerce('_spacenet7.kwcoco.view.v006.sqlite')

disconnect()

Drop references to any SQL or cache objects

connect(*readonly=False*)

Connects this instance to the underlying database.

References

details on read only mode, some of these didnt seem to work https://github.com/sqlalchemy/sqlalchemy/blob/master/lib/sqlalchemy/dialects/sqlite/pysqlite.py#L71 https://github.com/pudo/dataset/issues/136 https://writeonly.wordpress.com/2009/07/16/simple-read-only-sqlalchemy-sessions/

property fpath

delete()

populate_from(*dset*, *verbose=1*)

Copy the information in a `CocoDataset` into this SQL database.

Example

```
>>> # xdoctest: +REQUIRES(module:sqlalchemy)
>>> from kwcoco.coco_sql_dataset import _benchmark_dset_readtime  # NOQA
>>> import kwcoco
>>> from kwcoco.coco_sql_dataset import *
>>> dset2 = dset = kwcoco.CocoDataset.demo()
>>> dset1 = self = CocoSqlDatabase('sqlite:///:memory:')
>>> self.connect()
>>> self.populate_from(dset)
>>> assert_dsets_allclose(dset1, dset2, tag1='sql', tag2='dct')
>>> ti_sql = _benchmark_dset_readtime(dset1, 'sql')
>>> ti_dct = _benchmark_dset_readtime(dset2, 'dct')
>>> print('ti_sql.rankings = {}'.format(ub.repr2(ti_sql.rankings, nl=2,␣
↪precision=6, align=':')))
>>> print('ti_dct.rankings = {}'.format(ub.repr2(ti_dct.rankings, nl=2,␣
↪precision=6, align=':')))
```

property dataset

property anns

property cats

property imgs

property name_to_cat

raw_table(*table_name*)

Loads an entire SQL table as a pandas DataFrame

Parameters

table_name (*str*) – name of the table

Returns

pandas.DataFrame

Example

```
>>> # xdoctest: +REQUIRES(module:sqlalchemy)
>>> from kwcoco.coco_sql_dataset import *  # NOQA
>>> self, dset = demo()
>>> table_df = self.raw_table('annotations')
>>> print(table_df)
```

tabular_targets()

Convinience method to create an in-memory summary of basic annotation properties with minimal SQL overhead.

Example

```
>>> # xdoctest: +REQUIRES(module:sqlalchemy)
>>> from kwcoco.coco_sql_dataset import *  # NOQA
>>> self, dset = demo()
>>> targets = self.tabular_targets()
>>> print(targets.pandas())
```

property bundle_dpath

property data_fpath

data_fpath is an alias of fpath

kwcoco.coco_sql_dataset.**cached_sql_coco_view**(*dct_db_fpath=None*, *sql_db_fpath=None*, *dset=None*, *force_rewrite=False*)

Attempts to load a cached SQL-View dataset, only loading and converting the json dataset if necessary.

kwcoco.coco_sql_dataset.**ensure_sql_coco_view**(*dset*, *db_fpath=None*, *force_rewrite=False*)

Create a cached on-disk SQL view of an on-disk COCO dataset.

Note: This function is fragile. It depends on looking at file modified timestamps to determine if it needs to write the dataset.

---

kwcoco.coco_sql_dataset.**demo**(*num=10*)

kwcoco.coco_sql_dataset.**assert_dsets_allclose**(*dset1*, *dset2*, *tag1='dset1'*, *tag2='dset2'*)

kwcoco.coco_sql_dataset.**devcheck**()

Scratch work for things that should eventually become unit or doc tests

from kwcoco.coco_sql_dataset import * # NOQA self, dset = demo()

2.1.2.10 kwcoco.compat_dataset module

A wrapper around the basic kwcoco dataset with a pycocotools API.

We do not recommend using this API because it has some idiosyncrasies, where names can be missleading and APIs are not always clear / efficient: e.g.

(1) catToImgs returns integer image ids but imgToAnns returns annotation dictionaries.

(2) showAnns takes a dictionary list as an argument instead of an integer list

The cool thing is that this extends the kwcoco API so you can drop this for compatibility with the old API, but you still get access to all of the kwcoco API including dynamic addition / removal of categories / annotations / images.

class kwcoco.compat_dataset.**COCO**(*annotation_file=None*, ***kw*)

Bases: *CocoDataset*

A wrapper around the basic kwcoco dataset with a pycocotools API.

Example

```
>>> from kwcoco.compat_dataset import *  # NOQA
>>> import kwcoco
>>> basic = kwcoco.CocoDataset.demo('shapes8')
>>> self = COCO(basic.dataset)
>>> self.info()
>>> print('self.imgToAnns = {!r}'.format(self.imgToAnns[1]))
>>> print('self.catToImgs = {!r}'.format(self.catToImgs))
```

createIndex()

info()

Print information about the annotation file.

property imgToAnns

property catToImgs

unlike the name implies, this actually goes from category to image ids Name retained for backward compatibility

getAnnIds(*imgIds=[]*, *catIds=[]*, *areaRng=[]*, *iscrowd=None*)

Get ann ids that satisfy given filter conditions. default skips that filter

Parameters

- **imgIds** (*List[int]*) – get anns for given imgs
- **catIds** (*List[int]*) – get anns for given cats
- **areaRng** (*List[float]*) – get anns for given area range (e.g. [0 inf])
- **iscrowd** (*bool*) – get anns for given crowd label (False or True)

Returns

integer array of ann ids

Return type

List[int]

Example

```
>>> from kwcoco.compat_dataset import *  # NOQA
>>> import kwcoco
>>> self = COCO(kwcoco.CocoDataset.demo('shapes8').dataset)
>>> self.getAnnIds()
>>> self.getAnnIds(imgIds=1)
>>> self.getAnnIds(imgIds=[1])
>>> self.getAnnIds(catIds=[3])
```

getCatIds(*catNms=[]*, *supNms=[]*, *catIds=[]*)

filtering parameters. default skips that filter.

Parameters

- **catNms** (*List[str]*) – get cats for given cat names
- **supNms** (*List[str]*) – get cats for given supercategory names

- **catIds** (*List[int]*) – get cats for given cat ids

Returns
integer array of cat ids

Return type
List[int]

Example

```
>>> from kwcoco.compat_dataset import *  # NOQA
>>> import kwcoco
>>> self = COCO(kwcoco.CocoDataset.demo('shapes8').dataset)
>>> self.getCatIds()
>>> self.getCatIds(catNms=['superstar'])
>>> self.getCatIds(supNms=['raster'])
>>> self.getCatIds(catIds=[3])
```

getImgIds(*imgIds=[]*, *catIds=[]*)

Get img ids that satisfy given filter conditions.

Parameters

- **imgIds** (*List[int]*) – get imgs for given ids
- **catIds** (*List[int]*) – get imgs with all given cats

Returns
integer array of img ids

Return type
List[int]

Example

```
>>> from kwcoco.compat_dataset import *  # NOQA
>>> import kwcoco
>>> self = COCO(kwcoco.CocoDataset.demo('shapes8').dataset)
>>> self.getImgIds(imgIds=[1, 2])
>>> self.getImgIds(catIds=[3, 6, 7])
>>> self.getImgIds(catIds=[3, 6, 7], imgIds=[1, 2])
```

loadAnns(*ids=[]*)

Load anns with the specified ids.

Parameters
ids (*List[int]*) – integer ids specifying anns

Returns
loaded ann objects

Return type
List[dict]

loadCats(*ids=[]*)

Load cats with the specified ids.

Parameters
ids (*List[int]*) – integer ids specifying cats

Returns
loaded cat objects

Return type
List[dict]

`loadImgs`(*ids=[]*)

Load anns with the specified ids.

Parameters
ids (*List[int]*) – integer ids specifying img

Returns
loaded img objects

Return type
List[dict]

`showAnns`(*anns*, *draw_bbox=False*)

Display the specified annotations.

Parameters
anns (*List[Dict]*) – annotations to display

`loadRes`(*resFile*)

Load result file and return a result api object.

Parameters
resFile (*str*) – file name of result file

Returns
res result api object

Return type
object

`download`(*tarDir=None*, *imgIds=[]*)

Download COCO images from mscoco.org server.

Parameters

- **tarDir** (*str*) – COCO results directory name
- **imgIds** (*list*) – images to be downloaded

`loadNumpyAnnotations`(*data*)

Convert result data from a numpy array [Nx7] where each row contains {imageID,x1,y1,w,h,score,class}

Parameters
data (*numpy.ndarray*)

Returns
annotations (python nested list)

Return type
List[Dict]

`annToRLE`(*ann*)

Convert annotation which can be polygons, uncompressed RLE to RLE.

Returns
kwimage.Mask

Note:

- This requires the C-extensions for kwimage to be installed (i.e.

`pip install kwimage_ext`) due to the need to interface with the bytes RLE format.

Example

```
>>> from kwcoco.compat_dataset import *  # NOQA
>>> import kwcoco
>>> self = COCO(kwcoco.CocoDataset.demo('shapes8').dataset)
>>> try:
>>>     rle = self.annToRLE(self.anns[1])
>>> except NotImplementedError:
>>>     import pytest
>>>     pytest.skip('missing kwimage c-extensions')
>>> else:
>>>     assert len(rle['counts']) > 2
>>> # xdoctest: +REQUIRES(module:pycocotools)
>>> self.conform(legacy=True)
>>> orig = self._aspycoco().annToRLE(self.anns[1])
```

annToMask(*ann*)

Convert annotation which can be polygons, uncompressed RLE, or RLE to binary mask.

Returns
binary mask (numpy 2D array)

Return type
ndarray

Note: The mask is returned as a fortran (F-style) array with the same dimensions as the parent image.

2.1.2.11 kwcoco.exceptions module

exception kwcoco.exceptions.**AddError**

Bases: ValueError

Generic error when trying to add a category/annotation/image

exception kwcoco.exceptions.**DuplicateAddError**

Bases: ValueError

Error when trying to add a duplicate item

exception kwcoco.exceptions.**InvalidAddError**

Bases: ValueError

Error when trying to invalid data

exception kwcoco.exceptions.**CoordinateCompatibilityError**

Bases: ValueError

Error when trying to perform operations on data in different coordinate systems.

2.1.2.12 kwcoco.kpf module

WIP:

Conversions to and from KPF format.

kwcoco.kpf.**coco_to_kpf**(*coco_dset*)

import kwcoco coco_dset = kwcoco.CocoDataset.demo('shapes8')

kwcoco.kpf.**demo**()

2.1.2.13 kwcoco.kw18 module

A helper for converting COCO to / from KW18 format.

KW18 File Format https://docs.google.com/spreadsheets/d/1DFCwoTKnDv8qfy3raM7QXtir2Fjfj9j8-z8px5Bu0q8/edit#gid=10

The kw18.trk files are text files, space delimited; each row is one frame of one track and all rows have the same number of columns. The fields are:

```
01) track_ID         : identifies the track
02) num_frames:     number of frames in the track
03) frame_id         : frame number for this track sample
04) loc_x         : X-coordinate of the track (image/ground coords)
05) loc_y         : Y-coordinate of the track (image/ground coords)
06) vel_x         : X-velocity of the object (image/ground coords)
07) vel_y         : Y-velocity of the object (image/ground coords)
08) obj_loc_x        : X-coordinate of the object (image coords)
09) obj_loc_y        : Y-coordinate of the object (image coords)
10) bbox_min_x    : minimum X-coordinate of bounding box (image coords)
11) bbox_min_y    : minimum Y-coordinate of bounding box (image coords)
12) bbox_max_x    : maximum X-coordinate of bounding box (image coords)
13) bbox_max_y    : maximum Y-coordinate of bounding box (image coords)
14) area         : area of object (pixels)
15) world_loc_x    : X-coordinate of object in world
16) world_loc_y    : Y-coordinate of object in world
17) world_loc_z    : Z-coordiante of object in world
18) timestamp        : timestamp of frame (frames)
For the location and velocity of object centroids, use fields 4-7.
Bounding box is specified using coordinates of the top-left and bottom
right corners. Fields 15-17 may be ignored.

The kw19.trk and kw20.trk files, when present, add the following field(s):
19) object class: estimated class of the object, either 1 (person), 2
(vehicle), or 3 (other).
20) Activity ID -- refer to activities.txt for index and list of activities.
```

class kwcoco.kw18.**KW18**(*data*)

Bases: DataFrameArray

A DataFrame like object that stores KW18 column data

Example

```
>>> import kwcoco
>>> from kwcoco.kw18 import KW18
>>> coco_dset = kwcoco.CocoDataset.demo('shapes')
>>> kw18_dset = KW18.from_coco(coco_dset)
>>> print(kw18_dset.pandas())
```

DEFAULT_COLUMNS = ['track_id', 'track_length', 'frame_number', 'tracking_plane_loc_x', 'tracking_plane_loc_y', 'velocity_x', 'velocity_y', 'image_loc_x', 'image_loc_y', 'img_bbox_tl_x', 'img_bbox_tl_y', 'img_bbox_br_x', 'img_bbox_br_y', 'area', 'world_loc_x', 'world_loc_y', 'world_loc_z', 'timestamp', 'confidence', 'object_type_id', 'activity_type_id']

classmethod demo()

classmethod from_coco(*coco_dset*)

to_coco(*image_paths=None*, *video_name=None*)

Translates a kw18 files to a CocoDataset.

Note: kw18 does not contain complete information, and as such the returned coco dataset may need to be augmented.

Parameters

- **image_paths** (*Dict[int, str], default=None*) – if specified, maps frame numbers to image file paths.
- **video_name** (*str, default=None*) – if specified records the name of the video this kw18 belongs to

---

Todo:

- [X] allow kwargs to specify path to frames / videos

Example

```
>>> from kwcoco.kw18 import KW18
>>> from os.path import join
>>> import ubelt as ub
>>> import kwimage
>>> # Prep test data - autogen a demo kw18 and write it to disk
>>> dpath = ub.ensure_app_cache_dir('kwcoco/kw18')
>>> kw18_fpath = join(dpath, 'test.kw18')
```

(continues on next page)

(continued from previous page)

```
>>> KW18.demo().dump(kw18_fpath)
>>> #
>>> # Load the kw18 file
>>> self = KW18.load(kw18_fpath)
>>> # Pretend that these image correspond to kw18 frame numbers
>>> frame_names = [kwimage.grab_test_image_fpath(k) for k in kwimage.grab_test_
↪image.keys()]
>>> frame_ids = sorted(set(self['frame_number']))
>>> image_paths = dict(zip(frame_ids, frame_names))
>>> #
>>> # Convert the kw18 to kwcoco and specify paths to images
>>> coco_dset = self.to_coco(image_paths=image_paths, video_name='dummy.mp4')
>>> #
>>> # Now we can draw images
>>> canvas = coco_dset.draw_image(1)
>>> # xdoctest: +REQUIRES(--draw)
>>> kwimage.imwrite('foo.jpg', canvas)
>>> # Draw all iamges
>>> for gid in coco_dset.imgs.keys():
>>>     canvas = coco_dset.draw_image(gid)
>>>     fpath = join(dpath, 'gid_{}.jpg'.format(gid))
>>>     print('write fpath = {!r}'.format(fpath))
>>>     kwimage.imwrite(fpath, canvas)
```

classmethod load(*file*)

Example

```
>>> import kwcoco
>>> from kwcoco.kw18 import KW18
>>> coco_dset = kwcoco.CocoDataset.demo('shapes')
>>> kw18_dset = KW18.from_coco(coco_dset)
>>> print(kw18_dset.pandas())
```

classmethod loads(*text*)

Example

```
>>> self = KW18.demo()
>>> text = self.dumps()
>>> self2 = KW18.loads(text)
>>> empty = KW18.loads('')
```

dump(*file*)

dumps()

Example

```
>>> self = KW18.demo()
>>> text = self.dumps()
>>> print(text)
```

2.1.2.14 kwcoco.sensorchan_spec module

This is an extension of *kwcoco.channel_spec*, which augments channel information with an associated sensor attribute. Eventually, this will entirely replace the channel spec.

Example

```
>>> # xdoctest: +REQUIRES(module:lark)
>>> # hack for 3.6
>>> from kwcoco import sensorchan_spec
>>> import kwcoco
>>> kwcoco.SensorChanSpec = sensorchan_spec.SensorChanSpec
>>> self = kwcoco.SensorChanSpec.coerce('sensor0:B1|B8|B8a|B10|B11,sensor1:B11|X.2|Y:2:6,
↪sensor2:r|g|b|disparity|gauss|B8|B11,sensor3:r|g|b|flowx|flowy|distri|B10|B11')
>>> self.normalize()
```

class kwcoco.sensorchan_spec.**SensorSpec**(*spec*)

Bases: NiceRepr

A simple wrapper for sensors in case we want to do anything fancy with them later. For now they are just a string.

class kwcoco.sensorchan_spec.**SensorChanSpec**(*spec: str*)

Bases: NiceRepr

The public facing API for the sensor / channel specification

Example

```
>>> # xdoctest: +REQUIRES(module:lark)
>>> from kwcoco.sensorchan_spec import SensorChanSpec
>>> self = SensorChanSpec('(L8,S2):BGR,WV:BGR,S2:nir,L8:land.0:4')
>>> s1 = self.normalize()
>>> s2 = self.concise()
>>> streams = self.streams()
>>> print(s1)
>>> print(s2)
>>> print('streams = {}'.format(ub.repr2(streams, sv=1, nl=1)))
L8:BGR,S2:BGR,WV:BGR,S2:nir,L8:land.0|land.1|land.2|land.3
(L8,S2,WV):BGR,L8:land:4,S2:nir
streams = [
    L8:BGR,
    S2:BGR,
    WV:BGR,
    S2:nir,
```

(continues on next page)

(continued from previous page)

```
    L8:land.0|land.1|land.2|land.3,
]
```

Example

```
>>> # Check with generic sensors
>>> # xdoctest: +REQUIRES(module:lark)
>>> from kwcoco.sensorchan_spec import SensorChanSpec
>>> import kwcoco
>>> self = SensorChanSpec('(*):BGR,*:BGR,*:nir,*:land.0:4')
>>> self.concise().normalize()
>>> s1 = self.normalize()
>>> s2 = self.concise()
>>> print(s1)
>>> print(s2)
*:BGR,*:BGR,*:nir,*:land.0|land.1|land.2|land.3
(*,*):BGR,*:(nir,land:4)
>>> import kwcoco
>>> c = kwcoco.ChannelSpec.coerce('BGR,BGR,nir,land.0:8')
>>> c1 = c.normalize()
>>> c2 = c.concise()
>>> print(c1)
>>> print(c2)
```

Example

```
>>> # Check empty channels
>>> # xdoctest: +REQUIRES(module:lark)
>>> from kwcoco.sensorchan_spec import SensorChanSpec
>>> import kwcoco
>>> print(SensorChanSpec('*:').normalize())
*:
>>> print(SensorChanSpec('sen:').normalize())
sen:
>>> print(SensorChanSpec('sen:').normalize().concise())
sen:
>>> print(SensorChanSpec('sen:').concise().normalize().concise())
sen:
```

classmethod coerce(*data*)

Attempt to interpret the data as a channel specification

Returns

SensorChanSpec

Example

```
>>> # xdoctest: +REQUIRES(module:lark)
>>> from kwcoco.sensorchan_spec import *  # NOQA
>>> from kwcoco.sensorchan_spec import SensorChanSpec
>>> data = SensorChanSpec.coerce(3)
>>> assert SensorChanSpec.coerce(data).normalize().spec == '*:u0|u1|u2'
>>> data = SensorChanSpec.coerce(3)
>>> assert data.spec == 'u0|u1|u2'
>>> assert SensorChanSpec.coerce(data).spec == 'u0|u1|u2'
>>> data = SensorChanSpec.coerce('u:3')
>>> assert data.normalize().spec == '*:u.0|u.1|u.2'
```

normalize()

concise()

streams()

Returns
List of sensor-names and fused channel specs

Return type
List[*FusedSensorChanSpec*]

late_fuse(**others*)

Example

```
>>> # xdoctest: +REQUIRES(module:lark)
>>> import kwcoco
>>> from kwcoco import sensorchan_spec
>>> import kwcoco
>>> kwcoco.SensorChanSpec = sensorchan_spec.SensorChanSpec  # hack for 3.6
>>> a = kwcoco.SensorChanSpec.coerce('A|B|C,edf')
>>> b = kwcoco.SensorChanSpec.coerce('A12')
>>> c = kwcoco.SensorChanSpec.coerce('')
>>> d = kwcoco.SensorChanSpec.coerce('rgb')
>>> print(a.late_fuse(b).spec)
>>> print((a + b).spec)
>>> print((b + a).spec)
>>> print((a + b + c).spec)
>>> print(sum([a, b, c, d]).spec)
A|B|C,edf,A12
A|B|C,edf,A12
A12,A|B|C,edf
A|B|C,edf,A12
A|B|C,edf,A12,rgb
>>> import kwcoco
>>> a = kwcoco.SensorChanSpec.coerce('A|B|C,edf').normalize()
>>> b = kwcoco.SensorChanSpec.coerce('A12').normalize()
>>> c = kwcoco.SensorChanSpec.coerce('').normalize()
>>> d = kwcoco.SensorChanSpec.coerce('rgb').normalize()
```

(continues on next page)

(continued from previous page)

```
>>> print(a.late_fuse(b).spec)
>>> print((a + b).spec)
>>> print((b + a).spec)
>>> print((a + b + c).spec)
>>> print(sum([a, b, c, d]).spec)
*:A|B|C,*:edf,*:A12
*:A|B|C,*:edf,*:A12
*:A12,*:A|B|C,*:edf
*:A|B|C,*:edf,*:A12,*:
*:A|B|C,*:edf,*:A12,*:,*:rgb
>>> print((a.late_fuse(b)).concise())
>>> print(((a + b)).concise())
>>> print(((b + a)).concise())
>>> print(((a + b + c)).concise())
>>> print((sum([a, b, c, d])).concise())
*:(A|B|C,edf,A12)
*:(A|B|C,edf,A12)
*:(A12,A|B|C,edf)
*:(A|B|C,edf,A12,)
*:(A|B|C,edf,A12,,r|g|b)
```

Example

```
>>> # Test multi-arg case
>>> import kwcoco
>>> a = kwcoco.SensorChanSpec.coerce('A|B|C,edf')
>>> b = kwcoco.SensorChanSpec.coerce('A12')
>>> c = kwcoco.SensorChanSpec.coerce('')
>>> d = kwcoco.SensorChanSpec.coerce('rgb')
>>> others = [b, c, d]
>>> print(a.late_fuse(*others).spec)
>>> print(kwcoco.SensorChanSpec.late_fuse(a, b, c, d).spec)
A|B|C,edf,A12,rgb
A|B|C,edf,A12,rgb
```

matching_sensor(*sensor*)

Get the components corresponding to a specific sensor

Parameters

sensor (*str*) – the name of the sensor to match

Example

```
>>> # xdoctest: +REQUIRES(module:lark)
>>> import kwcoco
>>> self = kwcoco.SensorChanSpec.coerce('(S1,S2):(a|b|c),S2:c|d|e')
>>> sensor = 'S2'
>>> new = self.matching_sensor(sensor)
>>> print(f'new={new}')
new=S2:a|b|c,S2:c|d|e
```

(continues on next page)

(continued from previous page)

```
>>> print(self.matching_sensor('S1'))
S1:a|b|c
>>> print(self.matching_sensor('S3'))
S3:
```

property chans

Returns the channel-only spec, ONLY if all of the sensors are the same

Example

```
>>> # xdoctest: +REQUIRES(module:lark)
>>> import kwcoco
>>> self = kwcoco.SensorChanSpec.coerce('(S1,S2):(a|b|c),S2:c|d|e')
>>> import pytest
>>> with pytest.raises(Exception):
>>>     self.chans
>>> print(self.matching_sensor('S1').chans.spec)
>>> print(self.matching_sensor('S2').chans.spec)
a|b|c
a|b|c,c|d|e
```

class kwcoco.sensorchan_spec.**FusedSensorChanSpec**(*sensor*, *chans*)

Bases: *SensorChanSpec*

A single sensor a corresponding fused channels.

property chans

property spec

class kwcoco.sensorchan_spec.**SensorChanNode**(*sensor*, *chan*)

Bases: object

TODO: just replace this with the spec class itself?

property spec

class kwcoco.sensorchan_spec.**FusedChanNode**(*chan*)

Bases: object

TODO: just replace this with the spec class itself?

Example

s = FusedChanNode('a|b|c.0|c.1|c.2') c = s.concise() print(s) print(c)

property spec

concise()

class kwcoco.sensorchan_spec.**SensorChanTransformer**(*concise_channels=1*, *concise_sensors=1*)

Bases: Transformer

Given a parsed tree for a sensor-chan spec, can transform it into useful forms.

Todo: Make the classes that hold the underlying data more robust such that they either use the existing channel spec or entirely replace it. (probably the former). Also need to add either a FusedSensorChan node that is restircted to only a single sensor and group of fused channels.

---

chan_id(*items*)

chan_single(*items*)

chan_getitem(*items*)

chan_getslice_0b(*items*)

chan_getslice_ab(*items*)

chan_code(*items*)

sensor_seq(*items*)

fused_seq(*items*)

fused(*items*)

channel_rhs(*items*)

sensor_lhs(*items*)

nosensor_chan(*items*)

sensor_chan(*items*)

stream_item(*items*)

stream(*items*)

kwcoco.sensorchan_spec.**normalize_sensor_chan**(*spec*)

Example

```
>>> # xdoctest: +REQUIRES(module:lark)
>>> from kwcoco.sensorchan_spec import *  # NOQA
>>> spec = 'L8:mat:4,L8:red,S2:red,S2:forest|brush,S2:mat.0|mat.1|mat.2|mat.3'
>>> r1 = normalize_sensor_chan(spec)
>>> spec = 'L8:r|g|b,L8:r|g|b'
>>> r2 = normalize_sensor_chan(spec)
>>> print(f'r1={r1}')
>>> print(f'r2={r2}')
r1=L8:mat.0|mat.1|mat.2|mat.3,L8:red,S2:red,S2:forest|brush,S2:mat.0|mat.1|mat.
↪2|mat.3
r2=L8:r|g|b,L8:r|g|b
```

kwcoco.sensorchan_spec.**concise_sensor_chan**(*spec*)

Example

```
>>> # xdoctest: +REQUIRES(module:lark)
>>> from kwcoco.sensorchan_spec import *  # NOQA
>>> spec = 'L8:mat.0|mat.1|mat.2|mat.3,L8:red,S2:red,S2:forest|brush,S2:mat.0|mat.
↪1|mat.2|mat.3'
>>> concise_spec = concise_sensor_chan(spec)
>>> normed_spec = normalize_sensor_chan(concise_spec)
>>> concise_spec2 = concise_sensor_chan(normed_spec)
>>> assert concise_spec2 == concise_spec
>>> print(concise_spec)
(L8,S2):(mat:4,red),S2:forest|brush
```

kwcoco.sensorchan_spec.**sensorchan_concise_parts**(*spec*)

kwcoco.sensorchan_spec.**sensorchan_normalized_parts**(*spec*)

2.1.3 Module contents

The Kitware COCO module defines a variant of the Microsoft COCO format, originally developed for the "collected images in context" object detection challenge. We are backwards compatible with the original module, but we also have improved implementations in several places, including segmentations, keypoints, annotation tracks, multi-spectral images, and videos (which represents a generic sequence of images).

A kwcoco file is a "manifest" that serves as a single reference that points to all images, categories, and annotations in a computer vision dataset. Thus, when applying an algorithm to a dataset, it is sufficient to have the algorithm take one dataset parameter: the path to the kwcoco file. Generally a kwcoco file will live in a "bundle" directory along with the data that it references, and paths in the kwcoco file will be relative to the location of the kwcoco file itself.

The main data structure in this model is largely based on the implementation in https://github.com/cocodataset/cocoapi It uses the same efficient core indexing data structures, but in our implementation the indexing can be optionally turned off, functions are silent by default (with the exception of long running processes, which optionally show progress by default). We support helper functions that add and remove images, categories, and annotations.

The *kwcoco.CocoDataset* class is capable of dynamic addition and removal of categories, images, and annotations. Has better support for keypoints and segmentation formats than the original COCO format. Despite being written in Python, this data structure is reasonably efficient.

```
>>> import kwcoco
>>> import json
>>> # Create demo data
>>> demo = kwcoco.CocoDataset.demo()
>>> # Reroot can switch between absolute / relative-paths
>>> demo.reroot(absolute=True)
>>> # could also use demo.dump / demo.dumps, but this is more explicit
>>> text = json.dumps(demo.dataset)
>>> with open('demo.json', 'w') as file:
>>>     file.write(text)

>>> # Read from disk
>>> self = kwcoco.CocoDataset('demo.json')

>>> # Add data
>>> cid = self.add_category('Cat')
```

(continues on next page)

(continued from previous page)

```
>>> gid = self.add_image('new-img.jpg')
>>> aid = self.add_annotation(image_id=gid, category_id=cid, bbox=[0, 0, 100, 100])

>>> # Remove data
>>> self.remove_annotations([aid])
>>> self.remove_images([gid])
>>> self.remove_categories([cid])

>>> # Look at data
>>> import ubelt as ub
>>> print(ub.repr2(self.basic_stats(), nl=1))
>>> print(ub.repr2(self.extended_stats(), nl=2))
>>> print(ub.repr2(self.boxsize_stats(), nl=3))
>>> print(ub.repr2(self.category_annotation_frequency()))


>>> # Inspect data
>>> # xdoctest: +REQUIRES(module:kwplot)
>>> import kwplot
>>> kwplot.autompl()
>>> self.show_image(gid=1)

>>> # Access single-item data via imgs, cats, anns
>>> cid = 1
>>> self.cats[cid]
{'id': 1, 'name': 'astronaut', 'supercategory': 'human'}

>>> gid = 1
>>> self.imgs[gid]
{'id': 1, 'file_name': '...astro.png', 'url': 'https://i.imgur.com/KXhKM72.png'}

>>> aid = 3
>>> self.anns[aid]
{'id': 3, 'image_id': 1, 'category_id': 3, 'line': [326, 369, 500, 500]}

>>> # Access multi-item data via the annots and images helper objects
>>> aids = self.index.gid_to_aids[2]
>>> annots = self.annots(aids)

>>> print('annots = {}'.format(ub.repr2(annots, nl=1, sv=1)))
annots = <Annots(num=2)>

>>> annots.lookup('category_id')
[6, 4]

>>> annots.lookup('bbox')
[[37, 6, 230, 240], [124, 96, 45, 18]]

>>> # built in conversions to efficient kwimage array DataStructures
>>> print(ub.repr2(annots.detections.data, sv=1))
{
    'boxes': <Boxes(xywh,
```

(continues on next page)

(continued from previous page)

```
                array([[ 37.,   6., 230., 240.],
                       [124.,  96.,  45.,  18.]], dtype=float32))>,
    'class_idxs': [5, 3],
    'keypoints': <PointsList(n=2)>,
    'segmentations': <PolygonList(n=2)>,
}

>>> gids = list(self.imgs.keys())
>>> images = self.images(gids)
>>> print('images = {}'.format(ub.repr2(images, nl=1, sv=1)))
images = <Images(num=3)>

>>> images.lookup('file_name')
['...astro.png', '...carl.png', '...stars.png']

>>> print('images.annots = {}'.format(images.annots))
images.annots = <AnnotGroups(n=3, m=3.7, s=3.9)>

>>> print('images.annots.cids = {!r}'.format(images.annots.cids))
images.annots.cids = [[1, 2, 3, 4, 5, 5, 5, 5, 5], [6, 4], []]
```

2.1.3.1 CocoDataset API

The following is a logical grouping of the public kwcoco.CocoDataset API attributes and methods. See the in-code documentation for further details.

2.1.3.1.1 CocoDataset classmethods (via MixinCocoExtras)

- *kwcoco.CocoDataset.coerce* - Attempt to transform the input into the intended CocoDataset.
- *kwcoco.CocoDataset.demo* - Create a toy coco dataset for testing and demo puposes
- *kwcoco.CocoDataset.random* - Creates a random CocoDataset according to distribution parameters

2.1.3.1.2 CocoDataset classmethods (via CocoDataset)

- *kwcoco.CocoDataset.from_coco_paths* - Constructor from multiple coco file paths.
- *kwcoco.CocoDataset.from_data* - Constructor from a json dictionary
- *kwcoco.CocoDataset.from_image_paths* - Constructor from a list of images paths.

2.1.3.1.3 CocoDataset slots

- kwcoco.CocoDataset.index - an efficient lookup index into the coco data structure. The index defines its own attributes like anns, cats, imgs, gid_to_aids, file_name_to_img, etc. See CocoIndex for more details on which attributes are available.
- kwcoco.CocoDataset.hashid - If computed, this will be a hash uniquely identifing the dataset. To ensure this is computed see kwcoco.coco_dataset.MixinCocoExtras._build_hashid().
- kwcoco.CocoDataset.hashid_parts -
- kwcoco.CocoDataset.tag - A tag indicating the name of the dataset.
- kwcoco.CocoDataset.dataset - raw json data structure. This is the base dictionary that contains {'annotations': List, 'images': List, 'categories': List}
- kwcoco.CocoDataset.bundle_dpath - If known, this is the root path that all image file names are relative to. This can also be manually overwritten by the user.
- kwcoco.CocoDataset.assets_dpath -
- kwcoco.CocoDataset.cache_dpath -

2.1.3.1.4 CocoDataset properties

- kwcoco.CocoDataset.anns -
- kwcoco.CocoDataset.cats -
- kwcoco.CocoDataset.cid_to_aids -
- kwcoco.CocoDataset.data_fpath -
- kwcoco.CocoDataset.data_root -
- *kwcoco.CocoDataset.fpath* - if known, this stores the filepath the dataset was loaded from
- kwcoco.CocoDataset.gid_to_aids -
- kwcoco.CocoDataset.img_root -
- kwcoco.CocoDataset.imgs -
- kwcoco.CocoDataset.n_annots -
- kwcoco.CocoDataset.n_cats -
- kwcoco.CocoDataset.n_images -
- kwcoco.CocoDataset.n_videos -
- kwcoco.CocoDataset.name_to_cat -

2.1.3.1.5 CocoDataset methods (via MixinCocoAddRemove)

- *kwcoco.CocoDataset.add_annotation* - Add an annotation to the dataset (dynamically updates the index)
- *kwcoco.CocoDataset.add_annotations* - Faster less-safe multi-item alternative to add_annotation.
- *kwcoco.CocoDataset.add_category* - Adds a category
- *kwcoco.CocoDataset.add_image* - Add an image to the dataset (dynamically updates the index)
- *kwcoco.CocoDataset.add_images* - Faster less-safe multi-item alternative
- *kwcoco.CocoDataset.add_video* - Add a video to the dataset (dynamically updates the index)
- *kwcoco.CocoDataset.clear_annotations* - Removes all annotations (but not images and categories)
- *kwcoco.CocoDataset.clear_images* - Removes all images and annotations (but not categories)
- *kwcoco.CocoDataset.ensure_category* - Like `add_category()`, but returns the existing category id if it already exists instead of failing. In this case all metadata is ignored.
- *kwcoco.CocoDataset.ensure_image* - Like `add_image()`,, but returns the existing image id if it already exists instead of failing. In this case all metadata is ignored.
- *kwcoco.CocoDataset.remove_annotation* - Remove a single annotation from the dataset
- *kwcoco.CocoDataset.remove_annotation_keypoints* - Removes all keypoints with a particular category
- *kwcoco.CocoDataset.remove_annotations* - Remove multiple annotations from the dataset.
- *kwcoco.CocoDataset.remove_categories* - Remove categories and all annotations in those categories. Currently does not change any hierarchy information
- *kwcoco.CocoDataset.remove_images* - Remove images and any annotations contained by them
- *kwcoco.CocoDataset.remove_keypoint_categories* - Removes all keypoints of a particular category as well as all annotation keypoints with those ids.
- *kwcoco.CocoDataset.remove_videos* - Remove videos and any images / annotations contained by them
- *kwcoco.CocoDataset.set_annotation_category* - Sets the category of a single annotation

2.1.3.1.6 CocoDataset methods (via MixinCocoObjects)

- *kwcoco.CocoDataset.annots* - Return vectorized annotation objects
- *kwcoco.CocoDataset.categories* - Return vectorized category objects
- *kwcoco.CocoDataset.images* - Return vectorized image objects
- *kwcoco.CocoDataset.videos* - Return vectorized video objects

2.1.3.1.7 CocoDataset methods (via MixinCocoStats)

- *kwcoco.CocoDataset.basic_stats* - Reports number of images, annotations, and categories.
- *kwcoco.CocoDataset.boxsize_stats* - Compute statistics about bounding box sizes.
- *kwcoco.CocoDataset.category_annotation_frequency* - Reports the number of annotations of each category
- *kwcoco.CocoDataset.category_annotation_type_frequency* - Reports the number of annotations of each type for each category
- *kwcoco.CocoDataset.conform* - Make the COCO file conform a stricter spec, infers attibutes where possible.
- *kwcoco.CocoDataset.extended_stats* - Reports number of images, annotations, and categories.
- *kwcoco.CocoDataset.find_representative_images* - Find images that have a wide array of categories. Attempt to find the fewest images that cover all categories using images that contain both a large and small number of annotations.
- *kwcoco.CocoDataset.keypoint_annotation_frequency* -
- *kwcoco.CocoDataset.stats* - This function corresponds to *kwcoco.cli.coco_stats*.
- *kwcoco.CocoDataset.validate* - Performs checks on this coco dataset.

2.1.3.1.8 CocoDataset methods (via MixinCocoAccessors)

- *kwcoco.CocoDataset.category_graph* - Construct a networkx category hierarchy
- *kwcoco.CocoDataset.delayed_load* - Experimental method
- *kwcoco.CocoDataset.get_auxiliary_fpath* - Returns the full path to auxiliary data for an image
- *kwcoco.CocoDataset.get_image_fpath* - Returns the full path to the image
- *kwcoco.CocoDataset.keypoint_categories* - Construct a consistent CategoryTree representation of keypoint classes
- *kwcoco.CocoDataset.load_annot_sample* - Reads the chip of an annotation. Note this is much less efficient than using a sampler, but it doesn't require disk cache.
- *kwcoco.CocoDataset.load_image* - Reads an image from disk and
- *kwcoco.CocoDataset.object_categories* - Construct a consistent CategoryTree representation of object classes

2.1.3.1.9 CocoDataset methods (via CocoDataset)

- *kwcoco.CocoDataset.copy* - Deep copies this object
- *kwcoco.CocoDataset.dump* - Writes the dataset out to the json format
- *kwcoco.CocoDataset.dumps* - Writes the dataset out to the json format
- *kwcoco.CocoDataset.subset* - Return a subset of the larger coco dataset by specifying which images to port. All annotations in those images will be taken.
- *kwcoco.CocoDataset.union* - Merges multiple *CocoDataset* items into one. Names and associations are retained, but ids may be different.

- *kwcoco.CocoDataset.view_sql* - Create a cached SQL interface to this dataset suitable for large scale multiprocessing use cases.

2.1.3.1.10 CocoDataset methods (via MixinCocoExtras)

- *kwcoco.CocoDataset.corrupted_images* - Check for images that don't exist or can't be opened
- *kwcoco.CocoDataset.missing_images* - Check for images that don't exist
- *kwcoco.CocoDataset.rename_categories* - Rename categories with a potentially coarser categorization.
- *kwcoco.CocoDataset.reroot* - Rebase image/data paths onto a new image/data root.

2.1.3.1.11 CocoDataset methods (via MixinCocoDraw)

- *kwcoco.CocoDataset.draw_image* - Use kwimage to draw all annotations on an image and return the pixels as a numpy array.
- *kwcoco.CocoDataset.imread* - Loads a particular image
- *kwcoco.CocoDataset.show_image* - Use matplotlib to show an image with annotations overlaid

class kwcoco.**AbstractCocoDataset**

Bases: ABC

This is a common base for all variants of the Coco Dataset

At the time of writing there is kwcoco.CocoDataset (which is the dictionary-based backend), and the kwcoco.coco_sql_dataset.CocoSqlDataset, which is experimental.

class kwcoco.**CategoryTree**(*graph=None*, *checks=True*)

Bases: NiceRepr

Wrapper that maintains flat or hierarchical category information.

Helps compute softmaxes and probabilities for tree-based categories where a directed edge (A, B) represents that A is a superclass of B.

Note: There are three basic properties that this object maintains:

```
node:
    Alphanumeric string names that should be generally descriptive.
    Using spaces and special characters in these names is
    discouraged, but can be done.  This is the COCO category "name"
    attribute.  For categories this may be denoted as (name, node,
    cname, catname).

id:
    The integer id of a category should ideally remain consistent.
    These are often given by a dataset (e.g. a COCO dataset).  This
    is the COCO category "id" attribute. For categories this is
    often denoted as (id, cid).

index:
    Contigous zero-based indices that indexes the list of
    categories.  These should be used for the fastest access in
```

(continues on next page)

(continued from previous page)

```
    backend computation tasks. Typically corresponds to the
    ordering of the channels in the final linear layer in an
    associated model.  For categories this is often denoted as
    (index, cidx, idx, or cx).
```

Variables

- **idx_to_node** (*List[str]*) – a list of class names. Implicitly maps from index to category name.
- **id_to_node** (*Dict[int, str]*) – maps integer ids to category names
- **node_to_id** (*Dict[str, int]*) – maps category names to ids
- **node_to_idx** (*Dict[str, int]*) – maps category names to indexes
- **graph** (*networkx.Graph*) – a Graph that stores any hierarchy information. For standard mutually exclusive classes, this graph is edgeless. Nodes in this graph can maintain category attributes / properties.
- **idx_groups** (*List[List[int]]*) – groups of category indices that share the same parent category.

Example

```
>>> from kwcoco.category_tree import *
>>> graph = nx.from_dict_of_lists({
>>>     'background': [],
>>>     'foreground': ['animal'],
>>>     'animal': ['mammal', 'fish', 'insect', 'reptile'],
>>>     'mammal': ['dog', 'cat', 'human', 'zebra'],
>>>     'zebra': ['grevys', 'plains'],
>>>     'grevys': ['fred'],
>>>     'dog': ['boxer', 'beagle', 'golden'],
>>>     'cat': ['maine coon', 'persian', 'sphynx'],
>>>     'reptile': ['bearded dragon', 't-rex'],
>>> }, nx.DiGraph)
>>> self = CategoryTree(graph)
>>> print(self)
<CategoryTree(nNodes=22, maxDepth=6, maxBreadth=4...)>
```

Example

```
>>> # The coerce classmethod is the easiest way to create an instance
>>> import kwcoco
>>> kwcoco.CategoryTree.coerce(['a', 'b', 'c'])
<CategoryTree...nNodes=3, nodes=...'a', 'b', 'c'...
>>> kwcoco.CategoryTree.coerce(4)
<CategoryTree...nNodes=4, nodes=...'class_1', 'class_2', 'class_3', ...
>>> kwcoco.CategoryTree.coerce(4)
```

copy()

classmethod from_mutex(*nodes*, *bg_hack=True*)

Parameters
nodes (*List[str]*) – or a list of class names (in which case they will all be assumed to be mutually exclusive)

Example

```
>>> print(CategoryTree.from_mutex(['a', 'b', 'c']))
<CategoryTree(nNodes=3, ...)>
```

classmethod from_json(*state*)

Parameters
state (*Dict*) – see __getstate__ / __json__ for details

classmethod from_coco(*categories*)

Create a CategoryTree object from coco categories

Parameters
List[Dict] – list of coco-style categories

classmethod coerce(*data*, ***kw*)

Attempt to coerce data as a CategoryTree object.

This is primarily useful for when the software stack depends on categories being represent

This will work if the input data is a specially formatted json dict, a list of mutually exclusive classes, or if it is already a CategoryTree. Otherwise an error will be thrown.

Parameters

- **data** (*object*) – a known representation of a category tree.
- ****kwargs** – input type specific arguments

Returns
self

Return type
CategoryTree

Raises

- **TypeError - if the input format is unknown** –
- **ValueError - if kwargs are not compatible with the input format** –

Example

```
>>> import kwcoco
>>> classes1 = kwcoco.CategoryTree.coerce(3)  # integer
>>> classes2 = kwcoco.CategoryTree.coerce(classes1.__json__())  # graph dict
>>> classes3 = kwcoco.CategoryTree.coerce(['class_1', 'class_2', 'class_3'])  #␣
↪mutex list
>>> classes4 = kwcoco.CategoryTree.coerce(classes1.graph)  # nx Graph
>>> classes5 = kwcoco.CategoryTree.coerce(classes1)  # cls
>>> # xdoctest: +REQUIRES(module:ndsampler)
>>> import ndsampler
>>> classes6 = ndsampler.CategoryTree.coerce(3)
>>> classes7 = ndsampler.CategoryTree.coerce(classes1)
>>> classes8 = kwcoco.CategoryTree.coerce(classes6)
```

classmethod demo(*key='coco'*, ***kwargs*)

Parameters

key (*str*) – specify which demo dataset to use. Can be 'coco' (which uses the default coco demo data). Can be 'btree' which creates a binary tree and accepts kwargs 'r' and 'h' for branching-factor and height. Can be 'btree2', which is the same as btree but returns strings

CommandLine

```
xdoctest -m ~/code/kwcoco/kwcoco/category_tree.py CategoryTree.demo
```

Example

```
>>> from kwcoco.category_tree import *
>>> self = CategoryTree.demo()
>>> print('self = {}'.format(self))
self = <CategoryTree(nNodes=10, maxDepth=2, maxBreadth=4...)>
```

to_coco()

Converts to a coco-style data structure

Yields

Dict – coco category dictionaries

property id_to_idx

Example:

```
>>> import kwcoco
>>> self = kwcoco.CategoryTree.demo()
>>> self.id_to_idx[1]
```

property idx_to_id

Example:

```
>>> import kwcoco
>>> self = kwcoco.CategoryTree.demo()
>>> self.idx_to_id[0]
```

idx_to_ancestor_idxs(*include_self=True*)

Mapping from a class index to its ancestors

Parameters
include_self (*bool, default=True*) – if True includes each node as its own ancestor.

idx_to_descendants_idxs(*include_self=False*)

Mapping from a class index to its descendants (including itself)

Parameters
include_self (*bool, default=False*) – if True includes each node as its own descendant.

idx_pairwise_distance()

Get a matrix encoding the distance from one class to another.

Distances

- from parents to children are positive (descendants),
- from children to parents are negative (ancestors),
- between unreachable nodes (wrt to forward and reverse graph) are nan.

is_mutex()

Returns True if all categories are mutually exclusive (i.e. flat)

If true, then the classes may be represented as a simple list of class names without any loss of information, otherwise the underlying category graph is necessary to preserve all knowledge.

Todo:

- [ ] what happens when we have a dummy root?

---

property num_classes

property class_names

property category_names

property cats

Returns a mapping from category names to category attributes.

If this category tree was constructed from a coco-dataset, then this will contain the coco category attributes.

Returns
Dict[str, Dict[str, object]]

Example

```
>>> from kwcoco.category_tree import *
>>> self = CategoryTree.demo()
>>> print('self.cats = {!r}'.format(self.cats))
```

index(*node*)

Return the index that corresponds to the category name

show()

forest_str()

normalize()

Applies a normalization scheme to the categories.

Note: this may break other tasks that depend on exact category names.

Returns
CategoryTree

Example

```
>>> from kwcoco.category_tree import *  # NOQA
>>> import kwcoco
>>> orig = kwcoco.CategoryTree.demo('animals_v1')
>>> self = kwcoco.CategoryTree(nx.relabel_nodes(orig.graph, str.upper))
>>> norm = self.normalize()
```

class kwcoco.**ChannelSpec**(*spec*, *parsed=None*)

Bases: *BaseChannelSpec*

Parse and extract information about network input channel specs for early or late fusion networks.

Behaves like a dictionary of FusedChannelSpec objects

Todo:

- **[ ] Rename to something that indicates this is a collection of**
  FusedChannelSpec? MultiChannelSpec?

Note: This class name and API is in flux and subject to change.

Note: The pipe ('|') character represents an early-fused input stream, and order matters (it is non-communative).

The comma (',') character separates different inputs streams/branches for a multi-stream/branch network which will be lated fused. Order does not matter

---

Example

```
>>> from kwcoco.channel_spec import *  # NOQA
>>> # Integer spec
>>> ChannelSpec.coerce(3)
<ChannelSpec(u0|u1|u2) ...>
```

```
>>> # single mode spec
>>> ChannelSpec.coerce('rgb')
<ChannelSpec(rgb) ...>
```

```
>>> # early fused input spec
>>> ChannelSpec.coerce('rgb|disprity')
<ChannelSpec(rgb|disprity) ...>
```

```
>>> # late fused input spec
>>> ChannelSpec.coerce('rgb,disprity')
<ChannelSpec(rgb,disprity) ...>
```

```
>>> # early and late fused input spec
>>> ChannelSpec.coerce('rgb|ir,disprity')
<ChannelSpec(rgb|ir,disprity) ...>
```

Example

```
>>> self = ChannelSpec('gray')
>>> print('self.info = {}'.format(ub.repr2(self.info, nl=1)))
>>> self = ChannelSpec('rgb')
>>> print('self.info = {}'.format(ub.repr2(self.info, nl=1)))
>>> self = ChannelSpec('rgb|disparity')
>>> print('self.info = {}'.format(ub.repr2(self.info, nl=1)))
>>> self = ChannelSpec('rgb|disparity,disparity')
>>> print('self.info = {}'.format(ub.repr2(self.info, nl=1)))
>>> self = ChannelSpec('rgb,disparity,flowx|flowy')
>>> print('self.info = {}'.format(ub.repr2(self.info, nl=1)))
```

Example

```
>>> specs = [
>>>     'rgb',              # and rgb input
>>>     'rgb|disprity',     # rgb early fused with disparity
>>>     'rgb,disprity',     # rgb early late with disparity
>>>     'rgb|ir,disprity',  # rgb early fused with ir and late fused with disparity
>>>     3,                  # 3 unknown channels
>>> ]
>>> for spec in specs:
>>>     print('=======================')
>>>     print('spec = {!r}'.format(spec))
>>>     #
>>>     self = ChannelSpec.coerce(spec)
>>>     print('self = {!r}'.format(self))
>>>     sizes = self.sizes()
>>>     print('sizes = {!r}'.format(sizes))
>>>     print('self.info = {}'.format(ub.repr2(self.info, nl=1)))
>>>     #
>>>     item = self._demo_item((1, 1), rng=0)
>>>     inputs = self.encode(item)
>>>     components = self.decode(inputs)
>>>     input_shapes = ub.map_vals(lambda x: x.shape, inputs)
>>>     component_shapes = ub.map_vals(lambda x: x.shape, components)
```

(continues on next page)

(continued from previous page)

```
>>>     print('item = {}'.format(ub.repr2(item, precision=1)))
>>>     print('inputs = {}'.format(ub.repr2(inputs, precision=1)))
>>>     print('input_shapes = {}'.format(ub.repr2(input_shapes)))
>>>     print('components = {}'.format(ub.repr2(components, precision=1)))
>>>     print('component_shapes = {}'.format(ub.repr2(component_shapes, nl=1)))
```

property spec

property info

classmethod coerce(*data*)

Attempt to interpret the data as a channel specification

Returns
ChannelSpec

Example

```
>>> from kwcoco.channel_spec import *  # NOQA
>>> data = FusedChannelSpec.coerce(3)
>>> assert ChannelSpec.coerce(data).spec == 'u0|u1|u2'
>>> data = ChannelSpec.coerce(3)
>>> assert data.spec == 'u0|u1|u2'
>>> assert ChannelSpec.coerce(data).spec == 'u0|u1|u2'
>>> data = ChannelSpec.coerce('u:3')
>>> assert data.normalize().spec == 'u.0|u.1|u.2'
```

parse()

Build internal representation

Example

```
>>> from kwcoco.channel_spec import *  # NOQA
>>> self = ChannelSpec('b1|b2|b3|rgb,B:3')
>>> print(self.parse())
>>> print(self.normalize().parse())
>>> ChannelSpec('').parse()
```

Example

```
>>> base = ChannelSpec('rgb|disparity,flowx|r|flowy')
>>> other = ChannelSpec('rgb')
>>> self = base.intersection(other)
>>> assert self.numel() == 4
```

concise()

Example

```
>>> self = ChannelSpec('b1|b2,b3|rgb|B.0,B.1|B.2')
>>> print(self.concise().spec)
b1|b2,b3|r|g|b|B.0,B.1:3
```

normalize()

Replace aliases with explicit single-band-per-code specs

Returns
normalized spec

Return type
ChannelSpec

Example

```
>>> self = ChannelSpec('b1|b2,b3|rgb,B:3')
>>> normed = self.normalize()
>>> print('self   = {}'.format(self))
>>> print('normed = {}'.format(normed))
self   = <ChannelSpec(b1|b2,b3|rgb,B:3)>
normed = <ChannelSpec(b1|b2,b3|r|g|b,B.0|B.1|B.2)>
```

keys()

values()

items()

fuse()

Fuse all parts into an early fused channel spec

Returns
FusedChannelSpec

Example

```
>>> from kwcoco.channel_spec import *  # NOQA
>>> self = ChannelSpec.coerce('b1|b2,b3|rgb,B:3')
>>> fused = self.fuse()
>>> print('self  = {}'.format(self))
>>> print('fused = {}'.format(fused))
self  = <ChannelSpec(b1|b2,b3|rgb,B:3)>
fused = <FusedChannelSpec(b1|b2|b3|rgb|B:3)>
```

streams()

Breaks this spec up into one spec for each early-fused input stream

Example

self = ChannelSpec.coerce('r|g,B1|B2,fx|fy') list(map(len, self.streams()))

code_list()

as_path()

Returns a string suitable for use in a path.

Note, this may no longer be a valid channel spec

difference(*other*)

Set difference. Remove all instances of other channels from this set of channels.

Example

```
>>> from kwcoco.channel_spec import *
>>> self = ChannelSpec('rgb|disparity,flowx|r|flowy')
>>> other = ChannelSpec('rgb')
>>> print(self.difference(other))
>>> other = ChannelSpec('flowx')
>>> print(self.difference(other))
<ChannelSpec(disparity,flowx|flowy)>
<ChannelSpec(r|g|b|disparity,r|flowy)>
```

Example

```
>>> from kwcoco.channel_spec import *
>>> self = ChannelSpec('a|b,c|d')
>>> new = self - {'a', 'b'}
>>> len(new.sizes()) == 1
>>> empty = new - 'c|d'
>>> assert empty.numel() == 0
```

intersection(*other*)

Set difference. Remove all instances of other channels from this set of channels.

Example

```
>>> from kwcoco.channel_spec import *
>>> self = ChannelSpec('rgb|disparity,flowx|r|flowy')
>>> other = ChannelSpec('rgb')
>>> new = self.intersection(other)
>>> print(new)
>>> print(new.numel())
>>> other = ChannelSpec('flowx')
>>> new = self.intersection(other)
>>> print(new)
>>> print(new.numel())
<ChannelSpec(r|g|b,r)>
4
```

(continues on next page)

(continued from previous page)

```
<ChannelSpec(flowx)>
1
```

union(*other*)

Union simply tags on a second channel spec onto this one. Duplicates are maintained.

Example

```
>>> from kwcoco.channel_spec import *
>>> self = ChannelSpec('rgb|disparity,flowx|r|flowy')
>>> other = ChannelSpec('rgb')
>>> new = self.union(other)
>>> print(new)
>>> print(new.numel())
>>> other = ChannelSpec('flowx')
>>> new = self.union(other)
>>> print(new)
>>> print(new.numel())
<ChannelSpec(r|g|b|disparity,flowx|r|flowy,r|g|b)>
10
<ChannelSpec(r|g|b|disparity,flowx|r|flowy,flowx)>
8
```

issubset(*other*)

issuperset(*other*)

numel()

Total number of channels in this spec

sizes()

Number of dimensions for each fused stream channel

IE: The EARLY-FUSED channel sizes

Example

```
>>> self = ChannelSpec('rgb|disparity,flowx|flowy,B:10')
>>> self.normalize().concise()
>>> self.sizes()
```

unique(*normalize=False*)

Returns the unique channels that will need to be given or loaded

encode(*item*, *axis=0*, *mode=1*)

Given a dictionary containing preloaded components of the network inputs, build a concatenated (fused) network representations of each input stream.

Parameters

- **item** (*Dict[str, Tensor]*) – a batch item containing unfused parts. each key should be a single-stream (optionally early fused) channel key.

- **axis** (*int, default=0*) – concatenation dimension

Returns
mapping between input stream and its early fused tensor input.

Return type
Dict[str, Tensor]

Example

```
>>> from kwcoco.channel_spec import *  # NOQA
>>> import numpy as np
>>> dims = (4, 4)
>>> item = {
>>>     'rgb': np.random.rand(3, *dims),
>>>     'disparity': np.random.rand(1, *dims),
>>>     'flowx': np.random.rand(1, *dims),
>>>     'flowy': np.random.rand(1, *dims),
>>> }
>>> # Complex Case
>>> self = ChannelSpec('rgb,disparity,rgb|disparity|flowx|flowy,flowx|flowy')
>>> fused = self.encode(item)
>>> input_shapes = ub.map_vals(lambda x: x.shape, fused)
>>> print('input_shapes = {}'.format(ub.repr2(input_shapes, nl=1)))
>>> # Simpler case
>>> self = ChannelSpec('rgb|disparity')
>>> fused = self.encode(item)
>>> input_shapes = ub.map_vals(lambda x: x.shape, fused)
>>> print('input_shapes = {}'.format(ub.repr2(input_shapes, nl=1)))
```

Example

```
>>> # Case where we have to break up early fused data
>>> import numpy as np
>>> dims = (40, 40)
>>> item = {
>>>     'rgb|disparity': np.random.rand(4, *dims),
>>>     'flowx': np.random.rand(1, *dims),
>>>     'flowy': np.random.rand(1, *dims),
>>> }
>>> # Complex Case
>>> self = ChannelSpec('rgb,disparity,rgb|disparity,rgb|disparity|flowx|flowy,
↪flowx|flowy,flowx,disparity')
>>> inputs = self.encode(item)
>>> input_shapes = ub.map_vals(lambda x: x.shape, inputs)
>>> print('input_shapes = {}'.format(ub.repr2(input_shapes, nl=1)))
```

```
>>> # xdoctest: +REQUIRES(--bench)
>>> #self = ChannelSpec('rgb|disparity,flowx|flowy')
>>> import timerit
>>> ti = timerit.Timerit(100, bestof=10, verbose=2)
```

(continues on next page)

(continued from previous page)

```
>>> for timer in ti.reset('mode=simple'):
>>>     with timer:
>>>         inputs = self.encode(item, mode=0)
>>> for timer in ti.reset('mode=minimize-concat'):
>>>     with timer:
>>>         inputs = self.encode(item, mode=1)
```

decode(*inputs*, *axis=1*)

break an early fused item into its components

Parameters

- **inputs** (*Dict[str, Tensor]*) – dictionary of components
- **axis** (*int, default=1*) – channel dimension

Example

```
>>> from kwcoco.channel_spec import *  # NOQA
>>> import numpy as np
>>> dims = (4, 4)
>>> item_components = {
>>>     'rgb': np.random.rand(3, *dims),
>>>     'ir': np.random.rand(1, *dims),
>>> }
>>> self = ChannelSpec('rgb|ir')
>>> item_encoded = self.encode(item_components)
>>> batch = {k: np.concatenate([v[None, :], v[None, :]], axis=0)
...          for k, v in item_encoded.items()}
>>> components = self.decode(batch)
```

Example

```
>>> # xdoctest: +REQUIRES(module:netharn, module:torch)
>>> import torch
>>> import numpy as np
>>> dims = (4, 4)
>>> components = {
>>>     'rgb': np.random.rand(3, *dims),
>>>     'ir': np.random.rand(1, *dims),
>>> }
>>> components = ub.map_vals(torch.from_numpy, components)
>>> self = ChannelSpec('rgb|ir')
>>> encoded = self.encode(components)
>>> from netharn.data import data_containers
>>> item = {k: data_containers.ItemContainer(v, stack=True)
>>>         for k, v in encoded.items()}
>>> batch = data_containers.container_collate([item, item])
>>> components = self.decode(batch)
```

component_indices(*axis=2*)

Look up component indices within fused streams

Example

```
>>> dims = (4, 4)
>>> inputs = ['flowx', 'flowy', 'disparity']
>>> self = ChannelSpec('disparity,flowx|flowy')
>>> component_indices = self.component_indices()
>>> print('component_indices = {}'.format(ub.repr2(component_indices, nl=1)))
component_indices = {
    'disparity': ('disparity', (slice(None, None, None), slice(None, None,␣
↪None), slice(0, 1, None))),
    'flowx': ('flowx|flowy', (slice(None, None, None), slice(None, None, None),␣
↪slice(0, 1, None))),
    'flowy': ('flowx|flowy', (slice(None, None, None), slice(None, None, None),␣
↪slice(1, 2, None))),
}
```

class kwcoco.**CocoDataset**(*data=None*, *tag=None*, *bundle_dpath=None*, *img_root=None*, *fname=None*, *autobuild=True*)

Bases: *AbstractCocoDataset*, *MixinCocoAddRemove*, *MixinCocoStats*, *MixinCocoObjects*, *MixinCocoDraw*, *MixinCocoAccessors*, *MixinCocoExtras*, *MixinCocoIndex*, *MixinCocoDepricate*, NiceRepr

The main coco dataset class with a json dataset backend.

Variables

- **dataset** (*Dict*) – raw json data structure. This is the base dictionary that contains {'annotations': List, 'images': List, 'categories': List}
- **index** (CocoIndex) – an efficient lookup index into the coco data structure. The index defines its own attributes like anns, cats, imgs, gid_to_aids, file_name_to_img, etc. See CocoIndex for more details on which attributes are available.
- **fpath** (*PathLike | None*) – if known, this stores the filepath the dataset was loaded from
- **tag** (*str*) – A tag indicating the name of the dataset.
- **bundle_dpath** (*PathLike | None*) – If known, this is the root path that all image file names are relative to. This can also be manually overwritten by the user.
- **hashid** (*str | None*) – If computed, this will be a hash uniquely identifing the dataset. To ensure this is computed see kwcoco.coco_dataset.MixinCocoExtras._build_hashid().

References

http://cocodataset.org/#format http://cocodataset.org/#download

CommandLine

```
python -m kwcoco.coco_dataset CocoDataset --show
```

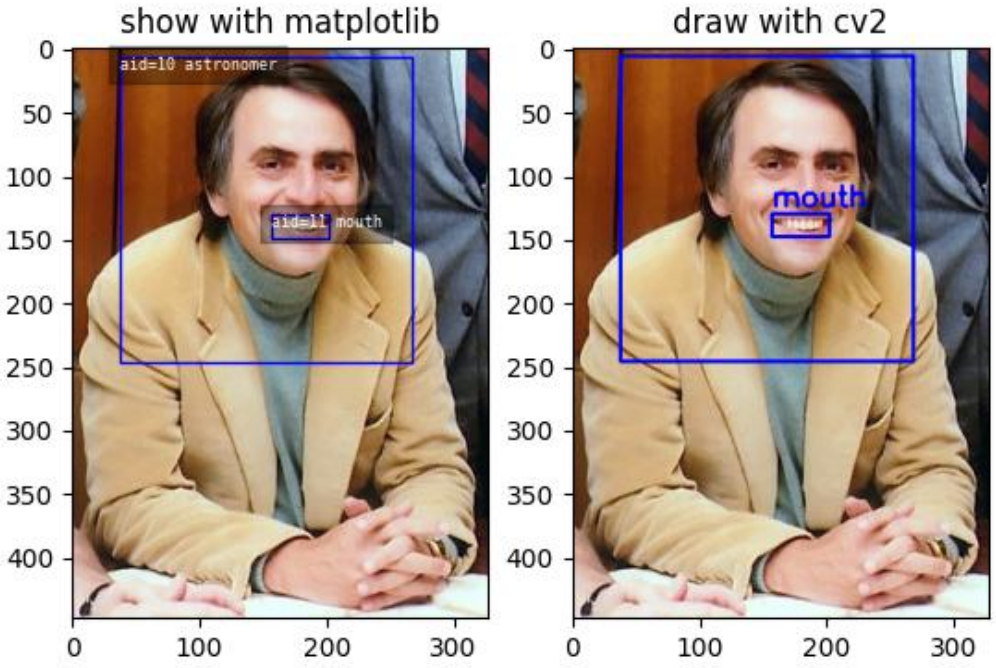
Example

```
>>> from kwcoco.coco_dataset import demo_coco_data
>>> import kwcoco
>>> import ubelt as ub
>>> # Returns a coco json structure
>>> dataset = demo_coco_data()
>>> # Pass the coco json structure to the API
>>> self = kwcoco.CocoDataset(dataset, tag='demo')
>>> # Now you can access the data using the index and helper methods
>>> #
>>> # Start by looking up an image by it's COCO id.
>>> image_id = 1
>>> img = self.index.imgs[image_id]
>>> print(ub.repr2(img, nl=1, sort=1))
{
    'file_name': 'astro.png',
    'id': 1,
    'url': 'https://i.imgur.com/KXhKM72.png',
}
>>> #
>>> # Use the (gid_to_aids) index to lookup annotations in the iamge
>>> annotation_id = sorted(self.index.gid_to_aids[image_id])[0]
>>> ann = self.index.anns[annotation_id]
>>> print(ub.repr2(ub.dict_diff(ann, {'segmentation'}), nl=1))
{
    'bbox': [10, 10, 360, 490],
    'category_id': 1,
    'id': 1,
    'image_id': 1,
    'keypoints': [247, 101, 2, 202, 100, 2],
}
>>> #
>>> # Use annotation category id to look up that information
>>> category_id = ann['category_id']
>>> cat = self.index.cats[category_id]
>>> print('cat = {}'.format(ub.repr2(cat, nl=1, sort=1)))
cat = {
    'id': 1,
    'name': 'astronaut',
    'supercategory': 'human',
}
>>> #
>>> # Now play with some helper functions, like extended statistics
>>> extended_stats = self.extended_stats()
>>> # xdoctest: +IGNORE_WANT
>>> print('extended_stats = {}'.format(ub.repr2(extended_stats, nl=1, precision=2,␣
```

(continues on next page)

(continued from previous page)

```
↪sort=1)))
extended_stats = {
    'annots_per_img': {'mean': 3.67, 'std': 3.86, 'min': 0.00, 'max': 9.00, 'nMin':␣
↪1, 'nMax': 1, 'shape': (3,)},
    'imgs_per_cat': {'mean': 0.88, 'std': 0.60, 'min': 0.00, 'max': 2.00, 'nMin': 2,
↪ 'nMax': 1, 'shape': (8,)},
    'cats_per_img': {'mean': 2.33, 'std': 2.05, 'min': 0.00, 'max': 5.00, 'nMin': 1,
↪ 'nMax': 1, 'shape': (3,)},
    'annots_per_cat': {'mean': 1.38, 'std': 1.49, 'min': 0.00, 'max': 5.00, 'nMin':␣
↪2, 'nMax': 1, 'shape': (8,)},
    'imgs_per_video': {'empty_list': True},
}
>>> # You can "draw" a raster of the annotated image with cv2
>>> canvas = self.draw_image(2)
>>> # Or if you have matplotlib you can "show" the image with mpl objects
>>> # xdoctest: +REQUIRES(--show)
>>> from matplotlib import pyplot as plt
>>> fig = plt.figure()
>>> ax1 = fig.add_subplot(1, 2, 1)
>>> self.show_image(gid=2)
>>> ax2 = fig.add_subplot(1, 2, 2)
>>> ax2.imshow(canvas)
>>> ax1.set_title('show with matplotlib')
>>> ax2.set_title('draw with cv2')
>>> plt.show()
```



property fpath

In the future we will deprecate img_root for bundle_dpath

classmethod from_data(*data*, *bundle_dpath=None*, *img_root=None*)

Constructor from a json dictionary

classmethod from_image_paths(*gpaths*, *bundle_dpath=None*, *img_root=None*)

Constructor from a list of images paths.

This is a convinience method.

Parameters

gpaths (*List[str]*) – list of image paths

Example

```
>>> import kwcoco
>>> coco_dset = kwcoco.CocoDataset.from_image_paths(['a.png', 'b.png'])
>>> assert coco_dset.n_images == 2
```

classmethod from_coco_paths(*fpaths*, *max_workers=0*, *verbose=1*, *mode='thread'*, *union='try'*)

Constructor from multiple coco file paths.

Loads multiple coco datasets and unions the result

Note: if the union operation fails, the list of individually loaded files is returned instead.

Parameters

- **fpaths** (*List[str]*) – list of paths to multiple coco files to be loaded and unioned.
- **max_workers** (*int, default=0*) – number of worker threads / processes
- **verbose** (*int*) – verbosity level
- **mode** (*str*) – thread, process, or serial
- **union** (*str | bool, default='try'*) – If True, unions the result datasets after loading. If False, just returns the result list. If 'try', then try to preform the union, but return the result list if it fails.

copy()

Deep copies this object

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> new = self.copy()
>>> assert new.imgs[1] is new.dataset['images'][0]
>>> assert new.imgs[1] == self.dataset['images'][0]
>>> assert new.imgs[1] is not self.dataset['images'][0]
```

dumps(*indent=None*, *newlines=False*)

Writes the dataset out to the json format

Parameters

newlines (*bool*) – if True, each annotation, image, category gets its own line

Note:

Using newlines=True is similar to:
print(ub.repr2(dset.dataset, nl=2, trailsep=False)) However, the above may not output valid json if it contains ndarrays.

Example

```
>>> import kwcoco
>>> import json
>>> self = kwcoco.CocoDataset.demo()
>>> text = self.dumps(newlines=True)
>>> print(text)
>>> self2 = kwcoco.CocoDataset(json.loads(text), tag='demo2')
>>> assert self2.dataset == self.dataset
>>> assert self2.dataset is not self.dataset
```

```
>>> text = self.dumps(newlines=True)
>>> print(text)
>>> self2 = kwcoco.CocoDataset(json.loads(text), tag='demo2')
>>> assert self2.dataset == self.dataset
>>> assert self2.dataset is not self.dataset
```

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.coerce('vidshapes1-msi-multisensor', verbose=3)
>>> self.remove_annotations(self.annots())
>>> text = self.dumps(newlines=True, indent='  ')
>>> print(text)
```

dump(*file*, *indent=None*, *newlines=False*, *temp_file=True*)

Writes the dataset out to the json format

Parameters

- **file** (*PathLike | IO*) – Where to write the data. Can either be a path to a file or an open file pointer / stream.
- **newlines** (*bool*) – if True, each annotation, image, category gets its own line.
- **temp_file** (*bool | str, default=True*) – Argument to `safer.open()`. Ignored if `file` is not a PathLike object.

Example

```
>>> import tempfile
>>> import json
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> file = tempfile.NamedTemporaryFile('w')
>>> self.dump(file)
>>> file.seek(0)
>>> text = open(file.name, 'r').read()
>>> print(text)
>>> file.seek(0)
>>> dataset = json.load(open(file.name, 'r'))
>>> self2 = kwcoco.CocoDataset(dataset, tag='demo2')
>>> assert self2.dataset == self.dataset
>>> assert self2.dataset is not self.dataset
```

```
>>> file = tempfile.NamedTemporaryFile('w')
>>> self.dump(file, newlines=True)
>>> file.seek(0)
>>> text = open(file.name, 'r').read()
>>> print(text)
>>> file.seek(0)
>>> dataset = json.load(open(file.name, 'r'))
>>> self2 = kwcoco.CocoDataset(dataset, tag='demo2')
```

(continues on next page)

(continued from previous page)

```
>>> assert self2.dataset == self.dataset
>>> assert self2.dataset is not self.dataset
```

union(*, *disjoint_tracks=True*, ***kwargs*)

Merges multiple `CocoDataset` items into one. Names and associations are retained, but ids may be different.

Parameters

- ***others** – a series of CocoDatasets that we will merge. Note, if called as an instance method, the "self" instance will be the first item in the "others" list. But if called like a classmethod, "others" will be empty by default.
- **disjoint_tracks** (*bool, default=True*) – if True, we will assume track-ids are disjoint and if two datasets share the same track-id, we will disambiguate them. Otherwise they will be copied over as-is.
- ****kwargs** – constructor options for the new merged CocoDataset

Returns

a new merged coco dataset

Return type

kwcoco.CocoDataset

CommandLine

```
xdoctest -m kwcoco.coco_dataset CocoDataset.union
```

Example

```
>>> import kwcoco
>>> # Test union works with different keypoint categories
>>> dset1 = kwcoco.CocoDataset.demo('shapes1')
>>> dset2 = kwcoco.CocoDataset.demo('shapes2')
>>> dset1.remove_keypoint_categories(['bot_tip', 'mid_tip', 'right_eye'])
>>> dset2.remove_keypoint_categories(['top_tip', 'left_eye'])
>>> dset_12a = kwcoco.CocoDataset.union(dset1, dset2)
>>> dset_12b = dset1.union(dset2)
>>> dset_21 = dset2.union(dset1)
>>> def add_hist(h1, h2):
>>>     return {k: h1.get(k, 0) + h2.get(k, 0) for k in set(h1) | set(h2)}
>>> kpfreq1 = dset1.keypoint_annotation_frequency()
>>> kpfreq2 = dset2.keypoint_annotation_frequency()
>>> kpfreq_want = add_hist(kpfreq1, kpfreq2)
>>> kpfreq_got1 = dset_12a.keypoint_annotation_frequency()
>>> kpfreq_got2 = dset_12b.keypoint_annotation_frequency()
>>> assert kpfreq_want == kpfreq_got1
>>> assert kpfreq_want == kpfreq_got2
```

```
>>> # Test disjoint gid datasets
>>> dset1 = kwcoco.CocoDataset.demo('shapes3')
```

(continues on next page)

(continued from previous page)

```
>>> for new_gid, img in enumerate(dset1.dataset['images'], start=10):
>>>     for aid in dset1.gid_to_aids[img['id']]:
>>>         dset1.anns[aid]['image_id'] = new_gid
>>>     img['id'] = new_gid
>>> dset1.index.clear()
>>> dset1._build_index()
>>> # ------
>>> dset2 = kwcoco.CocoDataset.demo('shapes2')
>>> for new_gid, img in enumerate(dset2.dataset['images'], start=100):
>>>     for aid in dset2.gid_to_aids[img['id']]:
>>>         dset2.anns[aid]['image_id'] = new_gid
>>>     img['id'] = new_gid
>>> dset1.index.clear()
>>> dset2._build_index()
>>> others = [dset1, dset2]
>>> merged = kwcoco.CocoDataset.union(*others)
>>> print('merged = {!r}'.format(merged))
>>> print('merged.imgs = {}'.format(ub.repr2(merged.imgs, nl=1)))
>>> assert set(merged.imgs) & set([10, 11, 12, 100, 101]) == set(merged.imgs)
```

```
>>> # Test data is not preserved
>>> dset2 = kwcoco.CocoDataset.demo('shapes2')
>>> dset1 = kwcoco.CocoDataset.demo('shapes3')
>>> others = (dset1, dset2)
>>> cls = self = kwcoco.CocoDataset
>>> merged = cls.union(*others)
>>> print('merged = {!r}'.format(merged))
>>> print('merged.imgs = {}'.format(ub.repr2(merged.imgs, nl=1)))
>>> assert set(merged.imgs) & set([1, 2, 3, 4, 5]) == set(merged.imgs)
```

```
>>> # Test track-ids are mapped correctly
>>> dset1 = kwcoco.CocoDataset.demo('vidshapes1')
>>> dset2 = kwcoco.CocoDataset.demo('vidshapes2')
>>> dset3 = kwcoco.CocoDataset.demo('vidshapes3')
>>> others = (dset1, dset2, dset3)
>>> for dset in others:
>>>     [a.pop('segmentation', None) for a in dset.index.anns.values()]
>>>     [a.pop('keypoints', None) for a in dset.index.anns.values()]
>>> cls = self = kwcoco.CocoDataset
>>> merged = cls.union(*others, disjoint_tracks=1)
>>> print('dset1.anns = {}'.format(ub.repr2(dset1.anns, nl=1)))
>>> print('dset2.anns = {}'.format(ub.repr2(dset2.anns, nl=1)))
>>> print('dset3.anns = {}'.format(ub.repr2(dset3.anns, nl=1)))
>>> print('merged.anns = {}'.format(ub.repr2(merged.anns, nl=1)))
```

Example

```
>>> import kwcoco
>>> # Test empty union
>>> empty_union = kwcoco.CocoDataset.union()
>>> assert len(empty_union.index.imgs) == 0
```

Todo:

- [ ] are supercategories broken?
- [ ] reuse image ids where possible
- [ ] reuse annotation / category ids where possible
- [X] handle case where no inputs are given
- [x] disambiguate track-ids
- [x] disambiguate video-ids

---

subset(*gids*, *copy=False*, *autobuild=True*)

Return a subset of the larger coco dataset by specifying which images to port. All annotations in those images will be taken.

Parameters

- **gids** (*List[int]*) – image-ids to copy into a new dataset
- **copy** (*bool, default=False*) – if True, makes a deep copy of all nested attributes, otherwise makes a shallow copy.
- **autobuild** (*bool, default=True*) – if True will automatically build the fast lookup index.

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> gids = [1, 3]
>>> sub_dset = self.subset(gids)
>>> assert len(self.index.gid_to_aids) == 3
>>> assert len(sub_dset.gid_to_aids) == 2
```

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo('vidshapes2')
>>> gids = [1, 2]
>>> sub_dset = self.subset(gids, copy=True)
>>> assert len(sub_dset.index.videos) == 1
>>> assert len(self.index.videos) == 2
```

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> sub1 = self.subset([1])
>>> sub2 = self.subset([2])
>>> sub3 = self.subset([3])
>>> others = [sub1, sub2, sub3]
>>> rejoined = kwcoco.CocoDataset.union(*others)
>>> assert len(sub1.anns) == 9
>>> assert len(sub2.anns) == 2
>>> assert len(sub3.anns) == 0
>>> assert rejoined.basic_stats() == self.basic_stats()
```

view_sql(*force_rewrite=False*, *memory=False*)

Create a cached SQL interface to this dataset suitable for large scale multiprocessing use cases.

Parameters

- **force_rewrite** (*bool, default=False*) – if True, forces an update to any existing cache file on disk
- **memory** (*bool, default=False*) – if True, the database is constructed in memory.

Note: This view cache is experimental and currently depends on the timestamp of the file pointed to by `self.fpath`. In other words dont use this on in-memory datasets.

---

class kwcoco.**CocoImage**(*img*, *dset=None*)

Bases: NiceRepr

An object-oriented representation of a coco image.

It provides helper methods that are specific to a single image.

This operates directly on a single coco image dictionary, but it can optionally be connected to a parent dataset, which allows it to use CocoDataset methods to query about relationships and resolve pointers.

This is different than the Images class in coco_object1d, which is just a vectorized interface to multiple objects.

Example

```
>>> import kwcoco
>>> dset1 = kwcoco.CocoDataset.demo('shapes8')
>>> dset2 = kwcoco.CocoDataset.demo('vidshapes8-multispectral')
```

```
>>> self = CocoImage(dset1.imgs[1], dset1)
>>> print('self = {!r}'.format(self))
>>> print('self.channels = {}'.format(ub.repr2(self.channels, nl=1)))
```

```
>>> self = CocoImage(dset2.imgs[1], dset2)
>>> print('self.channels = {}'.format(ub.repr2(self.channels, nl=1)))
>>> self.primary_asset()
```

classmethod from_gid(*dset*, *gid*)

property bundle_dpath

property video

Helper to grab the video for this image if it exists

detach()

Removes references to the underlying coco dataset, but keeps special information such that it wont be needed.

stats()

keys()

Proxy getter attribute for underlying *self.img* dictionary

get(*key*, *default=NoParam*)

Proxy getter attribute for underlying *self.img* dictionary

Example

```
>>> import pytest
>>> # without extra populated
>>> import kwcoco
>>> self = kwcoco.CocoImage({'foo': 1})
>>> assert self.get('foo') == 1
>>> assert self.get('foo', None) == 1
>>> # with extra populated
>>> self = kwcoco.CocoImage({'extra': {'foo': 1}})
>>> assert self.get('foo') == 1
>>> assert self.get('foo', None) == 1
>>> # without extra empty
>>> self = kwcoco.CocoImage({})
>>> with pytest.raises(KeyError):
>>>     self.get('foo')
>>> assert self.get('foo', None) is None
>>> # with extra empty
>>> self = kwcoco.CocoImage({'extra': {'bar': 1}})
>>> with pytest.raises(KeyError):
>>>     self.get('foo')
>>> assert self.get('foo', None) is None
```

property channels

property num_channels

property dsize

primary_image_filepath(*requires=None*)

primary_asset(*requires=None*)

Compute a “main” image asset.

Notes

Uses a heuristic.

- First, try to find the auxiliary image that has with the smallest

distortion to the base image (if known via warp_aux_to_img)

- Second, break ties by using the largest image if w / h is known
- Last, if previous information not available use the first auxiliary image.

Parameters
requires (*List[str]*) – list of attribute that must be non-None to consider an object as the primary one.

Returns
the asset dict or None if it is not found

Return type
None | dict

Todo:

- [ ] Add in primary heuristics

Example

```
>>> import kwarray
>>> from kwcoco.coco_image import *  # NOQA
>>> rng = kwarray.ensure_rng(0)
>>> def random_auxiliary(name, w=None, h=None):
>>>     return {'file_name': name, 'width': w, 'height': h}
>>> self = CocoImage({
>>>     'auxiliary': [
>>>         random_auxiliary('1'),
>>>         random_auxiliary('2'),
>>>         random_auxiliary('3'),
>>>     ]
>>> })
>>> assert self.primary_asset()['file_name'] == '1'
>>> self = CocoImage({
>>>     'auxiliary': [
>>>         random_auxiliary('1'),
>>>         random_auxiliary('2', 3, 3),
>>>         random_auxiliary('3'),
>>>     ]
>>> })
>>> assert self.primary_asset()['file_name'] == '2'
```

iter_image_filepaths(*with_bundle=True*)
Could rename to iter_asset_filepaths

Parameters

with_bundle (*bool*) – If True, prepends the bundle dpath to fully specify the path. Otherwise, just returns the registered string in the file_name attribute of each asset. Defaults to True.

`iter_asset_objs`()

Iterate through base + auxiliary dicts that have file paths

Yields

dict – an image or auxiliary dictionary

`find_asset_obj`(*channels*)

Find the asset dictionary with the specified channels

Example

```
>>> import kwcoco
>>> coco_img = kwcoco.CocoImage({'width': 128, 'height': 128})
>>> coco_img.add_auxiliary_item(
>>>     'rgb.png', channels='red|green|blue', width=32, height=32)
>>> assert coco_img.find_asset_obj('red') is not None
>>> assert coco_img.find_asset_obj('green') is not None
>>> assert coco_img.find_asset_obj('blue') is not None
>>> assert coco_img.find_asset_obj('red|blue') is not None
>>> assert coco_img.find_asset_obj('red|green|blue') is not None
>>> assert coco_img.find_asset_obj('red|green|blue') is not None
>>> assert coco_img.find_asset_obj('black') is None
>>> assert coco_img.find_asset_obj('r') is None
```

Example

```
>>> # Test with concise channel code
>>> import kwcoco
>>> coco_img = kwcoco.CocoImage({'width': 128, 'height': 128})
>>> coco_img.add_auxiliary_item(
>>>     'msi.png', channels='foo.0:128', width=32, height=32)
>>> assert coco_img.find_asset_obj('foo') is None
>>> assert coco_img.find_asset_obj('foo.3') is not None
>>> assert coco_img.find_asset_obj('foo.3:5') is not None
>>> assert coco_img.find_asset_obj('foo.3000') is None
```

`add_auxiliary_item`(*file_name=None*, *channels=None*, *imdata=None*, *warp_aux_to_img=None*, *width=None*, *height=None*, *imwrite=False*)

Adds an auxiliary / asset item to the image dictionary.

This operation can be done purely in-memory (the default), or the image data can be written to a file on disk (via the imwrite=True flag).

Parameters

- **file_name** (*str | None*) – The name of the file relative to the bundle directory. If unspecified, imdata must be given.
- **channels** (*str | kwcoco.FusedChannelSpec*) – The channel code indicating what each of the bands represents. These channels should be disjoint wrt to the existing data in this image (this is not checked).

- **imdata** (*ndarray | None*) – The underlying image data this auxiliary item represents. If unspecified, it is assumed file_name points to a path on disk that will eventually exist. If imdata, file_name, and the special imwrite=True flag are specified, this function will write the data to disk.
- **warp_aux_to_img** (*kwimage.Affine*) – The transformation from this auxiliary space to image space. If unspecified, assumes this item is related to image space by only a scale factor.
- **width** (*int*) – Width of the data in auxiliary space (inferred if unspecified)
- **height** (*int*) – Height of the data in auxiliary space (inferred if unspecified)
- **imwrite** (*bool*) – If specified, both imdata and file_name must be specified, and this will write the data to disk. Note: it it recommended that you simply call imwrite yourself before or after calling this function. This lets you better control imwrite parameters.

Todo:

- [ ] Allow imwrite to specify an executor that is used to

return a Future so the imwrite call does not block.

Example

```
>>> from kwcoco.coco_image import *  # NOQA
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('vidshapes8-multispectral')
>>> coco_img = dset.coco_image(1)
>>> imdata = np.random.rand(32, 32, 5)
>>> channels = kwcoco.FusedChannelSpec.coerce('Aux:5')
>>> coco_img.add_auxiliary_item(imdata=imdata, channels=channels)
```

Example

```
>>> import kwcoco
>>> dset = kwcoco.CocoDataset()
>>> gid = dset.add_image(name='my_image_name', width=200, height=200)
>>> coco_img = dset.coco_image(gid)
>>> coco_img.add_auxiliary_item('path/img1_B0.tif', channels='B0', width=200,␣
↪height=200)
>>> coco_img.add_auxiliary_item('path/img1_B1.tif', channels='B1', width=200,␣
↪height=200)
>>> coco_img.add_auxiliary_item('path/img1_B2.tif', channels='B2', width=200,␣
↪height=200)
>>> coco_img.add_auxiliary_item('path/img1_TCI.tif', channels='r|g|b',␣
↪width=200, height=200)
```

add_asset(*file_name=None*, *channels=None*, *imdata=None*, *warp_aux_to_img=None*, *width=None*, *height=None*, *imwrite=False*)

Adds an auxiliary / asset item to the image dictionary.

This operation can be done purely in-memory (the default), or the image data can be written to a file on disk (via the imwrite=True flag).

Parameters

- **file_name** (*str | None*) – The name of the file relative to the bundle directory. If unspecified, imdata must be given.
- **channels** (*str | kwcoco.FusedChannelSpec*) – The channel code indicating what each of the bands represents. These channels should be disjoint wrt to the existing data in this image (this is not checked).
- **imdata** (*ndarray | None*) – The underlying image data this auxiliary item represents. If unspecified, it is assumed file_name points to a path on disk that will eventually exist. If imdata, file_name, and the special imwrite=True flag are specified, this function will write the data to disk.
- **warp_aux_to_img** (*kwimage.Affine*) – The transformation from this auxiliary space to image space. If unspecified, assumes this item is related to image space by only a scale factor.
- **width** (*int*) – Width of the data in auxiliary space (inferred if unspecified)
- **height** (*int*) – Height of the data in auxiliary space (inferred if unspecified)
- **imwrite** (*bool*) – If specified, both imdata and file_name must be specified, and this will write the data to disk. Note: it it recommended that you simply call imwrite yourself before or after calling this function. This lets you better control imwrite parameters.

Todo:

- [ ] Allow imwrite to specify an executor that is used to

return a Future so the imwrite call does not block.

Example

```
>>> from kwcoco.coco_image import *  # NOQA
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('vidshapes8-multispectral')
>>> coco_img = dset.coco_image(1)
>>> imdata = np.random.rand(32, 32, 5)
>>> channels = kwcoco.FusedChannelSpec.coerce('Aux:5')
>>> coco_img.add_auxiliary_item(imdata=imdata, channels=channels)
```

Example

```
>>> import kwcoco
>>> dset = kwcoco.CocoDataset()
>>> gid = dset.add_image(name='my_image_name', width=200, height=200)
>>> coco_img = dset.coco_image(gid)
>>> coco_img.add_auxiliary_item('path/img1_B0.tif', channels='B0', width=200,␣
↪height=200)
>>> coco_img.add_auxiliary_item('path/img1_B1.tif', channels='B1', width=200,␣
↪height=200)
>>> coco_img.add_auxiliary_item('path/img1_B2.tif', channels='B2', width=200,␣
↪height=200)
```

(continues on next page)

(continued from previous page)

```
>>> coco_img.add_auxiliary_item('path/img1_TCI.tif', channels='r|g|b',␣
↪width=200, height=200)
```

delay(*channels=None*, *space='image'*, *bundle_dpath=None*, *interpolation='linear'*, *antialias=True*, *nodata_method=None*, *mode=1*)

Perform a delayed load on the data in this image.

The delayed load can load a subset of channels, and perform lazy warping operations. If the underlying data is in a tiled format this can reduce the amount of disk IO needed to read the data if only a small crop or lower resolution view of the data is needed.

Note: This method is experimental and relies on the delayed load proof-of-concept.

Parameters

- **gid** (*int*) – image id to load
- **channels** (*kwcoco.FusedChannelSpec*) – specific channels to load. if unspecified, all channels are loaded.
- **space** (*str*) – can either be “image” for loading in image space, or “video” for loading in video space.

Todo:

- **[X] Currently can only take all or none of the channels from each**
  base-image / auxiliary dict. For instance if the main image is r|g|b you can’t just select g|b at the moment.
- **[X] The order of the channels in the delayed load should**
  match the requested channel order.
- [X] TODO: add nans to bands that don’t exist or throw an error
- **[ ] This function could stand to have a better name. Maybe imread**
  with a delayed=True flag? Or maybe just delayed_load?

Example

```
>>> from kwcoco.coco_image import *  # NOQA
>>> import kwcoco
>>> gid = 1
>>> #
>>> dset = kwcoco.CocoDataset.demo('vidshapes8-multispectral')
>>> self = CocoImage(dset.imgs[gid], dset)
>>> delayed = self.delay()
>>> print('delayed = {!r}'.format(delayed))
>>> print('delayed.finalize() = {!r}'.format(delayed.finalize()))
>>> print('delayed.finalize() = {!r}'.format(delayed.finalize()))
>>> #
>>> dset = kwcoco.CocoDataset.demo('shapes8')
```

(continues on next page)

(continued from previous page)

```
>>> delayed = dset.coco_image(gid).delay()
>>> print('delayed = {!r}'.format(delayed))
>>> print('delayed.finalize() = {!r}'.format(delayed.finalize()))
>>> print('delayed.finalize() = {!r}'.format(delayed.finalize()))
```

```
>>> crop = delayed.crop((slice(0, 3), slice(0, 3)))
>>> crop.finalize()
```

```
>>> # TODO: should only select the "red" channel
>>> dset = kwcoco.CocoDataset.demo('shapes8')
>>> delayed = CocoImage(dset.imgs[gid], dset).delay(channels='r')
```

```
>>> import kwcoco
>>> gid = 1
>>> #
>>> dset = kwcoco.CocoDataset.demo('vidshapes8-multispectral')
>>> delayed = dset.coco_image(gid).delay(channels='B1|B2', space='image')
>>> print('delayed = {!r}'.format(delayed))
>>> print('delayed.finalize() = {!r}'.format(delayed.finalize()))
>>> delayed = dset.coco_image(gid).delay(channels='B1|B2|B11', space='image')
>>> print('delayed = {!r}'.format(delayed))
>>> print('delayed.finalize() = {!r}'.format(delayed.finalize()))
>>> delayed = dset.coco_image(gid).delay(channels='B8|B1', space='video')
>>> print('delayed = {!r}'.format(delayed))
>>> print('delayed.finalize() = {!r}'.format(delayed.finalize()))
```

```
>>> delayed = dset.coco_image(gid).delay(channels='B8|foo|bar|B1', space='video
↪')
>>> print('delayed = {!r}'.format(delayed))
>>> print('delayed.finalize() = {!r}'.format(delayed.finalize()))
```

Example

```
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo()
>>> coco_img = dset.coco_image(1)
>>> # Test case where nothing is registered in the dataset
>>> delayed = coco_img.delay()
>>> final = delayed.finalize()
>>> assert final.shape == (512, 512, 3)
```

```
>>> delayed = coco_img.delay(mode=1)
>>> final = delayed.finalize()
>>> print('final.shape = {}'.format(ub.repr2(final.shape, nl=1)))
>>> assert final.shape == (512, 512, 3)
```

Example

```
>>> # Test that delay works when imdata is stored in the image
>>> # dictionary itself.
>>> from kwcoco.coco_image import *  # NOQA
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('vidshapes8-multispectral')
>>> coco_img = dset.coco_image(1)
>>> imdata = np.random.rand(6, 6, 5)
>>> imdata[:] = np.arange(5)[None, None, :]
>>> channels = kwcoco.FusedChannelSpec.coerce('Aux:5')
>>> coco_img.add_auxiliary_item(imdata=imdata, channels=channels)
>>> delayed = coco_img.delay(channels='B1|Aux:2:4', mode=1)
>>> final = delayed.finalize()
```

Example

```
>>> # Test delay when loading in asset space
>>> from kwcoco.coco_image import *  # NOQA
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('vidshapes8-msi-multisensor')
>>> coco_img = dset.coco_image(1)
>>> stream1 = coco_img.channels.streams()[0]
>>> stream2 = coco_img.channels.streams()[1]
>>> aux_delayed = coco_img.delay(stream1, space='asset')
>>> img_delayed = coco_img.delay(stream1, space='image')
>>> vid_delayed = coco_img.delay(stream1, space='video')
>>> #
>>> aux_imdata = aux_delayed.as_xarray().finalize()
>>> img_imdata = img_delayed.as_xarray().finalize()
>>> assert aux_imdata.shape != img_imdata.shape
>>> # Cannot load multiple asset items at the same time in
>>> # asset space
>>> import pytest
>>> fused_channels = stream1 | stream2
>>> with pytest.raises(kwcoco.exceptions.CoordinateCompatibilityError):
>>>     aux_delayed2 = coco_img.delay(fused_channels, space='asset')
```

valid_region(*space='image'*)

If this image has a valid polygon, return it in image, or video space

property warp_vid_from_img

property warp_img_from_vid

class kwcoco.**FusedChannelSpec**(*parsed*, *_is_normalized=False*)

Bases: *BaseChannelSpec*

A specific type of channel spec with only one early fused stream.

The channels in this stream are non-communative

Behaves like a list of atomic-channel codes (which may represent more than 1 channel), normalized codes always represent exactly 1 channel.

Note: This class name and API is in flux and subject to change.

Todo: A special code indicating a name and some number of bands that that names contains, this would primarilly be used for large numbers of channels produced by a network. Like:

resnet_d35d060_L5:512

or

resnet_d35d060_L5[:512]

might refer to a very specific (hashed) set of resnet parameters with 512 bands

maybe we can do something slicly like:

resnet_d35d060_L5[A:B] resnet_d35d060_L5:A:B

Do we want to "just store the code" and allow for parsing later?

Or do we want to ensure the serialization is parsed before we construct the data structure?

Example

```
>>> from kwcoco.channel_spec import *  # NOQA
>>> import pickle
>>> self = FusedChannelSpec.coerce(3)
>>> recon = pickle.loads(pickle.dumps(self))
>>> self = ChannelSpec.coerce('a|b,c|d')
>>> recon = pickle.loads(pickle.dumps(self))
```

classmethod concat(*items*)

property spec

unique()

classmethod parse(*spec*)

classmethod coerce(*data*)

Example

```
>>> from kwcoco.channel_spec import *  # NOQA
>>> FusedChannelSpec.coerce(['a', 'b', 'c'])
>>> FusedChannelSpec.coerce('a|b|c')
>>> FusedChannelSpec.coerce(3)
>>> FusedChannelSpec.coerce(FusedChannelSpec(['a']))
>>> assert FusedChannelSpec.coerce('').numel() == 0
```

concise()

Shorted the channel spec by de-normaliz slice syntax

Returns

concise spec

Return type
FusedChannelSpec

Example

```
>>> from kwcoco.channel_spec import *  # NOQA
>>> self = FusedChannelSpec.coerce(
>>>      'b|a|a.0|a.1|a.2|a.5|c|a.8|a.9|b.0:3|c.0')
>>> short = self.concise()
>>> long = short.normalize()
>>> numels = [c.numel() for c in [self, short, long]]
>>> print('self.spec  = {!r}'.format(self.spec))
>>> print('short.spec = {!r}'.format(short.spec))
>>> print('long.spec  = {!r}'.format(long.spec))
>>> print('numels = {!r}'.format(numels))
self.spec  = 'b|a|a.0|a.1|a.2|a.5|c|a.8|a.9|b.0:3|c.0'
short.spec = 'b|a|a:3|a.5|c|a.8:10|b:3|c.0'
long.spec  = 'b|a|a.0|a.1|a.2|a.5|c|a.8|a.9|b.0|b.1|b.2|c.0'
numels = [13, 13, 13]
>>> assert long.concise().spec == short.spec
```

normalize()

Replace aliases with explicit single-band-per-code specs

Returns
normalize spec

Return type
FusedChannelSpec

Example

```
>>> from kwcoco.channel_spec import *  # NOQA
>>> self = FusedChannelSpec.coerce('b1|b2|b3|rgb')
>>> normed = self.normalize()
>>> print('self = {}'.format(self))
>>> print('normed = {}'.format(normed))
self = <FusedChannelSpec(b1|b2|b3|rgb)>
normed = <FusedChannelSpec(b1|b2|b3|r|g|b)>
>>> self = FusedChannelSpec.coerce('B:1:11')
>>> normed = self.normalize()
>>> print('self = {}'.format(self))
>>> print('normed = {}'.format(normed))
self = <FusedChannelSpec(B:1:11)>
normed = <FusedChannelSpec(B.1|B.2|B.3|B.4|B.5|B.6|B.7|B.8|B.9|B.10)>
>>> self = FusedChannelSpec.coerce('B.1:11')
>>> normed = self.normalize()
>>> print('self = {}'.format(self))
>>> print('normed = {}'.format(normed))
self = <FusedChannelSpec(B.1:11)>
normed = <FusedChannelSpec(B.1|B.2|B.3|B.4|B.5|B.6|B.7|B.8|B.9|B.10)>
```

numel()

Total number of channels in this spec

sizes()

Returns a list indicating the size of each atomic code

Returns
List[int]

Example

```
>>> from kwcoco.channel_spec import *  # NOQA
>>> self = FusedChannelSpec.coerce('b1|Z:3|b2|b3|rgb')
>>> self.sizes()
[1, 3, 1, 1, 3]
>>> assert(FusedChannelSpec.parse('a.0').numel()) == 1
>>> assert(FusedChannelSpec.parse('a:0').numel()) == 0
>>> assert(FusedChannelSpec.parse('a:1').numel()) == 1
```

code_list()

Return the expanded code list

as_list()

as_oset()

as_set()

to_set()

to_oset()

to_list()

as_path()

Returns a string suitable for use in a path.

Note, this may no longer be a valid channel spec

difference(*other*)

Set difference

Example

```
>>> FCS = FusedChannelSpec.coerce
>>> self = FCS('rgb|disparity|flowx|flowy')
>>> other = FCS('r|b')
>>> self.difference(other)
>>> other = FCS('flowx')
>>> self.difference(other)
>>> FCS = FusedChannelSpec.coerce
>>> assert len((FCS('a') - {'a'}).parsed) == 0
>>> assert len((FCS('a.0:3') - {'a.0'}).parsed) == 2
```

intersection(*other*)

Example

```
>>> FCS = FusedChannelSpec.coerce
>>> self = FCS('rgb|disparity|flowx|flowy')
>>> other = FCS('r|b|XX')
>>> self.intersection(other)
```

union(*other*)

Example

```
>>> from kwcoco.channel_spec import *  # NOQA
>>> FCS = FusedChannelSpec.coerce
>>> self = FCS('rgb|disparity|flowx|flowy')
>>> other = FCS('r|b|XX')
>>> self.union(other)
```

issubset(*other*)

issuperset(*other*)

component_indices(*axis=2*)

Look up component indices within this stream

Example

```
>>> FCS = FusedChannelSpec.coerce
>>> self = FCS('disparity|rgb|flowx|flowy')
>>> component_indices = self.component_indices()
>>> print('component_indices = {}'.format(ub.repr2(component_indices, nl=1)))
component_indices = {
    'disparity': (slice(...), slice(...), slice(0, 1, None)),
    'flowx': (slice(...), slice(...), slice(4, 5, None)),
    'flowy': (slice(...), slice(...), slice(5, 6, None)),
    'rgb': (slice(...), slice(...), slice(1, 4, None)),
}
```

streams()

Idempotence with *ChannelSpec.streams()*

fuse()

Idempotence with *ChannelSpec.streams()*

class kwcoco.**SensorChanSpec**(*spec: str*)

Bases: NiceRepr

The public facing API for the sensor / channel specification

Example

```
>>> # xdoctest: +REQUIRES(module:lark)
>>> from kwcoco.sensorchan_spec import SensorChanSpec
>>> self = SensorChanSpec('(L8,S2):BGR,WV:BGR,S2:nir,L8:land.0:4')
>>> s1 = self.normalize()
>>> s2 = self.concise()
>>> streams = self.streams()
>>> print(s1)
>>> print(s2)
>>> print('streams = {}'.format(ub.repr2(streams, sv=1, nl=1)))
L8:BGR,S2:BGR,WV:BGR,S2:nir,L8:land.0|land.1|land.2|land.3
(L8,S2,WV):BGR,L8:land:4,S2:nir
streams = [
    L8:BGR,
    S2:BGR,
    WV:BGR,
    S2:nir,
    L8:land.0|land.1|land.2|land.3,
]
```

Example

```
>>> # Check with generic sensors
>>> # xdoctest: +REQUIRES(module:lark)
>>> from kwcoco.sensorchan_spec import SensorChanSpec
>>> import kwcoco
>>> self = SensorChanSpec('(*):BGR,*:BGR,*:nir,*:land.0:4')
>>> self.concise().normalize()
>>> s1 = self.normalize()
>>> s2 = self.concise()
>>> print(s1)
>>> print(s2)
*:BGR,*:BGR,*:nir,*:land.0|land.1|land.2|land.3
(*,*):BGR,*:(nir,land:4)
>>> import kwcoco
>>> c = kwcoco.ChannelSpec.coerce('BGR,BGR,nir,land.0:8')
>>> c1 = c.normalize()
>>> c2 = c.concise()
>>> print(c1)
>>> print(c2)
```

Example

```
>>> # Check empty channels
>>> # xdoctest: +REQUIRES(module:lark)
>>> from kwcoco.sensorchan_spec import SensorChanSpec
>>> import kwcoco
>>> print(SensorChanSpec('*:').normalize())
*:
>>> print(SensorChanSpec('sen:').normalize())
sen:
>>> print(SensorChanSpec('sen:').normalize().concise())
sen:
>>> print(SensorChanSpec('sen:').concise().normalize().concise())
sen:
```

classmethod coerce(*data*)

Attempt to interpret the data as a channel specification

Returns
SensorChanSpec

Example

```
>>> # xdoctest: +REQUIRES(module:lark)
>>> from kwcoco.sensorchan_spec import *  # NOQA
>>> from kwcoco.sensorchan_spec import SensorChanSpec
>>> data = SensorChanSpec.coerce(3)
>>> assert SensorChanSpec.coerce(data).normalize().spec == '*:u0|u1|u2'
>>> data = SensorChanSpec.coerce(3)
>>> assert data.spec == 'u0|u1|u2'
>>> assert SensorChanSpec.coerce(data).spec == 'u0|u1|u2'
>>> data = SensorChanSpec.coerce('u:3')
>>> assert data.normalize().spec == '*:u.0|u.1|u.2'
```

normalize()

concise()

streams()

Returns
List of sensor-names and fused channel specs

Return type
List[*FusedSensorChanSpec*]

late_fuse(**others*)

Example

```
>>> # xdoctest: +REQUIRES(module:lark)
>>> import kwcoco
>>> from kwcoco import sensorchan_spec
>>> import kwcoco
>>> kwcoco.SensorChanSpec = sensorchan_spec.SensorChanSpec  # hack for 3.6
>>> a = kwcoco.SensorChanSpec.coerce('A|B|C,edf')
>>> b = kwcoco.SensorChanSpec.coerce('A12')
>>> c = kwcoco.SensorChanSpec.coerce('')
>>> d = kwcoco.SensorChanSpec.coerce('rgb')
>>> print(a.late_fuse(b).spec)
>>> print((a + b).spec)
>>> print((b + a).spec)
>>> print((a + b + c).spec)
>>> print(sum([a, b, c, d]).spec)
A|B|C,edf,A12
A|B|C,edf,A12
A12,A|B|C,edf
A|B|C,edf,A12
A|B|C,edf,A12,rgb
>>> import kwcoco
>>> a = kwcoco.SensorChanSpec.coerce('A|B|C,edf').normalize()
>>> b = kwcoco.SensorChanSpec.coerce('A12').normalize()
>>> c = kwcoco.SensorChanSpec.coerce('').normalize()
>>> d = kwcoco.SensorChanSpec.coerce('rgb').normalize()
>>> print(a.late_fuse(b).spec)
>>> print((a + b).spec)
>>> print((b + a).spec)
>>> print((a + b + c).spec)
>>> print(sum([a, b, c, d]).spec)
*:A|B|C,*:edf,*:A12
*:A|B|C,*:edf,*:A12
*:A12,*:A|B|C,*:edf
*:A|B|C,*:edf,*:A12,*:
*:A|B|C,*:edf,*:A12,*:,*:rgb
>>> print((a.late_fuse(b)).concise())
>>> print(((a + b)).concise())
>>> print(((b + a)).concise())
>>> print(((a + b + c)).concise())
>>> print((sum([a, b, c, d])).concise())
*:(A|B|C,edf,A12)
*:(A|B|C,edf,A12)
*:(A12,A|B|C,edf)
*:(A|B|C,edf,A12,)
*:(A|B|C,edf,A12,,r|g|b)
```

Example

```
>>> # Test multi-arg case
>>> import kwcoco
>>> a = kwcoco.SensorChanSpec.coerce('A|B|C,edf')
>>> b = kwcoco.SensorChanSpec.coerce('A12')
>>> c = kwcoco.SensorChanSpec.coerce('')
>>> d = kwcoco.SensorChanSpec.coerce('rgb')
>>> others = [b, c, d]
>>> print(a.late_fuse(*others).spec)
>>> print(kwcoco.SensorChanSpec.late_fuse(a, b, c, d).spec)
A|B|C,edf,A12,rgb
A|B|C,edf,A12,rgb
```

matching_sensor(*sensor*)

Get the components corresponding to a specific sensor

Parameters

sensor (*str*) – the name of the sensor to match

Example

```
>>> # xdoctest: +REQUIRES(module:lark)
>>> import kwcoco
>>> self = kwcoco.SensorChanSpec.coerce('(S1,S2):(a|b|c),S2:c|d|e')
>>> sensor = 'S2'
>>> new = self.matching_sensor(sensor)
>>> print(f'new={new}')
new=S2:a|b|c,S2:c|d|e
>>> print(self.matching_sensor('S1'))
S1:a|b|c
>>> print(self.matching_sensor('S3'))
S3:
```

property chans

Returns the channel-only spec, ONLY if all of the sensors are the same

Example

```
>>> # xdoctest: +REQUIRES(module:lark)
>>> import kwcoco
>>> self = kwcoco.SensorChanSpec.coerce('(S1,S2):(a|b|c),S2:c|d|e')
>>> import pytest
>>> with pytest.raises(Exception):
>>>     self.chans
>>> print(self.matching_sensor('S1').chans.spec)
>>> print(self.matching_sensor('S2').chans.spec)
a|b|c
a|b|c,c|d|e
```

class kwcoco.CocoSqlDatabase(*uri=None*, *tag=None*, *img_root=None*)

Bases: *AbstractCocoDataset*, *MixinCocoAccessors*, *MixinCocoObjects*, *MixinCocoStats*, *MixinCocoDraw*, NiceRepr

Provides an API nearly identical to kwcoco.CocoDatabase, but uses an SQL backend data store. This makes it robust to copy-on-write memory issues that arise when forking, as discussed in[1].

Note: By default constructing an instance of the CocoSqlDatabase does not create a connection to the databse. Use the *connect()* method to open a connection.

References

Example

```
>>> # xdoctest: +REQUIRES(module:sqlalchemy)
>>> from kwcoco.coco_sql_dataset import *  # NOQA
>>> sql_dset, dct_dset = demo()
>>> dset1, dset2 = sql_dset, dct_dset
>>> tag1, tag2 = 'dset1', 'dset2'
>>> assert_dsets_allclose(sql_dset, dct_dset)
```

MEMORY_URI = 'sqlite:///:memory:'

classmethod coerce(*data*)

Create an SQL CocoDataset from the input pointer.

Example

import kwcoco dset = kwcoco.CocoDataset.demo('shapes8') data = dset.fpath self = CocoSqlDatabase.coerce(data)

from kwcoco.coco_sql_dataset import CocoSqlDatabase import kwcoco dset = kwcoco.CocoDataset.coerce('spacenet7.kwcoco.json')

self = CocoSqlDatabase.coerce(dset)

from kwcoco.coco_sql_dataset import CocoSqlDatabase sql_dset = CocoSqlDatabase.coerce('spacenet7.kwcoco.json')

from kwcoco.coco_sql_dataset import CocoSqlDatabase import kwcoco sql_dset = kwcoco.CocoDataset.coerce('_spacenet7.kwcoco.view.v006.sqlite')

disconnect()

Drop references to any SQL or cache objects

connect(*readonly=False*)

Connects this instance to the underlying database.

[1] https://github.com/pytorch/pytorch/issues/13246

References

details on read only mode, some of these didnt seem to work https://github.com/sqlalchemy/sqlalchemy/blob/master/lib/sqlalchemy/dialects/sqlite/pysqlite.py#L71 https://github.com/pudo/dataset/issues/136 https://writeonly.wordpress.com/2009/07/16/simple-read-only-sqlalchemy-sessions/

property fpath

delete()

populate_from(*dset*, *verbose=1*)

Copy the information in a *CocoDataset* into this SQL database.

Example

```
>>> # xdoctest: +REQUIRES(module:sqlalchemy)
>>> from kwcoco.coco_sql_dataset import _benchmark_dset_readtime  # NOQA
>>> import kwcoco
>>> from kwcoco.coco_sql_dataset import *
>>> dset2 = dset = kwcoco.CocoDataset.demo()
>>> dset1 = self = CocoSqlDatabase('sqlite:///:memory:')
>>> self.connect()
>>> self.populate_from(dset)
>>> assert_dsets_allclose(dset1, dset2, tag1='sql', tag2='dct')
>>> ti_sql = _benchmark_dset_readtime(dset1, 'sql')
>>> ti_dct = _benchmark_dset_readtime(dset2, 'dct')
>>> print('ti_sql.rankings = {}'.format(ub.repr2(ti_sql.rankings, nl=2,␣
↪precision=6, align=':')))
>>> print('ti_dct.rankings = {}'.format(ub.repr2(ti_dct.rankings, nl=2,␣
↪precision=6, align=':')))
```

property dataset

property anns

property cats

property imgs

property name_to_cat

raw_table(*table_name*)

Loads an entire SQL table as a pandas DataFrame

Parameters
table_name (*str*) – name of the table

Returns
pandas.DataFrame

Example

```
>>> # xdoctest: +REQUIRES(module:sqlalchemy)
>>> from kwcoco.coco_sql_dataset import *  # NOQA
>>> self, dset = demo()
>>> table_df = self.raw_table('annotations')
>>> print(table_df)
```

tabular_targets()

Convinience method to create an in-memory summary of basic annotation properties with minimal SQL overhead.

Example

```
>>> # xdoctest: +REQUIRES(module:sqlalchemy)
>>> from kwcoco.coco_sql_dataset import *  # NOQA
>>> self, dset = demo()
>>> targets = self.tabular_targets()
>>> print(targets.pandas())
```

property bundle_dpath

property data_fpath

data_fpath is an alias of fpath

BIBLIOGRAPHY

[PowersMetrics] https://csem.flinders.edu.au/research/techreps/SIE07001.pdf

[MatlabBM] https://www.mathworks.com/matlabcentral/fileexchange/5648-bm-cm-?requestedDomain=www.mathworks.com

[MulticlassMCC] Jurman, Riccadonna, Furlanello, (2012). A Comparison of MCC and CEN Error Measures in MultiClass Prediction

[CocoFormat] http://cocodataset.org/#format-data

[PyCocoToolsMask] https://github.com/nightrome/cocostuffapi/blob/master/PythonAPI/pycocotools/mask.py

[CocoTutorial] https://www.immersivelimit.com/tutorials/create-coco-annotations-from-scratch/#coco-dataset-format

PYTHON MODULE INDEX

k

INDEX

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X