kwcoco Documentation

Release 0.3.4

Jon Crall

Aug 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

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'>: '/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
```

(continues on next page)

(continued from previous page)

```
>>> #
>>> 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
>>> }
```

(continues on next page)

(continued from previous page)

```
>>> 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
```

(continues on next page)

(continued from previous page)

```
>>> 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

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)
>>> 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()
>>>     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)
```

(continues on next page)

(continued from previous page)

```
>>> pred_dset._check_json_serializable()
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> gid = 1
>>> canvas = true_dset.delayed_load(gid).finalize()
>>> 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

2.1.1.3.1.4 kwcoco.demo.toydata_image module

2.1.1.3.1.5 kwcoco.demo.toydata_video module

2.1.1.3.1.6 kwcoco.demo.toypatterns module

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')
```

(continues on next page)

(continued from previous page)

```
>>> 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?

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

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:

Args:

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)
>>> print(report['confusion'])
pred  1  2  3  r
real
1     3  1  1   5
2     0  4  1   5
3     1  1  6   8
```

(continues on next page)

(continued from previous page)

```
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('
```

‘.join(logs))

Ignore:

```
>>> size = 100
>>> rng = np.random.RandomState(0)
>>> p_classes = np.array([.90, .05, .05][0:2])
>>> p_classes = p_classes / p_classes.sum()
>>> p_wrong   = np.array([.03, .01, .02][0:2])
>>> y_true = testdata_ytrue(p_classes, p_wrong, size, rng)
>>> rs = []
>>> for x in range(17):
>>>     p_wrong += .05
>>>     y_pred = testdata_ypred(y_true, p_wrong, rng)
>>>     report = classification_report(y_true, y_pred, verbose='hack')
>>>     rs.append(report)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> import pandas as pd
>>> df = pd.DataFrame(rs).drop(['raw'], axis=1)
>>> delta = df.subtract(df['target'], axis=0)
>>> sqrd_error = np.sqrt((delta ** 2).sum(axis=0))
>>> print('Error')
```

(continues on next page)

(continued from previous page)

```
>>> print(sqrd_error.sort_values())
>>> ys = df.to_dict(orient='list')
>>> kwplot.multi_plot(ydata_list=ys)
```

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)
```

(continues on next page)

(continued from previous page)

```
>>> 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'])))
```

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)
```

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))
```

(continues on next page)

(continued from previous page)

```
>>> 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))
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))
>>> 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()
```

(continues on next page)

(continued from previous page)

```
>>> 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
```

(continues on next page)

(continued from previous page)

```
>>> info = ub.peek(cfsn_vecs.binarize_ovr().measures()['perclass'].values())
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> 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)
>>> # 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)
>>> # 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
```

(continues on next page)

(continued from previous page)

```
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.**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))
```

(continues on next page)

(continued from previous page)

```
>>>     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]
```

(continues on next page)

(continued from previous page)

```
>>>     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))
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)
>>> 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],
>>> })
```

(continues on next page)

(continued from previous page)

```
>>> # 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)
```

(continues on next page)

(continued from previous page)

```
>>> 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':␣
```

(continues on next page)

(continued from previous page)

```
↪[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'])))
```

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 Submodules

2.1.1.6.1.1 kwcoco.util.dict_like module

2.1.1.6.1.2 kwcoco.util.jsonschema_elements module

2.1.1.6.1.3 kwcoco.util.lazy_frame_backends module

2.1.1.6.1.4 kwcoco.util.util_archive module

2.1.1.6.1.5 kwcoco.util.util_delayed_poc module

2.1.1.6.1.6 kwcoco.util.util_futures module

2.1.1.6.1.7 kwcoco.util.util_json module

2.1.1.6.1.8 kwcoco.util.util_monkey module

2.1.1.6.1.9 kwcoco.util.util_reroot module

2.1.1.6.1.10 kwcoco.util.util_sklearn module

2.1.1.6.1.11 kwcoco.util.util_truncate module

2.1.1.6.2 Module contents

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
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
```

(continues on next page)

(continued from previous page)

```
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)>
```

(continues on next page)

(continued from previous page)

```
>>> 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'
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)
```

(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))),
}
```

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
}

# 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
    '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
```

(continues on next page)

(continued from previous page)

```
        {
            '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]>,
            '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
        }
```

(continues on next page)

(continued from previous page)

```
# 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, ...],
    '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
```

(continues on next page)

(continued from previous page)

```
    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
                }, ...
            ]
        }, ...
    ],
    '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:
```

(continues on next page)

(continued from previous page)

```
    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,
        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",
```

(continues on next page)

(continued from previous page)

```
↪"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:

    {
        '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
- [X] remove videos

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()))
>>> #
```

(continues on next page)

(continued from previous page)

```
>>> 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

```
>>> self = 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

```
>>> self = 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

```
>>> self = 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*, ***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.
- ****kw** – passed to whatever constructor is chosen (if any)

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 = {
>>>     '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))
>>>     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'])
```

(continues on next page)

(continued from previous page)

```
>>> 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*, *vidid=None*, *names=None*)

Return vectorized image objects

Parameters

- **gids** (*List[int]*) – image ids to reference, if unspecified all images are returned.
- **vidid** (*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')
>>> vidid = 1
>>> images = self.images(vidid=vidid)
>>> assert all(v == vidid 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

```
>>> from kwcoco.coco_dataset import *
>>> self = 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

```
>>> from kwcoco.coco_dataset import *
>>> self = 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

```
>>> self = 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

```
>>> from kwcoco.coco_dataset import *
>>> self = 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,
```

(continues on next page)

(continued from previous page)

```
    '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

```
>>> self = 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*)

DEPRECATE: 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*, ***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.
- ****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(--show)
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('vidshapes8-msi')
>>> dset.show_image(gid=1, channels='B8')
```

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

```
>>> self = 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

```
>>> self = 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

```
>>> self = 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

```
>>> imgs = CocoDataset.demo().dataset['images']
>>> self = 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

```
>>> self = 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

```
>>> self = 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

```
>>> self = 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

```
>>> from kwcoco.coco_dataset import *
>>> self = 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

```
>>> from kwcoco.coco_dataset import *
>>> self = 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

```
>>> self = 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',
}
>>> #
>>> # 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,)},
```

(continues on next page)

(continued from previous page)

```
    '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

```
>>> coco_dset = 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

```
>>> from kwcoco.coco_dataset import *
>>> self = 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

```
>>> from kwcoco.coco_dataset import *
>>> self = CocoDataset.demo()
>>> text = self.dumps(newlines=True)
>>> print(text)
>>> self2 = 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 = CocoDataset(json.loads(text), tag='demo2')
>>> assert self2.dataset == self.dataset
>>> assert self2.dataset is not self.dataset
```

Example

```
>>> from kwcoco.coco_dataset import *
>>> self = 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
>>> from kwcoco.coco_dataset import *
>>> self = 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 = 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 = CocoDataset(dataset, tag='demo2')
>>> 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

```
>>> # Test union works with different keypoint categories
>>> dset1 = CocoDataset.demo('shapes1')
>>> dset2 = CocoDataset.demo('shapes2')
>>> dset1.remove_keypoint_categories(['bot_tip', 'mid_tip', 'right_eye'])
>>> dset2.remove_keypoint_categories(['top_tip', 'left_eye'])
>>> dset_12a = 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
>>> import kwcoco
>>> dset1 = kwcoco.CocoDataset.demo('shapes3')
>>> 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):
```

(continues on next page)

(continued from previous page)

```
>>>     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

```
>>> self = 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

```
>>> self = CocoDataset.demo()
>>> sub1 = self.subset([1])
>>> sub2 = self.subset([2])
>>> sub3 = self.subset([3])
>>> others = [sub1, sub2, sub3]
>>> rejoined = 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)
>>> from kwcoco.coco_dataset import demo_coco_data, CocoDataset
>>> dataset = demo_coco_data()
>>> self = 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

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.

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.

Args:
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.

wc

- [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))
```

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

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), (300, 250), (256, 256)]

property area

Example: >>> import kwcoco >>> self = kwcoco.CocoDataset.demo().images() >>> self._dset._ensure_imgsize() >>> print(self.area) [262144, 75000, 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()

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

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],
           [124,  96,  45,  18]]))>
```

property xywh

Returns raw boxes

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo().annots([1, 2, 11])
>>> print(self.xywh)
```

class kwcoco.coco_objects1d.**AnnotGroups**(*groups*, *dset*)

Bases: *ObjectGroups*

property cids

property cnames

class kwcoco.coco_objects1d.**ImageGroups**(*groups*, *dset*)

Bases: *ObjectGroups*

2.1.2.8 kwcoco.coco_schema module

2.1.2.9 kwcoco.coco_sql_dataset module

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 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)
```

(continues on next page)

(continued from previous page)

```
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')
>>> 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.**Transformer**

Bases: object

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,
    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(*other*)

Example

```
>>> # xdoctest: +REQUIRES(module:lark)
>>> import kwcoco
>>> 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)
```

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.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')
>>> 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]
```

(continues on next page)

(continued from previous page)

```
>>> 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))
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)
>>>     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))),
}
```

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

```
>>> coco_dset = 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

```
>>> from kwcoco.coco_dataset import *
>>> self = 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

```
>>> from kwcoco.coco_dataset import *
>>> self = CocoDataset.demo()
>>> text = self.dumps(newlines=True)
>>> print(text)
>>> self2 = 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 = CocoDataset(json.loads(text), tag='demo2')
>>> assert self2.dataset == self.dataset
>>> assert self2.dataset is not self.dataset
```

Example

```
>>> from kwcoco.coco_dataset import *
>>> self = 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
>>> from kwcoco.coco_dataset import *
>>> self = 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 = 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 = CocoDataset(dataset, tag='demo2')
>>> 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

```
>>> # Test union works with different keypoint categories
>>> dset1 = CocoDataset.demo('shapes1')
>>> dset2 = CocoDataset.demo('shapes2')
>>> dset1.remove_keypoint_categories(['bot_tip', 'mid_tip', 'right_eye'])
>>> dset2.remove_keypoint_categories(['top_tip', 'left_eye'])
>>> dset_12a = 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
>>> import kwcoco
>>> dset1 = kwcoco.CocoDataset.demo('shapes3')
>>> 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):
```

(continues on next page)

(continued from previous page)

```
>>>     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

```
>>> self = 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

```
>>> self = CocoDataset.demo()
>>> sub1 = self.subset([1])
>>> sub2 = self.subset([2])
>>> sub3 = self.subset([3])
>>> others = [sub1, sub2, sub3]
>>> rejoined = 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.

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.

Args:
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.

wc

- [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
```

(continues on next page)

(continued from previous page)

```
>>> 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()*

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

D

E

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X