kwcoco Documentation

Release 0.7.6

Jon Crall

Jan 30, 2024

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.

Documentation about higher level kwcoco concepts can be found here. 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.urepr(self.basic_stats(), nl=1))
>>> print(ub.urepr(self.extended_stats(), nl=2))
>>> print(ub.urepr(self.boxsize_stats(), nl=3))
>>> print(ub.urepr(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.urepr(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.urepr(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.urepr(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.__main__ module

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

kw = dict(command='stats') cmdline = False

2.1.1.1.1.2 kwcoco.cli.coco_conform module

class `kwcoco.cli.coco_conform.CocoConformCLI`

Bases: `object`

name = 'conform'

class CLIConfig(**args*, ***kwargs*)

Bases: `DataConfig`

Infer properties to make the COCO file conform to different specs.

Arguments can be used to control which information is inferred. By default, information such as image size, annotation area, are added to the file.

Other arguments like `--legacy` and `--mmlab` can be used to conform to specifications expected by external tooling.

Valid options: []

Parameters

- ***args** – positional arguments for this data config
- ****kwargs** – keyword arguments for this data config

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

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

Example

```
>>> from kwcoco.cli.coco_conform import *  # NOQA
>>> import kwcoco
>>> import ubelt as ub
>>> dpath = ub.Path.appdir('kwcoco/tests/cli/conform').ensuredir()
>>> dst = dpath / 'out.kwcoco.json'
>>> kw = {'src': 'special:shapes8', 'dst': dst, 'compress': True}
>>> cmdline = False
>>> cls = CocoConformCLI
>>> cls.main(cmdline, **kw)
```

kwcoco.cli.coco_conform.**_CLI**

alias of *CocoConformCLI*

2.1.1.1.1.3 kwcoco.cli.coco_eval module

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

class kwcoco.cli.coco_eval.**CocoEvalCLIConfig**(**args*, ***kwargs*)

Bases: *CocoEvalConfig*

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

Valid options: []

Parameters

- ***args** – positional arguments for this data config
- ****kwargs** – keyword arguments for this data config

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

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

Bases: object

name = 'eval'

CLIConfig
alias of *CocoEvalCLIConfig*

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

Example

```
>>> # xdoctest: +REQUIRES(module:kwplot)
>>> from kwcoco.cli.coco_eval import *  # NOQA
>>> import ubelt as ub
>>> from kwcoco.cli.coco_eval import *  # NOQA
>>> from os.path import join
>>> import kwcoco
>>> dpath = ub.Path.appdir('kwcoco/tests/eval').ensuredir()
>>> true_dset = kwcoco.CocoDataset.demo('shapes8')
>>> from kwcoco.demo.perterb import perterb_coco
>>> kwargs = {
>>>     'box_noise': 0.5,
>>>     'n_fp': (0, 10),
>>>     'n_fn': (0, 10),
>>> }
>>> pred_dset = perterb_coco(true_dset, **kwargs)
>>> true_dset.fpath = join(dpath, 'true.mscoco.json')
>>> pred_dset.fpath = join(dpath, 'pred.mscoco.json')
>>> true_dset.dump(true_dset.fpath)
>>> pred_dset.dump(pred_dset.fpath)
>>> draw = False  # set to false for faster tests
>>> CocoEvalCLI.main(cmdline=False,
>>>     true_dataset=true_dset.fpath,
>>>     pred_dataset=pred_dset.fpath,
>>>     draw=draw, out_dpath=dpath)
```

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

Todo:

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

CommandLine

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

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

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

`kwcoco.cli.coco_eval._CLI`
alias of *CocoEvalCLI*

2.1.1.1.1.4 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

Parameters

- **data** (*object*) – filepath, dict, or None
- **default** (*dict* | *None*) – overrides the class defaults
- **cmdline** (*bool* | *List[str]* | *str* | *dict*) – If False, then no command line information is used. If True, then sys.argv is parsed and used. If a list of strings that used instead of sys.argv. If a string, then that is parsed using shlex and used instead

  of sys.argv.

  If a dictionary grants fine grained controls over the args passed to `Config._read_argv()`. Can contain:

  – strict (bool): defaults to False

  – argv (List[str]): defaults to None

  – special_options (bool): defaults to True

  – autocomplete (bool): defaults to False

  Defaults to False.

Note: Avoid setting `cmdline` parameter here. Instead prefer to use the `cli` classmethod to create a command line aware config instance..

---

default = {'dpath': <Path(Path('/home/docs/.cache/kwcoco/data'))>, 'names': <Value([])>}

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

`kwcoco.cli.coco_grab._CLI`
alias of *CocoGrabCLI*

2.1.1.1.1.5 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

Parameters

- **data** (*object*) – filepath, dict, or None
- **default** (*dict* | *None*) – overrides the class defaults
- **cmdline** (*bool* | *List[str]* | *str* | *dict*) – If False, then no command line information is used. If True, then sys.argv is parsed and used. If a list of strings that used instead of sys.argv. If a string, then that is parsed using shlex and used instead

  of sys.argv.

  If a dictionary grants fine grained controls over the args passed to `Config._read_argv()`. Can contain:

  - strict (bool): defaults to False
  - argv (List[str]): defaults to None
  - special_options (bool): defaults to True
  - autocomplete (bool): defaults to False

  Defaults to False.

Note: Avoid setting `cmdline` parameter here. Instead prefer to use the `cli` classmethod to create a command line aware config instance..

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 = {'compress': <Value('auto')>, 'dst': <Value(None)>, 'keep': <Value(None)>, 'keep_annots': <Value(False)>, 'remove': <Value(None)>, 'rename': <Value(None)>, 'src': <Value(None)>}

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

Example

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

kwcoco.cli.coco_modify_categories.**_CLI**

alias of *CocoModifyCatsCLI*

2.1.1.1.1.6 kwcoco.cli.coco_move module

class kwcoco.cli.coco_move.**CocoMove**(**args*, ***kwargs*)

Bases: DataConfig

Move a kwcoco file to a new location while maintaining relative paths. This is equivalent to a regular copy followed by `kwcoco reroot` followed by a delete of the original.

TODO: add option to move the assets as well?

Valid options: []

Parameters

- ***args** – positional arguments for this data config
- ****kwargs** – keyword arguments for this data config

classmethod main(*cmdline=1*, ***kwargs*)

Example

```
>>> import ubelt as ub
>>> from kwcoco.cli import coco_move
>>> import kwcoco
>>> dpath = ub.Path.appdir('kwcoco/doctest/move')
>>> dpath.delete().ensuredir()
>>> dset = kwcoco.CocoDataset.demo('vidshapes2', dpath=dpath)
>>> cmdline = 0
>>> dst = (ub.Path(dset.bundle_dpath) / 'new_dpath').ensuredir()
>>> kwargs = dict(src=dset.fpath, dst=dst)
>>> coco_move.CocoMove.main(cmdline=cmdline, **kwargs)
>>> assert dst.exists()
>>> assert not ub.Path(dset.fpath).exists()
```

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

2.1.1.1.1.7 kwcoco.cli.coco_reroot module

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

Bases: `object`

name = 'reroot'

class CocoRerootConfig(**args*, ***kwargs*)

Bases: `DataConfig`

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

Valid options: []

Parameters

- ***args** – positional arguments for this data config
- ****kwargs** – keyword arguments for this data config

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

CLIConfig

alias of *CocoRerootConfig*

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

Example

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

kwcoco.cli.coco_reroot.**find_reroot_autofix**(*dset*)

kwcoco.cli.coco_reroot.**_check_candidates**(*candidate*, *bundle_dpath*, *missing_gpaths*, *fastfail=True*)

kwcoco.cli.coco_reroot.**_CLI**

alias of *CocoRerootCLI*

2.1.1.1.1.8 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

Parameters

- **data** (*object*) – filepath, dict, or None
- **default** (*dict | None*) – overrides the class defaults
- **cmdline** (*bool | List[str] | str | dict*) – If False, then no command line information is used. If True, then sys.argv is parsed and used. If a list of strings that used instead of sys.argv. If a string, then that is parsed using shlex and used instead

  of sys.argv.

  If a dictionary grants fine grained controls over the args passed to `Config._read_argv()`. Can contain:

  – strict (bool): defaults to False

  – argv (List[str]): defaults to None

  – special_options (bool): defaults to True

  – autocomplete (bool): defaults to False

  Defaults to False.

Note: Avoid setting `cmdline` parameter here. Instead prefer to use the `cli` classmethod to create a command line aware config instance..

---

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

kwcoco.cli.coco_show.**_CLI**

alias of *CocoShowCLI*

2.1.1.1.1.9 kwcoco.cli.coco_split module

class kwcoco.cli.coco_split.**CocoSplitCLI**

Bases: object

Splits a coco files into two parts base on some criteria.

Useful for generating quick and dirty train/test splits, but in general users should opt for using `kwcoco subset` instead to explicitly construct these splits based on domain knowledge.

name = 'split'

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

Bases: Config

Split a single COCO dataset into two sub-datasets.

Parameters

- **data** (*object*) – filepath, dict, or None
- **default** (*dict* | *None*) – overrides the class defaults
- **cmdline** (*bool* | *List[str]* | *str* | *dict*) – If False, then no command line information is used. If True, then sys.argv is parsed and used. If a list of strings that used instead of sys.argv. If a string, then that is parsed using shlex and used instead

  of sys.argv.

  If a dictionary grants fine grained controls over the args passed to `Config._read_argv()`. Can contain:

  – strict (bool): defaults to False

  – argv (List[str]): defaults to None

  – special_options (bool): defaults to True

  – autocomplete (bool): defaults to False

  Defaults to False.

Note: Avoid setting `cmdline` parameter here. Instead prefer to use the `cli` classmethod to create a command line aware config instance..

default = {'balance_categories': <Value(True)>, 'compress': <Value('auto')>, 'dst1': <Value('split1.kwcoco.json')>, 'dst2': <Value('split2.kwcoco.json')>, 'factor': <Value(3)>, 'num_write': <Value(1)>, 'rng': <Value(None)>, 'splitter': <Value('auto')>, 'src': <Value(None)>}

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:vidshapes8',
>>>       'dst1': dpath / 'train.json',
>>>       'dst2': dpath / 'test.json'}
>>> cmdline = False
>>> cls = CocoSplitCLI
>>> cls.main(cmdline, **kw)
```

kwcoco.cli.coco_split.**_CLI**

alias of *CocoSplitCLI*

2.1.1.1.1.10 kwcoco.cli.coco_stats module

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

Bases: `object`

name = 'stats'

class CLIConfig(**args*, ***kwargs*)

Bases: `DataConfig`

Compute summary statistics about a COCO dataset

Valid options: []

Parameters

- ***args** – positional arguments for this data config
- ****kwargs** – keyword arguments for this data config

default = {'annot_attrs': <Value(False)>, 'basic': <Value(True)>, 'boxes': <Value(False)>, 'catfreq': <Value(True)>, 'embed': <Value(False)>, 'extended': <Value(True)>, 'image_attrs': <Value(False)>, 'image_size': <Value(False)>, 'io_workers': <Value(0)>, 'src': <Value(['special:shapes8'])>, 'video_attrs': <Value(False)>}

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

Example

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

`kwcoco.cli.coco_stats._CLI`

alias of *CocoStatsCLI*

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

Example

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

2.1.1.1.1.11 kwcoco.cli.coco_subset module

class `kwcoco.cli.coco_subset.CocoSubsetCLI`

Bases: `object`

name = 'subset'

class CocoSubetConfig(**args*, ***kwargs*)

Bases: `DataConfig`

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

Valid options: []

Parameters

- ***args** – positional arguments for this data config
- ****kwargs** – keyword arguments for this data config

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

CLIConfig

alias of *CocoSubetConfig*

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

Example

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

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

Example

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

Example

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

kwcoco.cli.coco_subset.**_CLI**

alias of *CocoSubsetCLI*

2.1.1.1.1.12 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.

Parameters

- **data** (*object*) – filepath, dict, or None
- **default** (*dict | None*) – overrides the class defaults
- **cmdline** (*bool | List[str] | str | dict*) – If False, then no command line information is used. If True, then sys.argv is parsed and used. If a list of strings that used instead of sys.argv. If a string, then that is parsed using shlex and used instead

  of sys.argv.

  If a dictionary grants fine grained controls over the args passed to `Config._read_argv()`. Can contain:

  – strict (bool): defaults to False

  – argv (List[str]): defaults to None

  – special_options (bool): defaults to True

  – autocomplete (bool): defaults to False

  Defaults to False.

Note: Avoid setting `cmdline` parameter here. Instead prefer to use the `cli` classmethod to create a command line aware config instance..

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 '

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

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)
```

kwcoco.cli.coco_toydata.**_CLI**

alias of *CocoToyDataCLI*

2.1.1.1.1.13 kwcoco.cli.coco_union module

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

Bases: object

name = 'union'

class CLIConfig(**args*, ***kwargs*)

Bases: DataConfig

Combine multiple COCO datasets into a single merged dataset.

Valid options: []

Parameters

- ***args** – positional arguments for this data config
- ****kwargs** – keyword arguments for this data config

default = {'absolute': <Value(False)>, 'compress': <Value('auto')>, 'dst': <Value('combo.kwcoco.json')>, 'io_workers': <Value('avail-2')>, 'remember_parent': <Value(False)>, 'src': <Value([])>}

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

Example

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

`kwcoco.cli.coco_union.`**`_postprocess_absolute`**(*dset*)

`kwcoco.cli.coco_union.`**`_CLI`**

alias of *`CocoUnionCLI`*

2.1.1.1.1.14 kwcoco.cli.coco_validate module

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

Bases: `object`

`name = 'validate'`

`class CLIConfig`(**args*, ***kwargs*)

Bases: `DataConfig`

Validates that a coco file satisfies expected properties.

Checks that a coco file conforms to the json schema, that assets exist, and that other expected properties are satisfied.

This also has the ability to fix corrupted assets by removing them, but that functionality may be moved to a new command in the future.

Valid options: []

Parameters

- ***args** – positional arguments for this data config
- ****kwargs** – keyword arguments for this data config

`default = {'channels': <Value(True)>, 'corrupted': <Value(False)>, 'dst': <Value(None)>, 'fastfail': <Value(False)>, 'fix': <Value(None)>, 'img_attrs': <Value('warn')>, 'missing': <Value(True)>, 'require_relative': <Value(False)>, 'schema': <Value(True)>, 'src': <Value(None)>, 'unique': <Value(True)>, 'verbose': <Value(1)>, 'workers': <Value(0)>}`

`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)
```

kwcoco.cli.coco_validate.**_CLI**

alias of *CocoValidateCLI*

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.**_devcheck_sample_full_image**()

kwcoco.data.grab_camvid.**_devcheck_load_sub_image**()

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

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

Grab a kwcoco.CocoSampler object for the CamVid dataset.

Returns
sampler

Return type
kwcoco.CocoSampler

Example

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

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

Example

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

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

Grab the raw camvid data.

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

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

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

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

Example

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

kwcoco.data.grab_camvid.**_define_camvid_class_hierarcy**(*dset*)

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

Dump the paths to the coco file to stdout

By default these will go to in the path:
~/.cache/kwcoco/camvid/camvid-master

The four files will be:
~/.cache/kwcoco/camvid/camvid-master/camvid-full.mscoco.json ~/.cache/kwcoco/camvid/camvid-master/camvid-train.mscoco.json ~/.cache/kwcoco/camvid/camvid-master/camvid-vali.mscoco.json ~/.cache/kwcoco/camvid/camvid-master/camvid-test.mscoco.json

2.1.1.2.1.2 kwcoco.data.grab_cifar module

2.1.1.2.1.3 kwcoco.data.grab_datasets module

Todo:

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

2.1.1.2.1.4 kwcoco.data.grab_domainnet module

References

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

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

Todo:

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

pattern across the data grab scripts.

2.1.1.2.1.5 kwcoco.data.grab_spacenet module

References

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

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

References

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

Requires:
awscli

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

Converts the raw SpaceNet7 dataset to kwcoco

Note:

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

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

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

2.1.1.2.1.6 kwcoco.data.grab_voc module

`kwcoco.data.grab_voc.`**`__torrent_voc`**`()`

Requires:
pip install deluge pip install python-libtorrent-bin

References

https://academictorrents.com/details/f6ddac36ac7ae2ef79dc72a26a065b803c9c7230

Todo:

- [ ] Is there a pythonic way to download a torrent programatically?

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

`kwcoco.data.grab_voc.`**`_convert_voc_split`**`(`*devkit_dpath*, *classes*, *split*, *year*, *root*`)`

split, year = 'train', 2012 split, year = 'train', 2007

`kwcoco.data.grab_voc.`**`_read_split_paths`**`(`*devkit_dpath*, *split*, *year*`)`

split = 'train' self = VOCDataset('test') year = 2007 year = 2012 `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* | *None*) – 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()
```

(continues on next page)

(continued from previous page)

```
>>>     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)
```

(continues on next page)

(continued from previous page)

```
>>> 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.urepr(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.urepr(square_mat, nl=1)))
```

kwcoco.demo.boids.**_spatial_index_scratch**()

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)
```

kwcoco.demo.boids.**_pygame_render_boids**()

Fast and responsive BOID rendering. This is an easter egg.

Requirements:
pip install pygame

CommandLine

```
python -m kwcoco.demo.boids
pip install pygame kwcoco -U && python -m kwcoco.demo.boids
```

kwcoco.demo.boids.**_yeah_boid**()

2.1.1.3.1.2 kwcoco.demo.perterb module

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

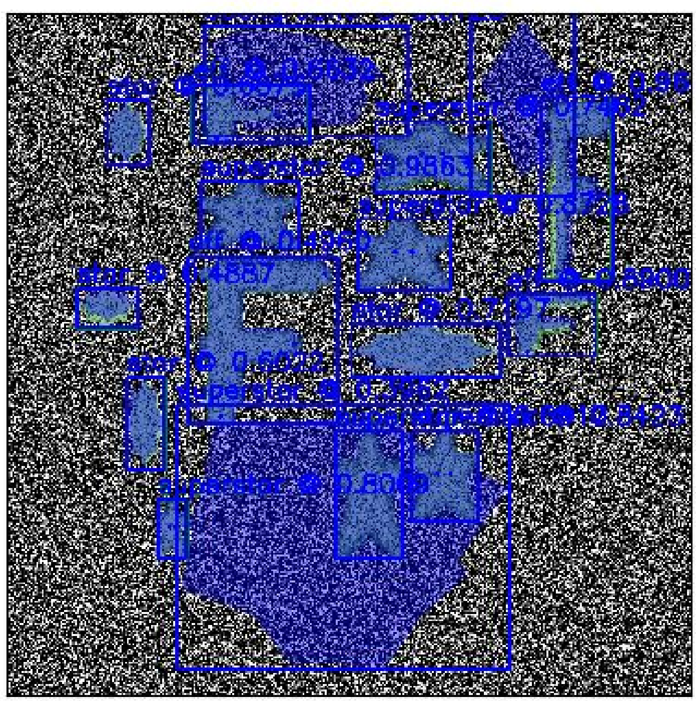
Perterbs a coco dataset

Parameters

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

Example

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



kwcoco.demo.perterb.**_demo_construct_probs**(*pred_cxs*, *pred_scores*, *classes*, *rng*, *hacked=1*)

Constructs random probabilities for demo data

Example

```
>>> import kwcoco
>>> import kwarray
>>> rng = kwarray.ensure_rng(0)
>>> classes = kwcoco.CategoryTree.coerce(10)
>>> hacked = 1
>>> pred_cxs = rng.randint(0, 10, 10)
>>> pred_scores = rng.rand(10)
>>> probs = _demo_construct_probs(pred_cxs, pred_scores, classes, rng, hacked)
>>> probs.sum(axis=1)
```

2.1.1.3.1.3 kwcoco.demo.toydata module

Generates "toydata" for demo and testing purposes.

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

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

Create a toy detection problem

Parameters

- **image_size** (*Tuple[int, int]*) – The width and height of the generated images
- **n_imgs** (*int*) – number of images to generate
- **rng** (*int* | *RandomState* | *None*) – random number generator or seed. Defaults to 0.
- **newstyle** (*bool*) – create newstyle kwcoco data. default=True
- **dpath** (*str* | *PathLike* | *None*) – path to the directory that will contain the bundle, (defaults to a kwcoco cache dir). Ignored if *bundle_dpath* is given.
- **fpath** (*str* | *PathLike* | *None*) – path to the kwcoco file. The parent will be the bundle if it is not specified. Should be a descendant of the dpath if specified.
- **bundle_dpath** (*str* | *PathLike* | *None*) – path to the directory that will store images. If specified, dpath is ignored. If unspecified, a bundle will be written inside *dpath*.
- **aux** (*bool* | *None*) – if True generates dummy auxiliary channels
- **verbose** (*int*) – verbosity mode. default=3
- **use_cache** (*bool*) – if True caches the generated json in the *dpath*. Default=True
- ****kwargs** – used for old backwards compatible argument names gsize - alias for image_size

Return type

kwcoco.CocoDataset

SeeAlso:

random_video_dset

CommandLine

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

Todo:

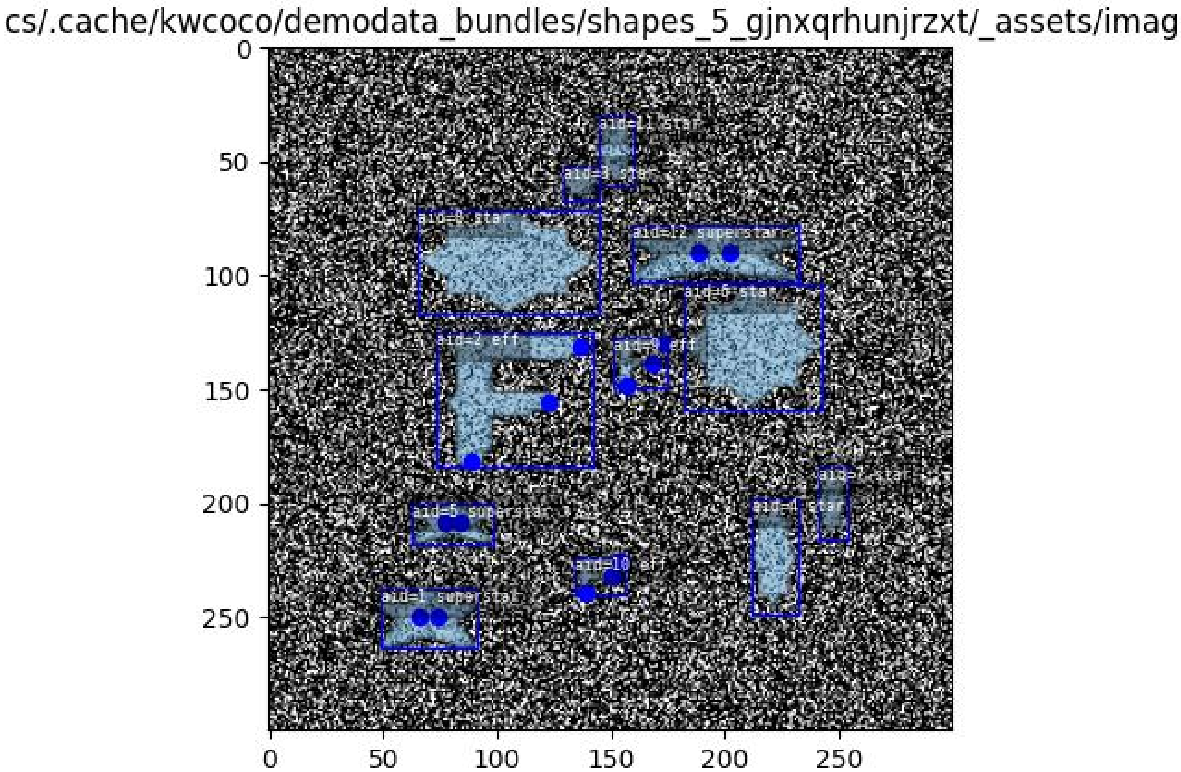
- [ ] Non-homogeneous images sizes

Example

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

dset._tree()

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



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

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

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

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

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

Create the video scene layout of object positions.

Note: Does not render the data unless specified.

Parameters

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

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

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

Todo:

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

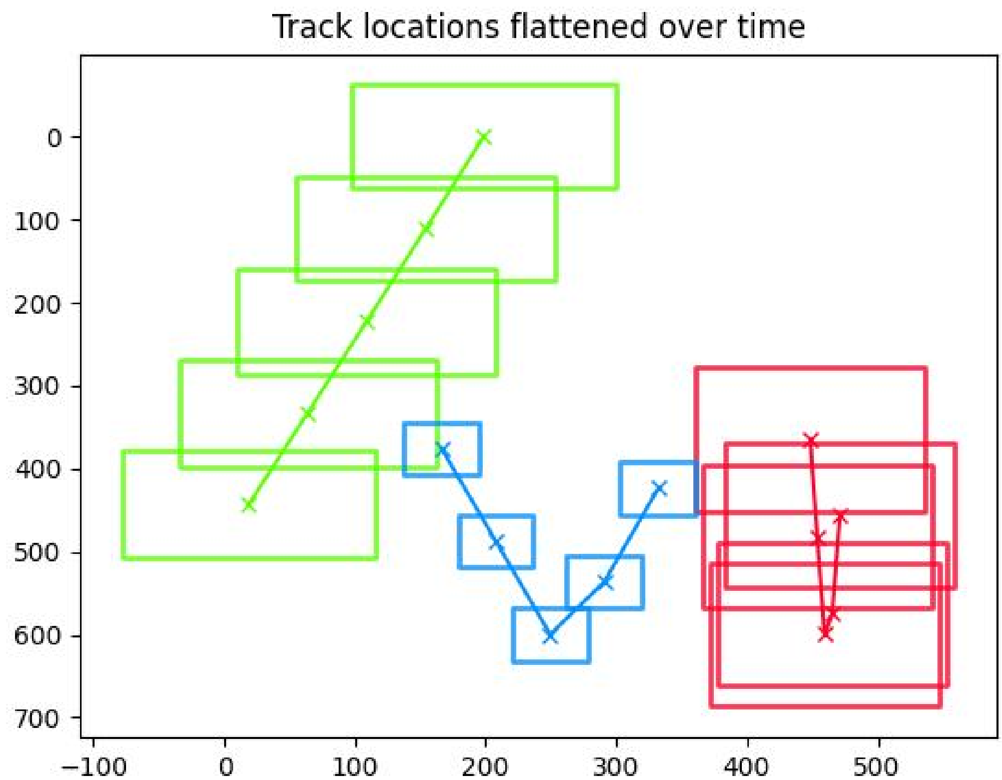
Example

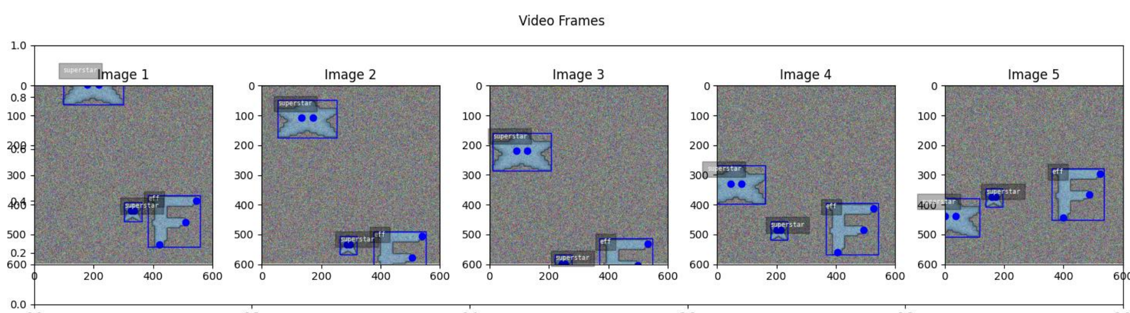
```
>>> import numpy as np
>>> from kwcoco.demo.toydata_video import random_single_video_dset
>>> anchors = np.array([ [0.3, 0.3],  [0.1, 0.1]])
>>> dset = random_single_video_dset(render=True, num_frames=5,
>>>                                 num_tracks=3, anchors=anchors,
>>>                      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

(continues on next page)

(continued from previous page)

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





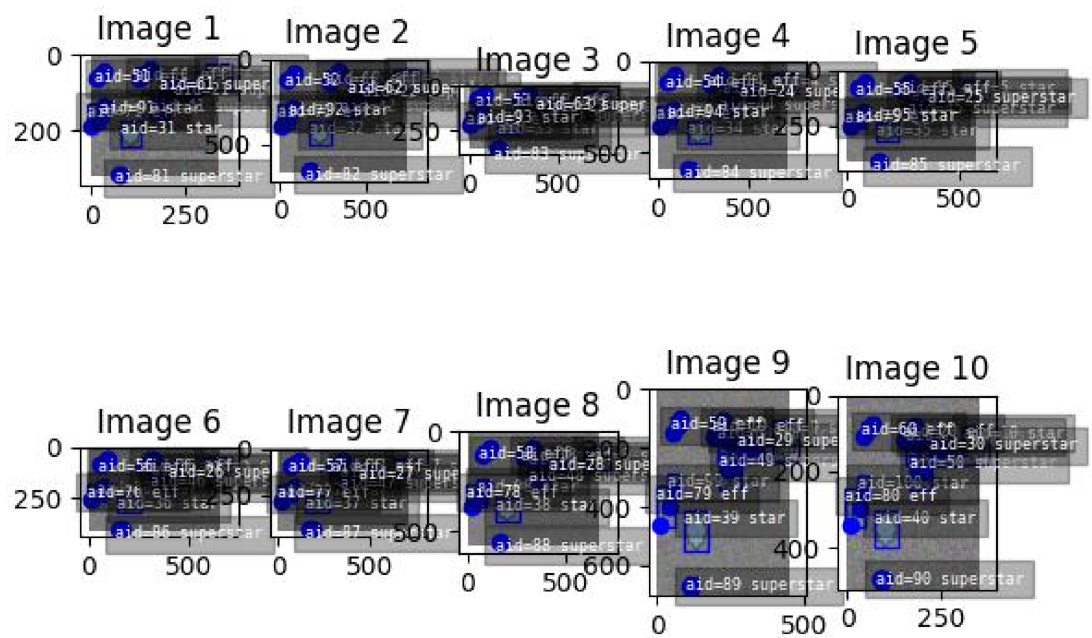
Example

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

(continues on next page)

(continued from previous page)

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



Example

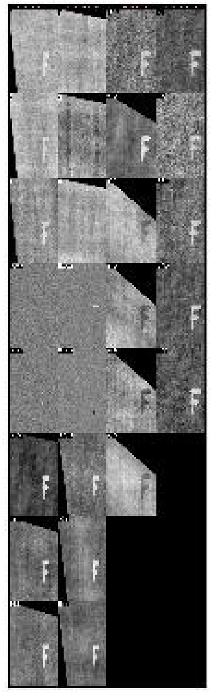
```
>>> from kwcoco.demo.toydata_video import *  # NOQA
>>> dset = random_single_video_dset(num_frames=10, num_tracks=10, aux=True)
>>> assert 'auxiliary' in dset.imgs[1]
>>> assert dset.imgs[1]['auxiliary'][0]['channels']
>>> assert dset.imgs[1]['auxiliary'][1]['channels']
```

Example

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

Example

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



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

Create a toy Coco Video Dataset

Parameters

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

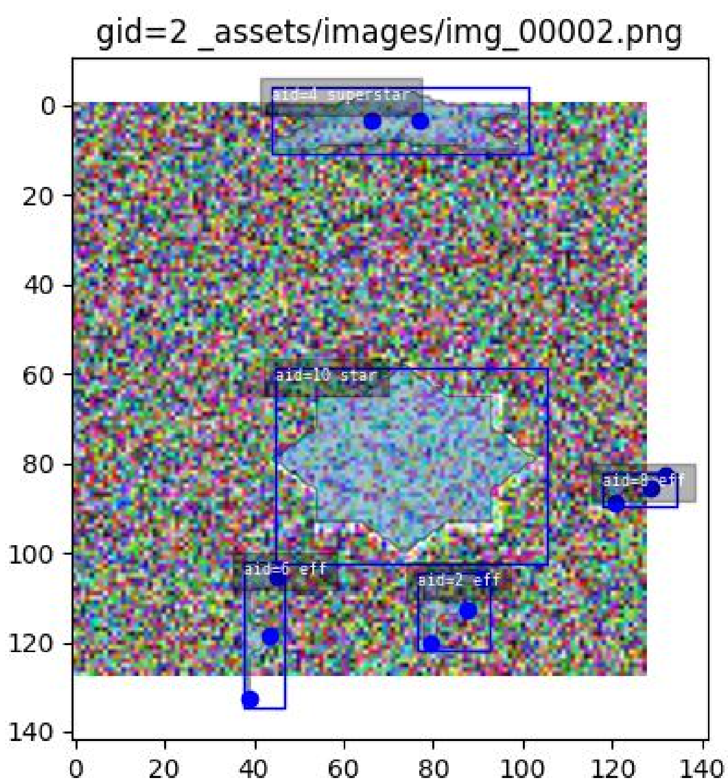
- ****kwargs** – used for old backwards compatible argument names gsize - alias for image_size

SeeAlso:
random_single_video_dset

Example

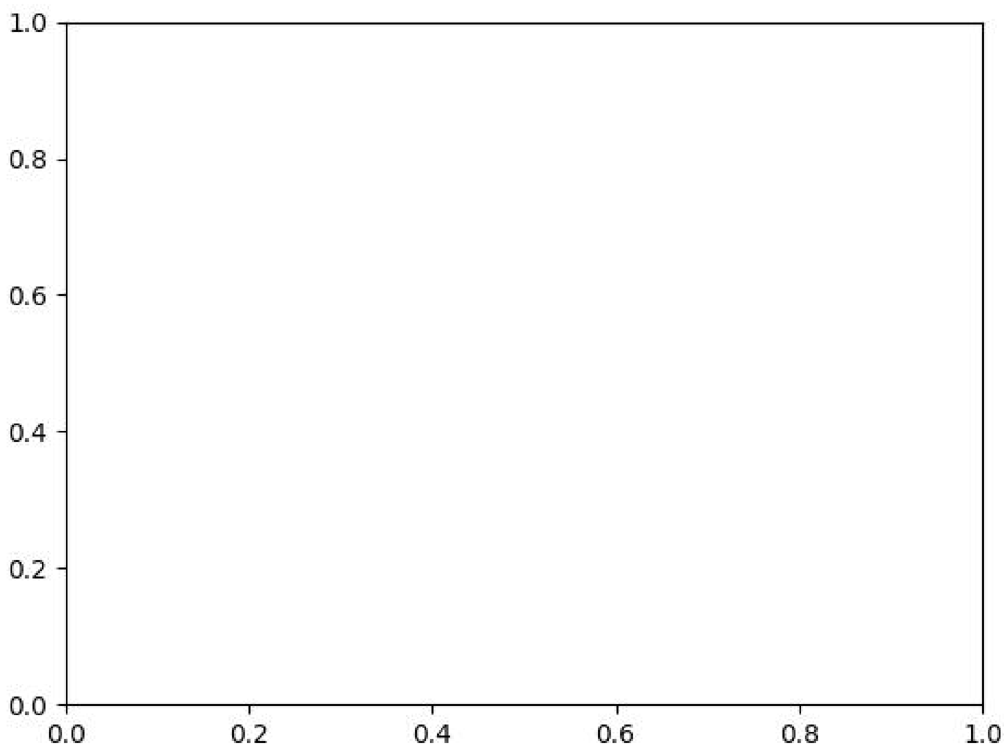
```
>>> from kwcoco.demo.toydata_video import *  # NOQA
>>> dset = random_video_dset(render=True, num_videos=3, num_frames=2,
>>>                          num_tracks=5, image_size=(128, 128))
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> dset.show_image(1, doclf=True)
>>>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

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



Example

```
>>> from kwcoco.demo.toydata_video import *  # NOQA
>>> # Test small images
>>> dset = random_video_dset(render=True, num_videos=1, num_frames=1,
>>>                          num_tracks=1, image_size=(2, 2))
>>> ann = dset.annots().peek()
>>> print('ann = {}'.format(ub.urepr(ann, nl=2)))
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> dset.show_image(1, doclf=True)
```



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

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

Todo:

DEPRECATE IN FAVOR OF
random_single_video_dset + render_toy_image

Parameters

- **anchors** (*ndarray | None*) – Nx2 base width / height of boxes
- **gsize** (*Tuple[int, int]*) – width / height of the image
- **categories** (*List[str] | None*) – list of category names
- **n_annots** (*Tuple | int*) – controls how many annotations are in the image. if it is a tuple, then it is interpreted as uniform random bounds
- **fg_scale** (*float*) – standard deviation of foreground intensity
- **bg_scale** (*float*) – standard deviation of background intensity
- **bg_intensity** (*float*) – mean of background intensity
- **fg_intensity** (*float*) – mean of foreground intensity
- **centerobj** (*bool | None*) – if 'pos', then the first annotation will be in the center of the image, if 'neg', then no annotations will be in the center.
- **exact** (*bool*) – if True, ensures that exactly the number of specified annots are generated.
- **newstyle** (*bool*) – use new-sytle kwcoco format
- **rng** (*RandomState | int | None*) – the random state used to seed the process
- **aux** (*bool | None*) – if specified builds auxiliary channels
- ****kwargs** – used for old backwards compatible argument names. gsize - alias for image_size

CommandLine

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

Example

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

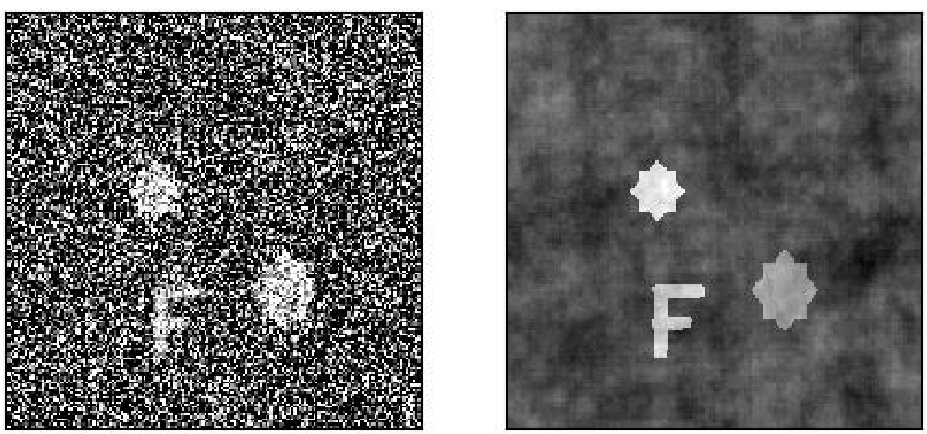
(continues on next page)

(continued from previous page)

```
{'bbox': [4, 4, 8, 6],
 'category_name': 'superstar',
 'keypoints': [{'keypoint_category': 'left_eye', 'xy': [7.25, 6.8125]}, {'keypoint_
↪category': 'right_eye', 'xy': [8.75, 6.8125]}],
 'segmentation': {'counts': 'U4210j0300O01010O00MVO0ed0', 'size': [32, 32]},},
{'bbox': [3, 20, 6, 7],
 'category_name': 'star',
 'keypoints': [],
 'segmentation': {'counts': 'g31m04N000002L[f0', 'size': [32, 32]},},]
```

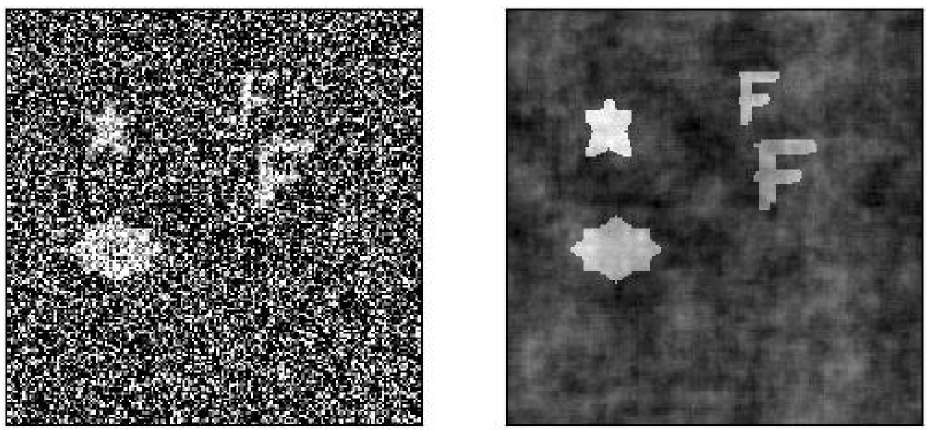
Example

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



Example

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



2.1.1.3.1.4 kwcoco.demo.toydata_image module

Generates "toydata" for demo and testing purposes.

Loose image version of the toydata generators.

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

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

Create a toy detection problem

Parameters

- **image_size** (*Tuple[int, int]*) – The width and height of the generated images
- **n_imgs** (*int*) – number of images to generate
- **rng** (*int* | *RandomState* | *None*) – random number generator or seed. Defaults to 0.
- **newstyle** (*bool*) – create newstyle kwcoco data. default=True
- **dpath** (*str* | *PathLike* | *None*) – path to the directory that will contain the bundle, (defaults to a kwcoco cache dir). Ignored if *bundle_dpath* is given.
- **fpath** (*str* | *PathLike* | *None*) – path to the kwcoco file. The parent will be the bundle if it is not specified. Should be a descendant of the dpath if specified.
- **bundle_dpath** (*str* | *PathLike* | *None*) – path to the directory that will store images. If specified, dpath is ignored. If unspecified, a bundle will be written inside *dpath*.
- **aux** (*bool* | *None*) – if True generates dummy auxiliary channels
- **verbose** (*int*) – verbosity mode. default=3
- **use_cache** (*bool*) – if True caches the generated json in the *dpath*. Default=True
- ****kwargs** – used for old backwards compatible argument names gsize - alias for image_size

Return type

kwcoco.CocoDataset

SeeAlso:

random_video_dset

CommandLine

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

Todo:

- [ ] Non-homogeneous images sizes

Example

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

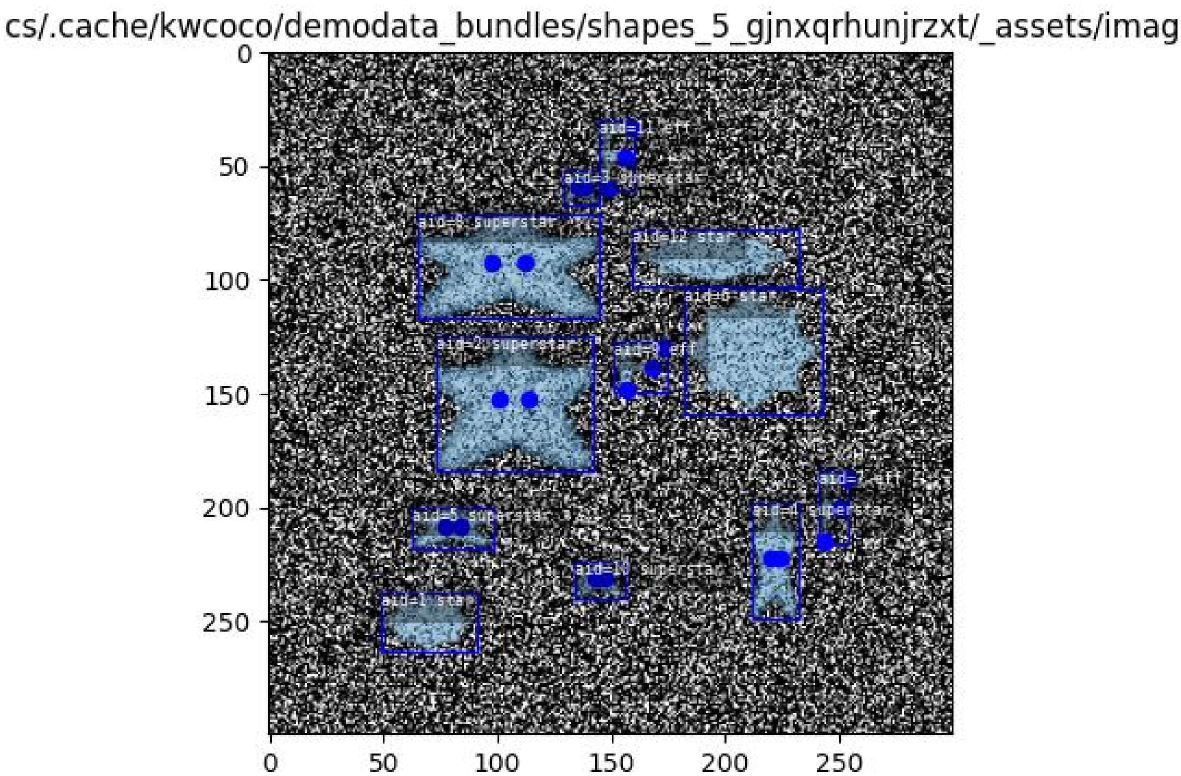
(continues on next page)

(continued from previous page)

```
>>> dset.show_image(gid=1)
>>> ub.startfile(dset.bundle_dpath)
```

dset._tree()

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



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

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

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

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

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

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

Todo:

DEPRECATE IN FAVOR OF
random_single_video_dset + render_toy_image

Parameters

- **anchors** (*ndarray | None*) – Nx2 base width / height of boxes
- **gsize** (*Tuple[int, int]*) – width / height of the image
- **categories** (*List[str] | None*) – list of category names
- **n_annots** (*Tuple | int*) – controls how many annotations are in the image. if it is a tuple, then it is interpreted as uniform random bounds
- **fg_scale** (*float*) – standard deviation of foreground intensity
- **bg_scale** (*float*) – standard deviation of background intensity
- **bg_intensity** (*float*) – mean of background intensity
- **fg_intensity** (*float*) – mean of foreground intensity
- **centerobj** (*bool | None*) – if 'pos', then the first annotation will be in the center of the image, if 'neg', then no annotations will be in the center.
- **exact** (*bool*) – if True, ensures that exactly the number of specified annots are generated.
- **newstyle** (*bool*) – use new-sytle kwcoco format
- **rng** (*RandomState | int | None*) – the random state used to seed the process
- **aux** (*bool | None*) – if specified builds auxiliary channels
- ****kwargs** – used for old backwards compatible argument names. gsize - alias for image_size

CommandLine

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

Example

```
>>> from kwcoco.demo.toydata_image import *  # NOQA
>>> img, anns = demodata_toy_img(image_size=(32, 32), anchors=[[.3, .3]], rng=0)
>>> img['imdata'] = '<ndarray shape={}>'.format(img['imdata'].shape)
>>> print('img = {}'.format(ub.urepr(img)))
>>> print('anns = {}'.format(ub.urepr(anns, nl=2, cbr=True)))
>>> # xdoctest: +IGNORE_WANT
img = {
    'height': 32,
    'imdata': '<ndarray shape=(32, 32, 3)>',
    'width': 32,
}
anns = [{'bbox': [15, 10, 9, 8],
  'category_name': 'star',
  'keypoints': [],
  'segmentation': {'counts': '[`06j0000O20N1000e8', 'size': [32, 32]},},
```

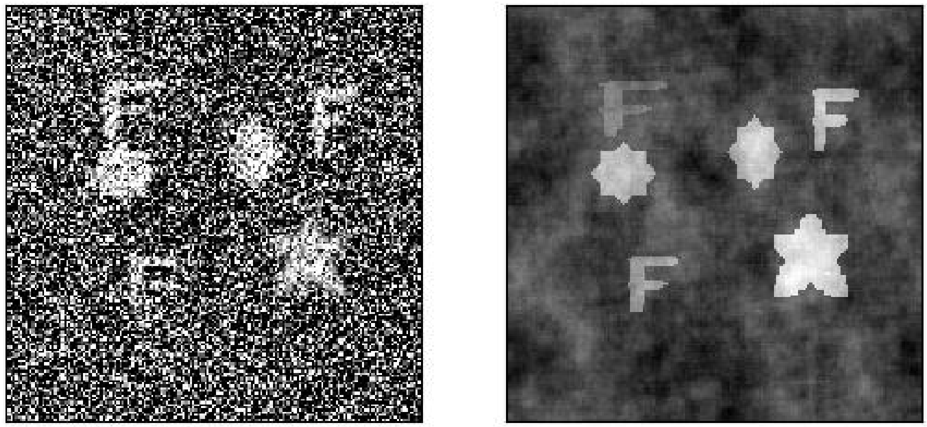
(continues on next page)

(continued from previous page)

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

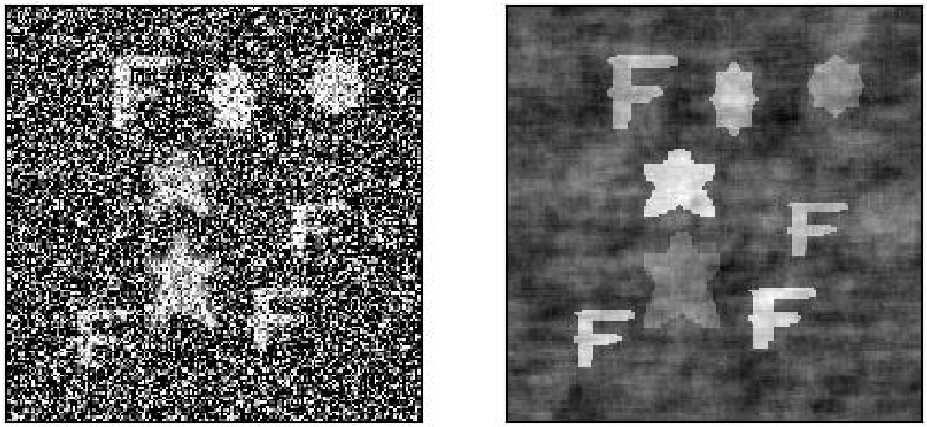
Example

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



Example

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



2.1.1.3.1.5 kwcoco.demo.toydata_video module

Generates "toydata" for demo and testing purposes.

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

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

Create a toy Coco Video Dataset

Parameters

- **num_videos** (*int*) – number of videos
- **num_frames** (*int*) – number of images per video
- **num_tracks** (*int*) – number of tracks per video
- **image_size** (*Tuple[int, int]*) – The width and height of the generated images
- **render** (*bool | dict*) – if truthy the toy annotations are synthetically rendered. See `render_toy_image()` for details.
- **rng** (*int | None | RandomState*) – random seed / state

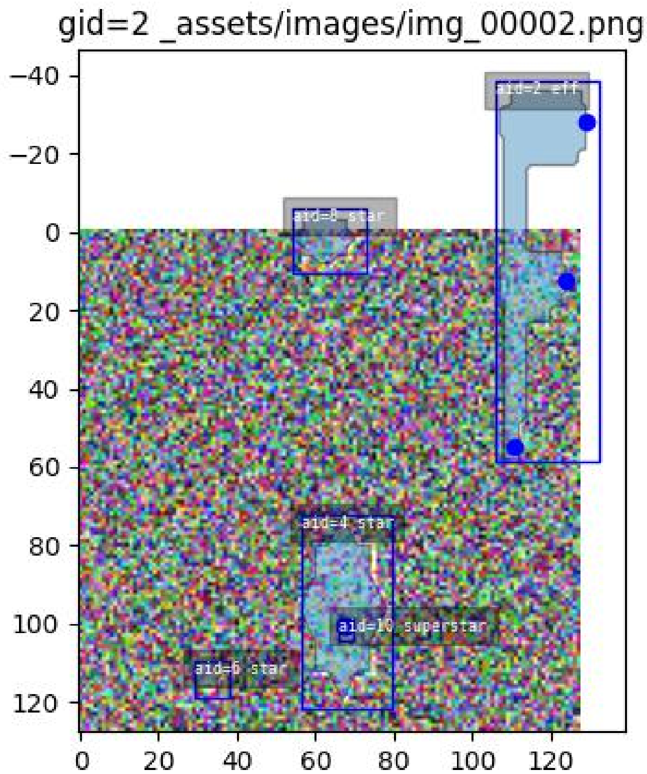
- **dpath** (*str | PathLike | None*) – only used if render is truthy, place to write rendered images.
- **verbose** (*int*) – verbosity mode, default=3
- **aux** (*bool | None*) – if True generates dummy auxiliary / asset channels
- **multispectral** (*bool*) – similar to aux, but does not have the concept of a "main" image.
- **max_speed** (*float*) – max speed of movers
- **channels** (*str | None*) – experimental new way to get MSI with specific band distributions.
- ****kwargs** – used for old backwards compatible argument names gsize - alias for image_size

SeeAlso:
random_single_video_dset

Example

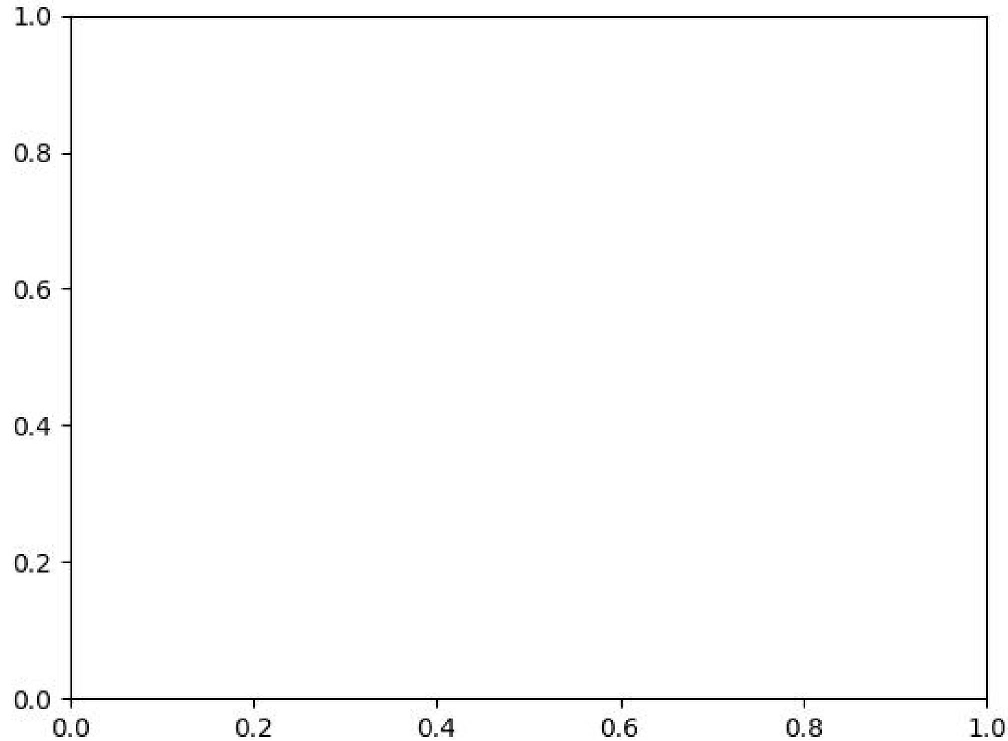
```
>>> from kwcoco.demo.toydata_video import *  # NOQA
>>> dset = random_video_dset(render=True, num_videos=3, num_frames=2,
>>>                          num_tracks=5, image_size=(128, 128))
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> dset.show_image(1, doclf=True)
>>>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

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



Example

```
>>> from kwcoco.demo.toydata_video import *  # NOQA
>>> # Test small images
>>> dset = random_video_dset(render=True, num_videos=1, num_frames=1,
>>>                          num_tracks=1, image_size=(2, 2))
>>> ann = dset.annots().peek()
>>> print('ann = {}'.format(ub.urepr(ann, nl=2)))
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> dset.show_image(1, doclf=True)
```



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

Create the video scene layout of object positions.

Note: Does not render the data unless specified.

Parameters

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

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

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

Todo:

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

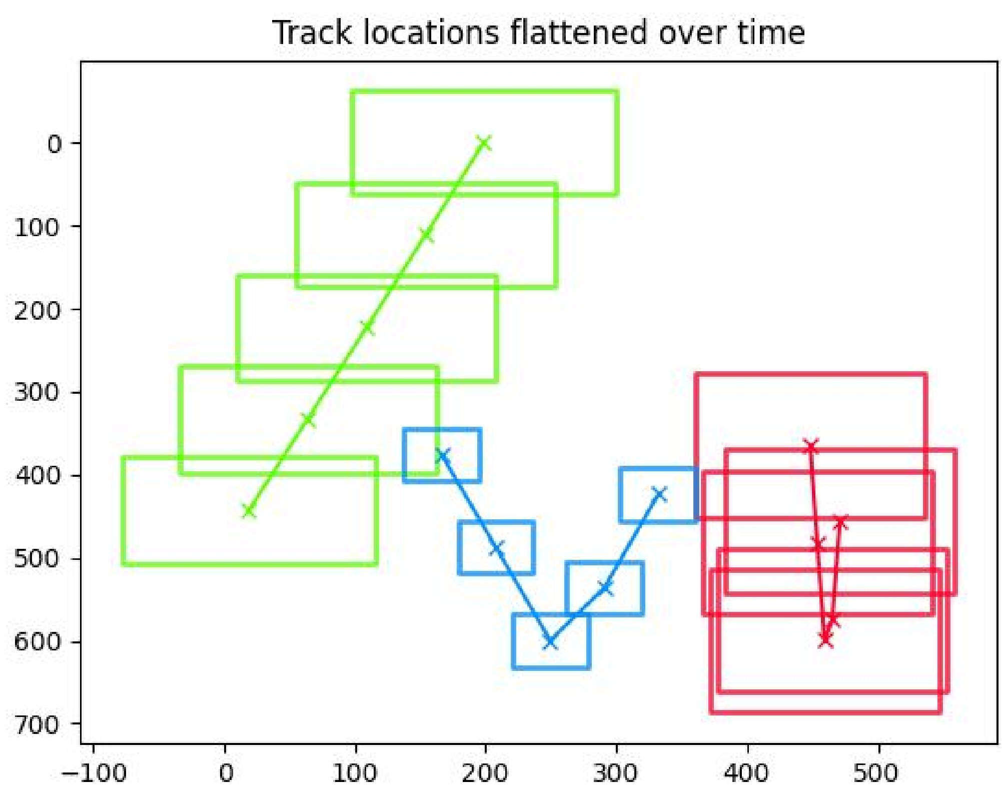
Example

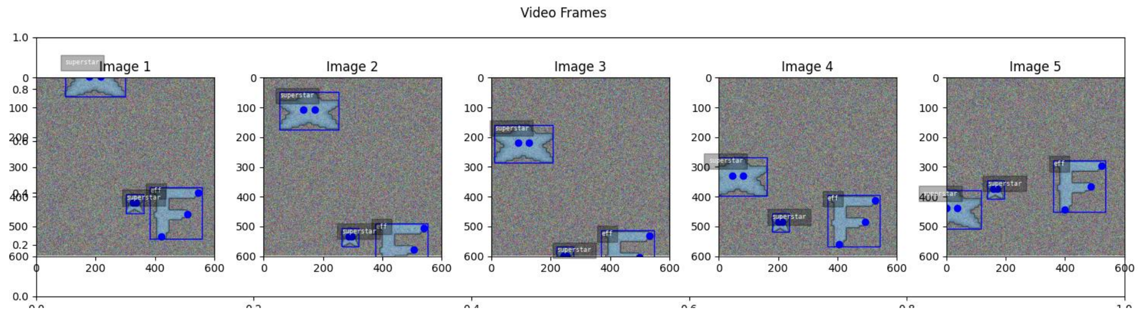
```
>>> import numpy as np
>>> from kwcoco.demo.toydata_video import random_single_video_dset
>>> anchors = np.array([ [0.3, 0.3],  [0.1, 0.1]])
>>> dset = random_single_video_dset(render=True, num_frames=5,
>>>                                 num_tracks=3, anchors=anchors,
>>>                      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

(continues on next page)

(continued from previous page)

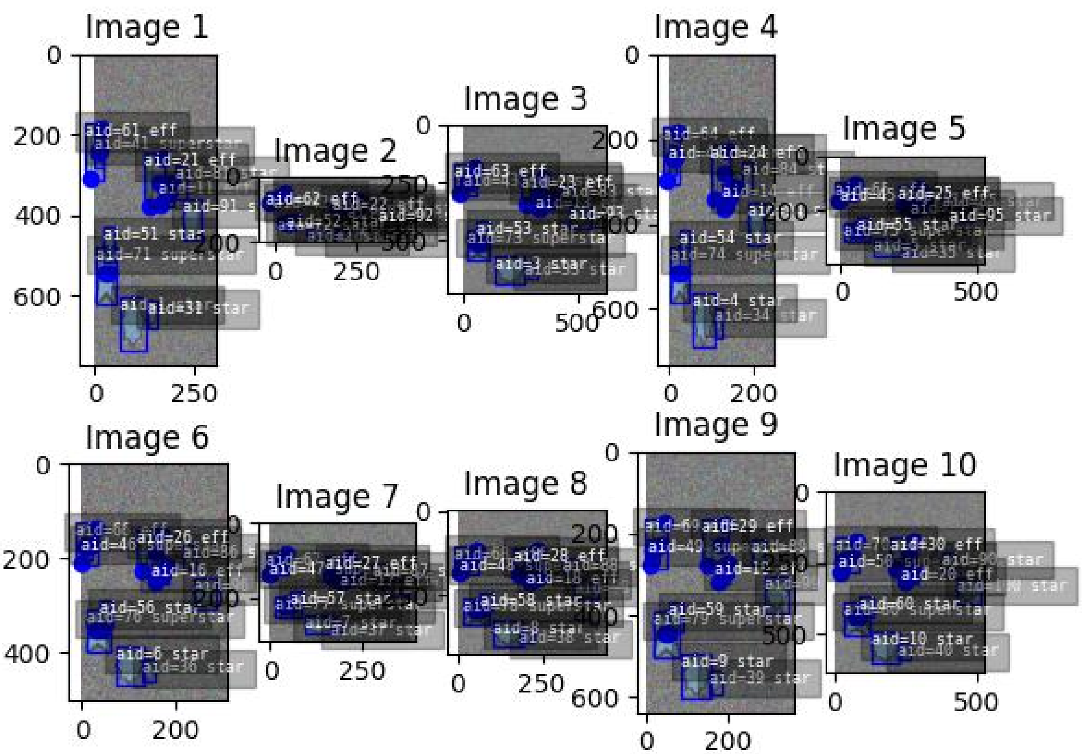
```
>>>     color = colors[i]
>>>     path = boxes.data[:, 0:2]
>>>     boxes.draw(color=color, centers={'radius': 0.01}, alpha=0.8)
>>>     ax.plot(path.T[0], path.T[1], 'x-', color=color)
>>> ax.invert_yaxis()
>>> ax.set_title('Track locations flattened over time')
>>> # Plot the image sequence
>>>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





Example

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



Example

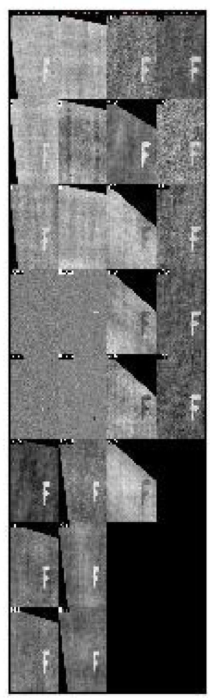
```
>>> from kwcoco.demo.toydata_video import *  # NOQA
>>> dset = random_single_video_dset(num_frames=10, num_tracks=10, aux=True)
>>> assert 'auxiliary' in dset.imgs[1]
>>> assert dset.imgs[1]['auxiliary'][0]['channels']
>>> assert dset.imgs[1]['auxiliary'][1]['channels']
```

Example

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

Example

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



kwcoco.demo.toydata_video.**_draw_video_sequence**(*dset*, *gids*)

Helper to draw a multi-sensor sequence

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

Create toydata_video renderings for a preconstructed coco dataset.

Parameters

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

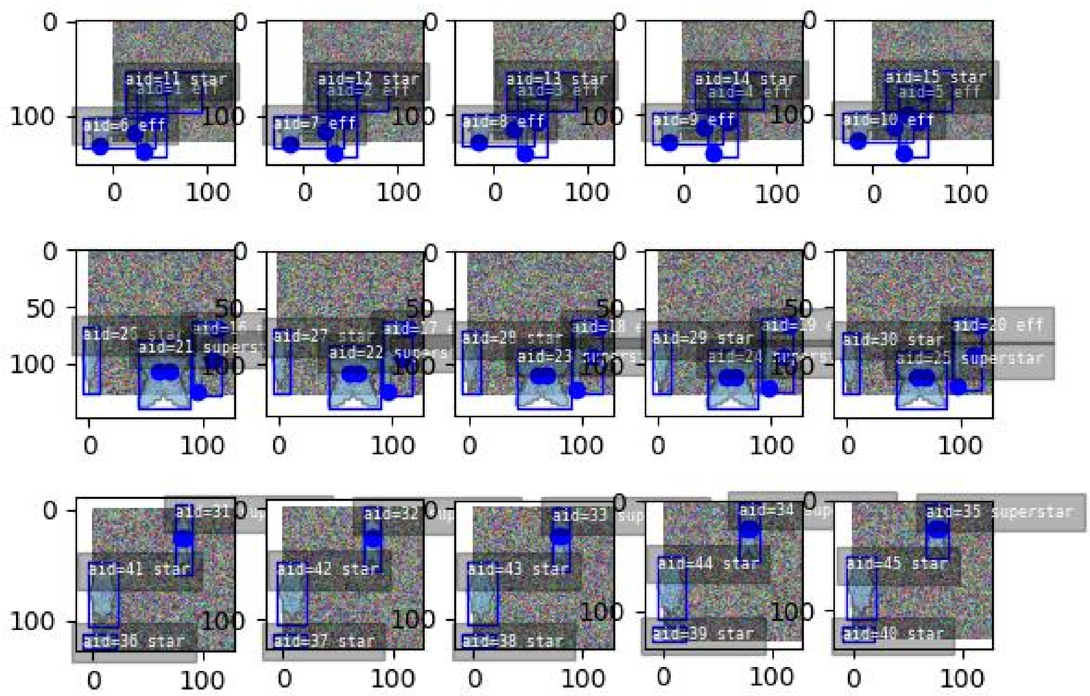
  Currently this process is simple, it just creates a noisy image with the shapes superimposed over where they should exist as indicated by the annotations. In the future this may become more sophisticated.

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

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

Example

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



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

Modifies dataset inplace, rendering synthetic annotations.

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

Parameters

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

Returns
the inplace-modified image dictionary

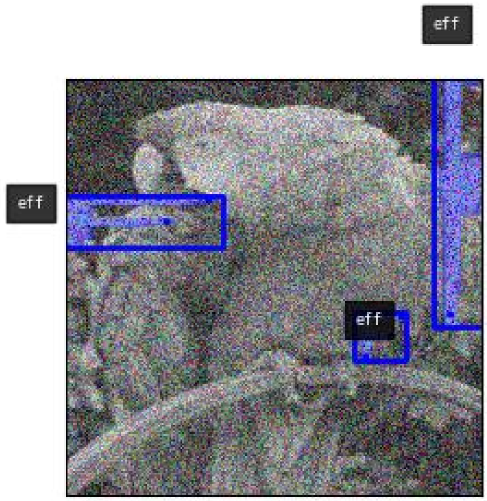
Return type
Dict

Example

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

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

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



Example

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

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

Renders demo annoations on top of a demo background

kwcoco.demo.toydata_video.**render_background**(*img*, *rng*, *gray*, *bg_intensity*, *bg_scale*, *imgspace_background=None*)

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

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

kwcoco.demo.toydata_video.**random_multi_object_path**(*num_objects*, *num_frames*, *rng=None*, *max_speed=0.01*) kwcoco.demo.toydata_video.**random_path**(*num*, *degree=1*, *dimension=2*, *rng=None*, *mode='boid'*)

Create a random path using a somem ethod curve.

Parameters

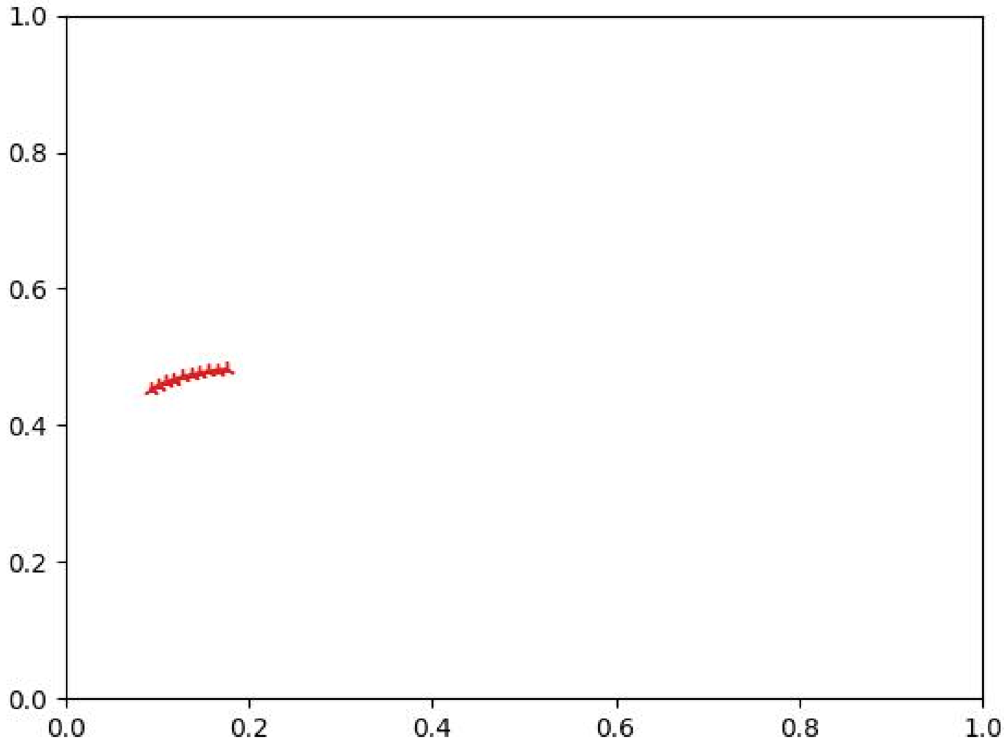
- **num** (*int*) – number of points in the path
- **degree** (*int*) – degree of curvieness of the path, default=1
- **dimension** (*int*) – number of spatial dimensions, default=2
- **mode** (*str*) – can be boid, walk, or bezier
- **rng** (*RandomState* | *None* | *int*) – seed, default=None

References

https://github.com/dhermes/bezier

Example

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



Example

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

(continues on next page)

(continued from previous page)

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

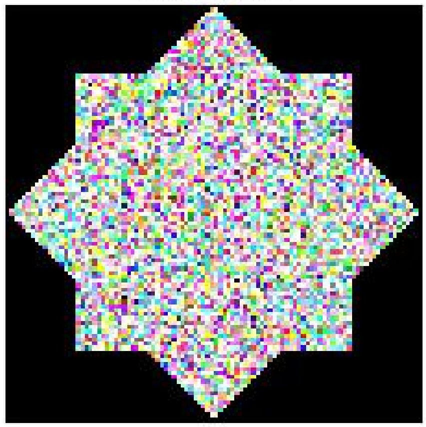
2.1.1.3.1.6 kwcoco.demo.toypatterns module

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

Bases: object

Example

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



Parameters
categories (*List[Dict] | None*) – List of coco category dictionaries

_default_categories = [{'id': 0, 'keypoints': [], 'name': 'background'}, {'id': 1, 'keypoints': [], 'name': 'box', 'supercategory': 'vector'}, {'id': 2, 'keypoints': [], 'name': 'circle', 'supercategory': 'vector'}, {'id': 3, 'keypoints': [], 'name': 'star', 'supercategory': 'vector'}, {'id': 4, 'keypoints': [], 'name': 'octagon', 'supercategory': 'vector'}, {'id': 5, 'keypoints': [], 'name': 'diamond', 'supercategory': 'vector'}, {'id': 6, 'keypoints': ['left_eye', 'right_eye'], 'name': 'superstar', 'supercategory': 'raster'}, {'id': 7, 'keypoints': ['top_tip', 'mid_tip', 'bot_tip'], 'name': 'eff', 'supercategory': 'raster'}, {'id': 8, 'keypoints': [], 'name': 'raster', 'supercategory': 'raster'}, {'id': 9, 'keypoints': [], 'name': 'vector', 'supercategory': 'shape'}, {'id': 10, 'keypoints': [], 'name': 'shape'}]

_default_keypoint_categories = [{'id': 1, 'name': 'left_eye', 'reflection_id': 2}, {'id': 2, 'name': 'right_eye', 'reflection_id': 1}, {'id': 3, 'name': 'top_tip', 'reflection_id': None}, {'id': 4, 'name': 'mid_tip', 'reflection_id': None}, {'id': 5, 'name': 'bot_tip', 'reflection_id': None}]

_default_catnames = ['star', 'eff', 'superstar']

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

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

Example

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

index(*name*)

get(*index, default=NoParam*)

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

Example

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

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

Example

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

Example

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

_todo_refactor_geometric_info(*cname, xy_offset, dims*)

This function is used to populate kpts and sseg information in the autogenerated coco dataset before rendering. It is redundant with other functionality.

TODO: rectify with _from_elem

Example

```
>>> self = CategoryPatterns.coerce(['superstar'])
>>> dims = (64, 64)
>>> cname = 'superstar'
>>> xy_offset = None
>>> self._todo_refactor_geometric_info(cname, xy_offset, dims)
```

Example

```
>>> from kwcoco.demo.toypatterns import *  # NOQA
>>> cname = 'star'
>>> xy_offset = None
>>> self = CategoryPatterns.coerce([cname])
>>> for d in range(0, 5):
...     dims = (d, d)
...     info = self._todo_refactor_geometric_info(cname, xy_offset, dims)
...     print(info['segmentation'].data)
```

_package_info(*cname*, *data*, *mask*, *kpts*, *xy_offset*, *dims*, *newstyle*)

packages data from _from_elem into coco-like annotation

_from_elem(*cname*, *chip*, *size=None*)

Example

```
>>> # hack to allow chip to be None
>>> chip = None
>>> size = (32, 32)
>>> cname = 'superstar'
>>> self = CategoryPatterns.coerce()
>>> self._from_elem(cname, chip, size)
```

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

Generates a star shaped structuring element.

Much faster than skimage.morphology version

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

Bases: object

static superstar()

test data patch

static eff()

test data patch

2.1.1.3.2 Module contents

2.1.1.4 kwcoco.examples package

2.1.1.4.1 Submodules

2.1.1.4.1.1 kwcoco.examples.bench_large_hyperspectral module

2.1.1.4.1.2 kwcoco.examples.demo_kwcoco_spaces module

2.1.1.4.1.3 kwcoco.examples.demo_sql_and_zip_files module

2.1.1.4.1.4 kwcoco.examples.draw_gt_and_predicted_boxes module

2.1.1.4.1.5 kwcoco.examples.faq module

2.1.1.4.1.6 kwcoco.examples.getting_started_existing_dataset module

2.1.1.4.1.7 kwcoco.examples.loading_multispectral_data module

2.1.1.4.1.8 kwcoco.examples.modification_example module

2.1.1.4.1.9 kwcoco.examples.shifting_annots module

2.1.1.4.1.10 kwcoco.examples.simple_kwcoco_torch_dataset module

2.1.1.4.1.11 kwcoco.examples.vectorized_interface module

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)

---

kwcoco.metrics.assignment.**_assign_confusion_vectors**(*true_dets*, *pred_dets*, *bg_weight=1.0*, *iou_thresh=0.5*, *bg_cidx=-1*, *bias=0.0*, *classes=None*, *compat='all'*, *prioritize='iou'*, *ignore_classes='ignore'*, *max_dets=None*)

Create confusion vectors for detections by assigning to ground true boxes

Given predictions and truth for an image return (y_pred, y_true, y_score), which is suitable for sklearn classification metrics

Parameters

- **true_dets** (*Detections*) – groundtruth with boxes, classes, and weights
- **pred_dets** (*Detections*) – predictions with boxes, classes, and scores
- **iou_thresh** (*float, default=0.5*) – bounding box overlap iou threshold required for assignment
- **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.
- **bg_cidx** (*int, default=-1*) – The index of the background class. The index used in the truth column when a predicted bounding box does not match any true bounding box.
- **classes** (*List[str] | kwcoco.CategoryTree*) – mapping from class indices to class names. Can also contain class heirarchy information.
- **ignore_classes** (*str | List[str]*) – class name(s) indicating ignore regions
- **max_dets** (*int*) – maximum number of detections to consider

---

Todo:

- [ ] This is a bottleneck function. An implementation in C / C++ /

Cython would likely improve the overall system.

- **[ ] Implement crowd truth. Allow multiple predictions to match any**
  truth objet marked as "iscrowd".

Returns

with relevant confusion vectors. This keys of this dict can be
interpreted as columns of a data frame. The *txs* / *pxs* columns represent the indexes of the true / predicted annotations that were assigned as matching. Additionally each row also contains the true and predicted class index, the predicted score, the true weight and the iou of the true and predicted boxes. A *txs* value of -1 means that the predicted box was not assigned to a true annotation and a *pxs* value of -1 means that the true annotation was not assigne to any predicted annotation.

Return type
dict

Example

```
>>> # xdoctest: +REQUIRES(module:pandas)
>>> import pandas as pd
>>> import kwimage
>>> # Given a raw numpy representation construct Detection wrappers
>>> true_dets = kwimage.Detections(
>>>     boxes=kwimage.Boxes(np.array([
>>>         [ 0,  0, 10, 10], [10,  0, 20, 10],
>>>         [10,  0, 20, 10], [20,  0, 30, 10]]), 'tlbr'),
>>>     weights=np.array([1, 0, .9, 1]),
>>>     class_idxs=np.array([0, 0, 1, 2]))
>>> pred_dets = kwimage.Detections(
>>>     boxes=kwimage.Boxes(np.array([
>>>         [6, 2, 20, 10], [3,  2, 9, 7],
>>>         [3,  9, 9, 7],  [3,  2, 9, 7],
>>>         [2,  6, 7, 7],  [20,  0, 30, 10]]), 'tlbr'),
>>>     scores=np.array([.5, .5, .5, .5, .5, .5]),
>>>     class_idxs=np.array([0, 0, 1, 2, 0, 1]))
>>> bg_weight = 1.0
>>> compat = 'all'
>>> iou_thresh = 0.5
>>> bias = 0.0
>>> import kwcoco
>>> classes = kwcoco.CategoryTree.from_mutex(list(range(3)))
>>> bg_cidx = -1
>>> y = _assign_confusion_vectors(true_dets, pred_dets, bias=bias,
>>>                               bg_weight=bg_weight, iou_thresh=iou_thresh,
>>>                               compat=compat)
>>> y = pd.DataFrame(y)
>>> print(y)  # xdoc: +IGNORE_WANT
   pred  true  score  weight     iou  txs  pxs
0     1     2 0.5000  1.0000  1.0000    3    5
1     0    -1 0.5000  1.0000 -1.0000   -1    4
2     2    -1 0.5000  1.0000 -1.0000   -1    3
3     1    -1 0.5000  1.0000 -1.0000   -1    2
4     0    -1 0.5000  1.0000 -1.0000   -1    1
5     0     0 0.5000  0.0000  0.6061    1    0
6    -1     0 0.0000  1.0000 -1.0000    0   -1
7    -1     1 0.0000  0.9000 -1.0000    2   -1
```

Example

```
>>> # xdoctest: +REQUIRES(module:pandas)
>>> from kwcoco.metrics.assignment import _assign_confusion_vectors
>>> import pandas as pd
>>> import ubelt as ub
>>> from kwcoco.metrics import DetectionMetrics
>>> dmet = DetectionMetrics.demo(nimgs=1, nclasses=8,
>>>                              nboxes=(0, 20), n_fp=20,
>>>                              box_noise=.2, cls_noise=.3)
>>> classes = dmet.classes
>>> gid = ub.peek(dmet.gid_to_pred_dets)
>>> true_dets = dmet.true_detections(gid)
>>> pred_dets = dmet.pred_detections(gid)
>>> y = _assign_confusion_vectors(true_dets, pred_dets,
>>>                               classes=dmet.classes,
>>>                               compat='all', prioritize='class')
>>> y = pd.DataFrame(y)
>>> print(y)  # xdoc: +IGNORE_WANT
>>> y = _assign_confusion_vectors(true_dets, pred_dets,
>>>                               classes=dmet.classes,
>>>                               compat='ancestors', iou_thresh=.5)
>>> y = pd.DataFrame(y)
>>> print(y)  # xdoc: +IGNORE_WANT
```

kwcoco.metrics.assignment.**_critical_loop**(*true_dets*, *pred_dets*, *iou_lookup*, *isvalid_lookup*, *cx_to_matchable_txs*, *bg_weight*, *prioritize*, *iou_thresh_*, *pdist_priority*, *cx_to_ancestors*, *bg_cidx*, *ignore_classes*, *max_dets*)

kwcoco.metrics.assignment.**_fast_pdist_priority**(*classes*, *prioritize*, *_cache={}*)

Custom priority computation. Needs some vetting.

This is the priority used when deciding which prediction to assign to which truth.

Todo:

- **[ ] Look at absolute difference in sibling entropy when deciding**
  whether to go up or down in the tree.

---

kwcoco.metrics.assignment.**_filter_ignore_regions**(*true_dets*, *pred_dets*, *ioaa_thresh=0.5*, *ignore_classes='ignore'*)

Determine which true and predicted detections should be ignored.

Parameters

- **true_dets** (*Detections*)
- **pred_dets** (*Detections*)
- **ioaa_thresh** (*float*) – intersection over other area thresh for ignoring a region.

Returns

flags indicating which true and predicted
detections should be ignored.

Return type
Tuple[ndarray, ndarray]

Example

```
>>> from kwcoco.metrics.assignment import *  # NOQA
>>> from kwcoco.metrics.assignment import _filter_ignore_regions
>>> import kwimage
>>> pred_dets = kwimage.Detections.random(classes=['a', 'b', 'c'])
>>> true_dets = kwimage.Detections.random(
>>>     segmentations=True, classes=['a', 'b', 'c', 'ignore'])
>>> ignore_classes = {'ignore', 'b'}
>>> ioaa_thresh = 0.5
>>> print('true_dets = {!r}'.format(true_dets))
>>> print('pred_dets = {!r}'.format(pred_dets))
>>> flags1, flags2 = _filter_ignore_regions(
>>>     true_dets, pred_dets, ioaa_thresh=ioaa_thresh, ignore_classes=ignore_
↪classes)
>>> print('flags1 = {!r}'.format(flags1))
>>> print('flags2 = {!r}'.format(flags2))
```

```
>>> flags3, flags4 = _filter_ignore_regions(
>>>     true_dets, pred_dets, ioaa_thresh=ioaa_thresh,
>>>     ignore_classes={c.upper() for c in ignore_classes})
>>> assert np.all(flags1 == flags3)
>>> assert np.all(flags2 == flags4)
```

2.1.1.5.1.2 kwcoco.metrics.clf_report module

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

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

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

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

References

Parameters

- **y_true** (*ndarray*) – true labels for each item
- **y_pred** (*ndarray*) – predicted labels for each item
- **target_names** (*List | None*) – mapping from label to category name
- **sample_weight** (*ndarray | None*) – weight for each item
- **verbose** (*int*) – print if True
- **log** (*callable | None*) – print or logging function
- **remove_unsupported** (*bool*) – removes categories that have no support. Defaults to False.
- **ascii_only** (*bool*) – if True dont use unicode characters. if the environ ASCII_ONLY is present this is forced to True and cannot be undone. Defaults to False.

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
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
```

(continues on next page)

(continued from previous page)

```
>>> sample_weight = None
>>> logs = []
>>> report = classification_report(y_true, y_pred, verbose=1, ascii_only=True,␣
↪log=logs.append)
>>> print('\n'.join(logs))
```

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

One-vs-rest classification report

Parameters

- **mc_y_true** (*ndarray*) – 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] | None*) – mapping from int label to string name
- **sample_weight** (*ndarray | None*) – weight for each item. Shape [N].
- **metrics** (*List[str] | None*) – 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

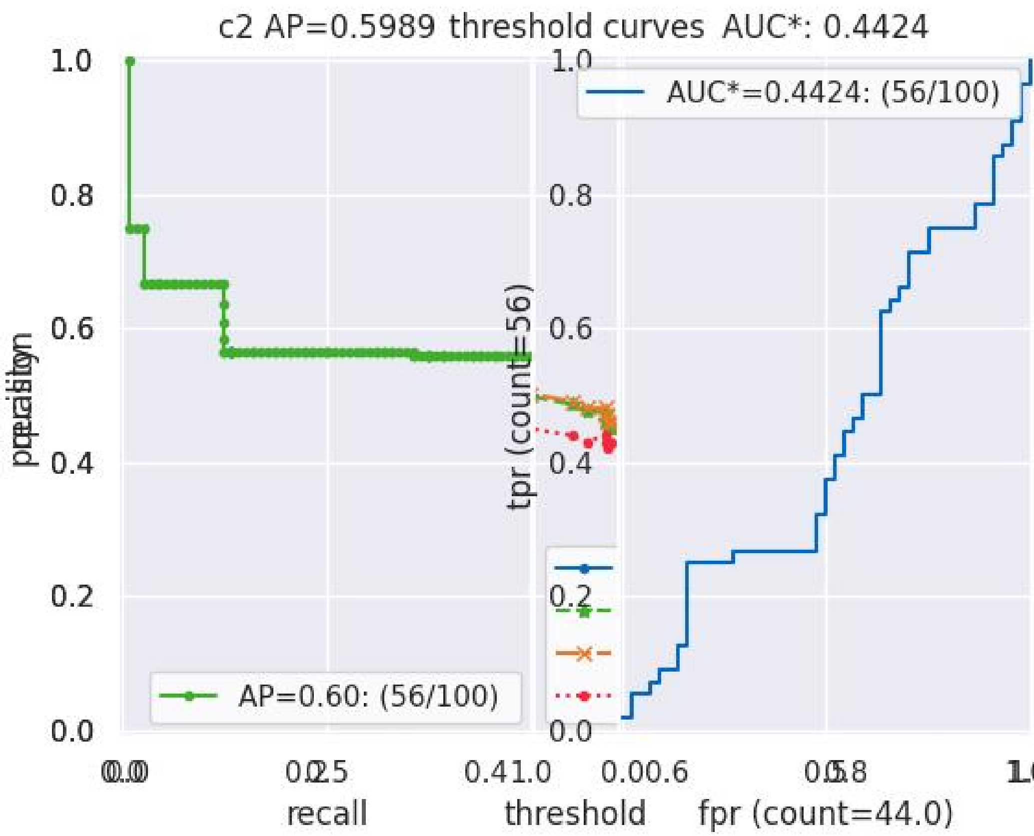
class kwcoco.metrics.confusion_measures.**Measures**(*info*)

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)
```

(continues on next page)

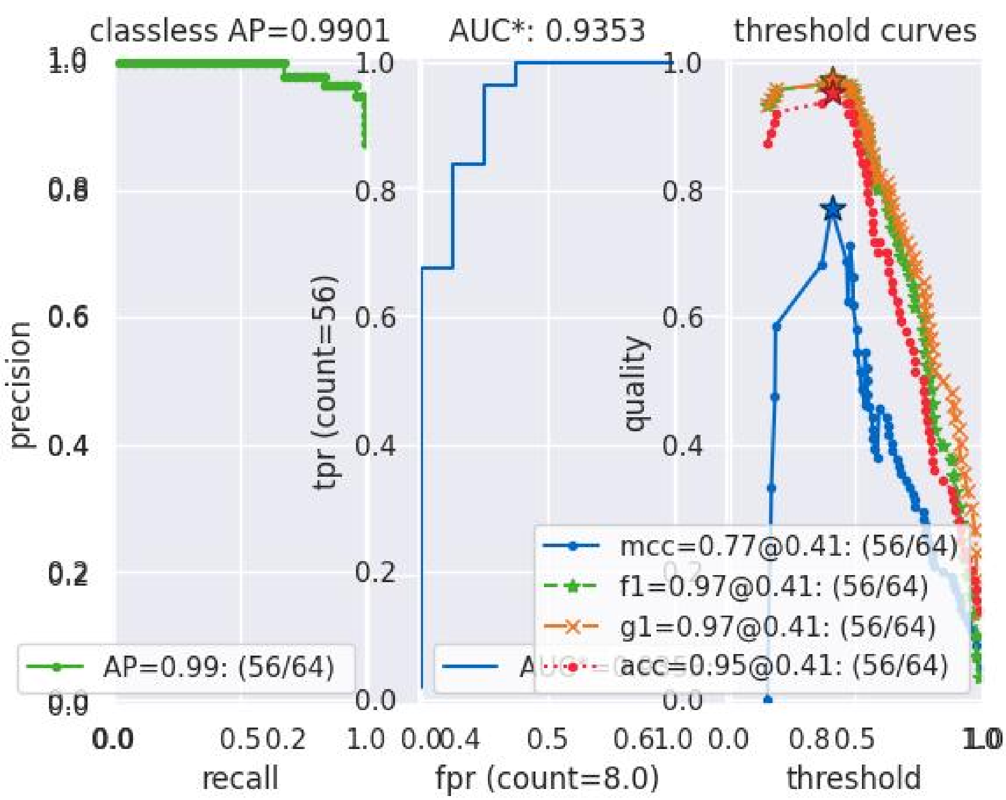
(continued from previous page)

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

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

Example

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



classmethod demo(***kwargs*)

Create a demo Measures object for testing / demos

Parameters

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

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

Combine binary confusion metrics

Parameters

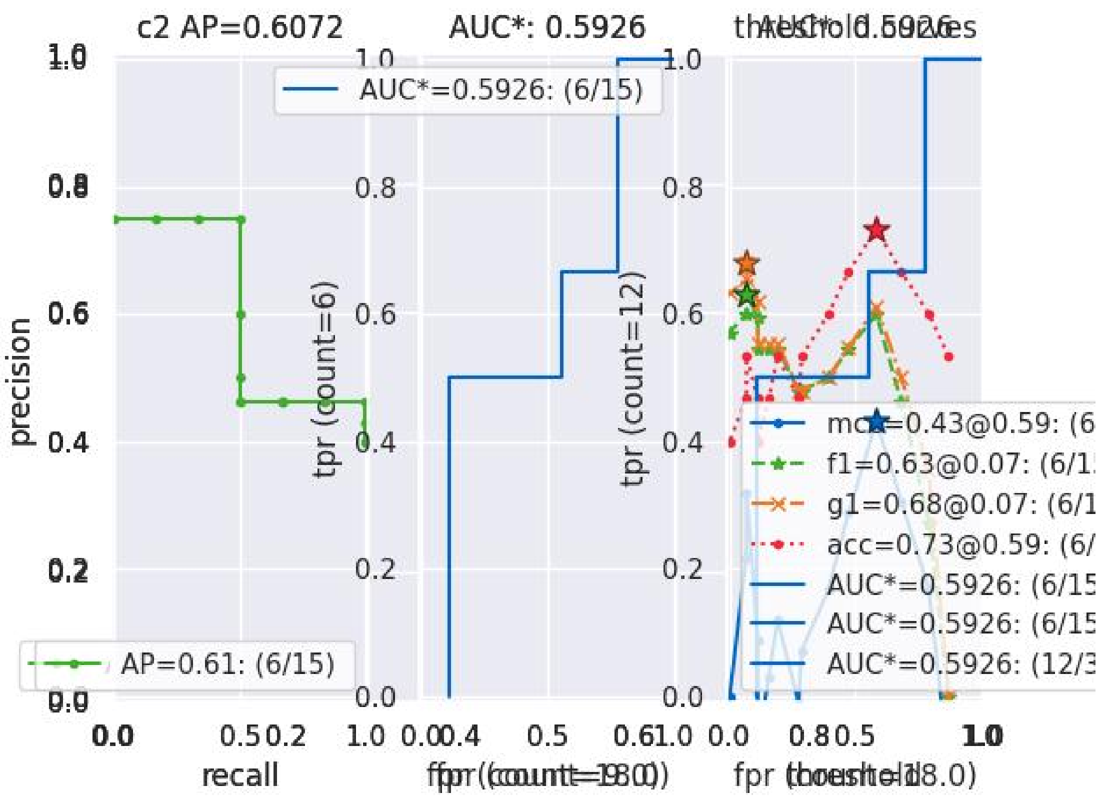
- **tocombine** (*List[Measures]*) – a list of measures to combine into one
- **precision** (*int* | *None*) – If specified rounds thresholds to this precision which can prevent a RAM explosion when combining a large number of measures. However, this is a lossy operation and will impact the underlying scores. NOTE: use `growth` instead.
- **growth** (*int* | *None*) – if specified this limits how much the resulting measures are allowed to grow by. If None, growth is unlimited. Otherwise, if growth is 'max', the growth is limited to the maximum length of an input. We might make this more numerical in the future.
- **thresh_bins** (*int* | *None*) – 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.urepr(measures1.__json__(), nl=1, sort=0))
>>> print(ub.urepr(measures2.__json__(), nl=1, sort=0))
>>> print(ub.urepr(new_measures.__json__(), nl=1, sort=0))
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.figure(fnum=1)
>>> new_measures.summary_plot()
>>> measures1.summary_plot()
>>> measures1.draw('roc')
>>> measures2.draw('roc')
>>> new_measures.draw('roc')
```



Example

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

(continues on next page)

(continued from previous page)

```
>>>     measures_n = Measures(base_n).reconstruct()
>>>     tocombine.append(measures_n)
>>> for precision in [6, 5, 2]:
>>>     combo = Measures.combine(tocombine, precision=precision).reconstruct()
>>>     print('precision = {!r}'.format(precision))
>>>     print('combo = {}'.format(ub.urepr(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.urepr(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.urepr(tocombine, nl=2)))
>>> print('thresh_bins = {!r}'.format(thresh_bins))
```

(continues on next page)

(continued from previous page)

```
>>> print(ub.urepr(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.urepr(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.urepr(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.urepr(combo, nl=1)))
>>>     print('num_thresholds = {}'.format(len(combo['thresholds'])))
```

kwcoco.metrics.confusion_measures.**_combine_threshold**(*tocombine_thresh*, *thresh_bins*, *growth*, *precision*)

Logic to take care of combining thresholds in the case bins are not given

This can be fairly slow and lead to unnecessary memory usage

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

Does a reversable array difference operation.

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

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

Bases: NiceRepr, *DictProxy*

summary()

classmethod **from_json**(*state*)

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

Example

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

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

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

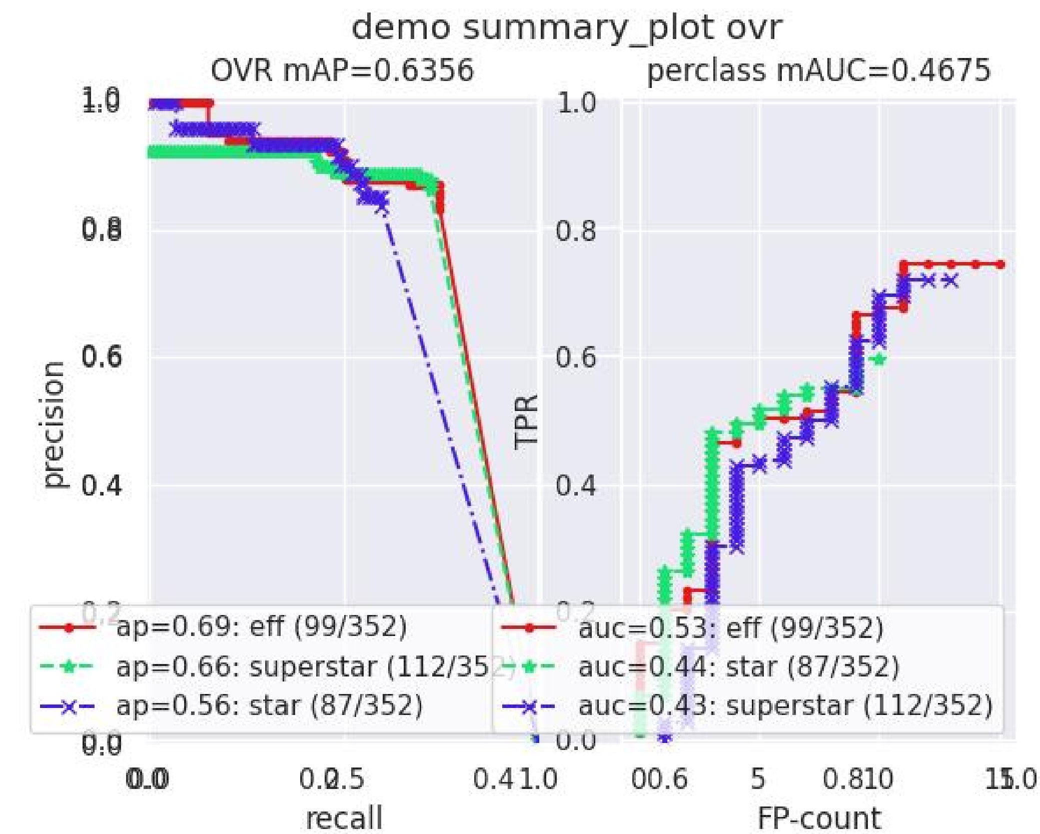
summary_plot(*fnum=1*, *title=''*, *subplots='auto'*)

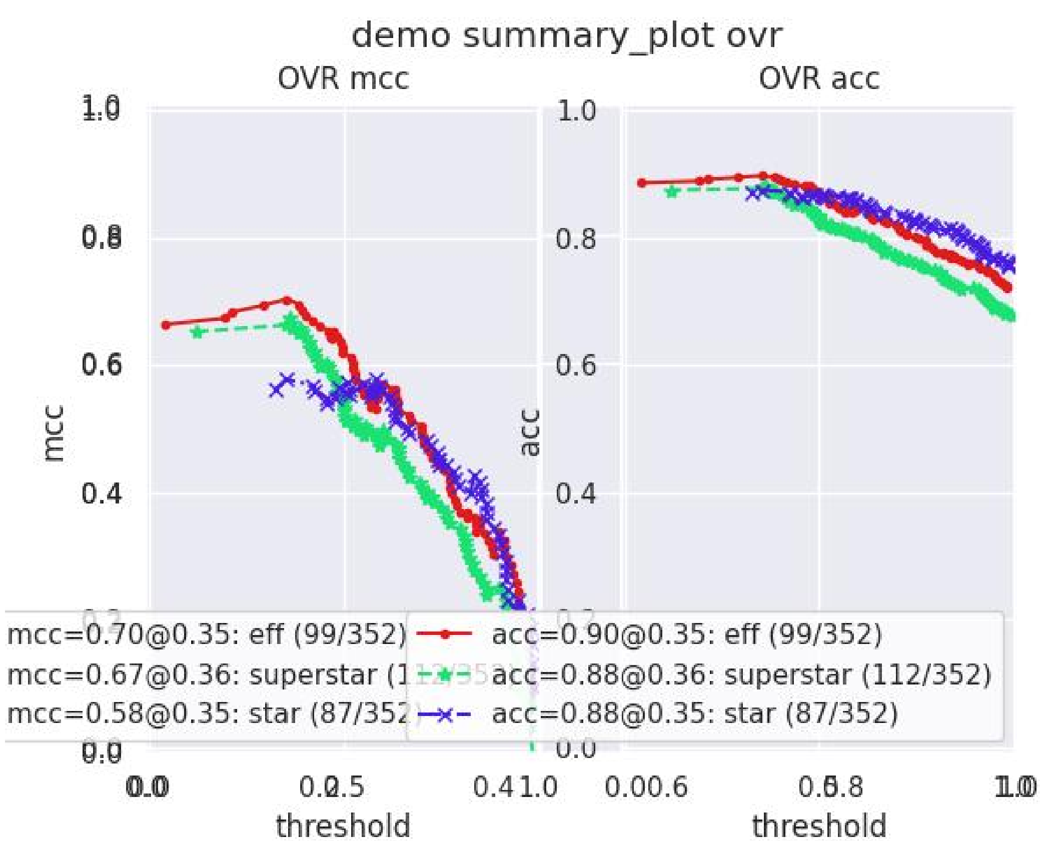
CommandLine

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

Example

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


demo summary_plot ovr
OVR mAP=0.6356
perclass mAUC=0.4675
precision
recall
TPR
FP-count
ap=0.69: eff (99/352)
ap=0.66: superstar (112/352)
ap=0.56: star (87/352)
auc=0.53: eff (99/352)
auc=0.44: star (87/352)
auc=0.43: superstar (112/352)


demo summary_plot ovr
OVR mcc
OVR acc
mcc
acc
threshold
threshold
mcc=0.70@0.35: eff (99/352)
mcc=0.67@0.36: superstar (112/352)
mcc=0.58@0.35: star (87/352)
acc=0.90@0.35: eff (99/352)
acc=0.88@0.36: superstar (112/352)
acc=0.88@0.35: star (87/352)

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*)

_summary()

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…

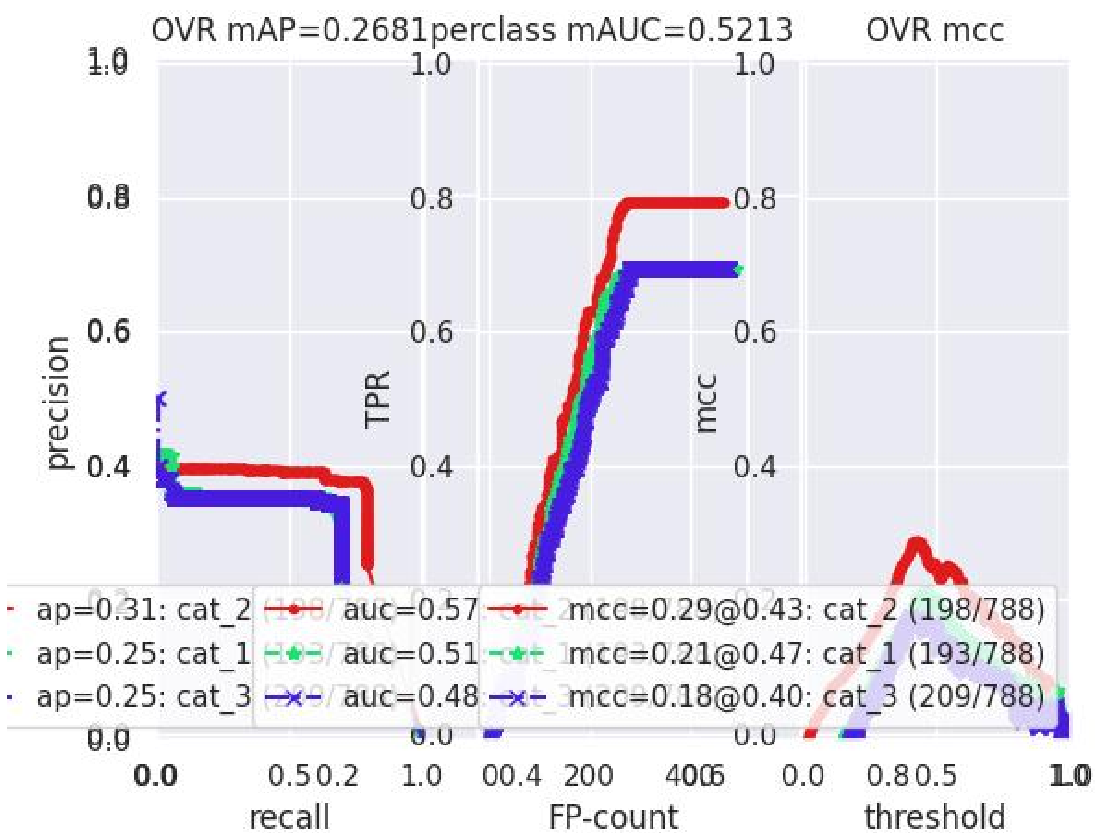
Variables

- **data** (*kwarray.DataFrameArray*) – should at least have keys true, pred, weight
- **classes** (*Sequence | CategoryTree*) – list of category names or category graph
- **probs** (*ndarray | None*) – probabilities for each class

Example

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

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



classmethod from_json(*state*)

classmethod demo(***kw*)

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

Returns
ConfusionVectors

Example

```
>>> from kwcoco.metrics.confusion_vectors import *  # NOQA
>>> 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, newstyle=False)
>>> 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] | None*) – 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*) – if fewer than this many data points inserts dummy stabilization data so curves can still be drawn. Default to 7.
- **fp_cutoff** (*int | None*) – maximum number of false positives in the truncated roc curves. The default None is equivalent to float('inf')
- **monotonic_ppv** (*bool*) – 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. Default to True.

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.urepr(self.measures())))
```

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

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

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

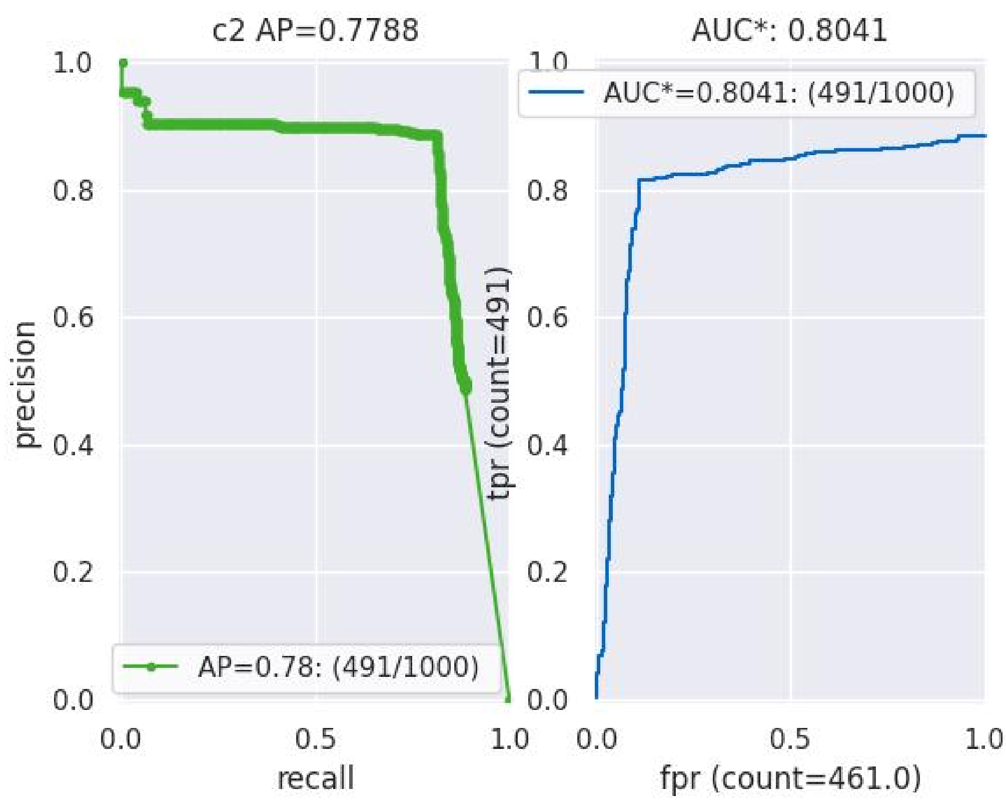
- **n** (*int*) – number of rows
- **p_true** (*float*) – fraction of real positive cases
- **p_error** (*float*) – probability of making a recoverable mistake
- **p_miss** (*float*) – probability of making a unrecoverable mistake
- **rng** (*int* | *RandomState* | *None*) – random seed / state

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.urepr(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* | *None*) – maximum number of false positives in the truncated roc curves. The default of `None` is equivalent to `float('inf')`
- **monotonic_ppv** (*bool*) – 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.urepr(self.measures())))
>>> self = BinaryConfusionVectors.demo(n=1, p_true=0.5, p_error=0.5)
>>> print('measures = {}'.format(ub.urepr(self.measures())))
>>> self = BinaryConfusionVectors.demo(n=3, p_true=0.5, p_error=0.5)
>>> print('measures = {}'.format(ub.urepr(self.measures())))
```

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

_binary_clf_curves(*stabalize_thresh=7*, *fp_cutoff=None*)

Compute TP, FP, TN, and FN counts for this binary confusion vector.

Code common to ROC, PR, and threshold measures, computes the elements of the binary confusion matrix at all relevant operating point thresholds.

Parameters

- **stabalize_thresh** (*int*) – if fewer than this many data points insert stabalization data.
- **fp_cutoff** (*int* | *None*) – maximum number of false positives

Example

```
>>> from kwcoco.metrics.confusion_vectors import *  # NOQA
>>> self = BinaryConfusionVectors.demo(n=1, p_true=0.5, p_error=0.5)
>>> self._binary_clf_curves()
```

```
>>> self = BinaryConfusionVectors.demo(n=0, p_true=0.5, p_error=0.5)
>>> self._binary_clf_curves()
```

```
>>> self = BinaryConfusionVectors.demo(n=100, p_true=0.5, p_error=0.5)
>>> self._binary_clf_curves()
```

draw_distribution()

_3dplot()

Example

```
>>> # xdoctest: +REQUIRES(module:kwplot)
>>> # xdoctest: +REQUIRES(module:pandas)
>>> from kwcoco.metrics.confusion_vectors import *  # NOQA
>>> from kwcoco.metrics.detect_metrics import DetectionMetrics
>>> dmet = DetectionMetrics.demo(
>>>     n_fp=(0, 1), n_fn=(0, 2), nimgs=256, nboxes=(0, 10),
>>>     bbox_noise=10,
>>>     classes=1)
>>> cfsn_vecs = dmet.confusion_vectors()
>>> self = bin_cfsn = cfsn_vecs.binarize_classless()
>>> #dmet.summarize(plot=True)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.figure(fnum=3)
>>> self._3dplot()
```

kwcoco.metrics.confusion_vectors.**_stabalize_data**(*y_true*, *y_score*, *sample_weight*, *npad=7*)

Adds ideally calibrated dummy values to curves with few positive examples. This acts somewhat like a Bayesian prior and smooths out the curve.

Example

y_score = np.array([0.5, 0.6]) y_true = np.array([1, 1]) sample_weight = np.array([1, 1]) npad = 7 _stabalize_data(y_true, y_score, sample_weight, npad=npad)

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[int, kwimage.Detections]*) – maps image ids to truth
- **gid_to_pred_dets** (*Dict[int, kwimage.Detections]*) – maps image ids to predictions
- **classes** (kwcoco.CategoryTree | *None*) – the categories to be scored, if unspecified attempts to determine these from the truth detections

Example

```
>>> # Demo how to use detection metrics directly given detections only
>>> # (no kwcoco file required)
>>> from kwcoco.metrics import detect_metrics
>>> import kwimage
>>> # Setup random true detections (these are just boxes and scores)
>>> true_dets = kwimage.Detections.random(3)
>>> # Peek at the simple internals of a detections object
>>> print('true_dets.data = {}'.format(ub.urepr(true_dets.data, nl=1)))
>>> # Create similar but different predictions
>>> true_subset = true_dets.take([1, 2]).warp(kwimage.Affine.coerce({'scale': 1.1}))
>>> false_positive = kwimage.Detections.random(3)
>>> pred_dets = kwimage.Detections.concatenate([true_subset, false_positive])
>>> dmet = DetectionMetrics()
>>> dmet.add_predictions(pred_dets, imgname='image1')
>>> dmet.add_truth(true_dets, imgname='image1')
>>> # Raw confusion vectors
>>> cfsn_vecs = dmet.confusion_vectors()
>>> print(cfsn_vecs.data.pandas().to_string())
>>> # Our scoring definition (derived from confusion vectors)
>>> print(dmet.score_kwcoco())
>>> # VOC scoring
>>> print(dmet.score_voc(bias=0))
>>> # Original pycocotools scoring
>>> # xdoctest: +REQUIRES(module:pycocotools)
>>> print(dmet.score_pycocotools())
```

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()

enrich_confusion_vectors(*cfsn_vecs*)

Adds annotation id information into confusion vectors computed via this detection metrics object.

TODO: should likely use this at the end of the function that builds the confusion vectors.

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*) – coco dataset with ground truth
- **pred_coco** (*kwcoco.CocoDataset*) – coco dataset with predictions

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()
```

_register_imagename(*imgname*, *gid=None*)

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

Register/Add predicted detections for an image

Parameters

- **pred_dets** (*kwimage.Detections*) – predicted detections
- **imgname** (*str* | *None*) – 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* | *None*) – 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

property classes

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

- **prioritize** (*str*) – 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. Default to 'iou'
- **ignore_classes** (*set | str*) – class names indicating ignore regions. Default={'ignore'}
- **background_class** (*str | NoParamType*) – 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*) – verbosity flag. Default to 'auto'. In auto mode, verbose=1 if len(gids) > 1000.
- **workers** (*int*) – number of parallel assignment processes. Defaults to 0
- **track_probs** (*str*) – can be 'try', 'force', or False. if truthy, we assume probabilities for multiple classes are available. default='try'

Returns
ConfusionVectors | Dict[float, 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...
```

_to_coco()

Convert to a coco representation of truth and predictions

with inverse aid mappings

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.urepr(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.urepr(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.

rng (int | None | RandomState): random seed / state

CommandLine

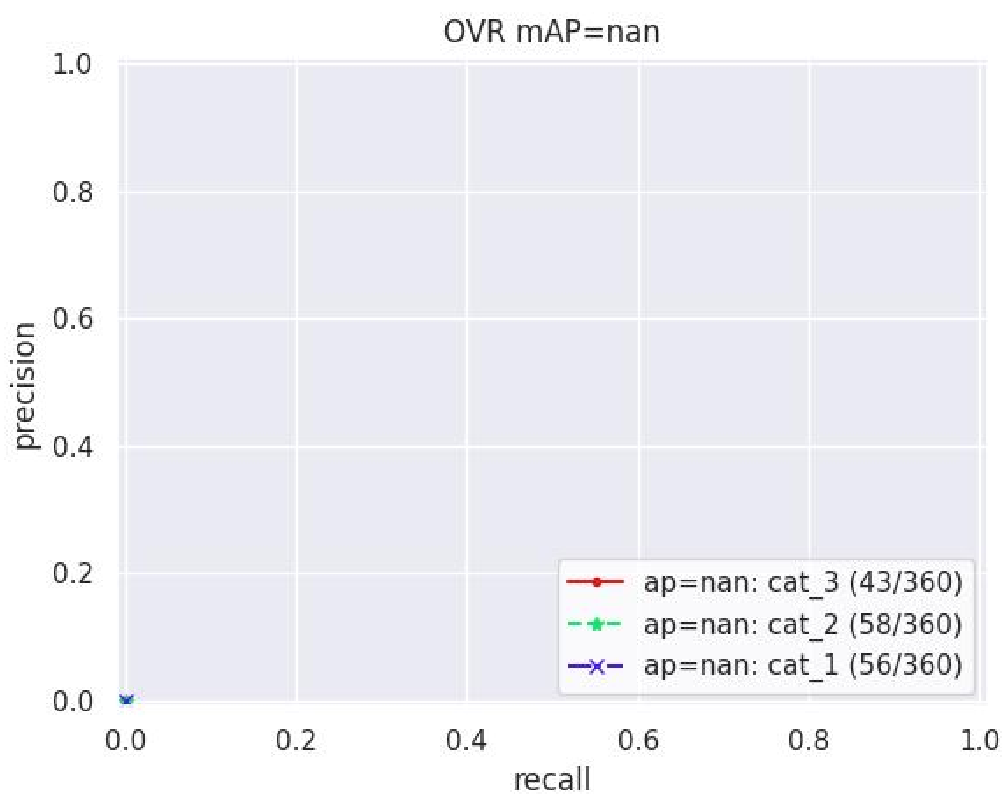
```
xdoctest -m kwcoco.metrics.detect_metrics DetectionMetrics.demo:2 --show
```

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'*)

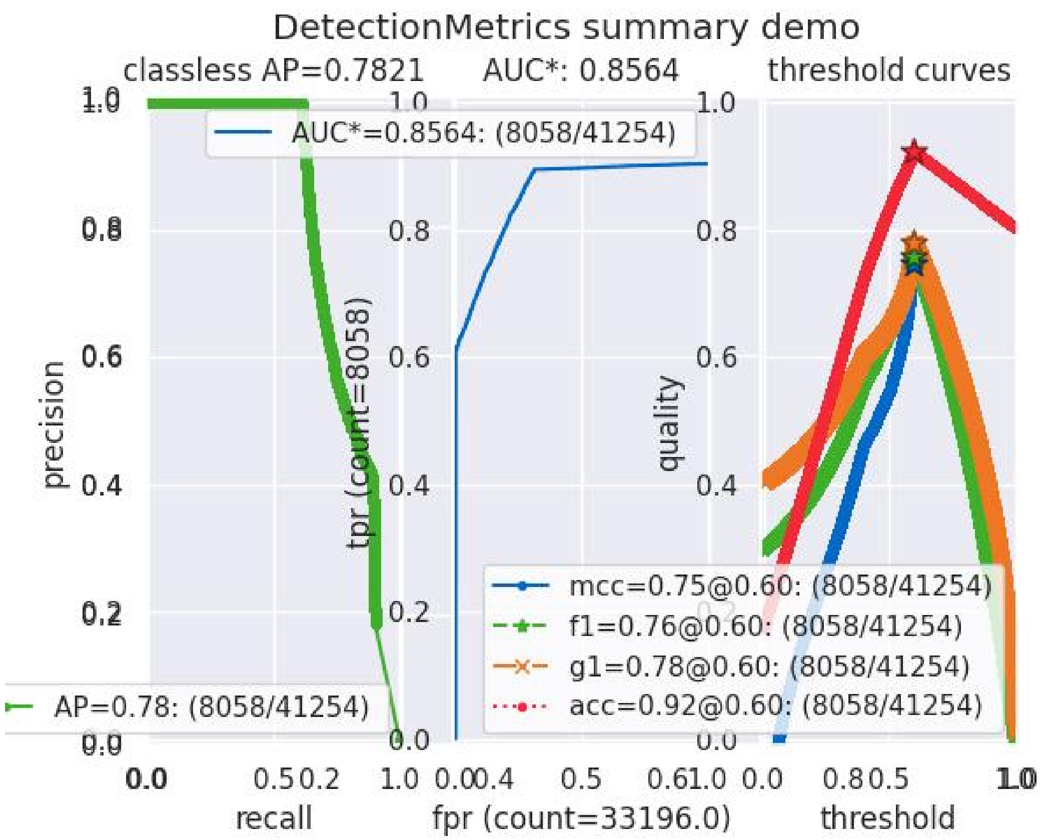
Create summary one-versus-rest and binary metrics.

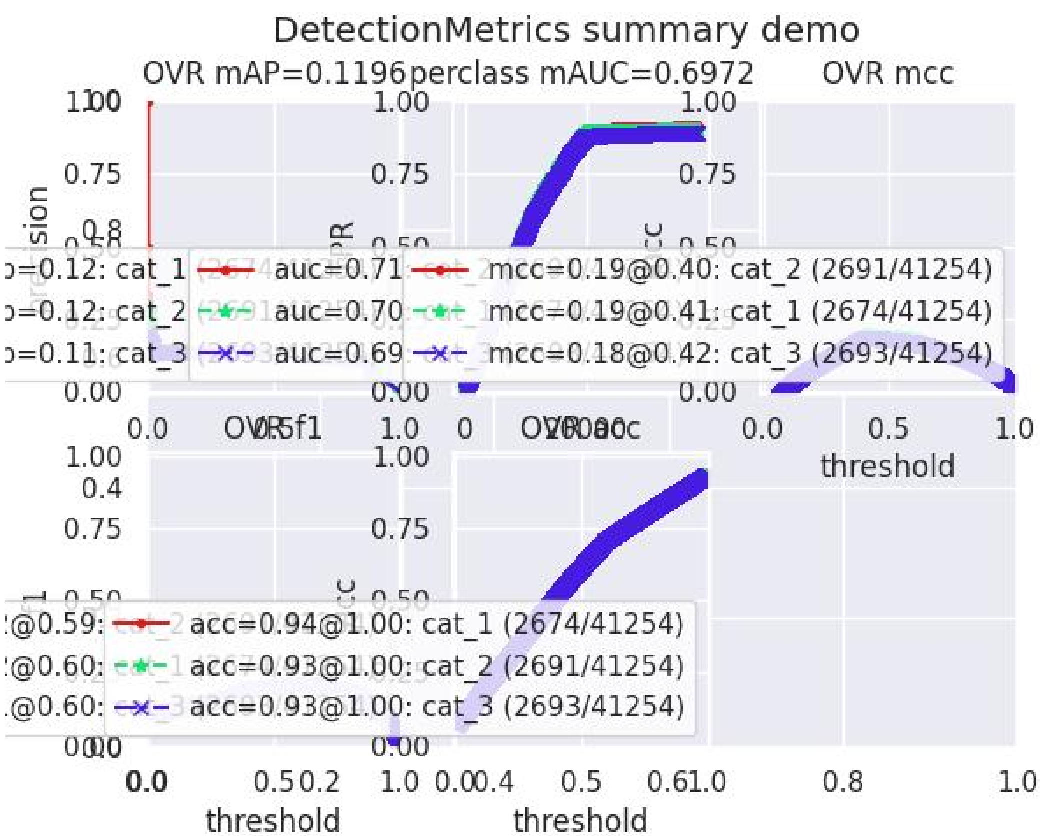
Parameters

- **out_dpath** (*pathlib.Path | None*) – FIXME: not hooked up
- **with_ovr** (*str | bool*) – include one-versus-rest metrics (wrt the classes). If 'auto' enables if possible. FIXME: auto is not working.
- **with_bin** (*str | bool*) – include binary classless metrics (i.e. detected or not). If 'auto' enables if possible. FIXME: auto is not working.
- **plot** (*bool*) – if true, also write plots. Defaults to False.
- **title** (*str*) – passed if plot is given

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()
```


DetectionMetrics summary demo
classless AP=0.7821
AUC*: 0.8564
threshold curves
AUC*=0.8564: (8058/41254)
precision
recall
AP=0.78: (8058/41254)
tpr (count=8058)
fpr (count=33196.0)
quality
threshold
mcc=0.75@0.60: (8058/41254)
f1=0.76@0.60: (8058/41254)
g1=0.78@0.60: (8058/41254)
acc=0.92@0.60: (8058/41254)


DetectionMetrics summary demo
OVR mAP=0.1196
perclass mAUC=0.6972
OVR mcc
auc=0.71
auc=0.70
auc=0.69
mcc=0.19@0.40: cat_2 (2691/41254)
mcc=0.19@0.41: cat_1 (2674/41254)
mcc=0.18@0.42: cat_3 (2693/41254)
threshold
acc=0.94@1.00: cat_1 (2674/41254)
acc=0.93@1.00: cat_2 (2691/41254)
acc=0.93@1.00: cat_3 (2693/41254)
threshold
threshold

kwcoco.metrics.detect_metrics.**_demo_construct_probs**(*pred_cxs*, *pred_scores*, *classes*, *rng*, *hacked=1*)
Constructs random probabilities for demo data

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.**_summarize**(*self*, *ap=1*, *iouThr=None*, *areaRngLbl='all'*, *maxDets=100*)

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(01.00000000000000000000000000043))
```

kwcoco.metrics.drawing.**_realpos_label_suffix**(*info*)

Creates a label suffix that indicates the number of real positive cases versus the total amount of cases considered for an evaluation curve.

Parameters
info (*Dict*) – with keys, nsuppert, realpos_total

Example

```
>>> from kwcoco.metrics.drawing import *  # NOQA
>>> info = {'nsupport': 10, 'realpos_total': 10}
>>> _realpos_label_suffix(info)
10/10
>>> info = {'nsupport': 10.0, 'realpos_total': 10.0}
>>> _realpos_label_suffix(info)
10/10
>>> info = {'nsupport': 10.3333, 'realpos_total': 10.22222}
>>> _realpos_label_suffix(info)
10.22/10.33
>>> info = {'nsupport': 10.000000001, 'realpos_total': None}
```

(continues on next page)

(continued from previous page)

```
>>> _realpos_label_suffix(info)
10
>>> info = {'nsupport': 10.009}
>>> _realpos_label_suffix(info)
10.01
>>> info = {'nsupport': 3331033334342.432, 'realpos_total': 1033334342.432}
>>> _realpos_label_suffix(info)
1e9/3.3e12
>>> info = {'nsupport': 0.007, 'realpos_total': 0.0000893}
>>> _realpos_label_suffix(info)
8.9e-5/0.007
```

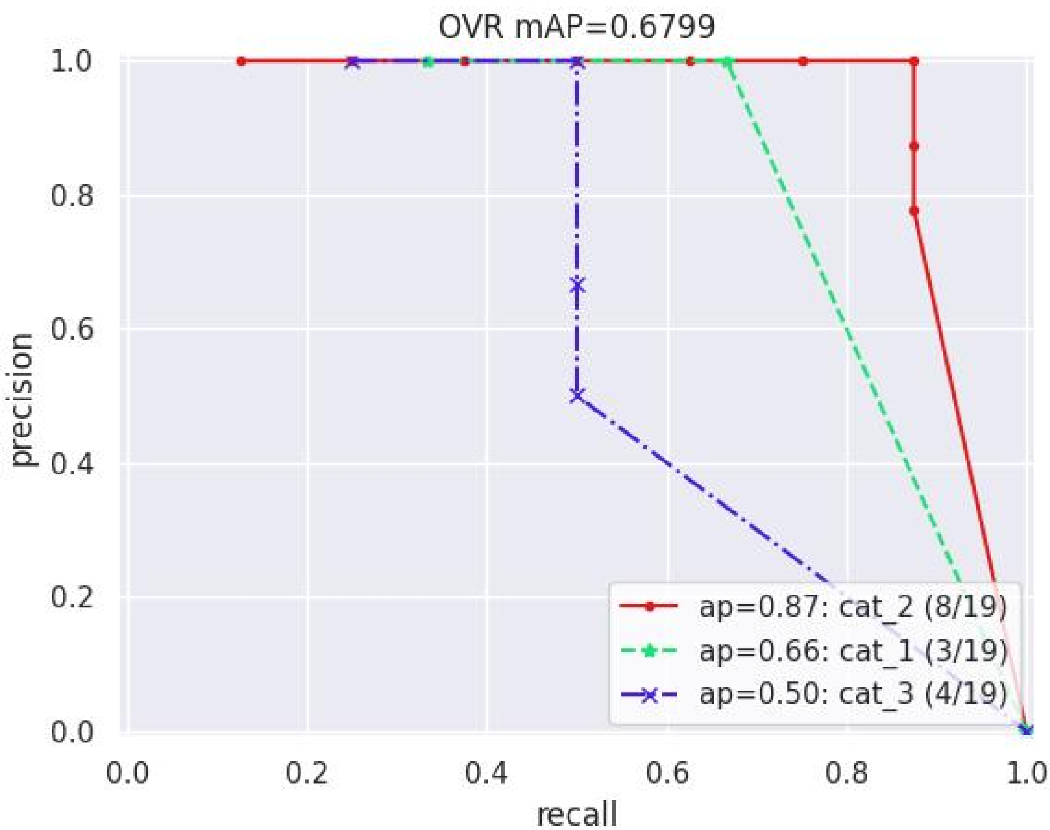
kwcoco.metrics.drawing.**draw_perclass_prcurve**(*cx_to_info*, *classes=None*, *prefix=''*, *fnum=1*, ***kw*)

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.urepr(cx_to_info, nl=1)))
>>> import kwplot
>>> kwplot.autompl()
>>> draw_perclass_prcurve(cx_to_info, classes)
>>> # xdoctest: +REQUIRES(--show)
>>> kwplot.show_if_requested()
```



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

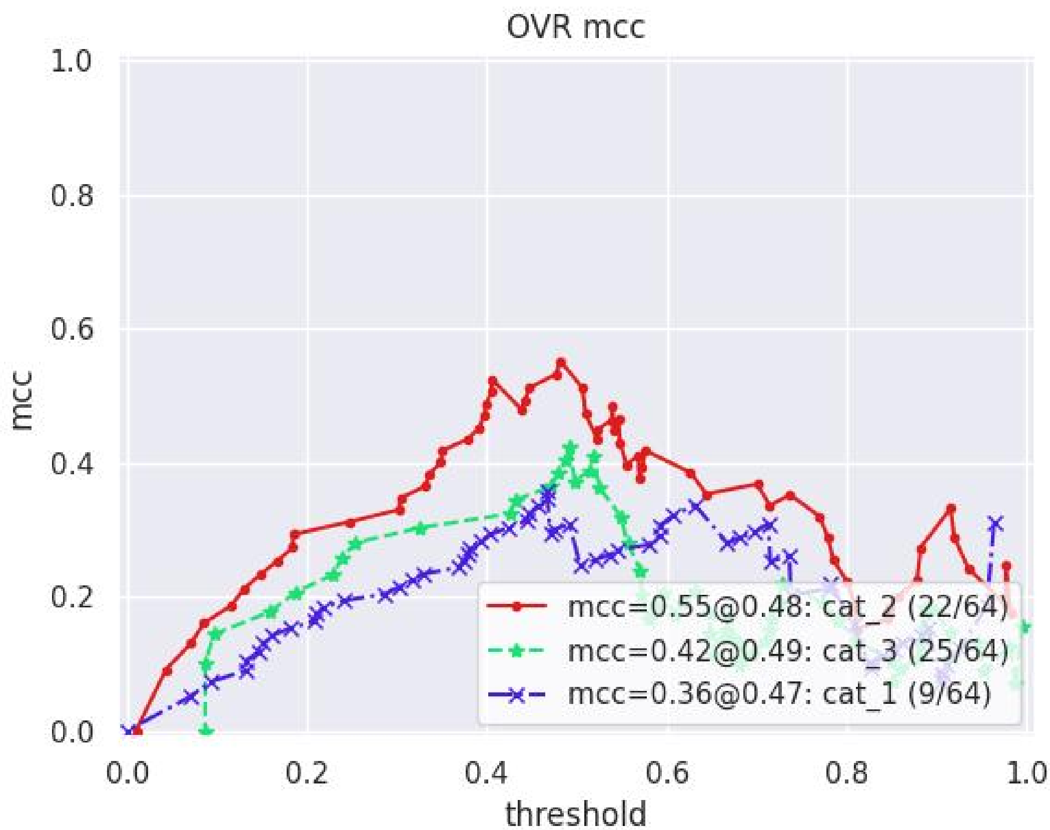
Parameters

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

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

Example

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



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

Parameters
info (*Measures* | *Dict*)

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

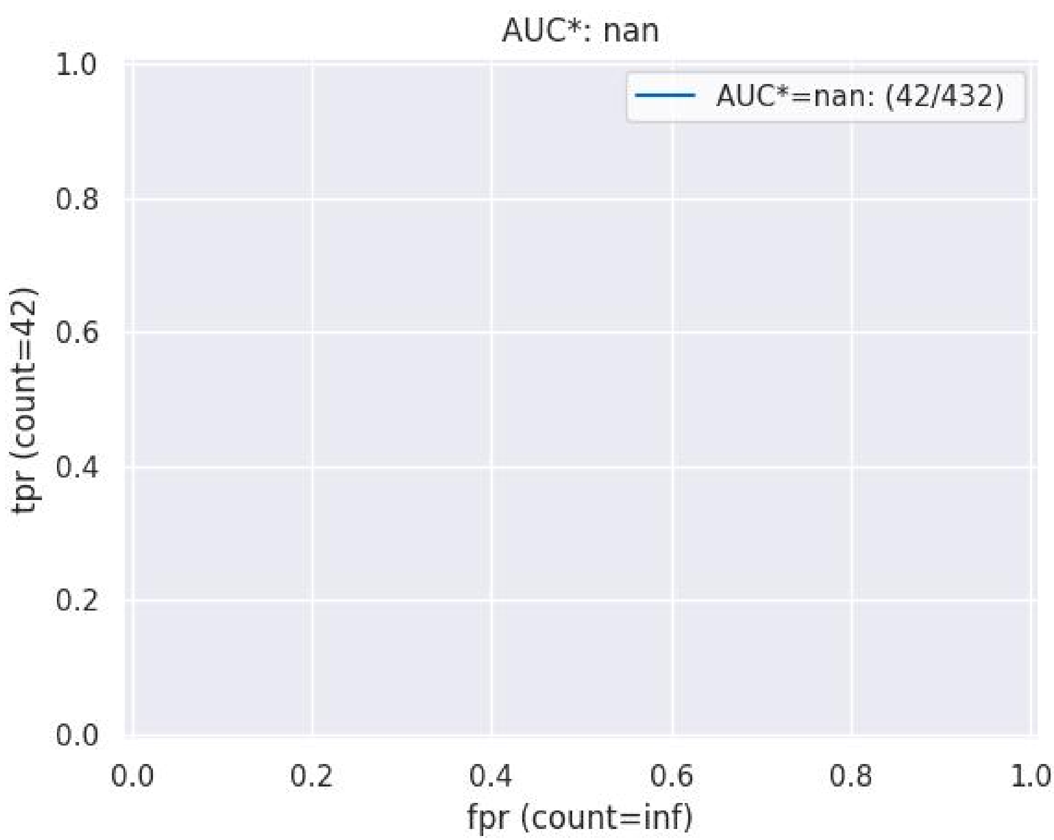
Example

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

(continues on next page)

(continued from previous page)

```
>>> 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

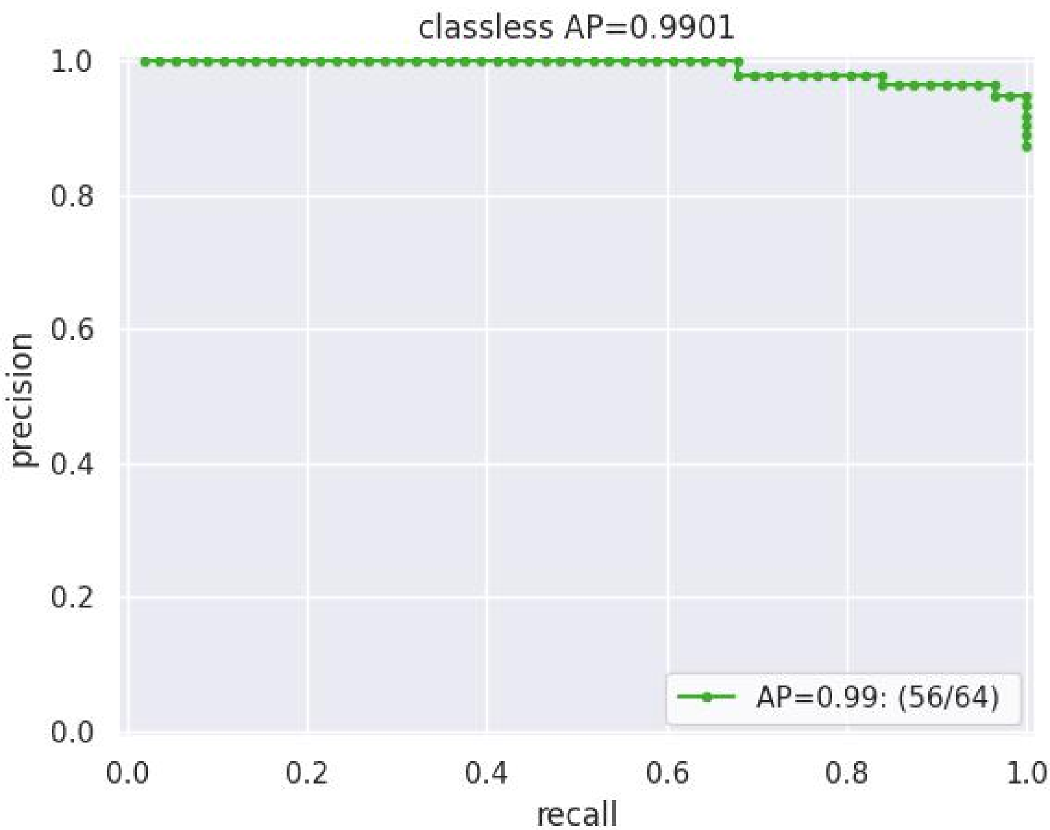
```
>>> # xdoctest: +REQUIRES(module:kwplot)
>>> from kwcoco.metrics import DetectionMetrics
>>> dmet = DetectionMetrics.demo(
>>>     nimgs=10, nboxes=(0, 10), n_fp=(0, 1), classes=3)
>>> cfsn_vecs = dmet.confusion_vectors()
```

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

(continues on next page)

(continued from previous page)

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



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

Parameters

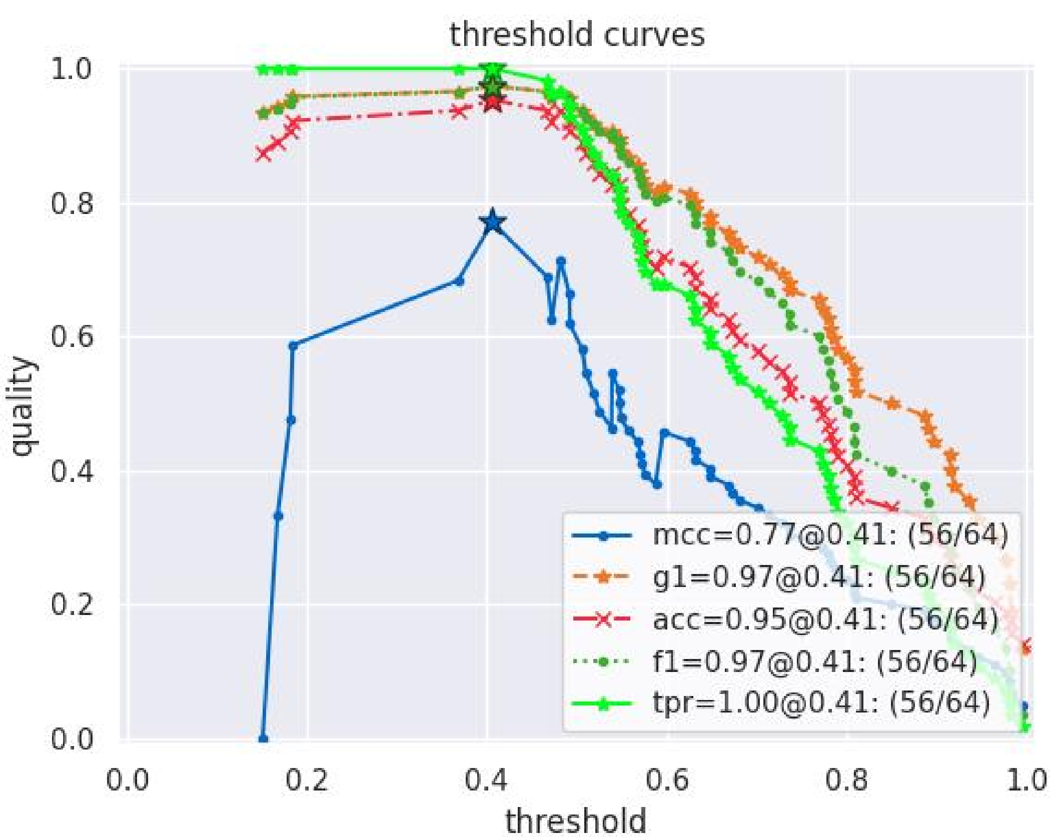
- **info** (*Measures* | *Dict*)
- **keys** (*None* | *List[str]*) – the metrics to view over threhsolds

CommandLine

```
xdoctest -m kwcoco.metrics.drawing draw_threshold_curves --show
```

Example

```
>>> # xdoctest: +REQUIRES(module:kwplot)
>>> 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()
>>> keys = {'g1', 'f1', 'acc', 'mcc', 'tpr'}
>>> draw_threshold_curves(info, keys)
>>> # xdoctest: +REQUIRES(--show)
>>> kwplot.show_if_requested()
```



kwcoco.metrics.drawing.**determenistic_colors**(*keys*, *preset_colors*)

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*) – ground truth class label for each sample
- **y_pred** (*ndarray*) – predicted class label for each sample
- **n_labels** (*int*) – number of labels
- **sample_weight** (*ndarray | None*) – 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]...)
```

kwcoco.metrics.functional.**_truncated_roc**(*y_df*, *bg_idx=-1*, *fp_cutoff=None*)

Computes truncated ROC info

kwcoco.metrics.functional.**_pr_curves**(*y*)

Compute a PR curve from a method

Parameters

y (*pd.DataFrame | DataFrameArray*) – output of detection_confusions

Returns

Tuple[float, ndarray, ndarray]

Example

```
>>> # xdoctest: +REQUIRES(module:sklearn)
>>> import pandas as pd
>>> y1 = pd.DataFrame.from_records([
>>>     {'pred': 0, 'score': 10.00, 'true': -1, 'weight': 1.00},
>>>     {'pred': 0, 'score':  1.65, 'true':  0, 'weight': 1.00},
>>>     {'pred': 0, 'score':  8.64, 'true': -1, 'weight': 1.00},
>>>     {'pred': 0, 'score':  3.97, 'true':  0, 'weight': 1.00},
>>>     {'pred': 0, 'score':  1.68, 'true':  0, 'weight': 1.00},
```

(continues on next page)

(continued from previous page)

```
>>>     {'pred': 0, 'score':  5.06, 'true':  0, 'weight': 1.00},
>>>     {'pred': 0, 'score':  0.25, 'true':  0, 'weight': 1.00},
>>>     {'pred': 0, 'score':  1.75, 'true':  0, 'weight': 1.00},
>>>     {'pred': 0, 'score':  8.52, 'true':  0, 'weight': 1.00},
>>>     {'pred': 0, 'score':  5.20, 'true':  0, 'weight': 1.00},
>>> ])
>>> import kwcoco as nh
>>> import kwarray
>>> y2 = kwarray.DataFrameArray(y1)
>>> _pr_curves(y2)
>>> _pr_curves(y1)
```

kwcoco.metrics.functional.**_average_precision**(*tpr*, *ppv*)

Compute average precision of a binary PR curve. This is simply the area under the curve.

Parameters

- **tpr** (*ndarray*) – true positive rate - aka recall
- **ppv** (*ndarray*) – positive predictive value - aka precision

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*)

kwcoco.metrics.sklearn_alts.**_binary_clf_curve2**(*y_true*, *y_score*, *pos_label=None*, *sample_weight=None*)

MODIFIED VERSION OF SCIKIT-LEARN API

Calculate true and false positives per binary classification threshold.

Parameters

- **y_true** (*array, shape = [n_samples]*) – True targets of binary classification
- **y_score** (*array, shape = [n_samples]*) – Estimated probabilities or decision function
- **pos_label** (*int or str, default=None*) – The label of the positive class
- **sample_weight** (*array-like of shape (n_samples,), default=None*) – Sample weights.

Returns

- **fps** (*array, shape = [n_thresholds]*) – A count of false positives, at index i being the number of negative samples assigned a score >= thresholds[i]. The total number of negative samples is equal to fps[-1] (thus true negatives are given by fps[-1] - fps).
- **tps** (*array, shape = [n_thresholds <= len(np.unique(y_score))]*) – An increasing count of true positives, at index i being the number of positive samples assigned a score >= thresholds[i]. The total number of positive samples is equal to tps[-1] (thus false negatives are given by tps[-1] - tps).
- **thresholds** (*array, shape = [n_thresholds]*) – Decreasing score values.

Example

```
>>> y_true  = [      1,    1,    1,    1,    1,    1,    0]
>>> y_score = [ np.nan, 0.2, 0.3, 0.4, 0.5, 0.6, 0.3]
>>> sample_weight = None
>>> pos_label = None
>>> fps, tps, thresholds = _binary_clf_curve2(y_true, y_score)
```

2.1.1.5.1.9 kwcoco.metrics.util module

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>]].
- **classes** (*None | List[str] |* kwcoco.CategoryTree) – class names

add_truth(*true_dets, gid*)

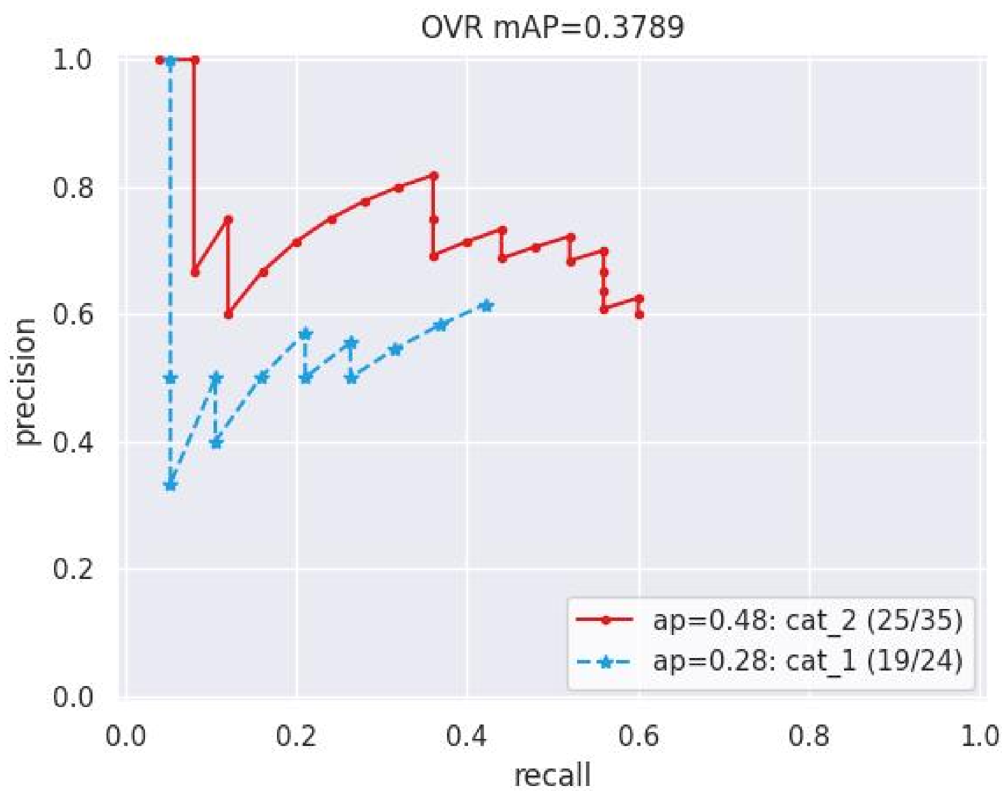
add_predictions(*pred_dets, gid*)

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, newstyle=0)
>>> gid = ub.peek(dmet.gid_to_pred_dets)
>>> self = VOC_Metrics(classes=dmet.classes)
>>> self.add_truth(dmet.true_detections(gid), gid)
>>> self.add_predictions(dmet.pred_detections(gid), gid)
>>> voc_scores = self.score()
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.figure(fnum=1, doclf=True)
>>> voc_scores['perclass'].draw(key='pr')
```



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)

kwcoco.metrics.voc_metrics.**_pr_curves**(*y*, *method='voc2012'*)

Compute a PR curve from a method

Parameters

y (*pd.DataFrame* | *DataFrameArray*) – output of detection_confusions

Returns

Tuple[float, ndarray, ndarray]

Example

```
>>> import pandas as pd
>>> y1 = pd.DataFrame.from_records([
>>>     {'pred': 0, 'score': 10.00, 'true': -1, 'weight': 1.00},
>>>     {'pred': 0, 'score':  1.65, 'true':  0, 'weight': 1.00},
>>>     {'pred': 0, 'score':  8.64, 'true': -1, 'weight': 1.00},
>>>     {'pred': 0, 'score':  3.97, 'true':  0, 'weight': 1.00},
>>>     {'pred': 0, 'score':  1.68, 'true':  0, 'weight': 1.00},
>>>     {'pred': 0, 'score':  5.06, 'true':  0, 'weight': 1.00},
>>>     {'pred': 0, 'score':  0.25, 'true':  0, 'weight': 1.00},
>>>     {'pred': 0, 'score':  1.75, 'true':  0, 'weight': 1.00},
>>>     {'pred': 0, 'score':  8.52, 'true':  0, 'weight': 1.00},
```

(continues on next page)

(continued from previous page)

```
>>>     {'pred': 0, 'score':  5.20, 'true':  0, 'weight': 1.00},
>>> ])
>>> import kwarray
>>> y2 = kwarray.DataFrameArray(y1)
>>> _pr_curves(y2)
>>> _pr_curves(y1)
```

`kwcoco.metrics.voc_metrics._voc_eval`(*lines*, *recs*, *classname*, *iou_thresh=0.5*, *method='voc2012'*, *bias=1.0*)

VOC AP evaluation for a single category.

Parameters

- **lines** (*List[list]*) – VOC formatted predictions. Each "line" is a list of [[<imgid>, <score>, <tl_x>, <tl_y>, <br_x>, <br_y>]].
- **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.
- **classname** (*str*) – the category to evaluate.
- **method** (*str*) – code for how the AP is computed.
- **bias** (*float*) – either 1.0 or 0.0.

Returns
info about the evaluation containing AP. Contains fp, tp, prec, rec,

Return type
Dict

Note: Raw replication of matlab implementation of creating assignments and the resulting PR-curves and AP. Based on MATLAB code [1].

References

[1] http://host.robots.ox.ac.uk/pascal/VOC/voc2012/VOCdevkit_18-May-2011.tar

`kwcoco.metrics.voc_metrics._voc_ave_precision`(*rec*, *prec*, *method='voc2012'*)

Compute AP from precision and recall Based on MATLAB code in[1],[2], and[3].

Parameters

- **rec** (*ndarray*) – recall
- **prec** (*ndarray*) – precision
- **method** (*str*) – either voc2012 or voc2007

Returns
ap: average precision

Return type
float

1 http://host.robots.ox.ac.uk/pascal/VOC/voc2012/VOCdevkit_18-May-2011.tar
2 https://github.com/rbgirshick/voc-dpm/blob/master/test/pascal_eval.m
3 https://github.com/rbgirshick/voc-dpm/blob/c0b88564bd668bcc6216bbffe96cb061613be768/utils/bootstrap/VOCevaldet_bootstrap.m

References

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.urepr(self.measures())))
```

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

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

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

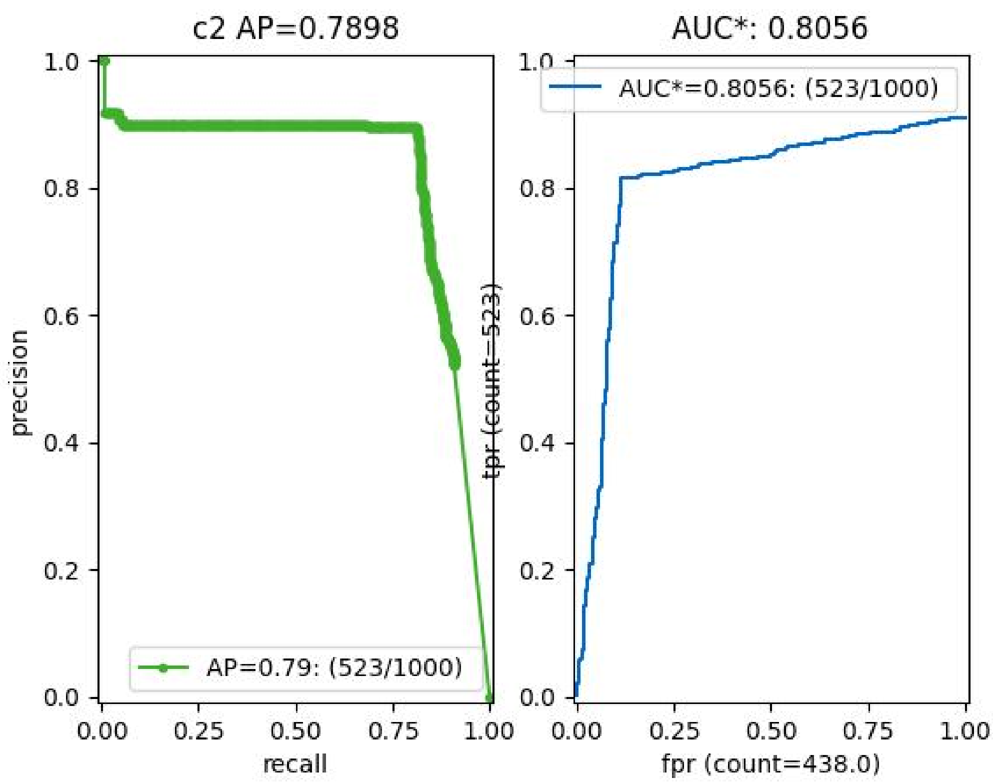
Parameters

- **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* | *None*) – 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.urepr(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* | *None*) – maximum number of false positives in the truncated roc curves. The default of `None` is equivalent to `float('inf')`
- **monotonic_ppv** (*bool*) – 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.urepr(self.measures())))
>>> self = BinaryConfusionVectors.demo(n=1, p_true=0.5, p_error=0.5)
>>> print('measures = {}'.format(ub.urepr(self.measures())))
>>> self = BinaryConfusionVectors.demo(n=3, p_true=0.5, p_error=0.5)
>>> print('measures = {}'.format(ub.urepr(self.measures())))
```

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

_binary_clf_curves(*stabalize_thresh=7*, *fp_cutoff=None*)

Compute TP, FP, TN, and FN counts for this binary confusion vector.

Code common to ROC, PR, and threshold measures, computes the elements of the binary confusion matrix at all relevant operating point thresholds.

Parameters

- **stabalize_thresh** (*int*) – if fewer than this many data points insert stabalization data.
- **fp_cutoff** (*int* | *None*) – maximum number of false positives

Example

```
>>> from kwcoco.metrics.confusion_vectors import *  # NOQA
>>> self = BinaryConfusionVectors.demo(n=1, p_true=0.5, p_error=0.5)
>>> self._binary_clf_curves()
```

```
>>> self = BinaryConfusionVectors.demo(n=0, p_true=0.5, p_error=0.5)
>>> self._binary_clf_curves()
```

```
>>> self = BinaryConfusionVectors.demo(n=100, p_true=0.5, p_error=0.5)
>>> self._binary_clf_curves()
```

draw_distribution()

_3dplot()

Example

```
>>> # xdoctest: +REQUIRES(module:kwplot)
>>> # xdoctest: +REQUIRES(module:pandas)
>>> from kwcoco.metrics.confusion_vectors import *  # NOQA
>>> from kwcoco.metrics.detect_metrics import DetectionMetrics
>>> dmet = DetectionMetrics.demo(
>>>     n_fp=(0, 1), n_fn=(0, 2), nimgs=256, nboxes=(0, 10),
>>>     bbox_noise=10,
>>>     classes=1)
>>> cfsn_vecs = dmet.confusion_vectors()
>>> self = bin_cfsn = cfsn_vecs.binarize_classless()
>>> #dmet.summarize(plot=True)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.figure(fnum=3)
>>> self._3dplot()
```

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 | None*) – probabilities for each class

Example

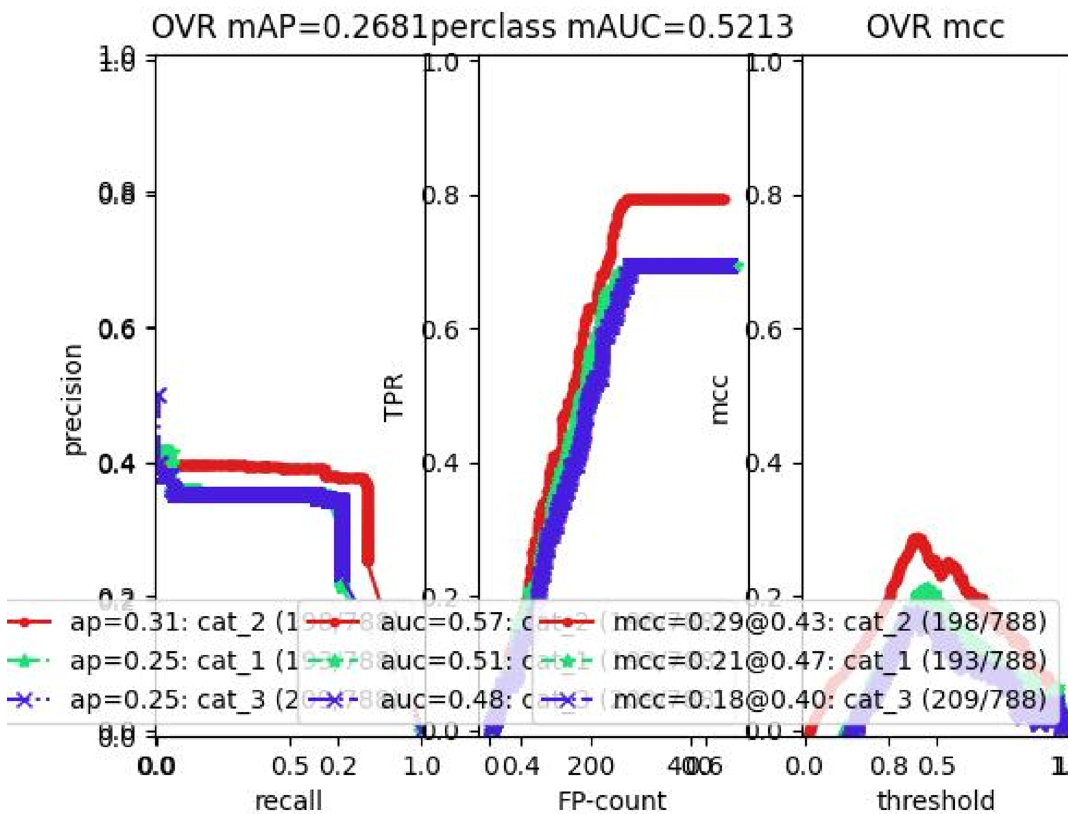
```
>>> # xdoctest: IGNORE_WANT
>>> # xdoctest: +REQUIRES(module:pandas)
>>> from kwcoco.metrics import DetectionMetrics
>>> dmet = DetectionMetrics.demo(
>>>     nimgs=10, nboxes=(0, 10), n_fp=(0, 1), classes=3)
>>> cfsn_vecs = dmet.confusion_vectors()
>>> print(cfsn_vecs.data._pandas())
     pred 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

(continues on next page)

(continued from previous page)

```
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

```
>>> from kwcoco.metrics.confusion_vectors import *  # NOQA
>>> 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, newstyle=False)
>>> cfsn_vecs = dmet.confusion_vectors()
>>> cm = cfsn_vecs.confusion_matrix()
...
>>> print(cm.to_string(float_format=lambda x: '%.2f' % x))
pred        background  cat_1  cat_2  cat_3
real
background        0.00   1.00   2.00   3.00
cat_1             3.00  12.00   0.00   0.00
```

(continues on next page)

(continued from previous page)

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

coarsen(*cxs*)

Creates a coarsened set of vectors

Returns
ConfusionVectors

binarize_classless(*negative_classes=None*)

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

Parameters
negative_classes (*List[str | int] | None*) – 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[int, kwimage.Detections]*) – maps image ids to truth
- **gid_to_pred_dets** (*Dict[int, kwimage.Detections]*) – maps image ids to predictions
- **classes** (kwcoco.CategoryTree | *None*) – the categories to be scored, if unspecified attempts to determine these from the truth detections

Example

```
>>> # Demo how to use detection metrics directly given detections only
>>> # (no kwcoco file required)
>>> from kwcoco.metrics import detect_metrics
>>> import kwimage
>>> # Setup random true detections (these are just boxes and scores)
>>> true_dets = kwimage.Detections.random(3)
>>> # Peek at the simple internals of a detections object
>>> print('true_dets.data = {}'.format(ub.urepr(true_dets.data, nl=1)))
>>> # Create similar but different predictions
>>> true_subset = true_dets.take([1, 2]).warp(kwimage.Affine.coerce({'scale': 1.1}))
>>> false_positive = kwimage.Detections.random(3)
>>> pred_dets = kwimage.Detections.concatenate([true_subset, false_positive])
```

(continues on next page)

(continued from previous page)

```
>>> dmet = DetectionMetrics()
>>> dmet.add_predictions(pred_dets, imgname='image1')
>>> dmet.add_truth(true_dets, imgname='image1')
>>> # Raw confusion vectors
>>> cfsn_vecs = dmet.confusion_vectors()
>>> print(cfsn_vecs.data.pandas().to_string())
>>> # Our scoring definition (derived from confusion vectors)
>>> print(dmet.score_kwcoco())
>>> # VOC scoring
>>> print(dmet.score_voc(bias=0))
>>> # Original pycocotools scoring
>>> # xdoctest: +REQUIRES(module:pycocotools)
>>> print(dmet.score_pycocotools())
```

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()

enrich_confusion_vectors(*cfsn_vecs*)

Adds annotation id information into confusion vectors computed via this detection metrics object.

TODO: should likely use this at the end of the function that builds the confusion vectors.

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*) – coco dataset with ground truth
- **pred_coco** (*kwcoco.CocoDataset*) – coco dataset with predictions

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()
```

_register_imagename(*imgname*, *gid=None*)

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

Register/Add predicted detections for an image

Parameters

- **pred_dets** (*kwimage.Detections*) – predicted detections
- **imgname** (*str* | *None*) – 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* | *None*) – 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

property classes

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]*) – bounding box overlap iou threshold required for assignment if a list, then return type is a dict. Defaults to 0.5
- **bias** (*float*) – for computing bounding box overlap, either 1 or 0 Defaults to 0.
- **gids** (*List[int]* | *None*) – which subset of images ids to compute confusion metrics on. If not specified all images are used. Defaults to None.
- **compat** (*str*) – 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. Defaults to all.
- **prioritize** (*str*) – 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. Default to 'iou'
- **ignore_classes** (*set* | *str*) – class names indicating ignore regions. Default={'ignore'}

- **background_class** (*str | NoParamType*) – 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*) – verbosity flag. Default to 'auto'. In auto mode, verbose=1 if len(gids) > 1000.
- **workers** (*int*) – number of parallel assignment processes. Defaults to 0
- **track_probs** (*str*) – can be 'try', 'force', or False. if truthy, we assume probabilities for multiple classes are available. default='try'

Returns
ConfusionVectors | Dict[float, 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...
```

_to_coco()

Convert to a coco representation of truth and predictions

with inverse aid mappings

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.urepr(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.urepr(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.

rng (int | None | RandomState): random seed / state

CommandLine

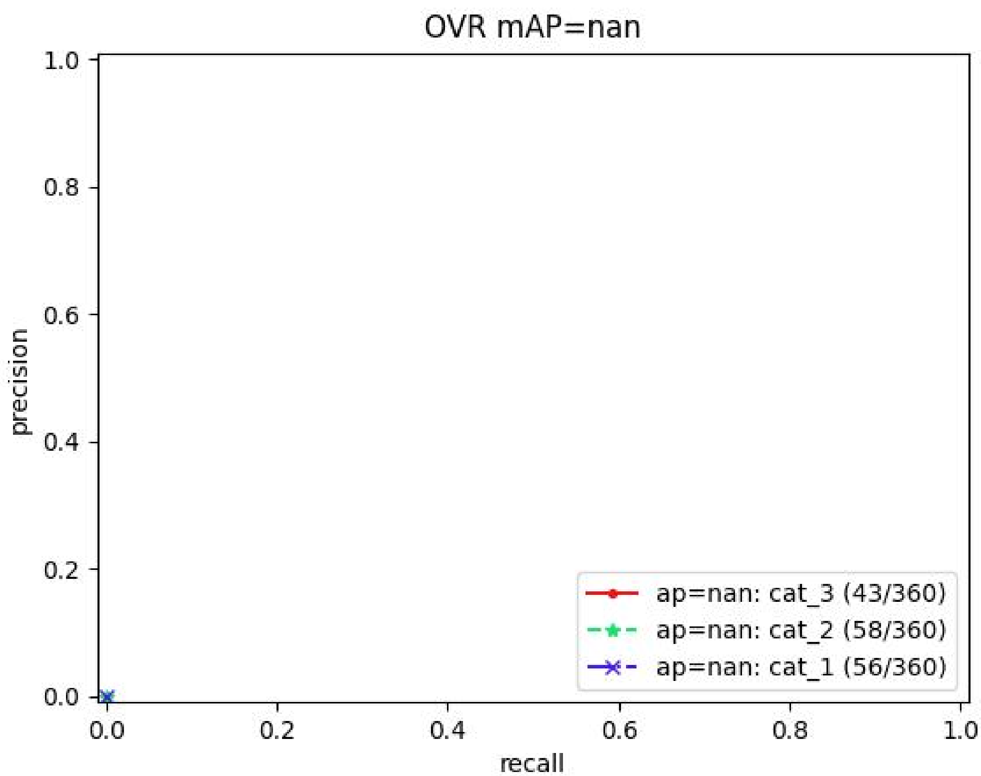
```
xdoctest -m kwcoco.metrics.detect_metrics DetectionMetrics.demo:2 --show
```

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'*)

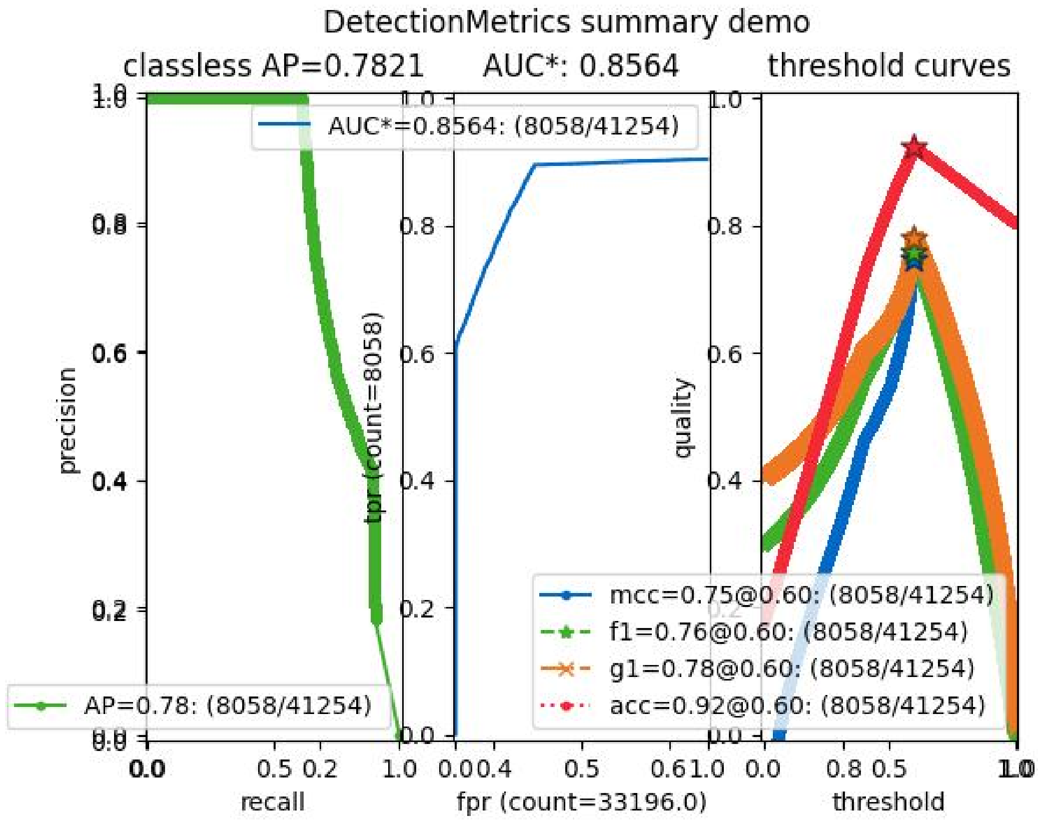
Create summary one-versus-rest and binary metrics.

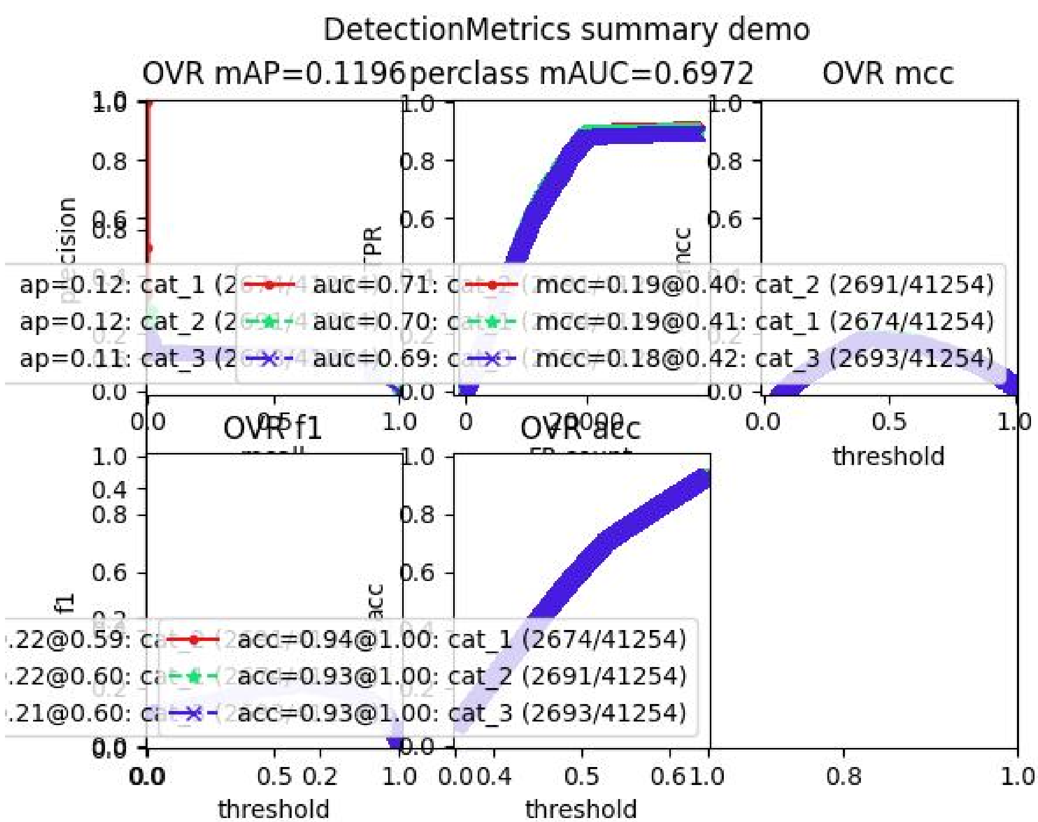
Parameters

- **out_dpath** (*pathlib.Path | None*) – FIXME: not hooked up
- **with_ovr** (*str | bool*) – include one-versus-rest metrics (wrt the classes). If 'auto' enables if possible. FIXME: auto is not working.
- **with_bin** (*str | bool*) – include binary classless metrics (i.e. detected or not). If 'auto' enables if possible. FIXME: auto is not working.
- **plot** (*bool*) – if true, also write plots. Defaults to False.
- **title** (*str*) – passed if plot is given

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()
```


DetectionMetrics summary demo
classless AP=0.7821
AUC*: 0.8564
threshold curves
AUC*=0.8564: (8058/41254)
precision
recall
tpr (count=8058)
fpr (count=33196.0)
quality
threshold
AP=0.78: (8058/41254)
mcc=0.75@0.60: (8058/41254)
f1=0.76@0.60: (8058/41254)
g1=0.78@0.60: (8058/41254)
acc=0.92@0.60: (8058/41254)


DetectionMetrics summary demo
OVR mAP=0.1196
perclass mAUC=0.6972
OVR mcc
ap=0.12: cat_1
ap=0.12: cat_2
ap=0.11: cat_3
auc=0.71:
auc=0.70:
auc=0.69:
mcc=0.19@0.40: cat_2 (2691/41254)
mcc=0.19@0.41: cat_1 (2674/41254)
mcc=0.18@0.42: cat_3 (2693/41254)
OVR f1
OVR acc
.22@0.59:
.22@0.60:
.21@0.60:
acc=0.94@1.00: cat_1 (2674/41254)
acc=0.93@1.00: cat_2 (2691/41254)
acc=0.93@1.00: cat_3 (2693/41254)
threshold

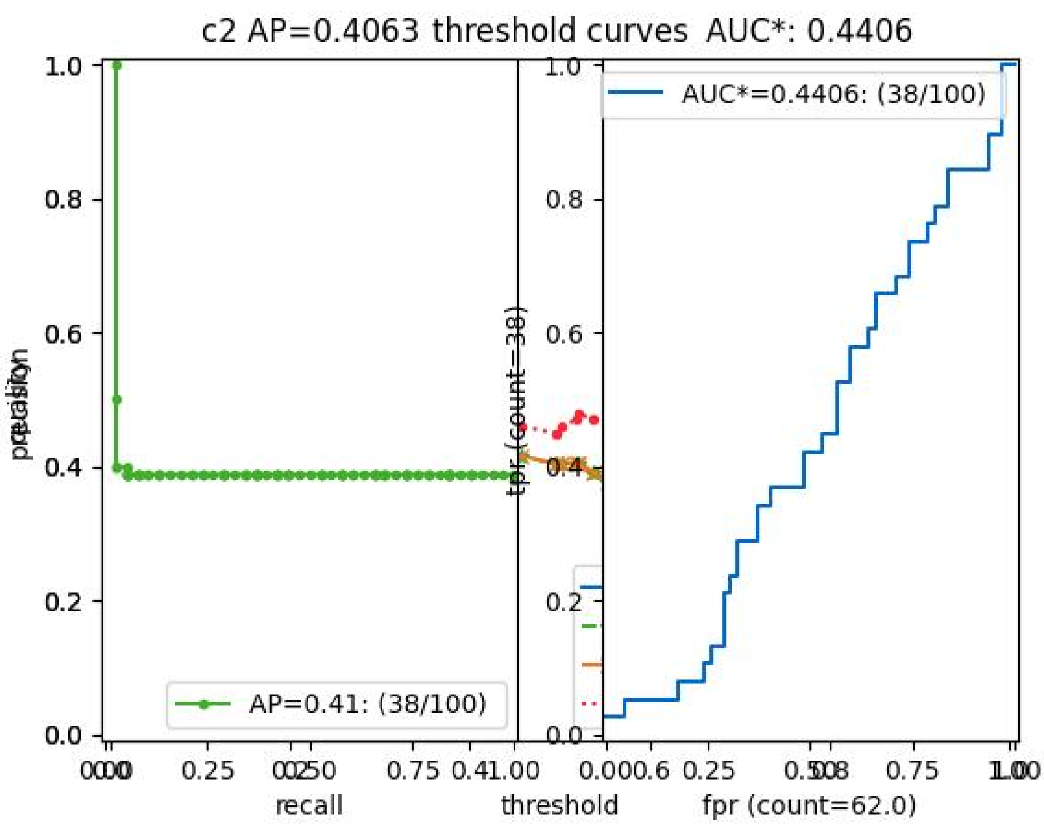
class kwcoco.metrics.**Measures**(*info*)

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*)

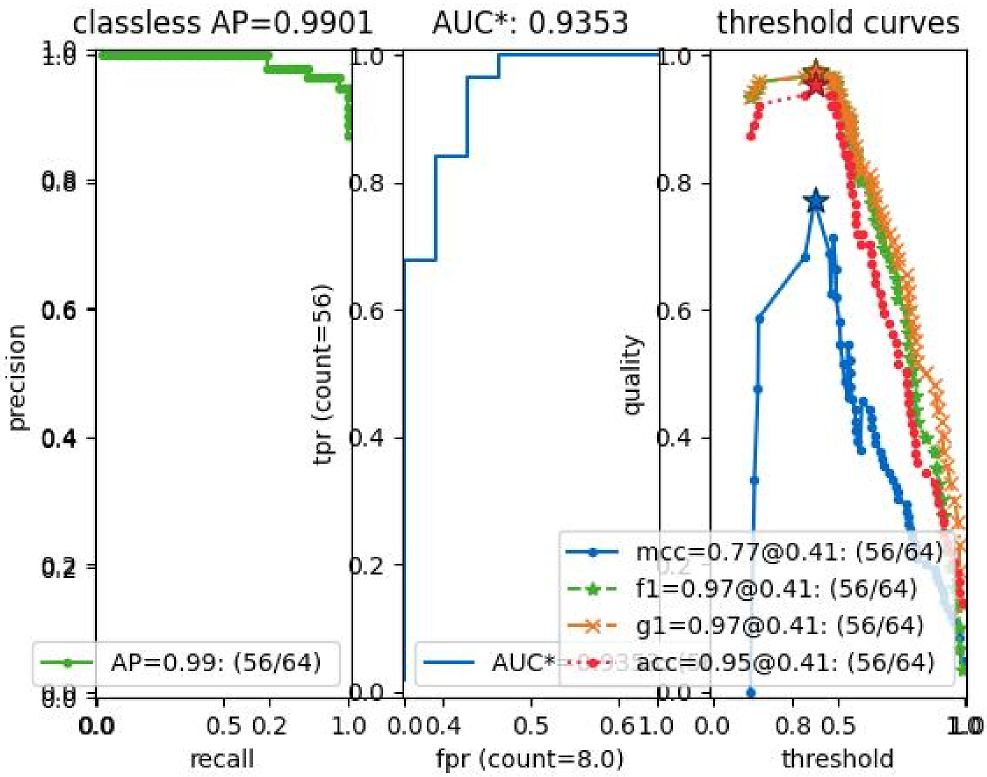
Example

```
>>> # 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

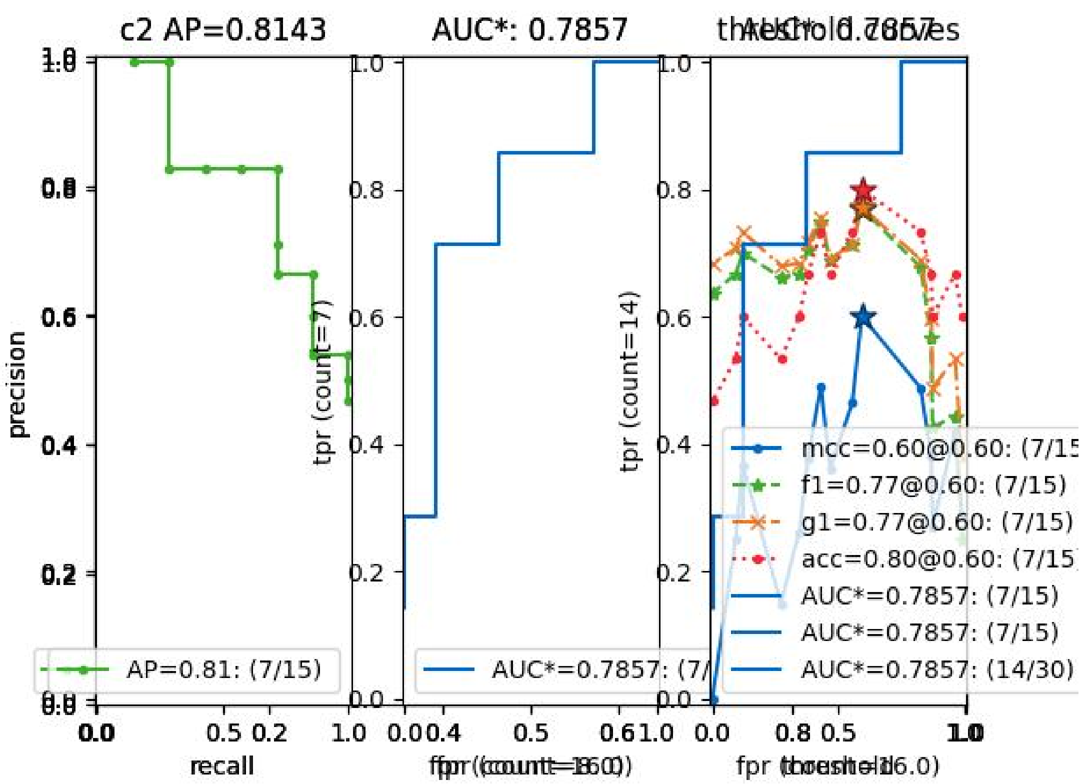
- **tocombine** (*List[Measures]*) – a list of measures to combine into one
- **precision** (*int | None*) – If specified rounds thresholds to this precision which can prevent a RAM explosion when combining a large number of measures. However, this is a lossy operation and will impact the underlying 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 | None*) – 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.urepr(measures1.__json__(), nl=1, sort=0))
>>> print(ub.urepr(measures2.__json__(), nl=1, sort=0))
>>> print(ub.urepr(new_measures.__json__(), nl=1, sort=0))
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.figure(fnum=1)
>>> new_measures.summary_plot()
>>> measures1.summary_plot()
>>> measures1.draw('roc')
>>> measures2.draw('roc')
>>> new_measures.draw('roc')
```



Example

```
>>> # Demonstrate issues that can arrise from choosing a precision
>>> # that is too low  when combining metrics. Breakpoints
>>> # between different metrics can get muddled, but choosing a
>>> # precision that is too high can overwhelm memory.
>>> from kwcoco.metrics.confusion_measures import *  # NOQA
>>> base = ub.map_vals(np.asarray, {
>>>     'tp_count':   [ 1,   1,   2,   2,   2,   2,   3],
>>>     'fp_count':   [ 0,   1,   1,   2,   3,   4,   5],
>>>     'fn_count':   [ 1,   1,   0,   0,   0,   0,   0],
>>>     'tn_count':   [ 5,   4,   4,   3,   2,   1,   0],
>>>     'thresholds': [.0,  .0,  .0,  .0,  .0,  .0,  .0],
>>> })
>>> # Make tiny offsets to thresholds
>>> rng = kwarray.ensure_rng(0)
>>> n = len(base['thresholds'])
>>> offsets = [
>>>     sorted(rng.rand(n) * 10 ** -rng.randint(4, 7))[::-1]
>>>     for _ in range(20)
>>> ]
>>> tocombine = []
>>> for offset in offsets:
>>>     base_n = base.copy()
>>>     base_n['thresholds'] += offset
>>>     measures_n = Measures(base_n).reconstruct()
>>>     tocombine.append(measures_n)
>>> for precision in [6, 5, 2]:
>>>     combo = Measures.combine(tocombine, precision=precision).reconstruct()
>>>     print('precision = {!r}'.format(precision))
>>>     print('combo = {}'.format(ub.urepr(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.urepr(combo, nl=1)))
>>>     print('num_thresholds = {}'.format(len(combo['thresholds'])))
>>>     #print(combo.counts().pandas())
```

Example

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

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

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

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

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

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

Example

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

(continues on next page)

(continued from previous page)

```
↪count': [0.0], 'tp_count': [9804.0]},
>>>     {'fn_count': [0.0], 'fp_count': [677826.0], 'thresholds': [0.0], 'tn_
↪count': [0.0], 'tp_count': [2470.0]},
>>>     {'fn_count': [0.0], 'fp_count': [677834.0], 'thresholds': [0.0], 'tn_
↪count': [0.0], 'tp_count': [2470.0]},
>>>     {'fn_count': [0.0], 'fp_count': [677835.0], 'thresholds': [0.0], 'tn_
↪count': [0.0], 'tp_count': [2470.0]},
>>>     {'fn_count': [11123.0, 0.0], 'fp_count': [0.0, 676754.0], 'thresholds':␣
↪[0.0002442002442002442, 0.0], 'tn_count': [676754.0, 0.0], 'tp_count': [2.0,␣
↪11125.0]},
>>>     {'fn_count': [7738.0, 0.0], 'fp_count': [0.0, 676466.0], 'thresholds':␣
↪[0.0002442002442002442, 0.0], 'tn_count': [676466.0, 0.0], 'tp_count': [0.0,␣
↪7738.0]},
>>>     {'fn_count': [8653.0, 0.0], 'fp_count': [0.0, 676341.0], 'thresholds':␣
↪[0.0002442002442002442, 0.0], 'tn_count': [676341.0, 0.0], 'tp_count': [0.0,␣
↪8653.0]},
>>> ]
>>> thresh_bins = np.linspace(0, 1, 4)
>>> combo = Measures.combine(tocombine, thresh_bins=thresh_bins).reconstruct()
>>> print('tocombine = {}'.format(ub.urepr(tocombine, nl=2)))
>>> print('thresh_bins = {!r}'.format(thresh_bins))
>>> print(ub.urepr(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.urepr(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.urepr(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.urepr(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*) – if fewer than this many data points inserts dummy stabilization data so curves can still be drawn. Default to 7.
- **fp_cutoff** (*int | None*) – maximum number of false positives in the truncated roc curves. The default `None` is equivalent to `float('inf')`
- **monotonic_ppv** (*bool*) – 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. Default to True.

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*)

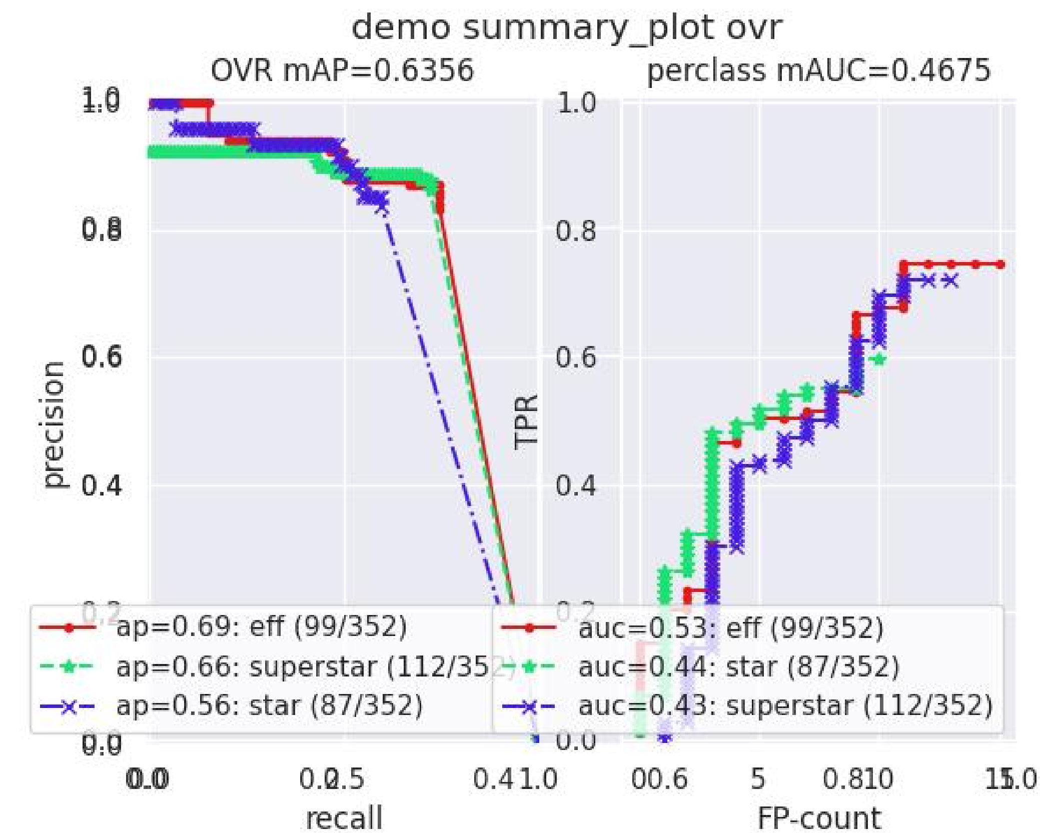
summary_plot(*fnum=1*, *title=''*, *subplots='auto'*)

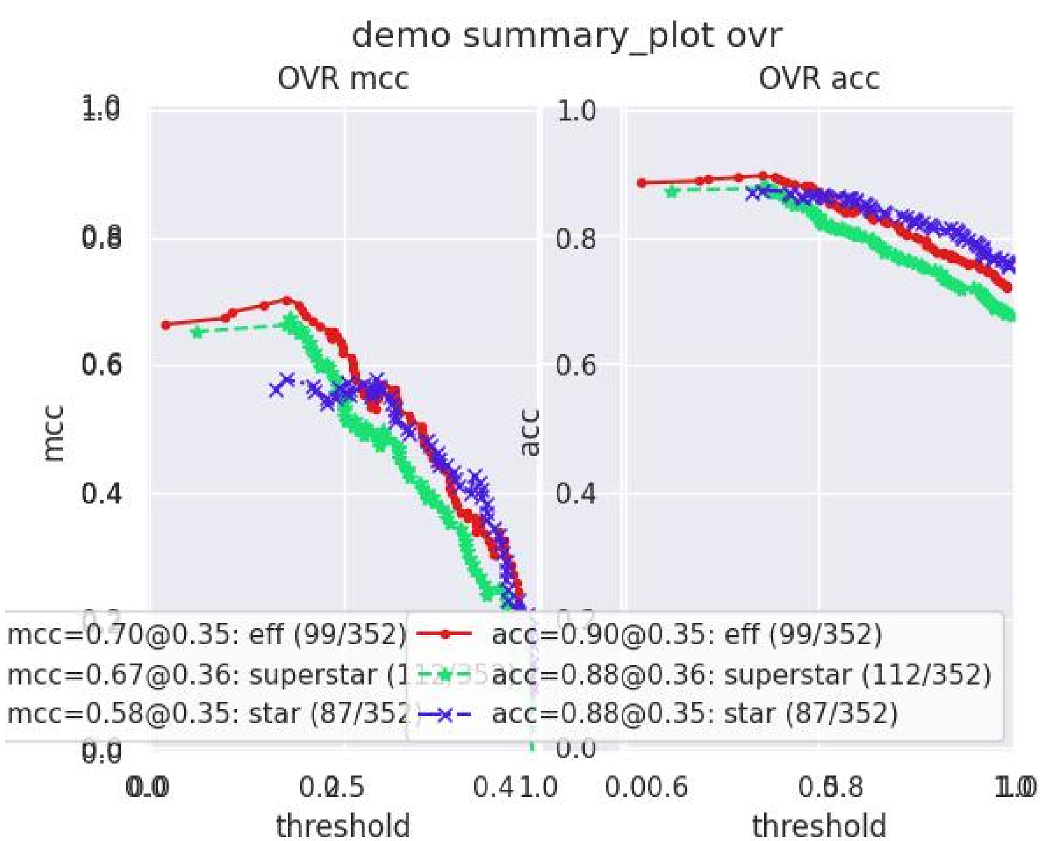
CommandLine

```
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)
```


demo summary_plot ovr
OVR mAP=0.6356
perclass mAUC=0.4675
precision
recall
TPR
FP-count
ap=0.69: eff (99/352)
ap=0.66: superstar (112/352)
ap=0.56: star (87/352)
auc=0.53: eff (99/352)
auc=0.44: star (87/352)
auc=0.43: superstar (112/352)


demo summary_plot ovr
OVR mcc
OVR acc
mcc
acc
threshold
threshold
mcc=0.70@0.35: eff (99/352)
mcc=0.67@0.36: superstar (112/352)
mcc=0.58@0.35: star (87/352)
acc=0.90@0.35: eff (99/352)
acc=0.88@0.36: superstar (112/352)
acc=0.88@0.35: star (87/352)

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

DEPRECATED USE *kwcoco eval* instead

CommandLine

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

2.1.1.6 kwcoco.util package

2.1.1.6.1 Subpackages

2.1.1.6.1.1 kwcoco.util.delayed_ops package

2.1.1.6.1.2 Module contents

Functionality has been ported to delayed_image

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

Bases: *DelayedUnaryOperation*

A generic NDArray.

Parameters
subdata (*DelayedArray*)

property shape

Returns: None | Tuple[int | None, …]

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

Bases: *DelayedImage*

Casts the data to an xarray object in the finalize step

Example;

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

Parameters

- **subdata** (*DelayedArray*)

- **dsize** (*None | Tuple[int | None, int | None]*) – overrides subdata dsize
- **channels** (*None | int | FusedChannelSpec*) – overrides subdata channels

_finalize()

Returns
ArrayLike

optimize()

Returns
DelayedImage

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

Bases: *ImageOpsMixin*, *DelayedConcat*

Stacks multiple arrays together.

Example

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

Example

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

Parameters

- **parts** (*List[DelayedArray]*) – data to concat

- **dsize** (*Tuple[int, int] | None*) – size if known a-priori

property channels

Returns: None | FusedChannelSpec

property shape

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

_finalize()

Returns
ArrayLike

optimize()

Returns
DelayedImage

take_channels(*channels*)

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

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

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

Return type
DelayedArray

Example

```
>>> # xdoctest: +REQUIRES(module:kwcoco)
>>> from delayed_image.delayed_nodes import *  # NOQA
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('vidshapes8-multispectral')
>>> self = delayed = dset.coco_image(1).delay()
>>> channels = 'B11|B8|B1|B10'
>>> new = self.take_channels(channels)
```

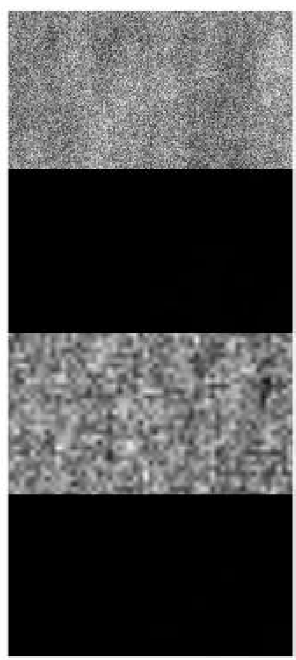
Example

```
>>> # xdoctest: +REQUIRES(module:kwcoco)
>>> # Complex case
>>> import kwcoco
>>> from delayed_image.delayed_nodes import *  # NOQA
>>> from delayed_image.delayed_leafs import DelayedLoad
>>> dset = kwcoco.CocoDataset.demo('vidshapes8-multispectral')
>>> delayed = dset.coco_image(1).delay()
>>> astro = DelayedLoad.demo('astro', channels='r|g|b').prepare()
>>> aligned = astro.warp(kwimage.Affine.scale(600 / 512), dsize='auto')
>>> self = combo = DelayedChannelConcat(delayed.parts + [aligned])
```

(continues on next page)

(continued from previous page)

```
>>> channels = 'B1|r|B8|g'
>>> new = self.take_channels(channels)
>>> new_cropped = new.crop((slice(10, 200), slice(12, 350)))
>>> new_opt = new_cropped.optimize()
>>> datas = new_opt.finalize()
>>> if 1:
>>>     new_cropped.write_network_text(with_labels='name')
>>>     new_opt.write_network_text(with_labels='name')
>>> vizable = kwimage.normalize_intensity(datas, axis=2)
>>> self._validate()
>>> new._validate()
>>> new_cropped._validate()
>>> new_opt._validate()
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> stacked = kwimage.stack_images(vizable.transpose(2, 0, 1))
>>> kwplot.imshow(stacked)
```



Example

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

property num_overviews

Returns: int

as_xarray()

Returns
DelayedAsXarray

_push_operation_under(*op*, *kwargs*)

_validate()

Check that the delayed metadata corresponds with the finalized data

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

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

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

Parameters

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

Returns
The List[DelayedImage] are the `parts` i.e. the new images with the warping undone. The List[kwimage.Affine]: is the transforms from `self` to each item in `parts`

Return type
List[*DelayedImage*] | Tuple[List[*DelayedImage*] | List[kwimage.Affine]]

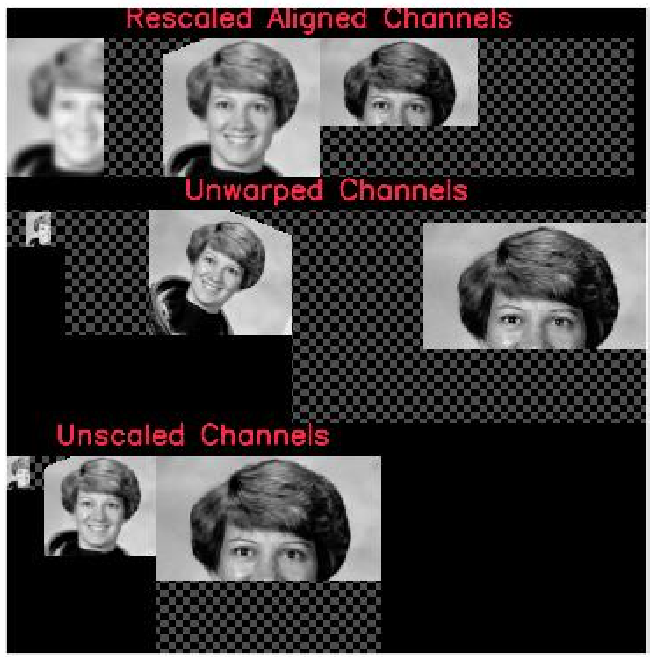
Example

```
>>> from delayed_image.delayed_nodes import *  # NOQA
>>> from delayed_image.delayed_leafs import DelayedLoad
>>> from delayed_image.delayed_leafs import DelayedNans
>>> import ubelt as ub
>>> import kwimage
>>> import kwarray
>>> import numpy as np
>>> # Demo case where we have different channels at different resolutions
>>> base = DelayedLoad.demo(channels='r|g|b').prepare().dequantize({'quant_max
↪': 255})
>>> bandR = base[:, :, 0].scale(100 / 512)[:, :-50].evaluate()
>>> bandG = base[:, :, 1].scale(300 / 512).warp({'theta': np.pi / 8, 'about':␣
↪(150, 150)}).evaluate()
>>> bandB = base[:, :, 2].scale(600 / 512)[:150, :].evaluate()
>>> bandN = DelayedNans((600, 600), channels='N')
>>> # Make a concatenation of images of different underlying native resolutions
>>> delayed_vidspace = DelayedChannelConcat([
>>>     bandR.scale(6, dsize=(600, 600)).optimize(),
>>>     bandG.warp({'theta': -np.pi / 8, 'about': (150, 150)}).scale(2,␣
↪dsize=(600, 600)).optimize(),
>>>     bandB.scale(1, dsize=(600, 600)).optimize(),
>>>     bandN,
>>> ]).warp({'scale': 0.7}).optimize()
>>> vidspace_box = kwimage.Boxes([[100, 10, 270, 160]], 'ltrb')
>>> vidspace_poly = vidspace_box.to_polygons()[0]
>>> vidspace_slice = vidspace_box.to_slices()[0]
>>> self = delayed_vidspace[vidspace_slice].optimize()
>>> print('--- Aligned --- ')
>>> self.write_network_text()
>>> squash_nans = True
>>> undone_all_parts, tfs1 = self.undo_warps(squash_nans=squash_nans, return_
↪warps=True)
>>> undone_scale_parts, tfs2 = self.undo_warps(remove=['scale'], squash_
↪nans=squash_nans, return_warps=True)
>>> stackable_aligned = self.finalize().transpose(2, 0, 1)
>>> stackable_undone_all = []
>>> stackable_undone_scale = []
>>> print('--- Undone All --- ')
>>> for undone in undone_all_parts:
...     undone.write_network_text()
...     stackable_undone_all.append(undone.finalize())
>>> print('--- Undone Scale --- ')
>>> for undone in undone_scale_parts:
...     undone.write_network_text()
...     stackable_undone_scale.append(undone.finalize())
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
```

(continues on next page)

(continued from previous page)

```
>>> kwplot.autompl()
>>> canvas0 = kwimage.stack_images(stackable_aligned, axis=1)
>>> canvas1 = kwimage.stack_images(stackable_undone_all, axis=1)
>>> canvas2 = kwimage.stack_images(stackable_undone_scale, axis=1)
>>> canvas0 = kwimage.draw_header_text(canvas0, 'Rescaled Aligned Channels')
>>> canvas1 = kwimage.draw_header_text(canvas1, 'Unwarped Channels')
>>> canvas2 = kwimage.draw_header_text(canvas2, 'Unscaled Channels')
>>> canvas = kwimage.stack_images([canvas0, canvas1, canvas2], axis=0)
>>> canvas = kwimage.fill_nans_with_checkers(canvas)
>>> kwplot.imshow(canvas)
```



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

Bases: `DelayedNaryOperation`

Stacks multiple arrays together.

Parameters

- **parts** (*List[DelayedArray]*) – data to concat
- **axis** (*int*) – axes to concat on

property shape

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

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

Bases: `DelayedImage`

Crops an image along integer pixel coordinates.

Example

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

Example

```
>>> from delayed_image.delayed_nodes import *  # NOQA
>>> from delayed_image import DelayedLoad
>>> base = DelayedLoad.demo(dsize=(16, 16)).prepare()
>>> # Test Discontiguous Channel Select Via Index
>>> self = base[:, :, [0, 2]]
>>> self.write_network_text()
>>> final = self._finalize()
>>> assert final.shape == (16, 16, 2)
```

Parameters

- **subdata** (*DelayedArray*) – data to operate on
- **space_slice** (*Tuple[slice, slice]*) – if speficied, take this y-slice and x-slice.
- **chan_idxs** (*List[int] | None*) – if specified, take these channels / bands

_finalize()

Returns
ArrayLike

_transform_from_subdata()

optimize()

Returns
DelayedImage

Example

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

_opt_fuse_crops()

Combine two consecutive crops into a single operation.

Example

```
>>> from delayed_image.delayed_nodes import *  # NOQA
>>> from delayed_image.delayed_leafs import DelayedLoad
>>> base = DelayedLoad.demo(dsize=(16, 16)).prepare()
>>> # Test Fuse Crops Space Only
>>> crop1 = base[4:12, 0:16]
>>> crop2 = self = crop1[2:6, 0:8]
>>> opt = crop2._opt_fuse_crops()
>>> self.write_network_text()
>>> opt.write_network_text()
>>> opt._validate()
>>> self._validate()
```

Example

```
>>> # Test Fuse Crops Channels Only
>>> from delayed_image.delayed_nodes import *  # NOQA
>>> from delayed_image.delayed_leafs import DelayedLoad
>>> base = DelayedLoad.demo(dsize=(16, 16)).prepare()
>>> crop1 = base.crop(chan_idxs=[0, 2, 1])
>>> crop2 = crop1.crop(chan_idxs=[1, 2])
>>> crop3 = self = crop2.crop(chan_idxs=[0, 1])
>>> opt = self._opt_fuse_crops()._opt_fuse_crops()
>>> self.write_network_text()
>>> opt.write_network_text()
>>> finalB = base._validate()._finalize()
>>> final1 = opt._validate()._finalize()
>>> final2 = self._validate()._finalize()
>>> assert np.all(final2[..., 0] == finalB[..., 2])
>>> assert np.all(final2[..., 1] == finalB[..., 1])
>>> assert np.all(final2[..., 0] == final1[..., 0])
>>> assert np.all(final2[..., 1] == final1[..., 1])
```

Example

```
>>> # Test Fuse Crops Space  And Channels
>>> from delayed_image.delayed_nodes import *  # NOQA
>>> from delayed_image.delayed_leafs import DelayedLoad
>>> base = DelayedLoad.demo(dsize=(16, 16)).prepare()
>>> crop1 = base[4:12, 0:16, [1, 2]]
>>> self = crop1[2:6, 0:8, [1]]
>>> opt = self._opt_fuse_crops()
>>> self.write_network_text()
>>> opt.write_network_text()
>>> self._validate()
>>> crop1._validate()
```

_opt_warp_after_crop()

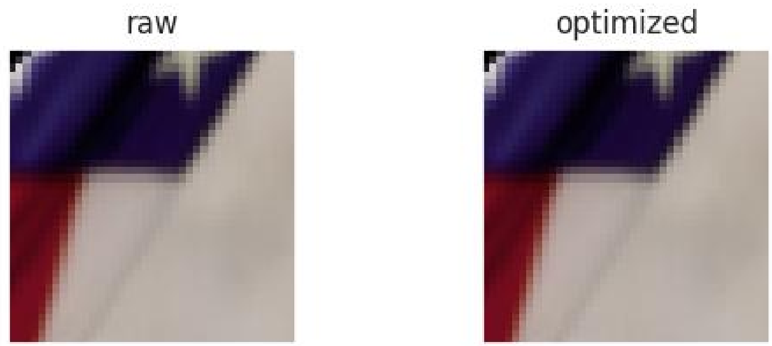
If the child node is a warp, move it after the crop.

This is more efficient because:

1. The crop is closer to the load.
2. we are warping with less data.

Example

```
>>> from delayed_image.delayed_nodes import *  # NOQA
>>> from delayed_image.delayed_leafs import DelayedLoad
>>> fpath = kwimage.grab_test_image_fpath()
>>> node0 = DelayedLoad(fpath, channels='r|g|b').prepare()
>>> node1 = node0.warp({'scale': 0.432,
>>>                     'theta': np.pi / 3,
>>>                     'about': (80, 80),
>>>                     'shearx': .3,
>>>                     'offset': (-50, -50)})
>>> node2 = node1[10:50, 1:40]
>>> self = node2
>>> new_outer = node2._opt_warp_after_crop()
>>> print(ub.urepr(node2.nesting(), nl=-1, sort=0))
>>> print(ub.urepr(new_outer.nesting(), nl=-1, sort=0))
>>> final0 = self._finalize()
>>> final1 = new_outer._finalize()
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(final0, pnum=(2, 2, 1), fnum=1, title='raw')
>>> kwplot.imshow(final1, pnum=(2, 2, 2), fnum=1, title='optimized')
```



Example

```
>>> # xdoctest: +REQUIRES(module:osgeo)
>>> from delayed_image import *  # NOQA
>>> from delayed_image.delayed_leafs import DelayedLoad
>>> fpath = kwimage.grab_test_image_fpath(overviews=3)
>>> node0 = DelayedLoad(fpath, channels='r|g|b').prepare()
>>> node1 = node0.warp({'scale': 1000 / 512})
>>> node2 = node1[250:750, 0:500]
>>> self = node2
>>> new_outer = node2._opt_warp_after_crop()
>>> print(ub.urepr(node2.nesting(), nl=-1, sort=0))
>>> print(ub.urepr(new_outer.nesting(), nl=-1, sort=0))
```

_opt_dequant_after_crop()

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

Bases: *DelayedImage*

Rescales image intensities from int to floats.

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

Parameters

- **subdata** (*DelayedArray*) – data to operate on
- **quantization** (*Dict*) – see `delayed_image.helpers.dequantize()`

_finalize()

Returns
ArrayLike

optimize()

Returns
DelayedImage

Example

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

_opt_dequant_before_other()

_transform_from_subdata()

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

Bases: *DelayedStack*

Stacks multiple arrays together.

Parameters
parts (*List[DelayedArray]*) – data to stack

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

Bases: *DelayedImageLeaf*

Returns an ndarray as-is

Example

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

Example

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

_finalize()

Returns
ArrayLike

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

Bases: `ImageOpsMixin`, `DelayedArray`

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

Parameters

- **subdata** (*DelayedArray*)
- **dsize** (*None* | *Tuple[int* | *None, int* | *None]*) – overrides subdata dsize
- **channels** (*None* | *int* | *FusedChannelSpec*) – overrides subdata channels

property shape

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

property num_channels

Returns: None | int

property dsize

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

property channels

Returns: None | FusedChannelSpec

property num_overviews

Returns: int

take_channels(*channels*)

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

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

Returns
a new delayed load with a fused take channel operation

Return type
DelayedCrop

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

Example

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

Example

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

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

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

Example

```
>>> from delayed_image.delayed_nodes import *  # NOQA
>>> from delayed_image import DelayedLoad
>>> self = DelayedLoad.demo(dsize=(16, 16), channels='r|g|b').prepare()
>>> # Case where a channel doesn't exist
>>> channels = 'r|b|magic'
>>> new = self.take_channels(channels)
>>> assert len(new.parts) == 2
>>> new._validate()
```

_validate()

Check that the delayed metadata corresponds with the finalized data

_transform_from_subdata()

get_transform_from_leaf()

Returns the transformation that would align data with the leaf

evaluate()

Evaluate this node and return the data as an identity.

Returns
DelayedIdentity

_opt_push_under_concat()

Push this node under its child node if it is a concatenation operation

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

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

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

Parameters

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

SeeAlso:
DelayedChannelConcat.undo_warps

Example

```
>>> # Test similar to undo_warps, but on each channel separately
>>> from delayed_image.delayed_nodes import *  # NOQA
>>> from delayed_image.delayed_leafs import DelayedLoad
>>> from delayed_image.delayed_leafs import DelayedNans
>>> import ubelt as ub
>>> import kwimage
>>> import kwarray
>>> import numpy as np
>>> # Demo case where we have different channels at different resolutions
>>> base = DelayedLoad.demo(channels='r|g|b').prepare().dequantize({'quant_max
↪': 255})
>>> bandR = base[:, :, 0].scale(100 / 512)[:, :-50].evaluate()
>>> bandG = base[:, :, 1].scale(300 / 512).warp({'theta': np.pi / 8, 'about':␣
↪(150, 150)}).evaluate()
>>> bandB = base[:, :, 2].scale(600 / 512)[:150, :].evaluate()
>>> bandN = DelayedNans((600, 600), channels='N')
>>> B0 = bandR.scale(6, dsize=(600, 600)).optimize()
```

(continues on next page)

(continued from previous page)

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

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

Bases: `DelayedImage`

Parameters

- **subdata** (*DelayedArray*)
- **dsize** (*None | Tuple[int | None, int | None]*) – overrides subdata dsize
- **channels** (*None | int | FusedChannelSpec*) – overrides subdata channels

get_transform_from_leaf()

Returns the transformation that would align data with the leaf

Returns
kwimage.Affine

optimize()

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

Bases: `DelayedImageLeaf`

Points to an image on disk to be loaded.

This is the starting point for most delayed operations. Disk IO is avoided until the `finalize` operation is called. Calling `prepare` can read image headers if metadata like the image width, height, and number of channels is not provided, but most operations can be performed while these are still unknown.

If a gdal backend is available, and the underlying image is in the appropriate formate (e.g. COG), finalize will return a lazy reference that enables fast overviews and crops. For image formats that do not allow for tiling / overviews, then there is no way to avoid reading entire image as an ndarray.

Example

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

Example

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

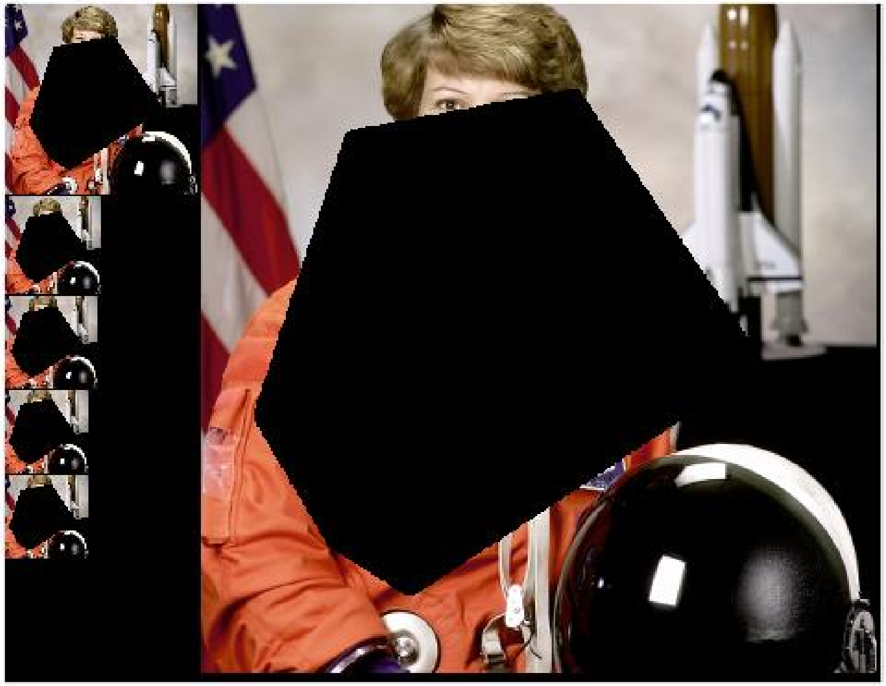
Example

```
>>> # xdoctest: +REQUIRES(module:osgeo)
>>> # Test delayed ops with int16 and nodata values
>>> from delayed_image import *  # NOQA
>>> import kwimage
>>> from delayed_image.helpers import quantize_float01
>>> import ubelt as ub
>>> dpath = ub.Path.appdir('delayed_image/tests/test_delay_nodata').ensuredir()
>>> fpath = dpath / 'data.tif'
>>> data = kwimage.ensure_float01(kwimage.grab_test_image())
>>> poly = kwimage.Polygon.random(rng=321032).scale(data.shape[0])
>>> poly.fill(data, np.nan)
>>> data_uint16, quantization = quantize_float01(data)
>>> nodata = quantization['nodata']
>>> kwimage.imwrite(fpath, data_uint16, nodata=nodata, backend='gdal', overviews=3)
```

(continues on next page)

(continued from previous page)

```
>>> # Test loading the data
>>> self = DelayedLoad(fpath, channels='r|g|b', nodata_method='float').prepare()
>>> node0 = self
>>> node1 = node0.dequantize(quantization)
>>> node2 = node1.warp({'scale': 0.51}, interpolation='lanczos')
>>> node3 = node2[13:900, 11:700]
>>> node4 = node3.warp({'scale': 0.9}, interpolation='lanczos')
>>> node4.write_network_text()
>>> node5 = node4.optimize()
>>> node5.write_network_text()
>>> node6 = node5.warp({'scale': 8}, interpolation='lanczos').optimize()
>>> node6.write_network_text()
>>> #
>>> data0 = node0._validate().finalize()
>>> data1 = node1._validate().finalize()
>>> data2 = node2._validate().finalize()
>>> data3 = node3._validate().finalize()
>>> data4 = node4._validate().finalize()
>>> data5 = node5._validate().finalize()
>>> data6 = node6._validate().finalize()
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> stack1 = kwimage.stack_images([data1, data2, data3, data4, data5])
>>> stack2 = kwimage.stack_images([stack1, data6], axis=1)
>>> kwplot.imshow(stack2)
```



Parameters

- **fpath** (*str* | *PathLike*) – URI pointing at the image data to load
- **channels** (*int* | *str* | *FusedChannelSpec* | *None*) – the underlying channels of the image if known a-priori
- **dsize** (*Tuple[int, int]*) – The width / height of the image if known a-priori
- **nodata_method** (*str* | *None*) – How to handle nodata values in the file itself. Can be "auto", "float", or "ma".

`property fpath`

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

Creates a demo DelayedLoad node that points to a file generated by `kwimage.grab_test_image_fpath()`.

If metadata like dsize and channels are not provided, then the *prepare()* can be used to auto-populate them at the cost of the disk IO to read image headers.

Parameters

- **key** (*str*) – which test image to grab. Valid choices are: astro - an astronaught carl - Carl Sagan paraview - ParaView logo stars - picture of stars in the sky
- **channels** (*str*) – if specified, these channels will be stored in the delayed load metadata. Note: these are not auto-populated. Usually the key corresponds to 3-channel data,
- **dsize** (*None* | *Tuple[int, int]*) – if specified, we will return a variant of the data with the specific dsize

- **nodata_method** (*str | None*) – How to handle nodata values in the file itself. Can be "auto", "float", or "ma".
- **overviews** (*None | int*) – if specified, will return a variant of the data with overviews

Returns
DelayedLoad

Example

```
>>> from delayed_image.delayed_leafs import *  # NOQA
>>> import delayed_image
>>> delayed = delayed_image.DelayedLoad.demo()
>>> print(f'delayed={delayed}')
>>> delayed.prepare()
>>> print(f'delayed={delayed}')
>>> delayed = DelayedLoad.demo(channels='r|g|b', nodata_method='float')
>>> print(f'delayed={delayed}')
>>> delayed.prepare()
>>> print(f'delayed={delayed}')
>>> delayed.finalize()
```

_load_reference()

prepare()

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

Returns
DelayedLoad

_load_metadata()

_finalize()

Returns
ArrayLike

Example

```
>>> # Check difference between finalize and _finalize
>>> from delayed_image.delayed_leafs import *  # NOQA
>>> self = DelayedLoad.demo().prepare()
>>> final_arr = self.finalize()
>>> assert isinstance(final_arr, np.ndarray), 'finalize should always return an␣
↪array'
>>> final_ref = self._finalize()
>>> if self.lazy_ref is not NotImplemented:
>>>     assert not isinstance(final_ref, np.ndarray), (
>>>         'A pure load with gdal should return a reference that is '
>>>         'similiar to but not quite an array')
```

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

Bases: *DelayedImageLeaf*

Constructs nan channels as needed

Example

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

Example

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

_finalize()

Returns
ArrayLike

_optimized_crop(*space_slice=None*, *chan_idxs=None*)

Crops an image along integer pixel coordinates.

Parameters

- **space_slice** (*Tuple[slice, slice]*) – y-slice and x-slice.
- **chan_idxs** (*List[int]*) – indexes of bands to take

Returns
DelayedImage

_optimized_warp(*transform*, *dsize=None*, ***warp_kwargs*)

Returns
DelayedImage

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

Bases: *DelayedOperation*

For operations that have multiple input arrays

children()

Yields
Any

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

Bases: NiceRepr

nesting()

Returns
Dict[str, dict]

as_graph(*fields='auto'*)

Builds the underlying graph structure as a networkx graph with human readable labels.

Parameters
fields (*str* | *List[str]*) – Add the specified fields as labels. If 'auto' then does somthing "reasonable". If 'all' then shows everything. TODO: only implemented for "auto" and "all", implement general field selection (PR Wanted).

Returns
networkx.DiGraph

_traverse()

A flat list of all descendent nodes and their parents

Yields
Tuple[None | *DelayedOperation, DelayedOperation]* – tules of parent / child nodes. Discarding the parents will be a list of all nodes.

leafs()

Iterates over all leafs in the tree.

Yields
Tuple[DelayedOperation]

_leafs()

Iterates over all leafs in the tree.

Yields
Tuple[DelayedOperation]

_leaf_paths()

Builds all independent paths to leafs.

Yields
Tuple[DelayedOperation, DelayedOperation] – The leaf, and the path to it,

Example

```
>>> from delayed_image import demo
>>> self = demo.non_aligned_leafs()
>>> for leaf, part in list(self._leaf_paths()):
...     leaf.write_network_text()
...     part.write_network_text()
```

Example

```
>>> from delayed_image import demo
>>> import delayed_image
>>> orig = delayed_image.DelayedLoad.demo().prepare()
>>> part1 = orig[0:100, 0:100].scale(2, dsize=(128, 128))
>>> part2 = delayed_image.DelayedNans(dsize=(128, 128))
>>> self = delayed_image.DelayedChannelConcat([part2, part1])
>>> for leaf, part in list(self._leaf_paths()):
...     leaf.write_network_text()
...     part.write_network_text()
```

_traversed_graph()

A flat list of all descendent nodes and their parents

print_graph(*fields='auto'*, *with_labels=True*, *rich='auto'*, *vertical_chains=True*)

Alias for write_network_text

Parameters

- **fields** (*str | List[str]*) – Add the specified fields as labels. If 'auto' then does somthing "reasonable". If 'all' then shows everything. TODO: only implemented for "auto" and "all", implement general field selection (PR Wanted).
- **with_labels** (*bool*) – set to false for no label data
- **rich** (*bool | str*) – defaults to 'auto'
- **vertical_chains** (*bool*) – Defaults to True. Set to false to save vertical space at the cost of horizontal space.

write_network_text(*fields='auto'*, *with_labels=True*, *rich='auto'*, *vertical_chains=True*)

Alias for `DelayedOperation.print_graph()`

property shape

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

children()

Yields

Any

prepare()

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

Returns

DelayedOperation

_finalize()

This is the method that new nodes should overload.

Conceptually this works just like the finalize method with the exception that it happens at every node in the tree, whereas the public facing method only happens once, calls this, and is able to do one-time pre and post operations.

Returns

ArrayLike

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

Evaluate the operation tree in full.

Parameters

- **prepare** (*bool*) – ensure prepare is called to ensure metadata exists if possible before optimizing. Defaults to True.
- **optimize** (*bool*) – ensure the graph is optimized before loading. Default to True.
- ****kwargs** – for backwards compatibility, these will allow for in-place modification of select nested parameters.

Returns
ArrayLike

Notes

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

optimize()

Returns
DelayedOperation

_set_nested_params(***kwargs*)

Hack to override nested params on all warps for things like interplation / antialias

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

Bases: `DelayedImage`

Downsamples an image by a factor of two.

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

Example

```
>>> # xdoctest: +REQUIRES(module:osgeo)
>>> # Make a complex chain of operations and optimize it
>>> from delayed_image import *  # NOQA
>>> import kwimage
>>> fpath = kwimage.grab_test_image_fpath(overviews=3)
>>> dimg = DelayedLoad(fpath, channels='r|g|b').prepare()
>>> dimg = dimg.get_overview(1)
>>> dimg = dimg.get_overview(1)
>>> dimg = dimg.get_overview(1)
>>> dopt = dimg.optimize()
>>> if 1:
>>>     import networkx as nx
>>>     dimg.write_network_text()
>>>     dopt.write_network_text()
>>> print(ub.urepr(dopt.nesting(), nl=-1, sort=0))
>>> final0 = dimg._finalize()[:]
>>> final1 = dopt._finalize()[:]
>>> assert final0.shape == final1.shape
```

(continues on next page)

(continued from previous page)

```
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(final0, pnum=(1, 2, 1), fnum=1, title='raw')
>>> kwplot.imshow(final1, pnum=(1, 2, 2), fnum=1, title='optimized')
```

Parameters

- **subdata** (*DelayedArray*) – data to operate on
- **overview** (*int*) – the overview to use (assuming it exists)

property num_overviews

Returns: int

_finalize()

Returns
ArrayLike

optimize()

Returns
DelayedImage

_transform_from_subdata()

_opt_overview_as_warp()

Sometimes it is beneficial to replace an overview with a warp as an intermediate optimization step.

_opt_crop_after_overview()

Given an outer overview and an inner crop, switch places. We want the overview to be as close to the load as possible.

Example

```
>>> # xdoctest: +REQUIRES(module:osgeo)
>>> from delayed_image import *  # NOQA
>>> fpath = kwimage.grab_test_image_fpath(overviews=3)
>>> node0 = DelayedLoad(fpath, channels='r|g|b').prepare()
>>> node1 = node0[100:400, 120:450]
>>> node2 = node1.get_overview(2)
>>> self = node2
>>> new_outer = node2.optimize()
>>> print(ub.urepr(node2.nesting(), nl=-1, sort=0))
>>> print(ub.urepr(new_outer.nesting(), nl=-1, sort=0))
>>> final0 = self._finalize()
>>> final1 = new_outer._finalize()
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(final0, pnum=(1, 2, 1), fnum=1, title='raw')
>>> kwplot.imshow(final1, pnum=(1, 2, 2), fnum=1, title='optimized')
```

Example

```
>>> # xdoctest: +REQUIRES(module:osgeo)
>>> from delayed_image import *  # NOQA
>>> fpath = kwimage.grab_test_image_fpath(overviews=3)
>>> node0 = DelayedLoad(fpath, channels='r|g|b').prepare()
>>> node1 = node0[:, :, 0:2]
>>> node2 = node1.get_overview(2)
>>> self = node2
>>> new_outer = node2.optimize()
>>> node2.write_network_text()
>>> new_outer.write_network_text()
>>> assert node2.shape[2] == 2
>>> assert new_outer.shape[2] == 2
```

_opt_fuse_overview()

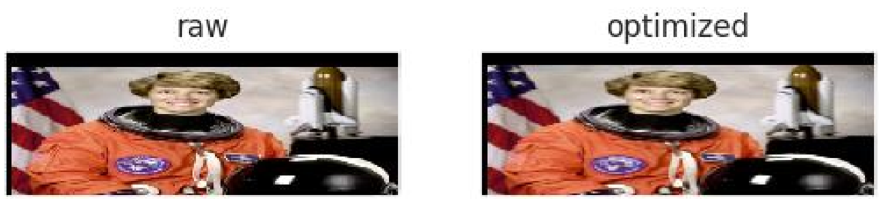
_opt_dequant_after_overview()

_opt_warp_after_overview()

Given an warp followed by an overview, move the warp to the outer scope such that the overview is first.

Example

```
>>> # xdoctest: +REQUIRES(module:osgeo)
>>> from delayed_image import *  # NOQA
>>> fpath = kwimage.grab_test_image_fpath(overviews=3)
>>> node0 = DelayedLoad(fpath, channels='r|g|b').prepare()
>>> node1 = node0.warp({'scale': (2.1, .7), 'offset': (20, 40)})
>>> node2 = node1.get_overview(2)
>>> self = node2
>>> new_outer = node2.optimize()
>>> print(ub.urepr(node2.nesting(), nl=-1, sort=0))
>>> print(ub.urepr(new_outer.nesting(), nl=-1, sort=0))
>>> final0 = self._finalize()
>>> final1 = new_outer._finalize()
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(final0, pnum=(1, 2, 1), fnum=1, title='raw')
>>> kwplot.imshow(final1, pnum=(1, 2, 2), fnum=1, title='optimized')
```



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

Bases: *DelayedNaryOperation*

Stacks multiple arrays together.

Parameters

- **parts** (*List[DelayedArray]*) – data to stack
- **axis** (*int*) – axes to stack on

property shape

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

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

Bases: *DelayedOperation*

For operations that have a single input array

children()

Yields
Any

class kwcoco.util.delayed_ops.**DelayedWarp**(*subdata*, *transform*, *dsize='auto'*, *antialias=True*, *interpolation='linear'*, *border_value='auto'*, *noop_eps=0*)

Bases: *DelayedImage*

Applies an affine transform to an image.

Example

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

Parameters

- **subdata** (*DelayedArray*) – data to operate on
- **transform** (*ndarray | dict | kwimage.Affine*) – a coercable affine matrix. See `kwimage.Affine` for details on what can be coerced.
- **dsize** (*Tuple[int, int] | str*) – The width / height of the output canvas. If 'auto', dsize is computed such that the positive coordinates of the warped image will fit in the new canvas. In this case, any pixel that maps to a negative coordinate will be clipped. This has the property that the input transformation is not modified.
- **antialias** (*bool*) – if True determines if the transform is downsampling and applies antialiasing via gaussian a blur. Defaults to False
- **interpolation** (*str*) – interpolation code or cv2 integer. Interpolation codes are linear, nearest, cubic, lancsoz, and area. Defaults to "linear".
- **noop_eps** (*float*) – This is the tolerance for optimizing a warp away. If the transform has all of its decomposed parameters (i.e. scale, rotation, translation, shear) less than this value, the warp node can be optimized away. Defaults to 0.

property transform

Returns: kwimage.Affine

_finalize()

Returns
ArrayLike

optimize()

Returns
DelayedImage

Example

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

Example

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

Example

```
>>> # Test optimize nans
>>> from delayed_image import DelayedLoad
>>> import kwimage
>>> base = DelayedLoad.demo(channels='r|g|b').prepare()
>>> transform = kwimage.Affine.scale(1.0 + 1e-7)
>>> self = base.warp(transform, dsize=base.dsize)
>>> # An optimize will not remove a warp if there is any
>>> # doubt if it is the identity.
>>> new = self.optimize()
>>> assert len(self.as_graph().nodes) == 2
>>> assert len(new.as_graph().nodes) == 2
>>> # But we can specify a threshold where it will
>>> self._set_nested_params(noop_eps=1e-6)
>>> new = self.optimize()
>>> assert len(self.as_graph().nodes) == 2
>>> assert len(new.as_graph().nodes) == 1
```

_transform_from_subdata()

_opt_fuse_warps()

Combine two consecutive warps into a single operation.

_opt_absorb_overview()

Remove any deeper overviews that would be undone by this warp.

Given this warp node, if it has a scale component could undo an overview (i.e. the scale factor is greater than 2), we want to:

1. determine if there is an overview deeper in the tree.
2. remove that overview and that scale factor from this warp
3. modify any intermediate nodes that will be changed by having the deeper overview removed.

Note: This optimization is currently the most dubious one in the code, and is likely where some of the bugs are coming from. Help wanted.

CommandLine

```
xdoctest -m delayed_image.delayed_nodes DelayedWarp._opt_absorb_overview
```

Example

```
>>> # xdoctest: +REQUIRES(module:osgeo)
>>> from delayed_image.delayed_nodes import *  # NOQA
>>> from delayed_image import DelayedLoad
>>> import kwimage
>>> fpath = kwimage.grab_test_image_fpath(overviews=3)
>>> base = DelayedLoad(fpath, channels='r|g|b').prepare()
>>> # Case without any operations between the overview and warp
>>> self = base.get_overview(1).warp({'scale': 4})
>>> self.write_network_text()
>>> opt = self._opt_absorb_overview()._validate()
>>> opt.write_network_text()
>>> opt_data = [d for n, d in opt.as_graph().nodes(data=True)]
>>> assert 'DelayedOverview' not in [d['type'] for d in opt_data]
>>> # Case with a chain of operations between overview and warp
>>> self = base.get_overview(1)[0:101, 0:100].warp({'scale': 4})
>>> self.write_network_text()
>>> opt = self._opt_absorb_overview()._validate()
>>> opt.write_network_text()
>>> opt_data = [d for n, d in opt.as_graph().nodes(data=True)]
>>> #assert opt_data[1]['meta']['space_slice'] == (slice(0, 202, None), slice(0,␣
↪200, None))
>>> assert opt_data[1]['meta']['space_slice'] == (slice(0, 204, None), slice(0,␣
↪202, None))
>>> # Any sort of complex chain does prevents this optimization
>>> # from running.
>>> self = base.get_overview(1)[0:101, 0:100][0:50, 0:50].warp({'scale': 4})
>>> opt = self._opt_absorb_overview()._validate()
>>> opt.write_network_text()
>>> opt_data = [d for n, d in opt.as_graph().nodes(data=True)]
>>> assert 'DelayedOverview' in [d['type'] for d in opt_data]
```

_opt_split_warp_overview()

Split this node into a warp and an overview if possible

Example

```
>>> # xdoctest: +REQUIRES(module:osgeo)
>>> from delayed_image.delayed_nodes import *  # NOQA
>>> from delayed_image import DelayedLoad
>>> import kwimage
>>> fpath = kwimage.grab_test_image_fpath(overviews=3)
>>> self = DelayedLoad(fpath, channels='r|g|b').prepare()
>>> print(f'self={self}')
>>> print('self.meta = {}'.format(ub.urepr(self.meta, nl=1)))
>>> warp0 = self.warp({'scale': 0.2})
>>> warp1 = warp0._opt_split_warp_overview()
>>> warp2 = self.warp({'scale': 0.25})._opt_split_warp_overview()
>>> print(ub.urepr(warp0.nesting(), nl=-1, sort=0))
>>> print(ub.urepr(warp1.nesting(), nl=-1, sort=0))
>>> print(ub.urepr(warp2.nesting(), nl=-1, sort=0))
>>> warp0_nodes = [d['type'] for d in warp0.as_graph().nodes.values()]
>>> warp1_nodes = [d['type'] for d in warp1.as_graph().nodes.values()]
>>> warp2_nodes = [d['type'] for d in warp2.as_graph().nodes.values()]
>>> assert warp0_nodes == ['DelayedWarp', 'DelayedLoad']
>>> assert warp1_nodes == ['DelayedWarp', 'DelayedOverview', 'DelayedLoad']
>>> assert warp2_nodes == ['DelayedOverview', 'DelayedLoad']
```

Example

```
>>> # xdoctest: +REQUIRES(module:osgeo)
>>> from delayed_image.delayed_nodes import *  # NOQA
>>> from delayed_image import DelayedLoad
>>> import kwimage
>>> fpath = kwimage.grab_test_image_fpath(overviews=3)
>>> self = DelayedLoad(fpath, channels='r|g|b').prepare()
>>> warp0 = self.warp({'scale': 1 / 2 ** 6})
>>> opt = warp0.optimize()
>>> print(ub.urepr(warp0.nesting(), nl=-1, sort=0))
>>> print(ub.urepr(opt.nesting(), nl=-1, sort=0))
>>> warp0_nodes = [d['type'] for d in warp0.as_graph().nodes.values()]
>>> opt_nodes = [d['type'] for d in opt.as_graph().nodes.values()]
>>> assert warp0_nodes == ['DelayedWarp', 'DelayedLoad']
>>> assert opt_nodes == ['DelayedWarp', 'DelayedOverview', 'DelayedLoad']
```

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

Bases: object

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

Crops an image along integer pixel coordinates.

Parameters

- **space_slice** (*Tuple[slice, slice]*) – y-slice and x-slice.
- **chan_idxs** (*List[int]*) – indexes of bands to take
- **clip** (*bool*) – if True, the slice is interpreted normally, where it won't go past the image extent, otherwise slicing into negative regions or past the image bounds will result in padding.

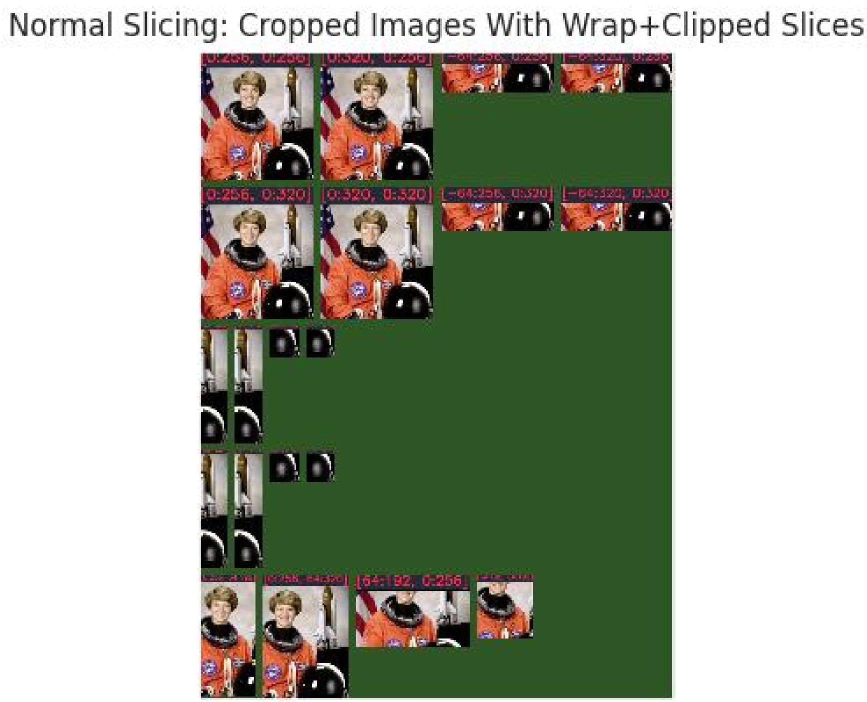
Defaults to True.

- **wrap** (*bool*) – if True, negative indexes "wrap around", otherwise they are treated as is. Defaults to True.
- **pad** (*int | List[Tuple[int, int]]*) – if specified, applies extra padding

Returns
DelayedImage

Example

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



Example

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

(continues on next page)

(continued from previous page)

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

_padded_crop(*space_slice*, *pad=0*)

Does the type of padded crop we want, but inefficiently using a warp. Reimplementing would be good, but this is good enough for now.

warp(*transform*, *dsize='auto'*, ***warp_kwargs*)

Applys an affine transformation to the image. See `DelayedWarp`.

Parameters

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

Returns

DelayedImage

scale(*scale, dsize='auto', **warp_kwargs*)

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

resize(*dsize, **warp_kwargs*)

Resize an image to a specific width/height by scaling it.

dequantize(*quantization*)

Rescales image intensities from int to floats.

Parameters

quantization (*Dict[str, Any]*) – quantization information dictionary to undo. see `delayed_image.helpers.dequantize()` Expected keys are: orig_dtype (str) orig_min (float) orig_max (float) quant_min (float) quant_max (float) nodata (None | int)

Returns

DelayedDequantize

Example

```
>>> from delayed_image.delayed_leafs import DelayedLoad
>>> self = DelayedLoad.demo().prepare()
>>> quantization = {
>>>     'orig_dtype': 'float32',
>>>     'orig_min': 0,
>>>     'orig_max': 1,
>>>     'quant_min': 0,
>>>     'quant_max': 255,
>>>     'nodata': None,
>>> }
>>> new = self.dequantize(quantization)
```

(continues on next page)

(continued from previous page)

```
>>> assert self.finalize().max() > 1
>>> assert new.finalize().max() <= 1
```

get_overview(*overview*)

Downsamples an image by a factor of two.

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

Returns
DelayedOverview

as_xarray()

Returns
DelayedAsXarray

get_transform_from(*src*)

Find a transform from a given node (src) to this node (self / dst).

Given two delayed images src and dst that share a common leaf, find the transform from src to dst.

Parameters
src (*DelayedOperation*) – the other view to get a transform to. This must share a leaf with self (which is the dst).

Returns
The transform that warps the space of src to the space of self.

Return type
kwimage.Affine

Example

```
>>> from delayed_image import *  # NOQA
>>> from delayed_image.delayed_leafs import DelayedLoad
>>> base = DelayedLoad.demo().prepare()
>>> src = base.scale(2)
>>> dst = src.warp({'scale': 4, 'offset': (3, 5)})
>>> transform = dst.get_transform_from(src)
>>> tf = transform.decompose()
>>> assert tf['scale'] == (4, 4)
>>> assert tf['offset'] == (3, 5)
```

Example

```
>>> from delayed_image import demo
>>> self = demo.non_aligned_leafs()
>>> leaf = list(self._leaf_paths())[0][0]
>>> tf1 = self.get_transform_from(leaf)
>>> tf2 = leaf.get_transform_from(self)
>>> np.allclose(np.linalg.inv(tf2), tf1)
```

2.1.1.6.2 Submodules

2.1.1.6.2.1 kwcoco.util.dict_like module

class kwcoco.util.dict_like.**DictLike**

Bases: NiceRepr

An inherited class must specify the getitem, setitem, and
keys methods.

A class is dictionary like if it has:

`__iter__`, `__len__`, `__contains__`, `__getitem__`, `items`, `keys`, `values`, `get`,

and if it should be writable it should have: `__delitem__`, `__setitem__`, `update`,

And perhaps: `copy`,

`__iter__`, `__len__`, `__contains__`, `__getitem__`, `items`, `keys`, `values`, `get`,

and if it should be writable it should have: `__delitem__`, `__setitem__`, `update`,

And perhaps: `copy`,

getitem(*key*)

Parameters
key (*Any*) – a key

Returns
a value

Return type
Any

setitem(*key*, *value*)

Parameters

- **key** (*Any*)
- **value** (*Any*)

delitem(*key*)

Parameters
key (*Any*)

keys()

Yields
Any – a key

items()

Yields
Tuple[Any, Any] – a key value pair

values()

Yields
Any – a value

copy()

Return type
Dict

to_dict()

Return type
Dict

asdict()

Return type
Dict

update(*other*)

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

Parameters

- **key** (*Any*)
- **default** (*Any*)

Return type
Any

class kwcoco.util.dict_like.**DictProxy**

Bases: `DictLike`

Allows an object to proxy the behavior of a dict attribute

keys()

2.1.1.6.2.2 kwcoco.util.dict_proxy2 module

class kwcoco.util.dict_proxy2.**DictInterface**

Bases: `object`

An inherited class must specify the `getitem`, `setitem`, and
`keys` methods.

A class is dictionary like if it has:

`__iter__`, `__len__`, `__contains__`, `__getitem__`, `items`, `keys`, `values`, `get`,

and if it should be writable it should have: `__delitem__`, `__setitem__`, `update`,

And perhaps: `copy`,

`__iter__`, `__len__`, `__contains__`, `__getitem__`, `items`, `keys`, `values`, `get`,

and if it should be writable it should have: `__delitem__`, `__setitem__`, `update`,

And perhaps: `copy`,

Example

from scriptconfig.dict_like import DictLike class DuckDict(DictLike):

def __init__(self, _data=None):

if _data is None:
_data = {}

self._data = _data

def getitem(self, key):
return self._data[key]

def keys(self):
return self._data.keys()

self = DuckDict({1: 2, 3: 4}) print(f'self._data={self._data}') cast = dict(self) print(f'cast={cast}') print(f'self={self}')

keys()

Yields
str

items()

Yields
Tuple[Any, Any] – a key value pair

values()

Yields
Any – a value

update(*other*)

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

Parameters

- **key** (*Any*)
- **default** (*Any*)

Return type
Any

class kwcoco.util.dict_proxy2.**DictProxy2**

Bases: *DictInterface*

Allows an object to proxy the behavior of a _proxy dict attribute

keys()

class kwcoco.util.dict_proxy2.**_AliasMetaclass**(*name*, *bases*, *namespace*, **args*, ***kwargs*)

Bases: type

Populates the __alias_to_aliases__ field at class definition time to reduce the overhead of instance creation.

class kwcoco.util.dict_proxy2.**AliasedDictProxy**

Bases: *DictProxy2*

Can have a class attribute called ``__alias_to_primary__`` which is a Dict[str, str] mapping alias-keys to primary-keys.

Need to handle cases:

- **image dictionary contains no primary / aliased keys**
  - primary keys used
- **image dictionary only has aliased keys**
  - aliased keys are updated
- **image dictionary only has primary keys**
  - primary keys are updated
- **image dictionary only both primary and aliased keys**
  - both keys are updated

Example

```
>>> from kwcoco.util.dict_proxy2 import *  # NOQA
>>> class MyAliasedObject(AliasedDictProxy):
>>>     __alias_to_primary__ = {
>>>         'foo_alias1': 'foo_primary',
>>>         'foo_alias2': 'foo_primary',
>>>         'bar_alias1': 'bar_primary',
>>>     }
>>>     def __init__(self, obj):
>>>         self._proxy = obj
>>>     def __repr__(self):
>>>         return repr(self._proxy)
>>>     def __str__(self):
>>>         return str(self._proxy)
>>> # Test starting from empty
>>> obj = MyAliasedObject({})
>>> obj['regular_key'] = 'val0'
>>> assert 'foo_primary' not in obj
>>> assert 'foo_alias1' not in obj
>>> assert 'foo_alias2' not in obj
>>> obj['foo_primary'] = 'val1'
>>> assert 'foo_primary' in obj
>>> assert 'foo_alias1' in obj
>>> assert 'foo_alias2' in obj
>>> obj['foo_alias1'] = 'val2'
>>> obj['foo_alias2'] = 'val3'
>>> obj['bar_alias1'] = 'val4'
>>> obj['bar_primary'] = 'val5'
>>> assert obj._proxy == {
>>>     'regular_key': 'val0',
>>>     'foo_primary': 'val3',
>>>     'bar_primary': 'val5'}
```

(continues on next page)

(continued from previous page)

```
>>> # Test starting with primary keys
>>> obj = MyAliasedObject({
>>>     'foo_primary': 123,
>>>     'bar_primary': 123,
>>> })
>>> assert 'foo_alias1' in obj
>>> assert 'bar_alias1' in obj
>>> obj['bar_alias1'] = 456
>>> obj['foo_primary'] = 789
>>> assert obj._proxy == {
>>>     'foo_primary': 789,
>>>     'bar_primary': 456}
>>> # Test that if aliases keys are existant we dont add primary keys
>>> obj = MyAliasedObject({
>>>     'foo_alias1': 123,
>>> })
>>> assert 'foo_alias1' in obj
>>> assert 'foo_primary' in obj
>>> obj['foo_alias1'] = 456
>>> obj['foo_primary'] = 789
>>> assert obj._proxy == {
>>>     'foo_alias1': 789,
>>> }
>>> # Test that if primary and aliases keys exist, we update both
>>> obj = MyAliasedObject({
>>>     'foo_primary': 3,
>>>     'foo_alias2': 5,
>>> })
>>> # We do not attempt to detect conflicts
>>> assert obj['foo_primary'] == 3
>>> assert obj['foo_alias1'] == 3
>>> assert obj['foo_alias2'] == 5
>>> obj['foo_alias1'] = 23
>>> assert obj['foo_primary'] == 23
>>> assert obj['foo_alias1'] == 23
>>> assert obj['foo_alias2'] == 23
>>> obj['foo_primary'] = -12
>>> assert obj['foo_primary'] == -12
>>> assert obj['foo_alias1'] == -12
>>> assert obj['foo_alias2'] == -12
>>> assert obj._proxy == {
>>>     'foo_primary': -12,
>>>     'foo_alias2': -12}
```

keys()

2.1.1.6.2.3 kwcoco.util.jsonschema_elements module

Functional interface into defining jsonschema structures.

See mixin classes for details.

Perhaps [Voluptuous] does this better and we should switch to that?

References

Example

```
>>> from kwcoco.util.jsonschema_elements import *  # NOQA
>>> elem = SchemaElements()
>>> for base in SchemaElements.__bases__:
>>>     print('\n\n====\nbase = {!r}'.format(base))
>>>     attrs = [key for key in dir(base) if not key.startswith('_')]
>>>     for key in attrs:
>>>         value = getattr(elem, key)
>>>         print('{} = {}'.format(key, value))
```

class kwcoco.util.jsonschema_elements.**Element**(*base*, *options={}*, *_magic=None*)

Bases: dict

A dictionary used to define an element of a JSON Schema.

The exact keys/values for the element will depend on the type of element being described. The *SchemaElements* defines exactly what these are for the core elements. (e.g. OBJECT, INTEGER, NULL, ARRAY, ANYOF)

Example

```
>>> from kwcoco.coco_schema import *  # NOQA
>>> self = Element(base={'type': 'demo'}, options={'opt1', 'opt2'})
>>> new = self(opt1=3)
>>> print('self = {}'.format(ub.urepr(self, nl=1, sort=1)))
>>> print('new = {}'.format(ub.urepr(new, nl=1, sort=1)))
>>> print('new2 = {}'.format(ub.urepr(new(), nl=1, sort=1)))
>>> print('new3 = {}'.format(ub.urepr(new(title='myvar'), nl=1, sort=1)))
>>> print('new4 = {}'.format(ub.urepr(new(title='myvar')(examples=['']), nl=1,
↪sort=1)))
>>> print('new5 = {}'.format(ub.urepr(new(badattr=True), nl=1, sort=1)))
self = {
    'type': 'demo',
}
new = {
    'opt1': 3,
    'type': 'demo',
}
new2 = {
    'opt1': 3,
    'type': 'demo',
}
new3 = {
```

(continues on next page)

(continued from previous page)

```
    'opt1': 3,
    'title': 'myvar',
    'type': 'demo',
}
new4 = {
    'examples': [''],
    'opt1': 3,
    'title': 'myvar',
    'type': 'demo',
}
new5 = {
    'opt1': 3,
    'type': 'demo',
}
```

Parameters

- **base** (*dict*) – the keys / values this schema must contain
- **options** (*dict*) – the keys / values this schema may contain
- **_magic** (*callable | None*) – called when creating an instance of this schema element. Allows convinience attributes to be converted to the formal jsonschema specs. TODO: _magic is a terrible name, we need to rename it with something descriptive.

`validate`(*instance=NoParam*)

If `instance` is given, validates that that dictionary conforms to this schema. Otherwise validates that this is a valid schema element.

Parameters

instance (*dict*) – a dictionary to validate

`class kwcoco.util.jsonschema_elements.ScalarElements`

Bases: `object`

Single-valued elements

`property NULL`

//json-schema.org/understanding-json-schema/reference/null.html

Type

https

`property BOOLEAN`

//json-schema.org/understanding-json-schema/reference/null.html

Type

https

`property STRING`

//json-schema.org/understanding-json-schema/reference/string.html

Type

https

`property NUMBER`

//json-schema.org/understanding-json-schema/reference/numeric.html#number

Type
https

property INTEGER
//json-schema.org/understanding-json-schema/reference/numeric.html#integer

Type
https

class kwcoco.util.jsonschema_elements.QuantifierElements

Bases: object

Quantifier types

https://json-schema.org/understanding-json-schema/reference/combining.html#allof

Example

```
>>> from kwcoco.util.jsonschema_elements import *  # NOQA
>>> elem.ANYOF(elem.STRING, elem.NUMBER).validate()
>>> elem.ONEOF(elem.STRING, elem.NUMBER).validate()
>>> elem.NOT(elem.NULL).validate()
>>> elem.NOT(elem.ANY).validate()
>>> elem.ANY.validate()
```

property ANY

ALLOF(**TYPES*)

ANYOF(**TYPES*)

ONEOF(**TYPES*)

NOT(*TYPE*)

class kwcoco.util.jsonschema_elements.ContainerElements

Bases: object

Types that contain other types

Example

```
>>> from kwcoco.util.jsonschema_elements import *  # NOQA
>>> print(elem.ARRAY().validate())
>>> print(elem.OBJECT().validate())
>>> print(elem.OBJECT().validate())
{'type': 'array', 'items': {}}
{'type': 'object', 'properties': {}}
{'type': 'object', 'properties': {}}
```

ARRAY(*TYPE={}*, ***kw*)
https://json-schema.org/understanding-json-schema/reference/array.html

Example

```
>>> from kwcoco.util.jsonschema_elements import *  # NOQA
>>> ARRAY(numItems=3)
>>> schema = ARRAY(minItems=3)
>>> schema.validate()
{'type': 'array', 'items': {}, 'minItems': 3}
```

OBJECT(*PROPERTIES={}*, ***kw*)

https://json-schema.org/understanding-json-schema/reference/object.html

Example

```
>>> import jsonschema
>>> schema = elem.OBJECT()
>>> jsonschema.validate({}, schema)
>>> #
>>> import jsonschema
>>> schema = elem.OBJECT({
>>>     'key1': elem.ANY(),
>>>     'key2': elem.ANY(),
>>> }, required=['key1'])
>>> jsonschema.validate({'key1': None}, schema)
>>> #
>>> import jsonschema
>>> schema = elem.OBJECT({
>>>     'key1': elem.OBJECT({'arr': elem.ARRAY()}),
>>>     'key2': elem.ANY(),
>>> }, required=['key1'], title='a title')
>>> schema.validate()
>>> print('schema = {}'.format(ub.urepr(schema, sort=1, nl=-1)))
>>> jsonschema.validate({'key1': {'arr': []}}, schema)
schema = {
    'properties': {
        'key1': {
            'properties': {
                'arr': {'items': {}, 'type': 'array'}
            },
            'type': 'object'
        },
        'key2': {}
    },
    'required': ['key1'],
    'title': 'a title',
    'type': 'object'
}
```

class kwcoco.util.jsonschema_elements.**SchemaElements**

Bases: *ScalarElements*, *QuantifierElements*, *ContainerElements*

Functional interface into defining jsonschema structures.

See mixin classes for details.

References

https://json-schema.org/understanding-json-schema/

Todo:

- [ ] Generics: title, description, default, examples

CommandLine

```
xdoctest -m /home/joncrall/code/kwcoco/kwcoco/util/jsonschema_elements.py␣
↪SchemaElements
```

Example

```
>>> from kwcoco.util.jsonschema_elements import *  # NOQA
>>> elem = SchemaElements()
>>> elem.ARRAY(elem.ANY())
>>> schema = OBJECT({
>>>     'prop1': ARRAY(INTEGER, minItems=3),
>>>     'prop2': ARRAY(STRING, numItems=2),
>>>     'prop3': ARRAY(OBJECT({
>>>         'subprob1': NUMBER,
>>>         'subprob2': NUMBER,
>>>     }))
>>> })
>>> print('schema = {}'.format(ub.urepr(schema, nl=2, sort=1)))
schema = {
    'properties': {
        'prop1': {'items': {'type': 'integer'}, 'minItems': 3, 'type': 'array'},
        'prop2': {'items': {'type': 'string'}, 'maxItems': 2, 'minItems': 2, 'type
↪': 'array'},
        'prop3': {'items': {'properties': {'subprob1': {'type': 'number'}, 'subprob2
↪': {'type': 'number'}}, 'type': 'object'}, 'type': 'array'},
    },
    'type': 'object',
}
```

```
>>> TYPE = elem.OBJECT({
>>>     'p1': ANY,
>>>     'p2': ANY,
>>> }, required=['p1'])
>>> import jsonschema
>>> inst = {'p1': None}
>>> jsonschema.validate(inst, schema=TYPE)
>>> #jsonschema.validate({'p2': None}, schema=TYPE)
```

kwcoco.util.jsonschema_elements.**ALLOF**(**TYPES*)

kwcoco.util.jsonschema_elements.**ANYOF**(**TYPES*) kwcoco.util.jsonschema_elements.**ARRAY**(*TYPE={}*, ***kw*)

https://json-schema.org/understanding-json-schema/reference/array.html

Example

```
>>> from kwcoco.util.jsonschema_elements import *  # NOQA
>>> ARRAY(numItems=3)
>>> schema = ARRAY(minItems=3)
>>> schema.validate()
{'type': 'array', 'items': {}, 'minItems': 3}
```

kwcoco.util.jsonschema_elements.**NOT**(*TYPE*)

kwcoco.util.jsonschema_elements.**OBJECT**(*PROPERTIES={}*, ***kw*)

https://json-schema.org/understanding-json-schema/reference/object.html

Example

```
>>> import jsonschema
>>> schema = elem.OBJECT()
>>> jsonschema.validate({}, schema)
>>> #
>>> import jsonschema
>>> schema = elem.OBJECT({
>>>     'key1': elem.ANY(),
>>>     'key2': elem.ANY(),
>>> }, required=['key1'])
>>> jsonschema.validate({'key1': None}, schema)
>>> #
>>> import jsonschema
>>> schema = elem.OBJECT({
>>>     'key1': elem.OBJECT({'arr': elem.ARRAY()}),
>>>     'key2': elem.ANY(),
>>> }, required=['key1'], title='a title')
>>> schema.validate()
>>> print('schema = {}'.format(ub.urepr(schema, sort=1, nl=-1)))
>>> jsonschema.validate({'key1': {'arr': []}}, schema)
schema = {
    'properties': {
        'key1': {
            'properties': {
                'arr': {'items': {}, 'type': 'array'}
            },
            'type': 'object'
        },
        'key2': {}
    },
    'required': ['key1'],
    'title': 'a title',
    'type': 'object'
}
```

kwcoco.util.jsonschema_elements.**ONEOF**(**TYPES*)

2.1.1.6.2.4 kwcoco.util.lazy_frame_backends module

Ported to delayed_image

2.1.1.6.2.5 kwcoco.util.util_archive module

class kwcoco.util.util_archive.**Archive**(*fpath=None*, *mode='r'*, *backend=None*, *file=None*)

Bases: object

Abstraction over zipfile and tarfile

Todo: see if we can use one of these other tools instead

SeeAlso:

https://github.com/RKrahl/archive-tools https://pypi.org/project/arlib/

Example

```
>>> from kwcoco.util.util_archive import Archive
>>> import ubelt as ub
>>> dpath = ub.Path.appdir('kwcoco', 'tests', 'util', 'archive')
>>> dpath.delete().ensuredir()
>>> # Test write mode
>>> mode = 'w'
>>> arc_zip = Archive(str(dpath / 'demo.zip'), mode=mode)
>>> arc_tar = Archive(str(dpath / 'demo.tar.gz'), mode=mode)
>>> open(dpath / 'data_1only.txt', 'w').write('bazbzzz')
>>> open(dpath / 'data_2only.txt', 'w').write('buzzz')
>>> open(dpath / 'data_both.txt', 'w').write('foobar')
>>> #
>>> arc_zip.add(dpath / 'data_both.txt')
>>> arc_zip.add(dpath / 'data_1only.txt')
>>> #
>>> arc_tar.add(dpath / 'data_both.txt')
>>> arc_tar.add(dpath / 'data_2only.txt')
>>> #
>>> arc_zip.close()
>>> arc_tar.close()
>>> #
>>> # Test read mode
>>> arc_zip = Archive(str(dpath / 'demo.zip'), mode='r')
>>> arc_tar = Archive(str(dpath / 'demo.tar.gz'), mode='r')
>>> # Test names
>>> name = 'data_both.txt'
>>> assert name in arc_zip.names()
>>> assert name in arc_tar.names()
>>> # Test read
```

(continues on next page)

(continued from previous page)

```
>>> assert arc_zip.read(name, mode='r') == 'foobar'
>>> assert arc_tar.read(name, mode='r') == 'foobar'
>>> #
>>> # Test extractall
>>> extract_dpath = ub.ensuredir(str(dpath / 'extracted'))
>>> extracted1 = arc_zip.extractall(extract_dpath)
>>> extracted2 = arc_tar.extractall(extract_dpath)
>>> for fpath in extracted2:
>>>     print(open(fpath, 'r').read())
>>> for fpath in extracted1:
>>>     print(open(fpath, 'r').read())
```

Parameters

- **fpath** (*str | None*) – path to open
- **mode** (*str*) – either r or w
- **backend** (*str | ModuleType | None*) – either tarfile, zipfile string or module.
- **file** (*tarfile.TarFile | zipfile.ZipFile | None*) – the open backend file if it already exists. If not set, than fpath will open it.

_available_backends = {'tarfile': <module 'tarfile' from '/home/docs/.asdf/installs/python/3.11.6/lib/python3.11/tarfile.py'>, 'zipfile': <module 'zipfile' from '/home/docs/.asdf/installs/python/3.11.6/lib/python3.11/zipfile.py'>}

classmethod _open(*fpath*, *mode*, *backend=None*)

names()

read(*name*, *mode='rb'*)

Read data directly out of the archive.

Parameters

- **name** (*str*) – the name of the archive member to read
- **mode** (*str*) – This is a conceptual parameter that emulates the usual open mode. Defaults to "rb", which returns data as raw bytes. If "r" will decode the bytes into utf8-text.

classmethod coerce(*data*)

Either open an archive file path or coerce an existing ZipFile or tarfile structure into this wrapper class

add(*fpath*, *arcname=None*)

close()

extractall(*output_dpath='.'*, *verbose=1*, *overwrite=True*)

kwcoco.util.util_archive.**unarchive_file**(*archive_fpath*, *output_dpath='.'*, *verbose=1*, *overwrite=True*)

kwcoco.util.util_archive.**_available_zipfile_compressions**()

kwcoco.util.util_archive.**_coerce_zipfile_compression**(*compression*)

2.1.1.6.2.6 kwcoco.util.util_deprecate module

Deprecation helpers

kwcoco.util.util_deprecate.**deprecated_function_alias**(*modname*, *old_name*, *new_func*, *deprecate=None*, *error=None*, *remove=None*)

Exposes an old deprecated alias of a new prefered function

2.1.1.6.2.7 kwcoco.util.util_eval module

Defines a safer eval function

exception kwcoco.util.util_eval.**RestrictedSyntaxError**

Bases: Exception

An exception raised by restricted_eval if a disallowed expression is given

kwcoco.util.util_eval.**restricted_eval**(*expr*, *max_chars=32*, *local_dict=None*, *builtins_passlist=None*)

A restricted form of Python's eval that is meant to be slightly safer

Parameters

- **expr** (*str*) – the expression to evaluate
- **max_char** (*int*) – expression cannot be more than this many characters
- **local_dict** (*Dict[str, Any]*) – a list of variables allowed to be used
- **builtins_passlist** (*List[str] | None*) – if specified, only allow use of certain builtins

References

https://realpython.com/python-eval-function/#minimizing-the-security-issues-of-eval

Notes

This function may not be safe, but it has as many mitigation measures that I know about. This function should be audited and possibly made even more restricted. The idea is that this should just be used to evaluate numeric expressions.

Example

```
>>> from kwcoco.util.util_eval import *  # NOQA
>>> builtins_passlist = ['min', 'max', 'round', 'sum']
>>> local_dict = {}
>>> max_chars = 32
>>> expr = 'max(3 + 2, 9)'
>>> result = restricted_eval(expr, max_chars, local_dict, builtins_passlist)
>>> expr = '3 + 2'
>>> result = restricted_eval(expr, max_chars, local_dict, builtins_passlist)
>>> expr = '3 + 2'
>>> result = restricted_eval(expr, max_chars)
>>> import pytest
```

(continues on next page)

(continued from previous page)

```
>>> with pytest.raises(RestrictedSyntaxError):
>>>     expr = 'max(a + 2, 3)'
>>>     result = restricted_eval(expr, max_chars, dict(a=3))
```

2.1.1.6.2.8 kwcoco.util.util_futures module

Deprecated and functionality moved to ubelt

class kwcoco.util.util_futures.**Executor**(*mode='thread'*, *max_workers=0*)

Bases: object

A concrete asynchronous executor with a configurable backend.

The type of parallelism (or lack thereof) is configured via the mode parameter, which can be: "process", "thread", or "serial". This allows the user to easily enable / disable parallelism or switch between processes and threads without modifying the surrounding logic.

SeeAlso:

- concurrent.futures.ThreadPoolExecutor
- concurrent.futures.ProcessPoolExecutor
- SerialExecutor
- *JobPool*

In the case where you cant or dont want to use ubelt.Executor you can get similar behavior with the following pure-python snippet:

```
def Executor(max_workers):
    # Stdlib-only "ubelt.Executor"-like behavior
    if max_workers == 1:
        import contextlib
        def submit_partial(func, *args, **kwargs):
            def wrapper():
                return func(*args, **kwargs)
            wrapper.result = wrapper
            return wrapper
        executor = contextlib.nullcontext()
        executor.submit = submit_partial
    else:
        from concurrent.futures import ThreadPoolExecutor
        executor = ThreadPoolExecutor(max_workers=max_workers)
    return executor

executor = Executor(0)
with executor:
    jobs = []

    for arg in range(1000):
        job = executor.submit(chr, arg)
        jobs.append(job)

    results = []
```

(continues on next page)

(continued from previous page)

```
    for job in jobs:
        result = job.result()
        results.append(result)

print('results = {}'.format(ub.urepr(results, nl=1)))
```

Variables

backend (*SerialExecutor | ThreadPoolExecutor | ProcessPoolExecutor*) –

Example

```
>>> import ubelt as ub
>>> # Prototype code using simple serial processing
>>> executor = ub.Executor(mode='serial', max_workers=0)
>>> jobs = [executor.submit(sum, [i + 1, i]) for i in range(10)]
>>> print([job.result() for job in jobs])
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

```
>>> # Enable parallelism by only changing one parameter
>>> executor = ub.Executor(mode='process', max_workers=0)
>>> jobs = [executor.submit(sum, [i + 1, i]) for i in range(10)]
>>> print([job.result() for job in jobs])
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

Parameters

- **mode** (*str*) – The backend parallelism mechanism. Can be either thread, serial, or process. Defaults to 'thread'.
- **max_workers** (*int*) – number of workers. If 0, serial is forced. Defaults to 0.

submit(*func*, **args*, ***kw*)

Calls the submit function of the underlying backend.

Returns

a future representing the job

Return type

concurrent.futures.Future

shutdown()

Calls the shutdown function of the underlying backend.

map(*fn*, **iterables*, ***kwargs*)

Calls the map function of the underlying backend.

CommandLine

```
xdoctest -m ubelt.util_futures Executor.map
```

Example

```
>>> import ubelt as ub
>>> import concurrent.futures
>>> import string
>>> with ub.Executor(mode='serial') as executor:
...     result_iter = executor.map(int, string.digits)
...     results = list(result_iter)
>>> print('results = {!r}'.format(results))
results = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> with ub.Executor(mode='thread', max_workers=2) as executor:
...     result_iter = executor.map(int, string.digits)
...     results = list(result_iter)
>>> # xdoctest: +IGNORE_WANT
>>> print('results = {!r}'.format(results))
results = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

class kwcoco.util.util_futures.**JobPool**(*mode='thread'*, *max_workers=0*, *transient=False*)

Bases: object

Abstracts away boilerplate of submitting and collecting jobs

This is a basic wrapper around *ubelt.util_futures.Executor* that simplifies the most basic case by 1. keeping track of references to submitted futures for you and 2. providing an as_completed method to consume those futures as they are ready.

Variables

- **executor** (Executor) – internal executor object
- **jobs** (*List[Future]*) – internal job list. Note: do not rely on this attribute, it may change in the future.

Example

```
>>> import ubelt as ub
>>> def worker(data):
>>>     return data + 1
>>> pool = ub.JobPool('thread', max_workers=16)
>>> for data in ub.ProgIter(range(10), desc='submit jobs'):
>>>     pool.submit(worker, data)
>>> final = []
>>> for job in pool.as_completed(desc='collect jobs'):
>>>     info = job.result()
>>>     final.append(info)
>>> print('final = {!r}'.format(final))
```

Parameters

- **mode** (*str*) – The backend parallelism mechanism. Can be either thread, serial, or process. Defaults to 'thread'.
- **max_workers** (*int*) – number of workers. If 0, serial is forced. Defaults to 0.
- **transient** (*bool*) – if True, references to jobs will be discarded as they are returned by `as_completed()`. Otherwise the `jobs` attribute holds a reference to all jobs ever submitted. Default to False.

`submit`(*func*, **args*, ***kwargs*)

Submit a job managed by the pool

Parameters

- **func** (*Callable[. . . , Any]*) – A callable that will take as many arguments as there are passed iterables.
- ***args** – positional arguments to pass to the function
- ***kwargs** – keyword arguments to pass to the function

Returns

a future representing the job

Return type

concurrent.futures.Future

`shutdown`()

`_clear_completed`()

`as_completed`(*timeout=None*, *desc=None*, *progkw=None*)

Generates completed jobs in an arbitrary order

Parameters

- **timeout** (*float | None*) – Specify the the maximum number of seconds to wait for a job. Note: this is ignored in serial mode.
- **desc** (*str | None*) – if specified, reports progress with a `ubelt.progiter.ProgIter` object.
- **progkw** (*dict | None*) – extra keyword arguments to `ubelt.progiter.ProgIter`.

Yields

concurrent.futures.Future – The completed future object containing the results of a job.

CommandLine

```
xdoctest -m ubelt.util_futures JobPool.as_completed
```

Example

```
>>> import ubelt as ub
>>> pool = ub.JobPool('thread', max_workers=8)
>>> text = ub.paragraph(
...     '''
...     UDP is a cool protocol, check out the wiki:
...
...     UDP-based Data Transfer Protocol (UDT), is a high-performance
...     data transfer protocol designed for transferring large
...     volumetric datasets over high-speed wide area networks. Such
...     settings are typically disadvantageous for the more common TCP
...     protocol.
...     ''')
>>> for word in text.split(' '):
...     pool.submit(print, word)
>>> for _ in pool.as_completed():
...     pass
>>> pool.shutdown()
```

join(***kwargs*)

Like *JobPool.as_completed()*, but executes the *result* method of each future and returns only after all processes are complete. This allows for lower-boilerplate prototyping.

Parameters
****kwargs** – passed to *JobPool.as_completed()*

Returns
list of results

Return type
List[Any]

Example

```
>>> import ubelt as ub
>>> # We just want to try replacing our simple iterative algorithm
>>> # with the embarrassingly parallel version
>>> arglist = list(zip(range(1000), range(1000)))
>>> func = ub.identity
>>> #
>>> # Original version
>>> for args in arglist:
>>>     func(*args)
>>> #
>>> # Potentially parallel version
>>> jobs = ub.JobPool(max_workers=0)
>>> for args in arglist:
>>>     jobs.submit(func, *args)
>>> _ = jobs.join(desc='running')
```

2.1.1.6.2.9 kwcoco.util.util_json module

kwcoco.util.util_json.**ensure_json_serializable**(*dict_*, *normalize_containers=False*, *verbose=0*)

Attempt to convert common types (e.g. numpy) into something json complient

Convert numpy and tuples into lists

Parameters
normalize_containers (*bool*) – if True, normalizes dict containers to be standard python structures. Defaults to False.

Example

```
>>> data = ub.ddict(lambda: int)
>>> data['foo'] = ub.ddict(lambda: int)
>>> data['bar'] = np.array([1, 2, 3])
>>> data['foo']['a'] = 1
>>> data['foo']['b'] = (1, np.array([1, 2, 3]), {3: np.int32(3), 4: np.float16(1.0)}
↪)
>>> dict_ = data
>>> print(ub.urepr(data, nl=-1))
>>> assert list(find_json_unserializable(data))
>>> result = ensure_json_serializable(data, normalize_containers=True)
>>> print(ub.urepr(result, nl=-1))
>>> assert not list(find_json_unserializable(result))
>>> assert type(result) is dict
```

kwcoco.util.util_json.**find_json_unserializable**(*data*, *quickcheck=False*)

Recurse through json datastructure and find any component that causes a serialization error. Record the location of these errors in the datastructure as we recurse through the call tree.

Parameters

- **data** (*object*) – data that should be json serializable
- **quickcheck** (*bool*) – if True, check the entire datastructure assuming its ok before doing the python-based recursive logic.

Returns

list of "bad part" dictionaries containing items

'value' - the value that caused the serialization error

'loc' - which contains a list of key/indexes that can be used to lookup the location of the unserializable value. If the "loc" is a list, then it indicates a rare case where a key in a dictionary is causing the serialization error.

Return type
List[Dict]

Example

```
>>> from kwcoco.util.util_json import *  # NOQA
>>> part = ub.ddict(lambda: int)
>>> part['foo'] = ub.ddict(lambda: int)
>>> part['bar'] = np.array([1, 2, 3])
>>> part['foo']['a'] = 1
>>> # Create a dictionary with two unserializable parts
>>> data = [1, 2, {'nest1': [2, part]}, {frozenset({'badkey'}): 3, 2: 4}]
>>> parts = list(find_json_unserializable(data))
>>> print('parts = {}'.format(ub.urepr(parts, nl=1)))
>>> # Check expected structure of bad parts
>>> assert len(parts) == 2
>>> part = parts[1]
>>> assert list(part['loc']) == [2, 'nest1', 1, 'bar']
>>> # We can use the "loc" to find the bad value
>>> for part in parts:
>>>     # "loc" is a list of directions containing which keys/indexes
>>>     # to traverse at each descent into the data structure.
>>>     directions = part['loc']
>>>     curr = data
>>>     special_flag = False
>>>     for key in directions:
>>>         if isinstance(key, list):
>>>             # special case for bad keys
>>>             special_flag = True
>>>             break
>>>         else:
>>>             # normal case for bad values
>>>             curr = curr[key]
>>>     if special_flag:
>>>         assert part['data'] in curr.keys()
>>>         assert part['data'] is key[1]
>>>     else:
>>>         assert part['data'] is curr
```

Example

```
>>> # xdoctest: +SKIP("TODO: circular ref detect algo is wrong, fix it")
>>> from kwcoco.util.util_json import *  # NOQA
>>> import pytest
>>> # Test circular reference
>>> data = [[], {'a': []}]
>>> data[1]['a'].append(data)
>>> with pytest.raises(ValueError, match="Circular reference detected at.*1, 'a', 1*
↪"):
...     parts = list(find_json_unserializable(data))
>>> # Should be ok here
>>> shared_data = {'shared': 1}
>>> data = [[shared_data], shared_data]
>>> parts = list(find_json_unserializable(data))
```

kwcoco.util.util_json.**indexable_allclose**(*dct1*, *dct2*, *return_info=False*)

Walks through two nested data structures and ensures that everything is roughly the same.

Note: Use the version in ubelt instead

Parameters

- **dct1** – a nested indexable item
- **dct2** – a nested indexable item

Example

```
>>> from kwcoco.util.util_json import indexable_allclose
>>> dct1 = {
>>>     'foo': [1.222222, 1.333],
>>>     'bar': 1,
>>>     'baz': [],
>>> }
>>> dct2 = {
>>>     'foo': [1.22222, 1.333],
>>>     'bar': 1,
>>>     'baz': [],
>>> }
>>> assert indexable_allclose(dct1, dct2)
```

kwcoco.util.util_json.**coerce_indent**(*indent*)

Example

2.1.1.6.2.10 kwcoco.util.util_monkey module

class kwcoco.util.util_monkey.**SupressPrint**(**mods*, ***kw*)

Bases: object

Temporarily replace the print function in a module with a noop

Parameters

- ***mods** – the modules to disable print in
- ****kw** – only accepts "enabled" enabled (bool, default=True): enables or disables this context

class kwcoco.util.util_monkey.**Reloadable**

Bases: type

This is a metaclass that overrides the behavior of isinstance and issubclass when invoked on classes derived from this such that they only check that the module and class names agree, which are preserved through module reloads, whereas class instances are not.

This is useful for interactive develoment, but should be removed in production.

Example

```
>>> from kwcoco.util.util_monkey import *  # NOQA
>>> # Illustrate what happens with a reload when using this utility
>>> # versus without it.
>>> class Base1:
>>>     ...
>>> class Derived1(Base1):
>>>     ...
>>> @Reloadable.add_metaclass
>>> class Base2:
>>>     ...
>>> class Derived2(Base2):
>>>     ...
>>> inst1 = Derived1()
>>> inst2 = Derived2()
>>> assert isinstance(inst1, Derived1)
>>> assert isinstance(inst2, Derived2)
>>> # Simulate reload
>>> class Base1:
>>>     ...
>>> class Derived1(Base1):
>>>     ...
>>> @Reloadable.add_metaclass
>>> class Base2:
>>>     ...
>>> class Derived2(Base2):
>>>     ...
>>> assert not isinstance(inst1, Derived1)
>>> assert isinstance(inst2, Derived2)
```

classmethod add_metaclass(*cls*)

Class decorator for creating a class with this as a metaclass

classmethod developing(*cls*)

Like add_metaclass, but warns the user that they are developing. This helps remind them to remove this in production

2.1.1.6.2.11 kwcoco.util.util_parallel module

kwcoco.util.util_parallel.**coerce_num_workers**(*num_workers='auto'*, *minimum=0*)

Return some number of CPUs based on a chosen hueristic

Parameters

- **num_workers** (*int* | *str*) – A special string code, or an exact number of cpus
- **minimum** (*int*) – minimum workers we are allowed to return

Returns

number of available cpus based on request parameters

Return type

int

CommandLine

```
xdoctest -m kwcoco.util.util_parallel coerce_num_workers
```

Example

```
>>> from kwcoco.util.util_parallel import *  # NOQA
>>> print(coerce_num_workers('all'))
>>> print(coerce_num_workers('avail'))
>>> print(coerce_num_workers('auto'))
>>> print(coerce_num_workers('all-2'))
>>> print(coerce_num_workers('avail-2'))
>>> print(coerce_num_workers('all/2'))
>>> print(coerce_num_workers('min(all,2)'))
>>> print(coerce_num_workers('[max(all,2)][0]'))
>>> import pytest
>>> with pytest.raises(Exception):
>>>     print(coerce_num_workers('all + 1' + (' + 1' * 100)))
>>> total_cpus = coerce_num_workers('all')
>>> assert coerce_num_workers('all-2') == max(total_cpus - 2, 0)
>>> assert coerce_num_workers('all-100') == max(total_cpus - 100, 0)
>>> assert coerce_num_workers('avail') <= coerce_num_workers('all')
>>> assert coerce_num_workers(3) == max(3, 0)
```

2.1.1.6.2.12 kwcoco.util.util_reroot module

Rerooting is harder than you would think

kwcoco.util.util_reroot.**special_reroot_single**(*dset*, *verbose=0*)

kwcoco.util.util_reroot.**resolve_relative_to**(*path*, *dpath*, *strict=False*)

Given a path, try to resolve its symlinks such that it is relative to the given dpath.

Example

```
>>> from kwcoco.util.util_reroot import *  # NOQA
>>> import os
>>> def _symlink(self, target, verbose=0):
>>>     return ub.Path(ub.symlink(target, self, verbose=verbose))
>>> ub.Path._symlink = _symlink
>>> #
>>> # TODO: try to enumerate all basic cases
>>> #
>>> base = ub.Path.appdir('kwcoco/tests/reroot')
>>> base.delete().ensuredir()
>>> #
>>> drive1 = (base / 'drive1').ensuredir()
>>> drive2 = (base / 'drive2').ensuredir()
>>> #
```

(continues on next page)

(continued from previous page)

```
>>> data_repo1 = (drive1 / 'data_repo1').ensuredir()
>>> cache = (data_repo1 / '.cache').ensuredir()
>>> real_file1 = (cache / 'real_file1').touch()
>>> #
>>> real_bundle = (data_repo1 / 'real_bundle').ensuredir()
>>> real_assets = (real_bundle / 'assets').ensuredir()
>>> #
>>> # Symlink file outside of the bundle
>>> link_file1 = (real_assets / 'link_file1')._symlink(real_file1)
>>> real_file2 = (real_assets / 'real_file2').touch()
>>> link_file2 = (real_assets / 'link_file2')._symlink(real_file2)
>>> #
>>> #
>>> # A symlink to the data repo
>>> data_repo2 = (drive1 / 'data_repo2')._symlink(data_repo1)
>>> data_repo3 = (drive2 / 'data_repo3')._symlink(data_repo1)
>>> data_repo4 = (drive2 / 'data_repo4')._symlink(data_repo2)
>>> #
>>> # A prediction repo TODO
>>> pred_repo5 = (drive2 / 'pred_repo5').ensuredir()
>>> #
>>> # _ = ub.cmd(f'tree -a {base}', verbose=3)
>>> #
>>> fpaths = []
>>> for r, ds, fs in os.walk(base, followlinks=True):
>>>     for f in fs:
>>>         if 'file' in f:
>>>             fpath = ub.Path(r) / f
>>>             fpaths.append(fpath)
>>> #
>>> #
>>> dpath = real_bundle.resolve()
>>> #
>>> for path in fpaths:
>>>     # print(f'{path}')
>>>     # print(f'{path.resolve()=}')
>>>     resolved_rel = resolve_relative_to(path, dpath)
>>>     print('resolved_rel = {!r}'.format(resolved_rel))
```

kwcoco.util.util_reroot.**resolve_directory_symlinks**(*path*)

Only resolve symlinks of directories, not the base file

2.1.1.6.2.13 kwcoco.util.util_sklearn module

Extensions to sklearn constructs

class kwcoco.util.util_sklearn.**StratifiedGroupKFold**(*n_splits=3*, *shuffle=False*, *random_state=None*)

Bases: `_BaseKFold`

Stratified K-Folds cross-validator with Grouping

Provides train/test indices to split data in train/test sets.

This cross-validation object is a variation of GroupKFold that returns stratified folds. The folds are made by preserving the percentage of samples for each class.

This is an old interface and should likely be refactored and modernized.

Parameters
n_splits (*int, default=3*) – Number of folds. Must be at least 2.

_make_test_folds(*X*, *y=None*, *groups=None*)

Parameters

- **X** (*ndarray*) – data
- **y** (*ndarray*) – labels
- **groups** (*ndarray*) – groupids for items. Items with the same groupid must be placed in the same group.

Returns
test_folds

Return type
list

Example

```
>>> from kwcoco.util.util_sklearn import *  # NOQA
>>> import kwarray
>>> rng = kwarray.ensure_rng(0)
>>> groups = [1, 1, 3, 4, 2, 2, 7, 8, 8]
>>> y      = [1, 1, 1, 1, 2, 2, 2, 3, 3]
>>> X = np.empty((len(y), 0))
>>> self = StratifiedGroupKFold(random_state=rng, shuffle=True)
>>> skf_list = list(self.split(X=X, y=y, groups=groups))
>>> import ubelt as ub
>>> print(ub.urepr(skf_list, nl=1, with_dtype=False))
[
    (np.array([2, 3, 4, 5, 6]), np.array([0, 1, 7, 8])),
    (np.array([0, 1, 2, 7, 8]), np.array([3, 4, 5, 6])),
    (np.array([0, 1, 3, 4, 5, 6, 7, 8]), np.array([2])),
]
```

_iter_test_masks(*X*, *y=None*, *groups=None*)

split(*X*, *y*, *groups=None*)

Generate indices to split data into training and test set.

_abc_impl = <_abc._abc_data object>

2.1.1.6.2.14 kwcoco.util.util_special_json module

Special non-general json functions

kwcoco.util.util_special_json.**_json_dumps**(*data*, *indent=None*)

kwcoco.util.util_special_json.**_json_lines_dumps**(*key*, *value*, *indent*)

kwcoco.util.util_special_json.**_special_kwcoco_pretty_dumps_orig**(*data*, *indent=None*)

The old way of doing "pretty" dumping, except it isn't that pretty.

See also:

Tried to do a "principled" lark version, but this this way is faster ~/code/kwcoco/dev/devcheck/json_dumps_experiments.py

2.1.1.6.2.15 kwcoco.util.util_truncate module

Truncate utility based on python-slugify.

https://pypi.org/project/python-slugify/1.2.2/

kwcoco.util.util_truncate.**_trunc_op**(*string*, *max_length*, *trunc_loc*, *trunc_char='~'*)

Example

```
>>> from kwcoco.util.util_truncate import _trunc_op
>>> string =
↪'DarnOvercastSculptureTipperBlazerConcaveUnsuitedDerangedHexagonRockband'
>>> max_length = 16
>>> trunc_loc = 0.5
>>> _trunc_op(string, max_length, trunc_loc)
```

```
>>> from kwcoco.util.util_truncate import _trunc_op
>>> max_length = 16
>>> string = 'a' * 16
>>> _trunc_op(string, max_length, trunc_loc)
```

```
>>> string = 'a' * 17
>>> _trunc_op(string, max_length, trunc_loc)
```

kwcoco.util.util_truncate.**smart_truncate**(*string*, *max_length=0*, *separator=' '*, *trunc_loc=0.5*, *trunc_char='~'*)

Truncate a string. :param string (str): string for modification :param max_length (int): output string length :param word_boundary (bool): :param save_order (bool): if True then word order of output string is like input string :param separator (str): separator between words :param trunc_loc (float): fraction of location where to remove the text

trunc_char (str): the character to denote where truncation is starting

Returns

2.1.1.6.2.16 kwcoco.util.util_windows module

kwcoco.util.util_windows.**fix_msys_path**(*path*)

Windows is so special. When using msys bash if you pass a path on the CLI it resolves /c to C:/, but if you have you path as part of a config string, it doesnt know how to do that, and at that point Python doesn't handle the msys style /c paths. This is a hack detects and fixes this in this location.

Example

```
>>> print(fix_msys_path('/c/Users/foobar'))
C:/Users/foobar
>>> print(fix_msys_path(r'\c\Users\foobar'))
C:/Users\foobar
>>> print(fix_msys_path(r'\d\Users\foobar'))
D:/Users\foobar
>>> print(fix_msys_path(r'\z'))
Z:/
>>> import pathlib
>>> assert fix_msys_path(pathlib.Path(r'\z')) == pathlib.Path('Z:/')
```

kwcoco.util.util_windows.**is_windows_path**(*path*)

Example

```
>>> assert is_windows_path('C:')
>>> assert is_windows_path('C:/')
>>> assert is_windows_path('C:\\')
>>> assert is_windows_path('C:/foo')
>>> assert is_windows_path('C:\\foo')
>>> assert not is_windows_path('/foo')
```

2.1.1.6.3 Module contents

mkinit ~/code/kwcoco/kwcoco/util/__init__.py -w mkinit ~/code/kwcoco/kwcoco/util/__init__.py –lazy

kwcoco.util.**ALLOF**(**TYPES*)

kwcoco.util.**ANYOF**(**TYPES*)

kwcoco.util.**ARRAY**(*TYPE={}, **kw*)

https://json-schema.org/understanding-json-schema/reference/array.html

Example

```
>>> from kwcoco.util.jsonschema_elements import *  # NOQA
>>> ARRAY(numItems=3)
>>> schema = ARRAY(minItems=3)
>>> schema.validate()
{'type': 'array', 'items': {}, 'minItems': 3}
```

class kwcoco.util.**Archive**(*fpath=None*, *mode='r'*, *backend=None*, *file=None*)

Bases: object

Abstraction over zipfile and tarfile

Todo: see if we can use one of these other tools instead

SeeAlso:
https://github.com/RKrahl/archive-tools https://pypi.org/project/arlib/

Example

```
>>> from kwcoco.util.util_archive import Archive
>>> import ubelt as ub
>>> dpath = ub.Path.appdir('kwcoco', 'tests', 'util', 'archive')
>>> dpath.delete().ensuredir()
>>> # Test write mode
>>> mode = 'w'
>>> arc_zip = Archive(str(dpath / 'demo.zip'), mode=mode)
>>> arc_tar = Archive(str(dpath / 'demo.tar.gz'), mode=mode)
>>> open(dpath / 'data_1only.txt', 'w').write('bazbzzz')
>>> open(dpath / 'data_2only.txt', 'w').write('buzzz')
>>> open(dpath / 'data_both.txt', 'w').write('foobar')
>>> #
>>> arc_zip.add(dpath / 'data_both.txt')
>>> arc_zip.add(dpath / 'data_1only.txt')
>>> #
>>> arc_tar.add(dpath / 'data_both.txt')
>>> arc_tar.add(dpath / 'data_2only.txt')
>>> #
>>> arc_zip.close()
>>> arc_tar.close()
>>> #
>>> # Test read mode
>>> arc_zip = Archive(str(dpath / 'demo.zip'), mode='r')
>>> arc_tar = Archive(str(dpath / 'demo.tar.gz'), mode='r')
>>> # Test names
>>> name = 'data_both.txt'
>>> assert name in arc_zip.names()
>>> assert name in arc_tar.names()
>>> # Test read
>>> assert arc_zip.read(name, mode='r') == 'foobar'
>>> assert arc_tar.read(name, mode='r') == 'foobar'
```

(continues on next page)

(continued from previous page)

```
>>> #
>>> # Test extractall
>>> extract_dpath = ub.ensuredir(str(dpath / 'extracted'))
>>> extracted1 = arc_zip.extractall(extract_dpath)
>>> extracted2 = arc_tar.extractall(extract_dpath)
>>> for fpath in extracted2:
>>>     print(open(fpath, 'r').read())
>>> for fpath in extracted1:
>>>     print(open(fpath, 'r').read())
```

Parameters

- **fpath** (*str | None*) – path to open
- **mode** (*str*) – either r or w
- **backend** (*str | ModuleType | None*) – either tarfile, zipfile string or module.
- **file** (*tarfile.TarFile | zipfile.ZipFile | None*) – the open backend file if it already exists. If not set, than fpath will open it.

_available_backends = {'tarfile': <module 'tarfile' from '/home/docs/.asdf/installs/python/3.11.6/lib/python3.11/tarfile.py'>, 'zipfile': <module 'zipfile' from '/home/docs/.asdf/installs/python/3.11.6/lib/python3.11/zipfile.py'>}

classmethod _open(*fpath*, *mode*, *backend=None*)

names()

read(*name*, *mode='rb'*)

Read data directly out of the archive.

Parameters

- **name** (*str*) – the name of the archive member to read
- **mode** (*str*) – This is a conceptual parameter that emulates the usual open mode. Defaults to "rb", which returns data as raw bytes. If "r" will decode the bytes into utf8-text.

classmethod coerce(*data*)

Either open an archive file path or coerce an existing ZipFile or tarfile structure into this wrapper class

add(*fpath*, *arcname=None*)

close()

extractall(*output_dpath='.'*, *verbose=1*, *overwrite=True*)

class kwcoco.util.**ContainerElements**

Bases: `object`

Types that contain other types

Example

```
>>> from kwcoco.util.jsonschema_elements import *  # NOQA
>>> print(elem.ARRAY().validate())
>>> print(elem.OBJECT().validate())
>>> print(elem.OBJECT().validate())
{'type': 'array', 'items': {}}
{'type': 'object', 'properties': {}}
{'type': 'object', 'properties': {}}
```

ARRAY(*TYPE={}*, ***kw*)

https://json-schema.org/understanding-json-schema/reference/array.html

Example

```
>>> from kwcoco.util.jsonschema_elements import *  # NOQA
>>> ARRAY(numItems=3)
>>> schema = ARRAY(minItems=3)
>>> schema.validate()
{'type': 'array', 'items': {}, 'minItems': 3}
```

OBJECT(*PROPERTIES={}*, ***kw*)

https://json-schema.org/understanding-json-schema/reference/object.html

Example

```
>>> import jsonschema
>>> schema = elem.OBJECT()
>>> jsonschema.validate({}, schema)
>>> #
>>> import jsonschema
>>> schema = elem.OBJECT({
>>>     'key1': elem.ANY(),
>>>     'key2': elem.ANY(),
>>> }, required=['key1'])
>>> jsonschema.validate({'key1': None}, schema)
>>> #
>>> import jsonschema
>>> schema = elem.OBJECT({
>>>     'key1': elem.OBJECT({'arr': elem.ARRAY()}),
>>>     'key2': elem.ANY(),
>>> }, required=['key1'], title='a title')
>>> schema.validate()
>>> print('schema = {}'.format(ub.urepr(schema, sort=1, nl=-1)))
>>> jsonschema.validate({'key1': {'arr': []}}, schema)
schema = {
    'properties': {
        'key1': {
            'properties': {
                'arr': {'items': {}, 'type': 'array'}
```

(continues on next page)

(continued from previous page)

```
            },
            'type': 'object'
        },
        'key2': {}
    },
    'required': ['key1'],
    'title': 'a title',
    'type': 'object'
}
```

class kwcoco.util.**DictLike**

Bases: `NiceRepr`

An inherited class must specify the getitem, setitem, and
keys methods.

A class is dictionary like if it has:

`__iter__`, `__len__`, `__contains__`, `__getitem__`, `items`, `keys`, `values`, `get`,

and if it should be writable it should have: `__delitem__`, `__setitem__`, `update`,

And perhaps: `copy`,

`__iter__`, `__len__`, `__contains__`, `__getitem__`, `items`, `keys`, `values`, `get`,

and if it should be writable it should have: `__delitem__`, `__setitem__`, `update`,

And perhaps: `copy`,

getitem(*key*)

Parameters
key (*Any*) – a key

Returns
a value

Return type
Any

setitem(*key*, *value*)

Parameters

- **key** (*Any*)
- **value** (*Any*)

delitem(*key*)

Parameters
key (*Any*)

keys()

Yields
Any – a key

items()

Yields
Tuple[Any, Any] – a key value pair

values()

Yields
Any – a value

copy()

Return type
Dict

to_dict()

Return type
Dict

asdict()

Return type
Dict

update(*other*)

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

Parameters

- **key** (*Any*)
- **default** (*Any*)

Return type
Any

class kwcoco.util.**Element**(*base*, *options={}*, *_magic=None*)

Bases: dict

A dictionary used to define an element of a JSON Schema.

The exact keys/values for the element will depend on the type of element being described. The *SchemaElements* defines exactly what these are for the core elements. (e.g. OBJECT, INTEGER, NULL, ARRAY, ANYOF)

Example

```
>>> from kwcoco.coco_schema import *  # NOQA
>>> self = Element(base={'type': 'demo'}, options={'opt1', 'opt2'})
>>> new = self(opt1=3)
>>> print('self = {}'.format(ub.urepr(self, nl=1, sort=1)))
>>> print('new = {}'.format(ub.urepr(new, nl=1, sort=1)))
>>> print('new2 = {}'.format(ub.urepr(new(), nl=1, sort=1)))
>>> print('new3 = {}'.format(ub.urepr(new(title='myvar'), nl=1, sort=1)))
>>> print('new4 = {}'.format(ub.urepr(new(title='myvar')(examples=['']), nl=1,␣
↪sort=1)))
>>> print('new5 = {}'.format(ub.urepr(new(badattr=True), nl=1, sort=1)))
self = {
```

(continues on next page)

(continued from previous page)

```
    'type': 'demo',
}
new = {
    'opt1': 3,
    'type': 'demo',
}
new2 = {
    'opt1': 3,
    'type': 'demo',
}
new3 = {
    'opt1': 3,
    'title': 'myvar',
    'type': 'demo',
}
new4 = {
    'examples': [''],
    'opt1': 3,
    'title': 'myvar',
    'type': 'demo',
}
new5 = {
    'opt1': 3,
    'type': 'demo',
}
```

Parameters

- **base** (*dict*) – the keys / values this schema must contain
- **options** (*dict*) – the keys / values this schema may contain
- **_magic** (*callable | None*) – called when creating an instance of this schema element. Allows convinience attributes to be converted to the formal jsonschema specs. TODO: _magic is a terrible name, we need to rename it with something descriptive.

`validate`(*instance=NoParam*)

If `instance` is given, validates that that dictionary conforms to this schema. Otherwise validates that this is a valid schema element.

Parameters
instance (*dict*) – a dictionary to validate

`class kwcoco.util.IndexableWalker`(*data*, *dict_cls=(<class 'dict'>,)*, *list_cls=(<class 'list'>, <class 'tuple'>)*)

Bases: `Generator`

Traverses through a nested tree-liked indexable structure.

Generates a path and value to each node in the structure. The path is a list of indexes which if applied in order will reach the value.

The `__setitem__` method can be used to modify a nested value based on the path returned by the generator.

When generating values, you can use "send" to prevent traversal of a particular branch.

RelatedWork:

- **https://pypi.org/project/python-benedict/ - implements a dictionary**
  subclass with similar nested indexing abilities.

Variables

- **data** (*dict | list | tuple*) – the wrapped indexable data
- **dict_cls** (*Tuple[type]*) – the types that should be considered dictionary mappings for the purpose of nested iteration. Defaults to `dict`.
- **list_cls** (*Tuple[type]*) – the types that should be considered list-like for the purposes of nested iteration. Defaults to `(list, tuple)`.

Example

```
>>> import ubelt as ub
>>> # Given Nested Data
>>> data = {
>>>     'foo': {'bar': 1},
>>>     'baz': [{'biz': 3}, {'buz': [4, 5, 6]}],
>>> }
>>> # Create an IndexableWalker
>>> walker = ub.IndexableWalker(data)
>>> # We iterate over the data as if it was flat
>>> # ignore the <want> string due to order issues on older Pythons
>>> # xdoctest: +IGNORE_WANT
>>> for path, val in walker:
>>>     print(path)
['foo']
['baz']
['baz', 0]
['baz', 1]
['baz', 1, 'buz']
['baz', 1, 'buz', 0]
['baz', 1, 'buz', 1]
['baz', 1, 'buz', 2]
['baz', 0, 'biz']
['foo', 'bar']
>>> # We can use "paths" as keys to getitem into the walker
>>> path = ['baz', 1, 'buz', 2]
>>> val = walker[path]
>>> assert val == 6
>>> # We can use "paths" as keys to setitem into the walker
>>> assert data['baz'][1]['buz'][2] == 6
>>> walker[path] = 7
>>> assert data['baz'][1]['buz'][2] == 7
>>> # We can use "paths" as keys to delitem into the walker
>>> assert data['baz'][1]['buz'][1] == 5
>>> del walker[['baz', 1, 'buz', 1]]
>>> assert data['baz'][1]['buz'][1] == 7
```

Example

```
>>> # Create nested data
>>> # xdoctest: +REQUIRES(module:numpy)
>>> import numpy as np
>>> import ubelt as ub
>>> data = ub.ddict(lambda: int)
>>> data['foo'] = ub.ddict(lambda: int)
>>> data['bar'] = np.array([1, 2, 3])
>>> data['foo']['a'] = 1
>>> data['foo']['b'] = np.array([1, 2, 3])
>>> data['foo']['c'] = [1, 2, 3]
>>> data['baz'] = 3
>>> print('data = {}'.format(ub.repr2(data, nl=True)))
>>> # We can walk through every node in the nested tree
>>> walker = ub.IndexableWalker(data)
>>> for path, value in walker:
>>>     print('walk path = {}'.format(ub.repr2(path, nl=0)))
>>>     if path[-1] == 'c':
>>>         # Use send to prevent traversing this branch
>>>         got = walker.send(False)
>>>         # We can modify the value based on the returned path
>>>         walker[path] = 'changed the value of c'
>>> print('data = {}'.format(ub.repr2(data, nl=True)))
>>> assert data['foo']['c'] == 'changed the value of c'
```

Example

```
>>> # Test sending false for every data item
>>> import ubelt as ub
>>> data = {1: [1, 2, 3], 2: [1, 2, 3]}
>>> walker = ub.IndexableWalker(data)
>>> # Sending false means you wont traverse any further on that path
>>> num_iters_v1 = 0
>>> for path, value in walker:
>>>     print('[v1] walk path = {}'.format(ub.repr2(path, nl=0)))
>>>     walker.send(False)
>>>     num_iters_v1 += 1
>>> num_iters_v2 = 0
>>> for path, value in walker:
>>>     # When we dont send false we walk all the way down
>>>     print('[v2] walk path = {}'.format(ub.repr2(path, nl=0)))
>>>     num_iters_v2 += 1
>>> assert num_iters_v1 == 2
>>> assert num_iters_v2 == 8
```

Example

```
>>> # Test numpy
>>> # xdoctest: +REQUIRES(CPython)
>>> # xdoctest: +REQUIRES(module:numpy)
>>> import ubelt as ub
>>> import numpy as np
>>> # By default we don't recurse into ndarrays because they
>>> # Are registered as an indexable class
>>> data = {2: np.array([1, 2, 3])}
>>> walker = ub.IndexableWalker(data)
>>> num_iters = 0
>>> for path, value in walker:
>>>     print('walk path = {}'.format(ub.repr2(path, nl=0)))
>>>     num_iters += 1
>>> assert num_iters == 1
>>> # Currently to use top-level ndarrays, you need to extend what the
>>> # list class is. This API may change in the future to be easier
>>> # to work with.
>>> data = np.random.rand(3, 5)
>>> walker = ub.IndexableWalker(data, list_cls=(list, tuple, np.ndarray))
>>> num_iters = 0
>>> for path, value in walker:
>>>     print('walk path = {}'.format(ub.repr2(path, nl=0)))
>>>     num_iters += 1
>>> assert num_iters == 3 + 3 * 5
```

send(*arg*) → send 'arg' into generator,

return next yielded value or raise StopIteration.

throw(*typ*[, *val*[, *tb*]]) → raise exception in generator,

return next yielded value or raise StopIteration.

_walk(*data=None*, *prefix=[]*)

Defines the underlying generator used by IndexableWalker

Yields

***Tuple[List, Any] | None* – path (List) - a "path" through the nested data structure**
value (Any) - the value indexed by that "path".

Can also yield None in the case that *send* is called on the generator.

allclose(*other*, *rel_tol=1e-09*, *abs_tol=0.0*, *return_info=False*)

Walks through this and another nested data structures and checks if everything is roughly the same.

Parameters

- **other** (*IndexableWalker | List | Dict*) – a nested indexable item to compare against.
- **rel_tol** (*float*) – maximum difference for being considered "close", relative to the magnitude of the input values
- **abs_tol** (*float*) – maximum difference for being considered "close", regardless of the magnitude of the input values
- **return_info** (*bool, default=False*) – if true, return extra info dict

Returns
A boolean result if `return_info` is false, otherwise a tuple of the boolean result and an "info" dict containing detailed results indicating what matched and what did not.

Return type
bool | Tuple[bool, Dict]

Example

```
>>> import ubelt as ub
>>> items1 = ub.IndexableWalker({
>>>     'foo': [1.222222, 1.333],
>>>     'bar': 1,
>>>     'baz': [],
>>> })
>>> items2 = ub.IndexableWalker({
>>>     'foo': [1.22222, 1.333],
>>>     'bar': 1,
>>>     'baz': [],
>>> })
>>> flag, return_info =  items1.allclose(items2, return_info=True)
>>> print('return_info = {}'.format(ub.repr2(return_info, nl=1)))
>>> print('flag = {!r}'.format(flag))
>>> for p1, v1, v2  in return_info['faillist']:
>>>     v1_ = items1[p1]
>>>     print('*fail p1, v1, v2 = {}, {}, {}'.format(p1, v1, v2))
>>> for p1 in return_info['passlist']:
>>>     v1_ = items1[p1]
>>>     print('*pass p1, v1_ = {}, {}'.format(p1, v1_))
>>> assert not flag
```

```
>>> import ubelt as ub
>>> items1 = ub.IndexableWalker({
>>>     'foo': [1.0000000000000000000000001, 1.],
>>>     'bar': 1,
>>>     'baz': [],
>>> })
>>> items2 = ub.IndexableWalker({
>>>     'foo': [0.9999999999999999, 1.],
>>>     'bar': 1,
>>>     'baz': [],
>>> })
>>> flag, return_info =  items1.allclose(items2, return_info=True)
>>> print('return_info = {}'.format(ub.repr2(return_info, nl=1)))
>>> print('flag = {!r}'.format(flag))
>>> assert flag
```

Example

```
>>> import ubelt as ub
>>> flag, return_info =  ub.IndexableWalker([]).allclose(ub.IndexableWalker([]),
↪ return_info=True)
>>> print('return_info = {!r}'.format(return_info))
>>> print('flag = {!r}'.format(flag))
>>> assert flag
```

Example

```
>>> import ubelt as ub
>>> flag =  ub.IndexableWalker([]).allclose([], return_info=False)
>>> print('flag = {!r}'.format(flag))
>>> assert flag
```

Example

```
>>> import ubelt as ub
>>> flag, return_info =  ub.IndexableWalker([]).allclose([1], return_info=True)
>>> print('return_info = {!r}'.format(return_info))
>>> print('flag = {!r}'.format(flag))
>>> assert not flag
```

Example

```
>>> # xdoctest: +REQUIRES(module:numpy)
>>> import ubelt as ub
>>> import numpy as np
>>> a = np.random.rand(3, 5)
>>> b = a + 1
>>> wa = ub.IndexableWalker(a, list_cls=(np.ndarray,))
>>> wb = ub.IndexableWalker(b, list_cls=(np.ndarray,))
>>> flag, return_info =  wa.allclose(wb, return_info=True)
>>> print('return_info = {!r}'.format(return_info))
>>> print('flag = {!r}'.format(flag))
>>> assert not flag
>>> a = np.random.rand(3, 5)
>>> b = a.copy() + 1e-17
>>> wa = ub.IndexableWalker([a], list_cls=(np.ndarray, list))
>>> wb = ub.IndexableWalker([b], list_cls=(np.ndarray, list))
>>> flag, return_info =  wa.allclose(wb, return_info=True)
>>> assert flag
>>> print('return_info = {!r}'.format(return_info))
>>> print('flag = {!r}'.format(flag))
```

_abc_impl = <_abc._abc_data object>

kwcoco.util.**NOT**(*TYPE*)

kwcoco.util.**OBJECT**(*PROPERTIES={}*, ***kw*)

https://json-schema.org/understanding-json-schema/reference/object.html

Example

```
>>> import jsonschema
>>> schema = elem.OBJECT()
>>> jsonschema.validate({}, schema)
>>> #
>>> import jsonschema
>>> schema = elem.OBJECT({
>>>     'key1': elem.ANY(),
>>>     'key2': elem.ANY(),
>>> }, required=['key1'])
>>> jsonschema.validate({'key1': None}, schema)
>>> #
>>> import jsonschema
>>> schema = elem.OBJECT({
>>>     'key1': elem.OBJECT({'arr': elem.ARRAY()}),
>>>     'key2': elem.ANY(),
>>> }, required=['key1'], title='a title')
>>> schema.validate()
>>> print('schema = {}'.format(ub.urepr(schema, sort=1, nl=-1)))
>>> jsonschema.validate({'key1': {'arr': []}}, schema)
schema = {
    'properties': {
        'key1': {
            'properties': {
                'arr': {'items': {}, 'type': 'array'}
            },
            'type': 'object'
        },
        'key2': {}
    },
    'required': ['key1'],
    'title': 'a title',
    'type': 'object'
}
```

kwcoco.util.**ONEOF**(**TYPES*)

class kwcoco.util.**QuantifierElements**

Bases: object

Quantifier types

https://json-schema.org/understanding-json-schema/reference/combining.html#allof

Example

```
>>> from kwcoco.util.jsonschema_elements import *  # NOQA
>>> elem.ANYOF(elem.STRING, elem.NUMBER).validate()
>>> elem.ONEOF(elem.STRING, elem.NUMBER).validate()
>>> elem.NOT(elem.NULL).validate()
>>> elem.NOT(elem.ANY).validate()
>>> elem.ANY.validate()
```

property ANY

ALLOF(**TYPES*)

ANYOF(**TYPES*)

ONEOF(**TYPES*)

NOT(*TYPE*)

class kwcoco.util.**ScalarElements**

Bases: `object`

Single-valued elements

property NULL

//json-schema.org/understanding-json-schema/reference/null.html

Type
https

property BOOLEAN

//json-schema.org/understanding-json-schema/reference/null.html

Type
https

property STRING

//json-schema.org/understanding-json-schema/reference/string.html

Type
https

property NUMBER

//json-schema.org/understanding-json-schema/reference/numeric.html#number

Type
https

property INTEGER

//json-schema.org/understanding-json-schema/reference/numeric.html#integer

Type
https

class kwcoco.util.**SchemaElements**

Bases: *ScalarElements*, *QuantifierElements*, *ContainerElements*

Functional interface into defining jsonschema structures.

See mixin classes for details.

References

https://json-schema.org/understanding-json-schema/

Todo:

- [ ] Generics: title, description, default, examples

CommandLine

```
xdoctest -m /home/joncrall/code/kwcoco/kwcoco/util/jsonschema_elements.py␣
↪SchemaElements
```

Example

```
>>> from kwcoco.util.jsonschema_elements import *  # NOQA
>>> elem = SchemaElements()
>>> elem.ARRAY(elem.ANY())
>>> schema = OBJECT({
>>>     'prop1': ARRAY(INTEGER, minItems=3),
>>>     'prop2': ARRAY(STRING, numItems=2),
>>>     'prop3': ARRAY(OBJECT({
>>>         'subprob1': NUMBER,
>>>         'subprob2': NUMBER,
>>>     }))
>>> })
>>> print('schema = {}'.format(ub.urepr(schema, nl=2, sort=1)))
schema = {
    'properties': {
        'prop1': {'items': {'type': 'integer'}, 'minItems': 3, 'type': 'array'},
        'prop2': {'items': {'type': 'string'}, 'maxItems': 2, 'minItems': 2, 'type
↪': 'array'},
        'prop3': {'items': {'properties': {'subprob1': {'type': 'number'}, 'subprob2
↪': {'type': 'number'}}, 'type': 'object'}, 'type': 'array'},
    },
    'type': 'object',
}
```

```
>>> TYPE = elem.OBJECT({
>>>     'p1': ANY,
>>>     'p2': ANY,
>>> }, required=['p1'])
>>> import jsonschema
>>> inst = {'p1': None}
>>> jsonschema.validate(inst, schema=TYPE)
>>> #jsonschema.validate({'p2': None}, schema=TYPE)
```

class kwcoco.util.**StratifiedGroupKFold**(*n_splits=3*, *shuffle=False*, *random_state=None*)

Bases: `_BaseKFold`

Stratified K-Folds cross-validator with Grouping

Provides train/test indices to split data in train/test sets.

This cross-validation object is a variation of GroupKFold that returns stratified folds. The folds are made by preserving the percentage of samples for each class.

This is an old interface and should likely be refactored and modernized.

Parameters
: **n_splits** (*int, default=3*) – Number of folds. Must be at least 2.

_make_test_folds(*X*, *y=None*, *groups=None*)

Parameters

- **X** (*ndarray*) – data
- **y** (*ndarray*) – labels
- **groups** (*ndarray*) – groupids for items. Items with the same groupid must be placed in the same group.

Returns
: test_folds

Return type
: list

Example

```
>>> from kwcoco.util.util_sklearn import *  # NOQA
>>> import kwarray
>>> rng = kwarray.ensure_rng(0)
>>> groups = [1, 1, 3, 4, 2, 2, 7, 8, 8]
>>> y      = [1, 1, 1, 1, 2, 2, 2, 3, 3]
>>> X = np.empty((len(y), 0))
>>> self = StratifiedGroupKFold(random_state=rng, shuffle=True)
>>> skf_list = list(self.split(X=X, y=y, groups=groups))
>>> import ubelt as ub
>>> print(ub.urepr(skf_list, nl=1, with_dtype=False))
[
    (np.array([2, 3, 4, 5, 6]), np.array([0, 1, 7, 8])),
    (np.array([0, 1, 2, 7, 8]), np.array([3, 4, 5, 6])),
    (np.array([0, 1, 3, 4, 5, 6, 7, 8]), np.array([2])),
]
```

_iter_test_masks(*X*, *y=None*, *groups=None*)

split(*X*, *y*, *groups=None*)
: Generate indices to split data into training and test set.

_abc_impl = <_abc._abc_data object>

kwcoco.util.**ensure_json_serializable**(*dict_*, *normalize_containers=False*, *verbose=0*)
: Attempt to convert common types (e.g. numpy) into something json complient

Convert numpy and tuples into lists

Parameters
normalize_containers (*bool*) – if True, normalizes dict containers to be standard python structures. Defaults to False.

Example

```
>>> data = ub.ddict(lambda: int)
>>> data['foo'] = ub.ddict(lambda: int)
>>> data['bar'] = np.array([1, 2, 3])
>>> data['foo']['a'] = 1
>>> data['foo']['b'] = (1, np.array([1, 2, 3]), {3: np.int32(3), 4: np.float16(1.0)}
↪)
>>> dict_ = data
>>> print(ub.urepr(data, nl=-1))
>>> assert list(find_json_unserializable(data))
>>> result = ensure_json_serializable(data, normalize_containers=True)
>>> print(ub.urepr(result, nl=-1))
>>> assert not list(find_json_unserializable(result))
>>> assert type(result) is dict
```

kwcoco.util.**find_json_unserializable**(*data*, *quickcheck=False*)

Recurse through json datastructure and find any component that causes a serialization error. Record the location of these errors in the datastructure as we recurse through the call tree.

Parameters

- **data** (*object*) – data that should be json serializable
- **quickcheck** (*bool*) – if True, check the entire datastructure assuming its ok before doing the python-based recursive logic.

Returns

list of "bad part" dictionaries containing items

'value' - the value that caused the serialization error

'loc' - which contains a list of key/indexes that can be used to lookup the location of the unserializable value. If the "loc" is a list, then it indicates a rare case where a key in a dictionary is causing the serialization error.

Return type
List[Dict]

Example

```
>>> from kwcoco.util.util_json import *  # NOQA
>>> part = ub.ddict(lambda: int)
>>> part['foo'] = ub.ddict(lambda: int)
>>> part['bar'] = np.array([1, 2, 3])
>>> part['foo']['a'] = 1
>>> # Create a dictionary with two unserializable parts
>>> data = [1, 2, {'nest1': [2, part]}, {frozenset({'badkey'}): 3, 2: 4}]
>>> parts = list(find_json_unserializable(data))
>>> print('parts = {}'.format(ub.urepr(parts, nl=1)))
```

(continues on next page)

(continued from previous page)

```
>>> # Check expected structure of bad parts
>>> assert len(parts) == 2
>>> part = parts[1]
>>> assert list(part['loc']) == [2, 'nest1', 1, 'bar']
>>> # We can use the "loc" to find the bad value
>>> for part in parts:
>>>     # "loc" is a list of directions containing which keys/indexes
>>>     # to traverse at each descent into the data structure.
>>>     directions = part['loc']
>>>     curr = data
>>>     special_flag = False
>>>     for key in directions:
>>>         if isinstance(key, list):
>>>             # special case for bad keys
>>>             special_flag = True
>>>             break
>>>         else:
>>>             # normal case for bad values
>>>             curr = curr[key]
>>>     if special_flag:
>>>         assert part['data'] in curr.keys()
>>>         assert part['data'] is key[1]
>>>     else:
>>>         assert part['data'] is curr
```

Example

```
>>> # xdoctest: +SKIP("TODO: circular ref detect algo is wrong, fix it")
>>> from kwcoco.util.util_json import *  # NOQA
>>> import pytest
>>> # Test circular reference
>>> data = [[], {'a': []}]
>>> data[1]['a'].append(data)
>>> with pytest.raises(ValueError, match="Circular reference detected at.*1, 'a', 1*
↪"):
...     parts = list(find_json_unserializable(data))
>>> # Should be ok here
>>> shared_data = {'shared': 1}
>>> data = [[shared_data], shared_data]
>>> parts = list(find_json_unserializable(data))
```

kwcoco.util.**indexable_allclose**(*dct1*, *dct2*, *return_info=False*)

Walks through two nested data structures and ensures that everything is roughly the same.

Note: Use the version in ubelt instead

Parameters

- **dct1** – a nested indexable item
- **dct2** – a nested indexable item

Example

```
>>> from kwcoco.util.util_json import indexable_allclose
>>> dct1 = {
>>>     'foo': [1.222222, 1.333],
>>>     'bar': 1,
>>>     'baz': [],
>>> }
>>> dct2 = {
>>>     'foo': [1.22222, 1.333],
>>>     'bar': 1,
>>>     'baz': [],
>>> }
>>> assert indexable_allclose(dct1, dct2)
```

kwcoco.util.**resolve_directory_symlinks**(*path*)

Only resolve symlinks of directories, not the base file

kwcoco.util.**resolve_relative_to**(*path*, *dpath*, *strict=False*)

Given a path, try to resolve its symlinks such that it is relative to the given dpath.

Example

```
>>> from kwcoco.util.util_reroot import *  # NOQA
>>> import os
>>> def _symlink(self, target, verbose=0):
>>>     return ub.Path(ub.symlink(target, self, verbose=verbose))
>>> ub.Path._symlink = _symlink
>>> #
>>> # TODO: try to enumerate all basic cases
>>> #
>>> base = ub.Path.appdir('kwcoco/tests/reroot')
>>> base.delete().ensuredir()
>>> #
>>> drive1 = (base / 'drive1').ensuredir()
>>> drive2 = (base / 'drive2').ensuredir()
>>> #
>>> data_repo1 = (drive1 / 'data_repo1').ensuredir()
>>> cache = (data_repo1 / '.cache').ensuredir()
>>> real_file1 = (cache / 'real_file1').touch()
>>> #
>>> real_bundle = (data_repo1 / 'real_bundle').ensuredir()
>>> real_assets = (real_bundle / 'assets').ensuredir()
>>> #
>>> # Symlink file outside of the bundle
>>> link_file1 = (real_assets / 'link_file1')._symlink(real_file1)
>>> real_file2 = (real_assets / 'real_file2').touch()
>>> link_file2 = (real_assets / 'link_file2')._symlink(real_file2)
>>> #
>>> #
>>> # A symlink to the data repo
>>> data_repo2 = (drive1 / 'data_repo2')._symlink(data_repo1)
```

(continues on next page)

(continued from previous page)

```
>>> data_repo3 = (drive2 / 'data_repo3')._symlink(data_repo1)
>>> data_repo4 = (drive2 / 'data_repo4')._symlink(data_repo2)
>>> #
>>> # A prediction repo TODO
>>> pred_repo5 = (drive2 / 'pred_repo5').ensuredir()
>>> #
>>> # _ = ub.cmd(f'tree -a {base}', verbose=3)
>>> #
>>> fpaths = []
>>> for r, ds, fs in os.walk(base, followlinks=True):
>>>     for f in fs:
>>>         if 'file' in f:
>>>             fpath = ub.Path(r) / f
>>>             fpaths.append(fpath)
>>> #
>>> #
>>> dpath = real_bundle.resolve()
>>> #
>>> for path in fpaths:
>>>     # print(f'{path}')
>>>     # print(f'{path.resolve()=}')
>>>     resolved_rel = resolve_relative_to(path, dpath)
>>>     print('resolved_rel = {!r}'.format(resolved_rel))
```

kwcoco.util.**smart_truncate**(*string*, *max_length=0*, *separator=' '*, *trunc_loc=0.5*, *trunc_char='~'*)

Truncate a string. :param string (str): string for modification :param max_length (int): output string length :param word_boundary (bool): :param save_order (bool): if True then word order of output string is like input string :param separator (str): separator between words :param trunc_loc (float): fraction of location where to remove the text

trunc_char (str): the character to denote where truncation is starting

Returns

kwcoco.util.**special_reroot_single**(*dset*, *verbose=0*)

kwcoco.util.**unarchive_file**(*archive_fpath*, *output_dpath='.'*, *verbose=1*, *overwrite=True*)

2.1.2 Submodules

2.1.2.1 kwcoco.__main__ module

2.1.2.2 kwcoco._helpers module

These items were split out of coco_dataset.py which is becoming too big

These are helper data structures used to do things like auto-increment ids, recycle ids, do renaming, extend sortedcontainers etc…

class kwcoco._helpers.**_NextId**(*parent*)

Bases: object

Helper class to tracks unused ids for new items

_update_unused(*key*)

Scans for what the next safe id can be for `key`

get(*key*)

Get the next safe item id for `key`

class kwcoco._helpers.**_ID_Remapper**(*reuse=False*)

Bases: `object`

Helper to recycle ids for unions.

For each dataset we create a mapping between each old id and a new id. If possible and reuse=True we allow the new id to match the old id. After each dataset is finished we mark all those ids as used and subsequent new-ids cannot be chosen from that pool.

Parameters

reuse (*bool*) – if True we are allowed to reuse ids as long as they haven't been used before.

Example

```
>>> video_trackids = [[1, 1, 3, 3, 200, 4], [204, 1, 2, 3, 3, 4, 5, 9]]
>>> self = _ID_Remapper(reuse=True)
>>> for tids in video_trackids:
>>>     new_tids = [self.remap(old_tid) for old_tid in tids]
>>>     self.block_seen()
>>>     print('new_tids = {!r}'.format(new_tids))
new_tids = [1, 1, 3, 3, 200, 4]
new_tids = [204, 205, 2, 206, 206, 207, 5, 9]
>>> #
>>> self = _ID_Remapper(reuse=False)
>>> for tids in video_trackids:
>>>     new_tids = [self.remap(old_tid) for old_tid in tids]
>>>     self.block_seen()
>>>     print('new_tids = {!r}'.format(new_tids))
new_tids = [0, 0, 1, 1, 2, 3]
new_tids = [4, 5, 6, 7, 7, 8, 9, 10]
```

remap(*old_id*)

Convert a old-id into a new-id. If self.reuse is True then we will return the same id if it hasn't been blocked yet.

block_seen()

Mark all seen ids as unable to be used. Any ids sent to remap will now generate new ids.

next_id()

Generate a new id that hasnt been used yet

class kwcoco._helpers.**UniqueNameRemapper**

Bases: `object`

helper to ensure names will be unique by appending suffixes

Example

```
>>> from kwcoco.coco_dataset import *  # NOQA
>>> self = UniqueNameRemapper()
>>> assert self.remap('foo') == 'foo'
>>> assert self.remap('foo') == 'foo_v001'
>>> assert self.remap('foo') == 'foo_v002'
>>> assert self.remap('foo_v001') == 'foo_v003'
>>> assert 'foo' in self
```

remap(*name*)

kwcoco._helpers.**_lut_image_frame_index**(*imgs*, *gid*)

kwcoco._helpers.**_lut_frame_index**(*imgs*, *gid*)

kwcoco._helpers.**_lut_annot_frame_index**(*imgs*, *anns*, *aid*)

class kwcoco._helpers.**SortedSet**(*iterable=None*, *key=None*)

Bases: SortedSet

Initialize sorted set instance.

Optional *iterable* argument provides an initial iterable of values to initialize the sorted set.

Optional *key* argument defines a callable that, like the *key* argument to Python's *sorted* function, extracts a comparison key from each value. The default, none, compares values directly.

Runtime complexity: *O(n*log(n))*

```
>>> ss = SortedSet([3, 1, 2, 5, 4])
>>> ss
SortedSet([1, 2, 3, 4, 5])
>>> from operator import neg
>>> ss = SortedSet([3, 1, 2, 5, 4], neg)
>>> ss
SortedSet([5, 4, 3, 2, 1], key=<built-in function neg>)
```

Parameters

- **iterable** – initial values (optional)
- **key** – function used to extract comparison key (optional)

_abc_impl = <_abc._abc_data object>

kwcoco._helpers.**SortedSetQuiet**

alias of *SortedSet*

kwcoco._helpers.**_delitems**(*items*, *remove_idxs*, *thresh=750*)

Parameters

- **items** (*List*) – list which will be modified
- **remove_idxs** (*List[int]*) – integers to remove (MUST BE UNIQUE)

kwcoco._helpers.**_load_and_postprocess**(*data*, *loader*, *postprocess*, ***loadkw*)

kwcoco._helpers.**_image_corruption_check**(*fpath*, *only_shape=False*)

2.1.2.3 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.

_abc_impl = <_abc._abc_data object>

2.1.2.4 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)
```

Parameters

- **graph** (*nx.DiGraph*) – either the graph representing a category hierarchy
- **checks** (*bool, default=True*) – if false, bypass input checks

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

Parameters
node (*str*) – the name of the category

Returns
int

`take(`*indexes*`)`

Create a subgraph based on the selected class indexes

`subgraph(`*subnodes*, *closure=True*`)`

Create a subgraph based on the selected class nodes (i.e. names)

Example

```
>>> self = CategoryTree.from_coco([
>>>     {'id': 130, 'name': 'n3', 'supercategory': 'n1'},
>>>     {'id': 410, 'name': 'n1', 'supercategory': None},
>>>     {'id': 640, 'name': 'n4', 'supercategory': 'n3'},
>>>     {'id': 220, 'name': 'n2', 'supercategory': 'n1'},
>>>     {'id': 560, 'name': 'n6', 'supercategory': 'n2'},
>>>     {'id': 350, 'name': 'n5', 'supercategory': 'n2'},
>>> ])
>>> self.print_graph()
>>> subnodes = ['n3', 'n6', 'n4', 'n1']
>>> new1 = self.subgraph(subnodes, closure=1)
>>> new1.print_graph()
...
>>> print('new1.idx_to_id = {}'.format(ub.urepr(new1.idx_to_id, nl=0)))
>>> print('new1.idx_to_node = {}'.format(ub.urepr(new1.idx_to_node, nl=0)))
new1.idx_to_id = [130, 560, 640, 410]
new1.idx_to_node = ['n3', 'n6', 'n4', 'n1']
```

```
>>> indexes = [2, 1, 0, 5]
>>> new2 = self.take(indexes)
>>> new2.print_graph()
...
>>> print('new2.idx_to_id = {}'.format(ub.urepr(new2.idx_to_id, nl=0)))
>>> print('new2.idx_to_node = {}'.format(ub.urepr(new2.idx_to_node, nl=0)))
new2.idx_to_id = [640, 410, 130, 350]
new2.idx_to_node = ['n4', 'n1', 'n3', 'n5']
```

```
>>> subnodes = ['n3', 'n6', 'n4', 'n1']
>>> new3 = self.subgraph(subnodes, closure=0)
>>> new3.print_graph()
```

_build_index()

construct lookup tables

show()

forest_str()

print_graph()

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()
```

2.1.2.5 kwcoco.channel_spec module

The ChannelSpec and FusedChannelSpec represent a set of channels or bands in an image. This could be as simple as `red|green|blue`, or more complex like: `red|green|blue|nir|swir16|swir22`.

This functionality has been moved to "delayed_image".

2.1.2.6 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*.

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.

An informal description of the spec given in: coco_schema_informal.rst.

For a formal description of the spec see the coco_schema.json, which is generated by :py:mod`kwcoco/coco_schema`.

Todo:

- **[ ] Use ijson (modified to support NaN) 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.
- [x] Document channel spec
- [x] Document sensor-channel spec
- [X] Add remove videos method
- **[ ] Efficiency: Make video annotations more efficient by only tracking**
  keyframes, provide an API to obtain a dense or interpolated annotation on an intermediate frame.
- **[ ] Efficiency: Allow each section of the kwcoco file to be written as a**
  separate json file. Perhaps allow genric pointer support? Might get messy.
- [ ] Reroot needs to be redesigned very carefully.
- [ ] Allow parts of the kwcoco file to be references to other json files.

- [X] Add top-level track table
- [ ] Fully transition to integer track ids (in progress)

References

class kwcoco.coco_dataset.**MixinCocoDepricate**

Bases: object

These functions are marked for deprication and will be removed

keypoint_annotation_frequency()

DEPRECATED

Example

```
>>> import kwcoco
>>> import ubelt as ub
>>> self = kwcoco.CocoDataset.demo('shapes', rng=0)
>>> hist = self.keypoint_annotation_frequency()
>>> hist = ub.odict(sorted(hist.items()))
>>> # FIXME: for whatever reason demodata generation is not determenistic when␣
↪seeded
>>> print(ub.urepr(hist))  # xdoc: +IGNORE_WANT
{
    'bot_tip': 6,
    'left_eye': 14,
    'mid_tip': 6,
    'right_eye': 14,
    'top_tip': 6,
}
```

category_annotation_type_frequency()

DEPRECATED

Reports the number of annotations of each type for each category

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> hist = self.category_annotation_frequency()
>>> print(ub.urepr(hist))
```

imread(*gid*)

DEPRECATED: use load_image or delayed_image

Loads a particular image

class kwcoco.coco_dataset.**MixinCocoAccessors**

Bases: object

TODO: better name

delayed_load(*gid*, *channels=None*, *space='image'*)

Experimental method

Parameters

- **gid** (*int*) – image id to load
- **channels** (*kwcoco.FusedChannelSpec*) – specific channels to load. if unspecified, all channels are loaded.
- **space** (*str*) – can either be "image" for loading in image space, or "video" for loading in video space.

Todo:

- **[X] Currently can only take all or none of the channels from each**
  base-image / auxiliary dict. For instance if the main image is r|g|b you can't just select g|b at the moment.
- **[X] The order of the channels in the delayed load should**
  match the requested channel order.
- [X] TODO: add nans to bands that don't exist or throw an error

---

Example

```
>>> import kwcoco
>>> gid = 1
>>> #
>>> self = kwcoco.CocoDataset.demo('vidshapes8-multispectral')
>>> delayed = self.delayed_load(gid)
>>> print('delayed = {!r}'.format(delayed))
>>> print('delayed.finalize() = {!r}'.format(delayed.finalize()))
>>> #
>>> self = kwcoco.CocoDataset.demo('shapes8')
>>> delayed = self.delayed_load(gid)
>>> print('delayed = {!r}'.format(delayed))
>>> print('delayed.finalize() = {!r}'.format(delayed.finalize()))
```

```
>>> crop = delayed.crop((slice(0, 3), slice(0, 3)))
>>> crop.finalize()
```

```
>>> # TODO: should only select the "red" channel
>>> self = kwcoco.CocoDataset.demo('shapes8')
>>> delayed = self.delayed_load(gid, channels='r')
```

```
>>> import kwcoco
>>> gid = 1
>>> #
>>> self = kwcoco.CocoDataset.demo('vidshapes8-multispectral')
>>> delayed = self.delayed_load(gid, channels='B1|B2', space='image')
>>> print('delayed = {!r}'.format(delayed))
>>> delayed = self.delayed_load(gid, channels='B1|B2|B11', space='image')
```

(continues on next page)

(continued from previous page)

```
>>> 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

Note: Prefer to use the CocoImage methods instead

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* | *None*) – if specified, return a path to data containing auxiliary channels instead

Note: Prefer to use the CocoImage methods instead

Returns
full path to the image

Return type
PathLike

_get_img_auxiliary(*gid_or_img*, *channels*)

returns the auxiliary dictionary for a specific channel

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)

Note: Prefer to use the CocoImage methods instead

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 deprecate?

Parameters

- **aid_or_int** (*int* | *dict*) – annot id or dict
- **image** (*ArrayLike* | *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()
```

_resolve_to_id(*id_or_dict*)
Ensures output is an id

_resolve_to_cid(*id_or_name_or_dict*)
Ensures output is an category id

Note: this does not resolve aliases (yet), for that see _alias_to_cat

Todo: we could maintain an alias index to make this fast

_resolve_to_gid(*id_or_name_or_dict*)
Ensures output is an category id

_resolve_to_vidid(*id_or_name_or_dict*)
Ensures output is an video id

_resolve_to_trackid(*id_or_name_or_dict*)

_resolve_to_ann(*aid_or_ann*)
Ensures output is an annotation dictionary

_resolve_to_img(*gid_or_img*)
Ensures output is an image dictionary

_resolve_to_kpcat(*kp_identifier*)
Lookup a keypoint-category dict via its name or id

Parameters
kp_identifier (*int | str | dict*) – either the keypoint category name, alias, or its keypoint_category_id.

Returns
keypoint category dictionary

Return type
Dict

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo('shapes')
>>> kpcat1 = self._resolve_to_kpcat(1)
>>> kpcat2 = self._resolve_to_kpcat('left_eye')
>>> assert kpcat1 is kpcat2
>>> import pytest
>>> with pytest.raises(KeyError):
>>>     self._resolve_to_cat('human')
```

`_resolve_to_cat`(*cat_identifier*)

Lookup a coco-category dict via its name, alias, or id.

Parameters
cat_identifier (*int | str | dict*) – either the category name, alias, or its category_id.

Raises
`KeyError` – if the category doesn't exist.

Note: If the index is not built, the method will work but may be slow.

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> cat = self._resolve_to_cat('human')
>>> import pytest
>>> assert self._resolve_to_cat(cat['id']) is cat
>>> assert self._resolve_to_cat(cat) is cat
>>> with pytest.raises(KeyError):
>>>     self._resolve_to_cat(32)
>>> self.index.clear()
>>> assert self._resolve_to_cat(cat['id']) is cat
>>> with pytest.raises(KeyError):
>>>     self._resolve_to_cat(32)
```

`_alias_to_cat`(*alias_catname*)

Lookup a coco-category via its name or an "alias" name. In production code, use `_resolve_to_cat()` instead.

Parameters
alias_catname (*str*) – category name or alias

Returns
coco category dictionary

Return type
dict

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> cat = self._alias_to_cat('human')
>>> import pytest
>>> with pytest.raises(KeyError):
>>>     self._alias_to_cat('person')
>>> cat['alias'] = ['person']
>>> self._alias_to_cat('person')
>>> cat['alias'] = 'person'
>>> self._alias_to_cat('person')
>>> assert self._alias_to_cat(None) is None
```

category_graph()

Construct a networkx category hierarchy

Returns
graph: a directed graph where category names are the nodes, supercategories define edges, and items in each category dict (e.g. category id) are added as node properties.

Return type
networkx.DiGraph

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> graph = self.category_graph()
>>> assert 'astronaut' in graph.nodes()
>>> assert 'keypoints' in graph.nodes['human']
```

object_categories()

Construct a consistent CategoryTree representation of object classes

Returns
category data structure

Return type
kwcoco.CategoryTree

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> classes = self.object_categories()
>>> print('classes = {}'.format(classes))
```

keypoint_categories()

Construct a consistent CategoryTree representation of keypoint classes

Returns
category data structure

Return type
kwcoco.CategoryTree

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> classes = self.keypoint_categories()
>>> print('classes = {}'.format(classes))
```

_keypoint_category_names()

Construct keypoint categories names.

Uses new-style if possible, otherwise this falls back on old-style.

Returns
names - list of keypoint category names

Return type
List[str]

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> names = self._keypoint_category_names()
>>> print(names)
```

_lookup_kpnames(*cid*)

Get the keypoint categories for a certain class

_coco_image(*gid*)

coco_image(*gid*)

Parameters
gid (*int*) – image id

Returns
CocoImage

class kwcoco.coco_dataset.**MixinCocoConstructors**

Bases: `object`

Classmethods for constructing CocoDataset objects

classmethod coerce(*key*, *sqlview=False*, ***kw*)

Attempt to transform the input into the intended CocoDataset.

Parameters

- **key** – this can either be an instance of a CocoDataset, a string URI pointing to an on-disk dataset, or a special key for creating demodata.
- **sqlview** (*bool | str*) – If truthy, will return the dataset as a cached sql view, which can be quicker to load and use in some instances. Can be given as a string, which sets the backend that is used: either sqlite or postgresql. Defaults to False.
- ****kw** – passed to whatever constructor is chosen (if any)

Returns

AbstractCocoDataset | kwcoco.CocoDataset | kwcoco.CocoSqlDatabase

Example

```
>>> # test coerce for various input methods
>>> import kwcoco
>>> from kwcoco.coco_sql_dataset import assert_dsets_allclose
>>> dct_dset = kwcoco.CocoDataset.coerce('special:shapes8')
>>> copy1 = kwcoco.CocoDataset.coerce(dct_dset)
>>> copy2 = kwcoco.CocoDataset.coerce(dct_dset.fpath)
>>> assert assert_dsets_allclose(dct_dset, copy1)
>>> assert assert_dsets_allclose(dct_dset, copy2)
>>> # xdoctest: +REQUIRES(module:sqlalchemy)
>>> sql_dset = dct_dset.view_sql()
>>> copy3 = kwcoco.CocoDataset.coerce(sql_dset)
>>> copy4 = kwcoco.CocoDataset.coerce(sql_dset.fpath)
>>> assert assert_dsets_allclose(dct_dset, sql_dset)
>>> assert assert_dsets_allclose(dct_dset, copy3)
>>> assert assert_dsets_allclose(dct_dset, copy4)
```

classmethod demo(*key='photos'*, ***kwargs*)

Create a toy coco dataset for testing and demo puposes

Parameters

- **key** (*str*) – Either 'photos' (default), '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.

(5) vidshapes2-tracks5 - generate 2 videos with 5 tracks each.

(6) 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()`.

  Notable options are:
  bundle_dpath (PathLike): location to write the demo bundle fpath (PathLike): location to write the demo kwcoco file

Kwargs:
image_size (Tuple[int, int]): width / height size of the images

dpath (str | PathLike):
path to the directory where any generated demo bundles will be written to. Defaults to using kwcoco cache dir.

aux (bool): if True generates dummy auxiliary channels

rng (int | RandomState | None):
random number generator or seed

verbose (int): verbosity mode. Defaults to 3.

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))
```

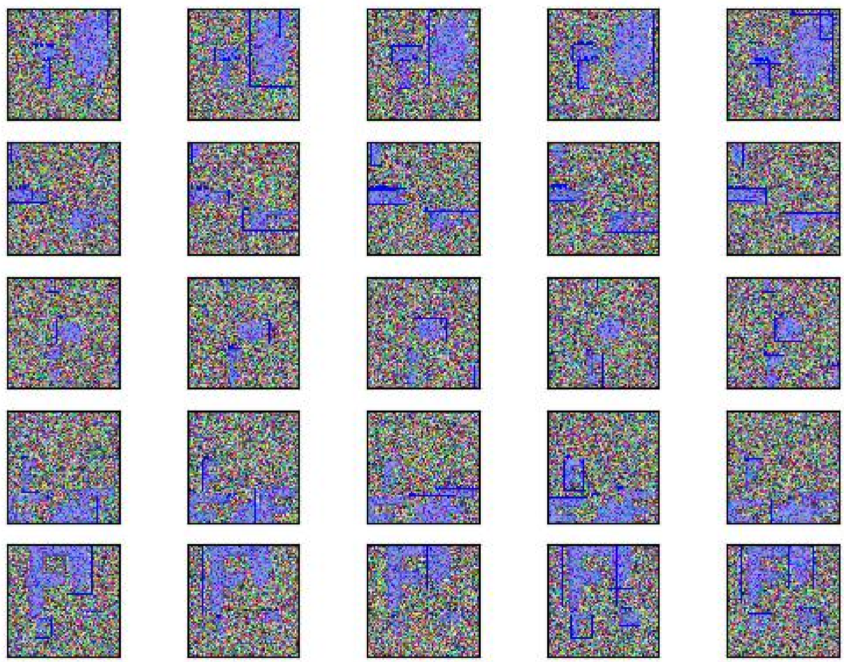
Example

```
>>> 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
```

(continues on next page)

(continued from previous page)

```
>>> 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])
```

(continues on next page)

(continued from previous page)

```
>>> 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

---

class kwcoco.coco_dataset.**MixinCocoExtras**

Bases: object

Misc functions for coco

_tree()

developer helper

_dataset_id()

A human interpretable name that can be used to uniquely identify the dataset.

Note: This function is currently subject to change.

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> print(self._dataset_id())
>>> self = kwcoco.CocoDataset.demo('vidshapes8')
>>> print(self._dataset_id())
>>> self = kwcoco.CocoDataset()
>>> print(self._dataset_id())
```

_ensure_imgsize(*workers=0*, *verbose=1*, *fail=False*)

Populate the imgsize field if it does not exist.

Parameters

- **workers** (*int*) – number of workers for parallel processing.
- **verbose** (*int*) – verbosity level
- **fail** (*bool*) – if True, raises an exception if anything size fails to load.

Returns

a list of "bad" image dictionaries where the size could
not be determined. Typically these are corrupted images and should be removed.

Return type

List[dict]

Example

```
>>> # Normal case
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> bad_imgs = self._ensure_imgsize()
>>> assert len(bad_imgs) == 0
>>> assert self.imgs[1]['width'] == 512
>>> assert self.imgs[2]['width'] == 328
>>> assert self.imgs[3]['width'] == 256
```

```
>>> # Fail cases
>>> self = kwcoco.CocoDataset()
>>> self.add_image('does-not-exist.jpg')
>>> bad_imgs = self._ensure_imgsize()
>>> assert len(bad_imgs) == 1
>>> import pytest
>>> with pytest.raises(Exception):
>>>     self._ensure_imgsize(fail=True)
```

_ensure_image_data(*gids=None*, *verbose=1*)

Download data from "url" fields if specified.

Parameters
gids (*List*) – subset of images to download

missing_images(*check_aux=True*, *verbose=0*)

Check for images that don't exist

Parameters

- **check_aux** (*bool*) – 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=True*, *verbose=0*, *workers=0*)

Check for images that don't exist or can't be opened

Parameters

- **check_aux** (*bool*) – if specified also checks auxiliary images
- **verbose** (*int*) – verbosity level
- **workers** (*int*) – number of background workers

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)
```

_ensure_json_serializable()

_aspycoco()

Converts to the official pycocotools.coco.COCO object

Todo:

- [ ] Maybe expose as a public API?

---

reroot(*new_root=None*, *old_prefix=None*, *new_prefix=None*, *absolute=False*, *check=True*, *safe=True*, *verbose=1*)

Modify the prefix of the image/data paths onto a new image/data root.

Parameters

- **new_root** (*str | PathLike | 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

```
>>> # xdoctest: +REQUIRES(module:rich)
>>> import kwcoco
>>> import ubelt as ub
>>> import rich
>>> def report(dset):
>>>     gid = 1
>>>     abs_fpath = ub.Path(dset.get_image_fpath(gid))
>>>     rel_fpath = dset.index.imgs[gid]['file_name']
>>>     color = 'green' if abs_fpath.exists() else 'red'
>>>     print(ub.color_text(f'abs_fpath = {abs_fpath!r}', color))
>>>     print(f'rel_fpath = {rel_fpath!r}')
>>> dset = self = kwcoco.CocoDataset.demo()
>>> # Change base relative directory
>>> bundle_dpath = ub.expandpath('~')
>>> rich.print('ORIG self.imgs = {}'.format(ub.urepr(self.imgs, nl=1)))
>>> rich.print('ORIG dset.bundle_dpath = {!r}'.format(dset.bundle_dpath))
>>> rich.print('NEW(1) bundle_dpath       = {!r}'.format(bundle_dpath))
>>> # Test relative reroot
>>> rich.print('[blue] --- 1. RELATIVE REROOT ---')
>>> self.reroot(bundle_dpath, verbose=3)
>>> report(self)
>>> rich.print('NEW(1) self.imgs = {}'.format(ub.urepr(self.imgs, nl=1)))
>>> if not ub.WIN32:
>>>     assert self.imgs[1]['file_name'].startswith('.cache')
>>> # Test absolute reroot
>>> rich.print('[blue] --- 2. ABSOLUTE REROOT ---')
>>> self.reroot(absolute=True, verbose=3)
>>> rich.print('NEW(2) self.imgs = {}'.format(ub.urepr(self.imgs, nl=1)))
>>> assert self.imgs[1]['file_name'].startswith(bundle_dpath)
```

```
>>> # Switch back to relative paths
>>> rich.print('[blue] --- 3. ABS->REL REROOT ---')
>>> self.reroot()
>>> rich.print('NEW(3) self.imgs = {}'.format(ub.urepr(self.imgs, nl=1)))
>>> if not ub.WIN32:
>>>     assert self.imgs[1]['file_name'].startswith('.cache')
```

Example

```
>>> # demo with auxiliary data
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo('shapes8', aux=True)
>>> bundle_dpath = ub.expandpath('~')
>>> print(self.imgs[1]['file_name'])
>>> print(self.imgs[1]['auxiliary'][0]['file_name'])
>>> self.reroot(new_root=bundle_dpath)
>>> print(self.imgs[1]['file_name'])
>>> print(self.imgs[1]['auxiliary'][0]['file_name'])
>>> if not ub.WIN32:
>>>     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.MixinCocoHashing

Bases: `object`

Mixin for creating and maintaining hashids (i.e. content identifiers)

_build_hashid(*hash_pixels=False*, *verbose=0*)

Construct a hash that uniquely identifies the state of this dataset.

Parameters

- **hash_pixels** (*bool*) – If False the image data is not included in the hash, which can speed up computation, but is not 100% robust. Defaults to False.
- **verbose** (*int*) – verbosity level

Returns
the hashid

Return type
str

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> self._build_hashid(hash_pixels=True, verbose=3)
...
>>> # Shorten hashes for readability
>>> import ubelt as ub
>>> walker = ub.IndexableWalker(self.hashid_parts)
>>> for path, val in walker:
>>>     if isinstance(val, str):
>>>         walker[path] = val[0:8]
>>> # Note: this may change in different versions of kwcoco
>>> print('self.hashid_parts = ' + ub.urepr(self.hashid_parts))
>>> print('self.hashid = {!r}'.format(self.hashid[0:8]))
self.hashid_parts = {
    'annotations': {
        'json': 'c1d1b9c3',
        'num': 11,
    },
    'images': {
        'pixels': '88e37cc3',
        'json': '9b8e8be3',
        'num': 3,
    },
    'categories': {
        'json': '82d22e00',
        'num': 8,
    },
}
self.hashid = 'bf69bf15'
```

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> self._build_hashid(hash_pixels=True, verbose=3)
>>> self.hashid_parts
>>> # Test that when we modify the dataset only the relevant
>>> # hashid parts are recomputed.
>>> orig = self.hashid_parts['categories']['json']
>>> self.add_category('foobar')
>>> assert 'categories' not in self.hashid_parts
>>> self.hashid_parts
>>> self.hashid_parts['images']['json'] = 'should not change'
>>> self._build_hashid(hash_pixels=True, verbose=3)
>>> assert self.hashid_parts['categories']['json']
>>> assert self.hashid_parts['categories']['json'] != orig
>>> assert self.hashid_parts['images']['json'] == 'should not change'
```

_invalidate_hashid(*parts=None*)

Called whenever the coco dataset is modified. It is possible to specify which parts were modified so unmodified parts can be reused next time the hash is constructed.

Todo:

- [ ] Rename to _notify_modification — or something like that

---

`_cached_hashid()`

Under Construction.

The idea is to cache the hashid when we are sure that the dataset was loaded from a file and has not been modified. We can record the modification time of the file (because we know it hasn't changed in memory), and use that as a key to the cache. If the modification time on the file is different than the one recorded in the cache, we know the cache could be invalid, so we recompute the hashid.

Todo:

- [ ] This is reasonably stable, elevate this to a public API function.

`classmethod _cached_hashid_for(`*fpath*`)`

Lookup the cached hashid for a kwcoco json file if it exists.

`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(`*annot_ids=None*, *image_id=None*, *track_id=None*, *video_id=None*, *trackid=None*, *aids=None*, *gid=None*`)`

Return vectorized annotation objects

Parameters

- **annot_ids** (*List[int]* | *None*) – annotation ids to reference, if unspecified all annotations are returned. An alias is "aids", which may be removed in the future.
- **image_id** (*int* | *None*) – return all annotations that belong to this image id. Mutually exclusive with other arguments. An alias is "gids", which may be removed in the future.
- **track_id** (*int* | *None*) – return all annotations that belong to this track. mutually exclusive with other arguments. An alias is "trackid", which may be removed in the future.
- **video_id** (*int* | *None*) – return all annotations that belong to this video. 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.urepr(sub_annots.get('bbox', None)))
[
    [350, 5, 130, 290],
    None,
    None,
]
```

images(*image_ids=None*, *video_id=None*, *names=None*, *gids=None*, *vidid=None*)

Return vectorized image objects

Parameters

- **image_ids** (*List[int] | None*) – image ids to reference, if unspecified all images are returned. An alias is *gids*.
- **video_id** (*int | None*) – returns all images that belong to this video id. mutually exclusive with *image_ids* arg.
- **names** (*List[str] | None*) – lookup images by their names.

Returns
vectorized image object

Return type
kwcoco.coco_objects1d.Images

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> images = self.images()
>>> print(images)
<Images(num=3)>
```

```
>>> self = kwcoco.CocoDataset.demo('vidshapes2')
>>> video_id = 1
>>> images = self.images(video_id=video_id)
>>> assert all(v == video_id for v in images.lookup('video_id'))
>>> print(images)
<Images(num=2)>
```

categories(*category_ids=None*, *cids=None*)

Return vectorized category objects

Parameters
category_ids (*List[int] | None*) – category ids to reference, if unspecified all categories are returned. The *cids* argument is an alias.

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(*video_ids=None*, *names=None*, *vidids=None*)

Return vectorized video objects

Parameters

- **video_ids** (*List[int] | None*) – video ids to reference, if unspecified all videos are returned. The *vidids* argument is an alias. Mutually exclusive with other args.
- **names** (*List[str] | None*) – lookup videos by their name. Mutually exclusive with other args.

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.urepr(videos.objs[0:2], nl=1)))
```

tracks(*track_ids=None*, *names=None*)

Return vectorized track objects

Parameters

- **track_ids** (*List[int] | None*) – track ids to reference, if unspecified all tracks are returned.
- **names** (*List[str] | None*) – lookup tracks by their name. Mutually exclusive with other args.

Returns

vectorized video object

Return type

kwcoco.coco_objects1d.Tracks

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo('vidshapes2')
>>> tracks = self.tracks()
>>> print(tracks)
>>> tracks.lookup('name')
>>> tracks.lookup('id')
>>> print('tracks.objs = {}'.format(ub.urepr(tracks.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_tracks

The number of tracks in the dataset

property n_videos

The number of videos in the dataset

category_annotation_frequency()

Reports the number of annotations of each category

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> hist = self.category_annotation_frequency()
>>> print(ub.urepr(hist))
{
    'astroturf': 0,
    'human': 0,
    'astronaut': 1,
    'astronomer': 1,
    'helmet': 1,
    'rocket': 1,
    'mouth': 2,
    'star': 5,
}
```

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

mmlab (default=False): if True tries to convert data to be compatible with open-mmlab tooling.

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

workers (int): number of workers for parallel checks. defaults to 0

fastfail (default=False): if True raise errors immediately

Returns

result containing keys -
status (bool): False if any errors occurred errors (List[str]): list of all error messages missing (List): List of any missing images corrupted (List): List of any corrupted images

Return type
dict

SeeAlso:
`_check_integrity()` - performs internal checks

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> import pytest
>>> with pytest.warns(UserWarning):
>>>     result = self.validate()
>>> assert not result['errors']
>>> assert result['warnings']
```

`stats`(***kwargs*)

Compute summary statistics to describe the dataset at a high level

This function corresponds to *kwcoco.cli.coco_stats*.

KWargs:
basic(bool): return basic stats', default=True extended(bool): return extended stats', default=True catfreq(bool): return category frequency stats', default=True boxes(bool): return bounding box stats', default=False

annot_attrs(bool): return annotation attribute information', default=True image_attrs(bool): return image attribute information', default=True

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.urepr(self.basic_stats()))
{
    'n_anns': 11,
    'n_imgs': 3,
    'n_videos': 0,
    'n_cats': 8,
    'n_tracks': 0,
}
```

```
>>> 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.urepr(dset.basic_stats()))
{
    'n_anns': 20,
    'n_imgs': 2,
    'n_videos': 1,
    'n_cats': 3,
    'n_tracks': 10,
}
```

extended_stats()

Reports number of images, annotations, and categories.

SeeAlso:
kwcoco.coco_dataset.MixinCocoStats.basic_stats() *kwcoco.coco_dataset.MixinCocoStats.extended_stats()*

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> print(ub.urepr(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* | *None*) – if specified also computes box anchors via KMeans clustering

- **perclass** (*bool*) – if True also computes stats for each category
- **gids** (*List[int] | None*) – if specified only compute stats for these image ids. Defaults to None.
- **aids** (*List[int] | None*) – if specified only compute stats for these annotation ids. Defaults to None.
- **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.urepr(infos, nl=-1, precision=2))
```

```
>>> infos = self.boxsize_stats(gids=[1], statskw=dict(median=True))
>>> print(ub.urepr(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

draw_image(*gid*, *channels=None*)

Use kwimage to draw all annotations on an image and return the pixels as a numpy array.

Parameters

- **gid** (*int*) – image id to draw
- **channels** (*kwcoco.ChannelSpec*) – the channel to draw on

Returns
canvas

Return type
ndarray

SeeAlso
kwcoco.coco_dataset.MixinCocoDraw.draw_image() *kwcoco.coco_dataset.MixinCocoDraw.show_image()*

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo('shapes8')
>>> self.draw_image(1)
>>> # Now you can dump the annotated image to disk / whatever
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(canvas)
```

show_image(*gid=None*, *aids=None*, *aid=None*, *channels=None*, *setlim=None*, ***kwargs*)

Use matplotlib to show an image with annotations overlaid

Parameters

- **gid** (*int | None*) – image id to show
- **aids** (*list | None*) – aids to highlight within the image
- **aid** (*int | None*) – a specific aid to focus on. If gid is not give, look up gid based on this aid.
- **setlim** (*None | str*) – if 'image' sets the limit to the image extent
- ****kwargs** – show_annots, show_aid, show_catname, show_kpname, show_segmentation, title, show_gid, show_filename, show_boxes,

SeeAlso

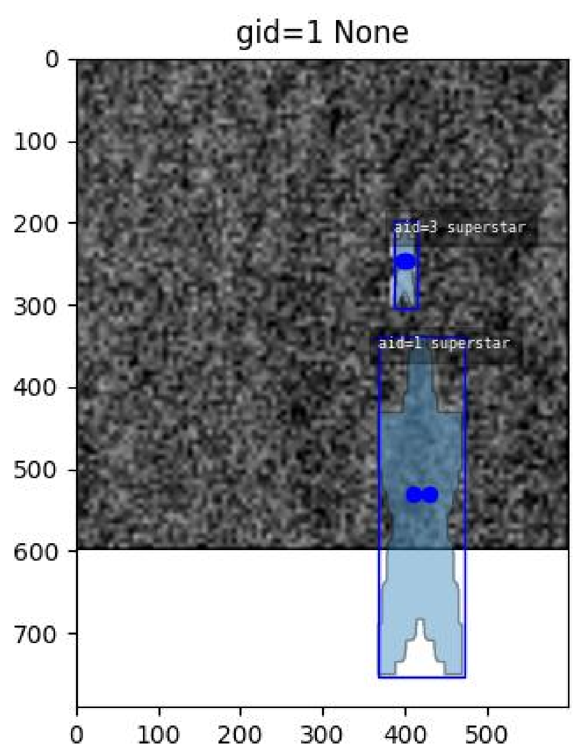
kwcoco.coco_dataset.MixinCocoDraw.draw_image() *kwcoco.coco_dataset.MixinCocoDraw.show_image()*

CommandLine

```
xdoctest -m kwcoco.coco_dataset MixinCocoDraw.show_image --show
```

Example

```
>>> # xdoctest: +REQUIRES(module:kwplot)
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('vidshapes8-msi')
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> # xdoctest: -REQUIRES(--show)
>>> dset.show_image(gid=1, channels='B8')
>>> # xdoctest: +REQUIRES(--show)
>>> kwplot.show_if_requested()
```



kwcoco.coco_dataset.**_normalize_intensity_if_needed**(*canvas*)

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.urepr(self.index.videos, nl=1)))
>>> print('self.index.imgs = {}'.format(ub.urepr(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.urepr(self.index.videos, nl=1)))
>>> print('self.index.imgs = {}'.format(ub.urepr(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.urepr(self.index.videos, nl=1)))
>>> print('self.index.imgs = {}'.format(ub.urepr(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 expect 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_asset()` 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
- **warp_img_to_vid** (*Dict*) – this transform is used to align the image to a video if it belongs to one.
- ****kw** – stores arbitrary key/value pairs in this new image

Returns
the image id assigned to the new image

Return type
int

SeeAlso:
add_image() add_images() ensure_image()

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> import kwimage
>>> gname = kwimage.grab_test_image_fpath('paraview')
>>> gid = self.add_image(gname)
>>> assert self.imgs[gid]['file_name'] == gname
```

add_asset(*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_asset()` for more details

Example

```
>>> import kwcoco
>>> dset = kwcoco.CocoDataset()
>>> gid = dset.add_image(name='my_image_name', width=200, height=200)
>>> dset.add_asset(gid, 'path/fake_B0.tif', channels='B0', width=200,
>>>                height=200, warp_aux_to_img={'scale': 1.0})
```

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_asset()` for more details

Example

```
>>> import kwcoco
>>> dset = kwcoco.CocoDataset()
>>> gid = dset.add_image(name='my_image_name', width=200, height=200)
>>> dset.add_asset(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*, *track_id=None*, ***kw*)

Register a new annotation with the dataset

Parameters

- **image_id** (*int*) – image_id the annotation is added to.
- **category_id** (*int | None*) – category_id for the new annotation
- **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.
- **track_id** (*int | str | None*) – Some value used to associate annotations that belong to the same "track". In the future we may remove support for strings.
- ****kw** – stores arbitrary key/value pairs in this new image, Common respected key/values include but are not limited to the following: 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.urepr(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.urepr(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.urepr(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.urepr(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.urepr(ann, nl=2)))
```

add_category(*name*, *supercategory=None*, *id=None*, ***kw*)

Register a new category with the dataset

Parameters

- **name** (*str*) – name of the new category
- **supercategory** (*str | None*) – parent of this category
- **id** (*int | None*) – use this category id, if it was not taken
- ****kw** – stores arbitrary key/value pairs in this new image

Returns

the category id assigned to the new category

Return type

int

SeeAlso:
kwcoco.coco_dataset.MixinCocoAddRemove.add_category() *kwcoco.coco_dataset.MixinCocoAddRemove.ensure_category()*

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> prev_n_cats = self.n_cats
>>> cid = self.add_category('dog', supercategory='object')
>>> assert self.cats[cid]['name'] == 'dog'
>>> assert self.n_cats == prev_n_cats + 1
>>> import pytest
>>> with pytest.raises(ValueError):
>>>     self.add_category('dog', supercategory='object')
```

add_track(*name*, *id=None*, ***kw*)

Register a new track with the dataset

Parameters

- **name** (*str*) – name of the new track
- **id** (*int | None*) – use this track id, if it was not taken
- ****kw** – stores arbitrary key/value pairs in this new image

Returns
the track id assigned to the new track

Return type
int

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> prev_n_tracks = self.n_tracks
>>> track_id = self.add_track('dog')
>>> assert self.index.tracks[track_id]['name'] == 'dog'
>>> assert self.n_tracks == prev_n_tracks + 1
>>> import pytest
>>> with pytest.raises(ValueError):
>>>     self.add_track('dog')
```

ensure_video(*name*, *id=None*, ***kw*)

Register a video if it is new or returns an existing id.

Like *kwcoco.coco_dataset.MixinCocoAddRemove.add_video()*, but returns the existing video id if it already exists instead of failing. In this case all metadata is ignored.

Parameters

- **file_name** (*str*) – relative or absolute path to video
- **id** (*None | int*) – ADVANCED. Force using this video id.

- ****kw** – stores arbitrary key/value pairs in this new video

Returns
the existing or new video id

Return type
int

SeeAlso:
kwcoco.coco_dataset.MixinCocoAddRemove.add_video() *kwcoco.coco_dataset.MixinCocoAddRemove.ensure_video()*

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> id1 = self.ensure_video('video1')
>>> id2 = self.ensure_video('video1')
>>> assert id1 == id2
```

ensure_track(*name*, *id=None*, ***kw*)

Register a track if it is new or returns an existing id.

Like *kwcoco.coco_dataset.MixinCocoAddRemove.add_track()*, but returns the existing track id if it already exists instead of failing. In this case all metadata is ignored.

Parameters

- **file_name** (*str*) – relative or absolute path to track
- **id** (*None | int*) – ADVANCED. Force using this track id.
- ****kw** – stores arbitrary key/value pairs in this new track

Returns
the existing or new track id

Return type
int

SeeAlso:
kwcoco.coco_dataset.MixinCocoAddRemove.add_track() *kwcoco.coco_dataset.MixinCocoAddRemove.ensure_track()*

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> track_id1 = self.ensure_track('dog')
>>> track_id2 = self.ensure_track('dog')
>>> assert track_id1 == track_id2
```

ensure_image(*file_name*, *id=None*, ***kw*)

Register an image if it is new or returns an existing id.

Like *kwcoco.coco_dataset.MixinCocoAddRemove.add_image()*, but returns the existing image id if it already exists instead of failing. In this case all metadata is ignored.

Parameters

- **file_name** (*str*) – relative or absolute path to image
- **id** (*None | int*) – ADVANCED. Force using this image id.
- ****kw** – stores arbitrary key/value pairs in this new image

Returns
the existing or new image id

Return type
int

SeeAlso:
kwcoco.coco_dataset.MixinCocoAddRemove.add_image() *kwcoco.coco_dataset.MixinCocoAddRemove.add_images()* *kwcoco.coco_dataset.MixinCocoAddRemove.ensure_image()*

ensure_category(*name*, *supercategory=None*, *id=None*, ***kw*)

Register a category if it is new or returns an existing id.

Like *kwcoco.coco_dataset.MixinCocoAddRemove.add_category()*, but returns the existing category id if it already exists instead of failing. In this case all metadata is ignored.

Returns
the existing or new category id

Return type
int

SeeAlso:
kwcoco.coco_dataset.MixinCocoAddRemove.add_category() *kwcoco.coco_dataset.MixinCocoAddRemove.ensure_category()*

add_annotations(*anns*)

Faster less-safe multi-item alternative to add_annotation.

We assume the annotations are well formatted in kwcoco compliant dictionaries, including the "id" field. No validation checks are made when calling this function.

Parameters
anns (*List[Dict]*) – list of annotation dictionaries

SeeAlso:
add_annotation() *add_annotations()*

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> anns = [self.anns[aid] for aid in [2, 3, 5, 7]]
>>> self.remove_annotations(anns)
>>> assert self.n_annots == 7 and self._check_integrity()
>>> self.add_annotations(anns)
>>> assert self.n_annots == 11 and self._check_integrity()
```

add_images(*imgs*)

Faster less-safe multi-item alternative

We assume the images are well formatted in kwcoco compliant dictionaries, including the "id" field. No validation checks are made when calling this function.

Note: THIS FUNCTION WAS DESIGNED FOR SPEED, AS SUCH IT DOES NOT CHECK IF THE IMAGE-IDs or FILE_NAMES ARE DUPLICATED AND WILL BLINDLY ADD DATA EVEN IF IT IS BAD. THE SINGLE IMAGE VERSION IS SLOWER BUT SAFER.

Parameters

imgs (*List[Dict]*) – list of image dictionaries

SeeAlso:

kwcoco.coco_dataset.MixinCocoAddRemove.add_image() *kwcoco.coco_dataset.MixinCocoAddRemove.add_images()* *kwcoco.coco_dataset.MixinCocoAddRemove.ensure_image()*

Example

```
>>> import kwcoco
>>> imgs = kwcoco.CocoDataset.demo().dataset['images']
>>> self = kwcoco.CocoDataset()
>>> self.add_images(imgs)
>>> assert self.n_images == 3 and self._check_integrity()
```

clear_images()

Removes all images and annotations (but not categories)

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> self.clear_images()
>>> print(ub.urepr(self.basic_stats(), nobr=1, nl=0, si=1))
n_anns: 0, n_imgs: 0, n_videos: 0, n_cats: 8, n_tracks: 0
```

clear_annotations()

Removes all annotations and tracks (but not images and categories)

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> self.clear_annotations()
>>> print(ub.urepr(self.basic_stats(), nobr=1, nl=0, si=1))
n_anns: 0, n_imgs: 3, n_videos: 0, n_cats: 8, n_tracks: 0
```

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_integrity()
```

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*) – if True, we perform checks to remove duplicates and non-existing identifiers. Defaults to True.

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_integrity()
```

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*) – if True, keeps annotations, but removes category labels. Defaults to False.
- **safe** (*bool*) – if True, we perform checks to remove duplicates and non-existing identifiers. Defaults to True.

Returns
num_removed: information on the number of items removed

Return type
Dict

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> cat_identifiers = [self.cats[1], 'rocket', 3]
>>> self.remove_categories(cat_identifiers)
>>> assert len(self.dataset['categories']) == 5
>>> self._check_integrity()
```

remove_tracks(*track_identifiers*, *keep_annots=False*, *verbose=0*, *safe=True*)

Remove tracks and all annotations in those tracks.

Parameters

- **track_identifiers** (*List*) – list of track dicts, names, or ids
- **keep_annots** (*bool*) – if True, keeps annotations, but removes tracks labels. Defaults to False.
- **safe** (*bool*) – if True, we perform checks to remove duplicates and non-existing identifiers. Defaults to True.

Returns
num_removed: information on the number of items removed

Return type
Dict

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo('vidshapes1')
>>> for ann in self.dataset['annotations']:
...     ann.pop('segmentation')
...     ann.pop('keypoints')
>>> print('self.dataset = {}'.format(ub.urepr(self.dataset, nl=2)))
>>> track_identifiers = [2]
>>> assert len(self.dataset['tracks']) == 2
>>> self.remove_tracks(track_identifiers)
>>> print('self.dataset = {}'.format(ub.urepr(self.dataset, nl=2)))
>>> assert len(self.dataset['tracks']) == 1
>>> self._check_integrity()
```

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*) – if True, we perform checks to remove duplicates and non-existing identifiers.

Returns
num_removed: information on the number of items removed

Return type
Dict

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> assert len(self.dataset['images']) == 3
>>> gids_or_imgs = [self.imgs[2], 'astro.png']
>>> self.remove_images(gids_or_imgs)  # xdoc: +IGNORE_WANT
{'annotations': 11, 'images': 2}
>>> assert len(self.dataset['images']) == 1
>>> self._check_integrity()
>>> gids_or_imgs = [3]
>>> self.remove_images(gids_or_imgs)
>>> assert len(self.dataset['images']) == 0
>>> self._check_integrity()
```

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*) – if True, we perform checks to remove duplicates and non-existing identifiers.

Returns
num_removed: information on the number of items removed

Return type
Dict

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo('vidshapes8')
>>> assert len(self.dataset['videos']) == 8
>>> vidids_or_videos = [self.dataset['videos'][0]['id']]
>>> self.remove_videos(vidids_or_videos)  # xdoc: +IGNORE_WANT
{'annotations': 4, 'images': 2, 'videos': 1}
>>> assert len(self.dataset['videos']) == 7
>>> self._check_integrity()
```

remove_annotation_keypoints(*kp_identifiers*)

Removes all keypoints with a particular category

Parameters
kp_identifiers (*List*) – list of keypoint category dicts, names, or ids

Returns
num_removed: information on the number of items removed

Return type
Dict

remove_keypoint_categories(*kp_identifiers*)

Removes all keypoints of a particular category as well as all annotation keypoints with those ids.

Parameters
kp_identifiers (*List*) – list of keypoint category dicts, names, or ids

Returns
num_removed: information on the number of items removed

Return type
Dict

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo('shapes', rng=0)
>>> kp_identifiers = ['left_eye', 'mid_tip']
>>> remove_info = self.remove_keypoint_categories(kp_identifiers)
>>> print('remove_info = {!r}'.format(remove_info))
>>> # FIXME: for whatever reason demodata generation is not determenistic when␣
↪seeded
>>> # assert remove_info == {'keypoint_categories': 2, 'annotation_keypoints': 16,
↪'reflection_ids': 1}
>>> assert self._resolve_to_kpcat('right_eye')['reflection_id'] is None
```

set_annotation_category(*aid_or_ann*, *cid_or_cat*)

Sets the category of a single annotation

Parameters

- **aid_or_ann** (*dict* | *int*) – annotation dict or id
- **cid_or_cat** (*dict* | *int*) – category dict or id

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> old_freq = self.category_annotation_frequency()
>>> aid_or_ann = aid = 2
>>> cid_or_cat = new_cid = self.ensure_category('kitten')
>>> self.set_annotation_category(aid, new_cid)
>>> new_freq = self.category_annotation_frequency()
>>> print('new_freq = {}'.format(ub.urepr(new_freq, nl=1)))
>>> print('old_freq = {}'.format(ub.urepr(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
- **tracks** (*Dict[int, dict]*) – mapping between track ids and the track 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.
- **name_to_track** (*Dict[str, dict]*) – mapping between a track name and the track dictionary.
- **file_name_to_img** (*Dict[str, dict]*) – mapping between a image file_name and the image dictionary.

_set

alias of `set`

_images_set_sorted_by_frame_index(*gids=None*)

Helper for ensuring that vidid_to_gids returns image ids ordered by frame index.

_set_sorted_by_frame_index(*gids=None*)

Helper for ensuring that vidid_to_gids returns image ids ordered by frame index.

_annots_set_sorted_by_frame_index(*aids=None*)

Helper for ensuring that vidid_to_gids returns image ids ordered by frame index.

property cid_to_gids

Example:

```
>>> import kwcoco
>>> self = dset = kwcoco.CocoDataset()
>>> self.index.cid_to_gids
```

_add_video(*vidid*, *video*)

_add_image(*gid*, *img*)

Example

```
>>> # Test adding image to video that doesnt exist
>>> import kwcoco
>>> self = dset = kwcoco.CocoDataset()
>>> dset.add_image(file_name='frame1', video_id=1, frame_index=0)
>>> dset.add_image(file_name='frame2', video_id=1, frame_index=0)
>>> dset._check_integrity()
>>> print('dset.index.vidid_to_gids = {!r}'.format(dset.index.vidid_to_gids))
>>> assert len(dset.index.vidid_to_gids) == 1
>>> dset.add_video(name='foo-vid', id=1)
>>> assert len(dset.index.vidid_to_gids) == 1
>>> dset._check_integrity()
```

_add_images(*imgs*)

See ../dev/bench/bench_add_image_check.py

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.

_add_annotation(*aid*, *gid*, *cid*, *tid*, *ann*)

_add_annotations(*anns*)

_add_category(*cid*, *name*, *cat*)

_add_track(*trackid*, *name*, *track*)

_remove_all_annotations()

_remove_all_images()

_remove_annotations(*remove_aids*, *verbose=0*)

_remove_tracks(*remove_trackids*, *verbose=0*)

_remove_categories(*remove_cids*, *verbose=0*)

_remove_images(*remove_gids*, *verbose=0*)

_remove_videos(*remove_vidids*, *verbose=0*)

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 tid - Track 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*, *MixinCocoConstructors*, *MixinCocoExtras*, *MixinCocoHashing*, *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 | None*) – 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
```

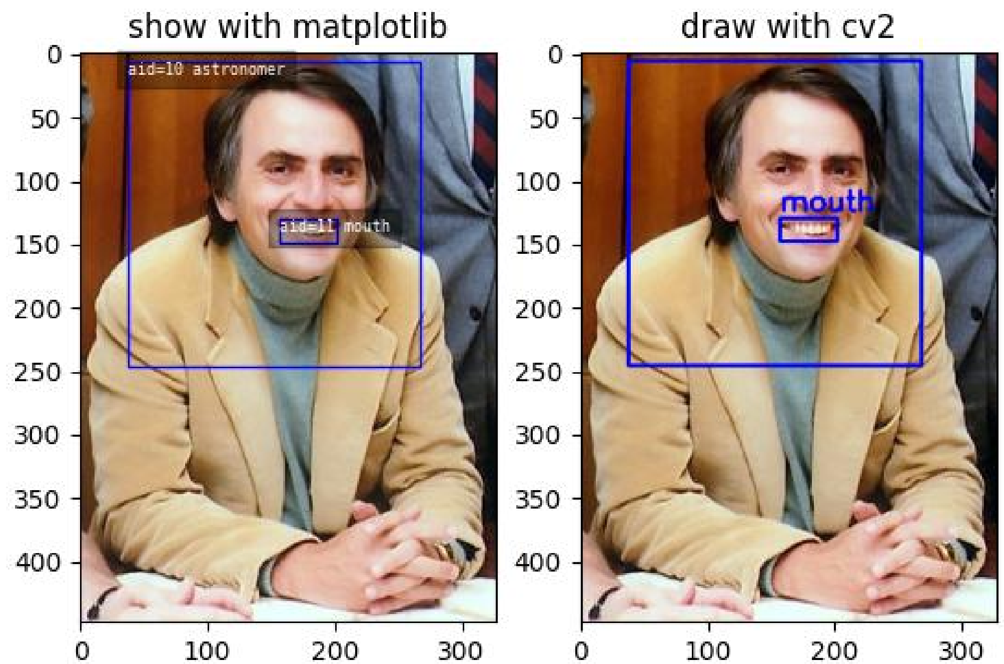
Example

```
>>> 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.urepr(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.urepr((ub.udict(ann) - {'segmentation'}).sorted_keys(), 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.urepr(cat, nl=1, sort=1)))
cat = {
    'id': 1,
    'name': 'astronaut',
    'supercategory': 'human',
}
>>> #
```

(continues on next page)

(continued from previous page)

```
>>> # Now play with some helper functions, like extended statistics
>>> extended_stats = self.extended_stats()
>>> # xdoctest: +IGNORE_WANT
>>> print('extended_stats = {}'.format(ub.urepr(extended_stats, nl=1, precision=2,␣
↪sort=1)))
extended_stats = {
    'annots_per_img': {'mean': 3.67, 'std': 3.86, 'min': 0.00, 'max': 9.00, 'nMin':␣
↪1, 'nMax': 1, 'shape': (3,)},
    'imgs_per_cat': {'mean': 0.88, 'std': 0.60, 'min': 0.00, 'max': 2.00, 'nMin': 2,
↪ 'nMax': 1, 'shape': (8,)},
    'cats_per_img': {'mean': 2.33, 'std': 2.05, 'min': 0.00, 'max': 5.00, 'nMin': 1,
↪ 'nMax': 1, 'shape': (3,)},
    'annots_per_cat': {'mean': 1.38, 'std': 1.49, 'min': 0.00, 'max': 5.00, 'nMin':␣
↪2, 'nMax': 1, 'shape': (8,)},
    'imgs_per_video': {'empty_list': True},
}
>>> # You can "draw" a raster of the annotated image with cv2
>>> canvas = self.draw_image(2)
>>> # Or if you have matplotlib you can "show" the image with mpl objects
>>> # xdoctest: +REQUIRES(--show)
>>> from matplotlib import pyplot as plt
>>> fig = plt.figure()
>>> ax1 = fig.add_subplot(1, 2, 1)
>>> self.show_image(gid=2)
>>> ax2 = fig.add_subplot(1, 2, 2)
>>> ax2.imshow(canvas)
>>> ax1.set_title('show with matplotlib')
>>> ax2.set_title('draw with cv2')
>>> plt.show()
```



Parameters

- **data** (*str* | *PathLike* | *dict* | *None*) – Either a filepath to a coco json file, or a dictionary containing the actual coco json structure. For a more generally coercable constructor see func:*CocoDataset.coerce*.
- **tag** (*str* | *None*) – Name of the dataset for display purposes, and does not influence behavior of the underlying data structure, although it may be used via convinience methods. We attempt to autopopulate this via information in `data` if available. If unspecfied and `data` is a filepath this becomes the basename.
- **bundle_dpath** (*str* | *None*) – the root of the dataset that images / external data will be assumed to be relative to. If unspecfied, we attempt to determine it using information in `data`. If `data` is a filepath, we use the dirname of that path. If `data` is a dictionary, we look for the "img_root" key. If unspecfied and we fail to introspect then, we fallback to the current working directory.
- **img_root** (*str* | *None*) – deprecated alias for bundle_dpath

property fpath

In the future we will deprecate img_root for bundle_dpath

_update_fpath(*new_fpath*)

_infer_dirs()

classmethod from_data(*data*, *bundle_dpath=None*, *img_root=None*)

Constructor from a json dictionary

Return type
CocoDataset

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

Return type
CocoDataset

Example

```
>>> import kwcoco
>>> coco_dset = kwcoco.CocoDataset.from_image_paths(['a.png', 'b.png'])
>>> assert coco_dset.n_images == 2
```

classmethod from_class_image_paths(*root*)

Ingest classification data in the common format where images of different categories are stored in folders with the category label.

Parameters
root (*str | PathLike*) – the path to a directory containing class-subdirectories

Return type
CocoDataset

classmethod coerce_multiple(*datas*, *workers=0*, *mode='process'*, *verbose=1*, *postprocess=None*, *ordered=True*, ***kwargs*)

Coerce multiple CocoDataset objects in parallel.

Parameters

- **datas** (*List*) – list of kwcoco coercables to load
- **workers** (*int | str*) – number of worker threads / processes. Can also accept coerceable workers.
- **mode** (*str*) – thread, process, or serial. Defaults to process.
- **verbose** (*int*) – verbosity level
- **postprocess** (*Callable | None*) – A function taking one arg (the loaded dataset) to run on the loaded kwcoco dataset in background workers. This can be more efficient when postprocessing is independent per kwcoco file.
- **ordered** (*bool*) – if True yields datasets in the same order as given. Otherwise results are yielded as they become available. Defaults to True.
- ****kwargs** – arguments passed to the constructor

Yields
CocoDataset

SeeAlso:

- load_multiple - like this function but is a strict file-path-only loader

CommandLine

```
xdoctest -m kwcoco.coco_dataset CocoDataset.coerce_multiple
```

Example

```
>>> import kwcoco
>>> dset1 = kwcoco.CocoDataset.demo('shapes1')
>>> dset2 = kwcoco.CocoDataset.demo('shapes2')
>>> dset3 = kwcoco.CocoDataset.demo('vidshapes8')
>>> dsets = [dset1, dset2, dset3]
>>> input_fpaths = [d.fpath for d in dsets]
>>> results = list(kwcoco.CocoDataset.coerce_multiple(input_fpaths,␣
↪ordered=True))
>>> result_fpaths = [r.fpath for r in results]
>>> assert result_fpaths == input_fpaths
>>> # Test unordered
>>> results1 = list(kwcoco.CocoDataset.coerce_multiple(input_fpaths,␣
↪ordered=False))
>>> result_fpaths = [r.fpath for r in results]
>>> assert set(result_fpaths) == set(input_fpaths)
>>> #
>>> # Coerce from existing datasets
>>> results2 = list(kwcoco.CocoDataset.coerce_multiple(dsets, ordered=True,␣
↪workers=0))
>>> assert results2[0] is dsets[0]
```

classmethod load_multiple(*fpaths*, *workers=0*, *mode='process'*, *verbose=1*, *postprocess=None*, *ordered=True*, ***kwargs*)

Load multiple CocoDataset objects in parallel.

Parameters

- **fpaths** (*List[str | PathLike]*) – list of paths to multiple coco files to be loaded
- **workers** (*int*) – number of worker threads / processes
- **mode** (*str*) – thread, process, or serial. Defaults to process.
- **verbose** (*int*) – verbosity level
- **postprocess** (*Callable | None*) – A function taking one arg (the loaded dataset) to run on the loaded kwcoco dataset in background workers and returns the modified dataset. This can be more efficient when postprocessing is independent per kwcoco file.
- **ordered** (*bool*) – if True yields datasets in the same order as given. Otherwise results are yielded as they become available. Defaults to True.
- ****kwargs** – arguments passed to the constructor

Yields

CocoDataset

SeeAlso:

- **coerce_multiple - like this function but accepts general** coercable inputs.

classmethod _load_multiple(*_loader, inputs, workers=0, mode='process', verbose=1, postprocess=None, ordered=True, **kwargs*)

Shared logic for multiprocessing loaders.

SeeAlso:

- coerce_multiple
- load_multiple

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*) – number of worker threads / processes
- **verbose** (*int*) – verbosity level
- **mode** (*str*) – thread, process, or serial
- **union** (*str | bool*) – 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. Default='try'

Note: This may be deprecated. Use load_multiple or coerce_multiple and then manually perform the union.

copy()

Deep copies this object

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> new = self.copy()
>>> assert new.imgs[1] is new.dataset['images'][0]
>>> assert new.imgs[1] == self.dataset['images'][0]
>>> assert new.imgs[1] is not self.dataset['images'][0]
```

dumps(*indent=None, newlines=False*)

Writes the dataset out to the json format

Parameters

- **newlines** (*bool*) – if True, each annotation, image, category gets its own line
- **indent** (*int | str | None*) – indentation for the json file. See `json.dump()` for details.
- **newlines** (*bool*) – if True, each annotation, image, category gets its own line.

Note:

Using newlines=True is similar to:
print(ub.urepr(dset.dataset, nl=2, trailsep=False)) However, the above may not output valid json if it contains ndarrays.

Example

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

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

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.coerce('vidshapes1-msi-multisensor', verbose=3)
>>> self.remove_annotations(self.annots())
>>> text = self.dumps(newlines=0, indent='  ')
>>> print(text)
>>> text = self.dumps(newlines=True, indent='  ')
>>> print(text)
```

_compress_dump_to_fileptr(*file*, *arcname=None*, *indent=None*, *newlines=False*)

Experimental method to save compressed kwcoco files, may be folded into dump in the future.

_dump(*file*, *indent*, *newlines*, *compress*)

Case where we are dumping to an open file pointer. We assume this means the dataset has been written to disk.

dump(*file=None*, *indent=None*, *newlines=False*, *temp_file='auto'*, *compress='auto'*)

Writes the dataset out to the json format

Parameters

- **file** (*PathLike | IO | None*) – Where to write the data. Can either be a path to a file or an open file pointer / stream. If unspecified, it will be written to the current `fpath` property.
- **indent** (*int | str | None*) – indentation for the json file. See `json.dump()` for details.
- **newlines** (*bool*) – if True, each annotation, image, category gets its own line.

- **temp_file** (*bool* | *str*) – Argument to `safer.open()`. Ignored if `file` is not a PathLike object. Defaults to 'auto', which is False on Windows and True everywhere else.
- **compress** (*bool* | *str*) – if True, dumps the kwcoco file as a compressed zipfile. In this case a literal IO file object must be opened in binary write mode. If auto, then it will default to False unless it can introspect the file name and the name ends with .zip

Example

```
>>> import kwcoco
>>> import ubelt as ub
>>> dpath = ub.Path.appdir('kwcoco/demo/dump').ensuredir()
>>> dset = kwcoco.CocoDataset.demo()
>>> dset.fpath = dpath / 'my_coco_file.json'
>>> # Calling dump writes to the current fpath attribute.
>>> dset.dump()
>>> assert dset.dataset == kwcoco.CocoDataset(dset.fpath).dataset
>>> assert dset.dumps() == dset.fpath.read_text()
>>> #
>>> # Using compress=True can save a lot of space and it
>>> # is transparent when reading files via CocoDataset
>>> dset.dump(compress=True)
>>> assert dset.dataset == kwcoco.CocoDataset(dset.fpath).dataset
>>> assert dset.dumps() != dset.fpath.read_text(errors='replace')
```

Example

```
>>> import kwcoco
>>> import ubelt as ub
>>> # Compression auto-defaults based on the file name.
>>> dpath = ub.Path.appdir('kwcoco/demo/dump').ensuredir()
>>> dset = kwcoco.CocoDataset.demo()
>>> fpath1 = dset.fpath = dpath / 'my_coco_file.zip'
>>> dset.dump()
>>> fpath2 = dset.fpath = dpath / 'my_coco_file.json'
>>> dset.dump()
>>> assert fpath1.read_bytes()[0:8] != fpath2.read_bytes()[0:8]
```

_check_json_serializable(*verbose=1*)

Debug which part of a coco dataset might not be json serializable

_check_integrity()

perform most checks

_check_index()

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> self._check_index()
>>> # Force a failure
>>> self.index.anns.pop(1)
>>> self.index.anns.pop(2)
>>> import pytest
>>> with pytest.raises(AssertionError):
>>>     self._check_index()
```

_abc_impl = <_abc._abc_data object>

_check_pointers(*verbose=1*)

Check that all category and image ids referenced by annotations exist

_build_index()

union(*, *disjoint_tracks=True*, *remember_parent=False*, ***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*) – if True, we will assume track-names are disjoint and if two datasets share the same track-name, we will disambiguate them. Otherwise they will be copied over as-is. Defaults to True. In most cases you do not want to set this to False.
- **remember_parent** (*bool*) – if True, videos and images will save information about their parent in the "union_parent" field.
- ****kwargs** – constructor options for the new merged CocoDataset

Returns

a new merged coco dataset

Return type

kwcoco.CocoDataset

CommandLine

```
xdoctest -m kwcoco.coco_dataset CocoDataset.union
```

Example

```
>>> import kwcoco
>>> # Test union works with different keypoint categories
>>> dset1 = kwcoco.CocoDataset.demo('shapes1')
>>> dset2 = kwcoco.CocoDataset.demo('shapes2')
>>> dset1.remove_keypoint_categories(['bot_tip', 'mid_tip', 'right_eye'])
>>> dset2.remove_keypoint_categories(['top_tip', 'left_eye'])
>>> dset_12a = kwcoco.CocoDataset.union(dset1, dset2)
>>> dset_12b = dset1.union(dset2)
>>> dset_21 = dset2.union(dset1)
>>> def add_hist(h1, h2):
>>>     return {k: h1.get(k, 0) + h2.get(k, 0) for k in set(h1) | set(h2)}
>>> kpfreq1 = dset1.keypoint_annotation_frequency()
>>> kpfreq2 = dset2.keypoint_annotation_frequency()
>>> kpfreq_want = add_hist(kpfreq1, kpfreq2)
>>> kpfreq_got1 = dset_12a.keypoint_annotation_frequency()
>>> kpfreq_got2 = dset_12b.keypoint_annotation_frequency()
>>> assert kpfreq_want == kpfreq_got1
>>> assert kpfreq_want == kpfreq_got2
```

```
>>> # Test disjoint gid datasets
>>> dset1 = kwcoco.CocoDataset.demo('shapes3')
>>> for new_gid, img in enumerate(dset1.dataset['images'], start=10):
>>>     for aid in dset1.gid_to_aids[img['id']]:
>>>         dset1.anns[aid]['image_id'] = new_gid
>>>     img['id'] = new_gid
>>> dset1.index.clear()
>>> dset1._build_index()
>>> # ------
>>> dset2 = kwcoco.CocoDataset.demo('shapes2')
>>> for new_gid, img in enumerate(dset2.dataset['images'], start=100):
>>>     for aid in dset2.gid_to_aids[img['id']]:
>>>         dset2.anns[aid]['image_id'] = new_gid
>>>     img['id'] = new_gid
>>> dset1.index.clear()
>>> dset2._build_index()
>>> others = [dset1, dset2]
>>> merged = kwcoco.CocoDataset.union(*others)
>>> print('merged = {!r}'.format(merged))
>>> print('merged.imgs = {}'.format(ub.urepr(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.urepr(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.urepr(dset1.anns, nl=1)))
>>> print('dset2.anns = {}'.format(ub.urepr(dset2.anns, nl=1)))
>>> print('dset3.anns = {}'.format(ub.urepr(dset3.anns, nl=1)))
>>> print('merged.anns = {}'.format(ub.urepr(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*) – if True, makes a deep copy of all nested attributes, otherwise makes a shallow copy. Defaults to True.
- **autobuild** (*bool*) – if True will automatically build the fast lookup index. Defaults to True.

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> gids = [1, 3]
>>> sub_dset = self.subset(gids)
>>> assert len(self.index.gid_to_aids) == 3
>>> assert len(sub_dset.gid_to_aids) == 2
```

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo('vidshapes2')
>>> gids = [1, 2]
>>> sub_dset = self.subset(gids, copy=True)
>>> assert len(sub_dset.index.videos) == 1
>>> assert len(self.index.videos) == 2
```

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> sub1 = self.subset([1])
>>> sub2 = self.subset([2])
>>> sub3 = self.subset([3])
>>> others = [sub1, sub2, sub3]
>>> rejoined = kwcoco.CocoDataset.union(*others)
>>> assert len(sub1.anns) == 9
>>> assert len(sub2.anns) == 2
>>> assert len(sub3.anns) == 0
>>> assert rejoined.basic_stats() == self.basic_stats()
```

`view_sql`(*force_rewrite=False*, *memory=False*, *backend='sqlite'*, *sql_db_fpath=None*)

Create a cached SQL interface to this dataset suitable for large scale multiprocessing use cases.

Parameters

- **force_rewrite** (*bool*) – if True, forces an update to any existing cache file on disk
- **memory** (*bool*) – if True, the database is constructed in memory.
- **backend** (*str*) – sqlite or postgresql
- **sql_db_fpath** (*str* | *PathLike* | *None*) – overrides the database uri

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.

CommandLine

```
KWCOCO_WITH_POSTGRESQL=1 xdoctest -m /home/joncrall/code/kwcoco/kwcoco/coco_
↪dataset.py CocoDataset.view_sql
```

Example

```
>>> # xdoctest: +REQUIRES(module:sqlalchemy)
>>> # xdoctest: +REQUIRES(env:KWCOCO_WITH_POSTGRESQL)
>>> # xdoctest: +REQUIRES(module:psycopg2)
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('vidshapes32')
>>> postgres_dset = dset.view_sql(backend='postgresql', force_rewrite=True)
>>> sqlite_dset = dset.view_sql(backend='sqlite', force_rewrite=True)
>>> list(dset.anns.keys())
>>> list(postgres_dset.anns.keys())
>>> list(sqlite_dset.anns.keys())
```

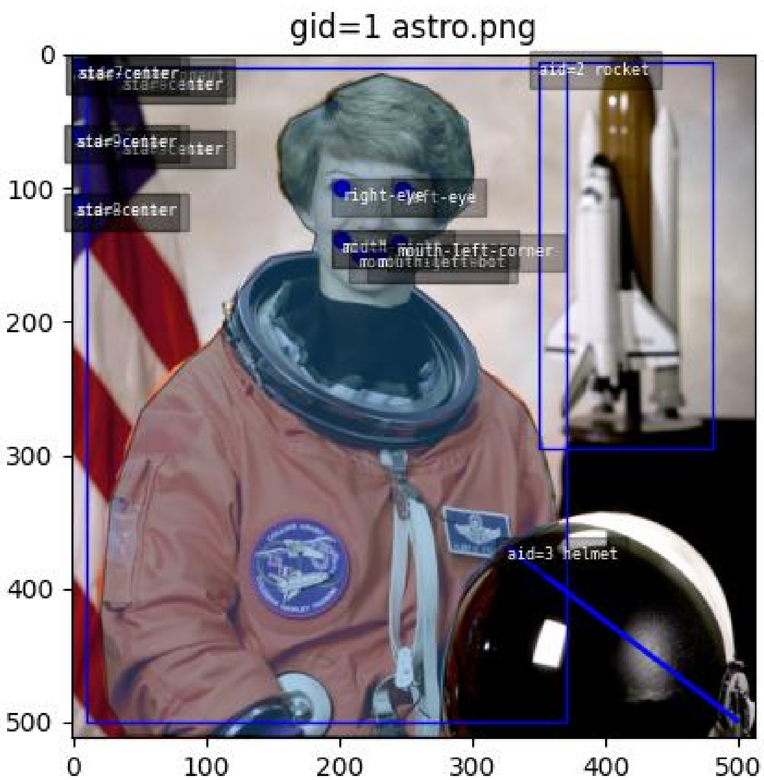
kwcoco.coco_dataset.**demo_coco_data**()

Simple data for testing.

This contains several non-standard fields, which help ensure robustness of functions tested with this data. For more compliant demodata see the `kwcoco.demodata` submodule.

Example

```
>>> # xdoctest: +REQUIRES(--show)
>>> import kwcoco
>>> from kwcoco.coco_dataset import demo_coco_data
>>> dataset = demo_coco_data()
>>> self = kwcoco.CocoDataset(dataset, tag='demo')
>>> import kwplot
>>> kwplot.autompl()
>>> self.show_image(gid=1)
>>> kwplot.show_if_requested()
```



2.1.2.7 kwcoco.coco_evaluator module

Evaluates a predicted coco dataset against a truth coco dataset.

This currently computes detection-level metrics.

The components in this module work programatically or as a command line script.

Todo:

- **[ ] does evaluate return one result or multiple results**
  based on different configurations?
- [ ] max_dets - TODO: in original pycocoutils but not here
- [ ] Flag that allows for polygon instead of bounding box overlap
- **[ ] How do we note what iou_thresh and area-range were in**
  the result plots?

CommandLine

```
xdoctest -m kwcoco.coco_evaluator __doc__:0 --vd --slow
```

Example

```
>>> from kwcoco.coco_evaluator import *  # NOQA
>>> from kwcoco.coco_evaluator import CocoEvaluator
>>> import kwcoco
>>> # note: increase the number of images for better looking metrics
>>> true_dset = kwcoco.CocoDataset.demo('shapes8')
>>> from kwcoco.demo.perterb import perterb_coco
>>> kwargs = {
>>>     'box_noise': 0.5,
>>>     'n_fp': (0, 10),
>>>     'n_fn': (0, 10),
>>>     'with_probs': True,
>>> }
>>> pred_dset = perterb_coco(true_dset, **kwargs)
>>> print('true_dset = {!r}'.format(true_dset))
>>> print('pred_dset = {!r}'.format(pred_dset))
>>> config = {
>>>     'true_dataset': true_dset,
>>>     'pred_dataset': pred_dset,
>>>     'area_range': ['all', 'small'],
>>>     'iou_thresh': [0.3, 0.95],
>>> }
>>> coco_eval = CocoEvaluator(config)
>>> results = coco_eval.evaluate()
>>> # Now we can draw / serialize the results as we please
>>> dpath = ub.Path.appdir('kwcoco/tests/test_out_dpath').ensuredir()
>>> results_fpath = dpath / 'metrics.json'
>>> print('results_fpath = {!r}'.format(results_fpath))
>>> results.dump(results_fpath, indent='    ')
>>> measures = results['area_range=all,iou_thresh=0.3'].nocls_measures
>>> import pandas as pd
>>> print(pd.DataFrame(ub.dict_isect(
>>>     measures, ['f1', 'g1', 'mcc', 'thresholds',
>>>                'ppv', 'tpr', 'tnr', 'npv', 'fpr',
>>>                'tp_count', 'fp_count',
>>>                'tn_count', 'fn_count'])).iloc[::100])
>>> # xdoctest: +REQUIRES(module:kwplot)
>>> # xdoctest: +REQUIRES(--slow)
>>> results.dump_figures(dpath)
>>> print('dpath = {!r}'.format(dpath))
>>> # xdoctest: +REQUIRES(--vd)
>>> if ub.argflag('--vd') or 1:
>>>     import xdev
>>>     xdev.view_directory(dpath)
```

class kwcoco.coco_evaluator.**CocoEvalConfig**(**args*, ***kwargs*)

Bases: DataConfig

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

Valid options: []

Parameters

- ***args** – positional arguments for this data config
- ****kwargs** – keyword arguments for this data config

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

normalize()

class kwcoco.coco_evaluator.**CocoEvaluator**(*config*)

Bases: object

Abstracts the evaluation process to execute on two coco datasets.

This can be run as a standalone script where the user specifies the paths to the true and predited dataset explicitly, or this can be used by a higher level script that produces the predictions and then sends them to this evaluator.

Example

```
>>> from kwcoco.coco_evaluator import CocoEvaluator
>>> from kwcoco.demo.perterb import perterb_coco
>>> import kwcoco
>>> true_dset = kwcoco.CocoDataset.demo('shapes8')
>>> kwargs = {
>>>     'box_noise': 0.5,
>>>     'n_fp': (0, 10),
>>>     'n_fn': (0, 10),
>>>     'with_probs': True,
>>> }
>>> pred_dset = perterb_coco(true_dset, **kwargs)
>>> config = {
>>>     'true_dataset': true_dset,
>>>     'pred_dataset': pred_dset,
>>>     'classes_of_interest': [],
>>> }
>>> coco_eval = CocoEvaluator(config)
>>> results = coco_eval.evaluate()
```

log(*msg*, *level='INFO'*)

_init()

Performs initial coercion from given inputs into dictionaries of kwimage.Detection objects and attempts to ensure comparable category and image ids.

_ensure_init()

classmethod _rectify_classes(*true_classes*, *pred_classes*)

classmethod _coerce_dets(*dataset*, *verbose=0*, *workers=0*)

Coerce the input to a mapping from image-id to kwimage.Detection

Also capture a CocoDataset if possible.

Returns
gid_to_det: mapping from gid to dets extra: any extra information we gathered via coercion

Return type
Tuple[Dict[int, Detections], Dict]

Example

```
>>> from kwcoco.coco_evaluator import *  # NOQA
>>> import kwcoco
>>> coco_dset = kwcoco.CocoDataset.demo('shapes8')
>>> gid_to_det, extras = CocoEvaluator._coerce_dets(coco_dset)
```

Example

```
>>> # xdoctest: +REQUIRES(module:sqlalchemy)
>>> from kwcoco.coco_evaluator import *  # NOQA
>>> import kwcoco
>>> coco_dset = kwcoco.CocoDataset.demo('shapes8').view_sql()
>>> gid_to_det, extras = CocoEvaluator._coerce_dets(coco_dset)
```

_build_dmet()

Builds the detection metrics object

Returns

DetectionMetrics - object that can perform assignment and
build confusion vectors.

evaluate()

Executes the main evaluation logic. Performs assignments between detections to make DetectionMetrics object, then creates per-item and ovr confusion vectors, and performs various threshold-vs-confusion analyses.

Returns

container storing (and capable of drawing /
serializing) results

Return type
CocoResults

kwcoco.coco_evaluator.**dmet_area_weights**(*dmet*, *orig_weights*, *cfsn_vecs*, *area_ranges*, *coco_eval*, *use_area_attr=False*)

Hacky function to compute confusion vector ignore weights for different area thresholds. Needs to be slightly refactored.

class kwcoco.coco_evaluator.**CocoResults**(*resdata=None*)

Bases: NiceRepr, *DictProxy*

CommandLine

```
xdoctest -m /home/joncrall/code/kwcoco/kwcoco/coco_evaluator.py CocoResults --
↪profile
```

Example

```
>>> from kwcoco.coco_evaluator import *  # NOQA
>>> from kwcoco.coco_evaluator import CocoEvaluator
>>> import kwcoco
>>> true_dset = kwcoco.CocoDataset.demo('shapes2')
>>> from kwcoco.demo.perterb import perterb_coco
>>> kwargs = {
>>>     'box_noise': 0.5,
>>>     'n_fp': (0, 10),
>>>     'n_fn': (0, 10),
>>> }
>>> pred_dset = perterb_coco(true_dset, **kwargs)
>>> print('true_dset = {!r}'.format(true_dset))
>>> print('pred_dset = {!r}'.format(pred_dset))
>>> config = {
>>>     'true_dataset': true_dset,
>>>     'pred_dataset': pred_dset,
>>>     'area_range': ['small'],
>>>     'iou_thresh': [0.3],
>>> }
>>> coco_eval = CocoEvaluator(config)
>>> results = coco_eval.evaluate()
>>> # Now we can draw / serialize the results as we please
>>> dpath = ub.Path.appdir('kwcoco/tests/test_out_dpath').ensuredir()
>>> #
>>> # test deserialization works
>>> state = results.__json__()
>>> self2 = CocoResults.from_json(state)
>>> #
>>> # xdoctest: +REQUIRES(module:kwplot)
>>> results.dump_figures(dpath, figsize=(3, 2), tight=False)  # make this go faster
>>> results.dump(dpath / 'metrics.json', indent='    ')
```

dump_figures(*out_dpath*, *expt_title=None*, *figsize='auto'*, *tight=True*)

classmethod from_json(*state*)

dump(*file*, *indent=' '*)

Serialize to json file

class kwcoco.coco_evaluator.**CocoSingleResult**(*nocls_measures*, *ovr_measures*, *cfsn_vecs*, *meta=None*)

Bases: NiceRepr

Container class to store, draw, summarize, and serialize results from CocoEvaluator.

Example

```
>>> # xdoctest: +REQUIRES(--slow)
>>> from kwcoco.coco_evaluator import *  # NOQA
>>> from kwcoco.coco_evaluator import CocoEvaluator
>>> import kwcoco
>>> true_dset = kwcoco.CocoDataset.demo('shapes8')
>>> from kwcoco.demo.perterb import perterb_coco
>>> kwargs = {
>>>     'box_noise': 0.2,
>>>     'n_fp': (0, 3),
>>>     'n_fn': (0, 3),
>>>     'with_probs': False,
>>> }
>>> pred_dset = perterb_coco(true_dset, **kwargs)
>>> print('true_dset = {!r}'.format(true_dset))
>>> print('pred_dset = {!r}'.format(pred_dset))
>>> config = {
>>>     'true_dataset': true_dset,
>>>     'pred_dataset': pred_dset,
>>>     'area_range': [(0, 32 ** 2), (32 ** 2, 96 ** 2)],
>>>     'iou_thresh': [0.3, 0.5, 0.95],
>>> }
>>> coco_eval = CocoEvaluator(config)
>>> results = coco_eval.evaluate()
>>> result = ub.peek(results.values())
>>> state = result.__json__()
>>> print('state = {}'.format(ub.urepr(state, nl=-1)))
>>> recon = CocoSingleResult.from_json(state)
>>> state = recon.__json__()
>>> print('state = {}'.format(ub.urepr(state, nl=-1)))
```

classmethod from_json(*state*)

dump(*file*, *indent=' '*)

Serialize to json file

dump_figures(*out_dpath*, *expt_title=None*, *figsize='auto'*, *tight=True*, *verbose=1*)

kwcoco.coco_evaluator.**_writefig**(*fig*, *metrics_dpath*, *fname*, *figsize*, *verbose*, *tight*)

kwcoco.coco_evaluator.**_load_dets**(*pred_fpaths*, *workers=0*)

Example

```
>>> from kwcoco.coco_evaluator import _load_dets, _load_dets_worker
>>> import ubelt as ub
>>> import kwcoco
>>> dpath = ub.Path.appdir('kwcoco/tests/load_dets').ensuredir()
>>> N = 4
>>> pred_fpaths = []
>>> for i in range(1, N + 1):
>>>     dset = kwcoco.CocoDataset.demo('shapes{}'.format(i))
```

(continues on next page)

(continued from previous page)

```
>>>     dset.fpath = dpath / 'shapes_{}.mscoco.json'.format(i)
>>>     dset.dump(dset.fpath)
>>>     pred_fpaths.append(dset.fpath)
>>> dets, coco_dset = _load_dets(pred_fpaths)
>>> print('dets = {!r}'.format(dets))
>>> print('coco_dset = {!r}'.format(coco_dset))
```

kwcoco.coco_evaluator.**_load_dets_worker**(*single_pred_fpath*, *with_coco=True*)

2.1.2.8 kwcoco.coco_image module

Defines the CocoImage class which is an object oriented way of manipulating data pointed to by a COCO image dictionary.

Notably this provides the `.imdelay` method for delayed image loading (which enables things like fast loading of subimage-regions / coarser scales in images that contain tiles / overviews - e.g. Cloud Optimized Geotiffs or COGs (Medical image formats may be supported in the future).

Todo: This file no longer is only images, it has logic for generic single-class objects. It should be refactored into coco_objects0d.py or something.

---

class kwcoco.coco_image.**_CocoObject**(*obj*, *dset=None*, *bundle_dpath=None*)

Bases: *AliasedDictProxy*, NiceRepr

General coco scalar object

property bundle_dpath

detach()

Removes references to the underlying coco dataset, but keeps special information such that it wont be needed.

class kwcoco.coco_image.**CocoImage**(*img*, *dset=None*)

Bases: *_CocoObject*

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 = kwcoco.CocoImage(dset1.imgs[1], dset1)
>>> print('self = {!r}'.format(self))
>>> print('self.channels = {}'.format(ub.urepr(self.channels, nl=1)))
```

```
>>> self = kwcoco.CocoImage(dset2.imgs[1], dset2)
>>> print('self.channels = {}'.format(ub.urepr(self.channels, nl=1)))
>>> self.primary_asset()
>>> assert 'auxiliary' in self
```

classmethod from_gid(*dset*, *gid*)

property video

Helper to grab the video for this image if it exists

property name

detach()

Removes references to the underlying coco dataset, but keeps special information such that it wont be needed.

property assets

CocoImage.iter_assets.

Type
Convinience wrapper around

Type
func

property datetime

Try to get datetime information for this image. Not always possible.

Returns
datetime.datetime | None

annots()

Returns
a 1d annotations object referencing annotations in this image

Return type
Annots

stats()

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

keys()

Proxy getter attribute for underlying *self.img* dictionary

property channels

property n_assets

The number of on-disk files associated with this coco image

property num_channels

property dsize

primary_image_filepath(*requires=None*)

primary_asset(*requires=None*, *as_dict=True*)

Compute a "main" image asset.

Note: 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] | None*) – list of attribute that must be non-None to consider an object as the primary one.
- **as_dict** (*bool*) – if True the return type is a raw dictionary. Otherwise use a newer object-oriented wrapper that should be duck-type swappable. In the future this default will change to False.

Returns

the asset dict or None if it is not found

Return type

None | dict

Todo:

- [ ] Add in primary heuristics

Example

```
>>> import kwarray
>>> from kwcoco.coco_image import *  # NOQA
>>> rng = kwarray.ensure_rng(0)
>>> def random_asset(name, w=None, h=None):
>>>     return {'file_name': name, 'width': w, 'height': h}
>>> self = CocoImage({
>>>     'auxiliary': [
>>>         random_asset('1'),
>>>         random_asset('2'),
>>>         random_asset('3'),
>>>     ]
>>> })
>>> assert self.primary_asset()['file_name'] == '1'
>>> self = CocoImage({
>>>     'auxiliary': [
>>>         random_asset('1'),
>>>         random_asset('2', 3, 3),
>>>         random_asset('3'),
>>>     ]
```

(continues on next page)

(continued from previous page)

```
>>> })
>>> assert self.primary_asset()['file_name'] == '2'
>>> #
>>> # Test new objct oriented output
>>> self = CocoImage({
>>>     'file_name': 'foo',
>>>     'assets': [
>>>         random_asset('1'),
>>>         random_asset('2'),
>>>         random_asset('3'),
>>>     ],
>>> })
>>> assert self.primary_asset(as_dict=False) is self
>>> self = CocoImage({
>>>     'assets': [
>>>         random_asset('1'),
>>>         random_asset('3'),
>>>     ],
>>>     'auxiliary': [
>>>         random_asset('1'),
>>>         random_asset('2', 3, 3),
>>>         random_asset('3'),
>>>     ]
>>> })
>>> assert self.primary_asset(as_dict=False)['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.

Yields

ub.Path

iter_assets()

Iterate through assets (which could include the image itself it points to a file path).

Object-oriented alternative to *CocoImage.iter_asset_objs()*

Yields

CocoImage | *CocoAsset* – an asset object (or image object if it points to a file)

Example

```
>>> import kwcoco
>>> coco_img = kwcoco.CocoImage({'width': 128, 'height': 128})
>>> assert len(list(coco_img.iter_assets())) == 0
>>> dset = kwcoco.CocoDataset.demo('vidshapes8-multispectral')
>>> self = dset.coco_image(1)
>>> assert len(list(self.iter_assets())) > 1
>>> dset = kwcoco.CocoDataset.demo('vidshapes8')
```

(continues on next page)

(continued from previous page)

```
>>> self = dset.coco_image(1)
>>> assert list(self.iter_assets()) == [self]
```

iter_asset_objs()

Iterate through base + auxiliary dicts that have file paths

Note: In most cases prefer *iter_assets()* instead.

Yields
dict – an image or auxiliary dictionary

find_asset(*channels*)

Find the asset dictionary with the specified channels

Parameters
channels (*str* | *FusedChannelSpec*) – channel names the asset must have.

Returns
CocoImage | CocoAsset

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoImage({
>>>     'file_name': 'raw',
>>>     'channels': 'red|green|blue',
>>>     'assets': [
>>>         {'file_name': '1', 'channels': 'spam'},
>>>         {'file_name': '2', 'channels': 'eggs|jam'},
>>>     ],
>>>     'auxiliary': [
>>>         {'file_name': '3', 'channels': 'foo'},
>>>         {'file_name': '4', 'channels': 'bar|baz'},
>>>     ]
>>> })
>>> assert self.find_asset('blah') is None
>>> assert self.find_asset('red|green|blue') is self
>>> self.find_asset('foo')['file_name'] == '3'
>>> self.find_asset('baz')['file_name'] == '4'
```

find_asset_obj(*channels*)

Find the asset dictionary with the specified channels

In most cases use `CocoImge.find_asset()` instead.

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
```

_assets_key()

Internal helper for transition from auxiliary -> assets in the image spec

add_annotation(***ann*)

Adds an annotation to this image.

This is a convinience method, and requires that this CocoImage is still connected to a parent dataset.

Parameters
****ann** – annotation attributes (e.g. bbox, category_id)

Returns
the new annotation id

Return type
int

SeeAlso:
`kwcoco.CocoDataset.add_annotation()`

add_asset(*file_name=None*, *channels=None*, *imdata=None*, *warp_aux_to_img=None*, *width=None*, *height=None*, *imwrite=False*, *image_id=None*, ***kw*)

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 | PathLike | None*) – The name of the file relative to the bundle directory. If unspecified, imdata must be given.
- **channels** (*str | kwcoco.FusedChannelSpec | None*) – 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 | None*) – 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 | None*) – Width of the data in auxiliary space (inferred if unspecified)
- **height** (*int | None*) – 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.
- **image_id** (*int | None*) – An asset dictionary contains an image-id, but it should *not* be specified here. If it is, then it *must* agree with this image's id.
- ****kw** – stores arbitrary key/value pairs in this new asset.

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_asset(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_asset('path/img1_B0.tif', channels='B0', width=200, height=200)
>>> coco_img.add_asset('path/img1_B1.tif', channels='B1', width=200, height=200)
>>> coco_img.add_asset('path/img1_B2.tif', channels='B2', width=200, height=200)
```

(continues on next page)

(continued from previous page)

```
>>> coco_img.add_asset('path/img1_TCI.tif', channels='r|g|b', width=200,␣
↪height=200)
```

imdelay(*channels=None*, *space='image'*, *resolution=None*, *bundle_dpath=None*, *interpolation='linear'*, *antialias=True*, *nodata_method=None*, *RESOLUTION_KEY=None*)

Perform a delayed load on the data in this image.

The delayed load can load a subset of channels, and perform lazy warping operations. If the underlying data is in a tiled format this can reduce the amount of disk IO needed to read the data if only a small crop or lower resolution view of the data is needed.

Note: This method is experimental and relies on the delayed load proof-of-concept.

Parameters

- **gid** (*int*) – image id to load
- **channels** (*kwcoco.FusedChannelSpec*) – specific channels to load. if unspecified, all channels are loaded.
- **space** (*str*) – can either be "image" for loading in image space, or "video" for loading in video space.
- **resolution** (*None | str | float*) – If specified, applies an additional scale factor to the result such that the data is loaded at this specified resolution. This requires that the image / video has a registered resolution attribute and that its units agree with this request.

Todo:

- **[ ] 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.imdelay()
>>> 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).imdelay()
>>> 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).imdelay(channels='r')
```

```
>>> import kwcoco
>>> gid = 1
>>> #
>>> dset = kwcoco.CocoDataset.demo('vidshapes8-multispectral')
>>> delayed = dset.coco_image(gid).imdelay(channels='B1|B2', space='image')
>>> print('delayed = {!r}'.format(delayed))
>>> print('delayed.finalize() = {!r}'.format(delayed.finalize()))
>>> delayed = dset.coco_image(gid).imdelay(channels='B1|B2|B11', space='image')
>>> print('delayed = {!r}'.format(delayed))
>>> print('delayed.finalize() = {!r}'.format(delayed.finalize()))
>>> delayed = dset.coco_image(gid).imdelay(channels='B8|B1', space='video')
>>> print('delayed = {!r}'.format(delayed))
>>> print('delayed.finalize() = {!r}'.format(delayed.finalize()))
```

```
>>> delayed = dset.coco_image(gid).imdelay(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.imdelay()
>>> final = delayed.finalize()
>>> assert final.shape == (512, 512, 3)
```

```
>>> delayed = coco_img.imdelay()
>>> final = delayed.finalize()
>>> print('final.shape = {}'.format(ub.urepr(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.imdelay(channels='B1|Aux:2:4')
>>> 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]
>>> asset_delayed = coco_img.imdelay(stream1, space='asset')
>>> img_delayed = coco_img.imdelay(stream1, space='image')
>>> vid_delayed = coco_img.imdelay(stream1, space='video')
>>> #
>>> aux_imdata = asset_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
>>> from delayed_image.delayed_nodes import CoordinateCompatibilityError
>>> with pytest.raises(CoordinateCompatibilityError):
>>>     aux_delayed2 = coco_img.imdelay(fused_channels, space='asset')
```

Example

```
>>> # Test loading at a specific resolution.
>>> from kwcoco.coco_image import *  # NOQA
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('vidshapes8-msi-multisensor')
>>> coco_img = dset.coco_image(1)
>>> coco_img.img['resolution'] = '1 meter'
>>> img_delayed1 = coco_img.imdelay(space='image')
>>> vid_delayed1 = coco_img.imdelay(space='video')
>>> # test with unitless request
```

(continues on next page)

(continued from previous page)

```
>>> img_delayed2 = coco_img.imdelay(space='image', resolution=3.1)
>>> vid_delayed2 = coco_img.imdelay(space='video', resolution='3.1 meter')
>>> np.ceil(img_delayed1.shape[0] / 3.1) == img_delayed2.shape[0]
>>> np.ceil(vid_delayed1.shape[0] / 3.1) == vid_delayed2.shape[0]
>>> # test with unitless data
>>> coco_img.img['resolution'] = 1
>>> img_delayed2 = coco_img.imdelay(space='image', resolution=3.1)
>>> vid_delayed2 = coco_img.imdelay(space='video', resolution='3.1 meter')
>>> np.ceil(img_delayed1.shape[0] / 3.1) == img_delayed2.shape[0]
>>> np.ceil(vid_delayed1.shape[0] / 3.1) == vid_delayed2.shape[0]
```

valid_region(*space='image'*)

If this image has a valid polygon, return it in image, or video space

Returns
None | kwimage.MultiPolygon

property warp_vid_from_img

Affine transformation that warps image space -> video space.

Returns
The transformation matrix

Return type
kwimage.Affine

property warp_img_from_vid

Affine transformation that warps video space -> image space.

Returns
The transformation matrix

Return type
kwimage.Affine

_warp_for_resolution(*space*, *resolution=None*)

Compute a transform from image-space to the requested space at a target resolution.

_annot_segmentation(*ann*, *space='video'*, *resolution=None*)

” Load annotation segmentations in a requested space at a target resolution.

Example

```
>>> from kwcoco.coco_image import *  # NOQA
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('vidshapes8-msi-multisensor')
>>> coco_img = dset.coco_image(1)
>>> coco_img.img['resolution'] = '1 meter'
>>> ann = coco_img.annots().objs[0]
>>> img_sseg = coco_img._annot_segmentation(ann, space='image')
>>> vid_sseg = coco_img._annot_segmentation(ann, space='video')
>>> img_sseg_2m = coco_img._annot_segmentation(ann, space='image', resolution=
↪'2 meter')
>>> vid_sseg_2m = coco_img._annot_segmentation(ann, space='video', resolution=
```

(continues on next page)

(continued from previous page)

```
↪'2 meter')
>>> print(f'img_sseg.area    = {img_sseg.area}')
>>> print(f'vid_sseg.area    = {vid_sseg.area}')
>>> print(f'img_sseg_2m.area = {img_sseg_2m.area}')
>>> print(f'vid_sseg_2m.area = {vid_sseg_2m.area}')
```

_annot_segmentations(*anns*, *space='video'*, *resolution=None*)

" Load multiple annotation segmentations in a requested space at a target resolution.

Example

```
>>> from kwcoco.coco_image import *  # NOQA
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('vidshapes8-msi-multisensor')
>>> coco_img = dset.coco_image(1)
>>> coco_img.img['resolution'] = '1 meter'
>>> ann = coco_img.annots().objs[0]
>>> img_sseg = coco_img._annot_segmentations([ann], space='image')
>>> vid_sseg = coco_img._annot_segmentations([ann], space='video')
>>> img_sseg_2m = coco_img._annot_segmentations([ann], space='image',␣
↪resolution='2 meter')
>>> vid_sseg_2m = coco_img._annot_segmentations([ann], space='video',␣
↪resolution='2 meter')
>>> print(f'img_sseg.area    = {img_sseg[0].area}')
>>> print(f'vid_sseg.area    = {vid_sseg[0].area}')
>>> print(f'img_sseg_2m.area = {img_sseg_2m[0].area}')
>>> print(f'vid_sseg_2m.area = {vid_sseg_2m[0].area}')
```

resolution(*space='image'*, *channel=None*, *RESOLUTION_KEY=None*)

Returns the resolution of this CocoImage in the requested space if known. Errors if this information is not registered.

Parameters

- **space** (*str*) – the space to the resolution of. Can be either "image", "video", or "asset".
- **channel** (*str* | *kwcoco.FusedChannelSpec* | *None*) – a channel that identifies a single asset, only relevant if asking for asset space

Returns

has items mag (with the magnitude of the resolution) and unit, which is a convinience and only loosely enforced.

Return type

Dict

Example

```
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('vidshapes8-multispectral')
>>> self = dset.coco_image(1)
>>> self.img['resolution'] = 1
>>> self.resolution()
>>> self.img['resolution'] = '1 meter'
>>> self.resolution(space='video')
{'mag': (1.0, 1.0), 'unit': 'meter'}
>>> self.resolution(space='asset', channel='B11')
>>> self.resolution(space='asset', channel='B1')
```

_scalefactor_for_resolution(*space*, *resolution*, *channel=None*, *RESOLUTION_KEY=None*)

Given image or video space, compute the scale factor needed to achieve the target resolution.

Use this to implement scale_resolution_from_img scale_resolution_from_vid

Parameters

- **space** (*str*) – the space to the resolution of. Can be either "image", "video", or "asset".
- **resolution** (*str* | *float* | *int*) – the resolution (ideally with units) you want.
- **channel** (*str* | *kwcoco.FusedChannelSpec* | *None*) – a channel that identifies a single asset, only relevant if asking for asset space

Returns

the x and y scale factor that can be used to scale the underlying "space" to acheive the requested resolution.

Return type

Tuple[float, float]

_detections_for_resolution(*space='video'*, *resolution=None*, *aids=None*, *RESOLUTION_KEY=None*)

This is slightly less than ideal in terms of API, but it will work for now.

add_auxiliary_item(***kwargs*)

delay(***kwargs*)

show(***kwargs*)

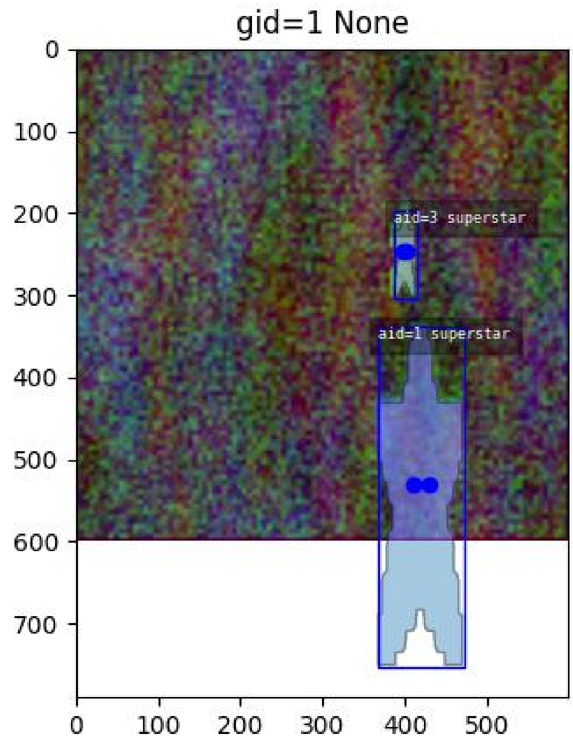
Show the image with matplotlib if possible

SeeAlso:

`kwcoco.CocoDataset.show_image()`

Example

```
>>> # xdoctest: +REQUIRES(module:kwplot)
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('vidshapes8-multispectral')
>>> self = dset.coco_image(1)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autoplt()
>>> self.show()
```



draw(***kwargs*)

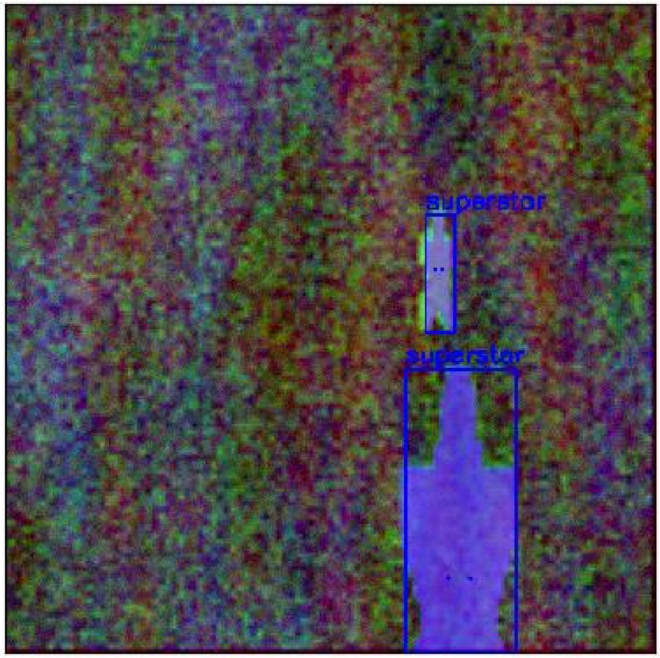
Draw the image on an ndarray using opencv

SeeAlso:

kwcoco.CocoDataset.draw_image()

Example

```
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('vidshapes8-multispectral')
>>> self = dset.coco_image(1)
>>> canvas = self.draw()
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(canvas)
```



class kwcoco.coco_image.**CocoAsset**(*asset*, *bundle_dpath=None*)

Bases: *_CocoObject*

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.

Example

```
>>> from kwcoco.coco_image import *  # NOQA
>>> self = CocoAsset({'warp_aux_to_img': 'foo'})
>>> assert 'warp_aux_to_img' in self
>>> assert 'warp_img_from_asset' in self
>>> assert 'warp_wld_from_asset' not in self
>>> assert 'warp_to_wld' not in self
>>> self['warp_aux_to_img'] = 'bar'
>>> assert self._proxy == {'warp_aux_to_img': 'bar'}
```

image_filepath()

class kwcoco.coco_image.**CocoVideo**(*obj*, *dset=None*, *bundle_dpath=None*)

Bases: *_CocoObject*

Object representing a single video.

Example

```
>>> from kwcoco.coco_image import *  # NOQA
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('vidshapes1')
>>> obj = dset.videos().objs[0]
>>> self = CocoVideo(obj, dset)
>>> print(f'self={self}')
```

class kwcoco.coco_image.**CocoAnnotation**(*obj*, *dset=None*, *bundle_dpath=None*)

Bases: *_CocoObject*

Object representing a single annotation.

Example

```
>>> from kwcoco.coco_image import *  # NOQA
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('vidshapes1')
>>> obj = dset.annots().objs[0]
>>> self = CocoAnnotation(obj, dset)
>>> print(f'self={self}')
```

class kwcoco.coco_image.**CocoCategory**(*obj*, *dset=None*, *bundle_dpath=None*)

Bases: *_CocoObject*

Object representing a single category.

Example

```
>>> from kwcoco.coco_image import *  # NOQA
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('vidshapes1')
>>> obj = dset.categories().objs[0]
>>> self = CocoCategory(obj, dset)
>>> print(f'self={self}')
```

class kwcoco.coco_image.**CocoTrack**(*obj*, *dset=None*, *bundle_dpath=None*)

Bases: *_CocoObject*

Object representing a single track.

Example

```
>>> from kwcoco.coco_image import *  # NOQA
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('vidshapes1')
>>> obj = dset.tracks().objs[0]
>>> self = CocoTrack(obj, dset)
>>> print(f'self={self}')
```

annots()

kwcoco.coco_image.**_delay_load_imglike**(*bundle_dpath*, *obj*, *nodata_method=None*)

kwcoco.coco_image.**parse_quantity**(*expr*)

kwcoco.coco_image.**coerce_resolution**(*expr*)

2.1.2.9 kwcoco.coco_objects1d module

Vectorized ORM-like objects used in conjunction with coco_dataset.

This powers the `.images()`, `.videos()`, and `.annotation()` methods of *kwcoco.CocoDataset*.

Todo:

- [ ] The use of methods vs properties is inconsistent. This needs to be fixed, but backwards compatability is a consideration.

See:
kwcoco.coco_dataset.MixinCocoObjects.categories() *kwcoco.coco_dataset.MixinCocoObjects.videos()* *kwcoco.coco_dataset.MixinCocoObjects.images()* *kwcoco.coco_dataset.MixinCocoObjects.annots()* *kwcoco.coco_dataset.MixinCocoObjects.tracks()*

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.

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])
```

(continues on next page)

(continued from previous page)

```
>>> self.lookup('id')
>>> self.lookup(['id'])
```

Parameters

- **ids** (*List[int]*) – list of ids
- **dset** (*CocoDataset*) – parent dataset
- **key** (*str*) – main object name (e.g. 'images', 'annotations')

property _id_to_obj

unique()

Removes any duplicates entries in this object

Returns
ObjectList1D

property ids

Returns: List[int]

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'])
```

(continues on next page)

(continued from previous page)

```
>>> 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)
```

sort_values(*by*, *reverse=False*, *key=None*)

Reorders the items by an attribute.

Parameters

- **by** (*str*) – The column attribute to sort by
- **key** (*Callable | None*) – Apply the key function to the values before sorting.

Returns
copy of this object with new ids

Return type
ObjectList1D

Example

```
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('vidshapes8')
>>> self = dset.images()
>>> new = self.sort_values('frame_index')
>>> frame_idxs = new.lookup('frame_index')
>>> assert sorted(frame_idxs) == frame_idxs
```

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

Return type
Dict[int, Any] | List[Any]

Example

```
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo()
>>> self = dset.annots()
>>> self.get('id')
>>> self.get(key='foo', default=None, keepid=True)
```

Example

```
>>> # xdoctest: +REQUIRES(module:sqlalchemy)
>>> import kwcoco
>>> dct_dset = kwcoco.CocoDataset.demo('vidshapes8', rng=303232)
>>> dct_dset.anns[3]['blorgo'] = 3
>>> dct_dset.annots().lookup('blorgo', default=None)
>>> for a in dct_dset.anns.values():
...     a['wizard'] = '10!'
>>> dset = dct_dset.view_sql(force_rewrite=1)
>>> assert dset.anns[3]['blorgo'] == 3
>>> assert dset.anns[3]['wizard'] == '10!'
>>> assert 'blorgo' not in dset.anns[2]
>>> dset.annots().lookup('blorgo', default=None)
>>> dset.annots().lookup('wizard', default=None)
>>> import pytest
>>> with pytest.raises(KeyError):
>>>     dset.annots().lookup('blorgo')
>>> dset.annots().lookup('wizard')
>>> #self = dset.annots()
```

_iter_get(*key*, *default=NoParam*)

Iterator version of get, not in stable API yet.

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.urepr(dset.imgs, nl=1)))
>>> self.get('my-key2')
```

_set(*key*, *values*)

faster less safe version of set

_lookup(*key*, *default=NoParam*)

Example

```
>>> # xdoctest: +REQUIRES(--benchmark)
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('shapes256')
>>> self = annots = dset.annots()
>>> #
>>> import timerit
>>> ti = timerit.Timerit(100, bestof=10, verbose=2)
>>> #
>>> for timer in ti.reset('lookup'):
>>>     with timer:
>>>         self.lookup('image_id')
>>> #
>>> for timer in ti.reset('_lookup'):
>>>     with timer:
>>>         self._lookup('image_id')
>>> #
>>> for timer in ti.reset('image_id'):
>>>     with timer:
>>>         self.image_id
>>> #
>>> for timer in ti.reset('raw1'):
>>>     with timer:
>>>         key = 'image_id'
>>>         [self._dset.anns[_id][key] for _id in self._ids]
>>> #
>>> for timer in ti.reset('raw2'):
>>>     with timer:
>>>         anns = self._dset.anns
>>>         key = 'image_id'
>>>         [anns[_id][key] for _id in self._ids]
>>> #
>>> for timer in ti.reset('lut-gen'):
>>>     with timer:
>>>         _lut = self._obj_lut
>>>         objs = (_lut[_id] for _id in self._ids)
>>>         [obj[key] for obj in objs]
>>> #
>>> for timer in ti.reset('lut-gen-single'):
>>>     with timer:
>>>         _lut = self._obj_lut
>>>         [_lut[_id][key] for _id in self._ids]
```

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.urepr(attrs, nl=1)))
```

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

Bases: NiceRepr

An object for holding a groups of *ObjectList1D* objects

Parameters

- **groups** (*List[ObjectList1D]*) – list of object lists
- **dset** (*CocoDataset*) – parent dataset

_lookup(*key*)

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))
```

Parameters

- **ids** (*List[int]*) – list of category ids
- **dset** (*CocoDataset*) – parent dataset

property cids

property name

property supercategory

class kwcoco.coco_objects1d.**Videos**(*ids*, *dset*)

Bases: *ObjectList1D*

Vectorized access to video attributes

SeeAlso:
kwcoco.coco_dataset.MixinCocoObjects.videos()

Example

```
>>> from kwcoco.coco_objects1d import Videos  # NOQA
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('vidshapes5')
>>> ids = list(dset.index.videos.keys())
>>> self = Videos(ids, dset)
>>> print('self = {!r}'.format(self))
self = <Videos(num=5) at ...>
```

Parameters

- **ids** (*List[int]*) – list of video ids
- **dset** (*CocoDataset*) – parent dataset

property images

Returns
ImageGroups

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo('vidshapes8').videos()
>>> print(self.images)
<ImageGroups(n=8, m=2.0, s=0.0)>
```

class kwcoco.coco_objects1d.**Images**(*ids*, *dset*)

Bases: *ObjectList1D*

Vectorized access to image attributes

Example

```
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('photos')
>>> images = dset.images()
>>> print('images = {}'.format(images))
images = <Images(num=3)...>
>>> print('images.gname = {}'.format(images.gname))
images.gname = ['astro.png', 'carl.jpg', 'stars.png']
```

SeeAlso:
kwcoco.coco_dataset.MixinCocoObjects.images()

Parameters

- **ids** (*List[int]*) – list of image ids

- **dset** (*CocoDataset*) – parent dataset

property coco_images

property gids

property gname

property gpath

property width

property height

property size

Example:

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo().images()
>>> self._dset._ensure_imgsize()
...
>>> print(self.size)
[(512, 512), (328, 448), (256, 256)]
```

property area

Returns
List[float]

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo().images()
>>> self._dset._ensure_imgsize()
...
>>> print(self.area)
[262144, 146944, 65536]
```

property n_annots

Returns
List[int]

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo().images()
>>> print(ub.urepr(self.n_annots, nl=0))
[9, 2, 0]
```

property aids

Returns
List[set]

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo().images()
>>> print(ub.urepr(list(map(list, self.aids)), nl=0))
[[1, 2, 3, 4, 5, 6, 7, 8, 9], [10, 11], []]
```

property annots

Returns
AnnotGroups

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo().images()
>>> print(self.annots)
<AnnotGroups(n=3, m=3.7, s=3.9)>
```

class kwcoco.coco_objects1d.**Annots**(*ids*, *dset*)

Bases: *ObjectList1D*

Vectorized access to annotation attributes

SeeAlso:
kwcoco.coco_dataset.MixinCocoObjects.annots()

Example

```
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('photos')
>>> annots = dset.annots()
>>> print('annots = {}'.format(annots))
annots = <Annots(num=11)>
>>> image_ids = annots.lookup('image_id')
>>> print('image_ids = {}'.format(image_ids))
image_ids = [1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2]
```

Parameters

- **ids** (*List[int]*) – list of annotation ids
- **dset** (*CocoDataset*) – parent dataset

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[str]

property category_names

Get the column of category names

Returns
List[str]

property detections

Get the kwimage-style detection objects

Returns
kwimage.Detections

Example

```
>>> # xdoctest: +REQUIRES(module:kwimage)
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo('shapes32').annots([1, 2, 11])
>>> dets = self.detections
>>> print('dets.data = {!r}'.format(dets.data))
>>> print('dets.meta = {!r}'.format(dets.meta))
```

property boxes

Get the column of kwimage-style bounding boxes

Returns
kwimage.Boxes

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo().annots([1, 2, 11])
>>> print(self.boxes)
<Boxes(xywh,
    array([[ 10,  10, 360, 490],
           [350,   5, 130, 290],
           [156, 130,  45,  18]]))>
```

property xywh

Returns raw boxes

DEPRECATED.

Returns
raw boxes in xywh format

Return type
List[List[int]]

Example

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

class kwcoco.coco_objects1d.Tracks(*ids*, *dset*)

Bases: *ObjectList1D*

Vectorized access to track attributes

SeeAlso:
kwcoco.coco_dataset.MixinCocoObjects.tracks()

Example

```
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('vidshapes1', num_tracks=4)
>>> tracks = dset.tracks()
>>> print('tracks = {}'.format(tracks))
tracks = <Tracks(num=4)>
>>> tracks.name
['track_001', 'track_002', 'track_003', 'track_004']
```

Parameters

- **ids** (*List[int]*) – list of track ids
- **dset** (*CocoDataset*) – parent dataset

property track_ids

The annotation ids of this column of annotations

property name

property annots

Example:

```
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('vidshapes1', num_tracks=4)
>>> self = dset.tracks()
>>> print(self.annots)
<AnnotGroups(n=4, m=2.0, s=0.0)>
```

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

Bases: *ObjectGroups*

Annotation groups are vectorized lists of lists.

Each item represents a set of annotations that corresopnds with something (i.e. belongs to a particular image).

Example

```
>>> from kwcoco.coco_objects1d import ImageGroups
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('photos')
>>> images = dset.images()
>>> # Requesting the "annots" property from a Images object
>>> # will return an AnnotGroups object
>>> group: AnnotGroups = images.annots
>>> # Printing the group gives info on the mean/std of the number
>>> # of items per group.
>>> print(group)
<AnnotGroups(n=3, m=3.7, s=3.9)...>
>>> # Groups are fairly restrictive, they dont provide property level
>>> # access in many cases, but the lookup method is available
>>> print(group.lookup('id'))
[[1, 2, 3, 4, 5, 6, 7, 8, 9], [10, 11], []]
>>> print(group.lookup('image_id'))
[[1, 1, 1, 1, 1, 1, 1, 1, 1], [2, 2], []]
>>> print(group.lookup('category_id'))
[[1, 2, 3, 4, 5, 5, 5, 5, 5], [6, 4], []]
```

Parameters

- **groups** (*List[ObjectList1D]*) – list of object lists
- **dset** (*CocoDataset*) – parent dataset

property cids

Get the grouped category ids for annotations in this group

Return type

List[List[int]]

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo('photos').images().annots
>>> print('self.cids = {}'.format(ub.urepr(self.cids, nl=0)))
self.cids = [[1, 2, 3, 4, 5, 5, 5, 5, 5], [6, 4], []]
```

property cnames

Get the grouped category names for annotations in this group

Return type
List[List[str]]

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo('photos').images().annots
>>> print('self.cnames = {}'.format(ub.urepr(self.cnames, nl=0)))
self.cnames = [['astronaut', 'rocket', 'helmet', 'mouth', 'star', 'star', 'star
↪', 'star', 'star'], ['astronomer', 'mouth'], []]
```

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

Bases: *ObjectGroups*

Image groups are vectorized lists of other Image objects.

Each item represents a set of images that corresopnds with something (i.e. belongs to a particular video).

Example

```
>>> from kwcoco.coco_objects1d import ImageGroups
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('vidshapes8')
>>> videos = dset.videos()
>>> # Requesting the "images" property from a Videos object
>>> # will return an ImageGroups object
>>> group: ImageGroups = videos.images
>>> # Printing the group gives info on the mean/std of the number
>>> # of items per group.
>>> print(group)
<ImageGroups(n=8, m=2.0, s=0.0)...>
>>> # Groups are fairly restrictive, they dont provide property level
>>> # access in many cases, but the lookup method is available
>>> print(group.lookup('id'))
[[1, 2], [3, 4], [5, 6], [7, 8], [9, 10], [11, 12], [13, 14], [15, 16]]
>>> print(group.lookup('video_id'))
[[1, 1], [2, 2], [3, 3], [4, 4], [5, 5], [6, 6], [7, 7], [8, 8]]
>>> print(group.lookup('frame_index'))
[[0, 1], [0, 1], [0, 1], [0, 1], [0, 1], [0, 1], [0, 1], [0, 1]]
```

Parameters

- **groups** (*List[ObjectList1D]*) – list of object lists

- **dset** (*CocoDataset*) – parent dataset

2.1.2.10 kwcoco.coco_schema module

The place where the formal KWCOCO schema is defined.

CommandLine

```
python -m kwcoco.coco_schema
xdoctest -m kwcoco.coco_schema __doc__
```

Todo:

- [ ] Perhaps use voluptuous instead?

Example

```
>>> import kwcoco
>>> from kwcoco.coco_schema import COCO_SCHEMA
>>> import jsonschema
>>> dset = kwcoco.CocoDataset.demo('shapes1')
>>> # print('dset.dataset = {}'.format(ub.urepr(dset.dataset, nl=2)))
>>> COCO_SCHEMA.validate(dset.dataset)
```

```
>>> try:
>>>     jsonschema.validate(dset.dataset, schema=COCO_SCHEMA)
>>> except jsonschema.exceptions.ValidationError as ex:
>>>     vali_ex = ex
>>>     print('ex = {!r}'.format(ex))
>>>     raise
>>> except jsonschema.exceptions.SchemaError as ex:
>>>     print('ex = {!r}'.format(ex))
>>>     schema_ex = ex
>>>     print('schema_ex.instance = {}'.format(ub.urepr(schema_ex.instance, nl=-1)))
>>>     raise
```

```
>>> # Test the multispectral image defintino
>>> import copy
>>> dataset = dset.copy().dataset
>>> img = dataset['images'][0]
>>> img.pop('file_name')
>>> import pytest
>>> with pytest.raises(jsonschema.ValidationError):
>>>     COCO_SCHEMA.validate(dataset)
>>> import pytest
>>> img['auxiliary'] = [{'file_name': 'foobar'}]
>>> with pytest.raises(jsonschema.ValidationError):
>>>     COCO_SCHEMA.validate(dataset)
```

(continues on next page)

(continued from previous page)

```
>>> img['name'] = 'asset-only images must have a name'
>>> COCO_SCHEMA.validate(dataset)
```

kwcoco.coco_schema.**deprecated**(**args*)

kwcoco.coco_schema.**TUPLE**(**args*, ***kw*)

2.1.2.11 kwcoco.coco_sql_dataset module

Finally got a baseline implementation of an SQLite backend for COCO datasets. This mostly plugs into my existing tools (as long as only read operations are used; haven't impelmented writing yet) by duck-typing the dict API.

This solves the issue of forking and then accessing nested dictionaries in the JSON-style COCO objects. (When you access the dictionary Python will increment a reference count which triggers copy-on-write for whatever memory page that data happened to live in. Non-contiguous access had the effect of excessive memory copies).

For "medium sized" datasets its quite a bit slower. Running through a torch DataLoader with 4 workers for 10,000 images executes at a rate of 100Hz but takes 850MB of RAM. Using the duck-typed SQL backend only uses 500MB (which includes the cost of caching), but runs at 45Hz (which includes the benefit of caching).

However, once I scale up to 100,000 images I start seeing benefits. The in-memory dictionary interface chugs at 1.05HZ, and is taking more than 4GB of memory at the time I killed the process (eta was over an hour). The SQL backend ran at 45Hz and took about 3 minutes and used about 2.45GB of memory.

Without a cache, SQL runs at 30HZ and takes 400MB for 10,000 images, and for 100,000 images it gets 30Hz with 1.1GB. There is also a much larger startup time. I'm not exactly sure what it is yet, but its probably some preprocessing I'm doing.

Using a LRU cache we get 45Hz and 1.05GB of memory, so that's a clear win. We do need to be sure to disable the cache if we ever implement write mode.

I'd like to be a bit faster on the medium sized datasets (I'd really like to avoid caching rows, which is why the speed is currently semi-reasonable), but I don't think I can do any better than this because single-row lookup time is *O(log(N))* for sqlite, whereas its *O(1)* for dictionaries. (I wish sqlite had an option to create a hash-table index for a table, but I dont think it does). I optimized as many of the dictionary operations as possible (for instance, iterating through keys, values, and items should be O(N) instead of *O(N log(N))*), but the majority of the runtime cost is in the single-row lookup time.

There are a few questions I still have if anyone has insight:

- Say I want to select a subset of K rows from a table with N entries, and I have a list of all of the rowids that I want. Is there any way to do this better than `O(K log(N))`? I tried using a `SELECT col FROM table WHERE id IN (?, ?, ?, ?, ...)` filling in enough *?* as there are rows in my subset. I'm not sure what the complexity of using a query like this is. I'm not sure what the *IN* implementation looks like. Can this be done more efficiently by with a temporary table and a `JOIN`?
- There really is no way to do `O(1)` row lookup in sqlite right? Is there a way in PostgreSQL or some other backend sqlalchemy supports?

I found that PostgreSQL does support hash indexes: https://www.postgresql.org/docs/13/indexes-types.html I'm really not interested in setting up a global service though . I also found a 10-year old thread with a hash-index feature request for SQLite, which I unabashedly resurrected http://sqlite.1065341.n5.nabble.com/Feature-request-hash-index-td23367.html https://web.archive.org/web/20210326010915/http://sqlite.1065341.n5.nabble.com/Feature-request-hash-index-td23367.html

---

Todo:

- **[ ] We get better speeds with raw SQL over alchemy. Can we mitigate the**
  speed difference so we can take advantage of alchemy's expressiveness?

---

class kwcoco.coco_sql_dataset.**FallbackCocoBase**

Bases: `object`

_decl_class_registry = {}

class kwcoco.coco_sql_dataset.**Category**(***kwargs*)

Bases: `Base`

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in `kwargs`.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

id
: unique internal id

name
: unique external name or identifier

alias
: list of alter egos

supercategory
: coarser category name

_unstructured

_sa_class_manager = {'_unstructured': <sqlalchemy.orm.attributes.InstrumentedAttribute object>, 'alias': <sqlalchemy.orm.attributes.InstrumentedAttribute object>, 'id': <sqlalchemy.orm.attributes.InstrumentedAttribute object>, 'name': <sqlalchemy.orm.attributes.InstrumentedAttribute object>, 'supercategory': <sqlalchemy.orm.attributes.InstrumentedAttribute object>}

class kwcoco.coco_sql_dataset.**KeypointCategory**(***kwargs*)

Bases: `Base`

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in `kwargs`.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

id
: unique internal id

name
: unique external name or identifier

alias
: list of alter egos

supercategory

coarser category name

reflection_id

if augmentation reflects the image, change keypoint id to this

_unstructured

_sa_class_manager = {'_unstructured': <sqlalchemy.orm.attributes.InstrumentedAttribute object>, 'alias': <sqlalchemy.orm.attributes.InstrumentedAttribute object>, 'id': <sqlalchemy.orm.attributes.InstrumentedAttribute object>, 'name': <sqlalchemy.orm.attributes.InstrumentedAttribute object>, 'reflection_id': <sqlalchemy.orm.attributes.InstrumentedAttribute object>, 'supercategory': <sqlalchemy.orm.attributes.InstrumentedAttribute object>}

class kwcoco.coco_sql_dataset.**Video**(***kwargs*)

Bases: Base

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in kwargs.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

id

unique internal id

name

caption

width

height

_unstructured

_sa_class_manager = {'_unstructured': <sqlalchemy.orm.attributes.InstrumentedAttribute object>, 'caption': <sqlalchemy.orm.attributes.InstrumentedAttribute object>, 'height': <sqlalchemy.orm.attributes.InstrumentedAttribute object>, 'id': <sqlalchemy.orm.attributes.InstrumentedAttribute object>, 'name': <sqlalchemy.orm.attributes.InstrumentedAttribute object>, 'width': <sqlalchemy.orm.attributes.InstrumentedAttribute object>}

class kwcoco.coco_sql_dataset.**Image**(***kwargs*)

Bases: Base

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in kwargs.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

id

unique internal id

name

file_name

width

height

video_id

timestamp

frame_index

channels

See ChannelSpec

warp_img_to_vid

See TransformSpec

auxiliary

_unstructured

_sa_class_manager = {'_unstructured': <sqlalchemy.orm.attributes.InstrumentedAttribute object>, 'auxiliary': <sqlalchemy.orm.attributes.InstrumentedAttribute object>, 'channels': <sqlalchemy.orm.attributes.InstrumentedAttribute object>, 'file_name': <sqlalchemy.orm.attributes.InstrumentedAttribute object>, 'frame_index': <sqlalchemy.orm.attributes.InstrumentedAttribute object>, 'height': <sqlalchemy.orm.attributes.InstrumentedAttribute object>, 'id': <sqlalchemy.orm.attributes.InstrumentedAttribute object>, 'name': <sqlalchemy.orm.attributes.InstrumentedAttribute object>, 'timestamp': <sqlalchemy.orm.attributes.InstrumentedAttribute object>, 'video_id': <sqlalchemy.orm.attributes.InstrumentedAttribute object>, 'warp_img_to_vid': <sqlalchemy.orm.attributes.InstrumentedAttribute object>, 'width': <sqlalchemy.orm.attributes.InstrumentedAttribute object>}

class kwcoco.coco_sql_dataset.**Track**(***kwargs*)

Bases: Base

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in kwargs.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

id

unique internal id

name

_unstructured

_sa_class_manager = {'_unstructured': <sqlalchemy.orm.attributes.InstrumentedAttribute object>, 'id': <sqlalchemy.orm.attributes.InstrumentedAttribute object>, 'name': <sqlalchemy.orm.attributes.InstrumentedAttribute object>}

class kwcoco.coco_sql_dataset.**Annotation**(***kwargs*)

Bases: Base

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in kwargs.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

id

image_id

category_id

track_id

segmentation

keypoints

bbox

_bbox_x

_bbox_y

_bbox_w

_bbox_h

score

weight

prob

iscrowd

caption

_unstructured

_sa_class_manager = {'_bbox_h': <sqlalchemy.orm.attributes.InstrumentedAttribute object>, '_bbox_w': <sqlalchemy.orm.attributes.InstrumentedAttribute object>, '_bbox_x': <sqlalchemy.orm.attributes.InstrumentedAttribute object>, '_bbox_y': <sqlalchemy.orm.attributes.InstrumentedAttribute object>, '_unstructured': <sqlalchemy.orm.attributes.InstrumentedAttribute object>, 'bbox': <sqlalchemy.orm.attributes.InstrumentedAttribute object>, 'caption': <sqlalchemy.orm.attributes.InstrumentedAttribute object>, 'category_id': <sqlalchemy.orm.attributes.InstrumentedAttribute object>, 'id': <sqlalchemy.orm.attributes.InstrumentedAttribute object>, 'image_id': <sqlalchemy.orm.attributes.InstrumentedAttribute object>, 'iscrowd': <sqlalchemy.orm.attributes.InstrumentedAttribute object>, 'keypoints': <sqlalchemy.orm.attributes.InstrumentedAttribute object>, 'prob': <sqlalchemy.orm.attributes.InstrumentedAttribute object>, 'score': <sqlalchemy.orm.attributes.InstrumentedAttribute object>, 'segmentation': <sqlalchemy.orm.attributes.InstrumentedAttribute object>, 'track_id': <sqlalchemy.orm.attributes.InstrumentedAttribute object>, 'weight': <sqlalchemy.orm.attributes.InstrumentedAttribute object>}

kwcoco.coco_sql_dataset.**cls**

alias of *Video*

kwcoco.coco_sql_dataset.**orm_to_dict**(*obj*)

kwcoco.coco_sql_dataset.**dict_restructure**(*item*)

Removes the unstructured field so the API is transparent to the user.

kwcoco.coco_sql_dataset.**_orm_yielder**(*query*, *size=300*)

TODO: figure out the best way to yield, in batches or otherwise

kwcoco.coco_sql_dataset.**_raw_yielder**(*result*, *size=300*)

TODO: figure out the best way to yield, in batches or otherwise

kwcoco.coco_sql_dataset.**_new_proxy_cache**()

By returning None, we wont use item caching

class kwcoco.coco_sql_dataset.**SqlListProxy**(*session*, *cls*)

Bases: NiceRepr

A view of an SQL table that behaves like a Python list

class kwcoco.coco_sql_dataset.**SqlDictProxy**(*session*, *cls*, *keyattr=None*, *ignore_null=False*)

Bases: *DictLike*

Duck-types an SQL table as a dictionary of dictionaries.

The key is specified by an indexed column (by default it is the *id* column). The values are dictionaries containing all data for that row.

Note: With SQLite indexes are B-Trees so lookup is O(log(N)) and not O(1) as will regular dictionaries. Iteration should still be O(N), but databases have much more overhead than Python dictionaries.

Parameters

- **session** (*sqlalchemy.orm.session.Session*) – the sqlalchemy session
- **cls** (*Type*) – the declarative sqlalchemy table class
- **keyattr** (*Column*) – the indexed column to use as the keys
- **ignore_null** (*bool*) – if True, ignores any keys set to NULL, otherwise NULL keys are allowed.

Example

```
>>> # xdoctest: +REQUIRES(module:sqlalchemy)
>>> from kwcoco.coco_sql_dataset import *  # NOQA
>>> import pytest
>>> sql_dset, dct_dset = demo(num=10)
>>> proxy = sql_dset.index.anns
```

```
>>> keys = list(proxy.keys())
>>> values = list(proxy.values())
>>> items = list(proxy.items())
>>> item_keys = [t[0] for t in items]
>>> item_vals = [t[1] for t in items]
>>> lut_vals = [proxy[key] for key in keys]
>>> assert item_vals == lut_vals == values
>>> assert item_keys == keys
>>> assert len(proxy) == len(keys)
```

```
>>> goodkey1 = keys[1]
>>> badkey1 = -100000000000
>>> badkey2 = 'foobarbazbiz'
>>> assert goodkey1 in proxy
>>> assert badkey1 not in proxy
>>> assert badkey2 not in proxy
>>> with pytest.raises(KeyError):
>>>     proxy[badkey1]
>>> with pytest.raises(KeyError):
>>>     proxy[badkey2]
>>> badkey3 = object()
>>> assert badkey3 not in proxy
>>> with pytest.raises(KeyError):
>>>     proxy[badkey3]
```

```
>>> # xdoctest: +SKIP
>>> from kwcoco.coco_sql_dataset import _benchmark_dict_proxy_ops
>>> ti = _benchmark_dict_proxy_ops(proxy)
>>> print('ti.measures = {}'.format(ub.urepr(ti.measures, nl=2, align=':',␣
↪precision=6)))
```

Example

```
>>> # xdoctest: +REQUIRES(module:sqlalchemy)
>>> from kwcoco.coco_sql_dataset import *  # NOQA
>>> import kwcoco
>>> # Test the variant of the SqlDictProxy where we ignore None keys
>>> # This is the case for name_to_img and file_name_to_img
>>> dct_dset = kwcoco.CocoDataset.demo('shapes1')
>>> dct_dset.add_image(name='no_file_image1')
>>> dct_dset.add_image(name='no_file_image2')
>>> dct_dset.add_image(name='no_file_image3')
>>> sql_dset = dct_dset.view_sql(memory=True)
>>> assert len(dct_dset.index.imgs) == 4
>>> assert len(dct_dset.index.file_name_to_img) == 1
>>> assert len(dct_dset.index.name_to_img) == 3
>>> assert len(sql_dset.index.imgs) == 4
>>> assert len(sql_dset.index.file_name_to_img) == 1
>>> assert len(sql_dset.index.name_to_img) == 3
```

```
>>> proxy = sql_dset.index.file_name_to_img
>>> assert len(list(proxy.keys())) == 1
>>> assert len(list(proxy.values())) == 1
```

```
>>> proxy = sql_dset.index.name_to_img
>>> assert len(list(proxy.keys())) == 3
>>> assert len(list(proxy.values())) == 3
```

```
>>> proxy = sql_dset.index.imgs
>>> assert len(list(proxy.keys())) == 4
>>> assert len(list(proxy.values())) == 4
```

_uncached_getitem(*key*)

The uncached getitem call

keys()

values()

items()

class kwcoco.coco_sql_dataset.**SqlIdGroupDictProxy**(*session*, *valattr*, *keyattr*, *parent_keyattr=None*, *order_attr=None*, *order_id=None*)

Bases: *DictLike*

Similar to *SqlDictProxy*, but maps ids to groups of other ids.

Simulates a dictionary that maps ids of a parent table to all ids of another table corresponding to rows where a specific column has that parent id.

The items in the group can be sorted by the `order_attr` if specified. The `order_attr` can belong to another table if `parent_order_id` and `self_order_id` are specified.

For example, imagine two tables: images with one column (id) and annotations with two columns (id, image_id). This class can help provide a mpaping from each *image.id* to a *Set[annotation.id]* where those annotation rows have *annotation.image_id = image.id*.

Example

```
>>> # xdoctest: +REQUIRES(module:sqlalchemy)
>>> from kwcoco.coco_sql_dataset import *  # NOQA
>>> sql_dset, dct_dset = demo(num=10)
>>> proxy = sql_dset.index.gid_to_aids
```

```
>>> keys = list(proxy.keys())
>>> values = list(proxy.values())
>>> items = list(proxy.items())
>>> item_keys = [t[0] for t in items]
>>> item_vals = [t[1] for t in items]
>>> lut_vals = [proxy[key] for key in keys]
>>> assert item_vals == lut_vals == values
>>> assert item_keys == keys
>>> assert len(proxy) == len(keys)
```

```
>>> # xdoctest: +SKIP
>>> from kwcoco.coco_sql_dataset import _benchmark_dict_proxy_ops
>>> ti = _benchmark_dict_proxy_ops(proxy)
>>> print('ti.measures = {}'.format(ub.urepr(ti.measures, nl=2, align=':',␣
↪precision=6)))
```

Example

```
>>> # xdoctest: +REQUIRES(module:sqlalchemy)
>>> from kwcoco.coco_sql_dataset import *  # NOQA
>>> import kwcoco
>>> # Test the group sorted variant of this by using vidid_to_gids
>>> # where the "gids" must be sorted by the image frame indexes
>>> dct_dset = kwcoco.CocoDataset.demo('vidshapes1')
>>> dct_dset.add_image(name='frame-index-order-demo1', frame_index=-30, video_id=1)
>>> dct_dset.add_image(name='frame-index-order-demo2', frame_index=10, video_id=1)
>>> dct_dset.add_image(name='frame-index-order-demo3', frame_index=3, video_id=1)
>>> dct_dset.add_video(name='empty-video1')
>>> dct_dset.add_video(name='empty-video2')
>>> dct_dset.add_video(name='empty-video3')
>>> sql_dset = dct_dset.view_sql(memory=True)
>>> orig = dct_dset.index.vidid_to_gids
>>> proxy = sql_dset.index.vidid_to_gids
>>> from kwcoco.util.util_json import indexable_allclose
>>> assert indexable_allclose(orig, dict(proxy))
>>> items = list(proxy.items())
>>> vals = list(proxy.values())
>>> keys = list(proxy.keys())
>>> assert len(keys) == len(vals)
>>> assert dict(zip(keys, vals)) == dict(items)
```

Parameters

- **session** (*sqlalchemy.orm.session.Session*) – the sqlalchemy session
- **valattr** (*InstrumentedAttribute*) – The column to lookup as a value
- **keyattr** (*InstrumentedAttribute*) – The column to use as a key
- **parent_keyattr** (*InstrumentedAttribute | None*) – The column of the table corresponding to the key. If unspecified the column in the indexed table is used which may be less efficient.
- **order_attr** (*InstrumentedAttribute | None*) – This is the attribute that the returned results will be ordered by
- **order_id** (*InstrumentedAttribute | None*) – if order_attr belongs to another table, then this must be a column of the value table that corresponds to the primary key of the table used for ordering (e.g. when ordering annotations by image frame index, this must be the annotation image id)

_uncached_getitem(*key*)

getitem without the cache

keys()

items()

values()

class kwcoco.coco_sql_dataset.**CocoSqlIndex**

Bases: object

Simulates the dictionary provided by *kwcoco.coco_dataset.CocoIndex*

build(*parent*)

_set_alchemy_mode(*mode*)

kwcoco.coco_sql_dataset.**_handle_sql_uri**(*uri*)

Temporary function to deal with URI. Modern tools seem to use RFC 3968 URIs, but sqlalchemy uses RFC 1738. Attempt to gracefully handle special cases. With a better understanding of the above specs, this function may be able to be written more eloquently.

class kwcoco.coco_sql_dataset.**CocoSqlDatabase**(*uri=None*, *tag=None*, *img_root=None*)

Bases: *AbstractCocoDataset*, *MixinCocoAccessors*, *MixinCocoObjects*, *MixinCocoStats*, *MixinCocoDraw*, NiceRepr

Provides an API nearly identical to kwcoco.CocoDatabase, but uses an SQL backend data store. This makes it robust to copy-on-write memory issues that arise when forking, as discussed in[1].

Note: By default constructing an instance of the CocoSqlDatabase does not create a connection to the databse. Use the *connect()* method to open a connection.

References

Example

```
>>> # xdoctest: +REQUIRES(module:sqlalchemy)
>>> from kwcoco.coco_sql_dataset import *  # NOQA
>>> sql_dset, dct_dset = demo()
>>> dset1, dset2 = sql_dset, dct_dset
>>> tag1, tag2 = 'dset1', 'dset2'
>>> assert_dsets_allclose(sql_dset, dct_dset)
```

MEMORY_URI = 'sqlite:///:memory:'

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

Create an SQL CocoDataset from the input pointer.

[1] https://github.com/pytorch/pytorch/issues/13246

Example

import kwcoco dset = kwcoco.CocoDataset.demo('shapes8') data = dset.fpath self = CocoSqlDatabase.coerce(data)

from kwcoco.coco_sql_dataset import CocoSqlDatabase import kwcoco dset = kwcoco.CocoDataset.coerce('spacenet7.kwcoco.json')

self = CocoSqlDatabase.coerce(dset)

from kwcoco.coco_sql_dataset import CocoSqlDatabase sql_dset = CocoSqlDatabase.coerce('spacenet7.kwcoco.json')

from kwcoco.coco_sql_dataset import CocoSqlDatabase import kwcoco sql_dset = kwcoco.CocoDataset.coerce('_spacenet7.kwcoco.view.v006.sqlite')

disconnect()

Drop references to any SQL or cache objects

connect(*readonly=False*, *verbose=0*, *must_exist=False*)

Connects this instance to the underlying database.

References

details on read only mode, some of these didnt seem to work https://github.com/sqlalchemy/sqlalchemy/blob/master/lib/sqlalchemy/dialects/sqlite/pysqlite.py#L71 https://github.com/pudo/dataset/issues/136 https://writeonly.wordpress.com/2009/07/16/simple-read-only-sqlalchemy-sessions/

CommandLine

```
KWCOCO_WITH_POSTGRESQL=1 xdoctest -m /home/joncrall/code/kwcoco/kwcoco/coco_sql_
↪dataset.py CocoSqlDatabase.connect
```

Example

```
>>> # xdoctest: +REQUIRES(env:KWCOCO_WITH_POSTGRESQL)
>>> # xdoctest: +REQUIRES(module:sqlalchemy)
>>> # xdoctest: +REQUIRES(module:psycopg2)
>>> from kwcoco.coco_sql_dataset import *  # NOQA
>>> dset = CocoSqlDatabase('postgresql+psycopg2://kwcoco:kwcoco_
↪pw@localhost:5432/mydb')
>>> self = dset
>>> dset.connect(verbose=1)
```

property fpath

delete(*verbose=0*)

table_names()

populate_from(*dset*, *verbose=1*)

Copy the information in a `CocoDataset` into this SQL database.

CommandLine

```
xdoctest -m kwcoco.coco_sql_dataset CocoSqlDatabase.populate_from:1
```

Example

```
>>> # xdoctest: +REQUIRES(module:sqlalchemy)
>>> from kwcoco.coco_sql_dataset import _benchmark_dset_readtime  # NOQA
>>> import kwcoco
>>> from kwcoco.coco_sql_dataset import *
>>> dset2 = dset = kwcoco.CocoDataset.demo()
>>> dset2.clear_annotations()
>>> dset1 = self = CocoSqlDatabase('sqlite:///:memory:')
>>> self.connect()
>>> self.populate_from(dset)
>>> dset1_images = list(dset1.dataset['images'])
>>> print('dset1_images = {}'.format(ub.urepr(dset1_images, nl=1)))
>>> print(dset2.dumps(newlines=True))
>>> assert_dsets_allclose(dset1, dset2, tag1='sql', tag2='dct')
>>> ti_sql = _benchmark_dset_readtime(dset1, 'sql')
>>> ti_dct = _benchmark_dset_readtime(dset2, 'dct')
>>> print('ti_sql.rankings = {}'.format(ub.urepr(ti_sql.rankings, nl=2,␣
↪precision=6, align=':')))
>>> print('ti_dct.rankings = {}'.format(ub.urepr(ti_dct.rankings, nl=2,␣
↪precision=6, align=':')))
```

Example

```
>>> # xdoctest: +REQUIRES(module:sqlalchemy)
>>> from kwcoco.coco_sql_dataset import _benchmark_dset_readtime  # NOQA
>>> import kwcoco
>>> from kwcoco.coco_sql_dataset import *
>>> dset2 = dset = kwcoco.CocoDataset.demo('vidshapes1')
>>> dset1 = self = CocoSqlDatabase('sqlite:///:memory:')
>>> self.connect()
>>> self.populate_from(dset)
>>> for tablename in dset1.dataset.keys():
>>>     print(tablename)
>>>     table = dset1.pandas_table(tablename)
>>>     print(table)
```

Example

```
>>> # xdoctest: +REQUIRES(module:sqlalchemy)
>>> from kwcoco.coco_sql_dataset import _benchmark_dset_readtime  # NOQA
>>> import kwcoco
>>> from kwcoco.coco_sql_dataset import *
>>> dset2 = dset = kwcoco.CocoDataset.demo()
>>> dset1 = self = CocoSqlDatabase('sqlite:///:memory:')
>>> self.connect()
>>> self.populate_from(dset)
>>> assert_dsets_allclose(dset1, dset2, tag1='sql', tag2='dct')
>>> ti_sql = _benchmark_dset_readtime(dset1, 'sql')
>>> ti_dct = _benchmark_dset_readtime(dset2, 'dct')
>>> print('ti_sql.rankings = {}'.format(ub.urepr(ti_sql.rankings, nl=2,␣
↪precision=6, align=':')))
>>> print('ti_dct.rankings = {}'.format(ub.urepr(ti_dct.rankings, nl=2,␣
↪precision=6, align=':')))
```

CommandLine

```
KWCOCO_WITH_POSTGRESQL=1 xdoctest -m /home/joncrall/code/kwcoco/kwcoco/coco_sql_
↪dataset.py CocoSqlDatabase.populate_from:1
```

Example

```
>>> # xdoctest: +REQUIRES(env:KWCOCO_WITH_POSTGRESQL)
>>> # xdoctest: +REQUIRES(module:sqlalchemy)
>>> # xdoctest: +REQUIRES(module:psycopg2)
>>> from kwcoco.coco_sql_dataset import *  # NOQA
>>> import kwcoco
>>> dset = dset2 = kwcoco.CocoDataset.demo()
>>> self = dset1 = CocoSqlDatabase('postgresql+psycopg2://kwcoco:kwcoco_
↪pw@localhost:5432/test_populate')
>>> self.delete(verbose=1)
>>> self.connect(verbose=1)
>>> #self.populate_from(dset)
```

property dataset

property anns

property cats

property imgs

property name_to_cat

pandas_table(*table_name*, *strict=False*)

Loads an entire SQL table as a pandas DataFrame

Parameters

table_name (*str*) – name of the table

Returns
pandas.DataFrame

Example

```
>>> # xdoctest: +REQUIRES(module:sqlalchemy)
>>> # xdoctest: +REQUIRES(module:pandas)
>>> from kwcoco.coco_sql_dataset import *  # NOQA
>>> self, dset = demo()
>>> table_df = self.pandas_table('annotations')
>>> print(table_df)
>>> table_df = self.pandas_table('categories')
>>> print(table_df)
>>> table_df = self.pandas_table('videos')
>>> print(table_df)
>>> table_df = self.pandas_table('images')
>>> print(table_df)
>>> table_df = self.pandas_table('tracks')
>>> print(table_df)
```

raw_table(*table_name*)

_raw_tables()

Example

```
>>> # xdoctest: +REQUIRES(module:sqlalchemy)
>>> from kwcoco.coco_sql_dataset import *  # NOQA
>>> import pandas as pd
>>> self, dset = demo()
>>> targets = self._raw_tables()
>>> for tblname, table in targets.items():
...     print(f'tblname={tblname}')
...     print(pd.DataFrame(table))
```

_column_lookup(*tablename*, *key*, *rowids*, *default=NoParam*, *keepid=False*)

Convinience method to lookup only a single column of information

Example

```
>>> # xdoctest: +REQUIRES(module:sqlalchemy)
>>> from kwcoco.coco_sql_dataset import *  # NOQA
>>> self, dset = demo(10)
>>> tablename = 'annotations'
>>> key = 'category_id'
>>> rowids = list(self.anns.keys())[::3]
>>> cids1 = self._column_lookup(tablename, key, rowids)
>>> cids2 = self.annots(rowids).get(key)
>>> cids3 = dset.annots(rowids).get(key)
>>> assert cids3 == cids2 == cids1
```

(continues on next page)

(continued from previous page)

```
>>> # Test json columns work
>>> vals1 = self._column_lookup(tablename, 'bbox', rowids)
>>> vals2 = self.annots(rowids).lookup('bbox')
>>> vals3 = dset.annots(rowids).lookup('bbox')
>>> assert vals1 == vals2 == vals3
>>> vals1 = self._column_lookup(tablename, 'segmentation', rowids)
>>> vals2 = self.annots(rowids).lookup('segmentation')
>>> vals3 = dset.annots(rowids).lookup('segmentation')
>>> assert vals1 == vals2 == vals3
```

_all_rows_column_lookup(*tablename*, *keys*)

Convinience method to look up all rows from a table and only a few columns.

Example

```
>>> # xdoctest: +REQUIRES(module:sqlalchemy)
>>> from kwcoco.coco_sql_dataset import *  # NOQA
>>> self, dset = demo(10)
>>> tablename = 'annotations'
>>> keys = ['id', 'category_id']
>>> rows = self._all_rows_column_lookup(tablename, keys)
```

tabular_targets()

Convinience method to create an in-memory summary of basic annotation properties with minimal SQL overhead.

Example

```
>>> # xdoctest: +REQUIRES(module:sqlalchemy)
>>> from kwcoco.coco_sql_dataset import *  # NOQA
>>> self, dset = demo()
>>> targets = self.tabular_targets()
>>> print(targets.pandas())
```

_table_names()

property bundle_dpath

property data_fpath

data_fpath is an alias of fpath

_orig_coco_fpath()

Hack to reconstruct the original name. Makes assumptions about how naming is handled elsewhere. There should be centralized logic about how to construct side-car names that can be queried for inversed like this.

Returns
ub.Path | None

_abc_impl = <_abc._abc_data object>

_cached_hashid()

Compatibility with the way the exiting cached hashid in the coco dataset is used. Both of these functions are private and subject to change (and need optimization).

kwcoco.coco_sql_dataset.**cached_sql_coco_view**(*dct_db_fpath=None*, *sql_db_fpath=None*, *dset=None*, *force_rewrite=False*, *backend=None*)

Attempts to load a cached SQL-View dataset, only loading and converting the json dataset if necessary.

kwcoco.coco_sql_dataset.**ensure_sql_coco_view**(*dset*, *db_fpath=None*, *force_rewrite=False*, *backend=None*)

Create a cached on-disk SQL view of an on-disk COCO dataset.

DEPREICATE, use cache function instead

Note: This function is fragile. It depends on looking at file modified timestamps to determine if it needs to write the dataset.

---

kwcoco.coco_sql_dataset.**demo**(*num=10*, *backend=None*)

kwcoco.coco_sql_dataset.**assert_dsets_allclose**(*dset1*, *dset2*, *tag1='dset1'*, *tag2='dset2'*)

kwcoco.coco_sql_dataset.**_benchmark_dset_readtime**(*dset*, *tag='?'*, *n=4*, *post_iterate=False*)

Helper for understanding the time differences between backends

---

Note: post_iterate ensures that all of the returned data is looked at by the python interpreter. Makes this a more fair comparison because python can just return pointers to the data, but only in the case where most of the data will touched. For one attribute lookups it is not a good test.

---

kwcoco.coco_sql_dataset.**_benchmark_dict_proxy_ops**(*proxy*)

Get insight on the efficiency of operations

kwcoco.coco_sql_dataset.**devcheck**()

Scratch work for things that should eventually become unit or doc tests

from kwcoco.coco_sql_dataset import * # NOQA self, dset = demo()

2.1.2.12 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* | *None*) – 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 | PathLike | None*) – COCO results directory name
- **imgIds** (*list*) – images to be downloaded

`loadNumpyAnnotations`(*data*)

Convert result data from a numpy array [Nx7] where each row contains {imageID,x1,y1,w,h,score,class}

Parameters
data (*numpy.ndarray*)

Returns
annotations (python nested list)

Return type
List[Dict]

`annToRLE`(*ann*)

Convert annotation which can be polygons, uncompressed RLE to RLE.

Returns
kwimage.Mask

Note:

- This requires the C-extensions for kwimage to be installed (i.e.

`pip install kwimage_ext`) due to the need to interface with the bytes RLE format.

Example

```
>>> from kwcoco.compat_dataset import *  # NOQA
>>> import kwcoco
>>> self = COCO(kwcoco.CocoDataset.demo('shapes8').dataset)
>>> try:
>>>     rle = self.annToRLE(self.anns[1])
>>> except NotImplementedError:
>>>     import pytest
>>>     pytest.skip('missing kwimage c-extensions')
>>> else:
>>>     assert len(rle['counts']) > 2
>>> # xdoctest: +REQUIRES(module:pycocotools)
>>> self.conform(legacy=True)
>>> orig = self._aspycoco().annToRLE(self.anns[1])
```

annToMask(*ann*)
Convert annotation which can be polygons, uncompressed RLE, or RLE to binary mask.

Returns
binary mask (numpy 2D array)

Return type
ndarray

Note: The mask is returned as a fortran (F-style) array with the same dimensions as the parent image.

_abc_impl = <_abc._abc_data object>

2.1.2.13 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

2.1.2.14 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**()

2.1.2.15 kwcoco.kw18 module

A helper for converting COCO to / from KW18 format.

KW18 File Format https://docs.google.com/spreadsheets/d/1DFCwoTKnDv8qfy3raM7QXtir2Fjfj9j8-z8px5Bu0q8/edit#gid=10

The kw18.trk files are text files, space delimited; each row is one frame of one track and all rows have the same number of columns. The fields are:

```
01) track_ID         : identifies the track
02) num_frames:     number of frames in the track
03) frame_id        : frame number for this track sample
04) loc_x        : X-coordinate of the track (image/ground coords)
05) loc_y        : Y-coordinate of the track (image/ground coords)
06) vel_x        : X-velocity of the object (image/ground coords)
07) vel_y        : Y-velocity of the object (image/ground coords)
08) obj_loc_x        : X-coordinate of the object (image coords)
09) obj_loc_y        : Y-coordinate of the object (image coords)
10) bbox_min_x    : minimum X-coordinate of bounding box (image coords)
11) bbox_min_y    : minimum Y-coordinate of bounding box (image coords)
12) bbox_max_x    : maximum X-coordinate of bounding box (image coords)
13) bbox_max_y    : maximum Y-coordinate of bounding box (image coords)
14) area        : area of object (pixels)
15) world_loc_x    : X-coordinate of object in world
16) world_loc_y    : Y-coordinate of object in world
17) world_loc_z    : Z-coordiante of object in world
18) timestamp        : timestamp of frame (frames)
For the location and velocity of object centroids, use fields 4-7.
Bounding box is specified using coordinates of the top-left and bottom
right corners. Fields 15-17 may be ignored.

The kw19.trk and kw20.trk files, when present, add the following field(s):
19) object class: estimated class of the object, either 1 (person), 2
(vehicle), or 3 (other).
20) Activity ID -- refer to activities.txt for index and list of activities.
```

class kwcoco.kw18.**KW18**(*data*)

Bases: DataFrameArray

A DataFrame like object that stores KW18 column data

Example

```
>>> import kwcoco
>>> from kwcoco.kw18 import KW18
>>> coco_dset = kwcoco.CocoDataset.demo('shapes')
>>> kw18_dset = KW18.from_coco(coco_dset)
>>> print(kw18_dset.pandas())
```

Parameters

data – the kw18 data frame.

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] | None*) – if specified, maps frame numbers to image file paths.
- **video_name** (*str | 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
>>> import ubelt as ub
>>> import kwimage
>>> import kwcoco
>>> # Prep test data - autogen a demo kw18 and write it to disk
>>> dpath = ub.Path.appdir('kwcoco/kw18').ensuredir()
>>> kw18_fpath = ub.Path(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= kwcoco.CocoDataset.demo('shapes8').images().lookup('file_name')
>>> 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 = 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)
```

kwcoco.kw18.**_ensure_kw18_column_order**(*df*)

Ensure expected kw18 columns exist and are in the correct order.

Example

```
>>> import pandas as pd
>>> df = pd.DataFrame(columns=KW18.DEFAULT_COLUMNS[0:18])
>>> _ensure_kw18_column_order(df)
>>> df = pd.DataFrame(columns=KW18.DEFAULT_COLUMNS[0:19])
>>> _ensure_kw18_column_order(df)
>>> df = pd.DataFrame(columns=KW18.DEFAULT_COLUMNS[0:18] + KW18.DEFAULT_
↪COLUMNS[20:21])
>>> assert np.all(_ensure_kw18_column_order(df).columns == df.columns)
```

2.1.2.16 kwcoco.sensorchan_spec module

This functionality has been moved to "delayed_image"

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.urepr(self.basic_stats(), nl=1))
>>> print(ub.urepr(self.extended_stats(), nl=2))
>>> print(ub.urepr(self.boxsize_stats(), nl=3))
>>> print(ub.urepr(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'}
```

(continues on next page)

(continued from previous page)

```
>>> 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.urepr(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.urepr(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.urepr(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.

_abc_impl = <_abc._abc_data object>

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
    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)
```

(continues on next page)

(continued from previous page)

```
>>> 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)
```

Parameters

- **graph** (*nx.DiGraph*) – either the graph representing a category hierarchy
- **checks** (*bool, default=True*) – if false, bypass input checks

`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

Parameters
node (*str*) – the name of the category

Returns
int

take(*indexes*)

Create a subgraph based on the selected class indexes

subgraph(*subnodes*, *closure=True*)

Create a subgraph based on the selected class nodes (i.e. names)

Example

```
>>> self = CategoryTree.from_coco([
>>>     {'id': 130, 'name': 'n3', 'supercategory': 'n1'},
>>>     {'id': 410, 'name': 'n1', 'supercategory': None},
>>>     {'id': 640, 'name': 'n4', 'supercategory': 'n3'},
>>>     {'id': 220, 'name': 'n2', 'supercategory': 'n1'},
>>>     {'id': 560, 'name': 'n6', 'supercategory': 'n2'},
>>>     {'id': 350, 'name': 'n5', 'supercategory': 'n2'},
>>> ])
>>> self.print_graph()
>>> subnodes = ['n3', 'n6', 'n4', 'n1']
>>> new1 = self.subgraph(subnodes, closure=1)
>>> new1.print_graph()
...
>>> print('new1.idx_to_id = {}'.format(ub.urepr(new1.idx_to_id, nl=0)))
>>> print('new1.idx_to_node = {}'.format(ub.urepr(new1.idx_to_node, nl=0)))
new1.idx_to_id = [130, 560, 640, 410]
new1.idx_to_node = ['n3', 'n6', 'n4', 'n1']
```

```
>>> indexes = [2, 1, 0, 5]
>>> new2 = self.take(indexes)
>>> new2.print_graph()
...
>>> print('new2.idx_to_id = {}'.format(ub.urepr(new2.idx_to_id, nl=0)))
>>> print('new2.idx_to_node = {}'.format(ub.urepr(new2.idx_to_node, nl=0)))
new2.idx_to_id = [640, 410, 130, 350]
new2.idx_to_node = ['n4', 'n1', 'n3', 'n5']
```

```
>>> subnodes = ['n3', 'n6', 'n4', 'n1']
>>> new3 = self.subgraph(subnodes, closure=0)
>>> new3.print_graph()
```

_build_index()

construct lookup tables

show()

forest_str()

print_graph()

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 delayed_image.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.urepr(self.info, nl=1)))
>>> self = ChannelSpec('rgb')
>>> print('self.info = {}'.format(ub.urepr(self.info, nl=1)))
>>> self = ChannelSpec('rgb|disparity')
>>> print('self.info = {}'.format(ub.urepr(self.info, nl=1)))
>>> self = ChannelSpec('rgb|disparity,disparity')
>>> print('self.info = {}'.format(ub.urepr(self.info, nl=1)))
>>> self = ChannelSpec('rgb,disparity,flowx|flowy')
>>> print('self.info = {}'.format(ub.urepr(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))
>>>     #
```

(continues on next page)

(continued from previous page)

```
>>>     self = ChannelSpec.coerce(spec)
>>>     print('self = {!r}'.format(self))
>>>     sizes = self.sizes()
>>>     print('sizes = {!r}'.format(sizes))
>>>     print('self.info = {}'.format(ub.urepr(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.urepr(item, precision=1)))
>>>     print('inputs = {}'.format(ub.urepr(inputs, precision=1)))
>>>     print('input_shapes = {}'.format(ub.urepr(input_shapes)))
>>>     print('components = {}'.format(ub.urepr(components, precision=1)))
>>>     print('component_shapes = {}'.format(ub.urepr(component_shapes, nl=1)))
```

property spec

property info

classmethod coerce(*data*)

Attempt to interpret the data as a channel specification

Returns
ChannelSpec

Example

```
>>> from delayed_image.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 delayed_image.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 delayed_image.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

Example

```
>>> from delayed_image.channel_spec import *
>>> self = ChannelSpec('rgb|disparity,flowx|r|flowy')
>>> self.as_path()
rgb_disparity,flowx_r_flowy
```

difference(*other*)

Set difference. Remove all instances of other channels from this set of channels.

Example

```
>>> from delayed_image.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 delayed_image.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 delayed_image.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 delayed_image.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

_item_shapes(*dims*)

Expected shape for an input item

Parameters
dims (*Tuple[int, int]*) – the spatial dimension

Returns
Dict[int, tuple]

_demo_item(*dims=(4, 4)*, *rng=None*)

Create an input that satisfies this spec

Returns

an item like it might appear when its returned from the
__getitem__ method of a `torch...Dataset`.

Return type
dict

Example

```
>>> dims = (1, 1)
>>> ChannelSpec.coerce(3)._demo_item(dims, rng=0)
>>> ChannelSpec.coerce('r|g|b|disaprity')._demo_item(dims, rng=0)
>>> ChannelSpec.coerce('rgb|disaprity')._demo_item(dims, rng=0)
>>> ChannelSpec.coerce('rgb,disaprity')._demo_item(dims, rng=0)
>>> ChannelSpec.coerce('rgb')._demo_item(dims, rng=0)
>>> ChannelSpec.coerce('gray')._demo_item(dims, rng=0)
```

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 delayed_image.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.urepr(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.urepr(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.urepr(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 delayed_image.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.urepr(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*, *MixinCocoConstructors*, *MixinCocoExtras*, *MixinCocoHashing*, *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 | None*) – 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
```

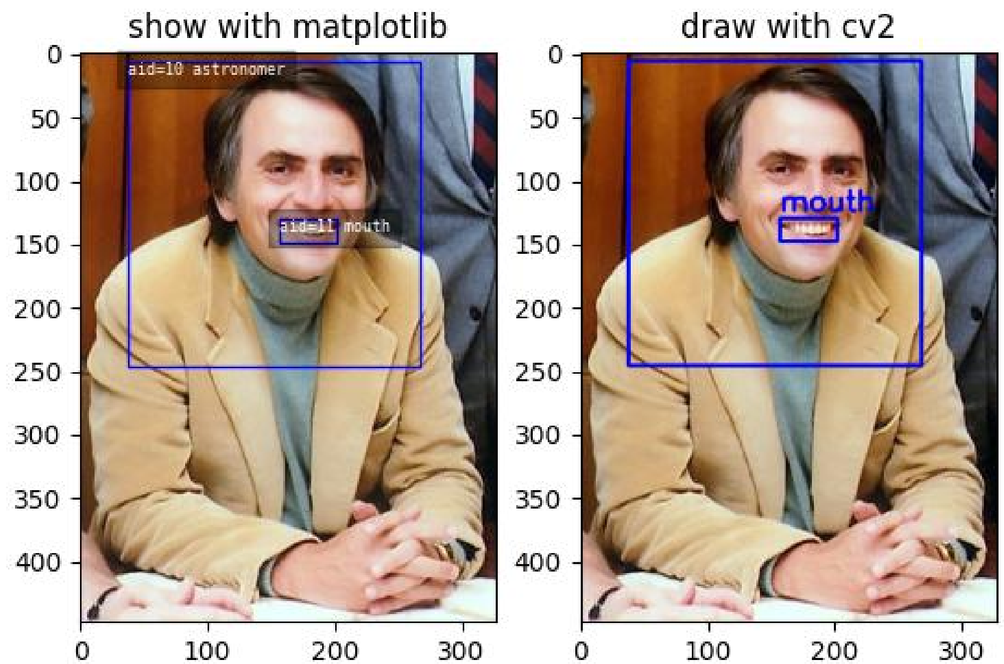
Example

```
>>> 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.urepr(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.urepr((ub.udict(ann) - {'segmentation'}).sorted_keys(), 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.urepr(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.urepr(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()
```



Parameters

- **data** (*str* | *PathLike* | *dict* | *None*) – Either a filepath to a coco json file, or a dictionary containing the actual coco json structure. For a more generally coercable constructor see func:*CocoDataset.coerce*.
- **tag** (*str* | *None*) – Name of the dataset for display purposes, and does not influence behavior of the underlying data structure, although it may be used via convinience methods. We attempt to autopopulate this via information in `data` if available. If unspecfied and `data` is a filepath this becomes the basename.
- **bundle_dpath** (*str* | *None*) – the root of the dataset that images / external data will be assumed to be relative to. If unspecfied, we attempt to determine it using information in `data`. If `data` is a filepath, we use the dirname of that path. If `data` is a dictionary, we look for the "img_root" key. If unspecfied and we fail to introspect then, we fallback to the current working directory.
- **img_root** (*str* | *None*) – deprecated alias for bundle_dpath

property fpath

In the future we will deprecate img_root for bundle_dpath

_update_fpath(*new_fpath*)

_infer_dirs()

classmethod from_data(*data*, *bundle_dpath=None*, *img_root=None*)

Constructor from a json dictionary

Return type
CocoDataset

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

Return type
CocoDataset

Example

```
>>> import kwcoco
>>> coco_dset = kwcoco.CocoDataset.from_image_paths(['a.png', 'b.png'])
>>> assert coco_dset.n_images == 2
```

classmethod from_class_image_paths(*root*)

Ingest classification data in the common format where images of different categories are stored in folders with the category label.

Parameters
root (*str | PathLike*) – the path to a directory containing class-subdirectories

Return type
CocoDataset

classmethod coerce_multiple(*datas*, *workers=0*, *mode='process'*, *verbose=1*, *postprocess=None*, *ordered=True*, ***kwargs*)

Coerce multiple CocoDataset objects in parallel.

Parameters

- **datas** (*List*) – list of kwcoco coercables to load
- **workers** (*int | str*) – number of worker threads / processes. Can also accept coerceable workers.
- **mode** (*str*) – thread, process, or serial. Defaults to process.
- **verbose** (*int*) – verbosity level
- **postprocess** (*Callable | None*) – A function taking one arg (the loaded dataset) to run on the loaded kwcoco dataset in background workers. This can be more efficient when postprocessing is independent per kwcoco file.
- **ordered** (*bool*) – if True yields datasets in the same order as given. Otherwise results are yielded as they become available. Defaults to True.
- ****kwargs** – arguments passed to the constructor

Yields
CocoDataset

SeeAlso:

- load_multiple - like this function but is a strict file-path-only loader

CommandLine

```
xdoctest -m kwcoco.coco_dataset CocoDataset.coerce_multiple
```

Example

```
>>> import kwcoco
>>> dset1 = kwcoco.CocoDataset.demo('shapes1')
>>> dset2 = kwcoco.CocoDataset.demo('shapes2')
>>> dset3 = kwcoco.CocoDataset.demo('vidshapes8')
>>> dsets = [dset1, dset2, dset3]
>>> input_fpaths = [d.fpath for d in dsets]
>>> results = list(kwcoco.CocoDataset.coerce_multiple(input_fpaths,␣
↪ordered=True))
>>> result_fpaths = [r.fpath for r in results]
>>> assert result_fpaths == input_fpaths
>>> # Test unordered
>>> results1 = list(kwcoco.CocoDataset.coerce_multiple(input_fpaths,␣
↪ordered=False))
>>> result_fpaths = [r.fpath for r in results]
>>> assert set(result_fpaths) == set(input_fpaths)
>>> #
>>> # Coerce from existing datasets
>>> results2 = list(kwcoco.CocoDataset.coerce_multiple(dsets, ordered=True,␣
↪workers=0))
>>> assert results2[0] is dsets[0]
```

classmethod load_multiple(*fpaths*, *workers=0*, *mode='process'*, *verbose=1*, *postprocess=None*, *ordered=True*, ***kwargs*)

Load multiple CocoDataset objects in parallel.

Parameters

- **fpaths** (*List[str | PathLike]*) – list of paths to multiple coco files to be loaded
- **workers** (*int*) – number of worker threads / processes
- **mode** (*str*) – thread, process, or serial. Defaults to process.
- **verbose** (*int*) – verbosity level
- **postprocess** (*Callable | None*) – A function taking one arg (the loaded dataset) to run on the loaded kwcoco dataset in background workers and returns the modified dataset. This can be more efficient when postprocessing is independent per kwcoco file.
- **ordered** (*bool*) – if True yields datasets in the same order as given. Otherwise results are yielded as they become available. Defaults to True.
- ****kwargs** – arguments passed to the constructor

Yields

CocoDataset

SeeAlso:

- **coerce_multiple - like this function but accepts general**
  coercable inputs.

classmethod _load_multiple(*_loader*, *inputs*, *workers=0*, *mode='process'*, *verbose=1*, *postprocess=None*, *ordered=True*, ***kwargs*)

Shared logic for multiprocessing loaders.

SeeAlso:

- coerce_multiple
- load_multiple

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*) – number of worker threads / processes
- **verbose** (*int*) – verbosity level
- **mode** (*str*) – thread, process, or serial
- **union** (*str | bool*) – 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. Default='try'

Note: This may be deprecated. Use load_multiple or coerce_multiple and then manually perform the union.

copy()

Deep copies this object

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> new = self.copy()
>>> assert new.imgs[1] is new.dataset['images'][0]
>>> assert new.imgs[1] == self.dataset['images'][0]
>>> assert new.imgs[1] is not self.dataset['images'][0]
```

dumps(*indent=None*, *newlines=False*)

Writes the dataset out to the json format

Parameters

- **newlines** (*bool*) – if True, each annotation, image, category gets its own line
- **indent** (*int | str | None*) – indentation for the json file. See `json.dump()` for details.
- **newlines** (*bool*) – if True, each annotation, image, category gets its own line.

Note:

Using newlines=True is similar to:
print(ub.urepr(dset.dataset, nl=2, trailsep=False)) However, the above may not output valid json if it contains ndarrays.

Example

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

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

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.coerce('vidshapes1-msi-multisensor', verbose=3)
>>> self.remove_annotations(self.annots())
>>> text = self.dumps(newlines=0, indent='  ')
>>> print(text)
>>> text = self.dumps(newlines=True, indent='  ')
>>> print(text)
```

_compress_dump_to_fileptr(*file*, *arcname=None*, *indent=None*, *newlines=False*)

Experimental method to save compressed kwcoco files, may be folded into dump in the future.

_dump(*file*, *indent*, *newlines*, *compress*)

Case where we are dumping to an open file pointer. We assume this means the dataset has been written to disk.

dump(*file=None*, *indent=None*, *newlines=False*, *temp_file='auto'*, *compress='auto'*)

Writes the dataset out to the json format

Parameters

- **file** (*PathLike | IO | None*) – Where to write the data. Can either be a path to a file or an open file pointer / stream. If unspecified, it will be written to the current `fpath` property.
- **indent** (*int | str | None*) – indentation for the json file. See `json.dump()` for details.
- **newlines** (*bool*) – if True, each annotation, image, category gets its own line.

- **temp_file** (*bool | str*) – Argument to `safer.open()`. Ignored if `file` is not a PathLike object. Defaults to 'auto', which is False on Windows and True everywhere else.
- **compress** (*bool | str*) – if True, dumps the kwcoco file as a compressed zipfile. In this case a literal IO file object must be opened in binary write mode. If auto, then it will default to False unless it can introspect the file name and the name ends with .zip

Example

```
>>> import kwcoco
>>> import ubelt as ub
>>> dpath = ub.Path.appdir('kwcoco/demo/dump').ensuredir()
>>> dset = kwcoco.CocoDataset.demo()
>>> dset.fpath = dpath / 'my_coco_file.json'
>>> # Calling dump writes to the current fpath attribute.
>>> dset.dump()
>>> assert dset.dataset == kwcoco.CocoDataset(dset.fpath).dataset
>>> assert dset.dumps() == dset.fpath.read_text()
>>> #
>>> # Using compress=True can save a lot of space and it
>>> # is transparent when reading files via CocoDataset
>>> dset.dump(compress=True)
>>> assert dset.dataset == kwcoco.CocoDataset(dset.fpath).dataset
>>> assert dset.dumps() != dset.fpath.read_text(errors='replace')
```

Example

```
>>> import kwcoco
>>> import ubelt as ub
>>> # Compression auto-defaults based on the file name.
>>> dpath = ub.Path.appdir('kwcoco/demo/dump').ensuredir()
>>> dset = kwcoco.CocoDataset.demo()
>>> fpath1 = dset.fpath = dpath / 'my_coco_file.zip'
>>> dset.dump()
>>> fpath2 = dset.fpath = dpath / 'my_coco_file.json'
>>> dset.dump()
>>> assert fpath1.read_bytes()[0:8] != fpath2.read_bytes()[0:8]
```

_check_json_serializable(*verbose=1*)

Debug which part of a coco dataset might not be json serializable

_check_integrity()

perform most checks

_check_index()

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> self._check_index()
>>> # Force a failure
>>> self.index.anns.pop(1)
>>> self.index.anns.pop(2)
>>> import pytest
>>> with pytest.raises(AssertionError):
>>>     self._check_index()
```

_abc_impl = <_abc._abc_data object>

_check_pointers(*verbose=1*)

Check that all category and image ids referenced by annotations exist

_build_index()

union(*, *disjoint_tracks=True*, *remember_parent=False*, ***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*) – if True, we will assume track-names are disjoint and if two datasets share the same track-name, we will disambiguate them. Otherwise they will be copied over as-is. Defaults to True. In most cases you do not want to set this to False.
- **remember_parent** (*bool*) – if True, videos and images will save information about their parent in the "union_parent" field.
- ****kwargs** – constructor options for the new merged CocoDataset

Returns

a new merged coco dataset

Return type

kwcoco.CocoDataset

CommandLine

```
xdoctest -m kwcoco.coco_dataset CocoDataset.union
```

Example

```
>>> import kwcoco
>>> # Test union works with different keypoint categories
>>> dset1 = kwcoco.CocoDataset.demo('shapes1')
>>> dset2 = kwcoco.CocoDataset.demo('shapes2')
>>> dset1.remove_keypoint_categories(['bot_tip', 'mid_tip', 'right_eye'])
>>> dset2.remove_keypoint_categories(['top_tip', 'left_eye'])
>>> dset_12a = kwcoco.CocoDataset.union(dset1, dset2)
>>> dset_12b = dset1.union(dset2)
>>> dset_21 = dset2.union(dset1)
>>> def add_hist(h1, h2):
>>>     return {k: h1.get(k, 0) + h2.get(k, 0) for k in set(h1) | set(h2)}
>>> kpfreq1 = dset1.keypoint_annotation_frequency()
>>> kpfreq2 = dset2.keypoint_annotation_frequency()
>>> kpfreq_want = add_hist(kpfreq1, kpfreq2)
>>> kpfreq_got1 = dset_12a.keypoint_annotation_frequency()
>>> kpfreq_got2 = dset_12b.keypoint_annotation_frequency()
>>> assert kpfreq_want == kpfreq_got1
>>> assert kpfreq_want == kpfreq_got2
```

```
>>> # Test disjoint gid datasets
>>> dset1 = kwcoco.CocoDataset.demo('shapes3')
>>> for new_gid, img in enumerate(dset1.dataset['images'], start=10):
>>>     for aid in dset1.gid_to_aids[img['id']]:
>>>         dset1.anns[aid]['image_id'] = new_gid
>>>     img['id'] = new_gid
>>> dset1.index.clear()
>>> dset1._build_index()
>>> # ------
>>> dset2 = kwcoco.CocoDataset.demo('shapes2')
>>> for new_gid, img in enumerate(dset2.dataset['images'], start=100):
>>>     for aid in dset2.gid_to_aids[img['id']]:
>>>         dset2.anns[aid]['image_id'] = new_gid
>>>     img['id'] = new_gid
>>> dset1.index.clear()
>>> dset2._build_index()
>>> others = [dset1, dset2]
>>> merged = kwcoco.CocoDataset.union(*others)
>>> print('merged = {!r}'.format(merged))
>>> print('merged.imgs = {}'.format(ub.urepr(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.urepr(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.urepr(dset1.anns, nl=1)))
>>> print('dset2.anns = {}'.format(ub.urepr(dset2.anns, nl=1)))
>>> print('dset3.anns = {}'.format(ub.urepr(dset3.anns, nl=1)))
>>> print('merged.anns = {}'.format(ub.urepr(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*) – if True, makes a deep copy of all nested attributes, otherwise makes a shallow copy. Defaults to True.
- **autobuild** (*bool*) – if True will automatically build the fast lookup index. Defaults to True.

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> gids = [1, 3]
>>> sub_dset = self.subset(gids)
>>> assert len(self.index.gid_to_aids) == 3
>>> assert len(sub_dset.gid_to_aids) == 2
```

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo('vidshapes2')
>>> gids = [1, 2]
>>> sub_dset = self.subset(gids, copy=True)
>>> assert len(sub_dset.index.videos) == 1
>>> assert len(self.index.videos) == 2
```

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> sub1 = self.subset([1])
>>> sub2 = self.subset([2])
>>> sub3 = self.subset([3])
>>> others = [sub1, sub2, sub3]
>>> rejoined = kwcoco.CocoDataset.union(*others)
>>> assert len(sub1.anns) == 9
>>> assert len(sub2.anns) == 2
>>> assert len(sub3.anns) == 0
>>> assert rejoined.basic_stats() == self.basic_stats()
```

`view_sql`(*force_rewrite=False*, *memory=False*, *backend='sqlite'*, *sql_db_fpath=None*)

Create a cached SQL interface to this dataset suitable for large scale multiprocessing use cases.

Parameters

- **force_rewrite** (*bool*) – if True, forces an update to any existing cache file on disk
- **memory** (*bool*) – if True, the database is constructed in memory.
- **backend** (*str*) – sqlite or postgresql
- **sql_db_fpath** (*str* | *PathLike* | *None*) – overrides the database uri

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.

CommandLine

```
KWCOCO_WITH_POSTGRESQL=1 xdoctest -m /home/joncrall/code/kwcoco/kwcoco/coco_
↪dataset.py CocoDataset.view_sql
```

Example

```
>>> # xdoctest: +REQUIRES(module:sqlalchemy)
>>> # xdoctest: +REQUIRES(env:KWCOCO_WITH_POSTGRESQL)
>>> # xdoctest: +REQUIRES(module:psycopg2)
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('vidshapes32')
>>> postgres_dset = dset.view_sql(backend='postgresql', force_rewrite=True)
>>> sqlite_dset = dset.view_sql(backend='sqlite', force_rewrite=True)
>>> list(dset.anns.keys())
>>> list(postgres_dset.anns.keys())
>>> list(sqlite_dset.anns.keys())
```

class kwcoco.**CocoImage**(*img*, *dset=None*)

Bases: *_CocoObject*

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 = kwcoco.CocoImage(dset1.imgs[1], dset1)
>>> print('self = {!r}'.format(self))
>>> print('self.channels = {}'.format(ub.urepr(self.channels, nl=1)))
```

```
>>> self = kwcoco.CocoImage(dset2.imgs[1], dset2)
>>> print('self.channels = {}'.format(ub.urepr(self.channels, nl=1)))
>>> self.primary_asset()
>>> assert 'auxiliary' in self
```

classmethod from_gid(*dset*, *gid*)

property video

Helper to grab the video for this image if it exists

property name

`detach()`

Removes references to the underlying coco dataset, but keeps special information such that it wont be needed.

`property assets`

CocoImage.iter_assets.

Type
Convinience wrapper around

Type
func

`property datetime`

Try to get datetime information for this image. Not always possible.

Returns
datetime.datetime | None

`annots()`

Returns
a 1d annotations object referencing annotations in this image

Return type
Annots

`stats()`

`get(key, default=NoParam)`

`keys()`

Proxy getter attribute for underlying *self.img* dictionary

`property channels`

`property n_assets`

The number of on-disk files associated with this coco image

`property num_channels`

`property dsize`

`primary_image_filepath(requires=None)`

`primary_asset(requires=None, as_dict=True)`

Compute a “main” image asset.

Note: 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] | None*) – list of attribute that must be non-None to consider an object as the primary one.
- **as_dict** (*bool*) – if True the return type is a raw dictionary. Otherwise use a newer object-oriented wrapper that should be duck-type swappable. In the future this default will change to False.

Returns
the asset dict or None if it is not found

Return type
None | dict

Todo:

- [ ] Add in primary heuristics

Example

```
>>> import kwarray
>>> from kwcoco.coco_image import *  # NOQA
>>> rng = kwarray.ensure_rng(0)
>>> def random_asset(name, w=None, h=None):
>>>     return {'file_name': name, 'width': w, 'height': h}
>>> self = CocoImage({
>>>     'auxiliary': [
>>>         random_asset('1'),
>>>         random_asset('2'),
>>>         random_asset('3'),
>>>     ]
>>> })
>>> assert self.primary_asset()['file_name'] == '1'
>>> self = CocoImage({
>>>     'auxiliary': [
>>>         random_asset('1'),
>>>         random_asset('2', 3, 3),
>>>         random_asset('3'),
>>>     ]
>>> })
>>> assert self.primary_asset()['file_name'] == '2'
>>> #
>>> # Test new objct oriented output
>>> self = CocoImage({
>>>     'file_name': 'foo',
>>>     'assets': [
>>>         random_asset('1'),
>>>         random_asset('2'),
>>>         random_asset('3'),
>>>     ],
>>> })
>>> assert self.primary_asset(as_dict=False) is self
>>> self = CocoImage({
```

(continues on next page)

(continued from previous page)

```
>>>     'assets': [
>>>         random_asset('1'),
>>>         random_asset('3'),
>>>     ],
>>>     'auxiliary': [
>>>         random_asset('1'),
>>>         random_asset('2', 3, 3),
>>>         random_asset('3'),
>>>     ]
>>> })
>>> assert self.primary_asset(as_dict=False)['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.

Yields
ub.Path

iter_assets()

Iterate through assets (which could include the image itself it points to a file path).

Object-oriented alternative to *CocoImage.iter_asset_objs()*

Yields
CocoImage | *CocoAsset* – an asset object (or image object if it points to a file)

Example

```
>>> import kwcoco
>>> coco_img = kwcoco.CocoImage({'width': 128, 'height': 128})
>>> assert len(list(coco_img.iter_assets())) == 0
>>> dset = kwcoco.CocoDataset.demo('vidshapes8-multispectral')
>>> self = dset.coco_image(1)
>>> assert len(list(self.iter_assets())) > 1
>>> dset = kwcoco.CocoDataset.demo('vidshapes8')
>>> self = dset.coco_image(1)
>>> assert list(self.iter_assets()) == [self]
```

iter_asset_objs()

Iterate through base + auxiliary dicts that have file paths

Note: In most cases prefer *iter_assets()* instead.

Yields
dict – an image or auxiliary dictionary

find_asset(*channels*)

Find the asset dictionary with the specified channels

Parameters

channels (*str* | *FusedChannelSpec*) – channel names the asset must have.

Returns

CocoImage | CocoAsset

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoImage({
>>>     'file_name': 'raw',
>>>     'channels': 'red|green|blue',
>>>     'assets': [
>>>         {'file_name': '1', 'channels': 'spam'},
>>>         {'file_name': '2', 'channels': 'eggs|jam'},
>>>     ],
>>>     'auxiliary': [
>>>         {'file_name': '3', 'channels': 'foo'},
>>>         {'file_name': '4', 'channels': 'bar|baz'},
>>>     ]
>>> })
>>> assert self.find_asset('blah') is None
>>> assert self.find_asset('red|green|blue') is self
>>> self.find_asset('foo')['file_name'] == '3'
>>> self.find_asset('baz')['file_name'] == '4'
```

find_asset_obj(*channels*)

Find the asset dictionary with the specified channels

In most cases use `CocoImge.find_asset()` instead.

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
```

_assets_key()

Internal helper for transition from auxiliary -> assets in the image spec

add_annotation(***ann*)

Adds an annotation to this image.

This is a convinience method, and requires that this CocoImage is still connected to a parent dataset.

Parameters
****ann** – annotation attributes (e.g. bbox, category_id)

Returns
the new annotation id

Return type
int

SeeAlso:
`kwcoco.CocoDataset.add_annotation()`

add_asset(*file_name=None*, *channels=None*, *imdata=None*, *warp_aux_to_img=None*, *width=None*, *height=None*, *imwrite=False*, *image_id=None*, ***kw*)

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 | PathLike | None*) – The name of the file relative to the bundle directory. If unspecified, imdata must be given.
- **channels** (*str | kwcoco.FusedChannelSpec | None*) – 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 | None*) – 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 | None*) – Width of the data in auxiliary space (inferred if unspecified)

- **height** (*int | None*) – 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.
- **image_id** (*int | None*) – An asset dictionary contains an image-id, but it should *not* be specified here. If it is, then it *must* agree with this image's id.
- ****kw** – stores arbitrary key/value pairs in this new asset.

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_asset(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_asset('path/img1_B0.tif', channels='B0', width=200, height=200)
>>> coco_img.add_asset('path/img1_B1.tif', channels='B1', width=200, height=200)
>>> coco_img.add_asset('path/img1_B2.tif', channels='B2', width=200, height=200)
>>> coco_img.add_asset('path/img1_TCI.tif', channels='r|g|b', width=200,␣
↪height=200)
```

imdelay(*channels=None*, *space='image'*, *resolution=None*, *bundle_dpath=None*, *interpolation='linear'*, *antialias=True*, *nodata_method=None*, *RESOLUTION_KEY=None*)

Perform a delayed load on the data in this image.

The delayed load can load a subset of channels, and perform lazy warping operations. If the underlying data is in a tiled format this can reduce the amount of disk IO needed to read the data if only a small crop or lower resolution view of the data is needed.

Note: This method is experimental and relies on the delayed load proof-of-concept.

Parameters

- **gid** (*int*) – image id to load

- **channels** (*kwcoco.FusedChannelSpec*) – specific channels to load. if unspecified, all channels are loaded.
- **space** (*str*) – can either be "image" for loading in image space, or "video" for loading in video space.
- **resolution** (*None | str | float*) – If specified, applies an additional scale factor to the result such that the data is loaded at this specified resolution. This requires that the image / video has a registered resolution attribute and that its units agree with this request.

Todo:

- **[ ] 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.imdelay()
>>> 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).imdelay()
>>> 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).imdelay(channels='r')
```

```
>>> import kwcoco
>>> gid = 1
>>> #
>>> dset = kwcoco.CocoDataset.demo('vidshapes8-multispectral')
>>> delayed = dset.coco_image(gid).imdelay(channels='B1|B2', space='image')
>>> print('delayed = {!r}'.format(delayed))
>>> print('delayed.finalize() = {!r}'.format(delayed.finalize()))
>>> delayed = dset.coco_image(gid).imdelay(channels='B1|B2|B11', space='image')
>>> print('delayed = {!r}'.format(delayed))
>>> print('delayed.finalize() = {!r}'.format(delayed.finalize()))
```

(continues on next page)

(continued from previous page)

```
>>> delayed = dset.coco_image(gid).imdelay(channels='B8|B1', space='video')
>>> print('delayed = {!r}'.format(delayed))
>>> print('delayed.finalize() = {!r}'.format(delayed.finalize()))
```

```
>>> delayed = dset.coco_image(gid).imdelay(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.imdelay()
>>> final = delayed.finalize()
>>> assert final.shape == (512, 512, 3)
```

```
>>> delayed = coco_img.imdelay()
>>> final = delayed.finalize()
>>> print('final.shape = {}'.format(ub.urepr(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.imdelay(channels='B1|Aux:2:4')
>>> 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]
>>> asset_delayed = coco_img.imdelay(stream1, space='asset')
>>> img_delayed = coco_img.imdelay(stream1, space='image')
>>> vid_delayed = coco_img.imdelay(stream1, space='video')
>>> #
>>> aux_imdata = asset_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
>>> from delayed_image.delayed_nodes import CoordinateCompatibilityError
>>> with pytest.raises(CoordinateCompatibilityError):
>>>     aux_delayed2 = coco_img.imdelay(fused_channels, space='asset')
```

Example

```
>>> # Test loading at a specific resolution.
>>> from kwcoco.coco_image import *  # NOQA
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('vidshapes8-msi-multisensor')
>>> coco_img = dset.coco_image(1)
>>> coco_img.img['resolution'] = '1 meter'
>>> img_delayed1 = coco_img.imdelay(space='image')
>>> vid_delayed1 = coco_img.imdelay(space='video')
>>> # test with unitless request
>>> img_delayed2 = coco_img.imdelay(space='image', resolution=3.1)
>>> vid_delayed2 = coco_img.imdelay(space='video', resolution='3.1 meter')
>>> np.ceil(img_delayed1.shape[0] / 3.1) == img_delayed2.shape[0]
>>> np.ceil(vid_delayed1.shape[0] / 3.1) == vid_delayed2.shape[0]
>>> # test with unitless data
>>> coco_img.img['resolution'] = 1
>>> img_delayed2 = coco_img.imdelay(space='image', resolution=3.1)
>>> vid_delayed2 = coco_img.imdelay(space='video', resolution='3.1 meter')
>>> np.ceil(img_delayed1.shape[0] / 3.1) == img_delayed2.shape[0]
>>> np.ceil(vid_delayed1.shape[0] / 3.1) == vid_delayed2.shape[0]
```

valid_region(*space='image'*)

If this image has a valid polygon, return it in image, or video space

Returns

None | kwimage.MultiPolygon

property warp_vid_from_img

Affine transformation that warps image space -> video space.

Returns
The transformation matrix

Return type
kwimage.Affine

property warp_img_from_vid

Affine transformation that warps video space -> image space.

Returns
The transformation matrix

Return type
kwimage.Affine

_warp_for_resolution(*space*, *resolution=None*)

Compute a transform from image-space to the requested space at a target resolution.

_annot_segmentation(*ann*, *space='video'*, *resolution=None*)

" Load annotation segmentations in a requested space at a target resolution.

Example

```
>>> from kwcoco.coco_image import *  # NOQA
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('vidshapes8-msi-multisensor')
>>> coco_img = dset.coco_image(1)
>>> coco_img.img['resolution'] = '1 meter'
>>> ann = coco_img.annots().objs[0]
>>> img_sseg = coco_img._annot_segmentation(ann, space='image')
>>> vid_sseg = coco_img._annot_segmentation(ann, space='video')
>>> img_sseg_2m = coco_img._annot_segmentation(ann, space='image', resolution=
↪'2 meter')
>>> vid_sseg_2m = coco_img._annot_segmentation(ann, space='video', resolution=
↪'2 meter')
>>> print(f'img_sseg.area    = {img_sseg.area}')
>>> print(f'vid_sseg.area    = {vid_sseg.area}')
>>> print(f'img_sseg_2m.area = {img_sseg_2m.area}')
>>> print(f'vid_sseg_2m.area = {vid_sseg_2m.area}')
```

_annot_segmentations(*anns*, *space='video'*, *resolution=None*)

" Load multiple annotation segmentations in a requested space at a target resolution.

Example

```
>>> from kwcoco.coco_image import *  # NOQA
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('vidshapes8-msi-multisensor')
>>> coco_img = dset.coco_image(1)
>>> coco_img.img['resolution'] = '1 meter'
>>> ann = coco_img.annots().objs[0]
>>> img_sseg = coco_img._annot_segmentations([ann], space='image')
>>> vid_sseg = coco_img._annot_segmentations([ann], space='video')
>>> img_sseg_2m = coco_img._annot_segmentations([ann], space='image',␣
↪resolution='2 meter')
>>> vid_sseg_2m = coco_img._annot_segmentations([ann], space='video',␣
↪resolution='2 meter')
>>> print(f'img_sseg.area    = {img_sseg[0].area}')
>>> print(f'vid_sseg.area    = {vid_sseg[0].area}')
>>> print(f'img_sseg_2m.area = {img_sseg_2m[0].area}')
>>> print(f'vid_sseg_2m.area = {vid_sseg_2m[0].area}')
```

resolution(*space='image'*, *channel=None*, *RESOLUTION_KEY=None*)

Returns the resolution of this CocoImage in the requested space if known. Errors if this information is not registered.

Parameters

- **space** (*str*) – the space to the resolution of. Can be either "image", "video", or "asset".
- **channel** (*str* | *kwcoco.FusedChannelSpec* | *None*) – a channel that identifies a single asset, only relevant if asking for asset space

Returns

has items mag (with the magnitude of the resolution) and unit, which is a convinience and only loosely enforced.

Return type

Dict

Example

```
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('vidshapes8-multispectral')
>>> self = dset.coco_image(1)
>>> self.img['resolution'] = 1
>>> self.resolution()
>>> self.img['resolution'] = '1 meter'
>>> self.resolution(space='video')
{'mag': (1.0, 1.0), 'unit': 'meter'}
>>> self.resolution(space='asset', channel='B11')
>>> self.resolution(space='asset', channel='B1')
```

_scalefactor_for_resolution(*space*, *resolution*, *channel=None*, *RESOLUTION_KEY=None*)

Given image or video space, compute the scale factor needed to achieve the target resolution.

Use this to implement scale_resolution_from_img scale_resolution_from_vid

Parameters

- **space** (*str*) – the space to the resolution of. Can be either "image", "video", or "asset".
- **resolution** (*str | float | int*) – the resolution (ideally with units) you want.
- **channel** (*str | kwcoco.FusedChannelSpec | None*) – a channel that identifies a single asset, only relevant if asking for asset space

Returns
the x and y scale factor that can be used to scale the underlying "space" to acheive the requested resolution.

Return type
Tuple[float, float]

_detections_for_resolution(*space='video'*, *resolution=None*, *aids=None*, *RESOLUTION_KEY=None*)

This is slightly less than ideal in terms of API, but it will work for now.

add_auxiliary_item(***kwargs*)

delay(***kwargs*)

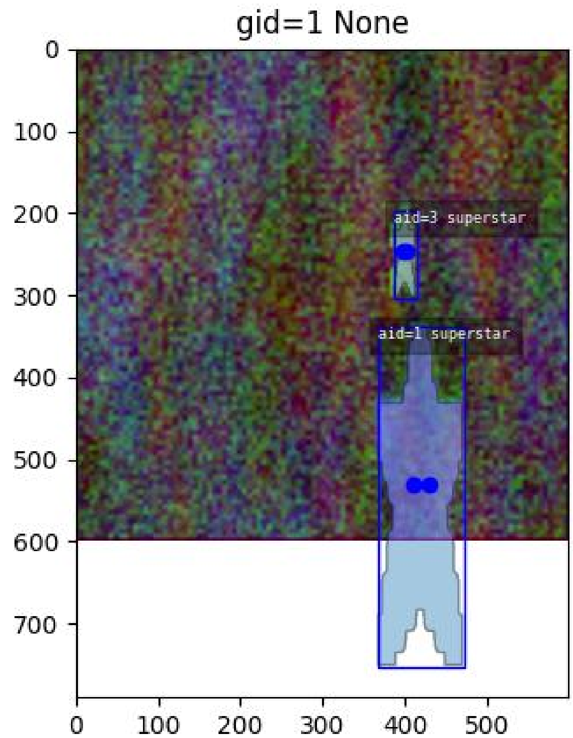
show(***kwargs*)

Show the image with matplotlib if possible

SeeAlso:
kwcoco.CocoDataset.show_image()

Example

```
>>> # xdoctest: +REQUIRES(module:kwplot)
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('vidshapes8-multispectral')
>>> self = dset.coco_image(1)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autoplt()
>>> self.show()
```



draw(***kwargs*)

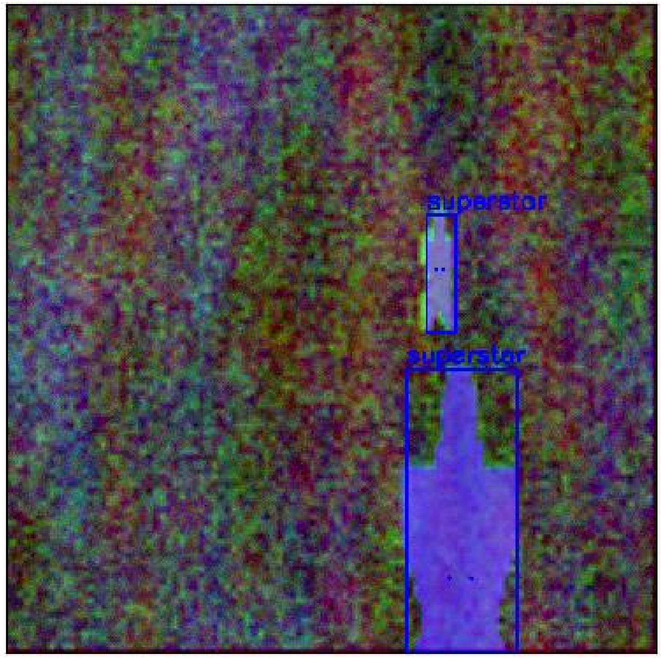
Draw the image on an ndarray using opencv

SeeAlso:

`kwcoco.CocoDataset.draw_image()`

Example

```
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('vidshapes8-multispectral')
>>> self = dset.coco_image(1)
>>> canvas = self.draw()
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(canvas)
```



class kwcoco.**CocoSqlDatabase**(*uri=None*, *tag=None*, *img_root=None*)

Bases: *AbstractCocoDataset*, *MixinCocoAccessors*, *MixinCocoObjects*, *MixinCocoStats*, *MixinCocoDraw*, NiceRepr

Provides an API nearly identical to `kwcoco.CocoDatabase`, but uses an SQL backend data store. This makes it robust to copy-on-write memory issues that arise when forking, as discussed in[1].

Note: By default constructing an instance of the CocoSqlDatabase does not create a connection to the databse. Use the *connect()* method to open a connection.

References

Example

```
>>> # xdoctest: +REQUIRES(module:sqlalchemy)
>>> from kwcoco.coco_sql_dataset import *  # NOQA
>>> sql_dset, dct_dset = demo()
>>> dset1, dset2 = sql_dset, dct_dset
>>> tag1, tag2 = 'dset1', 'dset2'
>>> assert_dsets_allclose(sql_dset, dct_dset)
```

MEMORY_URI = 'sqlite:///:memory:'

[1] https://github.com/pytorch/pytorch/issues/13246

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

Create an SQL CocoDataset from the input pointer.

Example

import kwcoco dset = kwcoco.CocoDataset.demo('shapes8') data = dset.fpath self = CocoSqlDatabase.coerce(data)

from kwcoco.coco_sql_dataset import CocoSqlDatabase import kwcoco dset = kwcoco.CocoDataset.coerce('spacenet7.kwcoco.json')

self = CocoSqlDatabase.coerce(dset)

from kwcoco.coco_sql_dataset import CocoSqlDatabase sql_dset = CocoSqlDatabase.coerce('spacenet7.kwcoco.json')

from kwcoco.coco_sql_dataset import CocoSqlDatabase import kwcoco sql_dset = kwcoco.CocoDataset.coerce('_spacenet7.kwcoco.view.v006.sqlite')

disconnect()

Drop references to any SQL or cache objects

connect(*readonly=False*, *verbose=0*, *must_exist=False*)

Connects this instance to the underlying database.

References

details on read only mode, some of these didnt seem to work https://github.com/sqlalchemy/sqlalchemy/blob/master/lib/sqlalchemy/dialects/sqlite/pysqlite.py#L71 https://github.com/pudo/dataset/issues/136 https://writeonly.wordpress.com/2009/07/16/simple-read-only-sqlalchemy-sessions/

CommandLine

```
KWCOCO_WITH_POSTGRESQL=1 xdoctest -m /home/joncrall/code/kwcoco/kwcoco/coco_sql_
↪dataset.py CocoSqlDatabase.connect
```

Example

```
>>> # xdoctest: +REQUIRES(env:KWCOCO_WITH_POSTGRESQL)
>>> # xdoctest: +REQUIRES(module:sqlalchemy)
>>> # xdoctest: +REQUIRES(module:psycopg2)
>>> from kwcoco.coco_sql_dataset import *  # NOQA
>>> dset = CocoSqlDatabase('postgresql+psycopg2://kwcoco:kwcoco_
↪pw@localhost:5432/mydb')
>>> self = dset
>>> dset.connect(verbose=1)
```

property fpath

delete(*verbose=0*)

table_names()

populate_from(*dset*, *verbose=1*)

Copy the information in a `CocoDataset` into this SQL database.

CommandLine

```
xdoctest -m kwcoco.coco_sql_dataset CocoSqlDatabase.populate_from:1
```

Example

```
>>> # xdoctest: +REQUIRES(module:sqlalchemy)
>>> from kwcoco.coco_sql_dataset import _benchmark_dset_readtime  # NOQA
>>> import kwcoco
>>> from kwcoco.coco_sql_dataset import *
>>> dset2 = dset = kwcoco.CocoDataset.demo()
>>> dset2.clear_annotations()
>>> dset1 = self = CocoSqlDatabase('sqlite:///:memory:')
>>> self.connect()
>>> self.populate_from(dset)
>>> dset1_images = list(dset1.dataset['images'])
>>> print('dset1_images = {}'.format(ub.urepr(dset1_images, nl=1)))
>>> print(dset2.dumps(newlines=True))
>>> assert_dsets_allclose(dset1, dset2, tag1='sql', tag2='dct')
>>> ti_sql = _benchmark_dset_readtime(dset1, 'sql')
>>> ti_dct = _benchmark_dset_readtime(dset2, 'dct')
>>> print('ti_sql.rankings = {}'.format(ub.urepr(ti_sql.rankings, nl=2,␣
↪precision=6, align=':')))
>>> print('ti_dct.rankings = {}'.format(ub.urepr(ti_dct.rankings, nl=2,␣
↪precision=6, align=':')))
```

Example

```
>>> # xdoctest: +REQUIRES(module:sqlalchemy)
>>> from kwcoco.coco_sql_dataset import _benchmark_dset_readtime  # NOQA
>>> import kwcoco
>>> from kwcoco.coco_sql_dataset import *
>>> dset2 = dset = kwcoco.CocoDataset.demo('vidshapes1')
>>> dset1 = self = CocoSqlDatabase('sqlite:///:memory:')
>>> self.connect()
>>> self.populate_from(dset)
>>> for tablename in dset1.dataset.keys():
>>>     print(tablename)
>>>     table = dset1.pandas_table(tablename)
>>>     print(table)
```

Example

```
>>> # xdoctest: +REQUIRES(module:sqlalchemy)
>>> from kwcoco.coco_sql_dataset import _benchmark_dset_readtime  # NOQA
>>> import kwcoco
>>> from kwcoco.coco_sql_dataset import *
>>> dset2 = dset = kwcoco.CocoDataset.demo()
>>> dset1 = self = CocoSqlDatabase('sqlite:///:memory:')
>>> self.connect()
>>> self.populate_from(dset)
>>> assert_dsets_allclose(dset1, dset2, tag1='sql', tag2='dct')
>>> ti_sql = _benchmark_dset_readtime(dset1, 'sql')
>>> ti_dct = _benchmark_dset_readtime(dset2, 'dct')
>>> print('ti_sql.rankings = {}'.format(ub.urepr(ti_sql.rankings, nl=2,␣
↪precision=6, align=':')))
>>> print('ti_dct.rankings = {}'.format(ub.urepr(ti_dct.rankings, nl=2,␣
↪precision=6, align=':')))
```

CommandLine

```
KWCOCO_WITH_POSTGRESQL=1 xdoctest -m /home/joncrall/code/kwcoco/kwcoco/coco_sql_
↪dataset.py CocoSqlDatabase.populate_from:1
```

Example

```
>>> # xdoctest: +REQUIRES(env:KWCOCO_WITH_POSTGRESQL)
>>> # xdoctest: +REQUIRES(module:sqlalchemy)
>>> # xdoctest: +REQUIRES(module:psycopg2)
>>> from kwcoco.coco_sql_dataset import *  # NOQA
>>> import kwcoco
>>> dset = dset2 = kwcoco.CocoDataset.demo()
>>> self = dset1 = CocoSqlDatabase('postgresql+psycopg2://kwcoco:kwcoco_
↪pw@localhost:5432/test_populate')
>>> self.delete(verbose=1)
>>> self.connect(verbose=1)
>>> #self.populate_from(dset)
```

property dataset

property anns

property cats

property imgs

property name_to_cat

pandas_table(*table_name*, *strict=False*)

Loads an entire SQL table as a pandas DataFrame

Parameters

table_name (*str*) – name of the table

Returns
pandas.DataFrame

Example

```
>>> # xdoctest: +REQUIRES(module:sqlalchemy)
>>> # xdoctest: +REQUIRES(module:pandas)
>>> from kwcoco.coco_sql_dataset import *  # NOQA
>>> self, dset = demo()
>>> table_df = self.pandas_table('annotations')
>>> print(table_df)
>>> table_df = self.pandas_table('categories')
>>> print(table_df)
>>> table_df = self.pandas_table('videos')
>>> print(table_df)
>>> table_df = self.pandas_table('images')
>>> print(table_df)
>>> table_df = self.pandas_table('tracks')
>>> print(table_df)
```

raw_table(*table_name*)

_raw_tables()

Example

```
>>> # xdoctest: +REQUIRES(module:sqlalchemy)
>>> from kwcoco.coco_sql_dataset import *  # NOQA
>>> import pandas as pd
>>> self, dset = demo()
>>> targets = self._raw_tables()
>>> for tblname, table in targets.items():
...     print(f'tblname={tblname}')
...     print(pd.DataFrame(table))
```

_column_lookup(*tablename*, *key*, *rowids*, *default=NoParam*, *keepid=False*)

Convinience method to lookup only a single column of information

Example

```
>>> # xdoctest: +REQUIRES(module:sqlalchemy)
>>> from kwcoco.coco_sql_dataset import *  # NOQA
>>> self, dset = demo(10)
>>> tablename = 'annotations'
>>> key = 'category_id'
>>> rowids = list(self.anns.keys())[::3]
>>> cids1 = self._column_lookup(tablename, key, rowids)
>>> cids2 = self.annots(rowids).get(key)
>>> cids3 = dset.annots(rowids).get(key)
>>> assert cids3 == cids2 == cids1
```

(continues on next page)

(continued from previous page)

```
>>> # Test json columns work
>>> vals1 = self._column_lookup(tablename, 'bbox', rowids)
>>> vals2 = self.annots(rowids).lookup('bbox')
>>> vals3 = dset.annots(rowids).lookup('bbox')
>>> assert vals1 == vals2 == vals3
>>> vals1 = self._column_lookup(tablename, 'segmentation', rowids)
>>> vals2 = self.annots(rowids).lookup('segmentation')
>>> vals3 = dset.annots(rowids).lookup('segmentation')
>>> assert vals1 == vals2 == vals3
```

_all_rows_column_lookup(*tablename*, *keys*)

Convinience method to look up all rows from a table and only a few columns.

Example

```
>>> # xdoctest: +REQUIRES(module:sqlalchemy)
>>> from kwcoco.coco_sql_dataset import *  # NOQA
>>> self, dset = demo(10)
>>> tablename = 'annotations'
>>> keys = ['id', 'category_id']
>>> rows = self._all_rows_column_lookup(tablename, keys)
```

tabular_targets()

Convinience method to create an in-memory summary of basic annotation properties with minimal SQL overhead.

Example

```
>>> # xdoctest: +REQUIRES(module:sqlalchemy)
>>> from kwcoco.coco_sql_dataset import *  # NOQA
>>> self, dset = demo()
>>> targets = self.tabular_targets()
>>> print(targets.pandas())
```

_table_names()

property bundle_dpath

property data_fpath

data_fpath is an alias of fpath

_orig_coco_fpath()

Hack to reconstruct the original name. Makes assumptions about how naming is handled elsewhere. There should be centralized logic about how to construct side-car names that can be queried for inversed like this.

Returns

ub.Path | None

_abc_impl = <_abc._abc_data object>

_cached_hashid()

Compatibility with the way the exiting cached hashid in the coco dataset is used. Both of these functions are private and subject to change (and need optimization).

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 delayed_image.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))
```

_alias_lut = {'dxdy': ['dx', 'dy'], 'fxfy': ['fx', 'fy'], 'rgb': ['r', 'g', 'b'], 'rgba': ['r', 'g', 'b', 'a']}

_memo = {'B1': <FusedChannelSpec(B1)>, 'B10': <FusedChannelSpec(B10)>, 'B11': <FusedChannelSpec(B11)>, 'B8': <FusedChannelSpec(B8)>, 'B8a': <FusedChannelSpec(B8a)>}

_size_lut = {'dxdy': 2, 'fxfy': 2, 'rgb': 3, 'rgba': 4}

classmethod concat(*items*)

property spec

unique()

classmethod parse(*spec*)

classmethod coerce(*data*)

Example

```
>>> from delayed_image.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 delayed_image.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 delayed_image.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 delayed_image.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

Example

```
>>> from delayed_image.channel_spec import *  # NOQA
>>> self = FusedChannelSpec.coerce('b1|Z:3|b2|b3|rgb')
>>> self.as_path()
b1_Z..3_b2_b3_rgb
```

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 delayed_image.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.urepr(component_indices, nl=1, _
↪dict_sort_behavior='old')))
component_indices = {
    'disparity': (slice(...), slice(...), slice(0, 1, None)),
    'flowx': (slice(...), slice(...), slice(4, 5, None)),
    'flowy': (slice(...), slice(...), slice(5, 6, None)),
    'rgb': (slice(...), slice(...), slice(1, 4, None)),
}
```

streams()

Idempotence with *ChannelSpec.streams()*

fuse()

Idempotence with *ChannelSpec.streams()*

class kwcoco.**SensorChanSpec**(*spec: str*)

Bases: NiceRepr

The public facing API for the sensor / channel specification

Example

```
>>> # xdoctest: +REQUIRES(module:lark)
>>> from delayed_image.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.urepr(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 delayed_image.sensorchan_spec import SensorChanSpec
>>> import delayed_image
>>> 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 delayed_image
>>> c = delayed_image.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 delayed_image.sensorchan_spec import SensorChanSpec
>>> import delayed_image
>>> 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 delayed_image.sensorchan_spec import *  # NOQA
>>> from delayed_image.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()

Example

```
>>> # xdoctest: +REQUIRES(module:lark)
>>> from delayed_image import SensorChanSpec
>>> a = SensorChanSpec.coerce('Cam1:(red,blue)')
>>> b = SensorChanSpec.coerce('Cam2:(blue,green)')
>>> c = (a + b).concise()
>>> print(c)
(Cam1,Cam2):blue,Cam1:red,Cam2:green
>>> # Note the importance of parenthesis in the previous example
>>> # otherwise channels will be assigned to `*` the generic sensor.
>>> a = SensorChanSpec.coerce('Cam1:red,blue')
>>> b = SensorChanSpec.coerce('Cam2:blue,green')
>>> c = (a + b).concise()
>>> print(c)
(*,Cam2):blue,*:green,Cam1:red
```

streams()

Returns
List of sensor-names and fused channel specs

Return type
List[FusedSensorChanSpec]

late_fuse(**others*)

Example

```
>>> # xdoctest: +REQUIRES(module:lark)
>>> import delayed_image
>>> from delayed_image import sensorchan_spec
>>> import delayed_image
>>> delayed_image.SensorChanSpec = sensorchan_spec.SensorChanSpec  # hack for 3.
↪6
>>> a = delayed_image.SensorChanSpec.coerce('A|B|C,edf')
>>> b = delayed_image.SensorChanSpec.coerce('A12')
>>> c = delayed_image.SensorChanSpec.coerce('')
>>> d = delayed_image.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 delayed_image
>>> a = delayed_image.SensorChanSpec.coerce('A|B|C,edf').normalize()
>>> b = delayed_image.SensorChanSpec.coerce('A12').normalize()
>>> c = delayed_image.SensorChanSpec.coerce('').normalize()
>>> d = delayed_image.SensorChanSpec.coerce('rgb').normalize()
>>> print(a.late_fuse(b).spec)
>>> print((a + b).spec)
>>> print((b + a).spec)
>>> print((a + b + c).spec)
>>> print(sum([a, b, c, d]).spec)
*:A|B|C,*:edf,*:A12
*:A|B|C,*:edf,*:A12
*:A12,*:A|B|C,*:edf
*:A|B|C,*:edf,*:A12,*:
*:A|B|C,*:edf,*:A12,*:,*:rgb
>>> print((a.late_fuse(b)).concise())
>>> print(((a + b)).concise())
>>> print(((b + a)).concise())
>>> print(((a + b + c)).concise())
>>> print((sum([a, b, c, d])).concise())
*:(A|B|C,edf,A12)
*:(A|B|C,edf,A12)
*:(A12,A|B|C,edf)
*:(A|B|C,edf,A12,)
*:(A|B|C,edf,A12,,r|g|b)
```

Example

```
>>> # Test multi-arg case
>>> import delayed_image
>>> a = delayed_image.SensorChanSpec.coerce('A|B|C,edf')
>>> b = delayed_image.SensorChanSpec.coerce('A12')
>>> c = delayed_image.SensorChanSpec.coerce('')
>>> d = delayed_image.SensorChanSpec.coerce('rgb')
>>> others = [b, c, d]
>>> print(a.late_fuse(*others).spec)
>>> print(delayed_image.SensorChanSpec.late_fuse(a, b, c, d).spec)
A|B|C,edf,A12,rgb
A|B|C,edf,A12,rgb
```

matching_sensor(*sensor*)

Get the components corresponding to a specific sensor

Parameters
sensor (*str*) – the name of the sensor to match

Example

```
>>> # xdoctest: +REQUIRES(module:lark)
>>> import delayed_image
>>> self = delayed_image.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 delayed_image
>>> self = delayed_image.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
```

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

[Voluptuous] https://pypi.org/project/voluptuous

[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

Symbols

A

B

C

D

E

F

J

K

L

M

O

P

S

T

U

V

W

X