imageio Documentation

Release 2.8.0

imageio contributors

Feb 18, 2020

Contents

Imageio is a Python library that provides an easy interface to read and write a wide range of image data, including animated images, volumetric data, and scientific formats. It is cross-platform, runs on Python 3.5+, and is easy to install.

Main website: http://imageio.github.io

Contents:

CHAPTER 1

Getting started

1.1 Installing imageio

Imageio is written in pure Python, so installation is easy. Imageio works on Python 3.5+. It also works on Pypy. Imageio depends on Numpy and Pillow. For some formats, imageio needs additional libraries/executables (e.g. ffmpeg), which imageio helps you to download/install.

To install imageio, use one of the following methods:

- If you are in a conda env: `conda install -c conda-forge imageio`
- If you have pip: `pip install imageio`
- Good old `python setup.py install`

After installation, checkout the *examples* and *user api*.

Still running Python 2.7? Read here.

1.1.1 Developers

For developers, we provide a simple mechanism to allow importing imageio from the cloned repository. See the file `imageio.proxy.py` for details.

Further imageio has the following dev-dependencies:

```
pip install black flake8 pytest pytest-cov sphinx numpydoc
```

1.2 Imageio usage examples

Some of these examples use Visvis to visualize the image data, but one can also use Matplotlib to show the images.

Imageio provides a range of *example images*, which can be used by using a URI like `'imageio:chelsea.png'`. The images are automatically downloaded if not already present on your system. Therefore most examples below should just work.

1.2.1 Read an image of a cat

Probably the most important thing you'll ever need.

```
import imageio

im = imageio.imread('imageio:chelsea.png')
print(im.shape)
```

If the image is a GIF:

```
import imageio

im = imageio.get_reader('cat.gif')
for frame in im:
    print(im.shape)  # Each frame is a numpy matrix
```

If the GIF is stored in memory:

```
import imageio

im = imageio.get_reader(image_bytes, '.gif')
```

1.2.2 Read from fancy sources

Imageio can read from filenames, file objects, http, zipfiles and bytes.

```
import imageio
import visvis as vv

im = imageio.imread('http://upload.wikimedia.org/wikipedia/commons/d/de/Wikipedia_
↪Logo_1.0.png')
vv.imshow(im)
```

Note: reading from HTTP and zipfiles works for many formats including png and jpeg, but may not work for all formats (some plugins "seek" the file object, which HTTP/zip streams do not support). In such a case one can download/extract the file first. For HTTP one can use something like `imageio.imread(imageio.core.urlopen(url).read(), '.gif')`.

1.2.3 Iterate over frames in a movie

```
import imageio

reader = imageio.get_reader('imageio:cockatoo.mp4')
for i, im in enumerate(reader):
    print('Mean of frame %i is %1.1f' % (i, im.mean()))
```

1.2.4 Grab screenshot or image from the clipboard

(Screenshots are supported on Windows and OS X, clipboard on Windows only.)

```
import imageio

im_screen = imageio.imread('<screen>')
im_clipboard = imageio.imread('<clipboard>')
```

1.2.5 Grab frames from your webcam

Use the special `<video0>` uri to read frames from your webcam (via the ffmpeg plugin). You can replace the zero with another index in case you have multiple cameras attached. You need to `pip install imageio-ffmpeg` in order to use this plugin.

```
import imageio
import visvis as vv

reader = imageio.get_reader('<video0>')
t = vv.imshow(reader.get_next_data(), clim=(0, 255))
for im in reader:
    vv.processEvents()
    t.SetData(im)
```

1.2.6 Convert a movie

Here we take a movie and convert it to gray colors. Of course, you can apply any kind of (image) processing to the image here ... You need to `pip install imageio-ffmpeg` in order to use the ffmpeg plugin.

```
import imageio

reader = imageio.get_reader('imageio:cockatoo.mp4')
fps = reader.get_meta_data()['fps']

writer = imageio.get_writer('~/cockatoo_gray.mp4', fps=fps)

for im in reader:
    writer.append_data(im[:, :, 1])
writer.close()
```

1.2.7 Read medical data (DICOM)

```
import imageio
dirname = 'path/to/dicom/files'

# Read as loose images
ims = imageio.mimread(dirname, 'DICOM')
# Read as volume
vol = imageio.volread(dirname, 'DICOM')
# Read multiple volumes (multiple DICOM series)
vols = imageio.mvolread(dirname, 'DICOM')
```

1.2.8 Volume data

```
import imageio
import visvis as vv

vol = imageio.volread('imageio:stent.npz')
vv.volshow(vol)
```

1.2.9 Writing videos with FFMPEG and vaapi

Using vaapi (on Linux only) (intel only?) can help free up resources on your laptop while you are encoding videos. One notable difference between vaapi and x264 is that vaapi doesn't support the color format yuv420p.

Note, you will need ffmpeg compiled with vaapi for this to work.

```
import imageio
import numpy as np

# All images must be of the same size
image1 = np.stack([imageio.imread('imageio:camera.png')] * 3, 2)
image2 = imageio.imread('imageio:astronaut.png')
image3 = imageio.imread('imageio:immunohistochemistry.png')

w = imageio.get_writer('my_video.mp4', format='FFMPEG', mode='I', fps=1,
                       codec='h264_vaapi',
                       output_params=['-vaapi_device',
                                      '/dev/dri/renderD128',
                                      '-vf',
                                      'format=gray|nv12,hwupload'],
                       pixelformat='vaapi_vld')
w.append_data(image1)
w.append_data(image2)
w.append_data(image3)
w.close()
```

A little bit of explanation:

- `output_params`
  - `vaapi_device` speficifies the encoding device that will be used.
  - `vf` and `format` tell ffmpeg that it must upload to the dedicated hardware. Since vaapi only supports a subset of color formats, we ensure that the video is in either gray or nv12 before uploading it. The `or` operation is acheived with `|`.
- `pixelformat`: set to `'vaapi_vld'` to avoid a warning in ffmpeg.
- `codec`: the code you wish to use to encode the video. Make sure your hardware supports the chosen codec. If your hardware supports h265, you may be able to encode using `'hevc_vaapi'`

1.2.10 Optimizing a GIF using pygifsicle

When creating a GIF using imageio the resulting images can get quite heavy, as the created GIF is not optimized. This can be useful when the elaboration process for the GIF is not finished yet (for instance if some elaboration on specific frames stills need to happen), but it can be an issue when the process is finished and the GIF is unexpectedly big.

GIF files can be compressed in several ways, the most common one method (the one used here) is saving just the differences between the following frames. In this example, we apply the described method to a given GIF *my_gif* using pygifsicle, a porting of the general-purpose GIF editing command-line library gifsicle. To install pygifsicle and gifsicle, read the setup on the project page: it boils down to installing the package using pip and following the console instructions:

```
pip install pygifsicle
```

Now, let's start by creating a gif using imageio:

```
import imageio
import matplotlib.pyplot as plt

n = 100
gif_path = "test.gif"
frames_path = "{i}.jpg"

n = 100
plt.figure(figsize=(4,4))
for i, x in enumerate(range(n)):
    plt.scatter(x/n, x/n)
    plt.xlim(0, 1)
    plt.ylim(0, 1)
    plt.savefig("{i}.jpg".format(i=i))

with imageio.get_writer(gif_path, mode='I') as writer:
    for i in range(n):
        writer.append_data(imageio.imread(frames_path.format(i=i)))
```

This way we obtain a 2.5MB gif.

We now want to compress the created GIF. We can either overwrite the initial one or create a new optimized one: We start by importing the library method:

```
from pygifsicle import optimize

optimize(gif_path, "optimized.gif") # For creating a new one
optimize(gif_path) # For overwriting the original one
```

The new optimized GIF now weights 870KB, almost 3 times less.

Putting everything together:

```
import imageio
import matplotlib.pyplot as plt
from pygifsicle import optimize

n = 100
gif_path = "test.gif"
frames_path = "{i}.jpg"

n = 100
plt.figure(figsize=(4,4))
for i, x in enumerate(range(n)):
    plt.scatter(x/n, x/n)
    plt.xlim(0, 1)
    plt.ylim(0, 1)
    plt.savefig("{i}.jpg".format(i=i))
```

(continues on next page)

(continued from previous page)

```
with imageio.get_writer(gif_path, mode='I') as writer:
    for i in range(n):
        writer.append_data(imageio.imread(frames_path.format(i=i)))

optimize(gif_path)
```

1.3 Transitioning from Scipy's imread

Scipy is deprecating their image I/O functionality.

This document is intended to help people coming from Scipy to adapt to Imageio's *imread* function. We recommend reading the *user api* and checkout some *examples* to get a feel of imageio.

Imageio makes use of variety of plugins to support reading images (and volumes/movies) from many different formats. Fortunately, Pillow is the main plugin for common images, which is the same library as used by Scipy's `imread`. Note that Imageio automatically selects a plugin based on the image to read (unless a format is explicitly specified), but uses Pillow where possible.

In short terms: For images previously read by Scipy's imread, imageio should generally use Pillow as well, and imageio provides the same functionality as Scipy in these cases. But keep in mind:

- Instead of `mode`, use the `pilmode` keyword argument.
- Instead of `flatten`, use the `as_gray` keyword argument.
- The documentation for the above arguments is not on *imread*, but on the docs of the individual formats, e.g. PNG.
- Imageio's functions all return numpy arrays, albeit as a subclass (so that meta data can be attached).

CHAPTER 2

Reference

2.1 Imageio's user API

These functions represent imageio's main interface for the user. They provide a common API to read and write image data for a large variety of formats. All read and write functions accept keyword arguments, which are passed on to the format that does the actual work. To see what keyword arguments are supported by a specific format, use the *help()* function.

Functions for reading:

- *imread()* - read an image from the specified uri
- *mimread()* - read a series of images from the specified uri
- *volread()* - read a volume from the specified uri
- *mvolread()* - read a series of volumes from the specified uri

Functions for saving:

- *imwrite()* - write an image to the specified uri
- *mimwrite()* - write a series of images to the specified uri
- *volwrite()* - write a volume to the specified uri
- *mvolwrite()* - write a series of volumes to the specified uri

More control:

For a larger degree of control, imageio provides functions *get_reader()* and *get_writer()*. They respectively return an *Reader* and an *Writer* object, which can be used to read/write data and meta data in a more controlled manner. This also allows specific scientific formats to be exposed in a way that best suits that file-format.

All read-functions return images as numpy arrays, and have a `meta` attribute; the meta-data dictionary can be accessed with `im.meta`. To make this work, imageio actually makes use of a subclass of `np.ndarray`. If needed, the image can be converted to a plain numpy array using `np.asarray(im)`.

Supported resource URI's:

All functions described here accept a URI to describe the resource to read from or write to. These can be a wide range of things. (Imageio takes care of handling the URI so that plugins can access the data in an easy way.)

For reading and writing:

- a normal filename, e.g. `'c:\foo\bar.png'`
- a file in a zipfile, e.g. `'c:\foo\bar.zip\eggs.png'`
- a file object with a `read()` / `write()` method.

For reading:

- an http/ftp address, e.g. `'http://example.com/foo.png'`
- the raw bytes of an image file
- `get_reader("<video0>")` to grab images from a (web) camera.
- `imread("<screen>")` to grab a screenshot (on Windows or OS X).
- `imread("<clipboard>")` to grab an image from the clipboard (on Windows).

For writing one can also use `'<bytes>'` or `imageio.RETURN_BYTES` to make a write function return the bytes instead of writing to a file.

Note that reading from HTTP and zipfiles works for many formats including png and jpeg, but may not work for all formats (some plugins "seek" the file object, which HTTP/zip streams do not support). In such a case one can download/extract the file first. For HTTP one can use something like `imageio.imread(imageio.core.urlopen(url).read(), '.gif')`.

`imageio.`**`help`**(*name=None*)

Print the documentation of the format specified by name, or a list of supported formats if name is omitted.

Parameters

name [str] Can be the name of a format, a filename extension, or a full filename. See also the *formats page*.

`imageio.`**`show_formats`**()

Show a nicely formatted list of available formats

`imageio.`**`imread`**(*uri*, *format=None*, ***kwargs*)

Reads an image from the specified file. Returns a numpy array, which comes with a dict of meta data at its 'meta' attribute.

Note that the image data is returned as-is, and may not always have a dtype of uint8 (and thus may differ from what e.g. PIL returns).

Parameters

uri [{str, pathlib.Path, bytes, file}] The resource to load the image from, e.g. a filename, pathlib.Path, http address or file object, see the docs for more info.

format [str] The format to use to read the file. By default imageio selects the appropriate for you based on the filename and its contents.

kwargs [...] Further keyword arguments are passed to the reader. See `help()` to see what arguments are available for a particular format.

imageio.**imwrite**(*uri*, *im*, *format=None*, ***kwargs*)
Write an image to the specified file.

Parameters

uri [{str, pathlib.Path, file}] The resource to write the image to, e.g. a filename, pathlib.Path or file object, see the docs for more info.

im [numpy.ndarray] The image data. Must be NxM, NxMx3 or NxMx4.

format [str] The format to use to read the file. By default imageio selects the appropriate for you based on the filename and its contents.

kwargs [...] Further keyword arguments are passed to the writer. See `help()` to see what arguments are available for a particular format.

---

imageio.**mimread**(*uri*, *format=None*, *memtest="256MB"*, ***kwargs*)
Reads multiple images from the specified file. Returns a list of numpy arrays, each with a dict of meta data at its 'meta' attribute.

Parameters

uri [{str, pathlib.Path, bytes, file}] The resource to load the images from, e.g. a filename,pathlib.Path, http address or file object, see the docs for more info.

format [str] The format to use to read the file. By default imageio selects the appropriate for you based on the filename and its contents.

memtest [{bool, int, float, str}] If truthy, this function will raise an error if the resulting list of images consumes greater than the amount of memory specified. This is to protect the system from using so much memory that it needs to resort to swapping, and thereby stall the computer. E.g. `mimread('hunger_games.avi')`.

If the argument is a number, that will be used as the threshold number of bytes.

If the argument is a string, it will be interpreted as a number of bytes with SI/IEC prefixed units (e.g. '1kB', '250MiB', '80.3YB').

- Units are case sensitive
- k, M etc. represent a 1000-fold change, where Ki, Mi etc. represent 1024-fold
- The "B" is optional, but if present, must be capitalised

If the argument is True, the default will be used, for compatibility reasons.

Default: '256MB'

kwargs [...] Further keyword arguments are passed to the reader. See `help()` to see what arguments are available for a particular format.

imageio.**mimwrite**(*uri*, *ims*, *format=None*, ***kwargs*)
Write multiple images to the specified file.

Parameters

uri [{str, pathlib.Path, file}] The resource to write the images to, e.g. a filename, pathlib.Path or file object, see the docs for more info.

ims [sequence of numpy arrays] The image data. Each array must be NxM, NxMx3 or NxMx4.

format [str] The format to use to read the file. By default imageio selects the appropriate for you based on the filename and its contents.

kwargs [...] Further keyword arguments are passed to the writer. See `help()` to see what arguments are available for a particular format.

---

imageio.**volread**(*uri*, *format=None*, ***kwargs*)

Reads a volume from the specified file. Returns a numpy array, which comes with a dict of meta data at its 'meta' attribute.

Parameters

uri [{str, pathlib.Path, bytes, file}] The resource to load the volume from, e.g. a filename, pathlib.Path, http address or file object, see the docs for more info.

format [str] The format to use to read the file. By default imageio selects the appropriate for you based on the filename and its contents.

kwargs [...] Further keyword arguments are passed to the reader. See `help()` to see what arguments are available for a particular format.

imageio.**volwrite**(*uri*, *vol*, *format=None*, ***kwargs*)

Write a volume to the specified file.

Parameters

uri [{str, pathlib.Path, file}] The resource to write the image to, e.g. a filename, pathlib.Path or file object, see the docs for more info.

vol [numpy.ndarray] The image data. Must be NxMxL (or NxMxLxK if each voxel is a tuple).

format [str] The format to use to read the file. By default imageio selects the appropriate for you based on the filename and its contents.

kwargs [...] Further keyword arguments are passed to the writer. See `help()` to see what arguments are available for a particular format.

---

imageio.**mvolread**(*uri*, *format=None*, *memtest='1GB'*, ***kwargs*)

Reads multiple volumes from the specified file. Returns a list of numpy arrays, each with a dict of meta data at its 'meta' attribute.

Parameters

uri [{str, pathlib.Path, bytes, file}] The resource to load the volumes from, e.g. a filename, pathlib.Path, http address or file object, see the docs for more info.

format [str] The format to use to read the file. By default imageio selects the appropriate for you based on the filename and its contents.

memtest [{bool, int, float, str}] If truthy, this function will raise an error if the resulting list of images consumes greater than the amount of memory specified. This is to protect the system from using so much memory that it needs to resort to swapping, and thereby stall the computer. E.g. `mimread('hunger_games.avi')`.

If the argument is a number, that will be used as the threshold number of bytes.

If the argument is a string, it will be interpreted as a number of bytes with SI/IEC prefixed units (e.g. '1kB', '250MiB', '80.3YB').

- Units are case sensitive

- k, M etc. represent a 1000-fold change, where Ki, Mi etc. represent 1024-fold
- The "B" is optional, but if present, must be capitalised

If the argument is True, the default will be used, for compatibility reasons.

Default: '1GB'

kwargs [...] Further keyword arguments are passed to the reader. See *help()* to see what arguments are available for a particular format.

imageio.**mvolwrite**(*uri*, *vols*, *format=None*, ***kwargs*)
Write multiple volumes to the specified file.

Parameters

uri [{str, pathlib.Path, file}] The resource to write the volumes to, e.g. a filename, pathlib.Path or file object, see the docs for more info.

ims [sequence of numpy arrays] The image data. Each array must be NxMxL (or NxMxLxK if each voxel is a tuple).

format [str] The format to use to read the file. By default imageio selects the appropriate for you based on the filename and its contents.

kwargs [...] Further keyword arguments are passed to the writer. See *help()* to see what arguments are available for a particular format.

---

imageio.**get_reader**(*uri*, *format=None*, *mode='?'*, ***kwargs*)
Returns a *Reader* object which can be used to read data and meta data from the specified file.

Parameters

uri [{str, pathlib.Path, bytes, file}] The resource to load the image from, e.g. a filename, pathlib.Path, http address or file object, see the docs for more info.

format [str] The format to use to read the file. By default imageio selects the appropriate for you based on the filename and its contents.

mode [{'i', 'I', 'v', 'V', '?'}] Used to give the reader a hint on what the user expects (default "?"): "i" for an image, "I" for multiple images, "v" for a volume, "V" for multiple volumes, "?" for don't care.

kwargs [...] Further keyword arguments are passed to the reader. See *help()* to see what arguments are available for a particular format.

imageio.**get_writer**(*uri*, *format=None*, *mode='?'*, ***kwargs*)
Returns a *Writer* object which can be used to write data and meta data to the specified file.

Parameters

uri [{str, pathlib.Path, file}] The resource to write the image to, e.g. a filename, pathlib.Path or file object, see the docs for more info.

format [str] The format to use to write the file. By default imageio selects the appropriate for you based on the filename.

mode [{'i', 'I', 'v', 'V', '?'}] Used to give the writer a hint on what the user expects (default '?'): "i" for an image, "I" for multiple images, "v" for a volume, "V" for multiple volumes, "?" for don't care.

kwargs [...] Further keyword arguments are passed to the writer. See *help()* to see what arguments are available for a particular format.

class imageio.core.format.**Reader**(*format*, *request*)

The purpose of a reader object is to read data from an image resource, and should be obtained by calling *get_reader()*.

A reader can be used as an iterator to read multiple images, and (if the format permits) only reads data from the file when new data is requested (i.e. streaming). A reader can also be used as a context manager so that it is automatically closed.

Plugins implement Reader's for different formats. Though rare, plugins may provide additional functionality (beyond what is provided by the base reader class).

Attributes

closed Whether the reader/writer is closed.

format The *Format* object corresponding to the current read/write operation.

request The *Request* object corresponding to the current read/write operation.

close(*self*)

Flush and close the reader/writer. This method has no effect if it is already closed.

closed

Whether the reader/writer is closed.

format

The *Format* object corresponding to the current read/write operation.

get_data(*index*, ***kwargs*)

Read image data from the file, using the image index. The returned image has a 'meta' attribute with the meta data. Raises IndexError if the index is out of range.

Some formats may support additional keyword arguments. These are listed in the documentation of those formats.

get_length()

Get the number of images in the file. (Note: you can also use `len(reader_object)`.)

The result can be:

- 0 for files that only have meta data
- 1 for singleton images (e.g. in PNG, JPEG, etc.)
- N for image series
- inf for streams (series of unknown length)

get_meta_data(*index=None*)

Read meta data from the file. using the image index. If the index is omitted or None, return the file's (global) meta data.

Note that `get_data` also provides the meta data for the returned image as an atrribute of that image.

The meta data is a dict, which shape depends on the format. E.g. for JPEG, the dict maps group names to subdicts and each group is a dict with name-value pairs. The groups represent the different metadata formats (EXIF, XMP, etc.).

get_next_data(***kwargs*)

Read the next image from the series.

Some formats may support additional keyword arguments. These are listed in the documentation of those formats.

iter_data()
Iterate over all images in the series. (Note: you can also iterate over the reader object.)

request
The *Request* object corresponding to the current read/write operation.

set_image_index(*index*)
Set the internal pointer such that the next call to get_next_data() returns the image specified by the index

class imageio.core.format.**Writer**(*format*, *request*)
The purpose of a writer object is to write data to an image resource, and should be obtained by calling *get_writer()*.

A writer will (if the format permits) write data to the file as soon as new data is provided (i.e. streaming). A writer can also be used as a context manager so that it is automatically closed.

Plugins implement Writer's for different formats. Though rare, plugins may provide additional functionality (beyond what is provided by the base writer class).

Attributes

closed Whether the reader/writer is closed.

format The *Format* object corresponding to the current read/write operation.

request The *Request* object corresponding to the current read/write operation.

append_data(*im*, *meta={}*)
Append an image (and meta data) to the file. The final meta data that is used consists of the meta data on the given image (if applicable), updated with the given meta data.

close(*self*)
Flush and close the reader/writer. This method has no effect if it is already closed.

closed
Whether the reader/writer is closed.

format
The *Format* object corresponding to the current read/write operation.

request
The *Request* object corresponding to the current read/write operation.

set_meta_data(*meta*)
Sets the file's (global) meta data. The meta data is a dict which shape depends on the format. E.g. for JPEG the dict maps group names to subdicts, and each group is a dict with name-value pairs. The groups represents the different metadata formats (EXIF, XMP, etc.).

Note that some meta formats may not be supported for writing, and individual fields may be ignored without warning if they are invalid.

2.2 Imageio formats

This page lists all formats currently supported by imageio. Each format can support extra keyword arguments for reading and writing, which can be specified in the call to `get_reader()`, `get_writer()`, `imread()`, `imwrite()` etc. Further, formats are free to provide additional methods on their Reader and Writer objects. These parameters and extra methods are specified in the documentation for each format.

2.2.1 Single images

- TIFF - TIFF format
- BMP-PIL - Windows Bitmap
- BUFR-PIL - BUFR
- CUR-PIL - Windows Cursor
- DCX-PIL - Intel DCX
- DDS-PIL - DirectDraw Surface
- DIB-PIL - Windows Bitmap
- EPS-PIL - Encapsulated Postscript
- FITS-PIL - FITS
- FLI-PIL - Autodesk FLI/FLC Animation
- FPX-PIL - FlashPix
- FTEX-PIL - Texture File Format (IW2:EOC)
- GBR-PIL - GIMP brush file
- GIF-PIL - Static and animated gif (Pillow)
- GRIB-PIL - GRIB
- HDF5-PIL - HDF5
- ICNS-PIL - Mac OS icns resource
- ICO-PIL - Windows Icon
- IM-PIL - IFUNC Image Memory
- IMT-PIL - IM Tools
- IPTC-PIL - IPTC/NAA
- JPEG-PIL - JPEG (ISO 10918)
- JPEG2000-PIL - JPEG 2000 (ISO 15444)
- MCIDAS-PIL - McIdas area file
- MIC-PIL - Microsoft Image Composer
- MPO-PIL - MPO (CIPA DC-007)
- MSP-PIL - Windows Paint
- PCD-PIL - Kodak PhotoCD
- PCX-PIL - Paintbrush
- PIXAR-PIL - PIXAR raster image
- PNG-PIL - Portable network graphics
- PPM-PIL - Pbmplus image
- PSD-PIL - Adobe Photoshop
- SGI-PIL - SGI Image File Format
- SPIDER-PIL - Spider 2D image

- SUN-PIL - Sun Raster File
- TGA-PIL - Targa
- TIFF-PIL - TIFF format (Pillow)
- WMF-PIL - Windows Metafile
- XBM-PIL - X11 Bitmap
- XPM-PIL - X11 Pixel Map
- XVTHUMB-PIL - XV thumbnail image
- SCREENGRAB - Grab screenshots (Windows and OS X only)
- CLIPBOARDGRAB - Grab from clipboard (Windows only)
- BMP-FI - Windows or OS/2 Bitmap
- CUT-FI - Dr. Halo
- DDS-FI - DirectX Surface
- EXR-FI - ILM OpenEXR
- G3-FI - Raw fax format CCITT G.3
- HDR-FI - High Dynamic Range Image
- IFF-FI - IFF Interleaved Bitmap
- J2K-FI - JPEG-2000 codestream
- JNG-FI - JPEG Network Graphics
- JP2-FI - JPEG-2000 File Format
- JPEG-FI - JPEG - JFIF Compliant
- JPEG-XR-FI - JPEG XR image format
- KOALA-FI - C64 Koala Graphics
- PBM-FI - Portable Bitmap (ASCII)
- PBMRAW-FI - Portable Bitmap (RAW)
- PCD-FI - Kodak PhotoCD
- PCX-FI - Zsoft Paintbrush
- PFM-FI - Portable floatmap
- PGM-FI - Portable Greymap (ASCII)
- PGMRAW-FI - Portable Greymap (RAW)
- PICT-FI - Macintosh PICT
- PNG-FI - Portable Network Graphics
- PPM-FI - Portable Pixelmap (ASCII)
- PPMRAW-FI - Portable Pixelmap (RAW)
- PSD-FI - Adobe Photoshop
- RAS-FI - Sun Raster Image
- RAW-FI - RAW camera image

- SGI-FI - SGI Image Format
- TARGA-FI - Truevision Targa
- TIFF-FI - Tagged Image File Format
- WBMP-FI - Wireless Bitmap
- WEBP-FI - Google WebP image format
- XBM-FI - X11 Bitmap Format
- XPM-FI - X11 Pixmap Format
- ICO-FI - Windows icon
- GIF-FI - Static and animated gif (FreeImage)
- BSDF - Format based on the Binary Structured Data Format
- DICOM - Digital Imaging and Communications in Medicine
- NPZ - Numpy's compressed array format
- FEI - FEI-SEM TIFF format
- FITS - Flexible Image Transport System (FITS) format
- ITK - Insight Segmentation and Registration Toolkit (ITK) format
- GDAL - Geospatial Data Abstraction Library
- LYTRO-LFR - Lytro Illum lfr image file
- LYTRO-ILLUM-RAW - Lytro Illum raw image file
- LYTRO-LFP - Lytro F01 lfp image file
- LYTRO-F01-RAW - Lytro F01 raw image file
- SPE - SPE file format
- DUMMY - An example format that does nothing.

2.2.2 Multiple images

- TIFF - TIFF format
- GIF-PIL - Static and animated gif (Pillow)
- ICO-FI - Windows icon
- GIF-FI - Static and animated gif (FreeImage)
- FFMPEG - Many video formats and cameras (via ffmpeg)
- BSDF - Format based on the Binary Structured Data Format
- DICOM - Digital Imaging and Communications in Medicine
- NPZ - Numpy's compressed array format
- SWF - Shockwave flash
- FITS - Flexible Image Transport System (FITS) format
- ITK - Insight Segmentation and Registration Toolkit (ITK) format
- GDAL - Geospatial Data Abstraction Library

- SPE - SPE file format
- DUMMY - An example format that does nothing.

2.2.3 Single volumes

- TIFF - TIFF format
- BSDF - Format based on the Binary Structured Data Format
- DICOM - Digital Imaging and Communications in Medicine
- NPZ - Numpy's compressed array format
- FEI - FEI-SEM TIFF format
- FITS - Flexible Image Transport System (FITS) format
- ITK - Insight Segmentation and Registration Toolkit (ITK) format
- GDAL - Geospatial Data Abstraction Library
- SPE - SPE file format

2.2.4 Multiple volumes

- TIFF - TIFF format
- BSDF - Format based on the Binary Structured Data Format
- DICOM - Digital Imaging and Communications in Medicine
- NPZ - Numpy's compressed array format
- FITS - Flexible Image Transport System (FITS) format
- ITK - Insight Segmentation and Registration Toolkit (ITK) format
- GDAL - Geospatial Data Abstraction Library
- SPE - SPE file format

2.3 Imageio command line scripts

This page lists the command line scripts provided by imageio. To see all options for a script, execute it with the `--help` option, e.g. `imageio_download_bin --help`.

- `imageio_download_bin`: Download binary dependencies for imageio plugins to the users application data directory. This script accepts the parameter `--package-dir` which will download the binaries to the directory where imageio is installed. This option is useful when freezing an application with imageio. It is supported out-of-the-box by PyInstaller version>=3.2.2.
- `imageio_remove_bin`: Remove binary dependencies of imageio plugins from all directories managed by imageio. This script is useful when there is a corrupt binary or when the user prefers the system binary over the binary provided by imageio.

2.4 Imageio environment variables

This page lists the environment variables that imageio uses. You can set these to control some of imageio's behavior. Each operating system has its own way for setting environment variables, but to set a variable for the current Python process use `os.environ['IMAGEIO_VAR_NAME'] = 'value'`.

- `IMAGEIO_NO_INTERNET`: If this value is "1", "yes", or "true" (case insensitive), makes imageio not use the internet connection to retrieve files (like libraries or sample data). Some plugins (e.g. freeimage and ffmpeg) will try to use the system version in this case.
- `IMAGEIO_FFMPEG_EXE`: Set the path to the ffmpeg executable. Set to simply "ffmpeg" to use your system ffmpeg executable.
- `IMAGEIO_FREEIMAGE_LIB`: Set the path to the freeimage library. If not given, will prompt user to download the freeimage library.
- `IMAGEIO_FORMAT_ORDER`: Determine format preference. E.g. setting this to `"TIFF, -FI"` will prefer the FreeImage plugin over the Pillow plugin, but still prefer TIFF over that. Also see the `formats.sort()` method.
- `IMAGEIO_USERDIR`: Set the path to the default user directory. If not given, imageio will try `~` and if that's not available `/var/tmp`.

2.5 Imageio standard images

Imageio provides a number of standard images. These include classic 2D images, as well as animated and volumetric images. To the best of our knowledge, all the listed images are in public domain.

The image names can be loaded by using a special URI, e.g. `imread('imageio:astronaut.png')`. The images are automatically downloaded (and cached in your appdata directory).

- chelsea.bsdf: The chelsea.png in a BSDF file(for testing)
- newtonscradle.gif: Animated GIF of a newton's cradle
- cockatoo.mp4: Video file of a cockatoo
- stent.npz: Volumetric image showing a stented abdominal aorta
- astronaut.png: Image of the astronaut Eileen Collins
- camera.png: Classic grayscale image of a photographer
- checkerboard.png: Black and white image of a chekerboard
- chelsea.png: Image of Stefan's cat
- clock.png: Photo of a clock with motion blur (Stefan van der Walt)
- coffee.png: Image of a cup of coffee (Rachel Michetti)
- coins.png: Image showing greek coins from Pompeii
- horse.png: Image showing the silhouette of a horse (Andreas Preuss)
- hubble_deep_field.png: Photograph taken by Hubble telescope (NASA)
- immunohistochemistry.png: Immunohistochemical (IHC) staining
- moon.png: Image showing a portion of the surface of the moon
- page.png: A scanned page of text

- text.png: A photograph of handdrawn text
- wikkie.png: Image of Almar's cat
- chelsea.zip: The chelsea.png in a zipfile (for testing)

CHAPTER 3

Developer documentation

3.1 Imageio's developer API

This page lists the developer documentation for imageio. Normal users will generally not need this, except perhaps the *Format* class. All these functions and classes are available in the `imageio.core` namespace.

This subpackage provides the core functionality of imageio (everything but the plugins).

Functions: *appdata_dir()*, *asarray()*, *get_platform()*, *get_remote_file()*, *has_module()*, *image_as_uint()*, *load_lib()*, *read_n_bytes()*, *resource_dirs()*, *urlopen()*

Classes: *Array*, *BaseProgressIndicator*, *Dict*, *Format*, *FormatManager*, Image, *InternetNotAllowedError*, *NeedDownloadError*, *Request*, *StdoutProgressIndicator*

`imageio.core.`**`appdata_dir`**(*appname=None*, *roaming=False*)
Get the path to the application directory, where applications are allowed to write user specific files (e.g. configurations). For non-user specific data, consider using common_appdata_dir(). If appname is given, a subdir is appended (and created if necessary). If roaming is True, will prefer a roaming directory (Windows Vista/7).

`imageio.core.`**`asarray`**(*a*)
Pypy-safe version of np.asarray. Pypy's np.asarray consumes a *lot* of memory if the given array is an ndarray subclass. This function does not.

`imageio.core.`**`get_platform`**()
Get a string that specifies the platform more specific than sys.platform does. The result can be: linux32, linux64, win32, win64, osx32, osx64. Other platforms may be added in the future.

`imageio.core.`**`get_remote_file`**(*fname*, *directory=None*, *force_download=False*, *auto=True*)
Get a the filename for the local version of a file from the web

Parameters

fname [str] The relative filename on the remote data repository to download. These correspond to paths on `https://github.com/imageio/imageio-binaries/`.

directory [str | None] The directory where the file will be cached if a download was required to obtain the file. By default, the appdata directory is used. This is also the first directory that is checked for a local version of the file. If the directory does not exist, it will be created.

force_download [bool | str] If True, the file will be downloaded even if a local copy exists (and this copy will be overwritten). Can also be a YYYY-MM-DD date to ensure a file is up-to-date (modified date of a file on disk, if present, is checked).

auto [bool] Whether to auto-download the file if its not present locally. Default True. If False and a download is needed, raises NeedDownloadError.

Returns

fname [str] The path to the file on the local system.

imageio.core.**has_module**(*module_name*)
Check to see if a python module is available.

imageio.core.**image_as_uint**(*im*, *bitdepth=None*)
Convert the given image to uint (default: uint8)

If the dtype already matches the desired format, it is returned as-is. If the image is float, and all values are between 0 and 1, the values are multiplied by np.power(2.0, bitdepth). In all other situations, the values are scaled such that the minimum value becomes 0 and the maximum value becomes np.power(2.0, bitdepth)-1 (255 for 8-bit and 65535 for 16-bit).

imageio.core.**load_lib**(*exact_lib_names*, *lib_names*, *lib_dirs=None*)
Load a dynamic library.

This function first tries to load the library from the given exact names. When that fails, it tries to find the library in common locations. It searches for files that start with one of the names given in lib_names (case insensitive). The search is performed in the given lib_dirs and a set of common library dirs.

Returns `(ctypes_library, library_path)`

imageio.core.**read_n_bytes**(*file*, *n*)
Read n bytes from the given file, or less if the file has less bytes. Returns zero bytes if the file is closed.

imageio.core.**resource_dirs**()
Get a list of directories where imageio resources may be located. The first directory in this list is the "resources" directory in the package itself. The second directory is the appdata directory (~/.imageio on Linux). The list further contains the application directory (for frozen apps), and may include additional directories in the future.

imageio.core.**urlopen**(**args*, ***kwargs*)
Compatibility function for the urlopen function. Raises an RuntimeError if urlopen could not be imported (which can occur in frozen applications.

class imageio.core.**Array**(*array*, *meta=None*)
A subclass of np.ndarray that has a meta attribute. Get the dictionary that contains the meta data using `im.meta`. Convert to a plain numpy array using `np.asarray(im)`.

Attributes

T The transposed array.

base Base object if memory is from some other object.

ctypes An object to simplify the interaction of the array with the ctypes module.

data Python buffer object pointing to the start of the array's data.

dtype Data-type of the array's elements.

flags Information about the memory layout of the array.

flat A 1-D iterator over the array.

imag The imaginary part of the array.

itemsize Length of one array element in bytes.

meta The dict with the meta data of this image.

nbytes Total bytes consumed by the elements of the array.

ndim Number of array dimensions.

real The real part of the array.

shape Tuple of array dimensions.

size Number of elements in the array.

strides Tuple of bytes to step in each dimension when traversing an array.

meta
The dict with the meta data of this image.

class imageio.core.**BaseProgressIndicator**(*name*)
A progress indicator helps display the progres of a task to the user. Progress can be pending, running, finished or failed.

Each task has:

- a name - a short description of what needs to be done.
- an action - the current action in performing the task (e.g. a subtask)
- progress - how far the task is completed
- max - max number of progress units. If 0, the progress is indefinite
- unit - the units in which the progress is counted
- status - 0: pending, 1: in progress, 2: finished, 3: failed

This class defines an abstract interface. Subclasses should implement _start, _stop, _update_progress(progressText), _write(message).

fail(*message=None*)
Stop the progress with a failure, optionally specifying a message.

finish(*message=None*)
Finish the progress, optionally specifying a message. This will not set the progress to the maximum.

increase_progress(*extra_progress*)
Increase the progress by a certain amount.

set_progress(*progress=0*, *force=False*)
Set the current progress. To avoid unnecessary progress updates this will only have a visual effect if the time since the last update is > 0.1 seconds, or if force is True.

start(*action=''*, *unit=''*, *max=0*)
Start the progress. Optionally specify an action, a unit, and a maxium progress value.

status()
Get the status of the progress - 0: pending, 1: in progress, 2: finished, 3: failed

write(*message*)
Write a message during progress (such as a warning).

class imageio.core.**Dict**
A dict in which the keys can be get and set as if they were attributes. Very convenient in combination with autocompletion.

This Dict still behaves as much as possible as a normal dict, and keys can be anything that are otherwise valid keys. However, keys that are not valid identifiers or that are names of the dict class (such as 'items' and 'copy') cannot be get/set as attributes.

class imageio.core.**Format**(*name*, *description*, *extensions=None*, *modes=None*)
Represents an implementation to read/write a particular file format

A format instance is responsible for 1) providing information about a format; 2) determining whether a certain file can be read/written with this format; 3) providing a reader/writer class.

Generally, imageio will select the right format and use that to read/write an image. A format can also be explicitly chosen in all read/write functions. Use `print(format)`, or `help(format_name)` to see its documentation.

To implement a specific format, one should create a subclass of Format and the Format.Reader and Format.Writer classes. see *Creating imageio plugins* for details.

Parameters

name [str] A short name of this format. Users can select a format using its name.

description [str] A one-line description of the format.

extensions [str | list | None] List of filename extensions that this format supports. If a string is passed it should be space or comma separated. The extensions are used in the documentation and to allow users to select a format by file extension. It is not used to determine what format to use for reading/saving a file.

modes [str] A string containing the modes that this format can handle ('iIvV'), "i" for an image, "I" for multiple images, "v" for a volume, "V" for multiple volumes. This attribute is used in the documentation and to select the formats when reading/saving a file.

Attributes

Reader

Writer

description A short description of this format.

doc The documentation for this format (name + description + docstring).

extensions A list of file extensions supported by this plugin.

modes A string specifying the modes that this format can handle.

name The name of this format.

can_read(*request*)
Get whether this format can read data from the specified uri.

can_write(*request*)
Get whether this format can write data to the speciefed uri.

description
A short description of this format.

doc
The documentation for this format (name + description + docstring).

extensions
A list of file extensions supported by this plugin. These are all lowercase with a leading dot.

get_reader(*request*)
Return a reader object that can be used to read data and info from the given file. Users are encouraged to use imageio.get_reader() instead.

get_writer(*request*)
Return a writer object that can be used to write data and info to the given file. Users are encouraged to use imageio.get_writer() instead.

modes
A string specifying the modes that this format can handle.

name
The name of this format.

class imageio.core.**FormatManager**
There is exactly one FormatManager object in imageio: `imageio.formats`. Its purpose it to keep track of the registered formats.

The format manager supports getting a format object using indexing (by format name or extension). When used as an iterator, this object yields all registered format objects.

See also *help()*.

add_format(*format*, *overwrite=False*)
Register a format, so that imageio can use it. If a format with the same name already exists, an error is raised, unless overwrite is True, in which case the current format is replaced.

get_format_names(*self*)
Get the names of all registered formats.

search_read_format(*request*)
Search a format that can read a file according to the given request. Returns None if no appropriate format was found. (used internally)

search_write_format(*request*)
Search a format that can write a file according to the given request. Returns None if no appropriate format was found. (used internally)

show(*self*)
Show a nicely formatted list of available formats

sort(*name1*, *name2*, *name3*, ...)
Sort the formats based on zero or more given names; a format with a name that matches one of the given names will take precedence over other formats. A match means an equal name, or ending with that name (though the former counts higher). Case insensitive.

Format preference will match the order of the given names: using `sort('TIFF', '-FI', '-PIL')` would prefer the FreeImage formats over the Pillow formats, but prefer TIFF even more. Each time this is called, the starting point is the default format order, and calling `sort()` with no arguments will reset the order.

Be aware that using the function can affect the behavior of other code that makes use of imageio.

Also see the `IMAGEIO_FORMAT_ORDER` environment variable.

imageio.core.**Image**
alias of `imageio.core.util.Array`

exception imageio.core.**InternetNotAllowedError**
Plugins that need resources can just use get_remote_file(), but should catch this error and silently ignore it.

exception imageio.core.**NeedDownloadError**
Is raised when a remote file is requested that is not locally available, but which needs to be explicitly downloaded by the user.

class imageio.core.**Request**(*uri*, *mode*, ***kwargs*)
Represents a request for reading or saving an image resource. This object wraps information to that request and acts as an interface for the plugins to several resources; it allows the user to read from filenames, files, http, zipfiles, raw bytes, etc., but offer a simple interface to the plugins via `get_file()` and `get_local_filename()`.

For each read/write operation a single Request instance is used and passed to the can_read/can_write method of a format, and subsequently to the Reader/Writer class. This allows rudimentary passing of information between different formats and between a format and associated reader/writer.

Parameters

uri [{str, bytes, file}] The resource to load the image from.

mode [str] The first character is "r" or "w", indicating a read or write request. The second character is used to indicate the kind of data: "i" for an image, "I" for multiple images, "v" for a volume, "V" for multiple volumes, "?" for don't care.

Attributes

extension The (lowercase) extension of the requested filename.

filename The uri for which reading/saving was requested.

firstbytes The first 256 bytes of the file.

kwargs The dict of keyword arguments supplied by the user.

mode The mode of the request.

extension
The (lowercase) extension of the requested filename. Suffixes in url's are stripped. Can be None if the request is not based on a filename.

filename
The uri for which reading/saving was requested. This can be a filename, an http address, or other resource identifier. Do not rely on the filename to obtain the data, but use `get_file()` or `get_local_filename()` instead.

finish(*self*)
For internal use (called when the context of the reader/writer exits). Finishes this request. Close open files and process results.

firstbytes
The first 256 bytes of the file. These can be used to parse the header to determine the file-format.

get_file(*self*)
Get a file object for the resource associated with this request. If this is a reading request, the file is in read mode, otherwise in write mode. This method is not thread safe. Plugins should not close the file when done.

This is the preferred way to read/write the data. But if a format cannot handle file-like objects, they should use `get_local_filename()`.

get_local_filename(*self*)
If the filename is an existing file on this filesystem, return that. Otherwise a temporary file is created on the local file system which can be used by the format to read from or write to.

get_result(*self*)
For internal use. In some situations a write action can have a result (bytes data). That is obtained with this function.

kwargs
The dict of keyword arguments supplied by the user.

mode
The mode of the request. The first character is "r" or "w", indicating a read or write request. The second character is used to indicate the kind of data: "i" for an image, "I" for multiple images, "v" for a volume, "V" for multiple volumes, "?" for don't care.

class imageio.core.**StdoutProgressIndicator**(*name*)
A progress indicator that shows the progress in stdout. It assumes that the tty can appropriately deal with backspace characters.

3.2 Creating imageio plugins

Imagio is plugin-based. Every supported format is provided with a plugin. You can write your own plugins to make imageio support additional formats. And we would be interested in adding such code to the imageio codebase!

3.2.1 What is a plugin

In imageio, a plugin provides one or more *Format* objects, and corresponding *Reader* and *Writer* classes. Each Format object represents an implementation to read/write a particular file format. Its Reader and Writer classes do the actual reading/saving.

The reader and writer objects have a `request` attribute that can be used to obtain information about the read or write *Request*, such as user-provided keyword arguments, as well get access to the raw image data.

3.2.2 Registering

Strictly speaking a format can be used stand alone. However, to allow imageio to automatically select it for a specific file, the format must be registered using `imageio.formats.add_format()`.

Note that a plugin is not required to be part of the imageio package; as long as a format is registered, imageio can use it. This makes imageio very easy to extend.

3.2.3 What methods to implement

Imageio is designed such that plugins only need to implement a few private methods. The public API is implemented by the base classes. In effect, the public methods can be given a descent docstring which does not have to be repeated at the plugins.

For the Format class, the following needs to be implemented/specified:

- The format needs a short name, a description, and a list of file extensions that are common for the file-format in question. These ase set when instantiation the Format object.
- Use a docstring to provide more detailed information about the format/plugin, such as parameters for reading and saving that the user can supply via keyword arguments.
- Implement `_can_read(request)`, return a bool. See also the *Request* class.
- Implement `_can_write(request)`, dito.

For the Format.Reader class:

- Implement `_open(**kwargs)` to initialize the reader. Deal with the user-provided keyword arguments here.
- Implement `_close()` to clean up.
- Implement `_get_length()` to provide a suitable length based on what the user expects. Can be `inf` for streaming data.
- Implement `_get_data(index)` to return an array and a meta-data dict.
- Implement `_get_meta_data(index)` to return a meta-data dict. If index is None, it should return the 'global' meta-data.

For the Format.Writer class:

- Implement `_open(**kwargs)` to initialize the writer. Deal with the user-provided keyword arguments here.
- Implement `_close()` to clean up.
- Implement `_append_data(im, meta)` to add data (and meta-data).
- Implement `_set_meta_data(meta)` to set the global meta-data.

3.2.4 Example / template plugin

```
# -*- coding: utf-8 -*-
# imageio is distributed under the terms of the (new) BSD License.

""" Example plugin. You can use this as a template for your own plugin.
"""

import numpy as np

from .. import formats
from ..core import Format


class DummyFormat(Format):
    """ The dummy format is an example format that does nothing.
    It will never indicate that it can read or write a file. When
    explicitly asked to read, it will simply read the bytes. When
    explicitly asked to write, it will raise an error.

    This documentation is shown when the user does ``help('thisformat')``.

    Parameters for reading
    ----------------------
    Specify arguments in numpy doc style here.

    Parameters for saving
    ---------------------
    Specify arguments in numpy doc style here.

    """

    def _can_read(self, request):
        # This method is called when the format manager is searching
        # for a format to read a certain image. Return True if this format
        # can do it.
```

(continues on next page)

(continued from previous page)

```
        #
        # The format manager is aware of the extensions and the modes
        # that each format can handle. It will first ask all formats
        # that *seem* to be able to read it whether they can. If none
        # can, it will ask the remaining formats if they can: the
        # extension might be missing, and this allows formats to provide
        # functionality for certain extensions, while giving preference
        # to other plugins.
        #
        # If a format says it can, it should live up to it. The format
        # would ideally check the request.firstbytes and look for a
        # header of some kind.
        #
        # The request object has:
        # request.filename: a representation of the source (only for reporting)
        # request.firstbytes: the first 256 bytes of the file.
        # request.mode[0]: read or write mode
        # request.mode[1]: what kind of data the user expects: one of 'iIvV?'

        if request.mode[1] in (self.modes + "?"):
            if request.extension in self.extensions:
                return True

    def _can_write(self, request):
        # This method is called when the format manager is searching
        # for a format to write a certain image. It will first ask all
        # formats that *seem* to be able to write it whether they can.
        # If none can, it will ask the remaining formats if they can.
        #
        # Return True if the format can do it.

        # In most cases, this code does suffice:
        if request.mode[1] in (self.modes + "?"):
            if request.extension in self.extensions:
                return True

    # -- reader

    class Reader(Format.Reader):
        def _open(self, some_option=False, length=1):
            # Specify kwargs here. Optionally, the user-specified kwargs
            # can also be accessed via the request.kwargs object.
            #
            # The request object provides two ways to get access to the
            # data. Use just one:
            #  - Use request.get_file() for a file object (preferred)
            #  - Use request.get_local_filename() for a file on the system
            self._fp = self.request.get_file()
            self._length = length  # passed as an arg in this case for testing
            self._data = None

        def _close(self):
            # Close the reader.
            # Note that the request object will close self._fp
            pass

        def _get_length(self):
```

(continues on next page)

(continued from previous page)

```
            # Return the number of images. Can be np.inf
            return self._length

        def _get_data(self, index):
            # Return the data and meta data for the given index
            if index >= self._length:
                raise IndexError("Image index %i > %i" % (index, self._length))
            # Read all bytes
            if self._data is None:
                self._data = self._fp.read()
            # Put in a numpy array
            im = np.frombuffer(self._data, "uint8")
            im.shape = len(im), 1
            # Return array and dummy meta data
            return im, {}

        def _get_meta_data(self, index):
            # Get the meta data for the given index. If index is None, it
            # should return the global meta data.
            return {}  # This format does not support meta data

    # -- writer

    class Writer(Format.Writer):
        def _open(self, flags=0):
            # Specify kwargs here. Optionally, the user-specified kwargs
            # can also be accessed via the request.kwargs object.
            #
            # The request object provides two ways to write the data.
            # Use just one:
            #  - Use request.get_file() for a file object (preferred)
            #  - Use request.get_local_filename() for a file on the system
            self._fp = self.request.get_file()

        def _close(self):
            # Close the reader.
            # Note that the request object will close self._fp
            pass

        def _append_data(self, im, meta):
            # Process the given data and meta data.
            raise RuntimeError("The dummy format cannot write image data.")

        def set_meta_data(self, meta):
            # Process the given meta data (global for all images)
            # It is not mandatory to support this.
            raise RuntimeError("The dummy format cannot write meta data.")


# Register. You register an *instance* of a Format class. Here specify:
format = DummyFormat(
    "dummy",  # short name
    "An example format that does nothing.",  # one line descr.
    ".foobar .nonexistentext",  # list of extensions
    "iI",  # modes, characters in iIvV
)
formats.add_format(format)
```

Python Module Index

i

Index