

---

# **imageio Documentation**

*Release 2.2.0dev*

**imageio contributors**

**Sep 12, 2017**



---

## Contents

---

|          |   |           |
|----------|---|-----------|
| <b>1</b> | <b>Getting started</b>                  | <b>3</b>  |
| 1.1      | Installing imageio . . . . .            | 3         |
| 1.2      | Imageio usage examples . . . . .        | 3         |
| 1.3      | Release notes . . . . .                 | 5         |
| <b>2</b> | <b>Reference</b>                        | <b>13</b> |
| 2.1      | Imageio's user API . . . . .            | 13        |
| 2.2      | Imageio formats . . . . .               | 19        |
| 2.3      | Imageio command line scripts . . . . .  | 23        |
| 2.4      | Imageio environment variables . . . . . | 23        |
| 2.5      | Imageio standard images . . . . .       | 24        |
| <b>3</b> | <b>Developer documentation</b>          | <b>25</b> |
| 3.1      | Imageio's developer API . . . . .       | 25        |
| 3.2      | Creating imageio plugins . . . . .      | 30        |
|          | <b>Python Module Index</b>              | <b>35</b> |



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 2.x and 3.x, and is easy to install.

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

Contents:



### Installing imageio

Imageio is written in pure Python, so installation is easy. Imageio works on Python 2.6 and up (including Python 3). 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 download and store in a folder in your application data.

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`

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

### 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.

### Read an image of a cat

Probably the most important thing you ever need.

```
import imageio
```

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

## 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)
```

## 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()))
```

## 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>')
```

## 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.

```
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)
```

## 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 ...



```
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()
```

## 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')
```

## Volume data

```
import imageio
import visvis as vv

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

## Release notes

### Version 2.2.0 (25-05-2017)

- New format for grabbing screenshots (for Windows and OS X).
- New format for grabbing image data from clipboard (Window only).
- Multipage Tiff files can now be read using `volread()` to obtain the image data as one array.
- Updated the ffmpeg executables that imageio provides.
- The ffmpeg format can now also use the ffmpeg exe provided by the ffmpeg conda package (`conda install ffmpeg -c conda-forge`).
- Fixes to ffmpeg format in general.
- Improve docs and rounding in animated GIF duration.
- Fix for setting number of loops in animated GIF.
- Fixes for transparent images in Pillow.

- Fixes for float indexing that is disallowed in new Numpy (Freeimage plugin).
- Fix for using missing `close()` on Pillow images.
- Updated version of tiffle plugin.

## **Version 2.1.2 (02-02-2017)**

A bugfix release:

- Fix animated gif writer that was broken in newer Pillow version.
- FFMPEG plugin improvements: more reliable fps detection, can deal with missing FPS, more reliable subprocess termination,
- Mimread allows a few missing frames to better deal with certain video files.
- Allow question marks in url's.
- Allow Pillow plugin to read remote files by “enabling” `seek()` and `tell()`.
- Use `invoke` to run development tasks instead of custom “make” module.

## **Version 2.1.1 (24-12-2016)**

Minor improvements related to Debian packaging.

## **Version 2.1 (22-12-2016)**

- Standard images now have to be specified using e.g. `imageio.imread('imageio:chelsea.png')` to be more explicit about being a special case and potentially involving a download.
- Improvements and fixes for the ffmpeg plugin (including improved seeking).
- Several tweaks to the tests and setup script to make it pass the Debian build system.

## **Version 2.0 (10-12-2016)**

This release introduces a new plugin based on Pillow, which will take care of the “common formats” like PNG and JPEG, which was previously the role of the FreeImage plugin. The latter is still available but the FreeImage library is no longer distributed by default.

- New Pillow plugin to provide the common formats.
- FreeImage plugin gets lower priority w.r.t. resolving a format.
- No more automatic downloading of libraries and executable (for FreeImage, FFMPEG and AVBIN plugins).
- Pillow plugin comes with a format to read/write animated GIF to supersede the one provided by FreeImage.
- Various improvements/fixes to the ffmpeg plugin.
- Fixes and improvements of the DICOM plugin.
- Better support of exr images via FreeImage (by Joel Nises).
- New FEI format (for images produced by the FEI SEM microscope).

## Version 1.6 (19-09-2016)

- Got rid of Lena image because it can be regarded offensive and is not (explicitly) publicly licensed.
- Fix issue with ffmpeg reader being slow on particular systems (#152).
- Tiff plugin updated.
- Add Tiff resolution support (Antony Lee).
- Support for 16bit PNG's (#150, by OrganicIrradiation).
- Fixes to ffmpeg plugin (#149, #145, #129).
- Fix in using IMAGEIO\_FREEIMAGE\_LIB (#141, by Radomirs Cirskis)
- Better ffmpeg verbosity and exe detection ( #138, #139, by Tim D. Smith).

## Version 1.5 (31-01-2016)

- Freeimage conda package (in main channel) is updated and works on all major OS's.
- Conda install imageio!
- Fix bug where the ffmpeg plugin fails on certain video files (#131).
- Fix how dicom uses dcmk for JPEG compressed files.

## Version 1.4 (18-11-2015)

- Various improvements to the ffmpeg plugin.
- New tiffle plugin that should support most scientific formats.
- New simpleITK wrapper plugin.
- New gdal plugin.
- Freeimage plugin can load freeimage lib provided by conda.
- Dicom plugin improved handling of compressed files.
- Most plugins adopt lazy loading to keep imageio lean, fast, and scalable.
- We now build wheels for Pypi.
- Travis also tests Python 3.5.

## Version 1.3 (02-07-2015)

This release features several fixes and small improvements, especially to the ffmpeg plugin.

- Fix 'FrameTime' in first frame of GIF image (#90)
- Fix that writing video could freeze on Windows (#84)
- Fix that ffmpeg process was sometimes not closed correctly (#79)
- Also protect user from clogging the machine for mvolread (#89)
- Better support for platforms other than Win/Linux/OSX (#87 )
- Support for reading from webcam on OSX (#83, #85)

- Support for dpx via the ffmpeg plugin (#81)
- Support for wmv via the ffmpeg plugin (#83)
- The ffmpeg plugin allows specifying pixelformat. The new default is more widely supported (#83)
- Allow passing additional arguments to ffmpeg command (#83)
- Quality of ffmpeg output now set via quality param instead of bitrate (#83)
- Imageio now has a few (documented) environment variables to specify the locations of plugin libraries/exes (thus preventing them from being automatically downloaded).

## Version 1.2 (23-02-2015)

Basically a hotfix release. But some new features were introduced.

- Fixed that pip-installing would put README.md and other files in sys.prefix.
- The used ffmpeg exe can be overridden with an environment variable 'IMAGEIO\_FFMPEG\_EXE'.
- Relative paths work again.
- FFMPEG plugin moved to correct timeframe when seeking (thanks Zulko)

## Version 1.1 (04-02-2015)

Imageio is now a dependency of [Moviepy](#), which exposed a few issues to fix. Imageio is now also available as a Debian package (thanks Ghislain!). Further, we tweaked our function names to be cleared and more consistent (the old names still work).

- All `Xsave()` functions are renamed to `Xwrite()`. Also `read()` and `save()` are now `get_reader()` and `get_writer()`. The old names are available as aliases (and will be for the foreseeable future) for backward compatibility.
- Protect user from bringing computer in swap-mode by doing e.g. `mimread('hunger_games.avi')`.
- Continuous integration for Windows via Appveyor.
- All imports are relative, so imageio can be used as a subpackage in a larger project.
- FFMPEG is the default plugin for reading video (since AVBIN has issues).
- Better handling on NaN and Inf when converting to uint8.
- Provide dist packages that include freeimage lib and a few example images.
- Several changes to ease building into Debian package.
- Fixed segfault when saving gif (thanks levskaya, <https://github.com/imageio/imageio/pull/53>).
- Don't fail when userdir is not writable.
- Gif plugin writer has fps param for consistency with avi/mp4 etc.

## Version 1.0 (13-11-2014)

In this release we did a lot of work to push imageio to a new level. The code is now properly tested, and we have several more formats.

The big changes:

- Many unit tests were written to cover over 95% of the code base. (the core of imageio has 100% coverage).
- Setup continuous integration (CI) using Travis.
- Imageio now follows PEP8 style guides (and this is tested with CI).
- Refactoring of the code base. Resulting in a cleaner namespace.
- Many improvements to the documentation.

Plugins:

- The FFMPEG format is now well supported. Binaries are provided.
- New AVBIN format for more efficient reading of video files.
- New NPZ format that can store (a series of) arbitrarily shaped numpy arrays.
- New SWF format (shockwave flash) for lossless animated images.
- Improvements to the GIF format. The GIF and ANIGIF formats are now merged.

Further:

- New simple website to act as a front page (<http://imageio.github.io>).
- Compatibility with Pypy.
- We provide a range of *standard images* that are automatically downloaded.
- Binaries (libs and executables) that plugins of imageio uses are now downloaded at runtime, not at build/install time. This simplifies things a lot.
- freeimage plugin now fully functional on pypy
- Added utilities for developers (run `python make` from the repo root).
- PNG, JPEG, BMP,GIF and other plugins can now handle float data (pixel values are assumed to be between 0 and 1).
- Imageio now expand the user dir when filename start with '~/'.
- Many improvements and fixes overall.

### Version 0.5.1 (23-06-2014)

- DICOM reader closes file after reading pixel data (avoid too-many-open-files error)
- Support for video data (import and export) via ffmpeg
- Read images from usb camera via ffmpeg (experimental)

### Version 0.4.1 (26-10-2013)

- We moved to github!
- Raise error if URI could not be understood.
- Small improvement for better error reporting.
- Fixes in mvolread and DICOM plugin

## Version 0.4 (27-03-2013)

Some more thorough testing resulted in several fixes and improvements over the last release.

- Fixes to reading of meta data in freeimage plugin which could cause errors when reading a file.
- Support for reading 4 bpp images.
- The color table for index images is now applied to yield an RGBA image.
- Basic support for Pypy.
- Better `__repr__` for the Image class.

## Version 0.3.2

- Fix in dicom reader (RescaleSlope and RescaleIntercept were not found)
- Fixed that progress indicator made things slow

## Version 0.3.1

- Fix installation/distribution issue.

## Version 0.3.0

This was a long haul. Implemented several plugins for animation and volumetric data to give an idea of what sort of API's work and which do not.

- Refactored for more conventional package layout (but importing without installing still supported)
- Put Reader and Writer classes in the namespace of the format. This makes a format a unified whole, and gets rid of the `_get_reader_class` and `_get_write_class` methods (at the cost of some extra indentation).
- Refactored Reader and Writer classes to come up with a better API for both users as plugins.
- The Request class acts as a smart bridging object. Therefore all plugins can now read from a zipfile, http/ftp, and bytes. And they don't have to do a thing.
- Implemented specific BMP, JPEG, PNG, GIF, ICON formats.
- Implemented animated gif plugin (based on freeimage).
- Implemented standalone DICOM plugin.

## Version 0.2.3

- Fixed issue 2 (fail at instal, introduced when implementing freezing)

## Version 0.2.2

- Improved documentation.
- Worked on distribution.
- Freezing should work now.

### Version 0.2.1

- Introduction of the imageio.help function.
- Wrote a lot of documentation.
- Added example (dummy) plugin.

### Version 0.2

- New plugin system implemented after discussions in group.
- Access to format information.

### Version 0.1

- First version with a preliminary plugin system.





## 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.

---

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.

---

`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` (*self*)

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, bytes, file}

The resource to load the image from, e.g. a filename, 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, file}

The resource to write the image to, e.g. a filename 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=True, **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, bytes, file}

The resource to load the images from, e.g. a filename, 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

If True (default), this function will raise an error if the resulting list of images consumes over 256 MB of memory. This is to protect the system using so much memory that it needs to resort to swapping, and thereby stall the computer. E.g. `mimread('hunger_games.avi')`.

**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, file}

The resource to write the images to, e.g. a filename 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, bytes, file}

The resource to load the volume from, e.g. a filename, 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, file}

The resource to write the image to, e.g. a filename 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=True, **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, bytes, file}

The resource to load the volumes from, e.g. a filename, 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

If True (default), this function will raise an error if the resulting list of images consumes over 1 GB of memory. This is to protect the system using so much memory that it needs to resort to swapping, and thereby stall the computer. E.g. `mimread('hunger_games.avi')`.

**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, file}

The resource to write the volumes to, e.g. a filename or file object, see the docs for more info.

`ims` : sequence of numpy arrays

The image data. Each array must be  $N \times M \times L$  (or  $N \times M \times L \times K$  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, bytes, file}

The resource to load the image from, e.g. a filename, 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, file}

The resource to write the image to, e.g. a filename 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).

**close()**

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.

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 attribute 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.

**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).

**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** ()

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.

## 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.

### 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)
- 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
- DUMMY - An example format that does nothing.

## 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)
- AVBIN - Many video formats (via AvBin, i.e. libav library)
- 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
- DUMMY - An example format that does nothing.

## Single volumes

- TIFF - TIFF 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

## Multiple volumes

- TIFF - TIFF 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

## 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  $\geq 3.2.2$ .
- `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.

## 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. If not given, will prompt the user to download the ffmpeg exe that imageio provides.
- `IMAGEIO_AVBIN_LIB`: Set the path to the avbin library. If not given, will prompt the user to download the avbin library that imageio provides.
- `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`.

## 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).

- `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 checkerboard
- `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)

## 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:** `BaseProgressIndicator`, `CannotReadFrameError`, `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.BaseProgressIndicator` (*name*)

A progress indicator helps display the progress 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 maximum 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).

**exception** `imageio.core.CannotReadFrameError`

Exception to be used by plugins to indicate that a frame could not be read, even though it should be a valid index. The length could be `inf`, or e.g. video sometimes reports a wrong length.

**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.

**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 specified 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 is 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** ()

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()**

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.

**class imageio.core.Image(*array*, *meta=None*)**

A subclass of `np.ndarray` that has a `meta` attribute. Following scikit-image, we leave this as a normal numpy array as much as we can.

**meta**

The dict with the meta data of this image.

**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.

**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()**

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()`

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 do not need to 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()`

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()`

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.

## Creating imageio plugins

Imageio 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!

### 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.

### 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.

## 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 are set when instantiating 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)`, ditto.

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.

If the plugin requires a binary download from the `imageio-binaries` repository, implement the `download` method (see e.g. the `ffmpeg` plugin). Make sure that the download directory base name matches the plugin name. Otherwise, the download and removal command line scripts (see `__main__.py`) might not work.

## Example / template plugin

```

1 # -*- coding: utf-8 -*-
2 # Copyright (c) 2015, imageio contributors
3 # imageio is distributed under the terms of the (new) BSD License.
4
5 """ Example plugin. You can use this as a template for your own plugin.
6 """
7
8 from __future__ import absolute_import, print_function, division
9
10 import numpy as np
11
12 from .. import formats
13 from ..core import Format
14

```

```

15
16 class DummyFormat (Format):
17     """ The dummy format is an example format that does nothing.
18     It will never indicate that it can read or write a file. When
19     explicitly asked to read, it will simply read the bytes. When
20     explicitly asked to write, it will raise an error.
21
22     This documentation is shown when the user does ``help('thisformat')``.
23
24     Parameters for reading
25     -----
26     Specify arguments in numpy doc style here.
27
28     Parameters for saving
29     -----
30     Specify arguments in numpy doc style here.
31
32     """
33
34     def _can_read(self, request):
35         # This method is called when the format manager is searching
36         # for a format to read a certain image. Return True if this format
37         # can do it.
38         #
39         # The format manager is aware of the extensions and the modes
40         # that each format can handle. It will first ask all formats
41         # that *seem* to be able to read it whether they can. If none
42         # can, it will ask the remaining formats if they can: the
43         # extension might be missing, and this allows formats to provide
44         # functionality for certain extensions, while giving preference
45         # to other plugins.
46         #
47         # If a format says it can, it should live up to it. The format
48         # would ideally check the request.firstbytes and look for a
49         # header of some kind.
50         #
51         # The request object has:
52         # request.filename: a representation of the source (only for reporting)
53         # request.firstbytes: the first 256 bytes of the file.
54         # request.mode[0]: read or write mode
55         # request.mode[1]: what kind of data the user expects: one of 'iIvV?'
56
57         if request.mode[1] in (self.modes + '?'):
58             if request.filename.lower().endswith(self.extensions):
59                 return True
60
61     def _can_write(self, request):
62         # This method is called when the format manager is searching
63         # for a format to write a certain image. It will first ask all
64         # formats that *seem* to be able to write it whether they can.
65         # If none can, it will ask the remaining formats if they can.
66         #
67         # Return True if the format can do it.
68
69         # In most cases, this code does suffice:
70         if request.mode[1] in (self.modes + '?'):
71             if request.filename.lower().endswith(self.extensions):
72                 return True

```

```

73
74 # -- reader
75
76 class Reader(Format.Reader):
77
78     def _open(self, some_option=False, length=1):
79         # Specify kwargs here. Optionally, the user-specified kwargs
80         # can also be accessed via the request.kwargs object.
81         #
82         # The request object provides two ways to get access to the
83         # data. Use just one:
84         # - Use request.get_file() for a file object (preferred)
85         # - Use request.get_local_filename() for a file on the system
86         self._fp = self.request.get_file()
87         self._length = length # passed as an arg in this case for testing
88         self._data = None
89
90     def _close(self):
91         # Close the reader.
92         # Note that the request object will close self._fp
93         pass
94
95     def _get_length(self):
96         # Return the number of images. Can be np.inf
97         return self._length
98
99     def _get_data(self, index):
100        # Return the data and meta data for the given index
101        if index >= self._length:
102            raise IndexError('Image index %i > %i' % (index, self._length))
103        # Read all bytes
104        if self._data is None:
105            self._data = self._fp.read()
106        # Put in a numpy array
107        im = np.frombuffer(self._data, 'uint8')
108        im.shape = len(im), 1
109        # Return array and dummy meta data
110        return im, {}
111
112     def _get_meta_data(self, index):
113        # Get the meta data for the given index. If index is None, it
114        # should return the global meta data.
115        return {} # This format does not support meta data
116
117 # -- writer
118
119 class Writer(Format.Writer):
120
121     def _open(self, flags=0):
122         # Specify kwargs here. Optionally, the user-specified kwargs
123         # can also be accessed via the request.kwargs object.
124         #
125         # The request object provides two ways to write the data.
126         # Use just one:
127         # - Use request.get_file() for a file object (preferred)
128         # - Use request.get_local_filename() for a file on the system
129         self._fp = self.request.get_file()
130

```

```
131     def _close(self):
132         # Close the reader.
133         # Note that the request object will close self._fp
134         pass
135
136     def _append_data(self, im, meta):
137         # Process the given data and meta data.
138         raise RuntimeError('The dummy format cannot write image data.')
139
140     def set_meta_data(self, meta):
141         # Process the given meta data (global for all images)
142         # It is not mandatory to support this.
143         raise RuntimeError('The dummy format cannot write meta data.')
144
145
146 # Register. You register an *instance* of a Format class. Here specify:
147 format = DummyFormat('dummy', # short name
148                     'An example format that does nothing.', # one line descr.
149                     '.foobar .nonexistenttext', # list of extensions
150                     'iI' # modes, characters in iIvV
151                     )
152 formats.add_format(format)
```

i

imageio, ??  
imageio.core, 25  
imageio.core.functions, 13  
imageio.plugins, 30





**A**

add\_format() (imageio.core.FormatManager method), 28  
 appdata\_dir() (in module imageio.core), 25  
 append\_data() (imageio.core.format.Writer method), 19  
 asarray() (in module imageio.core), 25

**B**

BaseProgressIndicator (class in imageio.core), 26

**C**

can\_read() (imageio.core.Format method), 28  
 can\_write() (imageio.core.Format method), 28  
 CannotReadFrameError, 27  
 close() (imageio.core.format.Reader method), 18  
 close() (imageio.core.format.Writer method), 19  
 closed (imageio.core.format.Reader attribute), 18  
 closed (imageio.core.format.Writer attribute), 19

**D**

description (imageio.core.Format attribute), 28  
 Dict (class in imageio.core), 27  
 doc (imageio.core.Format attribute), 28

**E**

extensions (imageio.core.Format attribute), 28

**F**

fail() (imageio.core.BaseProgressIndicator method), 27  
 filename (imageio.core.Request attribute), 29  
 finish() (imageio.core.BaseProgressIndicator method), 27  
 finish() (imageio.core.Request method), 29  
 firstbytes (imageio.core.Request attribute), 29  
 Format (class in imageio.core), 27  
 format (imageio.core.format.Reader attribute), 18  
 format (imageio.core.format.Writer attribute), 19  
 FormatManager (class in imageio.core), 28

**G**

get\_data() (imageio.core.format.Reader method), 18

get\_file() (imageio.core.Request method), 29  
 get\_format\_names() (imageio.core.FormatManager method), 28  
 get\_length() (imageio.core.format.Reader method), 18  
 get\_local\_filename() (imageio.core.Request method), 30  
 get\_meta\_data() (imageio.core.format.Reader method), 18  
 get\_next\_data() (imageio.core.format.Reader method), 18  
 get\_platform() (in module imageio.core), 25  
 get\_reader() (imageio.core.Format method), 28  
 get\_reader() (in module imageio), 17  
 get\_remote\_file() (in module imageio.core), 25  
 get\_result() (imageio.core.Request method), 30  
 get\_writer() (imageio.core.Format method), 28  
 get\_writer() (in module imageio), 17

**H**

has\_module() (in module imageio.core), 26  
 help() (in module imageio), 14

**I**

Image (class in imageio.core), 29  
 image\_as\_uint() (in module imageio.core), 26  
 imageio (module), 1  
 imageio.core (module), 25  
 imageio.core.functions (module), 13  
 imageio.plugins (module), 30  
 imread() (in module imageio), 14  
 imwrite() (in module imageio), 14  
 increase\_progress() (imageio.core.BaseProgressIndicator method), 27  
 InternetNotAllowedError, 29  
 iter\_data() (imageio.core.format.Reader method), 18

**K**

kwargs (imageio.core.Request attribute), 30

**L**

load\_lib() (in module imageio.core), 26

## M

meta (imageio.core.Image attribute), 29  
mimread() (in module imageio), 15  
mimwrite() (in module imageio), 15  
mode (imageio.core.Request attribute), 30  
modes (imageio.core.Format attribute), 28  
mvolread() (in module imageio), 16  
mvolwrite() (in module imageio), 17

## N

name (imageio.core.Format attribute), 28  
NeedDownloadError, 29

## R

read\_n\_bytes() (in module imageio.core), 26  
Reader (class in imageio.core.format), 18  
Request (class in imageio.core), 29  
request (imageio.core.format.Reader attribute), 18  
request (imageio.core.format.Writer attribute), 19  
resource\_dirs() (in module imageio.core), 26

## S

search\_read\_format() (imageio.core.FormatManager  
method), 28  
search\_write\_format() (imageio.core.FormatManager  
method), 28  
set\_meta\_data() (imageio.core.format.Writer method), 19  
set\_progress() (imageio.core.BaseProgressIndicator  
method), 27  
show() (imageio.core.FormatManager method), 28  
show\_formats() (in module imageio), 14  
sort() (imageio.core.FormatManager method), 29  
start() (imageio.core.BaseProgressIndicator method), 27  
status() (imageio.core.BaseProgressIndicator method), 27  
StdoutProgressIndicator (class in imageio.core), 30

## U

urlopen() (in module imageio.core), 26

## V

volread() (in module imageio), 15  
volwrite() (in module imageio), 16

## W

write() (imageio.core.BaseProgressIndicator method), 27  
Writer (class in imageio.core.format), 19