imageio Documentation

Release 2.22.2

imageio contributors

Oct 16, 2022

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: https://imageio.readthedocs.io/

Contents:

CHAPTER

ONE

USER GUIDE

1.1 Getting Started

1.1.1 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*.

1.1.2 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.

Read an image of a cat

Probably the most important thing you'll ever need.

```
import imageio.v3 as iio

im = iio.imread('imageio:chelsea.png')
print(im.shape)  # (300, 451, 3)
```

If the image is a GIF:

```
import imageio.v3 as iio

# index=None means: read all images in the file and stack along first axis
frames = iio.imread("imageio:newtonscradle.gif", index=None)
# ndarray with (num_frames, height, width, channel)
print(frames.shape)  # (36, 150, 200, 3)
```

Read from fancy sources

Imageio can read from filenames, file objects.

```
import imageio.v3 as iio
import io


# from HTTPS
web_image = "https://upload.wikimedia.org/wikipedia/commons/d/d3/Newtons_cradle_
↪animation_book_2.gif"
frames = iio.imread(web_image, index=None)

# from bytes
bytes_image = iio.imwrite("<bytes>", frames, extension=".gif")
frames = iio.imread(bytes_image, index=None)

# from byte streams
byte_stream = io.BytesIO(bytes_image)
frames = iio.imread(byte_stream, index=None)

# from file objects
class MyFileObject:
    def read(size:int=-1):
        return bytes_image

    def close():
        return  # nothing to do

frames = iio.imread(MyFileObject())
```

Read all Images in a Folder

```
import imageio.v3 as iio
from pathlib import Path

images = list()
for file in Path("path/to/folder").iterdir():
    if not file.is_file():
        continue

    images.append(iio.imread(file))
```

Note, however, that `Path().iterdir()` does not guarantees the order in which files are read.

Grab screenshot or image from the clipboard

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

```
import imageio.v3 as iio

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

Grab frames from your webcam

Note: For this to work, you need to install the ffmpeg backend:

```
pip install imageio[ffmpeg]
```

```
import imageio.v3 as iio
import numpy as np

for idx, frame in enumerate(iio.imiter("<video0>")):
    print(f"Frame {idx}: avg. color {np.sum(frame, axis=-1)}")
```

Note: You can replace the zero with another index in case you have multiple devices attached.

Convert a short movie to grayscale

Note: For this to work, you need to install the ffmpeg backend:

```
pip install imageio[ffmpeg]
```

```
import imageio as iio
import numpy as np

# read the video (it fits into memory)
# Note: this will open the image twice. Check the docs (advanced usage) if
# this is an issue for your use-case
metadata = iio.immeta("imageio:cockatoo.mp4", exclude_applied=False)
frames = iio.imread("imageio:cockatoo.mp4", index=None)

# manually convert the video
gray_frames = np.dot(frames, [0.2989, 0.5870, 0.1140])
gray_frames = np.round(gray_frames).astype(np.uint8)
gray_frames_as_rgb = np.stack([gray_frames] * 3, axis=-1)

# write the video
iio.imwrite("cockatoo_gray.mp4", gray_frames_as_rgb, fps=metadata["fps"])
```

Read or iterate frames in a video

Note: For this to work, you need to install the pyav backend:

```
pip install av
```

```
import imageio.v3 as iio

# read a single frame
frame = iio.imread(
    "imageio:cockatoo.mp4",
    index=42,
    plugin="pyav",
)

# bulk read all frames
# Warning: large videos will consume a lot of memory (RAM)
frames = iio.imread("imageio:cockatoo.mp4", plugin="pyav")

# iterate over large videos
for frame in iio.imiter("imageio:cockatoo.mp4", plugin="pyav"):
    print(frame.shape, frame.dtype)
```

Re-encode a (large) video

Note: For this to work, you need to install the pyav backend:

```
pip install av
```

```
import imageio.v3 as iio

# assume this is too large to keep all frames in memory
source = "imageio:cockatoo.mp4"
dest = "reencoded_cockatoo.mkv"

fps = iio.immeta(source, plugin="pyav")["fps"]

with iio.imopen(dest, "w", plugin="pyav") as out_file:
    out_file.init_video_stream("vp9", fps=fps)

    for frame in iio.imiter(source, plugin="pyav"):
        out_file.write_frame(frame)
```

Read medical data (DICOM)

```
import imageio.v3 as iio
dirname = 'path/to/dicom/files'

# Read multiple images of different shape
ims = [img for img in iio.imiter(dirname, plugin='DICOM')]
# Read as volume
vol = iio.imread(dirname, plugin='DICOM')
# Read multiple volumes of different shape
vols = [img for img in iio.imiter(dirname, plugin='DICOM')]
```

Volume data

```
import imageio.v3 as iio
import visvis as vv

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

Writing videos with FFMPEG and vaapi

Using vaapi can help free up CPU time on your device 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.v2 as iio
import numpy as np

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

w = iio.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 achieved 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'`

Writing to Bytes (Encoding)

You can convert ndimages into byte strings. For this, you have to hint the desired extension (using `extension=`), as a byte string doesn't specify any information about the format or color space to use. Note that, if the backend supports writing to file-like objects, the entire process will happen without touching your file-system.

```
import imageio.v3 as iio

# load an example image
img = iio.imread('imageio:astronaut.png')

# png-encoded bytes string
png_encoded = iio.imwrite("<bytes>", img, extension=".png")

# jpg-encoded bytes string
jpg_encoded = iio.imwrite("<bytes>", img, extension=".jpeg")

# RGBA bytes string
img = iio.imread('imageio:astronaut.png', mode="RGBA")
png_encoded = iio.imwrite("<bytes>", img, extension=".png")
```

Writing to BytesIO

Similar to writing to byte strings, you can also write to BytesIO directly.

```
import imageio.v3 as iio
import io

# load an example image
img = iio.imread('imageio:astronaut.png')

# write as PNG
output = io.BytesIO()
iio.imwrite(output, img, plugin="pillow", extension=".png")

# write as JPG
output = io.BytesIO()
iio.imwrite(output, img, plugin="pillow", extension=".jpeg")
```

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.v3 as iio
import matplotlib.pyplot as plt
import numpy as np

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

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

frames = np.stack(
    [iio.imread("{i}.jpg") for i in range(n)],
    axis=0
)

iio.imwrite(gif_path, frames, mode="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.v3 as iio
import matplotlib.pyplot as plt
```

(continues on next page)

(continued from previous page)

```
import numpy as np
from pygifsicle import optimize

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

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

frames = np.stack(
    [iio.imread("{i}.jpg") for i in range(n)],
    axis=0
)

iio.imwrite(gif_path, frames, mode="I")

optimize(gif_path)
```

Reading Images from ZIP archives

Note: In the future, this syntax will change to better match the URI standard by using fragments. The updated syntax will be "Path/to/file.zip#path/inside/zip/to/image.png".

```
import imageio.v3 as iio

image = iio.imread("Path/to/file.zip/path/inside/zip/to/image.png")
```

Reading Multiple Files from a ZIP archive

Assuming there are only image files in the ZIP archive you can iterate over them with a simple script like the one below.

```
import os
from zipfile import ZipFile
import imageio.v3 as iio

images = list()
with ZipFile("imageio.zip") as zf:
    for name in zf.namelist():
        im = iio.imread(name)
        images.append(im)
```

1.1.3 ImageResources

Reading images isn't always limited to simply loading a file from a local disk. Maybe you are writing a web application and your want to read images from HTTP, or your images are already in memory as a BytesIO object. Maybe you are doing machine learning and decided that it is smart to compress your images inside a ZIP file to reduce the IO bottleneck. All these are examples of different places where image data is stored (aka. resources).

ImageIO supports reading (and writing where applicable) for all of the above examples and more. To stay organized, we group all these sources/resources together and call them `ImageResource`. Often ImageResources are expressed as URIs though sometimes (e.g., in the case of byte streams) they can be python objects, too.

Here, you can find the documentation on what kind of ImageResources ImageIO currently supports together with documentation and an example on how to read/write them.

Files

Arguably the most common type of resource. You specify it using the path to the file, e.g.

```
img = iio.imread("path/to/my/image.jpg")  # relative path
img = iio.imread("/path/to/my/image.jpg")  # absolute path on Linux
img = iio.imread("C:\\path\\to\\my\\image.jpg")  # absolute path on Windows
```

Notice that this is a convenience shorthand (since it is so common). Alternatively, you can use the full URI to the resource on your disk

```
img = iio.imread("file://path/to/my/image.jpg")
img = iio.imread("file:///path/to/my/image.jpg")
img = iio.imread("file://C:\\path\\to\\my\\image.jpg")
```

Byte Streams

ImageIO can directly handle (binary) file objects. This includes `BytesIO` objects (and subclasses thereof) as well as generic objects that implement `close` and a `read` and/or `write` function. Simply pass them into ImageIO the same way you would pass a URI:

```
file_handle = open("some/image.jpg", "rb")
img = iio.imread(file_handle)
```

Standard Images

Standard images are a curated dataset of pictures in different formats that you can use to test your pipelines. You can access them via the `imageio` scheme

```
img = iio.imread("imageio://chelsea.png")
```

A list of all currently available standard images can be found in the section on *Standard Images*.

Web Servers (http/https)

Note: This is primarily intended for publically available ImageResources. If your server requires authentication, you will have to download the ImageResource yourself before handing it over to ImageIO.

Reading http and https is provided directly by ImageIO. This means that ImageIO takes care of requesting and (if necessary) downloading the image and then hands the image to a compatible backend to do the reading itself. It works with any backend. If the backend supports file objects directly, this processes will happen purely in memory.

You can read from public web servers using a URL string

```
img = iio.imread("https://my-domain.com/path/to/some/image.gif")
```

File Servers (ftp/ftps)

Note: This is primarily intended for publically available ImageResources. If your server requires authentication, you will have to download the ImageResource yourself before handing it over to ImageIO.

Reading ftp and ftps is provided directly by ImageIO following the same logic as reading from web servers:

```
img = iio.imread("ftp://my-domain.com/path/to/some/image.gif")
```

Webcam

Note: To access your webcam you will need to have the ffmpeg backend installed:

```
pip install imageio[ffmpeg]
```

With ImageIO you can directly read images (frame-by-frame) from a webcam that is connected to the computer. To do this, you can use the special target:

```
img = iio.imread("<video0>")
```

If you have multiple video sources, you can replace the `0` with the respective number, e.g:

```
img = iio.imread("<video2>")
```

If you need many frames, e.g., because you want to process a stream, it is often much more performant to open the webcam once, keep it open, and read frames on-demand. You can find an example of this in the *list of examples*.

Screenshots

Note: Taking screenshots are only supported on Windows and Mac.

ImageIO gives you basic support for taking screenshots via the special target `screen`:

```
img = iio.imread("<screen>")
```

Clipboard

Note: reading from clipboard is only supported on Windows.

ImageIO gives you basic support for reading from your main clipboard via the special target `clipboard`:

```
img = iio.imread("<clipboard>")
```

ZIP Archives

You can directly read ImageResources from within ZIP archives without extracting them. For this purpose ZIP archives are treated as normal folders; however, nested zip archives are not supported:

```
img = iio.imread("path/to/file.zip/abspath/inside/zipfile/to/image.png")
```

Note that in a future version of ImageIO the syntax for reading ZIP archives will be updated to use fragments, i.e., the path inside the zip file will become a URI fragment.

1.1.4 Bird's eye view on ImageIO

The aim of this page is to give you a high level overview of how ImageIO works under the hood. We think this is useful for two reasons:

1. If something doesn't work as it should, you need to know where to search for causes. The overview on this page aims to help you in this regard by giving you an idea of how things work, and - hence - where things may go sideways.
2. If you do find a bug and decide to report it, this page helps us establish some joint vocabulary so that we can quickly get onto the same page, figure out what's broken, and how to fix it.

Terminology

You can think of ImageIO in three parts that work in sequence in order to load an image.

ImageIO Core
: The user-facing APIs (legacy + v3) and a plugin manager. You send requests to iio.core and it uses a set of plugins (see below) to figure out which backend (see below) to use to fulfill your request. It does so by (intelligently) searching a matching plugin, or by sending the request to the plugin you specified explicitly.

Plugin
: A backend-facing adapter/wrapper that responds to a request from iio.core. It can convert a request that comes from iio.core into a sequence of instructions for the backend that fullfill the request (eg., read/write/iter). A plugin is also aware if its backend is or isn't installed and handles this case appropriately, e.g., it deactivates itself if the backend isn't present.

Backend Library
: A (potentially 3d party) library that can read and/or write images or image-like objects (videos, etc.). Every backend is optional, so it is up to you to decide which backends to install depending on your needs. Examples for backends are pillow, tifffile, or ffmpeg.

ImageResource
: A blob of data that contains image data. Typically, this is a file on your drive that is read by ImageIO. However, other sources such as HTTP or file objects are possible, too.

Issues

In this repo, we maintain ImageIO Core as well as all the plugins. If you find a bug or have a problem with either the core or any of the plugins, please open a new issue here.

If you find a bug or have problems with a specific backend, e.g. reading is very slow, please file an issue upstream (the place where the backend is maintained). When in doubt, you can always ask us, and we will redirect you appropriately.

New Plugins

If you end up writing a new plugin, we very much welcome you to contribute it back to us so that we can offer as expansive a list of backends and supported formats as possible. In return, we help you maintain the plugin - note: a plugin is typically different from the backend - and make sure that changes to ImageIO don't break your plugin. This we can only guarantee if it is part of the codebase, because we can (a) write unit tests for it and (b) update your plugin to take advantage of new features. That said, we generally try to be backward-compatible whenever possible.

The backend itself lives elsewhere, usually in a different repository. Not vendoring backends and storing a copy here keeps things lightweight yet powerful. We can, first of all, directly access any new updates or features that a backend may introduce. Second, we avoid forcing users that won't use a specific backend to go through its, potentially complicated, installation process.

1.2 Other Usage

1.2.1 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
- bricks.jpg: A (repeatable) texture of stone bricks
- meadow_cube.jpg: A cubemap image of a meadow, e.g. to render a skybox.
- wood.jpg: A (repeatable) texture of wooden planks

- 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: A 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)

1.2.2 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.

1.2.3 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_REQUEST_TIMEOUT`: Set the timeout of http/ftp request in seconds. If not set, this defaults to 5 seconds.
- `IMAGEIO_USERDIR`: Set the path to the default user directory. If not given, imageio will try ~ and if that's not available `/var/tmp`.

1.2.4 Transitioning from Scipy's imread

Scipy has depreciated 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 checking out some *examples* to get a feel of ImageIO.

ImageIO makes use of a variety of backends to support reading images (and volumes/movies) from many different formats. One of these backends is Pillow, which is the same library as used by Scipy's `imread`. This means that all image formats that were supported by scipy are supported by ImageIO. At the same time, Pillow is not the only backend of ImageIO, and those other backends give you give you access to additional formats. To manage this, ImageIO automatically selects the right backend to use based on the source to read (which you can of course control manually, should you wish to do so).

In short terms: In most places where you used Scipy's imread, ImageIO's imread is a drop-in replacement and ImageIO provides the same functionality as Scipy in these cases. In some cases, you will need to update the keyword arguments used.

- 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 plugins, e.g. *`Pillow`*.
- ImageIO's functions all return numpy arrays, albeit as a subclass (so that meta data can be attached).

1.2.5 Depreciating Python 2.7

Python 2.7 is deprecated from 2020. For more information on the scientific Python ecosystem's transition to Python3 only, see the python3-statement.

Imageio 2.6.x is the last release to support Python 2.7. This release will remain available on Pypi and conda-forge. The *py2* branch may be used to continue supporting 2.7, but one should not expect (free) contributions from the imageio developers.

For more information on porting your code to run on Python 3, see the python3-howto.

1.2.6 Freezing ImageIO

Note: Starting with `pyinstaller-hooks-contrib==2022.3` pyinstaller will automatically include any ImageIO plugins that are installed in your environment. The techniques described here target earlier versions.

When freezing ImageIO you need to make sure that you are collecting the plugins you need in addition to the core library. This will not happen automagically, because plugins are lazy-loaded upon first use for better platform support and reduced loading times. This lazy-loading is an interesting case as (1) these plugins are exactly the kind of thing

that PyInstaller tries desperately to avoid collecting to minimize package size and (2) these plugins are - by themself - super lightweight and won't bloat the package.

To add plugins to the application your first option is to add all of imageio to your package, which will also include the plugins. This can be done by using a command-line switch when calling pyinstaller that will gather all plugins:

```
pyinstaller --collect-submodules imageio <entry_script.py>
```

Note that it is generally recommended to do this from within a virtual environment in which you don't have unnecessary backends installed. Otherwise, any backend that is present will be included in the package and, if it is not being used, may increase package size unnecessarily.

Alternatively, if you want to limit the plugins used, you can include them individually using `--hidden-import`:

```
pyinstaller --hidden-import imageio.plugins.<plugin> <entry_script.py>
```

In addition, some plugins (e.g., ffmpeg) make use of external programs, and for these, you will need to take extra steps to also include necessary binaries. If you can't make it work, feel free to submit a new issue, so that we can see how to improve this documentation further.

CHAPTER

TWO

SUPPORTED FORMATS

Note: If you just want to know if a specific extension/format is supported you can search this page using `Ctrl+F` and then type the name of the extension or format.

ImageIO reads and writes images by deligating your request to one of many backends. Example backends are pillow, ffmpeg, tifffile among others. Each backend supports its own set of formats, which is how ImageIO manages to support so many of them.

To help you navigate this format jungle, ImageIO provides various curated lists of formats depending on your use-case

2.1 Formats by Plugin

Below you can find a list of each plugin that exists in ImageIO together with the formats that this plugin supports. This can be useful, for example, if you have to decide which plugins to install and/or depend on in your project.

FreeImage
: tif, tiff, jpeg, jpg, bmp, png, bw, dds, gif, ico, j2c, j2k, jp2,pbm, pcd, PCX, pgm, ppm, psd, ras, rgb, rgba, sgi, tga, xbm, xpm, pic, raw, 3fr, arw, bay, bmq, cap, cine, cr2, crw, cs1, cut, dc2, dcr, dng, drf, dsc, erf, exr, fff, g3, hdp, hdr, ia, iff, iiq, jif, jng, jpe, jxr, k25, kc2, kdc, koa, lbm, mdc, mef, mos, mrw, nef, nrw, orf, pct, pef, pfm, pict, ptx, pxn, qtk, raf, rdc, rw2, rwl, rwz, sr2, srf, srw, sti, targa, wap, wbm, wbmp, wdp, webp

Pillow
: tif, tiff, jpeg, jpg, bmp, png, bw, dds, gif, ico, j2c, j2k, jp2, pbm, pcd, PCX, pgm, ppm, psd, ras, rgb, rgba, sgi, tga, xbm, xpm, fit, fits, bufr, CLIPBOARDGRAB, cur, dcx, DIB, emf, eps, flc, fli, fpx, ftc, ftu, gbr, grib, h5, hdf, icns, iim, im, IMT, jfif, jpc, jpf, jpx, MCIDAS, mic, mpo, msp, ps, pxr, SCREENGRAB, SPIDER, wmf, XVTHUMB

ITK
: tif, tiff, jpeg, jpg, bmp, png, pic, img, lsm, dcm, dicom, gdcm, gipl, hdf5, hdr, img.gz, ipl, mgh, mha, mhd, mnc, mnc2, nhdr, nia, nii, nii.gz, nrrd, vtk

FFMPEG
: avi, mkv, mov, mp4, mpeg, mpg, WEBCAM, webm, wmv

GDAL
: tif, tiff, jpeg, jpg, img, ecw

tifffile
: tif,tiff,lsm,stk

FITS
: fit,fits,fts,fz

DICOM
: dcm,ct,mri

Lytro
: raw,lfp,lfr

FEI/SEM
: tif, tiff

Numpy
: npz

BSDF
: bsdf

SPE
: spe

SWF
: swf

2.2 Video Formats

Below you can find an alphabetically sorted list of the video extensions/formats that ImageIO is aware of. If an extension is listed here, it is supported. If an extension is not listed here, it may still be supported if one of the backends supports the extension/format. If you encounter the latter, please create a new issue so that we can keep below list up to date and add support for any missing formats.

Each entry in the list below follows the following format:

```
<extension> (<format name>): <plugin> <plugin> ...
```

where <plugin> is the name of a plugin that can handle the format. If you wish to use a specific plugin to load a format, you would use the name as specified. For example, if you have a MOV file that you wish to open with FFMPEG and the <plugin> is called FFMPEG you would call:

```
iio.imread("image.mov", format="FFMPEG")
```

Format List

Note: To complete this list we are looking for each format's full name and a link to the spec. If you come across this information, please consider sharing it by creating a new issue.

- **.264** (raw H.264 video): *pyav*
- **.265** (raw HEVC video): *pyav*
- **.3g2** (3GP (3GPP file format)): *pyav*
- **.3gp** (3GP (3GPP file format)): *pyav*
- **.A64** (a64 - video for Commodore 64): *pyav*
- **.a64** (a64 - video for Commodore 64): *pyav*
- **.adp** (Xilam DERF): *pyav*

- **.amr** (3GPP AMR): *pyav*
- **.amv** (AMV): *pyav*
- **.asf** (ASF (Advanced / Active Streaming Format)): *pyav*
- **.avc** (raw H.264 video): *pyav*
- **.avi** (Audio Video Interleave): *FFMPEG*
- **.avr** (AVR (Audio Visual Research)): *pyav*
- **.avs** (raw AVS2-P2/IEEE1857.4 video): *pyav*
- **.avs2** (raw AVS2-P2/IEEE1857.4 video): *pyav*
- **.avs3** (raw AVS3-P2/IEEE1857.10): *pyav*
- **.bmv** (Discworld II BMV): *pyav*
- **.cavs** (raw Chinese AVS (Audio Video Standard) video): *pyav*
- **.cdg** (CD Graphics): *pyav*
- **.cdxl** (Commodore CDXL video): *pyav*
- **.cgi** (raw Ingenient MJPEG): *pyav*
- **.chk** (WebM Chunk Muxer): *pyav*
- **.cif** (raw video): *pyav*
- **.cpk** (Sega FILM / CPK): *pyav*
- **.dat** (Video CCTV DAT): *pyav*
- **.dav** (Video DAV): *pyav*
- **.dif** (DV (Digital Video)): *pyav*
- **.dnxhd** (raw DNxHD (SMPTE VC-3)): *pyav*
- **.dnxhr** (raw DNxHD (SMPTE VC-3)): *pyav*
- **.drc** (raw Dirac): *pyav*
- **.dv** (DV (Digital Video)): *pyav*
- **.dvd** (MPEG-2 PS (DVD VOB)): *pyav*
- **.f4v** (3GP (3GPP file format)): *pyav*
- **.flm** (Adobe Filmstrip): *pyav*
- **.flv** (FLV (Flash Video)): *pyav*
- **.gsm** (raw GSM): *pyav*
- **.gxf** (GXF (General eXchange Format)): *pyav*
- **.h261** (raw H.261): *pyav*
- **.h263** (raw H.263): *pyav*
- **.h264** (raw H.264 video): *pyav*
- **.h265** (raw HEVC video): *pyav*
- **.h26l** (raw H.264 video): *pyav*
- **.hevc** (raw HEVC video): *pyav*

- **.idf** (iCE Draw File): *pyav*
- **.ifv** (IFV CCTV DVR): *pyav*
- **.imx** (Simbiosis Interactive IMX): *pyav*
- **.ipu** (raw IPU Video): *pyav*
- **.ism** (3GP (3GPP file format)): *pyav*
- **.isma** (3GP (3GPP file format)): *pyav*
- **.ismv** (3GP (3GPP file format)): *pyav*
- **.ivf** (On2 IVF): *pyav*
- **.ivr** (IVR (Internet Video Recording)): *pyav*
- **.j2k** (JPEG 2000): *pillow J2K-FI JPEG2000-PIL pyav*
- **.kux** (KUX (YouKu)): *pyav*
- **.lvf** (LVF): *pyav*
- **.m1v** (raw MPEG-1 video): *pyav*
- **.m2t** (MPEG-TS (MPEG-2 Transport Stream)): *pyav*
- **.m2ts** (MPEG-TS (MPEG-2 Transport Stream)): *pyav*
- **.m2v** (raw MPEG-2 video): *pyav*
- **.m4a** (3GP (3GPP file format)): *pyav*
- **.m4b** (3GP (3GPP file format)): *pyav*
- **.m4v** (iPod H.264 MP4 (MPEG-4 Part 14)): *pyav*
- **.mj2** (3GP (3GPP file format)): *pyav*
- **.mjpeg** (raw MJPEG video): *pyav*
- **.mjpg** (Loki SDL MJPEG): *pyav*
- **.mk3d** (Matroska / WebM): *pyav*
- **.mka** (Matroska / WebM): *pyav*
- **.mks** (Matroska / WebM): *pyav*
- **.mkv** (Matroska Multimedia Container): *FFMPEG pyav*
- **.mods** (MobiClip MODS): *pyav*
- **.moflex** (MobiClip MOFLEX): *pyav*
- **.mov** (QuickTime File Format): *FFMPEG pyav*
- **.mp4** (MPEG-4 Part 14): *FFMPEG pyav*
- **.mpc** (Musepack): *pyav*
- **.mpd** (DASH Muxer): *pyav*
- **.mpeg** (MPEG-1 Moving Picture Experts Group): *FFMPEG pyav*
- **.mpg** (Moving Picture Experts Group): *pillow FFMPEG pyav*
- **.mpo** (JPEG Multi-Picture Format): *pillow MPO-PIL*
- **.mts** (MPEG-TS (MPEG-2 Transport Stream)): *pyav*

- **.mvi** (Motion Pixels MVI): *pyav*
- **.mxf** (MXF (Material eXchange Format) Operational Pattern Atom): *pyav*
- **.mxg** (MxPEG clip): *pyav*
- **.nut** (NUT): *pyav*
- **.obu** (AV1 Annex B): *pyav*
- **.ogg** (Ogg): *pyav*
- **.ogv** (Ogg Video): *pyav*
- **.psp** (3GP (3GPP file format)): *pyav*
- **.qcif** (raw video): *pyav*
- **.rcv** (VC-1 test bitstream): *pyav*
- **.rgb** (Silicon Graphics Image): *pillow SGI-PIL*
- **.rm** (RealMedia): *pyav*
- **.roq** (raw id RoQ): *pyav*
- **.sdr2** (SDR2): *pyav*
- **.ser** (SER (Simple uncompressed video format for astronomical capturing)): *pyav*
- **.sga** (Digital Pictures SGA): *pyav*
- **.svag** (Konami PS2 SVAG): *pyav*
- **.svs** (Square SVS): *pyav*
- **.ts** (MPEG-TS (MPEG-2 Transport Stream)): *pyav*
- **.ty** (TiVo TY Stream): *pyav*
- **.ty+** (TiVo TY Stream): *pyav*
- **.v** (NC camera feed): *pyav*
- **.v210** (Uncompressed 4:2:2 10-bit): *pyav*
- **.vb** (Beam Software SIFF): *pyav*
- **.vc1** (raw VC-1 video): *pyav*
- **.vc2** (raw Dirac): *pyav*
- **.viv** (Vivo): *pyav*
- **.vob** (MPEG-2 PS (SVCD)): *pyav*
- **.webm** (Matroska): *FFMPEG pyav*
- **.wmv** (Windows Media Video): *FFMPEG*
- **.wtv** (Windows Television (WTV)): *pyav*
- **.xl** (Commodore CDXL video): *pyav*
- **.xmv** (Microsoft XMV): *pyav*
- **.y4m** (YUV4MPEG pipe): *pyav*
- **.yop** (Psygnosis YOP): *pyav*
- **.yuv** (raw video): *pyav*

- **.yuv10** (Uncompressed 4:2:2 10-bit): *pyav*

2.3 All Formats

Below you can find an alphabetically sorted list of *all* extensions/file-formats that ImageIO is aware of. If an extension is listed here, it is supported. If an extension is not listed here, it may still be supported if one of the backends supports the extension/format. If you encoutner the latter, please create a new issue so that we can keep below list up to date and add support for any missing formats.

Each entry in the list below follows the following format:

```
extension (format_name): plugin1 plugin2 ...
```

where the plugins refer to imageio plugins that can handle the format. If you wish to use a specific plugin to load a format, you would use the name as specified here. For example, if you have a PNG file that you wish to open with pillow you would call:

```
iio.imread("image.png", format="PNG-PIL")
```

Format List

Note: To complete this list we are looking for each format's full name and a link to the spec. If you come across this information, please consider sharing it by creating a new issue.

- **.264** (raw H.264 video): *pyav*
- **.265** (raw HEVC video): *pyav*
- **.3fr** (Hasselblad raw): *RAW-FI*
- **.3g2** (3GP (3GPP file format)): *pyav*
- **.3g2** (3GP2 (3GPP2 file format)): *pyav*
- **.3g2** (MP4 (MPEG-4 Part 14)): *pyav*
- **.3g2** (QuickTime / MOV): *pyav*
- **.3gp** (3GP (3GPP file format)): *pyav*
- **.3gp** (3GP2 (3GPP2 file format)): *pyav*
- **.3gp** (MP4 (MPEG-4 Part 14)): *pyav*
- **.3gp** (QuickTime / MOV): *pyav*
- **.A64** (a64 - video for Commodore 64): *pyav*
- **.IMT** (IM Tools): *pillow IMT-PIL*
- **.MCIDAS** (McIdas area file): *pillow MCIDAS-PIL*
- **.PCX** (Zsoft Paintbrush): *pillow PCX-FI PCX-PIL*
- **.SPIDER**: *pillow SPIDER-PIL*
- **.XVTHUMB** (Thumbnail Image): *pillow XVTHUMB-PIL*
- **.a64** (a64 - video for Commodore 64): *pyav*

- **.adp** (Xilam DERF): *pyav*
- **.amr** (3GPP AMR): *pyav*
- **.amv** (AMV): *pyav*
- **.apng** (Animated Portable Network Graphics): *pillow pyav*
- **.arw** (Sony alpha): *RAW-FI*
- **.asf** (ASF (Advanced / Active Streaming Format)): *pyav*
- **.asf** (ASF (Advanced / Active Streaming Format)): *pyav*
- **.avc** (raw H.264 video): *pyav*
- **.avi** (Audio Video Interleave): *FFMPEG*
- **.avi** (AVI (Audio Video Interleaved)): *pyav*
- **.avr** (AVR (Audio Visual Research)): *pyav*
- **.avs** (raw AVS2-P2/IEEE1857.4 video): *pyav*
- **.avs2** (raw AVS2-P2/IEEE1857.4 video): *pyav*
- **.avs3** (raw AVS3-P2/IEEE1857.10): *pyav*
- **.bay** (Casio raw format): *RAW-FI*
- **.blp**: *pillow*
- **.bmp** (Bitmap): *pillow BMP-PIL BMP-FI ITK pyav opencv*
- **.bmq** (Re-Volt mipmap): *RAW-FI*
- **.bmv** (Discworld II BMV): *pyav*
- **.bsdf** (Binary Structured Data Format): *BSDF*
- **.bufr** (Binary Universal Form for the Representation of meteorological data): *pillow BUFR-PIL*
- **.bw** (Silicon Graphics Image): *pillow SGI-PIL SGI-FI*
- **.cap** (Scirra Construct): *RAW-FI*
- **.cavs** (raw Chinese AVS (Audio Video Standard) video): *pyav*
- **.cdg** (CD Graphics): *pyav*
- **.cdxl** (Commodore CDXL video): *pyav*
- **.cgi** (raw Ingenient MJPEG): *pyav*
- **.chk** (WebM Chunk Muxer): *pyav*
- **.cif** (raw video): *pyav*
- **.cine** (AMETEK High Speed Camera Format): *RAW-FI*
- **.cpk** (Sega FILM / CPK): *pyav*
- **.cr2**: *RAW-FI*
- **.crw**: *RAW-FI*
- **.cs1**: *RAW-FI*
- **.ct** (Computerized Tomography): *DICOM*
- **.cur** (Windows Cursor Icons): *pillow CUR-PIL*

- **.cut** (Dr. Halo): *CUT-FI*
- **.dat** (Video CCTV DAT): *pyav*
- **.dav** (Video DAV): *pyav*
- **.dc2**: *RAW-FI*
- **.dcm** (DICOM file format): *DICOM ITK*
- **.dcr**: *RAW-FI*
- **.dcx** (Intel DCX): *pillow DCX-PIL*
- **.dds** (DirectX Texture Container): *pillow DDS-FI DDS-PIL*
- **.dib** (Windows Bitmap): *pillow DIB-PIL*
- **.dicom** (DICOM file format): *ITK*
- **.dif** (DV (Digital Video)): *pyav*
- **.dip** (Device-Independent Bitmap): *opencv*
- **.dng**: *RAW-FI*
- **.dnxhd** (raw DNxHD (SMPTE VC-3)): *pyav*
- **.dnxhr** (raw DNxHD (SMPTE VC-3)): *pyav*
- **.dpx**: *pyav*
- **.drc** (raw Dirac): *pyav*
- **.drf**: *RAW-FI*
- **.dsc**: *RAW-FI*
- **.dv** (DV (Digital Video)): *pyav*
- **.dvd** (MPEG-2 PS (DVD VOB)): *pyav*
- **.ecw** (Enhanced Compression Wavelet): *GDAL*
- **.emf** (Windows Metafile): *pillow WMF-PIL*
- **.eps** (Encapsulated Postscript): *pillow EPS-PIL*
- **.erf**: *RAW-FI*
- **.exr** (OpenEXR): *EXR-FI pyav opencv*
- **.f4v** (3GP (3GPP file format)): *pyav*
- **.f4v** (3GP2 (3GPP2 file format)): *pyav*
- **.f4v** (F4V Adobe Flash Video): *pyav*
- **.f4v** (MP4 (MPEG-4 Part 14)): *pyav*
- **.f4v** (QuickTime / MOV): *pyav*
- **.fff**: *RAW-FI*
- **.fit** (Flexible Image Transport System File): *pillow FITS-PIL FITS*
- **.fits** (Flexible Image Transport System File): *pillow FITS-PIL FITS pyav*
- **.flc** (Autodesk FLC Animation): *pillow FLI-PIL*
- **.fli** (Autodesk FLI Animation): *pillow FLI-PIL*

- **.flm** (Adobe Filmstrip): *pyav*
- **.flv** (FLV (Flash Video)): *pyav*
- **.flv** (live RTMP FLV (Flash Video)): *pyav*
- **.fpx** (Kodak FlashPix): *pillow FPX-PIL*
- **.ftc** (Independence War 2: Edge Of Chaos Texture Format): *pillow FTEX-PIL*
- **.fts** (Flexible Image Transport System File): *FITS*
- **.ftu** (Independence War 2: Edge Of Chaos Texture Format): *pillow FTEX-PIL*
- **.fz** (Flexible Image Transport System File): *FITS*
- **.g3** (Raw fax format CCITT G.3): *G3-FI*
- **.gbr** (GIMP brush file): *pillow GBR-PIL*
- **.gdcm** (Grassroots DICOM): *ITK*
- **.gif** (Graphics Interchange Format): *pillow GIF-PIL pyav*
- **.gipl** (UMDS GIPL): *ITK*
- **.grib** (gridded meteorological data): *pillow GRIB-PIL*
- **.gsm** (raw GSM): *pyav*
- **.gxf** (GXF (General eXchange Format)): *pyav*
- **.h261** (raw H.261): *pyav*
- **.h263** (raw H.263): *pyav*
- **.h264** (raw H.264 video): *pyav*
- **.h265** (raw HEVC video): *pyav*
- **.h26l** (raw H.264 video): *pyav*
- **.h5** (Hierarchical Data Format 5): *pillow HDF5-PIL*
- **.hdf** (Hierarchical Data Format 5): *pillow HDF5-PIL*
- **.hdf5** (Hierarchical Data Format 5): *ITK*
- **.hdp** (JPEG Extended Range): *JPEG-XR-FI*
- **.hdr** (High Dynamic Range Image): *HDR-FI ITK opencv*
- **.hevc** (raw HEVC video): *pyav*
- **.ia**: *RAW-FI*
- **.icb**: *pillow*
- **.icns** (Mac OS Icon File): *pillow ICNS-PIL*
- **.ico** (Windows Icon File): *pillow* `ICO-FI` *ICO-PIL pyav*
- **.idf** (iCE Draw File): *pyav*
- **.iff** (ILBM Interleaved Bitmap): *IFF-FI*
- **.ifv** (IFV CCTV DVR): *pyav*
- **.iim** (IPTC/NAA): *pillow IPTC-PIL*
- **.iiq**: *RAW-FI*

- **.im** (IFUNC Image Memory): *pillow IM-PIL*
- **.im1**: *pyav*
- **.im24**: *pyav*
- **.im8**: *pyav*
- **.img**: *ITK GDAL*
- **.img.gz**: *ITK*
- **.imx** (Simbiosis Interactive IMX): *pyav*
- **.ipl** (Image Processing Lab): *ITK*
- **.ipu** (raw IPU Video): *pyav*
- **.ism** (3GP (3GPP file format)): *pyav*
- **.ism** (3GP2 (3GPP2 file format)): *pyav*
- **.ism** (MP4 (MPEG-4 Part 14)): *pyav*
- **.ism** (QuickTime / MOV): *pyav*
- **.isma** (3GP (3GPP file format)): *pyav*
- **.isma** (3GP2 (3GPP2 file format)): *pyav*
- **.isma** (MP4 (MPEG-4 Part 14)): *pyav*
- **.isma** (QuickTime / MOV): *pyav*
- **.ismv** (3GP (3GPP file format)): *pyav*
- **.ismv** (3GP2 (3GPP2 file format)): *pyav*
- **.ismv** (MP4 (MPEG-4 Part 14)): *pyav*
- **.ismv** (QuickTime / MOV): *pyav*
- **.ivf** (On2 IVF): *pyav*
- **.ivr** (IVR (Internet Video Recording)): *pyav*
- **.j2c** (JPEG 2000): *pillow J2K-FI JPEG2000-PIL pyav*
- **.j2k** (JPEG 2000): *pillow J2K-FI JPEG2000-PIL pyav*
- **.j2k** (raw MJPEG 2000 video): *pyav*
- **.jfif** (JPEG): *pillow JPEG-PIL*
- **.jif** (JPEG): *JPEG-FI*
- **.jls**: *pyav*
- **.jng** (JPEG Network Graphics): *JNG-FI*
- **.jp2** (JPEG 2000): *pillow JP2-FI JPEG2000-PIL pyav opencv*
- **.jpc** (JPEG 2000): *pillow JPEG2000-PIL*
- **.jpe** (JPEG): *pillow JPEG-FI JPEG-PIL opencv*
- **.jpeg** (Joint Photographic Experts Group): *pillow JPEG-PIL JPEG-FI ITK GDAL pyav opencv*
- **.jpf** (JPEG 2000): *pillow JPEG2000-PIL*
- **.jpg** (Joint Photographic Experts Group): *pillow JPEG-PIL JPEG-FI ITK GDAL pyav opencv*

- **.jpx** (JPEG 2000): *pillow JPEG2000-PIL*
- **.jxr** (JPEG Extended Range): *JPEG-XR-FI*
- **.k25**: *RAW-FI*
- **.kc2**: *RAW-FI*
- **.kdc**: *RAW-FI*
- **.koa** (C64 Koala Graphics): *KOALA-FI*
- **.kux** (KUX (YouKu)): *pyav*
- **.lbm** (ILBM Interleaved Bitmap): *IFF-FI*
- **.lfp** (Lytro F01): *LYTRO-LFP*
- **.lfr** (Lytro Illum): *LYTRO-LFR*
- **.ljpg**: *pyav*
- **.lsm** (ZEISS LSM): *ITK TIFF*
- **.lvf** (LVF): *pyav*
- **.m1v** (raw MPEG-1 video): *pyav*
- **.m2t** (MPEG-TS (MPEG-2 Transport Stream)): *pyav*
- **.m2ts** (MPEG-TS (MPEG-2 Transport Stream)): *pyav*
- **.m2v** (raw MPEG-2 video): *pyav*
- **.m4a** (3GP (3GPP file format)): *pyav*
- **.m4a** (3GP2 (3GPP2 file format)): *pyav*
- **.m4a** (iPod H.264 MP4 (MPEG-4 Part 14)): *pyav*
- **.m4a** (MP4 (MPEG-4 Part 14)): *pyav*
- **.m4a** (QuickTime / MOV): *pyav*
- **.m4b** (3GP (3GPP file format)): *pyav*
- **.m4b** (3GP2 (3GPP2 file format)): *pyav*
- **.m4b** (iPod H.264 MP4 (MPEG-4 Part 14)): *pyav*
- **.m4b** (MP4 (MPEG-4 Part 14)): *pyav*
- **.m4b** (QuickTime / MOV): *pyav*
- **.m4v** (iPod H.264 MP4 (MPEG-4 Part 14)): *pyav*
- **.m4v** (raw MPEG-4 video): *pyav*
- **.mdc**: *RAW-FI*
- **.mef**: *RAW-FI*
- **.mgh** (FreeSurfer File Format): *ITK*
- **.mha** (ITK MetaImage): *ITK*
- **.mhd** (ITK MetaImage Header): *ITK*
- **.mic** (Microsoft Image Composer): *pillow MIC-PIL*
- **.mj2** (3GP (3GPP file format)): *pyav*

- **.mj2** (3GP2 (3GPP2 file format)): *pyav*
- **.mj2** (MP4 (MPEG-4 Part 14)): *pyav*
- **.mj2** (QuickTime / MOV): *pyav*
- **.mjpeg** (raw MJPEG video): *pyav*
- **.mjpg** (Loki SDL MJPEG): *pyav*
- **.mjpg** (MIME multipart JPEG): *pyav*
- **.mjpg** (raw MJPEG video): *pyav*
- **.mk3d** (Matroska / WebM): *pyav*
- **.mk3d** (WebM): *pyav*
- **.mka** (Matroska / WebM): *pyav*
- **.mka** (WebM): *pyav*
- **.mks** (Matroska / WebM): *pyav*
- **.mks** (WebM): *pyav*
- **.mkv** (Matroska Multimedia Container): *FFMPEG pyav*
- **.mnc** (Medical Imaging NetCDF): *ITK*
- **.mnc2** (Medical Imaging NetCDF 2): *ITK*
- **.mods** (MobiClip MODS): *pyav*
- **.moflex** (MobiClip MOFLEX): *pyav*
- **.mos** (Leaf Raw Image Format): *RAW-FI*
- **.mov** (QuickTime File Format): *FFMPEG pyav*
- **.mp4** (MPEG-4 Part 14): *FFMPEG pyav*
- **.mpc** (Musepack): *pyav*
- **.mpd** (DASH Muxer): *pyav*
- **.mpeg** (MPEG-1 Moving Picture Experts Group): *FFMPEG pyav*
- **.mpeg** (raw MPEG-1 video): *pyav*
- **.mpg** (Moving Picture Experts Group): *pillow FFMPEG pyav*
- **.mpg** (raw MPEG-1 video): *pyav*
- **.mpo** (JPEG Multi-Picture Format): *pillow MPO-PIL*
- **.mpo** (raw MJPEG video): *pyav*
- **.mri** (Magnetic resonance imaging): *DICOM*
- **.mrw**: *RAW-FI*
- **.msp** (Windows Paint): *pillow MSP-PIL*
- **.mts** (MPEG-TS (MPEG-2 Transport Stream)): *pyav*
- **.mvi** (Motion Pixels MVI): *pyav*
- **.mxf** (MXF (Material eXchange Format) Operational Pattern Atom): *pyav*
- **.mxf** (MXF (Material eXchange Format)): *pyav*

- **.mxg** (MxPEG clip): *pyav*
- **.nef**: *RAW-FI*
- **.nhdr**: *ITK*
- **.nia**: *ITK*
- **.nii**: *ITK*
- **.nii.gz** (nii.gz): *ITK*
- **.npz** (Numpy Array): *NPZ*
- **.nrrd**: *ITK*
- **.nrw**: *RAW-FI*
- **.nut** (NUT): *pyav*
- **.obu** (AV1 Annex B): *pyav*
- **.obu** (AV1 low overhead OBU): *pyav*
- **.ogg** (Ogg): *pyav*
- **.ogv** (Ogg Video): *pyav*
- **.orf**: *RAW-FI*
- **.palm**: *pillow*
- **.pam**: *pyav*
- **.pbm** (Portable Bitmap): *PGM-FI PGMRAW-FI pyav opencv*
- **.pbm** (Pbmplus image): *pillow PPM-PIL PPM-FI*
- **.pcd** (Kodak PhotoCD): *pillow PCD-FI PCD-PIL*
- **.pct** (Macintosh PICT): *PICT-FI*
- **.pcx**: *pyav*
- **.pdf**: *pillow*
- **.pef**: *RAW-FI*
- **.pfm**: *PFM-FI pyav opencv*
- **.pgm** (Portable Greymap): *pillow PGM-FI PGMRAW-FI pyav opencv*
- **.pgmyuv**: *pyav*
- **.pic** (Macintosh PICT): *PICT-FI ITK opencv*
- **.pict** (Macintosh PICT): *PICT-FI*
- **.pix**: *pyav*
- **.png** (Portable Network Graphics): *pillow PNG-PIL PNG-FI ITK pyav opencv*
- **.pnm** (Portable Image Format): *pillow opencv*
- **.ppm** (Pbmplus image): *pillow PPM-PIL pyav*
- **.ppm** (Portable Pixelmap (ASCII)): *PPM-FI opencv*
- **.ppm** (Portable Pixelmap (Raw)): *PPMRAW-FI*
- **.ppm**: *pyav*

- **.ps** (Ghostscript): *pillow EPS-PIL*
- **.psd** (Adope Photoshop 2.5 and 3.0): *pillow PSD-PIL PSD-FI*
- **.psp** (3GP (3GPP file format)): *pyav*
- **.psp** (3GP2 (3GPP2 file format)): *pyav*
- **.psp** (MP4 (MPEG-4 Part 14)): *pyav*
- **.psp** (PSP MP4 (MPEG-4 Part 14)): *pyav*
- **.psp** (QuickTime / MOV): *pyav*
- **.ptx**: *RAW-FI*
- **.pxm** (Portable image format): *opencv*
- **.pxn**: *RAW-FI*
- **.pxr** (PIXAR raster image): *pillow PIXAR-PIL*
- **.qcif** (raw video): *pyav*
- **.qtk**: *RAW-FI*
- **.raf**: *RAW-FI*
- **.ras** (Sun Raster File): *pillow SUN-PIL RAS-FI pyav opencv*
- **.raw**: *RAW-FI LYTRO-ILLUM-RAW LYTRO-F01-RAW*
- **.rcv** (VC-1 test bitstream): *pyav*
- **.rdc**: *RAW-FI*
- **.rgb** (Silicon Graphics Image): *pillow SGI-PIL*
- **.rgb** (raw video): *pyav*
- **.rgba** (Silicon Graphics Image): *pillow SGI-PIL*
- **.rm** (RealMedia): *pyav*
- **.roq** (raw id RoQ): *pyav*
- **.rs**: *pyav*
- **.rw2**: *RAW-FI*
- **.rwl**: *RAW-FI*
- **.rwz**: *RAW-FI*
- **.sdr2** (SDR2): *pyav*
- **.ser** (SER (Simple uncompressed video format for astronomical capturing)): *pyav*
- **.sga** (Digital Pictures SGA): *pyav*
- **.sgi** (Silicon Graphics Image): *pillow SGI-PIL pyav*
- **.spe** (SPE File Format): *SPE*
- **.sr** (Sun Raster File): *opencv*
- **.sr2**: *RAW-FI*
- **.srf**: *RAW-FI*
- **.srw**: *RAW-FI*

- **.sti**: *RAW-FI*
- **.stk**: *TIFF*
- **.sun**: *pyav*
- **.sunras**: *pyav*
- **.svag** (Konami PS2 SVAG): *pyav*
- **.svs** (Square SVS): *pyav*
- **.swf** (ShockWave Flash): *SWF pyav*
- **.targa** (Truevision TGA): *pillow TARGA-FI*
- **.tga** (Truevision TGA): *pillow TGA-PIL TARGA-FI pyav*
- **.tif** (Tagged Image File): *TIFF pillow TIFF-PIL TIFF-FI FEI ITK GDAL pyav opencv*
- **.tiff** (Tagged Image File Format): *TIFF pillow TIFF-PIL TIFF-FI FEI ITK GDAL pyav opencv*
- **.ts** (MPEG-TS (MPEG-2 Transport Stream)): *pyav*
- **.ty** (TiVo TY Stream): *pyav*
- **.ty+** (TiVo TY Stream): *pyav*
- **.v** (NC camera feed): *pyav*
- **.v210** (Uncompressed 4:2:2 10-bit): *pyav*
- **.vb** (Beam Software SIFF): *pyav*
- **.vc1** (raw VC-1 video): *pyav*
- **.vc2** (raw Dirac): *pyav*
- **.vda**: *pillow*
- **.viv** (Vivo): *pyav*
- **.vob** (MPEG-2 PS (SVCD)): *pyav*
- **.vob** (MPEG-2 PS (VOB)): *pyav*
- **.vst**: *pillow*
- **.vtk**: *ITK*
- **.wap** (Wireless Bitmap): *WBMP-FI*
- **.wbm** (Wireless Bitmap): *WBMP-FI*
- **.wbmp** (Wireless Bitmap): *WBMP-FI*
- **.wdp** (JPEG Extended Range): *JPEG-XR-FI*
- **.webm** (Matroska): *FFMPEG pyav*
- **.webp** (Google WebP): *pillow WEBP-FI pyav opencv*
- **.wmf** (Windows Meta File): *pillow WMF-PIL*
- **.wmv** (Windows Media Video): *FFMPEG*
- **.wmv** (ASF (Advanced / Active Streaming Format)): *pyav*
- **.wmv** (ASF (Advanced / Active Streaming Format)): *pyav*
- **.wtv** (Windows Television (WTV)): *pyav*

- **.xbm** (X11 Bitmap): *pillow XBM-PIL XBM-FI pyav*
- **.xface**: *pyav*
- **.xl** (Commodore CDXL video): *pyav*
- **.xmv** (Microsoft XMV): *pyav*
- **.xpm** (X11 Pixel Map): *pillow XPM-PIL XPM-FI*
- **.xwd**: *pyav*
- **.y**: *pyav*
- **.y4m** (YUV4MPEG pipe): *pyav*
- **.yop** (Psygnosis YOP): *pyav*
- **.yuv** (raw video): *pyav*
- **.yuv10** (Uncompressed 4:2:2 10-bit): *pyav*

CHAPTER

THREE

API REFERENCE

ImageIO's API follows the usual idea of choosing sensible defaults for the average user, but giving you fine grained control where you need it. As such the API is split into two parts: The Core API, which covers standard use-cases, and a plugin/backend specific API, which allows you to take full advantage of a backend and its unique features.

3.1 Core APIs

The core API is ImageIOs public frontend. It provides convenient (and powerful) access to the growing number of individual plugins on top of which ImageIO is built. Currently the following APIs exist:

3.1.1 Core API v3

Note: For migration instructions from the v2 API check *our migration guide*.

This API provides a functional interface to all of ImageIOs backends. It is split into two parts: *Everyday Usage* and *Advanced Usage*. The first part (everyday usage) focusses on sensible defaults and automation to cover everyday tasks such as plain reading or writing of images. The second part (advanced usage) focusses on providing fully-featured and performant access to all the features of a particular backend.

API Overview

`imageio.v3.imread`(uri, *[, index, plugin, ...])	Read an ndimage from a URI.
`imageio.v3.imwrite`(uri, image, *[, plugin, ...])	Write an ndimage to the given URI.
`imageio.v3.imiter`(uri, *[, plugin, ...])	Read a sequence of ndimages from a URI.
`imageio.v3.improps`(uri, *[, index, plugin, ...])	Read standardized metadata.
`imageio.v3.immeta`(uri, *[, index, plugin, ...])	Read format-specific metadata.
`imageio.v3.imopen`(uri, io_mode, *[, plugin, ...])	Open an ImageResource.

imageio.v3.imread

imageio.v3.**imread**(*uri*, *, *index=None*, *plugin=None*, *extension=None*, *format_hint=None*, ***kwargs*)

Read an ndimage from a URI.

Opens the given URI and reads an ndimage from it. The exact behavior depends on both the file type and plugin used to open the file. To learn about the exact behavior, check the documentation of the relevant plugin. Typically, imread attempts to read all data stored in the URI.

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.

index
[{int, Ellipsis, None}] If the ImageResource contains multiple ndimages, and index is an integer, select the index-th ndimage from among them and return it. If index is an ellipsis (...), read all ndimages in the file and stack them along a new batch dimension. If index is None, let the plugin decide.

plugin
[{str, None}] The plugin to use. If set to None (default) imread will perform a search for a matching plugin. If not None, this takes priority over the provided format hint (if present).

extension
[str] If not None, treat the provided ImageResource as if it had the given extension. This affects the order in which backends are considered.

format_hint
[str] A format hint to help optimize plugin selection given as the format's extension, e.g. ".png". This can speed up the selection process for ImageResources that don't have an explicit extension, e.g. streams, or for ImageResources where the extension does not match the resource's content.

****kwargs**
Additional keyword arguments will be passed to the plugin's read call.

Returns

image
[ndimage] The ndimage located at the given URI.

imageio.v3.imwrite

imageio.v3.**imwrite**(*uri*, *image*, *, *plugin=None*, *extension=None*, *format_hint=None*, ***kwargs*)

Write an ndimage to the given URI.

The exact behavior depends on the file type and plugin used. To learn about the exact behavior, check the documentation of the relevant plugin.

Parameters

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

image
[np.ndarray] The image to write to disk.

plugin
[{str, None}] The plugin to use. If set to None (default) imwrite will perform a search for a matching plugin. If not None, this takes priority over the provided format hint (if present).

extension
[str] If not None, treat the provided ImageResource as if it had the given extension. This affects the order in which backends are considered, and may also influence the format used when encoding.

format_hint
[str] A format hint to help optimize plugin selection given as the format's extension, e.g. ".png". This can speed up the selection process for ImageResources that don't have an explicit extension, e.g. streams, or for ImageResources where the extension does not match the resource's content. If the ImageResource lacks an explicit extension, it will be set to this format.

****kwargs**
Additional keyword arguments will be passed to the plugin's `write` call.

Returns

encoded_image
[None or Bytes] Returns `None` in all cases, except when `uri` is set to `<bytes>`. In this case it returns the encoded ndimage as a bytes string.

imageio.v3.imiter

`imageio.v3.imiter(`*uri*, *, *plugin=None*, *extension=None*, *format_hint=None*, ***kwargs*`)`

Read a sequence of ndimages from a URI.

Returns an iterable that yields ndimages from the given URI. The exact behavior depends on both, the file type and plugin used to open the file. To learn about the exact behavior, check the documentation of the relevant plugin.

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.

plugin
[{str, None}] The plugin to use. If set to None (default) imiter will perform a search for a matching plugin. If not None, this takes priority over the provided format hint (if present).

extension
[str] If not None, treat the provided ImageResource as if it had the given extension. This affects the order in which backends are considered.

format_hint
[str] A format hint to help optimize plugin selection given as the format's extension, e.g. ".png". This can speed up the selection process for ImageResources that don't have an explicit extension, e.g. streams, or for ImageResources where the extension does not match the resource's content. If the ImageResource lacks an explicit extension, it will be set to this format.

****kwargs**
Additional keyword arguments will be passed to the plugin's `iter` call.

Yields

image
[ndimage] The next ndimage located at the given URI.

imageio.v3.improps

imageio.v3.**improps**(*uri*, *, *index=None*, *plugin=None*, *extension=None*, ***kwargs*)

Read standardized metadata.

Opens the given URI and reads the properties of an ndimage from it. The properties represent standardized metadata. This means that they will have the same name regardless of the format being read or plugin/backend being used. Further, any field will be, where possible, populated with a sensible default (may be *None*) if the ImageResource does not declare a value in its metadata.

Parameters

index
[int] If the ImageResource contains multiple ndimages, and index is an integer, select the index-th ndimage from among them and return its properties. If index is an ellipsis (...), read all ndimages in the file and stack them along a new batch dimension and return their properties. If index is None, let the plugin decide.

plugin
[{str, None}] The plugin to be used. If None, performs a search for a matching plugin.

extension
[str] If not None, treat the provided ImageResource as if it had the given extension. This affects the order in which backends are considered.

****kwargs**
Additional keyword arguments will be passed to the plugin's `properties` call.

Returns

properties
[ImageProperties] A dataclass filled with standardized image metadata.

Notes

Where possible, this will avoid loading pixel data.

imageio.v3.immeta

imageio.v3.**immeta**(*uri*, *, *index=None*, *plugin=None*, *extension=None*, *exclude_applied=True*, ***kwargs*)

Read format-specific metadata.

Opens the given URI and reads metadata for an ndimage from it. The contents of the returned metadata dictionary is specific to both the image format and plugin used to open the ImageResource. To learn about the exact behavior, check the documentation of the relevant plugin. Typically, immeta returns a dictionary specific to the image format, where keys match metadata field names and values are a field's contents.

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.

index
[{int, None}] If the ImageResource contains multiple ndimages, and index is an integer, select the index-th ndimage from among them and return its metadata. If index is an ellipsis (...), return global metadata. If index is None, let the plugin decide the default.

plugin
[{str, None}] The plugin to be used. If None (default), performs a search for a matching plugin.

extension
[str] If not None, treat the provided ImageResource as if it had the given extension. This affects the order in which backends are considered.

****kwargs**
Additional keyword arguments will be passed to the plugin's metadata method.

Returns

image
[ndimage] The ndimage located at the given URI.

imageio.v3.imopen

imageio.v3.**imopen**(*uri*, *io_mode*, *, *plugin=None*, *extension=None*, *format_hint=None*, *legacy_mode=False*, ***kwargs*)

Open an ImageResource.

Warning: This warning is for pypy users. If you are not using a context manager, remember to deconstruct the returned plugin to avoid leaking the file handle to an unclosed file.

Parameters

uri
[str or pathlib.Path or bytes or file or Request] The *ImageResource* to load the image from.

io_mode
[str] The mode in which the file is opened. Possible values are:

```
``r`` - open the file for reading
``w`` - open the file for writing
```

Depreciated since v2.9: A second character can be added to give the reader a hint on what the user expects. This will be ignored by new plugins and will only have an effect on legacy plugins. Possible values are:

```
``i`` for a single image,
``I`` for multiple images,
``v`` for a single volume,
``V`` for multiple volumes,
``?`` for don't care (default)
```

plugin
[str, Plugin, or None] The plugin to use. If set to None (default) imopen will perform a search for a matching plugin. If not None, this takes priority over the provided format hint.

extension
[str] If not None, treat the provided ImageResource as if it had the given extension. This affects the order in which backends are considered, and when writing this may also influence the format used when encoding.

format_hint
[str] A format hint to help optimize plugin selection given as the format's extension, e.g. ".png". This can speed up the selection process for ImageResources that don't have an explicit extension, e.g. streams, or for ImageResources where the extension does not match the resource's content. If the ImageResource lacks an explicit extension, it will be set to this format.

legacy_mode
[bool] If true (default) use the v2 behavior when searching for a suitable plugin. This will ignore v3 plugins and will check `plugin` against known extensions if no plugin with the given name can be found.

****kwargs**
[Any] Additional keyword arguments will be passed to the plugin upon construction.

Notes

Registered plugins are controlled via the `known_plugins` dict in `imageio.config`.

Passing a `Request` as the uri is only supported if `legacy_mode` is `True`. In this case `io_mode` is ignored.

Using the kwarg `format_hint` does not enforce the given format. It merely provides a *hint* to the selection process and plugin. The selection processes uses this hint for optimization; however, a plugin's decision how to read a ImageResource will - typically - still be based on the content of the resource.

Examples

```
>>> import imageio.v3 as iio
>>> with iio.imopen("/path/to/image.png", "r") as file:
>>>     im = file.read()
```

```
>>> with iio.imopen("/path/to/output.jpg", "w") as file:
>>>     file.write(im)
```

Everyday Usage

Reading and Writing

Frequently, the image-related IO you wish to perform can be summarized as "I want to load X as a numpy array.", or "I want to save my array as a XYZ file". This should not be complicated, and it is, in fact, as simple as calling the corresponding function:

```
import imageio.v3 as iio

# reading:
img = iio.imread("imageio:chelsea.png")
```

(continues on next page)

(continued from previous page)

```
# writing:
iio.imwrite("out_image.jpg", img)

# (Check the examples section for many more examples.)
```

Both functions accept any *ImageResource* as their first argument, which means you can read/write almost anything image related; from simple JPGs through ImageJ hyperstacks to MP4 video or RAW formats. Check the *Supported Formats* docs for a list of all formats (that we are aware of).

If you do need to customize or tweak this process you can pass additional keyword arguments (*kwargs*) to overwrite any defaults ImageIO uses when reading or writing. The following *kwargs* are always available:

- `plugin`: The name of the plugin to use for reading/writing.
- `extension`: Treat the provided ImageResource as if it had the given extension.
- `index` (reading only): The index of the ndimage to read. Useful if the format supports storing more than one (GIF, multi-page TIFF, video, etc.).

Additional *kwargs* are specific to the plugin being used and you can find a list of available ones in the documentation of the specific plugin (where they are listed as parameters). A list of all available plugins can be found *here* and you can read more about the above kwargs in the documentation of the respective function (imread/imwrite/etc.).

Handling Metadata

ImageIO serves two types of metadata: ImageProperties and format-specific metadata.

ImageProperties are a collection of standardized metadata fields and are supported by all plugins and for all supported formats. If a file doesn't carry the relevant field or if a format doesn't support it, its value is set to a sensible default. You can access the properties of an image by calling *improps*:

```
import imageio.v3 as iio

props = iio.improps("imageio:newtonscradle.gif")
props.shape
props.dtype
# ...
```

As with the other functions of the API, you can pass generally available kwargs (*plugin*, *index*, ...) to *improps* to modify the behavior. Plugins may specify additional plugin-specific keyword arguments, and those are documented in the plugin-specific docs. Further, accessing this metadata is efficient in the sense that it doesn't load (decode) pixel data.

Format-specific metadata is handled by *immeta*:

```
import imageio.v3 as iio

meta = iio.immeta("imageio:newtonscradle.gif")
```

It returns a dictionary of non-standardized metadata fields. It, too, supports general kwargs (*plugin*, *index*, ...) which may be extended by a specific plugin. Further, it accepts a kwarg called `exclude_applied`. If set to True, this will remove any items from the dictionary that would be consumed by a read call to the plugin. For example, if the metadata sets a rotation flag (the raw pixel data should be rotated before displaying it) and the plugin's read call will rotate the image because if it, then setting `exclude_applied=True` will remove the rotation field from the returned metadata. This can be useful to keep an image and it's metadata in sync.

Further, this type of metadata is much more specific than ImageProperties because different plugins (and formats) may return and support different fields. This means that you can get much more specific metadata for a format, e.g., a frame's side data in a video format, but you need to be mindful of the plugin and format used because these fields may not exist all the time, e.g., jpeg and most other image formats have no side data. In general we try to match a fields name to the name it has in a format; however, we may adjust this if the name is not a valid python string. The best way to know which fields exist for your specific ImageResource is to call `immeta` on it and inspect the result.

Advanced Usage

Iterating Video and Multi-Page Formats

Some formats (GIF, MP4, TIFF, among others) allow storage of more than one ndimage in the same file, and you may wish to process all ndimages in such a file instead of just a single one. While `iio.imread(...)` allows you to read all ndimages and stack them into a ndarray (using *index=...*; check the docs), this comes with two problems:

- Some formats (e.g., SPE or TIFF) allow multiple ndimages with different shapes in the same file, which prevents stacking.
- Some files (especially video) are too big to fit into memory when loaded all at once.

To address these problems, the v3 API introduces `iio.imiter`; a generator yielding ndimages in the order in which they appear in the file:

```
import imageio.v3 as iio

for frame in iio.imiter("imageio:cockatoo.mp4"):
    pass # do something with each frame
```

Just like imread, imiter accepts additional *kwargs* to overwrite any defaults used by ImageIO. Like before, the `function-specific documentation` details the *kwargs* that are always present, and additional kwargs are plugin specific and documented by the respective plugin.

Low-Level Access

At times you may need low-level access to a plugin or file, for example, because:

- you wish to have fine-grained control over when the file is opened/closed.
- you need to perform multiple IO operations and don't want to open the file multiple times.
- a plugin/backend offers unique features not otherwise exposed by the high-level API.

For these cases the v3 API offers `iio.v3.imopen`. It provides a context manager that initializes the plugin and opens the file for reading (`"r"`) or writing (`"w"`), similar to the Python built-in function `open`:

```
import imageio.v3 as iio

with iio.imopen("imageio:chelsea.png", "r") as image_file:
    props = image_file.properties()
    # ... configure HPC pipeline and unicorns
    img = image_file.read()
    # image_file.plugin_specific_function()
```

Similar to above, you can pass the `plugin` *kwarg* to imopen to control the plugin that is being used. The returned plugin instance (*image_file*) exposes the *v3 plugin API*, and can be used for low-level access.

3.1.2 Core API v2

Warning: This API exists for backwards compatibility. It is a wrapper around calls to the v3 API and new code should use the v3 API directly. Check the migration instructions below for detailed information on how to migrate.

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 backend that does the actual work. To see what keyword arguments are supported by a specific format, use the *help()* function.

Note: 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)`.

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

Functions for reading

imageio.v2.imread(uri[, format])	Reads an image from the specified file.
imageio.v2.mimread(uri[, format, memtest])	Reads multiple images from the specified file.
imageio.v2.volread(uri[, format])	Reads a volume from the specified file.
imageio.v2.mvolread(uri[, format, memtest])	Reads multiple volumes from the specified file.

imageio.v2.imread

imageio.v2.**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.v2.mimread

imageio.v2.**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.v2.volread

imageio.v2.**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.v2.mvolread

imageio.v2.**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.

Functions for writing

imageio.v2.imwrite(uri, im[, format])	Write an image to the specified file.
imageio.v2.mimwrite(uri, ims[, format])	Write multiple images to the specified file.
imageio.v2.volwrite(uri, vol[, format])	Write a volume to the specified file.
imageio.v2.mvolwrite(uri, vols[, format])	Write multiple volumes to the specified file.

imageio.v2.imwrite

imageio.v2.**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 write 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.v2.mimwrite

imageio.v2.**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.v2.volwrite

imageio.v2.**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.v2.mvolwrite

imageio.v2.**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.

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.

imageio.v2.**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.v2.**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 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()

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

property closed

Whether the reader/writer is closed.

property 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.)

property 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 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()

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

property closed

Whether the reader/writer is closed.

property format

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

property 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.

Migrating to the v3 API

Note: Make sure to have a look at the *narrative docs for the v3 API* to build some intuition on how to use the new API.

The primary novelty of the v3 API compared to the v2 API is that images are now treated as ndimages, a generalization of (flat) 2D images to N dimensions. This change allows us to design a much simpler API; however, comes with a few backwards incompatible changes.

Avoiding Migration for Legacy Scripts

If you have an old script that you do not wish to migrate, you can avoid most of the migration by explicitly importing the v2 API:

```
import imageio.v2 as imageio
```

This will give you access to most of the old API. However, calls will still rely on the new (v3) plugins and backends, which may cause behavioral changes. As such, you should prefer a full migration to v3 and should avoid using the v2 API in new code. This is primarily as a quick way for you to postpone a full migration until it is convenient for you to do so.

Reading Images

iio.imread can now return a ndimage instead of being limited to flat images. As such, *iio.volread* has merged into *iio.imread* and is now gone. Similarily, *iio.mimread* and *iio.mvolread* have merged into a new function called *iio.imiter*, which returns a generator that yields ndimages from a file in the order in which they appear. Further, the default behavior of *iio.imread* has changed and it has become more context aware. Now it will either return the first stored ndimage (like before) or, depending on the format, all images if this is the more common use-case.

To reproduce similar behavior to the v2 API in cases where the incompatibility matters use one of the following snippets:

```
# img = iio.v2.imread(image_resource)
img = iio.v3.imread(image_resource, index=...)[0]

# img = iio.v2.volread(image_resource)
img = iio.v3.imread(image_resource)

# img = iio.v2.mimread(image_resource)
# img = iio.v2.mvolread(image_resource)
img = [im for im in iio.v3.imiter(image_resource)]
```

Writing Images

Similar to reading images, the new *iio.imwrite* can handle ndimages and lists of images, like `iio.mimwrite` and consorts. `iio.mimwrite`, `iio.volwrite`, and `iio.mvolwrite` have all dissapeared. The same goes for their aliases `iio.mimsave`, `iio.volsave`, `iio.mvolsave`, and `iio.imsave`. They are now all covered by `iio.imwrite`.

To reproduce similar behavior to the v2 API behavior use one of the following snippets:

```
# img = iio.v2.imwrite(image_resource, ndimage)
# img = iio.v2.volwrite(image_resource, ndimage)
img = iio.v3.imwrite(image_resource, ndimage)

# img = iio.v2.mimwrite(image_resource, list_of_ndimages)
# img = iio.v2.mvolwrite(image_resource, list_of_ndimages)
img = iio.v3.imwrite(image_resource, list_of_ndimages)
```

Reader and Writer

Previously, the *reader* and *writer* objects provided an advanced way to read/write data with more control. These are no longer needed. Likewise, the functions `iio.get_reader`, `iio.get_writer`, `iio.read`, and `iio.save` have become obsolete. Instead, the plugin object is used directly, which is instantiated in mode `r` (reading) or `w` (writing). To create such a plugin object, call `iio.imopen` and use it as a context manager:

```
# reader = iio.get_reader(image_resource)
# reader = iio.read(image_resource)
with iio.imopen(image_resource, "r") as reader:
    # file is only open/accessible here
    # img = reader.read(index=0)
    # meta = reader.metadata(index=0)
    # ...
    pass


# writer = iio.get_writer(image_resource)
# writer = iio.save(image_resource)
with iio.imopen(image_resource, "w") as writer:
    # file is only open/accessible here
    # writer.write(ndimage)
    # ...
    pass
```

Metadata

The v3 API makes handling of metadata much more consistent and explicit. Previously in v2, metadata was provided as a `dict` that was attached to the image by returning a custom subclass of `np.ndarray`. Now metadata is served independently from pixel data:

```
# metadata = iio.imread(image_resource).meta
metadata = iio.immeta(image_resource)
```

Further, ImageIO now provides a curated set of standardized metadata, which is called *ImageProperties*, in addition to above metadata. The difference between the two is as follows: The metadata dict contains metadata using format-specific keys to the extent the reading plugin supports them. The ImageProperties dataclass contains generally available metadata using standardized attribute names. Each plugin provides the same set of properties, and if the plugin or format doesn't provide a field it is set to a default value; usually `None`. To access ImageProperties use:

```
props = iio.improps(image_resource)
```

Caveats and Notes

This section is a collection of notes and feedback that we got from other developers that were migrating to ImageIO V3, which might be useful to know while you are migrating your own code.

- The old pillow plugin used to *np.squeeze* the image before writing it. This has been removed in V3 to match pillows native behavior. A trailing axis with dimension 1, e.g., `(256, 256, 1)` will now raise an exception. (see #842)

Core API v3

Note: To use this API import it using:

```
import imageio.v3 as iio
```

Note: Check the narrative documentation to build intuition for how to use this API. You can find them here: *narrative v3 API docs*.

imageio.v3.imread(uri, *[, index, plugin, ...])	Read an ndimage from a URI.
imageio.v3.imiter(uri, *[, plugin, ...])	Read a sequence of ndimages from a URI.
imageio.v3.improps(uri, *[, index, plugin, ...])	Read standardized metadata.
imageio.v3.immeta(uri, *[, index, plugin, ...])	Read format-specific metadata.
imageio.v3.imwrite(uri, image, *[, plugin, ...])	Write an ndimage to the given URI.
imageio.v3.imopen(uri, io_mode, *[, plugin, ...])	Open an ImageResource.

Core API v2

Warning: This API exists for backwards compatibility. It is a wrapper around calls to the v3 API and new code should use the v3 API directly.

Note: To use this API import it using:

```
import imageio.v2 as iio
```

Note: Check the narrative documentation to build intuition for how to use this API. You can find them here: *narrative v2 API docs*.

imageio.v2.imread(uri[, format])	Reads an image from the specified file.
imageio.v2.mimread(uri[, format, memtest])	Reads multiple images from the specified file.
imageio.v2.volread(uri[, format])	Reads a volume from the specified file.
imageio.v2.mvolread(uri[, format, memtest])	Reads multiple volumes from the specified file.
imageio.v2.imwrite(uri, im[, format])	Write an image to the specified file.
imageio.v2.mimwrite(uri, ims[, format])	Write multiple images to the specified file.
imageio.v2.volwrite(uri, vol[, format])	Write a volume to the specified file.
imageio.v2.mvolwrite(uri, vols[, format])	Write multiple volumes to the specified file.
imageio.v2.get_reader(uri[, format, mode])	Returns a *Reader* object which can be used to read data and meta data from the specified file.
imageio.v2.get_writer(uri[, format, mode])	Returns a *Writer* object which can be used to write data and meta data to the specified file.

3.2 Plugins & Backend Libraries

Sometimes, you need to do more than just "load an image, done". Sometimes you need to use a very specific feature of a very specific backend. ImageIO allows you to do so by allowing its plugins to extend the core API. Typically this is done in the form of additional keyword arguments (`kwarg`) or plugin-specific methods. Below you can find a list of available plugins and which arguments they support.

imageio.plugins.bsdf	Read/Write BSDF files.
imageio.plugins.dicom	Read DICOM files.
imageio.plugins.feisem	Read TIFF from FEI SEM microscopes.
imageio.plugins.ffmpeg	Read/Write video using FFMPEG
imageio.plugins.fits	Read FITS files.
imageio.plugins.freeimage	Read/Write images using FreeImage.
imageio.plugins.gdal	Read GDAL files.
imageio.plugins.lytro	Read LFR files (Lytro Illum).
imageio.plugins.npz	Read/Write NPZ files.
imageio.plugins.opencv	Read/Write images using OpenCV.
imageio.plugins.pillow	Read/Write images using Pillow/PIL.
imageio.plugins.pillow_legacy	Read/Write images using pillow/PIL (legacy).
imageio.plugins.pyav	Read/Write Videos (and images) using PyAV.
imageio.plugins.simpleitk	Read/Write images using SimpleITK.
imageio.plugins.spe	Read SPE files.
imageio.plugins.swf	Read/Write SWF files.
imageio.plugins.tifffile	Read/Write TIFF files.

3.2.1 imageio.plugins.bsdf

Read/Write BSDF files.

Backend Library: internal

The BSDF format enables reading and writing of image data in the BSDF serialization format. This format allows storage of images, volumes, and series thereof. Data can be of any numeric data type, and can optionally be compressed. Each image/volume can have associated meta data, which can consist of any data type supported by BSDF.

By default, image data is lazily loaded; the actual image data is not read until it is requested. This allows storing multiple images in a single file and still have fast access to individual images. Alternatively, a series of images can be read in streaming mode, reading images as they are read (e.g. from http).

BSDF is a simple generic binary format. It is easy to extend and there are standard extension definitions for 2D and 3D image data. Read more at http://bsdf.io.

Parameters

random_access
: [bool] Whether individual images in the file can be read in random order. Defaults to True for normal files, and to False when reading from HTTP. If False, the file is read in "streaming mode", allowing reading files as they are read, but without support for "rewinding". Note that setting this to True when reading from HTTP, the whole file is read upon opening it (since lazy loading is not possible over HTTP).

compression
: [int] Use `0` or "no" for no compression, `1` or "zlib" for Zlib compression (same as zip files and PNG), and `2` or "bz2" for Bz2 compression (more compact but slower). Default 1 (zlib). Note that some BSDF implementations may not support compression (e.g. JavaScript).

3.2.2 imageio.plugins.dicom

Read DICOM files.

Backend Library: internal

A format for reading DICOM images: a common format used to store medical image data, such as X-ray, CT and MRI.

This format borrows some code (and ideas) from the pydicom project. However, only a predefined subset of tags are extracted from the file. This allows for great simplifications allowing us to make a stand-alone reader, and also results in a much faster read time.

By default, only uncompressed and deflated transfer syntaxes are supported. If gdcm or dcmtk is installed, these will be used to automatically convert the data. See https://github.com/malaterre/GDCM/releases for installing GDCM.

This format provides functionality to group images of the same series together, thus extracting volumes (and multiple volumes). Using volread will attempt to yield a volume. If multiple volumes are present, the first one is given. Using mimread will simply yield all images in the given directory (not taking series into account).

Parameters

progress
: [{True, False, BaseProgressIndicator}] Whether to show progress when reading from multiple files. Default True. By passing an object that inherits from BaseProgressIndicator, the way in which progress is reported can be costumized.

3.2.3 imageio.plugins.feisem

Read TIFF from FEI SEM microscopes.

Backend Library: internal

This format is based on *TIFF*, and supports the same parameters. FEI microscopes append metadata as ASCII text at the end of the file, which this reader correctly extracts.

Parameters

discard_watermark
: [bool] If True (default), discard the bottom rows of the image, which contain no image data, only a watermark with metadata.

watermark_height
: [int] The height in pixels of the FEI watermark. The default is 70.

See Also

imageio.plugins.tifffile

3.2.4 imageio.plugins.ffmpeg

Read/Write video using FFMPEG

Note: We are in the process of (slowly) replacing this plugin with a new one that is based on pyav. It is faster and more flexible than the plugin documented here. Check the *pyav plugin's documentation* for more information about this plugin.

Backend Library: https://github.com/imageio/imageio-ffmpeg

Note: To use this plugin you have to install its backend:

```
pip install imageio[ffmpeg]
```

The ffmpeg format provides reading and writing for a wide range of movie formats such as .avi, .mpeg, .mp4, etc. as well as the ability to read streams from webcams and USB cameras. It is based on ffmpeg and is inspired by/based moviepy by Zulko.

Parameters for reading

fps
: [scalar] The number of frames per second to read the data at. Default None (i.e. read at the file's own fps). One can use this for files with a variable fps, or in cases where imageio is unable to correctly detect the fps. In case of trouble opening camera streams, it may help to set an explicit fps value matching a framerate supported by the camera.

loop
: [bool] If True, the video will rewind as soon as a frame is requested beyond the last frame. Otherwise, IndexError is raised. Default False. Setting this to True will internally call `count_frames()`, and set the reader's length to that value instead of inf.

size
: [str | tuple] The frame size (i.e. resolution) to read the images, e.g. (100, 100) or "640x480". For camera streams, this allows setting the capture resolution. For normal video data, ffmpeg will rescale the data.

dtype
: [str | type] The dtype for the output arrays. Determines the bit-depth that is requested from ffmpeg. Supported dtypes: uint8, uint16. Default: uint8.

pixelformat
[str] The pixel format for the camera to use (e.g. "yuyv422" or "gray"). The camera needs to support the format in order for this to take effect. Note that the images produced by this reader are always RGB.

input_params
[list] List additional arguments to ffmpeg for input file options. (Can also be provided as `ffmpeg_params` for backwards compatibility) Example ffmpeg arguments to use aggressive error handling: ['-err_detect', 'aggressive']

output_params
[list] List additional arguments to ffmpeg for output file options (i.e. the stream being read by imageio).

print_info
[bool] Print information about the video file as reported by ffmpeg.

Parameters for writing

fps
[scalar] The number of frames per second. Default 10.

codec
[str] the video codec to use. Default 'libx264', which represents the widely available mpeg4. Except when saving .wmv files, then the defaults is 'msmpeg4' which is more commonly supported for windows

quality
[float | None] Video output quality. Default is 5. Uses variable bit rate. Highest quality is 10, lowest is 0. Set to None to prevent variable bitrate flags to FFMPEG so you can manually specify them using output_params instead. Specifying a fixed bitrate using 'bitrate' disables this parameter.

bitrate
[int | None] Set a constant bitrate for the video encoding. Default is None causing 'quality' parameter to be used instead. Better quality videos with smaller file sizes will result from using the 'quality' variable bitrate parameter rather than specifiying a fixed bitrate with this parameter.

pixelformat: str
The output video pixel format. Default is 'yuv420p' which most widely supported by video players.

input_params
[list] List additional arguments to ffmpeg for input file options (i.e. the stream that imageio provides).

output_params
[list] List additional arguments to ffmpeg for output file options. (Can also be provided as `ffmpeg_params` for backwards compatibility) Example ffmpeg arguments to use only intra frames and set aspect ratio: ['-intra', '-aspect', '16:9']

ffmpeg_log_level: str
Sets ffmpeg output log level. Default is "warning". Values can be "quiet", "panic", "fatal", "error", "warning", "info" "verbose", or "debug". Also prints the FFMPEG command being used by imageio if "info", "verbose", or "debug".

macro_block_size: int
Size constraint for video. Width and height, must be divisible by this number. If not divisible by this number imageio will tell ffmpeg to scale the image up to the next closest size divisible by this number. Most codecs are compatible with a macroblock size of 16 (default), some can go smaller (4, 8). To disable this automatic feature set it to None or 1, however be warned many players can't decode videos that are odd in size and some codecs will produce poor results or fail. See https://en.wikipedia.org/wiki/Macroblock.

Notes

If you are using anaconda and `anaconda/ffmpeg` you will not be able to encode/decode H.264 (likely due to licensing concerns). If you need this format on anaconda install `conda-forge/ffmpeg` instead.

You can use the `IMAGEIO_FFMPEG_EXE` environment variable to force using a specific ffmpeg executable.

To get the number of frames before having read them all, you can use the `reader.count_frames()` method (the reader will then use `imageio_ffmpeg.count_frames_and_secs()` to get the exact number of frames, note that this operation can take a few seconds on large files). Alternatively, the number of frames can be estimated from the fps and duration in the meta data (though these values themselves are not always present/reliable).

3.2.5 imageio.plugins.fits

Read FITS files.

Backend Library: Astropy

Note: To use this plugin you have to install its backend:

```
pip install imageio[fits]
```

Flexible Image Transport System (FITS) is an open standard defining a digital file format useful for storage, transmission and processing of scientific and other images. FITS is the most commonly used digital file format in astronomy.

Parameters

cache
: [bool] If the file name is a URL, *~astropy.utils.data.download_file* is used to open the file. This specifies whether or not to save the file locally in Astropy's download cache (default: *True*).

uint
: [bool] Interpret signed integer data where `BZERO` is the central value and `BSCALE == 1` as unsigned integer data. For example, `int16` data with `BZERO = 32768` and `BSCALE = 1` would be treated as `uint16` data.

  Note, for backward compatibility, the kwarg **uint16** may be used instead. The kwarg was renamed when support was added for integers of any size.

ignore_missing_end
: [bool] Do not issue an exception when opening a file that is missing an `END` card in the last header.

checksum
: [bool or str] If *True*, verifies that both `DATASUM` and `CHECKSUM` card values (when present in the HDU header) match the header and data of all HDU's in the file. Updates to a file that already has a checksum will preserve and update the existing checksums unless this argument is given a value of 'remove', in which case the CHECKSUM and DATASUM values are not checked, and are removed when saving changes to the file.

disable_image_compression
: [bool, optional] If *True*, treats compressed image HDU's like normal binary table HDU's.

do_not_scale_image_data
: [bool] If *True*, image data is not scaled using BSCALE/BZERO values when read.

ignore_blank
: [bool] If *True*, the BLANK keyword is ignored if present.

scale_back
[bool] If *True*, when saving changes to a file that contained scaled image data, restore the data to the original type and reapply the original BSCALE/BZERO values. This could lead to loss of accuracy if scaling back to integer values after performing floating point operations on the data.

3.2.6 imageio.plugins.freeimage

Read/Write images using FreeImage.

Backend Library: FreeImage

Note: To use this plugin you have to install its backend:

```
imageio_download_bin freeimage
```

or you can download the backend using the function:

```
imageio.plugins.freeimage.download()
```

Each Freeimage format has the `flags` keyword argument. See the Freeimage documentation for more information.

Parameters

flags
[int] A freeimage-specific option. In most cases we provide explicit parameters for influencing image reading.

3.2.7 imageio.plugins.gdal

Read GDAL files.

Backend: GDAL

Note: To use this plugin you have to install its backend:

```
pip install imageio[gdal]
```

Parameters

none

3.2.8 imageio.plugins.lytro

Read LFR files (Lytro Illum).

Backend: internal

Plugin to read Lytro Illum .lfr and .raw files as produced by the Lytro Illum light field camera. It is actually a collection of plugins, each supporting slightly different keyword arguments

Parameters

meta_only
: [bool] Whether to only read the metadata.

include_thumbnail
: [bool] (only for lytro-lfr and lytro-lfp) Whether to include an image thumbnail in the metadata.

3.2.9 imageio.plugins.npz

Read/Write NPZ files.

Backend: Numpy

NPZ is a file format by numpy that provides storage of array data using gzip compression. This imageio plugin supports data of any shape, and also supports multiple images per file. However, the npz format does not provide streaming; all data is read/written at once. Further, there is no support for meta data.

See the BSDF format for a similar (but more fully featured) format.

Parameters

None

Notes

This format is not available on Pypy.

3.2.10 imageio.plugins.opencv

Read/Write images using OpenCV.

Backend Library: OpenCV

This plugin wraps OpenCV (also known as `cv2`), a popular image processing library. Currently, it exposes OpenCVs image reading capability (no video or GIF support yet); however, this may be added in future releases.

Methods

Note: Check the respective function for a list of supported kwargs and their documentation.

OpenCVPlugin.read(*[, index, colorspace, flags])	Read an image from the ImageResource.
OpenCVPlugin.iter([colorspace, flags])	Yield images from the ImageResource.
OpenCVPlugin.write(ndimage[, is_batch, params])	Save an ndimage in the ImageResource.
OpenCVPlugin.properties([index, colorspace, ...])	Standardized image metadata.
OpenCVPlugin.metadata([index, exclude_applied])	Format-specific metadata.

imageio.plugins.opencv.OpenCVPlugin.read

OpenCVPlugin.**read**(*, *index: Optional[int] = None*, *colorspace: Optional[Union[int, str]] = None*, *flags: int = cv2.IMREAD_COLOR*) → ndarray

Read an image from the ImageResource.

Parameters

index
[int, Ellipsis] If int, read the index-th image from the ImageResource. If `...`, read all images from the ImageResource and stack them along a new, prepended, batch dimension. If None (default), use `index=0` if the image contains exactly one image and `index=...` otherwise.

colorspace
[str, int] The colorspace to convert into after loading and before returning the image. If None (default) keep grayscale images as is, convert images with an alpha channel to `RGBA` and all other images to `RGB`. If int, interpret `colorspace` as one of OpenCVs conversion flags and use it for conversion. If str, convert the image into the given colorspace. Possible string values are: `"RGB"`, `"BGR"`, `"RGBA"`, `"BGRA"`, `"GRAY"`, `"HSV"`, or `"LAB"`.

flags
[int] The OpenCV flag(s) to pass to the reader. Refer to the OpenCV docs for details.

Returns

ndimage
[np.ndarray] The decoded image as a numpy array.

imageio.plugins.opencv.OpenCVPlugin.iter

OpenCVPlugin.**iter**(*colorspace: Optional[Union[int, str]] = None*, *flags: int = cv2.IMREAD_COLOR*) → ndarray

Yield images from the ImageResource.

Parameters

colorspace
[str, int] The colorspace to convert into after loading and before returning the image. If None (default) keep grayscale images as is, convert images with an alpha channel to `RGBA` and all other images to `RGB`. If int, interpret `colorspace` as one of OpenCVs conversion flags and use it for conversion. If str, convert the image into the given colorspace. Possible string values are: `"RGB"`, `"BGR"`, `"RGBA"`, `"BGRA"`, `"GRAY"`, `"HSV"`, or `"LAB"`.

flags
[int] The OpenCV flag(s) to pass to the reader. Refer to the OpenCV docs for details.

Yields

ndimage
[np.ndarray] The decoded image as a numpy array.

imageio.plugins.opencv.OpenCVPlugin.write

OpenCVPlugin.**write**(*ndimage: Union[Sequence[Sequence[Sequence[Sequence[Sequence[Any]]]]], _SupportsArray[dtype], Sequence[_SupportsArray[dtype]], Sequence[Sequence[_SupportsArray[dtype]]], Sequence[Sequence[Sequence[_SupportsArray[dtype]]]], Sequence[Sequence[Sequence[Sequence[_SupportsArray[dtype]]]]], bool, int, float, complex, str, bytes, Sequence[Union[bool, int, float, complex, str, bytes]], Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]], Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]], Sequence[Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]]], List[Union[Sequence[Sequence[Sequence[Sequence[Sequence[Any]]]]], _SupportsArray[dtype], Sequence[_SupportsArray[dtype]], Sequence[Sequence[_SupportsArray[dtype]]], Sequence[Sequence[Sequence[_SupportsArray[dtype]]]], Sequence[Sequence[Sequence[Sequence[_SupportsArray[dtype]]]]], bool, int, float, complex, str, bytes, Sequence[Union[bool, int, float, complex, str, bytes]], Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]], Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]], Sequence[Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]]]]], is_batch: bool = False, params: Optional[List[int]] = None*) → Optional[bytes]

Save an ndimage in the ImageResource.

Parameters

ndimage
[ArrayLike, List[ArrayLike]] The image data that will be written to the file. It is either a single image, a batch of images, or a list of images.

is_batch
[bool] If True, the provided ndimage is a batch of images. If False (default), the provided ndimage is a single image. If the provided ndimage is a list of images, this parameter has no effect.

params
[List[int]] A list of parameters that will be passed to OpenCVs imwrite or imwritemulti functions. Possible values are documented in the OpenCV documentation.

Returns

encoded_image
[bytes, None] If the ImageResource is "<bytes>" the call to write returns the encoded image as a bytes string. Otherwise it returns None.

imageio.plugins.opencv.OpenCVPlugin.properties

OpenCVPlugin.**properties**(*index: int = 0*, *colorspace: Optional[Union[int, str]] = None*, *flags: int = cv2.IMREAD_COLOR*) → *ImageProperties*

Standardized image metadata.

Parameters

index
[int, Ellipsis] If int, get the properties of the index-th image in the ImageResource. If `...`, get the properties of the image stack that contains all images. If None (default), use `index=0` if the image contains exactly one image and `index=...` otherwise.

colorspace
[str, int] The colorspace to convert into after loading and before returning the image. If None (default) keep grayscale images as is, convert images with an alpha channel to `RGBA` and all other images to `RGB`. If int, interpret `colorspace` as one of OpenCVs conversion flags and use it for conversion. If str, convert the image into the given colorspace. Possible string values are: `"RGB"`, `"BGR"`, `"RGBA"`, `"BGRA"`, `"GRAY"`, `"HSV"`, or `"LAB"`.

flags
[int] The OpenCV flag(s) to pass to the reader. Refer to the OpenCV docs for details.

Returns

props
[ImageProperties] A dataclass filled with standardized image metadata.

Notes

Reading properties with OpenCV involves decoding pixel data, because OpenCV doesn't provide a direct way to access metadata.

imageio.plugins.opencv.OpenCVPlugin.metadata

OpenCVPlugin.**metadata**(*index: Optional[int] = None*, *exclude_applied: bool = True*) → Dict[str, Any]

Format-specific metadata.

> **Warning:** OpenCV does not support reading metadata. When called, this function will raise a `NotImplementedError`.

Parameters

index
[int] This parameter has no effect.

exclude_applied
[bool] This parameter has no effect.

Pixel Formats (Colorspaces)

OpenCV is known to process images in BGR; however, most of the python ecosystem (in particular matplotlib and other pydata libraries) use the RGB. As such, images are converted to RGB, RGBA, or grayscale (where applicable) by default.

3.2.11 imageio.plugins.pillow

Read/Write images using Pillow/PIL.

Backend Library: Pillow

Plugin that wraps the the Pillow library. Pillow is a friendly fork of PIL (Python Image Library) and supports reading and writing of common formats (jpg, png, gif, tiff, ...). For, the complete list of features and supported formats please refer to pillows official docs (see the Backend Library link).

Parameters

request
: [Request] A request object representing the resource to be operated on.

Methods

PillowPlugin.read(*[, index, mode, rotate, ...])	Parses the given URI and creates a ndarray from it.
PillowPlugin.write(ndimage, *[, mode, ...])	Write an ndimage to the URI specified in path.
PillowPlugin.iter(*[, mode, rotate, apply_gamma])	Iterate over all ndimages/frames in the URI
PillowPlugin.get_meta(*[, index])	

imageio.plugins.pillow.PillowPlugin.read

PillowPlugin.**read**(*, *index=None*, *mode=None*, *rotate=False*, *apply_gamma=False*) → ndarray

Parses the given URI and creates a ndarray from it.

Parameters

index
: [{integer}] If the ImageResource contains multiple ndimages, and index is an integer, select the index-th ndimage from among them and return it. If index is an ellipsis (...), read all ndimages in the file and stack them along a new batch dimension and return them. If index is None, this plugin reads the first image of the file (index=0) unless the image is a GIF or APNG, in which case all images are read (index=...).

mode
: [{str, None}] Convert the image to the given mode before returning it. If None, the mode will be left unchanged. Possible modes can be found at: https://pillow.readthedocs.io/en/stable/handbook/concepts.html#modes

rotate
: [{bool}] If set to `True` and the image contains an EXIF orientation tag, apply the orientation before returning the ndimage.

apply_gamma
[{bool}] If `True` and the image contains metadata about gamma, apply gamma correction to the image.

Returns

ndimage
[ndarray] A numpy array containing the loaded image data

Notes

If you open a GIF - or any other format using color pallets - you may wish to manually set the *mode* parameter. Otherwise, the numbers in the returned image will refer to the entries in the color pallet, which is discarded during conversion to ndarray.

imageio.plugins.pillow.PillowPlugin.write

PillowPlugin.**write**(*ndimage: Union[Sequence[Sequence[Sequence[Sequence[Sequence[Any]]]]], _SupportsArray[dtype], Sequence[_SupportsArray[dtype]], Sequence[Sequence[_SupportsArray[dtype]]], Sequence[Sequence[Sequence[_SupportsArray[dtype]]]], Sequence[Sequence[Sequence[Sequence[_SupportsArray[dtype]]]]], bool, int, float, complex, str, bytes, Sequence[Union[bool, int, float, complex, str, bytes]], Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]], Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]], Sequence[Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]]], List[Union[Sequence[Sequence[Sequence[Sequence[Sequence[Any]]]]], _SupportsArray[dtype], Sequence[_SupportsArray[dtype]], Sequence[Sequence[_SupportsArray[dtype]]], Sequence[Sequence[Sequence[_SupportsArray[dtype]]]], Sequence[Sequence[Sequence[Sequence[_SupportsArray[dtype]]]]], bool, int, float, complex, str, bytes, Sequence[Union[bool, int, float, complex, str, bytes]], Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]], Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]], Sequence[Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]]]]]]]*, *, *mode: Optional[str] = None*, *format: Optional[str] = None*, *is_batch: Optional[bool] = None*, ***kwargs*) → Optional[bytes]

Write an ndimage to the URI specified in path.

If the URI points to a file on the current host and the file does not yet exist it will be created. If the file exists already, it will be appended if possible; otherwise, it will be replaced.

If necessary, the image is broken down along the leading dimension to fit into individual frames of the chosen format. If the format doesn't support multiple frames, and IOError is raised.

Parameters

image
[ndarray or list] The ndimage to write. If a list is given each element is expected to be an ndimage.

mode
[str] Specify the image's color format. If None (default), the mode is inferred from the array's shape and dtype. Possible modes can be found at: https://pillow.readthedocs.io/en/stable/handbook/concepts.html#modes

format
[str] Optional format override. If omitted, the format to use is determined from the filename extension. If a file object was used instead of a filename, this parameter must always be used.

is_batch
[bool] Explicitly tell the writer that `image` is a batch of images (True) or not (False). If None, the writer will guess this from the provided `mode` or `image.shape`. While the latter often works, it may cause problems for small images due to aliasing of spatial and color-channel axes.

kwargs
[...] Extra arguments to pass to pillow. If a writer doesn't recognise an option, it is silently ignored. The available options are described in pillow's image format documentation for each writer.

Notes

When writing batches of very narrow (2-4 pixels wide) gray images set the `mode` explicitly to avoid the batch being identified as a colored image.

imageio.plugins.pillow.PillowPlugin.iter

`PillowPlugin.iter`(*, *mode: Optional[str] = None*, *rotate: bool = False*, *apply_gamma: bool = False*) → Iterator[ndarray]

Iterate over all ndimages/frames in the URI

Parameters

mode
[{str, None}] Convert the image to the given mode before returning it. If None, the mode will be left unchanged. Possible modes can be found at: https://pillow.readthedocs.io/en/stable/handbook/concepts.html#modes

rotate
[{bool}] If set to `True` and the image contains an EXIF orientation tag, apply the orientation before returning the ndimage.

apply_gamma
[{bool}] If `True` and the image contains metadata about gamma, apply gamma correction to the image.

imageio.plugins.pillow.PillowPlugin.get_meta

`PillowPlugin.get_meta`(*, *index=0*) → Dict[str, Any]

3.2.12 imageio.plugins.pillow_legacy

Read/Write images using pillow/PIL (legacy).

Backend Library: Pillow

Pillow is a friendly fork of PIL (Python Image Library) and supports reading and writing of common formats (jpg, png, gif, tiff, ...). While these docs provide an overview of some of its features, pillow is constantly improving. Hence, the complete list of features can be found in pillows official docs (see the Backend Library link).

Parameters for Reading

pilmode
: [str] (Available for all formates except GIF-PIL) From the Pillow documentation:

- 'L' (8-bit pixels, grayscale)
- 'P' (8-bit pixels, mapped to any other mode using a color palette)
- 'RGB' (3x8-bit pixels, true color)
- 'RGBA' (4x8-bit pixels, true color with transparency mask)
- 'CMYK' (4x8-bit pixels, color separation)
- 'YCbCr' (3x8-bit pixels, color video format)
- 'I' (32-bit signed integer pixels)
- 'F' (32-bit floating point pixels)

PIL also provides limited support for a few special modes, including 'LA' ('L' with alpha), 'RGBX' (true color with padding) and 'RGBa' (true color with premultiplied alpha).

When translating a color image to grayscale (mode 'L', 'I' or 'F'), the library uses the ITU-R 601-2 luma transform:

```
L = R * 299/1000 + G * 587/1000 + B * 114/1000
```

as_gray
: [bool] (Available for all formates except GIF-PIL) If True, the image is converted using mode 'F'. When *mode* is not None and *as_gray* is True, the image is first converted according to *mode*, and the result is then "flattened" using mode 'F'.

ignoregamma
: [bool] (Only available in PNG-PIL) Avoid gamma correction. Default True.

exifrotate
: [bool] (Only available in JPEG-PIL) Automatically rotate the image according to exif flag. Default True.

Parameters for saving

optimize
[bool] (Only available in PNG-PIL) If present and true, instructs the PNG writer to make the output file as small as possible. This includes extra processing in order to find optimal encoder settings.

transparency:
(Only available in PNG-PIL) This option controls what color image to mark as transparent.

dpi: tuple of two scalars
(Only available in PNG-PIL) The desired dpi in each direction.

pnginfo: PIL.PngImagePlugin.PngInfo
(Only available in PNG-PIL) Object containing text tags.

compress_level: int
(Only available in PNG-PIL) ZLIB compression level, a number between 0 and 9: 1 gives best speed, 9 gives best compression, 0 gives no compression at all. Default is 9. When `optimize` option is True `compress_level` has no effect (it is set to 9 regardless of a value passed).

compression: int
(Only available in PNG-PIL) Compatibility with the freeimage PNG format. If given, it overrides compress_level.

icc_profile:
(Only available in PNG-PIL) The ICC Profile to include in the saved file.

bits (experimental): int
(Only available in PNG-PIL) This option controls how many bits to store. If omitted, the PNG writer uses 8 bits (256 colors).

quantize:
(Only available in PNG-PIL) Compatibility with the freeimage PNG format. If given, it overrides bits. In this case, given as a number between 1-256.

dictionary (experimental): dict
(Only available in PNG-PIL) Set the ZLIB encoder dictionary.

prefer_uint8: bool
(Only available in PNG-PIL) Let the PNG writer truncate uint16 image arrays to uint8 if their values fall within the range [0, 255]. Defaults to true for legacy compatibility, however it is recommended to set this to false to avoid unexpected behavior when saving e.g. weakly saturated images.

quality
[scalar] (Only available in JPEG-PIL) The compression factor of the saved image (1..100), higher numbers result in higher quality but larger file size. Default 75.

progressive
[bool] (Only available in JPEG-PIL) Save as a progressive JPEG file (e.g. for images on the web). Default False.

optimize
[bool] (Only available in JPEG-PIL) On saving, compute optimal Huffman coding tables (can reduce a few percent of file size). Default False.

dpi
[tuple of int] (Only available in JPEG-PIL) The pixel density, `(x,y)`.

icc_profile
[object] (Only available in JPEG-PIL) If present and true, the image is stored with the provided ICC profile. If this parameter is not provided, the image will be saved with no profile attached.

exif
[dict] (Only available in JPEG-PIL) If present, the image will be stored with the provided raw EXIF data.

subsampling
[str] (Only available in JPEG-PIL) Sets the subsampling for the encoder. See Pillow docs for details.

qtables
[object] (Only available in JPEG-PIL) Set the qtables for the encoder. See Pillow docs for details.

quality_mode
[str] (Only available in JPEG2000-PIL) Either *"rates"* or *"dB"* depending on the units you want to use to specify image quality.

quality
[float] (Only available in JPEG2000-PIL) Approximate size reduction (if quality mode is *rates*) or a signal to noise ratio in decibels (if quality mode is *dB*).

loop
[int] (Only available in GIF-PIL) The number of iterations. Default 0 (meaning loop indefinitely).

duration
[{float, list}] (Only available in GIF-PIL) The duration (in seconds) of each frame. Either specify one value that is used for all frames, or one value for each frame. Note that in the GIF format the duration/delay is expressed in hundredths of a second, which limits the precision of the duration.

fps
[float] (Only available in GIF-PIL) The number of frames per second. If duration is not given, the duration for each frame is set to 1/fps. Default 10.

palettesize
[int] (Only available in GIF-PIL) The number of colors to quantize the image to. Is rounded to the nearest power of two. Default 256.

subrectangles
[bool] (Only available in GIF-PIL) If True, will try and optimize the GIF by storing only the rectangular parts of each frame that change with respect to the previous. Default False.

Notes

To enable JPEG 2000 support, you need to build and install the OpenJPEG library, version 2.0.0 or higher, before building the Python Imaging Library. Windows users can install the OpenJPEG binaries available on the OpenJPEG website, but must add them to their PATH in order to use PIL (if you fail to do this, you will get errors about not being able to load the `_imaging` DLL).

GIF images read with this plugin are always RGBA. The alpha channel is ignored when saving RGB images.

3.2.13 imageio.plugins.pyav

Read/Write Videos (and images) using PyAV.

Note: To use this plugin you need to have PyAV installed:

```
pip install av
```

This plugin wraps pyAV, a pythonic binding for the FFMPEG library. It is similar to our FFMPEG plugin but offers a more performant and robut interface and aims to superseed the FFMPEG plugin in the future.

Methods

Note: Check the respective function for a list of supported kwargs and detailed documentation.

PyAVPlugin.read(*[, index, format, ...])	Read frames from the video.
PyAVPlugin.iter(*[, format, ...])	Yield frames from the video.
PyAVPlugin.write(ndimage, *[, codec, ...])	Save a ndimage as a video.
PyAVPlugin.properties([index, format])	Standardized ndimage metadata.
PyAVPlugin.metadata([index, ...])	Format-specific metadata.

imageio.plugins.pyav.PyAVPlugin.read

PyAVPlugin.**read**(*, *index: int = Ellipsis*, *format: str = 'rgb24'*, *filter_sequence: Optional[List[Tuple[str, Union[str, dict]]]] = None*, *filter_graph: Optional[Tuple[dict, List]] = None*, *constant_framerate: Optional[bool] = None*, *thread_count: int = 0*, *thread_type: Optional[str] = None*) → ndarray

Read frames from the video.

If `index` is an integer, this function reads the index-th frame from the file. If `index` is ... (Ellipsis), this function reads all frames from the video, stacks them along the first dimension, and returns a batch of frames.

Parameters

index
[int] The index of the frame to read, e.g. `index=5` reads the 5th frame. If `...`, read all the frames in the video and stack them along a new, prepended, batch dimension.

format
[str] Set the returned colorspace. If not None (default: rgb24), convert the data into the given format before returning it. If `None` return the data in the encoded format if it can be expressed as a strided array; otherwise raise an Exception.

filter_sequence
[List[str, str, dict]] If not None, apply the given sequence of FFmpeg filters to each ndimage. Check the (module-level) plugin docs for details and examples.

filter_graph
[(dict, List)] If not None, apply the given graph of FFmpeg filters to each ndimage. The graph is given as a tuple of two dicts. The first dict contains a (named) set of nodes, and the second dict contains a set of edges between nodes of the previous dict. Check the (module-level) plugin docs for details and examples.

constant_framerate
[bool] If True assume the video's framerate is constant. This allows for faster seeking inside the file. If False, the video is reset before each read and searched from the beginning. If None (default), this value will be read from the container format.

thread_count
[int] How many threads to use when decoding a frame. The default is 0, which will set the number using ffmpeg's default, which is based on the codec, number of available cores, threadding model, and other considerations.

thread_type
[str] The threading model to be used. One of

- *"SLICE"*: threads assemble parts of the current frame
- *"FRAME"*: threads may assemble future frames
- None (default): Uses `"FRAME"` if `index=...` and ffmpeg's default otherwise.

Returns

frame
[np.ndarray] A numpy array containing loaded frame data.

Notes

Accessing random frames repeatedly is costly (O(k), where k is the average distance between two keyframes). You should do so only sparingly if possible. In some cases, it can be faster to bulk-read the video (if it fits into memory) and to then access the returned ndarray randomly.

The current implementation may cause problems for b-frames, i.e., bidirectionaly predicted pictures. I lack test videos to write unit tests for this case.

Reading from an index other than `...`, i.e. reading a single frame, currently doesn't support filters that introduce delays.

imageio.plugins.pyav.PyAVPlugin.iter

`PyAVPlugin.iter`(*, *format: str = 'rgb24'*, *filter_sequence: Optional[List[Tuple[str, Union[str, dict]]]] = None*, *filter_graph: Optional[Tuple[dict, List]] = None*, *thread_count: int = 0*, *thread_type: Optional[str] = None*) → ndarray

Yield frames from the video.

Parameters

frame
[np.ndarray] A numpy array containing loaded frame data.

format
[str] Convert the data into the given format before returning it. If None, return the data in the encoded format if it can be expressed as a strided array; otherwise raise an Exception.

filter_sequence
[List[str, str, dict]] Set the returned colorspace. If not None (default: rgb24), convert the data into the given format before returning it. If `None` return the data in the encoded format if it can be expressed as a strided array; otherwise raise an Exception.

filter_graph
[(dict, List)] If not None, apply the given graph of FFmpeg filters to each ndimage. The graph is given as a tuple of two dicts. The first dict contains a (named) set of nodes, and the second dict contains a set of edges between nodes of the previous dict. Check the (module-level) plugin docs for details and examples.

thread_count
[int] How many threads to use when decoding a frame. The default is 0, which will set the number using ffmpeg's default, which is based on the codec, number of available cores, threadding model, and other considerations.

thread_type
[str] The threading model to be used. One of

- *"SLICE"* (default): threads assemble parts of the current frame

- *"FRAME"*: threads may assemble future frames (faster for bulk reading)

Yields

frame
[np.ndarray] A (decoded) video frame.

imageio.plugins.pyav.PyAVPlugin.write

PyAVPlugin.**write**(*ndimage: Union[ndarray, List[ndarray]]*, ***, *codec: Optional[str] = None*, *is_batch: bool = True*, *fps: int = 24*, *in_pixel_format: str = 'rgb24'*, *out_pixel_format: Optional[str] = None*, *filter_sequence: Optional[List[Tuple[str, Union[str, dict]]]] = None*, *filter_graph: Optional[Tuple[dict, List]] = None*) → Optional[bytes]

Save a ndimage as a video.

Given a batch of frames (stacked along the first axis) or a list of frames, encode them and add the result to the ImageResource.

Parameters

ndimage
[ArrayLike, List[ArrayLike]] The ndimage to encode and write to the ImageResource.

codec
[str] The codec to use when encoding frames. Only needed on first write and ignored on subsequent writes.

is_batch
[bool] If True (default), the ndimage is a batch of images, otherwise it is a single image. This parameter has no effect on lists of ndimages.

fps
[str] The resulting videos frames per second.

in_pixel_format
[str] The pixel format of the incoming ndarray. Defaults to "rgb24" and can be any stridable pix_fmt supported by FFmpeg.

out_pixel_format
[str] The pixel format to use while encoding frames. If None (default) use the codec's default.

filter_sequence
[List[str, str, dict]] If not None, apply the given sequence of FFmpeg filters to each ndimage. Check the (module-level) plugin docs for details and examples.

filter_graph
[(dict, List)] If not None, apply the given graph of FFmpeg filters to each ndimage. The graph is given as a tuple of two dicts. The first dict contains a (named) set of nodes, and the second dict contains a set of edges between nodes of the previous dict. Check the (module-level) plugin docs for details and examples.

Returns

encoded_image
[bytes or None] If the chosen ImageResource is the special target `"<bytes>"` then write will return a byte string containing the encoded image data. Otherwise, it returns None.

Notes

When writing <bytes>, the video is finalized immediately after the first write call and calling write multiple times to append frames is not possible.

imageio.plugins.pyav.PyAVPlugin.properties

PyAVPlugin.**properties**(*index: int = Ellipsis*, **, format: str = 'rgb24'*) → *ImageProperties*

Standardized ndimage metadata.

Parameters

index
[int] The index of the ndimage for which to return properties. If ... (Ellipsis, default), return the properties for the resulting batch of frames.

format
[str] If not None (default: rgb24), convert the data into the given format before returning it. If None return the data in the encoded format if that can be expressed as a strided array; otherwise raise an Exception.

Returns

properties
[ImageProperties] A dataclass filled with standardized image metadata.

Notes

This function is efficient and won't process any pixel data.

The provided metadata does not include modifications by any filters (through `filter_sequence` or `filter_graph`).

imageio.plugins.pyav.PyAVPlugin.metadata

PyAVPlugin.**metadata**(*index: int = Ellipsis*, *exclude_applied: bool = True*, *constant_framerate: bool = True*) → Dict[str, Any]

Format-specific metadata.

Returns a dictionary filled with metadata that is either stored in the container, the video stream, or the frame's side-data.

Parameters

index
[int] If … (Ellipsis, default) return global metadata (the metadata stored in the container and video stream). If not …, return the side data stored in the frame at the given index.

exclude_applied
[bool] Currently, this parameter has no effect. It exists for compliance with the ImageIO v3 API.

constant_framerate
[bool] If True assume the video's framerate is constant. This allows for faster seeking inside the file. If False, the video is reset before each read and searched from the beginning. If None (default), this value will be read from the container format.

Returns

metadata
[dict] A dictionary filled with format-specific metadata fields and their values.

Additional methods available inside the *imopen* context:

PyAVPlugin.init_video_stream(codec, *[, ...])	Initialize a new video stream.
PyAVPlugin.write_frame(frame, *[, pixel_format])	Add a frame to the video stream.
PyAVPlugin.set_video_filter([...])	Set the filter(s) to use.
PyAVPlugin.container_metadata	Container-specific metadata.
PyAVPlugin.video_stream_metadata	Stream-specific metadata.

imageio.plugins.pyav.PyAVPlugin.init_video_stream

PyAVPlugin.**init_video_stream**(*codec: str*, *, *fps: int = 24*, *pixel_format: Optional[str] = None*) → None

Initialize a new video stream.

This function adds a new video stream to the ImageResource using the selected encoder (codec), framerate, and colorspace.

Parameters

codec
[str] The codec to use, e.g. `"x264"` or `"vp9"`.

fps
[str] The desired framerate of the video stream (frames per second).

pixel_format
[str] The pixel format to use while encoding frames. If None (default) use the codec's default.

imageio.plugins.pyav.PyAVPlugin.write_frame

PyAVPlugin.**write_frame**(*frame: ndarray*, *, *pixel_format: str = 'rgb24'*) → None

Add a frame to the video stream.

This function appends a new frame to the video. It assumes that the stream previously has been initialized. I.e., `init_video_stream` has to be called before calling this function for the write to succeed.

Parameters

frame
[np.ndarray] The image to be appended/written to the video stream.

pixel_format
[str] The colorspace (pixel format) of the incoming frame.

Notes

Frames may be held in a buffer, e.g., by the filter pipeline used during writing or by FFMPEG to batch them prior to encoding. Make sure to `.close()` the plugin or to use a context manager to ensure that all frames are written to the ImageResource.

imageio.plugins.pyav.PyAVPlugin.set_video_filter

PyAVPlugin.**set_video_filter**(*filter_sequence: Optional[List[Tuple[str, Union[str, dict]]]] = None*, *filter_graph: Optional[Tuple[dict, List]] = None*) → None

Set the filter(s) to use.

This function creates a new FFMPEG filter graph to use when reading or writing video. In the case of reading, frames are passed through the filter graph before begin returned and, in case of writing, frames are passed through the filter before being written to the video.

Parameters

filter_sequence
[List[str, str, dict]] If not None, apply the given sequence of FFmpeg filters to each ndimage. Check the (module-level) plugin docs for details and examples.

filter_graph
[(dict, List)] If not None, apply the given graph of FFmpeg filters to each ndimage. The graph is given as a tuple of two dicts. The first dict contains a (named) set of nodes, and the second dict contains a set of edges between nodes of the previous dict. Check the (module-level) plugin docs for details and examples.

Notes

Changing a filter graph with lag during reading or writing will currently cause frames in the filter queue to be lost.

imageio.plugins.pyav.PyAVPlugin.container_metadata

property PyAVPlugin.**container_metadata**

Container-specific metadata.

A dictionary containing metadata stored at the container level.

imageio.plugins.pyav.PyAVPlugin.video_stream_metadata

property PyAVPlugin.**video_stream_metadata**

Stream-specific metadata.

A dictionary containing metadata stored at the stream level.

Advanced API

In addition to the default ImageIO v3 API this plugin exposes custom functions that are specific to reading/writing video and its metadata. These are available inside the *imopen* context and allow fine-grained control over how the video is processed. The functions are documented above and below you can find a usage example:

```
import imageio.v3 as iio

with iio.imopen("test.mp4", "w", plugin="pyav") as file:
    file.init_video_stream("libx264")
    file.container_metadata["comment"] = "This video has a rotation flag."
    file.video_stream_metadata["rotate"] = "90"

    for _ in range(5):
        for frame in iio.imiter("imageio:newtonscradle.gif"):
            file.write_frame(frame)

meta = iio.immeta("test.mp4", plugin="pyav")
assert meta["comment"] == "This video has a rotation flag."
```

Pixel Formats (Colorspaces)

By default, this plugin converts the video into 8-bit RGB (called `rgb24` in ffmpeg). This is a useful behavior for many use-cases, but sometimes you may want to use the video's native colorspace or you may wish to convert the video into an entirely different colorspace. This is controlled using the `format` kwarg. You can use `format=None` to leave the image in its native colorspace or specify any colorspace supported by FFMPEG as long as it is stridable, i.e., as long as it can be represented by a single numpy array. Some useful choices include:

- rgb24 (default; 8-bit RGB)
- rgb48le (16-bit lower-endian RGB)
- bgr24 (8-bit BGR; openCVs default colorspace)
- gray (8-bit grayscale)
- yuv444p (8-bit channel-first YUV)

Further, FFMPEG maintains a list of available formats, albeit not as part of the narrative docs. It can be found here (warning: C source code).

Filters

On top of providing basic read/write functionality, this plugin allows you to use the full collection of video filters available in FFMPEG. This means that you can apply excessive preprocessing to your video before retrieving it as a numpy array or apply excessive post-processing before you encode your data.

Filters come in two forms: sequences or graphs. Filter sequences are, as the name suggests, sequences of filters that are applied one after the other. They are specified using the `filter_sequence` kwarg. Filter graphs, on the other hand, come in the form of a directed graph and are specified using the `filter_graph` kwarg.

Note: All filters are either sequences or graphs. If all you want is to apply a single filter, you can do this by specifying a filter sequence with a single entry.

A `filter_sequence` is a list of filters, each defined through a 2-element tuple of the form `(filter_name, filter_parameters)`. The first element of the tuple is the name of the filter. The second element are the filter parameters, which can be given either as a string or a dict. The string matches the same format that you would use when specifying the filter using the ffmpeg command-line tool and the dict has entries of the form `parameter:value`. For example:

```
import imageio.v3 as iio

# using a filter_parameters str
img1 = iio.imread(
    "imageio:cockatoo.mp4",
    plugin="pyav",
    filter_sequence=[
        ("rotate", "45*PI/180")
    ]
)

# using a filter_parameters dict
img2 = iio.imread(
    "imageio:cockatoo.mp4",
    plugin="pyav",
    filter_sequence=[
        ("rotate", {"angle":"45*PI/180", "fillcolor":"AliceBlue"})
    ]
)
```

A `filter_graph`, on the other hand, is specified using a `(nodes, edges)` tuple. It is best explained using an example:

```
img = iio.imread(
    "imageio:cockatoo.mp4",
    plugin="pyav",
    filter_graph=(
        {
            "split": ("split", ""),
            "scale_overlay":("scale", "512:-1"),
            "overlay":("overlay", "x=25:y=25:enable='between(t,1,8)'"),
        },
        [
            ("video_in", "split", 0, 0),
            ("split", "overlay", 0, 0),
            ("split", "scale_overlay", 1, 0),
            ("scale_overlay", "overlay", 0, 1),
            ("overlay", "video_out", 0, 0),
        ]
    )
)
```

The above transforms the video to have picture-in-picture of itself in the top left corner. As you can see, nodes are specified using a dict which has names as its keys and filter tuples as values; the same tuples as the ones used when defining a filter sequence. Edges are a list of a 4-tuples of the form `(node_out, node_in, output_idx, input_idx)` and specify which two filters are connected and which inputs/outputs should be used for this.

Further, there are two special nodes in a filter graph: `video_in` and `video_out`, which represent the graph's input and output respectively. These names can not be chosen for other nodes (those nodes would simply be overwritten), and for a graph to be valid there must be a path from the input to the output and all nodes in the graph must be connected.

While most graphs are quite simple, they can become very complex and we recommend that you read through the FFMPEG documentation and their examples to better understand how to use them.

3.2.14 imageio.plugins.simpleitk

Read/Write images using SimpleITK.

Backend: Insight Toolkit

Note: To use this plugin you have to install its backend:

```
pip install imageio[itk]
```

The ItkFormat uses the ITK or SimpleITK library to support a range of ITK-related formats. It also supports a few common formats (e.g. PNG and JPEG).

Parameters

None

3.2.15 imageio.plugins.spe

Read SPE files.

Backend: internal

This plugin supports reading files saved in the Princeton Instruments SPE file format.

Parameters for reading

char_encoding
: [str] Character encoding used to decode strings in the metadata. Defaults to "latin1".

check_filesize
: [bool] The number of frames in the file is stored in the file header. However, this number may be wrong for certain software. If this is *True* (default), derive the number of frames also from the file size and raise a warning if the two values do not match.

sdt_meta
: [bool] If set to *True* (default), check for special metadata written by the *SDT-control* software. Does not have an effect for files written by other software.

Metadata for reading

ROIs
[list of dict] Regions of interest used for recording images. Each dict has the "top_left" key containing x and y coordinates of the top left corner, the "bottom_right" key with x and y coordinates of the bottom right corner, and the "bin" key with number of binned pixels in x and y directions.

comments
[list of str] The SPE format allows for 5 comment strings of 80 characters each.

controller_version
[int] Hardware version

logic_output
[int] Definition of output BNC

amp_hi_cap_low_noise
[int] Amp switching mode

mode
[int] Timing mode

exp_sec
[float] Alternative exposure in seconds

date
[str] Date string

detector_temp
[float] Detector temperature

detector_type
[int] CCD / diode array type

st_diode
[int] Trigger diode

delay_time
[float] Used with async mode

shutter_control
[int] Normal, disabled open, or disabled closed

absorb_live
[bool] on / off

absorb_mode
[int] Reference strip or file

can_do_virtual_chip
[bool] True or False whether chip can do virtual chip

threshold_min_live
[bool] on / off

threshold_min_val
[float] Threshold minimum value

threshold_max_live
[bool] on / off

threshold_max_val
[float] Threshold maximum value

time_local
[str] Experiment local time

time_utc
[str] Experiment UTC time

adc_offset
[int] ADC offset

adc_rate
[int] ADC rate

adc_type
[int] ADC type

adc_resolution
[int] ADC resolution

adc_bit_adjust
[int] ADC bit adjust

gain
[int] gain

sw_version
[str] Version of software which created this file

spare_4
[bytes] Reserved space

readout_time
[float] Experiment readout time

type
[str] Controller type

clockspeed_us
[float] Vertical clock speed in microseconds

readout_mode
[[“full frame”, “frame transfer”, “kinetics”, “”]] Readout mode. Empty string means that this was not set by the Software.

window_size
[int] Window size for Kinetics mode

file_header_ver
[float] File header version

chip_size
[[int, int]] x and y dimensions of the camera chip

virt_chip_size
[[int, int]] Virtual chip x and y dimensions

pre_pixels
[[int, int]] Pre pixels in x and y dimensions

post_pixels
[[int, int],] Post pixels in x and y dimensions

geometric
[list of {“rotate”, “reverse”, “flip”}] Geometric operations

sdt_major_version
[int] (only for files created by SDT-control) Major version of SDT-control software

sdt_minor_version
[int] (only for files created by SDT-control) Minor version of SDT-control software

sdt_controller_name
[str] (only for files created by SDT-control) Controller name

exposure_time
[float] (only for files created by SDT-control) Exposure time in seconds

color_code
[str] (only for files created by SDT-control) Color channels used

detection_channels
[int] (only for files created by SDT-control) Number of channels

background_subtraction
[bool] (only for files created by SDT-control) Whether background subtraction war turned on

em_active
[bool] (only for files created by SDT-control) Whether EM was turned on

em_gain
[int] (only for files created by SDT-control) EM gain

modulation_active
[bool] (only for files created by SDT-control) Whether laser modulation ("attenuate") was turned on

pixel_size
[float] (only for files created by SDT-control) Camera pixel size

sequence_type
[str] (only for files created by SDT-control) Type of sequnce (standard, TOCCSL, arbitrary, ...)

grid
[float] (only for files created by SDT-control) Sequence time unit ("grid size") in seconds

n_macro
[int] (only for files created by SDT-control) Number of macro loops

delay_macro
[float] (only for files created by SDT-control) Time between macro loops in seconds

n_mini
[int] (only for files created by SDT-control) Number of mini loops

delay_mini
[float] (only for files created by SDT-control) Time between mini loops in seconds

n_micro
[int (only for files created by SDT-control)] Number of micro loops

delay_micro
[float (only for files created by SDT-control)] Time between micro loops in seconds

n_subpics
[int] (only for files created by SDT-control) Number of sub-pictures

delay_shutter
[float] (only for files created by SDT-control) Camera shutter delay in seconds

delay_prebleach
[float] (only for files created by SDT-control) Pre-bleach delay in seconds

bleach_time
[float] (only for files created by SDT-control) Bleaching time in seconds

recovery_time
[float] (only for files created by SDT-control) Recovery time in seconds

comment
[str] (only for files created by SDT-control) User-entered comment. This replaces the "comments" field.

datetime
[datetime.datetime] (only for files created by SDT-control) Combines the "date" and "time_local" keys. The latter two plus "time_utc" are removed.

modulation_script
[str] (only for files created by SDT-control) Laser modulation script. Replaces the "spare_4" key.

3.2.16 imageio.plugins.swf

Read/Write SWF files.

Backend: internal

Shockwave flash (SWF) is a media format designed for rich and interactive animations. This plugin makes use of this format to store a series of images in a lossless format with good compression (zlib). The resulting images can be shown as an animation using a flash player (such as the browser).

SWF stores images in RGBA format. RGB or grayscale images are automatically converted. SWF does not support meta data.

Parameters for reading

loop
[bool] If True, the video will rewind as soon as a frame is requested beyond the last frame. Otherwise, IndexError is raised. Default False.

Parameters for saving

fps
[int] The speed to play the animation. Default 12.

loop
[bool] If True, add a tag to the end of the file to play again from the first frame. Most flash players will then play the movie in a loop. Note that the imageio SWF Reader does not check this tag. Default True.

html
[bool] If the output is a file on the file system, write an html file (in HTML5) that shows the animation. Default False.

compress
[bool] Whether to compress the swf file. Default False. You probably don't want to use this. This does not decrease the file size since the images are already compressed. It will result in slower read and write time. The only purpose of this feature is to create compressed SWF files, so that we can test the functionality to read them.

3.2.17 imageio.plugins.tifffile

Read/Write TIFF files.

Backend: internal

Provides support for a wide range of Tiff images using the tifffile backend.

Parameters for reading

offset
: [int] Optional start position of embedded file. By default this is the current file position.

size
: [int] Optional size of embedded file. By default this is the number of bytes from the 'offset' to the end of the file.

multifile
: [bool] If True (default), series may include pages from multiple files. Currently applies to OME-TIFF only.

multifile_close
: [bool] If True (default), keep the handles of other files in multifile series closed. This is inefficient when few files refer to many pages. If False, the C runtime may run out of resources.

Parameters for saving

bigtiff
: [bool] If True, the BigTIFF format is used.

byteorder
: [{'<', '>'}] The endianness of the data in the file. By default this is the system's native byte order.

software
: [str] Name of the software used to create the image. Saved with the first page only.

Metadata for reading

planar_configuration
: [{'contig', 'planar'}] Specifies if samples are stored contiguous or in separate planes. By default this setting is inferred from the data shape. 'contig': last dimension contains samples. 'planar': third last dimension contains samples.

resolution_unit
: [int] The resolution unit stored in the TIFF tag. Usually 1 means no/unknown unit, 2 means dpi (inch), 3 means dpc (centimeter).

resolution
: [(float, float, str)] A tuple formatted as (X_resolution, Y_resolution, unit). The unit is a string representing one of the following units:

```
NONE        # No unit or unit unknown
INCH        # dpi
CENTIMETER  # cpi
MILLIMETER
MICROMETER
```

compression
[int] Value indicating the compression algorithm used, e.g. 5 is LZW, 7 is JPEG, 8 is deflate. If 1, data are uncompressed.

predictor
[int] Value 2 indicates horizontal differencing was used before compression, while 3 indicates floating point horizontal differencing. If 1, no prediction scheme was used before compression.

orientation
[{'top_left', 'bottom_right', ... }] Oriented of image array.

is_rgb
[bool] True if page contains a RGB image.

is_contig
[bool] True if page contains a contiguous image.

is_tiled
[bool] True if page contains tiled image.

is_palette
[bool] True if page contains a palette-colored image and not OME or STK.

is_reduced
[bool] True if page is a reduced image of another image.

is_shaped
[bool] True if page contains shape in image_description tag.

is_fluoview
[bool] True if page contains FluoView MM_STAMP tag.

is_nih
[bool] True if page contains NIH image header.

is_micromanager
[bool] True if page contains Micro-Manager metadata.

is_ome
[bool] True if page contains OME-XML in image_description tag.

is_sgi
[bool] True if page contains SGI image and tile depth tags.

is_mdgel
[bool] True if page contains md_file_tag tag.

is_mediacy
[bool] True if page contains Media Cybernetics Id tag.

is_stk
[bool] True if page contains UIC2Tag tag.

is_lsm
[bool] True if page contains LSM CZ_LSM_INFO tag.

description
[str] Image description

description1
[str] Additional description

is_imagej
[None or str] ImageJ metadata

software
[str] Software used to create the TIFF file

datetime
[datetime.datetime] Creation date and time

Metadata for writing

photometric
[{'minisblack', 'miniswhite', 'rgb'}] The color space of the image data. By default this setting is inferred from the data shape.

planarconfig
[{'contig', 'planar'}] Specifies if samples are stored contiguous or in separate planes. By default this setting is inferred from the data shape. 'contig': last dimension contains samples. 'planar': third last dimension contains samples.

resolution
[(float, float) or ((int, int), (int, int))] X and Y resolution in dots per inch as float or rational numbers.

description
[str] The subject of the image. Saved with the first page only.

compress
[int] Values from 0 to 9 controlling the level of zlib (deflate) compression. If 0, data are written uncompressed (default).

predictor
[bool] If True, horizontal differencing is applied before compression. Note that using an int literal 1 actually means no prediction scheme will be used.

volume
[bool] If True, volume data are stored in one tile (if applicable) using the SGI image_depth and tile_depth tags. Image width and depth must be multiple of 16. Few software can read this format, e.g. MeVisLab.

writeshape
[bool] If True, write the data shape to the image_description tag if necessary and no other description is given.

extratags: sequence of tuples
Additional tags as [(code, dtype, count, value, writeonce)].

code
[int] The TIFF tag Id.

dtype
[str] Data type of items in 'value' in Python struct format. One of B, s, H, I, 2I, b, h, i, f, d, Q, or q.

count
[int] Number of data values. Not used for string values.

value
[sequence] 'Count' values compatible with 'dtype'.

writeonce
[bool] If True, the tag is written to the first page only.

Notes

Global metadata is stored with the first frame in a TIFF file. Thus calling `Format.Writer.set_meta_data()` after the first frame was written has no effect. Also, global metadata is ignored if metadata is provided via the *meta* argument of `Format.Writer.append_data()`.

If you have installed tifffile as a Python package, imageio will attempt to use that as backend instead of the bundled backend. Doing so can provide access to new performance improvements and bug fixes.

CHAPTER

FOUR

CONTRIBUTING

4.1 The ImageIO Plugin API

Here you can find documentation on how to write your own plugin to allow ImageIO to access a new backend. Plugins are quite object oriented, and the relevant classes and their interaction are documented here:

imageio.core.Format(name, description[, ...])	Represents an implementation to read/write a particular file format
imageio.core.Request(uri, mode, *[, ...])	ImageResource handling utility.

4.1.1 imageio.core.Format

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 *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

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.

Methods

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.
get_reader(request)	Return a reader object that can be used to read data and info from the given file.
get_writer(request)	Return a writer object that can be used to write data and info to the given file.

Attribute and Method Details

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.

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.

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.

4.1.2 imageio.core.Request

class imageio.core.**Request**(*uri*, *mode*, *, *extension=None*, *format_hint: Optional[str] = None*, ***kwargs*)

ImageResource handling utility.

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, zip-files, 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
Name of the ImageResource.

firstbytes
The first 256 bytes of the file.

format_hint

kwargs
The dict of keyword arguments supplied by the user.

mode
The mode of the request.

Methods

finish()	Wrap up this request.
get_file()	Get a file object for the resource associated with this request.
get_local_filename()	If the filename is an existing file on this filesystem, return that.
get_result()	For internal use.

Attribute and Method Details

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

Name of the ImageResource.

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.

firstbytes

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

format_hint

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.

finish() → None

Wrap up this request.

Finishes any pending reads or writes, closes any open files and frees any resources allocated by this request.

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 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()

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.

Note: You can always check existing plugins if you want to see examples.

4.1.3 What methods to implement

To implement a new plugin, create a new class that inherits from *imageio.core.Format*. and implement the following functions:

imageio.core.Format.__init__(name, description)	Initialize the Plugin.
imageio.core.Format._can_read(request)	Check if Plugin can read from ImageResource.
imageio.core.Format._can_write(request)	Check if Plugin can write to ImageResource.

imageio.core.Format.__init__

Format.**__init__**(*name*, *description*, *extensions=None*, *modes=None*)

Initialize the Plugin.

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.

imageio.core.Format._can_read

Format.**_can_read**(*request*)

Check if Plugin can read from ImageResource.

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.

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.

Parameters

request
[Request] A request that can be used to access the ImageResource and obtain metadata about it.

Returns

can_read
[bool] True if the plugin can read from the ImageResource, False otherwise.

imageio.core.Format._can_write

`Format._can_write(request)`

Check if Plugin can write to ImageResource.

Parameters

request
[Request] A request that can be used to access the ImageResource and obtain metadata about it.

Returns

can_read
[bool] True if the plugin can write to the ImageResource, False otherwise.

Further, each format contains up to two nested classes; one for reading and one for writing. To support reading and/or writing, the respective classes need to be defined.

For reading, create a nested class that inherits from `imageio.core.Format.Reader` and that implements the following functions:

- **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 writing, create a nested class that inherits from `imageio.core.Format.Writer` and implement the following functions:

- **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.

4.1.4 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.
```

(continues on next page)

(continued from previous page)

```
"""

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.

    """

```

(continues on next page)

(continued from previous page)

```
    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.
        #
        # 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

        if request.extension in self.extensions:
            return True

    def _can_write(self, request):
```

(continues on next page)

(continued from previous page)

```
        # 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.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

```

(continues on next page)

(continued from previous page)

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

    def _get_length(self):
        # 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
```

(continues on next page)

(continued from previous page)

```
    # -- 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.
```

(continues on next page)

(continued from previous page)

```
            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)
```

Becoming a contributor can take as little as 2 minutes (e.g., via a small doc PR) and doesn't require vast coding experience. We are open for contributions in many ways and, if you would like to get involved but aren't sure where to start, create a new issue and we will help you get started. In particular, we are also extremely happy for any help with

- bugfixes,
- improving our documentation,
- improving our DevOp pipeline,
- improving the design of our website,
- new backends/plugins,
- and new features.

Check the relevant section below for a guide on how to get started in any of these areas.

4.2 Small Changes to the Documentation

Note: A small documentation pull request includes changes that only affect a single file. Examples include fixing typos or broken links, rewriting a paragraph for clarity, or adding examples.

A small doc PR is very quick and doesn't require any local setup on your end. All you need is a GitHub account. Here are the steps involved:

1. Navigate to the page you wish to improve.
2. Click on the "Edit this Page" button on the right side pannel.
3. Make your changes to the file that opens.

4. At the bottom of the page under "Commit Changes" choose a descriptive commit message (e.g., "fixed typo in examples").
5. Click the button "Propose Changes".
6. That's it. The ImageIO team thanks you for your contribution!

4.3 Large Changes to the Documentation

Installation

For large changes to the documentation, you will have to set up a local development environment with the documentation dependencies installed:

```
# 1. Fork the ImageIO repository
# 2. If you want to use a virtual environment for your setup, create and activate it now.

git clone https://github.com/<replace_by_your_username>/imageio.git
cd imageio
pip install -e .[doc]

# create and checkout a new branch for your contribution
# Note: replace <branch_name> with a descriptive, but short, name
git checkout -b <branch_name>
git push --set-upstream origin <branch_name>

# building the docs locally on Linux or MacOS
# Note: to rebuild the docs, execute this line again
sphinx-build ./docs ./build

# building the docs locally on Windows
# Note: to rebuild the docs, execute this line again
sphinx-build .\docs .\build
```

Developing the PR

Next, you will have to add your modifications to the docs; remember to commit frequently. You are welcome to reach out during this process if you want feedback or have any doubts. Further, you may wish to create a draft PR with your changes, so that we can discuss more easily.

We use sphinx to generate the website and the API has narrative docs. Documentation of the actual functions is autogenerated using numpydoc, and is linked to from the narrative API docs. Further, the list of supported formats, too, is autogenerated using a custom sphinx extension (`imageio_ext.py`); essentially, it parses the respective format RST files using jinja and inserts the formats from `imageio.config.known_extensions`.

Submitting the PR

Once you are happy with your changes, push your changes to your local fork, head over to the main repo and create a new PR. In the description, briefly summarize what the changes are about (there is no fixed format). If we discussed them before in an issue, add a reference to that issue, so that we can track it. From there, we will help you by reviewing your changes, discussing any changes and eventually merging your work into ImageIO.

Thank you for contributing!

4.4 Fixing a Bug

Note: We currently don't have a dedicated maintainer that uses MacOS as their primary development platform. If you would like to suggest yourself, please get in touch, e.g., via a new issue.

The first step to fixing a bug is to open a new issue that describes the bug (ideally including a [mwe](https://en.wikipedia.org/wiki/Minimal_working_example)) to discuss where the fix should live. Alternatively, if an issue already exists, leave a comment to let us know you want to work on it.

In general, a bugfix follows the usual open-source routine. Here is a rough outline:

```
# Fork the ImageIO repository

git clone https://github.com/<replace_by_fork_location>/imageio.git
cd imageio

# Note: replace <plugin_name> with the plugin that has the bug to
#       install optional dependencies
pip install -e .[dev,<plugin_name>]

# Note: replace <branch_name> with a descriptive, but short, name
git checkout -b <branch_name>
git push --set-upstream origin <branch_name>

pytest  # run the existing test suite to verify install

# Add a unit test that reproduces the faulty behavior

# Apply the changes to fix the bug. Remember to commit frequently.

# check if your changes are covered by tests
coverage run -m pytest
coverage report -m

# If changes are not fully covered, add test(s) to cover the
# missing branches.

# Submit a PR. In the description, reference the abovementioned
# issue and give a brief description of the changes.
```

Thank you for contributing!

4.5 Adding a new Feature

Note: We currently don't have a dedicated maintainer that uses MacOS as their primary development platform. If you would like to suggest yourself, please get in touch, e.g., via a new issue.

The first step to contributing a new feature is to open a new issue on our issue page so we can discuss how the feature should look like, if it can work with the relevant backends, and fits within the scope of the library.

Once that is out of the way, the procedure usually follows the following steps:

```
# Fork the ImageIO repository

git clone https://github.com/<replace_by_fork_location>/imageio.git
cd imageio

# Note: replace <plugin_name> with the plugin that has the bug to
#       install optional dependencies
pip install -e .[dev,<plugin_name>]

# Note: replace <branch_name> with a descriptive, but short, name
git checkout -b <branch_name>
git push --set-upstream origin <branch_name>

pytest  # run the existing test suite to verify install

# Add the desired/discussed functionality

# For each new function you've created add a descriptive docstring
# compliant with numpydoc

# Add an example to ``docs/examples.rst`` showing off the new
# feature

# Add unit tests to verify that the feature does what it is
# intended to do

# check if your changes are covered by tests
coverage run -m pytest
coverage report -m

# If changes are not fully covered, add test(s) to cover the
# missing branches.

# Submit a PR. In the description, reference the feature issue and
# give a brief description of the changes.
```

Thank you for contributing!

4.6 Webdesign

Warning: We currently don't have a dedicated maintainer for this area. If you would like to suggest yourself, please get in touch, e.g., via a new issue.

There is currently no established way of suggestion webdesign related changes. If you have any suggestions regarding

- the structure/layout of the docs
- improvements of the template or its config
- styling of the website
- (other webdesign related items)

Please open a new issue and we will discuss with you.

4.7 DevOps

Before you go and improve our tooling, please start by first opening a new issue and discussing your idea with us. We ask this, because we want to make sure that there is at least one maintainer familiar with the technology you want to use. If anything breaks in the CI/CD pipeline, it will have a direct impact on our release cycle, and we would like to make sure that at least one maintainer is able to step in and fix it in a timely fashion.

Installation

While you may be able to complete some devops related contributions in GitHub itself, a local setup will perform better in most situations:

```
# 1. Fork the ImageIO repository

git clone https://github.com/<replace_by_your_username>/imageio.git
cd imageio
pip install -e .

# create and checkout a new branch for your contribution
# Note: replace <branch_name> with a descriptive, but short, name
git checkout -b <branch_name>
git push --set-upstream origin <branch_name>
```

Developing the PR

Do your changes, and remember to commit frequently. If you want to test your changes against our CI/CD provider (GH Actions), you can submit a draft PR, which we will review and run. Alternatively, you configure your local fork to execute actions and develop against it. You can find our worflows in the `.github` folder.

Continous Integration (CI): The core of our CI is a matrix test across all major OSes and supported python versions running a testsuite based on `pytest`. Additionally, we track linting errors (black + flake8), coverage (codecov), and have readthedocs build a preview of the documentation.

Continous Deployment (CD): Our continous deployment pipeline checks weekly (Monday night, EU time) for any changes to the repository. If there are any, it uses python-semantic-release to figure out if the version should bump and a new release should be made. If so, all tests are run against the master branch, the version is bumped, and a new release is made and published to GitHub and PyPI. Further, we have a feedstock on conda-forge which manages releases to conda.

Submitting the PR

Once you are happy with your changes, push your changes to your local fork, head over to the main repo and create a new PR. In the description, briefly summarize what the changes are about (there is no fixed format) and add a reference to the issue in which we discussed them. From there, we will help you by reviewing your changes, discussing them and eventually merging your work into ImageIO.

Thank you for contributing!

4.8 Implementing a new Plugin

Plugins allow integration of a new backends into ImageIO. They can exist independently of ImageIO; however, we encourage you to contribute any plugins back upstream. This way others, too, can benefit from using the new backend and you will get compatibility guarantees by virtue of the backend becoming part of our test suite.

Note: These instructions assume that you indeed wish to contribute the plugin.

If you don't, then you won't need a dev installation of ImageIO and can write the plugin directly. In this case, you will have to pass the plugin class to API calls using the plugin kwarg. For example:

```
import imageio as iio
from my_plugin import PluginClass

iio.imread(ImageResource, plugin=PluginClass)
```

Installation

To develop a new plugin, you can start off with a simple def install:

```
# 1. Fork the ImageIO repository

git clone https://github.com/<replace_by_your_username>/imageio.git
cd imageio
pip install -e .[dev]

# create and checkout a new branch for your contribution
# Note: replace <branch_name> with a descriptive, but short, name
git checkout -b <branch_name>
git push --set-upstream origin <branch_name>

# verify installation
pytest
```

Develop the Plugin

To write a new plugin, you have to create a new class that follows our plugin API, which is documented below:

v2 plugin docs

imageio.plugins	Here you can find documentation on how to write your own plugin to allow ImageIO to access a new backend.

v3 plugin docs

imageio.core.v3_plugin_api.PluginV3(request)	A ImageIO Plugin.
imageio.core.v3_plugin_api.ImageProperties(...)	Standardized Metadata

4.8.1 imageio.core.v3_plugin_api.PluginV3

class imageio.core.v3_plugin_api.**PluginV3**(*request:* Request)

A ImageIO Plugin.

This is an abstract plugin that documents the v3 plugin API interface. A plugin is an adapter/wrapper around a backend that converts a request from iio.core (e.g., read an image from file) into a sequence of instructions for the backend that fullfill the request.

Plugin authors may choose to subclass this class when implementing a new plugin, but aren't obliged to do so. As long as the plugin class implements the interface (methods) described below the ImageIO core will treat it just like any other plugin.

Parameters

request
[iio.Request] A request object that represents the users intent. It provides a standard interface to access the various ImageResources and serves them to the plugin as a file object (or file). Check the docs for details.

****kwargs**
[Any] Additional configuration arguments for the plugin or backend. Usually these match the configuration arguments available on the backend and are forwarded to it.

Raises

InitializationError
During `__init__` the plugin tests if it can fulfill the request. If it can't, e.g., because the request points to a file in the wrong format, then it should raise an `InitializationError` and provide a reason for failure. This reason may be reported to the user.

ImportError
Plugins will be imported dynamically when listed in `iio.config.known_plugins` to fullfill requests. This way, users only have to load plugins/backends they actually use. If this plugin's backend is not installed, it should raise an `ImportError` either during module import or during class construction.

Notes

Upon successful construction the plugin takes ownership of the provided request. This means that it is the plugin's responsibility to call request.finish() to close the resource when it is no longer needed.

Plugins _must_ implement a context manager that closes and cleans any resources held by the plugin upon exit.

Attributes

request

Methods

close()	Close the ImageResource.
iter()	Iterate the ImageResource.
metadata([index, exclude_applied])	Format-Specific ndimage metadata.
properties([index])	Standardized ndimage metadata.
read(*[, index])	Read a ndimage.
write(ndimage)	Write a ndimage to a ImageResource.

Attribute and Method Details

request

close() → None

Close the ImageResource.

This method allows a plugin to behave similar to the python build-in `open`:

```
image_file = my_plugin(Request, "r")
...
image_file.close()
```

It is used by the context manager and deconstructor below to avoid leaking ImageResources. If the plugin has no other cleanup to do it doesn't have to overwrite this method itself and can rely on the implementation below.

iter() → Iterator[ndarray]

Iterate the ImageResource.

This method returns a generator that yields ndimages in the order in which they appear in the file. This is roughly equivalent to:

```
idx = 0
while True:
    try:
        yield self.read(index=idx)
    except ValueError:
        break
```

It works very similar to `read`, and you can consult the documentation of that method for additional information on desired behavior.

Parameters

****kwargs**
[Any] The iter method may accept any number of plugin-specific keyword arguments to further customize the reading/iteration behavior. Usually these match the arguments available on the backend and are forwarded to it.

Yields

ndimage
[np.ndarray] A ndimage containing decoded pixel data (sometimes called bitmap).

See also:

PluginV3.read

metadata(*index: int = 0*, *exclude_applied: bool = True*) → Dict[str, Any]

Format-Specific ndimage metadata.

The method reads metadata stored in the ImageResource and returns it as a python dict. The plugin is free to choose which name to give a piece of metadata; however, if possible, it should match the name given by the format. There is no requirement regarding the fields a plugin must expose; however, if a plugin does expose any,``exclude_applied`` applies to these fields.

If the plugin does return metadata items, it must check the value of `exclude_applied` before returning them. If `exclude applied` is True, then any metadata item that would be applied to an ndimage returned by `read` (or `iter`) must not be returned. This is done to avoid confusion; for example, if an ImageResource defines the ExIF rotation tag, and the plugin applies the rotation to the data before returning it, then `exclude_applied` prevents confusion on whether the tag was already applied or not.

The *kwarg* `index` behaves similar to its counterpart in `read` with one exception: If the `index` is None, then global metadata is returned instead of returning a combination of all metadata items. If there is no global metadata, the Plugin should return an empty dict or raise an exception.

Parameters

index
[int] If the ImageResource contains multiple ndimages, and index is an integer, select the index-th ndimage from among them and return its metadata. If index is an ellipsis (...), return global metadata. If index is None, the plugin decides the default.

exclude_applied
[bool] If True (default), do not report metadata fields that the plugin would apply/consume while reading the image.

Returns

metadata
[dict] A dictionary filled with format-specific metadata fields and their values.

properties(*index: int = 0*) → *ImageProperties*

Standardized ndimage metadata.

Parameters

index
[int] If the ImageResource contains multiple ndimages, and index is an

integer, select the index-th ndimage from among them and return its properties. If index is an ellipsis (...), read all ndimages in the file and stack them along a new batch dimension and return their properties. If index is None, the plugin decides the default.

Returns

properties
[ImageProperties] A dataclass filled with standardized image metadata.

read(*, *index: int = 0*) → ndarray

Read a ndimage.

The `read` method loads a (single) ndimage, located at `index` from the requested ImageResource.

It is at the plugin's descretion to decide (and document) what constitutes a single ndimage. A sensible way to make this decision is to choose based on the ImageResource's format and on what users will expect from such a format. For example, a sensible choice for a TIFF file produced by an ImageJ hyperstack is to read

it as a volumetric ndimage (1 color dimension followed by 3 spatial dimensions). On the other hand, a sensible choice for a MP4 file produced by Davinci Resolve is to treat each frame as a ndimage (2 spatial dimensions followed by 1 color dimension).

The value `index=None` is special. It requests the plugin to load all ndimages in the file and stack them along a new first axis. For example, if a MP4 file is read with `index=None` and the plugin identifies single frames as ndimages, then the plugin should read all frames and stack them into a new ndimage which now contains a time axis as its first axis. If a PNG file (single image format) is read with `index=None` the plugin does a very similar thing: It loads all ndimages in the file (here it's just one) and stacks them along a new first axis, effectively prepending an axis with size 1 to the image. If a plugin does not wish to support `index=None` it should set a more sensible default and raise a `ValueError` when requested to read using `index=None`.

Parameters

index
[int] If the ImageResource contains multiple ndimages, and index is an integer, select the index-th ndimage from among them and return it. If index is an ellipsis (...), read all ndimages in the file and stack them along a new batch dimension. If index is None, let the plugin decide. If the index is out of bounds a `ValueError` is raised.

****kwargs**
[Any] The read method may accept any number of plugin-specific keyword arguments to further customize the read behavior. Usually these match the arguments available on the backend and are forwarded to it.

Returns

ndimage
[np.ndarray] A ndimage containing decoded pixel data (sometimes called bitmap).

Notes

The ImageResource from which the plugin should read is managed by the provided request object. Directly accessing the managed ImageResource is _not_ permitted. Instead, you can get FileLike access to the ImageResource via request.get_file().

If the backend doesn't support reading from FileLike objects, you can request a temporary file to pass to the backend via `request.get_local_filename()`. This is, however, not very performant (involves copying the Request's content into a temporary file), so you should avoid doing this whenever possible. Consider it a fallback method in case all else fails.

write(*ndimage: Union[Sequence[Sequence[Sequence[Sequence[Sequence[Any]]]]], _SupportsArray[dtype], Sequence[_SupportsArray[dtype]], Sequence[Sequence[_SupportsArray[dtype]]], Sequence[Sequence[Sequence[_SupportsArray[dtype]]]], Sequence[Sequence[Sequence[Sequence[_SupportsArray[dtype]]]]], bool, int, float, complex, str, bytes, Sequence[Union[bool, int, float, complex, str, bytes]], Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]], Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]], Sequence[Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]]], List[Union[Sequence[Sequence[Sequence[Sequence[Sequence[Any]]]]], _SupportsArray[dtype], Sequence[_SupportsArray[dtype]], Sequence[Sequence[_SupportsArray[dtype]]], Sequence[Sequence[Sequence[_SupportsArray[dtype]]]], Sequence[Sequence[Sequence[Sequence[_SupportsArray[dtype]]]]], bool, int, float, complex, str, bytes, Sequence[Union[bool, int, float, complex, str, bytes]], Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]], Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]], Sequence[Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]]]]*) → Optional[bytes]

Write a ndimage to a ImageResource.

The `write` method encodes the given ndimage into the format handled by the backend and writes it to the ImageResource. It overwrites any content that may have been previously stored in the file.

If the backend supports only a single format then it must check if the ImageResource matches that format and raise an exception if not. Typically, this should be done during initialization in the form of a `InitializationError`.

If the backend supports more than one format it must determine the requested/desired format. Usually this can be done by inspecting the ImageResource (e.g., by checking `request.extension`), or by providing a mechanism to explicitly set the format (perhaps with a - sensible - default value). If the plugin can not determine the desired format, it **must not** write to the ImageResource, but raise an exception instead.

If the backend supports at least one format that can hold multiple ndimages it should be capable of handling ndimage batches and lists of ndimages. If the `ndimage` input is a list of ndimages, the plugin should not assume that the ndimages are not stackable, i.e., ndimages may have different shapes. Otherwise, the `ndimage` may be a batch of multiple ndimages stacked along the first axis of the array. The plugin must be able to discover this, either automatically or via additional *kwargs*. If there is ambiguity in the process, the plugin must clearly document what happens in such cases and, if possible, describe how to resolve this ambiguity.

Parameters

ndimage
[ArrayLike] The ndimage to encode and write to the current ImageResource.

****kwargs**
[Any] The write method may accept any number of plugin-specific keyword arguments to customize the writing behavior. Usually these match the arguments available on the backend and are forwarded to it.

Returns

encoded_image
[bytes or None] If the chosen ImageResource is the special target `"<bytes>"` then write should return a byte string containing the encoded image data. Otherwise, it returns None.

Notes

The ImageResource to which the plugin should write to is managed by the provided request object. Directly accessing the managed ImageResource is _not_ permitted. Instead, you can get FileLike access to the ImageResource via request.get_file().

If the backend doesn't support writing to FileLike objects, you can request a temporary file to pass to the backend via `request.get_local_filename()`. This is, however, not very performant (involves copying the Request's content from a temporary file), so you should avoid doing this whenever possible. Consider it a fallback method in case all else fails.

4.8.2 imageio.core.v3_plugin_api.ImageProperties

class imageio.core.v3_plugin_api.**ImageProperties**(*shape: Tuple[int, ...]*, *dtype: dtype*, *is_batch: bool = False*, *spacing: Optional[tuple] = None*)

Standardized Metadata

ImageProperties represent a set of standardized metadata that is available under the same name for every supported format. If the ImageResource (or format) does not specify the value, a sensible default value is chosen instead.

Attributes

shape
[Tuple[int, ...]] The shape of the loaded ndimage.

dtype
[np.dtype] The dtype of the loaded ndimage.

is_batch
[bool] If True, the first dimension of the ndimage represents a batch dimension along which several images are stacked.

spacing
[Tuple] A tuple describing the spacing between pixels along each axis of the ndimage. If the spacing is uniform along an axis the value corresponding to that axis is a single float. If the spacing is non-uniform, the value corresponding to that axis is a tuple in which the i-th element indicates the spacing between the i-th and (i+1)-th pixel along that axis.

Methods

Attribute and Method Details

is_batch: bool = False

spacing: Optional[tuple] = None

shape: Tuple[int, ...]

dtype: dtype

The file itself should be placed into our plugins folder at `imageio\plugins\your_plugin.py`.

Declare Dependencies

Plugins typically have at least one external dependency: the backend they use to do the decoding/encoding. This dependency (and any others a plugin may have) have to be declared in our `setup.py`.

All plugins start out as optional dependencies (but may be promoted if they turn out stable and useful ;)). As such, to declare dependencies of a plugin add an item to the extra_requires dict. The key name typically matches the name of the backend, and the value is a list of dependencies:

```
extras_require = {
    # ...
    "my_backend": ["my_backend", ...],
}
```

That said, a plugin can assume that its dependencies are installed, i.e., it doesn't have to explicitly assert that imports resolve. We catch any ImportError internally and use it to inform the user that they have to install missing dependencies via:

```
pip install imageio[my_backend]
```

Register the Plugin

Registering the plugin with ImageIO enables various kinds of auto-magic. Currently this involves two steps:

1. Register the plugin as a known_plugin
2. Associate supported extensions with the plugin

First, add the plugin to the list of known plugins. This allows ImageIO to try the plugin in case no better suited one can be found and also enables all other optimizations. For this, add a new item to the dict in `imageio.config.plugins.py`. The key is the name under which the plugin will be known (usually the name of the backend) and the value is an instance of `PluginConfig`, for example:

```
known_plugins["my_plugin"] = PluginConfig(
    name="my_plugin",  # (same as key)
    class_name="MyPluginClass",  # the name of the class
    module_name="imageio.plugins.my_plugin" # where the class is implemented
    is_legacy: bool = True,  # True if plugin follows V2 API
    install_name: str = "my_backend",  # name of the optional dependency
)
```

For more details on the PluginConfig class, check the classes docstring.

Second, if the plugin adds support for any new formats that were not previously supported by ImageIO, declare those formats in `imageio.config.extensions.py`. For this, add items to the `extension_list`; items are instances of the `FileExtension` class:

```
FileExtension(
    name="Full Name of Format",
    extension=".file_extension", # e.g. ".png"
    priority=["my_plugin"], # a list of plugins that supports reading this format
),
```

Plugins listed in `priority` are assumed to be able to read the declared format. Further, ImageIO will prefer plugins that appear earlier over plugins that appear later in the list, i.e., they are tried first.

Finally, for each format that is already supported by other plugins, add the new plugin at the end of the `priority` list. (This avoids breaking existing downstream code.)

Document the Plugin

```
# build the docs
sphinx-build ./docs ./build  # MacOS / Linux
sphinx-build .\docs .\build  # Windows
```

Beyond the plugin itself, you will have to write documentation for it that tells others what features are available and how to use it. In ImageIO classes are documented using numpydoc, so they should follow numpy's documentation style. Most importantly, you will have to add a module docstring to the plugin file (check the other plugins for examples), which will be used as the entrypoint for your plugin's documentation.

Once you have written something, hook the documentation into our docs. For this add it's import path (imageio.plugins.your_module_name) in `docs\reference\index.rst`. It should be inside the autosummary block that lists all plugins.

Test the Plugin

```
# run tests
pytest  # run all
pytest path/to/test/file.py  # run specific module
pytest path/to/test/file.py::test_name_of_test  # run specific test

# check coverage
coverage run -m pytest  # update the coverage logs
coverage report -m  # report coverage in shell
```

To test your plugin, create a new test file at `tests\test_myplugin.py`. In the file, define functions that have their name begin with `test_` (example: `def test_png_reading():`) and fill them with code that uses your plugin.

Our main requirement for tests of new plugins is that they cover the full plugin. Ideally, they also test readiing and writing of all supported formats, but this is not strictly necessary. Check the commands above for how to run tests and check test coverage of your plugin.

Submitting the PR

Once you are happy with the plugin, push your changes to your local fork, head over to the main repo and create a new PR. In the description, briefly summarize what the plugin does (there is no fixed format) and, if it exists, add a reference to the issue in which the plugin is discussed. From there, we will help you by reviewing your changes, discussing them and eventually merging your plugin into ImageIO.

Thank you for contributing!

4.9 Adding a missing Format

Adding a new format (or updating an existing one) can be done directly in GitHub. Navigate to [this page](https://github.com/imageio/imageio/blob/master/imageio/config/extensions.py) and click on the "edit this file" button (looks like a pen). From here, either edit the existing format, e.g., by adding a new backend that supports it in `priority`, or by add a new format. For this add this snippet to the bottom of the `extension_list`:

```
FileExtension(
    name="<Full Name of File Format>", # e.g., Hasselblad raw
    extension="<extension>", # e.g., .3fr
    priority=<list of supporting backend names>, # e.g., ["RAW-FI"]
)
```

Thank you for contributing!

4.10 Full Developer Setup

Note: This section is intended for library maintainers and people that plan to make multiple contributions of various kinds.

If you plan to make a series of contributions, we recommend a development installation with all plugins:

```
# 1. Fork the ImageIO repository
# 2. Optional: Create and activate your virtual environment of choice
```

(continues on next page)

(continued from previous page)

```
# install
git clone https://github.com/<replace_by_fork_location>/imageio.git
cd imageio
pip install -e .[full]

# download test images and run all tests
pytest

# build the docs
sphinx-build ./docs ./build  # MacOS / Linux
sphinx-build .\docs .\build  # Windows

# check coverage
coverage run -m pytest  # update the coverage logs
coverage report -m  # report coverage in shell
```

PYTHON MODULE INDEX

i

INDEX

Symbols

A

C

D

E

F

G

H

I

K

M

N

P

R

S

V

W