plotly | Graphing Libraries (https://plotly.com/)(/graphing-libraries/)
utm_campaign=studio_cloud_launch&utm_content=sidebar)

Imshow in Python

How to display image data in Python with Plotly.

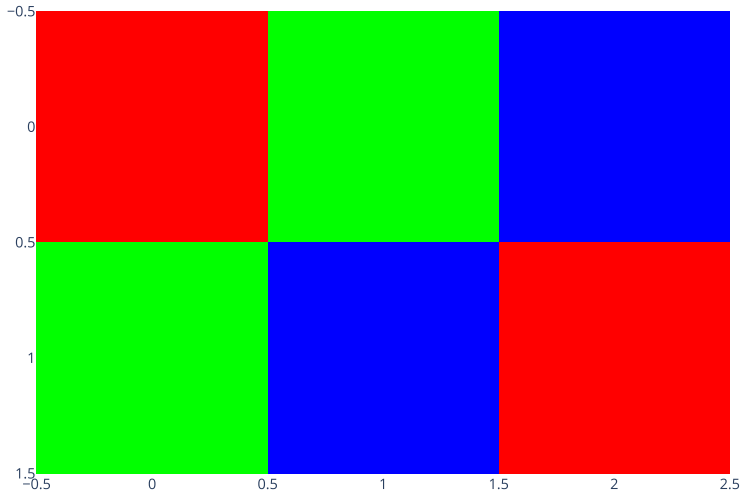
Plotly Studio: Transform any dataset into an interactive data application in minutes with AI. Sign up for early access now. (https://plotly.com/studio/?utm_medium=graphing_libraries&utm_campaign=studio_early_access&utm_content=sidebar)

This tutorial shows how to display and explore image data. If you would like instead a logo or static image, use `go.layout.Image` as explained here (/python/images).

Displaying RGB image data with px.imshow

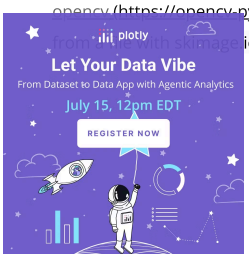
`px.imshow` displays multichannel (RGB) or single-channel ("grayscale") image data.

```
import plotly.express as px
import numpy as np
img_rgb = np.array([[[255, 0, 0], [0, 255, 0], [0, 0, 255]],
                    [[0, 255, 0], [0, 0, 255], [255, 0, 0]]
                   ], dtype=np.uint8)
fig = px.imshow(img_rgb)
fig.show()
```

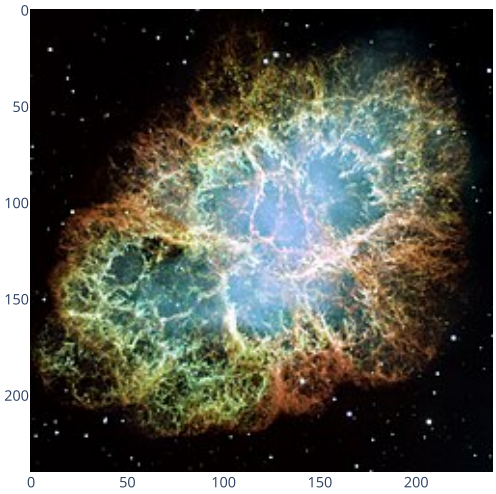


Read image arrays from image files

In order to create a numerical array to be passed to `px.imshow`, you can use a third-party library like PIL (https://pillow.readthedocs.io/en/stable/reference/Image.html#PIL.Image.open), scikit-image (https://scikit-image.org/docs/dev/user_guide/getting_started.html) or opencv (https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_gui/py_image_display/py_image_display.html). We show below how to open an image from file with `skimage.io.imread`, and alternatively how to load a demo image from `skimage.data`.



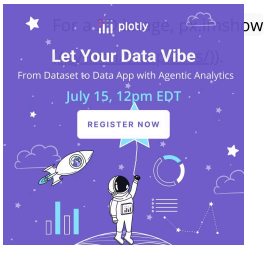
```
import plotly.express as px
from skimage import io
img = io.imread('https://upload.wikimedia.org/wikipedia/commons/thumb/0/00/Crab_Nebula.jpg/240px-Crab_Nebula.jpg')
fig = px.imshow(img)
fig.show()
```



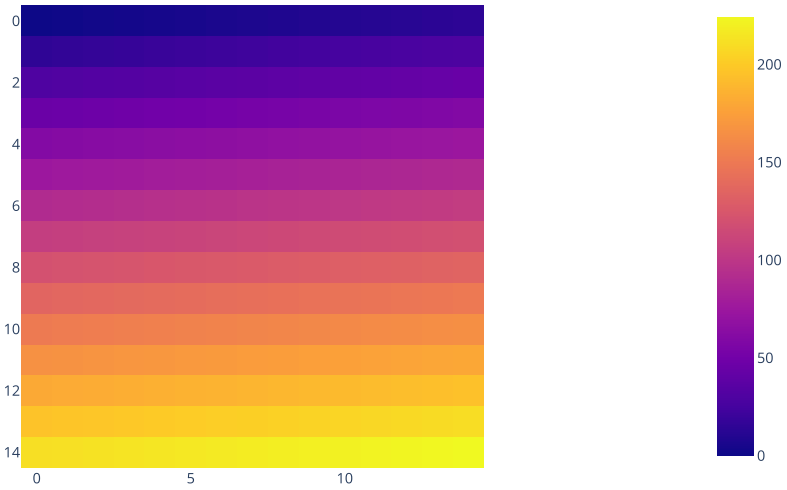

```
import plotly.express as px
from skimage import data
img = data.astronaut()
fig = px.imshow(img, binary_format="jpeg", binary_compression_level=0)
fig.show()
```

Display single-channel 2D data as a heatmap

For a single-channel image, `imshow` uses a colorscale to map scalar data to colors. The default colorscale is the one of the active template (see the tutorial on templates



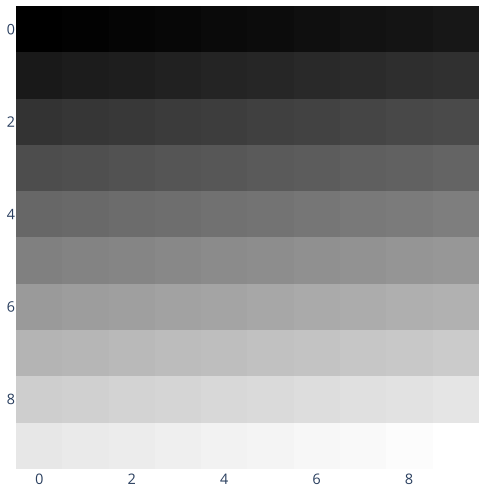
```
import plotly.express as px
import numpy as np
img = np.arange(15**2).reshape((15, 15))
fig = px.imshow(img)
fig.show()
```

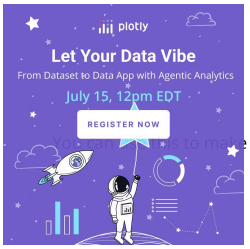


Choose the colorscale to display a single-channel image

You can customize the continuous color scale (/python/colorscales/) just like with any other Plotly Express function. However, color_continuous_scale is ignored when using binary_string=True, since the image is always represented as grayscale (and no colorbar is displayed).

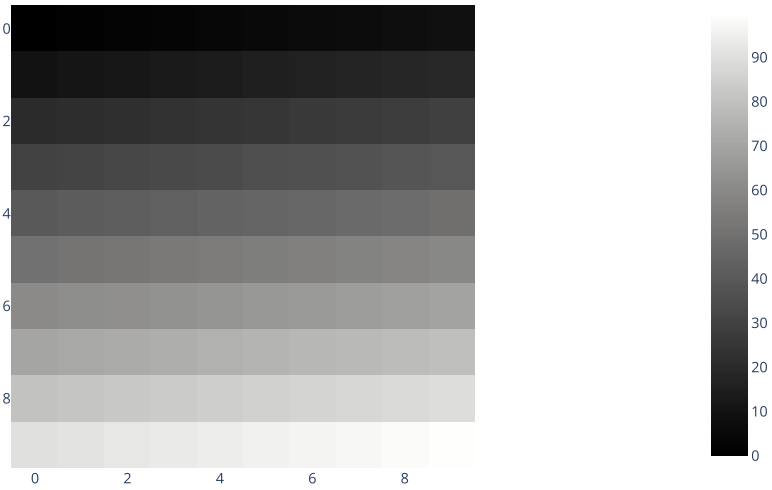
```
import plotly.express as px
import numpy as np
img = np.arange(100).reshape((10, 10))
fig = px.imshow(img, binary_string=True)
fig.show()
```





You can also use Plotly Express to make the image grayscale as well:

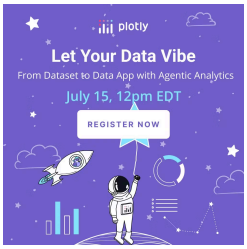
```
import plotly.express as px
import numpy as np
img = np.arange(100).reshape((10, 10))
fig = px.imshow(img, color_continuous_scale='gray')
fig.show()
```



Hiding the colorbar and axis labels

See the continuous color (/python/colorscales/) and cartesian axes (/python/axes/) pages for more details.

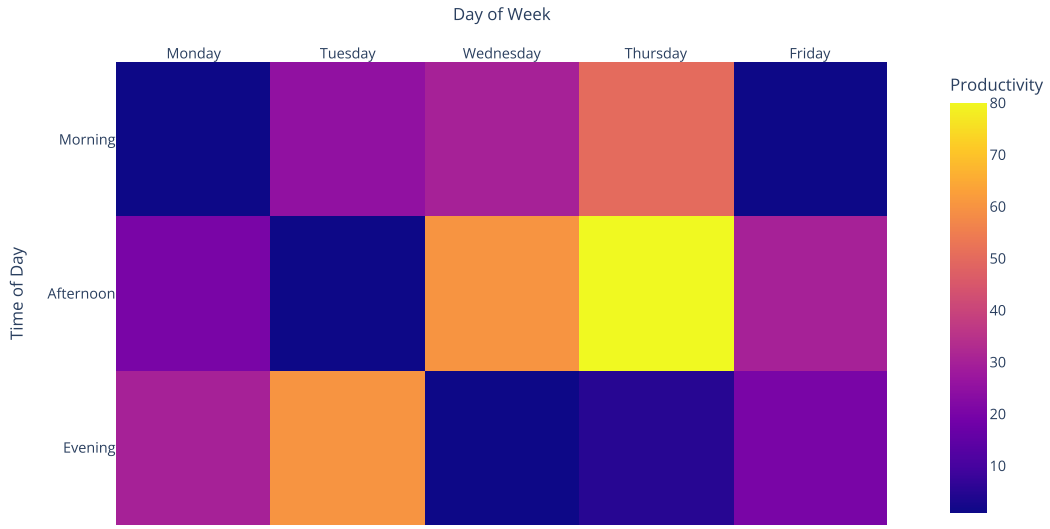
```
import plotly.express as px
from skimage import data
img = data.camera()
fig = px.imshow(img, color_continuous_scale='gray')
fig.update_layout(coloraxis_showscale=False)
fig.update_xaxes(showticklabels=False)
fig.update_yaxes(showticklabels=False)
fig.show()
```



Customizing the axes and labels on a single-channel image

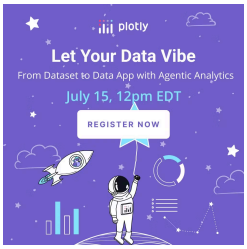
You can use the `x`, `y` and `labels` arguments to customize the display of a heatmap, and use `.update_xaxes()` to move the x axis tick labels to the top:

```
import plotly.express as px
data=[[1, 25, 30, 50, 1], [20, 1, 60, 80, 30], [30, 60, 1, 5, 20]]
fig = px.imshow(data,
                labels=dict(x="Day of Week", y="Time of Day", color="Productivity"),
                x=['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday'],
                y=['Morning', 'Afternoon', 'Evening']
               )
fig.update_xaxes(side="top")
fig.show()
```

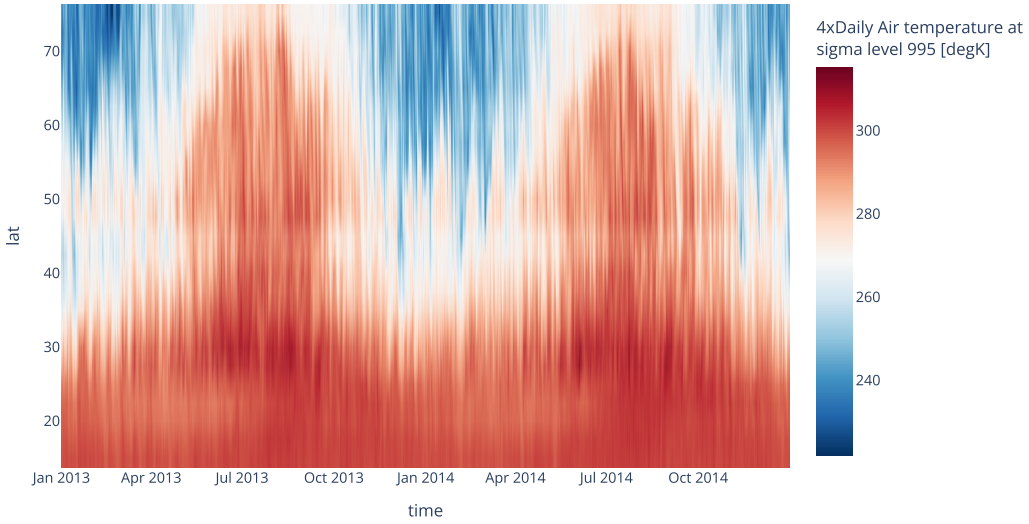


Display an xarray image with px.imshow

xarrays (http://xarray.pydata.org/en/stable/) are labeled arrays (with labeled axes and coordinates). If you pass an xarray image to `px.imshow`, its axes labels and coordinates will be used for axis titles. If you don't want this behavior, you can pass `img.values` which is a NumPy array if `img` is an xarray. Alternatively, you can override axis titles hover labels and colorbar title using the `labels` attribute, as above.



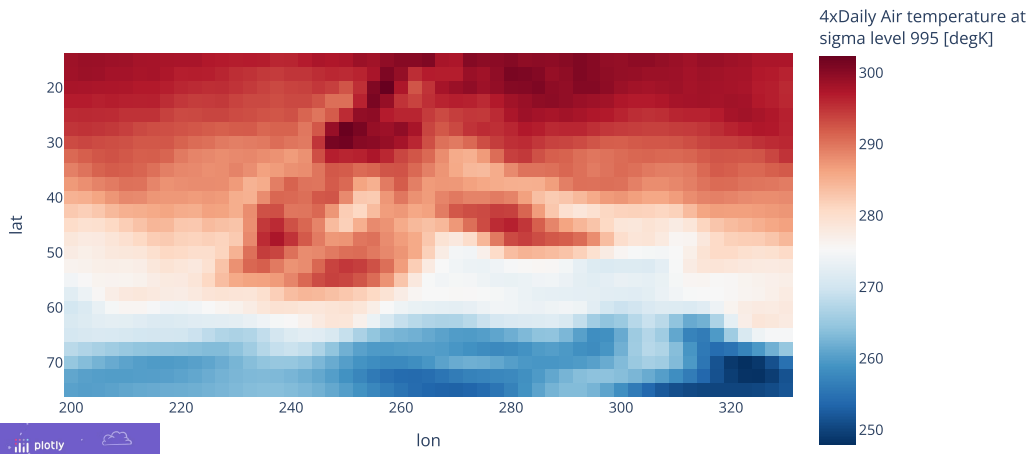
```
import plotly.express as px
import xarray as xr
# Load xarray from dataset included in the xarray tutorial
airtemps = xr.tutorial.open_dataset('air_temperature').air.sel(lon=250.0)
fig = px.imshow(airtemps.T, color_continuous_scale='RdBu_r', origin='lower')
fig.show()
```

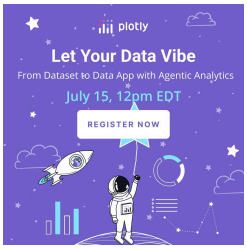


Display an xarray image with square pixels

For xarrays, by default `px.imshow` does not constrain pixels to be square, since axes often correspond to different physical quantities (e.g. time and space), contrary to a plain camera image where pixels are square (most of the time). If you want to impose square pixels, set the parameter `aspect` to "equal" as below.

```
import plotly.express as px
import xarray as xr
airtemps = xr.tutorial.open_dataset('air_temperature').air.isel(time=500)
colorbar_title = airtemps.attrs['var_desc'] + '<br>(%s)'%airtemps.attrs['units']
fig = px.imshow(airtemps, color_continuous_scale='RdBu_r', aspect='equal')
fig.show()
```





Display multichannel image data with go.Image

It is also possible to use the go.Image trace from the low-level graph_objects API in order to display image data. Note that go.Image only accepts multichannel images. For single-channel images, use go.Heatmap (/python/heatmaps).

Note that the go.Image trace is different from the go.layout.Image class, which can be used for adding background images or logos to figures (/python/images).

imshow

ingle-

a single-

show
e pixels
th

g to

the

mshow
ta
traces
am of

s

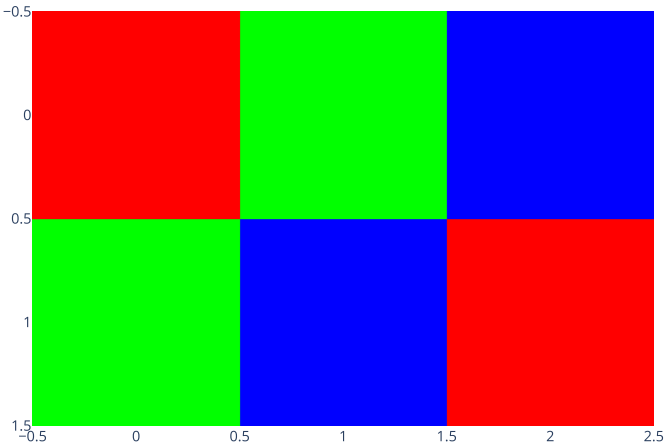
g
binary

of the

d

s with

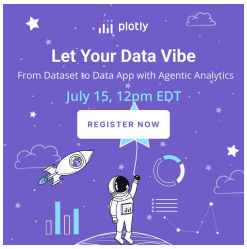
```
import plotly.graph_objects as go
img_rgb = [[[255, 0, 0], [0, 255, 0], [0, 0, 255]],
           [[0, 255, 0], [0, 0, 255], [255, 0, 0]]]
fig = go.Figure(go.Image(z=img_rgb))
fig.show()
```



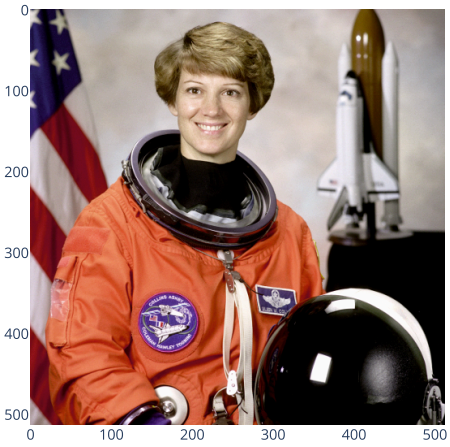
Passing image data as a binary string to go.Image

The z parameter of go.Image passes image data in the form of an array or a list of numerical values, but it is also possible to use the source parameter, which takes a b64 binary string. Thanks to png or jpg compression, using source is a way to reduce the quantity of data passed to the browser, and also to reduce the serialization time of the figure, resulting in increased performance.

Note than an easier way of creating binary strings with px.imshow is explained below.



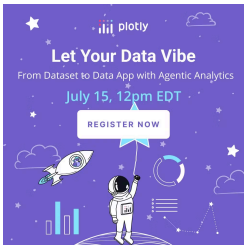
```
import plotly.graph_objects as go
from skimage import data
from PIL import Image
import base64
from io import BytesIO

img = data.astronaut()  # numpy array
pil_img = Image.fromarray(img) # PIL image object
prefix = ""
with BytesIO() as stream:
    pil_img.save(stream, format="png")
    base64_string = prefix + base64.b64encode(stream.getvalue()).decode("utf-8")
fig = go.Figure(go.Image(source=base64_string))
fig.show()
```



Defining the data range covered by the color range with zmin and zmax

The data range and color range are mapped together using the parameters `zmin` and `zmax` of `px.imshow` or `go.Image`, which correspond respectively to the data values mapped to black [0, 0, 0] and white [255, 255, 255], or to the extreme colors of the colorscale in the case of single-channel data.

For `go.Image`, `zmin` and `zmax` need to be given for all channels, whereas it is also possible to pass a scalar value (used for all channels) to `px.imshow`.



imshow

ingle-

a single-

show
e pixels
th

g to

the

mshow
ta
traces
am of

s

g
binary

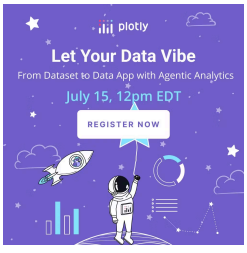
of the

d

s with

```
import plotly.express as px
from skimage import data
img = data.astronaut()
# Increase contrast by clipping the data range between 50 and 200
fig = px.imshow(img, zmin=50, zmax=200)
# We customize the hovertemplate to show both the data and the color values
# See https://plotly.com/python/hover-text-and-formatting/#customize-tooltip-text-with-a-hovertemplate
#fig.update_traces(hovertemplate="x: %{x} <br> y: %{y} <br> z: %{z} <br> color: %{color}")
fig.show()
```


```
import plotly.express as px
from skimage import data
img = data.astronaut()
# Stretch the contrast of the red channel only, resulting in a more red image
fig = px.imshow(img, zmin=[50, 0, 0], zmax=[200, 255, 255])
fig.show()
```



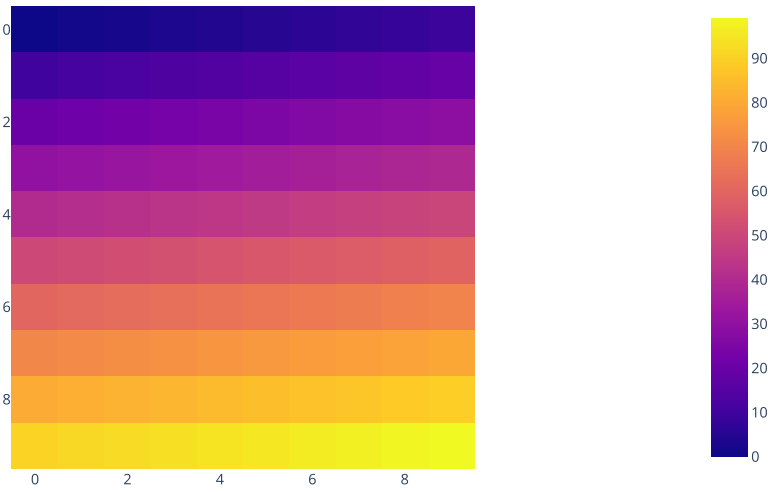
Automatic contrast rescaling in px.imshow

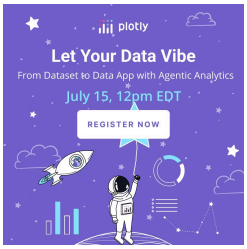
When zmin and zmax are not specified, the contrast_rescaling arguments determines how zmin and zmax are computed. For contrast_rescaling='minmax', the extrema of the data range are used. For contrast_rescaling='infer', a heuristic based on the data type is used:

- for integer data types, zmin and zmax correspond to the extreme values of the data type, for example 0 and 255 for uint8, 0 and 65535 for uint16, etc.
- for float numbers, the maximum value of the data is computed, and zmax is 1 if the max is smaller than 1, 255 if the max is smaller than 255, etc. (with higher thresholds 2**16 - 1 and 2**32 -1).

These two modes can be used for single- and multichannel data. The default value is to use 'minmax' for single-channel data (as in a Heatmap trace) and infer for multi-channel data (which often consist of uint8 data). In the example below we override the default value by setting contrast_rescaling='infer' for a single-channel image.

```
import plotly.express as px
img = np.arange(100, dtype=np.uint8).reshape((10, 10))
fig = px.imshow(img, contrast_rescaling='infer')
fig.show()
```



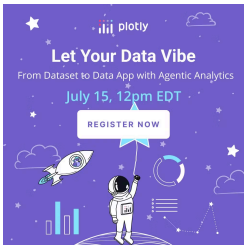


Ticks and margins around image data

```
import plotly.express as px
from skimage import data
img = data.astronaut()
fig = px.imshow(img)
fig.update_layout(width=400, height=400, margin=dict(l=10, r=10, b=10, t=10))
fig.update_xaxes(showticklabels=False).update_yaxes(showticklabels=False)
fig.show()
```

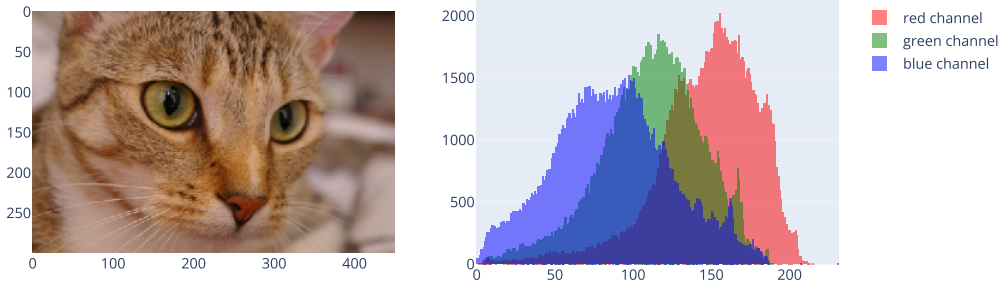
Combining image charts and other traces

```
import plotly.express as px
import plotly.graph_objects as go
from skimage import data
img = data.camera()
fig = px.imshow(img, color_continuous_scale='gray')
fig.add_trace(go.Contour(z=img, showscale=False,
                         contours=dict(start=0, end=70, size=70, coloring='lines'),
                         line_width=2))
fig.add_trace(go.Scatter(x=[230], y=[100], marker=dict(color='red', size=16)))
fig.show()
```



Displaying an image and the histogram of color values

```
from plotly.subplots import make_subplots
from skimage import data
img = data.chelsea()
fig = make_subplots(1, 2)
# We use go.Image because subplots require traces, whereas px functions return a figure
fig.add_trace(go.Image(z=img), 1, 1)
for channel, color in enumerate(['red', 'green', 'blue']):
    fig.add_trace(go.Histogram(x=img[..., channel].ravel(), opacity=0.5,
                               marker_color=color, name='%s channel' %color), 1, 2)
fig.update_layout(height=400)
fig.show()
```



imshow and datashader

Arrays of rasterized values build by datashader can be visualized using imshow. See the plotly and datashader tutorial (/python/datashader/) for examples on how to use plotly and datashader.

Annotating image traces with shapes

introduced in plotly 4.7

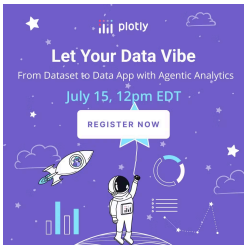
It can be useful to add shapes to an image trace, for highlighting an object, drawing bounding boxes as part of a machine learning training set, or identifying seeds for a segmentation algorithm.

In order to enable shape drawing, you need to

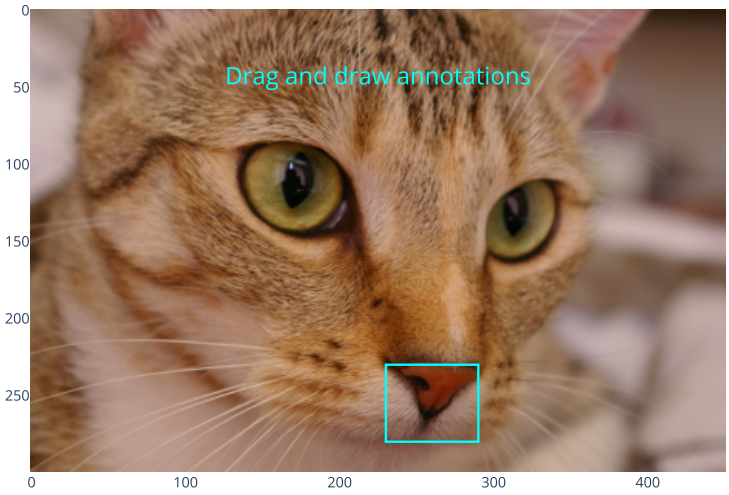
- define a dragmode corresponding to a drawing tool ('drawline','drawopenpath', 'drawclosedpath', 'drawcircle', or 'drawrect')
- add modebar buttons corresponding to the drawing tools you wish to use.

The style of new shapes is specified by the `newshape` layout attribute. Shapes can be selected and modified after they have been drawn. More details and examples are given in the tutorial on shapes (/python/shapes#drawing-shapes-on-cartesian-plots).

Drawing or modifying a shape triggers a `relayout` event, which can be captured by a callback inside a Dash application (https://dash.plotly.com/interactive-graphing).



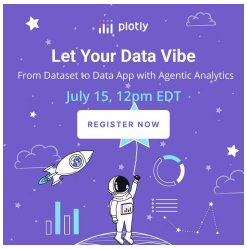
```
import plotly.express as px
from skimage import data
img = data.chelsea()
fig = px.imshow(img)
fig.add_annotation(
    x=0.5,
    y=0.9,
    text="Drag and draw annotations",
    xref="paper",
    yref="paper",
    showarrow=False,
    font_size=20, font_color='cyan')
# Shape defined programatically
fig.add_shape(
    type='rect',
    x0=230, x1=290, y0=230, y1=280,
    xref='x', yref='y',
    line_color='cyan'
)
# Define dragmode, newshape parameters, amd add modebar buttons
fig.update_layout(
    dragmode='drawrect',
    newshape=dict(line_color='cyan'))
fig.show(config={'modeBarButtonsToAdd':['drawline',
                                        'drawopenpath',
                                        'drawclosedpath',
                                        'drawcircle',
                                        'drawrect',
                                        'eraseshape'
                                       ]})
```



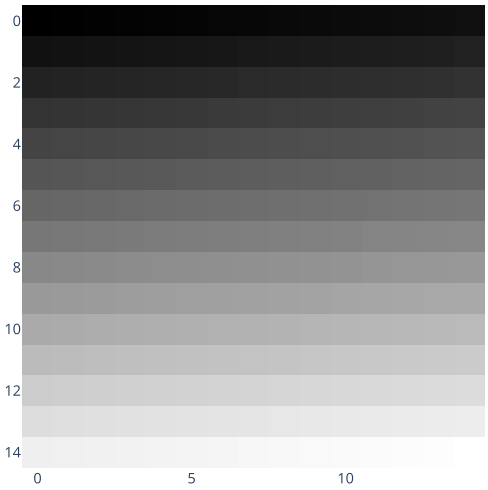
Passing image data as a binary string

introduced in plotly.py 4.10

px.imshow can pass the data to the figure object either as a list of numerical values, or as a png binary string which is passed directly to the browser. While the former solution offers more flexibility (values can be of float or int type, while values are rescaled to the range [0-255] for an image string), using a binary string is usually faster for large arrays. The parameter binary_string controls whether the image is passed as a png string (when True) or a list of values (False). Its default value is True for multi-channel images and False for single-channel images. When binary_string=True, image data are always represented using a go.Image trace.



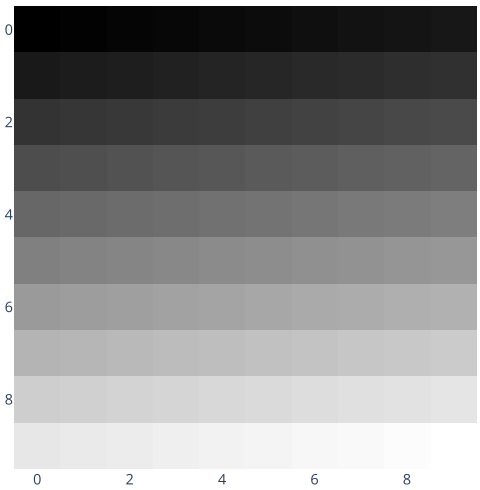
```
import plotly.express as px
import numpy as np
img = np.arange(15**2).reshape((15, 15))
fig = px.imshow(img, binary_string=True)
fig.show()
```

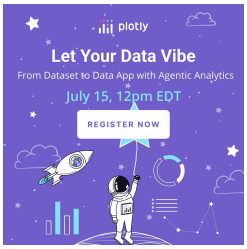


Contrast rescaling im imshow with binary string

When the image is passed to the plotly figure as a binary string (which is the default mode for RGB images), and when the image is rescaled to adjust the contrast (for example when setting zmin and zmax), the original intensity values are not passed to the plotly figure and therefore no intensity value is displayed in the hover.

```
import plotly.express as px
from skimage import data
import numpy as np
img = np.arange(100).reshape((10, 10))
fig = px.imshow(img, binary_string=True)
# You can check that only x and y are displayed in the hover
# You can use a hovertemplate to override the hover information
# See https://plotly.com/python/hover-text-and-formatting/#customize-tooltip-text-with-a-hovertemplate
fig.show()
```



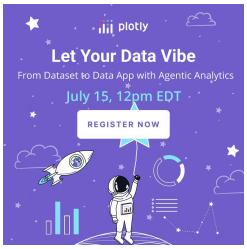


You can set `binary_string=False` if you want the intensity value to appear in the hover even for a rescaled image. In the example below we also modify the hovertemplate to display both z (the data of the original image array) and color (the pixel value displayed in the figure).

```
import plotly.express as px
from skimage import data
img = data.chelsea()
# Increase contrast by clipping the data range between 50 and 200
fig = px.imshow(img, binary_string=False, zmin=50, zmax=200)
# We customize the hovertemplate to show both the data and the color values
# See https://plotly.com/python/hover-text-and-formatting/#customize-tooltip-text-with-a-hovertemplate
fig.update_traces(hovertemplate="x: %{x} <br> y: %{y} <br> z: %{z} <br> color: %{color}")
fig.show()
```

Changing the level of compression of the binary string in px.imshow

The `binary_compression_level` parameter controls the level of compression to be used by the backend creating the png string. Two different backends can be used, `pypng` (which is a dependency of `plotly` and is therefore always available), and `pil` for Pillow, which is often more performant. The compression level has to be between 0 (no compression) and 9 (highest compression), although increasing the compression above 4 and 5 usually only offers diminishing returns (no significant compression gain, at the cost of a longer execution time).



```
import plotly.express as px
from skimage import data
img = data.camera()
for compression_level in range(0, 9):
    fig = px.imshow(img, binary_string=True, binary_compression_level=compression_level)
    print(f"compression level {compression_level}: length of {len(fig.data[0].source)}")
fig.show()
```

```
compression level 0: length of 350438
compression level 1: length of 211734
compression level 2: length of 209810
compression level 3: length of 206994
compression level 4: length of 190598
compression level 5: length of 190314
compression level 6: length of 189774
compression level 7: length of 189258
compression level 8: length of 188426
```

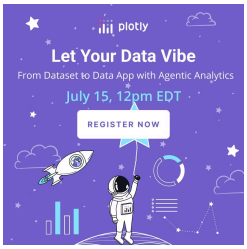
Exploring 3-D images, timeseries and sequences of images with facet_col

Introduced in plotly 4.14

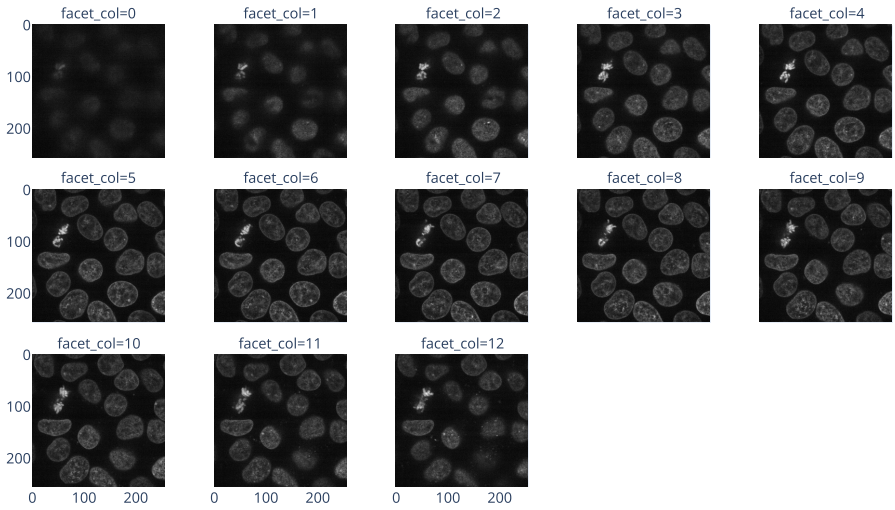
For three-dimensional image datasets, obtained for example by MRI or CT in medical imaging, one can explore the dataset by representing its different planes as facets. The facet_col argument specifies along which axis the image is sliced through to make the facets. With facet_col_wrap, one can set the maximum number of columns. For image datasets passed as xarrays, it is also possible to specify the axis by its name (label), thus passing a string to facet_col.

It is recommended to use binary_string=True for facetted plots of images in order to keep a small figure size and a short rendering time.

See the tutorial on facet plots (/python/facet-plots/) for more information on creating and styling facet plots.

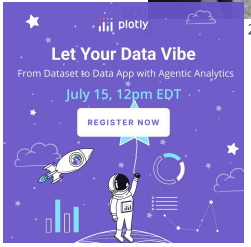


```
import plotly.express as px
from skimage import io
data = io.imread("https://github.com/scikit-image/skimage-tutorials/raw/main/images/cells.tif")
img = data[20:45:2]
fig = px.imshow(img, facet_col=0, binary_string=True, facet_col_wrap=5)
fig.show()
```



Facets can also be used to represent several images of equal shape, like in the example below where different values of the blurring parameter of a Gaussian filter are compared.

```
import plotly.express as px
import numpy as np
from skimage import data, filters, img_as_float
img = data.camera()
sigmas = [1, 2, 4]
img_sequence = [filters.gaussian(img, sigma=sigma) for sigma in sigmas]
fig = px.imshow(np.array(img_sequence), facet_col=0, binary_string=True,
                labels={'facet_col':'sigma'})
# Set facet titles
for i, sigma in enumerate(sigmas):
    fig.layout.annotations[i]['text'] = 'sigma = %d' %sigma
fig.show()
```

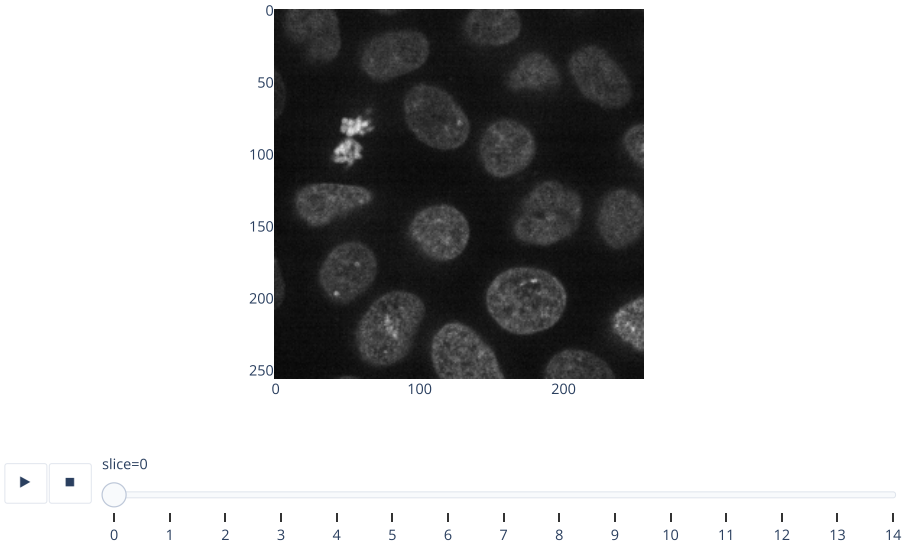


Exploring 3-D images and timeseries with `animation_frame`

Introduced in plotly 4.14

For three-dimensional image datasets, obtained for example by MRI or CT in medical imaging, one can explore the dataset by sliding through its different planes in an animation. The `animation_frame` argument of `px.imshow` sets the axis along which the 3-D image is sliced in the animation.

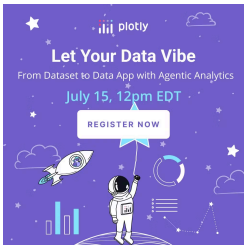
```
import plotly.express as px
from skimage import io
data = io.imread("https://github.com/scikit-image/skimage-tutorials/raw/main/images/cells.tif")
img = data[25:40]
fig = px.imshow(img, animation_frame=0, binary_string=True, labels=dict(animation_frame="slice"))
fig.show()
```



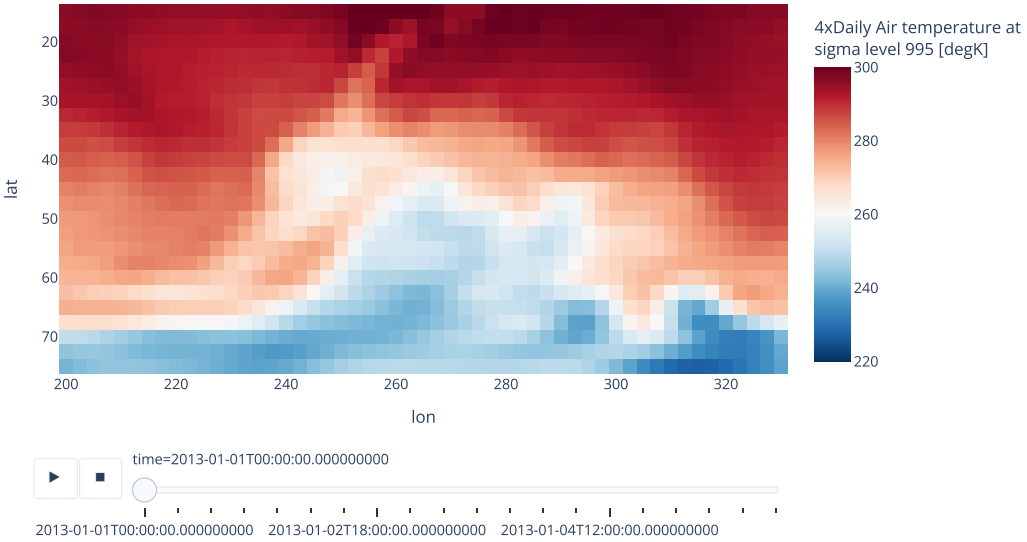
Animations of xarray datasets

Introduced in plotly 4.14

For xarray datasets, one can pass either an axis number or an axis name to `animation_frame`. Axis names and coordinates are automatically used for the labels, ticks and animation controls of the figure.



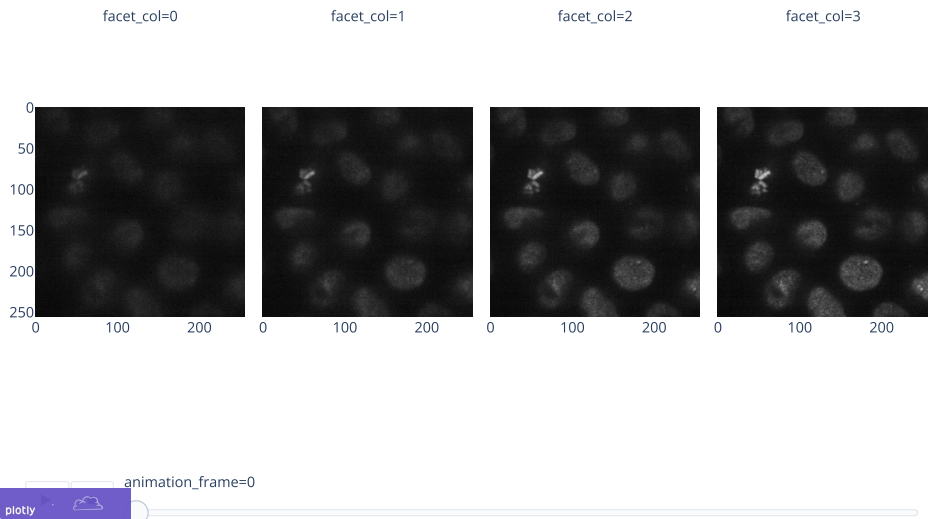
```
import plotly.express as px
import xarray as xr
# Load xarray from dataset included in the xarray tutorial
ds = xr.tutorial.open_dataset('air_temperature').air[:20]
fig = px.imshow(ds, animation_frame='time', zmin=220, zmax=300, color_continuous_scale='RdBu_r')
fig.show()
```

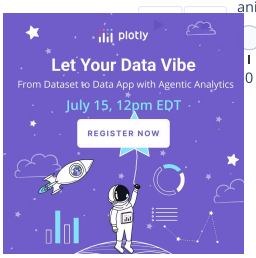


Combining animations and facets

It is possible to view 4-dimensional datasets (for example, 3-D images evolving with time) using a combination of `animation_frame` and `facet_col`.

```
import plotly.express as px
from skimage import io
data = io.imread("https://github.com/scikit-image/skimage-tutorials/raw/main/images/cells.tif")
data = data.reshape((15, 4, 256, 256))[5:]
fig = px.imshow(data, animation_frame=0, facet_col=1, binary_string=True)
fig.show()
```





Reference

See function reference for px.(imshow) (https://plotly.com/python-api-reference/generated/plotly.express.imshow) or https://plotly.com/python/reference/image/ (https://plotly.com/python/reference/image/) for more information and chart attribute options!

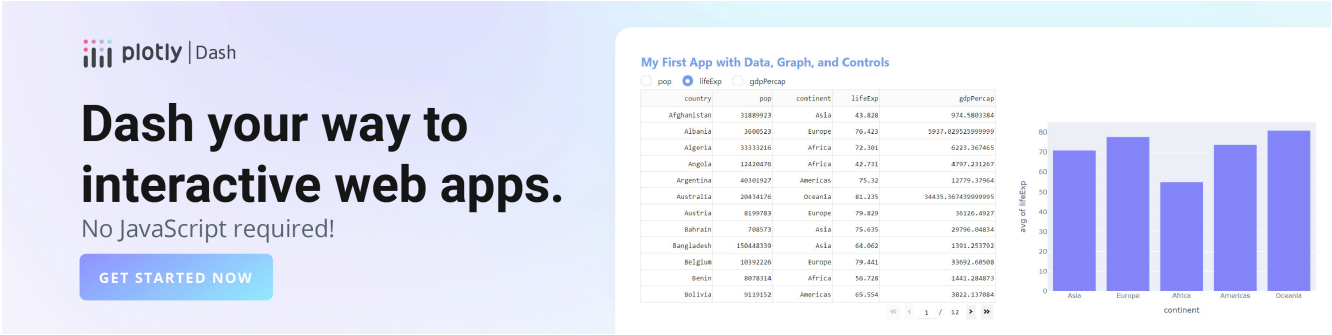
What About Dash?

Dash (https://dash.plot.ly/) is an open-source framework for building analytical applications, with no Javascript required, and it is tightly integrated with the Plotly graphing library.

Learn about how to install Dash at https://dash.plot.ly/installation (https://dash.plot.ly/installation).

Everywhere in this page that you see fig.show(), you can display the same figure in a Dash application by passing it to the figure argument of the Graph component (https://dash.plot.ly/dash-core-components/graph) from the built-in dash_core_components package like this:

```
import plotly.graph_objects as go # or plotly.express as px
fig = go.Figure() # or any Plotly Express function e.g. px.bar(...)
# fig.add_trace( ... )
# fig.update_layout( ... )

from dash import Dash, dcc, html

app = Dash()
app.layout = html.Div([
    dcc.Graph(figure=fig)
])

app.run(debug=True, use_reloader=False)  # Turn off reloader if inside Jupyter
```



(https://dash.plotly.com/tutorial?utm_medium=graphing_libraries&utm_content=python_footer)

JOIN OUR MAILING LIST

Sign up to stay in the loop with all things Plotly — from Dash Club to product updates, webinars, and more!

SUBSCRIBE (HTTPS://GO.PLOT.LY/SUBSCRIPTION)

Products

Dash (https://plotly.com/dash/)
Consulting and Training (https://plotly.com/consulting-and-oem/)

Pricing

Enterprise Pricing (https://plotly.com/get-pricing/)

About Us

Careers (https://plotly.com/careers)
Resources (https://plotly.com/resources/)
Blog (https://medium.com/@plotlygraphs)

Support

Community Support (https://community.plot.ly/)
Documentation (https://plotly.com/graphing-libraries)

Copyright © 2025 Plotly, Inc. All rights reserved.

Terms of Service (https://community.plotly.com/tos) Privacy Policy (https://plotly.com/privacy/)