Customizing Colorbars

Plot legends identify discrete labels of discrete points.

For **continuous labels** based on the color of points, lines, or regions, a **labeled colorbar** can be a great tool.

In Matplotlib, a colorbar is drawn as a **separate axes** that can provide a key for the meaning of colors in a plot.

We'll start by setting up the notebook for plotting and importing the functions we will use:

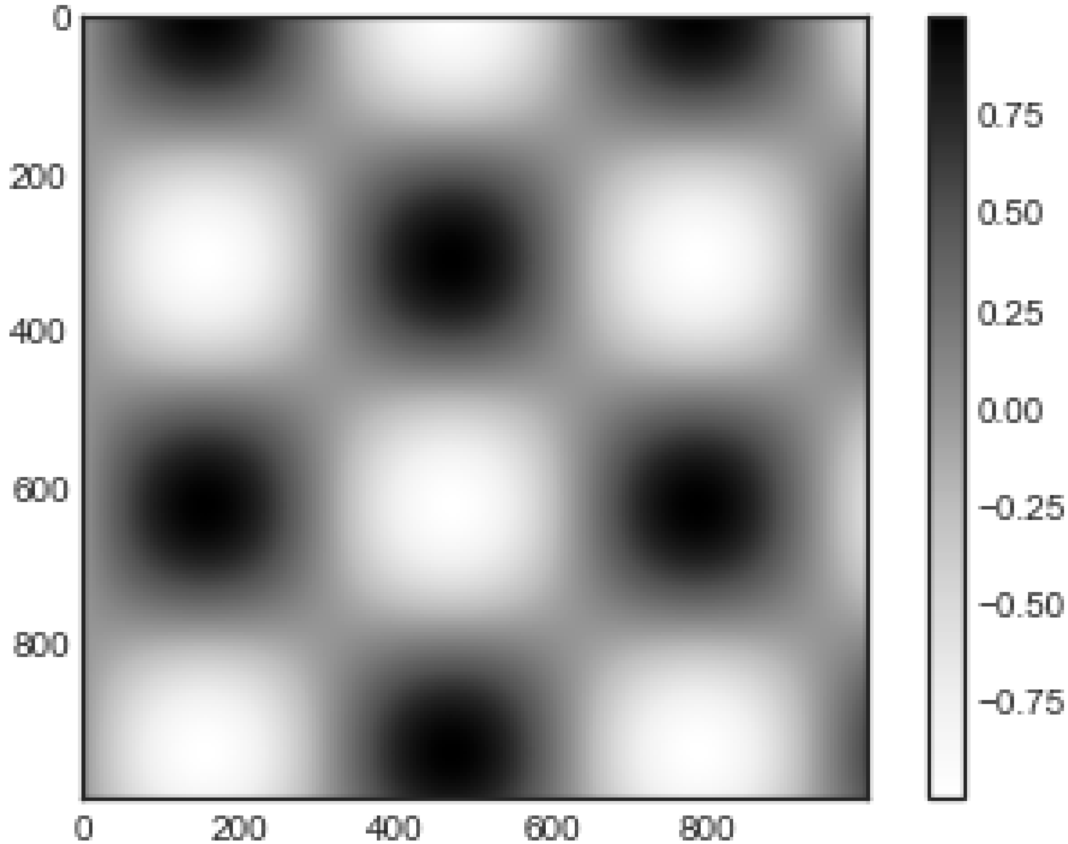
In [ ]:
```
import matplotlib.pyplot as plt
plt.style.use('seaborn-white')
```

In [ ]:
```
%matplotlib inline
import numpy as np
```

As we have seen several times already, the simplest colorbar can be created with the `plt.colorbar` function (see the following figure):

In [ ]:
```
x = np.linspace(0, 10, 1000)
I = np.sin(x) * np.cos(x[:, np.newaxis])

plt.imshow(I)
plt.colorbar();
```

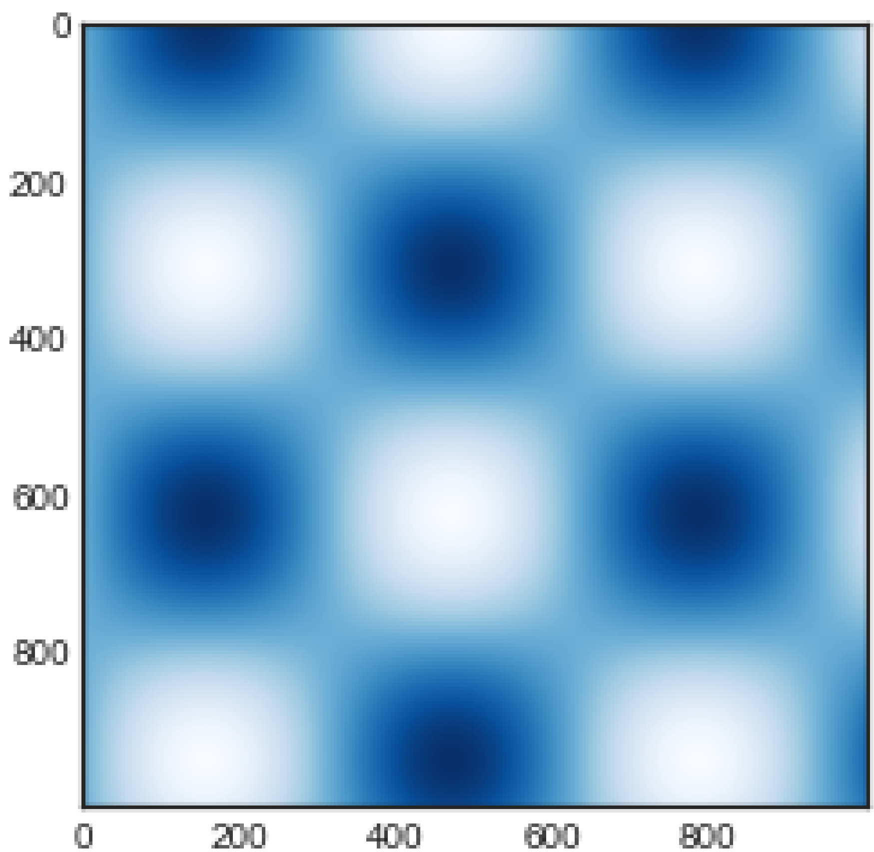


We'll now discuss a few ideas for **customizing these colorbars** and using them effectively in various situations.

Customizing Colorbars

The colormap can be specified using the `cmap` argument to the plotting function that is creating the visualization (see the following figure):

```
In [ ]: plt.imshow(I, cmap='Blues');
```



The **names of available colormaps** are in the `plt.cm` namespace;

using IPython's tab completion feature will give you a full list of built-in possibilities:

```
plt.cm.<TAB>
```

But being **able** to choose a colormap is just the first step: more important is how to **decide** among the possibilities!

The choice turns out to be **much more subtle** than you might initially expect.

Choosing the Colormap

Broadly, you should be aware of three **different categories of colormaps:**

- **Sequential colormaps:** These are made up of one continuous sequence of colors (e.g., `binary` or `viridis`).
- **Divergent colormaps:** These usually contain two distinct colors, which show positive and negative deviations from a mean (e.g., `RdBu` or `PuOr`).
- **Qualitative colormaps:** These mix colors with no particular sequence (e.g., `rainbow` or `jet`).

The `jet` colormap, which was the **default in Matplotlib** prior to version 2.0, is an example of a qualitative colormap.

Its status as the default was quite unfortunate, because qualitative maps are often a **poor choice** for representing quantitative data.

Among the problems is the fact that qualitative maps usually do not display any **uniform progression in brightness** as the scale increases.

We can see this by converting the `jet` colorbar into black and white (see the following figure):

In [ ]:

```
from matplotlib.colors import LinearSegmentedColormap

def grayscale_cmap(cmap):
    """Return a grayscale version of the given colormap"""
    cmap = plt.cm.get_cmap(cmap)
    colors = cmap(np.arange(cmap.N))

    # Convert RGBA to perceived grayscale luminance
    # cf. http://alienryderflex.com/hsp.html
    RGB_weight = [0.299, 0.587, 0.114]
    luminance = np.sqrt(np.dot(colors[:, :3] ** 2, RGB_weight
    colors[:, :3] = luminance[:, np.newaxis]

    return LinearSegmentedColormap.from_list(
        cmap.name + "_gray", colors, cmap.N)


def view_colormap(cmap):
    """Plot a colormap with its grayscale equivalent"""
    cmap = plt.cm.get_cmap(cmap)
```

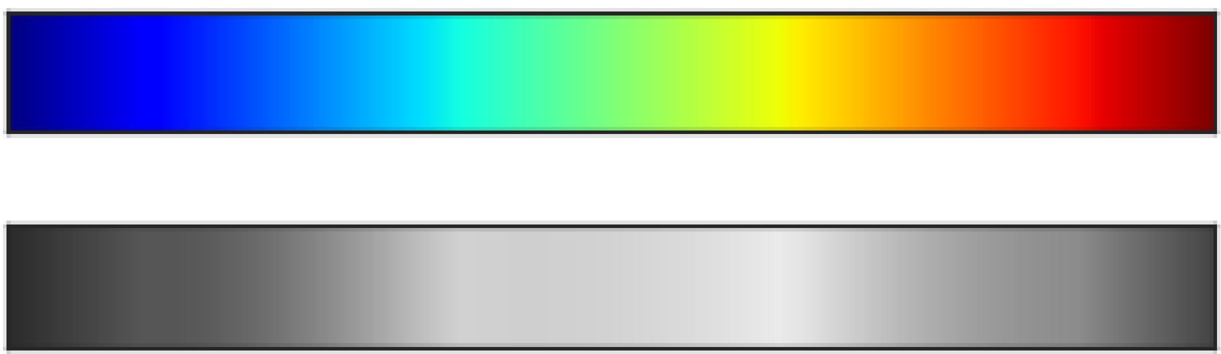
```
    colors = cmap(np.arange(cmap.N))

    cmap = grayscale_cmap(cmap)
    grayscale = cmap(np.arange(cmap.N))

    fig, ax = plt.subplots(2, figsize=(6, 2),
                           subplot_kw=dict(xticks=[], yticks=
    ax[0].imshow([colors], extent=[0, 10, 0, 1])
    ax[1].imshow([grayscale], extent=[0, 10, 0, 1])
```

In [ ]:

```
view_colormap('jet')
```



Notice the **bright stripes in the grayscale** image.

Even in full color, this **uneven brightness** means that the eye will be drawn to certain portions of the color range,

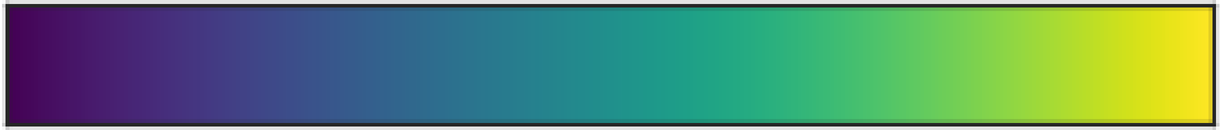
which will potentially **emphasize unimportant** parts of the dataset.

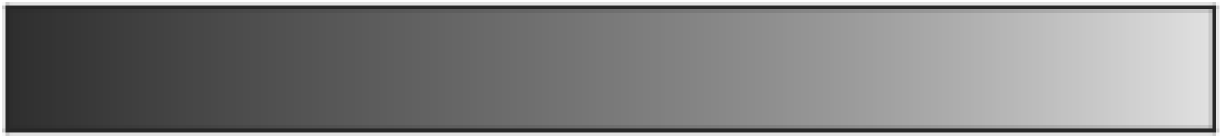
It's better to use a colormap such as `viridis` (the default as of Matplotlib 2.0),

which is specifically constructed to have an **even brightness variation across the range;**

thus, it not only plays well with our **color perception,** but also will **translate well to grayscale printing** (see the following figure):

In [ ]:
```
view_colormap('viridis')
```

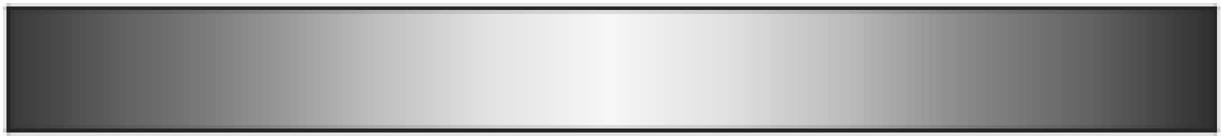




For other situations, such as showing **positive and negative deviations** from some mean, **dual-color colorbars** such as `RdBu` (*Red–Blue*) are helpful.

However, as you can see in the following figure, it's important to note that the **positive/negative information will be lost** upon translation to grayscale!

In [ ]:
```
view_colormap('RdBu')
```



We'll see examples of using some of these colormaps as we continue.

There are a **large number of colormaps available** in Matplotlib;

to see a list of them, you can use IPython to explore the `plt.cm` submodule.

Color Limits and Extensions

Matplotlib allows for a **large range** of **colorbar customization.**

The colorbar itself is simply an **instance** of `plt.Axes`,

so all of the axes and tick formatting tricks we've seen so far are applicable.

The colorbar has some interesting **flexibility:**

for example, we can narrow the color limits and indicate the out-of-bounds values with a **triangular arrow** at the top and bottom by setting the `extend` property.

This might come in handy, for example, if displaying an image that is subject to noise (see the following figure):

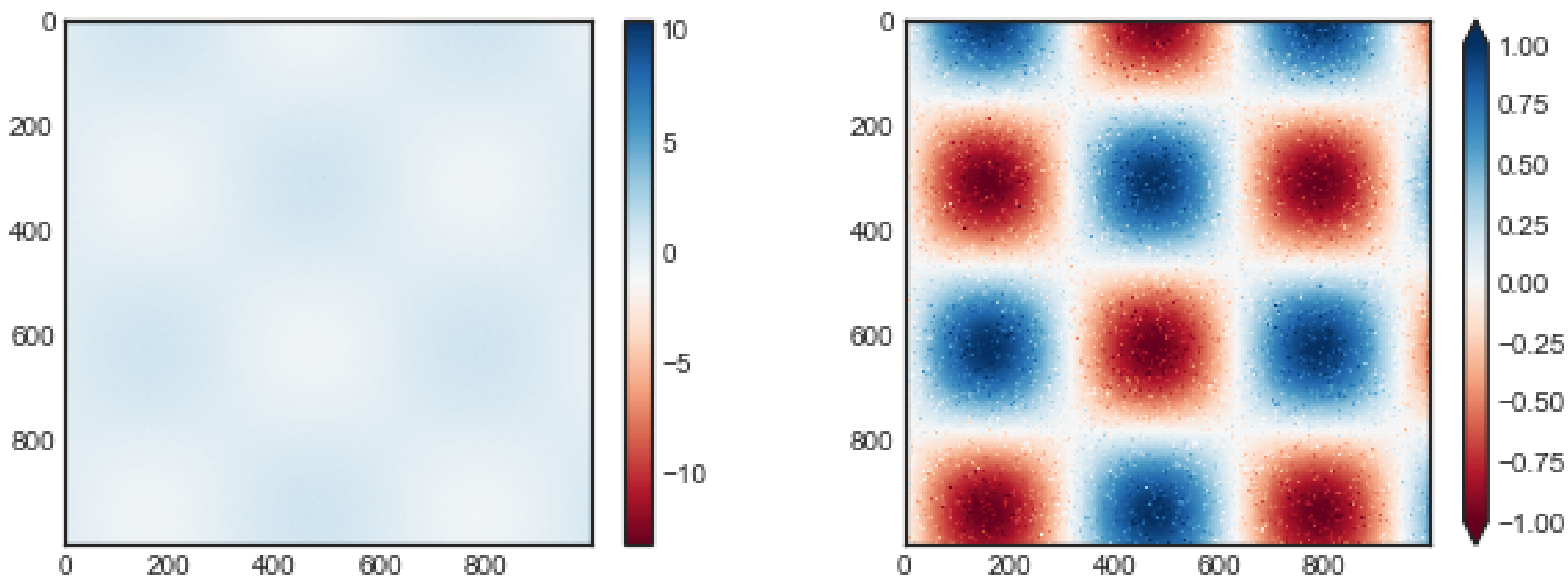
In [ ]:
```
# make noise in 1% of the image pixels
speckles = (np.random.random(I.shape) < 0.01)
I[speckles] = np.random.normal(0, 3, np.count_nonzero(speckle

plt.figure(figsize=(10, 3.5))

plt.subplot(1, 2, 1)
```

```
plt.imshow(I, cmap='RdBu')
plt.colorbar()

plt.subplot(1, 2, 2)
plt.imshow(I, cmap='RdBu')
plt.colorbar(extend='both')
plt.clim(-1, 1)
```



Notice that in the left panel, the default color limits respond to the noisy pixels, and the range of the noise completely **washes out the pattern** we are interested in.

In the right panel, we **manually** set the color limits and add extensions to indicate values that are above or below those limits.

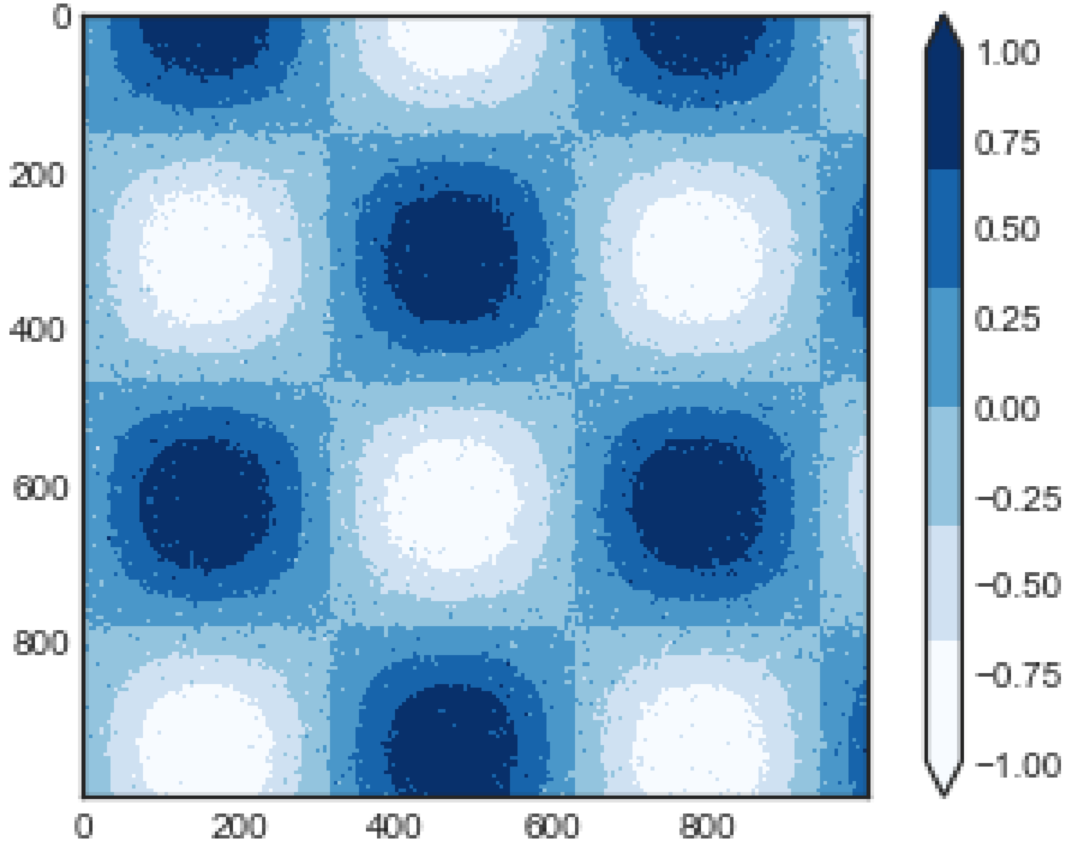
The result is a much more **useful visualization** of our data.

Discrete Colorbars

Colormaps are by default continuous, but sometimes you'd like to represent **discrete values.**

The easiest way to do this is to use the `plt.cm.get_cmap` function and pass the name of a suitable colormap along with the number of desired **bins** (see the following figure):

In [ ]:
```
plt.imshow(I, cmap=plt.cm.get_cmap('Blues', 6))
plt.colorbar(extend='both')
plt.clim(-1, 1);
```


0
200
400
600
800
0
200
400
600
800
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00