matplotlib-demo

January 27, 2026

1 Plotting in Python with Matplotlib

1.1 Winter visualization series

J. Yohai Meiron, PhD SciNet High Performance Computing Consortium University of Toronto

The University of Toronto is located on the traditional land ofthe Huron-Wendat, the Seneca, and the Mississaugas of the Credit.

Abstract Matplotlib is the foundational plotting library for Python, and is widely used in tandem with other scientific libraries (such as NumPy and Pandas) to visualize data across many different fields. It is a free and open source software library that offers fine-grained control over every aspect of a plot, making it a powerful tool for customizing figures to meet specific needs. In this workshop, we will use a Jupyter notebook to show you how to create common 2D plot types such as line, scatter, and heatmaps, and how to customize the labels, legend, and panel layout. We will briefly touch on using Matplotlib to produce animations and interactive visualizations. By the end of this one-hours session, you will have a basic understanding of the library's capabilities.

How to follow? Notebook link: https://pages.scinet.utoronto.ca/~ymeiron/matplotlib-demo.ipynb * I'm using SciNet Open OnDemand JupyterLab. Use that if you have access to the Trillium system. * If you don't, you can install the `notebook`, `matplotlib`, `pandas` and `ipympl` Python packages and the `ffmpeg` software *locally*, and launch a notebook server. * You can use a free (or paid) cloud Jupyter notebook environment (Google Colab, Binder, Kaggle). Note that not all support interactive plots. * We have a Magic Castle instance for this lecture (details provided separately). This *does not* support interactive plots however.

Unfortunately we won't have time to assist with technical difficulties today.

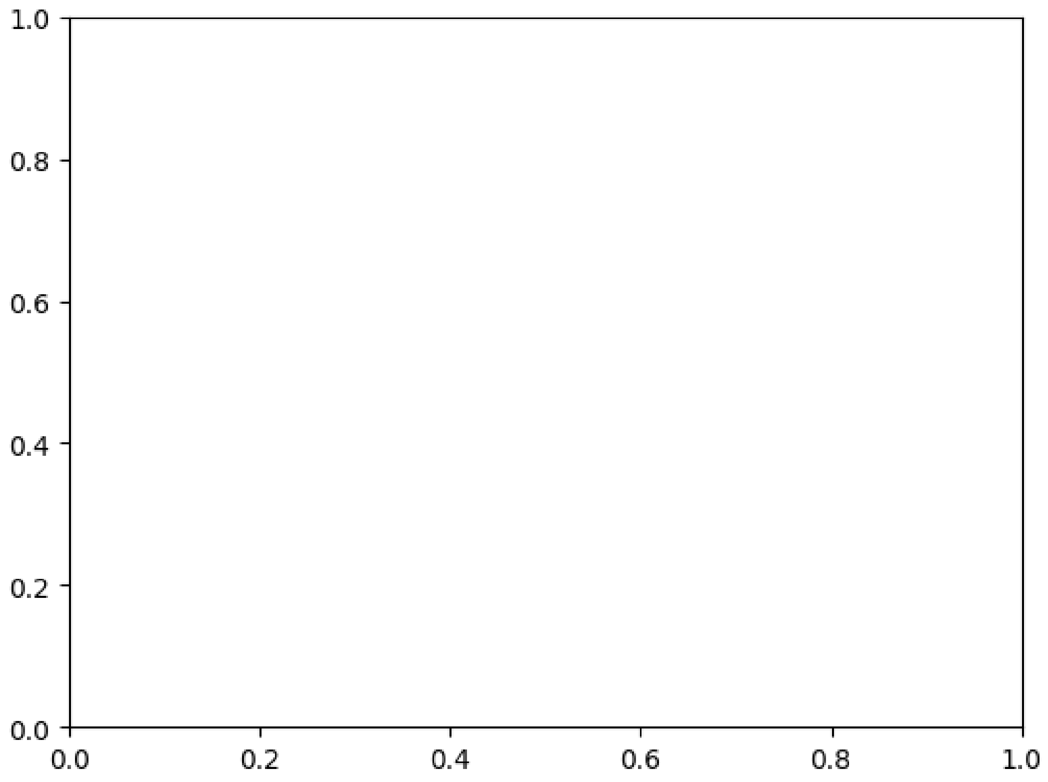
1.2 Introduction

Matplotlib (MPL) is a plotting library for Python, launched in 2003 and inspired by MATLAB. Today we are going to learn the basics of working with it, and I'm going to demo a few common types of plots, but what we can show in an hour is really the tip of the iceberg in terms of what's possible. The best way to learn further is to go to the gallery and find a figure with features you are interested in, and look at the source code to learn how it was made.

MPL provides a high level API (object-oriented as well as state-based) for creating figures, using several possible backends. This API is called PyPlot, let's import it and draw our first figure.

[1]:
```
from matplotlib import pyplot as plt
```

[3]:
```
fig, ax = plt.subplots()
```



The `subplots` function, called with no parameters here, caused this image to be added to our notebook and returned `Figure` and `Axes` objects. The figure is the top level container for all the plot elements. Each visible element, or an object that can draw itself on a figure, is called an *artist* in MPL terminology (the figure itself is also technically an artist). The axes object (or *subfigure*) is an artist contained in the figure, and it itself contains other artists (such as the ticks and labels, and plot elements once we add them). Artists have member functions that we can use to manipulate them, as we shall see. Axes are optional but in this lecture we'll always use them, and we can have more than one set of axes in the figure.

This is not the only way to create a figure with axes. In fact, there are usually multiple ways to do anything in Matplotlib, in violation of the Zen of Python. Here we'll stick to an object-oriented coding style and avoid shortcuts. This will produce a code that is slightly longer, but more readable.

1.2.1 Backends

Let's talk about the figure appearing in the notebook. While this was just one line, obviously there is a lot of low level *backend* code that's involved here. MPL can use several different backends to actually create an image (be it static or interactive). We don't need to know much about what they do and how, but just that we have the choice.

Jupyter notebooks The default backend in Jupyter notebooks is called `inline`. It essentially creates a PNG image (a static bitmap) from the figure, that is embedded into the notebook web page. Here, we cannot interact with the plot.

An alternative backend for Jupyter Notebooks is `ipympl` (requires a package of the same name to be installed). It renders the figure as an applet using web technologies (such as JavaScript) that get embedded in the notebook web page. Here, we can interact with the figure using the mouse as we are going to see. To switch to this backend use the `%matplotlib ipympl` Jupyter "magic" command.

In a notebook, if we create a figure in one cell, the `Figure` object still exists in the following cells (until it's discarded, e.g. the `fig` name reassigned). If it (or subordinate artists) is manipulated in the following cells, this will only affect what we see embedded in the notebook if we are using the `ipympl` backend. If the `inline` backend is used, commands run in the following cells won't affect the already-displayed figure. In this case you can draw the figure again using `display(fig)`.

[4]:
```
%matplotlib ipympl
# Run this cell and go back to the cell where we created the previous figure,␣
 ↪and run it again.
```

Interactive backends and the `show` function If the Python code runs as a standalone program through the command line, you'll usually use a backend that renders the figure in a graphical window. The backend will normally be chosen automatically based on the operating system and Python/MPL installation, but can be changed using the `matplotlib.use` function.

Importantly, when all figures are ready, use `plt.show()` to indicate to the backend to open the graphical windows (one per figure).

Static backends and the `savefig` function If the Python code runs in a non-interactive environment, such as when running a batch job in an HPC cluster, you'll use a *static* backend, that will not attempt to render anything to the screen. In most cases the library will know that there is no display available, but if MPL is trying to use an interactive backend for some reason and is producing an error, you can change it by using the `matplotlib.use` function (and choosing a static backend such as `agg`).

In those cases, you will save a figure to a file. The static backend will be chosen automatically based on the file extension. For example:

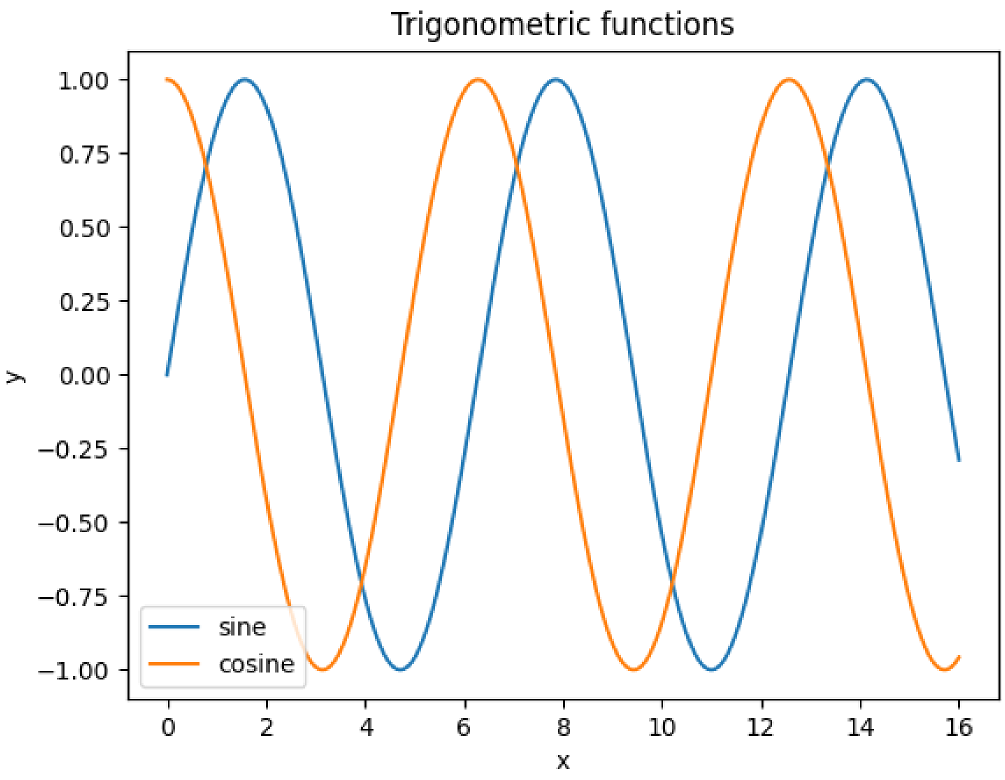
[5]:
```
fig.savefig('simple_figure.pdf')
```

1.3 Line plot

Let's create our first *plot*.

[6]:
```
import numpy as np
x  = np.linspace(0, 16, 256)
y1 = np.sin(x)
y2 = np.cos(x)

fig, ax = plt.subplots()
```

```
ax.plot(x, y1, label='sine')
ax.plot(x, y2, label='cosine')
ax.legend()
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_title('Trigonometric functions');

# The semicolon at the end of the last command just suppresses some unwanted␣
 ↪output in the Jupyter notebook
```



The alternative to this object-oriented approach is to use the `plt.plot` function, which creates the figure and axes if they don't already exist.

1.4 The global style

As we just saw, sensible defaults are used when creating a new figure. We didn't have to specify what font to use, what colour to use for each line, where to place the ticks, etc. In this case we just used MPL's defaults, giving us this signature look.

1.4.1 Runtime configuration settings

When producing many figures (e.g. for publication, or a poster), we might want to tweak these defaults, so our figures have a consistent look. To do this we can tweak the runtime configuration settings by manipulating `matplotlib.rcParams`. For example:

[7]:
```
import matplotlib as mpl
mpl.rcParams['font.family'] = 'Futura LT'
# Run this cell and go back to the cell where we created the previous figure,␣
 ↪and run it again.
```

Will change the default font of artists (reminder: this means figure elements) to Futura, in all future figures in this notebook.

There are (currently) 322 different settings (not all style-related)! So MPL plots are extremely customizable. You can find them listed here.

When choosing a font, it has to be installed on the machine where the Python interpreter (for the Jupyter notebook in this case) is running.

1.4.2 Style sheets

In addition to MPL's "signature" look, additional pre-defined styles are available (see them listed here). Load them with `plt.style.use` (and of course, you can still tweak the chosen style by manipulating `matplotlib.rcParams`)

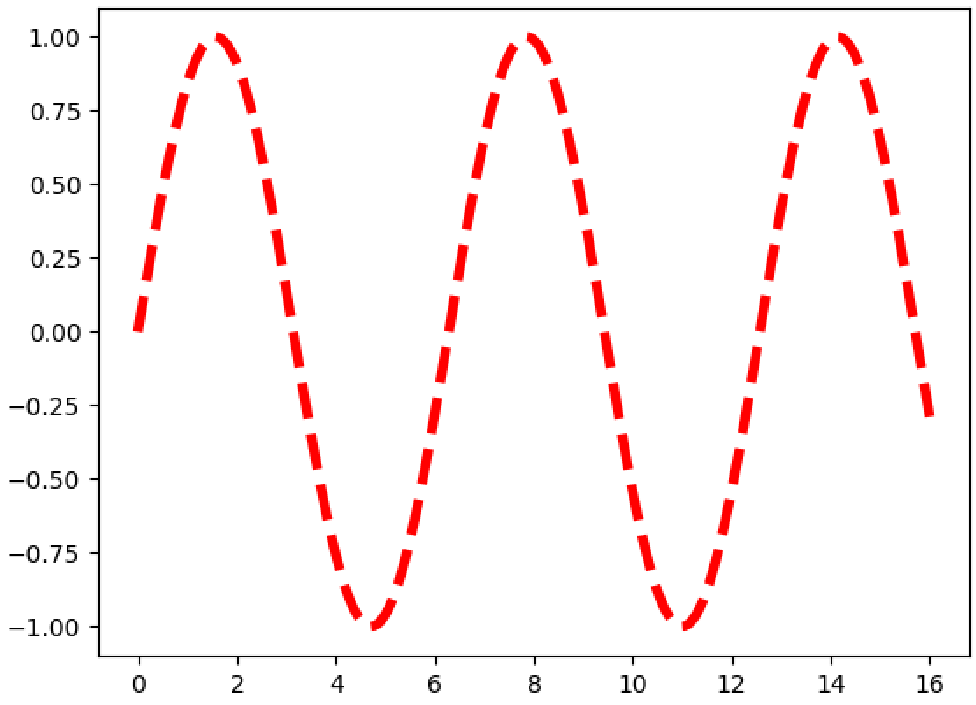
[8]:
```
plt.style.use('classic') # Also try: ggplot, fivethirtyeight, bmh
# Run this cell and go back to the cell where we created the previous figure,␣
 ↪and run it again.
```

[9]:
```
# Restore defaults values of all runtime configuration settings
plt.rcdefaults()
```

1.5 Keyword arguments

The most common way to customize part of the figure is to pass keyword arguments to the function that creates the artist. In the line plot example, the first line was blue and the second orange, this colour sequence is defined in the default style sheet. We can manipulate the line's characteristics by passing suitable keywords that the `plot` function can accept (see documentations).

[10]:
```
fig, ax = plt.subplots()
l, = ax.plot(x, y1, color='r', linewidth=4, linestyle='--')
```



- The string `r` is a single character shorthand notation for "red", there are many other ways to specify colours.
- The string `--` indicates a dashed line, see other options here.
- Capturing the output (artist) of `ax.plot` is not mandatory and not needed in most cases, but if we have it we can manipulate it (the line style **or even the data**) in the future (useful for animations!) For example:

[11]:
```
l.set_linewidth(0.5)
```

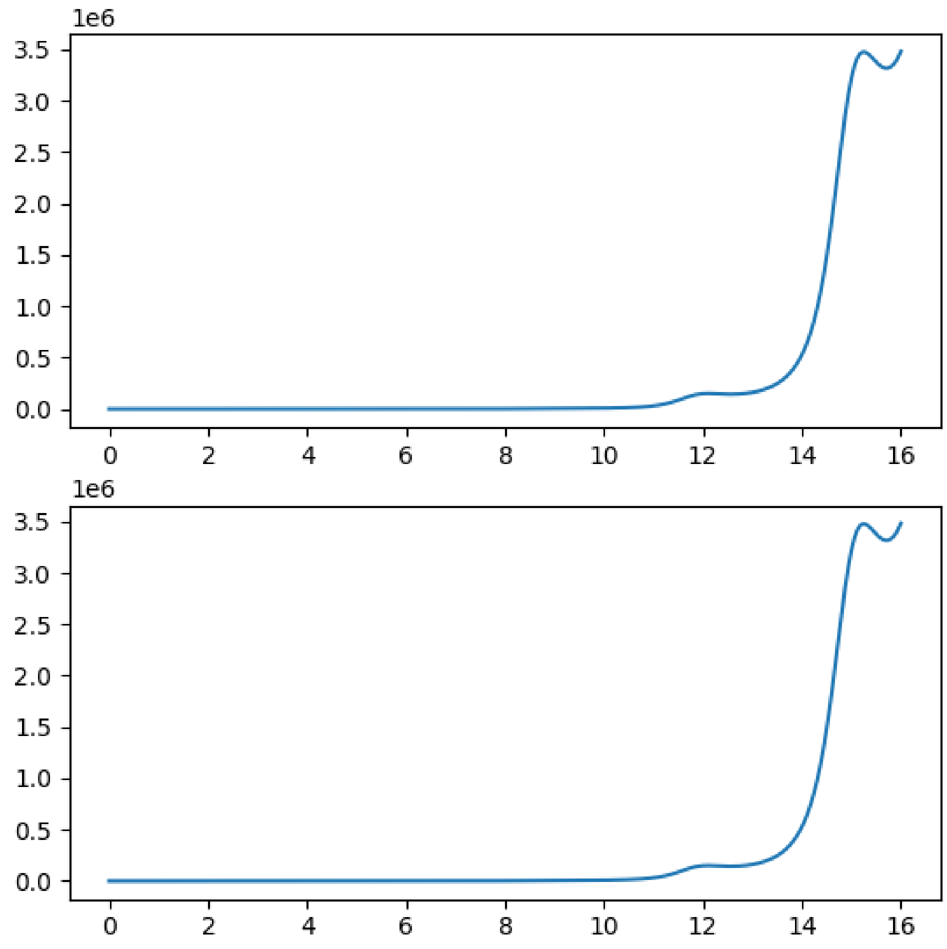
[12]:
```
l.set_ydata(np.sin(4*x))
```

- Some keywords have aliases: `color` → `c`, `linewidth` → `lw`, `linestyle` → `ls`
- The `plot` function also supports a third positional argument (after `x` and `y`) that can be used to specify colour and line style (as well as marker style), so the above is equivalent to `ax.plot(x, y1, 'r--', linewidth=4)`.

1.6 More about figures and axes

Let's create a figure with two sets of axes one on top of the other. We'll make the same line plot in both.

[13]:
```
fig, [ax1, ax2] = plt.subplots(nrows=2, figsize=(6.4, 6.4))
func = lambda x: np.exp(x)/(np.sin(2*x)+2)
ax1.plot(x, func(x))
ax2.plot(x, func(x));
```



- We used keyword arguments for `plt.subplots` to make two sets of axes in one column, and also change the figure size.
- The `figsize` keyword argument takes a tuple of width and height in the Imperial "inch" unit (2.54 cm). The default figure size is 6.4 × 4.8 of those units (`rcParams['figure.figsize']`), and the on-screen resolution is 100 pixels per "inch" (`rcParams['figure.dpi']`).
- We drew two independent sets of axes. We can lock the x-axis by adding `sharex=True` to `plt.subplots`.
- Of course, we can have bigger grids of subplots, and even more elaborate arrangements by

using GridSpec.

This plot isn't very good because of the large range of y-values. Let's change the lower subplot such that the y-axis is logarithmic:

[14]:
```
ax2.set_yscale('log')
```

By default, MPL leaves some margins around the data for determining axis limits (cf. `rcParams['axes.xmargin']`).

We can manually set the limits to whatever we want like so:

[15]:
```
ax2.set_xlim(0, 10);
```

- If the x-axis is shared (`sharex=True`) then whatever limits we set for `ax2` will apply to `ax1` as well (and vice versa)!
- `Axes` has convenience functions to create a line plot *and* simultaneously change the scaling to logarithmic, these are `semilogx`, `semilogy`, and `loglog`.

If we want to change the spacing between the subplots, we can call `Figure.subplots_adjust` (cf. `Figure.tight_layout`)

[16]:
```
fig.subplots_adjust(hspace=0.025)
```

1.7 Scatter plots

We can use `plt.plot` for simple scatter plots as well. We just have to make sure that there are markers but no line.

[17]:
```
x_peaks = np.arctan(-1/2) + np.arange(1, 6)*np.pi
ax2.plot(x_peaks, func(x_peaks), linestyle='', marker='o');
```

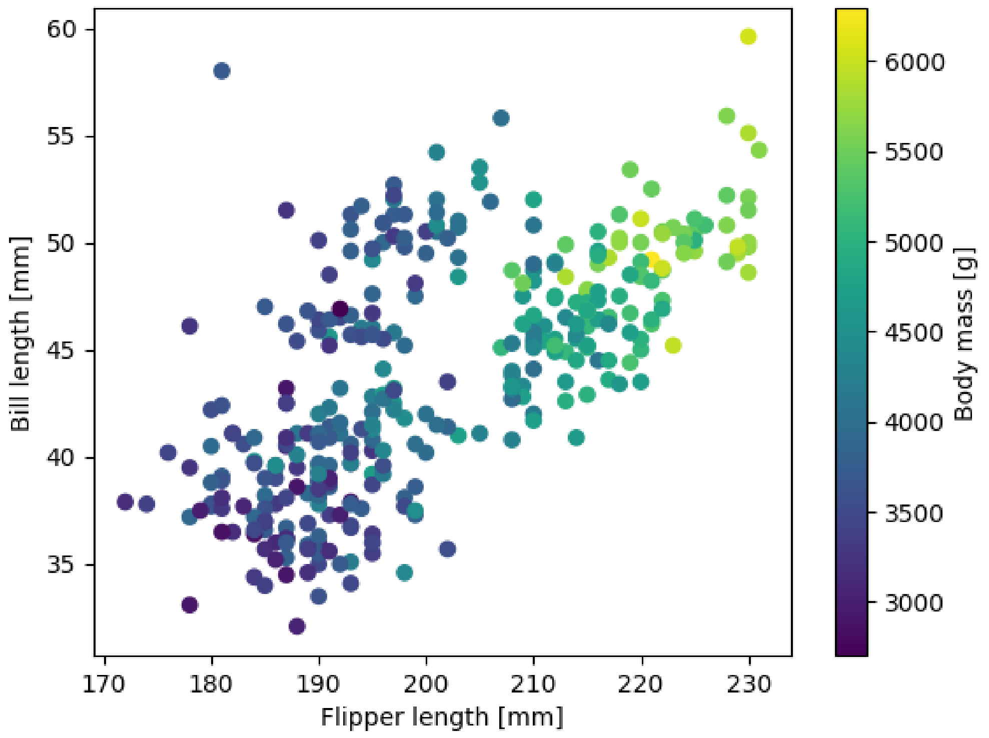
- There are quite a few markers to choose from (and you can make your own), see here.
- We could have created an equivalent plot with the third positional argument: `ax2.plot(x_peaks, func(x_peaks), 'o')`

The `Axes` class has many more plotting functions! While `plot` can create a scatter plot, all markers have the same shape, size, and colour. We can use `scatter` instead if we want to represent additional properties visually. In the example below we use Pandas to retrieve a small dataset related to penguins, and create a scatter plot of bill length vs. flipper length, where the colour represents a third property (body mass).

[18]:
```
import pandas as pd
df = pd.read_csv('https://raw.githubusercontent.com/allisonhorst/palmerpenguins/
  ↪master/inst/extdata/penguins.csv')
display(df)

x = df['flipper_length_mm']
y = df['bill_length_mm']
prop = df['body_mass_g']
```

```
       species     island  bill_length_mm  bill_depth_mm  flipper_length_mm  \
0       Adelie  Torgersen            39.1           18.7              181.0
1       Adelie  Torgersen            39.5           17.4              186.0
2       Adelie  Torgersen            40.3           18.0              195.0
3       Adelie  Torgersen             NaN            NaN                NaN
4       Adelie  Torgersen            36.7           19.3              193.0
..         …         …              …             …                  …
339  Chinstrap      Dream            55.8           19.8              207.0
340  Chinstrap      Dream            43.5           18.1              202.0
341  Chinstrap      Dream            49.6           18.2              193.0
342  Chinstrap      Dream            50.8           19.0              210.0
343  Chinstrap      Dream            50.2           18.7              198.0

     body_mass_g     sex  year
0         3750.0    male  2007
1         3800.0  female  2007
2         3250.0  female  2007
3            NaN     NaN  2007
4         3450.0  female  2007
..           …       …     …
339       4000.0    male  2009
340       3400.0  female  2009
341       3775.0    male  2009
342       4100.0    male  2009
343       3775.0  female  2009

[344 rows x 8 columns]
```

[19]:
```
fig, ax = plt.subplots()
scatter_plot = ax.scatter(x, y, c=prop)
ax.set_xlabel('Flipper length [mm]')
ax.set_ylabel('Bill length [mm]')
fig.colorbar(scatter_plot, label='Body mass [g]');
```



- This colour map is called *viridis* and is the default because it works well when printed in greyscale and is colourblind-friendly ("monotonic luminance"). It is also relatively perceptually uniform.
- There are many more options, and you can make your own.
  - Try passing `cmap='jet'` to `scatter`.
- Instead of the `c` keyword argument, try using `s` to have the marker size represent the property
  - You will usually have to transform the property somehow to get a useful visualization.

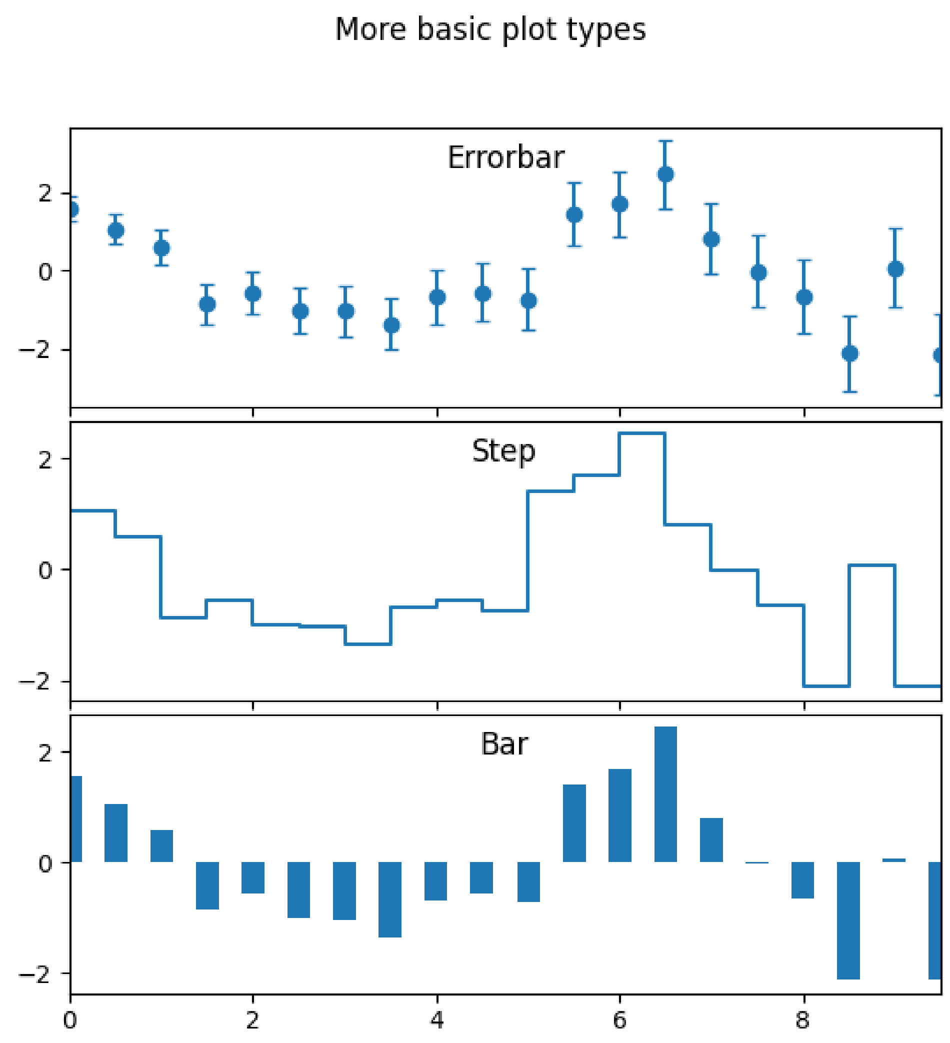
1.8 Text and annotation

We can place a text anywhere within the axes by calling the `text` function of the object (and there is also a `Figure.text` to place text in the figure, not associated with one axes/subplot object or another). In the example below we used `Axes.annotate` that is a bit more feature rich: it can place the text, but also shift it with respect to the point of interest, and draw and arrow to it.

[20]:
```
i_maxmass = np.argmax(prop)
x_maxmass, y_maxmass = x[i_maxmass], y[i_maxmass]
ax.annotate(
    f'chonkiest\npenguin\n({0.001*prop[i_maxmass]} kg)', # What text to place
    (x_maxmass, y_maxmass), # What point to annotate
    xytext=(15, -80), textcoords='offset points', # Where to put the annotation
    arrowprops=dict(arrowstyle='->'), # Arrow properties
    bbox=dict(boxstyle='round', facecolor='w', alpha=0.5), # Box properties
    horizontalalignment='center' # Other text properties
);
```

1.9 More baic plot types

[21]:
```
x = np.arange(0, 10, 0.5)
noise_envelope = np.sqrt((x+1)/10)
np.random.seed(3)
noise = np.random.randn(len(x))*noise_envelope
y = np.cos(x) + noise
fig, axs = plt.subplots(nrows=3, figsize=(6.4, 6.4), sharex=True)
axs[0].errorbar(x, y, yerr=noise_envelope, linestyle='', marker='o', capsize=3)
axs[0].set_title('Errorbar', y=0.8)
axs[1].step(x, y)
axs[1].set_title('Step', y=0.8)
axs[2].bar(x, y, width=0.25)
axs[2].set_title('Bar', y=0.8)
axs[0].set_xlim(x[0], x[-1])
fig.suptitle('More basic plot types')
fig.subplots_adjust(hspace=0.05)
```



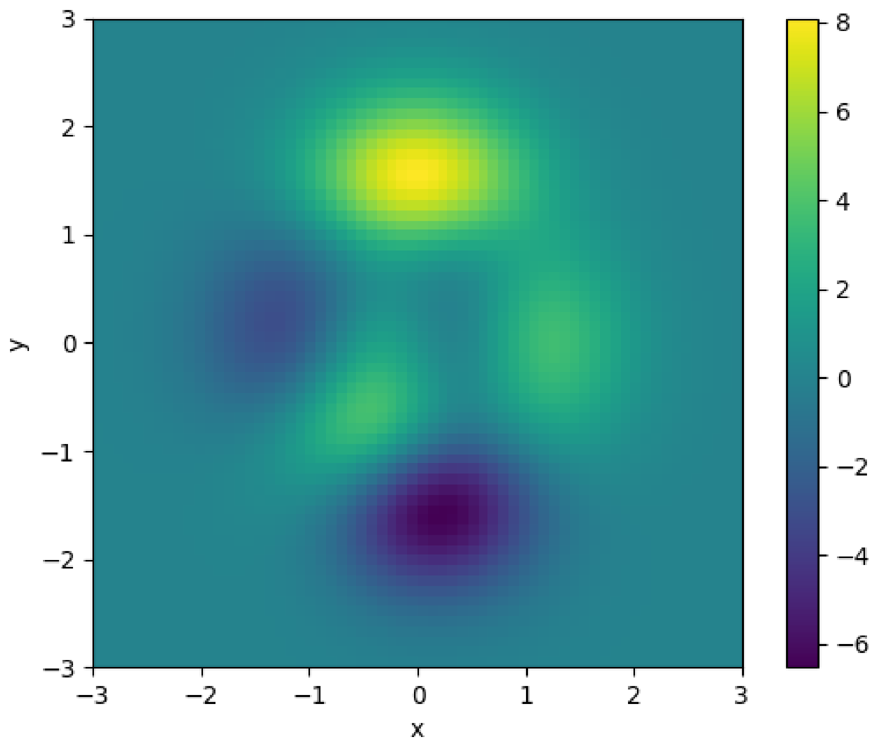
A useful convenience function is `Axes.hist`, which calculates a histogram from a dataset and draws a bar (or step) plot. Generally speaking, MPL is not a package for data analysis or numerical calculations, but it does have a few convenience functions for statistics and spectral analysis.

Seaborn is another library that uses MPL for data visualization. It provides a (even) higher-level API, that does both data analysis and plotting with fewer lines of code compared to MPL (check out Jarno's talk next week).

1.10 Visualizing 2D data

The simplest way to visualize the values of a 2D array is by using the `Axes.imshow` function, assuming a regular grid. This function treats the array values as image pixels, and maps it into the axes based on the `extent` argument.

[22]:
```
x_ = np.linspace(-3, 3, 64)
y_ = np.linspace(-3, 3, 64)
x, y = np.meshgrid(x_, y_)
z = 3*(1-x)**2*np.exp(-x**2-(y+1)**2) - 10*(x/5-x**3-y**5)*np.exp(-x**2-y**2) -␣
  ↪(1/3)*np.exp(-(x+1)**2-y**2)

fig, ax = plt.subplots()
im = ax.imshow(z, origin='lower', extent=[-3, 3, -3, 3])
fig.colorbar(im)
ax.set_xlabel('x')
ax.set_ylabel('y');
```



Like with `scatter`, you can choose a different colour map by passing the `cmap` parameter.

If the array is 3-dimensional, and the third dimension is size 3 or 4, this will be interpreted as a colour image (RGB or RGBA) and displayed accordingly.

`Axes.imshow` will show the array upside-down by default (with respect to how NumPy arrays are interpreted), use `origin='lower'` to change this behaviour.

An alternative to `imshow` is `pcolormesh`, that supports non-uniform grids and have a syntax that's more similar to the basic plotting functions. However it is slower and may produce very large output files if a vector graphics format is used (such as PDF or SVG).

You can also use `contour` or `contourf`:

[23]:
```
fig, ax = plt.subplots()
contour = ax.contour(x, y, z, levels=12)
ax.clabel(contour)
ax.set_xlabel('x')
ax.set_ylabel('y');
```

This plot looks squashed because by default it's adjusted to the size of the axes. We can force the aspect ratio like so:

[24]:
```
ax.set_aspect('equal')
```

1.11 3D plotting

Matplotlib can make basic and even reasonably advanced 3D plots (surfaces, wireframes, scatter, bars, etc.) It is not a full 3D rendering engine and lacks many features such as complex lighting and shadows, volumetric rendering, and advanced camera controls. Additionally it is not hardware accelerated, so will not handle large datasets as well as specialized tools.

Here is a demonstration of plotting the 2D array from before as a surface:

[25]:
```
fig = plt.figure()
ax = fig.add_subplot(projection='3d')
ax.plot_surface(x, y, z)
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z');
```

1.12 Animations

An animation is just a sequence of still frames. As we saw throughout this lecture, once an artist is created, we can use various functions to modify it. The MPL *animation* module provides a useful class called `FuncAnimation`: you need to pass to it a function that updates the plot as a side effect, and the resultant `Animation` object can be saved or displayed on screen. Here is an example:

[26]:
```
import matplotlib.animation as animation

frames = 60
x = np.linspace(0, 16, 256)
t = np.linspace(0, 2*np.pi, frames+1)[:-1]
y1 = np.sin(x)*np.sin(t[0])
y2 = np.cos(x)*np.cos(t[0])

fig, ax = plt.subplots()
l1, = ax.plot(x, y1, label='sine')
l2, = ax.plot(x, y2, label='cosine')
ax.legend(loc='upper left')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_title('Trigonometric functions')
text = ax.text(0,  0.65, 't = 0.00', bbox=dict(boxstyle='round', facecolor='w',␣
 ↪alpha=0.5), fontfamily='monospace')

def update(frame):
    y1 = np.sin(x)*np.sin(t[frame])
    y2 = np.cos(x)*np.cos(t[frame])
    l1.set_ydata(y1)
    l2.set_ydata(y2)
    text.set_text(f't = {t[frame]:.2f}')

ani = animation.FuncAnimation(fig=fig, func=update, frames=frames,␣
 ↪interval=3000/frames)

# Show animation in notebook
plt.close(fig)
from IPython.display import HTML
HTML(ani.to_html5_video())
```

[26]:
```
<IPython.core.display.HTML object>
```

- We created a figure much like we did before; we added two line plots, a legend, and a text.
- Notice we passed `loc='upper left'` to `legend` (otherwise it will jump around in the animation depending on the line plots).
- The `update` function recalculates the y values for both plots.
  - It updates the plot (`Line2D`) artists.
  - It updates the text (`Text`) artist.
- There are several other ways to embed an animation in a notebook, this one works quite well.
  - We closed the figure with `plt.close` so the static figure doesn't show up in addition to the animation.
  - Beware of very large animations in a Jupyter notebook, you might want to just save to a file with `Animation.save`.

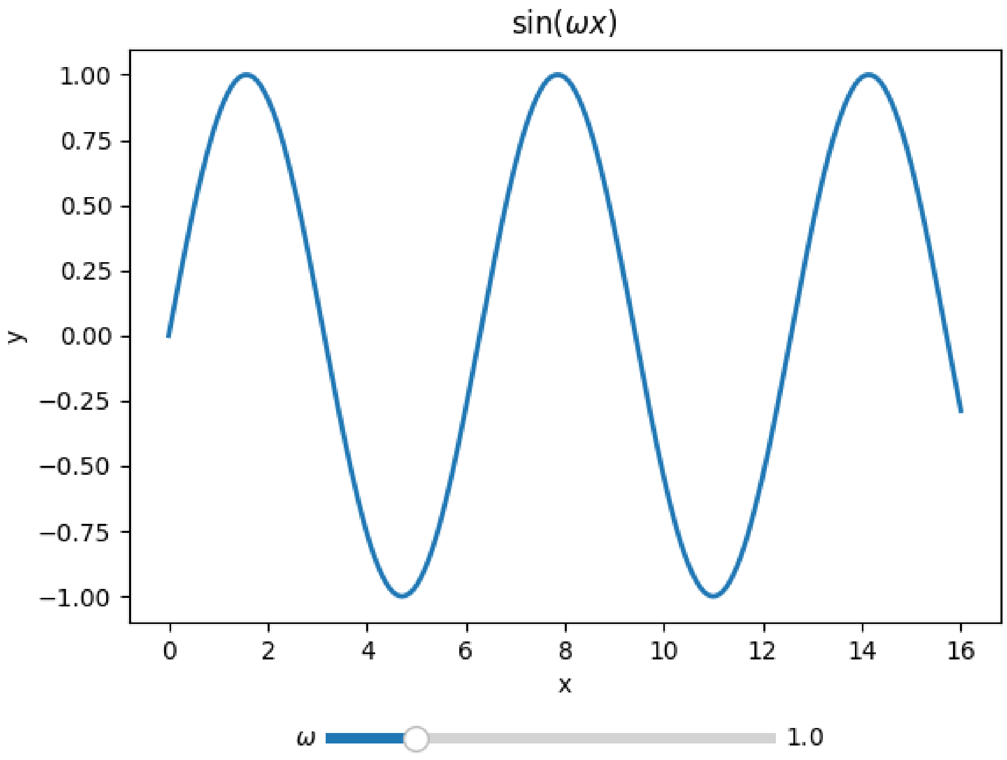
1.13 Interactivity

In addition to the built-in interactive capabilities of the axes object, you can add widgets such as buttons and sliders, capture events in the Python code and update the plot as needed.

This probably shouldn't be your first choice to develop a UI for an application, but can be useful to interactively adjust parameters. The example below adds a button to the figure, clicking it calls the `update` function that, like before, changes the existing artists.

[27]:
```
from matplotlib.widgets import Slider
fig, ax = plt.subplots()
fig.subplots_adjust(bottom=0.2)
x = np.linspace(0, 16, 256)
```

```
y = np.sin(x)
l, = ax.plot(x, y, linewidth=2)
ax.set_title(r'$\sin(\omega x)$')
ax.set_xlabel('x')
ax.set_ylabel('y')

def update(value):
    y = np.sin(x*value)
    l.set_ydata(y)

ax_slider = fig.add_axes([0.3, 0.05, 0.4, 0.025])
slider = Slider(ax_slider, r'$\omega$', 0.5, 3, valinit=1)
slider.on_changed(update);
```



[ ]:

```
```