

Beginner's Python Cheat Sheet - Matplotlib

What is Matplotlib?

Data visualization involves exploring data through visual representations. The Matplotlib library helps you make visually appealing representations of the data you're working with. Matplotlib is extremely flexible; these examples will help you get started with a few simple visualizations.

Many newer plotting libraries are wrappers around Matplotlib, and understanding Matplotlib will help you use those libraries more effectively as well.

Installing Matplotlib

Installing Matplotlib with pip

```
$ python -m pip install --user matplotlib
```

Line graphs and scatter plots

Making a line graph

The `fig` object represents the entire figure, or collection of plots; `ax` represents a single plot in the figure. This convention is used even when there's only one plot in the figure.

```
import matplotlib.pyplot as plt
```

```
x_values = [0, 1, 2, 3, 4, 5]
squares = [0, 1, 4, 9, 16, 25]
```

```
fig, ax = plt.subplots()
ax.plot(x_values, squares)
```

```
plt.show()
```

Making a scatter plot

`scatter()` takes a list of `x` and `y` values; the `s=10` argument controls the size of each point.

```
import matplotlib.pyplot as plt
```

```
x_values = list(range(1000))
squares = [x**2 for x in x_values]
```

```
fig, ax = plt.subplots()
ax.scatter(x_values, squares, s=10)
plt.show()
```

Customizing plots

Plots can be customized in a wide variety of ways. Just about any element of a plot can be modified.

Using built-in styles

Matplotlib comes with a number of built-in styles, which you can use with one additional line of code. The style must be specified before you create the figure.

```
import matplotlib.pyplot as plt
```

```
x_values = list(range(1000))
squares = [x**2 for x in x_values]
```

```
plt.style.use('seaborn-v0_8')
fig, ax = plt.subplots()
ax.scatter(x_values, squares, s=10)
```

```
plt.show()
```

Seeing available styles

You can see all available styles on your system. This can be done in a terminal session.

```
>>> import matplotlib.pyplot as plt
>>> plt.style.available
['Solarize_Light2', '_classic_test_patch', ...]
```

Adding titles and labels, and scaling axes

```
import matplotlib.pyplot as plt
```

```
x_values = list(range(1000))
squares = [x**2 for x in x_values]
```

```
# Set overall style to use, and plot data.
plt.style.use('seaborn-v0_8')
fig, ax = plt.subplots()
ax.scatter(x_values, squares, s=10)
```

```
# Set chart title and label axes.
ax.set_title('Square Numbers', fontsize=24)
ax.set_xlabel('Value', fontsize=14)
ax.set_ylabel('Square of Value', fontsize=14)
```

```
# Set scale of axes, and size of tick labels.
ax.axis([0, 1100, 0, 1_100_000])
ax.tick_params(axis='both', labelsize=14)
```

```
plt.show()
```

Using a colormap

A colormap varies the point colors from one shade to another, based on a certain value for each point. The value used to determine the color of each point is passed to the `c` argument, and the `cmap` argument specifies which colormap to use.

```
ax.scatter(x_values, squares, c=squares,
           cmap=plt.cm.Blues, s=10)
```

Customizing plots (cont.)

Emphasizing points

You can plot as much data as you want on one plot. Here we replot the first and last points larger to emphasize them.

```
import matplotlib.pyplot as plt
```

```
x_values = list(range(1000))
squares = [x**2 for x in x_values]
```

```
fig, ax = plt.subplots()
ax.scatter(x_values, squares, c=squares,
           cmap=plt.cm.Blues, s=10)
```

```
ax.scatter(x_values[0], squares[0], c='green',
           s=100)
ax.scatter(x_values[-1], squares[-1], c='red',
           s=100)
```

```
ax.set_title('Square Numbers', fontsize=24)
--snip--
```

Removing axes

You can customize or remove axes entirely. Here's how to access each axis, and hide it.

```
ax.get_xaxis().set_visible(False)
ax.get_yaxis().set_visible(False)
```

Setting a custom figure size

You can make your plot as big or small as you want by using the `figsize` argument. The `dpi` argument is optional; if you don't know your system's resolution you can omit the argument and adjust the `figsize` argument accordingly.

```
fig, ax = plt.subplots(figsize=(10, 6),
                       dpi=128)
```

Saving a plot

The Matplotlib viewer has a save button, but you can also save your visualizations programmatically by replacing `plt.show()` with `plt.savefig()`. The `bbox_inches` argument reduces the amount of whitespace around the figure.

```
plt.savefig('squares.png', bbox_inches='tight')
```

Online resources

The matplotlib gallery and documentation are at matplotlib.org/. Be sure to visit the Examples, Tutorials, and User guide sections.

Python Crash Course

A Hands-on, Project-Based
Introduction to Programming

ehmatthes.github.io/pcc_3e



Multiple plots

You can make as many plots as you want on one figure. When you make multiple plots, you can emphasize relationships in the data. For example you can fill the space between two sets of data.

Plotting two sets of data

Here we use `ax.scatter()` twice to plot square numbers and cubes on the same figure.

```
import matplotlib.pyplot as plt

x_values = list(range(11))
squares = [x**2 for x in x_values]
cubes = [x**3 for x in x_values]

plt.style.use('seaborn-v0_8')
fig, ax = plt.subplots()

ax.scatter(x_values, squares, c='blue', s=10)
ax.scatter(x_values, cubes, c='red', s=10)

plt.show()
```

Filling the space between data sets

The `fill_between()` method fills the space between two data sets. It takes a series of x-values and two series of y-values. It also takes a `facecolor` to use for the fill, and an optional `alpha` argument that controls the color's transparency.

```
ax.fill_between(x_values, cubes, squares,
               facecolor='blue', alpha=0.25)
```

Working with dates and times

Many interesting data sets have a date or time as the x value. Python's `datetime` module helps you work with this kind of data.

Generating the current date

The `datetime.now()` function returns a `datetime` object representing the current date and time.

```
from datetime import datetime as dt
```

```
today = dt.now()
date_string = today.strftime('%m/%d/%Y')
print(date_string)
```

Generating a specific date

You can also generate a `datetime` object for any date and time you want. The positional order of arguments is year, month, and day. The hour, minute, second, and microsecond arguments are optional.

```
from datetime import datetime as dt
```

```
new_years = dt(2023, 1, 1)
fall_equinox = dt(year=2023, month=9, day=22)
```

Working with dates and times (cont.)

Datetime formatting arguments

The `strftime()` function generates a `datetime` object from a string, and the `strptime()` method generates a formatted string from a `datetime` object. The following codes let you work with dates exactly as you need to.

%A	Weekday name, such as Monday
%B	Month name, such as January
%m	Month, as a number (01 to 12)
%d	Day of the month, as a number (01 to 31)
%Y	Four-digit year, such as 2021
%y	Two-digit year, such as 21
%H	Hour, in 24-hour format (00 to 23)
%I	Hour, in 12-hour format (01 to 12)
%p	AM or PM
%M	Minutes (00 to 59)
%S	Seconds (00 to 61)

Converting a string to a datetime object

```
new_years = dt.strptime('1/1/2023', '%m/%d/%Y')
```

Converting a datetime object to a string

```
ny_string = new_years.strftime('%B %d, %Y')
print(ny_string)
```

Plotting high temperatures

The following code creates a list of dates and a corresponding list of high temperatures. It then plots the high temperatures, with the date labels displayed in a specific format.

```
from datetime import datetime as dt

import matplotlib.pyplot as plt
from matplotlib import dates as mdates

dates = [
    dt(2023, 6, 21), dt(2023, 6, 22),
    dt(2023, 6, 23), dt(2023, 6, 24),
]

highs = [56, 57, 57, 64]

plt.style.use('seaborn-v0_8')
fig, ax = plt.subplots()
ax.plot(dates, highs, c='red')

ax.set_title("Daily High Temps", fontsize=24)
ax.set_ylabel("Temp (F)", fontsize=16)
x_axis = ax.get_xaxis()
x_axis.set_major_formatter(
    mdates.DateFormatter('%B %d %Y')
)
fig.autofmt_xdate()

plt.show()
```

Multiple plots in one figure

You can include as many individual graphs in one figure as you want.

Sharing an x-axis

The following code plots a set of squares and a set of cubes on two separate graphs that share a common x-axis. The `plt.subplots()` function returns a figure object and a tuple of axes. Each set of axes corresponds to a separate plot in the figure. The first two arguments control the number of rows and columns generated in the figure.

```
import matplotlib.pyplot as plt

x_values = list(range(11))
squares = [x**2 for x in x_values]
cubes = [x**3 for x in x_values]

fig, axs = plt.subplots(2, 1, sharex=True)

axs[0].scatter(x_values, squares)
axs[0].set_title('Squares')

axs[1].scatter(x_values, cubes, c='red')
axs[1].set_title('Cubes')

plt.show()
```

Sharing a y-axis

To share a y-axis, use the `sharey=True` argument.

```
import matplotlib.pyplot as plt

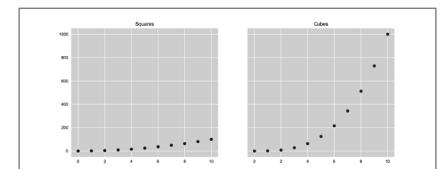
x_values = list(range(11))
squares = [x**2 for x in x_values]
cubes = [x**3 for x in x_values]

plt.style.use('seaborn-v0_8')
fig, axs = plt.subplots(1, 2, sharey=True)

axs[0].scatter(x_values, squares)
axs[0].set_title('Squares')

axs[1].scatter(x_values, cubes, c='red')
axs[1].set_title('Cubes')

plt.show()
```



Weekly posts about all things Python

mostlypython.substack.com