Lecture 3: Numpy and Matplotlib

Data arrays and plotting

Nils Olovsson
nils.olofsson@igp.uu.se

Introduction to data science and Python programming for medical research
Uppsala University

2025

UPPSALA UNIVERSITET

Content

Introduction

Introduction

Introduction: Numpy and Matplotlib

Previously saw **lists**, **tuples** and **dictionaries** for collecting things.

Flexible but **not** always computationally **efficient**.

Need **special class** for **numerical data**.
Numpy arrays are the standard in Python.

We also want to create plots from our data!
For this we will use the **matplotlib** package.

Numpy arrays

Numpy arrays

Numpy arrays: `np.ndarray`

Numpy arrays are similar to Python lists but are of **constant type**. Often numerical type, **int** or **float**.

Can have as many dimensions as required but commonly 1D, 2D or 3D.

1D Vector

2D Matrix

ND "Tensor"

To use **numpy** it is standard to **import** it using the `np` shorthand as:
`import numpy as np`.

```python
import numpy as np
a = np.array([1, 6, 8])
b = np.array([[1.0, 6.0],
              [5.0, 3.0]])
print(type(a))
print(a)
print(b)
```

```
>> <class 'numpy.ndarray'>
>> [1 6 8]
>> [[1. 6.]
>>  [5. 3.]]
```

Numpy arrays: Attributes

Numpy arrays are **instances** of the **class** np.ndarray. This class contains attributes we can inspect. Especially the `shape` and `dtype` is often important!

```python
import numpy as np
def print_attributes(
    arr: np.ndarray,
) -> None:
    """ Print important attributes.
    """
    print(f"shape: {arr.shape}")
    print(f"size:  {arr.size}")
    print(f"ndim:  {arr.ndim}")
    print(f"dtype: {arr.dtype}")
# Define two arrays
a = np.array([1, 6, 8])
b = np.array([[1.0, 6.0],
              [5.0, 3.0]])
# Print the attributes for  both arrays
print("a")
print_attributes(a)
print("\nb")
print_attributes(b)
```

```
>> a
>> shape: (3,)
>> size:  3
>> ndim:  1
>> dtype: int64
>>
>> b
>> shape: (2, 2)
>> size:  4
>> ndim:  2
>> dtype: float64
```

Numpy arrays: Initialization

Numpy arrays are **instances** of the **class** np.ndarray. This class contains attributes we can inspect. Especially the shape and dtype is often important!

```python
import numpy as np

# np.zeros: Arbitrary size with 0 everywhere
a = np.zeros(3)
# np.ones: Arbitrary size with 1 everywhere
b = np.ones(2)
# np.eye: 2D with 1 on diagonal, 0 elsewhere
c = np.eye(2, 2)
# np.empty: Arbitrary size, memory not set
d = np.empty(3)

print(f"np.zeros:\n{a}")
print(f"np.ones:\n{b}")
print(f"np.eye:\n{c}")
print(f"np.empty:\n{d}")
```

```
>> np.zeros:
>> [0. 0. 0.]
>> np.ones:
>> [1. 1.]
>> np.eye:
>> [[1. 0.]
>>  [0. 1.]]
>> np.empty:
>> [1. 1. 1.]
```

https://numpy.org/doc/stable/reference/routines.array-creation.html

Numpy arrays: Ranges

Sequences of evenly spaced numbers can be created using `np.arange` or `np.linspace`.

`np.arange` is useful when we **know the spacing** we want between the numbers.

`np.linspace` is useful when we **know the number of points** between two numbers.

```python
import numpy as np

# Linearly spaced integers from 0 to 5
# (not including!)
r0 = np.arange(5)
# Linearly spaced integers from 5 to 10
# (not including 10!)
r1 = np.arange(5, 10)
# Just like 'range', can count backwards,
# from 65 to 50
r2 = np.arange(65, 50, -5)
# 8 equally spaced numbers on [0, 4]
r3 = np.linspace(0, 4, 8)

# Print the arrays
print(f"{r0}")
print(f"{r1}")
print(f"{r2}")
print(f"{r3}")
```

```
>> [0 1 2 3 4]
>> [5 6 7 8 9]
>> [65 60 55]
>> [0.         0.57142857 1.14285714
     1.71428571 2.28571429 2.85714286
>>  3.42857143 4.        ]
```

https://numpy.org/doc/stable/reference/generated/numpy.arange.html
https://numpy.org/doc/stable/reference/generated/numpy.linspace.html

Numpy arrays: Random

Numpy can generate random numbers from several distributions. These are a few examples.

`np.uniform` distribution.
`np.normal` normal distribution.
`np.integers` on some specified interval.

Setting the seed parameter to some value in the random generator is good for testing.
Typically set to a more *random* number. *E.g.* from the `time.time()`.

https://numpy.org/doc/stable/reference/random/index.html

```python
import numpy as np

# Initiate a random number generator
# If we set the seed to a number like this
# we are guaranteed to get the same
# "random" numbers every time!
rng = np.random.default_rng(seed=0)

# 4 samples from uniform distribution
# on interval [0, 1)
u = rng.uniform(0.0, 1.0, 4)
# 4 samples from normal distribution
# with mean 1 and standard deviation 1.5
a = rng.normal(1, 1.5, 4)
# 5 random integers on interval 1 to
# 10 (excluding 10)
i = rng.integers(1, 10, 5)

# Print the arrays with random numbers
print(f"{u}")
print(f"{a}")
print(f"{i}")
```

```
>> [0.63696169 0.26978671 0.04097352
    0.01652764]
>> [0.19649594 1.54239258 2.95600007
    2.42062144]
>> [6 5 6 9 3]
```

Numpy arrays: Type

Regular Python has two **types** of numbers, **integer** and **float**.

Numpy arrays also has **several types of each kind**. *E.g.* 8, 16, 32 and 64 bit versions and signed and unsigned integers.

We will also see **Boolean** arrays.

`dtype` argument can be given during initialization. It can later be inspected as an attribute as we have already seen.

Initializing from lists of floats with dtype set to an integer type will silently truncate any decimals!

```python
import numpy as np

# Initialize an array of floats
# from list of integers
a = np.array([0, 1, 2, 4], dtype=float)

# Silent truncation of all decimals!
b = np.array([0.5, 1.3, 2.8], dtype=int)

# Initialize to 8-bit unsigned integers
# Used for gray scale images.
# Values supported are all integers
# between 0 and 255.
c = np.array([0, 5, 78], dtype=np.uint8)

# Print the arrays
print(f"a ({a.dtype}) {a}")
print(f"b ({b.dtype}) {b}")
print(f"c ({c.dtype}) {c}")
```

```
>> a (float64) [0. 1. 2. 4.]
>> b (int64) [0 1 2]
>> c (uint8) [ 0  5 78]
```

Array access

Array access

Array access: Indexing

Indexing is the same as for Python lists and **first element is at index zero**.

Numpy arrays are **mutable** so we can alter them after creation.

When indexing into **2D arrays the axes are separated by a comma**. They are indexed as `array[row, column]`.

This generalized to even higher dimensional arrays.

```python
import numpy as np

# Initialize a 1D array
a = np.array([0, 1, 2, 4])

# Initialize a 2D array
b = np.array([[0, 1, 2],
              [3, 4, 5],
              [6, 7, 8]])

# Can index to get values
print(a[3])

# Indexing a 2D array is done
# selecting row, then column
print(b[1, 2])

# Can index to set values
a[2] = 5
b[2, 0] = 0
print(a)
print(b)
```

```
>> 4
>> 5
>> [0 1 5 4]
>> [[0 1 2]
>>  [3 4 5]
>>  [0 7 8]]
```

Array access: Slicing

Slicing also same as for Python lists. A slice interval is defined by `[start:stop]`, not including `stop`.

Slicing is a very powerful way of accessing subarrays.

When slicing into **2D arrays the axes are separated by a comma**. Indices per axis are still separated by a colon.

This generalized to even higher dimensional arrays.

```python
import numpy as np

# Initialize a 1D array
a = np.array([0, 1, 2, 4])

# Initialize a 2D array
b = np.array([[0, 1, 2],
              [3, 4, 5],
              [6, 7, 8]])

# Get middle two elements
print(a[1:3])

# Get lower right quadrant
print(b[1:3, 1:3])

# Can slice to set values
# The sub-arrays must match in size!
print(a)
a[1:3] = np.array([3, 3])
print(a)
```

```
>> [1 2]
>> [[4 5]
>>  [7 8]]
>> [0 1 2 4]
>> [0 3 3 4]
```

Array access: Fancy index

Integer indexing can be done using an **array of indices**.

```python
import numpy as np

# Initialize a 1D array
a = np.array([0, 1, 2, 4])

# Can create index array of other shape
# and still index into array
indices = np.array([[0, 3], [3, 0]])
print(a[indices])
```

```
>> [[0 4]
>>  [4 0]]
```

Array access: View or copy

Just as when dealing with objects or lists we must be careful whether it is a **copy or reference**.

Slicing always **returns a view**. *I.e.*, changing any element in this view also changes it in the original array.

```python
import numpy as np

# Initialize a 2D array
b = np.array([[0, 1, 2],
              [3, 4, 5],
              [6, 7, 8]])

b_slice = b[0:2, 0:2]
print("Value before")
print(b)
b_slice[0, 0] = 5
print("Value after (top left element changes)"
     )
print(b)
```

```
>> Value before
>> [[0 1 2]
>>  [3 4 5]
>>  [6 7 8]]
>> Value after (top left element changes)
>> [[5 1 2]
>>  [3 4 5]
>>  [6 7 8]]
```

Array shape

Array shape

https://numpy.org/doc/stable/reference/routines.array-manipulation.html

Array shape: Reshape and resize

Create an array with the **same content** and **same size** but different shape.

`reshape` creates a copy of the array.

`resize` modifies the array in place.

Old and new size must match!

```
import numpy as np

# Initialize a 2D array
a = np.array([[0, 1, 2],
              [3, 4, 5]])
print(a)

# Reshape from a (2, 3) to a (3, 2) shaped array
b = a.reshape((3, 2))
print(b)

# Reshape it to a flat 1D array
a.resize(6)
print(a)
```

```
>> [[0 1 2]
>>  [3 4 5]]
>> [[0 1]
>>  [2 3]
>>  [4 5]]
>> [0 1 2 3 4 5]
```

https://numpy.org/doc/stable/reference/generated/numpy.reshape.html
https://numpy.org/doc/stable/reference/generated/numpy.resize.html

Array shape: Flatten and ravel

An array of any size and shape can be flattened to a 1D representation.

`flatten` creates a copy of the array.

`ravel` creates a view of the array.

```python
import numpy as np

# Initialize a 2D array
a = np.array([[0, 1, 2],
              [3, 4, 5]])
print(a)

# Flatten array using flatten and ravel
b = a.flatten()
print(b)
c = a.ravel()
print(c)

# Changing value in the raveled array changes
# in the original array as well!
c[0] = 9
print(a)
print(c)
```

```
>> [[0 1 2]
>>  [3 4 5]]
>> [0 1 2 3 4 5]
>> [0 1 2 3 4 5]
>> [[9 1 2]
>>  [3 4 5]]
>> [9 1 2 3 4 5]
```

https://numpy.org/doc/stable/reference/generated/numpy.ndarray.flatten.html
https://numpy.org/doc/stable/reference/generated/numpy.ravel.html

Array shape: Concatenate and transpose

Arrays can be **joined end-to-end** by **concatenation**.

The dimension(s) along which we join must match!

Arrays can be flipped on their side by **transposing** them.

Stack on preset axis:
`hstack`, `vstack` and `dstack`.

```python
import numpy as np

# Initialize 2D arrays (shape = 2x1)
a = np.array([[1], [2]])
b = np.array([[3], [4]])
c = np.array([[5], [6]])
print(a)
print(b)
print(c)

# Concatenate three 2x1 arrays into one 2x3
# Input is a list of arrays
d = np.concatenate([a, b, c], axis=1)
print(d)
# Concatenate three 2x1 arrays into one 6x1
d = np.concatenate([a, b, c], axis=0)
print(d.T) # Transpose to 1x6 array
```

```
>> [[1]
>>  [2]]
>> [[3]
>>  [4]]
>> [[5]
>>  [6]]
>> [[1 3 5]
>>  [2 4 6]]
>> [[1 2 3 4 5 6]]
```

https://numpy.org/doc/stable/reference/generated/numpy.concatenate.html

Array shape: Split

Arrays can also be **split** into sub-arrays.

`split` to divide into equally sized sub-array views.
`array_split` to allow non-equally sized sub-arrays.

Split on preset axis:
`hsplit`, `vsplit` and `dsplit`.

```
import numpy as np

# Initialize a 2D array
a = np.array([[1, 2, 3, 4],
              [5, 6, 7, 8]])
print(a)


# Split along axis 1, column
splits = np.array_split(a, 2, axis=1)

# Result is two arrays in a list, one with the
# first two columns, one with last two columns
for arr in splits:
    print(arr)
```

```
>> [[1 2 3 4]
>>  [5 6 7 8]]
>> [[1 2]
>>  [5 6]]
>> [[3 4]
>>  [7 8]]
```

https://numpy.org/doc/stable/reference/generated/numpy.split.html

Array shape: Broadcast

Broadcasting is an implicit shape conversion during certain operations between arrays. *E.g.* $+$, $-$ and other arithmetic operators.

This permits operations of arrays of **different but compatible shapes**.

(Implicit conversions like this should always be handled carefully!)

```
import numpy as np

# Initialize a 2D array
a = np.array([[1, 2, 3],
              [4, 5, 6]])

# Initialize a 1D array
b = np.array([1, 2, 3])

# The shapes of the arrays are not the
# same but we can still add 'a' to 'b'
# since 'a' is implicitly broadcast to
# match the shape of 'b'.
# As such 'b' is added to each row of 'a'.
#c = b + a
c = a + b

print(f'a:\n{a}')
print(f'b:\n{b}')
print(f'c:\n{c}')
```

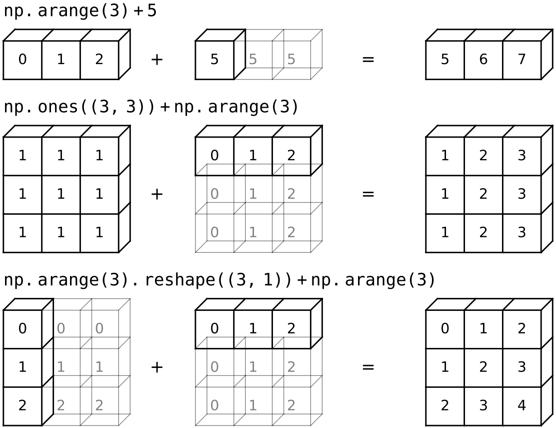
```
>> a:
>> [[1 2 3]
>>  [4 5 6]]
>> b:
>> [1 2 3]
>> c:
>> [[2 4 6]
>>  [5 7 9]]
```

https://numpy.org/doc/stable/reference/generated/numpy.broadcast.html

Array shape: Broadcast

Three examples that show different ways of initializing arrays.

They are then added using broadcast.



Adapted from VanderPlas "Python Data Science Handbook".

https://numpy.org/doc/stable/reference/generated/numpy.broadcast.html

https://github.com/jakevdp/PythonDataScienceHandbook

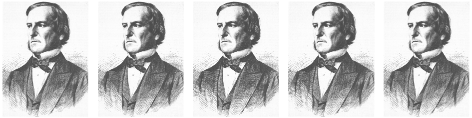
Array masks

Array masks

Array masks: Comparisons

Comparing a numeric array to a number results in an **array of Boolean type**.

All the regular comparison operators in Python are valid.



```python
import numpy as np

# Initialize a 2D array
a = np.array([[1, 5, 3],
              [4, 2, 6]])

# Comparison to create a Boolean array
b = a > 4

print(f'a:\n{a}')
print(f'b:\n{b}')
```

```
>> a:
>> [[1 5 3]
>>  [4 2 6]]
>> b:
>> [[False  True False]
>>  [False False  True]]
```

Array masks: Counting

```python
import numpy as np

# Initialize a 2D array
a = np.array([[1, 5, 3],
              [4, 2, 6]])

# Comparison to create a Boolean array
b = a > 4
print(f'a:\n{a}')
print(f'b:\n{b}')

# Count number of elements larger than 4
n = np.sum(b)
m = np.count_nonzero(b)
print(f"Number of elements > 4: {n}")
print(f"Number of elements > 4: {m}")

# Check if all or any elements are True
all_true = np.all(b)
any_true = np.any(b)
print(f"All elements > 4: {all_true}")
print(f"Any elements > 4: {any_true}")
```

This can be used to **count elements** of certain ranges or values using `np.count_nonzero()` or `np.sum()`.

To **summarize the result** of all Boolean elements in the array after a comparison use `np.any()` or `np.all()`.

```
>> a:
>> [[1 5 3]
>>  [4 2 6]]
>> b:
>> [[False  True False]
>>  [False False  True]]
>> Number of elements > 4: 2
>> Number of elements > 4: 2
>> All elements > 4: False
>> Any elements > 4: True
```

Array masks: Boolean operators

Can operate on these arrays just like regular Python Booleans but use operators usually reserved for bitwise logic.

- & Logical and.
- | Logical or.
- ^ Logical exclusive or.
- ~ Logical invert.

https://numpy.org/doc/stable/reference/routines.bitwise.html

```python
import numpy as np

# Initialize a 2D array
a = np.array([[1, 5, 3],
              [4, 2, 6]])

# Comparison to create a Boolean array
b = a > 4
c = a < 6

print(b)
print(~b)
print(b & c)
print(b | c)
print(b ^ c)
```

```
>> [[False  True False]
>>  [False False  True]]
>> [[ True False  True]
>>  [ True  True False]]
>> [[False  True False]
>>  [False False False]]
>> [[ True  True  True]
>>  [ True  True  True]]
>> [[ True False  True]
>>  [ True  True  True]]
```

Array masks: Mask indexing

```python
import numpy as np

# Initialize a 2D array
a = np.array([[1, 5, 3],
              [4, 2, 6]])

# Comparison to create a Boolean array
b = a > 4

# Use the Boolean array to index into
# original array.
print(f'a:\n{a}')
print(f'a>4:\n{a[b]}')

# Can use this to write to selected elements
a[b] = 9
print(f'a:\n{a}')
```

The Boolean arrays can be used to **mask** another array of the same shape to **index** only the elements where the Boolean, mask, array is True.

Can be used both to set and to get values.

The return value from indexing is always a 1D array.

```
>> a:
>> [[1 5 3]
>>  [4 2 6]]
>> a>4:
>> [5 6]
>> a:
>> [[1 9 3]
>>  [4 2 9]]
```

Numpy operators

Numpy operators

Numpy operators: Element wise functions

Supports **element wise operations** between other numbers or arrays of the same shape.
The operators are **same as** for float and int in **Python**.

```
+  np.add
-  np.subtract
*  multiply
** np.power
/  np.divide
// np.floor_divide
%  np.mod
```

```python
import numpy as np

# Initialize a 2D array
a = np.array([[1, 5, 3],
              [4, 2, 6]])

# Comparison to create a Boolean array
print(f'a:\n{a}')
a = a + 2
print(f'a:\n{a}')

a = a // 2
print(f'a:\n{a}')
```

```
>> a:
>> [[1 5 3]
>>  [4 2 6]]
>> a:
>> [[3 7 5]
>>  [6 4 8]]
>> a:
>> [[1 3 2]
>>  [3 2 4]]
```

Numpy operators: Math functions

All common mathematical functions exist in numpy.

The difference between the `math.sin()` and `np.sin()` is that the latter **operates on all values in an array**

`np.sin`

`np.cos`

`np.tan`

`np.exp`

...

https://numpy.org/doc/stable/reference/routines.math.html

```python
import numpy as np

# Initialize a 2D array
a = np.array([[1, 5, 3],
              [4, 2, 6]])

# Comparison to create a Boolean array
print(f'a:\n{a}')
s = np.sin(a)
print(f's:\n{s}')
```

```
>> a:
>> [[1 5 3]
>>  [4 2 6]]
>> s:
>> [[ 0.84147098 -0.95892427  0.14112001]
>>  [-0.7568025   0.90929743 -0.2794155 ]]
```

Numpy operators: Aggregates

Aggregates such as sum can be computed with specialized numpy functions.

`np.sum`

`np.prod`

`np.mean`

`np.std`

`np.var`

`np.min`

`np.max`

...

https://numpy.org/doc/stable/reference/routines.math.html

```python
import numpy as np

# Initialize a 2D array
a = np.array([[1, 5, 3],
              [4, 2, 6]])

# Print the array and aggregates
# or summary metrics
print(f'a:\n{a}\n')

print(f'min:  {np.min(a)}')
print(f'max:  {np.max(a)}')
print(f'sum:  {np.sum(a)}')
print(f'mean: {np.mean(a)}')
print(f'SD:   {np.std(a)}')
```

```
>> a:
>> [[1 5 3]
>>  [4 2 6]]
>>
>> min:  1
>> max:  6
>> sum:  21
>> mean: 3.5
>> SD:   1.707825127659933
```

Numpy summary

Numpy summary

Numpy summary: Sorting

```
import numpy as np

# Initialize a 1D and a 2D array
a = np.array([9, 5, 3, 4, 1, 2])
b = np.array([[9, 5, 3],
              [4, 1, 2]])

# The 1D array can be sorted in
# place using the array method .sort()
print(f'a:\n{a}')
a.sort()
print(f'a:\n{a}')

# The 2D arrays can be sorted
# along either axis using numpys
# function sort()
print(f'b:\n{b}')
c = np.sort(b, axis=0)
print(f'b sorted per column:\n{c}')
d = np.sort(b)
print(f'b sorted per row:\n{d}')
```

The content of an array can be sorted.

```
>> a:
>> [9 5 3 4 1 2]
>> a:
>> [1 2 3 4 5 9]
>> b:
>> [[9 5 3]
>>  [4 1 2]]
>> b sorted per column:
>> [[4 1 2]
>>  [9 5 3]]
>> b sorted per row:
>> [[3 5 9]
>>  [1 2 4]]
```

Numpy summary: Read and write

Numpy arrays can be **saved** and **loaded**.

Arrays can of course be saved using **pickle** or even as text files by opening and writing the content *manually*.

The simplest way is to **use numpys own functionality** and `np.save` and `np.load`.

```python
import os
import numpy as np

# Initialize a 2D array
a = np.array([[1, 5, 3],
              [4, 2, 6]])

# Save the array to a numpy .npy file
filepath = 'test-numpy.npy'
np.save(filepath, a)

# Load the array
b = np.load(filepath)
print(b)

# Delete the file
os.remove(filepath)
```

```
>> [[1 5 3]
>>  [4 2 6]]
```

Numpy summary: Summary

Numpy arrays are **similar to Python lists** but have several **advantages**.

- More compact and **memory efficient**.
- More **computationally efficient**.
- Offers **slicing** of high dimensional arrays.
- Comes with all the **mathematical functions** of Numpy.

Numpy arrays are at the heart of all Python packages we will encounter!

Therefore at the **heart of all data processing and machine learning** we will encounter!

Matplotlib figure

Matplotlib figure

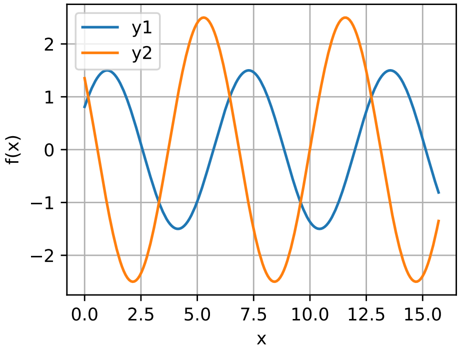
Matplotlib figure: Create a plot

```python
import numpy as np
import matplotlib.pyplot as plt

# Create a figure and get the axis
fig = plt.figure(figsize=(4,3))
ax: plt.Axes = fig.gca()

# Create data to plot
x = np.linspace(0, 5 * np.pi, 200)
y1 = 1.5 * np.cos(x - 1)
y2 = 2.5 * np.cos(x + 1)

# Plot data corresponding to two
# periodic mathematical functions
ax.plot(x, y1, label='y1')
ax.plot(x, y2, label='y2')

# Decorate the plot a little
ax.grid(True)
ax.set_xlabel('x')
ax.set_ylabel('f(x)')
ax.legend()

#plt.show()
```

https://matplotlib.org/stable/api/_as_gen/matplotlib.figure.Figure.html

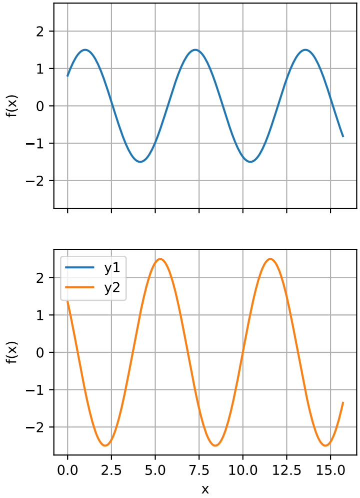
Matplotlibs plotting is commonly imported using the shorthand name `plt`.

Use the **axis** instance in the **figure** to plot.



Matplotlib figure: Subplots

```python
import numpy as np
import matplotlib.pyplot as plt

# Create subplots in two rows and one column
fig, axs = plt.subplots(2, 1, figsize=(4,6),
                        sharex=True, sharey=True)
ax: plt.Axes = fig.gca()

# Create data to plot
x = np.linspace(0, 5 * np.pi, 200)
y1 = 1.5 * np.cos(x - 1)
y2 = 2.5 * np.cos(x + 1)

# Plot data corresponding to two
# periodic mathematical functions
p1 = axs[0].plot(x, y1, 'tab:blue')
p2 = axs[1].plot(x, y2, 'tab:orange')

# Decorate the plot a little
for i, ax in enumerate(axs):
    ax.grid(True)
    ax.set_ylabel('f(x)')
axs[1].set_xlabel('x')
axs[1].legend(handles = [p1[0], p2[0]],
              labels = ['y1', 'y2'])
#plt.show()
```

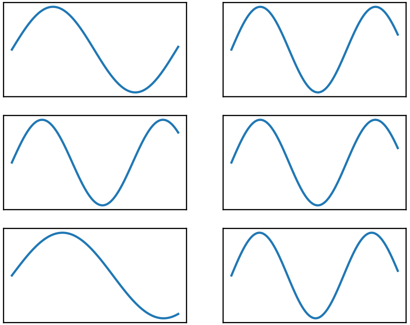


https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.subplots.html

Matplotlib figure: Subplot creation

```
import numpy as np
import matplotlib.pyplot as plt

# Initialize a random number generator
rng = np.random.default_rng(seed=0)

# Create subplots dynamically
# by adding them to figure
fig = plt.figure(figsize=(5,4))

x = np.linspace(0, 10, 100)
for index in range(1, 3 * 2 + 1):
    # Add a subplot
    ax = fig.add_subplot(3, 2, index)

    # Plot a random dummy function
    params = rng.uniform(0.5, 1, 3)
    b = params[0] * np.sin(params[1] * x) + params[2]
    ax.plot(x, b)
    ax.tick_params(axis='both', which='both', length=0)
    plt.setp(ax.get_xticklabels(), visible=False)
    plt.setp(ax.get_yticklabels(), visible=False)
#plt.show()
```

Subplots can also be created by adding them dynamically to a figure.

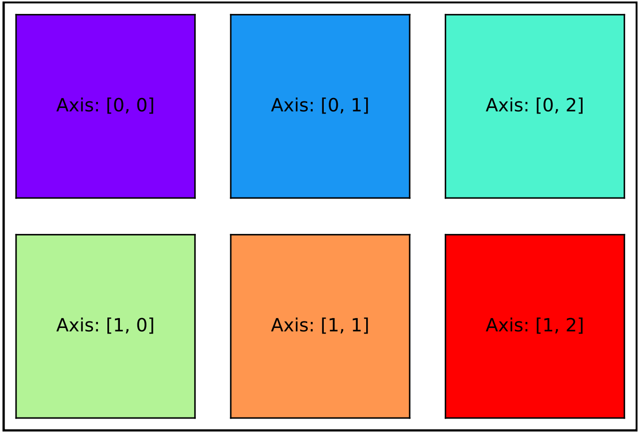


https://matplotlib.org/stable/api/_as_gen/matplotlib.figure.Figure.add_subplot.html

Matplotlib figure: Axes

The **axes** are where all the plotting is done.

The axes reside inside a **figure**.



https://matplotlib.org/stable/api/_as_gen/matplotlib.axes.Axes.html

Matplotlib plots

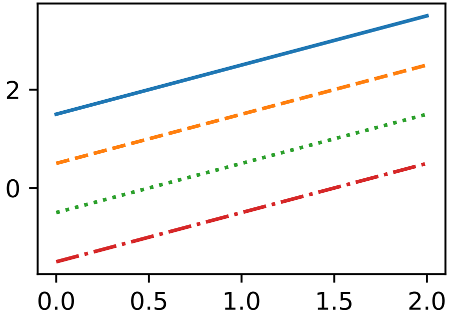
Matplotlib plots

Matplotlib plots: Plot

The standard plot draws a line between points defined by two arrays or lists of numbers, x and y. The arrays need to have the same size.

Plots can be differentiated using color and line style.

```python
import numpy as np
import matplotlib.pyplot as plt

fig = plt.figure(figsize=(3,2))
ax = fig.gca()

# Calculate x and line y = x
x = np.linspace(0, 2, 10)
y = x

# Plot the lines using different styles
ax.plot(x, y + 1.5, linestyle='-')
ax.plot(x, y + 0.5, linestyle='--')
ax.plot(x, y - 0.5, linestyle=':')
ax.plot(x, y - 1.5, linestyle='-.')

#plt.show()
```



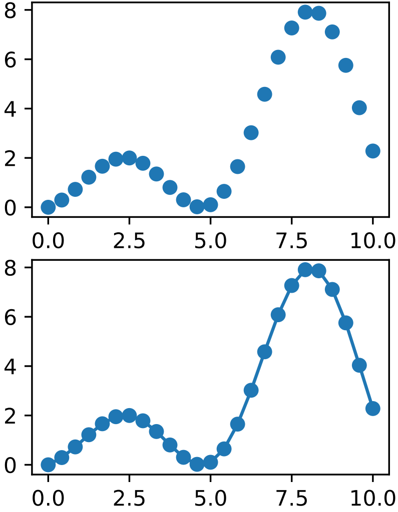
https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.plot.html
https://matplotlib.org/stable/gallery/lines_bars_and_markers/linestyles.html

Matplotlib plots: Scatter (`plot()`)

Points in the arrays can be plotted with scatter plots by setting a marker in the regular plot function.

```python
import numpy as np
import matplotlib.pyplot as plt

fig, axs = plt.subplots(2, 1, figsize=(3, 4))

# Calculate x and y(x)
x = np.linspace(0, 10, 25)
y = 0.5 * x * np.sin(x) + 0.5 * x

# Plot using round markers without
# and with line
axs[0].plot(x, y, 'o')
axs[1].plot(x, y, '-o')

#plt.show()
```

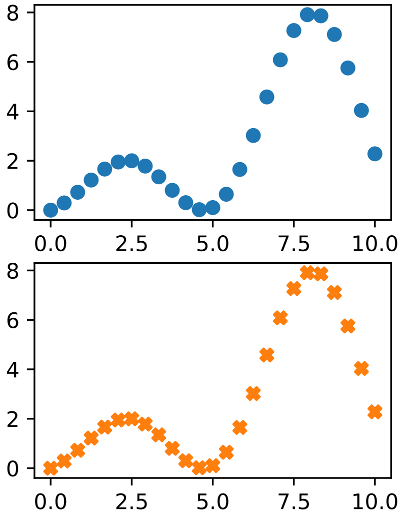
https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.plot.html



Matplotlib plots: Scatter (`scatter()`)

Scatter plots can, preferably, be created using the special scatter plot method.

```python
import numpy as np
import matplotlib.pyplot as plt

fig, axs = plt.subplots(2,1,figsize=(3,4))

# Calculate x and y(x)
x = np.linspace(0, 10, 25)
y = 0.5 * x * np.sin(x) + 0.5 * x

# Plot using two different markers
axs[0].scatter(x, y, marker='o')
axs[1].scatter(x, y, marker='X', color='tab:orange')

#plt.show()
```

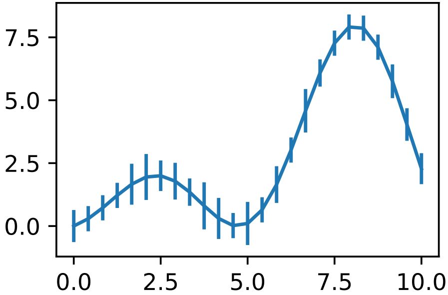


https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.scatter.html

Matplotlib plots: Errorbars (`errorbar()`)

Estimated errors or uncertainty can be visualized using error bars.

```
import numpy as np
import matplotlib.pyplot as plt

fig = plt.figure(figsize=(3,2))
ax = fig.gca()

# Calculate x and y(x)
x = np.linspace(0, 10, 25)
y = 0.5 * x * np.sin(x) + 0.5 * x

# Generate some random "errors"
rng = np.random.default_rng(seed=0)
y_err = rng.random(y.shape)
y_err[y_err<0.5] = 0.5

# Plot with error bars
ax.errorbar(x, y, yerr=y_err)

#plt.show()
```

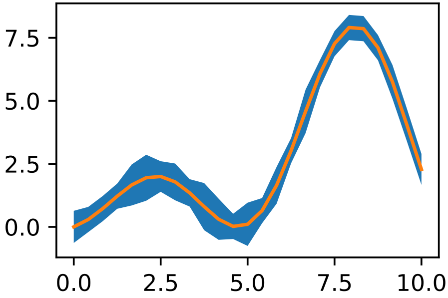


https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.errorbar.html

Matplotlib plots: Fill(`fill_between()`)

Estimated errors or uncertainty can also be visualized using filled curves. This is also very good for creating *area-under-curve* plots *e.g.* for integrals.

```
import numpy as np
import matplotlib.pyplot as plt

fig = plt.figure(figsize=(3,2))
ax = fig.gca()

# Calculate x and y(x)
x = np.linspace(0, 10, 25)
y = 0.5 * x * np.sin(x) + 0.5 * x

# Generate some random "errors"
rng = np.random.default_rng(seed=0)
y_err = rng.random(y.shape)
y_err[y_err<0.5] = 0.5

# Plot with filled curve
ax.fill_between(x, y - y_err, y + y_err)
ax.plot(x, y, color='tab:orange')

#plt.show()
```



https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.fill_between.html

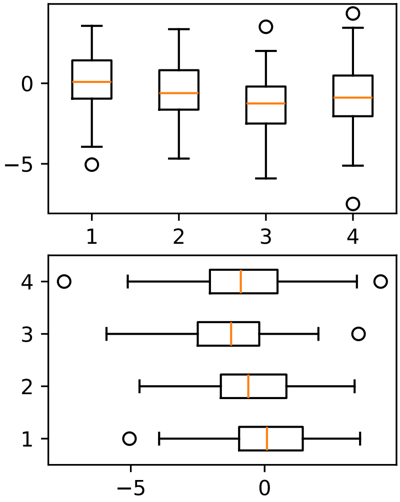
Matplotlib plots: Boxplots (boxplot())

Boxplots can be used to visualize distributions.

Shown is the median and the Q1 and Q3 quartiles. Whiskers are calculated (see the link) and outliers can be shown as points.

```python
import numpy as np
import matplotlib.pyplot as plt

fig, axs = plt.subplots(2, 1, figsize=(3, 4))
axs_list: List[plt.Axes] = list(axs.flatten())

rng = np.random.default_rng(seed=0)
N = 4
m = rng.uniform(-1, 1, N)
s = rng.uniform(1, 2, N)
v = rng.normal(m, s, size=(100, N))

axs_list[0].boxplot(v)
axs_list[1].boxplot(v, orientation='horizontal')

#plt.show()
```

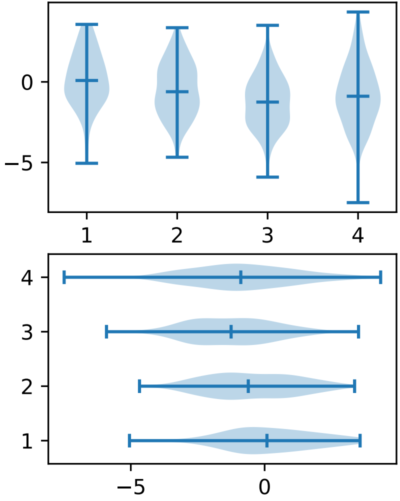
https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.boxplot.html



Matplotlib plots: Violin plots (`violinplot()`)

More general one dimensional distributions can also be visualized with violin plots.

```python
import numpy as np
import matplotlib.pyplot as plt

fig, axs = plt.subplots(2, 1, figsize=(3, 4))
axs_list: List[plt.Axes] = list(axs.flatten())

rng = np.random.default_rng(seed=0)
N = 4
m = rng.uniform(-1, 1, N)
s = rng.uniform(1, 2, N)
v = rng.normal(m, s, size=(100, N))

axs_list[0].violinplot(v, showmedians=True)
axs_list[1].violinplot(v, showmedians=True, orientation='horizontal')

#plt.show()
```



https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.violinplot.html

Matplotlib elements

Matplotlib elements

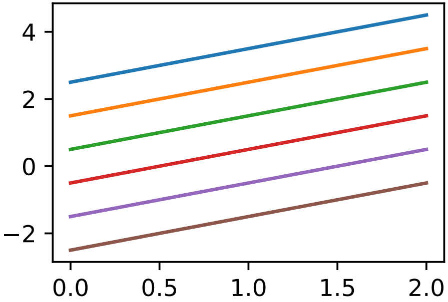
Matplotlib elements: Colors

```
import numpy as np
import matplotlib.pyplot as plt

fig = plt.figure(figsize=(3,2))
ax = fig.gca()

# Define a straight line y = x
x = np.linspace(0, 2, 10)
y = x

# Plot offset lines with different colors
ax.plot(x, y + 2.5, color='tab:blue')
ax.plot(x, y + 1.5, color='tab:orange')
ax.plot(x, y + 0.5, color='tab:green')
ax.plot(x, y - 0.5, color='tab:red')
ax.plot(x, y - 1.5, color='tab:purple')
ax.plot(x, y - 2.5, color='tab:brown')

#plt.show()
```

https://matplotlib.org/stable/gallery/color/named_colors.html

Indicate different plots by line style, glyphs/markers or color.

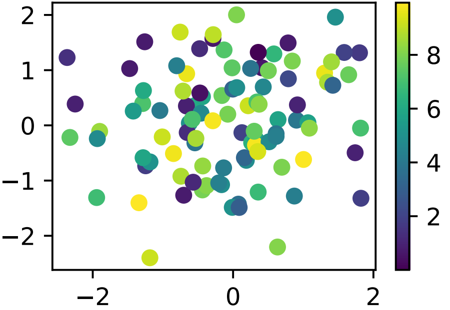
Matplotlib has many named colors. Use a limited palette! Be colorblindness and gray scale conversion aware!



Matplotlib elements: Colormaps and colorbars

```python
import numpy as np
import matplotlib.pyplot as plt

fig = plt.figure(figsize=(3,2))
ax = fig.gca()

# Create randomly scattered points
# Assign each point a value, z.
rng = np.random.default_rng(seed=0)
xy = rng.normal(0, 1, size=(100, 2))
z = rng.uniform(0, 10, 100)

# Use scatter plot and add a colorbar
sc = ax.scatter(xy[:,0], xy[:,1], c=z, cmap='viridis')
fig.colorbar(ax=ax, mappable=sc)

#plt.show()
```

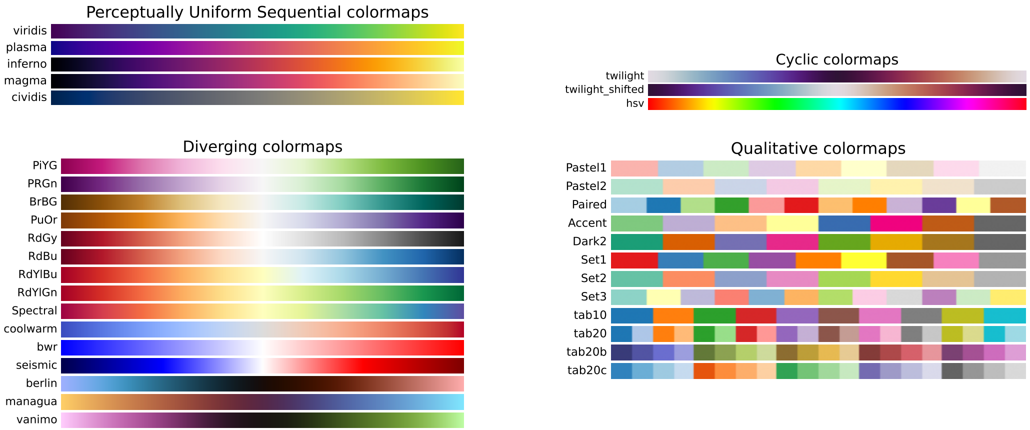
A colormap should be used to indicate the value of data when plotted without an axis for the value, *e.g.* in 2D plots.



https://matplotlib.org/stable/users/explain/colors/colormaps.html

Matplotlib elements: Choice of colormap

Matplotlib comes with many built in colormaps usable in different contexts.

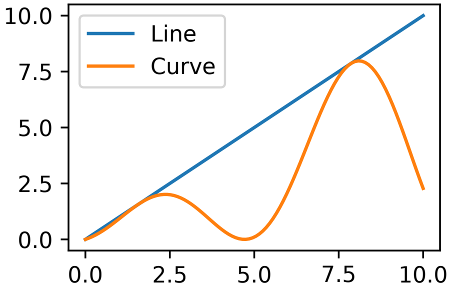


https://matplotlib.org/stable/users/explain/colors/colormaps.html

Matplotlib elements: Legends

```
import numpy as np
import matplotlib.pyplot as plt

fig = plt.figure(figsize=(3,2))
ax = fig.gca()

# Calculate x and y(x)
x = np.linspace(0, 10, 100)
y0 = x
y1 = 0.5 * x * np.sin(x) + 0.5 * x

# Add labels to the plots!
ax.plot(x, y0, color='tab:blue', label='Line')
ax.plot(x, y1, color='tab:orange', label='
    Curve')

# Create a legend
ax.legend(loc='upper left', bbox_to_anchor=(0,
    1))

#plt.show()
```

You should always add a legend to your plots to show what different plots represent.



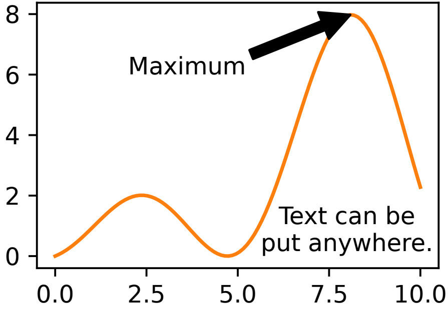
https://matplotlib.org/stable/users/explain/axes/legend_guide.html

Matplotlib elements: Text and annotations

```python
import numpy as np
import matplotlib.pyplot as plt

fig = plt.figure(figsize=(3,2))
ax = fig.gca()

# Calculate x and y(x) and find max y value
x = np.linspace(0, 10, 100)
y = 0.5 * x * np.sin(x) + 0.5 * x
index_y_max = np.argmax(y)
y_value = y[index_y_max]
x_value = x[index_y_max]

ax.plot(x, y, color='tab:orange')

# Add text and annotation with arrow
ax.text(x=8, y=0,
        s='Text can be\nput anywhere.',
        ha='center', va='bottom')
ax.annotate(text='Maximum',
            xy=(x_value, y_value),
            xytext=(2, 6),
            arrowprops=dict(facecolor='k'))

#plt.show()
```

https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.text.html

Text or annotations can highlight interesting parts in a plot.



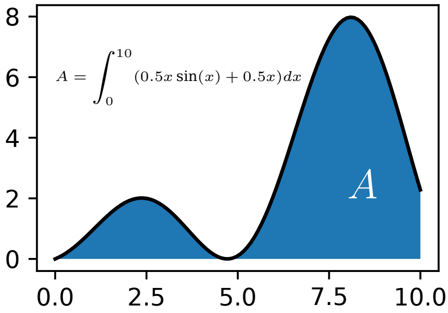
Matplotlib elements: Latex

```python
import numpy as np
import matplotlib
matplotlib.use('pgf') # Use the pgf backend!
import matplotlib.pyplot as plt

fig = plt.figure(figsize=(3,2))
ax = fig.gca()

x = np.linspace(0, 10, 100)
y = 0.5 * x * np.sin(x) + 0.5 * x

ax.fill_between(x, y, np.zeros(x.shape))
ax.plot(x, y, color='black')

# Latex often needs to be written as
# raw strings preceded by an "r"
tex = r'$A=\int_{0}^{10}(0.5x\sin(x)+0.5x)dx$'
ax.text(x=0, y=6,
        s=tex,
        ha='left', va='center',
        fontsize=6)
ax.text(x=8, y=2, s='$A$', color='white',
      fontsize=16)
#plt.show() # Error!!! Can only save figure!
```

https://www.latex-project.org/
https://matplotlib.org/stable/users/explain/text/usetex.html

LaTeX can be used to format text and typeset mathematical formulas.

It also gives access to special characters such as α, β etc and makes it possible to specify exact fonts.



Matplotlib elements: Save and show

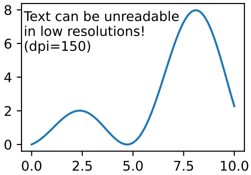
```
import os
import numpy as np
import matplotlib.pyplot as plt

# Create a figure axis and plot random stuff
ax = plt.figure(figsize=(3,3)).gca()
ax.plot(np.linspace(0,100, 101), np.random.random(101))

# Save the plot to file
filepath = os.path.join('.', 'save-and-show-file')
filepath_png = filepath + '.png'
filepath_pdf = filepath + '.pdf'

# Save as rasterized png and vector pdf
plt.savefig(filepath_png)
plt.savefig(filepath_pdf)

#plt.show()
```

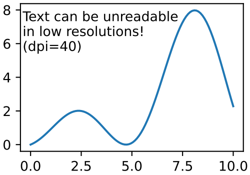
Matplotlib elements: DPI

When saving in rasterized format like jpg or png the dpi setting should be high enough for purpose of figure. Saving in vector format like pdf not affected.

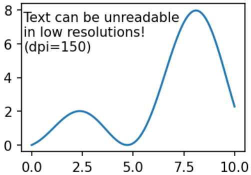
High DPI. Low DPI.

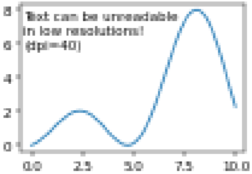
PDF





PNG





Styling

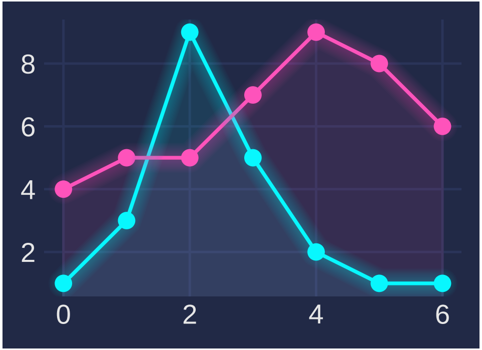
Styling

Styling: Styles

Matplotlib can use several predefined styles.
Additional styles can be downloaded or installed.

```
>> pip install mplcyberpunk
```

```python
""" Give the plot a cool cyberpunk look.
https://github.com/dhaitz/mplcyberpunk
"""
import matplotlib.pyplot as plt
import mplcyberpunk

plt.style.use("cyberpunk")
fig = plt.figure(figsize=(3,2))
ax = fig.gca()

ax.plot([1, 3, 9, 5, 2, 1, 1], marker='o')
ax.plot([4, 5, 5, 7, 9, 8, 6], marker='o')

mplcyberpunk.add_glow_effects()

#plt.show()
```

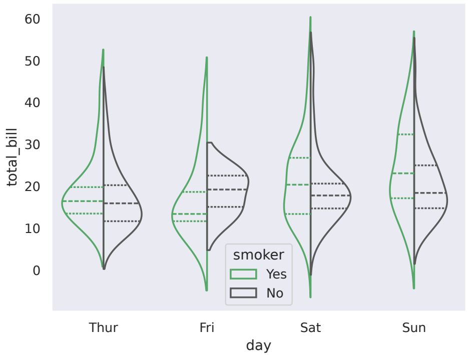


https://matplotlib.org/stable/gallery/style_sheets/style_sheets_reference.html

Styling: Seaborn

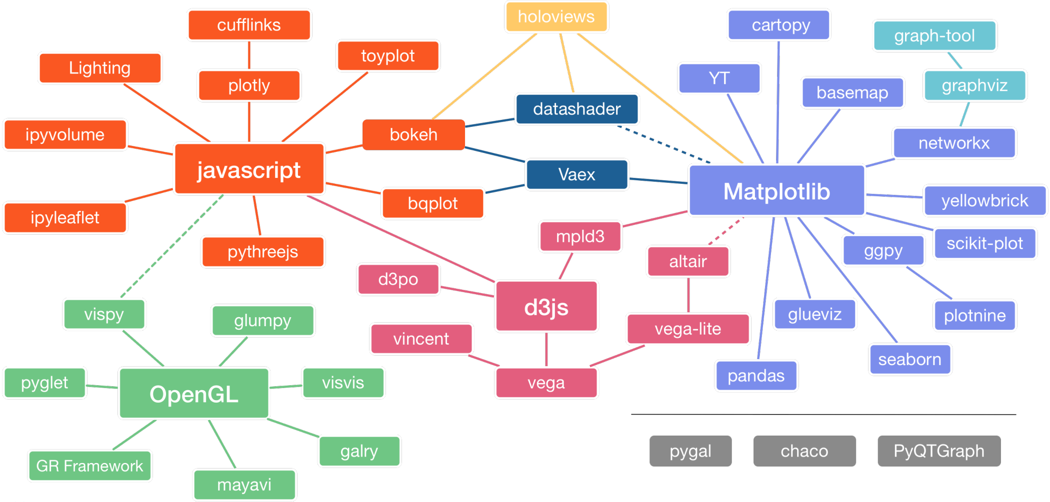
Seaborn is a data visualization package based on matplotlib.

```python
""" Seaborn example.
https://seaborn.pydata.org/examples/grouped_violinplots.html
"""
import seaborn as sns
sns.set_theme(style="dark")

# Load the example tips dataset
tips = sns.load_dataset("tips")

# Draw a nested violinplot and split the
# violins for easier comparison
palette = {"Yes": "g", "No": ".35"}
sns.violinplot(data=tips,
               x="day",
               y="total_bill",
               hue="smoker",
               split=True,
               inner="quart",
               fill=False,
               palette=palette)

#plt.show()
```



https://seaborn.pydata.org/
https://seaborn.pydata.org/examples/grouped_violinplots.html

Styling: Alternative visualization and plotting packages



https://pyviz.org
https://pyviz.org/overviews/index.html

Conclusion

Conclusion

Conclusion: Numpy and Matplotlib

Numpy arrays can store and operate on numerical data in an efficient way.

Using numpy arrays for data is faster and easier than using Python lists.

Numpy and matplotlib work together. Plotting numpy arrays is easy.

Numpy and matplotlib are imported using shorthand names:

```
import numpy as np
import matplotlib.pyplot as plt
```