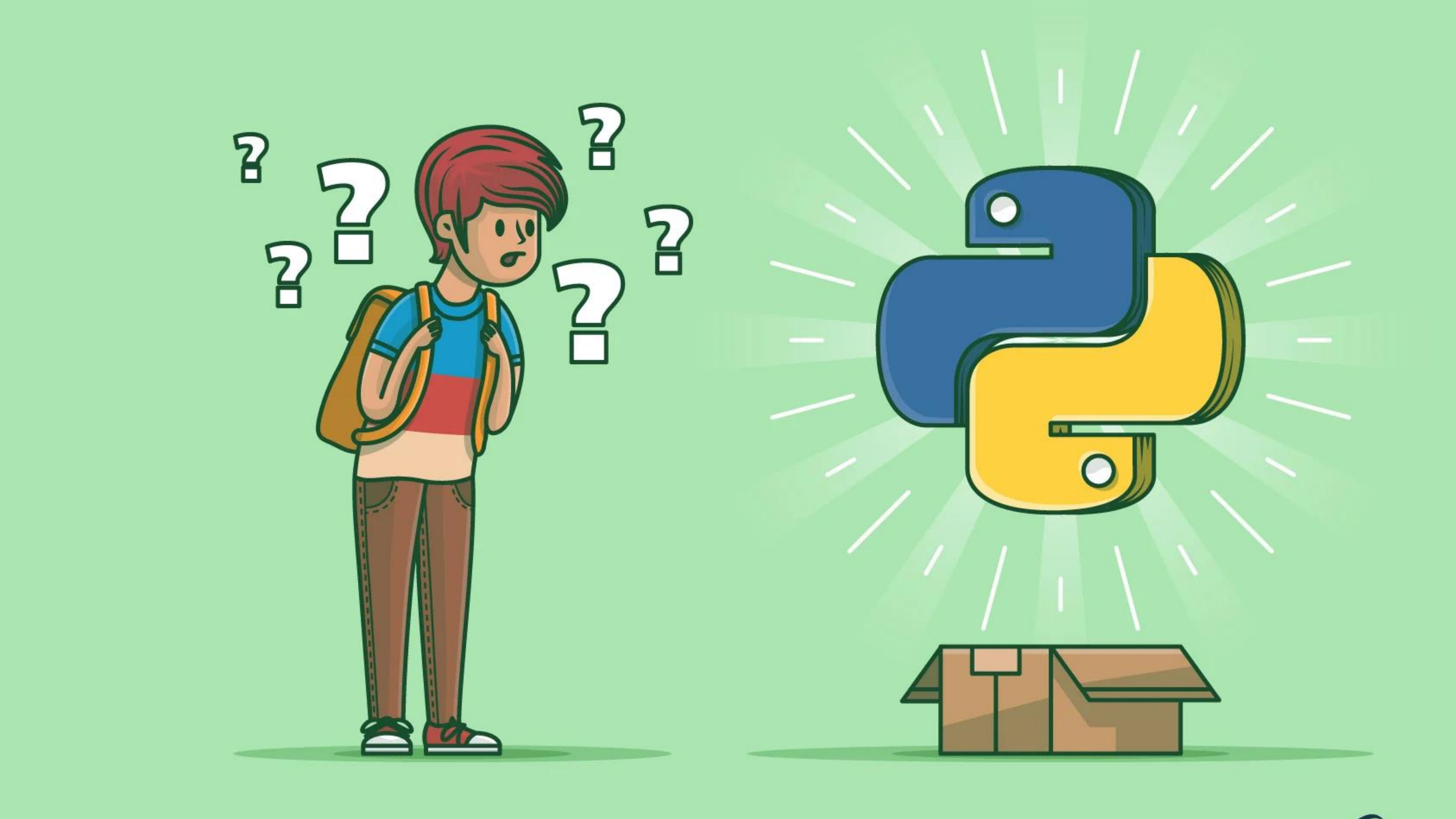
CSE 220:
Signals and Linear Systems

Introduction to Python:

OOP, Numpy and Matplotlib



01

Classes

Object Oriented Programming in Python

Object Orientated Programming

Object-oriented programming is a *programming paradigm* that provides a means of structuring programs so that properties and behaviors are bundled into individual **objects**.

Principles

1 Polymorphism

Allows objects of different classes to be treated as objects of a common class

2 Inheritance

Capability of one class to derive or inherit the properties from another class

3 Encapsulation

Restrictions on accessing variables and methods directly

4 Abstraction

Complex implementation details while exposing only essential information

OOP : Data Classes

We want to track employees of an organization by storing some of their information in a list. One way to do this is to represent each employee as a list:

```
kirk = ["James Kirk", 34, "Captain", 2265]
spock = ["Spock", 35, "Science Officer", 2254]
mccoy = ["Leonard McCoy", "Chief Medical Officer", 2266]
```

As our codebase get bigger and bigger, it gets more and more complicated to maintain each employee as a list.

OOP : Data Classes

An alternate way is to represent each employee as a python object.

```
class Employee:
    def __init__(self, name, age, position, start_year):

        self.name = name
        self.age = age
        self.position = position
        self.start_year = start_year

    def get_details(self):

        return f"Name: {self.name}, Age: {self.age},
Position: {self.position}, Start Year: {self.start_year}"
```

OOP : Data Classes

Attributes created in .__init__() are called *instance attributes*. An instance attribute's value is specific to a particular instance of the class.

On the other hand, *class attributes* are attributes that have the same value for all class instances. You can define a class attribute by assigning a value to a variable name outside of .__init__().

```
class Dog:
    species = "Canis familiaris"

    def __init__(self, name, age):
        self.name = name
        self.age = age
```

OOP : Instance Methods

Instance methods are functions that you define inside a class and can only call on an instance of that class. Just like .__init__(), an instance method always takes self as its first parameter.

```
class Dog:
    species = "Canis familiaris"

    def __init__(self, name, age):
        self.name = name
        self.age = age

    # Instance method
    def description(self):
        return f"{self.name} is {self.age} years old"

    # Another instance method
    def speak(self, sound):
        return f"{self.name} says {sound}"

tommy = Dog("Tommy", 3)
print(tommy.description())
```

OOP : Dunder Methods

Methods like .__init__() and .__str__() are called dunder methods because they begin and end with double underscores

```
class Dog:
    # ...

    def __str__(self):
        return f"{self.name} is {self.age} years old"

tommy = Dog("Tommy", 3)
print(tommy)
```

02

NumPy

Numerical Python Library

NumPy

NumPy (Numerical Python) is a powerful open-source library in Python used for numerical and scientific computing . It provides support for arrays, matrices, and a large collection of mathematical functions to operate on these data structures efficiently.

Why use numpy ?

* Faster
* Easier
* Rich Library Support

Installation :

```
pip install numpy
```

Import :

```
import numpy as np
```

Numpy Array Fundamentals

Numpy arrays behave very similar to python arrays. One way to initialize an array is using a Python sequence, such as a list. For example:

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

Numpy Array Fundamentals

Like the original list, the array is *mutable*. Also like the original list, Python *slice* notation can be used for indexing.

```
a[0] = 10
a
>> array([10,  2,  3,  4,  5,  6])

a[:3]
>> array([10, 2, 3])
```

Slicing

One major difference is that slice indexing of a list copies the elements into a new list, but slicing an array returns a *view*

```
b = a[3:]

b
>> array([4, 5, 6])

b[0] = 40

a
>>array([ 10,  2,  3, 40,  5,  6])
```

NumPy : Higher Dimensional Arrays

Two- and higher-dimensional arrays can be initialized from nested Python sequences.

```
a = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
a
>> array([[ 1,  2,  3,  4],
          [ 5,  6,  7,  8],
          [ 9, 10, 11, 12]])
```

NumPy

NumPy : Array Attributes

ndim, shape, size, *and* dtype

- The number of dimensions of an array is contained in the *ndim* attribute.
- The *shape* of an array is a tuple of non-negative integers that specify the number of elements along each dimension.
- The fixed, total number of elements in array is contained in the *size* attribute.
- Arrays are typically “homogeneous”, meaning that they contain elements of only one “data type”. The data type is recorded in the *dtype* attribute

NumPy

NumPy : Array Initialization

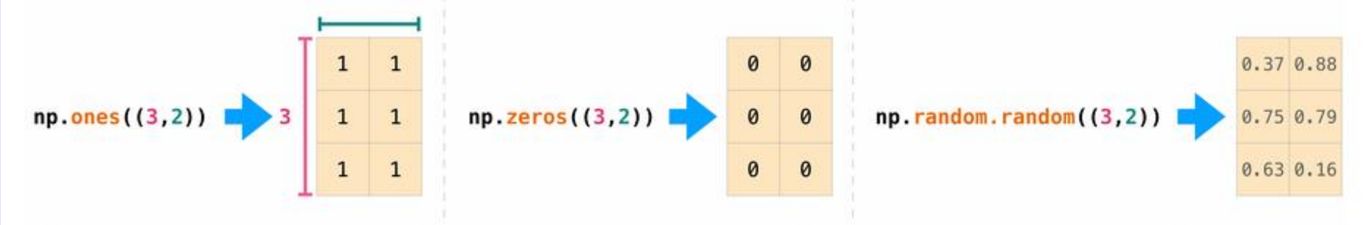
- *np.zeros()* : create an array filled with 0's
- *np.ones()* : create an array filled with 1's
- *np.empty()* : creates an array whose initial content is random and depends on the state of the memory
- *np.arrange()* : create an array with a range of elements
- *np.linspace()* : create an array with values that are spaced linearly in a specified interval

We can also specify the datatype using the optional *dtype=np.int64* parameter

NumPy

NumPy : Random Arrays

We can also use *ones()*, *zeros()*, and *random()* to create a 2D array if we give them a tuple describing the dimensions of the matrix:



NumPy : Sorting and Concat

```
arr = np.array([2, 1, 5, 3, 7, 4, 6, 8])

np.sort(arr) # returns a sorted copy of the array
>> array([1, 2, 3, 4, 5, 6, 7, 8])
```

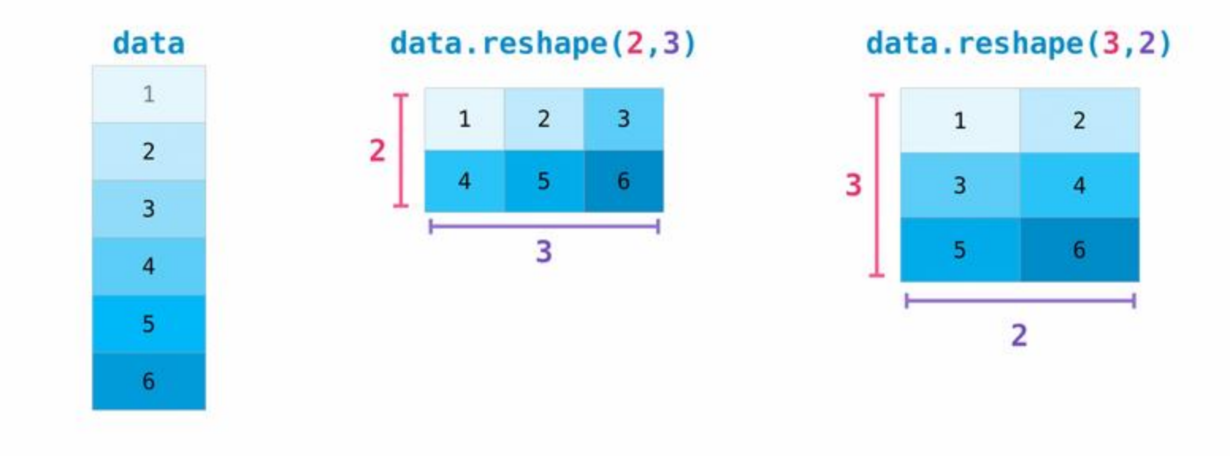
NumPy

NumPy : Sorting and Concat

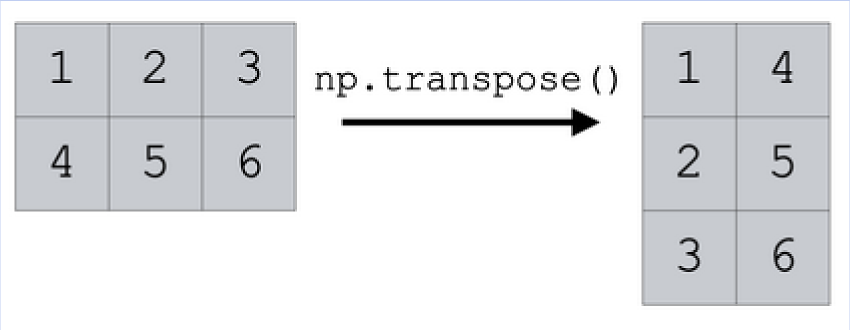
```
a = np.array([1, 2, 3, 4])
b = np.array([5, 6, 7, 8])
np.concatenate((a, b))
>> array([1, 2, 3, 4, 5, 6, 7, 8])
```

NumPy

NumPy : Reshape



NumPy : Transpose



We can also use *arr.T* instead of *arr.transpose()*

NumPy

NumPy : Selection

```
a = np.array([[1 , 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
a[a < 5]
>> array([1, 2, 3, 4])

divisible_by_2 = a[a%2==0]
>> array([ 2,  4,  6,  8, 10, 12])

c = a[(a > 2) & (a < 11)]
>> array([ 3  4  5  6  7  8  9 10])
```

NumPy

NumPy : Stacking

```
a1 = np.array([[1, 1],
               [2, 2]])

a2 = np.array([[3, 3],
               [4, 4]])

np.vstack((a1, a2))
>> array([[1, 1],
          [2, 2],
          [3, 3],
          [4, 4]])

np.hstack((a1, a2))
>> array([[1, 1, 3, 3],
          [2, 2, 4, 4]])
```

NumPy

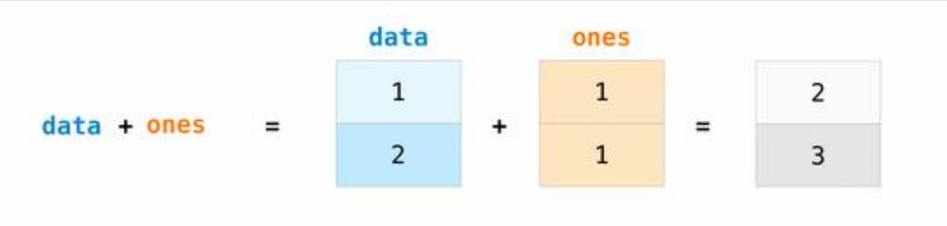
NumPy : Splitting

```
x = np.arange(1, 25).reshape(2, 12)
np.hsplit(x, 3)
>>[array([[ 1,  2,  3,  4],
         [13, 14, 15, 16]]), array([[ 5,  6,  7,  8],
         [17, 18, 19, 20]]), array([[ 9, 10, 11, 12],
         [21, 22, 23, 24]])]
```

NumPy

NumPy : Array Operations

NumPy supports arithmetic operations between two arrays of the same shape. The operations are carried out element wise.

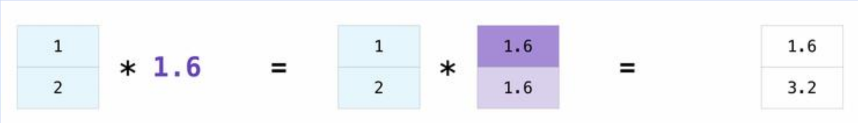


NumPy

NumPy : Broadcasting

Sometimes we want to carry out operations between a *vector* and a *scalar* or between arrays of *different sizes*. For example, our arrays might contain information about distances travelled in miles, which we want to convert into kilometers.

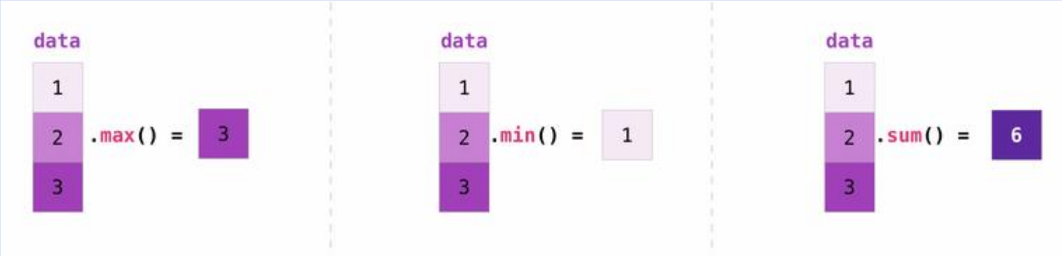
```
data = np.array([1.0, 2.0])

data * 1.6
>> array([1.6, 3.2])
```



NumPy

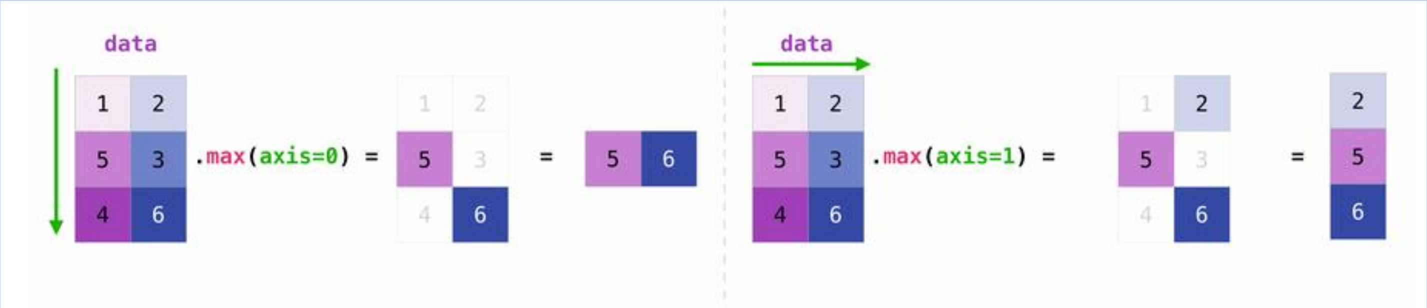
NumPy : Aggregation Operations

NumPy also performs aggregation functions. In addition to *min*, *max*, and *sum*, you can easily run *mean* to get the average, *prod* to get the result of multiplying the elements together, *std* to get the standard deviation.



NumPy : Aggregation Operations

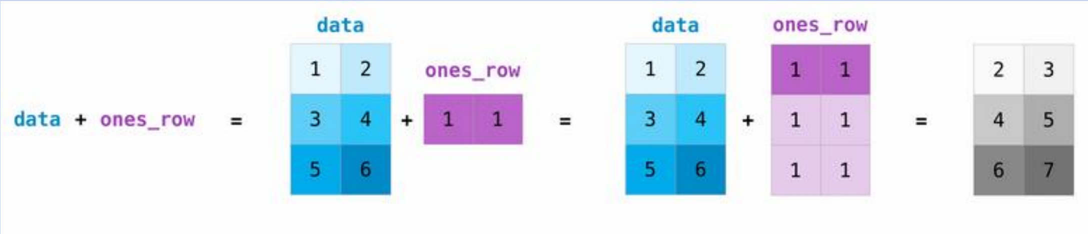
Aggregation can also be done across various axes. For example: we can apply the max aggregation to a 2D array in different axes to get different results.



NumPy : Broadcasting

You can do these arithmetic operations on matrices of different sizes, but only if one matrix has only *one column* or *one row*. In this case, NumPy will use its broadcast rules for the operation.

```
data = np.array([[1, 2], [3, 4], [5, 6]])
ones_row = np.array([[1, 1]])
```



NumPy : Unique

```
a = np.array([11, 11, 12, 13, 14, 15, 16, 17, 12, 13, 11, 14, 18, 19, 20])
unique_values, occurrence_count = np.unique(a, return_counts=True)

unique_values
>> [11 12 13 14 15 16 17 18 19 20]

occurrence_count
>> [3 2 2 2 1 1 1 1 1 1]
```

NumPy

NumPy : Mathematical Formulas

The ease of implementing mathematical formulas that work on arrays is one of the things that make NumPy so widely used in the scientific Python community.

$$MeanSquareError = \frac{1}{n}\sum_{i=1}^{n}\left(y_{Pred} - y_i\right)^2$$

NumPy

NumPy : Mathematical Formulas

The ease of implementing mathematical formulas that work on arrays is one of the things that make NumPy so widely used in the scientific Python community.

$$MeanSquareError = \frac{1}{n}\sum_{i=1}^{n}(y_{Pred} - y_i)^2$$

```
error = (1/n) * np.sum(np.square(predictions - label))
```

NumPy

NumPy : Mathematical Formulas

There are a lot of mathematical operations available in numpy. A few of them are as follows :

- *np.log()* : Computes the natural logarithm of each element.
- *np.sqrt()* : Computes the square root of each element in an array.
- *np.power()* : Raises each element of an array to a specified power.
- *np.sin()* : Computes the sine of each element (in radians).

NumPy : Mathematical Formulas

$$-\frac{1}{m}\sum_{i=1}^{m} y_{actual} \cdot \log(y_{pred})$$

Cross-entropy loss

NumPy : Mathematical Formulas

$$s(x_i) = \frac{e^{x_i}}{\sum_{j=i}^{n} e^{x_j}}$$

Softmax Function

NumPy

03

Matplotlib

Mathematical Visualization

Matplotlib

Matplotlib is a comprehensive library for creating *static*, *animated*, and *interactive* visualizations. It is a library for making 2D plots in Python

```
pip install matplotlib
```

matplotlib

Matplotlib : Import

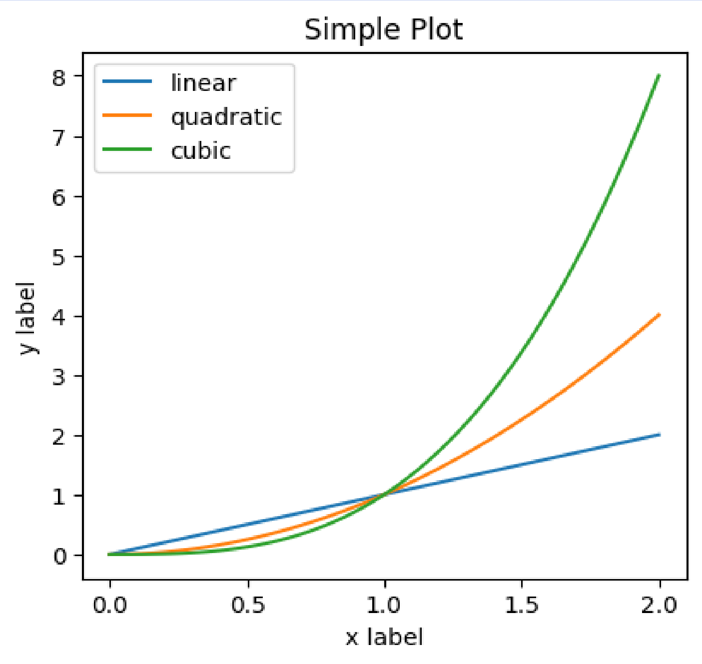
It is a general convention that we alias the import of *matplotlib.pyplot* as *plt*.

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

matplotlib

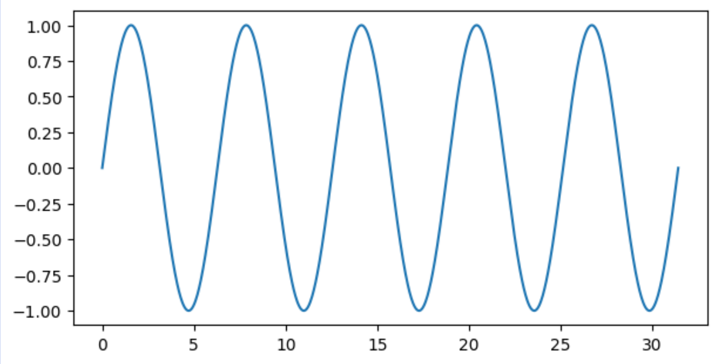
Matplotlib : A Simple Plot

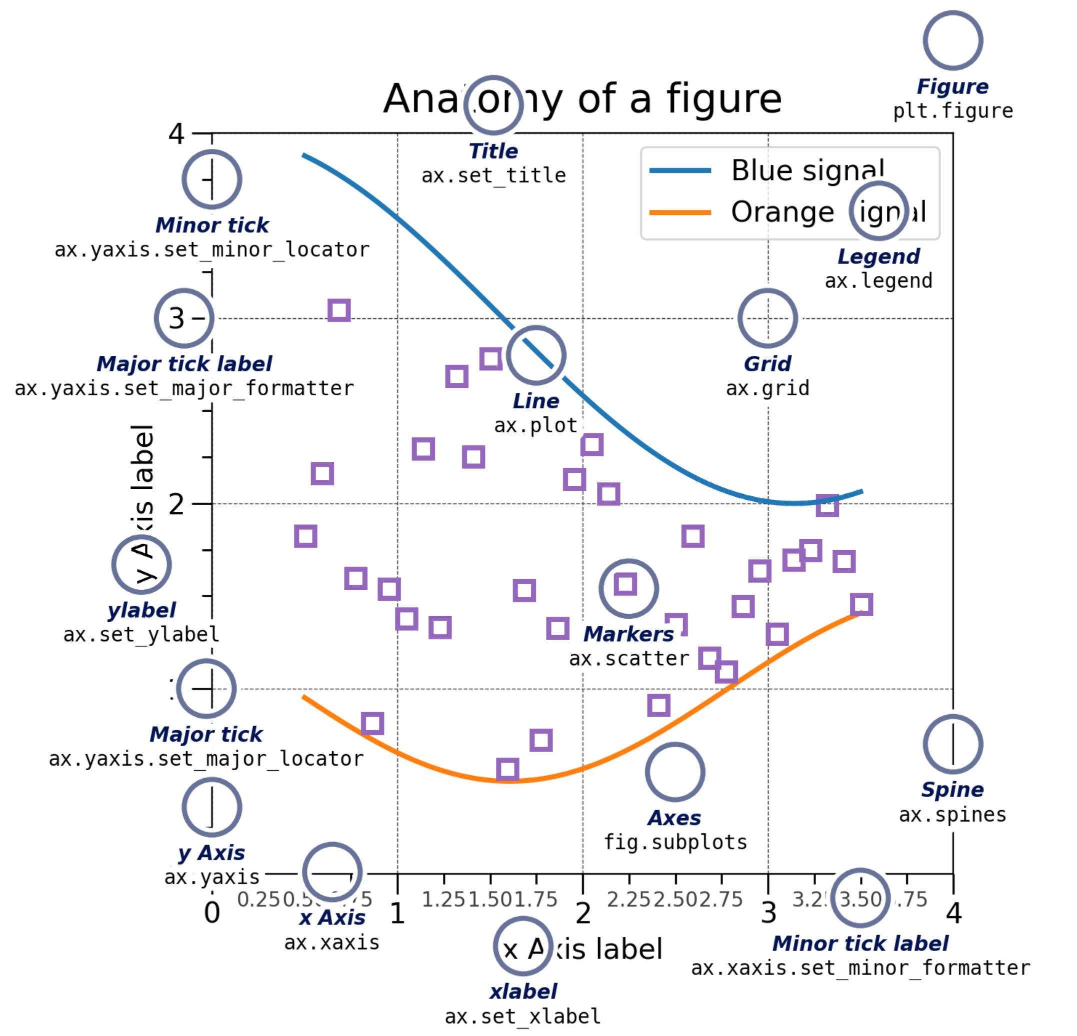
```
x = np.linspace(0, 2, 100)  # Sample data.

plt.figure(figsize=(4, 4), layout='constrained')
plt.plot(x, x, label='linear')
plt.plot(x, x**2, label='quadratic')  # etc.
plt.plot(x, x**3, label='cubic')
plt.xlabel('x label')
plt.ylabel('y label')
plt.title("Simple Plot")
plt.legend()
```



Matplotlib : Sine Wave

```
X = np.linspace(0, 10 * np.pi, 1000)
Y = np.sin(X)

plt.figure(figsize=(6, 3), layout='constrained')
plt.plot(X, Y)
plt.show()
```





Anatomy of a figure
Figure
plt.figure
Title
ax.set_title
Blue signal
Orange signal
Legend
ax.legend
Minor tick
ax.yaxis.set_minor_locator
Major tick label
ax.yaxis.set_major_formatter
Line
ax.plot
Grid
ax.grid
y Axis label
ylabel
ax.set_ylabel
Markers
ax.scatter
Major tick
ax.yaxis.set_major_locator
Spine
ax.spines
Axes
fig.subplots
y Axis
ax.yaxis
x Axis
ax.xaxis
x Axis label
xlabel
ax.set_xlabel
Minor tick label
ax.xaxis.set_minor_formatter

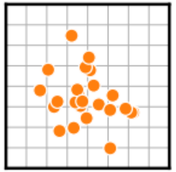
matplotlib

Matplotlib : Types of plots

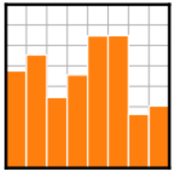
Matplotlib offers several kind of plots.

Here we demonstrate a *scatterplot*, *bar plot* and *image*.

```
X = np.random.uniform(0, 1, 100)
Y = np.random.uniform(0, 1, 100)
ax.scatter(X, Y)
```



```
X = np.arange(10)
Y = np.random.uniform(1, 10, 10)
ax.bar(X, Y)
```



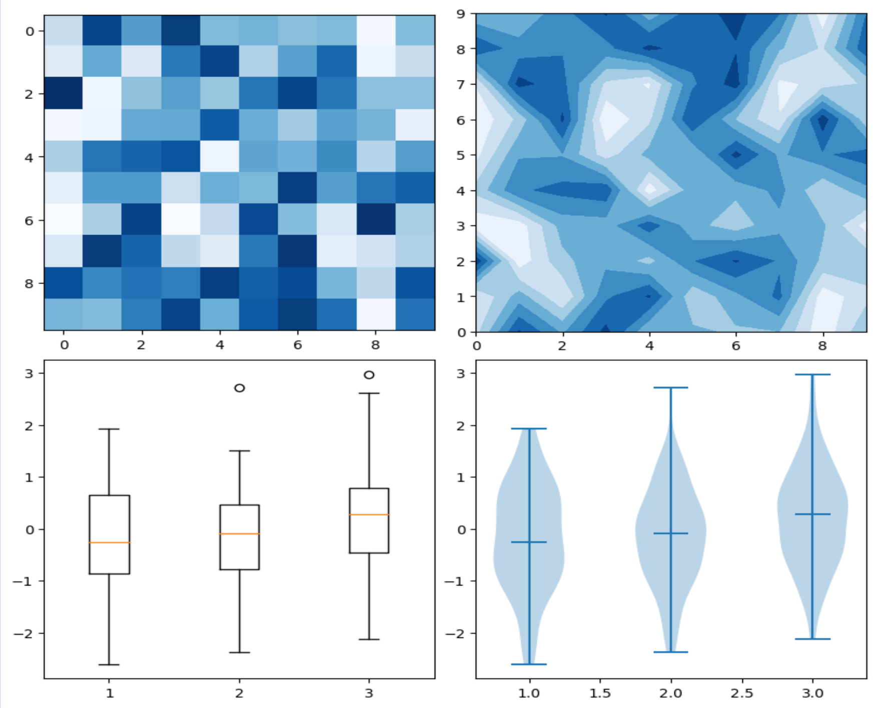
```
Z = np.random.uniform(0, 1, (8, 8))

ax.imshow(Z)
```

matplotlib

Matplotlib : Subplots

```
Z = np.random.uniform(0, 1, (10, 10))
Y = np.random.normal(0, 1, (100, 3))

fig, axs = plt.subplots(2, 2, figsize=(8, 8),
layout='constrained')
axs[0, 0].imshow(Z, cmap='Blues')
axs[0, 1].contourf(Z, cmap='Blues')
axs[1, 0].boxplot(Y )
axs[1, 1].violinplot(Y, showmedians=True)

fig.show()
```

Matplotlib : Subplots



matplotlib

References

- RealPython
- Numpy Documentation
- Matplotlib Documentation

Thank You