# Introduction to Python

Lecturer: Bo Li  TA: Chen Gao

2026-01-21

# A Brief Introduction to Python

**Python** is a widely used programming language with a rich ecosystem of libraries.

- In scientific computing, it plays a role similar to MATLAB or Octave,
- with strong support for numerical computation, data analysis, and visualization.

# A Brief Introduction to Python

**Python** is a widely used programming language with a rich ecosystem of libraries.

- In scientific computing, it plays a role similar to MATLAB or Octave,
- with strong support for numerical computation, data analysis, and visualization.

Many modern machine learning and deep learning frameworks—such as PyTorch and TensorFlow—are built around Python.

# A Brief Introduction to Python

**Python** is a widely used programming language with a rich ecosystem of libraries.

- In scientific computing, it plays a role similar to MATLAB or Octave,
- with strong support for numerical computation, data analysis, and visualization.

Many modern machine learning and deep learning frameworks—such as PyTorch and TensorFlow—are built around Python.

In this tutorial, we'll walk through the essentials of Python programming and help you get set up with a clean, reproducible workflow. No prior experience is required.

# Environment Setup

We will use `uv` (Astral) to manage Python environments and dependencies.

It is fast, lightweight, and makes projects easy to reproduce across machines.

# Install uv

Follow the official installation guide:

MacOS and Linux:

```
curl -LsSf https://astral.sh/uv/install.sh | sh
```

Windows:

```
powershell -ExecutionPolicy ByPass -c
\"irm https://astral.sh/uv/install.ps1 | iex\"
```

# Create a new project

Navigate to your working directory.

Initialize a new project.

```
uv init
```

This will create a minimal project structure along with a `pyproject.toml`.

Add packages to your project (and lock them):

```
uv add <package_name>
```

For example:

```
uv add numpy pandas matplotlib
```

# Run code

**Using Python**: Run a Python script within the project environment:

```
uv run example.py
```

**Using Jupyter Notebook**: If you want to use Jupyter notebooks, install Jupyter inside the project environment:

```
uv add jupyter
```

Then start Jupyter from within the project directory:

```
uv run jupyter lab
```

In most setups, you should see a kernel associated with your project's `.venv`.

# Sync dependencies across machines

To reproduce the exact same environment on another machine:

1. Copy **both** `pyproject.toml` and `uv.lock` to the new location.
2. Run:

```
uv sync
```

> **Note**
>
> `uv sync` is case-sensitive so make sure to use lowercase.

# IDE

Common choices:

- **PyCharm**
- **Visual Studio Code**
  - **Cursor** (an AI-enhanced VS Code fork)
- **Zed** (AI-native editor)
- **Vim / Neovim**

If you're new to Python, VS Code or PyCharm are the easiest places to start.

# Basic Operations

It's easy to assign values to variables and print them.

```
x = 10
y = 4
print(x, y)
```

```
10 4
```

And it's a good practice to add comments to your code.

```
# Comments start with hash #
```

# Basic Calculation

```python
print(f"Addition: x+y = {x+y}")
print(f"Power: x**y = {x**y}")
print(f"Integer Division: x//y = {x//y}")
print(f"Division: x/y = {x/y}")
concat = str(x) + "+" + str(y)
print(f"Concatenation: {concat}")
```

```
Addition: x+y = 14
Power: x**y = 10000
Integer Division: x//y = 2
Division: x/y = 2.5
Concatenation: 10+4
```

We can also use loops to perform repetitive calculations.

```
1 for i in range(5):
2     print(f"Loop: i = {i}")
```

```
Loop: i = 0
Loop: i = 1
Loop: i = 2
Loop: i = 3
Loop: i = 4
```

> **Note**
>
> We use a format called f-strings to format the output. For example, `f"Loop: i = {i}"` will replace `{i}` with the value of `i`.

# Containers

Python includes several built-in container types: lists, dictionaries, sets, and tuples. We will take a look at them.

# Lists

A **list** is the Python equivalent of an **array**, but is resizeable and can contain elements of different types.

# Lists

A **list** is the Python equivalent of an **array**, but is resizeable and can contain elements of different types.

```
1 xs = [3, 1, 2]
2 print(xs, xs[2])
```

```
[3, 1, 2] 2
```

# Lists

A **list** is the Python equivalent of an **array**, but is resizeable and can contain elements of different types.

```
1 xs = [3, 1, 2]
2 print(xs, xs[2])
```

```
[3, 1, 2] 2
```

Change the contents of a list (lists can contain elements of different types):

```
1 xs[2] = "foo"
2 print(xs)
```

```
[3, 1, 'foo']
```

Add a new element to the end of a list:

```
1 xs.append("bar")
2 print(xs)
```

```
[3, 1, 'foo', 'bar']
```

Add a new element to the end of a list:

```
xs.append("bar")
print(xs)
```

```
[3, 1, 'foo', 'bar']
```

Remove and return the last element from the list:

```
x = xs.pop()
print(x, xs)
```

```
bar [3, 1, 'foo']
```

# Slicing

In addition to accessing list elements one at a time, Python provides concise syntax to access sublists. This is known as slicing.

```
1 nums = list(range(5))
2 print(nums)
```

```
[0, 1, 2, 3, 4]
```

# Slicing

In addition to accessing list elements one at a time, Python provides concise syntax to access sublists. This is known as slicing.

```
1 nums = list(range(5))
2 print(nums)
```

```
[0, 1, 2, 3, 4]
```

```
1 print(nums[2:4])
2 print(nums[2:])
```

```
[2, 3]
[2, 3, 4]
```

# Loops

You can loop over the elements of a list:

```
1 animals = ["cat", "dog", "monkey"]
2 for animal in animals:
3     print(animal)
```

```
cat
dog
monkey
```

# Enumerate

If you want access to the index of each element within the body of a loop, use the built-in `enumerate` function:

```
1 animals = ["cat", "dog", "monkey"]
2 for idx, animal in enumerate(animals):
3     print(f"#{idx + 1}: {animal}")
```

```
#1: cat
#2: dog
#3: monkey
```

# List comprehensions

Frequently we want to transform one type of data into another. As a simple example, consider computing square numbers.

First, using a loop:

```
nums = [0, 1, 2, 3, 4]
squares = []
for x in nums:
    squares.append(x**2)
print(squares)
```

```
[0, 1, 4, 9, 16]
```

# List comprehensions

Now, using a list comprehension:

```
1 nums = [0, 1, 2, 3, 4]
2 print([x**2 for x in nums])
```

```
[0, 1, 4, 9, 16]
```

# List comprehensions

Now, using a list comprehension:

```
1 nums = [0, 1, 2, 3, 4]
2 print([x**2 for x in nums])
```

```
[0, 1, 4, 9, 16]
```

List comprehensions can also contain conditions:

```
1 nums = [0, 1, 2, 3, 4]
2 print([x**2 for x in nums if x % 2 == 0])
```

```
[0, 4, 16]
```

# Dictionaries

A dictionary stores (key, value) pairs. Create and use a dictionary:

```
1 d = {"cat": "cute", "dog": "furry"}
2 print(d["cat"])
3 print("cat" in d)
```

```
cute
True
```

`print(d['cat'])` gets an entry from the dictionary ("cute").

`'cat' in d` checks if the key exists.

# Dictionaries

Add a new key-value pair to the dictionary:

```
1 d["fish"] = "wet"
2 print(d["fish"])
```

```
wet
```

# Dictionaries

Add a new key-value pair to the dictionary:

```
d["fish"] = "wet"
print(d["fish"])
```

```
wet
```

Deleting keys and using get with a default:

```
del d["fish"]
print(d.get("fish", "N/A"))
```

```
N/A
```

# Dictionaries

**Loops**: Iterate over the keys in a dictionary:

```
d = {"person": 2, "cat": 4, "spider": 8}
for animal in d:
    legs = d[animal]
    print(f"A {animal} has {legs} legs")
```

```
A person has 2 legs
A cat has 4 legs
A spider has 8 legs
```

# Dictionaries

**items**: If you want access to keys and their corresponding values, use the `items` method:

```
d = {"person": 2, "cat": 4, "spider": 8}
for animal, legs in d.items():
    print(f"A {animal} has {legs} legs")
```

```
A person has 2 legs
A cat has 4 legs
A spider has 8 legs
```

# Dictionaries

**Dictionary comprehensions**: You can construct dictionaries easily with comprehensions:

```
1 nums = [0, 1, 2, 3, 4]
2 even_num_to_square = {x: x**2 for x in nums if x % 2 == 0}
3 print(even_num_to_square)
```

```
{0: 0, 2: 4, 4: 16}
```

# Sets

A set is an unordered collection of distinct elements.

Create a set and see how it behaves:

```
1 animals = {"cat", "dog"}
2 print("cat" in animals)
3 print("fish" in animals)
```

```
True
False
```

# Sets

Add a new element to the set:

```
1 animals.add("fish")
2 print("fish" in animals)
3 print(len(animals))
```

```
True
3
```

# Sets

Sets automatically avoid duplicates.

```
1 animals.add("cat")
2 print(len(animals))
3 animals.remove("cat")
4 print(len(animals))
```

```
3
2
```

# Sets

**Loops**: Iterate over a set:

```
1 animals = {"cat", "dog", "fish"}
2 for idx, animal in enumerate(animals):
3     print(f"#{idx + 1}: {animal}")
```

```
#1: fish
#2: cat
#3: dog
```

# Sets

**Set comprehensions**: Construct sets using comprehensions:

```
1 from math import sqrt
2
3 nums = {int(sqrt(x)) for x in range(30)}
4 print(nums)
```

```
{0, 1, 2, 3, 4, 5}
```

# Tuples

A tuple is an (*immutable*) ordered list of values. Tuples can be used as keys in dictionaries and as elements of sets, while lists cannot.

```
1 d = {(x, x + 1): x for x in range(10)}
2 t = (5, 6)
3 print(f"Type of t: {type(t)}")
4 print(f"Value of d[t]: {d[t]}")
5 print(f"Value of d[(1, 2)]: {d[(1, 2)]}")
```

```
Type of t: <class 'tuple'>
Value of d[t]: 5
Value of d[(1, 2)]: 1
```

# Functions

Python functions are defined using the `def` keyword.

For example, define the sign function as:

$$\text{sign}(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x = 0 \\ -1 & \text{if } x < 0 \end{cases}$$

# Functions

```
def sign(x):
    if x > 0:
        return 1
    elif x < 0:
        return -1
    else:
        return 0
```

Print the result for sample values:

```
for x in [-1, 0, 1]:
    print(f"sign({x}) = {sign(x)}", end=", ")
```

```
sign(-1) = -1, sign(0) = 0, sign(1) = 1,
```

# Arguments

We will often define functions to take **optional** keyword arguments, like this:

Define a function that prints a greeting, with an optional keyword argument `loud`:

```
1 def hello(name, loud=False):
2     if loud:
3         print(f"HELLO, {name.upper()}!")
4     else:
5         print(f"Hello, {name}")
```

# Arguments

Print the result for sample values:

```
1 hello("Bob")
2 hello("Fred", loud=True)
```

```
Hello, Bob
HELLO, FRED!
```

You will see various functions in real applications.

# Introduction to NumPy

## Install Packages

You can install NumPy in your project using `uv`:

```
uv add numpy
```

## Import Package Modules

- After the packages have been successfully installed, we can call particular functions from the package in our codes.
- Before we use these functions, we should first **import** them at the beginning of our codes.
- There are several ways to import package/modules.

# Import Package Modules

Import `numpy` under an alias:

```
1 import numpy as np
```

Import specific submodules/functions:

```
1 from numpy import linalg as la, dot as matrix_multiply
```

> **Caution**
>
> Be careful when you use alias. It may result in namespace collisions.

# NumPy Basics

- NumPy is an optimized library for vector/matrix computation.
- It makes use of C/C++ subroutines and memory-efficient data structures.
  - Therefore lots of computation can be efficiently done with `numpy`.
- The main data type is `np.ndarray`, which we will use to represent matrix/vector for computations.

# NumPy Basics

Construct `numpy` arrays:

```
x = np.array([1, 2, 3])
y = np.array([[3, 4, 5]])
print(x, y)
```

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

```
z = np.array([[6, 7], [8, 9]])
print(z)
```

```
[[6 7]
 [8 9]]
```

# NumPy Basics

Print the shape of arrays:

```
1 print(x.shape)
2 print(y.shape)
3 print(z.shape)
```

```
(3,)
(1, 3)
(2, 2)
```

# Indexing

In general, NumPy has similar indexing as Python.

There are also indexing methods specific to NumPy.

Mastery of indexing helps write efficient NumPy codes.

> **Note**
>
> For example, avoid explicit for-loops over indices because for-loops will dramatically slow down the code.

# Indexing

Create a random (3,4) matrix:

```
1 p = np.random.random((3, 4))
```

Select everything in p:

```
1 p[:]
```

```
array([[0.74542083, 0.81666596, 0.72452768, 0.82654059],
       [0.5789152 , 0.00729976, 0.94496863, 0.91998259],
       [0.968678  , 0.54588208, 0.63734426, 0.120711  ]])
```

# Indexing

Select the 0th and 2nd rows:

```
p[np.array([0, 2]), :]
```

```
array([[0.74542083, 0.81666596, 0.72452768, 0.82654059],
       [0.968678  , 0.54588208, 0.63734426, 0.120711  ]])
```

Select 1st row as 1-D vector and 1st through 2nd elements:

```
p[1, 1:3]
```

```
array([0.00729976, 0.94496863])
```

# Indexing

Boolean indexing:

```
1 p[p > 0.5]
```

```
array([0.74542083, 0.81666596, 0.72452768, 0.82654059, 0.5789152 ,
       0.94496863, 0.91998259, 0.968678  , 0.54588208, 0.63734426])
```

Create 3-D vector of shape(3,4,1) from p:

```
1 p[:, :, np.newaxis]
```

```
array([[[0.74542083],
        [0.81666596],
        [0.72452768],
        [0.82654059]],

       [[0.5789152 ],
        [0.00729976],
        [0.94496863],
```

# Array Math Operations

We can apply matrix operations to these objects just like linear algebra.

There are many advanced functions which are very useful in `SciPy` and `np.linalg`.

We will just show some of the most common operations below.

# Array Math Operations

Create two arrays for operations:

```
x = np.array([[1, 2], [3, 4]], dtype=np.float64)
y = np.array([[5, 6], [7, 8]], dtype=np.float64)
```

Elementwise sum:

```
print(x + y)
print(np.add(x, y))
```

```
[[ 6.  8.]
 [10. 12.]]
[[ 6.  8.]
 [10. 12.]]
```

# Array Math Operations

Elementwise difference:

```
1 print(x - y, np.allclose(x - y, np.subtract(x, y)))
```

```
[[-4. -4.]
 [-4. -4.]] True
```

Elementwise product:

```
1 print(x * y, np.allclose(x * y, np.multiply(x, y)))
```

```
[[ 5. 12.]
 [21. 32.]] True
```

# Array Math Operations

Dot product & matrix multiplication:

```
1 print(np.dot(x, y), np.allclose(np.dot(x, y), x @ y))
```

```
[[19. 22.]
 [43. 50.]] True
```

> **Note**
>
> Unlike MATLAB, $*$ is elementwise multiplication, not matrix multiplication. We instead use the dot function to compute inner products of vectors, to multiply a vector by a matrix, and to multiply matrices.

# Array Math Operations

Elementwise division:

```
1 print(x / y, np.allclose(x / y, np.divide(x, y)))
```

```
[[0.2        0.33333333]
 [0.42857143 0.5       ]] True
```

Elementwise square root:

```
1 print(np.sqrt(x))
```

```
[[1.         1.41421356]
 [1.73205081 2.        ]]
```

# Array Math Operations

Matrix Operations (Norm):

```
print(np.linalg.norm(y))
```

```
13.19090595827292
```

Transpose:

```
print(z.T)
```

```
[[6 8]
 [7 9]]
```

# Array Math Operations

**Sum Operation**: To compute sum of all elements, each column, or each row:

```
x = np.array([[1, 2], [3, 4]])

print(np.sum(x))  # sum of all elements
print(np.sum(x, axis=0))  # sum of each column
print(np.sum(x, axis=1))  # sum of each row
```

```
10
[4 6]
[3 7]
```

# Broadcast

Broadcasting is a powerful mechanism that allows numpy to work with arrays of different shapes when performing arithmetic operations.

Frequently we have a smaller array and a larger array, and we want to use the smaller array multiple times to perform some operation on the larger array.

Suppose that we want to add a constant vector to each row of a matrix. We could do it like this:

# Broadcast

**Adding a vector to each row with an explicit loop:**

```
x = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = np.empty_like(x)
for i in range(4):
    y[i, :] = x[i, :] + v
print(y)
```

```
[[ 2  2  4]
 [ 5  5  7]
 [ 8  8 10]
 [11 11 13]]
```

# Broadcast

This works; however, when the matrix `x` is very large, computing an explicit loop in Python could be slow.

Note that adding the vector `v` to each row of the matrix `x` is equivalent to forming a matrix `vv` by stacking multiple copies of `v` vertically.

Then performing elementwise summation of `x` and `vv`.

# Broadcast

**Stack multiple copies of a vector and add:**

```
x = np.array(
    [
        [1, 2, 3],
        [4, 5, 6],
        [7, 8, 9],
        [10, 11, 12],
    ]
)
v = np.array([1, 0, 1])
```

# Broadcast

Stack 4 copies of `v` on top of each other:

```
1 vv = np.tile(v, (4, 1))
2 print(f"vv={vv}")
```

```
vv=[[1 0 1]
 [1 0 1]
 [1 0 1]
 [1 0 1]]
```

# Broadcast

Add `x` and `vv` elementwise:

```
1 y = x + vv
2 print(f"y={y}")
```

```
y=[[ 2  2  4]
 [ 5  5  7]
 [ 8  8 10]
 [11 11 13]]
```

# Broadcast

Numpy broadcasting allows us to perform this computation without actually creating multiple copies of `v`.

**Using broadcasting to add a vector to each row:**

```
x = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = x + v  # Add v to each row of x using broadcasting
print(y)
```

```
[[ 2  2  4]
 [ 5  5  7]
 [ 8  8 10]
 [11 11 13]]
```

# Broadcast

The line `y = x + v` works even though `x` has shape (4, 3) and `v` has shape (3,) due to broadcasting;

This line works as if `v` actually had shape (4, 3), where each row was a copy of `v`, and the sum was performed elementwise.

# Broadcast Rules

- If the arrays do not have the same rank, prepend the shape of the lower rank array with 1s until both shapes have the same length.
- The two arrays are said to be compatible in a dimension if they have the same size in the dimension, or if one of the arrays has size 1 in that dimension.
- The arrays can be broadcast together if they are compatible in all dimensions.
- After broadcasting, each array behaves as if it had shape equal to the elementwise maximum of shapes of the two input arrays.
- In any dimension where one array had size 1 and the other array had size greater than 1, the first array behaves as if it were copied along that dimension

# Broadcast

More applications of broadcasting:

**Compute outer product of vectors:**

```
v = np.array([1, 2, 3])  # v has shape (3,)
w = np.array([4, 5])  # w has shape (2,)
print(np.reshape(v, (3, 1)) * w)
```

```
[[ 4  5]
 [ 8 10]
 [12 15]]
```

Here `v` has shape (3,) and `w` has shape (2,), so the outer product has shape (3, 2).

# Broadcast

To compute an outer product, we first reshape `v` to be a column vector of shape (3, 1); we can then broadcast it against `w` to yield an output of shape (3, 2), which is the outer product of `v` and `w`.

# Broadcast

**Add a vector to each row of a matrix:**

```
1  x = np.array([[1, 2, 3], [4, 5, 6]])
2
3  print(x + v)
```

```
[[2 4 6]
 [5 7 9]]
```

Here x has shape (2, 3) and v has shape (3,), so they broadcast to (2, 3)

# Broadcast

**Add a vector to each column of a matrix:** Here `x` has shape (2, 3) and `w` has shape (2,).

```
1 print((x.T + w).T)
2
3 print(x + np.reshape(w, (2, 1)))
```

```
[[ 5  6  7]
 [ 9 10 11]]
[[ 5  6  7]
 [ 9 10 11]]
```

# Broadcast

- If we transpose `x` then it has shape (3, 2) and can be broadcast against `w` to yield a result of shape (3, 2);
- transposing this result yields the final result of shape (2, 3) which is the matrix `x` with the vector `w` added to each column.
- Another solution is to reshape `w` to be a column vector of shape (2, 1); we can then broadcast it directly against `x` to produce the same output.

# Broadcast

**Multiply a matrix by a constant:** Here x has shape (2, 3). Numpy treats scalars as arrays of shape (), these can be broadcast together to shape (2, 3), producing the following array:

```
1 print(x * 2)
```

```
[[ 2  4  6]
 [ 8 10 12]]
```

# Introduction to Matplotlib

Matplotlib is a plotting library. In this section we give a brief introduction to the `matplotlib.pyplot` module, which provides a plotting system similar to MATLAB.

## Plotting

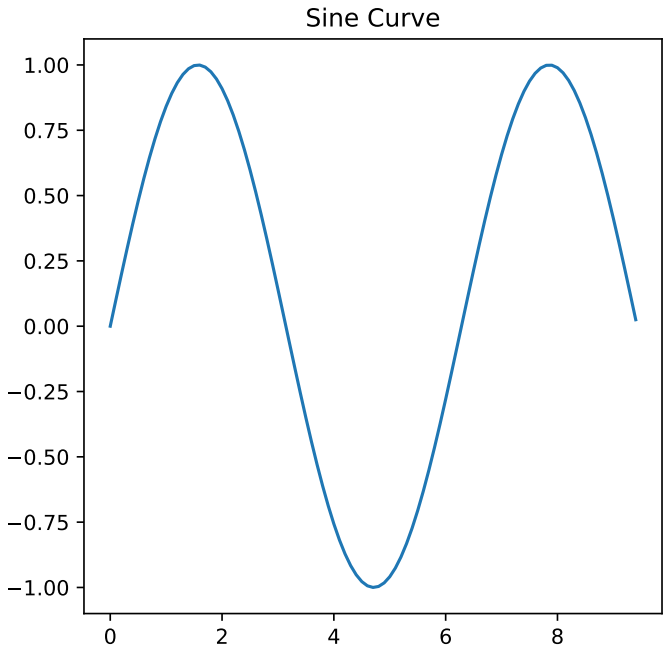
The most important function in matplotlib is `plot`, which allows you to plot 2D data. Here is a simple example:

# Example: Plotting a sine curve

- Import `matplotlib.pyplot` as `plt` and `numpy` as `np`.
- Compute the `x` and `y` coordinates for points on a sine curve.
- Plot the points using matplotlib.
- If you are **not** in a Jupyter notebook, you must call `plt.show()` to make the graphic appear.

# Plotting

```
1  import matplotlib.pyplot as plt
2  import numpy as np
3
4  x = np.arange(0, 3 * np.pi, 0.1)
5  y = np.sin(x)
6
7  plt.plot(x, y)
8  plt.title("Sine Curve")
9  plt.show()
```


Sine Curve
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
2
4
6
8

# Multiple Plots

With just a little bit of extra work, we can easily plot multiple lines at once, and add a title, legend, and axis labels.

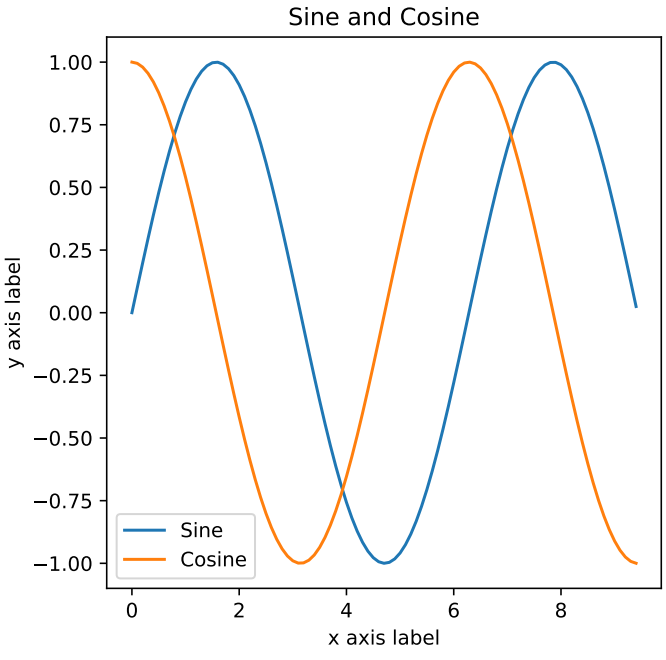
**Example: Plotting sine and cosine curves with axis labels and legend**

- Compute the x and y coordinates for both sine and cosine curves.
- Plot both lines on the same figure.
- Add x and y axis labels.
- Add a title and a legend.

# Multiple Plots

```
x = np.arange(0, 3 * np.pi, 0.1)
y_sin = np.sin(x)
y_cos = np.cos(x)

plt.plot(x, y_sin)
plt.plot(x, y_cos)
plt.xlabel("x axis label")
plt.ylabel("y axis label")
plt.title("Sine and Cosine")
plt.legend(["Sine", "Cosine"])
plt.show()
```


Sine and Cosine
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
y axis label
0
2
4
6
8
x axis label
Sine
Cosine

# Multiple Subplots

Sometimes we want multiple subplots in one figure.

**Example: Generate 6 histograms in a grid**

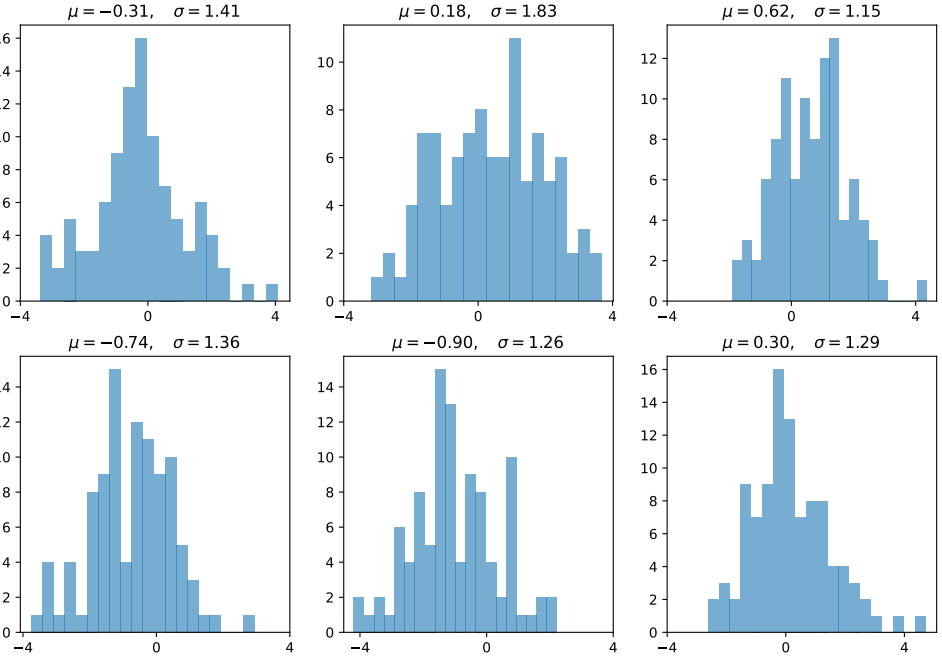
- Create a `3 x 2` grid of subplots.
- For each subplot, generate random data with a different mean `m` and standard deviation `s`.
- Plot a histogram in each subplot.
- Set titles and tick marks.

# Multiple Subplots

```
from numpy.random import uniform, normal as norm
import matplotlib.pyplot as plt

fig, axes = plt.subplots(2, 3)

for i in range(2):
    for j in range(3):
        m, s = uniform(-1, 1), uniform(1, 2)
        x = norm(loc=m, scale=s, size=100)
        axes[i, j].hist(x, alpha=0.6, bins=20)
        t = rf"$\mu = {m:.2f}, \quad \sigma = {s:.2f}$"
        axes[i, j].set(title=t, xticks=[-4, 0, 4])
```


$\mu = -0.31, \quad \sigma = 1.41$
$\mu = 0.18, \quad \sigma = 1.83$
$\mu = 0.62, \quad \sigma = 1.15$
$\mu = -0.74, \quad \sigma = 1.36$
$\mu = -0.90, \quad \sigma = 1.26$
$\mu = 0.30, \quad \sigma = 1.29$

# 3D Surface Plot

Matplotlib supports 3D plots.

**Example: 3D surface plot**

- Import the necessary 3D plotting modules.
- Define a function of two variables.
- Create meshgrid coordinates and evaluate the function.
- Plot the 3D surface.

# 3D Surface Plot

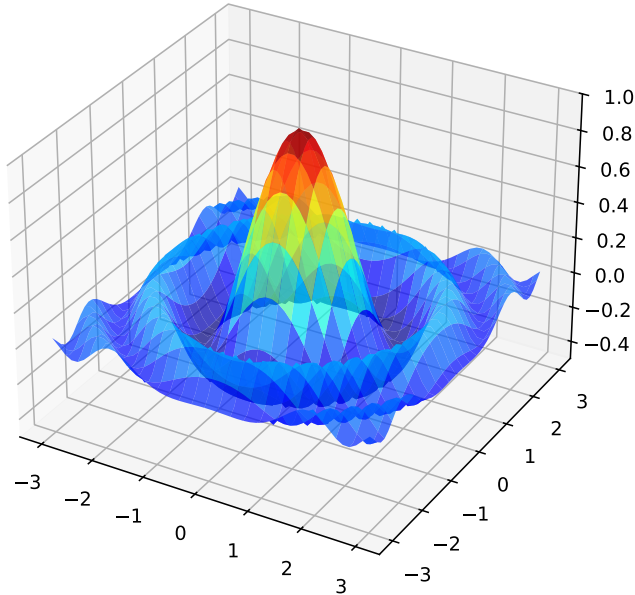
```
from mpl_toolkits.mplot3d.axes3d import Axes3D
from matplotlib import cm


def f(x, y):
    return np.cos(x**2 + y**2) / (1 + x**2 + y**2)


xgrid = np.linspace(-3, 3, 50)
ygrid = xgrid
x, y = np.meshgrid(xgrid, ygrid)
```

# 3D Surface Plot

```
fig = plt.figure(figsize=(10, 6))
ax = fig.add_subplot(111, projection="3d")
ax.plot_surface(
    x,
    y,
    f(x, y),
    rstride=2,
    cstride=2,
    cmap=cm.jet,
    alpha=0.7,
    linewidth=0.25,
)
ax.set_zlim(-0.5, 1.0)
```


1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−3
−2
−1
0
1
2
3
−3
−2
−1
0
1
2
3

# Further Reading

- The Matplotlib gallery provides many examples.
- A nice Matplotlib tutorial by Nicolas Rougier, Mike Muller and Gael Varoquaux.
- mpltools allows easy switching between plot styles: mpltools.
- Seaborn facilitates common statistics plots in Matplotlib: Seaborn

# References

The tutorial above is mainly from

- the Python Review Session of cs224n
- Python tutorial of CS231N

I strongly recommend these materials for you to get started.

There are more about Python to follow during practice. Here are some materials for your reference. From my personal experience, official documentations are often the most useful guides. `StackOverflow` and `Blogs` also help a lot.

- Official Python 3 documentation
- Official NumPy documentation