Python Foundations

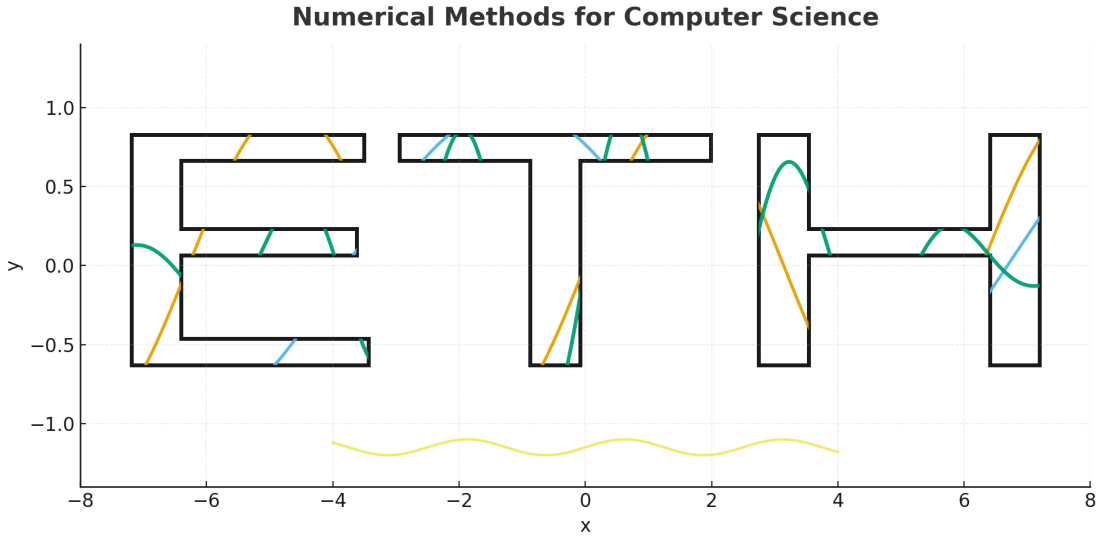
for Numerical Methods

ETH Zurich

David Mihnea-Stefan

mihdavid@ethz.ch

Fall Semester 2025



Remark: This document does not replace the official lectures. It is meant as a solid foundation for a first interaction with Python. Many additional notions will be introduced gradually during the semester, whenever they are needed.

Contents

Part I
Python Foundations

1 Introduction to python

1.1 Why Python?

Python is a high-level programming language created by Guido van Rossum in 1991. It was designed to be easy to read, simple to learn, yet powerful enough for real-world applications. Today, Python is maintained by the Python Software Foundation and has become one of the most widely used languages for science, engineering, and data analysis.

A central feature is that Python is an **interpreted language**. Unlike C or Java, which require compilation into machine code before execution, Python code is executed line by line by the Python interpreter.

What does this mean in practice?

- You can use Python interactively, typing commands and seeing results immediately.
- There is no compile–run cycle, making it ideal for rapid prototyping.
- The same script runs on any operating system where the interpreter is installed.

Example: Python (interpreted)

```
x = 5
print(x * 2)
```

Example: C (compiled)

```
// C program
#include <stdio.h>
int main() {
    int x = 5;
    printf("%d\n", x * 2);
    return 0;
}
```

In C, the code must be compiled with a tool such as `gcc` before execution. In Python, the code runs immediately in the interpreter.

Other key features of Python:

- Readable syntax close to natural language and mathematics,
- Dynamic typing: no need to declare variable types,
- Rich ecosystem of libraries (NumPy, SciPy, Matplotlib, Pandas, etc.),
- Open-source and free, with one of the largest programming communities.

Note. Python itself has a relatively small and clean core language, but its real strength comes from the vast ecosystem of external packages developed for specific domains (numerical computing, data analysis, machine learning, etc.). This design choice makes

Python both highly versatile and very powerful, allowing it to adapt to many different applications.

Conclusion: Python combines readability, interactivity, and powerful libraries, which makes it a preferred choice for both beginners and experts in scientific computing.

1.2 Why Python in this course?

In *Numerical Methods for Computer Science*, Python is chosen as the main working environment not because of its general popularity, but because it directly supports the type of problems we will study in this course. Rather than focusing on programming itself, our goal is to explore algorithms and numerical methods using a language that minimizes technical overhead and maximizes clarity.

- **Numerical libraries:** `NumPy` and `SciPy` provide efficient, ready-to-use implementations of algorithms for linear algebra, integration, differential equations, and optimization.
- **Data handling and visualization:** With `Pandas` for structured data and Matplotlib / Seaborn for high-quality plots, Python makes it straightforward to process input data and illustrate numerical results.
- **Interactive workflows:** Through `Jupyter Notebooks`, we can seamlessly combine explanations, code, and results in one document, which is ideal for both teaching and reproducibility.
- **Efficiency without complexity:** While Python is simple to write and read, the heavy computations are executed in optimized C and Fortran routines underneath, giving us the best of both worlds.
- **Transferable skills:** Beyond this course, Python is a standard tool in academia and industry, meaning that the techniques learned here are directly applicable in research and professional practice.

Conclusion: Python lets us concentrate on understanding numerical algorithms and their behavior, rather than on low-level programming details. This makes it the ideal companion for studying numerical methods in computer science.

2 Basics

2.1 First Steps

Python lets you write and run code without boilerplate (no class or `main` needed for simple scripts).

Run Python three ways:

1. **REPL / Interpreter** (type and execute line-by-line)

```
$ python
>>> print("Hello, world!")
Hello, world!
>>> 2 + 3 * 4
14
```

2. **Script** (save as `hello.py`, then run)

```
# hello.py
print("Hello from a script!")
x = 21
print(x * 2)

```

```
$ python hello.py
Hello from a script!
42
```

3. **Jupyter Notebook** (interactive documents mixing code, text, and math)

```
$ pip install jupyter
$ jupyter notebook    # open in browser
```

Java/C++ note: No compilation step is required before running simple Python code.

Important note: Throughout the semester we will *regularly use CodeExpert*, since this platform will also be required during the exam. Even if the interface may appear less flexible or elegant compared to other environments, it is extremely useful for training and for ensuring that you are well prepared for the final evaluation.

When working outside CodeExpert, there are excellent alternatives to Jupyter:

- **IPython** (which is in fact the foundation of Jupyter) combined with a text editor such as *vi*, *emacs*, or *gedit*;
- **Visual Studio Code**, a modern and widely used IDE that integrates well with Python.

Comments and docstrings

```
# Single-line comment
"""
Triple-quoted strings are often used as multi-line comments or
   docstrings.
"""
print("OK")
```

Printing and input

```
name = input("Your name: ")   # returns a string
print("Hello,", name)
```

Whitespace and indentation **Indentation is syntax in Python** (no braces). Use consistent 4 spaces.

```
# Good:
if True:
    print("Indented block")

# Bad (mixing tabs/spaces)      SyntaxError
```

2.2 Variables and Assignment

In Python, variables are **dynamically typed**: you do not declare their type explicitly. The interpreter determines the type of a value at runtime. This contrasts with Java/C++, where a variable must be declared with a fixed type (`int`, `double`, etc.).

Basic assignment

Assignment uses the = operator. A variable is created when you first assign to it.

```
x = 10              # int
y = 3.14            # float (double precision)
flag = True         # bool (note: capital T/F)
msg = "hello"       # str (Unicode by default)

print(type(x), type(y), type(flag), type(msg))
# <class 'int'> <class 'float'> <class 'bool'> <class 'str'>
```

Note: In Python, integers can grow arbitrarily large (no overflow like in C++/Java).

Re-assignment and dynamic typing

A variable can be reassigned to a value of a different type without error:

```
x = 42          # int
x = "forty"     # now a str
print(x)
```

This flexibility makes Python concise, but also means type errors only appear at runtime.

Multiple assignment and unpacking

Python supports assigning multiple variables at once, as well as unpacking tuples, lists, and iterables:

```
a, b = 1, 2                # tuple unpacking
a, b = b, a                # swap (no temp variable needed)
x, y, z = (10, 20, 30)     # unpacking a tuple
x, *rest = [1, 2, 3, 4]    # x=1, rest=[2,3,4]
```

This is particularly useful in numerical methods (e.g., updating iteration variables).

Chained assignment

A single value can be assigned to multiple variables:

```
x = y = z = 0
print(x, y, z)    # 0 0 0
```

Augmented assignment

Shorthand operators both update and reassign:

```
count = 0
count += 1    # equivalent to count = count + 1
count *= 2    # now count is 2
```

Java/C++ note: same as `+=`, `*=`, etc. in those languages.

Identity vs. equality

Two different concepts:

- **Equality** (`==`) checks whether two objects have the same value.
- **Identity** (`is`) checks whether two variables point to the same object in memory.

```
a = [1, 2]; b = [1, 2]
print(a == b)  # True  (same contents)
print(a is b)  # False (different objects)
```

Mutable vs. immutable types

- **Immutable:** int, float, bool, str, tuple
- **Mutable:** list, dict, set, custom objects

```
msg = "hi"
msg2 = msg
print(msg is msg2)  # True (both point to same string)

lst1 = [1,2]; lst2 = lst1
lst2.append(3)
print(lst1)  # [1,2,3] (both refer to the same list!)
```

Warning: assignment does not copy a mutable object; use `copy()` or `copy.deepcopy()` if needed.

Naming conventions (PEP 8)

Python follows standard style guidelines:

- variables/functions: `snake_case` (e.g., `max_value`)
- constants: `UPPER_CASE` (e.g., `PI = 3.1415`)
- classes: `CamelCase` (e.g., `MySolver`)

Good practices

- Choose descriptive names (`n_steps`, `tolerance`, etc.).
- Avoid shadowing built-ins: do not name variables `list`, `dict`, `str`, `sum`, etc.
- For numerical code, keep types consistent (e.g., floats vs. ints).

Micro-exercises

1. Swap two variables without using a temporary.
2. Write one line that assigns values `1, 2, 3` to `x, y, z`.
3. Show the difference between `is` and `==` using lists.
4. Demonstrate that strings are immutable by trying to change one character in a string.

2.3 Basic Data Types

Python comes with a rich set of built-in data types. Unlike C++/Java, you do not need to include headers or import libraries for these basics. Types are objects themselves, and can be inspected at runtime with `type()`.

Integers (arbitrary precision)

In Python, integers have **arbitrary precision**: they do not overflow at fixed 32/64-bit limits. You can also use underscores for readability in numeric literals.

```
n = 2**100              # very large integer
m = 1_000_000           # underscores improve readability
print(n.bit_length(), m)
# Output: number of bits to represent n, and m = 1000000
```

Java/C++ note: unlike `int` or `long`, Python's `int` grows automatically.

Floats (IEEE 754 double precision)

Floats in Python are implemented as C doubles. Thus they follow IEEE 754 rules and exhibit rounding errors.

```
x = 0.1 + 0.2
print(x)                # 0.30000000000000004
# Compare with tolerance instead of "==" :
import math
print(math.isclose(x, 0.3, rel_tol=1e-9))
```

Tip: always use `math.isclose()` or `numpy.isclose()` when testing float equality.

Complex numbers (built-in)

Unlike Java or C++, Python has native complex numbers.

```
z = 1 + 2j              # note: "j" is imaginary unit
print(z.real, z.imag, abs(z))
# Output: 1.0 2.0 2.236...
```

Useful in numerical methods: eigenvalues, FFTs, differential equations.

Booleans and truthiness

Python has a dedicated `bool` type, but also defines a general notion of *truthiness*: empty containers, zero, and `None` evaluate to `False`; everything else is `True`.

```
bools = [False, True, bool(0), bool(1), bool(""), bool("x")]
print(bools)    # [False, True, False, True, False, True]

print(bool([]), bool([0]), bool({}), bool({"a":1}))
# False True False True
```

Java/C++ note: In C/C++, any nonzero integer is treated as `true` and `0` as `false`. In Python, conditions are based on *truthiness*: `0`, `0.0`, `0j`, `None`, empty strings, and empty containers evaluate to `False`, while all other values (including `1`) evaluate to `True`.

Strings (Unicode by default)

Strings are sequences of Unicode characters. They are immutable.

```
s1 = "double quotes"; s2 = 'single quotes'
multi = """multi
line"""              # triple quotes
name, age = "Alice", 20
msg = f"{name} is {age} years old"    # f-string formatting
```

String operations:

```
print("abc" + "def")      # concatenation
print("ha" * 3)           # repetition: "hahaha"
print("Python"[0])        # indexing (0-based): 'P'
print("Python"[2:5])      # slicing: "tho"
```

Java/C++ note: strings are more powerful than `std::string` / `String`; no need for `char[]` or manual Unicode handling.

Operators and math

Python provides all the usual arithmetic operators.

```
# +, -, *, / (float), // (floor division), % (mod), ** (power)
print(7 / 2, 7 // 2, 7 % 2, 2 ** 10)
# 3.5 3 1 1024

import math
print(math.pi, math.e, math.sqrt(2))
```

Java/C++ note: be careful: `/` always yields float, even if both operands are ints. Floor division `//` is not truncation but mathematical floor.

Type conversions

You can convert explicitly between types:

```
int(3.7)          # 3
float(2)          # 2.0
str(42)           # "42"
complex(2, 3)     # (2+3j)
```

Special constants

- `None`: absence of value (like `null`).
- `float("inf")`, `float("-inf")`, `float("nan")` for numerical infinity and NaN.

```
print(float("inf"), float("-inf"), float("nan"))
print(math.isnan(float("nan")))
```

2.4 Data Structures (Containers)

Python ships four core container types: **list**, **tuple**, **dict**, **set**. They all support iteration, membership tests (`in`), and comprehensions.

Lists (mutable, ordered; like `ArrayList`)

Lists are dynamic arrays: they grow/shrink automatically, can contain mixed types, and allow slicing.

```
nums = [10, 20, 30]
nums.append(40)                 # [10, 20, 30, 40]
nums.extend([50, 60])           # concatenate
nums.insert(1, 99)              # insert at position 1
print(nums[0], nums[-1])        # first, last element
print(nums[1:4])                # slice [1..3]
print(nums[::-1])               # reversed copy
```

Modify/remove:

```python
nums[2] = 200                   # update by index
val = nums.pop()                # remove last, returns it
nums.remove(99)                 # remove first matching value
nums.clear()                    # empty list
```

Sorting:

```python
data = [("alice", 3), ("bob", 1), ("carol", 2)]
print(sorted(data))                     # sort by first element
print(sorted(data, key=lambda t: t[1], reverse=True))
```

Shallow copies and pitfalls:

```python
a = [[0]] * 3       # same inner list repeated!
a[0][0] = 99
print(a)            # [[99], [99], [99]]

b = [[0] for _ in range(3)]   # distinct inner lists
b[0][0] = 99
print(b)            # [[99], [0], [0]]
```

Java/C++ note: unlike fixed-size arrays, Python lists resize automatically.

Tuples (immutable, ordered)

Tuples are like fixed-size immutable lists. Useful for returning multiple values.

```python
pt = (3, 4)
# pt[0] = 5  # TypeError: tuples are immutable
single = (1,)           # note the trailing comma
x, y = pt               # unpacking
```

Dictionaries (hash maps / key–value)

Python's built-in mapping type.

```python
scores = {"Alice": 10, "Bob": 9}
scores["Carol"] = 8
print(scores["Alice"])              # 10
print(list(scores.keys()))          # keys
print(list(scores.values()))        # values
print(list(scores.items()))         # (key, value) pairs

print("Bob" in scores)              # membership checks keys
print(scores.get("Eve", 0))         # safe default

scores.update({"Bob": 10, "Eve": 7})   # merge/update
squared = {i: i*i for i in range(5)}   # dict comprehension
```

Note: Dicts preserve insertion order (since Python 3.7). Keys must be *hashable* (immutable).

Java/C++ note: equivalent to `HashMap<String,Integer>`.

Sets (unique elements, unordered)

Sets store unique elements with fast membership tests.

```
S = {1, 2, 2, 3}          # {1, 2, 3}
S.add(4)
S.discard(2)              # safe remove
A, B = {1,2,3}, {3,4}
print(A | B)              # union {1,2,3,4}
print(A & B)              # intersection {3}
print(A - B)              # difference {1,2}
print(A ^ B)              # symmetric diff {1,2,4}

evens = {i for i in range(10) if i % 2 == 0}   # set comp
```

Iteration patterns

All containers are iterable.

```
# Over items:
for x in [10, 20, 30]:
    print(x)

# With indices:
xs = ["a", "b", "c"]
for i, v in enumerate(xs):
    print(i, v)

# Over dicts:
for key, val in scores.items():
    print(key, "->", val)
```

Note: Iteration order is predictable (lists: insertion order; dicts: insertion order; sets: arbitrary but consistent within a run).

2.5 Strings (in depth)

Strings are immutable sequences of Unicode characters. They support slicing, concatenation, searching, and many built-in methods.

Indexing & slicing

```
s = "Numerical Methods"
print(s[0], s[-1])          # 'N' 's'
print(s[0:9])               # 'Numerical'
print(s[::2])               # step 2
print(s[::-1])              # reversed string
```

Common methods

```
t = "  data,science,python  "
print(t.strip())                # trim whitespace
print(t.upper(), t.lower())
```

```
print(t.replace("python", "NumCS"))

parts = t.strip().split(",")    # ["data","science","python"]
joined = "-".join(parts)        # "data-science-python"

print("num" in "numerical")     # True
print("abc".startswith("a"), "xyz".endswith("z"))
```

Formatting (f-strings recommended)

```
name, score = "Alice", 9.8765
print(f"{name} scored {score:.2f}")    # Alice scored 9.88
```

Java/C++ note: F-strings resemble `String.format` in Java or `printf` in C.

Escapes and raw strings

```
print("line1\nline2\t(tab)")
print(r"C:\Users\you\folder")    # raw string: no escapes
```

Bytes vs str (encoding)

Strings are Unicode; encode/decode converts to raw bytes.

```
text = "       3.14159"
b = text.encode("utf-8")     # bytes
print(b)                     # b'\xcf\x80 \xe2\x89\x88 3.14159'
print(b.decode("utf-8"))     # back to str
```

Immutability tip

```
s = "hello"
# s[0] = 'H'  # TypeError: strings are immutable
s = "H" + s[1:]  # create a new string
```

2.6 Exercises

The following exercises consolidate your understanding of Python basics (variables, types, data structures, strings). Try them in order.

Warm-up

1. Write a script that prints your name, age, and favorite course using f-strings.
2. Compute 2^{1000} and print the number of digits in the result.
3. Ask the user for an integer n and print the sum $1 + 2 + \cdots + n$ (use the formula, not a loop).

Numbers and math

1. Show that `0.1 + 0.2 == 0.3` is `False`, then fix the comparison with `math.isclose`.
2. Create a complex number $3 + 4i$ and print its magnitude (should be 5).
3. Convert `3.7` to an integer and `42` to a string. Print their types.

Variables and assignment

1. Swap two variables without using a temporary variable.
2. Demonstrate the difference between `==` and `is` using two identical lists.
3. Show that strings are immutable by trying to replace one character directly.

Containers

1. Create a list of the first 5 square numbers. Append the next square and print the list.
2. Sort a list of names `["Eve","Alice","Bob"]` alphabetically and then by length.
3. Create a tuple `(x,y)` and unpack it into two variables.
4. Create a dictionary mapping three students to their grades. Update one grade and add a new student.
5. From a list of words, build a set of unique first letters.

Strings

1. Print the first 3 characters of your name and the last 2 characters.
2. Given the string `" data,science,python "`, strip whitespace, split on commas, and join with dashes.
3. Check whether the substring `"num"` occurs in `"Numerical"` (case sensitive).
4. Use an f-string to format $\pi \approx 3.14159$ with exactly 3 decimal places.
5. Encode the string `"  3.14159"` into UTF-8 bytes and then decode back.

Challenge

1. Write a script that asks the user for a sentence and then:
   - prints the number of words,
   - prints the unique words (case-insensitive),
   - prints the sentence reversed (word order, not letters).

3 Control Flow

Programs need to make decisions and repeat actions. Python provides conditionals, loops, comprehensions, and other constructs for directing the flow of execution. Unlike Java/C++, blocks are controlled entirely by indentation.

3.1 Conditionals (if / elif / else)

```
x = 10
if x > 0:
    print("positive")
elif x == 0:
    print("zero")
else:
    print("negative")
```

Notes:

- `elif` is Python's `else if`.
- Indentation (4 spaces) replaces braces {}.

Truthiness in conditions Any value can be used in a condition. The following are considered **False**:

- `False`, `None`
- zero numeric values: `0`, `0.0`, `0j`
- empty containers: `""`, `[]`, `{}`, `set()`

Everything else is treated as **True**.

```
if []:
    print("non-empty list")
else:
    print("empty list")    # will be printed
```

Chained comparisons Python supports mathematical-style chaining:

```
x = 5
print(0 < x < 10)    # True
```

Conditional expressions (ternary operator) Short form for `if/else` inside an expression:

```
n = 7
parity = "even" if n % 2 == 0 else "odd"
print(parity)
```

Examples

```
name = input("Your name: ")
if name:
    print("Hello,", name)
else:
    print("You did not type a name")

score = 87
grade = "Pass" if score >= 60 else "Fail"
print(grade)
```

3.2 Loops

Loops repeat actions while a condition holds (`while`) or over items in an iterable (`for`). Blocks are defined by indentation (no braces).

While loops

```
# print first 5 natural numbers
i = 1
while i <= 5:
    print(i)
    i += 1
```

Java/C++ note: the semantics are identical; pay attention to indentation and to the fact that + + **and** — **-** operators do not exist in Python

For loops (iterate over sequences and iterables)

```
# iterate over a list
for name in ["Alice", "Bob", "Carol"]:
    print(name)

# iterate over a string (characters)
for ch in "NumPy":
    print(ch)

# iterate over a numeric range (0..4)
for k in range(5):
    print(k)

# range(start, stop, step) -- stop is exclusive
for k in range(2, 10, 2):  # 2,4,6,8
    print(k)
```

Java/C++ note: in Python, **for** iterates directly over elements (like a for-each); use range() for indices.

Loop control: `break`, `continue`, `pass`

```
# find first multiple of 7 in 1..100
for n in range(1, 101):
    if n % 7 == 0:
        print("first multiple:", n)
```

```
        break               # exit the loop early

# skip odd numbers
for n in range(10):
    if n % 2 == 1:
        continue            # skip the rest of this iteration
    print(n, "is even")

# placeholder for an empty block
def todo():
    pass                    # syntactic no-op; fill later
```

Loop `else` (runs only if loop wasn't broken)

```
# primality test (simple)
n = 29
for d in range(2, int(n**0.5) + 1):
    if n % d == 0:
        print("composite by", d)
        break
else:
    print("prime")  # runs only if no break occurred
```

Tip: `else` on loops is useful when nothing was found, without needing extra flags.

Iterating with indices (`enumerate`)

```
xs = ["a", "b", "c"]
for i, val in enumerate(xs):      # i from 0 by default
    print(i, val)
# enumerate(xs, start=1) indexing from 1
```

Parallel iteration (`zip`)

```
names = ["Alice", "Bob", "Carol"]
scores = [10, 7, 9]
for name, score in zip(names, scores):
    print(name, "->", score)
```

Safe iteration when mutating

```
# DON'T: mutating a list while iterating over it can skip items
xs = [1, 2, 3, 4]
for x in xs:
    if x % 2 == 0:
        xs.remove(x)    # risky

# DO: iterate over a copy or build a new list
xs = [1, 2, 3, 4]
xs = [x for x in xs if x % 2 == 1]   # list comprehension
```

3.3 Iteration Utilities

Beyond basic `for`/`while`, Python provides built-in functions to make iteration more expressive and concise.

`range()` (integer sequences)

```
for i in range(5):           # 0,1,2,3,4
    print(i)

for i in range(2, 10, 2):    # start, stop, step
    print(i)                 # 2,4,6,8

print(list(range(3)))        # [0,1,2]
```

Java/C++ note: `range` is like a generator of indices; stop is exclusive.

`enumerate()` (index + value)

```
names = ["Ada", "Alan", "Grace"]
for i, name in enumerate(names):
    print(i, name)
# 0 Ada; 1 Alan; 2 Grace

for i, name in enumerate(names, start=1):
    print(i, name)           # index starts at 1
```

`zip()` (parallel iteration)

```
names = ["Ada", "Alan", "Grace"]
scores = [10, 7, 9]
for name, score in zip(names, scores):
    print(name, "->", score)
# Ada -> 10; Alan -> 7; Grace -> 9

# zip stops at shortest sequence
```

`reversed()` and `sorted()`

```
nums = [3,1,4,1,5]
for n in reversed(nums):
    print(n)

for n in sorted(nums):
    print(n)        # 1,1,3,4,5

for n in sorted(nums, reverse=True):
    print(n)        # 5,4,3,1,1
```

Iteration over dicts

```
scores = {"Ada": 10, "Alan": 7, "Grace": 9}
for k in scores:                  # keys by default
    print(k)

for k, v in scores.items():       # key-value pairs
```

```
    print(k, v)

for v in scores.values():          # values only
    print(v)
```

Iteration protocol (advanced note) Any object that implements `__iter__()` (returns an iterator) can be used in `for`. Iterators implement `__next__()` and raise `StopIteration` when exhausted.

3.4 Comprehensions

Python provides a compact way to build new sequences and containers from existing iterables. Comprehensions are often more readable and efficient than writing explicit loops.

List comprehensions

```
squares = [x*x for x in range(5)]
print(squares)     # [0, 1, 4, 9, 16]

# with condition
evens = [x for x in range(10) if x % 2 == 0]
print(evens)       # [0, 2, 4, 6, 8]

# nested comprehensions
matrix = [[i*j for j in range(3)] for i in range(3)]
print(matrix)      # [[0,0,0],[0,1,2],[0,2,4]]
```

Dict comprehensions

```
# map numbers to their squares
squares = {x: x*x for x in range(5)}
print(squares)    # {0:0, 1:1, 2:4, 3:9, 4:16}

# invert a dict
scores = {"Alice": 10, "Bob": 7}
inverse = {v: k for k, v in scores.items()}
print(inverse)    # {10: "Alice", 7: "Bob"}
```

Set comprehensions

```
letters = {ch for ch in "abracadabra" if ch not in "ab"}
print(letters)    # unique letters excluding a/b
```

Generator expressions Like list comprehensions, but use parentheses `()` instead of brackets `[]`. They produce a lazy iterator (values computed on demand).

```
gen = (x*x for x in range(5))
print(gen)            # <generator object ...>
print(next(gen))      # 0
print(list(gen))      # [1,4,9,16] (rest of values)
```

Performance note

- List/Dict/Set comprehensions build the entire container immediately.
- Generator expressions are memory-efficient for large or infinite sequences.

Readability tip Comprehensions are best for simple transformations and filters. If logic becomes too complex (nested loops, multiple conditions), a regular loop may be clearer.

3.5 Built-in Control Helpers

Python provides higher-order functions and built-ins that often replace explicit loops. These make code shorter and closer to mathematical notation.

`any()` **and** `all()`

```
nums = [3, 5, 7, 8]

print(any(n % 2 == 0 for n in nums))    # True (at least one even)
print(all(n > 0 for n in nums))         # True (all positive)

# equivalent loop for all():
flag = True
for n in nums:
    if not (n > 0):
        flag = False
        break
print(flag)
```

`map()` **and** `filter()` Apply a function to every element, or filter elements by condition. They return iterators (convert to list to display).

```
nums = [1, 2, 3, 4, 5]

squares = list(map(lambda x: x*x, nums))
print(squares)    # [1,4,9,16,25]

evens = list(filter(lambda x: x % 2 == 0, nums))
print(evens)      # [2,4]
```

Pythonic alternative: list comprehensions.

```
squares = [x*x for x in nums]
evens   = [x for x in nums if x % 2 == 0]
```

Aggregation functions

```
nums = [3, 1, 4, 1, 5]
print(sum(nums))       # 14
print(min(nums))       # 1
print(max(nums))       # 5
print(sorted(nums))    # [1,1,3,4,5]
```

Itertools (for advanced iteration) The `itertools` module extends these patterns.

```
import itertools

for pair in itertools.combinations([1,2,3], 2):
    print(pair)    # (1,2), (1,3), (2,3)
```

Note: Itertools is part of the standard library and extremely useful for combinatorics and numerical methods.

4 Defining Functions

Functions let you group code into reusable blocks with a name, parameters, and (optionally) a return value. In Python, functions are **first-class objects**: they can be passed around, stored in variables, and even created at runtime. Defining functions is central to writing clean and modular programs.

4.1 Defining simple functions

Functions are defined with the keyword `def`, followed by the name, parameter list in parentheses, and a colon. The function body is indented. Use `return` to give back a value; without it, the function returns `None`.

```
def greet():
    print("Hello from a function!")

greet()     # call the function
```

With parameters

```
def square(x):
    return x * x

print(square(5))     # 25
```

Multiple parameters

```
def hypotenuse(a, b):
    return (a*a + b*b) ** 0.5

print(hypotenuse(3, 4))    # 5.0
```

No explicit return If you omit `return`, the function returns `None` by default.

```
def say_hi(name):
    print("Hi", name)

result = say_hi("Alice")
print(result)      # prints Hi Alice, then None
```

Docstrings Triple-quoted strings at the start of a function serve as documentation.

```
def factorial(n):
    """Compute n! (factorial) recursively."""
    if n == 0:
        return 1
    return n * factorial(n-1)

help(factorial)    # shows the docstring
```

Java/C++ note: Unlike Java, functions do not belong to classes by default (C++ style). You can define free-standing functions without wrapping them in a class.

4.2 Arguments and Parameters

Function parameters in Python are flexible: you can call functions with positional arguments, keyword arguments, or both. Defaults make parameters optional.

Positional arguments Arguments are matched to parameters in order.

```
def add(a, b):
    return a + b

print(add(2, 3))      # 5
```

Keyword arguments Arguments can also be passed by name, making the call clearer.

```
def greet(name, greeting):
    print(f"{greeting}, {name}!")

greet(name="Alice", greeting="Hello")
greet(greeting="Hi", name="Bob")
```

Default values Parameters can have default values, making them optional.

```
def greet(name, greeting="Hello"):
    print(f"{greeting}, {name}!")

greet("Alice")                # uses default
greet("Bob", greeting="Hi")   # override default
```

Mixing positional and keyword

```
def power(base, exp=2):
    return base ** exp

print(power(3))           # 9
print(power(3, 3))        # 27
print(power(base=2, exp=5))    # 32
```

Rules:

- Positional arguments must come before keyword arguments.
- Default parameters must come after non-default ones.

Keyword-only arguments (advanced) Using * enforces that certain arguments must be specified by name:

```
def divide(a, b, *, precision=2):
    return round(a / b, precision)

print(divide(5, 2))                 # 2.5
print(divide(5, 2, precision=4))    # 2.5 (4 decimals)
```

4.3 Variable-length Arguments

Sometimes the number of arguments is not known in advance. Python allows functions to accept a variable number of positional or keyword arguments.

`*args` (extra positional arguments) Using `*args` collects additional positional arguments into a tuple.

```
def total(*args):
    print(args)           # tuple of arguments
    return sum(args)

print(total(1, 2, 3))         # 6
print(total(4, 5, 6, 7, 8))   # 30
```

`**kwargs` (extra keyword arguments) Using `**kwargs` collects additional keyword arguments into a dictionary.

```
def describe(**kwargs):
    for key, value in kwargs.items():
        print(key, "->", value)

describe(name="Alice", age=20, course="NumCS")
# name -> Alice
# age -> 20
# course -> NumCS
```

Mixing fixed, `*args`, and `**kwargs`

```
def report(title, *items, **meta):
    print("Title:", title)
    print("Items:", items)
    print("Meta:", meta)

report("Shopping List", "apples", "bananas",
       date="2025-01-01", store="Market")
```

Argument unpacking The operators `*` and `**` can also be used when calling functions, to unpack sequences and dicts into arguments.

```
def add(a, b, c):
    return a + b + c

values = [1, 2, 3]
print(add(*values))     # equivalent to add(1,2,3)

params = {"a": 10, "b": 20, "c": 30}
print(add(**params))    # equivalent to add(a=10, b=20, c=30)
```

4.4 Scope and Lifetime

Each variable in Python has a **scope** (where it is visible) and a **lifetime** (how long it exists). By default, names defined inside a function are **local** to that function.

Local vs global variables

```
x = 10            # global variable

def f():
    x = 5         # local variable (shadows global)
    print("inside:", x)

f()
print("outside:", x)
# inside: 5
# outside: 10
```

`global` keyword If you want to modify a global variable inside a function, declare it with `global`.

```
count = 0

def increment():
    global count
    count += 1

increment()
print(count)    # 1
```

`nonlocal` keyword Inside nested functions, `nonlocal` lets you modify variables from the nearest enclosing (non-global) scope.

```
def outer():
    x = 0
    def inner():
        nonlocal x
        x += 1
        return x
    print(inner(), inner())

outer()    # 1 2
```

Lifetime

- Global variables live as long as the program runs.
- Local variables exist only while the function is executing.

Java/C++ note: In Python there are no block-scoped variables inside `if` or `for` (all names declared in a function belong to that function's scope).

4.5 First-class Functions

In Python, functions are **first-class objects**. This means they can be:

- assigned to variables,
- passed as arguments,
- returned from other functions,
- stored in data structures.

Assigning to variables

```
def greet(name):
    return f"Hello, {name}!"

say_hi = greet       # assign function to new name
print(say_hi("Alice"))
```

Passing functions as arguments

```
def apply_twice(func, x):
    return func(func(x))

def square(n): return n*n

print(apply_twice(square, 3))    # square(square(3)) = 81
```

Returning functions

```
def make_multiplier(k):
    def multiply(x):
        return k * x
    return multiply

times3 = make_multiplier(3)
print(times3(10))    # 30
```

Storing in collections

```
def inc(x): return x + 1
def dec(x): return x - 1

funcs = [inc, dec]
for f in funcs:
    print(f(10))    # 11, then 9
```

Note: This ability enables functional programming patterns, callbacks, and higher-order functions.

4.6 Lambdas (anonymous functions)

Python allows defining functions without a name using the keyword `lambda`. These are often called **anonymous functions** or simply **lambdas**. They are useful for short, throwaway functions, especially when passing functions as arguments.

Basic syntax

```
# general form:
# lambda parameters: expression

f = lambda x: x * x
print(f(5))    # 25
```

Equivalent with:

```
def f(x):
    return x * x
```

Multiple parameters

```
add = lambda a, b: a + b
print(add(3, 4))    # 7
```

Using with built-in functions Lambdas are commonly used with functions like `sorted`, `map`, `filter`, `reduce`.

```
# sort by length of string
words = ["Python", "C", "Haskell", "Go"]
print(sorted(words, key=lambda w: len(w)))

# map and filter
nums = [1, 2, 3, 4, 5]
squares = list(map(lambda x: x*x, nums))
evens   = list(filter(lambda x: x%2==0, nums))
print(squares, evens)
```

Conditional expressions inside lambda Because lambdas can only contain **expressions**, conditional logic must use the ternary operator:

```
sign = lambda x: "positive" if x > 0 else ("zero" if x == 0 else "
    negative")
print(sign(-3), sign(0), sign(5))
```

Scope and closures Lambdas capture variables from their enclosing scope (just like inner functions).

```
def make_adder(k):
    return lambda x: x + k

plus10 = make_adder(10)
print(plus10(5))    # 15
```

Limitations

- Lambdas must be a **single expression** (cannot contain statements like `for`, `while`, or `return`).
- They are less readable for complex logic. Use `def` for clarity.

Style guide (PEP 8)

- Use `lambda` for short, local functions passed as arguments.
- Prefer `def` for anything non-trivial or reused.

Java/C++ note Similar to anonymous classes in Java (before Java 8) or to `[]() {...}` lambdas in C++11+. However, Python's lambdas are much more limited: only expressions, no statements.

4.7 Exercises

The following exercises will help you practice defining and using functions in Python.

1. **Simple definitions**
   (a) Write a function `square(x)` that returns x^2 and test it.
   (b) Write a function `greet(name)` that prints `"Hello, <name>"`.
2. **Parameters and return values**
   (a) Write a function `hypotenuse(a, b)` that computes $\sqrt{a^2 + b^2}$.
   (b) Implement a recursive function `factorial(n)`.
3. **Default and keyword arguments**
   (a) Define `power(base, exp=2)` with a default exponent.
   (b) Define `greet(name, greeting="Hello")` and call it with and without the second argument.
4. **Variable-length arguments**
   (a) Write a function `total(*args)` that computes the sum of any number of arguments.
   (b) Write a function `describe(**kwargs)` that prints all keyword arguments in the form `key=value`.
5. **Scope**
   (a) Demonstrate the difference between local and global variables by reusing the same name inside and outside a function.
   (b) Write a function `outer()` with an inner function `inner()` that modifies a variable using `nonlocal`.
6. **First-class functions**
   (a) Write a function `apply_twice(func, x)` and test it with `square`.
   (b) Write a function `make_multiplier(k)` that returns a new function multiplying its argument by k.
7. **Lambdas**

(a) Sort a list of tuples `(name, score)` by score using a `lambda`.

(b) Use `map` + `lambda` to compute the squares of the first 10 numbers.

(c) Use `filter` + `lambda` to keep only the even numbers from a list.

5 Input and Output

Python provides simple ways to interact with the user and with files. The two main built-ins are `input()` for reading from standard input and `print()` for writing to standard output. For persistent data, Python supports file I/O using the `open()` function and context managers.

5.1 User Input

Reading input

```
name = input("Your name: ")    # always returns a string
print("Hello,", name)
```

Type conversion Use `int()`, `float()`, etc. to convert.

```
n = int(input("Enter an integer: "))
x = float(input("Enter a float: "))
print(n, x)
```

Java/C++ note Unlike `Scanner` or `cin`, `input()` always returns a string; explicit conversion is required.

5.2 Output with `print()`

Basic printing

```
print("Hello, world!")
```

Multiple arguments

```
a, b = 2, 3
print("a =", a, "b =", b)
# automatically inserts spaces
```

Separator and end

```
print(1, 2, 3, sep=", ")    # 1, 2, 3
print("no newline", end="")
print(" <- same line")
```

Formatted output Prefer f-strings:

```
pi = 3.14159
print(f"pi      {pi:.3f}")    # pi      3.142
```

5.3 File I/O

Opening and reading a file

```
f = open("data.txt", "r")
contents = f.read()           # read entire file as string
f.close()
```

Reading line by line

```
with open("data.txt") as f:       # auto-close with 'with'
    for line in f:
        print(line.strip())
```

Writing to a file

```
with open("output.txt", "w") as f:
    f.write("First line\n")
    f.write("Second line\n")
```

Append mode

```
with open("output.txt", "a") as f:
    f.write("Appended line\n")
```

File modes

- `"r"` – read (default)
- `"w"` – write (overwrite)
- `"a"` – append
- `"b"` – binary mode (e.g., `"rb"`, `"wb"`)

Example: reading numbers from a file

```
nums = []
with open("numbers.txt") as f:
    for line in f:
        if line.strip():
            nums.append(float(line))
print("Mean =", sum(nums)/len(nums))
```

5.4 Standard streams (advanced note)

Python also exposes:

- `sys.stdin` – standard input
- `sys.stdout` – standard output
- `sys.stderr` – error output

```
import sys
sys.stdout.write("Hello without newline")
```

6 Modules and Imports

Python code can be organized into **modules** (single `.py` files) and **packages** (folders containing `__init__.py`). This allows reusing code and using external libraries like `numpy`.

6.1 Importing modules

Basic import

```
import math

print(math.sqrt(16))     # 4.0
print(math.pi)           # 3.14159...
```

Alias imports It is common to give modules a short alias.

```
import numpy as np
print(np.arange(5))     # [0 1 2 3 4]
```

Import selected names You can import only what you need from a module.

```
from math import sqrt, pi

print(sqrt(25))     # 5.0
print(pi)           # 3.14159...
```

Import everything (not recommended)

```
from math import *
print(sin(pi/2))    # 1.0
```

Warning: This pollutes the namespace and can cause name conflicts.

6.2 Creating your own module

Any `.py` file can be imported as a module. Suppose you have `mymath.py`:

```
# mymath.py
def square(x):
    return x * x

def cube(x):
    return x * x * x
```

In another file:

```
import mymath

print(mymath.square(4))    # 16
```

6.3 Packages (directories of modules)

A package is a folder with an `__init__.py` file. For example:

```
mypkg/
    __init__.py
    utils.py
    geometry/
        __init__.py
        circle.py
```

You can then import:

```
from mypkg import utils
from mypkg.geometry import circle
```

6.4 The `__name__` variable

Each module has a built-in variable `__name__`. When a file is run directly, `__name__ == "__main__"`. This is useful for code that should only run when executed as a script, not when imported.

```
# myscript.py
def main():
    print("Hello from main")

if __name__ == "__main__":
    main()
```

Tip: This pattern is common in Python programs and avoids executing test code when the file is imported as a module.

Why this matters. When a file is *run directly* (e.g., `python myscript.py`), `__name__` is set to `"__main__"`, so the code under the "main guard" `if __name__ == "__main__":` runs. When the same file is *imported* (e.g., `import myscript`), `__name__` is set to the module name (here `"myscript"`), so that block does *not* run. This lets a module behave differently when executed as a script versus when imported as a library.

Listing 1: Direct run vs import behavior

```
# util.py
def add(a, b):
    return a + b

if __name__ == "__main__":    # runs only when: python util.py
    print("Demo:", add(2, 3))  # -> Demo: 5
```

Listing 2: Using the module from another file

```
# app.py
import util
print(util.add(10, 20))          # -> 30 (no "Demo" printed)
```

Common pitfall: the string must be written exactly as `"__main__"` (no spaces).

7 Object-Oriented Programming (OOP)

Python supports object-oriented programming, but with a simpler and more flexible syntax than Java or C++. Classes bundle data (attributes) and behavior (methods) together. Objects are instances of classes.

7.1 Introduction to classes and objects

Defining a class and creating objects A class is defined with the keyword `class`. The special method `__init__` is the constructor. Every instance method must take `self` as its first parameter, referring to the current object.

```
class Point:
    def __init__(self, x, y):
        self.x = x        # instance attribute
        self.y = y

    def move(self, dx, dy):
        self.x += dx
        self.y += dy

    def display(self):
        print(f"({self.x}, {self.y})")

# create instances
p1 = Point(2, 3)
p2 = Point(0, 0)

p1.display()        # (2, 3)
p2.move(5, -1)
p2.display()        # (5, -1)
```

Key points

- The constructor is always named `__init__`.
- The first parameter `self` refers to the current instance.
- Attributes are created by assigning to `self.something`.
- Methods are defined like normal functions but inside the class.

Java/C++ comparison

- No need to declare fields in advance; they are created dynamically when assigned.
- No `new` keyword: `p1 = Point(2,3)` constructs directly.
- No explicit access modifiers (`public`, `private`); by convention, names starting with `_` are considered internal.

7.2 Instance vs Class Variables

In Python, attributes can belong either to each individual object (**instance variables**) or to the class itself (**class variables**).

Instance variables Defined inside `__init__` (or later) and unique to each object.

```
class Point:
    def __init__(self, x, y):
        self.x = x          # instance variable
        self.y = y

p1 = Point(1, 2)
p2 = Point(3, 4)

p1.x = 10
print(p1.x, p2.x)    # 10 3  (different values)
```

Class variables Defined directly in the class body, shared by all instances.

```
class Counter:
    count = 0    # class variable

    def __init__(self):
        Counter.count += 1

c1 = Counter()
c2 = Counter()
print(Counter.count)    # 2
```

Important notes

- Use `self.var` for per-object state.
- Use class variables for data shared across all objects (e.g., counters, constants).
- Class variables can be accessed as `ClassName.var` or via any instance, but changing them via an instance creates a new instance variable.

Example: pitfall

```
class Example:
    shared = []         # class variable

e1 = Example()
e2 = Example()
e1.shared.append(1)
print(e2.shared)        # [1] (same list for all instances!)
```

Lesson: for mutable defaults, use instance variables instead:

```
class Example:
    def __init__(self):
        self.data = []    # unique list per instance
```

Java/C++ note

- Instance variables ≈ fields in C++/Java.
- Class variables ≈ `static` fields in Java/C++.

7.3 Methods

Python supports three kinds of methods in classes: **instance methods**, **class methods**, and **static methods**. They differ in what the first argument represents.

Instance methods (default) The most common kind of method. They take `self` as the first parameter, which refers to the current object.

```
class Circle:
    def __init__(self, r):
        self.r = r

    def area(self):                 # instance method
        return 3.14 * self.r**2

c = Circle(5)
print(c.area())     # 78.5
```

Class methods Declared with `@classmethod` decorator. They take `cls` (the class itself) as the first parameter. Useful for alternative constructors or operations that apply to the whole class.

```
class Person:
    population = 0

    def __init__(self, name):
        self.name = name
        Person.population += 1

    @classmethod
    def how_many(cls):
        return cls.population

p1 = Person("Alice")
p2 = Person("Bob")
print(Person.how_many())    # 2
```

Static methods Declared with `@staticmethod` decorator. They do not receive `self` or `cls`. They are like normal functions, but placed inside a class for organization.

```
class Math:
    @staticmethod
    def add(a, b):
        return a + b

```

```
print(Math.add(2, 3))    # 5
```

Summary

- **Instance methods:** operate on object data (`self`).
- **Class methods:** operate on class-level data (`cls`).
- **Static methods:** do not use `self` or `cls`, just grouped in the class.

Java/C++ note

- Instance methods ≈ regular methods.
- Class methods ≈ `static` methods with class access.
- Static methods ≈ `static` methods in Java/C++, but Python makes the distinction explicit with decorators.

7.4 Special Methods (dunder methods)

Python classes can define **special methods** (also called *dunder methods*, from "double underscore") to customize how objects behave with built-in operations. This allows user-defined classes to act like built-in types.

String representations: `__str__` and `__repr__`

```
class Vector:
    def __init__(self, x, y):
        self.x, self.y = x, y

    def __str__(self):
        return f"({self.x}, {self.y})"      # user-friendly

    def __repr__(self):
        return f"Vector({self.x}, {self.y})"  # for debugging

v = Vector(1, 2)
print(v)        # (1, 2)    -> __str__
print([v])      # [Vector(1, 2)] -> __repr__
```

Operator overloading You can redefine how operators work on objects.

```
class Vector:
    def __init__(self, x, y):
        self.x, self.y = x, y

    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)

    def __eq__(self, other):
        return self.x == other.x and self.y == other.y
```

```
10
11 v1 = Vector(1, 2)
12 v2 = Vector(3, 4)
13 print(v1 + v2)        # (4, 6)
14 print(v1 == v2)       # False
```

Container protocols Implement methods to behave like lists/dicts:

```
1  class MyList:
2      def __init__(self, data):
3          self.data = data
4
5      def __len__(self):
6          return len(self.data)
7
8      def __getitem__(self, idx):
9          return self.data[idx]
10
11 nums = MyList([10,20,30])
12 print(len(nums))     # 3
13 print(nums[1])       # 20
```

Iteration protocol Make an object iterable by defining `__iter__`.

```
1  class Countdown:
2      def __init__(self, n):
3          self.n = n
4
5      def __iter__(self):
6          while self.n > 0:
7              yield self.n
8              self.n -= 1
9
10 for x in Countdown(3):
11     print(x)     # 3 2 1
```

Common special methods

- `__init__` – constructor
- `__del__` – destructor (rarely used)
- `__str__`, `__repr__` – string representations
- `__len__`, `__getitem__`, `__setitem__` – container behavior
- `__iter__`, `__next__` – iteration
- `__add__`, `__sub__`, `__eq__`, `__lt__`, etc. – operators

Java/C++ note Java does not support user-defined operator overloading (aside from `+` for `String` concatenation).
C++ explicitly supports operator overloading with some constraints.
Python achieves similar flexibility via dunder methods (e.g., `__add__`, `__eq__`), making custom classes feel "built-in".

7.5 Encapsulation

Encapsulation is the principle of hiding internal details of a class and exposing only what is necessary. In Python, access modifiers (`public`, `private`, `protected`) do not exist like in Java/C++. Instead, Python relies on **naming conventions** and special features like `@property`.

Naming conventions

- `public_name` – intended for external use.
- `_internal_name` – convention: "protected", intended for internal use only.
- `__private_name` – triggers **name mangling**, making it harder to access from outside.

```
class Example:
    def __init__(self):
        self.public = 1
        self._protected = 2
        self.__private = 3

obj = Example()
print(obj.public)        # 1
print(obj._protected)    # 2 (convention: don't use outside)
print(obj.__private)     # AttributeError
print(obj._Example__private)  # 3 (name mangling)
```

Getters and setters (Java/C++ style) In Java/C++ you often write explicit getter/setter methods. In Python this is rarely done manually:

```
class Person:
    def __init__(self, name):
        self._name = name

    def get_name(self):
        return self._name

    def set_name(self, value):
        self._name = value

p = Person("Alice")
print(p.get_name())
p.set_name("Bob")
```

The Pythonic way: `@property` Python provides `@property` to expose methods as if they were attributes, keeping syntax clean while allowing encapsulation.

```
class Person:
    def __init__(self, name):
        self._name = name

    @property
    def name(self):           # getter
        return self._name

    @name.setter
    def name(self, value):    # setter
        if not value:
            raise ValueError("Name cannot be empty")
        self._name = value

p = Person("Alice")
print(p.name)       # calls getter
p.name = "Bob"      # calls setter
```

Advantages of @property

- You can start with a simple attribute, later turn it into a property without changing client code.
- Properties allow validation, computed values, or lazy evaluation.

Java/C++ note

- Java/C++ enforce access with keywords `private`, `protected`, `public`.
- Python trusts the programmer: conventions (`_var`) are not enforced by the language.
- Properties are closer to C# `get`/`set` syntax than to Java boilerplate.

7.6 Inheritance

Inheritance allows defining new classes based on existing ones. The child class (subclass) inherits attributes and methods from the parent class (superclass), and can override or extend them.

Basic inheritance

```
class Animal:
    def speak(self):
        print("Some sound")

class Dog(Animal):              # Dog inherits from Animal
    def speak(self):            # override
        print("Woof!")

class Cat(Animal):
```

```
    def speak(self):
        print("Meow!")

a = Animal()
d = Dog()
c = Cat()

a.speak()    # Some sound
d.speak()    # Woof!
c.speak()    # Meow!
```

Constructor inheritance If the parent defines `__init__`, the child can call it using `super()`.

```
class Person:
    def __init__(self, name):
        self.name = name

class Student(Person):
    def __init__(self, name, major):
        super().__init__(name)      # call parent constructor
        self.major = major

s = Student("Alice", "CS")
print(s.name, s.major)    # Alice CS
```

Overriding methods Subclasses can redefine methods. To extend rather than replace, call `super()` inside the override.

```
class Logger:
    def log(self, msg):
        print("LOG:", msg)

class TimestampLogger(Logger):
    def log(self, msg):
        from datetime import datetime
        now = datetime.now().strftime("%H:%M:%S")
        super().log(f"[{now}] {msg}")    # call parent log

t = TimestampLogger()
t.log("Hello")
# LOG: [12:34:56] Hello
```

The `isinstance` and `issubclass` functions

```
print(isinstance(s, Student))    # True
print(isinstance(s, Person))     # True
print(issubclass(Student, Person))   # True
```

Java/C++ note

- Like Java/C++, Python supports single inheritance directly.
- Unlike Java, multiple inheritance is allowed (see next section).
- No need for explicit `virtual` keyword: methods are always virtual and can be overridden.

7.7 Exercises (OOP)

1. **Basic class**
   (a) Define a class `Point` with attributes `x`, `y` and a method `move(dx, dy)`.
   (b) Create two points and move them.
2. **Instance vs class variables**
   (a) Define a class `Counter` with a class variable `count` that tracks how many instances have been created.
   (b) Create several objects and print `Counter.count`.
3. **Methods**
   (a) Write a class `Circle` with a constructor that takes a radius.
   (b) Add an instance method `area()` and a static method `pi()` returning 3.14159.
   (c) Test them both.
4. **Special methods**
   (a) Implement a class `Vector(x,y)` with `__str__` to print like `(x, y)`.
   (b) Overload the `+` operator (`__add__`) to add two vectors.
5. **Encapsulation**
   (a) Write a class `Person` with a private attribute `_name`.
   (b) Use `@property` and `@name.setter` to control access.
6. **Inheritance**
   (a) Create a base class `Shape` with a method `area()` that raises `NotImplementedError`.
   (b) Derive classes `Circle` and `Rectangle` that override `area()`.
   (c) Test them with a list of shapes and a loop.

Part II
NumPy Essentials

8 Introduction to NumPy

NumPy (Numerical Python) is the core library for scientific and numerical computing in Python. It provides the `ndarray` object, a powerful, memory-efficient multidimensional array, together with optimized operations written in C. While Python lists are flexible and general-purpose, they are too slow for large-scale numerical tasks. NumPy solves this by offering a compact representation of data and vectorized computations.

8.1 Motivation

Why do we need NumPy in a course on Numerical Methods?

- **Python lists are slow:** They store references to objects, not raw numbers. Iterating element by element in Python is inefficient for large datasets.
- **NumPy arrays are fast:** They are implemented as contiguous blocks of homogeneous data, allowing the use of optimized low-level routines.
- **Vectorization:** Instead of explicit loops, operations apply to entire arrays at once.
- **Ecosystem:** NumPy is the foundation for higher-level libraries such as SciPy, Pandas, TensorFlow, and PyTorch.

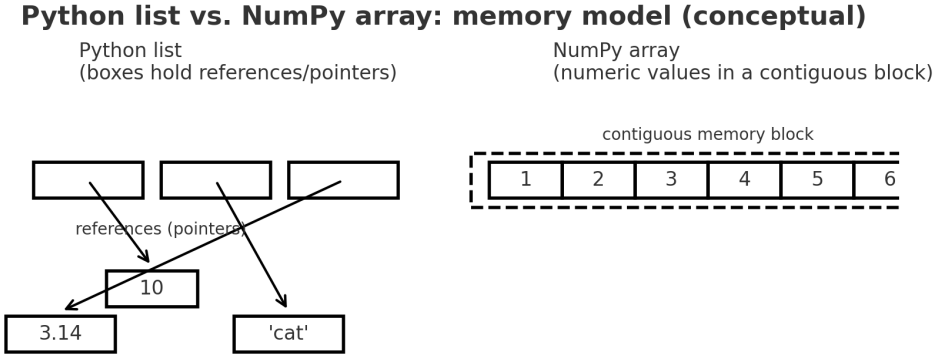


Figure 1: Python list vs. NumPy array: memory model (conceptual). Lists store references to heterogeneous objects, while NumPy arrays store homogeneous values in a contiguous memory block, enabling fast vectorized operations.

8.2 The `ndarray` Object

The central feature of NumPy is the `ndarray` (N-dimensional array). It is similar to arrays in C or matrices in MATLAB, but much more flexible.

Important attributes:

- `ndim` – number of dimensions (axes).
- `shape` – tuple with the size of each dimension.
- `dtype` – data type of elements (e.g., `int32`, `float64`).

```
import numpy as np

a = np.array([[1, 2, 3], [4, 5, 6]])
print(a.ndim)   # 2 (matrix has 2 dimensions)
print(a.shape)  # (2, 3) => 2 rows, 3 columns
print(a.dtype)  # int64 (on most systems)
```

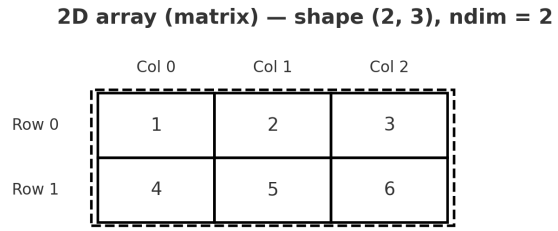


Figure 2: A 2D NumPy array (matrix) with shape `(2,3)` and `ndim = 2`. Rows and columns are indexed starting from 0.

8.3 Getting NumPy

If you are using CodeExpert, NumPy is already available. Otherwise, it can be installed via pip:

```
pip install numpy
```

By convention, NumPy is always imported under the alias `np`:

```
import numpy as np
```

This alias is universal in the scientific Python community and should always be used.

8.4 First Example

Let us compare a Python list with a NumPy array:

```python
lst = [1, 2, 3, 4]
print(lst * 2)
# Output: [1, 2, 3, 4, 1, 2, 3, 4]   (list repetition)

arr = np.array([1, 2, 3, 4])
print(arr * 2)
# Output: [2 4 6 8]   (element-wise multiplication)
```

Operation	Result
Python list * 2	[1, 2, 3, 4, 1, 2, 3, 4]
NumPy array * 2	[2, 4, 6, 8]

Figure 3: Multiplying a Python list versus a NumPy array. The list repeats its elements, while the array performs element-wise multiplication.

8.5 Why is NumPy Fast?

NumPy's performance advantage comes from two main reasons:

1. **Vectorization:** Array operations are implemented in optimized C code. Instead of looping in Python, the computation is executed at low level.
2. **Memory locality:** Elements are stored contiguously in memory, making access patterns efficient for the CPU cache.

Illustration:

```python
# Python loop (slow) -  0.04494023323059082 s
squares = [x*2 for x in range(1_000_000)]

# NumPy vectorized (fast) - 0.0020804405212402344 s
arr = np.arange(1_000_000)
squares_np = arr*2
```

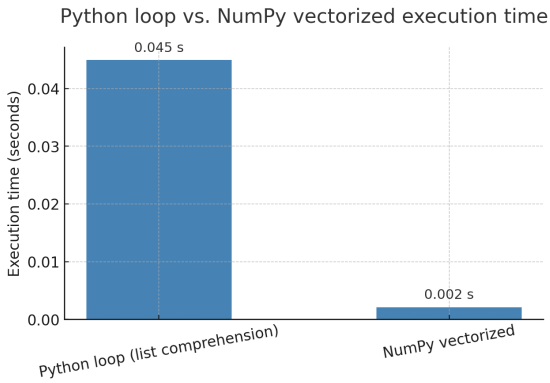


Figure 4: Execution time comparison between a Python loop and a NumPy vectorized operation. Vectorization in NumPy executes operations in optimized C code, leading to significant performance improvements.

8.6 Summary

NumPy enables us to write code that is:

- **Concise:** few lines instead of long loops,
- **Readable:** closer to mathematical notation,
- **Efficient:** executed by optimized C/Fortran routines,
- **Foundational:** forms the basis of the entire scientific Python ecosystem.

This makes NumPy the natural choice for studying and implementing numerical methods in computer science.

9 Constructing Arrays

Creating arrays is the fundamental starting point in NumPy. There are multiple ways to create arrays, depending on the use case: from existing Python objects, by using NumPy's built-in constructors, by generating numerical ranges, or by sampling random values.

9.1 From Python Sequences

The most direct way to build a NumPy array is from a Python list or tuple using the function `np.array`.

```
import numpy as np

# From a list
a = np.array([1, 2, 3, 4])

# From a tuple
b = np.array((5, 6, 7, 8))

print(type(a))  # <class 'numpy.ndarray'>
print(a)        # [1 2 3 4]
```

Unlike Python lists, NumPy arrays:

- contain elements of the *same data type*,
- are stored in contiguous memory,
- support element-wise operations efficiently.

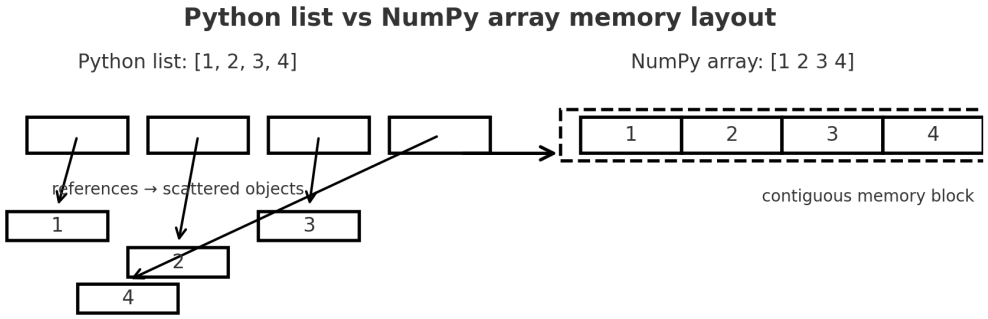


Figure 5: Creating a NumPy array from a Python list. A list stores references to objects scattered in memory, while a NumPy array stores values in a contiguous block.

9.2 Predefined Arrays

NumPy provides several constructors for common arrays:

```
zeros = np.zeros((2,3))    # 2x3 matrix of zeros
ones  = np.ones((3,3))     # 3x3 matrix of ones
full  = np.full((2,2), 7)  # 2x2 matrix filled with 7
empty = np.empty((2,2))    # uninitialized values
```

Notes:

- `np.zeros(shape)` creates an array filled with 0.
- `np.ones(shape)` creates an array filled with 1.
- `np.full(shape, value)` creates an array with a constant value.
- `np.empty(shape)` allocates memory without initialization (values will be arbitrary, depending on memory state).

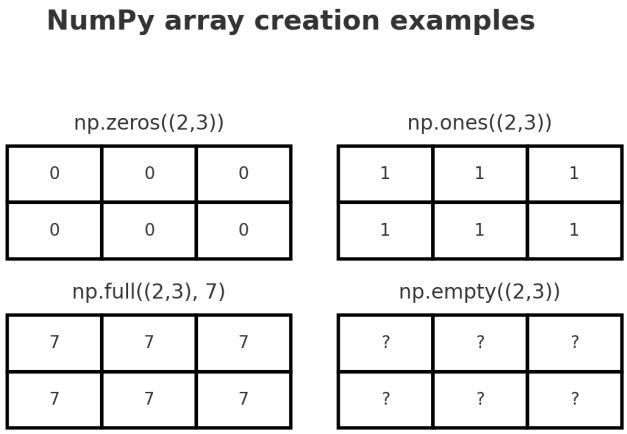


Figure 6: Examples of array constructors in NumPy. Functions like `np.zeros`, `np.ones`, `np.full`, and `np.empty` provide quick ways to generate arrays of a given shape with predefined values.

9.3 Identity and Diagonal Arrays

Special square matrices are often needed:

```
I = np.eye(3)      # identity matrix
D = np.diag([1, 2, 3])  # diagonal matrix
```

These arrays are useful in linear algebra and appear frequently in numerical algorithms.

9.4 Numerical Ranges

For evenly spaced sequences, NumPy provides `arange`, `linspace`, and `logspace`.

```
a = np.arange(0, 10, 2)   # [0 2 4 6 8]
b = np.linspace(0, 1, 5)  # [0.   0.25 0.5 0.75 1. ]
c = np.logspace(0, 2, 5)  # [  1.   10.  100.]
```

- `np.arange(start, stop, step)` works like Python's range but returns an array.
- `np.linspace(start, stop, num)` generates exactly `num` points between the bounds.

- `np.logspace(start, stop, num)` generates values evenly spaced on a logarithmic scale.

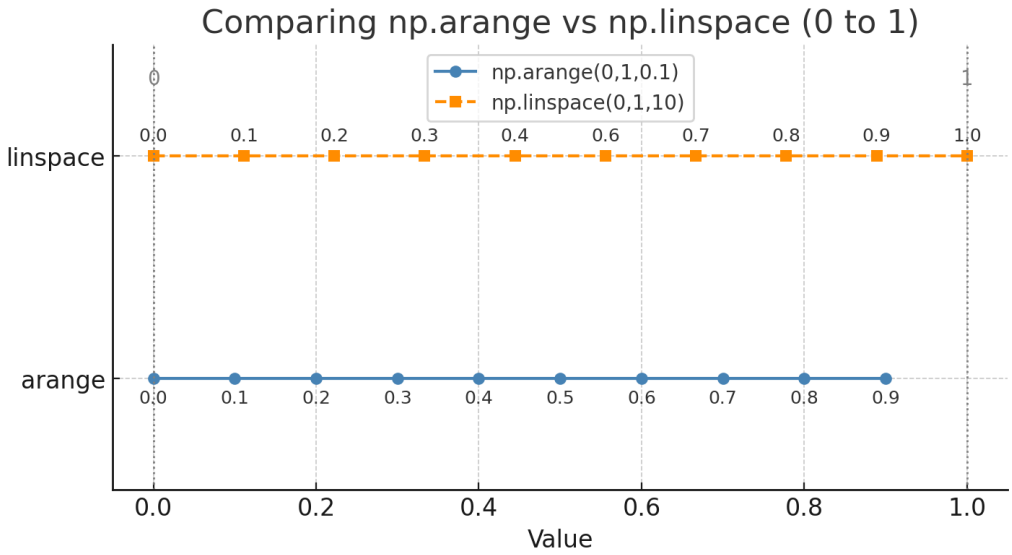


Figure 7: Comparison of `np.arange` and `np.linspace` between 0 and 1. `np.arange(0,1,0.1)` generates values with a fixed step, while `np.linspace(0,1,10)` generates exactly 10 equally spaced values.

9.5 Random Arrays

Random numbers are essential in simulations, Monte Carlo methods, and testing:

```
rand = np.random.rand(2,2)         # uniform in [0,1)
randn = np.random.randn(2,2)       # standard normal distribution
ints = np.random.randint(0,10,(2,3))  # random integers between 0 and 9
```

- `np.random.rand` draws from uniform distribution.
- `np.random.randn` draws from normal (Gaussian) distribution.
- `np.random.randint(low, high, size)` draws random integers.

A note on the normal distribution. The function `np.random.randn` draws values from the *standard normal distribution*, also known as the *Gaussian distribution*. It is the familiar "bell-shaped curve" that appears in many natural and social phenomena, centered at 0 with most values close to the mean.

If you are not familiar with the normal distribution, I recommend watching the excellent visual explanation by 3Blue1Brown: `https://www.youtube.com/watch?v=zeJD6dqJ5lo` (not fully relevant for this course, but really important in practice)

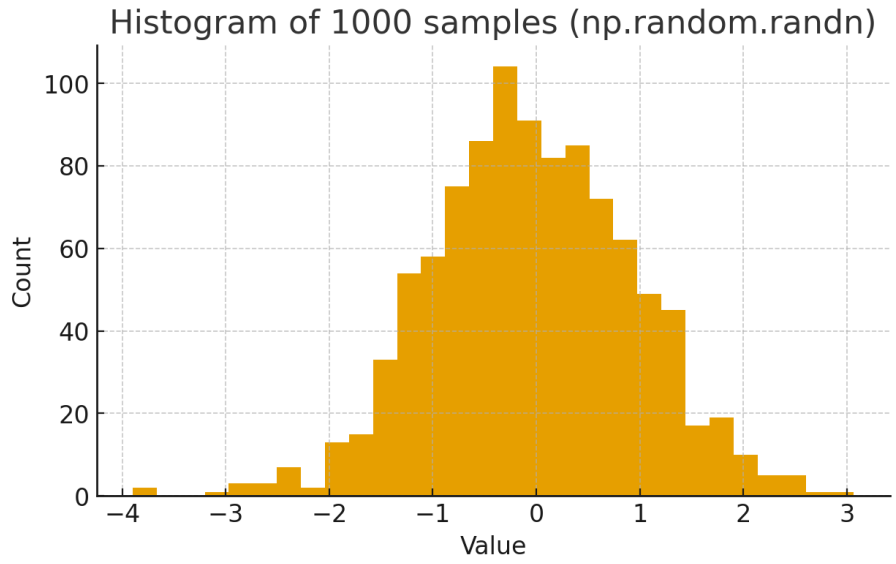


Figure 8: Histogram of 1000 samples generated with `np.random.randn`. The values follow the standard normal distribution with mean 0 and standard deviation 1.

9.6 Summary

NumPy provides many ways to construct arrays:

- From existing Python lists or tuples,
- Using constructors like `zeros`, `ones`, `full`, or `empty`,
- Generating special matrices like identities or diagonals,
- With evenly spaced sequences (`arange`, `linspace`, `logspace`),
- By sampling random numbers.

This flexibility makes NumPy arrays the natural building blocks for implementing numerical methods.

10 Inspecting Arrays

After creating arrays, one of the first tasks is to inspect their structure. This means checking how many dimensions they have, how large they are, what kind of data they store, and how we can reshape them for further computations. In numerical methods, being able to quickly understand the structure of an array is crucial for debugging and designing efficient algorithms.

10.1 Basic Attributes of an Array

Every NumPy array has several important attributes that describe its structure:

- `ndim` – number of dimensions (also called axes). A scalar has `ndim=0`, a vector has `ndim=1`, a matrix has `ndim=2`, and so on.
- `shape` – tuple giving the length along each dimension. For example, a 2×3 matrix has shape `(2,3)`.
- `size` – total number of elements (product of all shape entries).
- `dtype` – data type of elements (integers, floats, booleans, etc.).

```
import numpy as np

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

print(a.ndim)   # 2 (matrix has two dimensions)
print(a.shape)  # (2, 3) -> 2 rows, 3 columns
print(a.size)   # 6 (total elements)
print(a.dtype)  # int64 (on most systems)
```

Example: dimensions in practice.

```
scalar = np.array(42)
vector = np.array([1,2,3])
matrix = np.array([[1,2],[3,4],[5,6]])

print(scalar.ndim, scalar.shape)  # 0, ()
print(vector.ndim, vector.shape)  # 1, (3,)
print(matrix.ndim, matrix.shape)  # 2, (3,2)
```

—

10.2 Understanding Data Types

A key difference between Python lists and NumPy arrays is that arrays are *homogeneous* — all elements have the same type. This makes storage and computation much more efficient.

```
a = np.array([1, 2, 3], dtype=np.int32)
b = np.array([1.0, 2.0, 3.0])   # float64 by default

```

```
print(a.dtype)   # int32
print(b.dtype)   # float64
```

The method `astype()` can be used to explicitly convert types:

```
c = b.astype(np.int32)
print(c)         # [1 2 3]
print(c.dtype)   # int32
```

dtype	Description	Size (bytes)
int8	8-bit integer	1
int16	16-bit integer	2
int32	32-bit integer	4
int64	64-bit integer	8
uint8	8-bit unsigned integer	1
uint16	16-bit unsigned integer	2
uint32	32-bit unsigned integer	4
uint64	64-bit unsigned integer	8
float16	16-bit floating point	2
float32	32-bit floating point	4
float64	64-bit floating point	8
bool	Boolean (True/False)	1
complex64	Complex number (2×32-bit floats)	8
complex128	Complex number (2×64-bit floats)	16

Figure 9: Common NumPy `dtype` values, their description, and memory size.

10.3 Reshaping Arrays

NumPy allows us to reorganize the shape of arrays as long as the total number of elements stays the same.

```
a = np.arange(6)
print(a)         # [0 1 2 3 4 5]

b = a.reshape((2,3))
print(b)
# [[0 1 2]
#  [3 4 5]]

c = a.reshape((3,2))
print(c)
# [[0 1]
#  [2 3]
#  [4 5]]
```

If we are not sure about one of the dimensions, we can use `-1` and NumPy will infer it automatically:

```
d = a.reshape((-1, 2))    # 6 elements, grouped into 2 per row
print(d.shape)  # (3,2)
```

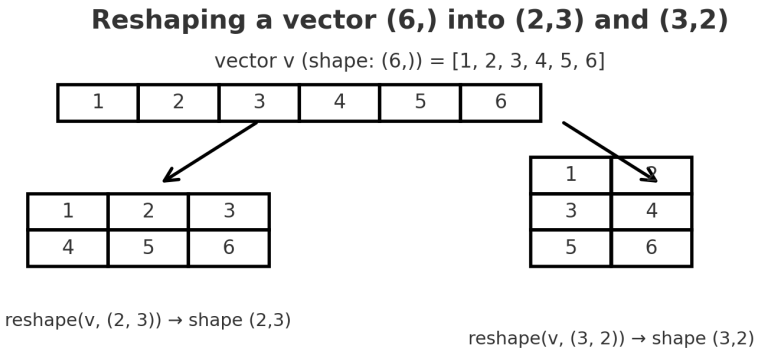


Figure 10: Reshaping a vector of shape `(6,)` into two different matrices. The same data can be reorganized into shape `(2,3)` or shape `(3,2)` without changing the underlying elements.

10.4 Flattening Arrays

Sometimes, we want to reduce a multi-dimensional array into a one-dimensional vector.

- `flatten()` – returns a new copy of the array.
- `ravel()` – returns a view when possible (faster, linked to original data).

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

flat1 = b.flatten()
print(flat1)    # [1 2 3 4 5 6]

flat2 = b.ravel()
print(flat2)    # [1 2 3 4 5 6]
```

If we modify the result of `ravel`, the original array is also modified (because they share memory), but this is not the case for `flatten`.

```
r = b.ravel()
r[0] = 99
print(b)
# [[99  2  3]
#  [ 4  5  6]]
```

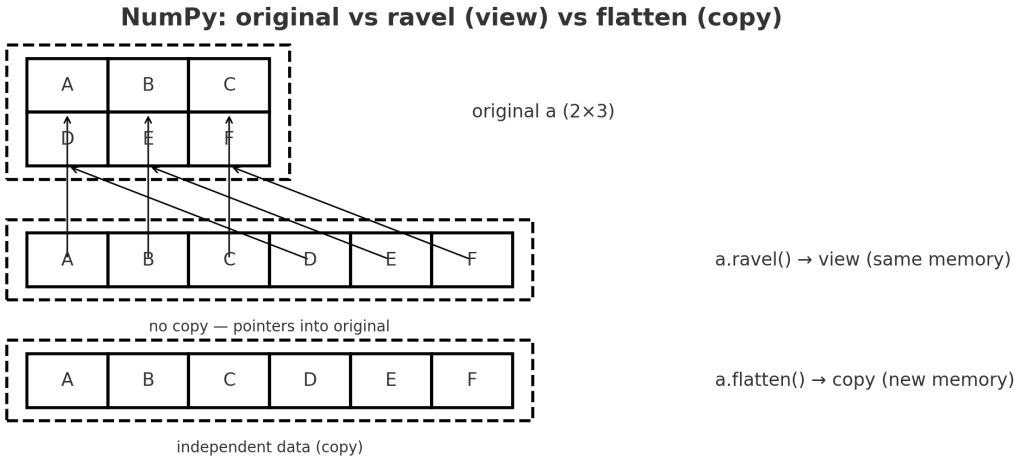


Figure 11: Flattening a 2D array into 1D. `a.ravel()` returns a view into the original array (no copy, shared memory), while `a.flatten()` returns a completely new copy of the data.

11 Indexing and Slicing Arrays

Indexing and slicing are the tools we use to access, select, and modify specific parts of arrays. They are essential in numerical methods, because real computations rarely use entire arrays at once — instead, we focus on sub-arrays, rows, columns, or even individual elements.

In NumPy, indexing is richer than in Python lists. We start with the basics and gradually move to more advanced forms.

11.1 Basic Indexing in 1D Arrays

Like Python lists, NumPy arrays are zero-indexed.

```
import numpy as np

a = np.array([10, 20, 30, 40])

print(a[0])    # 10  (first element)
print(a[2])    # 30  (third element)

a[1] = 99      # modify element at index 1
print(a)       # [10 99 30 40]
```

Notes:

- Index `0` refers to the first element.
- Negative indices count from the end: `a[-1]` is the last element.

```
print(a[-1])    # 40 (last element)
print(a[-2])    # 30 (second to last)
```

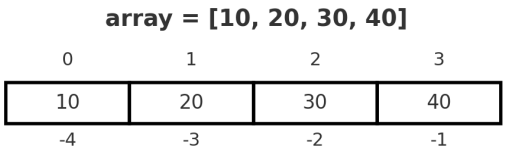


Figure 12: Indexing in a 1D NumPy array. Positive indices (0,1,2,3) count from the start, while negative indices (-4,-3,-2,-1) count from the end.

11.2 Indexing in 2D Arrays

For 2D arrays (matrices), we need two indices: one for rows and one for columns.

```
b = np.array([[1,2,3],
              [4,5,6],
              [7,8,9]])

print(b[0,0])   # 1 (row 0, col 0)
print(b[1,2])   # 6 (row 1, col 2)
print(b[-1,-1]) # 9 (last row, last col)
```

- First index = row - Second index = column
This is the same convention used in mathematics: $b_{i,j}$ refers to row i, column j.
—

11.3 Slicing in 1D Arrays

A slice extracts a portion of the array. The general form is:

`a[start:stop:step]`

- Includes `start`, excludes `stop` (half-open interval). - `step` defaults to 1. - Missing values default to array boundaries.

```
a = np.arange(10)  # [0 1 2 3 4 5 6 7 8 9]

print(a[2:7])    # [2 3 4 5 6]
print(a[:5])     # [0 1 2 3 4]
print(a[::2])    # [0 2 4 6 8]
print(a[::-1])   # [9 8 7 6 5 4 3 2 1 0]
```

11.4 Slicing in 2D Arrays

In multiple dimensions, we provide slices per axis.

```
b = np.array([[1,2,3],
              [4,5,6],
              [7,8,9]])

print(b[:2, 1:])
# [[2 3]
#  [5 6]]
```

Explanation: - Rows: `:2` → take rows 0 and 1 - Columns: `1:` → take columns 1 and onward
The result is a 2×2 submatrix.

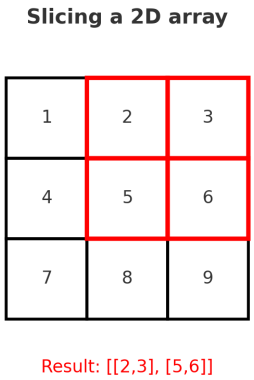


Figure 13: Slicing a 2D array. Using `b[:2, 1:]` selects the first two rows and the last two columns, giving the submatrix `[[2,3],[5,6]]`.

11.5 Slicing with Steps

Slices can also include steps.

```
c = np.arange(16).reshape(4,4)
print(c)
# [[ 0  1  2  3]
#  [ 4  5  6  7]
#  [ 8  9 10 11]
#  [12 13 14 15]]

print(c[::2, ::2])
# [[ 0  2]
#  [ 8 10]]
```

Here, every second row and every second column is taken.

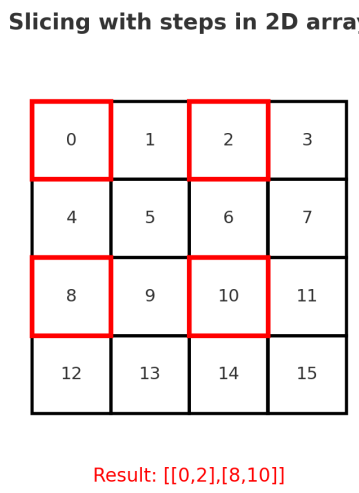


Figure 14: Slicing with steps in a 2D array. Using `c[::2, ::2]` selects every second row and every second column, resulting in the submatrix `[[0,2],[8,10]]`.

11.6 Advanced Indexing: Lists of Indices

NumPy allows selecting arbitrary elements by providing lists (or arrays) of indices.

```
a = np.array([10,20,30,40,50])
idx = [0, 2, 4]
print(a[idx])   # [10 30 50]
```

We can also use this in multiple dimensions:

```
b = np.array([[1,2],[3,4],[5,6]])
print(b[[0,2], [1,0]])  # [2 5]
# Explanation:
# take (row 0,col 1) and (row 2,col 0)
```

11.7 Boolean Indexing

A very powerful feature: arrays can be indexed with boolean masks.

```
a = np.arange(10)
mask = (a % 2 == 0)  # even numbers
print(mask)
# [ True False  True False  True False  True False  True False]
print(a[mask])
# [0 2 4 6 8]
```

This is essential in data analysis, because it allows conditions like "select all values greater than 5".

```
print(a[a > 5])    # [6 7 8 9]
```

11.8 Views vs Copies in Indexing and Slicing

Important: in NumPy, most slicing operations return a *view*, not a copy. This means they share the same memory. Modifying the view modifies the original array.

```
a = np.arange(10)
b = a[2:5]    # view
b[:] = 99
print(a)
# [0 1 99 99 99 5 6 7 8 9]
```

If we want an independent array, we must call `copy()`:

```
c = a[2:5].copy()
c[:] = -1
print(a)  # original unchanged
print(c)  # [-1 -1 -1]
```

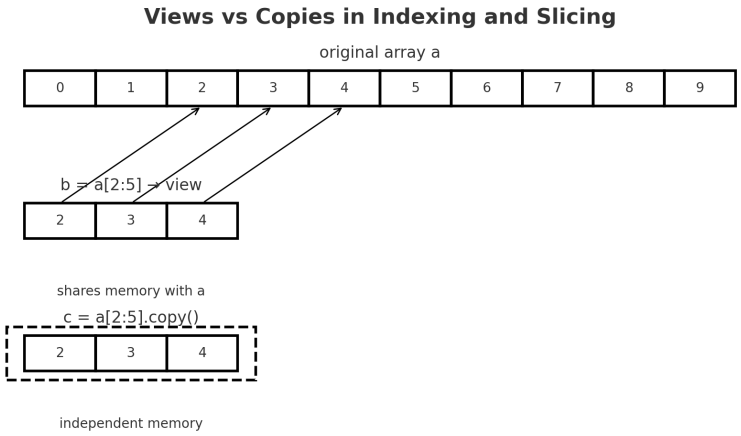


Figure 15: Views vs Copies in NumPy slicing. A slice like `a[2:5]` produces a view that shares memory with the original array. Using `copy()` creates a new independent block of memory.

12 Array Operations

NumPy arrays support fast, vectorized operations that apply to whole arrays at once. This section introduces element-wise arithmetic, broadcasting (how shapes align), universal functions ("ufuncs"), matrix multiplication, reductions (e.g. sums, means), comparisons and logical operations, and practical tips for numerical stability.

12.1 Element-wise Arithmetic

Unless stated otherwise, arithmetic operators apply element by element.

```
import numpy as np

a = np.array([1, 2, 3, 4])
b = np.array([10, 20, 30, 40])

print(a + b)    # [11 22 33 44]
print(a - b)    # [-9 -18 -27 -36]
print(a * b)    # [10 40 90 160]
print(b / a)    # [10. 10. 10. 10.]
print(a ** 2)   # [ 1  4  9 16]
```

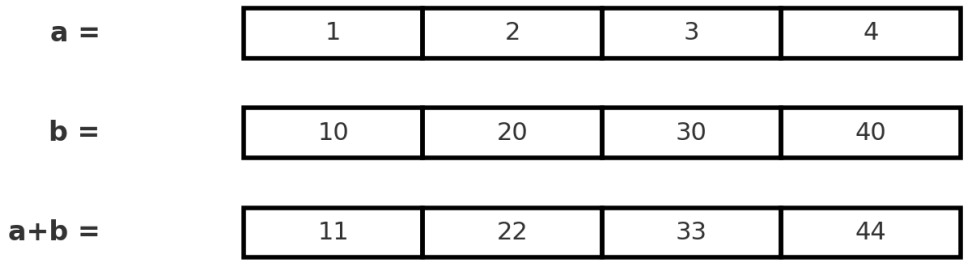


Figure 16: Element-wise addition in NumPy. Given $a = [1, 2, 3, 4]$ and $b = [10, 20, 30, 40]$, the result of $a + b$ is $[11, 22, 33, 44]$.

Type promotion. NumPy chooses a result dtype that can hold all inputs: adding `int32` and `float64` promotes to `float64`. Use `astype` if you need a specific dtype.

```
x = np.array([1, 2, 3], dtype=np.int32)
y = np.array([0.5, 1.5, 2.5])      # float64
z = x + y
print(z.dtype)  # float64
```

12.2 Scalar Operations and Broadcasting (Preview)

Operations with scalars are broadcast to every element:

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

print(a + 5)  # [6 7 8 9]
print(2 * a)  # [2 4 6 8]
```

Pitfall (Python lists). `[1,2,3] * 2` repeats the list, but `np.array([1,2,3]) * 2` multiplies element-wise.

12.3 Broadcasting: The Full Rules

Broadcasting lets NumPy perform element-wise operations on arrays of different shapes *without copying data*. Shapes are compared *from right to left*; two dimensions are compatible if they are equal or one of them is 1. The result shape is the element-wise maximum.

Operands	Compatible?	Result shape
$(3, 1)$ and $(1, 4)$	yes	$(3, 4)$
$(5, 1, 7)$ and $(1, 3, 1)$	yes	$(5, 3, 7)$
$(3,)$ and $(3, 1)$	no (compare from right: 1 vs $\varnothing$)	–
$(8, 1, 6, 1)$ and $(7, 1, 5)$	yes	$(8, 7, 6, 5)$

```
# Row vector (1, 4) + column vector (3, 1) -> (3, 4)
row = np.array([[1, 2, 3, 4]])   # shape (1, 4)
col = np.array([[10], [20], [30]])  # shape (3, 1)
print((col + row).shape)  # (3, 4)
print(col + row)
```

Making row/column vectors explicitly. Use `np.newaxis` (alias `None`) to add a length-1 dimension:

```
v = np.array([1, 2, 3, 4])  # shape (4,)
row = v[np.newaxis, :]      # (1,4)
col = v[:, np.newaxis]      # (4,1)
```

Common error. `ValueError: operands could not be broadcast together` means some pair of dimensions (right-aligned) are neither equal nor 1.

12.4 Universal Functions (ufuncs)

Ufuncs are fast vectorized functions in C that operate element-wise. They work on scalars or arrays, broadcast automatically, and support extra features like the `where` mask and `out` parameter.

```
x = np.linspace(0, np.pi, 5)
print(np.sin(x))            # element-wise sine
print(np.exp(x) - 1)        # expm1 (see numerical tips below)
```

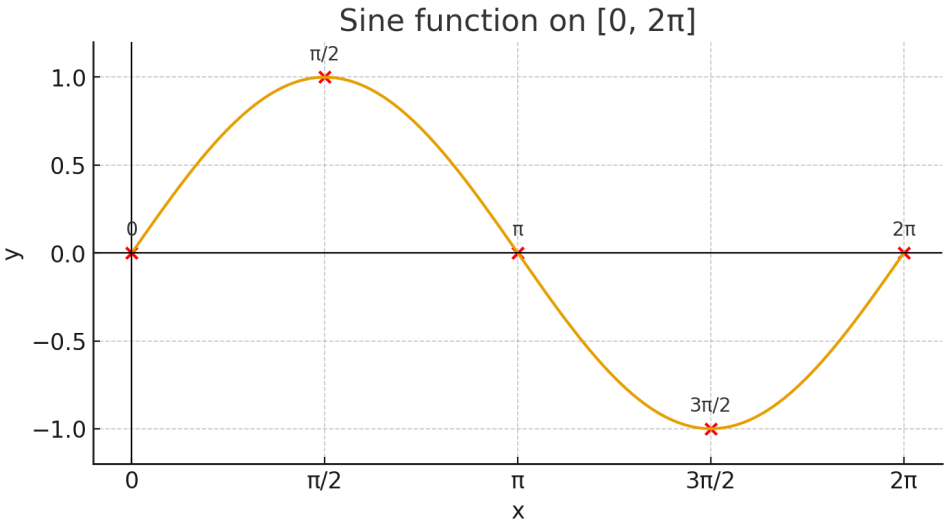


Figure 17: The sine function on the interval $[0, 2\pi]$. NumPy universal functions (ufuncs) such as `np.sin` apply element-wise to arrays, producing fast vectorized computations.

Binary ufuncs. Many ops have ufunc forms (e.g. `np.add`, `np.multiply`), and expose reductions/accumulations:

```
a = np.array([1,2,3,4])
print(np.add.reduce(a))      # 10   (same as a.sum())
print(np.add.accumulate(a)) # [1 3 6 10]  (prefix sums)
```

`where` and `out`. Apply a ufunc conditionally and/or write directly into a preallocated array:

```
x = np.linspace(-2, 2, 5)
y = np.empty_like(x)
np.square(x, out=y, where=(x >= 0))  # square only non-negatives; others left
print(y)
```

12.5 Comparisons and Logical Operations

Relational operators are element-wise and return boolean arrays (masks):

```
a = np.array([0, 1, 2, 3, 4, 5])
mask = (a % 2 == 0) & (a >= 2)    # use & and | with parentheses (not and/or)
print(mask)        # [False False  True False  True False]
print(a[mask])     # [2 4]
```

Logical reductions condense masks to scalars or to lower-rank arrays:

```
M = np.array([[True, False, True],
              [True, True,  False]])
print(M.any())           # True
print(M.all(axis=0))     # [ True  True False]  (per column)
print(M.any(axis=1))     # [ True  True]        (per row)
```

Useful helpers. `np.where(cond, a, b)`, `np.clip(x, lo, hi)`, `np.maximum`, `np.minimum` all broadcast and are vectorized.

12.6 Reductions and Aggregations

Reductions collapse one or more axes using functions like `sum`, `mean`, `min`, `max`, `std`, `var`, `median`, etc.

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

print(A.sum())           # 21 (all elements)
print(A.sum(axis=0))     # [5 7 9]  (column sums)
print(A.sum(axis=1))     # [6 15]   (row sums)
print(A.mean(keepdims=True))  # [[3.5]] (preserve rank)
```

NaN-aware variants. Use `np.nanmean`, `np.nansum`, etc. to ignore NaNs.

```
x = np.array([1.0, np.nan, 3.0])
print(np.nanmean(x))  # 2.0
```

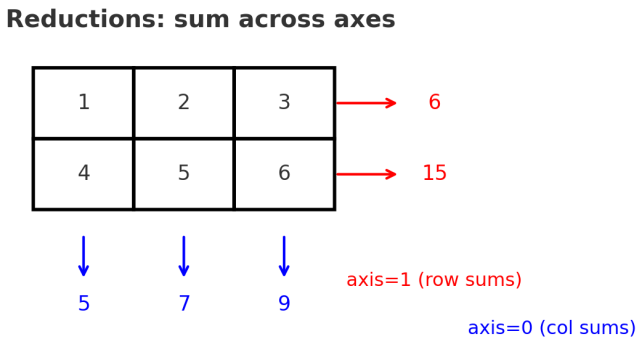


Figure 18: Reductions across axes in NumPy. The operation `a.sum(axis=1)` computes row sums (horizontal reduction, in red), while `a.sum(axis=0)` computes column sums (vertical reduction, in blue).

12.7 Matrix Multiplication vs Element-wise Multiplication

The operator `*` is element-wise; the `@` operator (and `np.matmul`/`np.dot`) performs matrix multiplication (or vector dot product).

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

print(A * B)   # element-wise: [[ 5 12]
               #                [21 32]]

print(A @ B)   # matrix product: [[19 22]
```

```
                #                     [43  50]]

v = np.array([1, 2])     # shape (2,)
print(A @ v)             # [5  11]
```

Row/column vectors explicitly. For 2D semantics, shape your vectors as $(1, n)$ or $(n, 1)$ with `np.newaxis`.

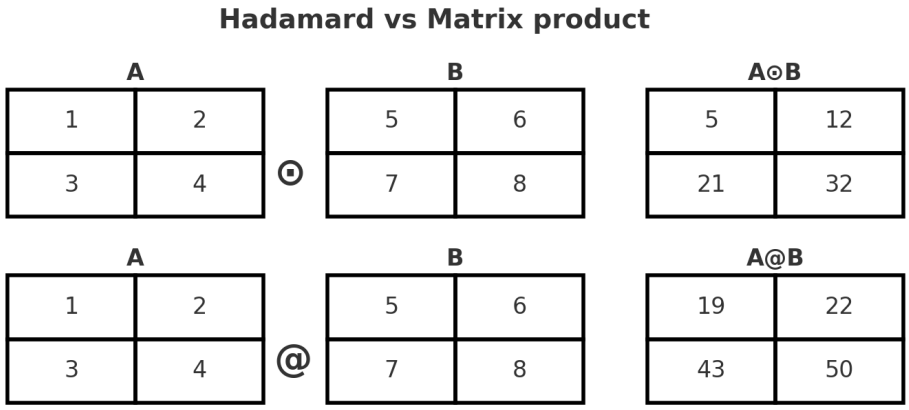


Figure 19: Element-wise (Hadamard) product vs Matrix multiplication. In NumPy, `A * B` multiplies elements position-wise, while `A @ B` or `np.dot(A,B)` performs the matrix product.

12.8 Outer Products, Dot Products, and Norms

```
u = np.array([1, 2, 3])
v = np.array([4, 5])

print(np.outer(u, v))
# [[ 4  5]
#  [ 8 10]
#  [12 15]]

print(np.dot(u, u))          # 1*1 + 2*2 + 3*3 = 14  (scalar)
print(np.linalg.norm(u))     # sqrt(14)
```

12.9 Axis Semantics (Mental Model)

For a 2D array:

- `axis=0` runs *down* the rows (operates across rows; one result per column).
- `axis=1` runs *across* the columns (operates across columns; one result per row).

For higher dimensions, apply the same idea: the axis you specify is the one being reduced.

```python
A = np.arange(24).reshape(2,3,4)  # (depth=2, rows=3, cols=4)
print(A.sum(axis=0).shape)  # (3,4)  (sum over depth)
print(A.sum(axis=1).shape)  # (2,4)  (sum over rows)
print(A.sum(axis=2).shape)  # (2,3)  (sum over cols)
```

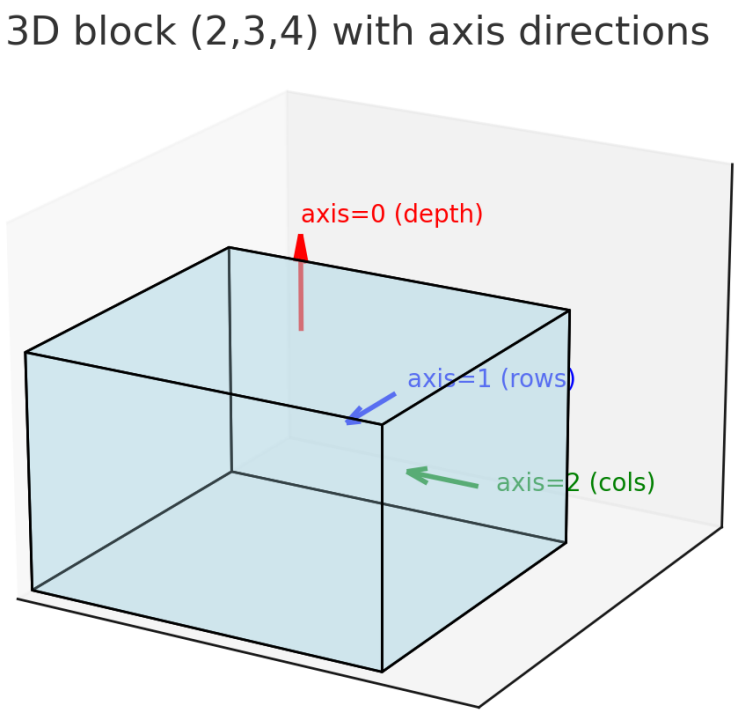


Figure 20: Axes in a 3D NumPy array of shape $(2, 3, 4)$. By convention, `axis=0` (red) corresponds to the depth dimension, `axis=1` (blue) to rows, and `axis=2` (green) to columns.

12.10 Numerical Stability and Practical Tips

Avoid Python loops. Prefer vectorized operations; they are faster and clearer.

Use `where` instead of branching in Python.

```python
x = np.linspace(-5, 5, 11)
y = np.where(x >= 0, np.sqrt(x), 0.0)    # vectorized "if"
```

Stable log/exp patterns. Use `np.log1p(x)` for $\log(1 + x)$ when x is small, and `np.expm1(x)` for $e^x - 1$. For sums of exponentials, subtract the maximum ("log-sum-exp"):

```python
a = np.array([1000.0, 1001.0, 999.0])
m = a.max()
lse = m + np.log(np.sum(np.exp(a - m)))  # stable log-sum-exp
```

Clipping and saturating.

```python
sig = np.linspace(-5, 5, 11)
bounded = np.clip(sig, -1.0, 1.0)    # now in [-1, 1]
```

Do not rely on `np.vectorize` for speed. It is a convenience wrapper (still loops in Python), not a performance tool.

12.11 Worked Examples

Broadcasting a row into all rows of a matrix.

```
M = np.arange(12).reshape(3,4)
r = np.array([10, 20, 30, 40])  # shape (4,)
print(M + r)  # r broadcasts to (3,4)
```

Normalizing rows to unit ℓ_2 norm.

```
X = np.array([[3., 4.],
              [1., 1.],
              [0., 2.]])
row_norms = np.linalg.norm(X, axis=1, keepdims=True)  # shape (3,1)
X_unit = X / row_norms
```

Mask-based replacement.

```
a = np.array([ -3., 0., 2., np.nan, 5.])
# Replace negatives and NaNs with 0
mask = (a < 0) | np.isnan(a)
a_fixed = a.copy()
a_fixed[mask] = 0.0
```

12.12 Exercises

1. Let $a =$ `np.array([1,2,3,4])` and $b =$ `np.array([4,3,2,1])`. Compute $2a - b$, $a \odot b$ (element-wise), and a/b.

2. Given $v =$ `np.array([1,2,3,4])`, create a row vector and a column vector (shapes $(1,4)$ and $(4,1)$), then form their outer product.

3. For $A =$ `np.arange(12).reshape(3,4)`, subtract the column means from each column (hint: compute `A.mean(axis=0)` and broadcast).

4. Implement row-wise normalization for a random matrix X so that each row has mean 0 and standard deviation 1 (use broadcasting).

5. Create a function that takes an array x and returns $y = \max(0, x)$ (ReLU) using vectorized operations only (e.g. `np.maximum`).

6. (Numerical stability) Evaluate $\log(1+x)$ for x in $\{-10^{-8}, \ldots, 10^{-8}\}$ using `np.log(1+x)` vs `np.log1p(x)` and compare errors to the Taylor approximation.

13 Joining and Splitting Arrays

In many applications, we need to combine arrays into larger structures or split arrays into smaller pieces. NumPy provides efficient functions for both operations.

13.1 Concatenation with np.concatenate

The simplest way to join arrays is with `np.concatenate`. It glues arrays along an existing axis.

```
import numpy as np

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

c = np.concatenate((a,b))
print(c)    # [1 2 3 4 5 6]
```

For 2D arrays, the axis matters:

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

print(np.concatenate((A,B), axis=0))
# [[1 2]
#  [3 4]
#  [5 6]]

print(np.concatenate((A,B.T), axis=1))
# [[1 2 5]
#  [3 4 6]]
```

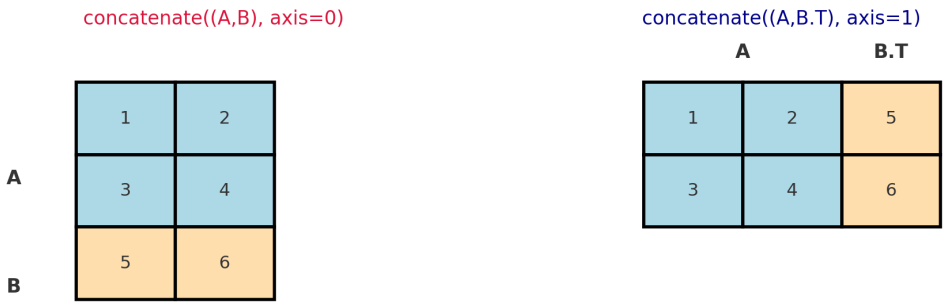


Figure 21: Concatenation with different axes. Left: `np.concatenate((A,B), axis=0)` stacks arrays along rows (vertical concatenation). Right: `np.concatenate((A,B.T), axis=1)` stacks along columns (horizontal concatenation after transposing `B`).

13.2 Stacking Arrays

Unlike `concatenate`, stacking creates a new dimension.

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

print(np.stack((a,b)))
# [[1 2 3]
#  [4 5 6]]

print(np.vstack((a,b)))
# [[1 2 3]
#  [4 5 6]]

print(np.hstack((a,b)))
# [1 2 3 4 5 6]
```

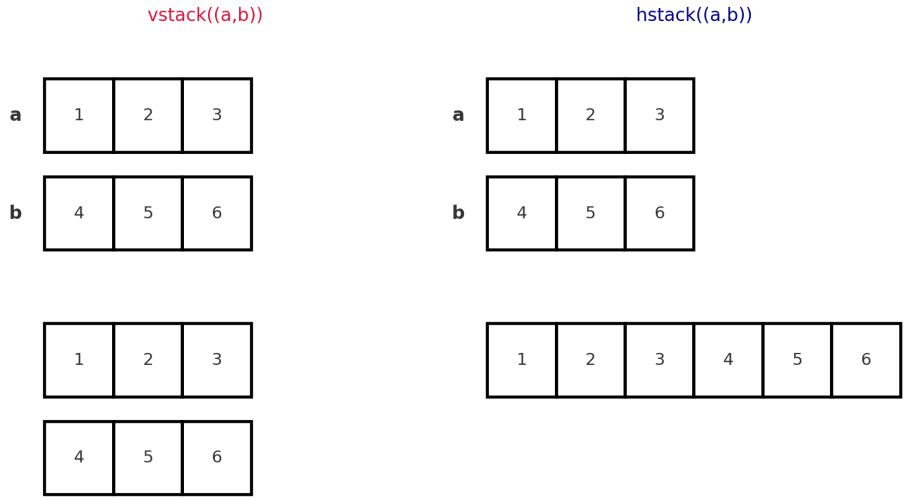


Figure 22: Stacking arrays. Left: `np.vstack((a,b))` stacks arrays vertically (row-wise). Right: `np.hstack((a,b))` stacks arrays horizontally (column-wise).

13.3 Splitting Arrays

Splitting is the inverse of concatenation. We can use `np.split` (equal parts) or `np.array_split` (unequal parts allowed).

```
a = np.arange(9)

print(np.split(a, 3))
# [array([0,1,2]), array([3,4,5]), array([6,7,8])]

print(np.array_split(a, 4))
# [array([0,1,2]), array([3,4]), array([5,6]), array([7,8])]
```

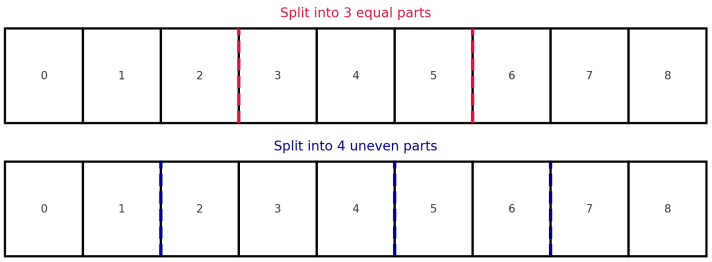


Figure 23: Splitting 1D arrays. Top: `np.split(arr, 3)` splits an array of length 9 into 3 equal parts. Bottom: `np.array_split(arr, 4)` splits into 4 uneven parts, since 9 is not divisible by 4.

For 2D arrays, we also have:

```
A = np.arange(16).reshape(4,4)

print(np.hsplit(A, 2))  # split into 2 column blocks
print(np.vsplit(A, 2))  # split into 2 row blocks
```

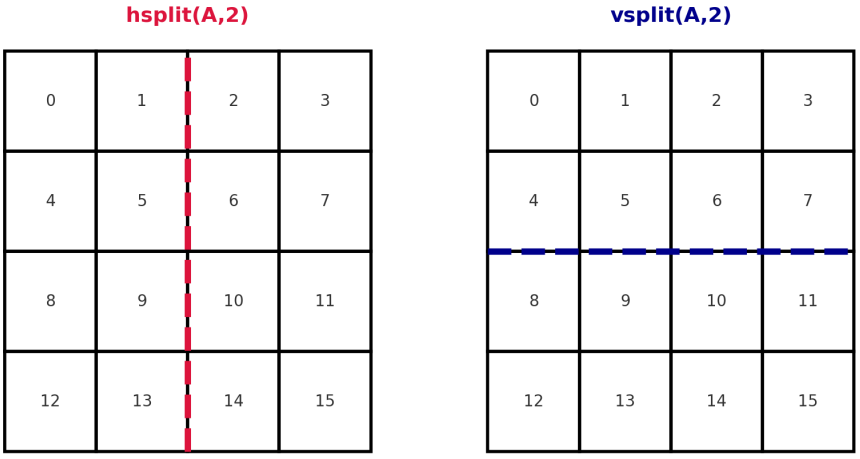


Figure 24: Splitting a 2D array into sub-arrays. Left: `np.hsplit(A,2)` divides into two column blocks (vertical cut). Right: `np.vsplit(A,2)` divides into two row blocks (horizontal cut).

14 Searching in NumPy Arrays

NumPy provides efficient functions to search for elements or their positions in arrays. These operations are vectorized and much faster than searching in plain Python lists. The most important functions include `np.where`, `np.argmax`, `np.argmin`, `np.nonzero`, `np.in1d`, and `np.unique`.

14.1 np.where

The `np.where` function returns the indices where a condition holds true. It can also be used for conditional selection.

```
import numpy as np

arr = np.array([10, 20, 30, 40, 50])
indices = np.where(arr > 25)

print(indices)        # (array([2, 3, 4]),)
print(arr[indices])   # [30 40 50]
```

14.2 np.argmax and np.argmin

`np.argmax(arr)` returns the index of the maximum element, while `np.argmin(arr)` returns the index of the minimum element.

```
arr = np.array([3, 8, 1, 10, 5])

print(np.argmax(arr))  # 3 (value 10)
print(np.argmin(arr))  # 2 (value 1)
```

14.3 np.nonzero

The function `np.nonzero` returns the indices of non-zero elements:

```
arr = np.array([0, 2, 0, 5, 0, 7])
print(np.nonzero(arr))  # (array([1, 3, 5]),)
```

14.4 np.in1d

Checks whether elements of one array are present in another array:

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

print(np.in1d(a, b))
# [False  True False  True False]
```

14.5 np.unique

Returns the unique values in an array and, optionally, their counts.

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

print(np.unique(arr))
# [1 2 3 4]

values, counts = np.unique(arr, return_counts=True)
print(values)  # [1 2 3 4]
print(counts)  # [1 2 3 1]
```

15 Iterating over Arrays

Iteration means accessing array elements one by one. While NumPy is designed for vectorized operations (where we avoid explicit loops), sometimes iteration is necessary or useful for teaching and debugging. NumPy offers both simple Python-style iteration and specialized tools like `np.nditer`.

15.1 Basic Iteration on 1D Arrays

Iterating over a 1D array works exactly like iterating over a Python list:

```
import numpy as np

a = np.array([10, 20, 30, 40])

for x in a:
    print(x)
# Output:
# 10
# 20
# 30
# 40
```

15.2 Iteration on 2D Arrays

When we loop over a 2D array, the iteration happens row by row:

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

for row in b:
    print(row)
# Output:
# [1 2 3]
# [4 5 6]
```

```
# Nested loop to access individual elements
for row in b:
    for elem in row:
        print(elem, end=" ")
# Output: 1 2 3 4 5 6
```

15.3 Iterating with np.nditer

For more control and efficiency, we use `np.nditer`, which allows us to iterate over every element regardless of dimension.

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

```

```python
for x in np.nditer(c):
    print(x)
# Output: 1 2 3 4 5 6
```

We can also control the memory order:

- `order='C'` → row-major (default, C-style).
- `order='F'` → column-major (Fortran-style).

```python
print(list(np.nditer(c, order='C')))  # [1 2 3 4 5 6]
print(list(np.nditer(c, order='F')))  # [1 3 5 2 4 6]
```

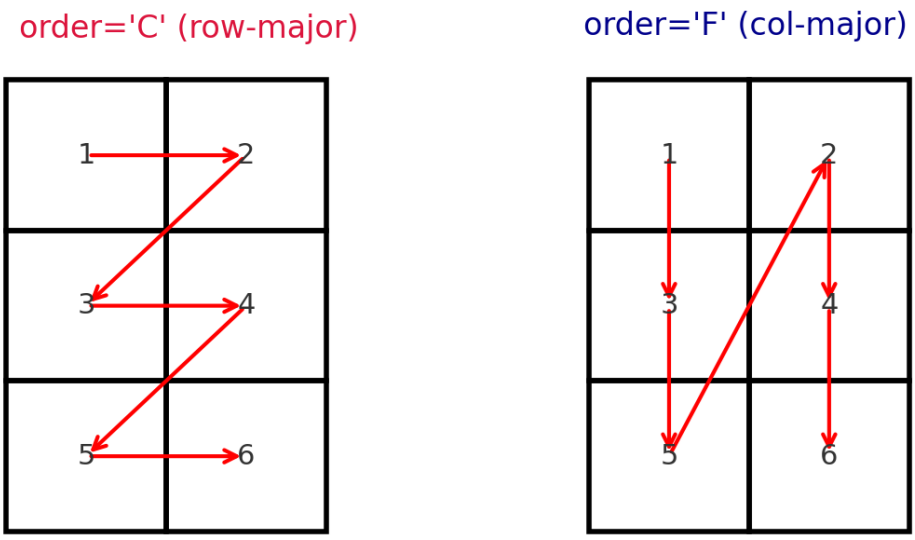


Figure 25: Iteration order with `np.nditer`. Left: C-style (row-major) iteration goes row by row. Right: Fortran-style (column-major) iteration goes column by column.

15.4 Enumerating Indices with np.ndenumerate

Sometimes we want both the index and the value. `np.ndenumerate` yields a tuple (index, value):

```python
for index, value in np.ndenumerate(c):
    print(index, value)

# Output:
# (0, 0) 1
# (0, 1) 2
# (1, 0) 3
# (1, 1) 4
# (2, 0) 5
# (2, 1) 6
```

16 Linear Algebra with NumPy

PS. This section only covers the *minimum essentials* of linear algebra in NumPy. Much more will be presented in detail throughout the semester. The purpose here is to provide just enough background so that the upcoming lessons will be easier to follow.

Linear algebra is fundamental to data science, physics, and engineering. NumPy provides efficient and reliable routines for common operations such as dot products, norms, matrix multiplication, decompositions, and solving systems. All functions are in the `numpy.linalg` submodule.

16.1 Vector and Matrix Operations

Dot product of vectors. For two vectors $a, b \in \mathbb{R}^n$, the dot product is defined as:

$$a \cdot b = \sum_{i=1}^{n} a_i b_i$$

It measures how aligned two vectors are: if a and b are orthogonal, the dot product is 0.

```
import numpy as np

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

print(np.dot(a, b))    # 32
print(a @ b)           # equivalent, 32
```

Matrix–vector product. Given a matrix $A \in \mathbb{R}^{m \times n}$ and a vector $x \in \mathbb{R}^n$, the product $y = Ax$ is a vector in $\mathbb{R}^m$. This is a linear combination of the columns of A weighted by components of x.

```
A = np.array([[1, 2, 3],
              [4, 5, 6]])
x = np.array([1, 0, -1])

y = A @ x
print(y)  # [-2 -2]
```

Matrix–matrix multiplication. For $A \in \mathbb{R}^{m \times n}$ and $B \in \mathbb{R}^{n \times p}$:

$$C = AB, \qquad C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$

The result C is an $m \times p$ matrix.

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

```
                [7, 8]])

C = A @ B
print(C)
# [[19 22]
#  [43 50]]
```

16.2 Norms and Distances

The most common norm is the ℓ_2 (Euclidean) norm:

$$\|x\|_2 = \sqrt{\sum_i x_i^2}$$

In NumPy:

```
x = np.array([3.0, 4.0])
print(np.linalg.norm(x))  # 5.0
```

Other norms include: - ℓ_1 norm: $\|x\|_1 = \sum_i |x_i|$ - ℓ_∞ norm: $\|x\|_\infty = \max_i |x_i|$

```
print(np.linalg.norm(x, 1))     # 7.0
print(np.linalg.norm(x, np.inf))  # 4.0
```

Matrix norms:
- Frobenius norm $\|A\|_F = \sqrt{\sum_{ij} A_{ij}^2}$
- Spectral norm (largest singular value)

```
A = np.array([[1, 2],
              [3, 4]])
print(np.linalg.norm(A, 'fro'))  # Frobenius norm
```

16.3 Solving Linear Systems

A central problem: solving $Ax = b$ for x.

```
A = np.array([[3, 1],
              [1, 2]])
b = np.array([9, 8])

x = np.linalg.solve(A, b)
print(x)   # [2. 3.]
```

Always prefer `np.linalg.solve` over inverting A (`np.linalg.inv`) for numerical stability and performance.

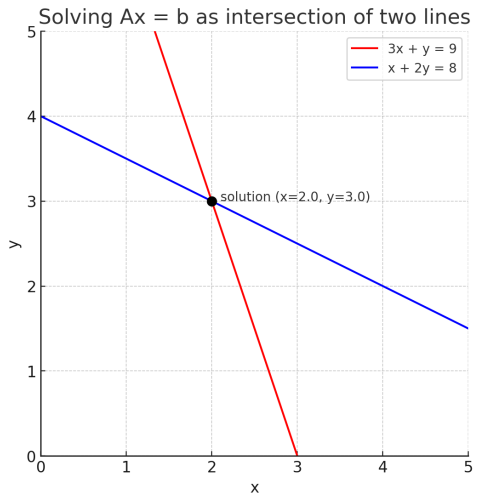


Figure 26: Solving $Ax = b$ as the intersection of two lines. The red line is $3x + y = 9$, the blue line is $x + 2y = 8$. Their intersection $(x = 2, y = 3)$ is the solution of the system.

16.4 Matrix Decompositions

Decompositions break a matrix into simpler factors, crucial for understanding its properties and for numerical algorithms.

Eigenvalues and eigenvectors. For $Av = \lambda v$, v is an eigenvector and λ its eigenvalue.

```
A = np.array([[4, -2],
              [1,  1]])

vals, vecs = np.linalg.eig(A)
print("Eigenvalues:", vals)
print("Eigenvectors:\n", vecs)
```

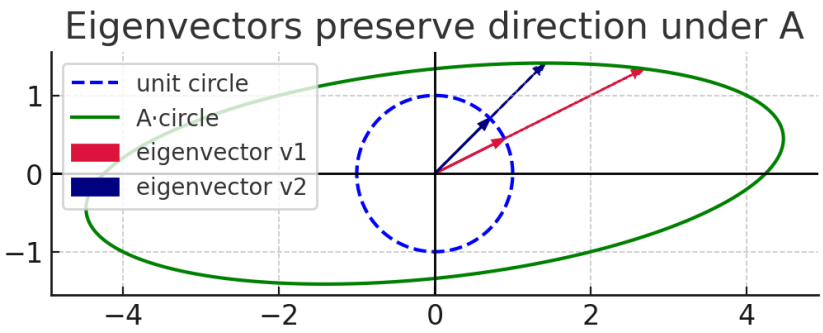


Figure 27: Eigenvectors preserve their direction under A. The unit circle (blue dashed) is transformed by A into an ellipse (green). The eigenvectors (red, blue) remain aligned with their original directions, scaled by their eigenvalues.

Singular Value Decomposition (SVD). Every matrix $A \in \mathbb{R}^{m \times n}$ can be written as:

$$A = U \Sigma V^T$$

where U and V are orthogonal, and Σ is diagonal with singular values.

```
U, s, Vt = np.linalg.svd(A)
print("U:", U)
print("Singular values:", s)
print("V^T:", Vt)
```

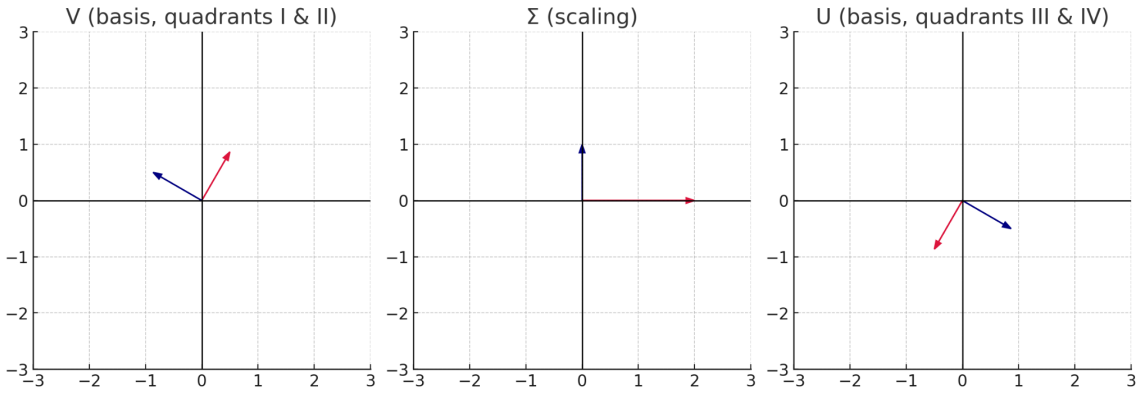


Figure 28: Geometric interpretation of the SVD. Left: V defines an orthogonal basis in the input space. Middle: Σ scales along coordinate axes (singular values). Right: U rotates into the output space.

QR decomposition. Factor $A = QR$ with Q orthogonal, R upper triangular.

```
Q, R = np.linalg.qr(A)
print("Q:", Q)
print("R:", R)
```

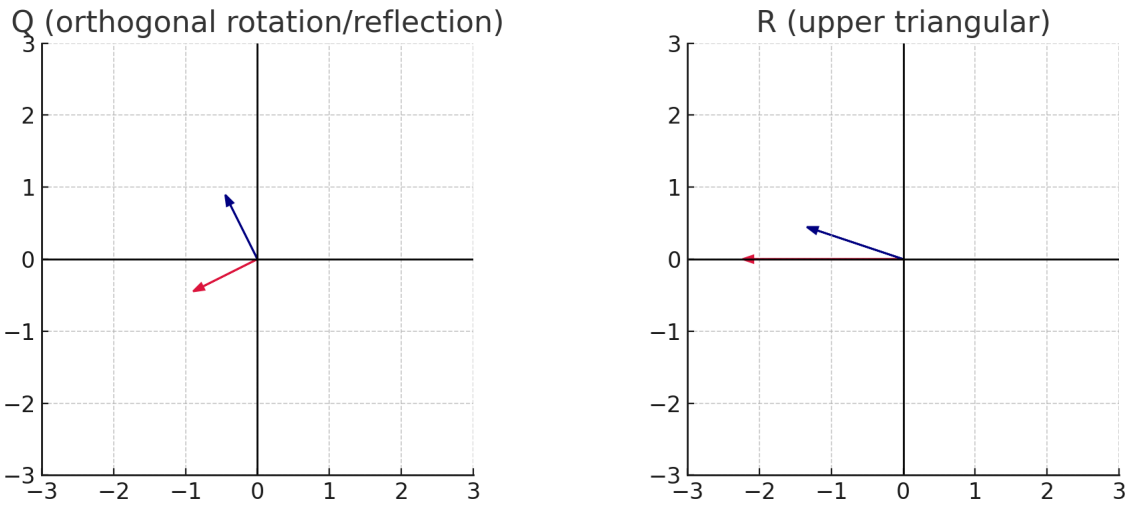


Figure 29: Geometric view of QR decomposition. Left: Q rotates or reflects the basis. Right: R applies an upper triangular shear and scaling.

Part III
SciPy Basics for Numerical Methods

17 Introduction to SciPy

PS. This chapter is meant only as an *introductory overview*. We will discover most of SciPy's functionality gradually, at the moments when we need it during the semester. Here the goal is simply to understand what SciPy is, how it relates to NumPy, and in which directions it may be useful for us.

17.1 What is SciPy?

SciPy ("Scientific Python") is a large collection of scientific computing routines built on top of NumPy. While NumPy provides the *data structure* (array objects) and the *basic numerical operations*, SciPy provides *algorithms and higher-level functionality* that are needed for more complex numerical tasks.

- NumPy focuses on:
  - array creation and manipulation,
  - vectorized operations,
  - basic linear algebra (`numpy.linalg`).
  - Fourier transforms (general case)
- SciPy focuses on:
  - advanced linear algebra routines (`scipy.linalg`),
  - optimization and root finding (`scipy.optimize`),
  - integration and solving ODEs (`scipy.integrate`),
  - interpolation of data (`scipy.interpolate`),
  - Fourier transforms - more advanced (`scipy.fft`),
  - sparse matrix structures (`scipy.sparse`),
  - and many more (signal processing, statistics, image analysis).

17.2 Why do we need SciPy?

In practice:

- We will continue to use NumPy arrays as the fundamental building block.
- Whenever we need a numerical method (a solver, an optimizer, an integrator, etc.), we will call the corresponding SciPy submodule.
- This way, we avoid implementing standard algorithms from scratch and rely on well-tested, efficient implementations.

17.3 NumPy vs SciPy: A First Look

It is useful to remember:

- NumPy provides the *core tools* (array container, basic linear algebra).
- SciPy provides the *specialized algorithms*.

For example, solving a linear system $Ax = b$:

```
import numpy as np
from scipy import linalg as sla

A = np.array([[3., 1.],
              [1., 2.]])
b = np.array([9., 8.])

# NumPy:
x_np = np.linalg.solve(A, b)

# SciPy:
x_sp = sla.solve(A, b)

print(x_np)    # [2. 3.]
print(x_sp)    # [2. 3.]
```

Both results are identical here, but SciPy also provides many extra routines (LU, QR, SVD, Schur decomposition, etc.) that are not directly available in NumPy.

NumPy vs SciPy — quick comparison

NumPy	SciPy
N-dimensional arrays (ndarray)	Algorithms built on top of NumPy
Vectorized math, broadcasting	Optimization (scipy.optimize)
Linear algebra (basic)	Integration & ODEs (scipy.integrate)
FFT, random numbers	Interpolation (scipy.interpolate)
Basic statistics	Statistics (scipy.stats)
I/O for simple formats	Signal & image processing
Array creation & indexing	Sparse matrices (scipy.sparse)

Figure 30: NumPy vs SciPy — quick comparison. Note: both provide FFT routines; for simple cases `numpy.fft` is sufficient, but for efficiency and extended functionality it is usually recommended to use `scipy.fft`.

18 Linear Algebra with `scipy.linalg`

Linear algebra is at the core of numerical computing. Just like `numpy.linalg`, SciPy provides `scipy.linalg` for linear algebra routines. The difference is that SciPy exposes many more functions, often with better numerical stability, and with direct access to LAPACK/BLAS routines.

NumPy vs SciPy. Both libraries allow you to solve common problems such as computing determinants, inverses, or solving systems. However:

- `numpy.linalg` provides only the most common routines.
- `scipy.linalg` provides everything in `numpy.linalg` and much more (LU, QR, Schur, Cholesky, SVD, Sylvester equations, etc.).

Determinant and inverse.

```
import numpy as np
from scipy import linalg

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

# determinant
print("det(A) =", linalg.det(A))

# inverse
print("A^{-1} =", linalg.inv(A))
```

Solving linear systems. Instead of computing the inverse (which is slow and unstable), always use `solve`:

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

# NumPy
x_np = np.linalg.solve(A, b)

# SciPy
x_sp = linalg.solve(A, b)

print("NumPy:", x_np)
print("SciPy:", x_sp)
```

Both produce the same result here, but SciPy allows you to choose specialized solvers for triangular systems, banded matrices, or sparse systems (via `scipy.sparse.linalg`).

LU decomposition. LU factorization splits A into PLU:

```
P, L, U = linalg.lu(A)
print("P =\n", P)
print("L =\n", L)
print("U =\n", U)
```

QR decomposition. QR factorization writes $A = QR$ with Q orthogonal and R upper triangular.

```
Q, R = linalg.qr(A)
print("Q =\n", Q)
print("R =\n", R)
```

Cholesky decomposition. For symmetric positive definite matrices:

$$A = LL^T$$

```
B = np.array([[4, 2],
              [2, 3]])
L = linalg.cholesky(B, lower=True)
print("L =\n", L)
```

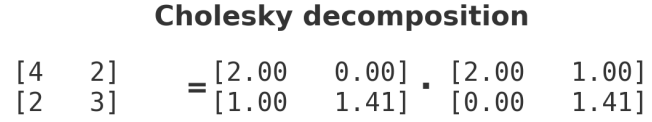


Figure 31: Cholesky decomposition of a symmetric positive definite matrix. Here $A = LL^T$, with L lower triangular.

Singular Value Decomposition (SVD). Every $A \in \mathbb{R}^{m \times n}$ can be written as:

$$A = U\Sigma V^T$$

```
U, s, Vh = linalg.svd(A)
print("Singular values:", s)
```

Eigenvalues and eigenvectors.

```
eigvals, eigvecs = linalg.eig(A)
print("Eigenvalues:", eigvals)
print("Eigenvectors:\n", eigvecs)
```

SciPy handles both real and complex matrices robustly.

Summary. Use `numpy.linalg` for basic problems in small scripts. Use `scipy.linalg` when you need advanced decompositions, better numerical stability, or direct LAPACK routines.

19 Overview of SciPy Modules

Note: This is only a quick orientation. We will return to many of these modules during the semester, when we need them in specific contexts. For now, the goal is to know what tools exist and how they can be accessed.

19.1 Optimization (`scipy.optimize`)

Used to find roots of equations or minimize functions.

```
from scipy import optimize
import numpy as np

# Root of x^2 - 2 = 0
f = lambda x: x**2 - 2
root = optimize.root_scalar(f, bracket=[0, 2])
print(root.root)   # approx sqrt(2)

# Minimize f(x) = x^2 + 5*sin(x)
res = optimize.minimize(lambda x: x**2 + 5*np.sin(x), x0=2.0)
print(res.x)
```

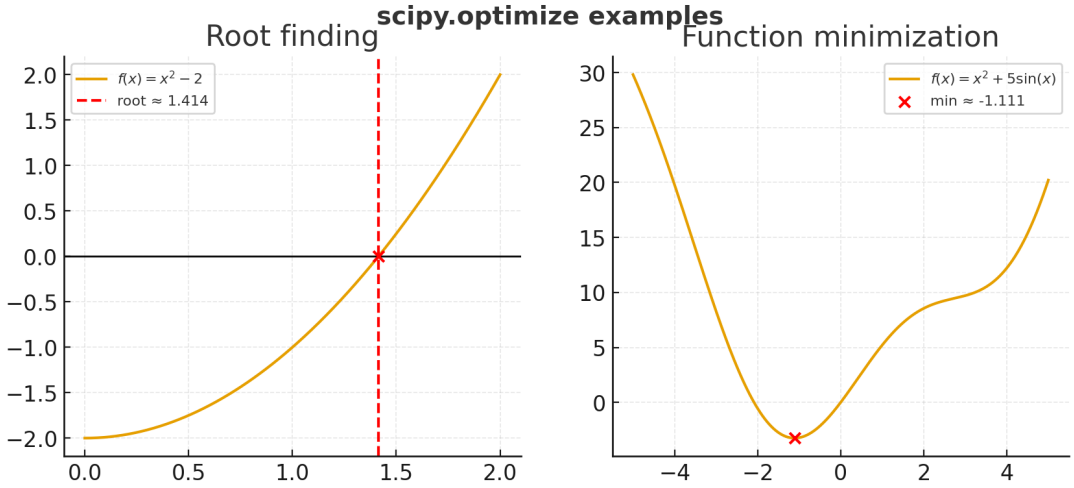


Figure 32: Examples from `scipy.optimize`. Left: root finding for $f(x) = x^2 - 2$. Right: minimization of $f(x) = x^2 + 5\sin(x)$.

19.2 Integration and ODEs (`scipy.integrate`)

Compute integrals and solve differential equations.

```
from scipy import integrate
import numpy as np

# Integral of exp(-x^2) from 0 to infinity
res, err = integrate.quad(lambda x: np.exp(-x**2), 0, np.inf)
print(res)

```

```
# Solve dy/dt = -2y, with y(0)=1
sol = integrate.solve_ivp(lambda t, y: -2*y, [0, 5], [1])
print(sol.y)
```

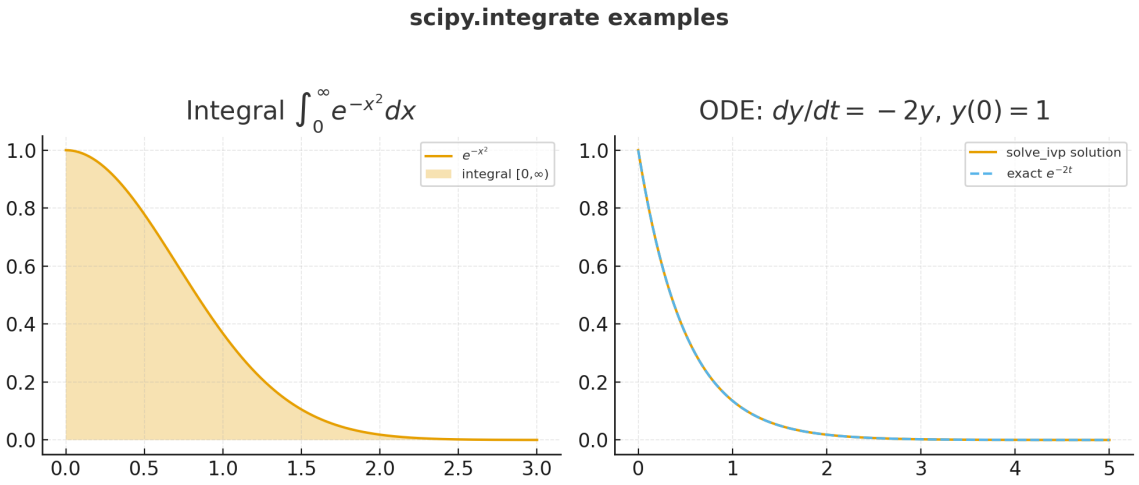


Figure 33: Examples from `scipy.integrate`. Left: numerical integration of $\int_0^\infty e^{-x^2}\,dx$. Right: solving the ODE $dy/dt = -2y, y(0) = 1$.

19.3 Interpolation (`scipy.interpolate`)

Construct interpolating functions from data.

```
from scipy import interpolate
import numpy as np

x = np.linspace(0, 10, 5)
y = np.sin(x)
f = interpolate.interp1d(x, y, kind="cubic")

print(f(5.5))  # evaluate interpolation
```

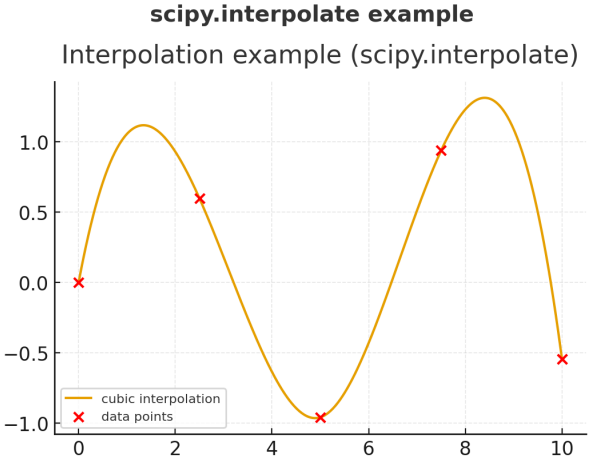


Figure 34: Interpolation with `scipy.interpolate.interp1d`. The original data points (red) are smoothly connected using a cubic spline.

19.4 Statistics (`scipy.stats`)

Large collection of probability distributions and statistical tests.

```
from scipy import stats

rv = stats.norm(loc=0, scale=1)  # normal distribution
print(rv.pdf(0))   # density at 0
print(rv.cdf(1.96)) # cumulative probability
```

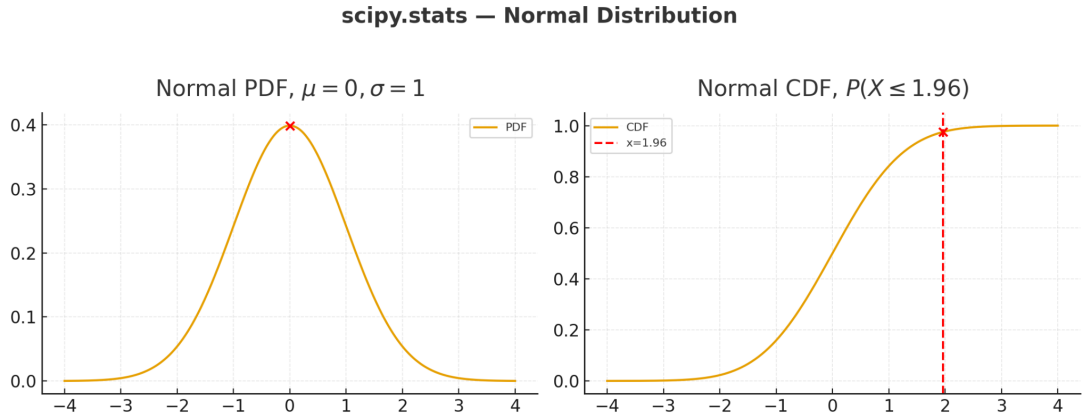


Figure 35: Normal distribution with `scipy.stats.norm`. Left: probability density function (PDF). Right: cumulative distribution function (CDF). More details will be presented next semester during the Probability and Statistics course.

19.5 Sparse Matrices (`scipy.sparse`)

Efficient storage and operations with sparse matrices.

```
from scipy import sparse
import numpy as np

A = sparse.csr_matrix([[0, 0, 1],
                       [1, 0, 0],
                       [0, 2, 0]])
print(A)
```

Output:

```
<Compressed Sparse Row sparse matrix of dtype 'int64'
        with 3 stored elements and shape (3, 3)>
  Coords        Values
  (0, 2)        1
  (1, 0)        1
  (2, 1)        2
```

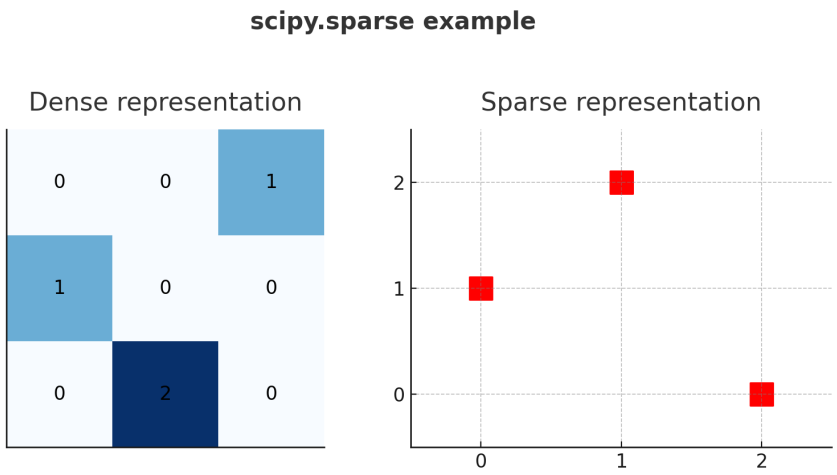


Figure 36: Dense vs sparse representation. Left: full matrix with many zeros. Right: sparse format only stores nonzero entries, saving memory and speeding up computations.

19.6 Signal and Image Processing

Basic tools for filtering and transforms.

```
from scipy import signal

b, a = signal.butter(3, 0.3)    # low-pass filter
print(b, a)
```

```
from scipy import ndimage
import numpy as np

image = np.random.rand(5,5)
blurred = ndimage.gaussian_filter(image, sigma=1)
print(blurred)
```

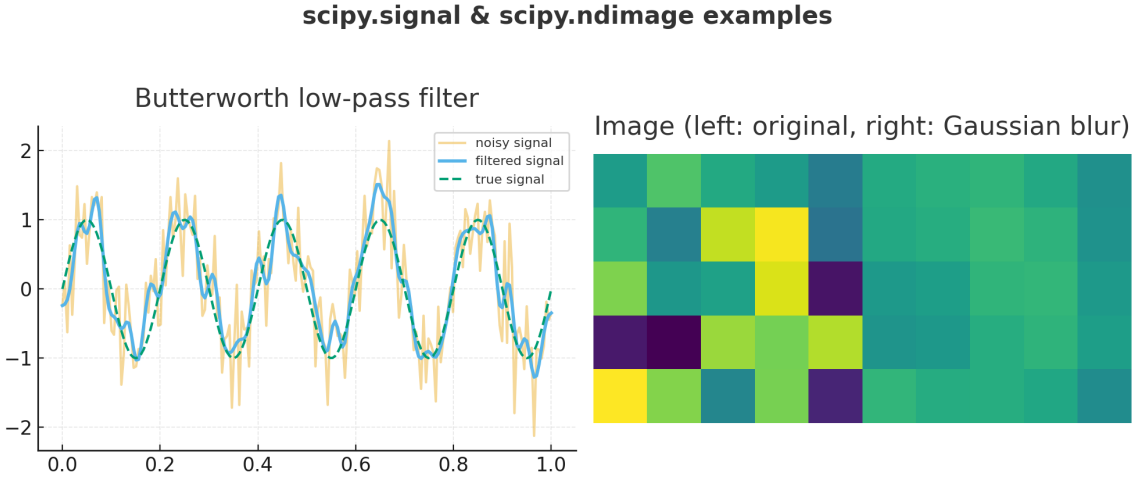


Figure 37: `scipy.signal` (left): filtering a noisy signal with a Butterworth low-pass filter. `scipy.ndimage` (right): applying a Gaussian blur to an image. Theoretical background (filters, Fourier, image processing) will be covered later.

Part IV
Plotting and Visualization with Matplotlib

20 Introduction to Matplotlib

PS. This part is meant only as an *introductory overview*. We will encounter Matplotlib repeatedly during the semester, whenever we need visualization for data analysis, numerical methods, or scientific results. Here, the goal is simply to learn the basic workflow of producing plots, so that visual feedback becomes a natural part of our experiments.

20.1 Why visualization?

In numerical computing, visualization serves several purposes:

- **Exploration:** understanding the shape of data, detecting patterns or anomalies.
- **Verification:** checking if numerical algorithms behave as expected.
- **Communication:** presenting results in a clear and intuitive form.

Matplotlib is the standard plotting library in Python. It is mature, flexible, and integrates directly with NumPy and SciPy. With Matplotlib, we can produce:

- simple line plots,
- multiple curves on the same axes,
- scatter plots, bar charts, histograms, pie charts,
- heatmaps and images,
- even 3D visualizations.

20.2 The Pyplot interface

The most common entry point is the `matplotlib.pyplot` interface, imported as:

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

This interface works in a state-machine style, similar to MATLAB: calling functions like `plt.plot`, `plt.title`, or `plt.show` affects the current figure and axes.

For advanced use, Matplotlib also has an *object-oriented* API (creating explicit `Figure` and `Axes` objects). We will briefly see both approaches, but for now we focus on the simpler Pyplot workflow.

20.3 A first plot

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

# Sample data: sine wave on [0, 2  ]
x = np.linspace(0, 2*np.pi, 200)
y = np.sin(x)

# Create plot
plt.plot(x, y)

# Add decorations
plt.title("Sine wave")
plt.xlabel("x")
plt.ylabel("sin(x)")

# Display on screen
plt.show()
```

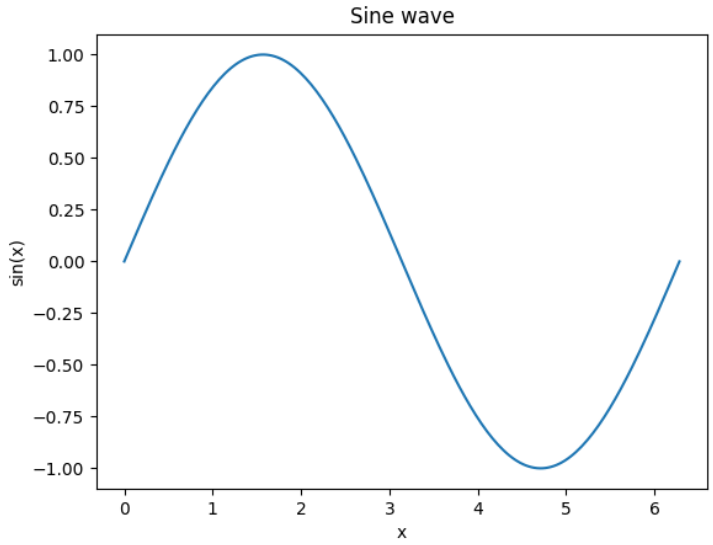


Figure 38: First plot with Matplotlib: sine function on $[0, 2\pi]$.

This shows the basic workflow:

1. Prepare data using NumPy.
2. Call a plotting function (here `plt.plot`).
3. Add titles, labels, legends, gridlines as needed.
4. Display with `plt.show()` or save with `plt.savefig("file.png")`.

20.4 Saving figures

Plots can be saved directly to disk in many formats (PNG, PDF, SVG, EPS).

```
plt.savefig("sine.png", dpi=150)    # save with resolution 150 dpi
```

This is especially useful for including results in reports or LaTeX documents.

20.5 Multiple plots on the same axes

We can call `plt.plot` several times to overlay multiple curves.

```
x = np.linspace(0, 2*np.pi, 200)

plt.plot(x, np.sin(x), label="sin(x)")
plt.plot(x, np.cos(x), label="cos(x)")

plt.title("Trigonometric functions")
plt.xlabel("x")
plt.ylabel("value")
plt.legend()
plt.grid(True)

plt.show()
```

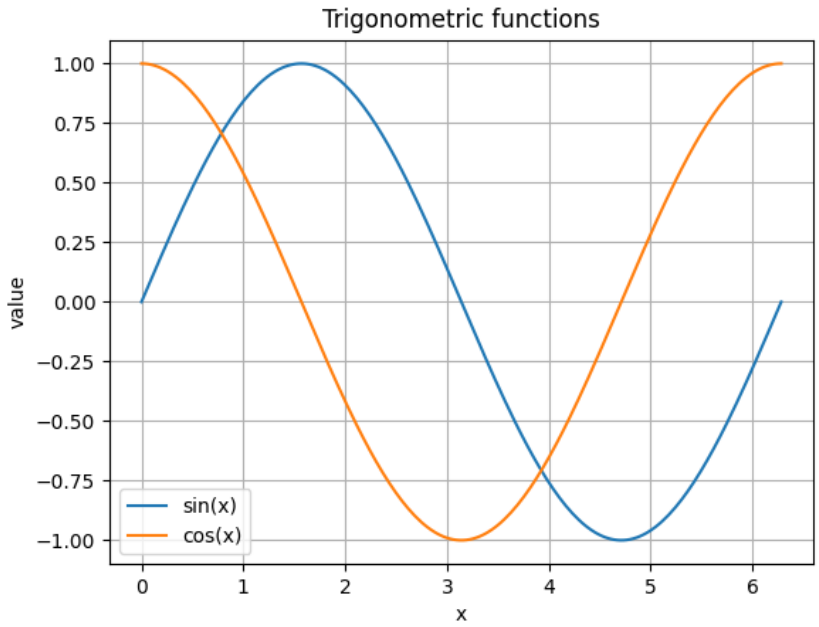


Figure 39: Sine and cosine plotted together, with legend and grid.

20.6 Figures and subplots

Each *figure* can contain one or more *subplots*. This allows structured layouts for comparing results.

```
fig, axs = plt.subplots(2, 2, figsize=(8,6))

x = np.linspace(0, 2*np.pi, 200)

axs[0,0].plot(x, np.sin(x))
axs[0,0].set_title("sin(x)")

axs[0,1].plot(x, np.cos(x))
axs[0,1].set_title("cos(x)")

axs[1,0].plot(x, np.sin(2*x))
axs[1,0].set_title("sin(2x)")

axs[1,1].plot(x, np.cos(2*x))
axs[1,1].set_title("cos(2x)")

plt.tight_layout()
plt.show()
```

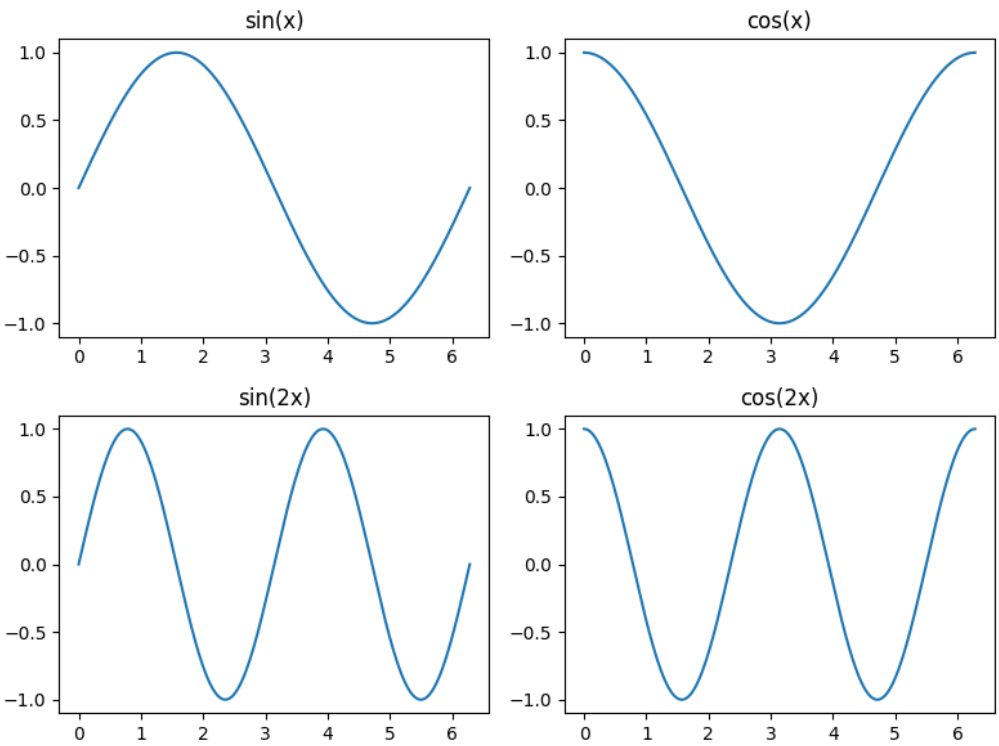


Figure 40: Using `subplots` to create a grid of multiple plots within one figure.

21 Basic Plotting Styles

Once we know how to make a simple line plot, the next step is to customize its appearance. Matplotlib offers fine control over colors, line styles, markers, grids, and axis labels.

21.1 Colors and line styles

We can specify colors and styles directly in `plt.plot()`. Common line styles:

- `'-'` solid line,
- `'-'` dashed line,
- `':'` dotted line,
- `'-.'` dash-dot line.

Common color codes: `'b'` (blue), `'r'` (red), `'g'` (green), `'k'` (black), `'m'` (magenta), `'c'` (cyan), `'y'` (yellow).

```
x = np.linspace(0, 2*np.pi, 200)

plt.plot(x, np.sin(x), 'r--', label="sin(x), dashed red")
plt.plot(x, np.cos(x), 'b:', label="cos(x), dotted blue")

plt.legend()
plt.show()
```

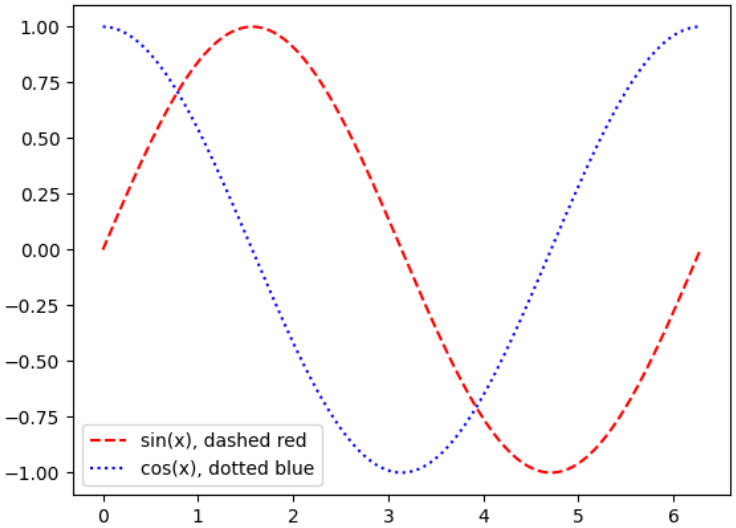


Figure 41: Customizing line styles in Matplotlib. Here, `sin(x)` is plotted as a dashed red line, while `cos(x)` is plotted as a dotted blue line.

21.2 Markers

We can add markers to emphasize data points. Examples: 'o' (circle), 's' (square), ^ (triangle), 'x' (cross).

```
x = np.linspace(0, 2*np.pi, 8)

plt.plot(x, np.sin(x), 'bo-', label="sin(x) with circles")
plt.plot(x, np.cos(x), 'gs--', label="cos(x) with squares")

plt.legend()
plt.show()
```

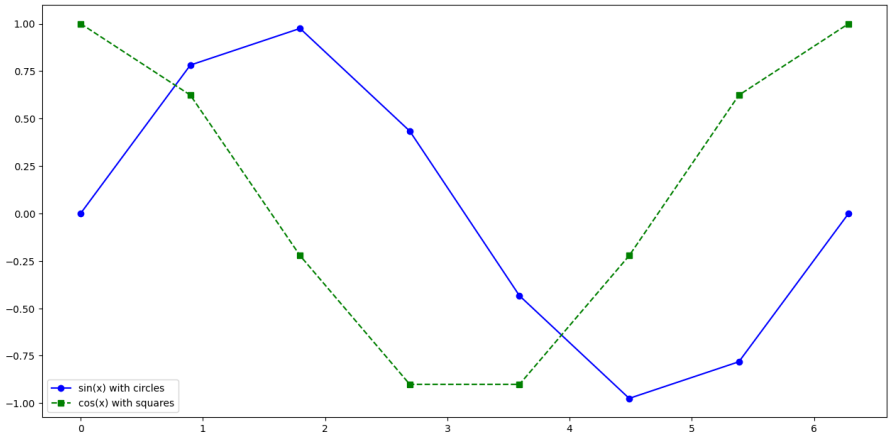


Figure 42: Customizing markers in Matplotlib. Here, `sin(x)` is plotted with circular markers (blue solid line), while `cos(x)` is plotted with square markers (green dashed line).

21.3 Axis limits and ticks

We can adjust the visible region of the plot with `xlim` and `ylim`.

```
x = np.linspace(-2*np.pi, 2*np.pi, 200)
y = np.sin(x)

plt.plot(x, y)
plt.xlim(0, 2*np.pi)
plt.ylim(-1.5, 1.5)
plt.show()
```

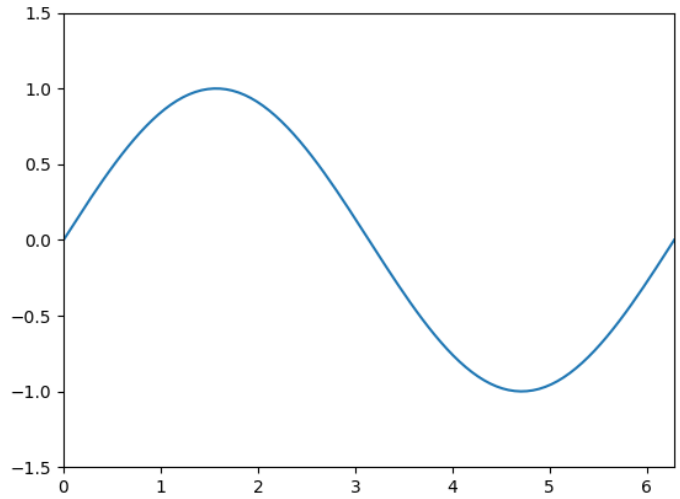


Figure 43: Basic plot in Matplotlib: plotting $\sin(x)$ with default settings. This illustrates the simplest use of `plt.plot` without extra formatting.

Custom ticks can be set with `xticks` and `yticks`.

```
plt.plot(x, y)
plt.xticks([0, np.pi, 2*np.pi],
           [r"$0$", r"$\pi$", r"$2\pi$"])
plt.show()
```

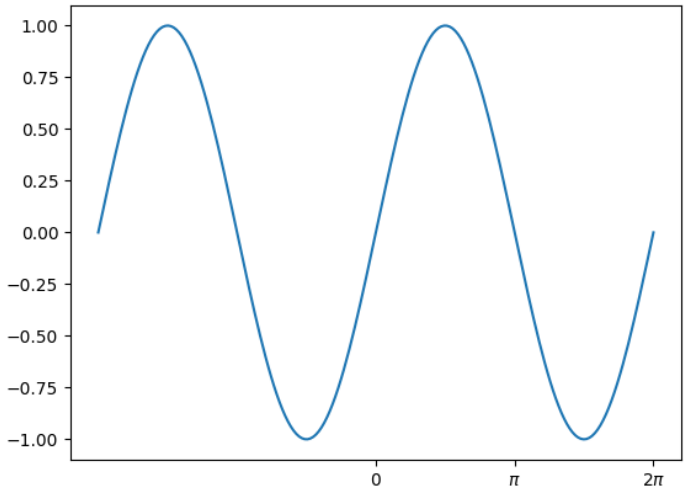


Figure 44: Customizing ticks and labels in Matplotlib. Here the x-axis is labeled using multiples of π, which makes trigonometric functions easier to interpret.

21.4 Grids and labels

Adding gridlines helps reading the values.
Labels can contain LaTeX math expressions.

```
plt.plot(x, y, label=r"$\sin(x)$")
plt.xlabel(r"$x$")
plt.ylabel(r"$y$")
plt.title("Sine function")
plt.grid(True)
plt.legend()
plt.show()
```

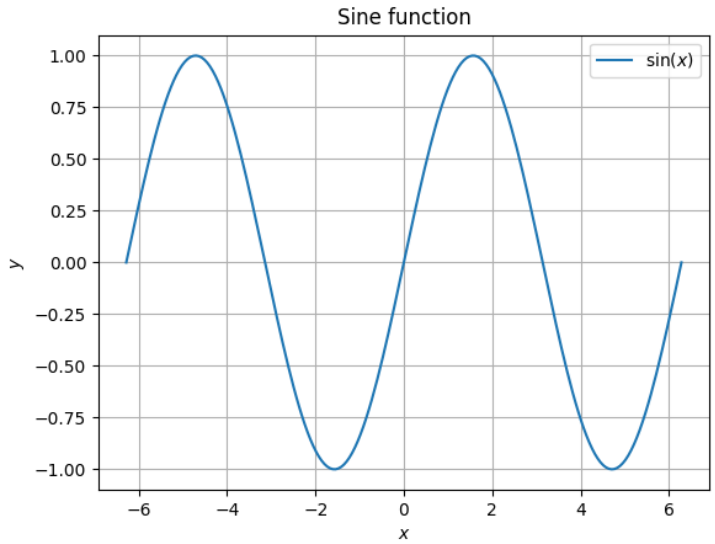


Figure 45: Adding titles, axis labels, limits, grid, and legend in Matplotlib. This makes the plot much more readable and informative.

22 Scatter Plots

Scatter plots visualize the relationship between two variables. Each point corresponds to one observation with coordinates (x, y).

22.1 Basic scatter plot

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

x = np.linspace(0, 10, 30)
y = np.sin(x) + 0.1*np.random.randn(30)

plt.scatter(x, y)
plt.xlabel("x")
plt.ylabel("y")
plt.title("Scatter plot: noisy sine values")
plt.show()
```

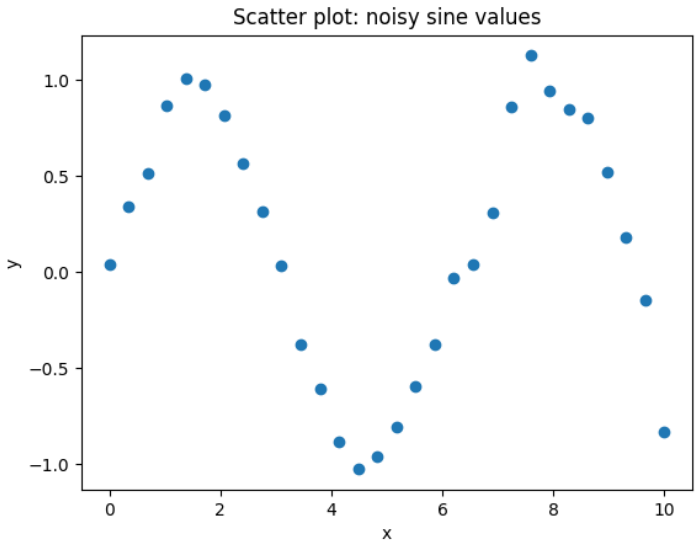


Figure 46: Scatter plot of a sine function with added noise.

22.2 Comparing two datasets

Scatter plots can show multiple datasets together.

```
np.random.seed(0)
x1 = np.random.normal(0, 1, 30)
y1 = np.random.normal(0, 1, 30)

x2 = np.random.normal(2, 1, 30)
y2 = np.random.normal(2, 1, 30)

plt.scatter(x1, y1, label="Cluster A")
```

```
plt.scatter(x2, y2, label="Cluster B")
plt.legend()
plt.title("Two clusters in 2D")
plt.show()
```

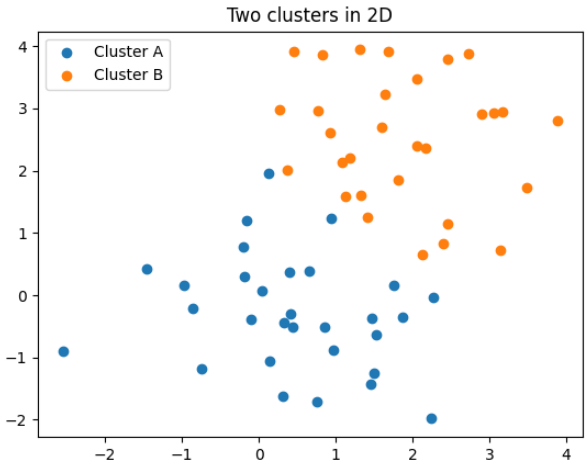


Figure 47: Scatter plot showing two clusters (blue and orange).

Colors, sizes, and transparency. Extra dimensions can be encoded in scatter plots:

- position (x, y),
- color (variable c),
- size (variable s),
- transparency (alpha).

```
x = np.random.rand(50)
y = np.random.rand(50)
c = x**2 + y**2          # color: distance from origin
s = 400 * (x+y)          # size: depends on sum

plt.scatter(x, y, c=c, s=s, cmap="viridis", alpha=0.6)
plt.colorbar(label="distance^2")
plt.title("Scatter plot with color and size encoding")
plt.show()
```

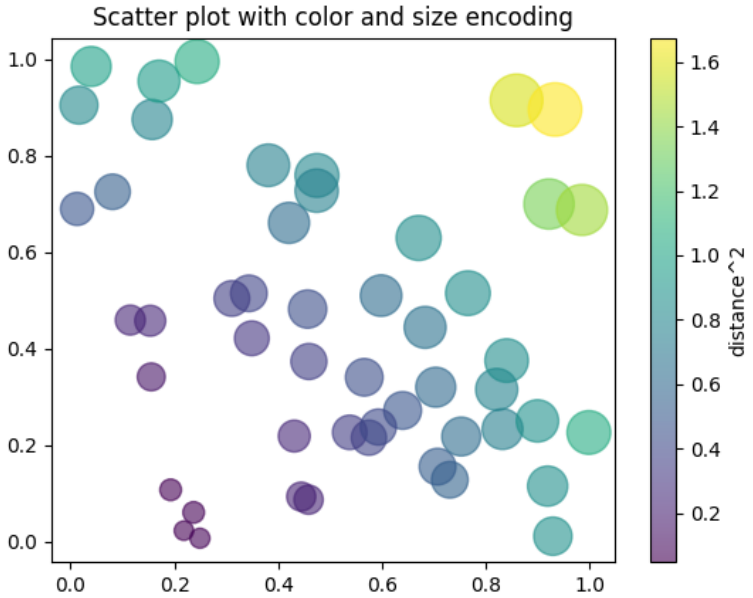


Figure 48: Scatter plot encoding multiple variables. Position (x, y), color (distance from origin), and size (based on $x + y$).

22.3 Scatter plot with colormap

In addition to manually setting colors, we can use a **colormap** to assign colors automatically based on numerical values (as we did in the last example). A colormap maps scalar values (e.g., between 0 and 100) to colors in a gradient. This is very useful when we want to highlight the relationship between a third variable and the (x, y) positions.

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

# generate random data
np.random.seed(0)
x = np.random.rand(50)
y = np.random.rand(50)
values = np.sqrt(x**2 + y**2) * 100  # third variable: distance from origin

plt.scatter(x, y, c=values, cmap="viridis")
plt.colorbar(label="Distance from origin")
plt.title("Scatter plot with colormap")
plt.xlabel("x")
plt.ylabel("y")
plt.show()
```

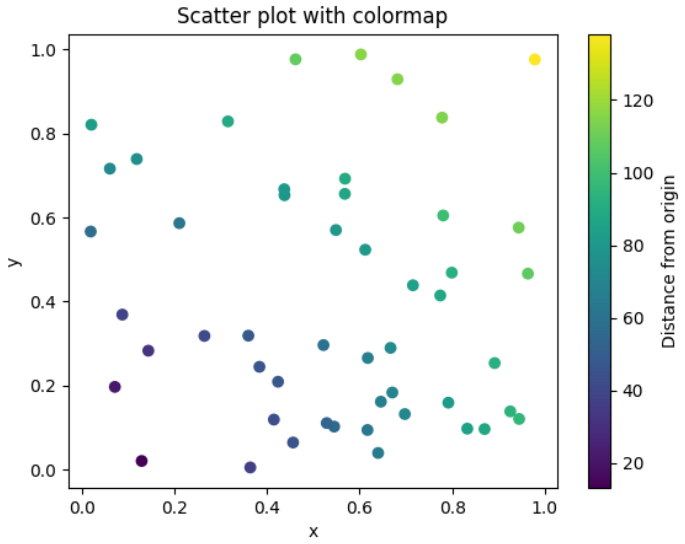


Figure 49: Scatter plot using a colormap ("viridis"). Each point's color represents its distance from the origin.

23 Bar plots

Bar plots are used to represent categorical data with rectangular bars. The length (or height) of each bar is proportional to the value it represents. With `pyplot`, the function `bar()` draws vertical bars, while `barh()` draws horizontal ones.

23.1 Basic bar plot

Simple bar plot with categories on the x-axis and values on the y-axis:

```
import matplotlib.pyplot as plt

categories = ["A", "B", "C", "D"]
values = [3, 7, 5, 2]

plt.bar(categories, values)
plt.title("Basic bar plot")
plt.xlabel("Categories")
plt.ylabel("Values")
plt.show()
```

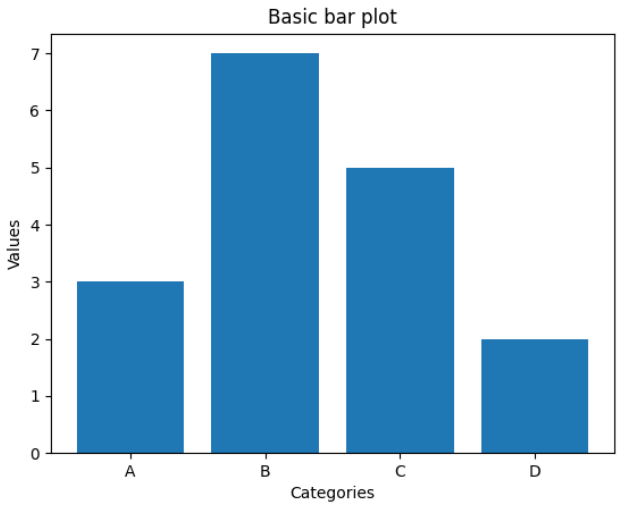


Figure 50: Basic bar plot with categories A–D.

23.2 Horizontal bar plot

One can also draw horizontal bars using `barh()`:

```
plt.barh(categories, values, color="orange")
plt.title("Horizontal bar plot")
plt.xlabel("Values")
plt.ylabel("Categories")
plt.show()
```

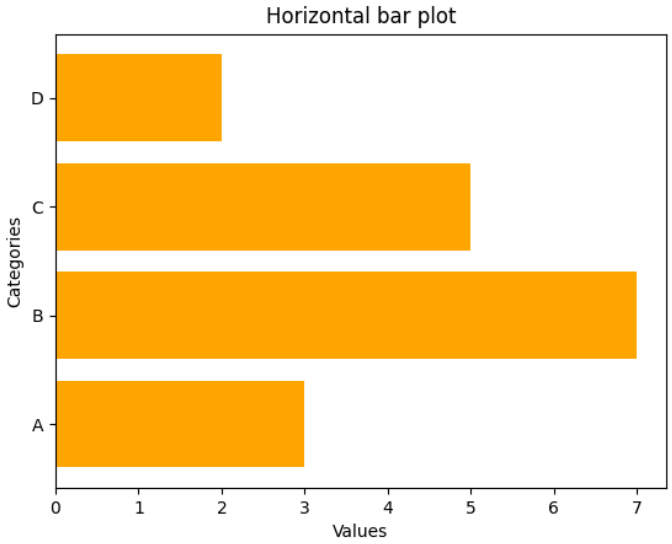


Figure 51: Horizontal bar plot with categories A–D in orange.

23.3 Grouped bar plot

It is often useful to compare two sets of values across the same categories. This is done with grouped bar plots by shifting the bar positions:

```
import numpy as np

categories = ["A", "B", "C", "D"]
values1 = [3, 7, 5, 2]
values2 = [4, 6, 4, 3]
x = np.arange(len(categories))

plt.bar(x - 0.2, values1, width=0.4, label="Group 1")
plt.bar(x + 0.2, values2, width=0.4, label="Group 2")
plt.xticks(x, categories)
plt.title("Grouped bar plot")
plt.legend()
plt.show()
```

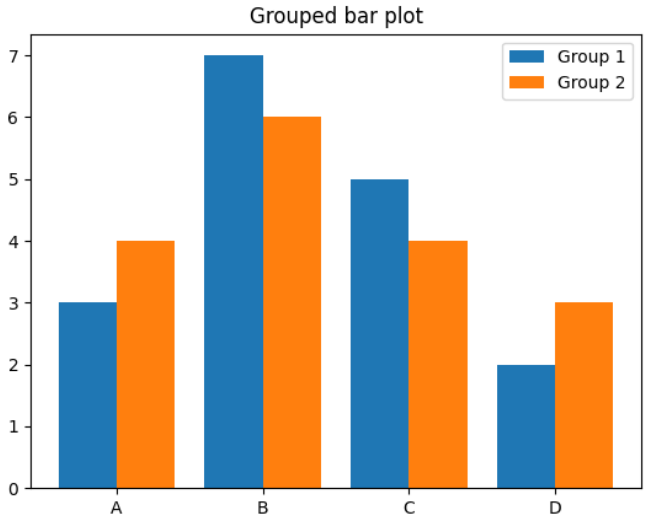


Figure 52: Grouped bar plot: categories A–D with two groups shown side by side.

23.4 Customization

Bars can be customized with different colors, widths, and edge styles. For example:

```
plt.bar(categories, values, color="skyblue", edgecolor="black", linewidth=1.2)
plt.title("Customized bar plot")
plt.show()
```

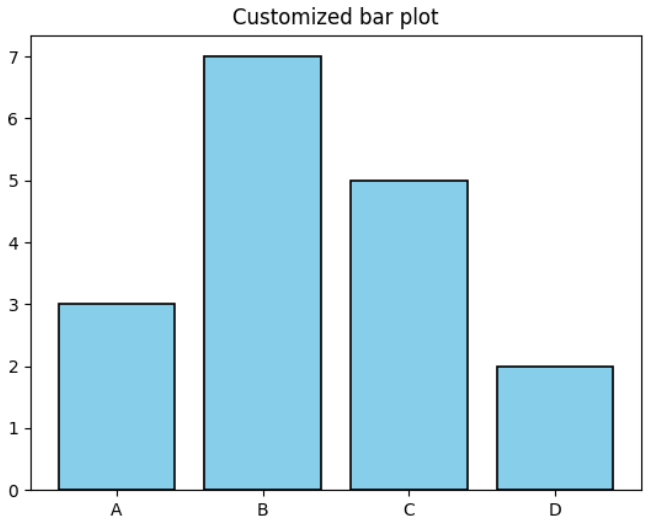


Figure 53: Customized bar plot: bars with `skyblue` fill and black edges.

24 Histograms

A **histogram** is a graphical representation of the distribution of numerical data. It works by dividing the range of the data into *bins* (intervals) and then counting how many data points fall into each bin. The result is similar to a bar plot, but with a continuous x-axis. The height of each bar shows the frequency (or probability, if normalized) of observations in that bin.

In `matplotlib`, histograms can be created using the function `plt.hist()`.

24.1 Classic histrogram

We can create a histogram for 1000 values sampled from a normal distribution. Here we use 30 bins, and customize the color of the bars.

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

# Generate 1000 samples from a normal distribution
data = np.random.randn(1000)

# Create histogram with 30 bins
plt.hist(data, bins=30, color="green", edgecolor="black")
plt.title("Histogram of normal distribution")
plt.xlabel("Value")
plt.ylabel("Frequency")
plt.show()
```

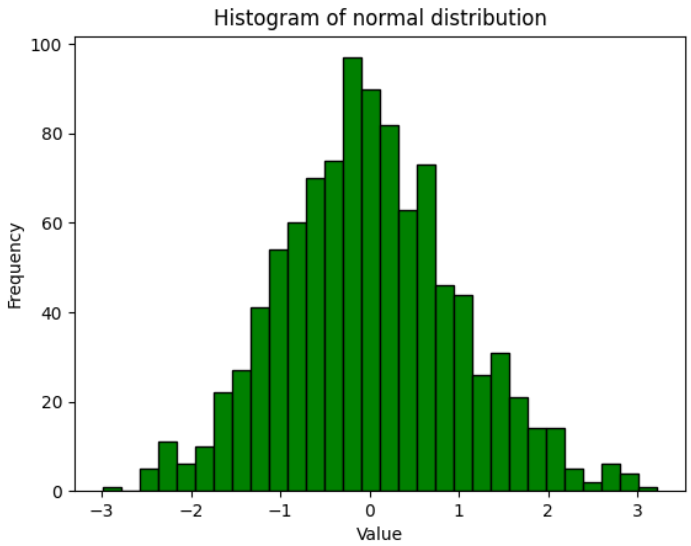


Figure 54: Histogram of 1000 normally distributed values with 30 bins.

24.2 Probability density.

By default, the histogram shows the *frequency* of values. If we want instead to show the *probability density*, we can use the argument `density=True`. In this case, the total

area under the histogram equals 1, making it easier to compare with probability density functions (PDFs).

```
plt.hist(data, bins=30, density=True, color="lightgreen", edgecolor="black")
plt.title("Normalized histogram (probability density)")
plt.xlabel("Value")
plt.ylabel("Density")
plt.show()
```

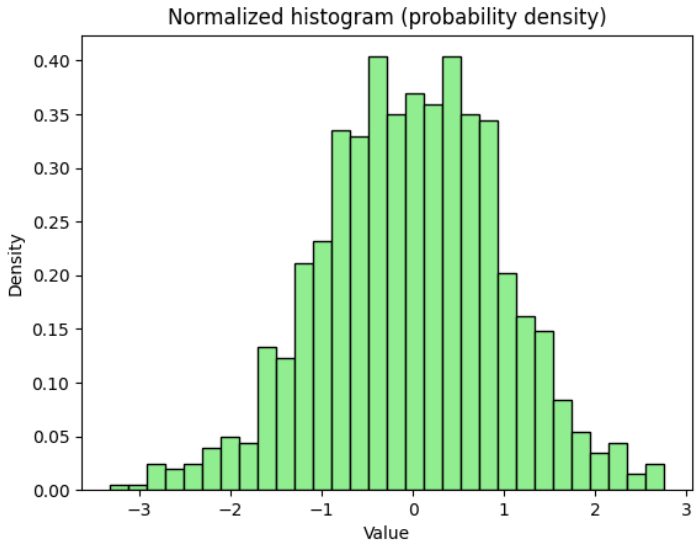


Figure 55: Normalized histogram (`density=True`) of 1000 normally distributed values.

Advanced example: comparing two groups. Histograms are often used to compare the distributions of two or more datasets. For example, suppose we measure the heights of men and women in a population. We can generate synthetic data and compare them in one figure using overlapping histograms.

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

# Synthetic height data (in cm)
men_heights = np.random.normal(loc=175, scale=7, size=1000)   # mean 175, std
women_heights = np.random.normal(loc=162, scale=6, size=1000) # mean 162, std

# Plot histograms on the same axes
plt.hist(men_heights, bins=30, alpha=0.6, color="blue", label="Men")
plt.hist(women_heights, bins=30, alpha=0.6, color="pink", label="Women")

plt.title("Comparison of Heights: Men vs Women")
plt.xlabel("Height (cm)")
plt.ylabel("Frequency")
plt.legend()
plt.grid(True)
plt.show()
```

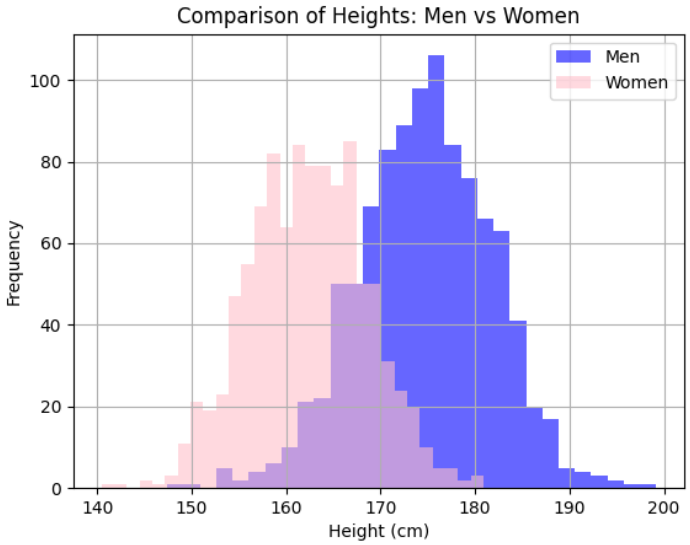


Figure 56: Two overlapping histograms with transparency: blue for men (centered at ~ 175 cm) and pink for women (centered at ~ 162 cm).

25 Stack plots (area plots)

Stack plots, also known as **area plots**, are used to show how multiple quantities evolve over the same range, stacked on top of each other. They are particularly useful when we want to visualize the contribution of each category to the total over time.

In `matplotlib`, the function `plt.stackplot()` creates such plots. It takes the x values (e.g., time or categories) and multiple y-series, which are then stacked.

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

# x-axis: time (days)
days = np.arange(1, 8)

# example data: hours spent by a student
sleep   = [7, 8, 6, 7, 8, 9, 8]
study   = [2, 3, 4, 3, 2, 4, 3]
exercise= [1, 1, 1, 1, 2, 1, 1]
leisure = [4, 3, 4, 5, 4, 3, 5]

# stack plot
plt.stackplot(days, sleep, study, exercise, leisure,
              labels=["Sleep","Study","Exercise","Leisure"],
              colors=["skyblue","orange","green","violet"],
              alpha=0.8)

plt.legend(loc="upper left")
plt.title("Daily activities over one week")
plt.xlabel("Day")
plt.ylabel("Hours")
plt.show()
```

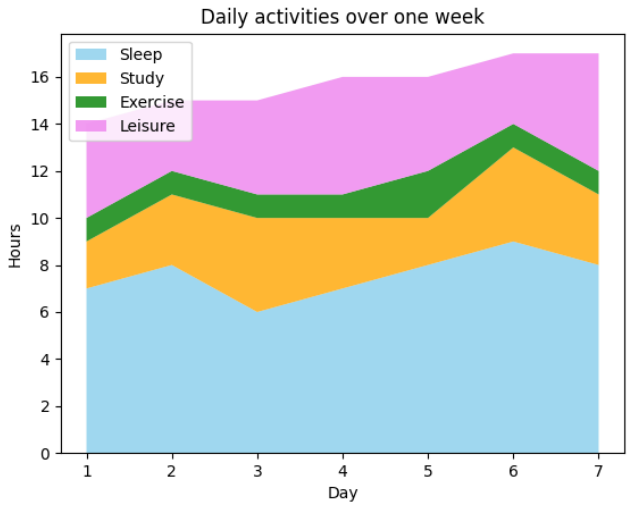


Figure 57: Stack plot of a student's activities during one week: sleep, study, exercise, and leisure time stacked together.

26 Pie charts

Pie charts are a common way to visualize **proportions of a whole**. Each slice of the circle corresponds to a category, and the angle of the slice is proportional to its percentage of the total. They are useful when we want to emphasize how different parts contribute to the overall sum.

Matplotlib provides the function `plt.pie()` to generate pie charts. The essential input is a list of values (one for each category). Optional arguments allow us to customize the chart with colors, labels, percentages, and even highlight specific slices.

26.1 Basic example

We start with a very simple pie chart showing the distribution of time in one day across different activities.

```
import matplotlib.pyplot as plt

# Daily activities in hours
activities = ["Sleep", "Study", "Exercise", "Leisure", "Other"]
hours = [8, 4, 2, 6, 4]

plt.pie(hours, labels=activities)
plt.title("Basic daily activities distribution")
plt.show()
```

Figure 58: Basic pie chart of daily activities.

26.2 Adding percentages and colors

We can use the `autopct` argument to display percentages on each slice, and the `colors` argument to manually select colors.

```
colors = ["skyblue", "orange", "green", "violet", "gray"]

plt.pie(hours,
        labels=activities,
        colors=colors,
        autopct="%1.1f%%",  # show percentages
        startangle=90       # rotate chart for better readability
       )
```

```
plt.title("Daily activities distribution")
plt.show()
```

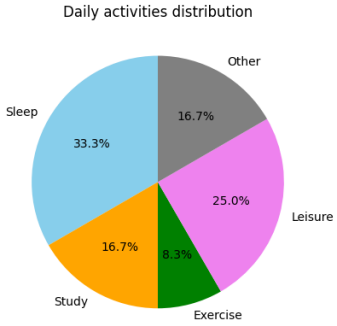


Figure 59: Pie chart showing daily activities with percentages.

26.3 Highlighting one category (explode)

Sometimes we want to emphasize one category (e.g. `Sleep`) by pulling it out from the chart.

```
explode = (0.1, 0, 0, 0, 0)  # highlight "Sleep"

plt.pie(hours,
        labels=activities,
        colors=colors,
        autopct="%1.1f%%",
        startangle=90,
        explode=explode
       )
plt.title("Daily activities with Sleep highlighted")
plt.show()
```

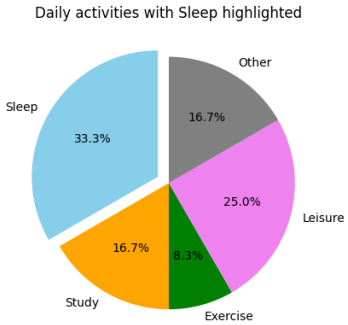


Figure 60: Pie chart with one slice (“Sleep”) emphasized using the `explode` option.

26.4 Comparing two groups

Pie charts can also be used for comparison, though bar plots are usually better. Here we show daily activity distributions for two different individuals side by side.

```
fig, axs = plt.subplots(1, 2, figsize=(8,4))

# Person A
hours_A = [7, 5, 1, 7, 4]
axs[0].pie(hours_A, labels=activities, autopct="%1.0f%%")
axs[0].set_title("Person A")

# Person B
hours_B = [8, 3, 2, 6, 5]
axs[1].pie(hours_B, labels=activities, autopct="%1.0f%%")
axs[1].set_title("Person B")

plt.suptitle("Comparison of daily activities")
plt.show()
```

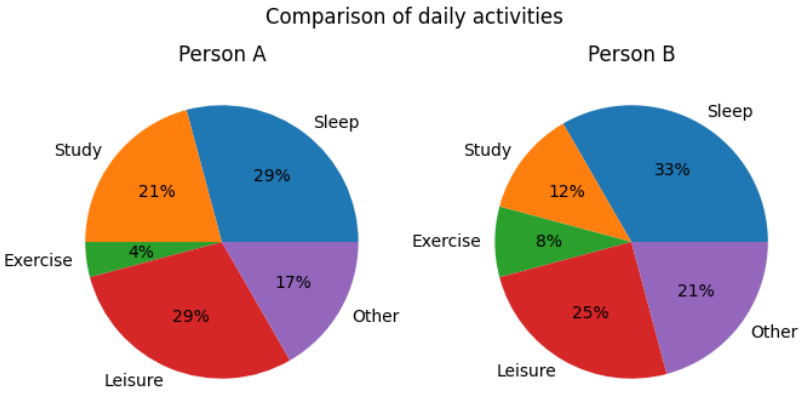


Figure 61: Side-by-side comparison of daily activities for two individuals.