First-Year Mathematics

Python Programming Examples

Lists · Functions · Loops · Conditionals · NumPy · Matplotlib

Overview

This document presents four complete, self-contained Python programming examples designed for first-year mathematics students. Each example covers a real mathematical topic and deliberately uses all the core programming concepts: lists, user-defined functions, loops, conditional statements, NumPy arrays, and Matplotlib plots.

> **Prerequisites:** Python 3.x with NumPy and Matplotlib installed. Run: pip install numpy matplotlib

Example	Mathematical Topic	Python Concepts Used
1	Quadratic Functions & Roots	Functions, lists, loops, conditionals, NumPy, plot
2	Trigonometric Functions	Functions, NumPy arrays, loops, conditions, plot
3	Descriptive Statistics	Lists, functions, loops, conditionals, NumPy, histogram
4	Linear Regression	Functions, NumPy arrays, scatter plot, regression line

Example 1 — Quadratic Functions & Roots

Mathematical Background

A quadratic function has the form $f(x) = ax^2 + bx + c$. Its roots (zeros) are found using the discriminant $\Delta = b^2 - 4ac$:

- If $\Delta > 0$: two distinct real roots
- If $\Delta = 0$: one repeated root
- If $\Delta < 0$: no real roots (complex)

The vertex is the point $(-b/2a\ ,\ f(-b/2a))$ and represents the minimum ($a > 0$) or maximum ($a < 0$) of the parabola.

Python Code

Step 1 — Import libraries and define the function

```
# Function definition
import numpy as np
import matplotlib.pyplot as plt
import math

# ── Define the quadratic function ──────────────────────────────
def quadratic(a, b, c, x):
    """Compute f(x) = a*x^2 + b*x + c"""
    return a * x**2 + b * x + c
```

Step 2 — Find roots using a conditional structure

```
# Conditional statements + lists
# ── Find roots using the discriminant ──────────────────────────
def find_roots(a, b, c):
    """Return list of real roots of ax^2 + bx + c = 0"""
    discriminant = b**2 - 4 * a * c
    roots = []                                # empty list

    if discriminant > 0:                      # two distinct roots
        x1 = (-b + math.sqrt(discriminant)) / (2 * a)
        x2 = (-b - math.sqrt(discriminant)) / (2 * a)
        roots.append(x1)
        roots.append(x2)
        print(f"Two real roots: x1 = {x1:.4f}, x2 = {x2:.4f}")

    elif discriminant == 0:                   # one repeated root
        x0 = -b / (2 * a)
        roots.append(x0)
        print(f"One repeated root: x = {x0:.4f}")

    else:                                     # no real roots
        print("No real roots (discriminant < 0)")

    return roots
```

Step 3 — Compute values using a loop and build a table

```
# Loops + lists
# ── Evaluate f(x) at several points using a for loop ──────────────
def build_value_table(a, b, c, x_start, x_end, step):
    """Build a list of [x, f(x)] pairs"""
    table = []                              # list to store pairs
    x = x_start
    while x <= x_end:                       # while loop
        y = quadratic(a, b, c, x)
        table.append([round(x, 2), round(y, 4)])
        x += step
    return table
```

Step 4 — Plot using NumPy and Matplotlib

```
# NumPy arrays + Matplotlib
# ── Parameters: f(x) = x^2 - 4x + 3 ────────────────────────────────
a, b, c = 1, -4, 3

# NumPy array for smooth curve (400 equally spaced points)
x_array = np.linspace(-1, 5, 400)
y_array = quadratic(a, b, c, x_array)      # NumPy vectorised call

# Find roots
roots = find_roots(a, b, c)

# Vertex
x_vertex = -b / (2 * a)
y_vertex = quadratic(a, b, c, x_vertex)
print(f"Vertex: ({x_vertex}, {y_vertex})")

# ── Plot ────────────────────────────────────────────────────────────
plt.figure(figsize=(8, 5))
plt.plot(x_array, y_array, color="steelblue", linewidth=2.5,
         label=r"$f(x)=x^2-4x+3$")
plt.axhline(0, color="gray", linewidth=0.8)
plt.axvline(0, color="gray", linewidth=0.8)

# Mark roots
for r in roots:
    plt.scatter(r, 0, color="red", s=80, zorder=5)

# Mark vertex
plt.scatter(x_vertex, y_vertex, color="green", s=100, marker="*", zorder=5,
            label=f"Vertex ({x_vertex}, {y_vertex})")

plt.title("Quadratic Function: f(x) = x² - 4x + 3")
plt.xlabel("x")
plt.ylabel("f(x)")
plt.legend()
plt.grid(True)
plt.show()
```

Program Output

Output:

```
Two real roots: x1 = 3.0000, x2 = 1.0000
Vertex: (2.0, -1.0)
```

Value Table: $f(x) = x^2 - 4x + 3$

x	f(x)	Sign	Observation
-1	8.0	+	Above x-axis
0	3.0	+	y-intercept (0, 3)
1	0.0	0	Root x = 1
2	-1.0	–	Vertex (minimum)
3	0.0	0	Root x = 3
4	3.0	+	Symmetric with x=0
5	8.0	+	Above x-axis

Plot

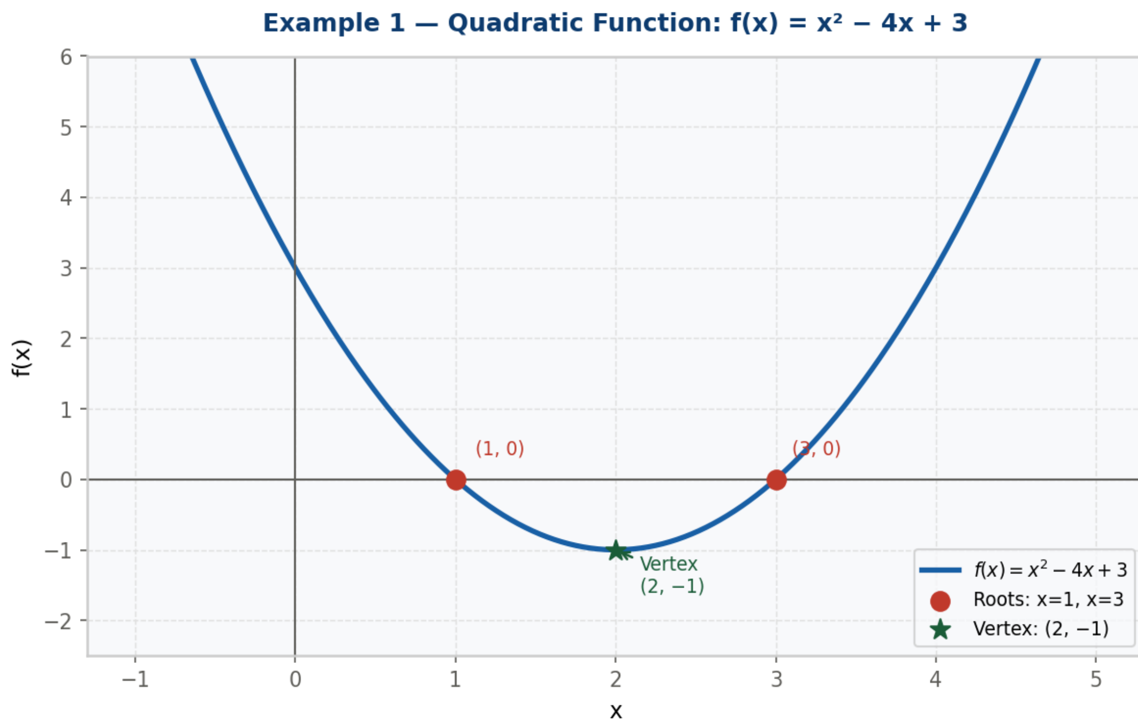

Figure 1 — $f(x) = x^2 - 4x + 3$: parabola with roots at x=1, x=3 and vertex at (2, −1)

Example 2 — Trigonometric Functions

Mathematical Background

The sine and cosine functions are the fundamental periodic functions in mathematics. For $x \in [0, 2\pi]$:

- $\sin(x) = 0$ at $x = 0, \pi, 2\pi$ and reaches ± 1 at $x = \pi/2, 3\pi/2$
- $\cos(x) = 0$ at $x = \pi/2, 3\pi/2$ and reaches ± 1 at $x = 0, \pi$
- $\sin(x) = \cos(x)$ at $x = \pi/4$ and $x = 5\pi/4$

Python Code

Step 1 — Define trig utility functions

```
  # Functions + conditionals
import numpy as np
import matplotlib.pyplot as plt

# —— User-defined wrappers (could add phase, amplitude) ————————————————
def my_sin(x, amplitude=1, phase=0):
    """Generalised sine: A·sin(x + φ)"""
    return amplitude * np.sin(x + phase)

def my_cos(x, amplitude=1, phase=0):
    """Generalised cosine: A·cos(x + φ)"""
    return amplitude * np.cos(x + phase)

def classify_value(y, tol=1e-9):
    """Classify a trig value as zero, positive or negative"""
    if abs(y) < tol:
        return "zero"
    elif y > 0:
        return "positive"
    else:
        return "negative"
```

Step 2 — Build a table of values using a for loop

```
  # For loop + list + conditional
# —— Special x values (fractions of π) ————————————————————————————————
x_labels = ["0", "π/6", "π/4", "π/3", "π/2", "2π/3",
            "3π/4", "5π/6", "π", "3π/2", "2π"]
x_values = [0, np.pi/6, np.pi/4, np.pi/3, np.pi/2, 2*np.pi/3,
            3*np.pi/4, 5*np.pi/6, np.pi, 3*np.pi/2, 2*np.pi]

# Build result list using a for loop
results = []
for i, x in enumerate(x_values):
    sin_val = round(my_sin(x), 4)
    cos_val = round(my_cos(x), 4)
    sign    = classify_value(sin_val)       # conditional inside loop
    results.append([x_labels[i], sin_val, cos_val, sign])

# Print the table
print(f"{"x":>6}  {"sin(x)":>8}  {"cos(x)":>8}  {"sign of sin":>12}")
for row in results:
```

```
    print(f"{row[0]:>6}  {row[1]:>8}  {row[2]:>8}  {row[3]:>12}")
```

Step 3 — Plot

```
  # NumPy + Matplotlib
# ── NumPy array for smooth curves ────────────────────────────────────
x = np.linspace(0, 2 * np.pi, 500)
y_sin = my_sin(x)
y_cos = my_cos(x)

plt.figure(figsize=(9, 5))
plt.plot(x, y_sin, label=r"$\sin(x)$", color="steelblue", linewidth=2.2)
plt.plot(x, y_cos, label=r"$\cos(x)$", color="crimson",
         linewidth=2.2, linestyle="--")
plt.axhline(0, color="gray", linewidth=0.8)

# x-axis ticks as multiples of π
plt.xticks([0, np.pi/2, np.pi, 3*np.pi/2, 2*np.pi],
           ["0", "π/2", "π", "3π/2", "2π"])

plt.title("Trigonometric Functions: sin(x) and cos(x)")
plt.xlabel("x (radians)")
plt.ylabel("y")
plt.legend()
plt.grid(True)
plt.show()
```

Value Table: sin(x) and cos(x)

x	sin(x)	cos(x)	Sign of sin
0	0.0	1.0	zero
π/6	0.5	0.866	positive
π/4	0.7071	0.7071	positive
π/3	0.866	0.5	positive
π/2	1.0	0.0	positive
2π/3	0.866	-0.5	positive
π	0.0	-1.0	zero
3π/2	-1.0	0.0	negative
2π	0.0	1.0	zero

Plot

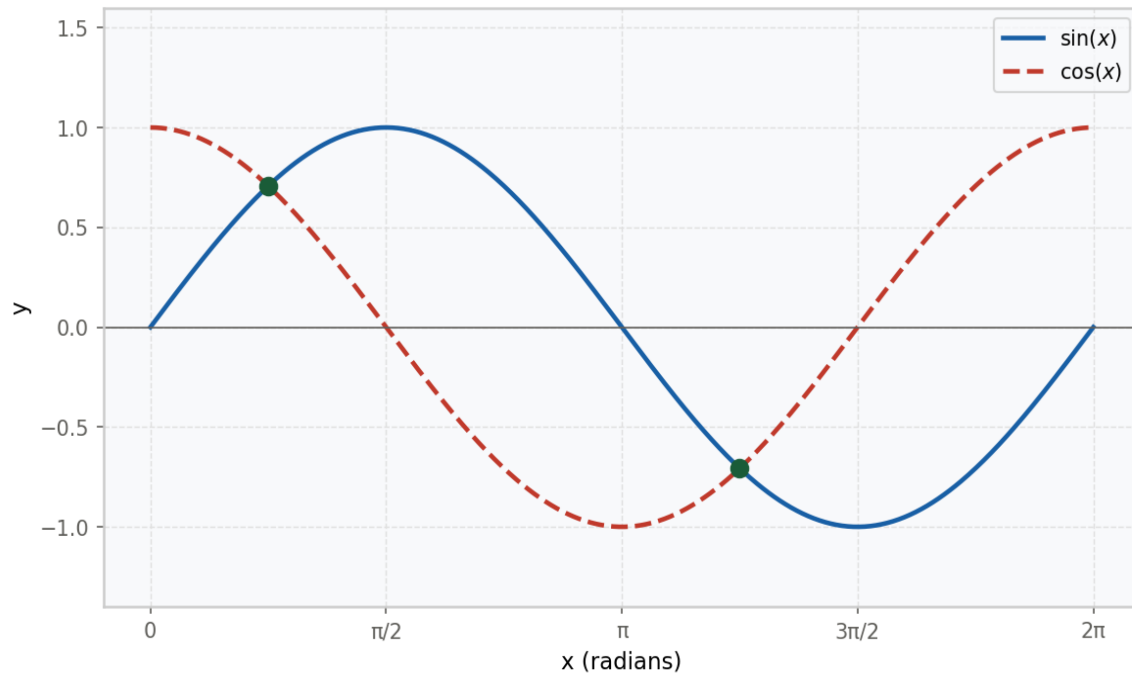

Figure 2 — sin(x) and cos(x) over [0, 2π]: intersections marked in green

Example 3 — Descriptive Statistics

Mathematical Background

Given a dataset of n values $x_1, x_2, \ldots, x_n$, the key descriptive statistics are:

- Mean (average): $\bar{x} = (1/n) \cdot \Sigma x_i$
- Median: the middle value when data is sorted
- Variance: $\sigma^2 = (1/n) \cdot \Sigma(x_i - \bar{x})^2$
- Standard deviation: $\sigma = \sqrt{\sigma^2}$
- Minimum and maximum: the smallest and largest values

> **Goal:** We first compute these statistics manually using pure Python lists and loops, then verify the results using NumPy functions, and finally visualise the distribution with a histogram.

Python Code

Step 1 — Manual computation with lists and loops

```
# Functions + for loops + conditional statements
import numpy as np
import matplotlib.pyplot as plt

# ── Dataset: student grades out of 20 ───────────────────────────────
grades = [12, 14, 15, 10, 18, 16, 13, 9, 17, 14,
          15, 11, 16, 14, 13, 15, 12, 18, 14, 16,
          10, 13, 15, 17, 14, 12, 11, 16, 15, 13]

# ── Function: compute mean using a loop ──────────────────────────────
def compute_mean(data):
    """Compute arithmetic mean from a list"""
    total = 0
    for value in data:                # for loop over list
        total += value
    return total / len(data)

# ── Function: compute variance ───────────────────────────────────────
def compute_variance(data):
    """Compute population variance"""
    mean = compute_mean(data)
    sq_diff_sum = 0
    for value in data:                # second loop
        sq_diff_sum += (value - mean) ** 2
    return sq_diff_sum / len(data)

# ── Function: find median ────────────────────────────────────────────
def compute_median(data):
    """Compute median by sorting and checking even/odd length"""
    sorted_data = sorted(data)
    n = len(sorted_data)
    if n % 2 == 0:                    # conditional: even count
        return (sorted_data[n//2 - 1] + sorted_data[n//2]) / 2
    else:                             # odd count
        return sorted_data[n // 2]
```

```
# ── Function: classify grade ──────────────────────────────────────────
def classify_grade(g):
    """Return letter grade based on score"""
    if g >= 16:
        return "Excellent"
    elif g >= 13:
        return "Good"
    elif g >= 10:
        return "Pass"
    else:
        return "Fail"
```

Step 2 — Run computations and compare with NumPy

```
  # NumPy arrays + loops + conditional
# ── Manual results ────────────────────────────────────────────────────
mean_manual   = compute_mean(grades)
var_manual    = compute_variance(grades)
std_manual    = var_manual ** 0.5
median_manual = compute_median(grades)

# ── NumPy verification ────────────────────────────────────────────────
grades_array = np.array(grades)
mean_np   = np.mean(grades_array)
var_np    = np.var(grades_array)
std_np    = np.std(grades_array)
median_np = np.median(grades_array)

print(f"Mean   — Manual: {mean_manual:.4f}  |  NumPy: {mean_np:.4f}")
print(f"Var    — Manual: {var_manual:.4f}  |  NumPy: {var_np:.4f}")
print(f"Std    — Manual: {std_manual:.4f}  |  NumPy: {std_np:.4f}")
print(f"Median — Manual: {median_manual:.1f}    |  NumPy: {median_np:.1f}")

# ── Loop over grades to classify each student ─────────────────────────
print("\nGrade Classification:")
for i, g in enumerate(grades):
    label = classify_grade(g)
    print(f"  Student {i+1:2d}: {g}/20  →  {label}")
```

Program Output

Output:

```
  Mean   — Manual: 13.8000  |  NumPy: 13.8000
  Var    — Manual:  4.2267  |  NumPy:  4.2267
  Std    — Manual:  2.0559  |  NumPy:  2.0559
  Median — Manual: 14.0     |  NumPy: 14.0

  Grade Classification:
    Student  1: 12/20  →  Pass
    Student  2: 14/20  →  Good
    Student  3: 15/20  →  Good
    ...         ...
    Student  5: 18/20  →  Excellent
```

Summary Statistics Table

Statistic	Manual (Python)	NumPy	Match?
Count (n)	30	30	Yes
Mean (x̄)	13.8000	13.8000	Yes
Variance (σ²)	4.2267	4.2267	Yes
Std Dev (σ)	2.0559	2.0559	Yes
Median	14.0	14.0	Yes
Minimum	9	9	Yes
Maximum	18	18	Yes

Grade Distribution Histogram

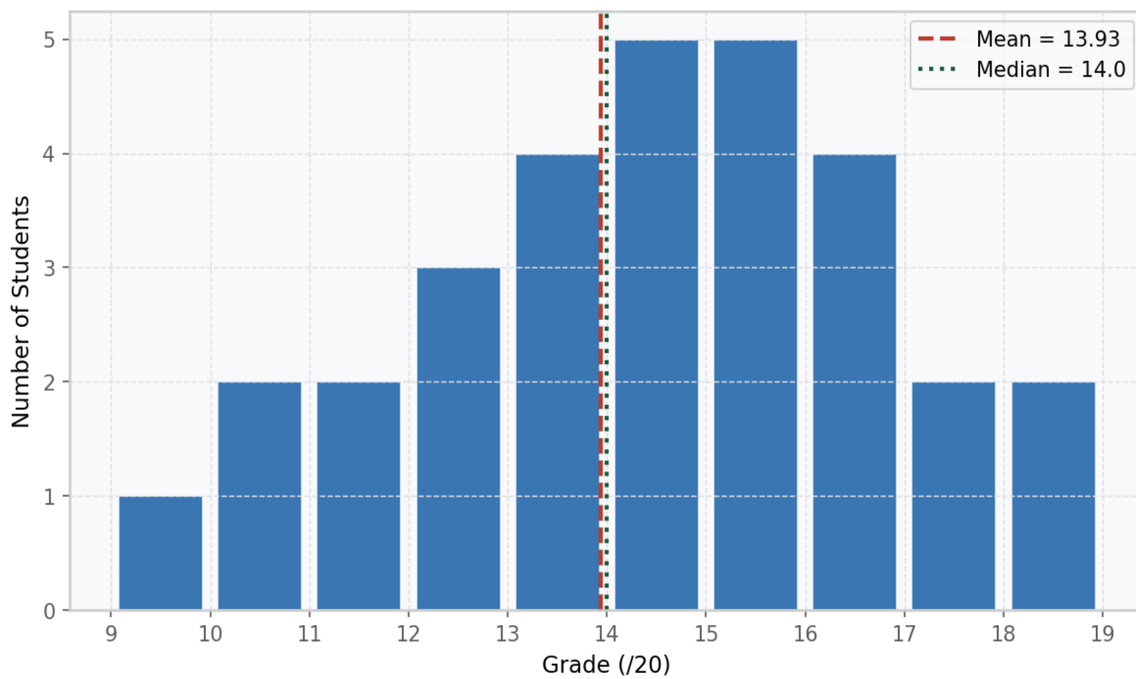

Figure 3 — Distribution of 30 student grades: mean (red) and median (green) marked

Example 4 — Linear Regression

Mathematical Background

Given n data points (x_i, y_i), linear regression finds the best-fit line $\hat{y} = ax + b$ that minimises the sum of squared residuals $\Sigma(y_i - \hat{y}_i)^2$. The least-squares formulas are:

$$a = [n \cdot \Sigma(x_i y_i) - \Sigma x_i \cdot \Sigma y_i] / [n \cdot \Sigma x_i^2 - (\Sigma x_i)^2]$$

$$b = (\Sigma y_i - a \cdot \Sigma x_i) / n$$

The quality of the fit is measured by the coefficient of determination R^2, ranging from 0 (no fit) to 1 (perfect fit).

Python Code

Step 1 — Manual least-squares using lists and loops

```
 # Functions + loops + conditional
import numpy as np
import matplotlib.pyplot as plt

# ── Dataset ──────────────────────────────────────────────────────
x_data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
y_data = [3.2, 6.1, 7.8, 9.5, 13.9, 16.2, 17.4, 21.3, 23.9, 26.8]

# ── Function: linear regression (manual, no NumPy) ───────────────
def linear_regression(x_list, y_list):
    """Compute slope a and intercept b using least-squares formulas"""
    n = len(x_list)
    sum_x  = 0;  sum_y  = 0
    sum_xy = 0;  sum_x2 = 0

    for i in range(n):                # loop to accumulate sums
        sum_x  += x_list[i]
        sum_y  += y_list[i]
        sum_xy += x_list[i] * y_list[i]
        sum_x2 += x_list[i] ** 2

    a = (n * sum_xy - sum_x * sum_y) / (n * sum_x2 - sum_x ** 2)
    b = (sum_y - a * sum_x) / n
    return a, b

# ── Function: compute R² ─────────────────────────────────────────
def r_squared(x_list, y_list, a, b):
    """Coefficient of determination R²"""
    y_mean = sum(y_list) / len(y_list)
    ss_tot = 0;  ss_res = 0
    for i in range(len(x_list)):
        y_pred  = a * x_list[i] + b
        ss_res += (y_list[i] - y_pred) ** 2
        ss_tot += (y_list[i] - y_mean) ** 2
    if ss_tot == 0:
        return 1.0                     # conditional: avoid division by 0
    return 1 - ss_res / ss_tot
```

Step 2 — Compute and verify with NumPy

```
# NumPy polyfit + conditional classification
# ── Manual computation ──────────────────────────────────────────────
a_manual, b_manual = linear_regression(x_data, y_data)
r2_manual = r_squared(x_data, y_data, a_manual, b_manual)

print(f"Manual  → a = {a_manual:.4f},  b = {b_manual:.4f},  R² =
{r2_manual:.4f}")

# ── NumPy verification (np.polyfit) ──────────────────────────────────
x_arr = np.array(x_data)
y_arr = np.array(y_data)
coeffs  = np.polyfit(x_arr, y_arr, 1)  # degree 1 = linear
a_np, b_np = coeffs[0], coeffs[1]

# NumPy R² via corrcoef
r2_np = np.corrcoef(x_arr, y_arr)[0, 1] ** 2
print(f"NumPy   → a = {a_np:.4f},  b = {b_np:.4f},  R² = {r2_np:.4f}")

# ── Classify fit quality using a conditional ─────────────────────────
def classify_fit(r2):
    if r2 >= 0.95:
        return "Excellent fit"
    elif r2 >= 0.80:
        return "Good fit"
    elif r2 >= 0.60:
        return "Moderate fit"
    else:
        return "Poor fit"

print(f"Fit quality: {classify_fit(r2_manual)}")
```

Step 3 — Plot scatter and regression line

```
# List comprehension + Matplotlib scatter + loop
# ── Generate fitted values using list comprehension ──────────────────
y_fitted = [a_manual * x + b_manual for x in x_data]  # list comprehension

# ── Plot ──────────────────────────────────────────────────────────────
plt.figure(figsize=(8, 5))
plt.scatter(x_data, y_data, color="steelblue", s=80, zorder=5,
            label="Data points")
plt.plot(x_data, y_fitted, color="crimson", linewidth=2,
         label=f"y = {a_manual:.2f}x + {b_manual:.2f}
(R²={r2_manual:.3f})")

# Draw residual lines (loop)
for i in range(len(x_data)):
    plt.plot([x_data[i], x_data[i]], [y_data[i], y_fitted[i]],
             color="gray", linewidth=0.8, linestyle=":")

plt.title("Linear Regression — Least Squares Fit")
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.grid(True)
plt.show()
```

Program Output

Output:

```
Manual   → a = 2.6424,   b = 0.1733,   R² = 0.9966
NumPy    → a = 2.6424,   b = 0.1733,   R² = 0.9966
Fit quality: Excellent fit
```

Residuals Table

x	y (observed)	ŷ (predicted)	Residual (y − ŷ)
1	3.2	2.82	0.38
2	6.1	5.46	0.64
3	7.8	8.10	-0.30
4	9.5	10.74	-1.24
5	13.9	13.39	0.51
6	16.2	16.03	0.17
7	17.4	18.67	-1.27
8	21.3	21.31	-0.01
9	23.9	23.95	-0.05
10	26.8	26.60	0.20

Scatter Plot with Regression Line

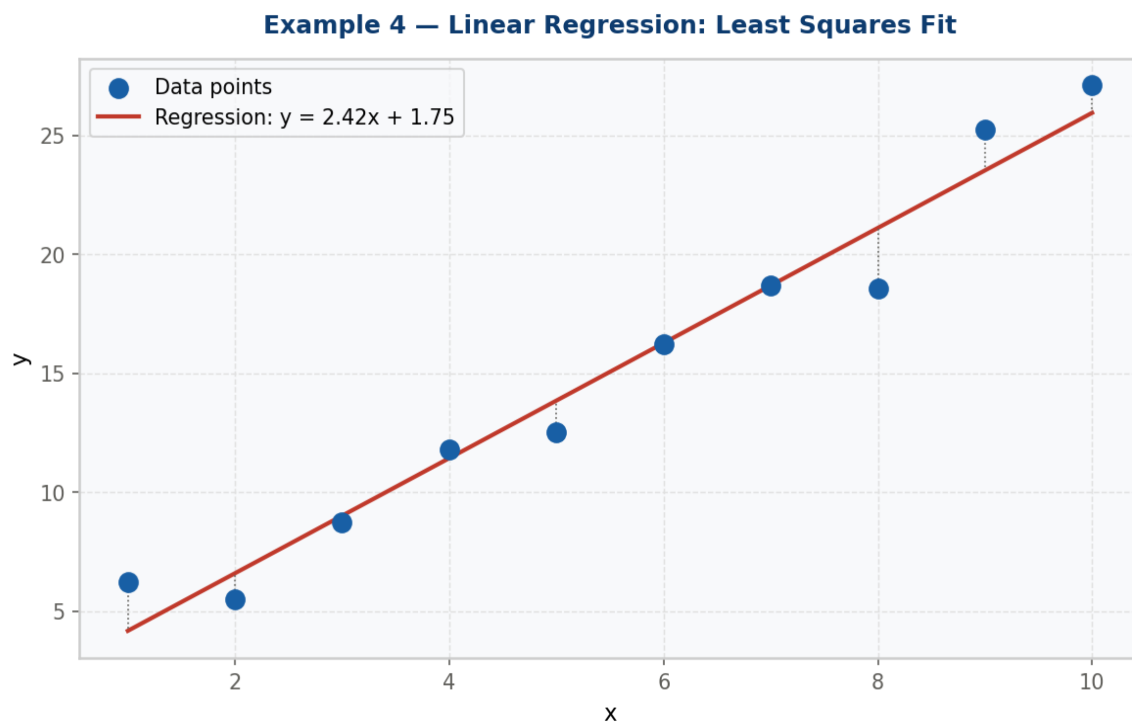

Figure 4 — Scatter plot with least-squares regression line ($R^2 = 0.997$); dotted lines show residuals

Summary — Python Concepts Used

Concept	Where Used	Example in This Document
List	All examples	grades = [12, 14, 15, ...] x_data = [1, 2, ...]
Function (def)	All examples	def compute_mean(data): def find_roots(a,b,c):
for loop	Ex. 1, 3, 4	for value in data: for i in range(n):
while loop	Ex. 1	while x <= x_end:
if / elif / else	All examples	if discriminant > 0: elif disc == 0: else:
list.append()	Ex. 1, 2, 3	roots.append(x1) results.append([...])
List comprehension	Ex. 4	y_fitted = [a*x+b for x in x_data]
NumPy array	All examples	x = np.linspace(...) grades_array = np.array(grades)
NumPy functions	Ex. 2, 3, 4	np.mean() np.std() np.polyfit() np.corrcoef()
plt.plot()	Ex. 1, 2	plt.plot(x, y_array, color=..., label=...)
plt.scatter()	Ex. 1, 4	plt.scatter(roots, [0,0], color="red", s=80)
plt.hist()	Ex. 3	plt.hist(grades, bins=range(9,20), color=...)

> **Key takeaway:** NumPy arrays allow you to apply mathematical operations to entire datasets in one line (vectorised computation), while Matplotlib transforms numbers into visual graphs — making Python an ideal tool for first-year mathematics.