Introduction to Programming with Python - Problem sheet

Solve the different problems by writing **Python programs**.

- Introduction to Programming with Python - Problem sheet
  - Module 02/03: Python basics and functions
    - Problem 1: Variables, arithmetic and comparison operators
    - Problem 2: Using loops and Loop Control
    - Problem 3: Preparing an invoice
    - Problem 4: Invoicing with Discount
    - Problem 5: Infinite loop and Loop Control
    - Problem 6: The *Parrot* program
    - Problem 7: Drawings
    - Problem 8: *Fizzbuzz*
    - Problem 9: *Guess my number* game
    - Problem 10 : Data manipulation with functions
    - Problem 11: Rolling dices and experiments
    - Problem 12: Band name generator
    - Problem 13: Body Mass Index Calculator
    - Problem 14: Automating Sales Commissions
      - Business Rules

    - Instructions

- Module 04/05 : Data analysis and visualization with Numpy and Matplotlib

    - Problem 1: Arrays and sales

    - Problem 2: The birthday paradox

    - Problem 3: Random walk in one dimension

        - Part 1: Our first experiment, a single random walk

        - Part 2: Exploring random walks statistically

    - Problem 4: Analyzing real-world population data with (only) NumPy and Matplotlib

        - Data extraction and preparation

        - Comparative line plot

        - Multi-panel visualization (with subplots)

        - Growth rate analysis

        - Generalization

        - Hints

    - Problem 5: Analyzing real-world population data with Pandas

- Additional exercises and problems to solve (practice!)

Module 02/03: Python basics and functions

Problem 1: Variables, arithmetic and comparison operators

1. **Write** a Python program that uses two variables `number1` and `number2` to store numbers: if the product of the two numbers is *less than or equal* to `1000`, print their *product*; otherwise print their *sum*.

Sample inputs and outputs:

- Case 1: `number1 = 20`, `number2 = 30`, output is `600`
- Case 2: `number1 = 40`, `number2 = 30`, output is `70`

2. **Modify** the program so that it asks the user to enter `number1` and `number2` using the `input()` function. (*Hint*: Remember to convert the input string to a number with the *casting* function `int()`)
3. **Introduce a third variable** `LIMIT`, a constant, and use it in your comparison instead of *hardcoding* the number `1000`.
4. **Add a check**: if either number is equal to 0, print "Zero multiplication detected" instead of computing the result.
5. **Check** whether each number is *even* or *odd* using the modulus operator (`%`) and print the result (e.g., "20 is Even", "31 is Odd").
6. *Bonus*: **Wrap** your program in a *while loop* to *continuously* prompt the user for new pairs of numbers. The loop should stop when the user enters 0 for `number1`.

Output sample:

```
Enter number1: 10
Enter number2: 3
30 # 3*10 < LIMIT, we print the product
10 is Even
3 is Odd
Enter number1: 500
Enter number2: 10
510 # 500*10 > LIMIT, we print the sum
500 is Even
10 is Even
Enter number1: 0 # end the program
```

Problem 2: Using loops and Loop Control

1. **Write a for loop** that displays the following output in the console:

```
1
2
3
4
5
6
7
8
9
10
```

2. **Write** a for loop that displays the following output in the console:

```
0
2
4
6
8
12
14
```

Note: Notice that 10 is skipped in the output sequence!

3. **Write** a for loop that displays the following output in the console:

```
1
2
4
8
16
32
64
128
256
```

4. **Write** a for loop that displays the following output in the console:

```
-1
1
-1
1
-1
1
-1
1
```

5. **Write** a for loop that displays the following outputs in the console:

```
"Yes!"
"No!"
"Yes!"
"No!"
"Yes!"
"No!"
```

Then, **modify** this loop to display the following output:

```
"Yes!"
"No!"
"Yess!"
"Noo!"
"Yesss!"
"Noooo!"
```

6. **Rewrite** all the previous loops using a **while loop** this time.

Problem 3: Preparing an invoice

We want to write a program that calculates and prints the details of an invoice.

1. **Declare a variable** set to the VAT rate percentage (20%). Choose the appropriate name and formatting.

2. **Declare a variable** that will store the subtotal amount in euros (before tax). Choose a relevant name in *snake_case*.

3. **Declare a variable** that will store `True` if VAT is applicable, or `False` if not.

4. **Write a procedure** that calculates the total amount including tax based on whether VAT applies or not for one invoice. Sample inputs:

- Invoice 1: Subtotal = 15000.50 EUROS, VAT applicable = No

- Invoice 2: Subtotal = 2500.00 EUROS, VAT applicable = Yes

5. **Display** the results in the console **using the following format**:

```
VAT=20%

Invoice {} :
Subtotal   : {} EUROS
Tax Amount : {} EUROS
Total (incl. taxes): {} EUROS
```

> **{ }** are *placeholders*: fill with appropriate variables and values.

6. Use the `input()` function to ask the user for invoice details (invoice number, subtotal and VAT applicable). Then, print the invoice details. Keep asking the user for next invoice details. To quit the program, the user must leave the invoice number blank.

```
To quit the program, leave invoice number blank (press Enter)
Enter invoice number: 71
Enter invoice subtotal: 1200
VAT applies: False

Invoice details:
VAT=20%

Invoice 71 :
Subtotal   : 1200.00 EUROS
Tax Amount : 00.00 EUROS
Total (incl. taxes): 1200.00 EUROS
```

Problem 4: Invoicing with Discount

Write a Python program named **`discount.py`** that calculates the final payable amount for a purchase based on a tiered discount system.

The program must prompt the user to enter a *net price* (excluding tax) and compute the corresponding *total price* including tax (using a constant VAT rate of 20%).

It then applies a *discount rate* based on the calculated *total price* including tax:

- 0% for an amount strictly less than €1,000

- 1% for an amount greater than or equal to €1,000 and strictly less than €2,000

- 3% for an amount greater than or equal to €2,000 and strictly less than €5,000

- 5% for an amount greater than or equal to €5,000

Finally, the program must **display** the details using the exact format shown below.

```
Enter net price (excl. VAT): 2500

VAT-inclusive price  3000.00
discount             90.00
net to pay           2910.00
```

Problem 5: Infinite loop and Loop Control

1. **Write** a Python program that runs an *infinite loop* using **`while`** and print “infinite loop” at each iteration. Terminate its execution manually using **`Ctrl+C`** (interrupt).

2. **Modify** the program so that it stops the loop **after the 7th iteration** using the `break` statement.

Output sample:

```
Infinite loop (0)
Infinite loop (1)
Infinite loop (2)
Infinite loop (3)
Infinite loop (4)
Infinite loop (5)
Infinite loop (6)
```

Problem 6: The *Parrot* program

Write a Python program `parrot.py` that continuously echoes the user's input. The program should run indefinitely until the user types "exit" to quit.

Sample output:

```
python parrot.py #run the program
hello #input
hello #output
Stop repeating what I say!
Stop repeating what I say!
exit #end the program
```

Problem 7: Drawings

1. **Write** a Python program named `triangle.py` that prompts the user to enter the size of a triangle to draw (the number of lines), up to a maximum of 20 lines:

- If the user enters a size greater than 20, the program must print an error message and terminate.

- Otherwise, the program draws the triangle using asterisks (`*`).

Sample inputs and outputs:

Valid case:

```
Enter the triangle size: 4

*
**
***
****
*****
```

Invalid case:

```
Enter the triangle size: 25
Error: The maximum allowed size is 20 lines.
```

2. **Building on the logic developed in the previous program**, **write a Python program** named **`half_diamond.py`** that prompts the user to enter the size of the upper half of a half diamond, up to a maximum of 10 lines, and print it.

Sample inputs and outputs:

Valid case:

```
Enter the maximum width: 4

-
**
---
****
---
**
-
```

> Note the alternation of hyphens (**`-`**) and asterisks (**`*`**) to draw each line

Invalid case:

```
Enter the maximum width: 25
Error: The size must be between 1 and 20 lines.
```

3. **Write** a program named **`rectangle.py`** that :

    1. Prompts the user to enter the dimensions (width and height) of the rectangle

    2. Prints the rectangle using asterisks (*) to the standard output

Modify the **`rectangle.py`** program to :

- **swap** height and height if necessary so that the rectangle is **always displayed horizontally** (largest dimension as the width)

- calculates and display its area (in arbitrary units)

Sample output:

```
Enter the dimensions of the rectangle: 3 10
The area of the rectangle is equal to 30 units

**********
**********
**********
```

4. *Bonus*: **Building on the logic developed in the `half_diamond.py` program**, **write a Python program** named **`diamond.py`** that prompts the user to enter the size of the upper half of a diamond, up to a maximum of 20 lines, and print it.

```
Enter the upper size of the diamond: 4

   *
  ***
 *****
*******
 *****
  ***
   *
```

Problem 8: *Fizzbuzz*

FizzBuzz (before being widely used by recruiters for technical interviews) is a counting game designed to help children learn division. The rules are simple: players count up to a predetermined positive integer N, applying the following rules:

- If the number is divisible by 3 (such as 9), replace it with “Fizz”;

- If it is divisible by 5 (such as 20), replace it with “Buzz”;

- If it is divisible by both 3 and 5 (such as 15), replace it with “Fizz Buzz”;

- Otherwise, simply say (*print*) the number.

Write a program that outputs the sequence for the game, where `N` represents the upper limit provided by the user.

Sample output:

```
Welcome to FizzBuzz. How high do you want to play? 15

1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
10
11
Fizz
13
14
Fizz Buzz
```

Problem 9: *Guess my number* game

Write the *“Guess My Number”* program in Python, a *classic* procedural program!

The game is simple: the computer generates a random number (integer) between 1 (included) and 100 (excluded) and the player has to guess it in minimum trials.

- For random generation, use the **`random`** module to select a *secret* integer between 1 (included) and 100 (excluded).

- For user input, use the **`input()`** function

Try to find the process, the different steps *before* coding! Use comments to help you design the process.

Hint: To help you, this a *pseudo-code* of the game:

```
1. Generate a random `secret` number
2. *Game loop*, repeat until the guess is correct:
   1. Ask user for a `guess`
   2. Print a message with a hint:
      *"It's greater than {guess} !"* if guess < secret
      *"It's lower than {guess} !"* otherwise
   3. Increment the number of attempts
3. Print a congratulations message with the total number of attempts:
  "Congratulations, the secret number was {secret}. It took you {attempts}
        attempts to guess this number."
4. End the program.
```

Problem 10 : Data manipulation with functions

Here are two data lists: one contains a series of numbers, the other a series of letters:

```
numbers = [1, 5, 32, 27, 12, -5, 100, -1.5, 230];
letters = ['a', 'z', 'b', 'u', 'r', 'g'];
```

For each expressed requirement, **write a function** that:

1. Returns the *sum* of the numbers contained in the numbers list;
2. Returns the *average* of the numbers contained in the numbers list;
3. Returns the *maximum* and *minimum* values of the numbers contained in the numbers list;
4. Returns the concatenation of the letters contained in the letters list;
5. Prints for each letter, whether it is a vowel or a consonant;
6. Returns the list of numbers with its elements sorted from smallest to largest;
7. Returns the list of letters with its elements sorted in alphabetical order;
8. Bonus: Returns the median value of the numbers contained in the numbers list.

Use list and string methods

Problem 11: Rolling dices and experiments

1. **Write** a function `d6` that simulates a single roll of a six-sided die.

2. Using this function, **simulate** rolling a six-sided die 50 times (or rolling 50 dices simultaneously!). **Print** the number of times a 6 was rolled, along with the empirical probability p of rolling a 6. (Recall that the probability of an event is defined as the number of successful outcomes divided by the total number of trials.)

3. Assuming the die is fair, **what** theoretical value $p_{\text{theoretical}}$ should you *expect*?

4. **Simulate** a series of n die rolls, where n ranges from 10 to 100,000, multiplying n by 10 at each step. For each value of n, **display** the calculated probability p of rolling a 6 with 3 digits for the decimal part.

Expected output:

```
n=10     p=..
n=100    p=..
n=1000   p=..
n=10000  p=..
# etc.
```

5. **Observe** the evolution of p as n increases. Does the empirical probability stabilize around $p_{\text{theoretical}}$? Which fundamental theorem of probability theory does this experiment illustrate?

Problem 12: Band name generator

Write a program that allows the user to generate random band names, using the `random` package. The system displays the names as a list. A band name follows this template: `The {adjective} {noun}` (e.g., *The Last Biscuits*, *The Midnight Llamas*). The program contains two list of 10 adjectives and 10 nouns (singular or plural).

Write a program with the **following functions**:

1. `find_adjectives()` and `find_nouns()`: functions that **retrieve** the list of adjectives and nouns from the “database”.

2. A function that **generates the band names**. It must takes as input the number of band names to generate and both lists of adjectives and nouns. **Find** an appropriate name for the function and parameters. It must return a list of generated band names

3. A function that **displays the band names** generated by the previous function. **Find** an appropriate signature for this function.

4. Use your functions to generate band names at each run

Problem 13: Body Mass Index Calculator

Write a small program that ask the user for its height (in cm) and weight (in kg) and computes its *Body Mass Index* (BMI):

$$\mathrm{BMI} = \frac{\mathrm{w}}{\mathrm{h}^2}$$

where w is the weight in kilograms and h is the height in meters.

Print the result with a small report:

- if BMI < 16, print “Severe thickness”

- if 16 < BMI < 18.5, print “Underweight”

- if 18.5 < BMI < 25, print “Normal”

- if 25 < BMI < 30, print “Overweight”

- if BMI > 30, print “Obese”

See all categories, you can add them all!

Problem 14: Automating Sales Commissions

You are a Finance and Business Analyst. At the end of each quarter, the HR and Sales Management teams calculate *performance commissions and bonuses* for the sales force.

Currently, this calculation is done manually in Excel by copying and pasting formulas for every single salesperson. This process is slow, prone to human error, and risky when sales policy rules change.

Your task is to write a reusable Python program that automates this process according *to the company’s business rules* (see below).

Business Rules

- **Base Commission Rate**: Every salesperson earns a standard base commission of 8% on their total sales revenue

- **Performance Tier Bonus**:

  - *Tier 1*: If sales revenue is 50 000 EUROS or below, no additional bonus rate applies (0%).

  - *Tier 2*: If sales revenue is between greater than 50 000 EUROS, they receive an additional 2% bonus rate on their total revenue (effective rate = 10%).

- **Customer Satisfaction (CSAT)**: each salesperson receives a *Customer Satisfaction score* rated from 1 to 5:

  - If score is below 3.0, a 500 EUROS penalty is deducted from their total payout (due to poor service quality).

  - If score is 4.5 or higher, a 1 000 EUROS accelerator bonus is added to their total payout.

Instructions

1. **Create** a new program `sales_comissions.py`

2. **Write a function** `calculate_total_payout(sales_amount, csat_score)` that returns the final total payout (commission + tier bonus + CSAT adjustment).

3. **Test** your function with the following sales reps:

    1. Alex: 40 000 EUROS revenue, CSAT: 4.8

    2. Beatrix: 75 000 EUROS revenue, CSAT: 2.5

    3. Charlie: 120 000 EUROS revenue, CSAT: 4.6

Represent sales force as a *list of dictionaries*. Each salesperson is represented by a *revenue* and a CSAT *score*.

4. **Export** this function into a dedicated module named `sales.py`. **Import** this module into your program. What is the benefice of doing this?

5. Imagine that next quarter, management decides to *increase the base rate to 9%*. **How many lines of code do you need to change** if you used a function compared to manual copy-pasting throughout your entire program?

Module 04/05 : Data analysis and visualization with Numpy and Matplotlib

Problem 1: Arrays and sales

You manage a local cinema. You have a record of ticket sales over 5 days (Monday through Friday) for 3 different ticket categories: Adult, Child, and Senior. Your data is stored in a 2D NumPy array called sales, where:

- Each *row* represents a day of the week: `['Mon', 'Tue', 'Wed', 'Thu', 'Fri']`

- Each *column* represents a ticket category: `['Adult', 'Child', 'Senior']`

Here is the starting code:

```
import numpy as np

# Rows: Days (Mon to Fri)
# Columns: Categories (Adult, Child, Senior)
sales = np.array([
    [120, 45, 30],  # Monday
    [95,  30, 25],  # Tuesday
    [110, 50, 40],  # Wednesday
    [150, 60, 35],  # Thursday
    [220, 90, 75]   # Friday
])
```

1. **Print** the *shape* of the array and *explain* what the dimensions represent.

2. **Print** how many *Child tickets* were sold on Wednesday? (using index coordinates)

3. **Extract** all ticket sales for *Friday* as a 1D array using *slicing*.

4. **Extract** the sales for the *Adult* category (1st column) across all 5 days. What was the **total** number of *Adult* tickets sold during the week?

5. **Compute** the *total number of tickets sold* per day (sum across columns, `axis=1`). Which day had the highest total attendance?

6. **Compute** the total revenue per category if *Adult* tickets cost 12 EUROS, *Child* tickets cost 8 EUROS, and Senior tickets cost 10 EUROS. (Hint: element-wise multiplication with a 1D price array!)

7. **Extract** the sales of the *Adult* category (1st column) for the first three days of the week (Monday, Tuesday and Wednesday)? **Compute** the percentage of the week's total sales made during the first three days. **Using** `np.round` function, **print** the percentage rounded to one decimal place.

8. **Find** on which days the *Child* sales are below 50. Convert the indices into day names with a list `days = ["Mon", "Tue", "Wed", "Thu", "Fri"]`.

9. **Store** the total number of tickets sold per day in a variable called `total_per_day`. **Count** the days on which we exceeded our target of 200 tickets. (Hint: use a boolean mask.)

10. Our daily sales targets vary from one day to the next. They are stored in the array `targets = np.array([180, 150, 200, 220, 350])` (Monday to Friday). **Print** the number of days on which the target was exceeded. (Hint: compare `total_per_day` and `targets` element by element, then use a boolean mask.)

11. **Plot a bar graph** of the total revenue by category computed previously. **Save** this graph to the file `total_revenue_bar.png` by using the `savefig` function.

Problem 2: The birthday paradox

The Birthday problem is a well-known and interesting problem. It asks for the *probability* that, in a set of N randomly chosen people, *at least two* will share the *same birthday*. As we will see, the result is quite counterintuitive, showing that our brains are *bad* at estimating *probabilities*!

The objective is to **compute** and **plot** the probability $P(N)$ that at least *two people* in a group of N people *share the same birthday*.

> Hypotheses: we assume 365 days in a year, an equal birthday probability for each day. Of course, $N > 0$ (the group must not be empty!)

It is easier to calculate the complementary probability $\bar{P}(N)$ that *no* two people share a birthday:

$$P(N) = 1 - \bar{P}(N)$$

> By definition $\bar{P}(N) + P(N) = 1$

Thus, for a group of size N, we get the following expression:

$$\bar{P}(N) = \frac{365}{365} \times \frac{364}{365} \times \frac{363}{365} \times ... \times \frac{365 - N + 1}{365}$$

Knowing $\bar{P}(N)$, we can compute $P(N)$ for each value of N.

1. **Write** a new program called **`birthday_problem.py`**
2. With numpy, **compute** $P(N)$ for $N = 1$ to $N = 100$.
3. **Plot** $P(N)$ with matplotlib, using **`plot()`** method. **Add** x-axis and y-axis labels
4. What is the *smallest* group size N where the probability of a shared birthday exceeds 50%?
5. What about 99%?

Problem 3: Random walk in one dimension

Random walk is a simple *process* and yet really surprising and fundamental.

In a 1D random walk, a *particle* (or someone, or something!) starts at some position, let say $x = 0$. At each step t, it moves one step forward (+1) or one step backward (-1) with equal probability ($p = 0.5$).

> The moving thing plays *Heads or Tails?* at each step! Sometimes, the random walk is called the "drunkard's walk"!

Part 1: Our first experiment, a single random walk

1. **Create** a new program **`random_walk.py`**.
2. **Define** a constant **`N_STEPS = 1000`**. It stores the *size* of the walk (the number of steps to take).
3. **Generate** **`N_STEPS`** random steps of $+1$ or -1 using **`np.random.choice()`**. Take the time to read the documentation of this function to understand how to use it correctly. **Store** the result in a variable called **`steps`**.

4. **Compute** the position of the particle at each step using `np.cumsum()` (*cumulative sum*). **Store** the resulting array in a variable called `positions`.

5. Using Matplotlib, **plot** `positions` versus step number.

6. **Add** the following to the plot:

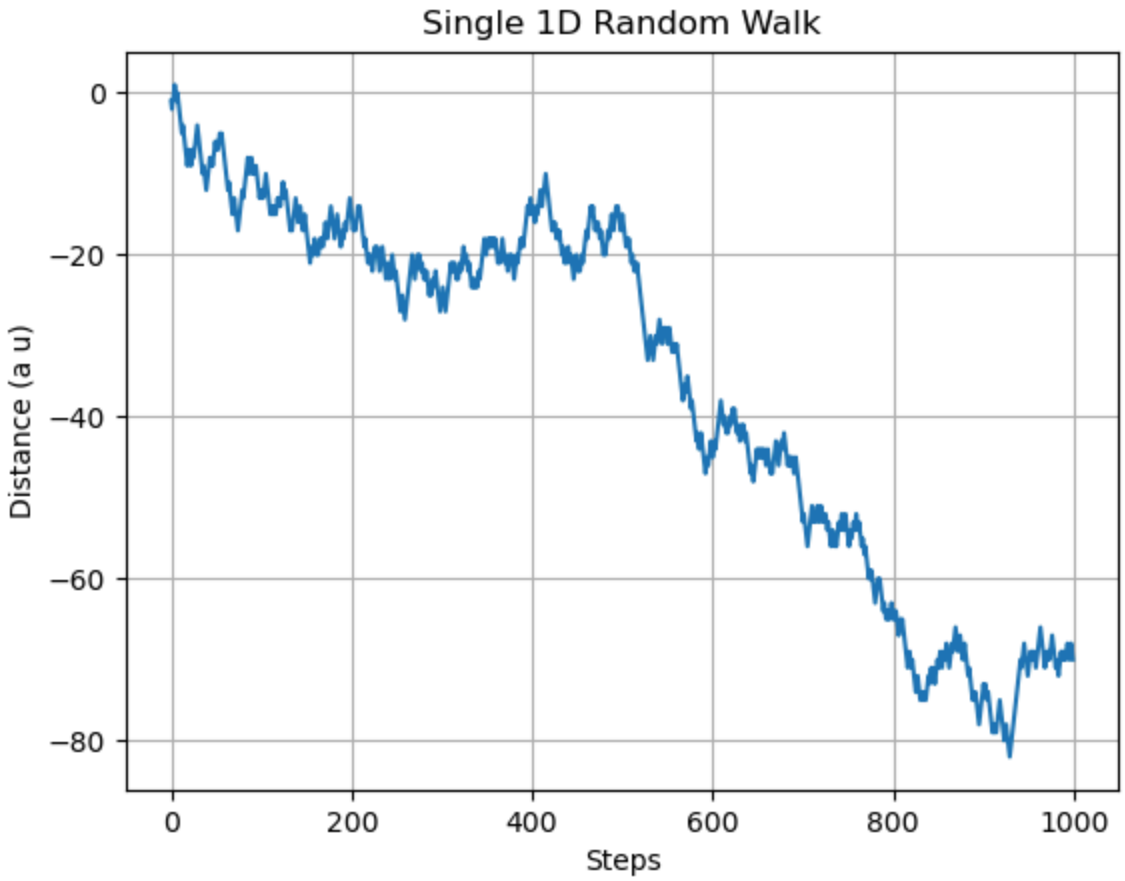
    1. an x-axis label: `"Steps"`

    2. a y-axis label: `"Distance"`

    3. a **title**: `"Single 1D random walk"`

    4. a **grid**

Sample output:



7. Mathematically, what is the *expected position* $\langle x_N \rangle$ of a particle after N steps?

8. Still using Matplotlib, **write** a loop to simulate and plot 100 *independent* walks on the same figure. What do you observe?

9. We are interested in the *distance* traveled by each walker after N steps, regardless of its direction (left or right). For each walk, take the *final position* x_N and compute its *square* $d_N^2 = x_N^2$. **Store** the results in a variable called `squared_distances`.

10. **Plot** a histogram of `squared_distances`.

11. **Compute** the *mean squared distance* $\langle d_N^2 \rangle$ over all walkers. What do you notice?

12. **Run** your program again with increasing walk sizes: N = 2 000, 5 000, then 10 000. For each value of N, **compute** the mean squared distance. Compare $\langle d_N^2 \rangle$ with N. What do you notice?

Part 2: Exploring random walks statistically

To understand the *average* behavior of random walks, we have to simulate *a lot* of them. This is where things become *really* interesting!

1. **Create** another program called `random_walks.py`. This way, you can still use and read the first program you wrote.

2. **Define** two constants: `N_WALKS = 500` and `N_STEPS = 1000`. `N_WALKS` is the number of random walks simulated (or *experiments*), and `N_STEPS` is the number of steps in each walk.

3. This time, **create** a *2D* array of *shape* `(N_WALKS, N_STEPS)` called `steps`, containing random steps of $+1$ or -1. Each row stores *one* random walk.

4. **Compute** the cumulative positions of each walk (along `axis=1`) using `np.cumsum()`, and **store** the result in a variable called `positions`.

5. We want to see where each walker ends up after its N steps. **Extract** the final position of each walker using 2D array slicing, and **store** the result in a variable called `last_positions`.

6. As before, we are interested in the distance traveled by each walker after N steps, regardless of its direction (left or right). From `last_positions`, **compute** the squared distances d_N^2 and **store** them in a variable called `squared_distances`.

7. **Compute** the *mean squared distance* $\left\langle d_N^2 \right\rangle$ over all walkers.

8. We now want to study the influence of the walk size N. **Write** a loop that runs the simulation for different values of N: 1000, 1500, 2000, 3000, 4000 and 5000.

9. Let's visualize our results. **Create** a figure with a 1×2 layout of subplots using `plt.subplots(1, 2)`:

   - *Left*: plot the mean squared distance $\left\langle d_N^2 \right\rangle$ as a function of N.

   - *Right*: plot the histograms of the *final positions* `last_positions` x_N for each value of N (set transparency`alpha = 0.4`)

What do you notice?

> **Why random walks?** A random walk is a simple model of how atoms and molecules move in a gas or a liquid. A particle undergoes countless collisions with its neighbors, each one changing its direction unpredictably. Its trajectory therefore looks like a sequence of random steps, a motion known as *Brownian motion*. We cannot predict the path of a single particle, but the *statistics* of many paths are remarkably regular: the mean squared displacement $\langle x^2 \rangle$ follows a simple law with time. This is the signature of *diffusion*, which Einstein analyzed in 1905, making a quantitative prediction that could be tested experimentally. Later, Jean Perrin used Einstein's result to *experimentally prove* the existence of atoms, a work for which he received the Nobel Prize in Physics in 1926! In this exercise, each walker represents one particle, and each step represents a time interval between collisions (in a simplified 1D model).

Problem 4: Analyzing real-world population data with (only) NumPy and Matplotlib

We have access to real-world dataset containing historical population statistics.

1. Download the world population dataset provided by World Bank Group
2. **Decompress** the archive and **rename** the file `API_SP.POP.TOTL_DS2_en_csv_v2_441303.csv` to `world_pop.csv`
3. **Open** the file with Excel or LibreOffice to *inspect* the data.

You can see that:

- The first 5 lines contain metadata/headers that must be skipped.
- Country codes are located in column index 1 (e.g., "FRA" for France, "KOR" for South Korea).

- Population counts starting from the year 1960 to 2025 onwards begin at column index 5.

Data extraction and preparation

1. **Open** and **read** `world_pop.csv` using Python's built-in csv module.
2. **Skip** the first 5 header lines.
3. **Extract** the population data for France (FRA) and South Korea (KOR).
4. **Convert** the raw string values into floating-point numbers representing population in millions (divide by 10^6)
5. **Create** a NumPy array for the years ranging from 1960 to 2025.

Comparative line plot

1. **Create** a line chart using `matplotlib.pyplot` to compare the population evolution of France and South Korea on the same plot from 1960 to 2025.
2. **Ensure** your plot includes:
    1. Clear axis labels (Year and Population (in millions)).
    2. A descriptive title.
    3. A legend distinguishing the two countries.

Multi-panel visualization (with subplots)

Create a single figure window containing **two subplots** stacked vertically (top and bottom) using `plt.subplots()`:

1. **Top subplot** (Line Chart): Plot the comparative population evolution of France and South Korea from 1960 to 2025. Include clear axis labels, a title, and a legend.
2. **Bottom subplot** (Bar Chart): Plot a bar chart illustrating the population evolution of France over the same years. Include appropriate labels and a title.

Growth rate analysis

1. **Calculate** the year-over-year *growth rate* (as a percentage) for both France and South Korea using NumPy. Formula for growth rate:

$$\text{Growth Rate } (\%) = \frac{P_t - P_{t-1}}{P_{t-1}} \times 100$$

where P_t is the population in the current year and P_{t-1} is the population in the previous year.

2. **Plot** and **compare** the year-over-year percentage growth rates for both France and South Korea.

3. **Extend your program** to automatically determine and **print**:

    1. The **minimum growth rate** across both countries and all years, along with the corresponding country and year.

    2. The **maximum growth rate** across both countries and all years, along with the corresponding country and year.

Generalization

Define a constant variable at the top of your script containing a list of two country codes to analyze, for example:

```
COUNTRIES_TO_ANALYZE = ["FRA", "KOR"] # you should be able to change both
        countries here easily
```

Your program should be able to **reproduce above analysis when both countries are changed** without any changes.

Hints

- Use **`np.diff()`** or slicing techniques (**`population[1:]`**) for efficient growth rate calculations.

- Remember that your growth rate array will have *one fewer element* than your years array (starting from 1961).

- Use **`np.min()`**, **`np.max()`**, and **`np.where()`** to locate the extreme values easily.

Problem 5: Analyzing real-world population data with Pandas

The World Bank publishes the total population of every country since 1960. In this problem, you will compare the **population** and the **year-over-year growth rate** of two countries. We have access to real-world dataset containing historical population statistics.

1. Download the world population dataset provided by World Bank Group

2. **Decompress** the archive and **rename** the file **`API_SP.POP.TOTL_DS2_en_csv_v2_441303.csv`** to **`world_pop.csv`**

3. **Open** the file with Excel or LibreOffice to *inspect* the data. The first 4 lines of this file are not data (metadata). Each row is a country, and there is one column per year (from `1960` to `2025`).

4. **Load** `world_pop.csv` into a `DataFrame`, using the *country code* as the index (*row-label*)

> *Hint*: Use `read_csv()` parameters `skiprows` to skip the metadata lines and `index_col` to use `Country Code` as the index. Check the result with `df.head()`.

5. **Extract** the population of **France** (`FRA`) and **South Korea** (`KOR`) from 1960 to 2025 as **NumPy arrays** of floats.

> *Hint*: `df.loc[row_label, first_column:last_column]` selects one row and a range of columns, **both ends included**. The column names are strings (`"1960"`, not `1960`). Use the `pandas` method `.to_numpy(dtype=float)` to get a NumPy array of floats.

6. **Compute** the *year-over-year growth rate* (in %) for both countries. **Store** the results in two NumPy arrays (without using a `for` loop). The year-over-year growth rate (in %) is defined as follow:

$$\text{Growth Rate } (\%) = \frac{P_t - P_{t-1}}{P_{t-1}} \times 100$$

where P_t is the population in the current year and P_{t-1} is the population in the previous year.

> You need an Numpy array of years for the x axis. For example, `np.arange(1960, 2026)`. How many growth rates do you get for 66 years of data?

> NumPy operations (`-`, `/`, `*`) are applied **element by element** on arrays of the same size. Can you combine `p[1:]` and `p[:-1]` to apply the growth rate formula?

7. Create **one figure with two subplots** (one above the other):

   - *Top*: population of both countries from 1960 to 2025

   - *Bottom*: year-over-year growth rate of both countries

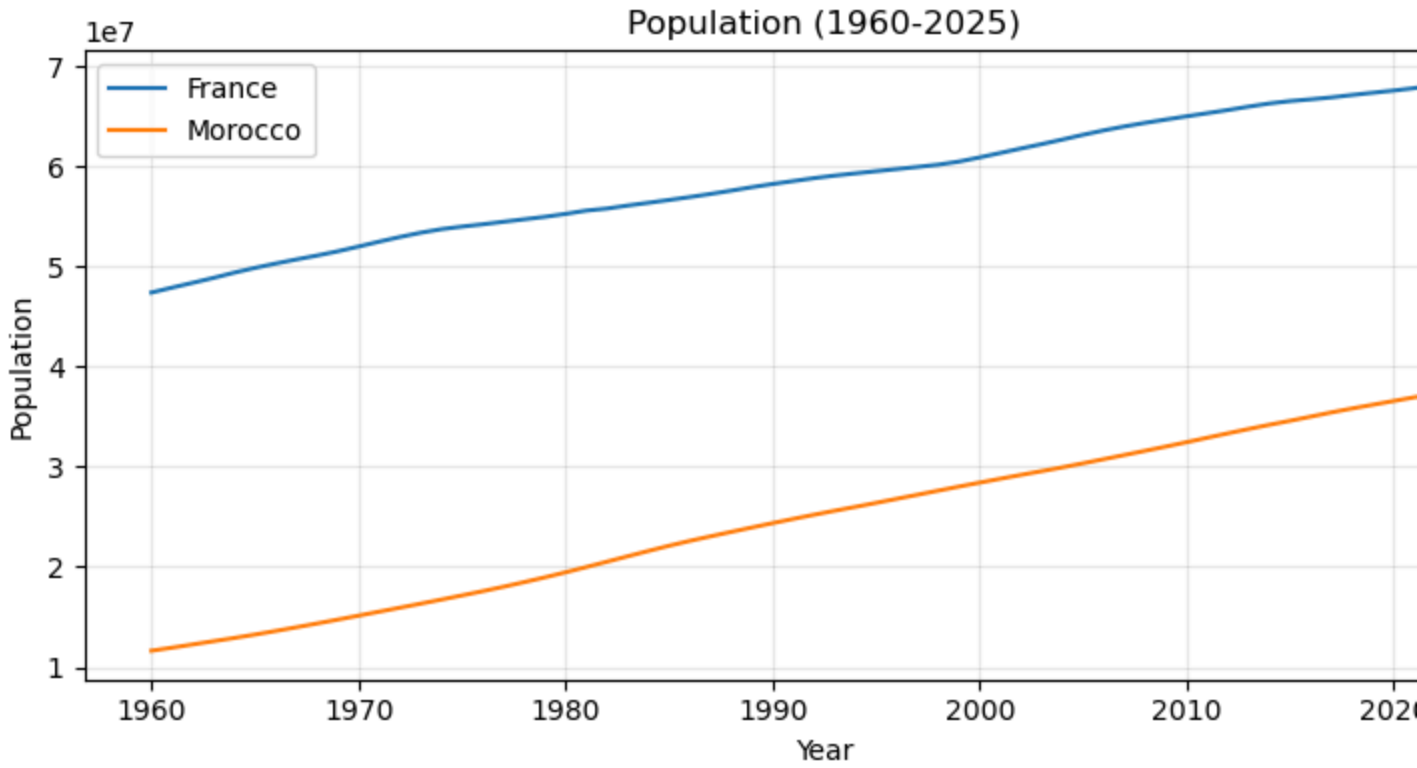
Each subplot must have a *title*, axis *labels*, a *legend* and a *grid*.

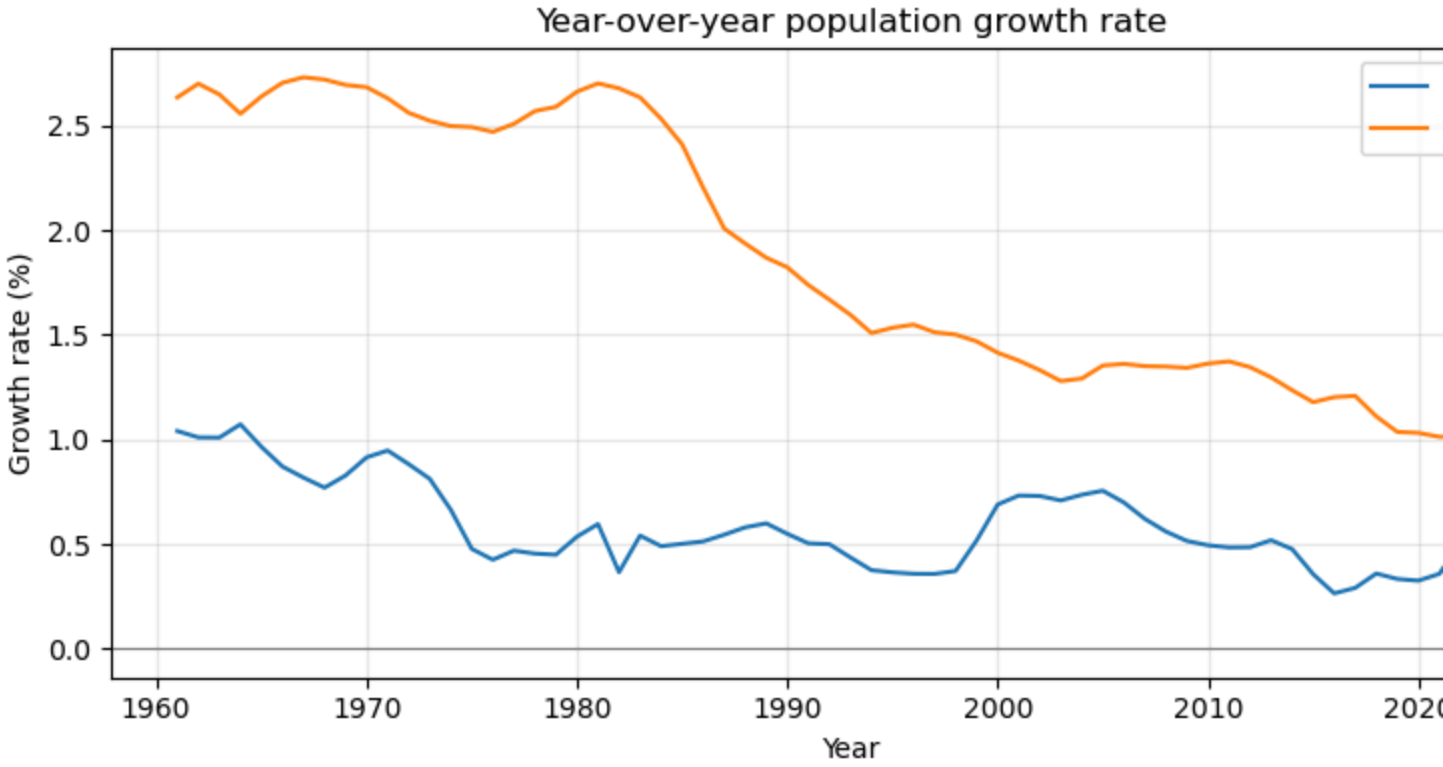
8. **Check** your results:

   - Your growth rates should be small numbers (a few percent at most, and often below 3%)

   - Both growth curves should have 65 points, starting in 1961

Example output with France and Morocco:





Bonus questions

1. Extend your program to automatically print:
    1. The **minimum** growth rate across both countries and all years, with the corresponding country and year
    2. The **maximum** growth rate across both countries and all years, with the corresponding country and year

Expected output format (values are placeholders):

```
Minimum growth rate: X.XX% (Country, Year)
Maximum growth rate: X.XX% (Country, Year)
```

> **`np.min(arr)`** returns the smallest value of an array, **`np.argmin(arr)`** returns the **position** of this smallest value (**`np.argmax`** for the largest). Use this position to find the corresponding year in your array of years

2. Restrict the analysis to the period 1990-2024 and observe what changes
3. Save the figure as an image with **`plt.savefig("population_growth.png")`** (before **`plt.show()`**)
4. Update your program to be able *to change countries easily* or to *add new countries* of your choice. You need to generalize what you have already done for two countries. For example, the user should be able to tell which countries he wants to analyze by defining the following variable: **`countries = {"FRA": "France", "KOR": "South Korea", "DEU": "Germany"}`**

Additional exercises and problems to solve (practice!)

- Coding Game (easy) Training Problems, solve some puzzles in an interactive environement in Python
- Practice Python Exercises & Challenges with Solutions [800+ Coding Questions], on PyNative
- *Ask me!*