PARIS PANTHÉON-ASSAS UNIVERSITÉ

TI101I – Programming in Python

FORT BOYARD SIMULATOR

In this project, you'll create interactive games in Python, using concepts like functions, conditions, loops, and JSON files to handle data. You'll also build a simple interface for users to interact with the game.

Intermediate deposit: Saturday 21/12/2024, noon on moodle

Final submission: Sunday 05/01/2025, noon on moodle

Defense: The week of 06/01/2025

Cherifa BEN KHELIL

EFREI – P1, P1-BN, P1-BDX, P1-Plus, P1-INT

Fort Boyard Simulator: Overcome Obstacles and Unlock the Treasure

In this module (TI101I), you will develop a simulator inspired by the famous TV game show *Fort Boyard*. The goal is to recreate an experience where a team of players must succeed in various challenges and collect three keys. Once the keys are gathered, the team can participate in the final challenge and attempt to unlock the treasure room.

Project Objectives

1. **Team Creation:** Allow the user to form a team of 1 to 3 players.
2. **Types of Challenges to Implement:** The program must offer the following 4 types of challenges:
    - **Mathematics:** (e.g., factorial calculations, equations)
    - **Chance:** (e.g., shell game, dice rolls)
    - **Logic:** (e.g., NIM game, Tic-Tac-Toe)
    - **Père Fouras' Riddle:** (A challenge based on riddles from the show)

For each challenge type, a minimum number of challenges must be included:

- At least 3 **Mathematics** challenges.
- 2 **Chance** challenges.
- At least 1 **Logic** challenge.

3. **Key Rewards:** During each challenge, a player is selected to participate. If the player succeeds, they win a key. The team can access the treasure room once they have collected 3 keys.
4. **Performance History and Saving (Optional - Bonus):** The performance history of players is saved in a text file. This file will allow the team to review their results at the end of the challenges.

Using JSON Files

In this project, you'll use JSON files to manage data. These files will contain real-world information about *Père Fouras' riddles* and the clues needed to access the treasure room, sourced from ***the Fan-FortBoyard.fr*** website.

Code Optimization and Personalization

Students are encouraged to:

- **Decompose** parts of the code to improve organization and clarity.
- **Modify or extend** the program as needed, providing a detailed explanation for each change made.
- **Add extra functions** to enhance the application and user experience.

Table of contents

1. Authorized native libraries and functions

The use of modules and predefined functions is strictly limited to those specifically authorized and listed below. It is forbidden to use other predefined functions, except in the following cases:

- ***List management :***
  - Add elements with `append()`.
  - Inserting elements with `insert()`.
  - Delete elements using `del` (also applicable to collections).
- ***Type conversion operations (cast) :***
  - Explicit conversion between types (e.g. `int()`, `str()`, etc.).
- ***File handling :***
  - Use the `open()` function to read or write to files.
  - Specific functions for reading and writing to a file (e.g. `write()`, `read()`, `readlines()`, etc.).
- **Message display :**
  - Use the `format()` function or `f-strings`.

1.1. The random library

The **`random`** module in Python offers functions for generating random numbers and performing random operations. In this project, you'll have the opportunity to use two functions:

a) The function random.randint(a, b): generates a random integer between a and b, including both ends. This means that calling `random.randint(1, 6)` will return a random integer between 1 and 6.

Example:

```
import random
random.seed(3) # Fixes the seed to make results reproducible.
print(random.randint(1, 6)) # Displays a random integer between 1 and 6 (here, always 2 with the seed fixed).
```

Output:

```
2
```

b) The random.choice(sequence) function: randomly selects an element from a sequence (such as a list, tuple or string). For example, if you have a list of ***names***, using `random.choice(names)` will return a name at random from this list. This is useful in games or simulations where a random choice is required.

Example:

```
import random
names = ['Alice', 'Bob', 'Charlie'] # List of names
random_choice = random.choice(names) # Choose a random name from the list
print(random_choice) # Displays 'Alice' , 'Bob' or 'Charlie' at random
```

1.2. The json library

The **JSON** (**J**ava**S**cript **O**bject **N**otation) format is a lightweight, human-readable format primarily used to exchange data between a server and a web application. It allows information to be structured in the form of key-value pairs, like a Python dictionary.
Here is an example of a JSON file named **data.json** :

```
[
{
"name": "Pierre",
"age": 30,
"city": "Paris",
"interests": ["reading", "sport", "music"]
},
{
"name": "Marie",
"age": 25,
"city": "Lyon",
"interests": ["cooking", "travel", "photography"]
}
 ]
```

This file consists of a list (delimited by square brackets []) containing two elements (each delimited by braces {} and separated by a comma). Each element corresponds to information about a person, organized in the form of key-value pairs for the attributes ***name***, ***age***, ***city*** and ***interests***.
To work with a JSON file in Python, you can use the **json** library. The **`json.load()`** function loads data from a JSON file and converts it into a Python structure such as a dictionary or list, depending on the organization of the data in the file, thus facilitating access to the information it contains.

Example:

If you want to load the ***data.json*** file from our example and use the data it contains, you can proceed as follows:

```
import json

# Open data.json file in read mode
with open('data.json', 'r', encoding='utf-8') as f:
    data = json.load(f) # Load JSON data into a Python structure

# Display the entire 'data' structure to view the loaded content
print(data)

# Display loaded data
for person in data:
    print("Name : {}".format(person['name']))
    print("Age : {}".format(person['age']))
    print("City : {}".format(person['city']))
    print("Interests : ")
    for interest in person['interests']:
        print("{}".format(interest))
    print() # Adds an empty line between persons
```

Output:

```
[{'name': 'Pierre', 'age': 30, 'city': 'Paris', 'interests': ['reading', 'sport', 'music']}, {'name': 'Marie', 'age': 25, 'city': 'Lyon', 'interests': ['cooking', 'travel', 'photography']}]
Name: Pierre
Age: 30
City: Paris
Interests :
reading
sport
music

Name : Marie
Age: 25
City: Lyon
Interests :
kitchen
travel
photography
```

The output begins by displaying the complete contents of the ***data.json*** file, which is a list composed of two elements (dictionaries). Each element represents a person, with this information (name, age, city and interests).

The code then displays the details of each person in a structured way:

- Name, age and city are displayed directly.
- Interests are listed under the heading "Interests".

For more details, see[1] : JSON Load in Python – GeeksforGeeks.

1.3. Other authorized native functions

a) The **abs()** function is a native Python function, part of the standard library. It returns the absolute value of a number.

Example :

```
number = -10
print(abs(number)) # Displays 10
```

b) The **lower()** function in Python converts all alphabetic characters in a string to lower case. It is part of the standard library and is used on a string to return a new string where all characters are lowercase. This method does not modify the original string, but returns a new, modified string.

Example:

```
text = "Hello World"
lowercase_text = text.lower()
print(lowercase_text)
```

Output :

```
hello world
```

c) **The split() function** in Python is used to split a string into several substrings according to a specified delimiter. By default, it uses a space as delimiter, but you can also specify another character as separator. It returns a list containing the resulting substrings.

Example1 :

```
text = "Python is a powerful language
words = text.split() # Default separation, i.e. by spaces
print(words)
```

Output:

```
['Python', 'is', 'a', 'powerful', 'language']
```

If you want to separate the string with another character, you can specify it as an argument:

[1] https://www.geeksforgeeks.org/json-load-in-python/

Example2 :

```
text = "1 ;2 ;3 ;4 ;5"
elements = text.split(";") # Semicolon separation
print(elements)
```

Output:

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

2. Project creation

2.1. First steps to start and manage your project in pairs

2.1.1. Instructions for organizing the group and using Git

This project must be carried out in **pairs** (only one pair is allowed for odd-numbered groups). Effective collaboration with your partner is essential to the success of this project.

You'll use **Git** and **GitHub** to manage the project. These tools will enable you to share code, version changes and track project progress collaboratively. **Before you , it's essential that each member creates an account on GitHub.**

Once your account has been created, you can **consult the document "User Guide - Using Git in PyCharm" available on Moodle**. This guide has been specially prepared to help you use these tools.

Make sure you share tasks fairly, communicate regularly and follow a schedule to ensure the project's success.

2.1.2. GitHub repository management

To get your project off to a good, carefully follow the steps below:

1. One of the members of the pair must initiate the project by creating the main repository on his or her **GitHub** account. This can be done directly on the **GitHub** platform or via **PyCharm** (refer to the user guide for details).
2. The repository must be named according to the following convention: "**pyfort-student1-name-student2-nameGroup**".
   Example: For a binomial composed of **Durant** and **Dupont**, belonging to group **INT1**, the repository should be called: **pyfort-durant-dupont-int1.**
3. Once the main repository has been created, the other member of the pair (or trio) must **clone** this repository on his or her computer via **PyCharm**.
4. The person who created the main repository must **add the other member of the binomial (or trinomial)** along with your **two lab instructors**, as **collaborators** on the **GitHub repository**. To do this, go to the **repository settings** on **GitHub**, then to the "**Collaborators**" section, and add the emails or **GitHub** usernames of the members and teachers.

5. Throughout the project, make sure you regularly **pull** the latest changes from the pair and push your own changes using Git **push**. This ensures that each member is working with the most up-to-date version of the code.

2.2. Project structure

creating your project by structuring your files and folders according to the tree structure shown below:

```
.
├── data
│   ├── PFRiddles.json
│   └── TRClues.json
├── pere_fouras_challenge.py
├── final_challenge.py
├── chance_challenges.py
├── logical_challenges.py
├── math_challenges.py
├── utility_functions.py
└── main.py
```

The **main.py** file will serve as the main entry point for the game, calling specific functions defined in the **utility_functions.py** module to manage the various stages of the game. Challenge-related functions will be distributed in specific modules as follows:

- **math_challenges.py**: Contains functions for math challenges, like calculations and solving equations.
- **chance_challenges.py**: Includes functions for games of chance, like the shell game or dice throwing.
- **logical_challenges.py**: Implements logic games, such as Battleship or tic-tac-toe.
- **pere_fouras_challenge.py**: Manages functions for the Père Fouras riddle challenge.
- **final_challenge.py**: Contains functions for the final game stage, where players guess a codeword to access the treasure room.

The last two challenges (Père Fouras and the treasure room) require the files ***PFRiddles.json*** and ***TRClues.json***, which are available on Moodle. Please download these files and place them in the data directory, as they contain the necessary data for these challenges.

3. Implementation

In this section, we'll detail the organization of the specific functions to be implemented in the project. Here are the key points to consider:

- **Function Structure:** Specific functions have been defined for this project. It's important to follow this structure for consistency, but feel free to add extra functions to improve readability and efficiency based on your own logic.
- **Choice of Games to Implement:** You will choose from several games based on the type of challenge. Each game comes with a difficulty level (low, medium, high). This level is a guideline and may change depending on your programming approach, so evaluate your options carefully before deciding.
- **Division of Tasks:** Clear task division is essential. Ensure each team member understands their responsibilities and the parts of the code they need to implement.

To make the project development easier, we recommend implementing the functions in the following order:

1. **Start with the simplest challenges**, namely the **mathematical** and **chance challenges**. These are relatively easier to implement and will enable you to quickly test the associated functions.
2. Continue with the implementation of your chosen **logic game**, the **Père Fouras riddles** and **the final challenge**. These challenges will progressively increase the complexity of the project, and you're free to tackle them in any order you like.
3. Finalize function implementation in the **utility_functions.py** module. Some of these functions can be developed earlier, particularly those that facilitate game and team management.
4. The **main.py** file should be the last to be completed, as it coordinates the calls to the various functions and manages the execution of the game as a whole.

3.1. The math_challenges.py module

In the module **math_challenges.py**, you must:

1. Choose and implement **3** of the **4 challenges** listed below. Some challenges only require implementation of the main function, while others also require specific sub-functions.

2. Implement the **math_challenge()** function, which randomly selects one of the challenges you've implemented and executes the corresponding function.

3.1.1. Factorial challenge (weak)

1. Implement a **factorial(n)** function that calculates the factorial of n (denoted n!). The factorial of n is defined as :
    - $0! = 1$
    - $n! = n \times (n-1) \times \cdots \times 1$ for $n > 0$

    For example, the factorial of 4 is calculated as follows: $4 \times 3 \times 2 \times 1 = 24$.

Example:

```
factorial(5) # Returns 120
```

2. Implement a function **math_challenge_factorial()** that generates a random number **n** between 1 and 10, then asks the player to calculate the factorial of **n**. The player's answer is compared with the value calculated by the **factorial(n)** function. If the answer is correct, a message indicating that the player has won a key is displayed, and the function returns **True**. Otherwise, the function returns **False**.

Example of user interface :

```
Math Challenge: Calculate the factorial of 5.
Your answer: 120
Correct! You win a key.
```

3.1.2. Linear equation test (weak)

1. Implement a function **solve_linear_equation()** that generates two random numbers **a** and **b** between 1 and 10, then solves the linear equation **ax + b = 0** for **x**. This function returns the values of **a**, **b** and the correct solution to the equation, i.e. the value of **x = -b/a**.
2. Implement a function **math_challenge_equation()** that calls the function **solve_linear_equation()** to obtain the generated values and the solution. The **math_challenge_equation()** function asks the player to solve the equation and compares the answer given with the correct solution. It returns **True** if the answer is correct, otherwise **False**.

Example of user interface:

```
Math Challenge: Solve the equation 4x + 6 = 0.
What is the value of x: -1.5
Correct! You win a key.
```

3.1.3. Prime Numbers challenge (average)

1. Implement two functions:
   - **is_prime(n):** checks whether **n** is a prime number. A number is prime if it is greater than 1 and has no divisor other than 1 and itself.
   - **nearest_prime(n):** returns the first prime number greater than or equal to **n**.
2. Implement the function **math_challenge_prime()** which generates a random number **n** between 10 and 20, then asks the player to find the prime number closest to **n**. The correct solution must be obtained by calling the function **nearest_prime(n)**. The function returns ***True*** if the player's answer is correct, otherwise it returns ***False***.

Example of user interface:

```
Math Challenge: Find the nearest prime to 14.
Your answer: 17
Correct! You win a key.
```

3.1.4. Math Roulette challenge (average)

Implement the **math_roulette_challenge()** function, which generates **five** random numbers between 1 and 20, then randomly selects an operation from ***addition***, ***subtraction*** and ***multiplication***.
Operations must be performed as follows:

- If the operation is addition, add all the five numbers.
- If the operation is subtraction, subtract all the five numbers in order.
- If the operation is multiplication, multiply all the five numbers together.

The player must calculate the result of the operation applied to the roulette numbers and give his answer. The function returns **True** if the answer is correct, indicating a win, and **False** otherwise.

Example of user interface:

```
Numbers on the roulette: [3, 5, 7, 8, 2]

Calculate the result by combining these numbers with addition

Your answer: 25

Correct answer! You've won a key.
```

3.1.5. math_challenge() function for random challenge selection

Implement an **math_challenge()** function that randomly chooses one of the mathematical challenges you have created:

1. Define a list called **challenges** and add the function names for each challenge as references (without parentheses), allowing you to call them later. In Python, this allows storing the functions themselves in the list to be executed dynamically.
2. Use the appropriate random function to randomly select a function from the list and assign it to the **challenge** variable.
3. Once the function is selected, call it by executing **challenge().** This will dynamically run the selected challenge and return the result (True or False) based on the player's response.

This method ensures that each call to the **math_challenge()** function executes a randomly selected challenge from the available options.

3.2. The chance_challenges.py module

The chance challenges and the logical challenges will be against the game master. In Fort Boyard, the game master oversees the challenges and guides the players through various tasks, while also acting as their opponent in certain challenges. You will implement games where the player competes directly against the game master.
In the **chance_challenges** module, you are required to implement the **two** challenges as well as the **chance_challenge()** function.

3.2.1. Shell game (weak)

The player must guess which of the three shells (A, B, or C) hides the key. They have two attempts to find it. On each attempt, the key is randomly placed under one of the shells. This game will be implemented by creating the **shell_game()** function, based on the following pseudo-algorithm:

Function : shell_game() → Boolean
Local variables[2] :

- a list containing three elements 'A', 'B' and 'C
- an integer representing the attempt number
- a variable storing the value of the randomly selected shell
- a variable storing the player's choice

Begin

1. Initialize the list of shells.
2. Display a welcome message explaining the rules of the game and specifying the number of attempts allowed.
3. Display the letters representing the shells, so that the player can make a choice.
4. **For each attempt** from 1 to 2, do the following:
   5. Randomly select the letter corresponding to the shell from the list and store it in a variable.
   6. Display the number of remaining attempts for the player.
   7. Ask the player to choose a shell (A, B or C) and convert the choice to uppercase if needed.
      a. If the choice is valid (present in the list of shells), then:
         - If the choice matches the one randomly selected:
            - Display a message indicating that the key has been found under the shell.
            - Return **True**.
         - If not, display that the player was unsuccessful in this attempt.
      b. Otherwise, if the choice is invalid:
         - Indicate this with a message.

[2] **The list of local variables is incomplete.**

8. Increment the number of attempts.

9. **After the two attempts**:
   - Display a message indicating that the player has lost and reveal which shell the key was under.
   - Return **False**.

End.

3.2.2. Rolling dice game (average)

The player and game master each roll two dice. The first to roll a 6 wins the game, with a maximum of three attempts. Implement the **roll_dice_game()** function using the following approximate pseudo-algorithm version:

Function: roll_dice_game() → Boolean

Local variables[3] **:**

- A variable (or constant) representing the number of allowed attempts (3)
- A tuple containing two randomly generated integer values between 1 and 6, representing the player's dice.
- A tuple containing two randomly generated integer values between 1 and 6, representing the game master's dice.

Begin

1. For each attempt from 1 to 3, do:
   2. Display number of attempts remaining
   3. Display a message inviting the player to roll the dice by pressing the "Enter" key.
      (*after the player presses the "Enter" key, the function generates the two random numbers*)
   4. Store in a tuple two values randomly generated by the appropriate function from the random library. Each value must be an integer between 1 and 6, representing the results of the dice roll.
   5. Display the values obtained by the player.
   6. If one of the two values is equal to 6, then :
      - Show that the player has won the game and the key.
      - Return True

   (*then it's the game master's turn*)
   7. Store two randomly generated values in a tuple.
   8. Display the values obtained by the game master.
   9. If one of the two values is equal to 6, then :
      - Show that the game master has won the game.
      - Return False

[3] **The list of local variables is incomplete.**

```
    10.   If no 6 is obtained :
        • Display a message indicating that you are moving on to the
          next attempt.
  (*After three tries, none of the players has won*)
  11.     Show that no player has scored a 6 after three tries and
    that it's a draw.
  12.     Return False
End.
```

3.2.3. chance_challenge() function for random challenge selection

Implement the **chance_challenge()** function using the same approach as the **math_challenge()** function:

1. Create a list called **challenges** and add function references to it.
2. Use a random selection method to pick a function from the list and assign it to the **challenge** variable.
3. Call the selected function by executing **challenge().**

3.3. The logical_challenges.py module

You are required to implement **one of the three proposed games**. The grade for this part will be based on the game you select. For each game, you will receive a breakdown of the required functions and an example of the expected execution to help guide you.

3.3.1. The game of Nim (Weak)

The player and the game master (AI) take it in turns to remove 1, 2 or 3 sticks from a total of 20. Whoever removes the last stick loses the game. The game master follows a strategy based on multiples of 4 to maximize his chances of victory.

List of functions to be implemented:

- **display_sticks (n):** Takes as parameter an integer **n**, representing the number of sticks remaining, and displays this number using bars (|), with each bar corresponding to one stick.
- **player_removal (n):** Takes an integer **n**, representing the number of sticks remaining. This function asks the player to choose how many sticks to remove (1, 2, or 3), ensuring that the input is valid. It returns the number of sticks removed by the player.
- **master_removal (n):** Takes as parameter an integer **n**, representing the number of sticks remaining. It applies an AI strategy based on the remainder of the division of n by 4. The AI (the game master) chooses the number of sticks to remove according to this strategy, with the goal of forcing the opponent to lose. It returns the number of sticks removed by the game master.
- **nim_game():** This is the main function of the stick game. It will call the functions defined above to manage the flow of the game as follows:
    1. Initialize the number of sticks to 20.
    2. Uses a Boolean variable to indicate that it's the player's turn to .

3. As long as the number of sticks is not zero:
    - Displays the remaining sticks.
    - An iteration represents one turn, either that of the player or that of the game master.
    - The number of sticks removed is retrieved by calling the corresponding function.
    - Update and display the number of sticks remaining after each round.
4. The game continues, alternating between players, until no sticks remain.
5. The player who removes the last stick is declared the loser.
6. The function returns True if the player wins, and False otherwise.

Example of the result of running the game:

```
Remaining sticks: ||||||||||||||||||||
Player's turn.
How many sticks do you want to remove (1, 2, or 3)? 3
Remaining sticks: ||||||||||||||||
The game master removes 1 stick(s).
Remaining sticks: |||||||||||||||
Player's turn.
How many sticks do you want to remove (1, 2, or 3)? 3
Remaining sticks: ||||||||||||
The game master removes 1 stick(s).
Remaining sticks: |||||||||||
Player's turn.
How many sticks do you want to remove (1, 2, or 3)? 3
Remaining sticks: |||||||||
The game master removes 1 stick(s).
Remaining sticks: ||||||||
Player's turn.
How many sticks do you want to remove (1, 2, or 3)? 3
Remaining sticks: |||||
The game master removes 1 stick(s).
Remaining sticks: ||||
Player's turn.
How many sticks do you want to remove (1, 2, or 3)? 3
Remaining sticks: |
The game master removes 1 stick(s).
The game master removed the last stick. The player wins!
```

3.3.2. The tic-tac-toe game (Medium)

The player and the game master (AI) compete in a classic game of tic-tac-toe. The first player to align three identical symbols (*horizontally*, *vertically* or *diagonally*) wins the game. The game master uses a simple strategy to try and win or block the player.

List of functions to be implemented:

- **display_grid (grid)** : The function takes as parameter a 3x3 grid, represented by a 2D list, and displays it. Each grid cell can be empty (" ") or contain a symbol ('X' or 'O'), with a separator between lines.
- **check_victory(grid, symbol) :** This function takes a 2D grid list and a symbol string ('X' or 'O') as parameters. It examines the rows, columns, and diagonals of the grid to check if the symbol has won, returning True if it has, otherwise False.
- **master_move(grid, symbol) :** The function takes as parameters a 2D grid list, representing the current state of the game grid, and a symbol string, representing the game master's symbol. This function determines the game master's move first to win, then to block the player if necessary, and finally plays a random move if none of these actions is required. It returns a tuple (row, column) corresponding to the coordinates of the chosen square.
- **player_turn(grid) :** This function takes a 2D grid list as a parameter, representing the current state of the game. It allows the player (symbol 'X') to place their symbol in an empty square. The player is prompted to enter the coordinates of the square in row, column format. Before placing the symbol, the function checks if the chosen square is empty. If the square is already occupied, the player is asked to select a different square. The grid is then updated with the player's move.
- **master_turn (grid) :** The function takes as parameter a 2D grid list, representing the current state of the game grid. This function directly modifies the grid by placing the game master's move ('O'). The game master plays according to the strategy defined in the **master_move()** function.
- **full_grid(grille) :** The function takes as parameter a 2D grid list, representing the current state of the game grid. It returns *True* if the grid is complete (no empty cells), otherwise *False*. This function is used to check whether the game has ended in a draw or can continue.
- **check_result(grid) :** This function takes a 2D grid list representing the current game state as a parameter. It returns True if the game has ended, meaning that either player 'X' or the game master (player 'O') has won, or if the game has ended in a draw. It checks the results using the **check_victory()** and **full_grid()** functions. If no end-of-game condition is met, it returns False, meaning that the game continues.
- **tictactoe_game() :** This is the main function that orchestrates the entire game. It takes no parameters and returns True if player 'X' wins the game, and False otherwise (i.e., if the game master wins or a draw occurs).
  Here's how it works:
  1. The grid (2D list) is initialized with empty spaces.
  2. A loop is defined to alternate turns between the player and the game master:
     - The **player_turn()** function is called to manage the player's move. It allows the player to place their symbol ('X') on the grid.

- The **check_result()** function is called to check whether the game is over (win or draw). If player 'X' wins, the function returns True.
- If the game hasn't ended after the player's turn, the **master_turn()** function is called to allow the game master (symbol 'O') to make their move.
- The **check_result()** function is called again to check if the game master wins or if the grid is complete. If the game master wins or a draw is detected, the function returns False.

3. The loop continues until a player wins or the grid is complete, indicating a draw. If either of these events occurs, the game ends.

Example of the result of running the game:

```
Welcome to Tic-Tac-Toe against the game master!
To play, enter your move coordinates in the form row,column (e.g., 1,2)
  |   |
---------
  |   |
---------
  |   |
---------
Player X, it's your turn. Where do you want to place your symbol? 1,1
Game master's turn (O)...
X |   |
---------
O |   |
---------
  |   |
---------
Player X, it's your turn. Where do you want to place your symbol? 2,2
Game master's turn (O)...
X |   |
---------
O | X |
---------
  |   | O
---------
Player X, it's your turn. Where do you want to place your symbol? 1,3
Game master's turn (O)...
X | O | X
---------
O | X |
---------
  |   | O
---------

Player X, it's your turn. Where do you want to place your symbol? 3,1
```

```
X | O | X
---------
O | X |
---------
X |   | O
---------
Player X has won!
```

3.1.1. Battleship game (High)

The player places two boats on a grid and tries to guess the position of his opponent's (the game master) two boats. The winner is the player who manages to hit all the opponent's boats. Each player has two grids: one for the position of his boats and another for recording his shots. The game alternates turn between players, each trying to hit the other's boats. The game ends when both of a player's boats are sunk.

List of functions to be implemented:

- **next_player(player)** : This function takes the index of the current player (0 or 1) as a parameter and returns the index of the next player, either 0 or 1.
- **empty_grid() :** This function generates and returns an empty grid as a 2D list of size 3x3. Each cell in the grid is initialized with an empty space (" ").
- **display_grid(grid, message) :** This function takes two parameters:
    - ***grid:*** A 2D list representing the grid.
    - **message:** A string that provides context for the grid, such as "Reminder of the history of the shots you have made:" or "Discover your game grid with your boats:".

  It displays the given message followed by the grid, line by line. Grid cells are separated by vertical bars (|), and a border is drawn below the grid for clarity. This function is used to display either the player's shot history or the positions of boats placed at the start of the game.
- **ask_position():** This function prompts the user to enter a valid position on the grid in the format row,column. It validates the input to ensure it is within bounds and correctly formatted. If valid, it returns a tuple (row, column) corresponding to the position.
- **initialize()**:The function returns the player's grid with the two boats placed. This function creates an empty grid for the player's boats. The user is prompted to place two boats by entering positions in the form row,column. If a position is valid and unoccupied, the function places a boat at that cell by marking it with '**B**'. Once both boats are placed, the function returns the player's grid with the boats positioned.
- **turn(player, player_shots_grid, opponent_grid) :** This function handles a single turn of the game, where either the player or the game master takes a shot. It takes as parameters:
    - ***player:*** The index of the current player (0 or 1).
    - ***player_shots_grid:*** The shooting grid of the current player.

- ***opponent_grid:*** The opponent's grid containing the positions of his boats.

- It displays the shot history, requests a position from the player (or generates a random position for the game master), and updates the shot grid with 'x' for a hit and '.' for a miss. The function modifies both the player's and the opponent's grids but does not return any value.
- **has_won(player_shots_grid):** This function takes the player's shot grid as a parameter. It checks whether all the opponent's boats have been sunk by counting the number of squares marked with 'x' on the shot grid. The function returns *True* if all boats have been sunk, otherwise it returns *False*.
- **battleship_game() :** This function orchestrates the entire Battleship game. It follows these steps:
  1. Displays a message explaining the rules, specifying that each player must place 2 boats on a 3x3 grid.
  2. Initializes the player's boat grid using the **initialize()** function and displays it.
  3. Creates an empty grid for the game master and randomly places 2 boats.
  4. Initializes empty shooting grids for both the player and the game master.
  5. Alternates turns between the player and the game master in a loop:
     - The game begins with the player (player 0).
     - On each turn, the **turn()** function is called for the current player to take a shot.
     - After each turn, the **has_won()** function checks if either the player or the game master has sunk all opponent boats.
     - If a player wins, a victory message is displayed, and the function returns True (if the player wins) or False (if the game master wins).
     - If no one has won, the game alternates to the other player using the **next_player()** function.
  6. The game ends as soon as one player sinks all the opponent's boats.

Example of the result of running the game:

```
Each player must place 2 boats on a 3x3 grid.
Boats are represented by 'B' and missed shots by '.'. Sunk boats are
marked by 'x'.
Place your boats:
Boat 1
Enter the position (row,column) between 1 and 3 (e.g., 1,2): 1,1
Boat 2
Enter the position (row,column) between 1 and 3 (e.g., 1,2): 2,3
Here is your game grid with your boats:
| B |   |   |
|   |   | B |
|   |   |   |
-------------
It's your turn to shoot!
```

```
History of your previous shots:
|   |   |   |
|   |   |   |
|   |   |   |
-------------
Enter the position (row,column) between 1 and 3 (e.g., 1,2): 3,1
Hit, sunk!

It's the game master's turn:
The game master shoots at position 1,2
Splash...

It's your turn to shoot!
History of your previous shots:
|   |   |   |
|   |   |   |
| x |   |   |
-------------
Enter the position (row,column) between 1 and 3 (e.g., 1,2): 2,1
Splash...

It's the game master's turn:
The game master shoots at position 2,2
Splash...

It's your turn to shoot!
History of your previous shots:
|   |   |   |
| . |   |   |
| x |   |   |
-------------
Enter the position (row,column) between 1 and 3 (e.g., 1,2): 2,2
Splash...

It's the game master's turn:
The game master shoots at position 1,2
Splash...

It's your turn to shoot!
History of your previous shots:
|   |   |   |
| . | . |   |
| x |   |   |
-------------
Enter the position (row,column) between 1 and 3 (e.g., 1,2): 1,3
Splash...

It's the game master's turn:
The game master shoots at position 1,3
```

```
Splash...

It's your turn to shoot!
History of your previous shots:
|   |   | . |
| . | . |   |
| x |   |   |
-------------
Enter the position (row,column) between 1 and 3 (e.g., 1,2): 1,2
Splash...

It's the game master's turn:
The game master shoots at position 1,1
Hit, sunk!

It's your turn to shoot!
History of your previous shots:
|   | . | . |
| . | . |   |
| x |   |   |
-------------
Enter the position (row,column) between 1 and 3 (e.g., 1,2): 3,3
Hit, sunk!
The player won!
```

3.4. The pere_fouras_challenge.py module

The Père Fouras riddles is a legendary challenge from the TV game show Fort Boyard, where players must solve a riddle posed by Père Fouras, the guardian of the fort. The challenge tests the player's logical thinking, as they must answer the riddle correctly in order to obtain a clue or key.

This module aims to recreate this challenge by using riddles extracted from the ***PFRiddles.json*** file. The goal is to implement two functions that simulate an encounter with Père Fouras: The function will present a riddle to the player, who has three attempts to guess the correct answer. If the player answers correctly, they win a key. If all three attempts are incorrect, a defeat message and the solution are displayed.

List of functions to be implemented:

- **load_riddles(file):** This function takes the path of the file ***'PFRiddles.json'*** as a parameter. It opens this file in read mode, loads its contents, then converts it into a list of JSON dictionaries. Each dictionary must represent riddle, with key-value pairs corresponding to the question and its answer. The function then returns this list of dictionaries.
- **pere_fouras_riddles() :** The function simulates an encounter with Père Fouras, where the player has three attempts to solve a riddle. Below is the pseudocode for the function, which can guide you in implementing **pere_fouras_riddles()**.

Function : pere_fouras_riddles() → Boolean
Local variables[4] **:**

- A list to store the riddles.
- A dictionary to store the randomly selected riddle.
- An integer initialized to 3, representing the number of attempts.

Begin

1. Use the **load_riddles(file)** function to retrieve the riddles from the JSON file and store them in a list.
2. Randomly select a riddle from the list and store it in a dictionary variable.
3. Display the riddle's question to the player.
4. As long as the number of attempts is greater than zero 0 do:
   - Prompt the player to enter their answer and convert the input to lowercase.
   - If the player's answer matches the correct answer from the chosen riddle:
     - Display a message indicating the answer is correct and that the player wins a key.
     - Return **True.**
   - Otherwise:
     - Decrease the number of remaining attempts by 1.
     - If any attempts remain:
       - Display a message that the answer is incorrect and show the number of remaining attempts.
     - Otherwise:
       - Display a failure message along with the correct answer to the riddle.
       - Return **False.**

End.

3.5. The final_challenge.py module

The treasure room represents the final stage of the game, where the team, having collected all the keys, must decipher the code needed to open the door and access the treasure. In this module, you will simulate this final step to access Fort Boyard's treasure room.

Implement the **treasure_room()** function by following the steps below:

[4] **The list of local variables is incomplete.**

1. Load the data from the **'TRClues.json'** file, which contains the necessary clues (indices) and codeword. Ensure you understand the structure of this file to access the data correctly.
2. Randomly select a program, then extract its clues and associated codeword.
3. Display the first three clues to help the player guess the code word. The player has three attempts in total.
4. After each wrong answer, provide an additional clue and display the number of remaining attempts.
5. If the player fails after three attempts, reveal the correct code word.
6. A final message indicate whether the player succeeded or failed at the end.

Here's a pseudocode of this function, which you can use as a guide to correctly implement **treasure_room*()*.**

Function : treasury_room()
Local variables[5] **:**

- **tv_game, show**: Dictionary
- **clues**: List
- **year, code_word**: String
- **attempts**: Integer
- **answer_correct**: Boolean

Begin

1. Load data from the file **'TRClues.json'** into the variable ***tv_game***.
2. Obtain the list of available years in ***tv_game*** and randomly select one. Store it in the variable ***year***.
3. Extract the programs associated with this year from ***tv_game***. Select one at random and store it in the variable ***show***.
4. Extract the clues and store them in the clue's variable the extract the corresponding codeword and store it in ***code_word***.
5. Display the first three clues from the clues list.
6. Set the attempt counter (***attempts***) to 3.
7. Initialize the ***answer_correct*** variable to False.
8. While the number of attempts is greater than 0:
    - Ask the player to enter an answer.
    - If the player's answer matches the code word:
        - Update ***answer_correct*** to True.
        - Exit the loop.
    - **Otherwise**:
        - Decrement the attempt counter.
        - **If** attempts remain:
            - Display the number of remaining attempts.

[5] **The list of local variables is incomplete.**

        - Display an additional clue.
    - **Else**:
        - Reveal the correct code word.
9. If ***answer_correct*** is True:
    - Display a victory message.
10. Otherwise:
    - Display a failure message.

End.

3.6. The utility_functions.py module

In the **utility_functions.py** module, you will implement a series of functions to manage players, events, and the recording of game results. These functions are essential for organizing and tracking the game's progress.

Below, you'll find the function prototypes. Ensure that each function is implemented according to its specified signature and described behavior.

3.6.1. Introduction()

Implement an **introduction()** function that displays a welcome message for the game and explains the basic rules:

- The player must complete challenges to earn keys and unlock the treasure room.
- The aim is to collect three keys to access the treasure room.

3.6.2. compose_equipe()

Implement the **compose_equipe()** function, which creates a team of players for the game, in the form of a **list**. The team can have up to 3 players.
The function must first ask the user how many players they wish to include in the team. If the number of players exceeds 3, an error message will be displayed, and the user will be prompted to re-enter the data.
For each player, the function must request and enter the following information: ***name***, ***profession***, and whether the player is the team ***leader***. Each player will be represented as a **dictionary** containing these three details, plus a fourth field, ***'keys_wons'***, initialized to zero.
If, at the end of the team composition, no player has been designated as the leader, the first player on the team will automatically become the leader.
The function must return the **list containing the players** who make up the team.

3.6.3. challenges_menu()

Implement the **challenges_menu()** function, which displays the following menu allowing the user to choose from the various types of challenges available:

1. Mathematics challenge
2. Logic challenge

```
3. Chance challenge
4. Père Fouras' riddle
```

The user enters the number corresponding to the challenge, and the function returns this choice.

3.6.4. choose_player(team)

Implement the **choose_player(team)** function, which takes a list of dictionaries representing players as a parameter and allows the user to select a player from the team to take on the challenge.

The function must display the list of players with their name, profession, and role (display "Member" if not the leader). ***For example:***

```
1. Jean Dupont (Engineer) - Leader
2. Marie Martin (Teacher) - Member
3. Paul Durand (Doctor) - Member
Enter the player's number:
```

The user can then input the number of the desired player, and the function will return the selected player as a dictionary containing their information.

3.6.5. Bonus function record_history(?)

This history recording function is optional but can earn you bonus points. You are free to define the parameters to be recorded in the **'output/history.txt'** file, such as the challenge name, the player, the result, or the number of keys obtained, among others. The most important thing is to structure and format the data in a clear and easily readable format.

3.7. Implementing the main function in main.py

In **main.py**, define the **game()** function, which will centralize all game actions using the functions of the other modules you've created in this project.

> ***Note:*** *When starting the game simulation, the user will first assemble the team of players, then make decisions throughout the game. The user will also assume the role of the selected player, participate in the challenge, and must win it.*

The **game()** function includes the following steps:

1. **Introduction and team composition**: The game begins with an introduction explaining the rules, followed by the user being invited to create a team of players. The team can consist of up to 3 players.
2. **Event loop**: The game continues in a loop until the team has won three keys:
   - A menu of challenge types is displayed, and the user selects a type.
   - Once the event type is validated, the list of players is shown, and the user must choose one.
   - A challenge is launched, chosen at random, except for the logical challenge.

- The user, taking on the role of the selected player, attempts to win the challenge.
- If the user wins the challenge, a key is awarded, represented by an increment in the number of keys won by the selected player.

3. **Final stage**: Once all three keys have been obtained, the final challenge of the treasure room is launched. The user is given clues to guess the code word and unlock the room. If successful, the team wins the game, otherwise, the team loses.

Recording results (bonus section): Throughout the game, the event history (selected challenges, players, results, etc.) is recorded. At the end of the game, this history is saved in the file **'output/history.txt'**.

4. Project evaluation: Criteria and expectations

The project evaluation includes:

- A grade for collaboration via Git
- A grade for documentation
- A grade for the source code
- A grade for the defense

4.1. Git repository: 10% of the Project Grade

Evidence of collaboration from all project team members through "commits" distributed throughout the project timeline.

4.2. Project documentation: 10% of the Project Grade

This documentation refers to the **README.md** file in the project's **Git** repository. It must follow the structure outlined below:

1. **General presentation :**
    - **Project Title**
    - **Contributors:** List of team members and their respective roles.
    - **Description:** A brief overview of the project.
    - **Key features:** A list of the main features offered by the application.
    - **Technologies used:** Mention programming languages, libraries (if applicable), and tools.
    - **Installation :**
        - Instructions for cloning the Git repository.
        - Steps to set up the development environment (installation requirements).
    - **How to use :**
        - Instructions on how to run the application.
        - If necessary, examples of commands or use cases.
2. **Technical documentation**
    - **Game algorithm :**

        - Present the algorithm you've developed for the project, including each step as a numbered list.
    - **Functions:**
        - Provide a list of function prototypes, including brief descriptions of their roles and parameter explanations.
    - **Input and error management :**
        - Describe how the code handles input values, intervals, and the methods used for error management.
        - Provide a list of known bugs.
3. **Logbook**
    - A logbook can help keep track of project progress and task allocation:
        - **Project chronology**: Dates and descriptions of milestones, decisions made, and problems encountered.
        - **Task distribution:** Who worked on what and how tasks were divided among team members.
4. **Testing and Validation**
    - **Test strategies :**
        - Specific test cases and results.
        - Screenshots showing the tests in action.

4.3. Code: 60% of the Project Grade

The code will be evaluated on two levels:

Content:

- The number of requested functionalities successfully completed.
- The program is modular:
    - Functions are used to logically distribute tasks across different files, improving the readability and comprehensibility of the code.
    - The chosen algorithms are "reasonably" efficient, avoiding unnecessary processing and variables.
- The code is robust against potential errors:
    - In particular, it ensures secure user input and prevents it from causing execution errors.
- The ergonomics of the user interface.

Code clarity:

- Each code file (**.py**) includes a global comment at the head of the file, with the name of the project, the authors and the role of this file in the project.
- Each function is preceded by a comment. It describes for each function:
    - Its role, the meaning of its parameters, the result returned (if none: explain why).

- Inside the functions, comments **only** indicate the **main** steps of the algorithm and explain tricky parts of the code.
- Functions and variables are named consistently.

4.4. Defense: 20% of the Project Grade

The project defense takes about **15 minutes** per team. Here are some elements of appreciation of the defense:

- Demonstration of the work done: Test scenarios covering all the performed functionalities.
- Quality of answers to questions
- Appropriation of the speaking time within the group.

4.5. Bonus points

- Parts marked as bonus in the project subject
- All forms of originality and innovation

5. Submitting work on Moodle

Project deposits must be made in the dedicated Moodle deposit area. Only one member of the pair will make the deposit.

- **Interim submission: Saturday 21/12/2024, at noon.** The ***.zip*** file must include the following:
  - The duly completed **interm deposit form**
  - All .py files already created in the project
  - All .json and/or .txt files used in the project
- **Final submission**: **Sunday 05/01/2025, at noon.** It is imperative to include the following elements in a ***.zip*** file:
  - The completed **final deposit form**
  - All .py, .json and/or .txt project files, organized according to the specified directory structure.
  - The **README.md** file in the requested format

Please note:

- Any **missing files** will result in a score of **0** for this part. It is therefore crucial to check carefully that all files are included, and not to wait until the last minute to make the deposit.
- **Once** the Moodle deposit area **has been closed, NO submissions** via email or Teams **will be accepted**.
- Any form of **plagiarism is strictly forbidden**, including copy/paste from external sources or code **generation using generative artificial intelligence**. Group members may use external sources only for inspiration or to stimulate ideas. However, they must ensure that their work is personal and original. Any **unoriginal**

work will result in an **immediate sanction of 0/20** and a **disciplinary hearing**. For more information on our anti-plagiarism policy and tools such as Compilatio and Studium, please refer to the documentation available in the Academic Department section of the pedagogical portal.