Práctica 7:
Bucles II

Índice

Objetivos

Objetivos

- Comprender el anidamiento de bucles de repetición.
  - Principales inconvenientes.
- Refactorización de bucles anidados.
- Entender problemas que impliquen múltiples bucles anidados y su implementación legible mediante funciones.

Objetivos

Se parte de un código desarrollado que incorpora múltiples bucles anidados. Además de asimilar el código y de ejecutarlo, hay que entender la problemática de las aplicaciones monolíticas (dificultad de extensión y de depuración, así como de mantenimiento). Consecuentemente, se aprenderá a refactorizar el código por medio de funciones que se resuelvan tareas, desde las más simples a las más complejas. Finalmente, se extenderá el código con nuevas funcionalidades.

Prueba el ejecutable de la aplicación disponible en Campus Virtual.

Planificación

Idea general

La sesión está dedicada a, partiendo de un código de un juego (Dungeon of Doom), se debe ser capaz de entender el código, así como de refactorizarlo para incrementar la legibilidad, facilitar el mantenimiento y la extensibilidad del mismo.

Todo ello está programado en 4 diferentes secciones o partes.

Organización

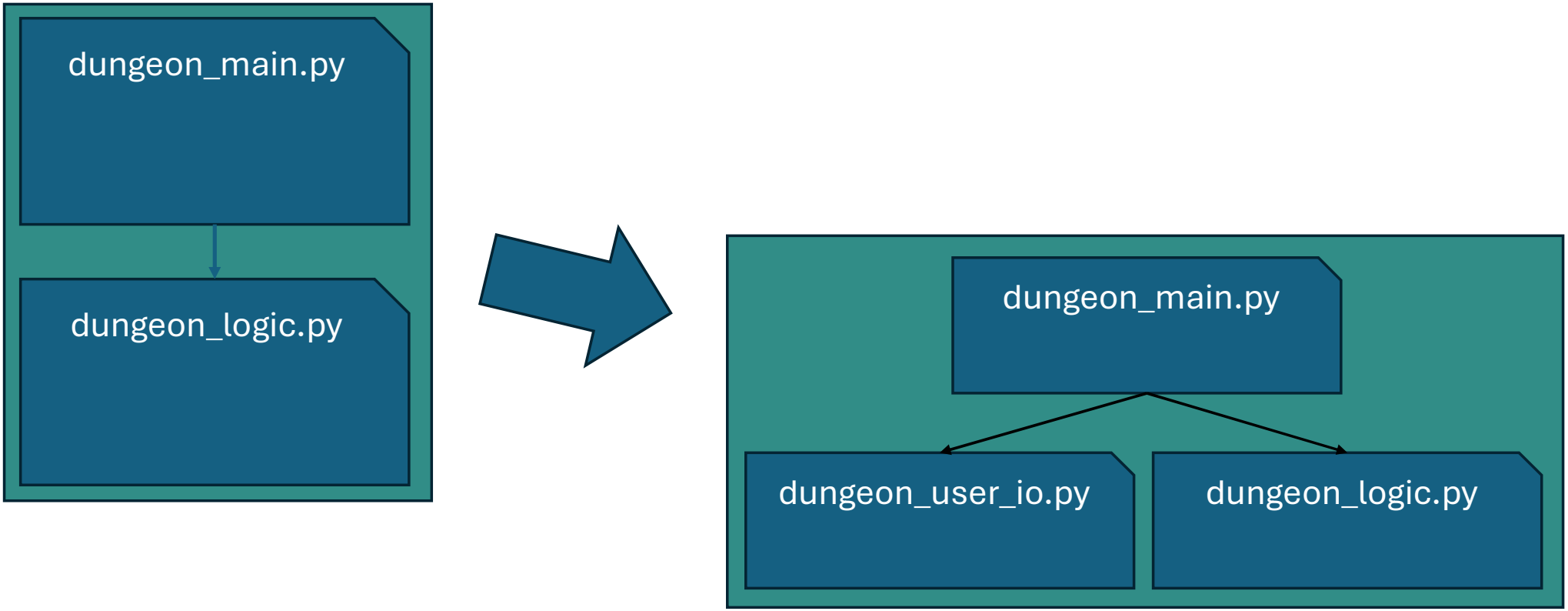
Parte 1: entender el código y la problemática asociada a códigos monolíticos.

Parte 2: Aprender a refactorizar, incrementando la legibilidad y la extensibilidad, así como facilitando el mantenimiento.

Parte 3: Introducir funcionalidad extra al programa refactorizado.

Parte 4: Trabajo autónomo.

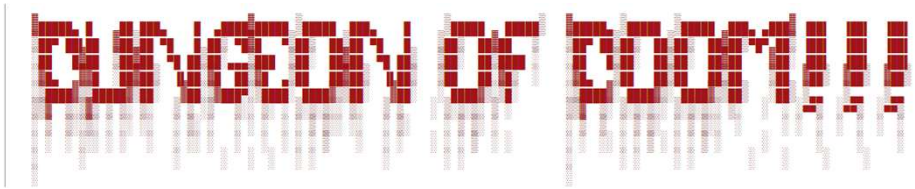
Organización del código



Parte 1: El anidamiento de bucles

Parte 1: El anidamiento de bucles

El código disponible



Descarga y ejecuta el código (archivos **dungeon_main.py** y **dungeon_logic.py**).

Se trata de un juego en el que se lucha de manera sucesiva con diferentes monstruos hasta que se agota la vida del jugador. El reto es llegar al nivel más alto posible.

Análisis del código

```
Mientras el jugador tenga vida
   Mostrar datos del turno actual
   Selección de monstruo
   Mientras el monstruo y el jugador tengan vida
      Mostrar datos de la ronda
      Selección de acción de jugador
      Si el jugador decide pelear
         Obtener jugada de jugador y de monstruo
         Determinar resultado de la ronda
         Actualizar valores de vida
      Si no, Si el jugador huye
         Pasar al siguiente turno
```

Vamos a estudiar el código!
Luego volvemos...

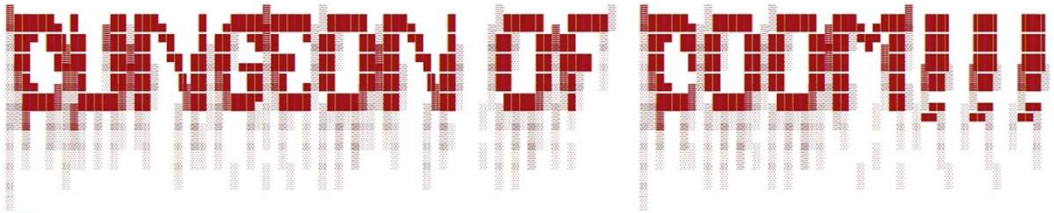
Parte 1: El anidamiento de bucles

```python
# -*- coding: utf-8 -*-

from random import randint

import dungeon_logic as logic

# It is a real code's quality metric is to have every block of code with
# a heading comment with the hint of what the block is intended for.


def main():
    # Print begin banner
    print("Welcome to the")
    print("""

    """)

    # Main game loop
    turn = 0
    health = 100
    level = 1
    while health > 0:
        turn += 1

        print()
        print("*" * 80)
        print(f"* Turn {turn}")
        print("*" * 3)
        print()

        print("Your health is", health)
        print("Your level is", level)
        print()
```

Parte 2: Refactorizando código

Parte 2: Refactorizando código

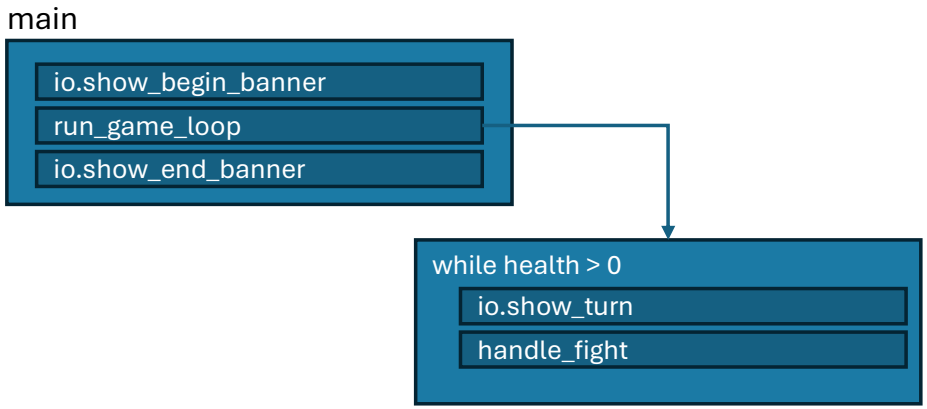
dungeon_main

main
roll_and_show

dungeon_logic

calculate_monster
calculate_fight_round
validate_Rolls
validate_roll
update_hero_health
update_health

```
Mientras el jugador tenga vida
    Mostrar datos del turno actual
    Selección de monstruo
    Mientras el monstruo y el jugador tengan vida
        Mostrar datos de la ronda
        Selección de acción de jugador
        Si el jugador decide pelear
            Obtener jugada de jugador y de monstruo
            Determinar resultado de la ronda
            Actualizar valores de vida
        Si no, Si el jugador huye
            Pasar al siguiente turno
```



dungeon_logic

calculate_monster
calculate_fight_round
validate_Rolls
validate_roll
update_hero_health
update_health

dungeon_user_io

show_begin_banner
show_end_banner
show_turn
show_fight_round
ask_fight_option

Parte 2: Refactorizando código

Objetivo

- Convertir código en funciones.
  - Simplificando el desarrollo.
  - Facilitando la lectura.
  - Reduciendo las tareas de mantenimiento

Pasos iniciales

- Factorizar las etapas del programa principal:
  - Funciones para I/O en el módulo dungeon_user_io.
  - Funciones para estructurar la lógica del juego

Parte 2: Refactorizando código
Funciones para I/O (I)

- Crea el módulo **dungeon_user_io.py**.
- Funciones a crear:
  - show_begin_banner(): muestra la pantalla de inicio del juego.
    - Introducir la llamada a esta función en el main, sustituyendo el código.
  - show_end_banner(level): muestra la pantalla final del juego.

    Recibe el nivel al que se ha llegado durante el juego.
    - Sustitute el código del main por la llamada a la función.

Parte 2: Refactorizando código
Funciones para I/O (II)

- Completar la factorización de **main**().
  - run_game_loop().
  - handle_fight(health, level).
- ¡Ambas funciones en el **dungeon_main.py** module!

Parte 2: Refactorizando código
Funciones para I/O (II)

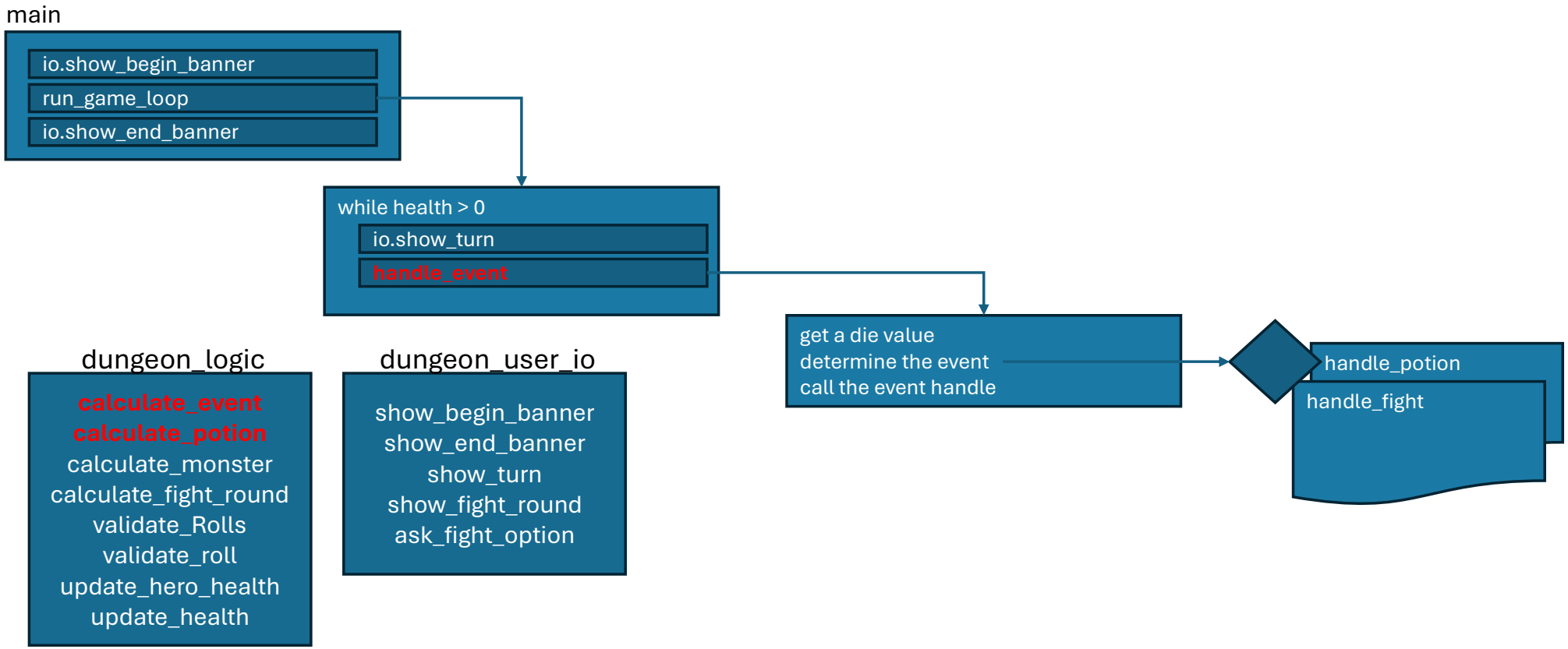
- Completar la factorización de **main**().
  - run_game_loop(): inicializa las variables del juego, contiene el bucle principal del juego, llama a la función que ejecuta una pelea, retornando el nivel del juego.
    - Retorna el nivel de juego porque es necesario para su uso posterior en la función con el mensaje de final de juego.

Parte 2: Refactorizando código
Funciones para I/O (II)

- Completar la factorización de **main**().
  - handle_fight(health, level): ejecuta una pelea… Elige el monstruo, contiene el bucle de la pelea, muestra información del estado de la pelea, y retorna la salud del jugador y el nivel alcanzado.
    - La salud del jugador se requiere retornar debido a que se debe actualizar para la siguiente pelea, o para determinar el final del juego.
    - El nivel alcanzado se requiere retornar para mantener esta variable actualizada a lo largo del juego.

Parte 3: Añadir más funcionalidades

Parte 3: Añadir más funcionalidades



Parte 3: Añadir más funcionalidades

¿Qué pasa si queremos más tipos de eventos?

- Modificar funciones de generación y de distribución de eventos.
- Incluir función gestora de cada nuevo evento.

Ejemplo

- El evento de pócima –bien venenosa o reparadora- puede aparecer aleatoriamente.
- Necesitamos introducir las funciones handle_event, handle_potion, logic.calculate_event y logic.calculate_potion.

Parte 3: Añadir más funcionalidades

- dungeon_main.handle_event(health, level):
  - Es una refactorización del código de la función run_game_loop, siendo llamada desde ésta.
  - Obtiene el evento mediante llamada a dungeon_logic.calculate_event.
  - En función del evento llamará a dungeon_main.handle_fight o dungeon_main.handle_potion.
  - Si se trata de un evento no válido, se genera una Excepción tipo ValueError.
  - Retorna la fuerza y nivel después de la gestión del evento.

Parte 3: Añadir más funcionalidades

- dungeon_logic.calculate_event(value):
  - Implica definir constantes para los posibles eventos (EVENT_POTION, EVENT_MONSTER, …).
  - Comprueba que el valor recibido es válido usando la función disponible.
  - Cuando el valor es uno retornar EVENT_POTION. En otro caso retorna EVENT_MONSTER.
  - Documenta adecuadamente la función con varios casos de uso.

Parte 3: Añadir más funcionalidades

- dungeon_main.handle_potion(health):
  - Muestra un mensaje del evento que está gestionándose.
  - Obtiene una jugada aleatoria mediante roll_and_show.
  - Calcula el efecto de la poción obtenida usando la función dungeon_logic.calculate_potion.
  - Si la nueva salud es mejor que la actual la poción fue sanadora; sino, la poción fue venenosa (en cualquier caso, muestra el mensaje adecuado). Si ambas son iguales, pese a que –como veremos- es una situación imposible, se mostrará un mensaje de poción vacía.
  - Retorna el nuevo valor de salud calculado.

Parte 3: Añadir más funcionalidades

- dungeon_logic.calculate_potion(value, health):
  - Definir las constantes POTION_THRESHOLD (a 3 para tener 50% de posibilidad de cada caso), POTION_DAMAGE (a -10) y POTION_CURE (a 10).
  - Se verificará que value sea válido usando la función disponible para ello.
  - Si value es mayor a POTION_THRESHOLD el incremento de salud será POTION_DAMAGE; sino, será POTION_CURE.
  - Se incrementará la salud del jugador por medio de la función disponible para ello.
  - Se retornará el nuevo valor de salud.

Parte 4: Ejercicios extra

Parte 4: Ejercicios extra 1

- A la hora de escapar, el monstruo puede atacar al jugador!
  - Aleatoriamente se decidirá si el monstruo ataca o no.
  - El ataque será exactamente igual un round de pelea entre el jugador y el monstruo (generar las dos tiradas, llamar a calculate_fight_round).
  - Se mostrarán mensajes informativos del estado de escape.

Parte 4: Ejercicios extra 1

- A la hora de escapar, el monstruo puede atacar al jugador!
  - Definir **dungeon_main.SAFE_SCAPE_THRESHOLD** con valor 4.
  - Implementar las siguientes funciones:
    - **dungeon_main.handle_scape** para codificar lo que es toda la etapa de huida. Es llamada desde *dungeon_main.handle_fight*.
    - **dungeon_main. run_a_fight** para realizar un ciclo de pelea.
    - **dungeon_user_io.show_death_message** para mostrar mensajes después de la lucha.
  - Modificar la función:
    - **dungeon_main.handle_fight** dado que las nuevas funciones simplifican la misma.

Parte 4: Ejercicios extra 1

dungeon_main. run_a_fight:

- Obtiene las 2 tiradas del jugador, así como las dos del monstruo.
- Llama a calculate_fight_round.
- Retorna los nuevos valores de salud.

- Para no repetir código, modificar *handle_fight* para simplificar su código debido a la existencia de la función que se acaba de escribir.

Parte 4: Ejercicios extra 1

dungeon_main.handle_scape:

- Notifica que está en escape.
- Obtiene la tirada con la función disponible para ello.
- Si la tirada es mayor que el umbral de escape seguro
  - Informar del ataque.
  - Obtener las jugadas de jugador y monstruo.
  - Ejecuta el round de pelea.
  - Actualiza las variables de salud del jugador y monstruo.
  - En caso de matar al monstruo, subir el nivel.
- Sino
  - Informar que se pudo escapar sin problemas.
- Informar del estado después del escape con show_death_message.
- Retornar los nuevos valores de salud y de nivel.

Parte 4: Ejercicios extra 1

dungeon_user_io.show_death_message

- Muestra mensaje acorde si el jugador o el monstruo perdieron su vida.
- Debe actualizarse cuidadosamente la función dungeon_main.handle_fight para simplificarla.

Parte 4: Ejercicios extra

- Laberinto aleatorio
  - El jugador tiene un número ENTERO de turnos linealmente dependiente del nivel: nivel 1 → 5 turnos; nivel 24 → 20 turnos.
  - El laberinto tiene una longitud ENTERA dependiente del nivel: nivel 1 → 2 posiciones; nivel 24 → 8 posiciones.
  - El jugador tiene los turnos indicados para avanzar las posiciones del laberinto.
  - Se avanza a la siguiente posición si la jugada obtenida por el jugador (entre arriba, abajo, izda o dcha) es igual a la dirección seleccionada aleatoriamente para dicha posición en el laberinto.
  - En caso de que la jugada y dirección real no coincidan se penaliza al jugador perdiendo salud. Si están en la misma dirección (arriba-abajo ó dcha-izda, pero en diferente sentido) se penaliza con 5, mientras que si son direcciones perpendiculares se penaliza con 10.
  - Se debe definir todas las constantes que sean necesarias.
  - Igualmente, se modificará la función de generación de eventos para que todos los posibles casos tengan igual probabilidad.

Parte 4: Ejercicios extra

- Laberinto aleatorio
  - Funciones a modificar:
    - dungeon_main.handle_event para incluir el nuevo tipo de evento.
    - dungeon_logic.calculate_event para que todo evento tenga igual probabilidad.
  - Funciones a crear (en el módulo adecuado):
    - handle_laberinth(health, level), implementando la prueba de laberinto, y retornando los nuevos valores de health y level.
    - calculate_player_number_of_turns(level) retornando el número de turnos.
    - calculate_LABERINTH_length(level) retornando la longitud del laberinto.
    - obtain_a_random_LABERINTH_move(subject) generando un movimiento aleatorio para ser usado como dirección real.
    - determine_punishment_for_the_current_movement(move, correct) calcula la penalización por el movimiento del jugador y la dirección actual.
    - ask_valid_laberinth_movement(pos, turn, max_turns, min_val, max_val) para solicitar al jugador la dirección que selecciona.
    - print_current_laberinth_state(pos, length, turn, max_turns) muestra los datos del estado actual del laberinto.
    - show_laberinth_result(pos, length, turn, max_turns, health, level) muestra los resultados finales de la prueba (desde que el jugador perdió la salud, salió del laberinto o si gastó todos los turnos. Si no se cumple ninguno de estos casos, se mostrará el estado del juego en ese momento.