Práctica 5: Condicionales

Índice

Objetivos

Objetivos

- Introducción a los condicionales
- Depuración de programas.
- Diseño e implementación de condicionales para resolver problemas.

Objetivos

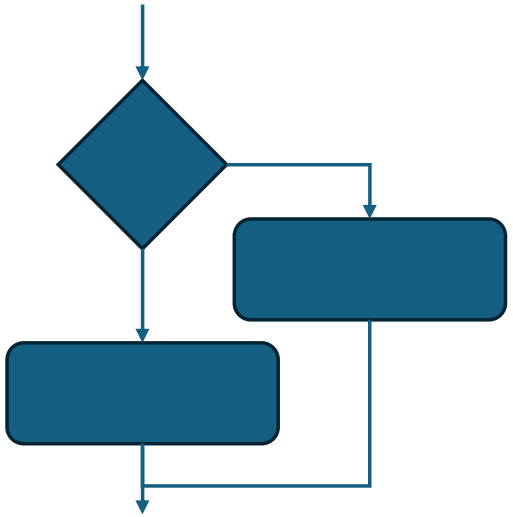
En la parte principal de esta práctica se programará una Gymkhana de Juegos en Secuencia (Dragon's Realm, A question of letters, y Piedra-Papel-Tijera, mas las Mazmorras). Se debe tener en cuenta los condicionales, el tratamiento de excepciones, y la validación de código automáticamente mediante doctest. Igualmente, se debe utilizar la depuración para corregir errores en el código.

Prueba el ejecutable de la Gymkhana disponible en Campus Virtual.

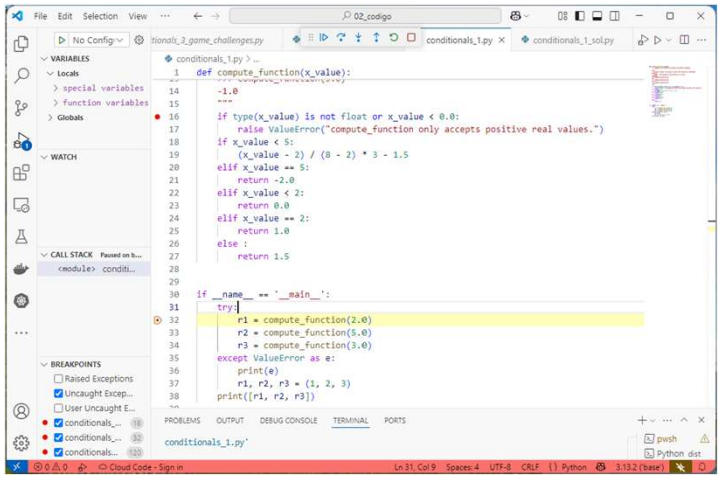
Planificación

Idea general

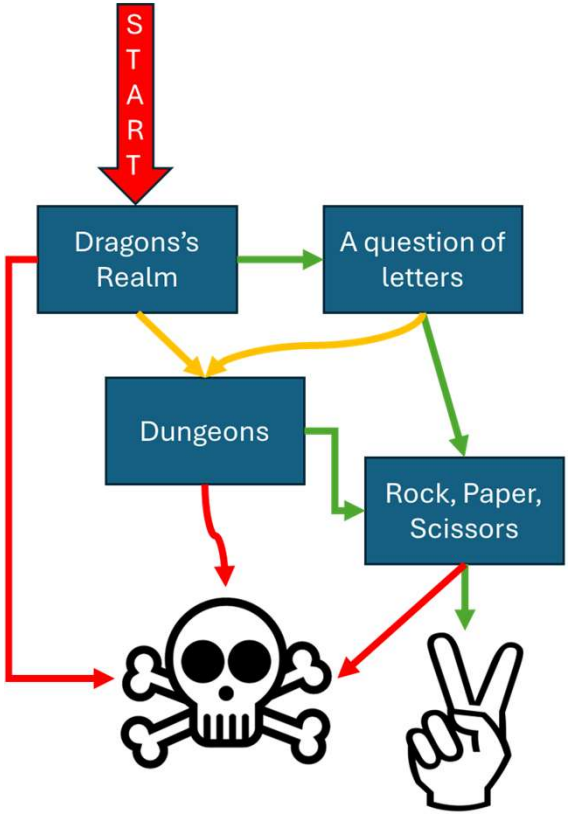
Introducción a los condicionales



Depuración de código



Gymkhana de programación



Idea general

Primero se realizará un ejercicio de introducción a los condicionales.

- Resolver tres funciones mediante el uso de la estructura condicional correcta.

Seguidamente, se depurará un código y se estudiará el manejo de excepciones.

- Resolver los errores de un código mediante la depuración.

Desarrollar un módulo para resolver los juegos incluidos en la Gymkhana.

- Resolver una serie de funciones para resolver los diferentes retos.
- Validar mediante doctest las funciones de lógica.
- Desarrollar el programa principal para la Gymkhana.

Organización

Parte 1: Introducción a los condicionales

Parte 2: Depuración de códigos. Excepciones.

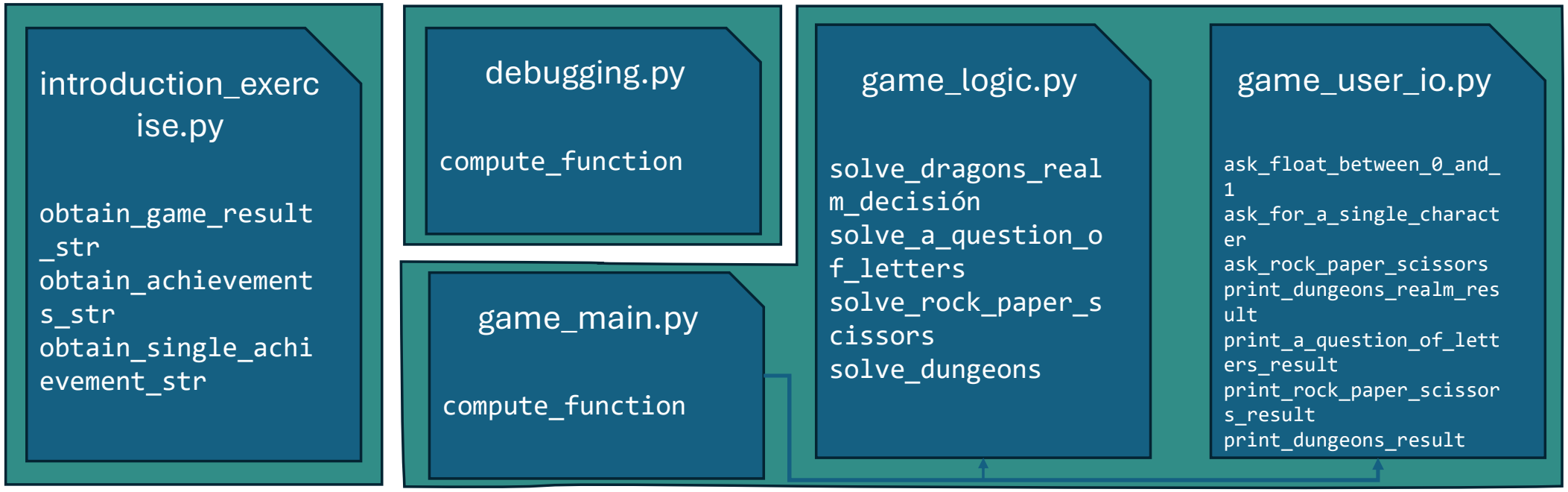
Parte 3: Gymkhana de programación.

Organización del código

Depuración y excepciones

Depuración de código

Gymkhana de programación



Parte 1: Introducción a los condicionales

Parte 1: Introducción a los condicionales

Objetivo: seleccionar, de entre las estructuras de condicionales, la mejor para solucionar el problema concreto:

- If-else
- If-elif-else
- If en secuencia
- If anidados

Parte 1: Introducción a los condicionales

Caso 1: implementar la función **obtain_game_result_str** que recibe los puntos anotados y el tiempo invertido (en minutos) por un jugador en conseguirlo, retornando una cadena con el resultado:

- Cuando bien los puntos o bien el tiempo invertido sean negativos se generará una excepción ValueError.
- En caso que anote más de 1000 puntos en menos de 90 minutos la cadena a retornar será “You win!”
- En otro caso, la cadena a retornar será “You lose!”

Parte 1: Introducción a los condicionales

Caso 2: implementar la función **obtain_achievements_str** con iguales parámetros y retorno que la anterior, generando excepción en las mismas condiciones. La cadena a retornar, inicialmente vacía, se forma concatenando mensajes según cada una de las siguientes condiciones:

- Más de 5000 puntos y tiempo mayor a 90, la cadena añadirá “-Slow but Steady!-”.
- Menos o igual a 1000 puntos y tiempo menor a 30, la cadena añadirá “-I just wanna finish fast!-”
- Si son más de 1000 puntos, la cadena añadirá “-High Score!-”
- Si es menos de 90 minutos, la cadena añadirá “-High Speed!-”

Parte 1: Introducción a los condicionales

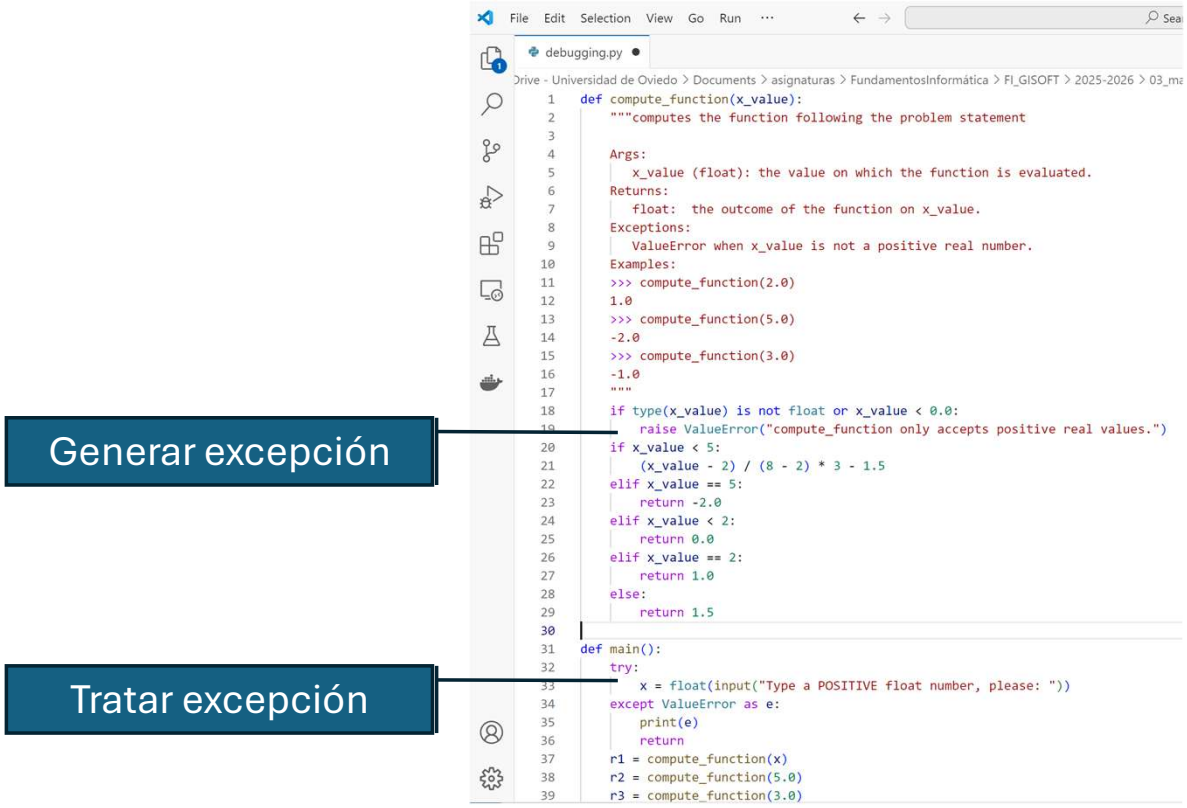
Caso 3: implementar la función **obtain_single_achievement_str** con iguales parámetros y retorno que la anterior, generando excepción en las mismas condiciones. La str a retornar se decide según el caso, debiendo analizarse en este orden:

- Más de 5000 puntos y tiempo mayor a 90, la cadena será “-Slow but Steady!-”.
- Menos de 1000 puntos y tiempo menor a 30, la cadena será “-I just wanna finish fast!-”
- Si son más de 1000 puntos, la cadena será “-High Score!-”
- Si es menos de 90 minutos, la cadena será “-High Speed!-”
- Resto de casos, la cadena será “-No achievements!-”

Parte 2: Depuración de código

Parte 2: Depuración de datos

Abre el archive **debugging.py**



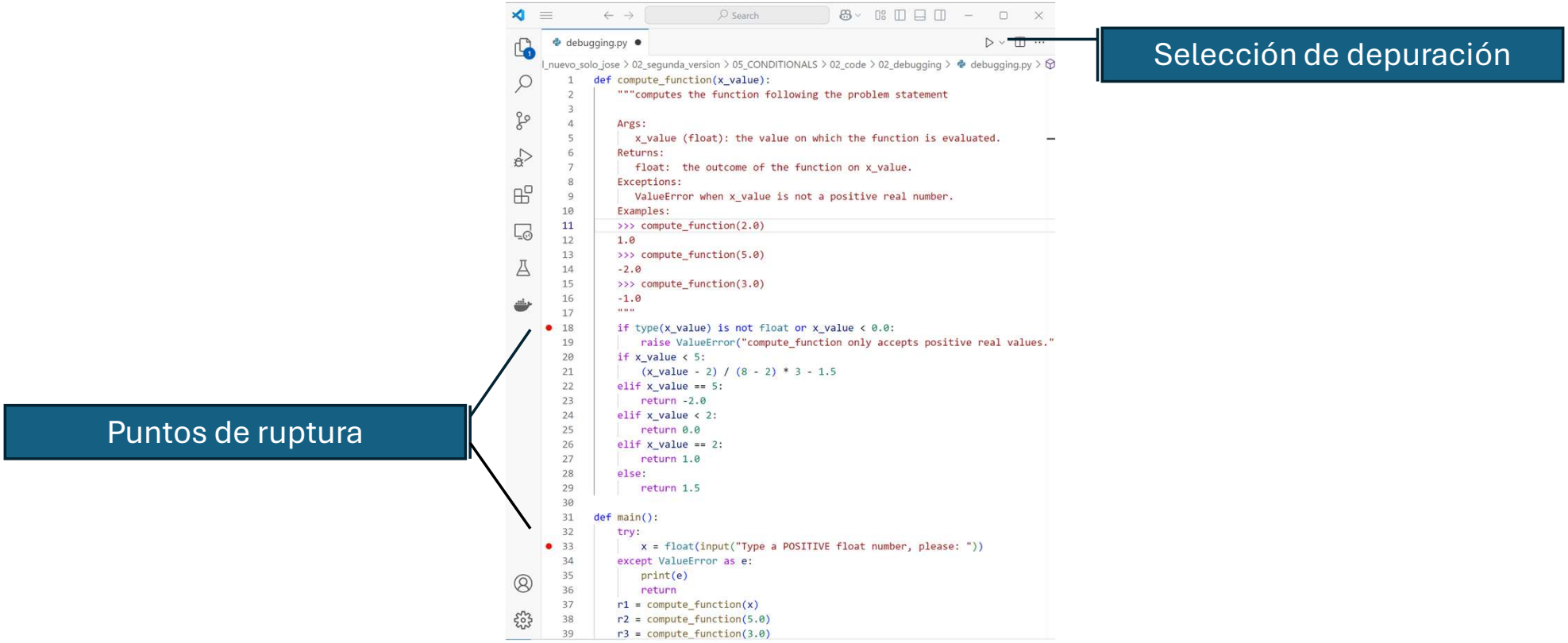
Función **compute_function**

- Documentación, casos base.
- Excepciones por errores de argumentos.
- Definida para $\mathbb{R}^+$:

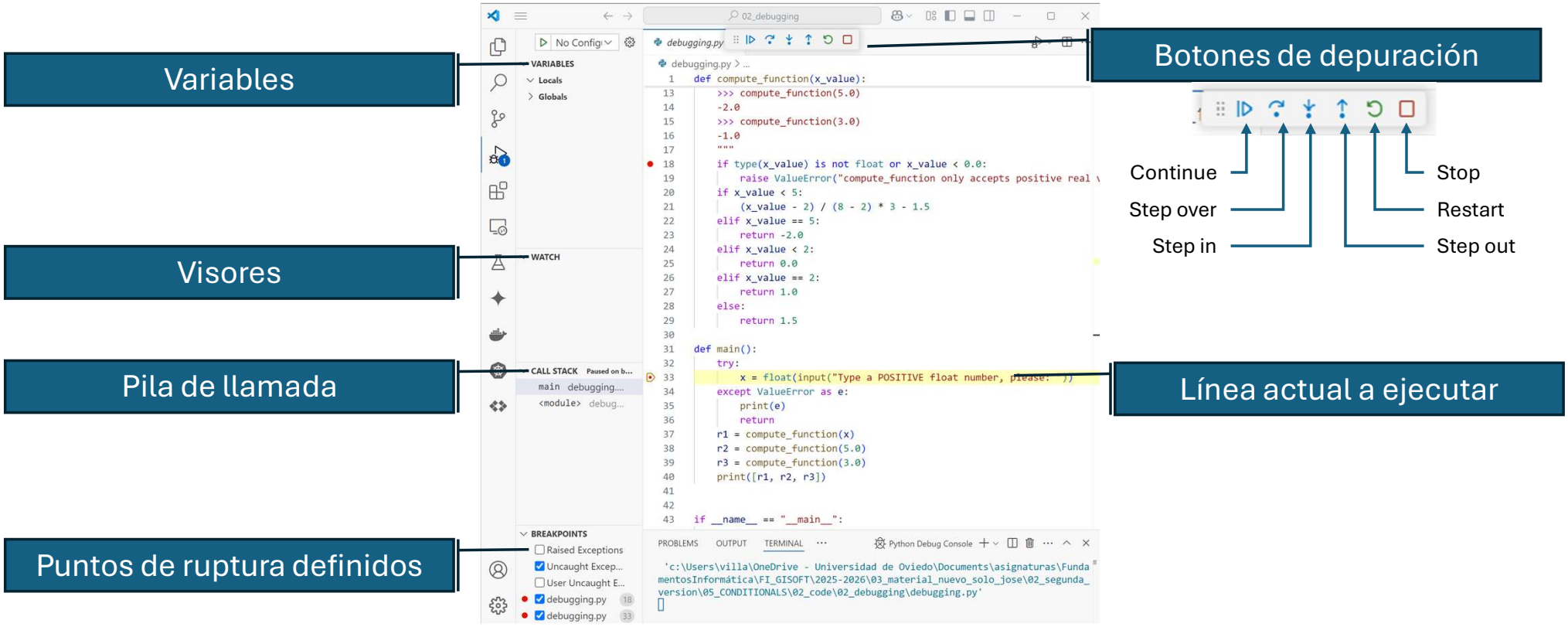
$$f(x) = \begin{cases} 0.0 & x < 2 \\ 1.0 & x = 2 \\ 3\dfrac{x-2}{8-2} - 1.5 & 2 < x < 5 \\ -2.0 & x = 5 \\ 1.5 & resto \end{cases}$$

- **Contiene errores!**

Parte 2: Depuración de datos



Parte 2: Depuración de datos



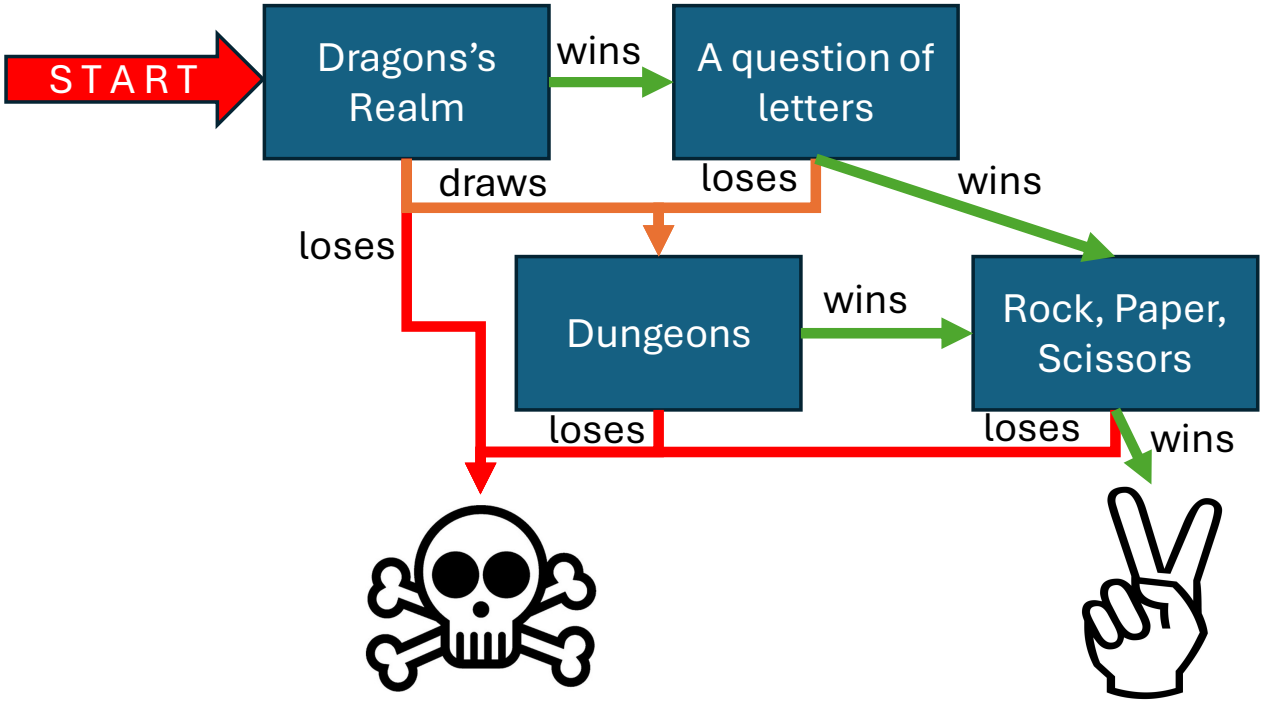
Parte 2: Depuración de datos

- Recoger el código debugging.py y, utilizando tu conocimiento y la depuración, encuentra y corrige los errores existentes.

Parte 3: Gymkhana de programación

Parte 3: Gymkhana de programación

Implementar el siguiente juego



game_main.py

```
compute_function
```

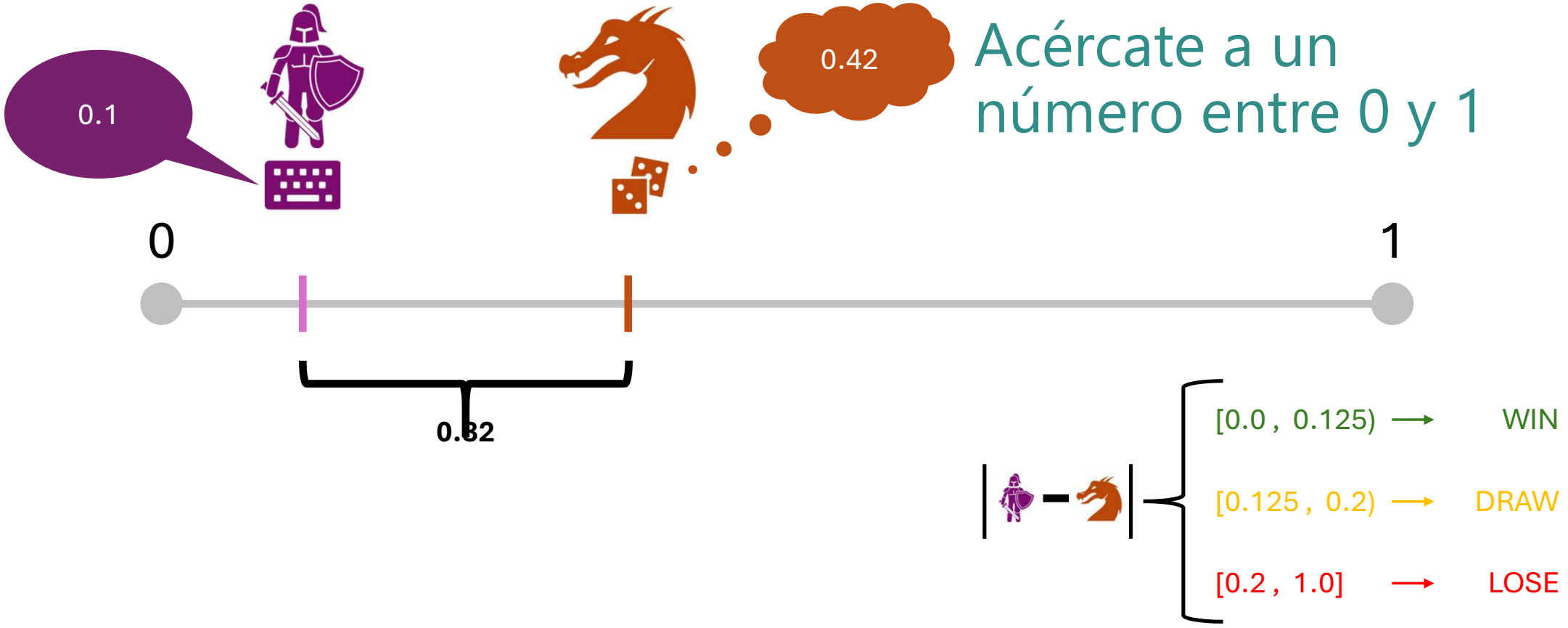
game_logic.py

```
solve_dragons_realm_decisión
solve_a_question_of_letters
solve_rock_paper_scissors
solve_dungeons
```

game_user_io.py

```
ask_float_between_0_and_1
ask_for_a_single_character
ask_rock_paper_scissors
print_dungeons_realm_result
print_a_question_of_letters_result
print_rock_paper_scissors_result
print_dungeons_result
```

Parte 3.1: Dragon's Realm



Parte 3.1: Dragon's Realm

Dragon's Realm: el programa genera un número aleatorio en [0.0, 1.0]. Tras ello, pregunta al usuario por un número en el mismo interval. Cuando la diferencia es menor a 0.125 el usuario gana (WIN_STATE), es un empate (DRAW_STATE) si la diferencia es menor a 0.2; en el resto de los casos el usuario pierde (LOSE_STATE).

- **solve_dragons_realm_decision**, en el módulo *game_logic.py*, retorna WIN_STATE, DRAW_STATE o LOSE_STATE en función del valor de usuario y del valor generado por el ordenador.
- **ask_float_between_0_and_1**, en el módulo *game_user_io.py*, pide al usuario un número real en [0.0, 1.0], retornándolo. Si el valor introducido esté fuera del intervalo, se retorna 0.0.
- **print_dungeons_realm_result**, en el módulo *game_user_io.py*, recibe las jugadas del usuario y del ordenador, así como el resultado de la partida. Imprime a pantalla el mensaje correspondiente según gane, pierda o empate.
- **run_dragons_realm**, en el módulo *game_main.py*, implementa el juego, preguntando al usuario, generando la jugada del ordenador, decidiendo resultado e imprimiento a pantalla el mismo. Retornará el resultado de la partida. Utiliza las funciones anteriores implementadas. Gestionará las excepciones en caso de errores en la introducción de datos.

Parte 3.1: Dragon's Realm

Dragon's Realm (y II): en el modulo game_main:

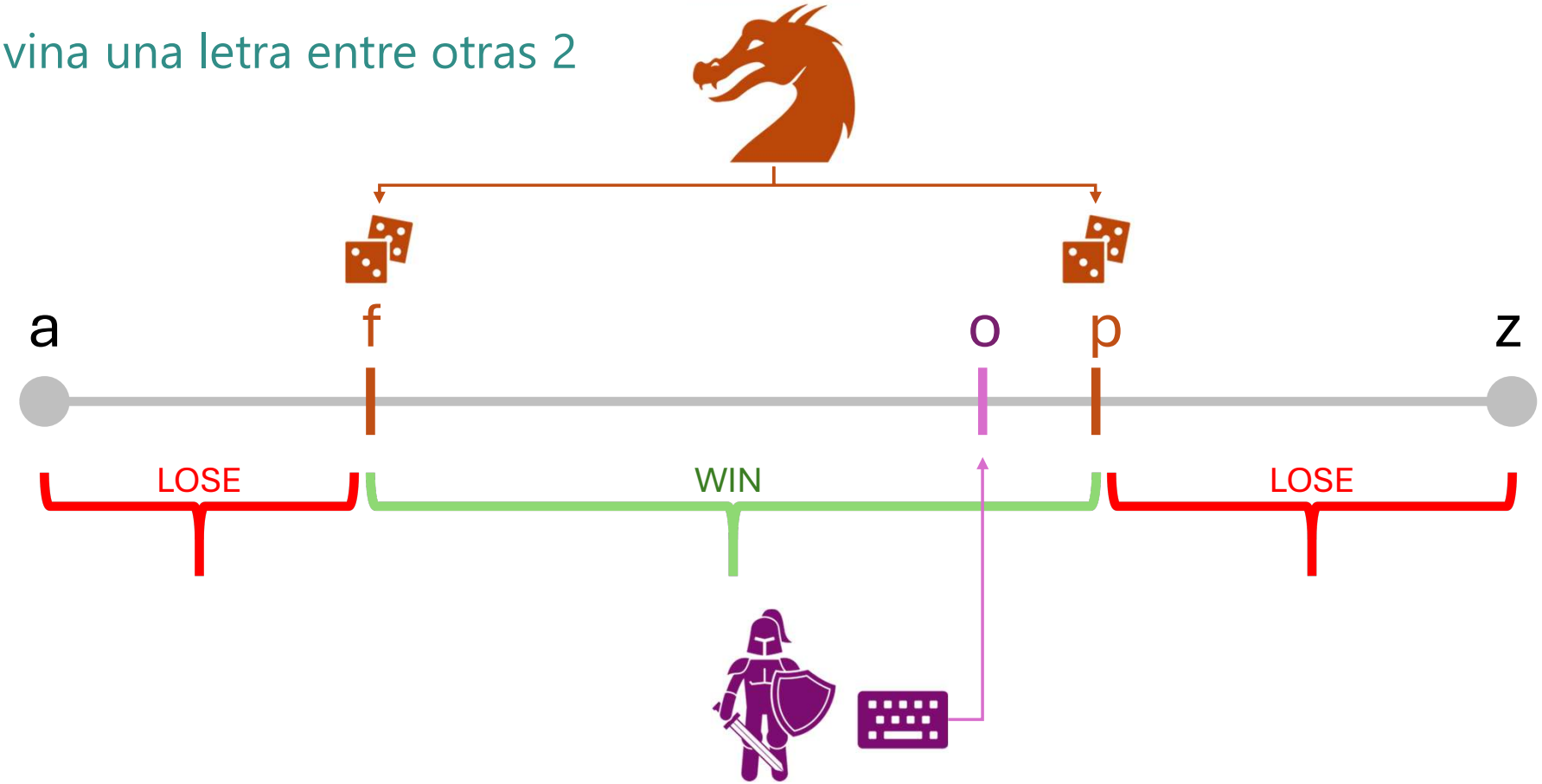
- Definir las etapas del juego como constantes globales:

```
LETTERS_STATE = "LETTERS"
DUNGEONS_STATE = "DUNGEONS"
ROCK_PAPER_SCISSORS_STATE = "RPS"
FINISH_WINNING = "WIN"
FINISH_LOSING = "LOSE"
```

- Implementa la función **main** que imprime un mensaje de bienvenida, llame al juego de Dragon's Realm, y decida cuál será la siguiente etapa actualizando la variable de control estado: LETTERS_STATE en caso de victoria del usuario, DUNGEONS_STATE en caso de empate, FINISH_LOSING en caso de derrota.

Parte 3.2: A Question of Letters

Adivina una letra entre otras 2



Parte 3.2: A Question of Letters

A Question of letters: el programa pide al usuario una letra, llevándola a minúsculas. Después, genera dos enteros aleatorios entre ord('a') y ord('z'), ambos incluidos. El usuario gana si el Código de la letra que introdujo está entre los dos enteros aleatorios.

- **solve_a_question_of_letters** (modulo game_logic) recibe tres caracteres: el elegido por el usuario, y los límites inferior y superior. Retorna WIN_STATE or LOSE_STATE en función de si la letra del usuario está entre los límites.
- **ask_for_a_single_character** (modulo game_user_io) pide al usuario un carácter. Si se introducen más de un carácter se retorna el carácter de menor código. Si se introduce la cadena vacía, se retorna el espacio en blanco. En cada uno de estos dos casos, se mostrarán mensajes oportunos.
- **print_a_question_of_letters_result** (modulo game_user_io) recibe la jugada del usuario, así como los valores límite y el resultado de la partida. Mostrará por pantalla el mensaje oportuno en función de si ganó o perdió.
- **run_a_question_of_letters** (módulo game_main) implementa el juego (pide la jugada al usuario, genera aleatoriamente los límites inferior y superior, decide el resultado e imprime los mensajes oportunos), retornando el resultado de la partida. Utiliza las funciones anteriormente definidas.
- Extender la función **main** para que, si el estado tras jugar Dragon's Realm es LETTERS_STATE, llamará al juego A Question of Letters. Posteriormente. En función del resultado de la partida, actualizará estado a ROCK_PAPER_SCISSORS_STATE o DUNGEONS_STATE.

Parte 3.3: Piedra-Papel-Tijeras

Juego clásico

Parte 3.3: Piedra-Papel-Tijeras

Rock-Papers-Scissors: lo conoces… Se pide al usuario su jugada -Piedra ('R'), Papel ('P') ó Tijeras ('T')-, mientras que el ordenador genera la suya propia del mismo tipo. Las reglas son Piedra gana a Tijeras, Tijeras gana a Papel, y Papel gana a Piedra.

- **solve_rock_paper_scissors** (modulo game_logic) recibe la jugada del usuario y la del ordenador. Retornará WIN_STATE, DRAW_STATE o LOSE_STATE en función de si la jugada del usuario es mejor, es igual, o es peor a la de la máquina.
- **ask_rock_paper_scissors** (modulo game_user_io)pregunta al usuario por la jugada concreta –'R', 'P' o 'T'-, almacenándola en mayúscula y retornándola. En caso de respuesta inválida se generará una excepción ValueError.
- **print_rock_paper_scissors_result** (modulo game_user_io) recibe las jugadas del usuario y ordenador, así como el resultado de la partida. Mostrará el mensaje correspondiente a si gana, pierde o empata.
- **get_random_rock_paper_scissors_choice** (modulo game_main) retorna aleatoriamente 'R', 'P' o 'T' como jugada del ordenador.
- **run_rock_paper_scissors** implementa el juego (obteniendo las jugadas, decidiendo el resultado y mostrando el mismo por pantalla), retornando el resultado de la partida. Utilizará las funciones anteriormente definidas.

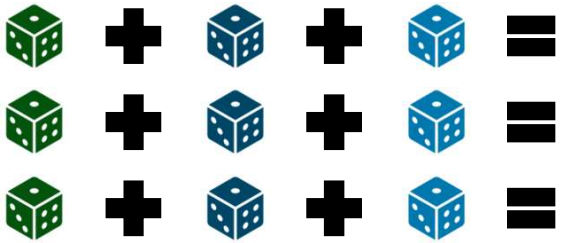
Parte 3.3: Piedra-Papel-Tijeras

Rock-Papers-Scissors (y II):

- Extiende el **main**, de manera que si el estado actual es ROCK_PAPER_SCISSORS_STATE entonces ejecute una partida de este juego. Si el usuario gana, entonces se pasará al estado FINISH_WINNING; sino, se pasará al estado FINISH_LOSING.
- Igualmente, añade la decisión final, mostrando el resultado final de la Gymkhana en función de si el estado es FINISH_WINNING o FINISH_LOSING.

Parte 3.4: Dungeons

Tres dados deciden



Par → WIN

Impar, todos dados impares → WIN

Impar, algún dado par → LOSE

Parte 3.4: Dungeons

Mazmorras: se lanzan 3 dados; si la suma es par o si la suma es impar pero todos los dados son impares, el usuario sale de la mazmorra. Si no, el usuario pierde.

- **solve_dungeons** (módulo game_logic) recibe los valores de los tres dados, retornando WIN_STATE o LOSE_STATE en función de si se cumple o no la condición de victoria.
- **print_dungeons_result** (módulo game_user_io) recibe los tres dados y el resultado de la partida, mostrando el mensaje oportuno.
- **run_dungeons** (módulo game_main) obtiene los valores aleatorios de los dados, obtiene el resultado e imprime el mismo a pantalla. Retorna el resultado de la partida. Llamará a las funciones antes implementadas.
- Extiende el main para que cuando el estado sea DUNGEONS_STATE ejecute dicho juego. Según el resultado de la partida, se pasará el estado a ROCK_PAPER_SCISSORS_STATE o a FINISH_LOSING.
  - Dónde se debe incrustar este trozo de código para cumplir con el flujograma de la Gymkhana?

Parte 3.4: Dungeons

Implementar el programa usando los módulos y las funciones indicadas.

- Para las siguientes funciones en el módulo gymkhana_logic.py se debe desarrollar la validación mediante **doctest**:
  - **get_dragons_realm_decision**
  - **dungeons**
- Para el resto de funciones, eliminar los errores en código por medio de la **depuración** (**debugging**).