# Documentación Entregable 1 - Programación Funcional

Integrantes: Barberousse Luciana y Sobral Alfonso

**Indicaciones de ejecución**

Para ejecutar el programa debe correrse el archivo game.py. Al iniciar, se solicitará seleccionar el modo de juego:

Modo simulación (s): El juego se desarrolla de manera automática. El usuario observa en la terminal la evolución de los jugadores en el tablero, las tiradas de dado y los efectos de bonus o penalizaciones. La partida finaliza cuando un jugador alcanza la casilla 30 o se queda sin monedas. En caso de que un jugador esté en la casilla 29 y obtenga un número mayor a 1, igualmente gana sin necesidad de un “rebote”. Una vez terminada la partida, se pregunta si se desea jugar nuevamente (s/n). En caso afirmativo, se reinicia el proceso de selección de modo.

Modo interactivo (i): Antes de comenzar se solicitan los nombres de los jugadores, que no pueden estar vacíos ni puede ser el mismo para ambos. El turno inicia con el primer nombre ingresado. Cada tirada de dado se realiza presionando Enter, y el programa informa el valor obtenido junto con los posibles bonus o penalizaciones en monedas y movimiento. En cada turno también se muestra el tablero actualizado con la posición de los jugadores. La partida finaliza cuando alguno alcanza la casilla 30 o se queda sin monedas.

**Decisiones de diseño y estructura**

*Estado del juego (state)*

Para representar el estado del juego decidimos utilizar un diccionario state, que encapsula toda la información necesaria para el desarrollo de la partida. Este enfoque permite trabajar de manera más clara y ordenada, ya que el estado no se modifica directamente, sino que en cada paso se genera un nuevo estado a partir del anterior (inmutabilidad).

El state es un diccionario (Dict[str, object]) con la siguiente estructura:

- "players": tupla con los nombres de los dos jugadores. Ejemplo: ("Ana", "Pepe").
- "positions": diccionario que indica la casilla actual de cada jugador. Ejemplo: {"Ana": 5, "Pepe": 3}.
- "coins": diccionario con la cantidad de monedas de cada jugador. Ejemplo: {"Ana": 2, "Pepe": 1}.
- "move": diccionario que asigna a ciertas casillas del tablero un bonus o penalización de movimiento. Ejemplo: {7: +2, 12: "reset"}.
- "econ": diccionario que asigna a ciertas casillas del tablero un bonus o penalización de monedas. Ejemplo: {8: +1, 15: -1}.

De esta manera, todo el flujo del juego puede modelarse a partir de este estado central, aplicando funciones puras que lo transforman paso a paso.

*Tablero en consola*

A la hora de realizar el tablero por consola hicimos uso de IA para poder llegar al resultado que queríamos pero siempre aprendiendo de lo que nos devuelve, por lo tanto decidimos dejarle una sección de este documento para describir lo que hace luego de haberla comprendido.

- Colores: Jugador1 en rojo, Jugador2 en azul; > magenta (movimiento), $ amarillo (economía).
- Celdas: números con dos dígitos para alinear (01, 02, …).
- Encabezado de turno: TURNO N — Jugador, para separar claramente la salida.

Funciones clave:
- visible_len y pad_center_visible: centran texto ignorando códigos ANSI.
- format_cell: compone la celda (jugador, > y/o $).
- render_board(state, boxes=BOXES, per_row=10, width=5): dibuja el tablero por filas (por defecto 10).

Limitación conocida (visual): si ambos jugadores caen en la misma casilla, se muestra sólo la inicial del primero. Es un compromiso estético de ancho fijo; se podría mejorar mostrando dos iniciales comprimidas (ej. [L|P]) si se quisiera, aunque optamos por dejar al que llegó primero.

Se conforma por 4 funciones:

1. def visible_len(s: str) -> int

Calcula la longitud visible de un string, eliminando los códigos de color ANSI (que no ocupan espacio real en la terminal). Esto es para alinear bien en consola, pues no importan los códigos de color.

2. def pad_center_visible(s: str, width: int) -> str

Centra un string en un espacio de ancho fijo, considerando solo los caracteres visibles (sin contar los códigos de color). Básicamente chequea cuanto ocupa el string que se debe imprimir en terminal, calcula la diferencia entre el espacio disponible y lo que ocupa para manejar dicho espacio libre así quede todo alineado.

3. def format_cell(i: int, state, width: int = 5) -> str

Construye el texto que representa una celda del tablero:

- Muestra el número de casilla (ej: 01, 02, ...). Decidimos utilizar dos dígitos para que cada número de casilla ocupe el mismo ancho y quede visualmente más estético.
- Si hay un jugador en esa casilla, muestra su inicial y la colorea. El jugador 1 aparece en rojo y el jugador 2 en azul.
- Si la casilla tiene bonus/penalización de salto, agrega un símbolo > magenta.
- Si la casilla tiene bonus/penalización de monedas, agrega un símbolo $ amarillo.
- Si hay ambos bonus se agregan ambos a la casilla con la misma lógica.

4. def render_board(state, boxes=BOXES, per_row=10, width=5): Dibuja el tablero completo en la terminal:

- Llama a format_cell para cada casilla del tablero.
- Agrupa las casillas en filas de 10 (por defecto). Este parámetro puede cambiarse dentro del código.
- Imprime cada fila.

**Algunas funciones**

- def generate_random_dice() -> Generator[int, None, None]

Esta función se encarga de generar tiradas infinitas de un dado de 6 caras. Utiliza *random.randint(1,6)* para producir un número aleatorio y yield para ir entregando cada valor. Gracias al uso de generadores, el estado se mantiene entre llamadas, lo que permite simular una secuencia ilimitada de lanzamientos.

- def generate_special_boxes() -> Tuple[Dict[int, object], Dict[int, int]]

Su función es crear casillas especiales en el tablero. Devuelve dos diccionarios:
- jumps: contiene casillas que generan efectos de movimiento (avanzar, retroceder o reiniciar a 0).
- econ: contiene casillas que otorgan o quitan monedas.

Para garantizar aleatoriedad, primero desordenan dos copias de las posiciones del tablero con random.shuffle, y luego asigna allí los bonus y penalizaciones. De esta forma, cada partida tiene una distribución distinta de casillas especiales, y permite que dos bonificaciones de distinto tipo aparezcan en la misma celda.

- def pure_step(state: State, player: str, throw: int) -> State

Esta función define cómo evoluciona el estado del juego tras un turno. Llamamos chain (cadena) a un evento poco usual en el juego, en el que un jugador tiene una muy buena racha de suerte y le aplican varias bonificaciones de corrido. Para esto, el código se encarga de calcular de forma recursiva todas las bonificaciones de la tirada aplicando una función. Primero, calcula el "paso cero" utilizando la posición inicial del jugador y el valor de la tirada del dado, y en base a esto, chequea las eventuales bonificaciones en cada "casilla intermedia" llamando a la función recursiva apply_chain.

El nombre pure_step resalta que es una función pura: no tiene efectos secundarios y siempre devuelve el mismo resultado dados los mismos parámetros.

- def apply_chain(pos: int, coins: int, calls: int = 0, cache=None) -> tuple[int, int]:

Esta función continua calculando bonificaciones hasta que el jugador cae en una casilla vacía o el juego termina. Además, utilizamos un "tope" para evitar que se llegue a un stack overflow por un bucle infinito (por ejemplo, que un jugador quede rebotando entre un +2 y un -2 de salto). Devuelve una tupla con la posición y las monedas finales del jugador.

- def endgame(state: State) -> bool:

Aquí chequeamos para cada jugador si llego al final, y si tiene monedas suficientes. Se utiliza para detectar si la partida terminó.
- Revisa si algún jugador llegó o superó la casilla final (BOXES) con al menos una moneda.
- Si ambos jugadores llegan al final, gana el que conserve más monedas (o se declara empate si tienen la misma cantidad).
- También contempla el caso en que un jugador se quede sin monedas en el medio de la partida, otorgándo la victoria automática al otro.

Es una función booleana (devuelve True si la partida terminó, o False en caso contrario).

- def simul(state: State, dice: Generator[int, None, None]) -> Generator[State, None, None]

Implementa el modo simulador del juego.

- Alterna los turnos de los jugadores, lanzando el dado con el generador dice.
- En cada turno actualiza el estado con pure_step y lo devuelve mediante yield.
- El bucle continúa hasta que endgame detecta que hay un ganador.

Esto permite recorrer la partida como una secuencia de estados, ideal para animaciones o ejecuciones automáticas.

**Implementación en "Python Funcional"**

Funciones puras

- ➔ compute_next_jump(position, bonus)
- ➔ compute_econ(coins, bonus)
- ➔ pure_step(state, player, throw) → retorna nuevo state.

Inmutabilidad

- ➔ No se muta state; se devuelve una copia con dict-unpacking ({**state, ...}).
- ➔ compute_econ asegura coins ≥ 0 con max(0, …).

Recursión

- ➔ apply_chain (anidada en pure_step) encadena premios/penas hasta alcanzar un estado estable o detectar ciclo.
- ➔ Detención garantizada: si no hay salto, corta; además, tope defensivo calls > 100 y cacheo de pares (pos, coins) para evitar bucles.

Generadores

- ➔ generate_random_dice() (flujo infinito de tiradas 1..6).
- ➔ simul(...) (flujo de estados previos al avance de cada turno).

Compresiones / map / filrter / reduce

- ➔ render_board usa map para celdas.
- ➔ En cada turno se listan jugadores con monedas con filter; el total de monedas usa reduce.

Composición de funciones

- ➔ En apply_chain se combinan compute_econ y compute_next_jump, y pure_step compone todo el flujo.

Decoradores de log

- ➔ @log_call en las funciones puras, con LOG_ENABLED para activar/desactivar.
- ➔ Muestra nombre de función, jugador/dado (si aplica) y un resumen (posición/monedas) sin spamear el estado completo.

**Guardado en .txt**

A la hora de realizar el guardado de la partida en un archivo .txt para tener registro no se respetó la inmutabilidad ni la pureza ya que:

1. El archivo .txt se edita en cada turno (no se crea uno nuevo).
2. La función create_log(players) recibe como parámetro players y crea un archivo con el nombre "jugador1vsjugador2" y se le agrega el Timestamp, por lo que para una misma entrada va a haber salidas diferentes.

La función create_log(players) se encarga de crear un archivo de texto único para registrar el desarrollo de cada partida. Utiliza los nombres de los jugadores y la fecha/hora actual para generar un nombre de archivo irrepetible, lo que permite guardar el historial de cada juego sin sobrescribir archivos anteriores. Así, cada vez que se inicia una partida, se crea automáticamente un archivo de log donde se puede ir escribiendo todo lo que sucede durante el juego: los turnos, los movimientos, los resultados de los dados y cualquier evento relevante.

Esto no solo facilita el seguimiento y la revisión de partidas pasadas, sino que también aporta un toque profesional y organizado al proyecto, permitiendo analizar jugadas, detectar errores o simplemente guardar recuerdos de las mejores partidas.

**Conclusiones de lo aprendido en el proyecto**

A lo largo de este proyecto logramos aplicar en un caso práctico los principales conceptos de la programación funcional dentro de Python, combinando teoría y práctica en un entorno lúdico (un juego por turnos). Entre los aprendizajes más relevantes destacamos:

Separación entre lógica pura y efectos de E/S: comprendimos la importancia de aislar las funciones puras (deterministas, sin efectos secundarios) de aquellas que interactúan con el exterior (impresión por consola, input del usuario, escritura en archivos). Esta distinción facilitó tanto el razonamiento sobre el código como la posibilidad de testear partes críticas de forma independiente.

Inmutabilidad como herramienta de claridad: al trabajar con un state inmutable, donde cada turno genera una copia actualizada en lugar de modificar directamente el original, experimentamos de primera mano cómo la inmutabilidad ayuda a prevenir errores difíciles de rastrear y hace más sencillo comprender la evolución del juego.

Uso de generadores: los generadores (yield) demostraron ser un recurso poderoso para modelar flujos potencialmente infinitos (como el dado o la secuencia de turnos). Esto no solo resultó útil para la simulación, sino que también nos permitió razonar de forma más declarativa sobre procesos que, en otro enfoque, habrían requerido estructuras más complejas.

Recursión y composición de funciones: la implementación de apply_chain nos permitió trabajar con un ejemplo claro de recursión controlada, asegurando la detención y evitando bucles infinitos. Asimismo, el diseño funcional favoreció la composición de funciones simples (por ejemplo, compute_econ y compute_next_jump) para construir lógica más compleja.

Compromisos prácticos: entendimos que no todo puede ser estrictamente funcional en un lenguaje multiparadigma como Python. Operaciones como el manejo de azar, la impresión en pantalla o la persistencia en archivos implican necesariamente impureza. Lo importante fue aprender a restringir esas impurezas a los bordes del programa, manteniendo el "núcleo" del juego lo más puro posible.

Más allá de la implementación técnica, este proyecto nos permitió ver cómo los principios de la programación funcional pueden aplicarse en contextos concretos y cotidianos, aportando claridad, modularidad y facilidad de mantenimiento.

En conclusión, el desarrollo del juego nos permitió consolidar conceptos fundamentales como funciones puras, inmutabilidad, recursión y generadores, a la vez que practicamos cómo lidiar con las inevitables limitaciones de la programación funcional en un lenguaje híbrido. El resultado final no solo fue un producto funcional y divertido, sino también una valiosa experiencia de aprendizaje que refuerza la utilidad práctica de este paradigma en el diseño de software.

*Inmutabilidad*

- Dónde sí:
    - pure_step no muta el estado; retorna nuevos diccionarios con las actualizaciones.
    - Funciones puras (compute_*) son deterministas y fáciles de testear.
    - El flujo de turnos (simul) y el dado (generate_random_dice) separan la generación del consumo de valores (composición).

- Dónde no / límites prácticos:
    - La E/S (prints, input) es inherentemente impura. La aislamos en funciones de orquestación (describe_and_apply_turn y main) para mantener el “núcleo” de lógica como puro.
    - El azar también es impuro; lo encapsulamos en generadores/constructores para poderlo “inyectar” (e incluso fijar con semilla).
    - El guardado en un archivo .txt de la partida.

En síntesis, logramos mantener el corazón del juego (reglas, transiciones de estado) en un estilo funcional (puro e inmutable por paso), relegando efectos a los bordes del programa.