

CÉSAR MANRIQUE

2ºDAW

ESTRUCTURAS DE CONTROL EN PYTHON

LABERINTO DE TEXTO



Eudaldo Álvaro Cal Saúl

Rodrigo Sanmartín Alaejos

Pablo Díaz Hernández

Índice

<i>INTRODUCCIÓN</i>	1
<i>OBJETIVOS</i>	2
ARQUITECTURA DEL PROYECTO	3
<i>ESTRUCTURA DEL PROGRAMA</i>	4
MAIN.PY	4
JUEGO.PY	5
MOVIMIENTO.PY	10
ACCIONES.PY	14
MUNDO.PY	17
ESTADO.PY	20
<i>MAPA DEL LABERINTO DE TEXTO</i>	22
<i>COMANDOS DISPONIBLES</i>	23
DECISIONES DE DISEÑO Y JUSTIFICACIÓN	24
USO DE FUNCIONES BUILT-IN Y DEFINIDAS POR EL USUARIO	27
DOCUMENTACIÓN DEL CÓDIGO CON SPHINX	29
Contenido Documentado	30
Ejemplo de Docstring Utilizado	31
Ventajas para el Desarrollo y Mantenimiento	32
<i>CONCLUSIÓN</i>	33
<i>BIBLIOGRAFÍA</i>	34

INTRODUCCIÓN

Este proyecto consiste en el desarrollo de un juego de aventura textual en Python, titulado Laberinto de Texto. El objetivo del juego es que el jugador explore un laberinto formado por distintas habitaciones, recoja objetos clave y consiga encontrar la salida.

El proyecto se basa en el **Reto 1: Diseño de Laberinto Simplificado**, que consiste en diseñar un laberinto con entre 8 y 10 habitaciones interconectadas, cada una con su propia descripción y con salidas en las cuatro direcciones básicas (N, S, E, O). Además, se implementa un sistema de inventario básico para gestionar entre 3 y 5 objetos clave y un mapa simple que muestra las habitaciones que el jugador ya ha visitado.

El juego se ejecuta por consola y se controla mediante la introducción de comandos de texto, permitiendo al jugador moverse por el laberinto, inspeccionar objetos y gestionar su inventario. A lo largo de la partida, el jugador debe interpretar la información que va encontrando para avanzar correctamente.

Además de los requisitos del **Reto 1**, se han añadido objetos con los que el jugador puede interactuar, como un pergamo que proporciona pistas sobre el laberinto y una llave necesaria para abrir la puerta final, lo que aporta un pequeño componente de exploración y lógica al juego.

El proyecto se ha desarrollado utilizando estructuras básicas de Python, como variables, condicionales, funciones, listas y diccionarios, prestando especial atención a la claridad del código y a una organización modular. Cada archivo del programa tiene una responsabilidad concreta, lo que facilita la comprensión y el mantenimiento del código.

Con este trabajo se pretende aplicar los conocimientos adquiridos en la asignatura, reforzando conceptos como la programación estructurada, el control de errores y la separación de responsabilidades en un programa de tamaño medio.

OBJETIVOS

Desarrollar un **juego de aventura textual en Python** que permita al jugador explorar un laberinto, interactuar con objetos y encontrar la salida, aplicando los conocimientos básicos de programación vistos en clase.

Objetivos específicos:

Diseñar un laberinto formado por varias habitaciones interconectadas, cada una con su propia descripción y salidas en las direcciones básicas.

Implementar un sistema de movimiento que permita al jugador desplazarse entre las distintas salas mediante comandos de texto.

Crear un sistema de inventario sencillo para recoger y soltar objetos durante la partida.

Incorporar objetos interactivos que aporten información o sean necesarios para completar el juego, como el pergamo y la llave.

Desarrollar un sistema de inspección de objetos para mostrar descripciones y pistas al jugador.

Implementar un mapa simple que muestre las habitaciones visitadas por el jugador.

Aplicar el manejo de errores para controlar entradas incorrectas y mejorar la experiencia de juego.

Organizar el código de forma modular, separando las distintas responsabilidades en varios archivos para facilitar su comprensión y mantenimiento.

ARQUITECTURA DEL PROYECTO

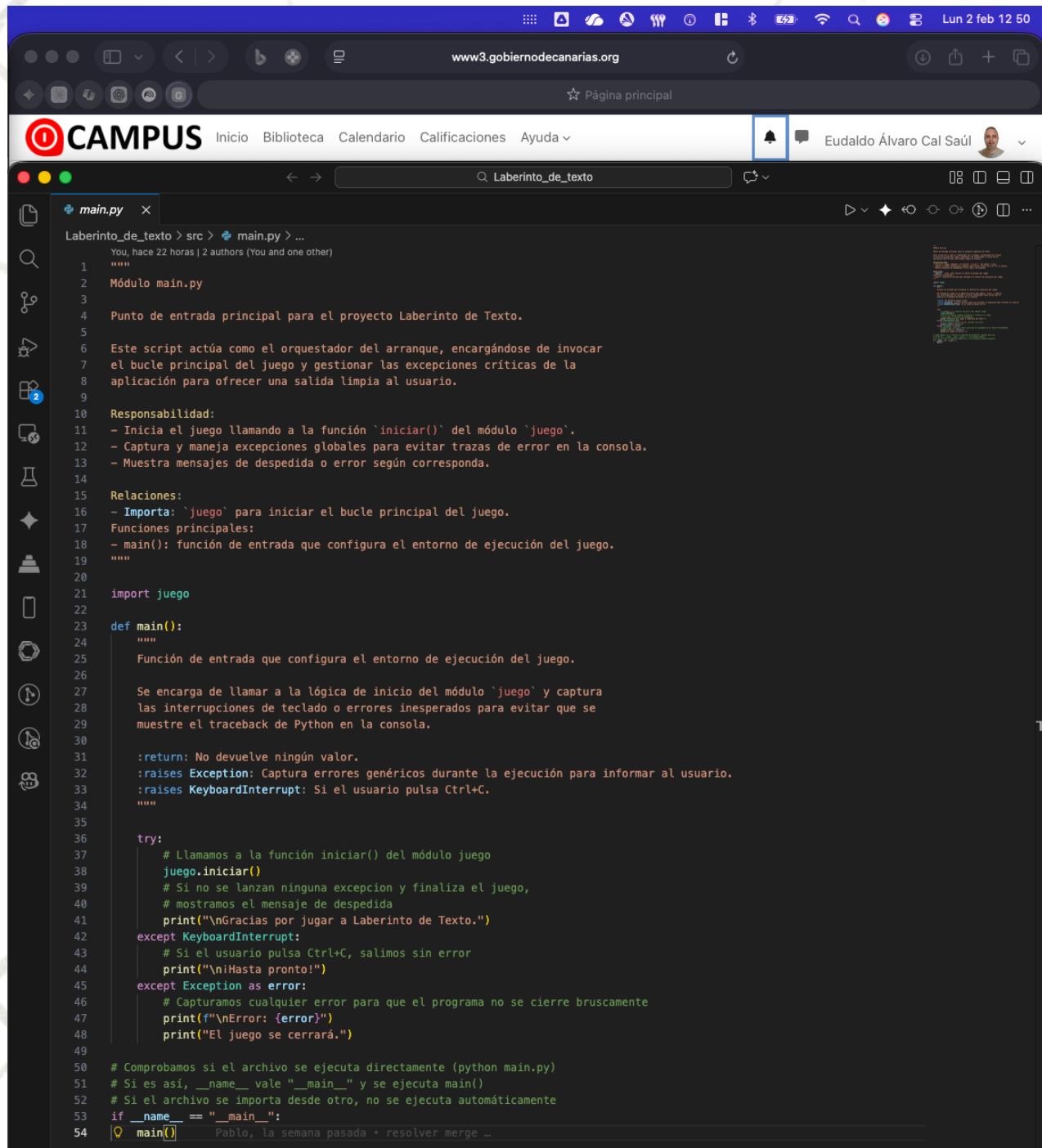
Nuestro ha sido diseñado siguiendo el principio de modularización y separación de responsabilidades. Esta arquitectura permite que cada módulo tenga una función específica y bien definida, facilitando el mantenimiento, la depuración y la escalabilidad del código.

El sistema se compone de seis módulos principales que interactúan entre sí de manera coordinada:

main.py	- Punto de entrada de la aplicación
juego.py	- Bucle principal y parser de comandos
estado.py	- Gestión del estado global del juego
mando.py	- Base de datos del laberinto
movimiento.py	- Lógica de navegación y exploración
acciones.py	- Gestión de objetos e inventario

ESTRUCTURA DEL PROGRAMA

MAIN.PY



The screenshot shows a web-based code editor interface. At the top, there's a toolbar with various icons. Below it, the URL bar shows "www3.gobiernodecanarias.org". The main area is a code editor with a dark theme, displaying Python code for "main.py". The code is annotated with comments explaining its functionality. The right side of the editor has a sidebar with some icons and a preview window showing the code structure.

```
main.py
-----
Laberinto_de_texto > src > main.py > ...
You, hace 22 horas | 2 authors (You and one other)
1 """
2 Módulo main.py
3
4 Punto de entrada principal para el proyecto Laberinto de Texto.
5
6 Este script actúa como el orquestador del arranque, encargándose de invocar
7 el bucle principal del juego y gestionar las excepciones críticas de la
8 aplicación para ofrecer una salida limpia al usuario.
9
10 Responsabilidad:
11 - Inicia el juego llamando a la función `iniciar()` del módulo `juego`.
12 - Captura y maneja excepciones globales para evitar trazas de error en la consola.
13 - Muestra mensajes de despedida o error según corresponda.
14
15 Relaciones:
16 - Importa: `juego` para iniciar el bucle principal del juego.
17 Funciones principales:
18 - main(): función de entrada que configura el entorno de ejecución del juego.
19 """
20
21 import juego
22
23 def main():
24     """
25     Función de entrada que configura el entorno de ejecución del juego.
26
27     Se encarga de llamar a la lógica de inicio del módulo `juego` y captura
28     las interrupciones de teclado o errores inesperados para evitar que se
29     muestre el traceback de Python en la consola.
30
31     :return: No devuelve ningún valor.
32     :raises Exception: Captura errores genéricos durante la ejecución para informar al usuario.
33     :raises KeyboardInterrupt: Si el usuario pulsa Ctrl+C.
34     """
35
36     try:
37         # Llamamos a la función iniciar() del módulo juego
38         juego.iniciar()
39         # Si no se lanzan ninguna excepción y finaliza el juego,
40         # mostramos el mensaje de despedida
41         print("\nGracias por jugar a Laberinto de Texto.")
42     except KeyboardInterrupt:
43         # Si el usuario pulsa Ctrl+C, salimos sin error
44         print("\nHasta pronto!")
45     except Exception as error:
46         # Capturamos cualquier error para que el programa no se cierre bruscamente
47         print(f"\nError: {error}")
48         print("El juego se cerrará.")
49
50     # Comprobamos si el archivo se ejecuta directamente (python main.py)
51     # Si es así, __name__ vale "__main__" y se ejecuta main()
52     # Si el archivo se importa desde otro, no se ejecuta automáticamente
53     if __name__ == "__main__":
54         main()  Pablo, la semana pasada + resolver merge ..
```

Es el punto de entrada del programa. Su única función es iniciar el juego llamando a `juego.iniciar()` y capturar posibles errores generales para que el programa no se cierre de forma brusca.

Se llama a la función `iniciar()`, que arranca el bucle principal del juego. Además, utiliza una estructura `try / except` para capturar posibles errores y permitir que el programa se cierre de forma controlada, incluyendo la interrupción por teclado mediante `Control + C`, evitando que el juego finalice de manera brusca.

Características principales:

- Manejo de excepciones global: Captura `KeyboardInterrupt (Ctrl+C)` y excepciones genéricas para evitar trazas de error desagradables al usuario.
- Separación de responsabilidades: No contiene lógica de juego, solo gestiona el arranque y la terminación.
- Punto de entrada controlado: Utiliza la convención `if __name__ == '__main__'` para permitir que el módulo sea importado sin ejecutarse automáticamente.
- Función clave: `main()` - Envuelve la llamada a `juego.iniciar()` en un bloque `try-except` robusto.

JUEGO.PY

Este archivo contiene el bucle principal del juego y el intérprete de comandos. Se encarga de mostrar los mensajes de bienvenida, las respuestas y los mensajes de errores. Además, lee lo que escribe el jugador, interpreta el comando y lo delega al módulo correspondiente. Por último, comprueba la condición de victoria cuando el jugador llega a la salida con la llave.

Características principales:

- Parser de comandos inteligente: La función `interpretar()` divide la entrada del usuario y delega a los módulos apropiados.
- Validación de entrada: Detecta comandos vacíos, argumentos faltantes y comandos desconocidos.
- Sistema de ayuda integrado: Constante `AYUDA` con todos los comandos disponibles.

- Condición de victoria automática: Detecta cuando el jugador alcanza la sala 'salida' con la 'llave' y finaliza el juego.
- Manejo de excepciones específico: Captura `NotImplementedError`, `ValueError` y `Exception` con mensajes informativos.
- Funciones clave:
- `interpretar(linea)`: Analiza el comando y devuelve la respuesta apropiada.
- `iniciar()`: Contiene el bucle principal del juego, mostrando la sala inicial y procesando comandos hasta que se cumpla la condición de victoria o el usuario salga.

A continuación, se explicamos las dos funciones más importantes del archivo `juego.py`, ya que concentran la lógica principal del funcionamiento del juego.

La función `interpretar()`:

Recibe la línea de texto introducida por el usuario, la divide en comandos y argumentos, y decide qué acción ejecutar en función del comando escrito. Para ello, delega las acciones en otros módulos del programa, como `movimiento.py` para el desplazamiento entre salas y `acciones.py` para la gestión del inventario y la inspección de objetos. Además, controla los comandos no válidos y devuelve mensajes adecuados al jugador.

Lun 2 feb 13:25

www3.gobiernodecanarias.org Página principal

CAMPUS Inicio Biblioteca Calendario Calificaciones Ayuda ▾

Eudaldo Álvaro Cal Saúl

Laberinto_de_texto

```
main.py iuego.py
```

```
45 def interpretar(linea) -> str:
46     """
47         Interpreta el comando introducido por el usuario y ejecuta la acción correspondiente.
48
49         Divide la cadena de entrada y determina mediante el primer token qué función
50         del motor de juego debe invocarse.
51
52         :param linea: La línea de comando completa introducida por el usuario.
53         :type linea: str
54         :return: El mensaje de respuesta generado tras la acción o un código de salida.
55         :rtype: str
56     """
57
58     # Dividimos la línea en palabras y limpiamos espacios
59     palabras = linea.strip().split()
60
61     # Si no hay palabras, devolvemos cadena vacía
62     if not palabras:
63         return ""
64
65     # Almacena la primera palabra como el comando y el resto como argumentos
66     comando = palabras[0].lower()
67     argumentos = palabras[1:]
68
69     # Si el comando es una dirección (n, s, e, o), llamamos a mover
70     if comando in ("n", "s", "e", "o"):
71         return movimiento.mover(comando)
72
73     # Si el comando es "ir" con dirección (n, s, e, o), llamamos a mover con
74     # la dirección asignada, si no hay dirección, mensaje de error
75     if comando == "ir":
76         if argumentos:
77             return movimiento.mover(argumentos[0])
78         else:
79             return "Te falta la dirección. Ejemplo: ir n"
80
81
82     # Muestra el mapa con las salas visitadas si el comando es "mapa"
83     if comando == "mapa":
84         if argumentos:
85             return "El comando 'mapa' no necesita argumentos."
86         return movimiento.mapa()
87
88     # Si el comando es "reiniciar", reinicia ubicación, inventario, visitadas y victoria.
89     # Finalmente muestra la sala inicial otra vez.
90     if comando == "reiniciar":
91         estado.resetear()
92         return movimiento.mirar()
93
94     # Si el comando es "mirar", llamamos a mirar
95     if comando == "mirar":
96         return movimiento.mirar()
97
98     # Si el comando es "inventario", llamamos a inventario_str
99     if comando == "inventario":
100        return acciones.inventario()
101
102    if comando == "coger" and argumentos:
```

A screenshot of a Mac OS X desktop environment. The window title is "www3.gobiernodecanarias.org". The page content is titled "CAMPUS" and shows a file editor with two tabs: "main.py" and "juego.py". The "juego.py" tab is active, displaying Python code for a text-based labyrinth game. The code defines a function "interpretar" that handles commands like "coger", "soltar", "inspeccionar", and "ayuda". It also checks for the command "salir" and provides help text if the command is not recognized.

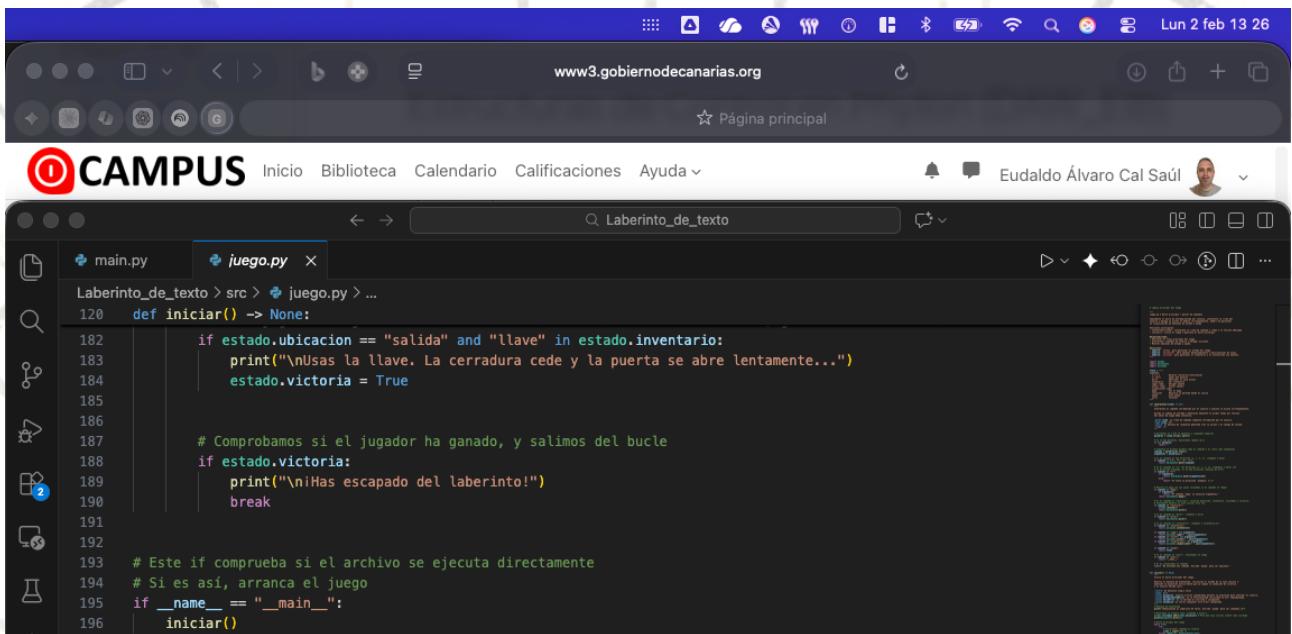
```
Laberinto_de_texto > src > juego.py > ...
45 def interpretar(linea) -> str:
101     if comando == "coger" and argumentos:
102         return acciones.coger(" ".join(argumentos))
103     if comando == "soltar" and argumentos:
104         return acciones.soltar(" ".join(argumentos))
105     if comando == "inspeccionar" and argumentos:
106         return acciones.inspeccionar(" ".join(argumentos))
107
108     if comando == "ayuda":
109         return AYUDA
110
111     # Si el comando es "salir", terminamos el juego
112     if comando == "salir":
113         return "__EXIT__"
114
115
116     # Si no reconocemos el comando
117     return "No entiendo ese comando. Escribe 'ayuda' para ver opciones."
118
```

La función iniciar():

Es la encargada de arrancar el juego. Muestra el mensaje de bienvenida, enseña la información de la sala inicial y mantiene el bucle principal de la partida. Dentro de este bucle se leen los comandos del jugador, se controlan posibles errores mediante estructuras try y except y se comprueba la condición de victoria cuando el jugador llega a la sala de salida con la llave en el inventario.

The screenshot shows a Mac OS X desktop environment with a terminal window open. The terminal window has two tabs: 'main.py' and 'juego.py'. The 'juego.py' tab is active and displays the following Python code:

```
Laberinto_de_texto > src > juego.py > ...
120 def iniciar() -> None:
121     Muestra el mensaje de bienvenida, inicializa el estado de la sala inicial y
122     mantiene la ejecución activa hasta que se cumpla la condición de victoria
123     o el usuario decide salir.
124
125     :return: No devuelve ningún valor.
126     :rtype: None
127     :raises Exception: Captura errores inesperados durante la ejecución para informar al usuario.
128     :raises NotImplementedError: Si la funcionalidad solicitada no está implementada.
129     :raises ValueError: Si la entrada es inválida o incompleta.
130     :raises Exception: Si ocurre cualquier otro error inesperado.
131     """
132
133     # Mensaje de bienvenida
134     print("\nBienvenido al Laberinto de Texto. Escribe 'ayuda' para ver comandos.\n")
135
136     # Mostramos la primera sala, llamando a mirar()
137     estado.visitadas.add(estado.ubicacion) # Para que sala inicial cuente como visitada
138     print(movimiento.mirar())
139
140     # Bucle principal del juego
141     while True:
142         try:
143             # Solicitamos comando al usuario
144             linea = input("\n> ")
145         except (EOFError, KeyboardInterrupt):
146             # Si el usuario pulsa Ctrl+C o Ctrl+D
147             print("\nInterrumpido.")
148             break
149
150         # Si la línea está vacía, solicitamos otro comando
151         if not linea.strip():
152             continue
153
154         try:
155             # Interpretamos el comando
156             respuesta = interpretar(linea)
157         except NotImplementedError as error:
158             # Funcionalidad aún no implementada
159             print("[Aún no disponible] {error}")
160             continue
161         except ValueError as error:
162             # Entrada no válida (por ejemplo, datos incorrectos)
163             print("[Entrada no válida] {error}")
164             continue
165         except Exception as error:
166             # Cualquier otro error inesperado
167             print("[Error inesperado] {error}")
168             continue
169
170         # Si el usuario escribió "salir", terminamos el juego
171         if respuesta == "__EXIT__":
172             print("¡Hasta pronto!")
173             break
174
175         # Mostramos la respuesta si existe
176         if respuesta:
177             print(respuesta)
178
179
180
```



```
main.py juego.py
Laberinto_de_texto > src > juego.py > ...
120 def iniciar() -> None:
182     if estado.ubicacion == "salida" and "llave" in estado.inventario:
183         print("\nUsas la llave. La cerradura cede y la puerta se abre lentamente...")
184         estado.victoria = True
185
186
187     # Comprobamos si el jugador ha ganado, y salimos del bucle
188     if estado.victoria:
189         print("\nHas escapado del laberinto!")
190         break
191
192
193     # Este if comprueba si el archivo se ejecuta directamente
194     # Si es así, arranca el juego
195     if __name__ == "__main__":
196         iniciar()
```

MOVIMIENTO.PY

Este archivo se encarga del movimiento del jugador y de mostrar información relacionada con la sala actual. Para ello utiliza **estado.py** para conocer y actualizar ubicación del jugador, y **mundo.py** para consultar descripciones, salidas y objetos disponibles en cada sala.

Centraliza el estado de la sesión de juego mediante variables globales, actuando como la 'memoria' del sistema.

Variables globales:

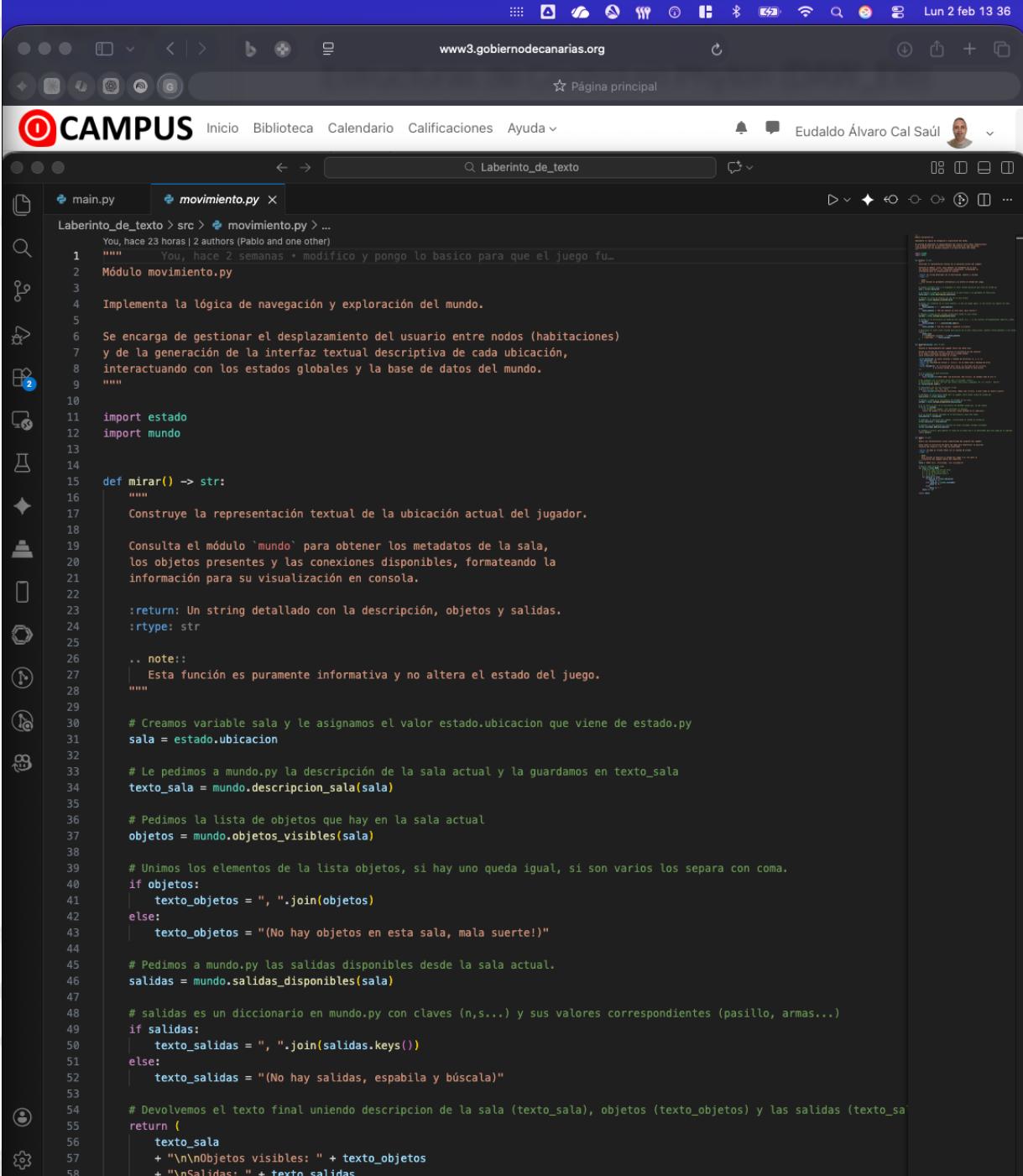
- **ubicacion**: String que identifica la sala actual (inicialmente 'entrada').
- **inventario**: Lista de objetos que el jugador ha recogido (inicialmente vacía).
- **visitadas**: Conjunto (set) de salas que el jugador ha explorado.
- **victoria**: Booleano que indica si el jugador ha completado el juego.

Función clave:

- **resetear ()**: Reinicia todas las variables a sus valores iniciales, permitiendo al jugador comenzar una nueva partida sin reiniciar el programa. Utiliza la palabra clave 'global' para modificar las variables del módulo.

A continuación, se explicamos las funciones más importantes del archivo **movimiento.py**, ya que gestionan la navegación y la información que ve el jugador durante la partida.

La función **mirar()**:



The screenshot shows a web-based code editor or notebook interface. The title bar indicates the URL is www3.gobiernodecanarias.org. The main content area displays the Python code for the **mirar()** function:

```
1  """ You, hace 23 horas | 2 authors (Pablo and one other)
2  Módulo movimiento.py
3
4  Implementa la lógica de navegación y exploración del mundo.
5
6  Se encarga de gestionar el desplazamiento del usuario entre nodos (habitaciones),
7  y de la generación de la interfaz textual descriptiva de cada ubicación,
8  interactuando con los estados globales y la base de datos del mundo.
9 """
10
11 import estado
12 import mundo
13
14 def mirar() -> str:
15     """
16         Construye la representación textual de la ubicación actual del jugador.
17
18         Consulta el módulo `mundo` para obtener los metadatos de la sala,
19         los objetos presentes y las conexiones disponibles, formateando la
20         información para su visualización en consola.
21
22         :return: Un string detallado con la descripción, objetos y salidas.
23         :rtype: str
24
25         .. note::
26             Esta función es puramente informativa y no altera el estado del juego.
27
28
29
30         # Creamos variable sala y le asignamos el valor estado.ubicación que viene de estado.py
31         sala = estado.ubicación
32
33         # Le pedimos a mundo.py la descripción de la sala actual y la guardamos en texto_sala
34         texto_sala = mundo.descripcion_sala(sala)
35
36         # Pedimos la lista de objetos que hay en la sala actual
37         objetos = mundo.objetos_visibles(sala)
38
39         # Unimos los elementos de la lista objetos, si hay uno queda igual, si son varios los separa con coma.
40         if objetos:
41             texto_objetos = ", ".join(objetos)
42         else:
43             texto_objetos = "(No hay objetos en esta sala, mala suerte!)"
44
45         # Pedimos a mundo.py las salidas disponibles desde la sala actual.
46         salidas = mundo.salidas_disponibles(sala)
47
48         # salidas es un diccionario en mundo.py con claves (n,s,...) y sus valores correspondientes (pasillo, armas...)
49         if salidas:
50             texto_salidas = ", ".join(salidas.keys())
51         else:
52             texto_salidas = "(No hay salidas, espabilá y búscalas)"
53
54         # Devolvemos el texto final uniendo descripción de la sala (texto_sala), objetos (texto_objetos) y las salidas (texto_sa
55         return (
56             texto_sala
57             + "\n\nObjetos visibles: " + texto_objetos
58             + "\nSalidas: " + texto_salidas
```

Construye y devuelve un texto con la información de la sala actual. Muestra el nombre y la descripción de la habitación, los objetos visibles en el suelo y las direcciones disponibles para moverse. Esta función no cambia el estado del juego, solo genera un mensaje informativo para que el jugador sepa dónde está y qué puede hacer.

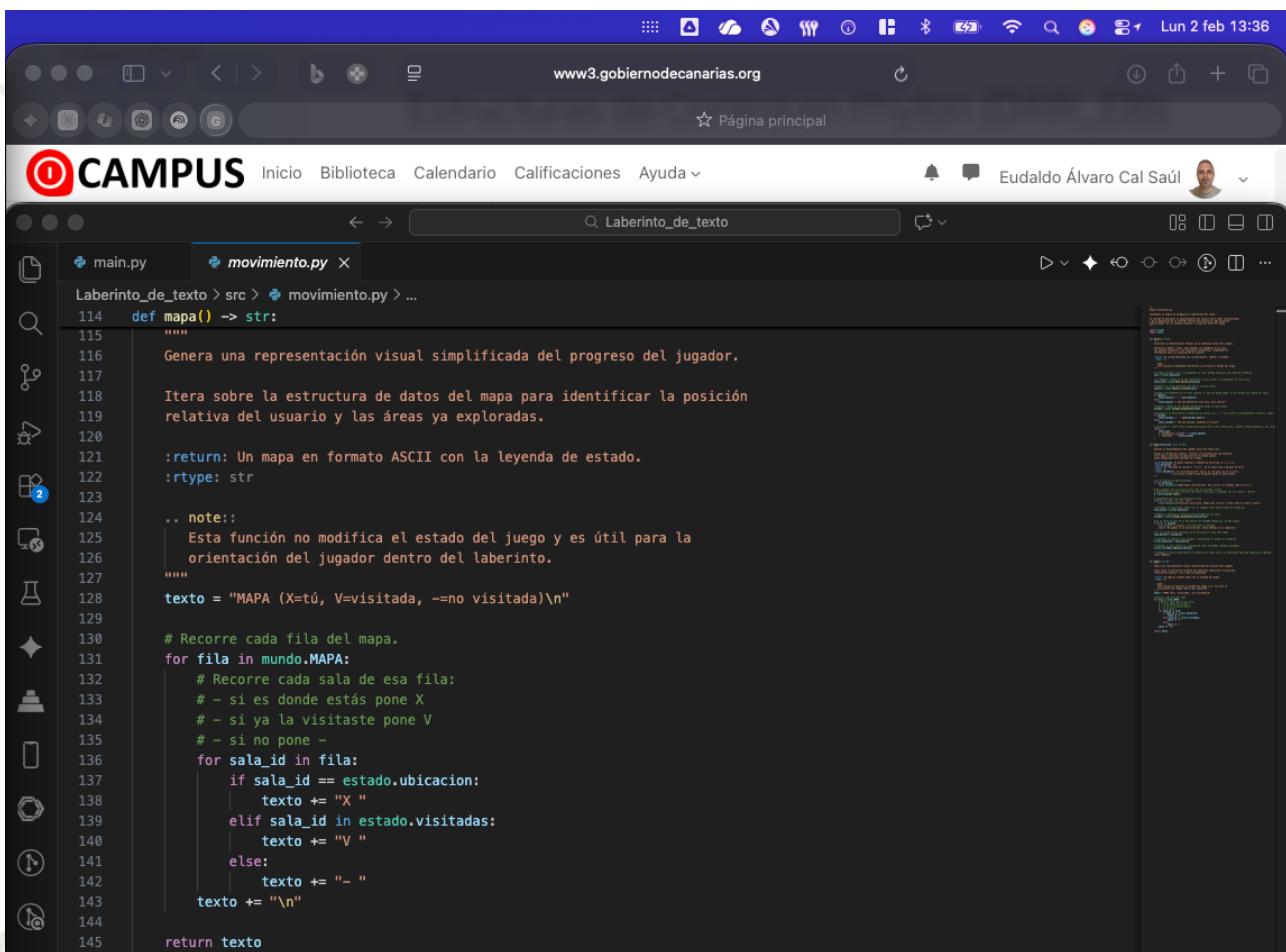
La función **mover(direccion):**

```
main.py | movimiento.py < ...>

61
62 def mover(direccion: str) -> str:
63     """
64     Ejecuta el desplazamiento del jugador hacia una nueva sala.
65
66     Valida la entrada del usuario, verifica la existencia de una conexión
67     en el mapa y actualiza la ubicación en el estado global.
68     Lanza excepciones ante entradas no válidas.
69
70     :param dirección: El punto cardinal o comando de dirección (n, s, e, o).
71     :type dirección: str
72     :return: El resultado de invocar a `mirar()` en la nueva sala o mensaje de error.
73     :rtype: str
74     :raises ValueError: Si la dirección está vacía, es inválida (no es n/s/e/o),
75     |   o no existe salida en esa dirección desde la sala actual.
76     """
77
78     # Si el usuario no puso dirección.
79     if not dirección:
80         raise ValueError("Debe haber una dirección. Usa n/s/e/o, no tenemos todo el día.")
81
82     # Nos quedamos con la primera letra (por si escriben 'norte')
83     # Convertimos el primer carácter del texto a minúscula, aceptamos "N", "n", "norte", "Norte".
84     d = dirección[0].lower()
85
86     # Comprobamos que sea una dirección válida
87     if d not in ("n", "s", "e", "o"):
88         raise ValueError("Dirección incorrecta. Debes usar n/s/e/o. A este ritmo no saldrás nunca")
89
90     # Guardamos en sala_actual dónde está el jugador ahora mismo (viene de estado.py)
91     sala_actual = estado.ubicacion
92
93     # Pedimos a mundo.py el diccionario de SALIDAS de esa sala.
94     salidas = mundo.salidas_disponibles(sala_actual)
95
96     # Si la letra no está en el diccionario de SALIDAS (mundo.py), no hay camino.
97     if d not in salidas:
98         # No movemos al jugador, solo mostramos el mensaje.
99         return "No puedes ir en esa dirección, estas perdido en el laberinto."
100
101     # Si la salida existe, buscamos en el diccionario a que sala vamos.
102     sala_destino = salidas[d]
103
104     # Cambiamos la ubicación del jugador, actualizando el estado en estado.py
105     estado.ubicacion = sala_destino
106
107     # Añadimos la sala destino al conjunto de salas visitadas (estado.visitadas)
108     estado.visitadas.add(sala_destino)
109
110     # Llamamos a mirar() para generar el texto de la nueva sala y lo devolvemos para que juego.py lo imprima.
111     return mirar()
112
```

Gestiona el desplazamiento del jugador entre salas. Primero valida la dirección introducida (aceptando n, s, e, o y también palabras como “norte” usando solo la primera letra). Si la dirección no es válida lanza un ValueError. Si no existe salida en esa dirección, devuelve un mensaje indicando que no se puede avanzar. Si la salida existe, actualiza estado.ubicacion, añade la sala al conjunto estado.visitadas y devuelve el resultado de mirar() para mostrar la nueva habitación.

La función mapa():



The screenshot shows a web-based code editor interface. At the top, there's a toolbar with various icons. Below it, the address bar shows "www3.gobiernodecanarias.org". The main area displays two tabs: "main.py" and "movimiento.py". The "movimiento.py" tab is active and contains the following Python code:

```
114 def mapa() -> str:
115     """
116         Genera una representación visual simplificada del progreso del jugador.
117
118         Itera sobre la estructura de datos del mapa para identificar la posición
119         relativa del usuario y las áreas ya exploradas.
120
121     :return: Un mapa en formato ASCII con la leyenda de estado.
122     :rtype: str
123
124     .. note::
125         Esta función no modifica el estado del juego y es útil para la
126         orientación del jugador dentro del laberinto.
127
128     texto = "MAPA (X=tú, V=visitada, --no visitada)\n"
129
130     # Recorre cada fila del mapa.
131     for fila in mundo.MAPA:
132         # Recorre cada sala de esa fila:
133         # - si es donde estás pone X
134         # - si ya la visitaste pone V
135         # - si no pone -
136         for sala_id in fila:
137             if sala_id == estado.ubicacion:
138                 texto += "X "
139             elif sala_id in estado.visitadas:
140                 texto += "V "
141             else:
142                 texto += "- "
143
144     texto += "\n"
145
146     return texto
```

Devuelve una representación simple del mapa del laberinto, indicando con X la posición actual del jugador, con V las salas ya visitadas y con - las salas todavía no visitadas. De esta forma el jugador puede orientarse y ver su progreso dentro del laberinto.

ACCIONES.PY

Este archivo se encarga de las acciones del jugador sobre los objetos del juego. Gestiona el inventario y permite interactuar con los objetos que se encuentran en las distintas salas. Para ello, utiliza `estado.py` para modificar el inventario y la ubicación del jugador, y `mundo.py` para comprobar qué objetos hay disponibles en cada sala.

Propósito: Implementa la lógica de interacción entre el jugador y los objetos del mundo, gestionando el inventario y las inspecciones detalladas.

Constante OBJ_DESCS: Diccionario que almacena las descripciones detalladas de cada objeto del juego.

Funciones principales:

- **coger(obj):** Verifica que el objeto esté en la sala actual, lo elimina de `mundo.OBJETOS_EN_SALA` y lo añade a `estado.inventario`. Retorna mensaje de confirmación o error.
- **soltar(obj):** Operación inversa a `coger()`. Valida que el objeto esté en el inventario antes de transferirlo a la sala actual.
- **inventario():** Genera representación textual del inventario, manejando el caso vacío con un mensaje apropiado.
- **inspeccionar(obj):** Devuelve descripción detallada si el objeto es accesible (en inventario o visible en sala). Incluye lógica especial para el 'pergamino', que solo puede leerse si está en posesión del jugador.

A continuación, se explicamos las funciones más importantes del archivo `acciones.py`, ya que controlan la interacción del jugador con los objetos.

La función coger():

Permite recoger un objeto que se encuentra en la sala actual. La función comprueba si el objeto está visible en la habitación y, si es así, lo elimina de la sala y lo añade al inventario del jugador. Si el objeto no está presente, se devuelve un mensaje indicando que no se puede coger.

A screenshot of a web-based code editor on a Mac OS X interface. The title bar shows the URL www3.gobiernodecanarias.org. The main window displays Python code for a game. The file path is `Laberinto_de_texto > src > acciones.py`. The code defines a function `coger` that moves an object from the player's inventory to the room's objects list. It includes docstrings and type annotations.

```
37 def coger(obj: str) -> str:
38     """
39         Transfiere un objeto desde la habitación actual al inventario del jugador.
40
41         Verifica la existencia del objeto en la sala y, en caso positivo, realiza
42         la mutación de las listas correspondientes en el estado global y el mundo.
43
44         :param obj: Nombre del objeto que se desea recoger.
45         :type obj: str
46         :return: Mensaje informativo sobre el éxito o fracaso de la acción.
47         :rtype: str
48
49         .. note::
50             Esta función modifica directamente la colección de objetos de la sala
51             en el módulo `mundo`.
52
53         objs = mundo.objetos_visibles(estado.ubicacion)
54         if obj not in objs:
55             return f"No hay '{obj}' aquí."
56
57         # Si está, lo quita de la sala(remove) y lo mete en inventario(append).
58         mundo.OBJETOS_EN_SALA[estado.ubicacion].remove(obj)
59         estado.inventario.append(obj)
60
61         # Finalmente devuelve texto de confirmación.
62         return f"Has cogido '{obj}'."
```

La función `soltar()`:

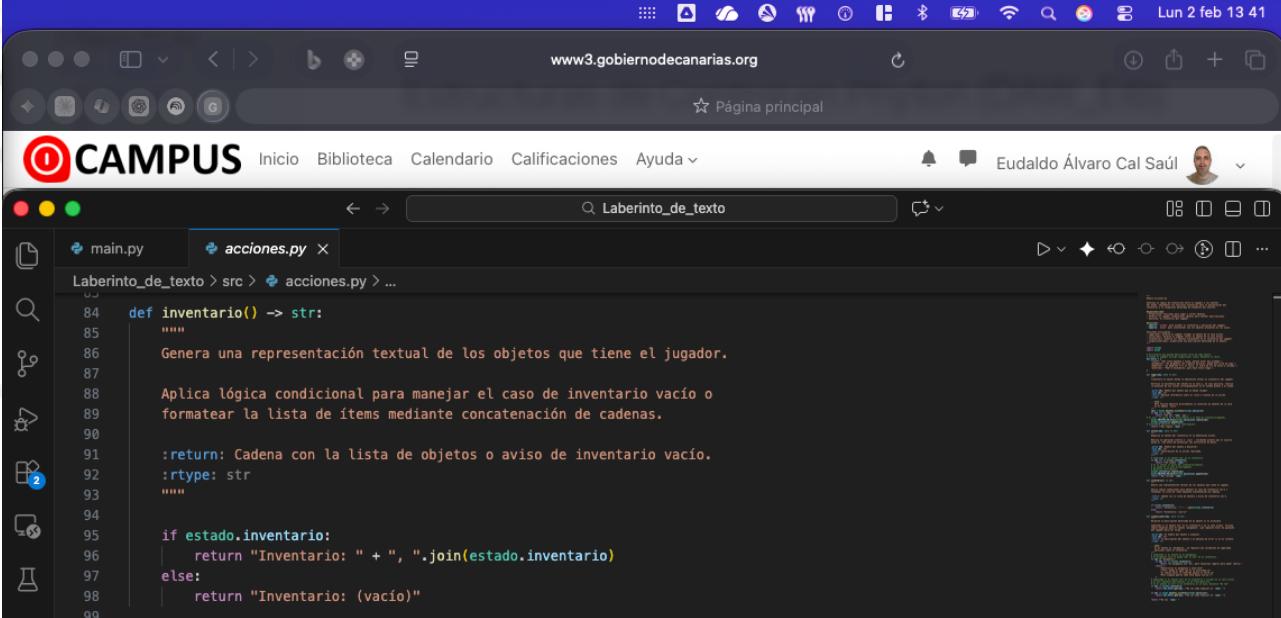
Permite dejar un objeto del inventario en la sala actual. Primero se comprueba si el jugador tiene el objeto en su inventario. Si no lo tiene, se muestra un mensaje de error. Si lo tiene, el objeto se elimina del inventario y se añade a la lista de objetos de la sala.

A screenshot of a web-based code editor on a Mac OS X interface. The title bar shows the URL www3.gobiernodecanarias.org. The main window displays Python code for a game. The file path is `Laberinto_de_texto > src > acciones.py`. The code defines a function `soltar` that moves an object from the room's objects list back to the player's inventory. It includes docstrings and type annotations.

```
61 def soltar(obj: str) -> str:
62     """
63         Deposita un objeto del inventario en la habitación actual.
64
65         Realiza la operación inversa a `coger`, validando primero que el usuario
66         posea el ítem antes de actualizar las estructuras de datos.
67
68         :param obj: Nombre del objeto a descartar.
69         :type obj: str
70         :return: Confirmación de la acción realizada.
71         :rtype: str
72
73         """
74
75         # Comprueba si el objeto está en el inventario.
76         if obj not in estado.inventario:
77             return f"No tienes '{obj}'."
78
79         # Si lo tienes lo quita del inventario(remove).
80         # Lo añade en la sala actual(append).
81         # Devuelve confirmación.
82         mundo.OBJETOS_EN_SALA[estado.ubicacion].append(obj)
83         estado.inventario.remove(obj)
84
85         # Finalmente devolverá el texto de confirmación.
86         return f"Has soltado '{obj}'."
```

La función inventario():

Devuelve un texto con los objetos que lleva el jugador en ese momento. Si el inventario está vacío, se muestra un mensaje indicándolo. Esta función permite al jugador conocer en todo momento los objetos que ha recogido.



The screenshot shows a web-based code editor interface. At the top, there's a toolbar with various icons. Below that is a header bar with the URL "www3.gobiernodecanarias.org" and a link to "Página principal". On the right side of the header is a user profile for "Eudaldo Álvaro Cal Saúl". The main area is a code editor with two tabs: "main.py" and "acciones.py". The "acciones.py" tab is active and contains the following Python code:

```
84 def inventario() -> str:
85     """
86         Genera una representación textual de los objetos que tiene el jugador.
87
88         Aplica lógica condicional para manejar el caso de inventario vacío o
89         formatear la lista de items mediante concatenación de cadenas.
90
91     :return: Cadena con la lista de objetos o aviso de inventario vacío.
92     :rtype: str
93     """
94
95     if estado.inventario:
96         return "Inventario: " + ", ".join(estado.inventario)
97     else:
98         return "Inventario: (vacío)"
```

La función inspeccionar():

Muestra información sobre un objeto del juego. Si el objeto está en el inventario o es visible en la sala, se devuelve su descripción. En el caso del pergamo, solo puede leerse cuando está en el inventario y proporciona una pista para ayudar al jugador a encontrar la salida.

```
100 def inspeccionar(obj: str) -> str:
101     """
102     Devuelve la descripción detallada de un objeto si es accesible.
103
104     Comprueba si el objeto está en el inventario o en la sala actual. Incluye
105     lógica especial para el objeto 'pergamino', que requiere estar en posesión
106     del jugador para ser leído.
107
108     :param obj: El nombre del objeto a examinar.
109     :type obj: str
110     :return: La descripción del objeto o un mensaje de error si no es visible.
111     :rtype: str
112
113     .. note::
114         Si el objeto es 'pergamino', se requiere una validación de seguridad
115         adicional sobre el inventario.
116
117     # Comprueba si el objeto es el pergamino.
118     # El pergamino solo se puede leer si está en el inventario.
119     if obj == "pergamino":
120         if obj not in estado.inventario:
121             return "El pergamino está ahí, pero necesitas cogerlo para poder leerlo."
122         return (
123             "Desenrollas el pergamino y lees:\n\n"
124             "\\"Alí donde el agua cae en la oscuridad,\n"
125             "el camino hacia la libertad apunta al norte.\n"
126             "Pero ninguna puerta cede ante manos vacías.'"
127         )
128
129     # Comprueba si el objeto está en tu inventario o visible en la sala actual.
130     # Si está devuelve descripción, si no frase genérica.
131     # Si el objeto no está ni en inventario ni en sala, devuelve "No ves"
132     if obj in estado.inventario:
133         return OBJ_DESCS.get(obj, f"No ves nada especial en '{obj}'")
134
135     if obj in mundo.objetos_visibles(estado.ubicacion):
136         return OBJ_DESCS.get(obj, f"No ves nada especial en '{obj}'")
137
138     return f"No ves '{obj}'."
```

MUNDO.PY

Este archivo contiene los datos del laberinto y no incluye lógica del juego. Su función es almacenar la información necesaria para que otros módulos puedan consultarla, como las habitaciones, las salidas y los objetos disponibles en cada sala.

En este módulo se definen las distintas salas del laberinto, cada una con su nombre y descripción, así como las conexiones entre ellas mediante salidas en las direcciones norte, sur, este y oeste. También se indican los objetos que hay inicialmente en cada habitación y un mapa simple que representa la distribución del laberinto.

Actúa como una base de datos del juego, separando los datos de la lógica y facilitando la organización del programa.

Además, el archivo incluye varias funciones de consulta que permiten obtener información sin modificar los datos originales.

Estructuras de datos:

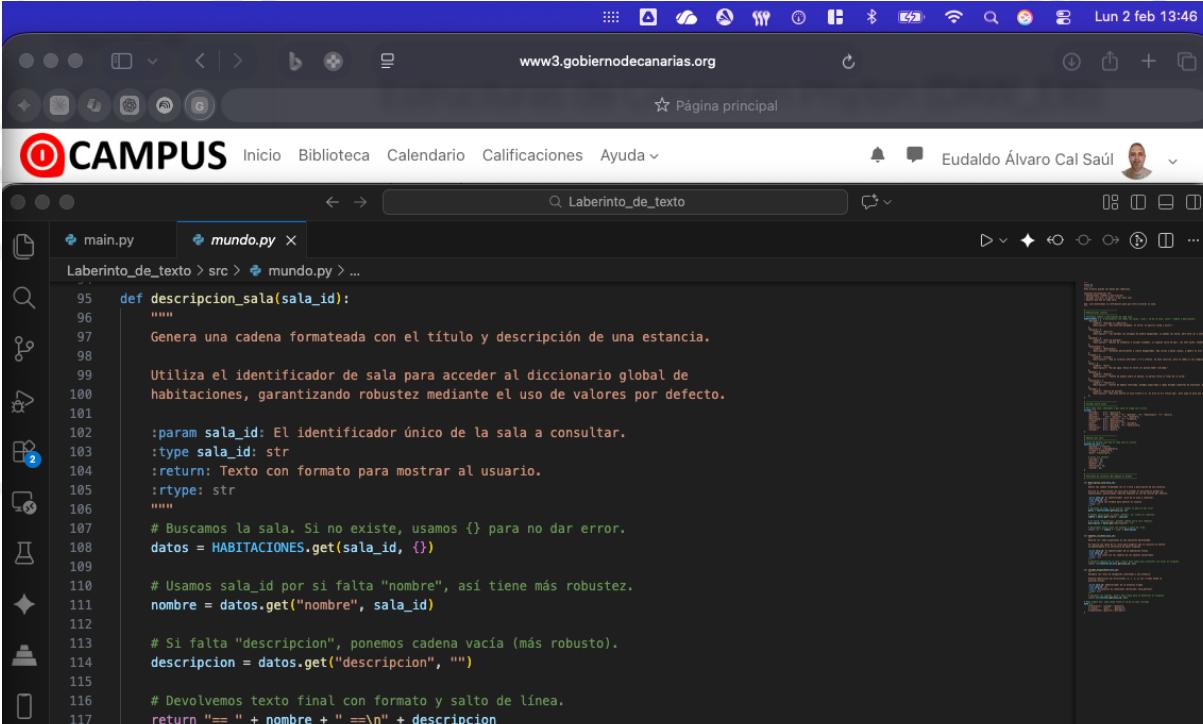
- **HABITACIONES**: Diccionario que mapea IDs de sala a sus metadatos (nombre y descripción).
- **SALIDAS**: Diccionario anidado que define las conexiones entre salas (n/s/e/o).
- **OBJETOS_EN_SALA**: Diccionario que lista los objetos iniciales en cada sala.
- **MAPA**: Matriz 3x3 que representa visualmente la disposición de las salas.

Funciones de consulta (no mutan el estado):

- **descripcion_sala(sala_id)**: Genera texto formateado con título y descripción de una sala.
- **objetos_visibles(sala_id)**: Retorna una copia de la lista de objetos presentes.
- **salidas_disponibles(sala_id)**: Devuelve una copia del diccionario de conexiones disponibles.

La función `descripcion_sala()`:

Devuelve el nombre y la descripción de una sala.



```
def descripcion_sala(sala_id):
    """
    Genera una cadena formateada con el título y descripción de una estancia.

    Utiliza el identificador de sala para acceder al diccionario global de
    habitaciones, garantizando robustez mediante el uso de valores por defecto.

    :param sala_id: El identificador único de la sala a consultar.
    :type sala_id: str
    :return: Texto con formato para mostrar al usuario.
    :rtype: str
    """

    # Buscamos la sala. Si no existe, usamos {} para no dar error.
    datos = HABITACIONES.get(sala_id, {})

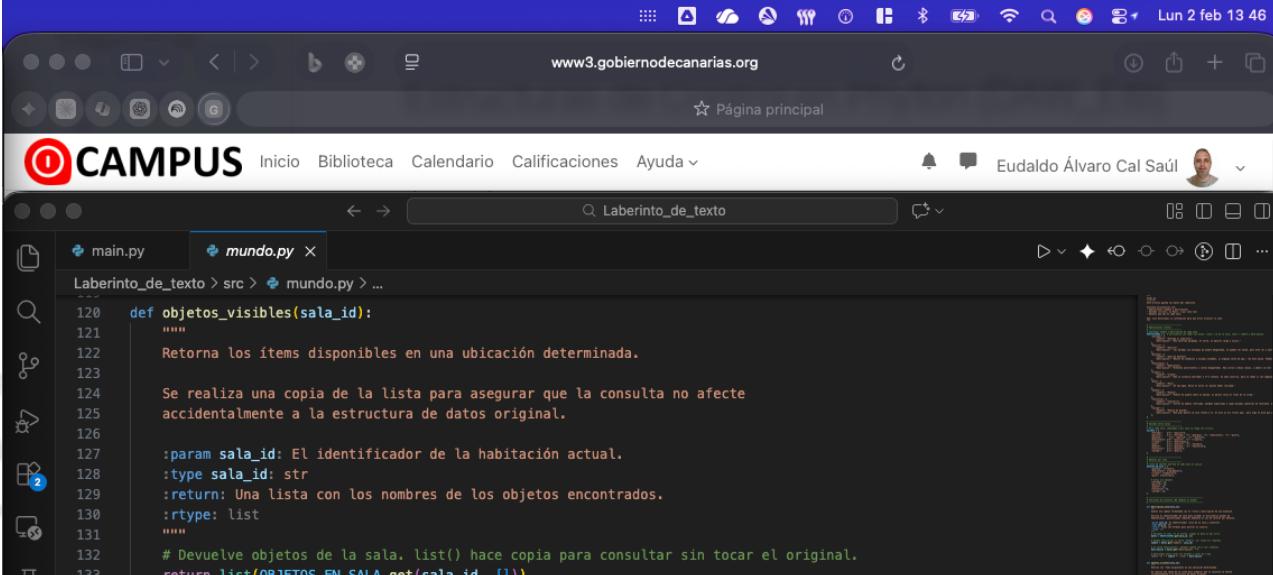
    # Usamos sala_id por si falta "nombre", así tiene más robustez.
    nombre = datos.get("nombre", sala_id)

    # Si falta "descripcion", ponemos cadena vacía (más robusto).
    descripcion = datos.get("descripcion", "")

    # Devolvemos texto final con formato y salto de línea.
    return "==" + nombre + " ==\n" + descripcion
```

La función `objetos_visibles()`:

Devuelve los objetos que hay en una habitación.



```
def objetos_visibles(sala_id):
    """
    Retorna los ítems disponibles en una ubicación determinada.

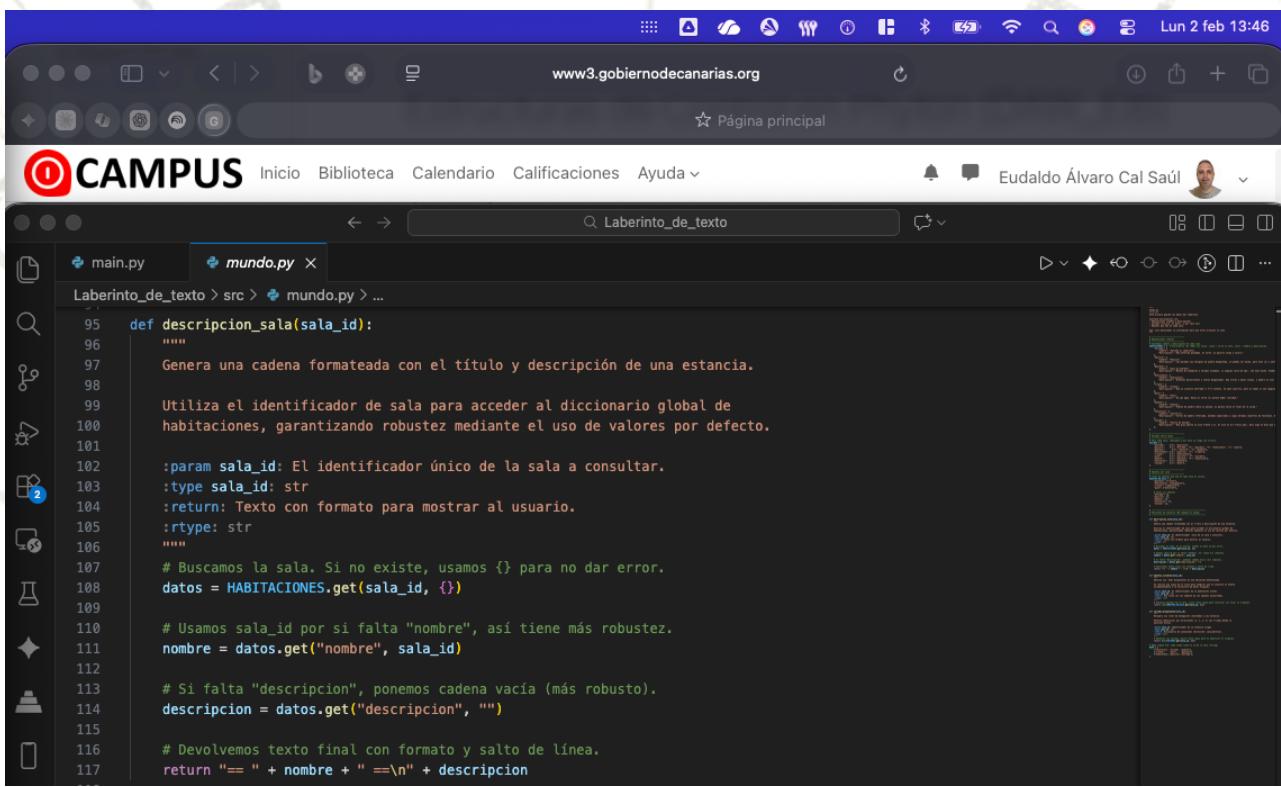
    Se realiza una copia de la lista para asegurar que la consulta no afecte
    accidentalmente a la estructura de datos original.

    :param sala_id: El identificador de la habitación actual.
    :type sala_id: str
    :return: Una lista con los nombres de los objetos encontrados.
    :rtype: list
    """

    # Devuelve objetos de la sala. list() hace copia para consultar sin tocar el original.
    return list(OBJETOS_EN_SALA.get(sala_id, []))
```

La función `salidas_disponibles()`:

Devuelve las salidas posibles desde una sala.



The screenshot shows a web browser window with the URL www3.gobiernodecanarias.org. The page title is "CAMPUS". The main content area is a code editor showing Python code for a labyrinth game. The file is named `mundo.py`. The code defines a function `descripcion_sala(sala_id)` which generates a formatted string with the title and description of a room. It uses a global dictionary `HABITACIONES` to get room data. If the room ID is missing, it adds "nombre". If the description is missing, it adds an empty string. Finally, it returns a string with the room name and description separated by a double newline. The code editor interface includes tabs for `main.py` and `mundo.py`, a search bar, and various icons for file operations.

```
def descripcion_sala(sala_id):
    """
    Genera una cadena formateada con el título y descripción de una estancia.

    Utiliza el identificador de sala para acceder al diccionario global de
    habitaciones, garantizando robustez mediante el uso de valores por defecto.

    :param sala_id: El identificador único de la sala a consultar.
    :type sala_id: str
    :return: Texto con formato para mostrar al usuario.
    :rtype: str
    """
    # Buscamos la sala. Si no existe, usamos {} para no dar error.
    datos = HABITACIONES.get(sala_id, {})

    # Usamos sala_id por si falta "nombre", así tiene más robustez.
    nombre = datos.get("nombre", sala_id)

    # Si falta "descripcion", ponemos cadena vacía (más robusto).
    descripcion = datos.get("descripcion", "")

    # Devolvemos texto final con formato y salto de línea.
    return "==" + nombre + " ==\n" + descripcion
```

ESTADO.PY

Este archivo se encarga de guardar el estado del juego, es decir, la información que cambia durante la partida. En él se almacenan datos como la ubicación actual del jugador, el inventario, las salas que ya han sido visitadas y si el jugador ha conseguido ganar la partida. Estos datos son utilizados y modificados por otros módulos a lo largo del juego.

Centraliza el estado de la sesión de juego mediante variables globales, actuando como la 'memoria' del sistema.

Variables globales:

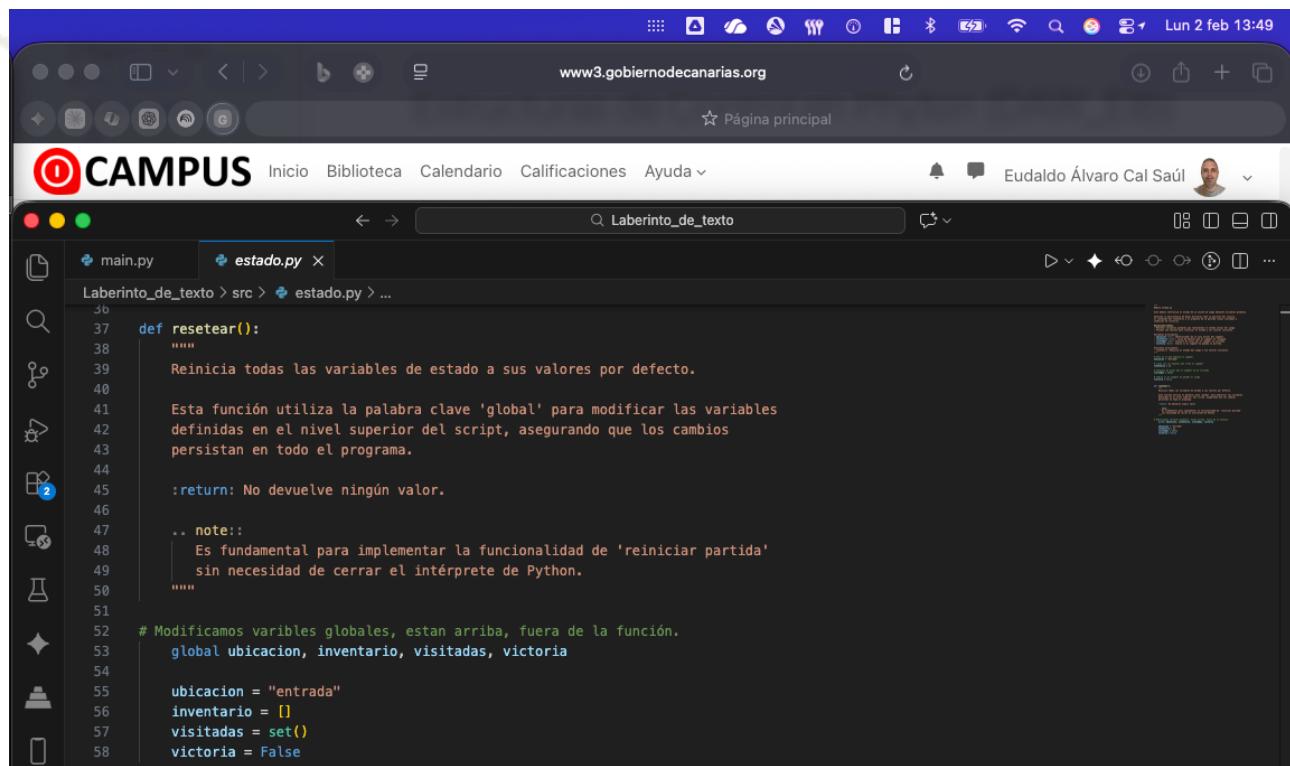
- **ubicacion**: String que identifica la sala actual (inicialmente 'entrada').
- **inventario**: Lista de objetos que el jugador ha recogido (inicialmente vacía).
- **visitadas**: Conjunto (set) de salas que el jugador ha explorado.
- **victoria**: Booleano que indica si el jugador ha completado el juego.

Función clave:

- **resetear():** Reinicia todas las variables a sus valores iniciales, permitiendo al jugador comenzar una nueva partida sin reiniciar el programa. Utiliza la palabra clave 'global' para modificar las variables del módulo.

La función resetear():

Permite volver el juego a su estado inicial. Esta función se utiliza cuando el jugador decide reiniciar la partida, restableciendo la ubicación inicial, vaciando el inventario y borrando las salas visitadas.



The screenshot shows a Mac OS X desktop environment. At the top, there's a system tray with icons for battery, signal, and volume. Below it is a browser window with the URL www3.gobiernodecanarias.org and a user profile for 'Eudaldo Álvaro Cal Saúl'. The main window is a terminal or code editor titled 'Laberinto_de_texto'. It displays two files: 'main.py' and 'estado.py'. The 'estado.py' file contains the following Python code:

```
36
37 def resetear():
38     """
39     Reinicia todas las variables de estado a sus valores por defecto.
40
41     Esta función utiliza la palabra clave 'global' para modificar las variables
42     definidas en el nivel superior del script, asegurando que los cambios
43     persistan en todo el programa.
44
45     :return: No devuelve ningún valor.
46
47     .. note::
48         Es fundamental para implementar la funcionalidad de 'reiniciar partida'
49         sin necesidad de cerrar el intérprete de Python.
50     """
51
52     # Modificamos variables globales, están arriba, fuera de la función.
53     global ubicacion, inventario, visitadas, victoria
54
55     ubicacion = "entrada"
56     inventario = []
57     visitadas = set()
58     victoria = False
```

MAPA DEL LABERINTO DE TEXTO



COMANDOS DISPONIBLES

El juego se controla mediante comandos de texto que el jugador introduce por consola. A continuación se muestran los comandos disponibles y su función:

n / s / e / o: Permiten moverse al norte, sur, este u oeste, siempre que exista una salida en esa dirección.

ir <dirección>: Permite moverse indicando la dirección (por ejemplo, ir n).

mirar: Muestra la descripción de la sala actual, los objetos visibles y las salidas disponibles.

coger <objeto>: Recoge un objeto que se encuentra en la sala actual y lo añade al inventario.

soltar <objeto>: Deja un objeto del inventario en la sala actual.

inventario: Muestra los objetos que lleva el jugador en ese momento.

inspeccionar <objeto>: Muestra información sobre un objeto. En el caso del pergamo, permite leer una pista.

mapa: Muestra un mapa simple del laberinto con las salas visitadas.

reiniciar: Reinicia la partida y devuelve al jugador a la sala inicial.

ayuda: Muestra la lista de comandos disponibles.

salir: Finaliza el juego.

DECISIONES DE DISEÑO Y JUSTIFICACIÓN

Modularización y Separación de Responsabilidades

Decisión: Dividir el código en seis módulos independientes con responsabilidades claramente definidas.

Justificación:

- **Mantenibilidad:** Cada módulo puede ser modificado sin afectar a los demás. Por ejemplo, cambiar el mapa (mundo.py) no requiere tocar la lógica de movimiento.
- **Reusabilidad:** Las funciones de consulta en mundo.py y acciones.py pueden ser utilizadas por cualquier otro módulo sin duplicar código.
- **Testabilidad:** Cada módulo puede ser probado de forma aislada.
- **Escalabilidad:** Facilita la adición de nuevas funcionalidades (por ejemplo, un módulo de combate o puzzles).

Uso de Variables Globales (estado.py)

Decisión: Centralizar el estado mutable del juego en un módulo dedicado con variables globales.

Justificación:

- **Simplicidad:** Evita pasar múltiples parámetros entre funciones. Todos los módulos pueden acceder al estado actual importando 'estado'.
- **Coherencia:** Un único punto de verdad para el estado del juego, reduciendo inconsistencias.
- **Función resetear():** Permite reiniciar el juego sin necesidad de reiniciar el intérprete de Python, mejorando la experiencia del usuario.

Funciones de Consulta con Copias Defensivas

Decisión: Las funciones de mundo.py (objetos_visibles, salidas_disponibles) devuelven copias de las estructuras de datos, no referencias directas.

Justificación:

- **Prevención de bugs:** Evita que modificaciones accidentales en el código de consulta afecten a la base de datos del mundo.

- **Encapsulación:** Los datos del mundo son de solo lectura para los módulos que los consultan.
- **Claridad:** Distingue claramente entre consultas (que no mutan) y mutaciones (que ocurren solo en acciones.py al modificar OBJETOS_EN_SALA).

Estrategia de Manejo de Excepciones

Decisión: Implementar tres niveles de manejo de excepciones:

- **Nivel 1 (main.py):** Captura excepciones globales (KeyboardInterrupt, Exception) para evitar trazas de error al usuario.
- **Nivel 2 (juego.py):** Captura excepciones específicas del bucle principal (NotImplementedError, ValueError, Exception) con mensajes informativos.
- **Nivel 3 (movimiento.py):** Lanza ValueError con mensajes descriptivos para entradas inválidas, delegando el manejo al nivel superior.

Justificación:

- **Experiencia de usuario:** Mensajes de error amigables en lugar de trazas de Python técnicas.
- **Robustez:** El juego nunca se cierra inesperadamente, siempre hay una salida controlada.
- **Depuración:** Mensajes específicos ayudan a identificar el tipo de error (entrada inválida vs funcionalidad no implementada vs error inesperado).
- **Separación de responsabilidades:** movimiento.py valida y lanza excepciones, juego.py las captura y formatea.

Validación Exhaustiva de Entrada del Usuario

Decisión: Implementar múltiples capas de validación en interpretar() y mover():

- Comandos vacíos son ignorados silenciosamente
- Comandos sin argumentos requeridos lanzan ValueError
- Direcciones son normalizadas (acepta 'n', 'N', 'norte', 'Norte')
- Comandos desconocidos generan sugerencia de usar 'ayuda'

Justificación:

- **Usabilidad:** El juego acepta variaciones naturales del lenguaje ('norte' vs 'n').

- **Prevención de errores:** Validación temprana evita que lleguen datos incorrectos a las funciones de lógica del juego.
- **Mensajes informativos:** Cada tipo de error de entrada genera un mensaje específico que ayuda al usuario a corregir su comando.

Lógica Especial para el Pergamino

Decisión: El objeto 'pergamino' solo puede ser inspeccionado si está en el inventario del jugador.

Justificación:

- **Mecánica de juego:** Añade profundidad al gameplay, requiriendo que el jugador interactúe con el objeto.
- **Pista narrativa:** El contenido del pergamino proporciona una pista crucial para completar el juego.
- **Diferenciación:** Demuestra que los objetos pueden tener comportamientos únicos, no solo descripciones diferentes.

Condición de Victoria Automática

Decisión: El juego detecta automáticamente la victoria cuando el jugador alcanza la sala 'salida' con la 'llave' en el inventario.

Justificación:

- **Fluidez:** El jugador no necesita escribir un comando adicional como 'usar llave'.
- **Claridad:** La condición de victoria es transparente y fácil de entender.
- **Satisfacción:** El mensaje de victoria aparece inmediatamente tras cumplir las condiciones, proporcionando feedback instantáneo.

Sistema de Mapa ASCII

Decisión: Implementar una matriz 3x3 en mundo.py y una función mapa() que genera representación ASCII del progreso.

Justificación:

- **Orientación:** Ayuda al jugador a visualizar la estructura del laberinto sin necesidad de dibujar un mapa manual.

- **Progreso visible:** Marca las salas visitadas, proporcionando sensación de avance.
- **Simplicidad:** Usa solo caracteres ASCII, compatible con cualquier terminal.

USO DE FUNCIONES BUILT-IN Y DEFINIDAS POR EL USUARIO

Funciones Built-in de Python Utilizadas

El proyecto hace uso extensivo de funciones integradas de Python:

Función	Uso en el Proyecto	Módulo
<code>input()</code>	Captura comandos del usuario	juego.py
<code>print()</code>	Muestra mensajes al usuario	juego.py, main.py
<code>str.strip()</code>	Limpia espacios en comandos	juego.py
<code>str.split()</code>	Separa comando y argumentos	juego.py
<code>str.lower()</code>	Normaliza comandos a minúsculas	juego.py, movimiento.py
<code>str.join()</code>	Formatea listas para mostrar	movimiento.py, acciones.py
<code>dict.get()</code>	Consulta segura de diccionarios	mando.py, acciones.py
<code>list()</code>	Crea copias defensivas	mando.py
<code>dict()</code>	Crea copias de diccionarios	mando.py

<code>set.add()</code>	Añade salas visitadas	movimiento.py, juego.py
<code>list.append()</code>	Añade objetos al inventario	acciones.py
<code>list.remove()</code>	Elimina objetos de listas	acciones.py

Funciones Definidas por el Usuario

El proyecto define **13 funciones principales** distribuidas estratégicamente entre los módulos:

Función	Propósito	Módulo
<code>main()</code>	Punto de entrada principal	main.py
<code>iniciar()</code>	Bucle principal del juego	juego.py
<code>interpretar(linea)</code>	Parser de comandos	juego.py
<code>resetear()</code>	Reinicia el estado del juego	estado.py
<code>mirar()</code>	Describe la sala actual	movimiento.py
<code>mover(direccion)</code>	Desplaza al jugador	movimiento.py
<code>mapa()</code>	Genera representación ASCII	movimiento.py
<code>coger(obj)</code>	Recoge un objeto	acciones.py
<code>soltar(obj)</code>	Deposita un objeto	acciones.py
<code>inventario()</code>	Muestra objetos del jugador	acciones.py

<code>inspeccionar(obj)</code>	Examina objetos detalladamente	acciones.py
<code>descripcion_sala()</code>	Formatea descripción de sala	mundo.py
<code>objetos_visibles()</code>	Retorna objetos en sala	mundo.py
<code>salidas_disponibles()</code>	Retorna conexiones de sala	mundo.py

DOCUMENTACIÓN DEL CÓDIGO CON SPHINX

Como parte de las buenas prácticas de desarrollo de software y para cumplir con los requisitos de documentación del proyecto, hemos generado **documentación técnica automatizada** utilizando **Sphinx**, una herramienta profesional ampliamente utilizada en proyectos Python.

Ubicación de la Documentación

La documentación completa del proyecto se encuentra disponible en formato HTML en la siguiente ruta:

`docs/build/html/index.html`

Esta documentación puede visualizarse abriendo el archivo `index.html` en cualquier navegador web moderno, proporcionando una interfaz interactiva y navegable para explorar toda la estructura del código.

Características de la Documentación Generada

La documentación con Sphinx ofrece las siguientes ventajas:

1. Generación Automática desde Docstrings

- Sphinx extrae automáticamente toda la información de los docstrings en formato reStructuredText (reST) presentes en cada módulo y función.
- Esto garantiza que la documentación está siempre sincronizada con el código fuente.

2. Estructura Organizada

- Página principal con introducción al proyecto
- Índice de módulos con enlaces directos a cada archivo
- Documentación de funciones con parámetros, tipos y valores de retorno
- Referencias cruzadas entre módulos relacionados

3. Navegación Intuitiva

- Menú lateral con jerarquía de módulos
- Barra de búsqueda para localizar funciones específicas
- Enlaces internos entre documentación relacionada

4. Formato Profesional

- Sintaxis resaltada para código
- Tablas de parámetros y tipos
- Secciones claramente diferenciadas
- Notas y advertencias destacadas

Contenido Documentado

La documentación generada incluye información completa sobre:

- **main.py**: Punto de entrada y manejo de excepciones globales
- **juego.py**: Bucle principal, parser de comandos y lógica de victoria
- **estado.py**: Variables globales y función de reinicio
- **mundo.py**: Estructuras de datos del laberinto y funciones de consulta
- **movimiento.py**: Lógica de navegación, exploración y mapa
- **acciones.py**: Gestión de inventario e interacción con objetos

Cada función documentada incluye:

- Descripción detallada del propósito
- Lista de parámetros con tipos y explicaciones
- Valor de retorno con tipo
- Excepciones que puede lanzar
- Notas adicionales sobre comportamiento especial

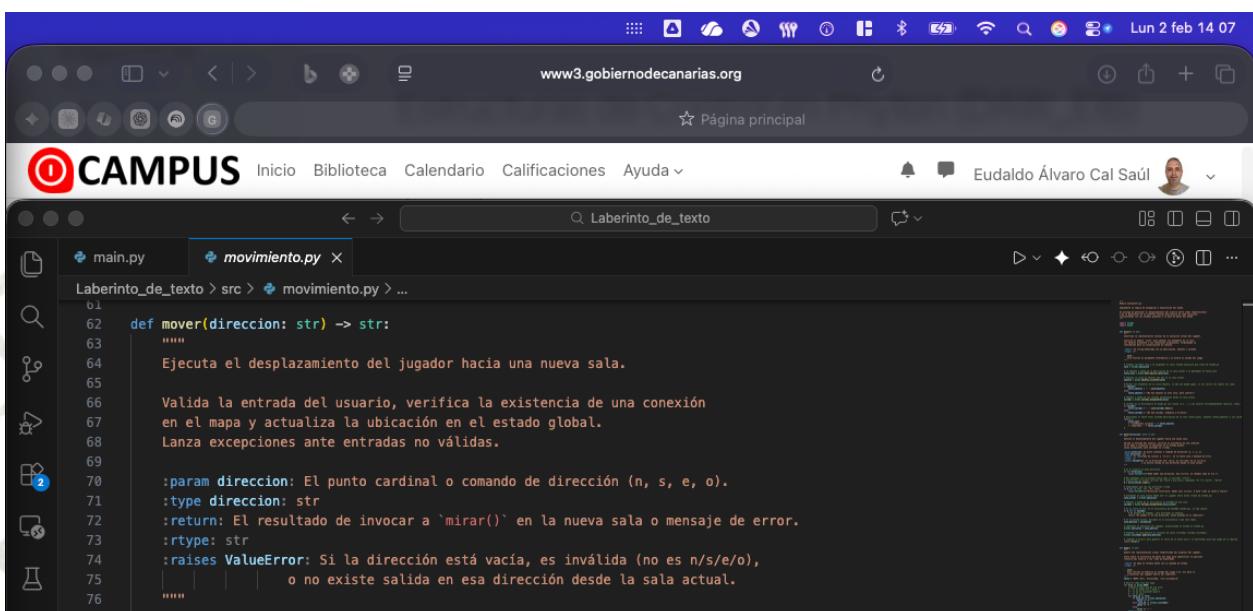
Decisión de Diseño: Docstrings en formato reStructuredText

Decisión: Utilizar docstrings en formato reStructuredText (reST) en todas las funciones del proyecto.

Justificación:

- **Compatibilidad con Sphinx:** reST es el formato nativo de Sphinx, permitiendo generar documentación HTML automáticamente.
- **Estándar profesional:** Es el formato más utilizado en proyectos Python de nivel empresarial.
- **Información estructurada:** Permite especificar tipos de parámetros, valores de retorno y excepciones de forma clara.
- **Legibilidad en código:** Los docstrings son legibles directamente en el código fuente, sin necesidad de herramientas adicionales.
- **Mantenibilidad:** Centraliza la documentación con el código, reduciendo inconsistencias.

Ejemplo de Docstring Utilizado



The screenshot shows a web browser window with the URL www3.gobiernodecanarias.org. The page title is "CAMPUS". The main content area displays a code editor for a Python file named "movimiento.py". The code contains a function definition for "mover" and its docstring:

```
def mover(direccion: str) -> str:
    """
    Ejecuta el desplazamiento del jugador hacia una nueva sala.

    Valida la entrada del usuario, verifica la existencia de una conexión
    en el mapa y actualiza la ubicación en el estado global.

    Lanza excepciones ante entradas no válidas.

    :param direccion: El punto cardinal o comando de dirección (n, s, e, o).
    :type direccion: str
    :return: El resultado de invocar a `mirar()` en la nueva sala o mensaje de error.
    :rtype: str
    :raises ValueError: Si la dirección está vacía, es inválida (no es n/s/e/o),
                        o no existe salida en esa dirección desde la sala actual.
    """

```

Ventajas para el Desarrollo y Mantenimiento

- 1. Onboarding de nuevos desarrolladores:** Un nuevo miembro del equipo puede comprender rápidamente la estructura del proyecto navegando la documentación HTML.
- 2. Referencia rápida:** Durante el desarrollo, cualquier desarrollador puede consultar la documentación sin necesidad de leer todo el código fuente.
- 3. Validación de diseño:** El proceso de documentar cada función obliga a reflexionar sobre su propósito, parámetros y comportamiento, mejorando la calidad del código.
- 4. Profesionalismo:** Contar con documentación técnica completa demuestra madurez en el proceso de desarrollo de software.

CONCLUSIÓN

Con este proyecto hemos aprendido a organizarnos mejor en grupo utilizando Git y a trabajar con estructuras como listas y diccionarios para representar las salas, salidas y objetos. También hemos mejorado nuestra forma de repartir el trabajo gracias a la división del programa en varios archivos.

Una de las partes más difíciles fue crear el mapa del laberinto usando bucles for anidados, ya que requería controlar bien el estado de cada sala. Aunque solo hemos desarrollado el Reto 1, añadimos algunos objetos interactivos para hacer el juego más completo.

Además, este proyecto nos ha ayudado a entender cómo se estructura un juego desde cero y a aplicar correctamente elementos como funciones, condicionales if-else, bucles, manejo de errores con try-except, variables, listas, conjuntos y diccionarios, reforzando así nuestros conocimientos de programación en Python

Logros clave del proyecto:

- **Modularización efectiva:** Seis módulos con responsabilidades claramente definidas.
- **Uso extensivo de funciones built-in:** 12 funciones integradas aplicadas estratégicamente.
- **Funciones propias bien diseñadas:** 13 funciones que implementan toda la lógica del juego.
- **Manejo robusto de excepciones:** Tres niveles de captura con mensajes informativos.
- **Código mantenible:** Documentación exhaustiva con docstrings en formato reStructuredText.
- **Experiencia de usuario cuidada:** Validación de entrada, mensajes claros y sistema de ayuda integrado.

Este diseño modular y las decisiones de arquitectura tomadas permiten que el proyecto sea fácilmente extensible. Nuevas funcionalidades como un sistema de combate, puzzles complejos, o múltiples niveles podrían añadirse sin necesidad de reestructurar el código existente.

BIBLIOGRAFÍA

Documentación oficial y recursos técnicos

- Python Software Foundation. *Python Documentation* (Version 3.10–3.12). Disponible en: <https://docs.python.org/3/>
- Python Software Foundation. *Built-in Functions*. Disponible en: <https://docs.python.org/3/library/functions.html>
- Python Software Foundation. *The Python Tutorial*. Disponible en: <https://docs.python.org/3/tutorial/>
- Sphinx Documentation Team. *Sphinx Documentation*. Disponible en: <https://www.sphinx-doc.org/>

Fuentes utilizadas en la memoria del proyecto

- Apuntes de clase. *Estructuras de control en Python*. Ciclo superior de Desarrollo de Aplicaciones Web.