

Manual técnico

SOFTWARE DE UN JUEGO BÁSICO DE SOLITARIO PARA LA ENTREGA DEL PROYECTO #1 DEL CURSO DE ESTRUCTURAS DE DATOS

Naydelin Jirón, Julieth K. Soto, Paula Vargas, y Mariángel Vega

Escuela de Informática, Universidad Nacional de Costa Rica, Sede
Interuniversitaria de Alajuela

EIF207: Estructura de datos

MT. Cristopher Montero Jiménez

17 de septiembre de 2024

Contenido

INTRODUCCIÓN	4
HERRAMIENTAS UTILIZADAS PARA EL DESARROLLO	5
EXPLICACIÓN DE LAS CLASES	6
CLASE COMMON.....	6
Método cargar imágenes:.....	10
CLASE CARTAS	11
Constructor:	12
Método settopleft:	13
Método arrastrar:	14
Método setcenter:.....	14
Método mostrar:.....	15
Método ocultar:	16
Método switch.....	16
CLASE PILASUBIR.....	17
Constructor:	17
Método settopleft:	18
CLASE MAZO	19
Constructor:	19
Método settopleft:	20
Método crear mazo:.....	20
Método revolver:	21

Método debe:.....	21
CLASE JUEGO	21
Constructor:	22
Método on_init:	23
Método on_execute:	25
Método on_loop:	26
Método on_render:	27
Método on_cleanup:	30
Método on_event:	30
Método solicita usuario:	36
Método update_timer:	36
Método game_over:	37
Método victory:	37
Método show_victory_screen:	38
Método show_game_over_screen:	39
Método restart_game:	41
Métodos de registrar victoria y derrota:	42
Método guardar estadísticas:	43
Método cargar estadística:	44
Método subir:	45
Método deal:	46

Método check_pila_area:	47
Método matchable:	50
Main:	51

Para este proyecto se tomo como base y referencia el siguiente proyecto:

<https://github.com/martinsantibanez/solitario/tree/master>

INTRODUCCIÓN

Este manual tiene como objetivo guiar a cualquier persona con conocimientos básicos en programación a través del código de un juego de solitario desarrollado en Python, utilizando la biblioteca pygame. El propósito es proporcionar una comprensión clara y completa de las estructuras y funcionalidades utilizadas en el código, con el fin de que el lector pueda aprender cómo construir un juego desde cero y manipular las distintas partes que lo componen.

El juego de solitario es un proyecto de programación que combina varios conceptos clave de desarrollo de software, como la programación orientada a objetos (POO), la gestión de eventos, el control del flujo del programa mediante ciclos y condiciones, así como la manipulación de gráficos y animaciones para crear una interfaz visual interactiva.

HERRAMIENTAS UTILIZADAS PARA EL DESARROLLO

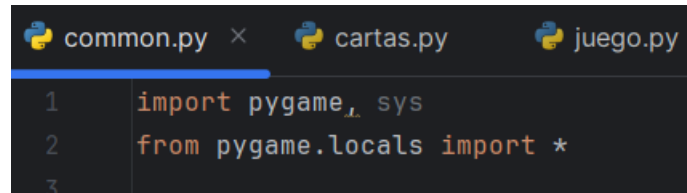
PYCHARM

Es un entorno de desarrollo integrado (IDE) diseñado específicamente para Python, creado por JetBrains. Este potente IDE proporciona un editor de código con autocompletado inteligente, herramientas avanzadas para depuración y pruebas, y soporte integrado para frameworks como Django y Flask. También facilita la gestión de entornos virtuales y la integración con sistemas de control de versiones, ofreciendo un entorno completo y eficiente para el desarrollo de aplicaciones web, proyectos de ciencia de datos y otros proyectos de software en Python. Su interfaz intuitiva, junto con funciones como el refactorizado automático de código y la compatibilidad con múltiples bases de datos, lo convierte en una herramienta ideal tanto para principiantes como para desarrolladores experimentados.

EXPLICACIÓN DE LAS CLASES

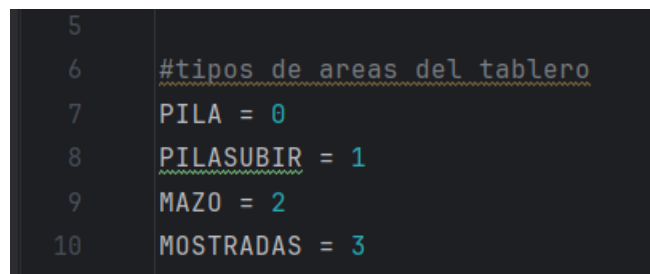
CLASE COMMON

En esta primera clase están los conjuntos de constantes y funciones relacionadas a la configuración y gestión de un juego tipo solitario utilizando la biblioteca pygame en Python.



```
common.py x cartas.py juego.py
1 import pygame, sys
2 from pygame.locals import *
3
```

Pygame es una biblioteca de Python diseñada para facilitar el desarrollo de videojuegos y aplicaciones multimedia. Proporciona herramientas para gestionar gráficos, sonido y entrada de usuario, permitiendo a los desarrolladores crear juegos y aplicaciones interactivas de manera más sencilla.



```
5
6 #tipos de areas del tablero
7 PILA = 0
8 PILASUBIR = 1
9 MAZO = 2
10 MOSTRADAS = 3
```

La imagen anterior define las constantes de las áreas que tendrá el tablero del juego. La constante llamada PILA representa las pilas de cartas que están en el tablero. La PILASUBIR representa las pilas en las cuales se podrán subir las cartas, el MAZO identifica el área donde se encontrará el mazo de cartas y MOSTRADAS representa el área donde se muestran las cartas visibles. Los números enteros de estas constantes se usan para identificar en las diferentes áreas del tablero y para facilitar su uso en las estructuras condicionales.

- **PILA:** representa el área del tablero donde se apilan las cartas. Se usa para identificar y gestionar esta área en el juego.
- **PILASUBIR:** representa las áreas donde las cartas pueden ser subidas y permite diferenciar entre las diferentes zonas de la interfaz del juego.
- **MAZO:** representa el área donde se encuentra el mazo de cartas. Es útil para gestionar la disposición del mazo en la interfaz.
- **MOSTRADA:** representa el área donde se muestran las cartas que se han sacado del mazo o que están disponibles para jugar. Facilita la visualización de las cartas en el juego.

Lo siguiente define las diferentes pintas de las cartas y una lista que las contiene a todas. Se usan las letras para abreviarlas y hacer el código más intuitivo.

```
11
12  #pintas
13  TREBOL = "T"
14  PICA = "P"
15  CORAZON = "C"
16  DIAMANTE = "D"
17  PINTAS = [TREBOL, PICA, CORAZON, DIAMANTE]
18
```

La siguiente imagen define las constantes para los colores que las cartas pueden tener: NEGRO para las de trébol y pica, ROJO para el corazón y el diamante. Se usa el 0 y el 1 para asociarlos a los palos de las cartas, y el uso de los números lo hace más eficiente y permite rápidas comparaciones.

```
18
19  #colores
20  NEGRO = 0
21  ROJO = 1
22
```

La siguiente imagen de código representa los dos estados en las que pueden estar las cartas:

- **ABAJO:** define el estado de la carta cuando está volteada hacia abajo, es decir, cuando debe estar oculta.
- **ARRIBA:** define el estado de la carta cuando está boca arriba, es decir, cuando está visible y utilizable para usar.

```
23  #estado
24  ABAJO = 0
25  ARRIBA = 1
26
```

Después sigue las constantes de las dimensiones de la ventana donde se mostrará el juego.

- **WIDTH:** representa el ancho que tendrá la ventana.
- **HEIGHT:** representa la altura que tendrá la ventana.

Esto se puede modificar a gusto del programador.

```

27     #ventana
28     WIDTH = 1000
29     HEIGHT = 800
30

```

Luego, determinamos el tamaño gráfico que queremos que tengan cada una de las cartas en la ventana del juego.

- **TAMX_CARTA:** representa el ancho que tendrá las cartas en pixeles.
- **TAMY_CARTA:** representa la altura que tendrán las cartas en pixeles.

```

31     # Escala para las cartas (ajusta según tus imágenes)
32     TAMX_CARTA = 100 # nuevo ancho de las cartas
33     TAMY_CARTA = 140 # nueva altura de las cartas
34

```

Definimos el espaciado y las posiciones iniciales de cada uno de los elementos en la ventana del juego.

```

35     # Ajustar las distancias entre cartas y posiciones iniciales
36
37     DISTX_PILAS = 102
38     DISTY_PILAS = 30
39     PILAS_XINICIAL = 200
40     PILAS_YINICIAL = 230
41     MAZO_XINICIAL = 50
42     MAZO_YINICIAL = 50
43     PILASUBIR_XINICIAL = MAZO_XINICIAL + DISTX_PILAS * 4.45
44     PILASUBIR_YINICIAL = MAZO_YINICIAL
45     MOSTRADA_POSX = MAZO_XINICIAL + DISTX_PILAS |
46     MOSTRADA_POSY = MAZO_YINICIAL
47

```

- **DISTX_PILAS:** representa la distancia horizontal entre las pilas de cartas que hay en el tablero, es decir, espaciado que hay para que no se superpongan las unas a las otras.
- **DISTY_PILAS:** representa la distancia vertical entre cartas en una pila, y controla el espaciado vertical para apilar las cartas adecuadamente.
- **PILAS_XINICIAL:** representa la posición horizontal inicial de las pilas en el tablero, es decir, donde comienzan las pilas en dirección horizontal
- **PILAS_YINICIAL:** representa la posición vertical inicial para las pilas y establece donde comienzan las pilas en la dirección vertical.

- **MAZO_XINICIAL**: representa la posición horizontal del mazo de cartas y define su ubicación en la pantalla.
- **MAZO_YINICIAL**: representa la posición vertical del mazo y también su ubicación en la pantalla.
- **PILASUBIR_XINICIAL**: establece la posición horizontal inicial de las pilas donde se pueden subir las cartas y se calcula en función de la posición del mazo y el espaciado horizontal entre pilas y la multiplicación por 4.45 ajusta el desplazamiento para que se alinee correctamente y haya una distancia adecuada.
- **PILASUBIR_YINICIAL**: establece la posición vertical inicial de las pilas donde se suben las cartas y se iguala a la posición vertical del mazo para que tanto el mazo y las pilas de subidas estén en la misma alineación vertical y evitar confusiones con el resto de las pilas.
- **MOSTRADA_POSX**: es la posición horizontal inicial para mostrar las cartas que han sido extraídas del mazo y define su ubicación horizontal para visualizarlas.
- **MOSTRADA_POSY**: es la posición vertical inicial para mostrar las cartas del mazo y define su ubicación vertical para visualizarlas.

Y debemos establecer la distancia que tendrán las cartas mostradas del mazo y la cantidad que van a mostrar cada vez que hagamos un clic en el mazo.

```
48  #distancia entre las que se muestran
49  DISTX_MOSTRADA = 10
50  #cartas a mostrar
51  CARTAS_MOSTRAR = 1
```

- **DISTX_MOSTRADA**: indica la distancia horizontal que hay entre las cartas que se muestran y ajusta el espaciado para que las cartas no se superpongan.
- **CARTAS_MOSTRAR**: representa el número de cartas que se van a mostrar cuando se extraiga del mazo.

También definimos un identificador de manejo de eventos del ratón en la ventana del juego con la siguiente constante.

```
52  #pygame
53  LEFT = 1
```

Esta constante LEFT se va a utilizar para identificar el botón izquierdo del ratón en eventos de pygame y su uso es para detectar clics en el área de juego. En pygame cuando se trata de eventos del ratón (MOUSEBUTTONDOWN o MOUSEBUTTONUP), el botón izquierdo del ratón se representa con el número 1 y se asigna a una constante para mejorar la legibilidad y mantenimiento del código.

Método cargar imágenes:

Ayuda cargar las imágenes de las cartas y el fondo en la ventana del juego. Esta función carga las imágenes desde un archivo utilizando pygame para gestionar errores en el proceso.

```
56 def load_image(filename, transparent=False):
57     try:
58         image = pygame.image.load(filename)
59     except pygame.error as message:
60         raise SystemExit (message)
61     image = image.convert()
62     if transparent:
63         color = image.get_at((0, 0))
64         image.set_colorkey(color, RLEACCEL)
65     return image
```

Este método recibe por parámetros el nombre del archivo de imagen a cargar (**filename**) y opcionalmente, el parámetro (**transparent**) que indica si se debe hacer transparente un color específico de la imagen, y en este caso no es necesario por lo que se pone "False".

El **bloque try** intenta ejecutar código que podría causar una excepción (error) si algo sale mal, y en este caso sería si no se puede cargar la imagen porque el archivo no existe o está corrupto. El programa capturará la excepción.

image = pygame.image.load(filename) es la responsable de cargar la imagen desde el archivo especificado por (filename).

Si ocurre algún error, el bloque **except pygame.error** capturará la excepción específica y el mensaje de error será almacenado en la variable **message**, y si se llegara a capturar, el programa termina y muestra el mensaje. **SystemExit** cerraría el programa y **message** mostraría que causó la excepción. Garantiza que, si no se pudo cargar la imagen, el programa no continúe en un estado incorrecto.

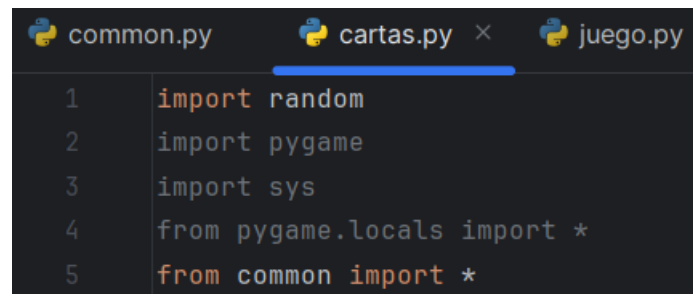
Si no entra en la excepción convierte la imagen en un formato más eficiente para ser procesado por pygame con el método **convert()** ajustando la imagen al formato de la pantalla actual, esto optimiza su rendimiento durante la ejecución y la velocidad de renderizado. El condicional **if transparent** verifica si el parámetro **transparent** fue pasado como True y si lo es, procede a hacer una parte de la imagen transparente. Si

fuera True, **color = image.get_at((0, 0))** obtendría el color del píxel ubicado en la esquina superior izquierda de la imagen por las coordenadas y sería utilizado como transparente. La función **set_colorkey(color, RLEACCEL)** establece un color transparente en todos los píxeles de la imagen que coincidan con el color en (0, 0), y RLEACCEL es una opción que permite optimizar la representación de transparencia usando RLE que mejora el rendimiento de la renderización. Por último, **return image** devuelve la imagen cargada.

CLASE CARTAS

Esta clase está diseñada para representar una carta de una baraja en el contexto de un juego. Cada carta tiene atributos visuales y funcionales como su valor, color, imagen y posición en el tablero y puede ser manipulada por varias operaciones como ser movida o volteada. En este archivo también se encuentran las clases Pila Subir y Mazo que serán explicadas más adelante.

Además, para que funcione correctamente esta clase de cartas y las demás que se encuentran, se necesitan algunas librerías como las siguientes

A screenshot of a code editor with three tabs: 'common.py', 'cartas.py' (which is active and highlighted with a blue underline), and 'juego.py'. The 'cartas.py' tab contains the following Python code:

```
1 import random
2 import pygame
3 import sys
4 from pygame.locals import *
5 from common import *
```

- **Random:** se utiliza para generar números aleatorios, y esto es esencial en el juego de cartas.
- **Pygame:** es una librería para la creación de videojuegos en Python. En esta clase se utiliza principalmente para cargar, manipular imágenes y gestionar las interacciones con las cartas en el entorno gráfico del juego.
- **Sys:** es parte de la biblioteca estándar de Python, y se utiliza principalmente para interactuar con el sistema operativo y proporcionar funciones para terminar el programa.
- **Pygame.locals:** contiene constantes que Pygame utiliza para eventos, teclas, y estados. En este caso es esencial para la interacción del jugador con el juego.
- **Common:** clase que contiene valores y constantes que son utilizados para proporcionar valores predefinidos para las dimensiones y posiciones de los elementos gráficos del juego.

Comenzamos directamente con la definición de la clase carta que es una subclase de `pygame.sprite.Sprite` que significa que hereda de la clase `Sprite` de `Pygame` que proporciona una estructura básica para todos los objetos gráficos. Al heredar de `Sprite`, la clase `Carta` puede ser gestionada por grupos de `Sprite` que facilitan la manipulación de varios objetos gráficos en conjunto.

```
8 class Carta(pygame.sprite.Sprite):
```

Constructor:

Seguimos con el constructor de la clase (`def __init__`). Este método inicializa la carta cuando se crea una nueva instancia de `Carta`.

```
8 class Carta(pygame.sprite.Sprite):
9     def __init__(self, numero, pinta, posX=-1, posY=-1):
10         self.pila = None
11         self.clicked = False
12         self.pinta = pinta
13         if pinta == TREBOL or pinta == PICA:
14             self.color = NEGRO
15         else:
16             self.color = ROJO
17         self.numero = numero
18         self.estado = ABAJO
19         # pygame
20         self.image = load_image("cards\\back.png")
21         self.image = pygame.transform.scale(self.image, size=(TAMX_CARTA, TAMY_CARTA))
22         self.settopleft(posx, posY)
```

Los atributos de esta clase son (`numero`, `pinta`, `posx`, `posy`), y se envían como parámetro al constructor para proporcionar información sobre el número y la pinta de la carta, también su posición en el tablero; como `posx` y `posy` tienen asignado el `-1` significa que la posición de la carta se determinará más adelante.

- **Self.pila** es un atributo que rastrea si la carta pertenece a alguna pila (grupo de cartas), y se inicializa como `None` porque al crear una carta aún no pertenece a ninguna pila.
- **Self.clicked** indica si la carta ha sido clicada o seleccionada, y se inicializa como `False` indicando que todavía no ha sido seleccionada.
- **Self.pinta** almacena la pinta de la carta, ya sea corazón, trébol, pica o diamante.
- **If pinta == TREBOL or pinta == PICA**, esta condicional determinará el color de la carta dependiendo de la pinta. Si es trébol o pica, el color de la carta será negro, y en caso contrario si la pica es corazón o diamante será color rojo.

- **Self.numero** almacena el número de la carta y sirve para identificar la jerarquía de la carta dentro de su pinta.
- **Self.estado** inicializa la carta boca abajo para que su cara no se vea, y se usa para alternar entre los estados de boca arriba y abajo.
- **Self.image = load_image()** asigna la imagen inicial de la carta llamando a la función `load_image` con una ruta a la carpeta `cards` a la imagen llamada `"back.png"`.
- **pygame.transform.scale()** redimensiona la imagen recibida por parámetro a un tamaño que está definido por los parámetros `TAMX_CARTA`, y `TAMY_CARTA` de la clase `common`.
- **Self.settopleft ()** es un método utilizado para establecer la posición inicial de la carta. Se llama al final para definir la posición de la carta una vez creada.

Método settopleft:

Este método se encarga de establecer la posición inicial de la carta en la pantalla por medio de coordenadas de la esquina superior izquierda del rectángulo que envuelve la carta.

```

23     def settopleft(self, x, y):
24         self.rect = self.image.get_rect()
25         self.posx = x
26         self.posy = y
27         self.posfx = self.posx
28         self.posfy = self.posy
29         self.rect.topleft = (self.posx, self.posy)

```

- **Self.rect = self.image.get_rect():** crea un rectángulo alrededor de la imagen de la carta. El rectángulo es un objeto en Pygame que se usa para manejar las posiciones, colisiones y áreas de los objetos. Al tener el rectángulo alrededor de la imagen, hace que sus dimensiones coincidan con las de la imagen de la carta.
- **Self.posx** representa la posición horizontal de la carta en la pantalla, y se le asigna un valor de `x` (coordenada horizontal).
- **Self.posy** similar al anterior, representa la posición vertical de la carta y se asigna un valor `y` (coordenada vertical)
- **Self.posfx** guarda una copia de la posición horizontal (`Self.posx`), ya que puede ser útil para restablecer la posición de la carta después de un movimiento o arrastre.

- **Self.posy** igual que lo anterior, guarda una copia de la posición vertical (Self.posy) para conservar la posición original o final.
- **Self.rect.topleft = (self.posx, self.posy)** la esquina superior izquierda del rectángulo y la imagen de la carta se ajusta a las coordenadas especificadas colocando la carta en la pantalla.

Método arrastrar:

Permite mover la carta por la pantalla cuando el jugador la arrastra con el ratón y ajusta la posición de la carta en función de las coordenadas del puntero del ratón.

```

1 usage (1 dynamic)
30     def arrastrar(self, x, y):
31         self.rect = self.image.get_rect()
32         self.posx = x - TAMX_CARTA/2
33         self.posy = y - TAMY_CARTA/2
34         self.rect.topleft = (self.posx, self.posy)
35

```

- **Self.rect = self.image.get_rect()** obtiene el rectángulo de la imagen de la carta, y vuelve a calcular para asegurar de cualquier actualización en la posición de la carta afecte correctamente al rectángulo asociado.
- **Self.posx = x - TAMX_CARTA / 2** esta ajusta la posición horizontal de la carta de acuerdo con el movimiento del ratón. La x es la posición actual del ratón, y se le resta la mitad del ancho de la carta para centrarla con la posición del cursor. Esto permite que el centro de la carta siga al cursor en lugar de la esquina superior izquierda.
- **Self.posy = y - TAMY_CARTA / 2** similar al anterior, se ajusta la posición de la carta en función del cursor y, restando la mitad de la altura de la carta para centrarla.
- **Self.rect.topleft = (self.posx, self.posy)** este actualiza la posición del rectángulo de la carta, estableciendo su esquina superior izquierda en las nuevas coordenadas enviadas por parámetro. Permite que la imagen se redibuje en la nueva posición mientras se arrastra.

Método setcenter:

Ajusta la posición de la carta en la pantalla, ubicando el centro de la carta en las coordenadas dadas como parámetro. A diferencia del método arrastrar este coloca la carta en una nueva posición fija sin que dependa del arrastre.

```

36     def setcenter(self, x, y):
37         self.rect = self.image.get_rect()
38         self.posx = x-TAMX_CARTA/2
39         self.posy = y-TAMY_CARTA/2
40         self.posfx = self.posx
41         self.posfy = self.posy
42         self.rect.topleft = (self.posx, self.posy)
43

```

- **Self.rect = self.image.get_rect():** obtiene el rectángulo de la imagen de la carta para asegurar que los cambios en la posición afecten bien la parte visual de la carta en la pantalla.
- **Self.posx = x - TAMX_CARTA / 2:** calcula la nueva posición horizontal de la carta, tomando la coordenada x y restándole la mitad del ancho de la carta para que quede centrada.
- **Self.posy = y - TAMY_CARTA / 2:** de forma similar al anterior, ajusta la posición vertical de la carta para centrarla en la coordenada y dada, y le resta la mitad de la altura.
- **Self.posfx = self.posx:** guarda la posición final en x de la carta posfx. Se puede usar para guardar o recuperar la posición futura.
- **Self.posfy = self.posy:** lo mismo que en el anterior, pero con la coordenada y.
- **Self.rect.topleft = (self.posx, self.posy):** esta actualiza la posición del rectángulo de la carta para que su esquina superior izquierda coincide con las coordenadas enviadas. Además, determina la posición en la que se dibuja la imagen de la carta en la pantalla.

Método mostrar:

Cambia el estado visual de una carta para mostrar su frente. Implica mostrar la cara en donde vienen su número y pinta, en lugar de la parte trasera.

```

44     def mostrar(self):
45         if self.estado == ABAJO:
46             self.estado = ARRIBA
47             self.image = load_image("cards\\" + self.pinta + str(self.numero + 1) + ".png")
48             self.image = pygame.transform.scale(self.image, size=(TAMX_CARTA, TAMY_CARTA))
49

```

- **if self.estado == ABAJO:** comprueba si la carta está boca abajo (no muestra su frente) y si lo está cambia su estado (self.estado = ARRIBA) a boca arriba, que quiere decir que muestre la carta que estaba oculta.

- **Self.image = load_image("cards\\" + self.pinta + str(self.numero + 1) + ".png"):** carga la imagen correspondiente a la carta y utiliza el atributo (self.pinta) y (self.numero) para hallarla en el archivo "cards".
- **Self.image = pygame.transform.scale(self.image, (TAMX_CARTA, TAMY_CARTA)):** después de cargar la imagen encontrada, la carta se redimensiona a las dimensiones predefinidas por los parámetros enviados para que encaje el tamaño en el juego. Asegura que tengan el mismo tamaño en la pantalla.

Método ocultar:

Cambia el estado visual de la carta para mostrar su parte trasera, ocultando la información del número y la pinta.

```

50     def ocultar(self):
51         if self.estado == ARRIBA:
52             self.estado = ABAJO
53             self.image = load_image("cards\\back.png")
54             self.image = pygame.transform.scale(self.image, size=(TAMX_CARTA, TAMY_CARTA))
55

```

- El **if self.estado == ARRIBA** verifica si la carta está mostrando su número y pinta, con el atributo **self.estado** de la carta.
- El **self.estado = ABAJO**: cambia el estado de la carta si esta se encuentra mostrando su información, para ocultarla. Es decir, a este atributo se le asigna un nuevo estado.
- El **self.image = load_image("cards\\back.png")**: cambia la imagen de la carta para mostrar la parte trasera. Al atributo self.image se le asigna el método para cargar la imagen primero con la dirección de ruta (cards) donde está la imagen "back.png"
- Por último, después de cargar la imagen con **self.image = pygame.transform.scale(self.image, (TAMX_CARTA, TAMY_CARTA))** se redimensiona la imagen enviada por parámetro para ajustar las dimensiones de la carta con los parámetros predefinidos. Esto hace que se mantenga un tamaño consistente en la pantalla.

Método switch

Este método hace que se alterne el estado de la carta entre "ARRIBA" o "ABAJO", es decir, si se muestra o no se muestra la información del número y la pinta.


```

56         def switch(self):
57             if(self.estado == ARRIBA):
58                 self.ocultar()
59             else:
60                 self.mostrar()
61

```

La condicional **if self.estado == ARRIBA** verifica el estado de la carta si se encuentra “ARRIBA”, y si lo está llama al método **ocultar()** para voltear la carta y no mostrar su información. Pero en caso contrario que la carta esté “ABAJO”, llama al método **mostrar()** para revelar la su información.

CLASE PILASUBIR

Esta clase representa una de las pilas donde se pueden organizar las cartas en orden ascendente (del AS al Rey) por cada palo. Y también maneja la visualización de la pila y su posición en la pantalla y las cartas que contiene.

La clase Pila Subir igual como la clase Carta, es una subclase de la clase base `pygame.sprite.Sprite`. Un “sprite” es cualquier elemento visual que puede moverse o interactuar con otros elementos del juego. Esta herencia permite manejar gráficos y colisiones.

```

62 class PilaSubir(pygame.sprite.Sprite):

```

Constructor:

El constructor de esta clase se define de la siguiente manera, y se ejecuta automáticamente cuando se crea una nueva instancia de Pila Subir:

```

62 class PilaSubir(pygame.sprite.Sprite):
63     def __init__(self, pinta, posx=-1, posy=-1):
64         self.pinta = pinta
65         self.image = load_image("cards\\" + pinta + ".png")
66         self.image = pygame.transform.scale(self.image, size=(TAMX_CARTA, TAMY_CARTA))
67         self.settopleft(posx, posy)
68         self.cartas = []

```

- El constructor (**def __init__**) recibe como parámetros la pinta, y las posiciones horizontales (posx) y vertical (posy) que tendrá. Estas posiciones se definirán más adelante.

- **Self.pinta = pinta:** crea un atributo llamado “pinta” que almacena el tipo de palo enviado por parámetro de las cartas que se colocarán en la pila.
- **Self.image = load_image("cards\\" + pinta + ".png"):** crea un atributo llamado “image” que almacena al método de cargar imágenes de la clase common para cargar la imagen que se asignará a esta pila. Este método buscará en el archivo “cards”, la letra inicial de la pinta que recibe por parámetro y el tipo de formato en la que está la imagen.
- **Self.image = pygame.transform.scale(self.image, (TAMX_CARTA, TAMY_CARTA)):** ajusta el tamaño de la carta con (pygame.transform.scale) y hace que coincida con las dimensiones de una carta representadas por las constantes enviadas por parámetro.
- **Self.settopleft(posx, posy):** este es un método para establecer la posición de la pila en la pantalla con los valores “posx” y “posy” que determinarán su ubicación.
- **Self.cartas = []:** inicializa una pila llamada cartas que contendrá todas las cartas que se coloquen en esta pila.

Método settopleft:

Este método definirá la posición de la pila en la pantalla del juego por medio de dos parámetros y que representa la posición horizontal (x) y vertical (y).

```

69         def settopleft(self, x, y):
70             self.rect = self.image.get_rect()
71             self.posx = x
72             self.posy = y
73             self.rect.topleft = (self.posx, self.posy)
74

```

- **Self.rect = self.image.get_rect():** crea primero un atributo llamado “rect” que es un objeto que representa el área rectangular de la imagen de la pila, y este objeto obtiene las dimensiones del rectángulo alrededor de la imagen con el método **get_rect()** de pygame.
- **Self.posx = x y posy = y:** son atributos que se crean para almacenar las coordenadas horizontales (x) y verticales (y) que se le asigna a la pila donde será dibujada en la pantalla.
- **Self.rect.topleft = (self.posx, self.posy):** es un atributo de pygame que define la posición de la esquina superior izquierda del rectángulo, al asignarle valores le estamos indicando en qué parte de la pantalla debe ubicarse el rectángulo, y por lo tanto también a la imagen del rectángulo.

CLASE MAZO

Esta clase define un mazo de cartas, que lo crea, lo baraja y lo posiciona visualmente en la pantalla.

La clase Mazo igual como la clase Carta y Pila Subir, es una subclase de la clase base `pygame.sprite.Sprite`, y hereda todas las funcionalidades de esta.

```
75 class Mazo(pygame.sprite.Sprite):
```

Constructor:

El constructor de esta clase se define de la siguiente manera, y no recibe nada por parámetros:

```
75 class Mazo(pygame.sprite.Sprite):
76     def __init__(self):
77         self.image = load_image("cards\\back.png")
78         self.image = pygame.transform.scale(self.image, size=(TAMX_CARTA, TAMY_CARTA))
79         self.settopleft(MAZO_XINICIAL, MAZO_YINICIAL)
80         self.cartas = []
81         self.crearmazo()
82         self.revolver()
```

- **Self.image = load_image("cards\\back.png")**: es un atributo que almacena el método de cargar imágenes, en este caso solo cargará la imagen de la parte trasera de las cartas para el mazo.
- **Self.image = pygame.transform.scale(self.image, (TAMX_CARTA, TAMY_CARTA))**: usa el atributo anterior para redimensionar la imagen del mazo a las dimensiones establecidas por el tamaño de las cartas.
- **self.settopleft(MAZO_XINICIAL, MAZO_YINICIAL)**: llama a este método que recibe por parámetro las posiciones que tendrá el mazo en la pantalla del juego. Se explica más adelante.
- **Self.cartas = []** : inicializa una lista vacía donde se almacenarán las cartas que tendrá el mazo.
- **Self.crearmazo()**: el constructor llama a este método para generar el conjunto de cartas que formarán el mazo.
- **Self.revolver()**: llama a este método para barajar las cartas y que sean aleatoriamente.

Método `setopleft`:

Este se encarga de posicionar el mazo en la pantalla

```
83     def setopleft(self, x, y):
84         self.rect = self.image.get_rect()
85         self.posx = x
86         self.posy = y
87         self.rect.topleft = (self.posx, self.posy)
88
```

- **`Self.rect = self.image.get_rect()`:** crea un rectángulo que rodea la imagen del mazo. Y obtiene las dimensiones del rectángulo alrededor de la imagen con el método `get_rect()`.
- **`Self.posx = x` y `self.posy = y`:** son atributos que se crean para almacenar las coordenadas horizontales (x) y verticales (y) que se le asigna al mazo donde será dibujado en la pantalla.
- **`Self.rect.topleft = (self.posx, self.posy)`:** es un atributo de pygame que define la posición de la esquina superior izquierda del rectángulo, al asignarle valores le estamos indicando en qué parte de la pantalla debe ubicarse el rectángulo.

Método `crear mazo`:

Este método es responsable de generar las 52 cartas que forman el mazo, es decir, las 13 cartas por cada palo.

```
89     def crearmazo(self):
90         for pinta in PINTAS:
91             for numero in range(13):
92                 new_carta = Carta(numero, pinta)
93                 self.cartas.append(new_carta)
```

- **`for pinta in PINTAS`:** itera sobre los diferentes palos de las cartas (trébol, pica, corazón, diamante) para crear un mazo completo de 52.
- **`for numero in range(13)`:** este itera sobre los valores numéricos de las cartas (del 1 al 13) y crea una carta para cada número dentro de un palo.
- **`new_carta = Carta(numero, pinta)`:** crea un objeto de la clase `Carta`, pasando como parámetros el número y el palo. Genera una carta por cada número y palo que hay.

- **Self.cartas.append(new_carta):** esta función llama a la lista vacía (cartas) que se inicializó en el constructor y con el “**append(new_carta)**” agrega a esa lista la carta que se creó en la línea de código anterior, y así hasta llenar el mazo.

Método revolver:

Este método mezcla el mazo de cartas utilizando la función “shuffle” de la librería random.

```
94     def revolver(self):
95         random.shuffle(self.cartas)
```

Random.shuffle es un método propio de la librería Random, y lo que hace es mezclar aleatoriamente el orden de la lista en donde están las cartas del mazo que recibió por parámetro.

Método debe:

Este método es solo para desarrollo y depuración. Permite imprimir la representación de cada carta en el mazo para verificar que se haya creado correctamente.

```
97     #dev only
98     def debug(self):
99         for c in self.cartas:
100             print(c.sprite)
```

El **for c in self.cartas** recorre cada carta que hay en la lista del mazo para examinar cada carta individualmente para verificar su estado o atributos durante el desarrollo o depuración. El **print(c.sprite)** imprime la representación de cada carta en la consola para que el desarrollador vea el estado o detalles de cada carta.

CLASE JUEGO

Esta última clase se encarga de gestionar el flujo y la lógica principal del juego, integrando la interfaz gráfica y la interacción del usuario.

Además, para que funcione correctamente esta clase Juego, se necesitan algunas librerías como las siguientes:

A screenshot of a code editor with three tabs: 'common.py', 'cartas.py', and 'juego.py'. The 'juego.py' tab is active and shows the following Python code:

```
1 import random
2 import pygame as pyg
3 import pygame.locals as pygl
4 import sys
5 from common import *
6 import time
7 from cartas import *
```

- **Random:** esta proporciona funciones para generar números aleatorios y realizar selecciones aleatorias.
- **Pygame as pyg:** simplifica el uso de la funcionalidad de su funcionalidad en el código y para manejar gráficos y eventos del juego.
- **Pygame.locals as pygl:** es un módulo dentro de la biblioteca pygame que incluye definiciones de constantes para eventos, teclas, y otros aspectos relacionados con la funcionalidad de la interfaz de usuario en Pygame.
- **Sys:** esta proporciona acceso a funcionalidades específicas del sistema, y principalmente en esta clase gestiona la salida estándar del juego y controla el comportamiento del programa al final de su ejecución.
- **Common:** importa la clase que contiene valores y constantes que son utilizados para proporcionar valores predefinidos para las dimensiones y posiciones de los elementos gráficos del juego.
- **Time:** proporciona funciones relacionadas con el manejo del tiempo. En esta clase se usa para controlar la temporización y el ritmo del juego.
- **Cartas:** importa la clase que contiene definiciones y funciones específicas relacionadas con la gestión de las cartas en el juego.

Constructor:

El constructor de esta clase se define de la siguiente manera:

```

10 class Juego:
11
12     def __init__(self):
13         self._running = True
14         self.screen = None
15         self.size = self.width, self.height = WIDTH, HEIGHT
16         self.time_limit = 600 # Límite del tiempo en segundos
17         self.start_time = time.time() # Hora de inicio del temporizador
18         self.estadisticas = {} # Parte del registro
19         self.usuario = ""

```

- **Self._running:** es una variable booleana que se utiliza para controlar el ciclo principal del juego. Mientras sea “True”, el juego seguirá ejecutándose, y si es “False”, el juego se detendrá.
- **Self.screen:** variable que se usa para almacenar el objeto de la pantalla que se crea más adelante en el método on_init(). Inicialmente, se establece en None porque aún no se ha creado.
- **Self.size = self.width, self.height = WIDTH, HEIGHT:** se crea una variable “size” que define las dimensiones de la ventana del juego con otras variables que se les asigna los valores de “WIDTH” y “HEIGHT” de la clase common.
- **Self.time_limit:** es una variable que establece el límite de tiempo del juego en segundos en el que el jugador debe completar el juego. Su valor puede variar.
- **Self.start_time = time.time():** una variable que almacena el tiempo actual en el momento que se inicializa el juego, y es para calcular cuánto tiempo ha pasado desde que se inició.
- **Self.estadisticas = {}:** inicializa un diccionario vacío para almacenar las estadísticas de los jugadores como el número de victorias y derrotas.
- **Self.usuario:** variable que almacena el nombre del jugador.

Método on_init:

Este método establece todas las bases necesarias para que el juego funcione correctamente y muestre adecuadamente la pantalla. También es la responsable de inicializar todos los componentes para que el juego comience.

```

22     def on_init(self):
23
24         pygame.init()
25         self.screen = pygame.display.set_mode(self.size)
26         pygame.display.set_caption("Un solitario más.")
27
28         # Cargar la imagen de fondo y redimensionarla al tamaño del tablero
29         self.background = load_image('bg.png')
30         self.background = pygame.transform.scale(self.background, size=(self.width, self.height))
31
32         self.pilas = []
33         self.pilas_subir = []
34         self.mostradas = []
35         self.maz = Mazo()
36         self.deal()

```

- **Pygame.init():** función propia de pygame, ya que inicializa todos los módulos de Pygame. Es esencial para usar cualquier funcionalidad de la librería como gráficos, eventos y sonidos.
- **Self.screen = pygame.display.set_mode(self.size):** se crea una variable llamada “screen” que crea la ventana del juego con el tamaño especificado de la variable “size” del constructor, la función (**pygame.transform.scale**) devuelve un objeto que representa la ventana, y almacena en la variable creada.
- **pygame.display.set_caption(" "):** esta función establece el título de la ventana del juego en la parte superior izquierda.
- **self.background = load_image('bg.png'):** se crea la variable de fondo y con el método de cargar la imagen, carga la imagen del fondo del juego desde el archivo enviado por parámetro.
- **self.background = pygame.transform.scale():** se usa la variable anterior para redimensionar la imagen de fondo para que coincida con el tamaño de la ventana del juego con la función “**pygame.transform.scale()**” que recibe como parámetro la variable donde está la imagen y el ancho y altura de la ventana.
- **self.pilas = []:** inicializa una lista vacía para almacenar las pilas de cartas en el juego. Cada pila tendrá una colección de cartas.
- **Self.pilas_subir = []:** inicializa una lista vacía para almacenar las pilas donde se colocan las cartas para ganar, estas pilas son el destino donde el jugador debe colocar las cartas.
- **Self.mostradas= []:** inicializa una lista vacía para almacenar las cartas que se han mostrado desde el mazo. Y son las cartas que están disponibles para el jugador durante el juego.
- **Self.maz = Mazo():** se crea un objeto de la clase Mazo, que representa el mazo de cartas del juego.

- **Self.deal():** llama al método “deal()” para repartir las cartas entre las pilas y ajustar sus posiciones en la pantalla. Además, organiza las cartas en las pilas de acuerdo con las reglas del juego, y lo prepara.

Método on_execute:

Este método es fundamental para la ejecución continua del juego, ya que controla el ciclo de vida de él, y gestiona eventos, actualiza el estado del juego, y también el de la pantalla para reflejar los cambios.

```

37     def on_execute(self):
38
39         self.solicitar_usuario()
40         self.dragging = []
41         self.clicked_sprites = []
42
43         if self.on_init() == False:
44             self._running = False
45
46         while self._running:
47             for event in pygame.event.get():
48                 self.on_event(event)
49             self.on_loop()
50             self.on_render()
51             self.on_cleanup()

```

- **Self.solicitar_usuario():** este método solicita al jugador que ingrese su nombre antes de empezar el juego.
- **Self.dragging = []:** inicializa una lista vacía para almacenar las cartas que el jugador va a arrastrar, y se implementará para la funcionalidad de arrastrar y soltar.
- **Self.clicked_sprites = []:** inicializa una lista vacía para almacenar y gestionar las cartas que el jugador ha clicado para una gestión de las cartas.
- **If self.on_init():** este condicional se asegura que, si el método de inicializar (on_init) está presentando algún fallo, el juego no continúe por medio de **self._running = False**, se encarga de evitar de que el juego se ejecute si la inicialización falla.
- **While self._running:** mientras que el self._running sea “True” este bucle controla la ejecución continua de los procesos del juego.

- **For event in pyg.event.get():** revisa cada evento que ocurre en el juego como los clics del ratón sobre las cartas y los pasa al método “**self.on_event(event)**” para que actúe según esas acciones del jugador como mover cartas o cerrar la ventana.
- **Self.on_loop():** se llama a este método dentro de la ejecución del juego, ya que actualiza el estado del juego en cada iteración del bucle. Y aplica las reglas del juego como verificar si se completó una pila o hay una victoria.
- **Self.on_render():** este método dibuja el estado actual del juego en la pantalla, actualizando la visualización en cada ciclo, y así el jugador ve los cambios y movimientos realizados.
- **Self.on_cleanup():** cuando el juego termina, este método se asegura de que todos los recursos como imágenes sean liberados correctamente para evitar pérdidas de memoria.

Método on_loop:

Se encarga de actualizar el estado del juego en cada iteración, gestionando principalmente el temporizador, la lógica de arrastrar cartas y verificar si el jugador ha ganado.

```

52     def on_loop(self):
53
54         self.update_timer()
55
56         if self.dragging:
57             desp = 0
58             for card in self.dragging:
59                 card.arrastrar(pyg.mouse.get_pos()[0], pyg.mouse.get_pos()[1]+desp)
60                 desp += DISTY_PILAS
61
62         #Verificar si gano
63         f = True
64         for pilasub in self.pilas_subir:
65             if len(pilasub.cartas) != 13:
66                 f = False
67         if f:
68             self.victory()

```

- **Self.update_timer():** este método actualiza el temporizador del juego que controla el límite de tiempo para completar el juego.

- **If self.dragging:** este condicional verifica si hay una carta clickeada en la lista, y si lo hay recorre por medio del (**for card in self.dragging**) la carta que hay para que se mueva la posición de la carta y seguir al ratón.
- **card.arrastrar(pyg.mouse.get_pos()[0], pyg.mouse.get_pos()[1]+desp):** este comando actualiza la posición de cada carta arrastrada según la posición actual del ratón (**pyg.mouse.get_pos()[0], pyg.mouse.get_pos()[1]**), haciendo que la carta se desplace verticalmente con el valor de “desp” para que las cartas apiladas se separen un poco en su presentación.
- **desp += DISTY_PILAS:** **DISTY_PILAS:** es una constante que define la separación vertical entre cartas. Se asegura que el jugador si arrastra varias cartas estas se apilen a una distancia visual agradable.
- Después de manejar el arrastre de las cartas, el juego verifica si el jugador ha ganado o todavía no. Por eso es por lo que se declara una variable (**f = True**) y que cambiará según la verificación.
- Primero se revisan todas las pilas de subida por medio del bucle (**for pilasub in self.pilas_subir**) y si el tamaño de alguna de las pilas de subida no tiene 13 cartas (**if len(pilasub.cartas) != 13**) el estado de la variable “f” cambia a “False” indicando que todavía el jugador no ha ganado
- **If f:** pero si “f” es “True” se llama al método victoria() y se indica que el jugador logró ganar el juego completando las pilas de subida.

Método on_render:

Este método se encarga de dibujar todos los elementos gráficos del juego como el fondo, las cartas, y el temporizador, en la pantalla.

```

70     def on_render(self):
71
72         self.screen.blit(self.background, (0, 0))
73         font = pyg.font.Font( name: None, size: 36)
74         remaining_time = max(0, self.time_limit - self.elapsed_time)
75         timer_text = font.render( text: f'Tiempo restante: {remaining_time}s', antialias: True, color: (255, 255, 255))
76         self.screen.blit(timer_text, (10, 10))

```

En este primer bloque de código del método:

- **self.screen.blit(self.background, (0, 0)):** se usa la variable “screen” con la función “blit” propia de pygame para que copie la imagen del fondo “self.background” a la pantalla en las coordenadas especificadas, y en este caso “(0, 0)” para colocarla en el fondo.
- **font = pyg.font.Font(None, 36):** se crea una fuente para texto con el comando de “pyg...” para mostrar en este caso el temporizador en la pantalla. No se le agina ninguna fuente (None) y se le da un tamaño de 36 pixeles.

- **remaining_time = max(0, self.time_limit - self.elapsed_time):** este calcula el tiempo restante del juego. La palabra “max(0,)” se asegura que el tiempo no sea negativo, y para calcular se resta el tiempo definido (self.time_limit) y el tiempo transcurrido (self.elapsed_time).
- **timer_text = font.render(f'Tiempo restante: {remaining_time}s', True, (255, 255, 255)):** este genera el texto que muestra el temporizador usando “font.render()” para convertir texto (en este caso, f'Tiempo restante: {remaining_time}s') en una imagen de superficie. Se le asigna “True” para indicar que el texto se vea suave y menos pixelada. Y se le asigna los colores.
- **self.screen.blit(timer_text, (10, 10)):** dibuja el texto del temporizador en la pantalla con la función “blit” y se colocan en las coordenadas (10, 10).

```

78         # Dibujar mazo
79         self.screen.blit(self.maz.image, self.maz.rect)
80
81         # Dibujar subidas
82         for pinta in self.pilas_subir:
83             self.screen.blit(pinta.image, pinta.rect)
84             for carta_p in pinta.cartas:
85                 self.screen.blit(carta_p.image, pinta.rect)
86

```

- **self.screen.blit(self.maz.image, self.maz.rect):** dibuja el mazo de cartas en la pantalla, y “blit” se encarga de eso y de colocarla en las coordenadas definidas por “self.maz.rect”.
- **for pinta in self.pilas_subir:** se itera sobre una de las pilas de subida, que son los lugares donde las cartas deben ser apiladas en orden para ganar.
- **self.screen.blit(pinta.image, pinta.rect):** dibuja la imagen de la pila respecto a su pinta en su posición correspondiente.
- **for carta_p in pinta.cartas:** si hay cartas en la pila, itera sobre cada una de ellas y las dibuja con la función **self.screen.blit(carta_p.image, pinta.rect)** con su respectiva imagen.

En este último bloque del método on_render

```

87         # Dibujar pilas
88         for pila in self.pilas:
89             for card in pila:
90                 if not card in self.dragging:
91                     self.screen.blit(card.image, card.rect)
92
93         # Dibujar mostradas
94         for ncart, card in enumerate(self.mostradas[-CARTAS_MOSTRAR:]):
95             if not card in self.dragging:
96                 card.settopleft(MOSTRADA_POSX+DISTX_MOSTRADA*ncart, MOSTRADA_POSY)
97                 self.screen.blit(card.image, card.rect)
98
99         # Dibujar arrastrando, al final para que no se tapen
100        for card in self.dragging:
101            self.screen.blit(card.image, card.rect)
102        pygame.display.flip()

```

- **Dibujo de pilas:** Se itera sobre las pilas de cartas en el área de juego por el (**for pila in self.pilas:**) y dentro de esa iteración también recorre cada carta que hay en una pila (**for card in pila:**). Con la condicional (**if not card in self.dragging:**) solo se dibujan las cartas que están en las pilas y que no están siendo arrastradas (**self.screen.blit(card.image, card.rect)**) por el jugador en ese momento, y se dibujarán en el último paso para no superponerse con otras cartas.
- **Dibujo mostradas:** Pasa similar para dibujar las cartas que se muestran del mazo. Se itera sobre las últimas cartas mostradas (**for ncart, card in enumerate(self.mostradas[-CARTAS_MOSTRAR:]):**) que son las que están visibles para el jugador después de haberlas sacado del mazo. Con el condicional (**if not card in self.dragging:**) si no están siendo arrastradas se dibujan. Además, se ajusta la posición de cada carta visible (**card.settopleft(MOSTRADA_POSX+DISTX_MOSTRADA*ncart, MOSTRADA_POSY)**) y se posiciona en los parámetros enviados, y cada carta siguiente se desplaza horizontalmente por un valor de DISTX_MOSTRADA. Y, por último, **self.screen.blit(card.image, card.rect):** dibuja cada carta mostrada en su posición ajustada.
- **Dibujo de las cartas arrastradas:** Se itera sobre todas las cartas que el jugador está arrastrando (**self.screen.blit(card.image, card.rect)**), y dibuja la carta arrastrada (**self.screen.blit(card.image, card.rect)**) en la pantalla después de todas las demás para que se muestre por encima de las otras.
- **Pygame.display.flip():** Actualiza la pantalla completa con los nuevos gráficos dibujados. Pygame usa un método de doble buffer, lo que significa que todas las

operaciones de dibujo se realizan en segundo plano, y luego se cambia todo el contenido visible de una vez con `flip()`.

Método `on_cleanup`:

Este método es responsable de limpiar los recursos y terminar el juego correctamente. Es crucial para que los elementos utilizados se liberen adecuadamente y el juego cierre sin problemas.

```
103         def on_cleanup(self):
104
105             pygame.quit()
```

El método **`pygame.quit()`** pertenece a Pygame y se utiliza para cerrar todos los módulos de Pygame y limpiar los recursos que utilizó.


Método `on_event`:

Esta es responsable de manejar los eventos del juego como los clics del ratón, las pulsaciones de teclas y el cierre de la ventana.

```
106         def on_event(self, evento):
107
108             if evento.type == QUIT:
109                 self._running = False
110             elif evento.type == pygame.KEYDOWN:
111                 if evento.key == pygame.K_ESCAPE:
112                     self.on_init()
```

- **`Evento.type`** se refiere al evento que ha sido detectado por el sistema que utiliza pygame para manejar diversas acciones y entradas del usuario como interacción con el teclado o ratón.
- **`If evento.type == QUIT`**: si el evento detectado es de tipo QUIT significa que el usuario ha intentado cerrar la ventana del juego por lo que el atributo “running” cambia a “False” y cierra el juego.
- **`Elif evento.type == pygame.KEYDOWN`**: verifica si el evento recibido es del tipo KEYDOWN (evento que se genera cuando el usuario presiona una tecla), y si lo es entra en otra condicional (**`if evento.key == pygame.K_ESCAPE`**;) que comprueba si la tecla (escape) ha sido presionada. `Pyg.K_ESCAPE` es una

constante en Pygame que representa la tecla Escape. Si se cumple llama al método **on_init** que reinicia el juego sin necesidad de cerrar y volver abrir la aplicación.

```
114         elif evento.type == pygame.MOUSEBUTTONDOWN:
115             pos = pygame.mouse.get_pos()
116
117             # Juntar todos los sprites y guardar los cliqueados.
118             tipo = None
119             for i in self.pilas:
120                 for x in i:
121                     if x.rect.collidepoint(pos):
122                         tipo = PILA
123                         self.clicked_sprites.append(x)
124
125             for i in self.mostradas:
126                 if i.rect.collidepoint(pos):
127                     tipo = MOSTRADAS
128                     self.clicked_sprites.append(i)
129
130             if self.clicked_sprites:
131                 cliqueada = self.clicked_sprites[-1]
132                 if tipo == MOSTRADAS:
133                     if cliqueada == self.mostradas[-1]:
134                         self.dragging.append(cliqueada)
135                 elif tipo == PILA:
136                     if cliqueada.estado == ARRIBA:
137                         cliquea_index_pila = cliqueada.pila.index(cliqueada)
138                  for cartasacar in cliqueada.pila[cliquea_index_pila:]:
139                     self.dragging.append(cartasacar)
140                 else: #si esta hacia abajo y se cliquea voltearla
141                     if cliqueada.pila and cliqueada == cliqueada.pila[-1]:
142                         cliqueada.mostrar()
```

- **Elif evento.type == pygame.MOUSEBUTTONDOWN:** Si el evento recibido es un clic del ratón (MOUSEBUTTONDOWN), se obtiene la posición actual del ratón (**pos = pygame.mouse.get_pos()**) en la ventana del juego en el momento que se produce el clic.
- **Tipo == None:** Se inicializa una variable para almacenar el tipo de sprite que ha sido clickeado, ya sea a una carta en una pila o en una carta mostrada. Se recorre todas las pilas de cartas (**for i in self.pilas:**) y también cada carta en la pila actual (**for x in i:**). Con el condicional (**if x.rect.collidepoint(pos)**) que verifica si el clic está dentro (collidepoint) del área del rectángulo (rect) de cada carta (x), y si así fue, cambia el estado de la variable **tipo en PILA** para indicar que el clic fue sobre una carta en una pila. Luego, añade la carta clickeada (x) a la lista (**self.clicked_sprites.append(x)**) que almacena las cartas clickeadas.
- **For i in self.mostradas:** Recorre todas las cartas que está mostrada y con el condicional (**if x.rect.collidepoint(pos)**) verifica si el clic está dentro del área de

alguna de las cartas, y si sí cambia el estado del tipo por MOSTRADAS y añade esa carta (i) a la lista (**`self.clicked_sprites.append(i)`**).

- **If `self.clicked_sprites`:** Verifica si hay alguna carta en la lista (`clicked_sprites`) , y si lo hay la almacena en la variable que toma la última carta clickeada (**`self.clicked_sprites[-1]`**). Luego verifica si el tipo de carta clickeada es MOSTRADAS (**`if tipo == MOSTRADAS:`**) , luego comprueba si la carta clickeada es la última en la lista (**`if clickeada == self.mostradas[-1]:`**). Y si lo fue, se añade a la lista (`self.dragging`), que indica que el jugador está intentando arrastrarla.
- **Elif `tipo == PILA`:** Verifica si el tipo de carta es clickeado en una pila, si lo es, comprueba que (**`if clickeada.estado == ARRIBA:`**) la carta clickeada esté en un estado de arriba (visible). Luego, obtiene el índice de la carta en su pila (**`clickea_index_pila = clickeada.pila.index(clickeada)`**) e itera desde la carta clickeada hasta el final de la pila (**`for cartasacar in clickeada.pila[clickea_index_pila:]`**). Y añade cada carta desde el índice hasta el final de la pila a la lista (**`self.dragging.append(cartasacar)`**). En caso contrario (**`else`**) si la carta clickeada está en estado “ABAJO” (**`if clickeada.pila and clickeada == clickeada.pila[-1]:`**), verifica si es la última carta en su pila, y si lo es, voltea (**`clickeada.mostrar():`**) la carta para que pueda ser arrastrada.

```
143         elif evento.type == pygame.MOUSEBUTTONUP:
144             pos = pygame.mouse.get_pos()
145             tipo_drop, piladrop_index = self.check_pila_area(pos[0], pos[1])
146             if(tipo_drop == PILA):
147                 if self.dragging:
148                     piladrop = self.pilas[piladrop_index]
149                     if (piladrop): # Si la pila de destino no está vacía
150                         if (piladrop_index != -1 and self.matchable(self.dragging[0], piladrop[-1])):
151                             for card in self.dragging:
152                                 if card.pila:
153                                     card.pila.remove(card)
154                                     # Verificar si la última carta de la pila anterior está boca abajo y voltearla
155                                     if card.pila and card.pila and card.pila[-1].estado == ABAJO:
156                                         card.pila[-1].mostrar()
157                                 else:
158                                     self.mostradas.remove(card)
159                                     card.settopleft(piladrop[-1].posx, piladrop[-1].posy + DISTY_PILAS)
160                                     piladrop.append(card)
161                                     card.pila = piladrop
```

`Elif evento.type == pygame.MOUSEBUTTONUP:` Verifica si el tipo de evento es MOUSEBUTTONUP (si el usuario soltó el botón del ratón), si sí lo hizo obtiene la posición actual del cursor del ratón cuando se ha soltado la carta (**`pos = pygame.mouse.get_pos()`**) y crea dos variables (**`tipo_drop, piladrop_index = self.check_pila_area(pos[0], pos[1])`**) para llamar al método “check_pila_area” es un método que evalúa la posición del mouse y devuelve dos valores: el tipo de área donde se soltó el mouse (PILA, MOSTRADAS, etc.) y el índice de la pila o el área correspondiente. Esto es necesario para saber dónde intentar colocar las cartas

arrastradas. Se verifica si el área de destino es una pila (***if(tipo_drop == PILA):***), luego se hace otra verificación (***if self.dragging:***) para saber si hay cartas en la lista ***self.dragging*** que contiene las cartas que han sido arrastradas. Se usa una variable para obtener la pila de destino (***piladrop = self.pilas[piladrop_index]***) y saber dónde colocar las cartas arrastradas. Se comprueba que la pila de destino no está vacía (***if (piladrop):***) y verificamos que el índice de la pila destino sea válido y no sea -1, y que la primera carta en la lista de cartas arrastradas sea compatible con la carta superior de la pila destino mediante el método “matchale” (***if (piladrop_index != -1 and self.matchable(self.dragging[0], piladrop[-1])):***). Luego se hace una iteración (***for card in self.dragging:***) en las cartas de la lista ***self.dragging*** y verifica si la carta (***card***) está actualmente en una pila (***if card.pila:***), y si lo está, se elimina la carta de su pila actual (***card.pila.remove(card)***), ya que se está moviendo a una nueva pila. Después se verifica si la pila original aún tiene cartas, y si la última carta en la pila está boca abajo (***if card.pila and card.pila[-1].estado == ABAJO:***), se voltee (***card.pila[-1].mostrar()***).

En caso contrario de que la carta no está en una pila (***else***), la elimina de la lista ***self.mostradas*** que contiene las cartas mostradas en la pantalla (***self.mostradas.remove(card)***). Y establece la nueva posición de la carta en la pila de destino (***piladrop***), esta posición se basa en la posición de la carta en la parte superior de la pila destino (***piladrop[-1]***), con un desplazamiento vertical adicional (***DISTY_PILAS***) para apilar las cartas (***card.settopleft(piladrop[-1].posx, piladrop[-1].posy + DISTY_PILAS)***). Después se añade la carta a la pila de destino (***piladrop.append(card)***) que el usuario ha escogido. Por último, establece que la pila de la carta ahora es pila de destino (***card.pila = piladrop***), es decir, que actualiza la referencia de la pila de carta.

```

162         else: # Si la pila de destino está vacía y la carta es un Rey
163             if piladrop_index != -1 and self.dragging[0].numero == 12:
164                 for card in self.dragging:
165                     if card.pila:
166                         card.pila.remove(card)
167                         # Verificar si la última carta de la pila anterior está boca abajo y voltearla
168                         if card.pila and card.pila[-1].estado == ABAJO:
169                             card.pila[-1].mostrar()
170                     else:
171                         self.mostradas.remove(card)
172                         if (card == self.dragging[0]):
173                             self.dragging[0].settopleft(PILAS_XINICIAL + (piladrop_index * DISTX_PILAS),
174                                                         PILAS_YINICIAL)
175                         else:
176                             card.settopleft(piladrop[-1].posx, piladrop[-1].posy + DISTY_PILAS)
177                         piladrop.append(card)
178                         card.pila = piladrop

```

En el caso contrario (***else***) que la pila de destino (***piladrop***) está vacía y si la primera carta en ***self.dragging*** (lista de cartas arrastrada) es un Rey (número 12) (***if***

piladrop_index != -1 and self.dragging[0].numero == 12:), recorre sobre cada carta de la lista de cartas arrastradas (***for card in self.dragging:***), y si la pila carta está en una pila (***if card.pila:***), esta se elimina de esa pila para colocarse en la nueva (***card.pila.remove(card)***). Después de eliminar la carta, verifica si la última carta en la pila original está ABAJO (***if card.pila and card.pila[-1].estado == ABAJO:***) para luego voltearla y que quede visible (***card.pila[-1].mostrar()***).

En caso contrario (***else***) que la carta no está en una pila es porque se encuentra en la zona de cartas mostradas del mazo (***self.mostradas***), por lo que se elimina de esa zona (***self.mostradas.remove(card)***). Y ahora se procede a actualizar la carta que se movió, pero hay que verificar si la carta es la primera en la lista de cartas arrastradas (***if (card == self.dragging[0]):***) y si lo es, la coloca en las posiciones iniciales de la pila (x, y), ajustándola por el índice de la pila de destino (***self.dragging[0].settopleft(PILAS_XINICIAL + (piladrop_index * DISTX_PILAS), PILAS_YINICIAL)***).

Y para las cartas que no son (***else***) la primera en la lista de arrastradas, se colocan encima de la última carta en la pila de destino con un desplazamiento vertical para apilar las cartas (***card.settopleft(piladrop[-1].posx, piladrop[-1].posy + DISTY_PILAS)***). Por último, se añade la carta arrastrada a la pila de destino (***piladrop.append(card)***), y se actualiza la referencia de la pila para que el estado de la carta esté con su nueva ubicación de pila (***card.pila = piladrop***).

```

181         elif(tipo_drop == PILASUBIR and len(self.dragging)==1):
182             if self.dragging:
183                 card = self.dragging[0]
184                 if(card.pinta == piladrop_index.pinta):
185                     if card.numero == 0:
186                         self.subir(card, piladrop_index)
187                     if piladrop_index.cartas:
188                         if card.numero == piladrop_index.cartas[-1].numero+1:
189                             self.subir(card, piladrop_index)
190             elif(tipo_drop == MAZO):
191                 if self.maz.cartas:
192                     for card in self.maz.cartas[:CARTAS_MOSTRAR]:
193                         self.maz.cartas.remove(card)
194                         card.mostrar()
195                         self.mostradas.append(card)
196                 else:
197                     while self.mostradas:
198                         card = self.mostradas[0]
199                         self.maz.cartas.append(card)
200                         self.mostradas.remove(card)

```

Este (elif) verifica si la carta arrastrada está siendo soltada sobre una "pila de subida" (PILASUBIR) y si solo se está arrastrando una carta (indicada por

`len(self.dragging) == 1)(elif(tipo_drop == PILASUBIR and len(self.dragging)==1):`). Y si se verifica que la lista de cartas arrastradas tiene cartas (**`if self.dragging:`**), se almacena la primera carta arrastrada en la variable (`card`) (**`card = self.dragging[0]`**). Si la pinta de la carta que está siendo arrastrada coincide con la pinta de la pila de subidas (**`if(card.pinta == piladrop_index.pinta):`**), se verifica que el número de la carta es 0 (una As)(**`if card.numero == 0:`**), si es así se llama al método de subir para mover esa carta a la pila de subida (**`self.subir(card, piladrop_index)`**). Si la pila de subida (`piladrop_index.cartas`) no está vacía (**`if piladrop_index.cartas:`**), otra vez se verifica si el número de la carta que está siendo arrastrada es un número más que el de la última carta en la pila (**`if card.numero == piladrop_index.cartas[-1].numero+1:`**), y si lo es se llama al método de subir para mover la carta a la pila de subida (**`self.subir(card, piladrop_index)`**).

El otro (**`elif`**) verifica si el jugador suelta una carta sobre el mazo (**`elif(tipo_drop == MAZO):`**), y verifica si hay cartas en el mazo (**`if self.maz.cartas:`**). Este for recorre cada una de las cartas que hay en el mazo, y toma la primera (`CARTAS_MOSTRAR`) carta del mazo, y la elimina del mazo(**`self.maz.cartas.remove(card)`**), para luego voltearla (**`card.mostrar()`**) y la añade a la lista de cartas mostradas (**`self.mostradas.append(card)`**).

En caso contrario que no haya cartas en el mazo (**`else`**), pero mientras sí haya en la pila de cartas mostradas (**`while self.mostradas`**), se toma todas las cartas de la pila de cartas mostrada que la almacena en una variable (**`card = self.mostradas[0]`**), las mueve devuelta al mazo (**`self.maz.cartas.append(card)`**) y las elimina de la lista de cartas mostradas (**`self.mostradas.remove(card)`**).

```

204         for card in self.dragging:
205             card.settopleft(card.posfx, card.posfy)
206         self.dragging = []
207         self.clicked_sprites = []

```

Por último, este for itera sobre todas las cartas de la lista de cartas que el jugador estaba arrastrando con el ratón (**`for card in self.dragging:`**), y establece la posición de la carta (`card`) usando el método “settopleft” con las variables que almacenaban las posiciones originales (**`card.settopleft(card.posfx, card.posfy)`**), si la carta no se puede soltar en una pila válida, esta regresa a su posición original con ese método. Y luego, se vacían la lista de cartas arrastradas eliminando a todas (**`self.dragging = []`**), y también se vacía la lista de clickeadas a las cartas (**`self.clicked_sprites = []`**).

Método solicita usuario:

Este método es el responsable de pedir al usuario que ingrese su nombre y luego cargar las estadísticas correspondientes a ese nombre.

```
209 # ----- FUNCIONES DEL JUEGO -----
    1 usage
210     def solicitar_usuario(self):...
211
212         self.usuario = input("Introduce tu nombre de usuario: ").strip()
213         self.cargar_estadisticas()
214
```

- **Self.usuario = input("Introduce tu nombre de usuario: ").strip():** Con la función “input” se muestra un mensaje en la consola para solicitar el nombre y capturarlo, el “strip()” es para eliminar los espacios en blanco antes y después del nombre que se haya ingresado. Y la variable “usuario” almacena el nombre que se haya registrado.
- **Self.cargar_estadisticas():** Llama al método “cargar_estadísticas()” que se encarga de buscar y cargar las victorias y derrotas que estén asociadas el nombre.

Método update_timer:

Se encarga de actualizar el temporizador del juego.

```
215     def update_timer(self):
216
217         self.elapsed_time = int(time.time() - self.start_time)
218         if self.elapsed_time >= self.time_limit and self._running:
219             self.game_over()
220
```

- **Self.elapsed_time = int(time.time() - self.start_time):** Calcula el tiempo transcurrido desde que comenzó el juego. El “int” convierte el tiempo en un número entero, la función “time.time()” devuelve el tiempo actual en segundos y se resta con el temporizador que comenzó cuando el juego inició, de esa manera calcula el tiempo que ha transcurrido.
- **if self.elapsed_time >= self.time_limit and self._running:** Verifica si el tiempo transcurrido ha alcanzado o superado el tiempo límite para completar el juego, si se cumplen estas condiciones se llama al método “game_over()” que finaliza el juego y pierde el jugador.

Método game_over:

Este método registra una derrota al jugador cuando no ha podido alcanzar a completar el juego en el tiempo límite.

```
221         def game_over(self):
222
223             print("¡Tiempo agotado!")
224             self._running = False
225             self.registrar_derrota()
226             self.show_game_over_screen()
227
```

- **print("¡Tiempo agotado!")**: Imprime un mensaje cuando se acabó el tiempo.
- **self._running = False**: Cambia el valor de running por "False", lo que hace que se detenga el juego.
- **self.registrar_derrota()**: Se llama al método que se encarga de guardar y actualizar los datos del jugador para indicar que ha perdido una partida.
- **self.show_game_over_screen()**: Llama al método "show_game_over_screen" que se encarga de mostrar una pantalla visual que indica al jugador que el juego ha terminado.

Método victory:

Este método es responsable de manejar todo lo que debe ocurrir cuando el jugador gana, como detener el juego y registrar la victoria.

```
228         def victory(self):
229
230             print("GANASTEEEEEEEEEEE")
231             self._running = False
232             self.registrar_victoria()
233
234             self.show_victory_screen()
235
```

- **print("GANASTEEEEEEEEEEE")**: Muestra un mensaje en la consola que el jugador ganó la partida antes del tiempo límite.
- **self._running = False**: Detiene el juego cambiando el valor de "running" por "False".

- **self.registrar_victoria():** Llama al método que guarda y actualiza las victorias del jugador.
- **self.show_victory_screen():** Llama al método que muestra una pantalla gráfica informando que el jugador ha ganado.

Método show_victory_screen:

Se encarga de presentar al jugador una pantalla de victoria y opciones después de ganar el juego.

```

237     def show_victory_screen(self):
238
239         font = pygame.font.Font( name: None, size: 36) # Establecer una fuente
240         running = True
241
242         while running:
243             self.screen.fill((255, 182, 193)) # Fondo rosadito
244             # Mensaje de felicitación
245             text = font.render( text: "¡Felicidades! Has ganado el juego.", antialias: True, color: (255, 255, 255))
246             self.screen.blit(text, (50, 150))
247
248             # Opciones del menú
249             play_again_text = font.render( text: "Presiona 1 para jugar de nuevo", antialias: True, color: (255, 255, 255))
250             quit_text = font.render( text: "Presiona 2 para salir", antialias: True, color: (255, 255, 255))
251             self.screen.blit(play_again_text, (50, 200))
252             self.screen.blit(quit_text, (50, 250))
253
254             # Muestra el nombre del usuario
255             user_text = font.render( text: f"Usuario: {self.usuario}", antialias: True, color: (255, 255, 255))
256             self.screen.blit(user_text, (50, 300))

```

- **font = pygame.font.Font(None, 36):** Crea un objeto de fuente de texto utilizando la clase pygame.font.Font. Se utiliza None para especificar la fuente por defecto, y el tamaño de la fuente se establece en 36 píxeles.
- **running = True:** Se inicializa una variable booleana running en True, que controlará el bucle principal de esta pantalla de victoria.
- **while running:** Inicia un ciclo que se ejecuta mientras “running” sea “True”, y así mantener la pantalla de victoria activa hasta que el usuario decida qué hacer.
- **self.screen.fill((255, 182, 193)):** Rellena toda la pantalla con un color rosado claro.
- **text = font.render(" ", True, (255, 255, 255)):** Crea una superficie de texto renderizado con el mensaje en color en blanco. Y self.screen.blit(text, (50, 150)), se encarga de dibujarlo en la pantalla en las coordenadas seleccionadas.
- **play_again_text = font.render y quit_text = font.render:** Renderiza dos textos que uno indica si quiere volver a jugar y el otro si desea salir del juego.
- **self.screen.blit(play_again_text, (50, 200)) y self.screen.blit(quit_text, (50, 250)):** La función “blit” se encarga de dibujar los textos de opciones en la pantalla con las ubicaciones dadas.

- **user_text = font.render(f"Usuario: {self.usuario}", True, (255, 255, 255))**: una variable que renderiza y muestra el nombre del usuario que está jugando en la pantalla. Y con **self.screen.blit(user_text, (50, 300))** se dibuja en la pantalla del juego.

```

258 # Muestra estadísticas de si pierde o gana
259 stats_text = font.render(text=f"Victorias: {self.estadisticas.get('ganados', 0)} | Derrotas: {self.estadisticas.get('perdidos', 0)}", antialias=True, color=(255, 255, 255))
260 self.screen.blit(stats_text, (50, 350))
261
262 pygame.display.flip()
263
264 # Maneja los eventos del menú
265 for event in pygame.event.get():
266     if event.type == pygame.QUIT:
267         running = False
268         sys.exit() # Salir del programa
269     elif event.type == pygame.KEYDOWN:
270         if event.key == pygame.K_1: # Opción para jugar de nuevo
271             self.restart_game()
272             running = False
273         elif event.key == pygame.K_2: # Opción para salir
274             running = False
275             sys.exit() # Salir del programa

```

- **stats_text = font.render()** y **self.screen.blit(stats_text, (50, 350))**: Renderiza y muestra las estadísticas del jugador, específicamente la cantidad de victorias y derrotas.
- **pygame.display.flip()**: Actualiza la pantalla con el contenido que se ha dibujado hasta ese momento.
- **for event in pygame.event.get()**: Inicia un bucle que itera sobre todos los eventos que han ocurrido desde la última iteración del ciclo. Si el evento (**if event.type == pygame.QUIT**) es de tipo QUIT (si el jugador cierra la ventana del juego), se establece **running = False** para salir del ciclo, y luego se llama a **sys.exit()** para finalizar el programa.
- **elif event.type == pygame.KEYDOWN**: Si el tipo de evento que detecta es igual a **pygame.KEYDOWN** y si el jugador presiona la tecla "1" (**if event.key == pygame.K_1**), se llama al método **self.restart_game()** para reiniciar el juego, y luego se establece **running = False** para salir del ciclo de la pantalla de victoria.
- **elif event.key == pygame.K_2**: Si el tipo de evento que detecta es igual a **pygame.KEYDOWN** y el jugador presiona la tecla "2", se establece **running = False** y se llama a **sys.exit()** para salir del juego.

Método show_game_over_screen:

Muestra la pantalla de fin de juego cuando el jugador pierde.

```

277 def show_game_over_screen(self):
278
279     font = pygame.font.Font( name=None, size=36)
280     running = True
281
282     while running:
283         self.screen.fill((255, 182, 193)) # Fondo rosadito
284
285         # Mensaje del fin del juego
286         text = font.render( text: "¡Se acabó el tiempo!", antialias=True, color: (255, 255, 255))
287         self.screen.blit(text, (50, 150))
288
289         # Opciones del menú
290         play_again_text = font.render( text: "Presiona 1 para jugar de nuevo", antialias=True, color: (255, 255, 255))
291         quit_text = font.render( text: "Presiona 2 para salir", antialias=True, color: (255, 255, 255))
292         self.screen.blit(play_again_text, (50, 200))
293         self.screen.blit(quit_text, (50, 250))
294
295         # Muestra el nombre del usuario
296         user_text = font.render( text: f"Usuario: {self.usuario}", antialias=True, color: (255, 255, 255))
297         self.screen.blit(user_text, (50, 300))
298
299         # Muestra estadísticas de si pierde o gana
300         stats_text = font.render(
301             text: f"Victorias: {self.estadisticas.get('ganados', 0)} | Derrotas: {self.estadisticas.get('perdidos', 0)}",
302             antialias=True, color: (255, 255, 255))
303         self.screen.blit(stats_text, (50, 350))
304
305         pygame.display.flip()

```

- **font = pygame.font.Font(None, 36):** Crea un objeto de fuente de texto utilizando la clase `pygame.font.Font`.
- **running = True:** Se inicializa una variable booleana `running` en `True`, que controlará el bucle principal de esta pantalla de victoria.
- **while running:** Inicia un ciclo que se ejecuta mientras “`running`” sea “`True`”, y así mantener la pantalla de derrota activa.
- **self.screen.fill((255, 182, 193)):** Rellena toda la pantalla con un color rosado claro.
- **text = font.render(" ", True, (255, 255, 255)):** Crea una superficie de texto renderizado con el mensaje en color en blanco. Y **self.screen.blit(text, (50, 150))**, se encarga de dibujarlo en la pantalla en las coordenadas seleccionadas.
- **play_again_text = font.render** y **quit_text = font.render:** Renderiza dos textos que uno indica si quiere volver a jugar y el otro si desea salir del juego.
- **self.screen.blit(play_again_text, (50, 200))** y **self.screen.blit(quit_text, (50, 250)):** La función “`blit`” se encarga de dibujar los textos de opciones en la pantalla con las ubicaciones dadas.
- **user_text = font.render(f"Usuario: {self.usuario}", True, (255, 255, 255)):** una variable que renderiza y muestra el nombre del usuario que está jugando en la pantalla. Y con **self.screen.blit(user_text, (50, 300))** se dibuja en la pantalla del juego.

- **stats_text = font.render() y self.screen.blit(stats_text, (50, 350)):** Renderiza y muestra las estadísticas del jugador, específicamente la cantidad de victorias y derrotas.
- **pygame.display.flip():** Actualiza la pantalla con el contenido que se ha dibujado hasta ese momento.

```

307         for event in pygame.event.get():
308             if event.type == pygame.QUIT:
309                 running = False
310                 sys.exit()
311             elif event.type == pygame.KEYDOWN:
312                 if event.key == pygame.K_1: # Jugar de nuevo
313                     self.restart_game()
314                     running = False
315                 elif event.key == pygame.K_2: # Salir
316                     running = False
317                     sys.exit()

```

- **for event in pygame.event.get():** Inicia un bucle que itera sobre todos los eventos que han ocurrido desde la última iteración del ciclo. Si el evento (**if event.type == pygame.QUIT:**) es de tipo QUIT (si el jugador cierra la ventana del juego), se establece `running = False` para salir del ciclo, y luego se llama a `sys.exit()` para finalizar el programa.
- **elif event.type == pygame.KEYDOWN:** Si el tipo de evento que detecta es igual a `pygame.KEYDOWN` y si el jugador presiona la tecla "1" (**if event.key == pygame.K_1:**), se llama al método `self.restart_game()` para reiniciar el juego, y luego se establece `running = False` para salir del ciclo de la pantalla de victoria.
- **elif event.key == pygame.K_2:** Si el tipo de evento que detecta es igual a `pygame.KEYDOWN` y el jugador presiona la tecla "2", se establece `running = False` y se llama a `sys.exit()` para salir del juego.

Método restart_game:

Se encarga de reiniciar el estado del juego para comenzar una nueva partida.

```

322         def restart_game(self):
323
324             self.start_time = time.time()
325             self.pilas = []
326             self.pilas_subir = []
327             self.mostradas = []
328             self.maz = Mazo()
329             self.deal() # Reparte las cartas
330             self._running = True
331             self.on_execute()

```

- **self.start_time = time.time():** Se actualiza el tiempo de inicio del juego con el tiempo actual (*time.time()*), que devuelve el número de segundos. Esto reinicia el cronómetro del juego para la nueva partida.
- **self.pilas = []:** Reinicia la lista de pilas, que contiene las pilas de cartas en el juego. Al vaciar esta lista, se elimina el estado de las pilas de la partida anterior.
- **self.pilas_subir = []:** Reinicia la lista de pilas para subir, donde se almacenan las cartas que el jugador debe colocar en pilas específicas durante el juego.
- **self.mostradas = []:** Reinicia la lista de cartas mostradas, que son las cartas visibles al jugador que han sido extraídas del mazo. Se vacía para que se pueda empezar con un nuevo conjunto de cartas.
- **self.maz = Mazo():** Crea una nueva instancia de Mazo, que representa el mazo de cartas del juego. Esto restablece el mazo a su estado inicial, listo para repartir nuevas cartas.
- **self.deal():** Llama al método deal (repartir cartas), que distribuye las cartas desde el mazo a las pilas y áreas del juego.
- **self._running = True:** Establece el estado del juego a True, indicando que el juego está en ejecución. Esto es necesario para reiniciar el ciclo principal del juego.
- **self.on_execute():** Llama al método on_execute, que inicia el ciclo principal del juego, gestionando eventos, actualizaciones y renderizado.

Métodos de registrar victoria y derrota:

Ambos métodos están diseñados para actualizar las estadísticas del juego de un usuario, y almacenarlas de manera persistente.

```

333     def registrar_victoria(self):_# Parte que registra las partidas
334         # Registra una victoria en el archivo de estadísticas
335         if self.usuario:
336             self.estadisticas['ganados'] = self.estadisticas.get('ganados', 0) + 1
337             self.guardar_estadisticas()
338
339         1 usage
340     def registrar_derrota(self):_# Registra una derrota en el archivo de estadísticas
341         if self.usuario:
342             self.estadisticas['perdidos'] = self.estadisticas.get('perdidos', 0) + 1
343             self.guardar_estadisticas()

```

Como ambos métodos poseen lo mismo, pero cada uno con su respectivo. Se explicará en general.

- **if self.usuario:** Verifica si el atributo `self.usuario` contiene un valor significativo (es decir, si no es `None` ni una cadena vacía). Esto asegura que solo se registre una victoria si hay un usuario asociado. Si no hay un usuario, no se realiza ninguna acción.
- **Self.estadisticas[' '] = self.estadisticas.get(' ', 0) + 1:** Actualiza el contador de victorias o derrotas en el diccionario `self.estadisticas`, y **self.estadisticas.get('ganados', 0)** intenta obtener el valor actual asociado con la clave 'ganados o derrotas' en el diccionario `self.estadisticas`. Si la clave 'ganados o derrotas' no existe, se utiliza un valor predeterminado de 0, y se suma 1 al valor obtenido y se actualiza el diccionario. Esto incrementa el conteo de victorias por una unidad.
- **self.guardar_estadisticas():** Llama al método “guardar_estadisticas”, que se encarga de guardar el diccionario de estadísticas actualizado en un archivo o base de datos para que persista entre sesiones del juego

Método guardar estadísticas:

Este método se encarga de guardar las estadísticas del usuario en un archivo de texto.

```

345     def guardar_estadisticas(self):_# Guarda las estadísticas en un archivo de texto
346
347         with open("estadisticas.txt", "w") as archivo:
348             with open(f"estadisticas_{self.usuario}.txt", "w") as archivo:
349                 archivo.write(f"Juegos ganados: {self.estadisticas.get('ganados', 0)}\n")
350                 archivo.write(f"Juegos perdidos: {self.estadisticas.get('perdidos', 0)}\n")
351

```

- **with open("estadisticas.txt", "w") as archivo:** Abre un archivo llamado estadisticas.txt en modo escritura ("w"). La sentencia with asegura que el archivo se cerrará automáticamente cuando se salga del bloque with, incluso si ocurre una excepción.
- **with open(f"estadisticas_{self.usuario}.txt", "w") as archivo:** se abre otro archivo, cuyo nombre es dinámico y depende del nombre del usuario (self.usuario). Se usa una (f"estadisticas_{self.usuario}.txt") para incluir el nombre del usuario en el nombre del archivo.
- **archivo.write(f"Juegos ganados: {self.estadisticas.get('ganados', 0)}\n"):** Escribe en el archivo una línea de texto que indica el número de juegos ganados. Utiliza **self.estadisticas.get('ganados', 0)** para obtener el número de victorias del diccionario estadísticas, devolviendo 0 si la clave 'ganados' no existe. El \n al final de la cadena añade un salto de línea para separar esta línea de la siguiente.
- **archivo.write(f"Juegos perdidos: {self.estadisticas.get('perdidos', 0)}\n"):** Igual que en el anterior, pero para los juegos perdidos.

Método cargar estadística:

Se asegura que el programa pueda cargar y manejar estadísticas anteriores del usuario o empezar con valores predeterminados si el archivo no está disponible.

```

352     def cargar_estadisticas(self):
353
354         try:
355             with open(f"estadisticas_{self.usuario}.txt", "r") as archivo:
356                 lineas = archivo.readlines()
357                 self.estadisticas['ganados'] = int(lineas[0].split(":")[1].strip())
358                 self.estadisticas['perdidos'] = int(lineas[1].split(":")[1].strip())
359         except FileNotFoundError:
360             self.estadisticas = {'ganados': 0, 'perdidos': 0}

```

- **try:** Se inicia un bloque try para intentar ejecutar el código que puede generar una excepción, en este caso, la posible falta del archivo de estadísticas.
- **with open(f"estadisticas_{self.usuario}.txt", "r") as archivo:** Intenta abrir el archivo de estadísticas específico del usuario (*estadísticas_{self.usuario}.txt*) en modo lectura ("r"). Usa un contexto with para garantizar que el archivo se cierre automáticamente después de su uso.
- **líneas = archivo.readlines():** Lee todas las líneas del archivo y las almacena en una lista llamada líneas. Cada elemento de la lista será una línea del archivo.
- **self.estadisticas['ganados'] = int(líneas[0].split(":")[1].strip()):** Toma la primera línea del archivo (líneas[0]), la divide en dos partes utilizando ":" como

delimitador, toma la segunda parte ([1]), elimina los espacios en blanco alrededor con strip(), y la convierte en un entero. Luego, asigna este valor al contador de juegos ganados ('ganados') en el diccionario self.estadisticas.

- **self.estadisticas['perdidos'] = int(líneas[1].split(":")[1].strip()):** Igual que el anterior, pero con los juegos perdidos.
- **except FileNotFoundError:** Si el archivo no se encuentra, se captura una excepción "FileNotFoundError". Esto evita que el programa se detenga inesperadamente.
- **self.estadisticas = {'ganados': 0, 'perdidos': 0}:** Si se produce la excepción, se inicializa el diccionario self.estadisticas con valores predeterminados de 0 para 'ganados' y 'perdidos'.

Método subir:

Se encarga de manejar el movimiento de una carta de su pila actual o de la lista de cartas mostradas a una nueva pila, actualizando tanto la posición visual de la carta como su referencia interna de pila.

```
363     def subir(self, card, piladrop_index):
364         if card.pila:
365             card.pila.remove(card)
366         else:
367             self.mostradas.remove(card)
368         card.settopleft(piladrop_index.posx, piladrop_index.posy)
369         piladrop_index.cartas.append(card)
370         card.pila = piladrop_index
```

- **if card.pila:** Verifica si la carta (card) está actualmente en alguna pila. Si es así, se elimina la carta de su pila actual (**card.pila.remove(card)**).
- **Else:** Si la carta no está en ninguna pila, significa que la carta está en la lista de cartas mostradas (self.mostradas). Por lo tanto, se elimina la carta de la lista de cartas mostradas utilizando remove.
- **card.settopleft(piladrop_index.posx, piladrop_index.posy):** Establece la nueva posición de la carta (card) usando el método settopleft. La nueva posición se obtiene de las coordenadas posx y posy del índice de la pila de destino (**piladrop_index**).
- **piladrop_index.cartas.append(card):** La carta se agrega a la lista de cartas de la pila de destino (**piladrop_index.cartas**). Aquí, cartas es una lista que representa todas las cartas en la pila de destino.

- **card.pila = piladrop_index:** Se actualiza la referencia de la pila (card.pila) de la carta para que apunte a la nueva pila de destino (piladrop_index).

Método deal:

Configura el tablero del juego al repartir cartas en pilas, asegurándose de que cada pila tenga el número correcto de cartas y configurando las pilas para cartas que se deben subir.

```

372     def deal(self): |
373
374         xact = PILAS_XINICIAL
375         for pilaact in range(7):
376             yact = PILAS_YINICIAL
377             pila = []
378             for numcarta in range(pilaact+1):
379                 carta_ins = self.maz.cartas[0]
380                 carta_ins.settopleft(xact, yact)
381                 carta_ins.pila = pila
382                 pila.append(carta_ins)
383                 self.maz.cartas.remove(carta_ins)
384                 if numcarta==pilaact:
385                     carta_ins.mostrar()
386                     yact += DISTY_PILAS
387                 self.pilas.append(pila)
388                 xact += DISTX_PILAS
389
390             # Crea pilas donde se suben las cartas
391             deltax = 0
392             for pinta in PINTAS:
393                 p = PilaSubir(pinta)
394                 p.settopleft(PILASUBIR_XINICIAL+deltax, PILASUBIR_YINICIAL)
395                 self.pilas_subir.append(p)
396                 deltax+= DISTX_PILAS

```

- **Inicializa xact:** Establece la posición horizontal inicial de las pilas donde se repartirán las cartas.
- **for pilaact in range(7):** Itera 7 veces para crear 7 pilas, una para cada columna de cartas en el juego.
- **yact = PILAS_YINICIAL:** Establece la posición vertical inicial para las cartas en la pila actual.
- **pila = []:** Inicializa una lista vacía para contener las cartas en la pila actual.

- **for numcarta in range(pilaact+1):** Itera desde 0 hasta pilaact (inclusive), donde pilaact representa el índice actual de la pila (empezando desde 0). Esto asegura que la primera pila tenga 1 carta, la segunda tenga 2 cartas, y así sucesivamente hasta la séptima pila.
- **carta_ins = self.maz.cartas[0]:** Obtiene la primera carta del mazo
- **carta_ins.settopleft(xact, yact):** Posiciona la carta en la pantalla en la posición (xact, yact).
- **carta_ins.pila = pila:** Establece que la carta actual pertenece a la pila recién creada.
- **pila.append(carta_ins):** Agrega la carta a la lista pila.
- **self.maz.cartas.remove(carta_ins):** Quita la carta del mazo para que no se vuelva a repartir.
- **if numcarta == pilaact:** Verifica si numcarta es igual a pilaact, y si lo es que muestre (**carta_ins.mostrar()**) la carta (esto significa que es la carta de fondo de la pila y debe ser visible).
- **yact += DISTY_PILAS:** Incrementa la posición vertical (yact) para la próxima carta en la pila.
- **self.pilas.append(pila):** Agrega la pila completa a la lista self.pilas.
- **xact += DISTX_PILAS:** Incrementa la posición horizontal (xact) para la próxima pila.
- **deltax = 0:** Establece la posición horizontal inicial para las pilas donde se subirán las cartas.
- **for pinta in PINTAS:** Itera sobre las pintas, y se crea una instancia de la clase PilaSubir, pasando el tipo de pinta actual al constructor (**p = PilaSubir(pinta)**). Luego, establece la posición en la pantalla donde se ubicará la pila de subida (**p.settopleft(PILASUBIR_XINICIAL + deltax, PILASUBIR_YINICIAL)**). Una vez que la pila ha sido creada y posicionada, se agrega a la lista (**self.pilas_subir.append(p)**) de pilas subir, ya que es una lista que almacena todas las pilas de subida en el juego. Y finalmente, se actualiza “deltax” con la distancia horizontal entre pilas. Deltax se incrementa en la distancia (**deltax += DISTX_PILAS**) para que la siguiente pila de subida se posicione más a la derecha en la pantalla.

Método check_pila_area:

Este método recibe por parámetros la posición que representa las coordenadas de un clic en la pantalla. Su objetivo es identificar si esas coordenadas del clic están sobre el mazo, pilas de cartas o las cartas mostradas, y devolver información sobre dicha área.

```

398     def check_pila_area(self, posx, posy):
399         p_xi = self.maz.posx
400         p_xf = p_xi + TAMX_CARTA
401         p_yi = self.maz.posy
402         p_yf = p_yi + TAMY_CARTA
403         if(posx > p_xi and posx < p_xf and posy > p_yi and posy < p_yf):
404             return MAZO, self.maz
405
406         # Revisa si el click es en las cartas mostradas
407         p_xi = MOSTRADA_POSX
408         p_xf = p_xi + TAMX_CARTA
409         p_yi = MOSTRADA_POSY
410         p_yf = p_yi + TAMY_CARTA
411         if(posx > p_xi and posx < p_xf and posy > p_yi and posy < p_yf):
412             return MOSTRADAS, None

```

- **p_xi y p_yi:** Son variables que almacenan las coordenadas del vértice superior izquierdo del mazo (**self.maz.posx, self.maz.posy**)
 - **p_xf y p_yf:** Son variables que almacenan las coordenadas del vértice inferior izquierdo del mazo. Y se calcula sumando las dimensiones de una carta a las coordenadas originales (**p_xi + TAMX_CARTA y p_yi + TAMY_CARTA**).
 - **if posx > p_xi and posx < p_xf and posy > p_yi and posy < p_yf:** Verifica si el clic ocurrió en el área del mazo, se utiliza una comparación doble para verificar que el valor de posx esté entre p_xi (límite izquierdo) y p_xf (límite derecho), y que el valor de posy esté entre p_yi (límite superior) y p_yf (límite inferior). Si estas condiciones son verdaderas, significa que el usuario ha hecho clic en el área del mazo, por lo que el método retorna MAZO (una constante que indica el mazo) y el objeto self.maz (**return MAZO, self.maz**), que representa el mazo en el juego.
-
- **p_xi y p_yi:** Son variables que almacenan las coordenadas iniciales del vértice superior izquierdo del área de las cartas mostradas (MOSTRADA_POSX, MOSTRADA_POSY)
 - **p_xf y p_yf:** Son variables que almacenan las coordenadas del vértice inferior izquierdo del área de las cartas mostradas. Y se calcula sumando las dimensiones de una carta a las coordenadas originales (p_xi + TAMX_CARTA y p_yi + TAMY_CARTA).

- **if posX > p_xi and posX < p_xf and posY > p_yi and posY < p_yf:** Similar al bloque anterior, verifica si el clic está dentro de los límites definidos para las cartas mostradas. Y si sí lo están, se retorna una constante que indica el área (MOSTRADAS) y “None” que significa que no necesita devolver ningún objeto específico (return MOSTRADAS, None).

```

414         for pilaact in range(7):
415             carta = -1
416             p_xi = PILAS_XINICIAL + (pilaact*DISTX_PILAS)
417             p_xf = p_xi + TAMX_CARTA
418             p_yi = PILAS_YINICIAL
419             p_yf = p_yi + TAMY_CARTA + (DISTY_PILAS*pilaact)
420             if(posx > p_xi and posX < p_xf and posY > p_yi):
421                 return PILA, pilaact
422         for pila_subir in self.pilas_subir:
423             p_xi = pila_subir.posx
424             p_xf = p_xi + TAMX_CARTA
425             p_yi = pila_subir.posy
426             p_yf = p_yi + TAMY_CARTA
427             if(posx > p_xi and posX < p_xf and posY > p_yi and posY < p_yf):
428                 return PILASUBIR, pila_subir
429         return -1, -1

```

- **for pilaact in range(7):** El bucle itera sobre las 7 pilas (desde pilaact = 0 hasta pilaact = 6).
- **p_xi:** Variable que almacena el calculo de la separación entre pilas (**PILAS_XINICIAL + (pilaact * DISTX_PILAS)**). Calcula la posición inicial de la primera pila, más un desplazamiento basado en el número de la pila actual, el desplazamiento de calcula (**pilaact * DISTX_PILAS**).
- **p_xf:** Variable que almacena el extremo derecho de la pila actual, calculado sumando el ancho de una carta por p_xi (**p_xi + TAMX_CARTA**).
- **p_yi:** Variable que almacena la posición inicial vertical fija para todas las pilas.
- **p_yf:** Variable que almacena el extremo inferior de la pila (**p_yi + TAMY_CARTA + (DISTY_PILAS * pilaact)**). Añade una cantidad proporcional al número de pila actual (pilaact) para acomodar el apilamiento vertical de las cartas en cada pila. La separación vertical entre cartas en una pila está determinada por **DISTY_PILAS**.
- **if posX > p_xi and posX < p_xf and posY > p_yi:** Verifica si clic ocurrió en alguna de las 7 pilas dentro del área definida por las coordenadas (**p_xi, p_xf, p_yi y p_yf**). Si se cumple esta condicional de que el clic cae dentro del área de

una pila, se retorna una pila de cartas, y el número de la pila actual (**return PILA, pilaact**) . Así identifica en qué pila ocurrió el clic.

- **for pila_subir in self.pilas_subir:** Itera sobre las pilas de subida.
- **p_xi y p_yi:** Variables que almacenan las coordenadas iniciales (superior izquierda) del área de la pila (pila_subir.posx, pila_subir.posy).
- **p_xf y p_yf:** Variables que almacenan las coordenadas del vértice inferior derecho, calculándolo con la suma de las variables anteriores y el tamaño de las cartas (p_xi + TAMX_CARTA, p_yi + TAMY_CARTA).
- **if posx > p_xi and posx < p_xf and posy > p_yi and posy < p_yf:** Verifica si el clic ocurrió dentro del área de alguna de las pilas de subida. Si lo fue, retorna una constante que indica una pila de subida, y la pila específica en donde ocurrió el clic (return PILASUBIR, pila_subir).
- **return -1, -1:** Si el clic no ocurrió en ninguna de las áreas definidas (mazo, cartas mostradas, pilas, o pilas de subida), el método retorna -1, -1, indicando que el clic no se realizó en ninguna zona interactiva del tablero.

Método matchable:

Esté método toma dos cartas, y verifica si es posible colocar el segundo parámetro sobre la carta del primer parámetro, siguiendo las reglas del juego.


```
431     def matchable(self, c1, c2):
432
433         alternancia_colores = (c1.color != c2.color)
434
435
436         valor_decreciente = (c1.numero == c2.numero - 1)
437
438         return alternancia_colores and valor_decreciente
439
```

- **alternancia_colores = (c1.color != c2.color):** Esta variable almacena la verificación de que las cartas recibidas por parámetros sean de diferente color (negro o rojo) para comprobar que una se pueda colocar encima de la otra.
- **valor_decreciente = (c1.numero == c2.numero - 1):** Esta variable almacena la verificación de si el valor numérico de c1 es igual al valor de c2 menos uno, es decir, si c2 es un número menor que c1 para poder colocar la carta encima.

- **return alternancia_colores and valor_decreciente:** Retorna el resultado de combinar ambas condiciones anteriores, es decir, ambas deben ser verdaderas para que retorne True y pueda colocar la carta encima de la otra. Si no se cumple "False", no se podrá colocar.

Main:

Este bloque de código ejecuta el juego.

```
441  if __name__ == "__main__":  
442     juego = Juego()  
443     juego.on_execute()
```

- **if __name__ == "__main__":** Este bloque condicional se usa para definir qué partes del código se deben ejecutar cuando el archivo es ejecutado directamente. Cuando el archivo es .py entonces "__name__" será igual a "__main__".
- **juego = Juego():** Se crea un objeto de la clase juego que inicializa todos los elementos que contiene la clase.
- **juego.on_execute():** Por medio del objeto "juego" creado se llama al método que inicia el ciclo del juego, permitiendo al jugador interactuar.