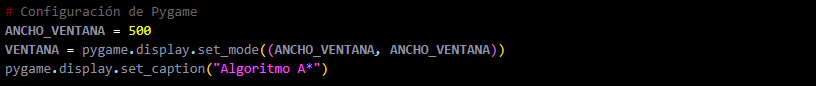
**1. Importar Pygame**

El primer paso es importar Pygame, que es una biblioteca para la creación de videojuegos en Python, además de otra librería que maneja colas de prioridad usando montículos (heaps) mínimos eficientes.



**2. Configuraciones iniciales**

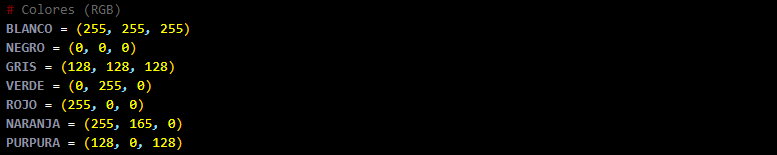
Se configuran la ventana y los colores que se usarán en la cuadrícula. `ANCHOVENTANA` define el tamaño de la ventana, y `VENTANA` se usa para crear la ventana donde se dibujarán los elementos:



* **`ANCHOVENTANA`**: Define que la ventana será de 500x500 píxeles.
* **`pygame.display.setmode`**: Crea la ventana con las dimensiones dadas.
* **`pygame.display.setcaption`**: Establece el título de la ventana.

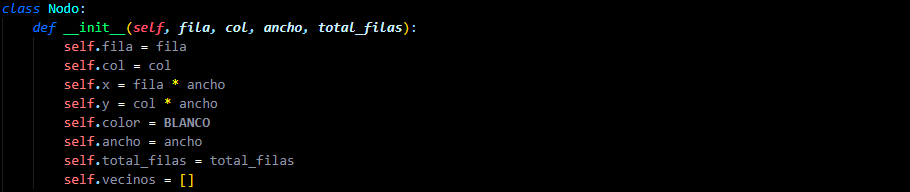
**3. Definir colores en formato RGB**

Definimos algunos colores que se usarán para los nodos y las paredes. Los colores se expresan en el formato RGB



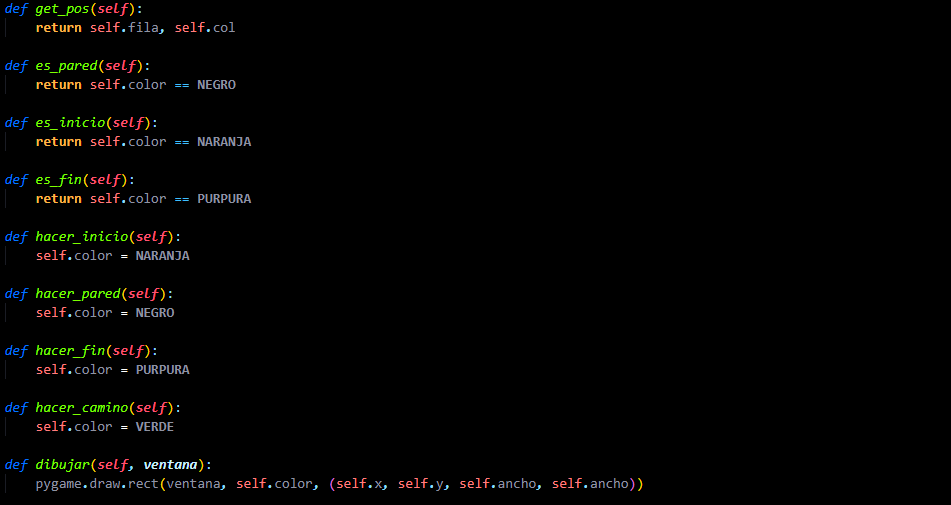
**4. Clase Nodo**

La clase `Nodo` representa cada celda de la cuadrícula. Cada nodo tiene una posición (fila y columna), un color, y puede tener diferentes estados (inicio, fin, pared, etc.):



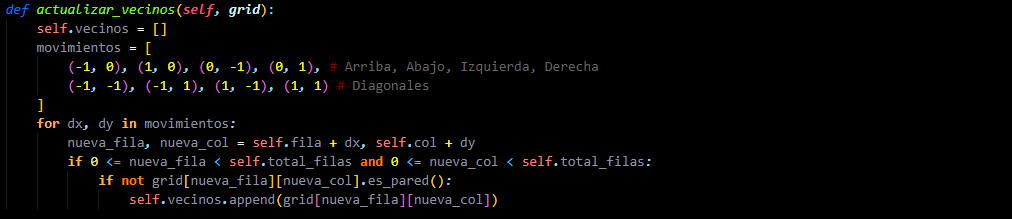
Esta clase contiene varios métodos útiles:

* **`getpos()`**: Devuelve la posición del nodo.
* **`espared()`, `esinicio()`, `esfin()`**: Determinan el estado del nodo.
* **`hacerpared()`, `hacerinicio()`, `hacerfin()`**: Cambian el estado del nodo.
* **`dibujar()`**: Dibuja el nodo en la ventana.



**5. Método Vecinos**

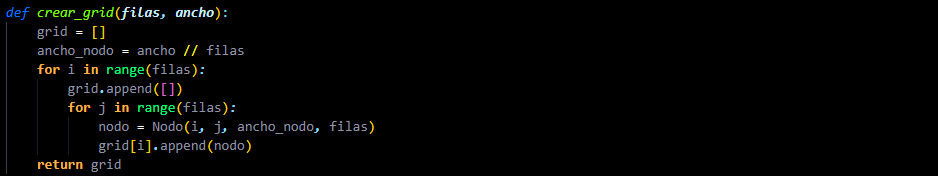
Este método actualiza la lista de vecinos accesibles (que no son paredes) del nodo actual. Considera las ocho direcciones posibles (arriba, abajo, izquierda, derecha y las diagonales). Sirve para determinar qué nodos se pueden visitar desde el nodo actual durante la ejecución de un algoritmo de búsqueda (como A\* o Dijkstra).



* self.vecinos = []: Inicializa la lista de vecinos vacía.
* movimientos: Define las 8 posibles direcciones alrededor del nodo actual.
* Se recorren los movimientos para calcular las coordenadas de cada vecino posible.
* Se verifica que las nuevas coordenadas estén dentro de los límites de la cuadrícula.
* Se descarta cualquier vecino que sea una pared (es\_pared()).
* Los vecinos válidos se agregan a la lista self.vecinos.

**6. Crear la cuadrícula**

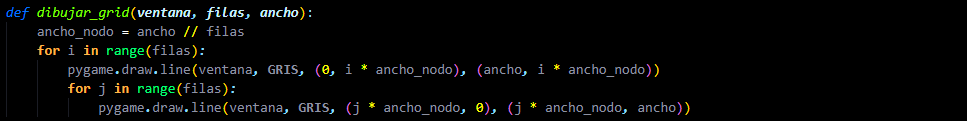
El método `creargrid` crea una lista de nodos organizados en una cuadrícula. Cada nodo tiene un tamaño calculado dividiendo el ancho total de la ventana por el número de filas.



Este método crea una lista bidimensional que representa la cuadrícula de nodos.

**7. Dibujar la cuadrícula y los nodos**

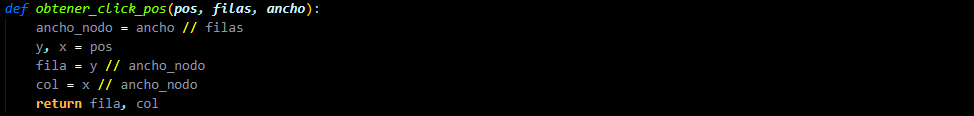
El método `dibujargrid` dibuja las líneas que separan los nodos en la cuadrícula, mientras que `dibujar` dibuja cada nodo en la ventana.



* **`dibujargrid`**: Dibuja las líneas horizontales y verticales para formar la cuadrícula.
* **`dibujar`**: Llama al método `dibujar()` de cada nodo para mostrar su estado (color).

**8.  Obtener la posición del clic**

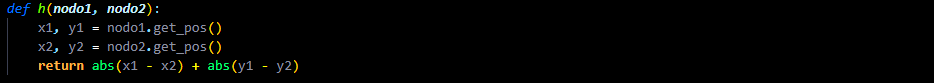
El método `obtenerclickpos` convierte la posición en píxeles del ratón a la posición de la cuadrícula (fila y columna).



Esto permite identificar qué nodo fue clicado con precisión.

**9. Calcula la distancia**

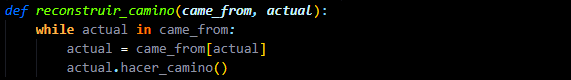
Esta función calcula la **distancia heurística** entre dos nodos utilizando la **distancia de Manhattan**, que es adecuada para movimientos en una cuadrícula.



* get\_pos(): Se obtienen las coordenadas (fila, columna) de cada nodo.
* abs(x1 - x2) + abs(y1 - y2): Se calcula la distancia de Manhattan, que mide la cantidad total de movimientos verticales y horizontales requeridos para llegar de un nodo a otro, sin considerar diagonales.

**10. El camino más corto**

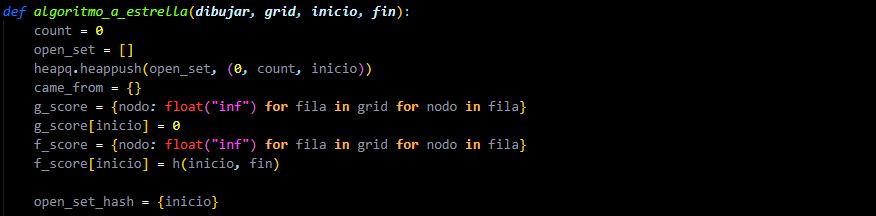
Esta función reconstruye el camino más corto desde el nodo final hasta el nodo inicial una vez que el algoritmo de búsqueda (como A\*) ha terminado.



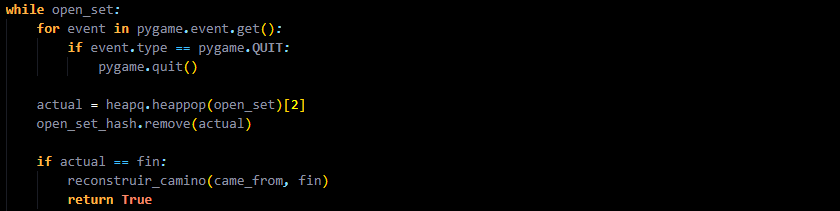
* came\_from: Es un diccionario que guarda la relación entre cada nodo y el nodo del cual fue visitado.
* actual: Es el nodo final (el objetivo alcanzado por el algoritmo).
* while actual in came\_from: Recorre el diccionario hacia atrás desde el nodo final hasta el nodo de inicio.
* actual = came\_from[actual]: Retrocede un paso en el camino.
* actual.hacer\_camino(): Marca ese nodo como parte del camino encontrado, normalmente cambiando su color (por ejemplo, a verde).

**11. Implementación del algoritmo – Evaluación de vecinos**

Esta función implementa el algoritmo **A\***, una técnica de búsqueda heurística que encuentra el camino más corto desde un nodo inicial hasta un nodo final en una cuadrícula.



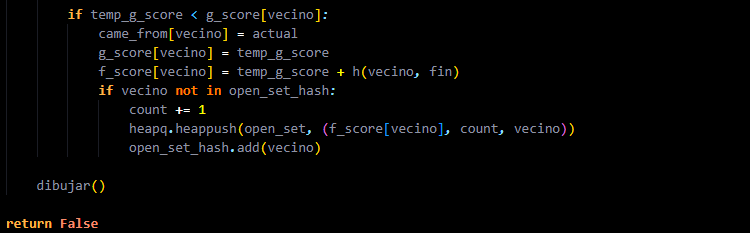
* open\_set: Cola de prioridad donde se guardan los nodos a evaluar.
* heapq: Se usa para ordenar los nodos por su f\_score (estimación del costo total).
* came\_from: Guarda el camino óptimo recorrido.
* g\_score: Costo real desde el nodo inicial hasta cada nodo. Inicialmente es infinito.
* f\_score: Estimación del costo total desde el inicio hasta el objetivo pasando por ese nodo (g\_score + heurística).
* open\_set\_hash: Conjunto auxiliar para verificar más rápido si un nodo ya está en la cola.



* Se mantiene activo mientras haya nodos por evaluar.
* Permite salir de la aplicación si se cierra la ventana (pygame.QUIT).
* Se extrae el nodo con menor f\_score de la cola.
* Se elimina de la estructura auxiliar.

**11.1 Implementación del algoritmo - Evaluación de vecinos**

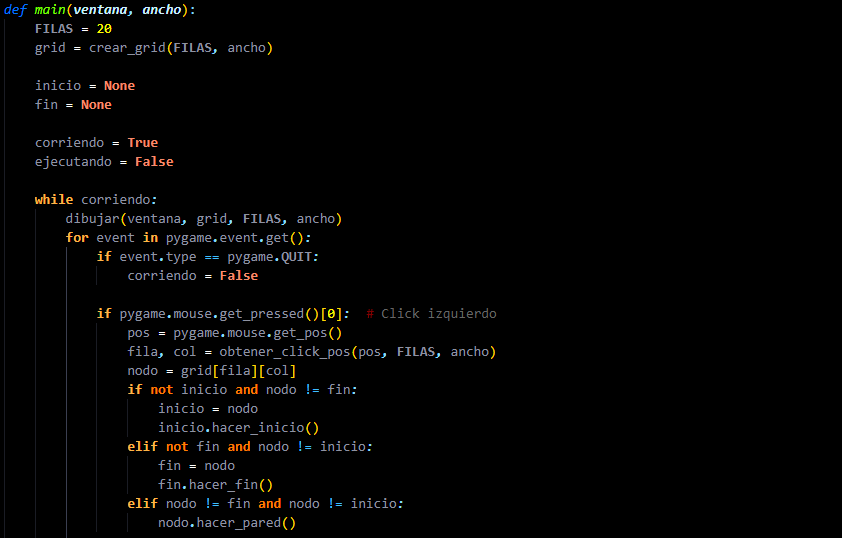
Si el nodo actual es el nodo final, se reconstruye el camino y se termina.

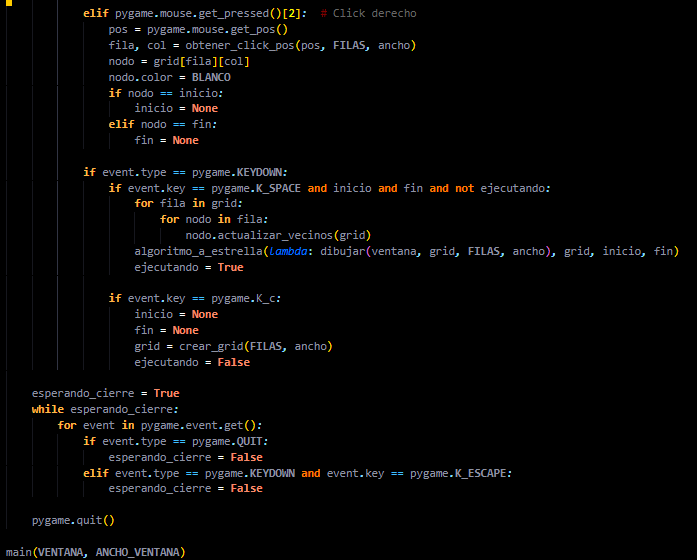


* Para cada vecino, calcula un nuevo g\_score.
* Si ese camino es mejor que el anterior, actualiza las puntuaciones y agrega el nodo a la cola si no está.
* count sirve para desempatar nodos con igual f\_score.
* Se actualiza la interfaz gráfica en cada paso para mostrar visualmente el progreso.
* Si se termina el bucle sin encontrar el nodo final, no existe camino posible.

**12. Ciclo principal**

El ciclo principal del programa está en la función `main`, que controla la interacción del usuario. Escucha eventos como el cierre de la ventana o clics del ratón para definir los nodos de inicio, fin, y las paredes.





Interacción del ratón:

* Clic izquierdo: Define el nodo de inicio, el nodo final, o establece una pared.
* Clic derecho: Restablece el nodo (lo borra).
* Finalmente, la función `pygame.quit()` se encarga de cerrar correctamente el programa.