Desafío picante

Wordle

Introducción

El Wordle es un juego que consiste en adivinar una palabra en un máximo de seis intentos. En la implementación original y más popular del juego[1], la palabra a ser adivinada es de cinco letras y se actualiza una vez por día.

Para adivinar la palabra secreta diaria (`target`), debemos sugerir palabras (`input`) de forma tal de ir ganando información a lo largo de nuestros sucesivos intentos. Cuando sugerimos una palabra, Wordle le asigna un valor distinto a cada letra según el grado de coincidencia entre nuestra sugerencia y la palabra secreta. Tomemos como ejemplo de palabra secreta a "alias". Para cada letra de una dada palabra sugerida:

- Si la letra no está en la palabra secreta, es marcada en gris (podríamos codificarla como un -1 en nuestro programa). *Ejemplo:* La "g" en "**algas**".
- Si la letra está en la palabra secreta y se encuentra en la misma posición que ocupa en nuestra sugerencia, es resaltada en verde (1). *Ejemplo:* La "a" en "**audio**".
- Si la letra está en la palabra secreta pero está en una posición distinta a la que ocupa en nuestra sugerencia, es resaltada en amarillo (0). *Ejemplo:* La segunda "a" en "**alada**".

Aprovechemos este último ejemplo para notar que la tercera "a" en "**alada**" no es resaltada en amarillo. El `input` es leído de izquierda a derecha, y al llegar a la tercera "a" ya no hay más necesidad de esta letra para reconstruir al `target`, que solo tiene dos letras "a". Además, las letras bien ubicadas (verde, 1) tienen prioridad sobre las demás. Por ejemplo, para el `target` "avispado", la tercera "a" del `input` "**albahaca**" tiene prioridad sobre la segunda por estar bien posicionada.

Vamos a implementar nuestro propio Wordle, sin simplificarlo en absoluto (salvo en lo que refiere a la interfaz de usuario, que por supollo queda de tarea).

A las palabras candidatas a ser `target` e `input` las vamos a leer del archivo **diccionario.txt**, el mismo que se usa para programar el scrabble. Es recomendable hacer una exploración rápida del archivo (puede ser algo tan simple como mirarlo un poco en un editor de texto) antes de seguir con los ejercicios.

Ejercicios

1. Inicializar las variables que nos son relevantes: `n_letras`, `n_intentos` y `abecedario` (string de letras del abecedario español, en lowercase). Para `abecedario` probar lo siguiente:

```
from string import ascii_lowercase
abecedario = ascii_lowercase
```

Importante: `ascii_lowercase` no incluye a la "ñ" ni a ninguna otra letra decorada (tildes, diéresis, etc.). Por ahora, solo agregar "ñ" a `abecedario` (no importa la posición).

2. Usar `diccionario = open("diccionario.txt").read().splitlines()` para leer el archivo, convertir su contenido en una lista de palabras (strings) y almacenarla en `diccionario`. Opcionalmente, se puede usar el módulo `os` (siempre con muchísimo cuidado) para modificar el directorio de trabajo y abrir el archivo.

Ahora que tenemos definidas todas las variables que necesitamos para jugar, vamos a programar la funcionalidad del Wordle paso por paso.

[1] Wordle, por Josh Wardle

3. Vamos a querer tener una función que nos diga si una dada palabra del diccionario se ajusta a nuestras necesidades de tamaño y caracteres. Implementar la función `se_forma(palabra, abecedario, n_letras)`, que devuelve `True` si el string `palabra` es de largo `n_letras` y además no contiene ningún caracter extraño a nuestro `abecedario`, o `False` si no cumple alguna o ninguna de estas condiciones.

4. Implementar la función `generar_subdiccionario(diccionario, abecedario, n_letras)`, que devuelve una lista `subdicc` con las palabras de `diccionario` que tienen `n_letras` de largo y que no contienen caracteres extraños. Notar que el archivo diccionario.txt contiene nombres propios, pero por nuestra definición de `abecedario` estas palabras nunca formaran parte de un subdiccionario (lo cual es muy bueno, porque no queremos nombres propios en un juego como este). Notar, también, que como ninguna palabra del diccionario contiene tildes (¿Cómo lo demostrarías?) no necesitamos preocuparnos porque nuestro abecedario no contenga letras con tilde.

5. Cuando el programa reciba una palabra como input, deberá procesarla para estandarizarla de acuerdo a la información que tenemos sobre nuestro diccionario. Definir `procesar(palabra)`, que convierte `palabra` a lowercase y reemplaza las vocales con tildes por sus versiones sin tilde. Por ejemplo, `print(procesar("Tildé"))` debe imprimir "`tilde`".

6. Hablando de inputs, vamos a escribir una función que nos sirva para recibirlos. Implementar `pedir_palabra()`. La función debe usar `input(...)` para recibir input del jugador, luego debe convertirlo a string y procesarlo para remover tildes y mayúsculas. Devuelve el input procesado.

7. Ahora armemos una función que nos diga si un input (procesado) es válido. Por ahora, un input válido será aquel que no contenga caracteres extraños y esté en el subdiccionario que estemos usando (notar que, implicitamente, estamos verificando que el input sea del largo adecuado). Implementar la función `es_valido(input, subdicc, abecedario)` que devuelve `True` si el input es válido y `False` si no lo es.

8. Importar el módulo `random` e implementar `palabra_target(subdicc)`, que elige al azar y devuelve una palabra de la lista `subdicc`. Vamos a usar esta función para elegir nuestra palabra secreta.

9. Llegó el desafío. Definir `comparar(target, input)`, que toma como parámetros una palabra `target` y un `input` válido y devuelve una lista `comparacion` en la que cada elemento solo puede valer `-1`, `0` o `1` y refiere a la codificación comentada en la introducción de este desafío. Se permite la implementación y uso de todas las funciones auxiliares que puedan parecer necesarias, pero no se permite el uso de bibliotecas como `numpy` o `itertools`. *Ejemplos:*

```
print(comparar('xxxxx', 'xxxxx'), # [ 1,  1,  1,  1,  1]
      comparar('xxxxx', 'yyyyy'), # [-1, -1, -1, -1, -1]
      comparar('xxyzz', 'zxxwx'), # [ 0,  1,  0, -1, -1]
      comparar('xyyyx', 'xxyxx')) # [ 1, -1,  1, -1,  1]
```

Sugerencia: Implemente un contador de letras por palabra que pueda ser dinámicamente modificado, para llevar la cuenta de las letras de `target` que ya fueron "resaltadas" en `input`.

10. Respiremos un poco. Después de que hayamos comparado a la sugerencia del jugador con la palabra secreta, debemos mostrar el resultado. Definir `mostrar_pistas(input, comparacion)`, que imprime en consola al `input` y, debajo, imprime alguna representación medianamente intuitiva de la codificación realizada por `comparar(...)`. Como sugerencia pueden usarse "_", "–" y "+" para representar a los valores `-1`, `0` y `1`, respectivamente. *Ejemplo:*

```
mostrar_pistas('patos', comparar('casio', 'patos'))
# patos
# _+_-
```

11. Definir `jugar_intento(target, subdicc, abecedario)`, que simula uno de los `n_intentos` intentos del jugador. Esta función debe, en orden:

1. Recibir un input del jugador.
2. Verificar la validez del input.
3. Si el input no es válido, debe comunicarle este infortunio al jugador (imprimiendo algún mensaje alentador en pantalla) y debe pedirle un nuevo input. Esto debe repetirse hasta que se obtenga un input válido.
4. Obtenido un input válido, realizar la comparación entre `target` y el input.
5. Mostrar el resultado de la comparación en consola con `mostrar_pistas(...)`.
6. Devolver `True` si el jugador ganó (el input es exactmente igual a `target`), `False` si el show debe continuar.

12. Finalmente definir `jugar_partida(n_letras, n_intentos, diccionario, abecedario)`. Esta función debe, combinando toda la funcionalidad escrita hasta ahora, simular una partida completa. La función debe generar un subdiccionario de acuerdo a `n_letras`, debe elegir una palabra random a actuar como target, y luego debe iterar `jugar_intento(...)` como máximo `n_intentos` veces. Al comienzo de un nuevo intento, se debe imprimir el número del intento actual en consola. Si el jugador logra adivinar la palabra secreta antes de superar el número máximo de intentos, la función imprime un mensaje bonito y devuelve `True`. Si al jugador se le acaban los intentos imprime algo así como "verás los malbones crecer desde abajo"(el Wordle se puede poner picante), le revela la palabra target al jugador, y devuelve `False`.

Y listo, tenemos un Wordle bueno, bello y barato. Pero todavía no es el Wordle prometido, el de las antiguas historias al cual han hecho referencia en sus jeroglíficos centenares de culturas de manera independiente a lo largo de la breve historia de la humanidad: el Wordle no simplificado. Para les valientes, los optativos. Para les no valientes, bueno, vamo a jugá'.

13. Repeticiones (optativo) Vamos a complejizar un poco la definición que dimos de input válido (ver ejercicio 7.) para que, además de todo lo que ya se ha dicho, un input válido sea un input no repetido. Esta no es una funcionalidad presente en la implementación más popular de Wordle, pero no tiene por qué no serlo. Así que agreguemosla.

Modificar `es_valido(...)` para que reciba un parámetro adicional, la lista `historial`, que contenga los inputs previos sugeridos por el jugador durante la partida. `es_valido(...)` debe devolver `True` solo si, además de cumplirse todos los demás requisitos, el input evaluado no está presente en `historial`. La lista debe inicializarse como lista vacía en `jugar_partida(...)` y debe actualizarse luego de cada intento, es decir, luego de que el programa reciba un input válido. **Sugerencia:** Actualizar `historial` *in-place* dentro de `jugar_intento(...)`.

14. Mensajes de error (optativo) Hay muchas razones por las cuales un input puede ser no válido, pero `jugar_intento()` tiene un único mensaje de error. Modificar `es_valido(...)` para que devuelva una lista de 4 booleanos, uno para cada condición de validez: largo adecuado, caracteres no extraños, pertenencia al subdiccionario y no repetición. Actualizar `jugar_intento(...)` para que el mensaje de error ante un input no válido dependa de cual de las condiciones de validez fue incumplida. Dependiendo de cómo hayas programado `es_valido(...)`, puede que tengas que modificar otras funciones.

15. Palabra diaria (optativo) La implementación más popular de Wordle ofrece una palabra secreta por día. En una fecha dada, esta palabra es la misma para cualquier persona que se meta a jugar desde cualquier computadora y desde cualquier parte del mundo. Para implementar esta dinámica en nuestro programa, vamos a necesitar que el accionar de `palabra_target(...)` (o más precisamente, el accionar de la función del modulo `random` llamada dentro de esta función) quede determinado por algún

identificador del día actual (como podría ser la cantidad de días transcurridos desde el 1 de Enero del año 1 D.C.). Además, la relación entre el indentificador del día actual y la elección de `palabra_target(...)` debe ser impredecible: en caso contrario, sería sencillo diseñar un algoritmo que prediga la palabra secreta del día actual o de un día futuro. Podríamos diseñar nuestro propio algoritmo para generar identificadores diarios gigantes y armar una función que haga muchas cosas raras y caóticas con estos identificadores para eventualmente devolver un elemento "random" de nuestro subdiccionario, pero vamos a ir por una alternativa un poco más sencilla usando `random.seed()`. Para obtener un identificador único del día actual, podemos usar el modulo `time` para acceder al tiempo Unix[2]. Probar:

```
import time
UTC_delta = - 3*60*60 # Buenos Aires está en la zona horaria UTC-3
semilla   = int(time.time() + UTC_delta)//(60*60*24) # Id. del día
```

Inicializar la variable booleana `diaria` junto al resto de las variables inicializadas en el ejercicio 1. Definir la función `semilla_diaria(diaria)`, que calcule al identificador del día actual `semilla` y llame a `random.seed(semilla)` si `diaria == True` o a `random.seed()` si no lo es. Finalmente, agregar el parámetro `diaria` a la función `jugar_partida(...)` y modificar la función para que use `semilla_diaria(...)` antes de llamar a `palabra_target(...)`.

16. Dificultad alta (optativo) Volvió el desafío. El Wordle ofrece una dificultad "alta" en la cual el jugador está obligado a usar, para su siguiente input, las pistas que tiene disponibles: si para una palabra secreta dada se encontró la identidad y posición de una letra (resaltada en verde, o codificada con un 1 dentro de nuestro programa), inputs siguientes deben necesariamente contener a esa letra en la posición que le corresponde. Si se determinó la identidad pero no la posición de una letra (amarillo, 0), inputs siguientes deben contener a esa letra en alguna posición, tantas veces como veces haya sido resaltada en amarillo en el input previo. De esta manera, los inputs válidos quedan aún más restringidos. Notar que no se prohibe el reuso de letras sobre las cuales es sabido que no pertenecen a la palabra secreta (gris, -1).

Inicializar la variable booleana `dificil` junto al resto de las variables inicializadas en el ejercicio 1. Implementar la función `usa_pistas(target, input, prev_input)`, que recibe el `target`, el `input`, y el input inmediatamente anterior al actual y devuelve `True` si se usaron las pistas disponibles o `False` en caso contrario. *Ejemplos:*

```
print(usa_pistas('yyyyy', 'xxxxx', 'xxxxx'), # True
      usa_pistas('zxvyu', 'ywxzw', 'yxwww'), # False
      usa_pistas('yyyyy', 'yyxyy', 'xxyxx'), # False
      usa_pistas('zxywv', 'wzxyw', 'xyxzw')) # True
```

Agregar el parámetro `dificil` a las funciones `es_valido(...)`, `jugar_intento(...)` y `jugar_partida(...)`. Si `dificil == True`, el resultado de `usa_pistas(...)` debe incorporarse a la definición de validez en `es_valido(...)`. Además, debe agregarse un nuevo mensaje de error en `jugar_intento(...)` que contemple la posibilidad de que un input no sea válido por no cumplir con esta nueva condición de validez. Hacé cualquier otra modificación a `es_valido(...)`, `jugar_intento(...)` y`jugar_partida(...)` que te sea necesaria para implementar esta nueva dificultad. **Sugerencia:** Hacé uso del parámetro `historial` para obtener el input previo al actual.

[2] Tiempo Unix, Wikipedia