

Práctico integrador - Word Game

Taller de Programación en Python y Aplicaciones
Curso formativo dentro del marco del Argentina Programa 4.0

Basado en Problem Set 3: The 6.00/6.0001 Word Game - MIT

Introducción

El juego es similar al Scrabble, pero para un solo jugador. Al jugador se le asignan letras con las que puede formar una o más palabras. Si cada palabra es correcta, el jugador suma puntos. La puntuación se calcula de acuerdo a la longitud y a las letras usadas en cada palabra. A continuación, se enuncian las reglas del juego.

Reparto

- Un jugador recibe una mano con `TAMANIO_MANO` letras del alfabeto. Estas letras son elegidas al azar y, puede que estén repetidas.
- El jugador puede formar las palabras que desee con su mano, pero usará cada letra una única vez.
- No es necesario utilizar todas las letras; sin embargo, el puntaje de una palabra se ve afectado por la cantidad de letras que quedan en la mano luego de ingresarla al juego.

Puntuación

- El puntaje de la mano es la suma del puntaje de cada palabra formada.
- El puntaje de una palabra es el producto de dos componentes:
 - Primer componente: la suma de los puntos de las letras usadas en la palabra. Similar al Scrabble, cada letra tiene un puntaje: A vale 1, B vale 3, etc.. Los puntajes están definidos dentro del proyecto en `VALORES_LETRAS`.
 - Segundo componente: el máximo valor entre 1 y la siguiente formulación:
$$[7 * longitud_palabra - 3 * (n - longitud_palabra)]$$
donde `longitud_palabra` es la cantidad de letras usadas en la palabra y `n` es la cantidad de letras disponibles en la mano actual (antes de jugar la palabra).

Ejemplos de puntuación

Suponemos $n = 6$ y las siguientes letras en la mano `['o', 'i', 'l', 'm', 'c', 'g']`. Jugar la palabra “mico” resultará en 176 puntos: $(3+1+3+1) * (7*4-3*(6-4)) = 176$. El primer término, es la suma de los valores de cada letra usada. El segundo término, es el cálculo especial que recompensa al jugador por elegir una palabra con muchas letras, y lo penaliza por las letras no usadas. Otro ejemplo, con $n = 7$ y una mano similar `['o', 'i', 'l', 'm', 'c', 'g', 'a']`. Jugar la palabra “mi” resulta en 4 puntos: $(3+1) * 1 = 4$. El segundo componente es 1; porque $(7*2) - 3*(7-2) = -1$ y elijo el máximo.

Desarrollo

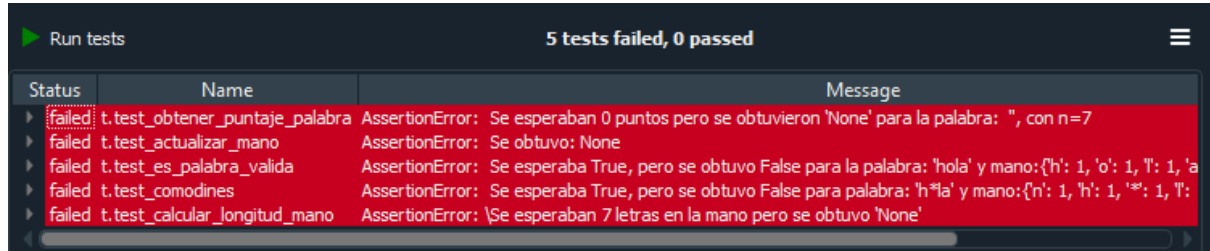
Ambiente de trabajo

Recomendamos utilizar Spyder debido a su simpleza al ejecutar testeos de unidad. Spyder incorpora un plugin que permite visualizar el resultado de estos testeos como parte de la interfaz del entorno. También, es simple definir nuevos casos de test o testeos de unidades para las funciones deseadas.

El proyecto se llama WordGame.zip y contiene:

- **WordGame.py**: un script Python que incluye un conjunto de funciones y datos para comenzar a desarrollar la solución al problema. Todo nuestro código deberá ser incorporado a este archivo.
- **test_WordGame.py**: utilizando la simplicidad que nos brinda pytest se construyó un conjunto de testeos de unidad para validar el correcto funcionamiento de las funciones desarrolladas en WordGame.py.
- **palabras.txt**: un conjunto de palabras válidas en castellano (en minúsculas y sin acentos).
- **words.txt**: un conjunto de palabras válidas en inglés (en minúsculas).

Abriendo el proyecto, y ejecutando los testeos de unidad definidos en test_WordGame.py deberíamos obtener el siguiente resultado. Esto nos permite validar que el proyecto fue ejecutado correctamente y que contamos con el plugin de pytest instalado correctamente en Spyder.



Si ejecutamos el script WordGame.py veremos el siguiente resultado:

Cargando lista de palabras desde el archivo...

80383 palabras cargadas.

jugar_partida no implementado.

Al inicio, el código carga la lista de palabras válidas desde un archivo llamado ARCHIVO_PALABRAS. Luego, llama a la función jugar_partida (aún no implementada). Si aparece una excepción de tipo IOError (es decir, no existe el archivo o directorio), deberemos validar que el archivo de texto palabras.txt esté en el mismo directorio que WordGame.py.

En el script WordGame.py hay un número de funciones ya implementadas que se podrán usar para construir la solución al problema.

Metodología de trabajo

Vamos a estructurar la resolución al problema mediante la construcción de un conjunto de funciones por separado. Luego, una vez validado el correcto funcionamiento de cada función, deberemos unir las para construir la lógica del juego completo. Este enfoque sigue los lineamientos de los test de unidad.

Para esto se plantean una serie de problemas donde cada uno aborda un aspecto particular del juego. En cada función a implementar hay comentarios de ayuda para intentar resolver el problema planteado. Después de cada ajuste, es recomendable ejecutar el test de unidad para verificar si se cumplió con el objetivo propuesto. Estos tests no son exhaustivos, es deseable que también se realicen otras pruebas (probablemente con otros datos de entrada).

El primer paso será implementar el esqueleto del ciclo de control principal del juego, comúnmente llamado **ciclo de juego**. El ciclo de juego comprende toda la acción del juego en sí. Una vez comenzado, se ejecutará hasta que se cumpla alguna condición particular que dé por finalizado el juego. Luego del ciclo de juego generalmente viene una etapa de presentación de resultados, donde se informa al jugador el motivo de la finalización del juego y su performance durante el mismo.

Problema 1: Construir el ciclo del juego

Como mencionamos anteriormente el primer problema con el que nos vamos a enfrentar es construir el esqueleto del ciclo de juego. ¿Por qué hablamos de esqueleto? Porque vamos a implementar completamente el ciclo de control del juego pero sin incorporar la lógica concreta que se debe llevar adelante en cada etapa del mismo.

Para este primer problema nos conformamos con visualizar por pantalla, mediante mensajes al usuario, en qué etapa se encuentra el juego a medida que el ciclo de juego se va ejecutando.

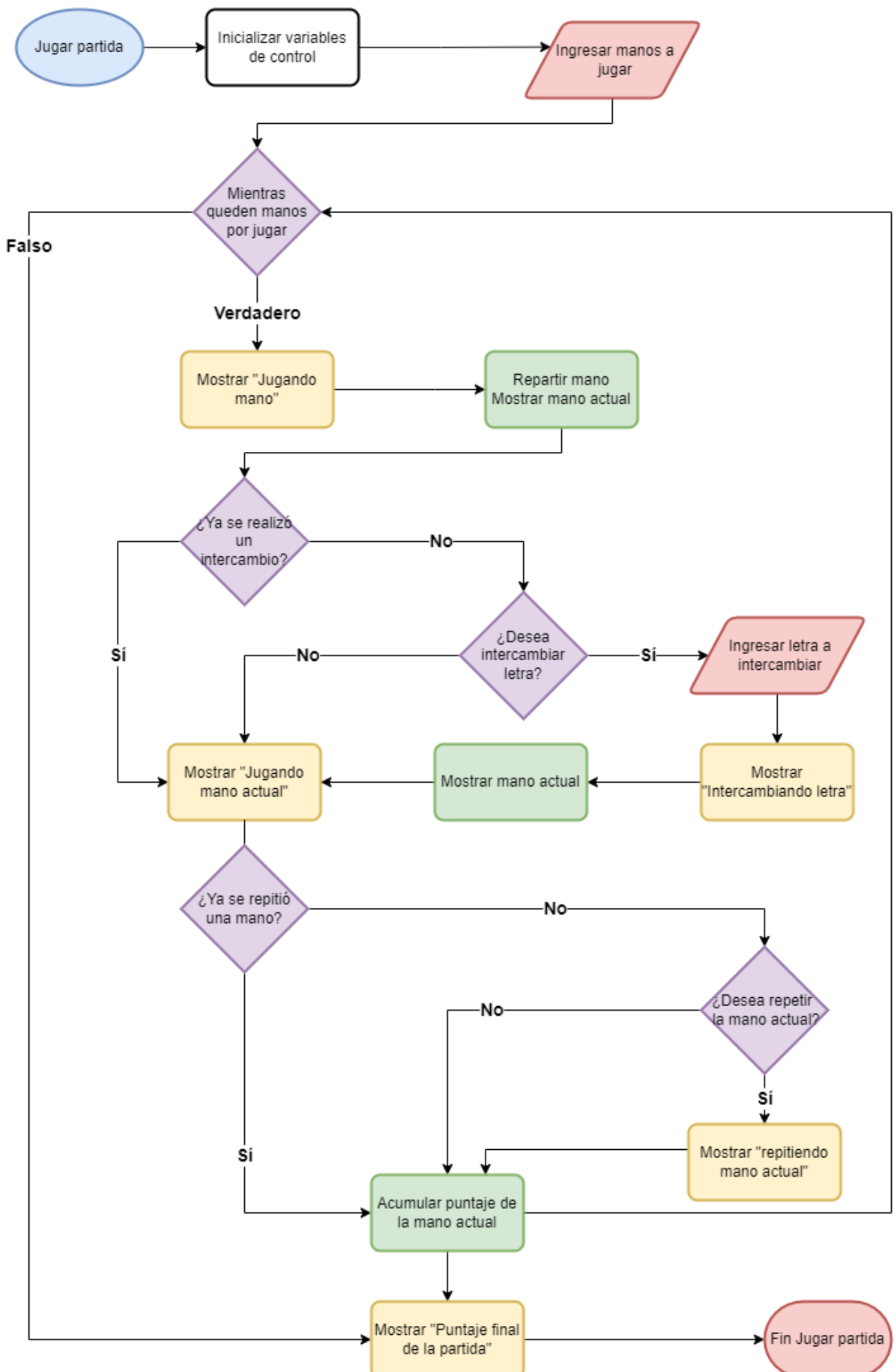
Este esqueleto será implementado en la función `jugar_partida()` utilizando como guía el siguiente diagrama de flujo.

Referencias:



Aclaración: En la resolución del problema 1 los procesos que forman parte de la lógica del juego se incluyen en el diagrama, para tener una idea completa del ciclo de juego, pero no quedarán implementados. Luego, a medida que avancemos en la resolución del juego iremos implementando las funciones específicas que permitirán resolver dichos procesos.

Se puede dar por resuelto este problema cuando, al realizar varias pruebas cambiando las opciones que se nos solicitan, se logran visualizar por pantalla los mensajes al usuario establecidos en el diagrama de flujo.



Problema 2: Puntuación de palabras

Implementar la función `obtener_puntaje_palabra(palabra, n)`.

El primer paso es implementar una función que calcule la puntuación para una palabra de acuerdo a las especificaciones de la función.

Recordatorio/pistas: Revisar las reglas de los puntajes. Usar el diccionario `VALORES_LETRAS` definido al principio de `WordGame.py`. No asumir que siempre habrá 7 letras en la mano. El parámetro `n` es el número total de letras en la mano cuando se ingresa la palabra. Considerar que el usuario puede construir la palabra usando letras en mayúscula y minúscula.

Testeo: si esta función es implementada correctamente, y se corre `test_WordGame.py`, el `test_obtener_puntaje_palabra()` debería terminar con éxito (*passed*). El test de comodines puede fallar, pero está bien por ahora.

Problema 3: Trabajando con las manos

Representando las manos

Una mano es el conjunto de letras que tiene un jugador durante el juego. Inicialmente se reparten un conjunto de letras al azar. Por ejemplo, el jugador podría empezar con la siguiente mano:

a, q, l, m, u, i, l.

En nuestro programa, una mano será representada mediante un diccionario: las claves son letras (en minúscula) y los valores indican el número de veces que esa letra se encuentra repetida en la mano. Por ejemplo, la mano presentada anteriormente será representada como:

`mano = {'a':1, 'q':1, 'l':2, 'm':1, 'u':1, 'i':1 }.`

Convirtiendo las palabras en una representación de diccionario

Una función que ya tenemos definida es `obtener_diccionario_frecuencias()`, la cual nos devuelve un diccionario que representa la palabra pasada. Por ejemplo:

```
In[]: obtener_diccionario_frecuencias("Taller")
>> {'t': 1, 'a': 1, 'l': 2, 'e': 1, 'r': 1}
```

Mostrando una mano

Dada una mano representada por un diccionario, queremos mostrarla por la terminal de una forma entendible para el usuario. Para esto, se cuenta con una función ya implementada llamada `mostrar_mano()`. Si miramos el código de la función veremos que muestra cada letra que contiene la mano separada por un espacio.

Por ejemplo:

```
In[]: mano = obtener_diccionario_frecuencias("Taller")
In[]: mostrar_mano(mano)
>> T a l l e r
```

Generando una mano al azar

Para esto tenemos una función, también ya implementada, que recibe un número entero positivo `n` y genera un diccionario con las letras de la mano, generada al azar, según las restricciones planteadas en las reglas del juego (`repartir_mano(n)`).

Eliminando letras de una mano

Implementar la función `actualizar_mano(mano, palabra)`.

El jugador empieza la partida con una mano compuesta de `n` letras. A medida que va construyendo palabras, las letras de la mano se van usando.

Por ejemplo, el jugador puede empezar con la siguiente mano: a, t, l, m, u, i, l y elegir jugar “multa”. Esto resultará en una mano con las letras l, i.

Es **nuestra tarea** implementar la función que tome una mano y una palabra como entrada, utilice las letras de la mano para generar esa palabra y retorne una nueva mano únicamente conteniendo las letras restantes. Esta implementación no debe modificar la mano que viene como entrada.

Por ejemplo:

```
In[]: mano = {'a':1, 't':1, 'l':2, 'm':1, 'u':1, 'i':1 }
In[]: mostrar_mano(mano)
>> a t l l m u i
In[]: mano_nueva = actualizar_mano(mano, 'multa')
>> mano_nueva {'l': 1, 'i': 1}

In[]: mostrar_mano(mano_nueva)
>> l i
In[]: mostrar_mano(mano)
>> a t l l m u i
```

Nota: En el ejemplo anterior, después de llamar a `actualizar_mano()` la información almacenada en `mano_nueva` también podría ser el diccionario `{'a':0, 't':0, 'l':1, 'm':0, 'u':0, 'i':1}`. Los valores exactos dependen de nuestra implementación, pero la salida generada por `mostrar_mano()` sí debería ser la misma en ambos casos.

Importante: Si el jugador arriesga una palabra que no es válida, ya sea porque no es una palabra real o porque no se puede formar con las letras en la mano, de todas formas perderá las letras de la mano que sí existan en la palabra, como castigo. Debemos validar que nuestra implementación considere esta situación. No asumir que la palabra dada sólo usa letras que existen en la mano.

Por ejemplo:

```
In[]: mano = {'j':2, 'o':1, 'l':1, 'v':2, 'n':1 }
In[]: mostrar_mano(mano)
>> j j o l v v n
In[]: mano = actualizar_mano(mano, 'novela')
>> mano {'j':2, 'v':1}
In[]: mostrar_mano(mano)
>> j j v
```

De la palabra “novela” se usan las letras 'n', 'l', 'o' y 'v' (una instancia porque había dos en la mano). Las letras 'e' y 'a' no tienen impacto en la mano ya que no existían.

Testeo: Verificar que el `test_actualizar_mano()` pase correctamente.

Problema 4. Palabras válidas

Implementar la función `es_palabra_valida(palabra, mano, lista_palabras)`.

Una palabra válida debe existir en la lista de palabras (ignorando las mayúsculas) y debe estar compuesta completamente de letras en la mano actual.

Testeo: Verificar que el `test_es_palabra_valida()` pase correctamente. Además, se debería probar la implementación con otras entradas razonables, por ejemplo, verificar si funciona correctamente al ser llamada repetidas veces con la misma mano.

Problema 5. Comodines

Ahora queremos que las manos pueden tener también comodines (*). Los comodines solo pueden reemplazar vocales. Cada mano debe contener un único comodín entre sus letras.

El jugador no recibe puntos adicionales por usar el comodín, pero si suma a la cuenta de letras usadas y no usadas.

Durante el juego, si se quiere usar un comodín se debe escribir el carácter “*” reemplazando a la letra deseada en una palabra. El código que verifica la validez de una palabra debe detectar la existencia de una palabra válida con una vocal en el lugar donde el comodín fue usado. Los ejemplos siguientes muestran cómo los comodines deben comportarse en el contexto de jugar una partida.

Ejemplo #1: Una palabra construida sin usar el comodín.

Mano actual: n a f l * w

Ingrese una palabra o “!!” para indicar que desea terminar: flan

“flan” resulta en 154 puntos.

Total: 154 puntos.

Mano actual: * w

Ingrese una palabra o “!!” para indicar que desea terminar: !!

Puntaje final: 154 puntos

Ejemplo #2: Una palabra válida usando un comodín.

Mano actual: n a f l * w

Ingrese una palabra o “!!” para indicar que desea terminar: fl*n

“fl*n” resulta en 132 puntos.

Total: 132 puntos.

Mano actual: a w

Ingrese una palabra o “!!” para indicar que desea terminar: !!

Puntaje final: 132 puntos

Ejemplo #3: Una palabra inválida usando el comodín.

Mano actual: n a f l * w

Ingrese una palabra o “!!” para indicar que desea terminar: fl*w

No es una palabra válida, por favor ingrese otra palabra.

Mano actual: a n

Ingrese una palabra o “!!” para indicar que desea terminar: !!

Puntaje final: 0 puntos

Ejemplo #4: Otra palabra inválida usando un comodín.

Mano actual: n a f l * w

Ingrese una palabra o “!!” para indicar que desea terminar: fla*

No es una palabra válida, por favor ingrese otra palabra.

Mano actual: n w

Ingrese una palabra o “!!” para indicar que desea terminar: !!

Puntaje final: 0 puntos

Modificaciones a realizar para incorporar los comodines

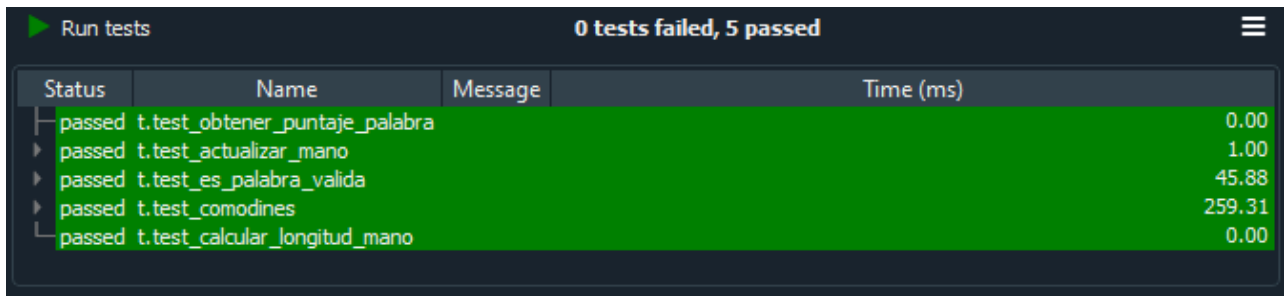
Adaptar la función `repartir_mano()` para que siempre entregue un comodín por mano. Actualmente esta función asegura que un tercio de las letras sean vocales y el resto consonantes; debemos dejar la cuenta de consonantes como está y reemplazar uno de los lugares para vocales con el comodín. Si es necesario, se pueden modificar las constantes definidas al principio de `WordGame.py`

Luego, modificar la función `es_palabra_valida()` para considerar los comodines.

Testeo: Verificar que el `test_comodines()` pase, además de probar otras entradas para validar si todo funciona correctamente.

Problema 6. Jugar una mano

Nos queda un último caso de test por resolver antes de empezar a implementar la lógica del juego: `calcular_longitud_mano()`. Esta función sólo debe retornar la cantidad de letras que hay en la mano. Cuando logremos pasar el `test_calcular_longitud_mano()` deberíamos ver:



The screenshot shows a test runner window with a dark background. At the top, it says "Run tests" and "0 tests failed, 5 passed". Below this is a table with four columns: Status, Name, Message, and Time (ms). All five tests listed are in the "passed" status.

Status	Name	Message	Time (ms)
passed	t.test_obtener_puntaje_palabra		0.00
passed	t.test_actualizar_mano		1.00
passed	t.test_es_palabra_valida		45.88
passed	t.test_comodines		259.31
passed	t.test_calcular_longitud_mano		0.00

Ahora sí podemos implementar la función `jugar_mano()`, la cual permitirá a un jugador jugar una mano individual.

Testeo: Probar la implementación como si fuéramos a jugar una mano. Ejecutar el programa, llamando a la función `jugar_mano()` pasándole una mano y la lista de palabras. La salida debería ser como los ejemplos que presentamos a continuación (respetar los mensajes y la forma de cada uno de ellos)

Ejemplo #1

Mano actual: a c f i * t l

Ingrese una palabra o "!!" para indicar que desea terminar: facil
"facil" resulta en 290 puntos. Total: 290 puntos
Mano actual: * t

Ingrese una palabra o "!!" para indicar que desea terminar: t*
"t*" resulta en 14 puntos. Total: 304 puntos
Se quedó sin letras
Puntaje final: 304 puntos

Ejemplo #2

Mano actual: a c f i * t l

Ingrese una palabra o "!!" para indicar que desea terminar: f*ca
"f*ca" resulta en 152 puntos. Total: 152 puntos
Mano actual: i t l

Ingrese una palabra o "!!" para indicar que desea terminar: li
No es una palabra válida, por favor ingrese otra palabra.
Mano actual: t

Ingrese una palabra o "!!" para indicar que desea terminar: !!
Puntaje final: 152 puntos

Problema 7. Jugar una partida

Una partida consiste de múltiples manos.

Necesitamos implementar dos funciones finales para completar nuestro Word Game: `intercambiar_mano()` y `jugar_partida()`. Esta última había sido estructurada en el problema 1, es hora de implementarlo por completo.

Para implementar `intercambiar_mano()` debemos seguir las especificaciones indicadas en el esqueleto de la función. También conviene revisar el código de `repartir_mano()` para ver el funcionamiento del código que realiza una elección al azar.

Recomendaciones: hacer un par de pruebas para ver si la longitud de la mano resultante es igual a la original cuando se reemplaza una letra con una ocurrencia, con varias o que no existe en la mano actual.

Para implementar la función `jugar_partida()` también se deben seguir las especificaciones. En este caso es recomendable siempre utilizar la constante `TAMANIO_MANO` para determinar el número de letras en la mano. No debemos asumir que siempre habrá 7 letra en la mano, así podemos construir un juego más flexible (si queremos jugar con más letras solo debemos cambiar el valor de esa constante)

Testeo: Probar esta implementación como si estuviéramos jugando el juego. Probar diferentes valores para `TAMANIO_MANO` en nuestro programa para verificar que todo funciona correctamente y que solo es necesario modificar el valor de la constante.

Ejemplo

Ingrese la cantidad de manos a jugar: 2

Mano actual:

o u j h h k *

----- ¿Desea intercambiar una letra ? (si/no):no

Mano actual: o u j h h k *

Ingrese una palabra o "!!" para indicar que desea terminar:hoj*

"hoj*" resulta en 247 puntos. Total: 247 puntos

Mano actual: u h k

Ingrese una palabra o "!!" para indicar que desea terminar:uh

"uh" resulta en 55 puntos. Total: 302 puntos

Mano actual: k

Ingrese una palabra o "!!" para indicar que desea terminar:!!

Puntaje final: 302 puntos

----- ¿Desea repetir la mano actual? (si/no):no

Mano actual:

o a k n f x *

----- ¿Desea intercambiar una letra ? (si/no):si

----- ¿Qué letra desea intercambiar?x

Mano actual:

o a k n f * u

Mano actual: o a k n f * u

Ingrese una palabra o “!!” para indicar que desea terminar:fan

"fan" resulta en 54 puntos. Total: 54 puntos

Mano actual: o k * u

Ingrese una palabra o “!!” para indicar que desea terminar:!!

Puntaje final: 54 puntos

----- ¿Desea repetir la mano actual? (si/no):si

Mano actual: o a k n f * u

Ingrese una palabra o “!!” para indicar que desea terminar:fauno

"fauno" resulta en 232 puntos. Total: 232 puntos

Mano actual: k *

Ingrese una palabra o “!!” para indicar que desea terminar:k*

"k*" resulta en 70 puntos. Total: 302 puntos

Se quedó sin letras

----- Puntaje final: 604 puntos

Problema 8. Programación Orientada a Objetos

Para cerrar este trabajo práctico integrador vamos a proponer un último problema a resolver: estructurar nuestro juego utilizando los conceptos de programación orientada a objetos.

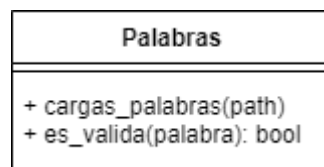
Hasta el momento resolvimos el problema trabajando con funciones directamente sobre las estructuras de datos que elegimos para representar los conceptos asociados al problema: la lista de palabras, el diccionario de letras para la mano, etc.

Ahora, nos vamos a preguntar ¿qué sucede si queremos independizar el juego y los algoritmos de las estructuras de datos concretas elegidas? Para esto debemos abstraer los datos y ocultar información, características fundamentales de la programación orientada a objetos.

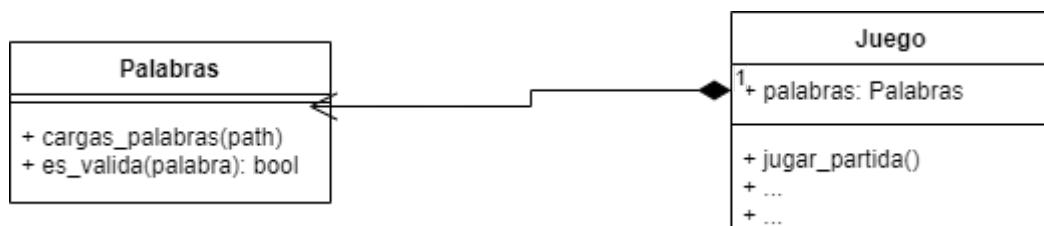
Este problema no va a requerir re-implementar el juego, sólo proponemos realizar un diseño de clases que permita entender qué entidades forman parte del juego y cuáles son sus responsabilidades.

Por ejemplo, la lista de palabras no necesariamente tiene que ser una lista o mejor aún, el juego no debería tener conocimiento de cómo se almacenan las palabras y, por lo tanto, no podría saber cómo verificar si una palabra existe o no en la lista de palabras.

Con esto en mente podemos detectar la necesidad de una clase Palabras con las siguientes responsabilidades:



El “juego” contendrá una referencia a las “Palabras” y usará la interfaz definida por la clase para construir la lógica del juego.



Ahora les proponemos completar este diagrama de clases con el resto de las entidades detectadas.