Autoría: Sabela Fiaño García (sabela.fgarcia@udc.es) y Sara Gende Longueira (sara.gende@udc.es)

Memoria

PARTE 1: Cómo ejecutar el programa (instrucciones para ejecutar el archivo principal → main.py):

Hay que descargar el archivo comprimido 'practica1_sabela_sara.zip' y extraer todos los archivos. A continuación abrimos los tres archivos con python. Debemos ejecutarlos todos, primero el 'unit.py', luego 'civilization.py' y, por último, 'main.py'. Además, para que funcione la batalla debemos tener en la misma carpeta los archivos de batalla. 'battle0.txt' y 'battle1.txt'.

PARTE 2: Explicación del proceso de programación:

1. Fase 1: Implementación de las clases:

Para comenzar, importamos ABC y @abstractmethod de abc para definir una clase abstracta Unit, así como la librería math (que utilizaremos para calcular la cantidad de daño que se infringe a cada unidad).

Luego, creamos la clase abstracta 'Unit' en la que definiremos los atributos característicos comunes a las clases hijas que se definirán más adelante. El único atributo que no definimos es el 'unit_type' puesto que será diferente para cada subclase.

Llamamos al método _init_ para construir un nuevo objeto de esa clase y distinguir un objeto de otro creado a partir de la misma clase, pasando el objeto creado como self y los argumentos que queremos almacenar en el objeto, de este modo definimos 'name' como 'str' (cadena de texto) y 'strength', 'defense',' hp' y 'total_hp' como 'int' (entero).

Además. también debemos implementar el método mágico (__str__()) para que la información que se nos muestre de cada civilización sea legible.

Usamos el decorador @abstractmethod para definir métodos para las subclases (effectiveness, attack y debilitated) y así garantizar que cada subclase proporciona su propia implementación específica de este método:

· 'effectiveness' será implementada en cada subclase y es diferente para cada una. Define la relación entre las subclases en base a su interacción en batalla, usando al objeto (self) y a la unidad oponente (opponent:'Unit') para devolver un entero (int) siendo -1 si es débil ante el oponente ,0 si es neutro contra el oponente o 1 si es efectivo ante el oponente.

· 'attack' aunque también es diferente para cada subclase si que se implementa también en la clase 'Unit' porque en el caso de que la unidad no disponga del atributo característico de su subclase (flechas, carga o furia) seguirá infringiendo el daño básico de 1 unidad, modificando dependiendo de la subclase, usando al objeto (self) y a la unidad oponente (opponent:'Unit') para devolver un entero (int).

· 'is_debilitated' solo se implementa en la clase 'Unit' porque es igual para todas las subclases. Devolverá 'True' cuando el oponente se quede sin vida y 'False' en caso contrario. Para ello toma el atributo hp (puntos de salud) lo compara con 0 y devuelve un booleano (bool)

Además, es importante que utilicemos las etiquetas @property y @setter puesto que sin ellas no podríamos tener acceso a los atributos privados y el programa no funcionaría. Además también comprueba que se haya introducido un valor válido. En el caso de los atributos numéricos que sea un valor entero y mayor que 0 y en el caso de las cadenas de texto que no sea una cadena vacía.

Ahora definimos cada una de las clases hijas. Para ello utilizamos en todas ellas la función 'super().__init__' para llamar a la superclase, añadimos el atributo propio de cada subclase ('Archer' → 'arrows', 'Cavalry' → 'charge', 'Infantry' → 'fury') y también definimos para cada una el 'unit_type' dependiendo de la unidad que sea.

Finalmente, dentro de cada subclase definimos el ataque y la efectividad de cada uno. En el caso del ataque será diferente para cada una respecto a 'n' que será el número de daño que realiza. Como precondición tenemos que comprobar que el atributo propio no sea inferior a 0 puesto que, en ese caso, el daño sería una única unidad. En el otro caso, en 'effectiveness' dependerá de la unidad a la que se está atacando.

Dentro de cada clase es importante volver a utilizar las etiquetas @property y @setter para que los atributos propios de cada una puedan ser llamados cuando se necesiten.

Como añadido, en la clase 'worker' definimos la función 'collect()' que lo que hace es devolver el valor 10 ya que cada vez que se utiliza esta unidad se recolectarán 10 recursos.

Además, para comprobar el funcionamiento correcto de esta primera parte del programa podemos utilizar el archivo 'unit_tests'. Para ello debemos tener ambos archivos en la misma carpeta y, una vez hecho, los ejecutamos a los dos. 'unit_tests' nos irá diciendo cuántos errores y fallos de ejecución tenemos para así poder corregirlos y llegar al 'OK', es decir, que el archivo funcione.

2. Fase 2: Implementación del módulo civilization.py:

En este archivo se implementa la clase civilization a la que le definiremos sus atributos propios: 'name', el nombre de cada civilización, 'resources', la cantidad de recursos que tiene cada civilización y 'units', cuántas y qué unidades tiene cada civilización. Tal como se hizo en la fase 1 se utilizará el método __init__ para definir todos estos atributos y que para que puedan ser accesibles volvemos a utilizar las etiquetas @property y @setter.

A continuación, definimos tres funciones propias de la clase que son:

· 'train_unit' esta función antes de nada comprueba qué unidad se quiere crear y, para dicha unidad, si hay suficientes recursos (en el caso del worker se necesitan 30 y para el resto de unidades 60). La unidad que se haya creado se introduce dentro de la lista 'units' y se le especifican los valores de cada atributo específicos a la unidad. Los valores siguen el orden de: fuerza, defensa, hp y total_hp, además para las unidades distintas de worker también añadimos el atributo especial dependiendo de la unidad (flechas, carga o furia)

Si no hay recursos suficientes se devolverá None y la lista de 'units' vacía.

· 'collect_resources' esta función lo que hace es recorrer la lista de 'units' y cuando la unidad sea de tipo 'worker' se invocará a la función collect que fue definida en el archivo 'unit.py' y el 'worker' desempeñará su función que es aumentar en 10 unidades los recursos totales de la civilización.

· 'all_debilitated' lo que hace es recorrer la lista de unidades y comprobar los hp de todas ellas. En el caso de que todos los hp sean igual a 0 se devolverá el valor booleano 'True' y la civilización perderá puesto que no tiene ninguna unidad con vida. En el caso contrario, se devolverá el valor booleano 'False' y se seguirá jugando hasta que alguna de las dos civilizaciones se quede sin unidades con vida.

3. Fase 3: Implementación de la lógica de la batalla entre dos civilizaciones:

Para esta fase utilizaremos el archivo ya creado 'main.py' al que le haremos nosotros algunos cambios que permitan la ejecución completa y correcta del programa.

Lo primero que hacemos es crear las civilizaciones. Para ello llamamos al módulo ya creado 'civilization' y cambiamos los valores de 'name' por civ (1 o 2 dependiendo de la civilización)_name y lo mismo con civ_resources. Además, para crear la misma cantidad de unidades en una civilización como en la otra programamos un bucle diferente para cada unidad e igual para las dos civilizaciones.

A continuación, debemos crear varias funciones diferentes que nos permitan programar las distintas fases (recolección, producción, batalla y estadisticas):

· 'def recoleccion()', antes de nada creamos una lista con ambas civilizaciones que iremos recorriendo con un bucle for y para cada una invocamos la función 'collect_resources()'. También muestra el estado actual de los recursos y muestra todas las unidades creadas, cuyo 'hp' es mayor de 0, junto con la vida que le queda.

· 'def produccion()', antes de nada comprobamos que los recursos no sean inferiores a 30 ya que en ese caso no se podría crear ninguna unidad. En caso contrario, se creará una unidad dependiendo del turno en el que estea la civilización: si el resto al dividirlo entre los 4 turnos que hai da 0 puede crear un 'Archer', si es 1 puede crear un 'Cavalry', si es 2 un 'Infantry' y si es 3 un 'Worker'. Al imprimirlo por pantalla podemos ver que se crean 5 'worker', 2 'archer', 2 'cavalry' y solo 1 'Infantry' porque, de acuerdo con los recursos iniciales de la batalla 0, no tiene suficientes recursos para crear el segundo 'Infantry'.

· 'def selec_objetivo', 'def vivas_no_workers' y 'def ha_ganado' no son una fase en sí sino que son implementadas en la función de batalla. La primera selecciona los oponentes a los que atacará cada atacante de cada civilización. Además también comprueba que el oponente tenga vida y que no sea un 'worker' a menos que el resto de unidades ya estén muertas. En la segunda función comprobamos si quedan unidades vivas que no sean 'worker'. En la tercera se comprueba si alguna de las civilizaciones tiene todas sus unidades muertas, si es así devuelve el valor booleano 'True'. Para comprobar esto, esta última función invoca el método 'all_debilitated()'.

· 'def batalla' simula el combate entre dos civilizaciones. Primero se comprueba si alguna de las civilizaciones ha perdido todas sus tropas con la función ha_ganado si ninguna ha ganado aún continúa la batalla. Se gestionan los ataques entre las unidades con vida y el objetivo seleccionado mediante la función selec_objetivo, para ello invoca a la función, del módulo 'unit.py', 'attack()' y este daño se le resta al hp del oponente. Luego, se comprueba si alguna de las civilizaciones tiene más unidades operativas que la otra. En ese caso, estas atacan al final del turno. A continuación, se comprueba si todas las unidades diferentes de 'worker' están sin vida, si es así los que atacan ahora son los worker a los cuales también les hay que comprobar si tienen hp > 0.

Después, con la ayuda de la librería pandas, registramos el daño medio por civilización, por unidad y por unidad respecto a cada civilización. Para eso creamos la función 'def estadisticas' que implementaremos en la función 'def batalla' tras el código de gestión del ataque.

Por último programamos los turnos, para eso usamos un bucle while que aumenta el turno hasta los deseados pero también se detiene si la variable "ganador" es True. En este incluiremos las funciones de cada fase (recolección, producción y batalla) y llamaremos con un if a ha_ganado para comprobar si tras la batalla alguna civilización queda sin tropas, si es así mostrará con un print a la civilización ganadora y cambiará el valor de "ganador" a True, terminando así el bucle y llamando a la función estadisticas.