杀死尖塔

作业2
第1学期，2023年
CSSE1001/CSSE7030

到期日：2023年4月28日 16:00 GMT+10

1 简介

*杀戮尖塔》*是一款类似于流氓的卡牌游戏，玩家必须建立一副卡牌，在遇到怪物的时候使用。原有的"*杀戮尖塔*"游戏的细节可以在这里找到。在作业2中，你将创建一个基于对象的文本游戏，其灵感来自于《*尖塔杀机》*[1]（尽管对其进行了大量的简化和修改）。

你需要实现本文件第5节中规定的类和方法的集合。你的程序的输出必须与预期的输出*完全一致*；输出的细微差别（如空格或大小写）*将*导致测试失败，导致这些测试的*零分*。本文件的任何变化都将列在文件末尾的变化日志中。

2 入门

从黑板上下载a2.zip--该档案包含开始这项作业的必要文件。一旦解压，a2.zip档案将提供以下文件/目录：

a2.py
: 游戏引擎。这是*你提交和修改的唯一文件*。*不要*对任何其他文件进行修改。

a2_support.py
: 支持代码，以协助进行更复杂的分配部分，并处理随机性。注意，在实现随机行为时，你**必须**使用支持函数，而不是自己从随机中调用函数。

游戏
: 一个包含几个可玩游戏文本文件的文件夹。

游戏_例子
: 一个包含运行已完成任务的示例输出的文件夹。

3 游戏玩法

在游戏开始时，用户选择一个玩家角色；不同的玩家角色有不同的优势和劣势，如开始时有更高的HP或更好的牌组。然后，用户选择一个游戏文件，其中规定了他们将进行的遭遇战。在这之后，游戏就可以开始了。

在游戏过程中，玩家使用一副卡牌来完成一系列与怪物的交锋。每场遭遇战涉及1至3个怪物，玩家必须在一系列回合中与它们平行作战。在每个回合开始时，用户从他们的卡组中随机抽出5张卡到他们的手中

。每张牌都要花费0到3个能量点才能发动。在他们的回合中，用户可以从他们的手牌中打出他们想要的牌，只要他们仍然有打出所需的能量点。用户在打完牌后可以选择结束自己的回合，这时遭遇的怪物会各自采取一个行动（可能会影响玩家的HP或其他属性，或者怪物自身的属性）。当一张牌被打出后，它会立即被送到玩家的弃牌堆中。在一个回合结束时，玩家手中的所有牌（无论它们是否在该回合被打出）都会被送到弃牌堆中。弃牌堆中的牌不能被抽出。

[1] 当原始游戏的行为与本规范不同时，应按本规范的要求实现分配（而不是按游戏的原始行为）。

直到整个牌组被抽出，这时牌组将被替换成弃牌堆中的所有牌。当玩家杀死了所有的怪物（将它们的HP减少到0）或者怪物杀死了玩家（将玩家的HP减少到0），一场遭遇战就会结束。如果玩家赢得了一场遭遇战，就会印出一场遭遇战的胜利信息，并开始下一场遭遇战。如果没有更多的遭遇战了，游戏就会以游戏胜利信息终止，如果玩家输掉了一场遭遇战，程序就会以损失信息终止。相关信息请参见a2_support.py。你可以在a2.zip中提供的game_examples文件夹，找到游戏的例子。关于main的行为的更多细节，见第5.4节。

4 类的概述和关系

你被*要求*实现一些类，以及一个主函数。你应该先开发这些类，并确保它们都能工作（至少要确保它们通过Gradescope的测试），然后再开始主函数的工作。图1中的类图提供了*所有*这些类的概况，以及它们之间的基本关系。这些类和它们的方法的细节将在第5节深入描述。

- 空心箭头表示*继承性*（即 "是-A "关系）。
- 带点的箭头表示*构成*（即 "有-a "关系）。标有1-1的箭头表示箭头底部的每个类的实例正好包含箭头头部的一个类的实例。标有1-n的箭头表示箭头底部的每个类的实例可能包含箭头头部的许多类的实例。例如，一个邂逅实例可能包含1到3个怪物实例，但只包含一个玩家实例。
- 绿色的类是*抽象的*类。在你的程序中，你只应该实例化蓝色的类，尽管你应该在开始对其子类工作之前实例化绿色的类来测试它们。

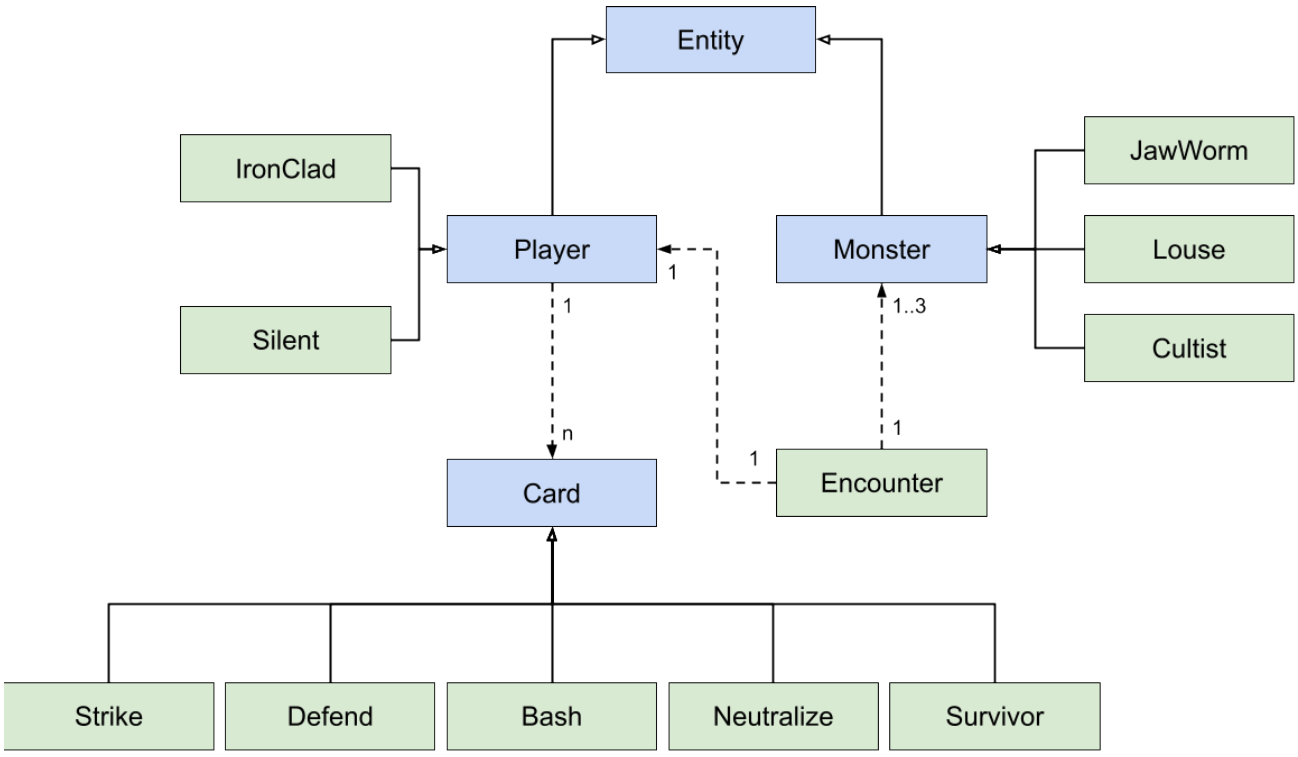

图1：本次作业需要实现的类的基本关系图。

5 实施

本节概述了你需要实现的类、方法和函数，作为你任务的一部分。建议你按照描述的顺序来实现这些类

。在进入下一个 类之前，确保每个类的行为符合示例的要求（以及在可能的情况下，Gradescope的测试）。

5.1 卡片

卡片在遭遇战中被玩家用来攻击怪物、防御或应用状态修改器。在实现这个类的时候，你还不需要了解*如何*应用这些效果的力学原理。你将在以后的实现中处理这个问题。卡片类只是提供每种类型的卡片所具有的效果的信息；它们不负责直接造成这些效果。

所有可实例化的卡片都继承自抽象的卡片类，并且应该继承默认的卡片行为，除非在每种特定类型的卡片的描述中指定。

卡片 *(抽象类)*

一个抽象的类，所有可实例化的卡片类型都继承于此。提供默认的卡片行为，它可以被特定类型的卡片所继承或覆盖。所有卡片的`init`方法除了`self`之外不接受任何参数。

get_damage_amount(self) -> int *(方法)*

返回这张牌对其目标（即被打的对手）造成的伤害量。默认情况下，一张牌造成的伤害为0。

get_block(self) -> int *(方法)*

返回此卡为其用户增加的块数。默认情况下，卡所提供的区块数量为0。

get_energy_cost(self) -> int *(方法)*

返回这张牌的能量消耗量。默认情况下，能量成本应该是1。

get_status_modifiers(self) -> dict[str, int] *(方法)*

返回一个字典，描述这张牌被打出时应用的每个状态修改器。默认情况下，不应用任何状态修改器；也就是说，这个方法在抽象的`Card`类中应该返回一个空的字典。

get_name(self) -> str *(方法)*

返回卡片的名称。在`Card`的超类中，这只是字符串 "Card"。

get_description(self) -> str *(方法)*

返回卡片的描述。在`Card`的超类中，这只是一个字符串 "A card"。

requires_target(self) -> bool *(方法)*

如果玩这张牌需要一个目标，则返回`True`，如果不需要，则返回`False`。默认情况下，一张牌需要一个目标。

str (self) -> str *(方法)*

返回卡片的字符串表示，格式为`'{卡片名称}：{卡描述}'`。

repr (self) -> str *(方法)*

返回创建一个与自己相同的该类的新实例所需的文本。

实例

```
>>> Card = Card()
>>> Card.get_damage_amount()
0
>>> card.get_block()
0
>>> card.get_energy_cost()
1
>>> card.get_status_modifiers()
{}
```

```
>>> card.get_name()
'卡'。
>>> card.get_description()
'一张卡片。
>>> card.requires_target()
True
>>> str(card) '卡:
一张卡。
>>>
Card
```

罢工 *(类)*

继承自卡片
攻击是一种对其目标造成6点伤害的卡。它需要花费1个能量点才能发动。

实例

```
>>> strike = Strike()
>> print( strike.get_damage_amount(), strike.get_block(), strike.get_energy_cost()
6 0 1
>>> strike.get_name()
'Strike'。
>>> strike.get_description()
'造成6次伤害。
>>> strike.required_target()
True
>>> str( strike) '攻击:
造成6点伤害。
>>>
Strike
```

保卫 *(类)*

继承自卡片
防御是一种可以为使用者增加5个格挡的卡片。防御不需要一个目标。它需要花费1个能量点才能发挥作用。

实例

```
>>> defend = Defend()
>> print(defense.get_damage_amount(), defend.get_block(), defend.get_energy_cost())
0 5 1
>>> defend.get_name() '防御'
。
>>> defend.get_description()
'获得5块。
>>> defend.requires_target()
False
>> str(defend) '防御:
获得5个阻挡。
>>> 防御
```

巴什 *(类)*

继承自卡片

撞击是一种卡牌，为其使用者增加5个格挡，并对目标造成7个伤害。它需要2个能量点才能发挥。

实例

```
>>> bash = Bash()
>> print(bash.get_damage_amount(), bash.get_block(), bash.get_energy_cost()
7 5 2
>>> bash.get_name()
'Bash'。
>>> bash.get_description() '造
成7点伤害。获得5个格挡。
>>> bash.requires_target()
True
>>> str(bash)
'Bash：造成7点伤害。获得5个格挡。
>>>
bash
```

中性化 *(类)*

继承自卡片

中和是一种对其目标造成3点伤害的牌。它还会对目标施加状态修饰；即施加1个虚弱和2个脆弱。中和不需要花费任何能量点来发挥。

实例

```
>>> neutralize = Neutralize()
>> print(neutralize.get_damage_amount(), neutralize.get_block(), neutralize.get_energy_cost()
3 0 0
>>> neutralize.get_status_modifiers()
{'弱': 1, '脆弱': 2}
>>> neutralize.get_name()
'Neutralize'。
>>> neutralize.get_description () 。
'造成3次伤害。应用1个弱点。应用2个脆弱的。
>>> str(neutralize)
'中和：造成3次伤害。应用1个弱点。应用2个弱点。
>>> neutralize
Neutralize()
```

幸存者 *(类)*

继承自卡片

幸存者是一种增加8块并对其使用者施加1个强度的牌。幸存者不需要目标。

实例

```
>>> survivor = Survivor()
>> print(survivor.get_damage_amount(), survivor.get_block(), survivor.get_energy_cost()
0 8 1
>>> survivor.get_status_modifiers()
{'强度': 1}
```

```
>>> survivor.requires_target()
False
>>> survivor.get_name()
'Survivor'（幸存者）。
>>> 幸存者.get_description() '
获得8个街区和1个力量。
>>> str(survivor)
'生存者：获得8个区块和1个力量。
>>> 幸存者
Survivor()
```

5.2 实体

游戏中的实体包括玩家和敌人（怪兽）。所有实体都有：

- 健康点数（HP）：这是从实体的最大HP开始的，并可能在一次或多次遭遇中减少。当一个实体的HP减少到0时，它就被打败了。
- 块：这是该实体所拥有的防御量。当一个实体被攻击时，伤害会首先作用于块。只有当区块被减少到0时，任何剩余的伤害才会对实体的HP造成影响。例如，如果一个拥有10点HP和5点块的实体受到8点伤害的攻击，他们的块会被减少到0，而HP会减少到7。
- 力量：这个实体拥有的额外力量。当实体打出一张对目标造成伤害的牌时，实体每拥有一个力量点，造成的伤害就会增加1。力量在遭遇战结束前不会消退。
- 虚弱：这个实体在多少个回合内是虚弱的。如果一个实体在某一回合是虚弱的，那么该实体打出的所有造成伤害的牌都会减少25%的伤害。
- 脆弱的：该实体易受攻击的回合数。如果一个实体在某一回合是脆弱的，对它造成的伤害将增加50%。

在这个作业中，你必须实现一个抽象的Entity类，它提供了所有实体都继承的基础实体功能。除了指定的地方，实体具有从Entity超类继承的默认行为。

实体 *(抽象类)*

抽象基类，所有实体都从它那里继承。

init (self, max_hp: int) -> None *(方法)*

用给定的max_hp设置一个新实体。一个实体以它能拥有的最大HP值开始。阻力、强度、弱点和脆弱都从0开始。

get_hp(self) -> int *(方法)*

返回该实体的当前HP。

get_max_hp(self) -> int *(方法)*

返回该实体的最大可能的HP。

get_block(self) -> int *(方法)*

返回这个实体的块的数量。

get_strength(self) -> int *(方法)*

返回这个实体的力量大小。

get_weak(self) -> int *(方法)*

返回这个实体是弱者的回合数。

get_vulnerable(self) -> int *(方法)*

返回该实体易受攻击的回合数。

get_name(self) -> int *(方法)*

返回实体的名称。实体的名称只是它所属的最具体类别的名称。

reduce_hp(self, amount: int) -> None *(方法)*

攻击实体，造成一定量的伤害。这涉及到减少块数，直到伤害量已经完成，或者直到块数减少到0，在这种情况下，HP被减少了剩余的量。例如，如果一个实体有20个HP和5个块，调用reduce_hp的金额为10，将导致15个HP和0个块。HP不能低于0。

is_defeated(self) -> bool *(方法)*

如果该实体被击败，则返回True，否则返回False。如果一个实体没有剩余的HP，它就被打败了。

add_block(self, amount: int) -> None *(方法)*

将给定的数量添加到该实体拥有的块的数量。

add_strength(self, amount: int) -> None *(方法)*

将给定的数量添加到这个实体所拥有的力量数量上。

add_weak(self, amount: int) -> None *(方法)*

将给定的数量添加到该实体拥有的弱点数量中。

add_vulnerable(self, amount: int) -> None *(方法)*

将给定的数量添加到这个实体所拥有的脆弱数量中。

new_turn(self) -> None *(方法)*

适用于新的回合开始时发生的任何状态变化。对于基本的实体类，这涉及到将阻挡设置为0，如果弱点和脆弱大于0，则分别减少1。

str (self) -> str *(方法)*

返回实体的字符串表示，格式为'{实体名称}: {current HP}/ {max HP}'。/ {最大HP}'。

repr (self) -> str *(方法)*

返回创建一个与自己相同的该类的新实例所需的文本。

实例

```
>> entity = Entity(20)
>>>实体.get_name() '实体'
>> print(entity.get_hp(), entity.get_max_hp(), entity.get_block())
20 20 0
>> print(entity.get_strength(), entity.get_weak(), entity.get_vulnerable())
0 0 0
>>实体.reduce_hp(2)
>>>实体.get_hp() 18
>>> entity.add_block(5)
>>实体.get_block() 5
>>实体.reduce_hp(10)
>>>实体.get_hp() 13
```

```
>>>实体.get_block() 0
>>>实体.is_defeated() False
>>>实体.add_strength(2)
>>>实体.add_weak(3)
>>>实体.add_vulnerable(4)
>> print(entity.get_strength(), entity.get_weak(), entity.get_vulnerable())
2 3 4
>>> entity.add_block(5)
>>实体.get_block() 5
>>>实体.new_turn()
>> print(entity.get_strength(), entity.get_weak(), entity.get_vulnerable())
2 2 3
>>>实体.get_block() 0
>>>实体.get_hp() 13
>>实体.reduce_hp(15)
>>>实体.get_hp() 0
>>>实体.is_defeated() True
```

队员 *(抽象类)*

继承自实体

玩家是一种由用户控制的实体类型。除了常规的实体功能外，玩家还拥有能量和牌。玩家必须管理三套牌；牌组（剩余可抽的牌），他们的手牌（当前回合可玩的牌），以及弃牌堆（这次遭遇中已经玩过的牌）。

init (self, max_hp: int, cards: list[Card] | None = None) -> None *(方法)*

除了执行实体超类的初始化程序外，该方法还必须初始化玩家的能量，该能量从3开始，以及三个牌表（牌组、手牌和弃牌堆）。如果牌组参数不是None，那么牌组将被初始化为牌。否则，它应该被初始化为一个空列表。玩家的手牌和弃牌堆开始是空列表。

get_energy(self) -> int *(方法)*

返回用户剩余的能量数量。

get_hand(self) -> list[Card] *(方法)*

返回玩家当前的手牌。

get_deck(self) -> list[Card] *(方法)*

返回玩家当前的牌组。

get_discarded(self) -> list[Card] *(方法)*

返回玩家当前的弃牌堆。

start_new_encounter(self) - > None *(方法)*

这种方法是先把玩家手中的所有牌加到他们的牌组末尾，再把玩家的弃牌堆中的所有牌加到他们的牌组末尾。请注意，这必须按照正确的顺序进行，才能使随机选牌选择到正确的牌。这个方法还必须将手牌和弃牌堆设置为空列表。

end_turn(self) -> None *(方法)*

这种方法是将玩家手中所有剩余的牌加到他们的弃牌堆的末尾，并将他们的手牌重新设置为空牌。

new_turn(self) -> None *(方法)*

这种方法为玩家准备了一个新的回合。这涉及到普通实体对新回合的所有要求，但也要求玩家发一张新的5张牌，并且能量被重置为3。

注意：你**必须**使用a2_support.py的draw_cards函数来实现给玩家发新牌。你必须从这个方法中准确地调用 draw_cards 函数**一次**。**不要**使用a2_support.py中的select_cards方法来实现随机选牌。

play_card(self, card_name: str) -> Card | None *(方法)*

试图从玩家的手中打出一张牌。如果玩家手中有一张给定名称的牌，并且玩家有足够的能量来打出这张牌，那么这张牌将从玩家手中移出并加入弃牌堆，所需的能量将从玩家的能量中扣除，并且这张牌将被返回。如果玩家手中没有给定名称的牌，或者玩家没有足够的能量来打出所要求的牌，这个函数将返回无。

实例

注意：如果你启动一个新的IDLE外壳（即在输入命令前重新运行你的程序），这个例子部分和后面的许多例子应该是完全可以复制的。如果你运行多个实例部分而不在中间重新启动IDLE shell，在new_turn期间分配到玩家手中的牌可能与实例中显示的不同。

```
>> player = Player(50, [Strike(), Strike(), Strike(), Defend(), Defend(), Bash() ] )
>>> player.get_name()
'Player'。
>>> player.get_hp()
50
>>> player.get_energy()
3
>> print(player.get_hand(, player.get_discarded() )
[] []
>>> player.get_deck()
[Strike(), Strike(), Strike(), Defend(), Defend(), Bash()] 。
>>> player.new_turn()
>>> player.get_hand()
[Strike(), Defend(), Strike(), Strike(), Bash()] 。
>>> player.get_deck()
[Defend(), Defend()] 。
>>> player.get_discarded()
[] 。
>>> player.play_card('Bash')
Bash()
>>> player.get_hand()
[罢工()，防御()，罢工()，罢工()] 。
>>> player.get_deck()
[Defend(), Defend()] 。
>>> player.get_discarded()
[Bash()] 。
>>> player.end_turn()
>>> player.get_hand()
[] 。
>>> player.get_deck()
[Defend(), Defend()] 。
>>> player.get_discarded()
```

```
>>> str(player)
'Player: 40/50 HP'
```

铁甲 *(类)*

继承自播放器

铁甲是一种开始有80HP的玩家。铁甲的卡组包含5张攻击卡、4张防御卡和1张冲撞卡。铁甲的初始_方法除了自我之外不接受任何参数。

实例

注意：在运行这些例子之前重新启动你的IDLE shell，以复制抽到手牌的情况。

```
>>> iron_clad = IronClad()
>>> iron_clad.get_name()
'IronClad'。
>> str(iron_clad)
'IronClad: 80/80 HP'
>>> iron_clad.get_hp()
80
>>> iron_clad.get_hand()
[] 。
>>> iron_clad.get_deck () 。
[Strike(), Strike(), Strike(), Strike(), Strike(), Defend(), Defend(), Defend(), Bash() ]
>>> iron_clad.new_turn () 。
>>> iron_clad.get_hand()
[Strike(), Strike(), Strike(), Bash(), Strike()] 。
>>> iron_clad.get_deck () 。
[Strike(), Defend(), Defend(), Defend()]。
```

无声 *(类)*

继承自播放器

沉默是一种类型的玩家，开始时有70HP。沉默者的卡组包含5张攻击卡、5张防御卡、1张中立卡和1张生存者卡。沉默者的初始_____方法除了自我之外不接受任何参数。

实例

注意：在运行这些例子之前重新启动你的IDLE shell，以复制抽到手牌的情况。

```
>>> silent = Silent()
>>> silent.get_name() '沉默'
。
>> str(silent) '无声
: 70/70 HP'。
>>> silent.get_hp()
70
>>> silent.get_hand()
[] 。
>>> silent.get_deck()
[Strike(), Strike(), Strike(), Strike(), Strike(), Defend(), Defend(), Defend(),
Neutralize(), Survivor() ]
>>> silent.new_turn () 。
>>> silent.get_hand()
[打击(), 打击(), 攻击(), 中和(), 防御()]
>>> silent.get_deck()
[Strike(), Strike(), Defend(), Defend(), Defend(), Survivor()]
```

怪物 *(抽象类)*

继承自实体

怪物是一种实体，用户在遭遇战中与之战斗。除了常规的实体功能外，每个怪物也有一个独特的ID，以及一个处理怪物行动对自身影响的行动方法，并描述怪物行动对其目标的影响。

在游戏过程中，遭遇战中的每个怪物每回合将获得一次行动机会（详见5.3节）。然而，在实现怪物类和子类时，了解轮回系统或怪物的行动效果将如何应用于玩家并不重要。怪物类只负责将怪物的行动效果应用于怪物本身（例如有些怪物会在行动中增加自己的属性），并返回一个描述该行动将如何影响玩家的字典。

init (self, max_hp: int) -> None *(方法)*

设置一个具有给定的最大HP和唯一ID号码的新怪物。第一个创建的怪物应该有一个0的ID，第二个创建的怪物应该有一个1的ID，等等。

get_id(self) -> int *(方法)*

返回这个怪物的唯一ID号码。

action(self) -> dict[str, int] *(方法)*

执行这个怪物的当前动作，并返回一个字典，描述这个怪物的动作应该对其目标造成的影响。在抽象的Monster超类中，这个方法应该只是引发一个NotImplementedError。这个方法必须被Monster的可实例化的子类所覆盖，其策略是针对每种类型的怪物的。

实例

注意：在运行这些例子之前，你可能需要重新启动你的IDLE shell来复制怪物的ID。

```
>>> monster = Monster(20)
>>> monster.get_id()
0
>>> another_monster = Monster(3)
>>> another_monster.get_id()
1
>>> monster.get_id()
0
>>> monster.action()
回溯（最近一次调用）：文件"<stdin>"，第
1行，在<module>中
...
raise NotImplementedError
NotImplementedError
>>> monster.get_name() '怪物
'。
>>> monster.reduce_hp(10)
>>> str(monster) '怪
```

虱子 *(类)*

继承自怪物

Louse的动作方法简单地返回一个{'伤害': 数量}的字典，其中数量是5到7（包括）之间的金额，在Louse实例被创建时随机生成。你**必须**使用 a2_support.py 中的 random_louse_amount 函数来生成每只老鼠的攻击量。你**必须**在创建虱子时，对每个虱子实例只调用一次 random_louse_amount 函数。

实例

注意：在运行这些例子之前，你可能需要重新启动你的IDLE shell来复制怪物的ID。

```
>>> louse = Louse(20)
>> str(louse)
'Louse: 20/20 HP'。
>>> louse.get_id()
0
>>> louse.action()
{'伤害': 6}
>>> louse.action() # 应该是相同的伤害量
{'伤害': 6}
>>> louse.action()
{'伤害': 6}
>>> another_louse = Louse(30)
>>> another_louse.action()
{'伤害': 7}
>>> another_louse.get_id()
1
```

崇拜者 *(类)*

继承自怪物

Cultist的攻击方法应该返回一个{'damage': damage_amount, 'weak': weak_amount}的字典。对于每个Cultist实例，在第一次调用动作时，damage_amount为0。对于每一个后续的行动调用，damage_amount = 6 + num_calls，其中num_calls是行动方法被调用的次数。
在这个特定的Cultist实例上。每次行动方法在0和1之间交替进行。
在一个特定的Cultist实例上调用，第一次调用时从0开始。

实例

```
>>> cultist = Cultist(20)
>>> cultist.action()
{'伤害': 0, '弱': 0}
>>> cultist.action()
{'伤害': 7, '弱': 1}
>>> cultist.action()
{'伤害': 8, '弱': 0}
>>> cultist.action()
{'伤害': 9, '弱': 1}
>>> cultist.action()
{'伤害': 10, '弱': 0}
>>> another_cultist = Cultist(30)
>>> Another_cultist.action()
{'伤害': 0, '弱': 0}
```

颚虫 *(类)*

继承自怪物

每次在一个JawWorm实例上调用动作，都会发生以下效果：

- 迄今为止，下巴虫所受伤害的一半（四舍五入）被加到下巴虫自身的阻挡量上。
- 迄今为止，颚虫所受伤害量的一半（向下取整）用于对目标的伤害。

到目前为止受到的伤害量是颚虫的最大HP和其当前HP之间的差值。

实例

```
>>> jaw_worm = JawWorm(20)
>>> jaw_worm.get_block()
0
>>> jaw_worm.action() # 在开始时应该产生0，因为jaw_worm还没有失去任何HP。
{'伤害': 0}
>>> jaw_worm.get_block()
0
>>> jaw_worm.reduce_hp(11)
>>> jaw_worm.action()
{'损害': 5}
>>> jaw_worm.get_block()
6
>>> jaw_worm.reduce_hp(5)
>>> jaw_worm.get_hp()
9
>>> jaw_worm.get_block()
1
>>> jaw_worm.reduce_hp(5)
>>> jaw_worm.get_hp()
5
>>> jaw_worm.action()
{'伤害': 7}
>>> jaw_worm.get_block()
8
```

5.3 邂逅

邂逅 *(类)*

游戏中的每一个遭遇都被表示为一个遭遇类的实例。这个类管理着一个玩家和一组1到3个怪物，并促进玩家和怪物之间的互动。本节描述了必须作为遭遇战类的一部分来实现的方法。

__init__(self, player: Player, monsters: list[tuple[str, int]]) -> None *(方法)*

遭遇战的初始化程序需要玩家实例，以及描述遭遇战中的怪物的元组列表。每个元组包含怪物的名称（类型）和怪物的最大HP。初始化器应该使用这些元组按照描述的顺序来构建怪物的实例。初始化器还应该告诉玩家开始一个新的遭遇（见Player.start_new_encounter），并且还应该开始一个新的回合（见下文start_new_turn的描述）。

start_new_turn(self) - > None *(方法)*

这个方法将其设定为玩家的回合（即允许玩家尝试应用牌），并称为
玩家实例上的new_turn。

end_player_turn(self) -> None *(方法)*

这个方法将其设定为不属于玩家的回合（即玩家不允许尝试应用牌），并确保玩家手中剩余的所有牌都移到他们的弃牌堆中。该方法还对遭遇战中剩余的所有怪物实例调用new_turn方法。

get_player(self) -> Player *(方法)*

返回本次遭遇战中的玩家。

get_monsters(self) -> list[Monster] *(方法)*

返回本次遭遇战中剩余的怪物。

is_active(self) - > bool *(方法)*

如果在这个遭遇中还有怪物，则返回 "真"，否则返回 "假"。

player_apply_card(self, card_name: str, target_id: int | None = None) -> bool *(方法)*

这个方法试图应用玩家手中第一张给定名称的牌（在相关情况下，该牌的目标由给定的目标_id指定）。这个方法的执行步骤如下：

1. 如果该卡的申请因以下任何原因而无效，则返回 "假"：
    - 如果没有轮到该玩家
    - 如果给定名称的卡片需要一个目标，但没有给定目标
    - 如果给了一个目标，但在这次遭遇战中没有留下该ID的怪物。
2. 玩家尝试打出一张给定名称的牌。如果不成功（即该牌不存在于玩家的手中，玩家没有足够的能量，或者该牌名没有映射到一张牌），该函数返回False。否则，该函数应执行剩余的步骤。
3. 这张牌的任何阻挡和力量都应该加到棋手身上。
4. 如果指定了一个目标：
    (a) 任何来自卡片的脆弱和虚弱都应适用于目标。
    (b) 伤害被计算并应用于目标。基本伤害是由牌造成的伤害量，加上玩家的力量。如果目标是脆弱的（即他们的脆弱状态非零），伤害应该乘以1.5，如果玩家是弱者（即他们的弱者状态非零），应该乘以0.75。在应用于目标之前，伤害量应该被转换为一个int。Int转换应该向下取整（注意，这是类型铸造到int的默认行为）。
    (c) 如果目标已被击败，则将其从怪物列表中移除。
5. 返回True，表示该函数成功执行。

注意：这些步骤的顺序很重要。例如，*在*计算应用于目标的伤害量*之前*，应该应用诸如力量、脆弱和虚弱等状态修饰符。在步骤1中检查所有会使牌无效的条件必须在步骤2之前发生，以便在牌的应用无效时不减少玩家的能量。

enemy_turn(self) - > None *(方法)*

这个方法试图让遭遇战中所有剩余的怪物采取一个行动。如果现在是玩家的回合，该方法立即返回。否则，每个怪物都会按以下方式进行轮流（按顺序）：

1. 怪物尝试其行动（见怪物类中的行动方法）。
2. 怪物的行动所产生的任何弱点和弱势都会加到玩家身上。
3. 任何由怪物行动产生的力量都会加到怪物身上。

4. 伤害被计算出来并应用到目标身上。基本伤害是由怪物的行动造成的伤害量，加上怪物的强度。如果玩家是脆弱的，伤害应该乘以1.5，如果怪物是脆弱的，应该乘以0.75。在应用于玩家之前，伤害量应该被转换为一个int。

一旦所有的怪物都采取了行动，这种方法就会开始一个新的回合。

5.4 主函数

当你的文件被运行时，主函数就会被运行，并管理整个游戏过程。关于主函数行为的例子，以及所需的确切提示，请参见作为a2.zip一部分提供的gameplay/文件夹。你也会发现a2_support.py中的常量对一些提示信息很有用（我们鼓励你也使用

在相关的地方定义你自己的全局常数）。主函数应该做以下工作：

1. 提示用户他们想要的玩家类型（"铁甲 "或 "无声"），并创建相关的玩家实例。你将在整个游戏中使用这个相同的球员实例。
2. 提示用户提供一个游戏文件。可以在a2_support.py中找到一个协助读取该文件的函数。
3. 对于文件中描述的每一个遭遇：
   (a) 用这组怪物开始一个新的遭遇，并展示它。
   (b) 在遭遇不再活动之前（或者在游戏因玩家失败而终止之前），应该不断地提示用户走棋。表1描述了可用的移动，以及它们应该引起的行为。如果在移动结束时，玩家已经赢得了这场遭遇战，那么在开始下一场遭遇战之前，他们的回合就应该结束。
4. 如果玩家通过了所有的遭遇战，程序应该以游戏胜利的信息终止。

你可以假设你的程序不会在不正确的输入下被测试，但播放命令可能会在无效的卡名和/或无效或缺失的target_id下被测试。

移动名称	行为
'结束转身'	当用户输入这个命令时，玩家的回合应该结束、 而敌人的回合应该被运行。如果玩家在敌方回合后被击败，游戏 应以游戏失败的信息终止。 否则就应该显示所产生的遭遇状态。
'与{deck | discard}相交'。	当用户输入'检查牌组'时，玩家的牌组应该是 印。当用户输入'检查弃牌'时，玩家的弃牌 应该打印出一堆。
'describe {card_name}'	当用户输入这条命令时，卡片的描述是 应该被打印出来。 需要注意的是 ，即使玩家不知道该牌的描述，也应该被打印出来。 没有所请求的卡的实例。
'播放 {card_name}' 或 '播放{card_name}{target_id}'	试图打出一张牌。如果指定了一个目标，但没有 如果在遭遇中发现有该ID的目标，应该打印出'无效目标'的文字 。如果卡片应用失败，应该打印卡片失败信息。否则，如果卡 片被成功应用，则 由此产生的遭遇状态应该被打印出来。

表1：用户可以在移动的提示下输入的四种类型的命令。

6 研究生任务

研究生需要实施另外两张牌和第三种类型的选手。本节简要介绍了所需的玩家和牌。由于这是一项高级任务，你要确定如何设计这些类，如何有效地测试它们，以及如何自己将它们整合到主 。

6.1 爆发

爆发是一种类型的卡，打出时需要2点能量，对目标造成9点伤害。

6.2 警惕性

警惕是一种卡，打出时需要消耗2个能量点，并为使用者增加8个格挡和1个强度。它 ，不需要一个目标。

6.3 观察者

守望者是一种类型的玩家，开始时有72HP，卡组包含4张攻击卡、4张防御卡、1张爆发卡和1张警戒卡。如果用户在玩家类型的提示中输入 "观察者"，那么使用的玩家应该是观察者实例。

7 评估和评分标准

这项作业评估了课程学习目标：

1. 应用程序结构，如变量、选择、迭代和子程序、
2. 应用基本的面向对象的概念，如类、实例和方法、
3. 阅读和分析他人编写的代码、
4. 分析一个问题并设计一个解决问题的算法、
5. 阅读和分析设计，并能够将设计转化为一个工作程序，以及
6. 应用测试和调试的技术。

7.1 标识细目

你在这项评估中的总成绩将由你的功能和风格分数组合而成。在这项作业中，功能和风格的权重是相同的，这意味着你至少应该在代码的正确风格设计上投入与你试图使其具有功能性同样多的时间。

7.2 功能标示

你的程序的功能将在50分的总分中被评分。与作业1一样，你的作业将通过一系列的测试，你的功能分数将与你通过的测试数量成正比。在作业到期日之前，你会得到一个功能测试的*子集*。你可能会因为部分工作方法，或者只实现了几个类而在每个类中得到部分分数。请注意，你不需要为了通过这项作业而实现主函数或Encounter类；以设计良好、风格鲜明的方式实现除Encounter以外的所有类，就足以获得这项作业的合格成绩。

你需要对你的程序进行*自己的*测试，以确保它符合作业中给出的*所有*规格。仅仅依靠所提供的测试，很可能导致你的程序在某些情况下失败，你会失去一些功能分数。注意：功能测试是自动化的，所以字符串输出需要与预期的*完全一致*。

你的程序必须在Python解释器（IDLE环境）中运行。部分解决方案将被标记，但如果你的代码中有错误，导致解释器不能执行你的程序，你的功能分数将为零。如果你的代码中有一部分导致解释器失败，请注释掉该代码，以便其余部分能够运行。你的程序必须使用 Python 3.11 解释器运行。如果它在另一个环境中运行 (例如 Python 3.10 或 PyCharm)，而不是在 Python 3.11 解释器中运行，你的功能分数将为零。

如果您的程序不能在Gradescope上运行，您将**得不到**功能方面的**分数。**您有责任及时上传到Gradescope，以调试任何可能导致Gradescope无法运行您提交的问题 。导师不会修复您代码的任何方面（包括文件名）。

7.3 样式标记

你的作业的风格将由导师评估。 风格的分数也将是满分50分。 对你的代码风格进行评分的主要考虑因素是代码是否容易理解，并显示出对面向对象编程概念的理解。在这项作业中，你的代码风格将根据以下标准进行评估。

- 可读性
  - 程序结构：代码的布局使其更容易阅读和遵循其逻辑。这包括使用空白来强调逻辑块。
  - 描述性的标识符名称：变量、常量、函数、类和方法的名称要清楚地描述它们在程序逻辑中代表的内容。**不要**对标识符使用所谓的 "*匈牙利符号*"。简而言之，这意味着不要将标识符的类型包含在其名称中（例如 `item_list`），而要使名称有意义。(例如，使用`items`，其中复数告知读者它是一个items的集合，它可以很容易地被改变为其他的集合而不是一个列表)。这个限制的主要原因是，大多数遵循 *匈牙利记号法*惯例的人，使用它的效果很差

(包括微软)。

  - 命名的常量：代码中所有非琐碎的固定值（字面常量）都由描述性的命名（符号）常量表示。

- 文件

  - 评论的清晰度：注释为代码提供有意义的描述。他们不应该重复那些通过阅读代码已经很明显的内容（例如：`# 将变量设置为0`）。注释不应该是冗长的或过多的，因为这样会使人难以理解代码的内容。

  - 翔实的文件串：每个类、方法和函数都应该有一个总结其目的的文档串。这包括描述参数和返回值，以便其他人能够理解如何正确使用该方法或函数。

  - 逻辑的描述：所有重要的代码块都应该有一个注释来解释逻辑是如何运作的。对于一个小的方法或函数，逻辑通常应该从代码和文档串中清楚地看到。对于长的或复杂的方法或函数，每个逻辑块都应该有一个行内注释来描述其逻辑。

结构将被评估为你的代码设计是否符合良好的面向对象编程实践。

- 面向对象的程序结构

  - 类和实例：对象被用作发送消息的实体，展示了对类和实例之间差异的理解。

  - 封闭性：类被设计成具有状态和行为的独立模块。方法只直接访问它们被调用的对象的状态。方法永远不会更新另一个对象的状态。

  - 继承与多态性：子类被设计成其超类的专门版本。子类扩展了它们的超类的行为，而没有重新实现行为，也没有破坏超类的行为或设计。子类重新定义适当方法的行为以扩展超类的类型。子类不会破坏其超类的接口。

- 算法逻辑

  - 单一的逻辑实例：在你的程序中不应该有重复的代码块。任何需要多次使用的代码都应该作为一个方法或函数来实现。

  - 变量范围：变量应该在需要它们的方法或函数中进行局部声明。属性应该在`init`方法中明确声明。除非是简化程序逻辑，否则应避免使用类变量。不应使用全局变量。

  - 控制结构：通过很好地使用控制结构 （如循环和条件语句），使逻辑结构简单清晰。

7.4 文件要求

在这项任务中，有大量的类和包含的方法需要你去实现。对于每一个，*你都必须*以docstring的形式*提供文档*。唯一的例外是在子类上重载的 方法，因为 python 的 docstrings 是继承的。

7.5 作业提交

本作业遵循与作业1相同的作业提交政策。请参考作业1的任务单。你必须以一个名为`a2.py`的Python文件的形式提交你的作业（使用这个名字--全部小写），而*不是其他*。你提交的文件将被自动运行以确定功能分数。如果你提交了一个*不同名字的*文件，测试将*失败*，你的功能分数将*为零*。*不要*提交

a2_support.py文件，或任何其他文件。*不要*提交任何形式的档案文件（如ZIP，RAR，7Z等）。

7.6 剽窃行为

本作业遵循与作业1相同的抄袭政策。请参考作业1的任务表。