Python Problems 08/09, Winter 2021/22

In this problem paper, we'll try to work on one bigger, contiguous problem. We will recreate the basis for a turn based adventure game[1]. Several players have come to the jungle to find a long forgotten temple and extract the treasure hidden within. On their way, they have to pass perils and they will find useful items that help them come closer to their goal. If a player passes a test, they gain a new item; if they fail to do so, they loose health.

The scope is a bit bigger than usual, so for this exercise, you **have time until after christmas** (i.e. you only need to be done by 2022-01-07).

The following tasks will guide you step by step to realizing this game concept. Don't worry by the length of the problem paper – the tasks themselves are actually rather simple ;)

Hint:
In the completed game, randomness will play a big role. However, this is difficult to debug. To understand an error, we usually need to be able to run the same code multiple times. For this reason, work with fixed values first. Only if everything works as intended and is tested for multiple input values, add the code for randomness (such as dice rolls).

Hint:
You'll need user input on some occasions. Experience shows that it takes a lot of time to enter and re-enter the same lines of data again and again. Like with randomness, I advise you to first work with fixed inputs and add user interfaces later.

1 Class Item (1 P)

Let's start by writing the code that describes an item.

An item has a name (e. g. *The Holy Hendgrenade of Antioch*). A player that owns an object gets a boost to one of their characteristics. Possible characteristics are health, strength, intelligence, speed and charisma. The magnitude of the boost is an integer value that can be different for different kinds of objects. Finally, there are consumable and permanent objects. Consumable objects can be used in only one peril, while permanent objects stay with a player throughout their journey.

Write the code for a Python class that can store these information. As shown in class, make it so that the instance attributes (name, characteristic, magnitude, consumable) are endered via the method `__init__`. Create two instances of that class that represent two different kinds of item. Add a method `__str__` to your class so that you can use your class in the following manner:

```
class Item :
    # your code here

items = []
items.append( Item("Holy Hendgrenade of Anitoch", "strength", 10, True) )
items.append( Item("Berret", "charisma", 3, False) )

```

[1] More precisely, we will recreate the core gameplay of *Fortune and Glory*, a wonderfully nerdy game inspired by the adventures of *Indiana Jones* and the pulp fiction of the 60s and 70s.

```
for item in items :
    print(item)
```

with the corresponding output:

```
Holy Hendgrenade of Anitoch (+10 strength, consumable)
Berret (+3 charisma, permanent)
```

2 Skeleton of the Player Class (2 P)

Now lets add the first lines of code that will represent a player. In the next few tasks, we will use the result of this task and expand on it.

A player has a name, a health-value, a strength-value, an intelligence-value, a speed-value and a charisma value, a progress-meter (i. e. the number of steps taken toward winning the game) as well as a bagpack (which, at first, shall be represented by an empty list).

We will, again, create a class `Player` that stores all of these data per player. This time, however, health, strength, intelligence, speed and charisma of a player shall be subject to randomness: All of these values shall be within a range `basevalue_X` $\pm$ `variation_X` (where `X` is one of the characteristics, e. g. strength). Convince yourself that it is prudent to store `basevalue_health`, `basevalue_strength`, ... as *class attributes* while the values actually describing the player should be *instance attributes*.

Now write the methods `__init__` and `__str__` for your class `Player`. For reasons we'll see later, it is best to put the characteristics in a instance attribute of type `dict` instead of using several instance attributes. This dict will store the values of the characteristics (health, strength, ...) as `int`s. Still, the backpack and the progress-meter should be instance attributes in their own right, not part of the `dict`.

You may expand on the following code:

```
class Player :
    baseHealth       = 10
    baseStrength     =  5
    baseIntelligence =  5
    baseSpeed        =  5
    baseCharisma     =  5

    fluctuationHealth       = 5
    fluctuationStrength     = 2
    fluctuationIntelligence = 2
    fluctuationSpeed        = 2
    fluctuationCharisma     = 2

    def __init__ (self, name) :
        # your code here

    def __str__ (self) :
        # your code here

players = []
players.append( Player("Dusky Joe") )
players.append( Player("Petra van Chameleon") )

```

```
24 print("Test access to player zero's health via characteristics dict:",
25       players[0].characteristics["health"])
26
27 for player in players :
28     print(player)
```

Possible output:

```
Test access to player zero's health via characteristics dict: 5
PLAYER:
        name                : Dusky Joe
        steps toward success: 0
        health              : 5
        strength            : 7
        intelligence        : 4
        speed               : 3
        charisma            : 6
        in their backpack:
        (nothing)

PLAYER:
        name                : Petra van Chameleon
        steps toward success: 0
        health              : 9
        strength            : 3
        intelligence        : 7
        speed               : 7
        charisma            : 5
        in their backpack:
        (nothing)
```

3 The Backpack (3 P)

Somewhere in your code, you should have written the line `self.backpack = []` to represent the fact that all players start with an empty backpack. Let's add the possibility to collect items!

In your class `Player`, write a method `add_item`. This method should take an instance of class `Item` as parameter and add it to the player's backpack, if the following conditions are satisfied:

- The backpack doesn't have unlimited space, but only holds a certain number of items, e. g. 3 items. It should only be allowed to add items to the backpack if that limit is not surpassed yet. (You may want to introduce a new class variable `backpack_limit` to your class `Player`.)
- While a player may have arbitrary many items of the same kind if the item is consumable, they cannot hold two identical permanent items. That is, a player may have two *Holy Hendgrenades of Antioch*, but only at most one *Berret*.

If the backpack limit is surpassed, the player should be asked whether they want to discard one of the items they already have in their backpack to pick up the new item. Make sure the second condition (no duplicate permanent items) is always satisfied.

Don't forget to update the method `__str__` of class `Player` to reflect the fact that a backpack is not always empty!

Example (assuming `Player.backpack_limit == 3`):

```
items = [Item("Holy Hendgrenade of Anitoch"    , "strength",     10, True),
         Item("Berret"                         , "charisma",     3 , False),
         Item("Bullwhip"                       , "strength"    , 1 , False),
         Item("Dune - The Desert Planet (book)", "intelligence", 3 , False)
]

player = Player("Dusky Joe")

player.add_item( items[0] )   # Holy Hendgrenade -- okay
player.add_item( items[1] )   # Berret -- okay
player.add_item( items[2] )   # Bullwhip -- okay

print("Test 1: duplicate permanent item")
player.add_item( items[1] )   # second Berret
print()

print("Test 2: too many items in backpack")
player.add_item( items[0] )   #
print()

print(player)
```

This code should give output similar to this:

```
Test 1: duplicate permanent item
This permanent item is already in your backpack

Test 2: too many items in backpack
Your backpack is full.
Which item do you want to leave behind?
0) Holy Hendgrenade of Anitoch (+10 strength, consumable)
1) Berret (+3 charisma, permanent)
2) Bullwhip (+1 strength, permanent)
3) Holy Hendgrenade of Anitoch (+10 strength, consumable)
Please enter the number of the item to discard now: 2

PLAYER:
        name                : Dusky Joe
        steps toward success: 0
        health              : 9
        strength            : 7
        intelligence        : 5
        speed               : 6
        charisma            : 3
        in their backpack:
        * Holy Hendgrenade of Anitoch (+10 strength, consumable)
        * Berret (+3 charisma, permanent)
        * Holy Hendgrenade of Anitoch (+10 strength, consumable)
```

4 Current Characteristics (2 P)

Write a method `get` for your class `Player`. This method should take a `str`ing as a parameter and return the current value of the characteristic identified by this parameter. It should also take the effect of permanent items into consideration, but ignore consumable items.

Hint:
You may want to implement helper methods `get_consumable_items` and `get_permanent_items`.

Example:

```
items = [Item("Holy Hendgrenade of Anitoch"    , "strength",     10, True),
         Item("Berret"                         , "charisma",     3 , False),
         Item("Bullwhip"                       , "strength"    , 1 , False),
         Item("Dune - The Desert Planet (book)", "intelligence", 3 , False)
]

player = Player("Dusky Joe")

print("Base strength:", player.get("strength"))

player.add_item( items[0] )
player.add_item( items[1] )
player.add_item( items[2] )

print("With items:", player.get("strength"))
```

Possible output:

```
Base strength: 5
With items: 6
```

5 Perils and Tasks (2 P)

Now let's tackle the perils our players have to overcome!

A *peril* comes with some text that describes the situation in which the players find themselves in. This description always ends in an *yes or no* question. Such a peril text might be:

The deep jungle suddenly ends on a cliff. In front of you, a chasm opens up and a deep gorge stands between you and the temple. There is a narrow and shaky suspension bridge over the gorge. Do you step on the bridge?

Associated with this situation are two *tasks*: one for when the player answers *yes* and one for when the answer is *no*.

A *task* is a collection of these information:

- Some text to be displayed before the player performs the test (e. g. *Half way over the bridge, you hear the hissing sound of ropes disintegrating: one of the ropes holding the bridge is about to snap! Run for your life!*
- The characteristic being tested (e. g. speed, if the player answered *yes*)

- A *cost*, i. e. an `int`eger, telling how much of the characteristic the player needs to have to pass the test
- A *penalty*, i. e. an `int`eger, telling how many health points are taken away from the player if they don't succeed
- Some text to be displayed if the player passes the test (e. g. *You dash forward as fast as you can. By a hair's breadth you make it to the other side before the bridge collapses. You are safe... for now...*)
- Some text to be displayed if the player does not pass the test (e. g. *In spite of your best effort you cannot make it to the other side before the bridge collapses. You fall down into the water. The fall hurts a lot.*)

Write the classes `Peril` and `Task`, together with their methods `__init__` and `__str__`. Begin with `Task`.

Example:

```
class Task :
    # your Code here

class Peril :
    # your Code here

perils = [
    Peril("The deep jungle suddenly ends on a cliff. In front of you, a chasm opens " + \
          "up and a deep gorge stands between you and the temple. There is a narrow " + \
          "and shaky suspension bridge over the gorge.\n" + \
          "Do you step on the bridge?",
          Task("speed", 16, 3,
               "Half way over the bridge, you hear the hissing sound of ropes " + \
               "disintegrating: one of the ropes holding the bridge is about to snap!\n" + \
               "Run for your life!",
               "You dash forward as fast as you can. By a hair's breadth you make it to " + \
               "the other side before the bridge collapses. You are safe... for now...",
               "In spite of your best effort you cannot make it to the other side before " + \
               "the bridge collapses. You fall down into the water. The fall hurts a lot."
          ),
          Task("skip turn", 1, 0,
               "You try to find another way across the gorge, which takes a lot of time.\n" + \
               "Skip one turn.",
               "", ""
          )
    )
]

print( perils[0] )
```

Possible output:

```
PERIL:
text: The deep jungle suddenly ends on a cliff. In front of you, a chasm opens up and a deep
 gorge stands between you and the temple. There is a narrow and shaky suspension bridge over
 the gorge.
Do you step on the bridge?
TASK:
```

```
category: speed
cost    : 16
penalty : 3
text    : Half way over the bridge, you hear the hissing sound of ropes disintegrating: one
 of the ropes holding the bridge is about to snap!
Run for your life!
on pass : You dash forward as fast as you can. By a hair's breadth you make it to the other
 side before the bridge collapses. You are safe... for now...
on fail : In spite of your best effort you cannot make it to the other side before the bridge
 collapses. You fall down into the water. The fall hurts a lot.
TASK:
category: skip turn
cost    : 1
penalty : 0
text    : You try to find another way across the gorge, which takes a lot of time.
Skip one turn.
on pass :
on fail :
```

6 Meeting the Requirements for a Task (3 P)

In the "real life game", players roll a number of dice to see whether or not they passed a test. The number of dice depends on their caracteristics, their permanent items in the backpack and on the consumable items they want to use for a given task. We will learn later, how many dice may be rolled in detail.

From all the dice rolled, only those showing 5 or 6 pips are counted. Both, fives and sixes count as *one success*. If the collected successes meet the *cost* of a task, it counts as passed. If the last roll did not meet the cost of a task but at least one success was collected, a player may continue rolling dice. Successes collected up to this point are not lost. Only if a roll gave no success at all, the task counts as failed.

Example 1:
A task has a cost of 3. Petra faces this problem with four dice. Her first roll is 1, 5, 6, 2. She gets two success tokens and continues to roll. Her second roll shows 6, 2, 4, 3. She thus has three successes in total and passes the test.

Example 2:
A task has a cost of 6. Dusky faces this problem with three dice. He rolls 2, 3 and 1. Since he has not a single success, he may not continue rolling but has failed the test.

Write a function `task_passed` (not a class method!) that replicates this rule. Your function should take two arguments (number of dice used and cost) and return a `bool`. The return value should be `True` if the task is passed and `False` if not.

7 Performing a Test (4 P)

Now make it so that a player can make an attempt at a peril. For that, add a method `face_peril` to your class `Player`. The method should take one parameter which identifies the peril faced. Your method should do the following things, in sequence:

- Print the peril description

- Ask the player whether they want to attempt the yes- or the no-task
- Show the characteristic and cost of the test
- Ask the player whether they want to use an item
  - if yes, also ask, which one and ...
  - ... remove that item from the player's backpack
- Check whether the task was passed or failed
- If the player was successful ...
  - print the success text from the instance of `Task`
  - increase the progress counter
  - add a new, random item to their backpack (using the method `add_item` from problem 3). You may use a global list of items in the game for this.
- If the player did not succeed ...
  - print the fail text from the instance of `Task`
  - reduce the health counter by the indicated cost

For this you need to know how many dice a player may use. This number is simply the player's value in the characteristic being tested plus the value from permanent items with the same characteristic, plus used consumable items with the same characteristic.

Example:
Dusky Joe faces a strength task. He has a strength of 5 and carries the Bull Whip (+1 strength, permanent), the Berret (+3 charisma, permanent) and the Holy Hendgrenade (+10 strength, consumable). If he uses the Holy Hendgrenade, he will have a total of 16 dice to roll. If he saves it for later, he may roll up 6 dice. The Berret does not matter in this test.

8 A Playable Game (0 P, but a lot of fun)

Expand on the code you have so far, to create a full, playable game. You may add any features you can think of and that you can implement.