The deadline for this exercise sheet is **Monday, 04.06.2018, 12:00.**

Introductory Words

Remember that from this homework onwards, you need proper documentation to pass the homework. The documentation doesn't need to be *perfect*, but everything that needs a docstring, should have a docstring. ¿¿¿¿¿¿¿ 5e08551c973dc3446cbfbd884decb4fe34f0d386

1 Warm-Up: The Land Before Time

Many many years ago, our planet was the habitat of many different types of dinosaurs.
We modeled several kinds of them and their behaviour (in a manner that may or may not be a little simplified) in a file named `dinos.py`.
Look at the dino classes given in the file. State for each class the following:

- All superclasses
- All subclasses
- All classes it extends
- All functions it can use
- All functions it overrides

Figure 1: Somewhat related to the task at hand.
Taken from naturesfancy at redbubble
`https://www.redbubble.com/de/people/naturesfancy/works/16720425-cute-t-rex-dinosaur-riding-red-bicycle`

2 I want to ride my bicycle, I want to ride my bike

Implement three classes: *Bike*, *Bicycle* and *Motorbike*.
Bike is the superclass of both *Bicycle* and *Motorbike*.
A *Bike* has a number of seats and a number of gears. It can start being ridden and end being ridden. Also, one is able to change the gears (to a gear number that actually exists.)
A *Bicycle*, additionally to the *Bike*, has a bell that can be rung.
A *Motorbike*, additionally to the *Bike*, has a tank that can be filled.
Write a module of the classes according to the description and add another module that tests them with some values. Don't forget to include proper documentation!

Additionally to the *Bike* functions, a bell on the *Bicycle* can be rung.
A *Motorbike*, additionally to the *Bike*, has a tank that can be either full or

empty. When you ride the *Motorbike*, the tank is empty afterwards.
Write a module of the classes according to the description and add another module that tests them with some values.
Think about which of your attributes need to be public and which should be set private.
Don't forget to include proper documentation!
Note: There is not only one solution. Your solution (e.g. the bell) can be more or less complex depending on how complex you want it to be.

3 To Battle

Let's play a game! In the .zip archive you will find the file `battle.py`. In this file, we implemented a game loop that has two knights battle. But there is no knight template yet, and this is where you come in!
The basic concept of battle is as follows: Each player is asked to create a knight, which consists of a name and attribute points (more on that below). Then one player is chosen at random to have the first turn. Each turn the remaining health of the players is displayed. The active player can choose an action (an attack) to perform, and the other player will take damage accordingly. This loop repeats for as long as both players are alive. After that the winning player is displayed, and you can choose to play another round.
Familiarise yourself with the code in `battle.py` to some extent. We tried to document it thoroughly, so the code is (hopefully) well understandable, but if you struggle with understanding the code just send us a message, and also you *do not* have to understand the code in full extent to do this homework. So do not worry if you cannot follow the code line by line.
You will also find the module `util` in the .zip. Make sure that it is in the same folder as the battle module, when running it. It contains two utility functions that unnecessarily bloated the battle code.

Every knight has the attributes *vitality, strength and luck* and a fixed amount of attribute points, that they can spend on their attributes. They always have to use all of their attribute points. You can freely choose how many attribute points your knights have (but all knights have the same).
The detailed outline of the class is below.

Your task is to write a module named `knights`. This module shall fullfill the following requirements:

- `MAX_ATTRIBUTE_POINTS`: a constant integer that holds how many attribute points a knight has
- `ATTRIBUTES`: a constant list that holds all attribute names as strings, in the same order they appear in the constructor
- `KnightError`: A special exception that is thrown when there is an error knighting a new knight. All custom errors extend the `Exception` class.

This custom error needs no properties, nor methods (not even `__init__`). It is enough to extend `Exception`.

- `Knight`: The very important class. This is the template for all our knights that we are going to send into battle.

Now, onto the important part: The **Knight class**. A knight instance is initialised with a **name**, the **vitality** attribute points, the **strength** points, and the **luck** points. The class needs to implement:

- **properties**: `name, health, strength, luck, crit_modifier, available_actions`
  - `name`: the name of the knight
  - `health`: this should be a property of the knight, which is read-only. The only way this is altered is through the `take_damage` function. It should be initialised to the *floored integer* value of $vitality * 1.5$, where `vitality` is the passed parameter on initialisation.
  - `strength`: the points this knight has in strength
  - `luck`: the points this knight has in luck, multiplied by 2. This is the critical hit chance in percent.
  - `crit_modifier`: The amount the attacks are multiplied with when they critically strike. We chose 3 for this.
  - `available_actions`: A list of all available actions this knight can take. Note that this should be a list of *functions* and not of strings, or other types.
- `__init__()`: this one should be mostly clear. It also should throw a `KnightError` if the sum of all attribute arguments are unequal to the set `MAX_ATTRIBUTE_POINTS` – so either if it is below or above.
- `alive()`: a function that returns whether the knight is still alive. The knight dies when their health is or falls below 0.
- `light_attack()`: A lighter attack with smaller damage and a higher critical strike chance. The damage is calculated by taking the *floored integer* value of the formula: $1 + 0.15 * strength$. If the hit was a critical hit, the damage is multiplied by the `crit_modifier` before flooring it. The critical chance is doubled for the light attack.
  The function returns a tuple consisting of the damage and a boolean whether the attack was critical, in this order.
- `heavy_hit()`: A heavier attack with higher damage, but lower critical chance. The damage is calculated by taking the *floored integer* value of the formula $3 + 0.4 * strength$.If the hit was a critical hit, the damage is multiplied by the `crit_modifier` before flooring it.
  The function returns a tuple consisting of the damage and a boolean whether the attack was critical, in this order.

- `all_actions()`: This function should return the list of all actions this knight can take.
- The string method should return only the name of the knight.

Feel free to play around with the values or implement more attributes. As long as the three mentioned above are implemented, everything (should) work fine. The battle code should also be flexible enough to deal with more attributes, if you put them into the `ATTRIBUTES` list, and the order is the same as in the `__init__` function.

3.1 Bonus: A New Challenger Approaches

Now that we have a basic knight, let us introduce some diversity. Build at least one more class that extends the basic `Knight` class. Also add a `description` *class method*, which shortly describes the knight to the basic `Knight` class and every subclass that you write. A class method is basically a method inside the class body, which *does not* take the self argument. It can hence not be called on an instance of those classes, but you can call it by doing `Knight.description()`. `description` shall take no arguments and return a string. The description string should shortly outline what the knight can do or how it differs from the basic knight.
One idea would be a berserker class, which deals more damage, but has less health.

The battle code should be flexible enough to deal with new attack methods and any subclass of Knight, if implemented correctly.