The deadline for this exercise sheet is **Tuesday, 04.06.2018, 14:00.**

Contents

DISCLAIMER: These are all just suggestions and not necessarily a complete or the best approach to a solution. It just offers hints, general approaches and ideas.
These are also a lot of pages of one-liners.

Task 2: Changing Gears to a "valid number"

Whenever the new gear is not within the number of gears the bike has, throw an exception.

Task 2: Ringing the Bell

Ringing the bell can be done by including a new variable or even class as the bell, but it is also fine to just print something whenever the bell is rung.

Task 2: What Class Properties and Functions Should I Include?

Properties correspond to what kinds of things a class *has*. However, it often happens that there are a few more properties only used for internal purposes that will need to be added to allow for a certain function to work.
Functions correspond to what someone *can do* with the class.

Task 3: The constants

The constants should be constants of the module, not the class. Just define them as variables atop your `knights` module before defining the class.
Technically, they could also be made constants of the class

Task 3: KnightError

pass

;)

Task 3: Knight Class Function Heads & Returns

- `__init__(self)`: does not return
- `alive(self)`: returns boolean
- `take_damage(self, damage)`: does not return / could return current health
- `light_attack(self)`: returns tuple (damage, crit_boolean)
- `heavy_hit(self)`: returns tuple (damage, crit_boolean)
- `all_actions(self)`: returns list of function objects
- `__str__(self)`: returns string

Task 3: The Health Property

Remember the @property decorator and how to define a private attribute. You can use the decorator and decorate a function named `health` with it, turning it into a property. This function then returns the value of the private health variable. This getter function is all you need. You don't need a setter function, since take_damage can access the private health property.

Task 3: The Critical Hit

You can use `random.randint` to calculate whether it was a critical hit. Generate a random integer between 0 and 99 and check whether it is below the critical hit chance. If it is, that is a critical hit! So a critical chance of 100 will always result in a hit, while a chance of 1 will only happen if `randint` returns 0.

Task 3: all_actions() & available_actions

The trick here is to store the function references in a list. Since you have the actions `light_attack` and `heavy_hit`, the list can be defined as
`self.available_actions = [self.light_attack, self.heavy_attack]`
Note that the parantheses are missing. This passes the actual *function object* (They are everywhere...). This function object is this very specific function inside this very specific instance, so we can do something like
`my_knight.all_actions()[0]()`
See how we just put a parentheses at the item we got from the list? We are just going to call the function object that is referenced in the list and it will execute that specific function in the specific instance of `my_knight` we got it from.

Task 3.1 Bonus: A Subclass

Remember that you can call `super().__init__` and afterwards add any additional attributes you need for your class.

Task 3.1 Bonus: Description()

description(): returns string *Note there is no self ;)*

Task 3.1 Bonus: New Attacks

If you want to give your knight new attacks, don't forget to update the `available_actions` list with the newly defined function, so that the battle code can find and call it.