Exercise 3

due 16.11.2017

This exercise covers Chapters 0-3 of the tutorial. Please submit your solution until 16.11., 23:59, via e-mail to `programming-l1-ws1718@ims.uni-stuttgart.de` as a **plain text** and/or Python file (which should end on `.txt` or `.py`). Please also submit in groups of **at least 3 students**, and clearly indicate the **names and immatriculation** numbers of all involved students. Submissions that do not fulfill these requirements are not accepted.

New: Please include in your submission how much time it took you (roughly) to complete the exercise. Thanks!

Questions

1. The functions you'll implement in the following scenario could be used to check whether a certain book exists in a library or not. Please submit a separate file for each part, and include the print statements below in your files.

   (a) The function takes two arguments (one int, one string) and returns a boolean value. The int-argument represents the year of publication, and the string argument the author of the book. Our library only contains books written by Franz Kafka that have been published before 1645. You may assume that the name is always written in the same way (users always write `'Franz Kafka'`, if they mean Franz Kafka.).

   ```
   print(exists(1543, "Franz Kafka"))        # should print True
   print(exists(1543, "Franz Beckenbauer")) # should return False
   ```

   (b) Some users might not know a date or a name. In this case, they use `None` as an argument value. Make sure that your function still returns something. If both arguments are missing, the function should always return False.

   ```
   print(exists(1543, None))                # should return True
   print(exists(None, "Franz Kafka"))       # should return True
   print(exists(None, "Franz Beckenbauer")) # should return False
   print(exists(None, None))                # should return False
   ```

(c) **Bonus question.** We now want to extend the function such that the arguments can be supplied in any order. The function should still behave properly. You may assume that the arguments are filled from left to right, i.e., if a single argument is given, it's always the first. In this case, the second is None.

```
print(exists(1800, None))          # should return False
print(exists(1543, None))          # should return True
print(exists("Franz Kafka", None)) # should return True
print(exists("Franz Kafka", 1543)) # should return True
print(exists("Franz Kafka", 1800)) # should return False
print(exists(1543, "Franz Kafka")) # should return True
```

2. We will now write a small program that records our financial situation. Initially, the program should set a variable to store what we have (100 units). Then, it should offer four functions:

   - `balance()` returns the current value of the variable.
   - `withdraw(amount)` removes the number given in the argument from our account. This function should be safe against wrong argument types: If a user calls the function with a string, the function should print an error message and not touch the account (i.e., not withdraw anything).
   - `interest()` calculates 10 percent interest (i.e., it calculates the amount of 10% of the current value and adds it to the account).
   - `project(rate)` returns the projected amount with the given interest rate, after interest has been applied 10 times. This function should not change our current status, but apply the interest rate (in the argument `rate`) 10 times (the function should take into account that the second application of the rate is based on the result of the first application of the rate, i.e., take into account interest on interest/Zinseszins).

   The following shows a possible sequence:

```
print(balance()) # prints 100
withdraw(10)
print(balance()) # prints 90
withdraw("ten")  # prints "Not a number"
interest()
print(balance()) # prints 99.0
withdraw(9)
print(balance()) # prints 90.0
withdraw(-20)
print(balance()) # prints 110.0
print(project(0.2))  # prints 567.5758...
print(balance()) # prints 110.0
```

3. We want to generate some ASCII art. In ASCII art, each character is considered to be a single Pixel. Multiple lines, in which some pixels are filled and some not, make an image. The following image shows the letters D and H in ASCII art.

```
.............................
....#######....##....##......
....##...###...##....##......
....##...###...########......
....##...###...##....##......
....#######....##....##......
.............................
```

We will assume that a single line is twenty characters long (line width), and we'll be using the dot `.` for an empty pixel and the hashtag `#` for a full pixel, like in the picture above.

The following program contains two functions and a few variables. Unfortunately, the functions are named `f1()` and `f2()`, which is not very memorable. Can you assign them better, more descriptive names (the names should of course reflect what they do)?

```python
linewidth = 20
empty = "."
full = "#"

def f1():
    print(empty*linewidth)

def f2(begin,end):
    s = ""
    for i in range(0,begin):
        s = s + empty
    for i in range(begin,end):
        s = s + full
    for i in range(end,linewidth):
        s = s + empty
    print(s)
```

Please write four functions in the following. Each function may call `f1()` and `f2()` (or your renamed versions).

(a) Please write a function that produces a diagonal line, from top left to bottom right.

(b) Change the direction of the line, from bottom left to top right.

(c) Please write a **function** `rect(x)` that takes one argument. The function should then print a quadratic image, in which a single quadratic rectangle is filled, starting at the top left. A call like `rect(5)` should produce the following image:

```
#####...............
#####...............
#####...............
#####...............
#####...............
....................
....................
....................
....................
....................
....................
....................
....................
....................
....................
....................
....................
....................
....................
....................
```

(d) Now extend `rect()`, so that it takes three arguments. The first one specifies the size of the rectangle (as before), the second argument specifies the vertical position and the third the horizontal position (of the top-left corner of the rectangle).