Exercise 4

due 23.11.2017

This exercise covers Chapters 0-4 of the tutorial. Please submit your solution until 23.11., 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. Please include in your submission how much **time** it took you (roughly) to complete the exercise. Thanks!

Outlook: The next exercise will also cover chapters 0 to 4 of the tutorial, in order to let everything sink in and advance a bit slower.

Questions

1. What is type and value of the expressions in lines 4-16, i.e., to what evaluate the expressions in those lines? Obviously, you can input this in an interpreter, but you'll learn more if you think about what you expect first. If your expectation deviates from what the interpreter gives you – find out why either by asking us or your group mates.

```
1  v = 'a'
2  l = ['a','b','c']
3  n = 17
4  []          # type: ? value: ?
5  v+'b'
6  l+[]
7  v*2
8  l*2
9  n*2
10 l[1]
11 l[1:2]
12 v == l[1]
13 l[-2]
14 l[1:1]
```

```
len(l)
l.append(l)
```

2. Let's revisit the ATM algorithm from two weeks ago. We are now given a list of banknote values at the beginning. For the euro, this list is `l = [5,10,20,50,100,200,500]`. Can you improve your algorithm, so that it uses this list? You may use our solution as a starting point (which is given below), but you should make sure that your algorithm also works for other denominations. For instance, in the fantastical land of Timonia, the denominations 5, 15, 25, 75 and 125 are in use (in which case your program would have a variable `l=[5,15,25,75,125]` at the beginning). In the Nathalands, all denominations are prime numbers: 1, 3, 5, 7, 11, 13 and 17. We still want as few bills as possible, i.e., high value bills have priority.

```

# This program runs forever
while(True):
    # Get the users input and convert it directly into an int
    amount = int(input("Enter amount you want to withdraw: "))

    # This is not very elegant, but we go over every possible
    # value. Each time, we calculate how many banknotes we need
    # of the given type (variables starting with ret_), and the
    # amount that remains.

    # 500s
    ret_500 = amount // 500
    amount = amount % 500

    # 200s
    ret_200 = amount // 200
    amount = amount % 200

    # 100s
    ret_100 = amount // 100
    amount = amount % 100

    # 50s
    ret_50 = amount // 50
    amount = amount % 50

    # 20s
    ret_20 = amount // 20
    amount = amount % 20

```

```
    # 10s
    ret_10 = amount // 10
    amount = amount % 10

    # 5s
    ret_5 = amount // 5
    amount = amount % 5

    # If there is something left we show
    # an error message.
    if amount > 0:
        print("The entered amount can not be withdrawn.")
    # If not, we print results, starting with the lowest
    # banknote type
    else:
        if ret_5 > 0:
            print("5: "+str(ret_5))
        if ret_10 > 0:
            print("10: "+str(ret_10))
        if ret_20 > 0:
            print("20: "+str(ret_20))
        if ret_50 > 0:
            print("50: "+str(ret_50))
        if ret_100 > 0:
            print("100: "+str(ret_100))
        if ret_200 > 0:
            print("200: "+str(ret_200))
        if ret_500 > 0:
            print("500: "+str(ret_500))
```

Bonus Exercise In Nilia, ATMs print banknotes of certain denominations on the fly, according to the following rule (but without upper limit!): A number n is a valid bill, if it is dividable by 7 *and* the number of its digits (using the decimal system) is odd. Valid denominations are (among others) 7, 105, 112, 700, 10.003, …. Invalid denominations (i.e., these should not be printed) are 14, 15, 17, 21, 3003, …(and many many more).

3. Again, ASCII art. What we could not do before is printing full pixels in different, non-continuous lines. We will do that now.

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

def emptyline():
    print(empty*linewidth)
```

```
7
8  def fromto(begin,end):
9      s = ""
10     for i in range(0,begin):
11         s = s + empty
12     for i in range(begin,end):
13         s = s + full
14     for i in range(end,linewidth):
15         s = s + empty
16     print(s)
```

(a) Write a function `dots()` that prints multiple, unconnected filled pixels in a line. The function should take a list as an argument which contains the x positions of the filled pixels on the line. A call to `dots([1,5,12])` should print a line that has pixels (i.e., #-characters) on position 1, 5 and 12, and is otherwise empty.

(b) Write a function called `cross()` that uses `dots()` to print such a cross:

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