General

Infinite loops

Take care to avoid infinite loops. An infinite loop is a loop that never stops executing: in most of the cases it concerns a `while`-loop where the statements inside the loop never take care to make the `while`-condition `False` after some time. As an example, take a look at the following code snippet

```
>>> i = 0
>>> a = 0
>>> while i < 4:
...     a += 1
```

Because the statement `a += 1` will never cause the value of the variable `i` to become larger than or equal to 4, the condition `i < 4` will evaluate to `True` forever.

Tip: If you work with PyCharm, you know that a program that was started is still running from the red square to the left of the Console or top right in the navigation bar. If you click the red square, you force the program to stop.

Counting starts at zero

Computer scientists by default start counting from zero, not from one. Python follows this tradition in many of its design decisions. As an example, the built-in function `range` generates a sequences of successive integers that starts at zero, if you only pass a single argument to the function. Here's how you count to 5 in Python

```
>>> for i in range(6):
...     print(i)
...
0
1
2
3
4
5
```

If you want counting to start at another value, you can pass this value as an extra argument to the `range` function.

```
>>> for i in range(1, 6):
...     print(i)
...
1
2
3
4
5
```

However, it is considered a more *Pythonic* solution to write the above as

```
>>> for i in range(5):
...     print(i + 1)
...
1
2
3
4
5
```

Early warning

Read multiple lines from input

If you know in advance how many lines must be read from input, you may use the following strategy

```
>>> lines = 4
>>> for _ in range(lines):
...     line = input()
...     # process the line
...
```

If the number of lines that must be read from input is not known in advance, but you know for example that the last line is an empty line, you may use the following strategy

```
>>> line = input()
>>> while line:
...     # process the line
...     line = input()
...
```

The frog prince

Premature abortion of loops

In Python, you can use the statements `break` and `continue` to abort a loop before it has come to completion. In general, however, these statements are considered bad programming style. So you'd better avoid them at all, as they will cost you points when used during an evaluation or an exam.

One situation where you may want a premature abortion of a loop occurs when you want to find a solution by trying all possible cases, and stop as soon as one solution has been found. Instead of using `break` or `continue` in this case, it is better to use an additional Boolean variable that indicates whether the solution has already been found.

```
>>> found = False
>>> while not found:
...     if <solution found>:  #solution found represents a condition
...         found = True
...
```

As soon as the solution has been found (represented here by the fact that the condition *solution found* evaluates to `True`), the variable `found` is assigned the value `True`. As a result, the `while`-loop ends the next time the `while`-condition is evaluated after the current iteration.

Rounding up floats

The `math` module contains a function `ceil` that can be used to round up *floating point* numbers. This function returns an integer (`int`).

```
>>> import math
>>> math.ceil(3.2)
4
>>> math.ceil(3.7)
4
```

The `math` module also contains the complementary function `floor` that can be used to round down *floating point* numbers. Use the built-in function `round` for the classic way of rounding *floating point* numbers.

Elevator paradox

Conditional expressions

Most programming languages have a ternary operator that is called the **conditional expression**. Its result depends on a condition `test`. Python uses the following syntax for conditional expressions:

```
<expr_true> if <test> else <expr_false>
```

Note that the order of the condition and the expressions differs from most other programming languages, where the condition `test` comes first in the syntax.

If the condition `test` evaluates to `True`, the expression `expr_true` yields the result of the conditional expression. Otherwise, the expression `expr_false` yields the result of the conditional expression.

The result of a conditional expression may for example be assigned to a variable `result`:

```
result = <expr_true> if <test> else <expr_false>
```

Specific information

For this assignment it is a good idea to first convert the number of hours and minutes on a 24-hour clock into a single variable `minutes_since_midnight` that indicates the number of minutes that has elapsed since the start of the day (midnight). For example, if time is *15:20*, the variable `minutes_since_midnight` is assigned the value $15 \times 60 + 20 = 920$. This makes it a lot easier to increase (or decrease) the timestamp with a fixed number of minutes m: simply add (subtract) m to the variable `minutes_since_midnight`.

Using integer division and the modulo operator, the variable `minutes_since_midnight` can be decomposed again in the number of hours and minutes on a 24-hour clock.

```
>>> hours = 15
>>> minutes = 20
>>> minutes_since_midnight = 60 * hours + minutes
>>> minutes_since_midnight
920
>>> minutes_since_midnight += 50            # increase the number of minutes by 50
>>> minutes_since_midnight
970
>>> (minutes_since_midnight // 60) % 24     # number of hours on a 24-hour clock
16
>>> minutes_since_midnight % 60             # number of minutes on a 24-hour clock
10
```

Note that we have added an extra modulo operation (`% 24`) when deriving the number of hours on a 24-hour clock. This operation ensures that the derivation still works in case the number of minutes since midnight exceeds the total number of minutes in a day (24 hours).

Billiards table

Specific information

The easiest way to solve this problem is to simulate the (x, y) coordinate of the billiard ball step by step. You can do this by keeping track of the position of the ball on the billiards table using two variables x and y.

The variables x and y are then adjusted by 1 or -1 in each step of the simulation, depending on the direction in which the ball moves.

After each simulation step you can check whether the ball bounces on a cushion or disappears in a pocket. If that's the case, an appropriate output message can be generated. If the ball bounces on a cushion, its

direction changes.