Bowling project

Petr Pošík

Department of Cybernetics, FEE CTU in Prague

EECS, BE5B33PRG: Programming Essentials, 2015

Specifications

Create function `format_score_card` that would take a **legal sequence** of at most 21 bowling throws (i.e. sequence of the pin counts shot by a ball in each throw) and would output a string which - when printed - would 'graphically' represent the score card. It should handle even incomplete games; in that case it should fill in only those scores which are known.

Example:

```
[1,4,4,5,6,4,5,5,10,0,1,7,3,6,4,10,2,8,6]

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|1|4|4|5|6|/|5|/| |X|0|1|7|/|6|/| |X|2|/|6|
| +-| +-| +-| +-| +-| +-| +-| +-| +-| +-+-|
| 5 | 14| 29| 49| 60| 61| 77| 97|117| 133 |
+---+---+---+---+---+---+---+---+---+-----+
```

Hint: Look at http://www.bowlinggenius.com/ (http://www.bowlinggenius.com/). You shall notice that while the first row is updated after each throw, the second row is updated only when a final score for each frame can be determined. This should suggest the natural elements over which you should iterate in the respective functions.

Initial setup of the testing framework

In [1]:
```
from testing import *
set_default_test_verbosity(0)
```

Function `format_score_card()`

How would you implement function `format_score_card()` on the top level?

Let's implement the function like this:

In [2]:
```
def format_score_card(throws):
    rows = []
    rows.append(format_top_rule())
    rows.append(format_row_1(throws))
    rows.append(format_mid_rule())
    rows.append(format_row_2(throws))
    rows.append(format_bot_rule())
    return '\n'.join(rows)
```

Our task is to implement those `format_xxxxx` functions.

Function `format_top_rule()`

This function is easy. We could return just the string consisting of the right number of + and `-`. But for the ease of testing, let's make it slightly more general. We will equip the function with a single parameter, number of frames, with default value equal to 10. This will allow us to test the function like this:

In [3]:
```python
def test_top_rule():
    test_equal(format_top_rule(1), '+-+-+-+')
    test_equal(format_top_rule(2), '+-+-+-+-+-+')
    test_equal(format_top_rule(), '+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+')
```

The tests cover the following cases:

- Single (last) frame could be composed of 3 throws, i.e. the table shall contain 3 cells in the top row, hence `+-+-+-+`.
- Game with 2 frames will contain one regular and one last frame, it can thus consist of 5 throws, i.e. the table shall contain 5 cells in the top row, hence `+-+-+-+-+-+`.
- The last test checks the usual case with 10 frames.

Let's implement the function such that it fulfills the tests:

In [4]:
```python
def format_top_rule(n_frames=10):
    """Return string representing the top border of the bowling score card"""
    return '+-+-' * (n_frames-1) + '+-+-+-+'
test_top_rule()
```

```
...
```

In [5]:
```python
print(format_top_rule())
```

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Function `format_mid_rule()`

The tests and implementation of function `format_mid_rule()` are analogous:

In [6]:
```python
def test_mid_rule():
    test_equal(format_mid_rule(1), '| +-+-|')
    test_equal(format_mid_rule(2), '| +-| +-+-|')
    test_equal(format_mid_rule(), '| +-| +-| +-| +-| +-| +-| +-| +-| +-| +-+-|')
```

And the implementation:

In [7]:
```python
def format_mid_rule(n_frames=10):
    """Return string representing the middle rule of the bowling score card"""
    return '| +-' * (n_frames-1) + '| +-+-|'
test_mid_rule()
```

```
...
```

In [8]:
```python
print(format_top_rule())
print(format_mid_rule())
```

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| +-| +-| +-| +-| +-| +-| +-| +-| +-| +-+-|
```

Function `format_bot_rule()`

And once again, almost the same process:

In [9]:
```python
def test_bot_rule():
    test_equal(format_bot_rule(1), '+-----+')
    test_equal(format_bot_rule(2), '+---+-----+')
    test_equal(format_bot_rule(), '+---+---+---+---+---+---+---+---+---+-----+')
```

And the implementation:

In [10]:
```python
def format_bot_rule(n_frames=10):
    """Return string representing the bottom rule of the bowling score card"""
    return '+---' * (n_frames-1) + '+-----+'
test_bot_rule()
```

```
...
```

In [11]:
```python
print(format_top_rule())
print(format_mid_rule())
print(format_bot_rule())
```

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| +-| +-| +-| +-| +-| +-| +-| +-| +-| +-+-|
+---+---+---+---+---+---+---+---+---+-----+
```

Functions `format_row_1()` and `format_row_2()`

These 2 functions are much more complex (and interesting!) than the 3 functions above. In general, both depend on the list of actual `throws` scored by a player so far. And both combine 2 different concerns:

- what to print, and
- how to format it properly.

How should we deal with that?

Let's implement these 2 concerns in separate functions like this:

In [12]:
```python
def format_row_1(throws):
    chars = throws_to_chars(throws)
    return format_throw_chars(chars)

def format_row_2(throws):
    scores = throws_to_scores(throws)
    return format_frame_scores(scores)
```

Although the 2 functions look similar, the inner data structure is different:

- Function `format_row_1()` is responsible for the first line where we shall print the numbers of pins knocked down in individual throws. The logic, however, is not easy: we should detect spares and strikes and shall replace the numbers by symbols like `'/'` for spare, `'X'` for strike; and for strike we shall also insert some spaces `' '`. Thus, function `throws_to_chars` shall return a list of characters, i.e. list of strings of length 1. Function `format_throw_chars` shall take this (perhaps incomplete) sequence of chars and produce correctly formatted string representing the first line, i.e. it shall insert some `|` characters to the right place and take care of the right length of the string.

- Function `format_row_2()` is responsible for the second line of the score card, i.e. it shall print the cumulative scores for individual frames. Again it is not trivial to determine the individual frame scores, so this responsibility is separated into function `throws_to_scores` which produces a list of integers, which is then passed to function `format_frame_scores` which takes care of correct formatting.

Let's first deal with the 2 functions responsible for formatting. Since we decomposed the task, we can easily test them if we provide them with the inputs that shall be printed by them.

Function `format_throw_chars()`

Function takes a list of at most 21 characters to be printed in the first line of the score card and returns a string representing the first line, including all the separators `'|'`. Again, we shall equip it with the possibility to set the number of frames in the game. Let's start with some tests.

```
In [13]: def test_format_throw_chars():
             test_equal(format_throw_chars([], 1), '| | | |')
             test_equal(format_throw_chars([], 2), '| | | | | |')
             test_equal(format_throw_chars([]), '| | | | | | | | | | | | | | | | | | | | | |')
             test_equal(format_throw_chars(['1'], 1), '|1| | |')
             test_equal(format_throw_chars(['1', '4'], 1), '|1|4| |')
             test_equal(format_throw_chars(['1', '/'], 1), '|1|/| |')
             test_equal(format_throw_chars(['X', 'X', 'X'], 1), '|X|X|X|')
             test_equal(format_throw_chars([' ', 'X', 'X', 'X', 'X'], 2), '| |X|X|X|X|')
             test_equal(format_throw_chars([' ', 'X', ' ', 'X', 'X', 'X', 'X'], 3), '| |X| |X|X|
         X|X|')
             test_equal(format_throw_chars(['1', '4', '4', '5', '6', '/', '5', '/', ' ', 'X',
                                            '0', '1', '7', '/', '6', '/', ' ', 'X', '2', '/', '6
         ']),
                                            '|1|4|4|5|6|/|5|/| |X|0|1|7|/|6|/| |X|2|/|6|')
```

And the implementation:

```
In [14]: def format_throw_chars(chars, n_frames=10):
             """Return string representing the first line of scorecard"""
             # If we do not have enough chars to fill in the first line,
             # append spaces
             n_cells = 2*n_frames+1
             n_spaces = n_cells - len(chars)
             chars.extend([' ' for i in range(n_spaces)])
             return '|' + '|'.join(chars) + '|'
```

```
In [15]: test_format_throw_chars()

         ..........
```

Function `format_frame_scores()`

This function is in principle similar to the last one. However, it will take a list of at most 10 numbers as input and produces a string containing textual representation of 10 cells. Each cell will be 3 'spaces' wide, except the last cell which shall be 5 'spaces' wide. It would be also nice if the numbers were centered, if possible, in their cells. Again, the function will take the number of frames in a game as a parameter. As usual, let's start with some tests.

```
In [16]: def test_format_frame_scores():
             test_equal(format_frame_scores([], 1), '|     |')
             test_equal(format_frame_scores([], 2), '|   |     |')
             test_equal(format_frame_scores([], 3), '|   |   |     |')
             test_equal(format_frame_scores([5, 14, 29, 49, 60, 61, 77, 97, 117, 133]),
                         '| 5 | 14| 29| 49| 60| 61| 77| 97|117| 133 |')
```

And the implementation:

In [17]:
```
def format_frame_scores(scores, n_frames=10):
    """Return string representing the second line of the scorecard"""
    # Transform scores to strings
    strings = [str(n) for n in scores]
    # Fill in empty cells if needed
    n_spaces = n_frames - len(scores)
    strings.extend(['' for i in range(n_spaces)])
    # Center the numbers in the cells
    centered = [s.center(3) for s in strings[:-1]]
    centered.append(strings[-1].center(5))
    return '|' + '|'.join(centered) + '|'
```

In [18]:
```
test_format_frame_scores()
```

```
....
```

Intermediate check

We still miss 2 crucial parts of the whole project: functions which shall determine what to print in the first and second line of the scorecard. But if we had this information, we now know how to construct the scorecard of it. Let's try the initial example:

In [19]:
```
print(format_top_rule())
print(format_throw_chars(['1', '4', '4', '5', '6', '/', '5', '/', ' ', 'X',
                          '0', '1', '7', '/', '6', '/', ' ', 'X', '2', '/', '6']))
print(format_mid_rule())
print(format_frame_scores([5, 14, 29, 49, 60, 61, 77, 97, 117, 133]))
print(format_bot_rule())
```

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|1|4|4|5|6|/|5|/| |X|0|1|7|/|6|/| |X|2|/|6|
| +-| +-| +-| +-| +-| +-| +-| +-| +-| +-+-|
| 5 | 14| 29| 49| 60| 61| 77| 97|117| 133 |
+---+---+---+---+---+---+---+---+---+-----+
```

What shall be printed?

The two functions which determine what shall be printed on the first and on the second line are non-trivial. We will construct them in separate notebooks:

- `throws_to_chars()` (bowling_line_1.ipynb)
- `throws_to_scores()` (bowling_line_2.ipynb)

Now, when we know how these functions should be defined, we can import them here and use them.

In [20]:
```
from bowling_1 import throws_to_chars
from bowling_2 import throws_to_scores
```

In [21]:
```
throws = [1,4,4,5,6,4,5,5,10,0,1,7,3,6,4,10,2,8,6]
```

In [22]:
```
print(format_score_card(throws))
```

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|1|4|4|5|6|/|5|/| |X|0|1|7|/|6|/| |X|2|/|6|
| +-| +-| +-| +-| +-| +-| +-| +-| +-| +-+-|
| 5 | 14| 29| 49| 60| 61| 77| 97|117| 133 |
+---+---+---+---+---+---+---+---+---+-----+
In [23]: # Perfect game. [30, 60, 90, 120, 150, 180, 210, 240, 270, 300]
         print(format_score_card([10,10,10,10,10,10,10,10,10,10,10,10]))
```

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| |X| |X| |X| |X| |X| |X| |X| |X| |X|X|X|X|
| +-| +-| +-| +-| +-| +-| +-| +-| +-| +-+-|
| 30| 60| 90|120|150|180|210|240|270| 300 |
+---+---+---+---+---+---+---+---+---+-----+
```

```
In [24]: # Spare in each frame. [11,23,36,50,65,81,98,116,135,155]
         print(format_score_card([0,10,1,9,2,8,3,7,4,6,5,5,6,4,7,3,8,2,9,1,10]))
```

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0|/|1|/|2|/|3|/|4|/|5|/|6|/|7|/|8|/|9|/|X|
| +-| +-| +-| +-| +-| +-| +-| +-| +-| +-+-|
| 11| 23| 36| 50| 65| 81| 98|116|135| 155 |
+---+---+---+---+---+---+---+---+---+-----+
```

```
In [25]: # Expecting [5,14,29,49,60,61,77,97,117,133]
         print(format_score_card([1,4,4,5,6,4,5,5,10,0,1,7,3,6,4,10,2,8,6]))
```

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|1|4|4|5|6|/|5|/| |X|0|1|7|/|6|/| |X|2|/|6|
| +-| +-| +-| +-| +-| +-| +-| +-| +-| +-+-|
| 5 | 14| 29| 49| 60| 61| 77| 97|117| 133 |
+---+---+---+---+---+---+---+---+---+-----+
```

And this was the initial example:

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|1|4|4|5|6|/|5|/| |X|0|1|7|/|6|/| |X|2|/|6|
| +-| +-| +-| +-| +-| +-| +-| +-| +-| +-+-|
| 5 | 14| 29| 49| 60| 61| 77| 97|117| 133 |
+---+---+---+---+---+---+---+---+---+-----+
```

Summary

We demonstrated a lot here:

- We started small with a single failing test and gradually built the functionality, by generalizing the code we already had.
- The test suite helped us to catch errors that we introduced to the code during the development process.
- At several places, we stopped generalizing the function (fulfilling more tests) and did a **refactoring** to make the code more understandable, readable, and intention revealing.
- The code after refactoring has a very simple structure, althought the underlying details are not that simple.

Final remarks

It is worth noting that there are many possible ways how to implement the same functionality. Maybe, it would be profitable to first split the throws into individual frames and then start to process them one by one. Usually, the order in which you create the test cases will drive the development process and will - in an implicit way - determine the structure of the code.

Notebook config

Some setup follows. Ignore it.

```
In [26]: from notebook.services.config import ConfigManager
         cm = ConfigManager()
         cm.update('livereveal', {
                       'theme': 'Simple',
                       'transition': 'slide',
                       'start_slideshow_at': 'selected',
                       'width': 1268,
                       'height': 768,
                       'minScale': 1.0
         })

Out[26]: {'height': 768,
          'minScale': 1.0,
          'start_slideshow_at': 'selected',
          'theme': 'Simple',
          'transition': 'slide',
          'width': 1268}
```

```
In [27]: %%HTML
         <style>
         .reveal #notebook-container { width: 90% !important; }
         .CodeMirror { max-width: 100% !important; }
         pre, code, .CodeMirror-code, .reveal pre, .reveal code {
             font-family: "Consolas", "Source Code Pro", "Courier New", Courier, monospace;
         }
         pre, code, .CodeMirror-code {
             font-size: inherit !important;
         }
         .reveal .code_cell {
             font-size: 130% !important;
             line-height: 130% !important;
         }
         </style>
```