Function `throws_to_scores()`

This file is part of the bowling project (bowling.ipynb).

Petr Pošík

Department of Cybernetics, FEE CTU in Prague

EECS, BE5B33PRG: Programming Essentials, 2015

Specifications

Here we shall develop function `throws_to_scores()` with the following specs:

- Input: a legal sequence of throws, i.e. list of pins knocked down by a player in his/her individual throws.
- Output: list of at most 10 integer numbers representing the cumulative scores in individual frames. Remember, in case of strike, the score shall contain also the scores for 2 more throws, in case of spare the score shall contain also the score of one more throw.

We must be able to handle incomplete games, but we shall return only those scores whose value is final, i.e. will not change with subsequent throws.

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

Simple frames

Let's create a first test case:

In [2]:
```
def test_throws_to_scores():
    test_equal(throws_to_scores([]), [], "No scores in the beginning")
```

And let's create function `throws_to_scores` with a very simple body.

In [3]:
```
def throws_to_scores(throws):
    return []
test_throws_to_scores()
```

```
.
```

Simple frames

Let's add another test case, with a single throw.

In [4]:
```
def test_throws_to_scores():
    test_equal(throws_to_scores([]), [], "No scores in the beginning")
    test_equal(throws_to_scores([0]), [], "No scores after a single throw")
test_throws_to_scores()
```

```
..
```

Hmm, this works as well. Add 3 other test cases for a regular frame.

```
In [5]: def test_throws_to_scores():
            test_equal(throws_to_scores([]), [], "No scores in the beginning")
            test_equal(throws_to_scores([0]), [], "No scores after a single throw")
            test_equal(throws_to_scores([0,0]), [0], "Single simple frame 1")
            test_equal(throws_to_scores([0,9]), [9], "Single simple frame 2")
            test_equal(throws_to_scores([9,0]), [9], "Single simple frame 3")
        test_throws_to_scores()
```

```
..
Test 'Single simple frame 1' at line 4 FAILED.
  [0] expected, but got [].

Test 'Single simple frame 2' at line 5 FAILED.
  [9] expected, but got [].

Test 'Single simple frame 3' at line 6 FAILED.
  [9] expected, but got [].
```

OK, we will have to start doing something useful now.

```
In [6]: def throws_to_scores(throws):
            return [sum(throws)]
        test_throws_to_scores()
```

```
Test 'No scores in the beginning' at line 2 FAILED.
  [] expected, but got [0].

Test 'No scores after a single throw' at line 3 FAILED.
  [] expected, but got [0].
...
```

HA! This fulfilled the last 3 tests but broke the simple ones. We have to do something more elaborate to fulfill all the tests.

```
In [7]: def throws_to_scores(throws):
            frame_scores = []
            if len(throws) >= 2:
                frame_scores.append(sum(throws))
            return frame_scores
        test_throws_to_scores()
```

```
.....
```

Add a test case for part of second frame.

```
In [8]: def test_throws_to_scores():
            test_equal(throws_to_scores([]), [], "No scores in the beginning")
            test_equal(throws_to_scores([0]), [], "No scores after a single throw")
            test_equal(throws_to_scores([0,0]), [0], "Single simple frame 1")
            test_equal(throws_to_scores([0,9]), [9], "Single simple frame 2")
            test_equal(throws_to_scores([9,0]), [9], "Single simple frame 3")
            test_equal(throws_to_scores([9,0,1]), [9], "Half of the second simple frame")
        test_throws_to_scores()
```

```
.....
Test 'Half of the second simple frame' at line 7 FAILED.
  [9] expected, but got [10].
```

Ah, we sumed all the throws, but we need to sum up only the first two of them.

In [9]:
```
def throws_to_scores(throws):
    frame_scores = []
    if len(throws) >= 2:
        frame_scores.append(sum(throws[:2]))
    return frame_scores
test_throws_to_scores()
```

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

Add a test case for 2 simple frames.

In [10]:
```
def test_throws_to_scores():
    test_equal(throws_to_scores([]), [], "No scores in the beginning")
    test_equal(throws_to_scores([0]), [], "No scores after a single throw")
    test_equal(throws_to_scores([0,0]), [0], "Single simple frame 1")
    test_equal(throws_to_scores([0,9]), [9], "Single simple frame 2")
    test_equal(throws_to_scores([9,0]), [9], "Single simple frame 3")
    test_equal(throws_to_scores([9,0,1]), [9], "Half of the second simple frame")
    test_equal(throws_to_scores([9,0,1,8]), [9,18], "Two simple frames")
test_throws_to_scores()
```

```
......
Test 'Two simple frames' at line 8 FAILED.
  [9, 18] expected, but got [9].
```

We need to handle the second frame as well.

In [11]:
```
def throws_to_scores(throws):
    frame_scores = []
    if len(throws) >= 2:
        frame_scores.append(sum(throws[:2]))
    if len(throws) >= 4:
        frame_scores.append(sum(throws[2:4]))
    return frame_scores
test_throws_to_scores()
```

```
......
Test 'Two simple frames' at line 8 FAILED.
  [9, 18] expected, but got [9, 9].
```

Well, that did work only partially. The frame scores shall be **cumulative** scores. Let's introduce variable `score` which will hold the cumulative score.

In [12]:
```
def throws_to_scores(throws):
    frame_scores = []
    score = 0
    if len(throws) >= 2:
        score += sum(throws[:2])
        frame_scores.append(score)
    if len(throws) >= 4:
        score += sum(throws[2:4])
        frame_scores.append(score)
    return frame_scores
test_throws_to_scores()
```

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

OK, this works, but look at the code! Seems that we are repeating ourselves. Let's introduce a loop here.

```
In [13]: def throws_to_scores(throws, n_frames=10):
             frame_scores = []
             score = 0
             for frame in range(n_frames):
                 if len(throws) >= 2*frame+2:
                     score += sum(throws[2*frame : 2*frame+2])
                     frame_scores.append(score)
             return frame_scores
         test_throws_to_scores()
```

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

It still works, we did not break anything with the loop. And the code shall be able to handle full game containing only simple frames... Let's try it.

```
In [14]: def test_throws_to_scores():
             test_equal(throws_to_scores([]), [], "No scores in the beginning")
             test_equal(throws_to_scores([0]), [], "No scores after a single throw")
             test_equal(throws_to_scores([0,0]), [0], "Single simple frame 1")
             test_equal(throws_to_scores([0,9]), [9], "Single simple frame 2")
             test_equal(throws_to_scores([9,0]), [9], "Single simple frame 3")
             test_equal(throws_to_scores([9,0,1]), [9], "Half of the second simple frame")
             test_equal(throws_to_scores([9,0,1,8]), [9,18], "Two simple frames")
             test_equal(throws_to_scores([0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]),
                  [0,0,0,0,0,0,0,0,0,0], "Full game - no pins")
             test_equal(throws_to_scores([0,9,1,8,2,7,3,6,4,5,5,4,6,3,7,2,8,1,9,0]),
                  [9,18,27,36,45,54,63,72,81,90], "Full nontrivial game - no specials")
         test_throws_to_scores()
```

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

Strikes

Let's try some strikes.

```
In [15]: def test_throws_to_scores():
             test_equal(throws_to_scores([]), [], "No scores in the beginning")
             test_equal(throws_to_scores([0]), [], "No scores after a single throw")
             test_equal(throws_to_scores([0,0]), [0], "Single simple frame 1")
             test_equal(throws_to_scores([0,9]), [9], "Single simple frame 2")
             test_equal(throws_to_scores([9,0]), [9], "Single simple frame 3")
             test_equal(throws_to_scores([9,0,1]), [9], "Half of the second simple frame")
             test_equal(throws_to_scores([9,0,1,8]), [9,18], "Two simple frames")
             test_equal(throws_to_scores([0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]),
                  [0,0,0,0,0,0,0,0,0,0], "Full game - no pins")
             test_equal(throws_to_scores([0,9,1,8,2,7,3,6,4,5,5,4,6,3,7,2,8,1,9,0]),
                  [9,18,27,36,45,54,63,72,81,90], "Full nontrivial game - no specials")
             # Strikes
             test_equal(throws_to_scores([10]), [], "Strike, need 2 more balls")
         test_throws_to_scores()
```

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

Hmm, this passes. What about strike and another ball?

In [16]:
```python
def test_throws_to_scores():
    test_equal(throws_to_scores([]), [], "No scores in the beginning")
    test_equal(throws_to_scores([0]), [], "No scores after a single throw")
    test_equal(throws_to_scores([0,0]), [0], "Single simple frame 1")
    test_equal(throws_to_scores([0,9]), [9], "Single simple frame 2")
    test_equal(throws_to_scores([9,0]), [9], "Single simple frame 3")
    test_equal(throws_to_scores([9,0,1]), [9], "Half of the second simple frame")
    test_equal(throws_to_scores([9,0,1,8]), [9,18], "Two simple frames")
    test_equal(throws_to_scores([0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]),
         [0,0,0,0,0,0,0,0,0,0], "Full game - no pins")
    test_equal(throws_to_scores([0,9,1,8,2,7,3,6,4,5,5,4,6,3,7,2,8,1,9,0]),
         [9,18,27,36,45,54,63,72,81,90], "Full nontrivial game - no specials")
    # Strikes
    test_equal(throws_to_scores([10]), [], "Strike, need 2 more balls")
    test_equal(throws_to_scores([10,9]), [], "Strike, need 1 more ball")
test_throws_to_scores()
```

```
..........
Test 'Strike, need 1 more ball' at line 15 FAILED.
  [] expected, but got [19].
```

Well, this did not pass, as expected. We need to identify the strikes. Strike happens when the first ball in a frame is 10. The frame is then composed of a single throw, we thus cannot assume that it will start at the index `2*frame`. Let's create a variable called `ball` which will in each iteration (in each frame) hold the index of the first throw in a frame (instead of the hardwired `2*frame` which works only when all the frames have length 2). We also need to increment it by 1 after strike, and by 2 after simple frame.

In [17]:
```python
def throws_to_scores(throws, n_frames=10):
    frame_scores = []
    score = 0
    ball = 0
    for frame in range(n_frames):
        if throws[ball] == 10:
            score += 10 + throws[ball+1] + throws[ball+2]
            frame_scores.append(score)
            ball += 1
        if len(throws) >= ball+2:
            score += sum(throws[ball : ball+2])
            frame_scores.append(score)
            ball += 2
    return frame_scores
test_throws_to_scores()
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-17-8e99bb013d1e> in <module>()
     13             ball += 2
     14     return frame_scores
---> 15 test_throws_to_scores()

<ipython-input-16-fc3944f82195> in test_throws_to_scores()
      1 def test_throws_to_scores():
----> 2     test_equal(throws_to_scores([]), [], "No scores in the beginning")
      3     test_equal(throws_to_scores([0]), [], "No scores after a single throw")
      4     test_equal(throws_to_scores([0,0]), [0], "Single simple frame 1")
      5     test_equal(throws_to_scores([0,9]), [9], "Single simple frame 2")

<ipython-input-17-8e99bb013d1e> in throws_to_scores(throws, n_frames)
      4     ball = 0
      5     for frame in range(n_frames):
----> 6         if throws[ball] == 10:
      7             score += 10 + throws[ball+1] + throws[ball+2]
      8             frame_scores.append(score)

IndexError: list index out of range
```

Ouch! This ended up with a runtime error, even on the first test! Why!?!?

Well, because the first thing we do in the for loop is accessing the 0th item in the `throws` list, which may not be there. We need to check the length of the `throws` list before accessing the item. The same issue holds for the body of the first `if` in the loop: we access items `ball+1` and `ball+2` without knowing that they are there. So, let's account for that.

In [18]:
```
def throws_to_scores(throws, n_frames=10):
    frame_scores = []
    score = 0
    ball = 0
    for frame in range(n_frames):
        if len(throws) >= ball + 1 and throws[ball] == 10:
            if len(throws) >= ball + 3:
                score += 10 + throws[ball+1] + throws[ball+2]
                frame_scores.append(score)
                ball += 1
        if len(throws) >= ball+2:
            score += sum(throws[ball : ball+2])
            frame_scores.append(score)
            ball += 2
    return frame_scores
test_throws_to_scores()
```

```
..........
Test 'Strike, need 1 more ball' at line 15 FAILED.
  [] expected, but got [19].
```

OK, the first tests pass again, but the last one still fails. It is because if the first `if` which should handle strike in current frame finishes, the second `if` is still executed and computed the score although it shouldn't. The correction is easy: replace the second `if` with `elif`, such that only one part of the for-loop body is executed.

In [19]:
```
def throws_to_scores(throws, n_frames=10):
    frame_scores = []
    score = 0
    ball = 0
    for frame in range(n_frames):
        if len(throws) >= ball + 1 and throws[ball] == 10:
            if len(throws) >= ball + 3:
                score += 10 + throws[ball+1] + throws[ball+2]
                frame_scores.append(score)
                ball += 1
        elif len(throws) >= ball+2:
            score += sum(throws[ball : ball+2])
            frame_scores.append(score)
            ball += 2
    return frame_scores
test_throws_to_scores()
```

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

Well, that's better.

But look at the code! It is not very readable. What do all those `len(throws) >= ball + 1` conditions actually test? They test whether there is a sufficient number of available balls to complete the subsequent operations.

Let's introduce function `num_available_throws()` which will return the number of throws still waiting to be processed in the current and the following frames. Its body will be simple, just `return len(throws) - ball`. And we will also modify the conditions in the main function body to use the new function. We will implement it as a nested function with nonlocal variables.

In [20]:
```
def throws_to_scores(throws, n_frames=10):
    frame_scores = []
    score = 0
    ball = 0

    def num_available_throws():
        return len(throws) - ball

    for frame in range(n_frames):
        if num_available_throws() >= 1 and throws[ball] == 10:
            if num_available_throws() >= 3:
                score += 10 + throws[ball+1] + throws[ball+2]
                frame_scores.append(score)
                ball += 1
        elif num_available_throws() >= 2:
            score += sum(throws[ball : ball+2])
            frame_scores.append(score)
            ball += 2
    return frame_scores
test_throws_to_scores()
```

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

Much better. The code starts to be readable. But wait again! The code structure looks like it can be described by the following pseudocode:

```
if strike:
    update score after strike
elif simple frame completed:
    update score after simple frame
```

Let's modify the code this way. Let's first introduce functions `strike` and `simple_frame_completed` for the conditions after the `if` statements:

In [21]:
```
def throws_to_scores(throws, n_frames=10):
    frame_scores = []
    score = 0
    ball = 0

    def num_available_throws():
        return len(throws) - ball

    def strike():
        return num_available_throws() >= 1 and throws[ball] == 10

    def simple_frame_completed():
        return num_available_throws() >= 2

    for frame in range(n_frames):
        if strike():
            if num_available_throws() >= 3:
                score += 10 + throws[ball+1] + throws[ball+2]
                frame_scores.append(score)
                ball += 1
        elif simple_frame_completed():
            score += sum(throws[ball : ball+2])
            frame_scores.append(score)
            ball += 2
    return frame_scores
test_throws_to_scores()
```

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

Now, let's also introduce 2 functions that would contain the bodies of the `if` statements. Let's call them `adjust_scores_for_strike` and `adjust_scores_for_simple_frame`.

```
In [22]: def throws_to_scores(throws, n_frames=10):
             frame_scores = []
             score = 0
             ball = 0

             def num_available_throws():
                 return len(throws) - ball

             def strike():
                 return num_available_throws() >= 1 and throws[ball] == 10

             def simple_frame_completed():
                 return num_available_throws() >= 2

             def adjust_scores_for_strike():
                 nonlocal score, frame_scores, ball
                 if num_available_throws() >= 3:
                     score += 10 + throws[ball+1] + throws[ball+2]
                     frame_scores.append(score)
                     ball += 1

             def adjust_scores_for_simple_frame():
                 nonlocal score, frame_scores, ball
                 score += sum(throws[ball : ball+2])
                 frame_scores.append(score)
                 ball += 2


             for frame in range(n_frames):
                 if strike():
                     adjust_scores_for_strike()
                 elif simple_frame_completed():
                     adjust_scores_for_simple_frame()
             return frame_scores
         test_throws_to_scores()
```

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

The code looks great now! Let's try to add some more tests for strikes:

```
In [23]: def test_throws_to_scores():
             test_equal(throws_to_scores([]), [], "No scores in the beginning")
             test_equal(throws_to_scores([0]), [], "No scores after a single throw")
             test_equal(throws_to_scores([0,0]), [0], "Single simple frame 1")
             test_equal(throws_to_scores([0,9]), [9], "Single simple frame 2")
             test_equal(throws_to_scores([9,0]), [9], "Single simple frame 3")
             test_equal(throws_to_scores([9,0,1]), [9], "Half of the second simple frame")
             test_equal(throws_to_scores([9,0,1,8]), [9,18], "Two simple frames")
             test_equal(throws_to_scores([0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]),
                  [0,0,0,0,0,0,0,0,0,0], "Full game - no pins")
             test_equal(throws_to_scores([0,9,1,8,2,7,3,6,4,5,5,4,6,3,7,2,8,1,9,0]),
                  [9,18,27,36,45,54,63,72,81,90], "Full nontrivial game - no specials")
             # Strikes
             test_equal(throws_to_scores([10]), [], "Strike, need 2 more balls")
             test_equal(throws_to_scores([10,9]), [], "Strike, need 1 more ball")
             test_equal(throws_to_scores([10,9,0]), [19,28], "Strike and simple frame")
         test_throws_to_scores()
```

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

The last one passes as well. So let's try a perfect game!

In [24]:
```
def test_throws_to_scores():
    test_equal(throws_to_scores([]), [], "No scores in the beginning")
    test_equal(throws_to_scores([0]), [], "No scores after a single throw")
    test_equal(throws_to_scores([0,0]), [0], "Single simple frame 1")
    test_equal(throws_to_scores([0,9]), [9], "Single simple frame 2")
    test_equal(throws_to_scores([9,0]), [9], "Single simple frame 3")
    test_equal(throws_to_scores([9,0,1]), [9], "Half of the second simple frame")
    test_equal(throws_to_scores([9,0,1,8]), [9,18], "Two simple frames")
    test_equal(throws_to_scores([0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]),
         [0,0,0,0,0,0,0,0,0,0], "Full game - no pins")
    test_equal(throws_to_scores([0,9,1,8,2,7,3,6,4,5,5,4,6,3,7,2,8,1,9,0]),
         [9,18,27,36,45,54,63,72,81,90], "Full nontrivial game - no specials")
    # Strikes
    test_equal(throws_to_scores([10]), [], "Strike, need 2 more balls")
    test_equal(throws_to_scores([10,9]), [], "Strike, need 1 more ball")
    test_equal(throws_to_scores([10,9,0]), [19,28], "Strike and simple frame")
    test_equal(throws_to_scores([10,10,10,10,10,10,10,10,10,10,10,10]),
         [30, 60, 90, 120, 150, 180, 210, 240, 270, 300], "Perfect game")
test_throws_to_scores()
```

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

Spares

Let's move on to spares and try to introduce some in the throws.

In [25]:
```
def test_throws_to_scores():
    test_equal(throws_to_scores([]), [], "No scores in the beginning")
    test_equal(throws_to_scores([0]), [], "No scores after a single throw")
    test_equal(throws_to_scores([0,0]), [0], "Single simple frame 1")
    test_equal(throws_to_scores([0,9]), [9], "Single simple frame 2")
    test_equal(throws_to_scores([9,0]), [9], "Single simple frame 3")
    test_equal(throws_to_scores([9,0,1]), [9], "Half of the second simple frame")
    test_equal(throws_to_scores([9,0,1,8]), [9,18], "Two simple frames")
    test_equal(throws_to_scores([0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]),
         [0,0,0,0,0,0,0,0,0,0], "Full game - no pins")
    test_equal(throws_to_scores([0,9,1,8,2,7,3,6,4,5,5,4,6,3,7,2,8,1,9,0]),
         [9,18,27,36,45,54,63,72,81,90], "Full nontrivial game - no specials")
    # Strikes
    test_equal(throws_to_scores([10]), [], "Strike, need 2 more balls")
    test_equal(throws_to_scores([10,9]), [], "Strike, need 1 more ball")
    test_equal(throws_to_scores([10,9,0]), [19,28], "Strike and simple frame")
    test_equal(throws_to_scores([10,10,10,10,10,10,10,10,10,10,10,10]),
         [30, 60, 90, 120, 150, 180, 210, 240, 270, 300], "Perfect game")
    # Spares
    test_equal(throws_to_scores([9,1]), [], "Spare, need 1 more ball")
test_throws_to_scores()
```

```
.............
Test 'Spare, need 1 more ball' at line 20 FAILED.
  [] expected, but got [10].
```

Well, this was expected. The code does not handle spares at all. But thanks to the nice structure of the function, it should be easy to update the code. Similarly to the cases of strikes and simple frames, we will introduce function `spare` that will detect if the current frame is spare, and function `adjust_score_for_spare`.

```
In [26]: def throws_to_scores(throws, n_frames=10):
             frame_scores = []
             score = 0
             ball = 0

             def num_available_throws():
                 return len(throws) - ball

             def strike():
                 return num_available_throws() >= 1 and throws[ball] == 10

             def spare():
                 return num_available_throws() >= 2 and sum(throws[ball : ball+2]) == 10

             def simple_frame_completed():
                 return num_available_throws() >= 2

             def adjust_scores_for_strike():
                 nonlocal score, frame_scores, ball
                 if num_available_throws() >= 3:
                     score += 10 + throws[ball+1] + throws[ball+2]
                     frame_scores.append(score)
                     ball += 1

             def adjust_scores_for_spare():
                 nonlocal score, frame_scores, ball
                 if num_available_throws() >= 3:
                     score += 10 + throws[ball + 2]
                     frame_scores.append(score)
                     ball += 2

             def adjust_scores_for_simple_frame():
                 nonlocal score, frame_scores, ball
                 score += sum(throws[ball : ball+2])
                 frame_scores.append(score)
                 ball += 2


             for frame in range(n_frames):
                 if strike():
                     adjust_scores_for_strike()
                 elif spare():
                     adjust_scores_for_spare()
                 elif simple_frame_completed():
                     adjust_scores_for_simple_frame()
             return frame_scores
         test_throws_to_scores()
```

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

Easy, wasn't it! Let's throw in another bunch of tests!

In [27]:
```
def test_throws_to_scores():
    test_equal(throws_to_scores([]), [], "No scores in the beginning")
    test_equal(throws_to_scores([0]), [], "No scores after a single throw")
    test_equal(throws_to_scores([0,0]), [0], "Single simple frame 1")
    test_equal(throws_to_scores([0,9]), [9], "Single simple frame 2")
    test_equal(throws_to_scores([9,0]), [9], "Single simple frame 3")
    test_equal(throws_to_scores([9,0,1]), [9], "Half of the second simple frame")
    test_equal(throws_to_scores([9,0,1,8]), [9,18], "Two simple frames")
    test_equal(throws_to_scores([0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]),
         [0,0,0,0,0,0,0,0,0,0], "Full game - no pins")
    test_equal(throws_to_scores([0,9,1,8,2,7,3,6,4,5,5,4,6,3,7,2,8,1,9,0]),
         [9,18,27,36,45,54,63,72,81,90], "Full nontrivial game - no specials")
    # Strikes
    test_equal(throws_to_scores([10]), [], "Strike, need 2 more balls")
    test_equal(throws_to_scores([10,9]), [], "Strike, need 1 more ball")
    test_equal(throws_to_scores([10,9,0]), [19,28], "Strike and simple frame")
    test_equal(throws_to_scores([10,10,10,10,10,10,10,10,10,10,10,10]),
         [30, 60, 90, 120, 150, 180, 210, 240, 270, 300], "Perfect game")
    # Spares
    test_equal(throws_to_scores([9,1]), [], "Spare, need 1 more ball")
    test_equal(throws_to_scores([9,1,5,4]), [15, 24], "Spare and simple frame")
    test_equal(throws_to_scores([9,1,9,1]), [19], "Two spares, need 1 more ball")
    test_equal(throws_to_scores([9,1,8,2,7,3,6,4,5,5,4,6,3,7,2,8,1,9,1,9,9]),
         [18,35,51,66,80,93,105,116,127,146], "All spares")
    test_equal(throws_to_scores([5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5]),
         [15,30,45,60,75,90,105,120,135,150], "All spares, all 5")
    test_equal(throws_to_scores([1,4,4,5,6,4,5,5,10,0,1,7,3,6,4,10,2,8,6]),
         [5,14,29,49,60,61,77,97,117,133], "Full game" )

test_throws_to_scores()
```

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

All tests pass. Are you able to come up with a test case, that would fail?

Output module just for reuse

In [28]:
```
%%writefile bowling_2.py
def throws_to_scores(throws, n_frames=10):
    frame_scores = []
    score = 0
    ball = 0

    def num_available_throws():
        return len(throws) - ball

    def strike():
        return num_available_throws() >= 1 and throws[ball] == 10

    def spare():
        return num_available_throws() >= 2 and sum(throws[ball : ball+2]) == 10

    def simple_frame_completed():
        return num_available_throws() >= 2

    def adjust_scores_for_strike():
        nonlocal score, frame_scores, ball
        if num_available_throws() >= 3:
            score += 10 + throws[ball+1] + throws[ball+2]
            frame_scores.append(score)
            ball += 1

    def adjust_scores_for_spare():
        nonlocal score, frame_scores, ball
        if num_available_throws() >= 3:
            score += 10 + throws[ball + 2]
            frame_scores.append(score)
            ball += 2

    def adjust_scores_for_simple_frame():
        nonlocal score, frame_scores, ball
        score += sum(throws[ball : ball+2])
        frame_scores.append(score)
        ball += 2


    for frame in range(n_frames):
        if strike():
            adjust_scores_for_strike()
        elif spare():
            adjust_scores_for_spare()
        elif simple_frame_completed():
            adjust_scores_for_simple_frame()
    return frame_scores
```

```
Overwriting bowling_2.py
```

Notebook config

Some setup follows. Ignore it.

In [29]:
```
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[29]:
```
{'height': 768,
 'minScale': 1.0,
 'start_slideshow_at': 'selected',
 'theme': 'Simple',
 'transition': 'slide',
 'width': 1268}
```

In [30]:
```
%%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>
```