# Software Testing Lecture 2

Justin Pearson

September 25, 2014

# Test Driven Development

- Test driven development (TDD) is a way of programming where all your development is driven by tests.
- Write tests before you write code.
- Only write code when you fail a test.

# The TDD Mantra

- Red: Write a test that fails.
- Green: Write code that passes the test.
- Refactor: If possible refactor your code.

It seems a bit counter intuitive at first. I must think about the algorithm not the tests. Can I write complex algorithms this way?

# TDD

- The key is to start small. Simple tests.
- Grow your code slowly.
- Only make your code more complex if the tests require it.
- Think of a good sequence of tests.

# 2 Examples

- Converting numbers to English.
- A score board for the game of darts.

# Refactoring

- For us refactoring will be taking a piece of code and making it simpler to understand and smaller. Hopefully it makes it more efficient. Although we won't worry too much about efficiency. Clear readable code is out goal. Efficiency comes when it is needed.
- "We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil." Donald Knuth.
- When you start doing OO design, then refactoring also means tinkering with the object hierarchy.

# `toEnglish`

- The idea is to write a function that given a number between 0 and 999 inclusive returns a string spelling out the number in English. For example `toEnglish(43)` should produce the string `'forty three'`.
- We are going to develop the function in Python. If you are not familiar with Python, then pretend it is pseudo code.
- See the web page for accompanying pdf file that explains how to set up the tests.

# Red

First let us write a test that will fail.

```
assertEqual(toEnglish.toEnglish(0),'zero')
```

Apart from setting up the minimal amount of files to get the modules going we have no code. This test will fail.

# Green

- Do not think ahead. Write the minimal code that will pass the test.

```
def toEnglish(n):
  return('zero')
```

Stupid? Well it gets us going.

# Red

We need to find a test that fails.

```
assertEqual(toEnglish.toEnglish(1),'one')
```

This test of course fails.

```
self.assertEqual(toEnglish.toEnglish(1),'one')
AssertionError: 'zero' != 'one'
```

# Green

Write the minimum to pass the test.

```
def toEnglish(n):
    if    n==0:
        return('zero')
    elif n==1:
        return('one')
```

Now all the tests pass. `elif` is short for `else if`

# Red

You can see where this is going.

```
assertEqual(toEnglish.toEnglish(2),'two')
```

# Green

```
def toEnglish(n):
    """ Converts a number between 0 and 999 to Engl
    if   n==0:
        return('zero')
    elif n==1:
        return('one')
    elif n==2:
        return('two')
```

We are on a roll. We understand our tests and understand our code. This will not go on for ever, but it is a start.

```
assertEqual(toEnglish.toEnglish(3),'three')
assertEqual(toEnglish.toEnglish(4),'four')
assertEqual(toEnglish.toEnglish(5),'five')
assertEqual(toEnglish.toEnglish(6),'six')
assertEqual(toEnglish.toEnglish(7),'seven')
assertEqual(toEnglish.toEnglish(8),'eight')
assertEqual(toEnglish.toEnglish(9),'nine')
assertEqual(toEnglish.toEnglish(10),'ten')
assertEqual(toEnglish.toEnglish(11),'eleven')
```

You can guess what the code looks like.

```
def toEnglish(n):
if   n==0:
    return('zero')
elif n==1:
    return('one')
elif n==2:
    return('two')
elif n==3:
    return('three')
elif n==4:
    return('four')
elif n==5:
  .......
```

# Refactor

Now we are being too slow and stupid. Refactor. Use a list instead of a bunch of `if` statements.

```
def toEnglish(n):
    numbers = ['zero','one','two','three','four',
               'five','six','seven','eight',
               'nine','ten','eleven']
    return(numbers[n])
```

Rerun the tests to make sure that your refactoring does not mess anything up.

# Red

We need to find tests that fail. While we know what is going on we take bigger steps.

```
assertEqual(toEnglish.toEnglish(12),'twelve')
assertEqual(toEnglish.toEnglish(13),'thirteen')
assertEqual(toEnglish.toEnglish(14),'fourteen')
assertEqual(toEnglish.toEnglish(15),'fifteen')
assertEqual(toEnglish.toEnglish(16),'sixteen')
assertEqual(toEnglish.toEnglish(17),'seventeen')
assertEqual(toEnglish.toEnglish(18),'eighteen')
assertEqual(toEnglish.toEnglish(19),'nineteen')
assertEqual(toEnglish.toEnglish(20),'twenty')
assertEqual(toEnglish.toEnglish(21),'twenty␣one')
```

# Green

Well the code should be no surprise.

```
def toEnglish(n):
    numbers = ['zero','one','two','three','four',
               'five','six','seven','eight',
               'nine','ten','eleven','twelve',
               'thirteen','fourteen','fifteen',
               'sixteen','seventeen','eighteen',
               'nineteen','twenty','twenty␣one']
    return(numbers[n])
```

# Refactor

Time to refactor 'twenty one' is 'twenty' + ' ' + 'one'.
This gives us the following code:

```
def toEnglish(n):
    numbers = ['zero','one','two','three','four',
               'five','six','seven','eight',
               'nine','ten','eleven','twelve',
               'thirteen','fourteen','fifteen',
               'sixteen','seventeen','eighteen',
               'nineteen','twenty']
    if n in range(0,20):
        return(numbers[n])
    else:
        return('twenty' + '␣' + numbers[n-20])
```

# Refactor

Don't forget to rerun your tests.

```
  File "test_toEnglish.py", line 27, in test_simple
    self.assertEqual(toEnglish.toEnglish(20),'twenty')
AssertionError: 'twenty zero' != 'twenty'
```

Studying the code gives us an error. I had assumed that `range(0,20)` produced all the numbers up to and including twenty. It only goes up to 19. It is in the manual, but I found out by testing.

# Refactor — New version

```
def toEnglish(n):
    numbers = ['zero','one','two','three','four',
               'five','six','seven','eight',
               'nine','ten','eleven','twelve',
               'thirteen','fourteen','fifteen',
               'sixteen','seventeen','eighteen',
               'nineteen','twenty']
    if n in range(0,21):
        return(numbers[n])
    else:
        return('twenty' + ' ' + numbers[n-20])
```

Now all the tests work.

# Red?

Well now that we have a feel for our problem we can start generating the test cases in larger steps.

```
assertEqual(toEnglish.toEnglish(22),'twenty two')
assertEqual(toEnglish.toEnglish(23),'twenty three')
assertEqual(toEnglish.toEnglish(29),'twenty nine')
```

Well we still are not at our red state because all our tests are passed.

# Red

```
assertEqual(toEnglish.toEnglish(30),'thirty')
```

Now we fail.

```
  File "test_toEnglish.py", line 32, in test_simple
    self.assertEqual(toEnglish.toEnglish(30),'thirty')
AssertionError: 'twenty ten' != 'thirty'
```

# Green

So thirty should treated as a special case.

```
def toEnglish(n):
    numbers = [ .... ]
    if n in range(0,21):
        return(numbers[n])
    elif n in range(21,30):
        return('twenty' + ' ' + numbers[n-20])
    elif n == 30 :
        return('thirty')
    elif n in range(31,40):
        return('thirty' + ' ' + numbers[n-30])
```

# Red

All is still going well tests are passing.

```
assertEqual(toEnglish.toEnglish(32),'thirty two')
assertEqual(toEnglish.toEnglish(39),'thirty nine')
```

Our goal is to find tests that fail. Don't develop any code until you can find a test that fails.

```
assertEqual(toEnglish.toEnglish(40),'forty')
assertEqual(toEnglish.toEnglish(41),'forty one')
assertEqual(toEnglish.toEnglish(49),'forty nine')
```

# Green

```
def toEnglish(n):
    numbers = [ .... ]
    if n in range(0,21):
        return(numbers[n])
    elif n in range(21,30):
        return('twenty' + ' ' + numbers[n-20])
    elif n == 30 :
        return('thirty')
    elif n in range(31,40):
        return('thirty' + ' ' + numbers[n-30])
    elif n == 40:
        return('forty')
    elif n in range(41,50):
        return('forty' + ' ' + numbers[n-40])
```

# Refactor

Now time to refactor. Looking at the code we can use a similar trick with a list instead of all those `if` statements.

```
def toEnglish(n):
    numbers = [...]
    tens = ['twenty','thirty','forty','fifty',
            'sixty','seventy','eighty','ninety']
    numberOfTens  = n // 10
    numberOfUnits = n % 10
    if n in range(0,20):
        return(numbers[n])
    return_string = tens[numberOfTens]
    if numberOfUnits > 0 :
        return_string += ' ' \
                         + numbers[numberOfUnits]
    return(return_string)
```

# Refactor

Run the tests we have to see if we didn't mess anything up.

```
  File "test_toEnglish.py", line 27, in test_simple
    self.assertEqual(toEnglish.toEnglish(20),'twenty')
AssertionError: 'forty' != 'twenty'
```

Problem with the `tens` list. I got the indexing wrong. One way to fix it is to add two dummy values at the head of the list.

# Refactor

```
def toEnglish(n):
    numbers = [...]
    tens = ['','','twenty','thirty','forty','fifty'
            'sixty','seventy','eighty','ninety']
    numberOfTens  = n // 10
    numberOfUnits = n % 10
    if n in range(0,20):
        return(numbers[n])
    return_string = tens[numberOfTens]
    if numberOfUnits > 0 :
        return_string += ' ' \
                         + numbers[numberOfUnits]
    return(return_string)
```

Now all the tests pass.

# Red

When adding more tests we don't need to add one for every value between zero and ninety nine, but we need to check border cases. We are still looking for test that make our code fail so we can write some code.

```
assertEqual(toEnglish.toEnglish(100),'one␣hundred')
```

So we rewrite the code with the minimal effort to pass the test (it was late in the day).

```
def toEnglish(n):
    numbers = [...]
    tens = [..]
    numberOfTens  = n // 10
    numberOfUnits = n % 10
    if n in range(0,20):
        return(numbers[n])
    elif numberOfTens in range(1,10):
        return_string = tens[numberOfTens]
        if numberOfUnits > 0 :
            return_string +=
             ' '+numbers[numberOfUnits]
        return(return_string)
    return('one hundred')
```

# Red

If we add the test

```
assertEqual(toEnglish.toEnglish(200),'two hundred')
```

then the code will fail and we will have to rewrite something.

# Green

```
def toEnglish(n):
    numberOfHundreds = n // 100
    numberOfTens  = n // 10
    numberOfUnits = n % 10
    return_string = ''
    if n in range(0,20):
        return(numbers[n])
    elif numberOfHundreds in range(1,10):
        return_string = return_string +\
                        numbers[numberOfHundreds] +
    elif numberOfTens in range(1,10):
        return_string = return_string + tens[number
        if numberOfUnits > 0 :
            return_string +=
             ' ' + numbers[numberOfUnits]
    return(return_string)
```

# Green/Refactor

Part of the problem is our numberOfTens calculation assumes that n is two digits and we got our logic wrong with the elif stuff. But we can do some refactoring and use toEnglish ourselves using the fact that:

```
'k*100 + w' = toEnglish(k) +
    'hundred and ' + toEnglish(w)
```

We are cheating a bit by doing the refactoring in one step, but by doing all these tests we have a deeper understanding of the problem.

# Green/Refactor

```
elif numberOfHundreds in range(1,10):
    return_string +=
        numbers[numberOfHundreds] + ' hundred'
    n = n - numberOfHundreds*100
    return_string +=  ' and ' + toEnglish(n)
    return(return_string)
elif numberOfTens in range(1,10):
    return_string += tens[numberOfTens]
    if numberOfUnits > 0 :
        return_string +=
            ' ' + numbers[numberOfUnits]
        return(return_string)
```

# Green

Lets run the tests.

```
AssertionError: 'one hundred and zero' != 'one hundred'
AssertionError: None != 'twenty'
AssertionError: None != 'thirty'
```

We seem to have messed up something with the numbers less than one hundred, but lets just fix the problems one at a time. We'll fix the `'one hundred and zero'` problem.

# Green

```
    elif numberOfHundreds in range(1,10):
        return_string +=
            numbers[numberOfHundreds] + ' hundred'
        n = n - numberOfHundreds*100
        if n>0:
            return_string += ' and ' + toEnglish(n)
        return(return_string)
    elif numberOfTens in range(1,10):
        return_string += tens[numberOfTens]
        if numberOfUnits > 0 :
            return_string +=
              ' ' + numbers[numberOfUnits]
            return(return_string)
```

# Green

But we still get the problems with

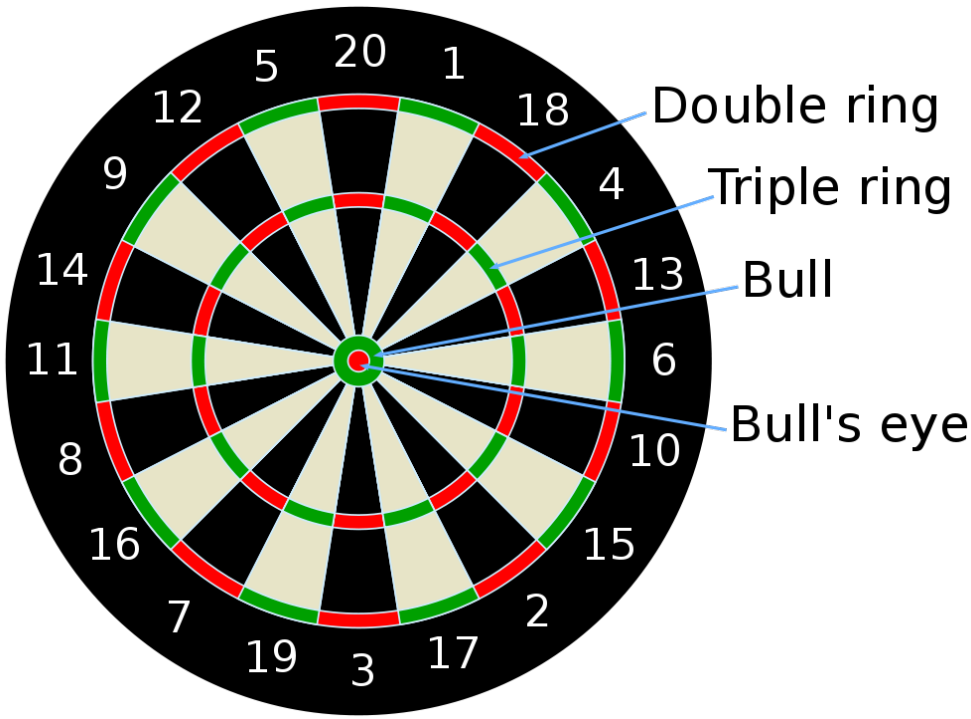
```
AssertionError: None != 'twenty'
```

```
AssertionError: None != 'thirty'
```

It means that I've got some of the if then else logic wrong; an easy problem in Python. Good job we have got tests to see if we have messed things up.
Look at the handout for the code. It does not really fit on a slide.

```
def toEnglish(n):
    numbers = [...  ]
    tens = [...]
    numberOfHundreds = n // 100
    numberOfTens  = n // 10
    numberOfUnits = n % 10
    return_string = ''
    if n in range(0,20):
        return(numbers[n])
    if numberOfHundreds in range(1,10):
        return_string = return_string + numbers[num
        n = n - numberOfHundreds*100
        if n>0:
            return_string = return_string + ' and '
        return(return_string)
    if numberOfTens in range(1,10):
        return_string = return_string + tens[number
        if numberOfUnits > 0 :
            return_string = return_string + ' ' + n
    return(return_string)
```

# Darts




Aim of the game to get your score to exactly 0 counting down from 301.

# Darts Scoreboard Class

We will implement a class that keeps track of the score, which player's turn it is and tells us if somebody has won. In a program this class would probably be interfaced into a GUI.

# Red

So the first test we do is to see if we can initialize an object of the class.

```
def test_init(self):
    game = darts.scoreboard()
```

This test fails. Since we haven't even written any object code yet.

# Green

```
class scoreboard:
    pass
```

This class does nothing.

# Red

What do we want to do with the class. It is a two player class. We want to know the score of player 1 and the score of player 2.
We assume that we are playing pub darts and start at 301. Games take to long otherwise and it takes time away from beer.

```
def test_score(self):
        game  = darts()
        self.assertEqual(game.playerscore(1),301)
        self.assertEqual(game.playerscore(2),301)
```

Notice that we are deciding the interfaces to the class by writing the tests.

# Green

```
class scoreboard:
    def __init__(self):
        self.playerscores = [301,301]
    def playerscore(self,player):
        return(self.playerscores[player-1])
```

Th `init` method is called when an object is called.

# Red

We only want two players. We want an exception thrown if we ask for the score of another player.

```
def test_exception(self):
        game = darts.scoreboard()
        self.assertRaises(NameError,
                                  game.playerscore,3)
```

# Green

```
def playerscore(self,player):
    if player == 1 or player == 2:
        return(self.playerscores[player-1])
    else:
        raise NameError('player out of range')
```

A darts board is divided into 20 regions. In each region you can score a single or double or triple times the score. So we want players to enter their score.

```
def test_scoring(self):
    game = darts.scoreboard()
    game.playerthrown(1,'single',15)
    self.assertEqual(game.playerscore(1),301-15)
    game.playerthrown(1,'double',20)
    self.assertEqual(game.playerscore(1),301-15-2*20)
    game.playerthrown(1,'triple',5)
    self.assertEqual(game.playerscore(1),
                     301-15-2*20-3*5)
```

The test will fail, we do not have a `playerthrown` method. So first attempt to correct the code.

```
def playerthrown(self,player,multiplier,number):
    if multiplier == 'double':
        number = number*2
    elif multiplier == 'triple':
        number = number*3
    self.playerscores[player-1] += number
```

At least it runs, but I still fail the test.

```
self.assertEqual(game.playerscore(1),301-15)
  AssertionError: 316 != 286
```

Did I get the code wrong or the tests wrong. The test looks fine, but I want to isolate things so I can pinpoint the problem.

```
    def test_scoring_single(self):
        game = darts.scoreboard()
        game.playerthrown(1,'single',15)
        self.assertEqual(game.playerscore(1),
                                 301-15)
    def test_scoring_double(self):
        game = darts.scoreboard()
        game.playerthrown(1,'double',20)
        self.assertEqual(game.playerscore(1),
                               301-(2*20))
    def test_scoring_triple(self):
        game = darts.scoreboard()
        game.playerthrown(1,'triple',5)
        self.assertEqual(game.playerscore(1),
                                 301-(3*5))
```

All three tests fail.

Looking at the code, I have made a stupid error.

```
self.playerscores[player-1] += number
```

should be

```
self.playerscores[player-1] -= number
```

Now finally all the tests pass.

# Red

Darts is a two player game. First player 1 plays 3 shots then player two plays three shots. We have to decide how the object behaves if players play out of turn. We'll throw exceptions. There are other ways of doing this, but *now* is the time to decide. Write the tests to capture the behaviour that you want. If you want to change the behaviour later you will have to rewrite the tests.

# Red

```
def test_player_1_plays_first(self):
        game = darts.scoreboard()
        #If a player 2 plays before player 1 then r
        self.assertRaises(NameError,
                game.playerthrown,2,'single',5)
```

Of course the test fails.

To solve this problem we need a variable that keeps track of the state of who's turn it is. So we modify the `init` method.

```
def __init__(self):
    self.playerscores = [301,301]
    self.turn = 1
```

And, modify the `playerthrown` method.

```
def playerthrown(self,player,multiplier,number):
    if player != turn:
        raise NameError('throw out of turn')
         .....
```

# Green

Tests still fail.

```
NameError: global name 'turn' is not defined
```

I forgot to use `self.turn` instead of `turn`. When it is fixed all the tests pass.

# Red

Now we have to think about how the game works. You make 3 throws and then it is the other players turn.
Since we are using exceptions we expect the following code to be exception free.

```
def test_three_throws(self):
        game = darts.scoreboard()
        game.playerthrown(1,'triple',5)
        game.playerthrown(1,'triple',5)
        game.playerthrown(1,'triple',5)
        game.playerthrown(2,'triple',20)
```

When the tests run we get an exception when player two tries to throw a dart.

So we need to keep track of how many throws have been. If we get to 3 then it is the other players turn.
First modify `init`

```
    def __init__(self):
            self.playerscores = [301,301]
            self.turn = 1
            self.throws = 0
```

And, then modify playerthrown to keep track of how many throws and flip the player when there have been 3 throws.

```
def playerthrown(self,player,multiplier,number):
    if player != self.turn:
            raise NameError('throw out of turn')
    self.throws = self.throws + 1
    if (self.throws == 3):
            self.turn = 1 - self.turn
```

But, the tests still fail.

```
======================================================================
ERROR: test_three_throws (__main__.TestDarts)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/var/folders/rE/rEvVYKXVEFWAJ9IyyNGDqk++42I/-Tmp-/
  File "darts.py", line 14, in playerthrown
    raise NameError('throw out of turn')
NameError: throw out of turn

----------------------------------------------------------------------
```

Ah, I'm stupid. I have a problem with indexing players both by 1 and by 0. This is going cause more errors in the future. From now on I will always index the players by 1. So I'll use the trick of having a dummy entry in the playerscores for the index 0 and rewrite the rest of the code accordingly.

```
def __init__(self):
    self.playerscores = [None,301,301]
    #  turn = 1 or 2 player's turn.
    self.turn = 1
    self.throws = 0
def playerscore(self,player):
    if player == 1 or player == 2:
        return(self.playerscores[player])
    else:
        raise NameError('player out of range')
```

```
def playerthrown(self,player,multiplier,number):
    if player != self.turn:
        raise NameError('throw out of turn')
    self.throws = self.throws + 1
    if (self.throws == 3):
        if (self.turn == 1):
            self.turn = 2
        else:
            self.turn = 1
    if multiplier == 'double':
        number = number*2
    elif multiplier == 'triple':
        number = number*3
    self.playerscores[player] -= number
```

# Red

Lets extend the previous test to make sure that we have got the logic of turns and throws correct.

```
def test_three_throws(self):
    game = darts.scoreboard()
    game.playerthrown(1,'triple',5)
    game.playerthrown(1,'triple',5)
    game.playerthrown(1,'triple',5)
    game.playerthrown(2,'triple',20)
    game.playerthrown(2,'triple',20)
    game.playerthrown(2,'triple',20)
    game.playerthrown(1,'triple',20)
    self.assertEqual(game.playerscore(1),301-3*(3*5)
    self.assertEqual(game.playerscore(2),301-3*20)
```

The tests still fail.

```
======================================================================
ERROR: test_three_throws (__main__.TestDarts)
----------------------------------------------------------------------
Traceback (most recent call last):

  File "darts.py", line 16, in playerthrown
    raise NameError('throw out of turn')
NameError: throw out of turn
```

Looking at the code, I've forgotten to reset the number of throws.

# Green

```
def playerthrown(self,player,multiplier,number):
    if player != self.turn:
        raise NameError('throw out of turn')
    self.throws = self.throws + 1
    if (self.throws == 3):
        self.throws = 0
        if (self.turn == 1):
            self.turn = 2
        else:
            self.turn = 1
    if multiplier == 'double':
        number = number*2
    elif multiplier == 'triple':
        number = number*3
    self.playerscores[player] -= number
```

Well the last test still fails.

```
  File "/var/folders/rE/rEvVYKXVEFWAJ9IyyNGDqk++42I/-Tmp-/
AssertionError: 121 != 241
```

Is it the code or the test? I've miscalculated the score. This test is getting a bit complicated. We've tested the scoring system for player 1. So lets divide the test up a bit.

```
def test_three_throws_3(self):
    game = darts.scoreboard()
    game.playerthrown(1,'double',5)
    game.playerthrown(1,'double',5)
    game.playerthrown(1,'double',5)
    game.playerthrown(2,'triple',20)
    game.playerthrown(2,'triple',20)
    game.playerthrown(2,'triple',20)
    self.assertEqual(game.playerscore(2),301-3*20)
```

The test still fails.

```
AssertionError: 121 != 241
```

Why do we get the answer 121? Because it is the right answer. I got the test wrong.

```
def test_three_throws_3(self):
    game = darts.scoreboard()
    game.playerthrown(1,'double',5)
    game.playerthrown(1,'double',5)
    game.playerthrown(1,'double',5)
    game.playerthrown(2,'triple',20)
    game.playerthrown(2,'triple',20)
    game.playerthrown(2,'triple',20)
    self.assertEqual(game.playerscore(2),301-3*3*20)
```

Now the tests pass.

# Hunting for Red

If you look in the handout there are more examples of tests of the scoring system.
Now I'm confident that the scoring code works and the players turn works. You have to stare at the code and think. TDD means you only have to think about little things at a time. Remember that passing a bunch of tests does not mean that your code is bug free, you have to think if you can move on.

# The end of a darts game

What happens when you finish a game of darts. The are variations in rule sets, but the version I played in the pub is that you must finish on exactly 0. Also you must score every throw. I'm not sure what the real rules are. Anyway this is a quick way for player 1 to get to 0.

```
def test_win(self):
    game = darts.scoreboard()
    game.playerthrown(1,'triple',20)
    game.playerthrown(1,'triple',20)
    game.playerthrown(1,'triple',20)
    game.playerthrown(2,'single',1)
    game.playerthrown(2,'single',1)
    game.playerthrown(2,'single',1)
    game.playerthrown(1,'double',19)
    game.playerthrown(1,'double',19)
    game.playerthrown(1,'double',19)
    game.playerthrown(2,'single',1)
    game.playerthrown(2,'single',1)
    game.playerthrown(2,'single',1)
    game.playerthrown(1,'single',1)
    game.playerthrown(1,'single',3)
    game.playerthrown(1,'single',3)
    self.assertEqual(game.playerscore(1),0)
```

# Red?

- Well we still have not reached condition red. This test passes. We have to decide what we want the class to do when a player wins. The alternatives are Exceptions, or Special value for the score?
- Personally I like using exceptions for non-error conditions, but some people think this is a bad idea. On some languages exceptions also have unnecessary extra overhead. So since this scoreboard will probably be interfaced to a GUI, we'll return a string 'WON' when the score goes to 0.

So lets modify the last line of the test.

```
self.assertEqual(game.playerscore(1),'WON')
```

The test fails.

# Green

So lets make the score function return 'WON' if the score is 0.

```
def playerscore(self,player):
    if player == 1 or player == 2:
        if (self.playerscores[player] != 0 ):
            return(self.playerscores[player])
        else:
            return('WON')
    else:
        raise NameError('player out of range')
```

All tests pass.

# Red

If you get a negative score during a throw then your score is reset back to what it was before the throw (in some variants you have to end on a double as well, we won't bother with that). So we can modify the previous test.
You can modify the test to make sure that your score is reset each round. See the handout
When you run the test you get:

```
AssertionError: -64 != 'WON'
```

# Green

So we'll need to keep track of the score in the current round. Modify `init`.

```
def __init__(self):
        self.playerscores = [None,301,301]
        self.turn = 1
        self.throws = 0
        self.current_round = 0
```

When we flip the current player we'll check if the score is -ve. If it is then we'll add the `current_round` score back to the score so the score should be reset to the score before the round.

# Green

```
def playerthrown(self,player,multiplier,number):
        if player != self.turn:
            raise NameError('throw out of turn')
        self.throws = self.throws + 1
        if (self.throws == 3):
            self.throws = 0
            if self.playerscores[player] < 0:
                self.playerscores[player] +=
                  self.current_round
            self.current_round = 0
            if (self.turn == 1):
                self.turn = 2
            else:
                self.turn = 1
      ......
        self.playerscores[player] -= number
        self.current_round += number
```

The test still fails.

```
FAIL: test_win_2 (__main__.TestDarts)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/var/folders/rE/rEvVYKXVEFWAJ9IyyNGDqk++42I/-Tmp-/
AssertionError: -59 != 'WON'

----------------------------------------------------------------------
```

I must of got the logic wrong for incrementing and decrementing the score. I got the current round logic wrong. Before when I flipped it didn't affect the rest of the code. Now I have to flip in the right place. I cheated I added print statements to see what was going wrong. Old habits die hard!

```
def playerthrown(self,player,multiplier,number):
  if player != self.turn:
      raise NameError('throw out of turn')
  self.throws = self.throws + 1
    ......  Numbers stuff
  self.playerscores[player] -= number
  self.current_round += number
  if (self.throws == 3):
      self.throws = 0
      if (self.turn == 1):
          self.turn = 2
      else:
          self.turn = 1
      if self.playerscores[player] < 0:
          self.playerscores[player] +=
                self.current_round
       self.current_round = 0
```

# Working Dart Class

- Now you have a working dart board class.
- You could extend it with more tests. If you hit the bull you get 50 and if you hit the outer bull you get 25.
- Handle non-scoring throws.
- Handle different variants of darts. This might mean that you start introducing sub-classing.

If you are designing a GUI it is good idea to separate logic from the GUI stuff. There are various design patterns out there MVC is one (Model, View, Controller).