1 Overview

In Project 5, we’re going to practice with objects some more. Along the way, we’ll also practice writing lots of `assert` statements.

For this project, you’ll be building a `Checkers` class. This class will represent the Checkers board at one point in time. As the game progresses, the program will create many of these objects - one for each different configuration of the board.

This class will have quite a few methods. Of course, it will have an `__init__()` method, to create a new board; this method will have the ability to create a brand-new game, or to load a game from a string. This class will have several methods to check the current state of the board (such as whether or not the game has ended); it will also have methods that allow us to create more objects - which represent moves that might happen. Finally, it has a “printable” output - which shows the state of the board in a human-readable way.

I am going to provide several example programs, which all use the same `Checkers` class. We will use these to test your code.

1.1 `assert` Statements

You will be making heavy use of `assert` statements in this project. You will validate the arguments passed to every method of your class; you will also provide a `sanity_check()` method, which my code can call, which checks everything about your internal variables.

You’ll find that my code will call `sanity_check()` very often (and I encourage you to call it from some of your methods, as well). Why do we do this so often? Why not call it once in a while, and then just assume that your code is working properly?

We call `sanity_check()` often in order to help you debug. If there’s a bug in your code somewhere, we want to find it as soon as possible, right after it first occurs - not long afterwards. Excessive checking is good debug.

Of course, in the Real World, programs don’t waste time with checking their data structures as often. (Performance is a big deal in real-world code.) However, we’re just learning about `assert` in this class - and so we’re going to be excessive.

In the future, when you have more experience, you’ll use a more balanced approach - you’ll use `assert` for key properties, instead of checking everything, every time.
1.2 Default Arguments
You will be required to use a default argument for \_init\_(). We’ll explain how they work below.

1.3 Don’t Print
In this project, I have provided three programs, which all use the Checkers class - so you should not have to write any code which prints anything. (You’re welcome to add some for debug purposes, but comment it out before you turn in your solution.)

2 Unit Testing
Since checkers.py is a library, it will not normally be run as a stand-alone program. However, as I mentioned in class, libraries often implement main() methods which perform “unit testing” on the library.

Your checkers.py must:

- Include a main() method, and code to run it:
  
  ```python
  if \_name\_ == \"\_main\_\":
     main()
  ```

- Include at least 30 different tests. Call various methods with certain parameters, and check that they return the correct values. Each method must be called at least once.

  (You will need to create at least two Checkers objects - to check the the two versions of \_init\_(). You probably will create more. But it’s OK to re-use the same object to check many different methods.)

- If a method ever returns the wrong value, print out an error message. Each message must be different (even if you only number them), so that you can know what test failed, when failures occur.

  Do not print out anything if a test passes.

- Count how many tests pass or fail. At the very end of the program, print out one of two messages, either:

  UNITTEST COMPLETE. NO FAILURES DETECTED.

  or

  Unittest complete. X out of Y tests passed.

  (Of course, X and Y are the number of tests that passed, and the total number of tests.)
2.1 Strategies for Unit Testing

So, what tests should you write for unit testing? To start with, do simple tests: create a default `Checkers` object, and then check to see that it has the right contents. Then perform a couple of moves and make sure that things work as expected. (Make sure that you try all of the methods in a few different scenarios.)

But here’s a hint: every time that you find a new bug in your code, write a unit test. That is, if you find a bug, then before you write the fix, write something in your unit test code which exposes the bug. Then, test the fix using your unit test, instead of the original scenario.

As you add more and more cases to your unit test, you’ll slowly build up a library of “things that must work” - these are things which will automatically get detected, if you accidentally introduce a bug later.

3 The Checkers Class

You will write a file named `checkers.py`, which contains the class `Checkers`. Each instance of this class represents the board at a certain state. This must include the complete state of the board (the position of all pieces, including accounting for Kings), as well as something to indicate which color moves next.

The `Checkers` class is immutable - that is, an object of this class, once created, cannot ever be modified. (In a perfect world, this means that you will do all of your setup in the `__init__()` function. However, when I wrote my solution, I found it a lot easier to duplicate an object, and then change a few things. I wrote a private helper method - the name started with a single underscore - to do the changes. You are allowed to do likewise, so long as the method is private.)

3.1 Implementation Hints

There are many ways that you might implement the board inside the class. The simplest (though not the most efficient) would be a 2D array\(^1\) with one element for each square on the board. Then define some way to encode all of the possible states:

- Empty space
- Red piece
- Red king
- Black piece
- Black king

\(1\) That is, a list of lists or tuple of tuples
You might encode them with characters, strings, integers, or any number of other options. In my solution, I used a tuple of strings - with one character for each square. But you’re welcome to use a different strategy if you’d like.

3.2 Required Methods

You must implement the following methods:

- \_\_init\_\_()
  As noted below, this must use a default argument. If called with no arguments, this will initialize the board to the standard start position. If called with an argument, that argument is a string which encodes the state of the board (see below).

  **assert Statements**
  This method must check that the input string (which encodes the board) is in the proper format. Check everything you can think of: is the string the right length, are the various characters reasonable, etc.

  After you have initialized the board and you are ready for \_\_init\_\_() to return, your last step must be to call the \_\_sanity\_check\_\_() method (see below).

- \_\_sanity\_check\_\_()
  This method must check everything you can think of about your object. Check to make sure that the board is the right size. 32 of the squares on the board must always be empty, no matter what; confirm that this is true. The other 32 might hold various pieces; check that each of those data fields has a reasonable value.

  This function should not perform a unit test - so don’t create any new objects, or run any methods. Instead, simply check to make sure that the current state of the data is reasonable.

  You must call this, as the last step in your \_\_init\_\_() method (for this project, at least). You are encouraged (but not required) to call it at the beginning or end of other methods in this class. My programs will call it on your objects quite often.

**Implementation Hint:**

Make use of the in operator for strings. For instance, the line

\[
\text{if var in "asd":}
\]

is equivalent to

\[
\text{if var == 'a' or var == 's' or var == 'd':}
\]
• **get_cur_player()**
  Returns either "red" or "black", indicating which player will play next.

• **get_square(x,y)**
  Returns the contents of one square on the board, encoded just like the input string. That is, an empty square is "."; a red piece is 'r', a red King is 'R', etc.
  The parameters are the x,y coordinates; each is in the range 0 to 7, inclusive.

• **assert Statements**
  Validate that the input values are valid. (It might be handy - though it’s not required - to also use the **sanity_check()** method, to make sure that you are returning reasonable values.)

• **get_printable_string()**
  Returns a string, which draws out the board. You may make this as simple or as fancy as you want - although it must include all of the current state of the board (except for the current player.)

**Examples:**
A simple picture might be to simply print out the rows:

```
 .b.b.b.b
 b.b.b.b.
 .b.b.b.b
 .........
 .........
 r.r.r.r.
 .r.r.r.r
 r.r.r.r.
```

A fancy version might have a lot more decoration:

```
 +---+---+---+---+---+---+---+---+
 8 | b | | b | | b | | b | |
 +---+---+---+---+---+---+---+---+
 7 b | | b | | b | | b | |
 +---+---+---+---+---+---+---+---+
 6 | | | | | | | |
 +---+---+---+---+---+---+---+---+
 5 | | | | | | | |
 +---+---+---+---+---+---+---+---+
 4 | | | | | | | |
 +---+---+---+---+---+---+---+---+
 3 r | | r | | r | | r | |
 +---+---+---+---+---+---+---+---+
```
encode()
Return the current state of the game, using the same code as the input to __init__().

assert Statements
This function must call sanity_check(), just in case.

get_piece_count(color)
Returns the number of pieces left for one of the players, as a 2-part tuple. The first part is the normal of normal pieces, and the second is the number of kings.
The parameter is either 'r' or 'b'.

assert Statements
Validate that the parameter is valid. You are encouraged (not required) to also use your sanity_check() method.

is_game_over()
Returns a boolean value, which is True if one player or the other has won. (This happens only when one player has no pieces left.2)

do_move(frm, to)
Performs a move, and returns a new Checkers object3 which represents the board after the move.
If the move is illegal, returns None.
The two arguments represent the “frm” and “to” positions on the board. They will be encoded as strings, using rank and file, like in chess:
http://www.dummies.com/games/chess/naming-ranks-and-files-in-chess/ for instance, the lower-left corner is "a1".
You must allow jumps, but multiple jumps are not supported by this program. Likewise, the “must jump if possible” rule is not enforced by this program.
If a piece reaches the last row (that is, if a red piece reaches the 8th rank, or a black piece reaches the 1st rank), then convert them to a King.

2I’ve been told that my definition is wrong here: that a player also loses if they have no valid moves left. That might be right, I’m not sure. But for this program, let’s define that as draw. Sorry if that’s not the official rules.
3Remember, Checkers objects are immutable, so you can’t change the current object’s state.
assert Statements
Use assert statements to confirm that the two arguments are both valid. Check as much as you can think of: for instance, check the length of both strings, and also that the characters have valid values.

Implementation Hints:
It will make calculations easier if you convert a rank-file string, like "b3", into (x,y) coordinates, like those used by get_square() - in this case, (1, 2).

How to convert letters into the appropriate integers? One option is a long, painful set of if/elif/else statements. Another way is a dictionary, that maps letters to numbers. But the best is the ord() function, which converts a letter to its ASCII encoding. What happens when you subtract 'c'-'a' (after converting both to ASCII)?

- calc_possible_moves()
Returns a list of tuples, which represent all legal moves from the current position. If the current player has no legal moves, then it returns an empty list. If the game is already over, it will fail an assertion. (But it doesn’t have to detect draws.)
The first field in each tuple is a new Checkers object, which shows what the board will be after the move. The second and third fields in each tuple are the from/to positions - exactly as encoded with do_move() - which were used to get to that position.

assert Statements
Assert that the game is not over. You are strongly encouraged (but not required) to also call sanity_check().

Implementation Hint
Write the do_move(frm, to) method first. Since that method returns None for invalid moves, one possible strategy for this function (slow, but it works) is to simply iterate over all possible moves and see which ones do_move(frm, to) will allow.

Since you’re iterating over pairs of positions - and each position is a coordinate pair - the easiest implementation will be 4 for loops, nested inside each other. Remember that you can do a for loop over a string - which will allow you to iterate through a set of characters.

3.3 Encoding a Board as a String
This project encodes boards as strings - not just as Checkers objects. This is used both as input to the _init_() method, and also as the output from the encode() method.

The encoding starts with a single character, either r or b, which indicates which player will move next. It then has a single space, followed by 8 words (also separated by spaces). Each word is exactly 8 characters long, and represents
the 8 squares in a single row. The first row is the row closest to the red player
(that is, where the red tokens start), and the last is the row closest to black
player (that is, where the red tokens are trying to get).

Use the following characters:

- . - empty space
- r - red token
- b - black token
- R - red king
- B - black king

For example, the starting position (black to move first, pieces start on the
bottom-left corner) would use the following encoding:

"b r.r.r.r .r.r.r.r r.r.r.r . . . . . . . . b.b.b.b b.b.b.b b.b.b.b b.b.b.b"

3.4 Default Argument for __init__()

The __init__() method for Checkers must take at least \[1\] one argument: a
string, which defines how the board is laid out. (See above. However, you must
also provide a default value for this argument, like this:

```python
def __init__(self, board_state=None):
    ... your code here ...
```

Now that you’ve provided a default value for the argument, Python will now
let you create a Checkers object in two ways:

```python
board1 = Checkers("....encoding here....")
board2 = Checkers()  # uses the default
```

Your program never knows the difference. It runs the same method in either
case - it’s just, in the second call, that Python provides the value automatically.

NOTE 1:
Default arguments work for all functions and methods, not just __init__().
And you can have many different parameters with default arguments in the
same function.

NOTE 2:
While it’s common to use None as the default value for an argument, it’s not
required. You can actually pass any value as the default. None is common
because it’s a good way to represent “nothing was given,” but other good options
might be:

\[1\] I don’t why you would want more than one argument. But now that you know how to
write default arguments, you can add as many as you want - if you have some reason for it!
• The empty string
• Some special string, such as ‘‘default’’
• The encoding of the starting board position
• Any (immutable) thing you can think of - so long as it’s clear what it means. (For subtle reasons, mutable objects are very bad choices for default arguments.)

3.5 **Hints for do_move()**

In my solution, do_move() was the longest method, by quite a bit. It was a little overwhelming at first, trying to figure out how to check all the things that I needed to check. So I thought I’d offer some hints and pointers. These are not necessarily in the order you need to do them - but they should give you a list of things to check.

Generally, my strategy was to look for errors - and immediately return None if I found one. So my code looked like this:

```
convert from/to inputs to handy integers

if something is bad:
    return None
if something else is bad:
    return None
if more checks:
    return None
if even more:
    return None

... build new object
update it to reflect the move
return the new object
```

Key things to check:
• Is the “from” square empty?
• Is the “to” square full?
• (After you’ve checked out if the move seems reasonable:) If this is a jump, then was the jumped-over square occupied by a piece of the other color?
• Was the move a 1-step diagonal move (a normal move), or a 2-step diagonal move? Or something else (invalid)?
• If the moving piece is not a King, was it moving in the correct direction?
• Did the piece reach the end of the board, and need to be made a King?
4 Testcases

In this project, we won’t have testcases in the sense we have had them before. Instead, I will be providing three demo programs that use your Checkers code:

- A program that simply prints out the board, given the encoding.
- A program that searches for the “best” next move from a position, using a very simple heuristic.
- A program that does some (not too smart) exploration of the next few moves (like you would do for an AI).

As before, I’ll provide demo versions of these online:
http://www.cs.arizona.edu/classes/cs120/summer17/projects/proj05/

(In addition, I have provided one interactive program - which I can’t run online - which allows you to play Checkers at the keyboard.)

Use these programs to test your code.

5 Turning in Your Solution

You must turn in your code using D2L, using the Assignment folder for this project.

Since this project only requires you to create one file - checkers.py - please turn in only this file to D2L. For simplicity, please DO NOT put it inside a .zip file.