Game: Othello

Rhodri James Revision 1.15, October 7, 2001

Credits

For the LaTeX source of this sheet, and for more information on LiveWires and on this course, see the LiveWires web site at http://www.livewires.org.uk/python/

Introduction

This is a Python games worksheet. When you've finished it, you'll have written a computerised version of the popular board game Othello.

What you need to know

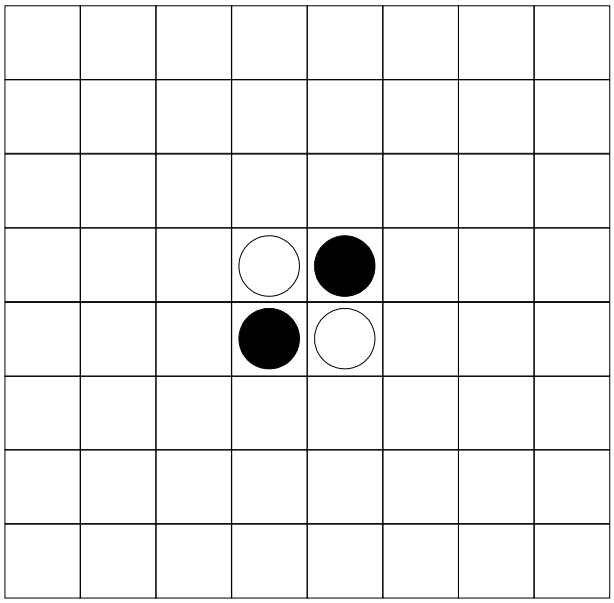
- The basics of Python (from Sheets 1 and 2)
- Function (from Sheet 3; you might want to look at Sheet F too)
- Classes and Objects (from Sheet O)
- Dictionaries (from Sheet D)

> *You should definitely look at Sheet O before starting this worksheet, even if you did the Beginners' course, as this sheet assumes you know about classes and objects, which aren't in that course.*

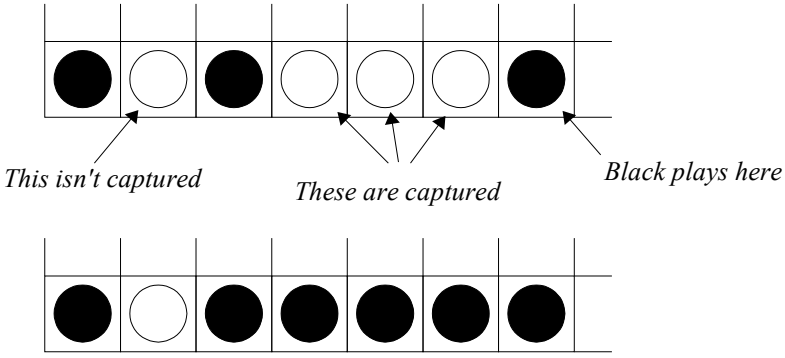
How to play Othello

Othello was originally invented in the 1880s in England, when it was called "Reversi." There is actually no difference between the two games, it's just that a canny games manufacturer repackaged the game in the 1970s and made a mint.

Othello is played on an 8×8 board. It starts with 2 white pieces and two black pieces occupying the central four squares as in the diagram below. Play then alternates between black and white; the player puts one of his pieces on the board so that he captures at least one of the other player's pieces.



Unlike chess or draughts, Othello uses a method of capture called *custodian capture*. One or more pieces are captured when they are sandwiched between two pieces of the opposite colour, as in the diagram. In Othello, one of the two pieces must be the one that is being played; the white piece on the left is safe.



As you see, captured pieces are not removed from the board, but instead change colour so that they belong to the capturer.

If on a player's go he cannot place a counter so as to capture some of his opponent's pieces, then he must miss his go.

The game ends when either the board is filled up, one player has no pieces of his colour left on the board, or neither player can make a legal move. The player with the most pieces on the board is the winner.

What do we need?

From the description, there's very little that we need to display on the screen for a game of Othello.

- A grid of squares
- Counters in two different colours (black and white)

That's it. The LiveWires library provides us with an easy way to make a grid, as well as shapes that we can use for counters. These come as classes in the library, as you might expect from Sheet O. We will need to make sub-classes from these classes to get them to do exactly what we want, however.

First things first

Before we get to the meat of our game, we need to tell Python to get the libraries and constants that we will want to use. It's best to put all these things in one place in the program so that you can find them easily later on, and the most obvious place to put them is right at the start.

Type the following into the editor window (the editor window is the one without the >>> characters before each line. In this window, Python doesn't run what you type as soon as you type it, but lets you type a lot of things to be run when you tell it).

```
from livewires import games
from livewires import boards
from livewires import colour
```

First, we tell Python that we want to access the things in the LiveWires `games` and `boards` modules, and also the colour names supplied by the `colour` module.

```
BOX_SIZE = 40
MARGIN = 60
COUNTER_SIZE = BOX_SIZE/2 - 3
TEXT_SIZE = 18

PLAYER1 = colour.white
PLAYER2 = colour.black
```

Here we set up values for quite a lot of things; the height and width of each box in our grid, the gap between the boxes and the edge of the window, and so on. We do this so that we can easily change the size of the boxes if we want to, just by changing the program at this point. If we decide that boxes of 40 units a side are too small, we can just change the program to read `BOX_SIZE = 60` (or whatever takes our fancy) and all the adjustments will be made for us. If we had just used the number 40 everywhere we were referring to the height or width of a box, we would have had to have tracked down every one of them and changed them individually. The chances of missing one are quite high, trust me!

Worse, there may be times when our box size is used to determine something else, and we are very likely to miss it. Take `COUNTER_SIZE` for example. We will use that constant as the radius of the circle that we will draw to represent a counter. To look neat on the board, this needs to be a bit less than half the width of one of our squares. If we were just typing numbers instead of using named constants, we would have typed in `18` (or whatever we thought looked best), and the odds are we wouldn't have thought twice about it when we changed the size of our boxes. This could look rather ridiculous, to put it mildly!

Notice that we're following a convention for naming our Python variables. All the variables we've made so far in this program are ones that we don't intended to change during the program. To be distinctive, we give them names that are ALL IN CAPITALS AND UNDERLINES. This gives us the hint that they are constants, and we shouldn't be changing them after setting them up for the first time. Python *doesn't* force us to do this, but it makes life a lot easier for someone else reading our program to see what is going on. It's worth doing; in six months' time, you won't remember what's what yourself!

```
NEXT_PLAYER = { PLAYER1 : PLAYER2,
                PLAYER2 : PLAYER1 }
```

This constant is a bit sneaky. We are making a Python dictionary called `NEXT_PLAYER`, which (as the name suggests) we will use to tell us who is going next. There are a lot of other ways we could have done this; in a more traditional programming language we would have had to give each player a number and performed a little calculation to work out who was next. Python however makes it easy for us just to identify each player by colour (something that we have to remember anyway), then look up the colour in `NEXT_PLAYER` to see who will be next. When we look up the first player,

`NEXT_PLAYER[PLAYER1]` tells us that it will be `PLAYER2` next, and vice versa.

Python's dictionaries (called *hash tables* in some other languages) are very handy for this sort of thing. We'll be using some more later on in the program to simplify our life.

```
COLOUR = { PLAYER1 : "White",
           PLAYER2 : "Black" }

ALL_DIRECTIONS = [ boards.LEFT, boards.UP_LEFT, boards.UP, boards.UP_RIGHT,
                   boards.RIGHT, boards.DOWN_RIGHT, boards.DOWN, boards.DOWN_LEFT ]
```

Finally, we define ourselves a list of all the `directions` that the LiveWires library gives us. This seems likely to be a useful thing to do; any time we put a piece down after all, we will have to check to see if it can make captures in any direction, so we will need to cycle through all the directions available.

On the starting grid

Now we get to putting our squares on the screen.

```
class Square(boards.Container, boards.GameCell):
  def __init__(self, board, i, j):
    self.init_container(['counter'])
    self.init_gamecell(board, i, j)
```

This says that our `Square` is a kind of `Container` and `GameCell`, both of which come from the `boards` module. A `Container` is a special class that the module provides which makes sure that everything that we put on top of the square, in this case a `counter`, stays visible on top of it. The `GameCell` class provides most of the bits and pieces that we need to put a box on the screen. Other than making sure that both of those subclasses are initialised, there's nothing more to do just yet.

Notice that we have another convention here. We've given out class a name that has a capital letter at the start. The names from the library go a little further since they contain several words; they have a capital letter at the start of each word and no underlines separating them. Once again, Python doesn't care what we call the class, it's just a trick to make it easier for us to notice what is going on.

> *Something happens in the call to* `init_container` *here that might cause confusion later. In order to prevent the* `Container` *class requiring you to do a lot of things in the right order, it secretly creates* `self.counter` *and sets it to* `None`*. This is almost always what we want it to do, but it might make you wonder why we can cheerfully test* `self.counter` *later on without ever having set it ourselves!*

But soft, what light through yonder window breaks?

We also need a class that represents the whole board, the collection of all the boxes. This will be a subclass of the `SingleBoard` class from the library, which as you might expect makes one board. It also handles creating and displaying the window that everything will be displayed in. Obviously we will still need to do a few extra things of our own, which means we need to make a subclass again. To start with, type in the following code:

```
class Othello(boards.SingleBoard):
  def __init__(self):
    self.init_singleboard((MARGIN, MARGIN), 8, 8, BOX_SIZE)
    self.create_directions()
    self.enable_cursor(3,3)

    self.current_player = PLAYER1
    self.game_over = 0
    self.num_counters = { PLAYER1 : 0,
                          PLAYER2 : 0 }
```

This tells Python that `Othello` is a kind of `SingleBoard` and records a few things that we will want to keep track of. In particular, we will need to know which player's turn it is at any given moment, whether or not we've finished the game, and how many counters of each colour are on the board at the moment. We need to know that last bit in order to work out if the game is over, and we could calculate it at the end of each turn, but it's a lot less work just to keep track as we go along.

We also tell Python to set up the `SingleBoard` parts of our class, which is mostly a matter of telling it how many rows and columns of boxes we need, how big they are and how much of a margin we want between the squares and the edge of the window. We also call `create_directions`, a function that the library provides that creates an ordered list called `direction` for each box. This list contains the squares, if any, which are next door to each square in any given direction. So for example, `box.direction[boards.LEFT]` tells us which square is immediately to the left of `box`. It contains `None` if there is no square in that direction.

Finally, the library can provide a special pointer for us that we call a "cursor". This puts a red outline around the square that we might want to drop a counter into, and moves it around for us according to the arrow keys on the keyboard. Since this makes life easier for us, we ask it to do that by calling `enable_cursor`.

We need one more thing in our `Othello` class before it will run. When we call `init_singleboard`, the class will go off and try to create boxes on the screen, as we said. To do this, it will call another method (*method* is just another name for a function that belongs to a class) called `new_gamecell`. The library does provide this function itself, but it only creates `GameCell` objects, not our `Square`s. We fix that by writing our own version inside the `Othello` class:

Line this up with the previous def

```
  def new_gamecell(self, i, j):
    return Square(self, i, j)
```

Not a very exciting function, but then glamour isn't for everyone.

First run

We only need to add two more lines to have a complete Python program, even if it's a Python program that doesn't do much. All we have to do is to tell Python to create an instance of the `Othello` class, and then to do all the work of running the screen for us.

```
othello = Othello()
othello.mainloop()
```

That's it. If you save your program and run it (from the command line — trying to run the program from Idle causes things to go somewhat astray!), you will see that it opens a new window on the screen containing an 8 by 8 grid of squares, one of which is outlined in red. You can move the red outline around the grid using the arrow keys; try it out now.

Counters

Now that we've put the scaffolding up, so to speak, we can start building the game proper. That means working out how to put counters down and capturing other counters, or at least working out how we are going to do that.

We haven't quite finished with all the setup yet. If you recall from the description, the middle four squares of the grid are supposed to have two black and two white counters in them. The next thing we need to do, then, is to work out how to display counters and just how to put them on the screen.

The obvious thing to represent a counter on the screen is a circle of the appropriate colour. Fortunately the library provides the `games.Circle` class for exactly this.

```
class Counter(games.Circle):
  def __init__(self, board, x, y, player):
    self.init_circle(board, x, y, COUNTER_SIZE, player, filled=1)
```

Easy. A little more complicated is the matter of putting the counter on the screen in the right place.

If you think about it, a counter lives in a square on the grid. In terms of the program, we have a `counter` in each `Square` which may or may not point to a real counter, depending on whether one has been put there or not. Logically, therefore, putting a counter in place is something that the square does, i.e. this function should be part of the `Square` class.

Add this to the `Square` *class*

```
def force_add_counter(self, player):
  self.counter = Counter(self.board, self.screen_x + BOX_SIZE/2,
                         self.screen_y + BOX_SIZE/2, player)
  self.board.num_counters[player] = self.board.num_counters[player] + 1
```

We call the function `force_add_counter` because it is the most basic "don't argue with me, just put the counter there" function that we're going to write. When we're really concerned with players putting counters down we are going to have to do all sorts of checks to ensure that it's a valid place to put a counter, and we don't need to be concerned about that just yet.

Notice that we do update the number of counters on the board. This helps us to determine when the game is over, saving a lot of work at the end of each turn. If we were being properly "object oriented" about our programming, we would have written a little function back in the `Othello` class that we would call to add one to the relevant total rather than doing it directly. Just this once, we'll take the short cut!

Finally, we need to put those initial counters on the board. This is just a few lines added to our `Othello.__init__` routine:

Add these lines to `Othello`*'s* `__init__` *function*

```
  self.grid[3][3].force_add_counter(PLAYER1)
  self.grid[4][4].force_add_counter(PLAYER1)
  self.grid[3][4].force_add_counter(PLAYER2)
  self.grid[4][3].force_add_counter(PLAYER2)
```

Run your program now, just to check the the counters do appear in the right place.

Placing counters

We need to decide how we want the players to tell us to put down a counter. The simplest thing from our point of view would be for the player to press some key when the cursor is over the box that he wants a counter in. Any key will do, so we will arbitrarily pick the `Return` key.

When the `boards` module sees a player press a key that it doesn't understand (i.e. that isn't one of the arrow keys), it calls the function `handle_keypress` to tell us. It includes a code in the call to say which key was pressed; in the case of the Return key, the code is that constant `games.K_RETURN` that the library provides for us.

This means that we need to add the following function to the `Othello` class:

Indent to match the rest

```
def handle_keypress(self, key):
  if self.game_over:
    return
  if key == games.K_RETURN:
    self.cursor.add_counter(self.current_player)
```

We haven't finished with this function yet, but we'll put it aside for a moment to concentrate on the rules for adding a counter.

Most obviously, a counter cannot go on a square that already has a counter in it. That's something that's easy to test for — is the `counter` bit of our `Square` the special Python object `None`?

The other rule of placement is that the counter that we're putting down must be able to capture other counters. In other words, we must be able to find some direction in which we have at least one counter of the opposite colour followed by a counter of our colour, with no empty spaces in between. We can do this by following along the `direction` line, taking care that we don't run off the board.

Building the test up bit by bit, we end up with a test for "can we capture?" that looks like this:

Put this in the `Square` *class*

```
def can_capture(self, player):
  other = NEXT_PLAYER[player]              We'll use this a lot
                                           For each direction...
  for d in ALL_DIRECTIONS:
                                           Check the square in that direction exists and has a counter
    if self.direction[d] <> None and self.direction[d].counter <> None:
                                           Check it's a counter of the opposite colour
      if self.direction[d].counter.is_player(other):
                                           Repeat this for as long as it's true
        s = self.direction[d].direction[d]
        while s <> None and s.counter <> None and s.counter.is_player(other):
          s = s.direction[d]
                                           We have something that isn't a counter of the opposite colour
                                           If it's a counter, it must be one of our colour, and we're done
        if s <> None and s.counter <> None:
          return 1                         Remember, non-zero means ''true''
                                           If we can't find anything, return false
  return 0
```

Phew! That's a good example of how to handle complicated issues, though. In any program, when you have something complicated, break it down into smaller and smaller pieces until it's small enough to understand. We've done that again here by fobbing off the question "Is this a counter belonging the `other` player?" onto our `Counter` class.

That turns out to be an easy question to answer, since we decided to identify our players by colour in the first place. We can find out what colour any object on the screen is by calling the library function `get_colour`, so the `is_player` function becomes just this:

Add to the `Counter` *class*

```
def is_player(self, player):
  return player == self.get_colour()
```

We've skipped a stage though. We haven't yet written our `add_counter` routine. Now that we have our `can_capture` test, it's easy enough to write:

Add to the `Square` *class again*

```
def add_counter(self, player):
  if self.counter == None and self.can_capture(player):
    self.force_add_counter(player)
    self.capture(player)
```

We've already written `force_add_counter`, so now we need to write `capture` as well. That's going to look very much like `can_capture`, except that instead of returning a true value when we work out that this is a good direction, we need to run through all the counters between `self` (the counter we're putting down) and `s` (the next one of our counters) changing the colour of each counter as we go. That in turn means that we need some help from the `Counter` class:

Add this to `Counter`

```
def flip(self):
  old = self.get_colour()
  new = NEXT_PLAYER[old]
  self.set_colour(new)
  self.screen.num_counters[old] = self.screen.num_counters[old] - 1
  self.screen.num_counters[new] = self.screen.num_counters[new] + 1
```

> As we said, `capture` looks very much like `can_capture`. Given what we've said, can you write the `capture` function yourself?

White and black

Thus far we can put white counters down on the board and take black pieces. This is all very well if we want to play white, but it's a bit boring for black.

Fortunately, that's very easy to fix. All we have to do is to set `self.current_player` to the next player (from `NEXT_PLAYER`) after each move, and that's easy to do.

> Go on, then. We're not going to give you every last line of code, you know; you are supposed to be doing some of the work yourself!

There's a snag, though, one that it won't take you too long to find out. If we swap players every time someone presses the `Return` key, then if one player tries to put a piece down in an illegal place he loses his turn. While you might chose to regard this as evolution in action, it's a bit harsh and not actually what the rules say. Therefore we need to figure out some way of not swapping players when they haven't really had a turn. That in turn means returning a yes or no from our `add_counter` routine.

> That's pretty simple. All we have to do is `return` a true value (1) if we actually drop a counter in `add_counter`, and a false value (0) if we don't. Then we just make `add_counter` the condition of an `if` statement in our `handle_keypress` routine, and only swap players if it returns true.

Messages

It would be useful, and would have made the last problem a whole lot less confusing, if the players had some way of telling whose turn it was. Let's add a useful message to our game, since the library makes this easy for us.

First we need to create the initial message. Since this is something that goes on the window as a whole rather than being associated with a particular square, we put it in the `Othello` class. Add these lines to the `__init__` function:

Don't forget to line it up!

```
x, y = self.cell_to_coords(4, -1)
self.status = games.Text(self, x, y, COLOUR[PLAYER1] + "'s turn",
                         TEXT_SIZE, colour.white)
```

This simply asks where the middle of the "-1"th row of squares is (as good a place to plonk our message as any), and makes a `Text` object centred there. `Text` is a class that the library supplies exactly for this purpose, putting words and numbers up in the window. Remember it, because we'll want it later.

> You might decide that you want to move the message slightly to get just the right effect. If you only want to move it a bit, remember what we said earlier about not just writing in numbers because you have to remember to change them all if you change one? That suggests that you ought to be nudging the `status` message by some fraction of `BOX_SIZE`, so that it'll end up proportionately in the same place if you ever change the box size! Try it and see.

We also need to change the message whenever we change whose go it is. The library gives us an easy way of doing this: the function `set_text` for a `Text` object does exactly what we want. Just add the line:

Correctly lined up, of course

```
self.status.set_text(COLOUR[self.current_player] + "'s turn")
```

after you change `self.current_player`.

Mouse movement

It's all very well racing around our window using the keyboard, but it would be nice if the players could just use the mouse to play Othello with. Fortunately, the `games` library makes this easy for us to do.

Just like with keypresses, when the library sees a mouse click that it doesn't understand (which is to say most of them), it calls a function that we can override. In this case we'll use the `mouse_up` function, which is called when the player releases a mouse button. The library gives us two parameters to this function; first the coordinates of where the mouse was on the screen when we let go, and second which button it was that was just released. We don't much care which button gets used, but other people might.

The function that we end up with looks a lot like `handle_keypress`: in fact it looks so much like it that we really should separate out the bits that are the same into a common function:

Remember, align the defs

```
def mouse_up(self, (x, y), button):
  if self.game_over:
    return
  i, j = self.coords_to_cell(x, y)
  if self.on_board(i, j):
    self.move_cursor(i, j)
    self.do_counter()

def handle_keypress(self, key):
  if self.game_over:
    return
  if key == games.K_RETURN:
    self.do_counter()
```

> It shouldn't take you long to work out what goes into `do_counter`. After all, you've already written it once!

Can you go?

It's possible for our game to come crashing to a premature halt at the moment. The rules say that if a player can't find anywhere legal to put a counter, he misses his go. We don't actually check for this yet, so the player would be left with nowhere to put a counter and no way of passing the move to the other player.

What we need is some way of checking if there is a legal move for a player. Our `can_capture` function will tell us if there is a legal move for a particular square, so what we need is to check all the squares to see if `can_capture` tells us there is a valid move.

The library provides an odd function that helps us with this. If we call `map_grid` and give it the name of a function, it will call that function on every square of the grid. The only problem is that the function is only allowed to take one parameter, and that has to be a `Square`. We need to communicate which player we're talking about somehow, and get back a yes-or-no response.

While there are several ways of doing this, we'll use the most straightforward method even if it is a bit inelegant. We'll put the player we want to check out into a `global` variable (remember those?), and expect to find the result in another global.

This all makes our `do_counter` routine messier:

```
                                                    You must align the defs correctly
  def do_counter(self):
    global capture_found, player_to_check           This is new
    if self.cursor.add_counter(self.current_player):
      capture_found = 0                             New
      player_to_check = NEXT_PLAYER[self.current_player] New
      self.map_grid(check_any_captures)             New
      if capture_found:                             New
        ...                                         Put what you had before here
      else:
        self.status.set_text("No move for " +
                             COLOUR[NEXT_PLAYER[self.current_player]] +
                             ": " + COLOUR[self.current_player] +
                             "'s turn")
```

Compared with that, the `check_any_captures` function is dead easy to write. We will take one opportunity to speed things up; if we've already found one capture, we don't need to check any more squares.

```
                                                    This is outside all the class definitions
                                                    i.e. no spaces to the left of this def!
def check_any_captures(square):
  global capture_found, player_to_check
  if not capture_found and square.counter == None:
    capture_found = square.can_capture(player_to_check)
```

Inelegant, but it works.

Winning and losing

After each move, we really ought to check to see if the game has ended before blithely letting the next player move. According to the rules, there are three conditions that end the game. We'll consider the first two for now: the game is over when one player has no counters on the board, or when all spaces on the board are filled.

This is the reason why we've been keeping the `self.num_counters` dictionaries up to date all the time. At the end of a player's turn, it may be that he has eliminated all of his opponent's pieces. He can't have eliminated his own, so we don't bother testing that. This makes for a very easy check, added again to `do_counter` before we do all the new checking for valid moves for the new player:

```
                                                    We'll leave you to find the right spot for this
      if self.num_counters[NEXT_PLAYER[self.current_player]] == 0:
         self.end_game()
         return
```

The second condition may sound like a lot of work; do we really have to check through every square in the grid, seeing if the all contain counters? No, actually, we don't. If you think about it, there can be only one counter in each space, so all that we need to check is that the sum of the counters for each player adds up to the number of spaces on the board. In other words:

```
                                                    Put this after the previous test!
      if self.num_counters[PLAYER1] + self.num_counters[PLAYER2] == 64:
        self.end_game()
        return
```

You can amalgamate the two tests into one `if` statement if you like.

The final condition is a little more complicated: the game is also over if neither player can move. We do currently check if one player can move, so now we need to check for the other one as well. This complicates our `do_counter` function again. You need to replace the existing `else` clause with the following:

```
                                          Remember, it must all line up
    else:                                 This you have already
      capture_found = 0                   Everything from here down is new
      player_to_check = self.current_player
      self.map_grid(check_any_captures)
      if capture_found:
        self.status.set_text("No move for"... As before
      else:
        self.end_game()
```

This just leaves one final problem, what to do when the game is over. The simplest solution is to display an appropriate message and prevent anyone from making any more moves. We might as well turn the cursor off while we're at it.

```
                                          This goes in the Othello class, of course
def end_game(self):
  self.game_over = 1
  self.disable_cursor()
  if self.num_counters[PLAYER1] > self.num_counters[PLAYER2]:
    self.status.set_text("Game over: " + COLOUR[PLAYER1] + " wins!")
  elif self.num_counters[PLAYER1] < self.num_counters[PLAYER2]:
    self.status.set_text("Game over: " + COLOUR[PLAYER2] + " wins!")
  else:
    self.status.set_text("Game over: Draw!")
```

And that's it. You now have a fully functioning game of Othello. All you need is another player!

Possible extras

There are still plenty of bells and whistles that you can add to your program if you want to make it more interesting. See if you can figure out how to do any of the following:

- Currently you can only play one game of Othello. When it's over, you have to start the program again to get another game. See if you can work out how to start a new game once the players press a given key after the end of the game. *Hint:* you will need to `remove` all the counters from the screen before you can start. You might find `map_grid` useful for this again.
- If you are doing that, you might want to keep a score of how many games each player has won, or how many counters they had at the end of each game, or some other score like that. That sounds like more `Text` objects to me!
- Since we keep track of the numbers of each colour of counter on the board anyway, you might also want to display these. Beware that the point you give to put a `Text` object at is the *centre* of the text string, which may look a little ugly.
- You could change the size of the board. Bear in mind that a smaller board makes for a quicker game, while a larger board could end up with a game lasting for ages! More interestingly, you could change the layout, introducing "holes" where pieces can't be put.
- If you're feeling very ambitious, you could turn this into an Othello *playing* program. You'll need to play a fair bit before you can work out a decent strategy, but you could start with something easy like picking a random square that has a valid move, or picking the square that offers the most captures (not always the best move by a long shot!). Take a look at the *Draughts* worksheet to see how to generate lists of valid places to move.
- Or failing that, you could make the capturing of pieces look snazzier. Flash the colours, or the backgrounds of the squares, or use an `Animation` of an explosion — whatever takes your fancy!
- If you have any other ideas for things to do, feel free!