Game: Draughts

Rhodri James

Revision 1.15, October 7, 2001

Credits

Introduction

This is a Python games worksheet. When you've finished it, you'll have written a computerised version of Draughts.

What you need to know

- The basics of Python (from Sheets 1 and 2)
- Functions (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 Draughts

Yes, yes, I know you know how to play draughts, but it's worth writing everything down in any case. It always helps to list what needs to be done. That way you know what you haven't done yet.

Draughts is played on an 8×8 chequerboard. It starts with 12 white pieces placed on the white squares on one side of the board, and 12 black pieces placed on the white squares on the other side of the board. Each piece can only move one space diagonally, and only towards the opposite side of the board. If a piece of the opposite colour is in the way but the space beyond it is empty, then a piece can jump over the opponent, removing it from the game.

If on a player's go a piece can be taken as above, then it must be. If more than one capture is possible, the player gets to choose which one to play. If after a capture the piece can capture again, then it must do so immediately without waiting for the other player to take a turn. This carries on until the piece cannot capture any more of the opponents' pieces.

Once a piece reaches the opposite side of the board, it is "crowned", allowing it to move diagonally backwards as well as diagonally forwards.

The game ends when one player has no pieces left on the board.

What do we need?

There's not actually a lot that we need to be able to put on the screen for this game. From the description, we need:

- a grid of squares (preferably of alternating colours),
- counters in two different colours (black and white), and
- some way of marking a "crowned" piece.

The LiveWires library provides us with an easy way to make a grid, as well as shapes that we can use for counters. We'll have to think a bit about how we mark crowns; on a real draughts board, the player would put a second piece on top of the first one to make it stand out. That won't look so good on a computer, so we'll put a red blob on the piece instead.

All of these things are provided by the library as classes, as you might expect from Sheet O. We will need to make sub-classes from them to get them to behave exactly as we want, however.

Initial bits and pieces

Before we get to the meat of our game, we need to tell Python to get the libraries and constants that we will want for us. 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 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 - 2
CROWN_SIZE = BOX_SIZE/10
TEXT_SIZE = 18
BOARD_SIZE = 8

CROWN_COLOUR = colour.red
WHITE = colour.white
BLACK = colour.black
BOARD_WHITE = colour.light_grey
BOARD_BLACK = colour.dark_grey
SELECTED = colour.red
```

This is setting 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 screen, 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!

It might seem a little odd to be creating something called `BOARD_WHITE` and then assigning it the colour `light_grey`, but think about it for a moment. We will be putting white pieces on white squares in order to play the game. If both the pieces and the squares really are white, then we won't see the pieces at all! This way we get to refer to both as "white" but still have the pieces visible, which is much less confusing.

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 = { WHITE : BLACK,
                BLACK : WHITE }
```

This last constant is rather 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 white, `NEXT_PLAYER[WHITE]` tells us that it will be `BLACK` 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.

Square dance

Now we get on to our board. We need to think about this a bit in order to get the colour of the square right. On a proper draughts board we alternate between white squares and black squares. If the square at (0, 0) is black, then (0, 1) is white, (0, 2) is black and so on. Moreover (1, 0) will be white too, and (2, 0) black, and so on. Oh, and while we're at it, (1, 1) will be black.

There is a pattern here, obviously enough. On column 0, the square on an even numbered row is black. On column 1, the black squares are on odd numbered rows instead. On column 2 we're back to the even numbers. And so on.

In other words, if you add together the row number and column number of the square, you have a black square if the result is even and a white square if it's odd.

The easiest way to work out if you have an even number or an odd number on a computer is to think of how that number looks in binary (base 2). An even number is a multiple of 2, so when you write it out in binary it will have a 0 in the last digit, just like all multiples of 10 have a 0 in the last place in base 10. For example, 6 is written as `110` in base 2; it's a multiple of two, and it has a zero in the last digit.

We can use this together with what we call *bitwise* operators to write our test in Python. The bitwise operators are like `and`, `or` and `not` which are mentioned in Sheet C, except that they work on each binary digit (or *bit* for short) separately. There's a long and tedious explanation, but for now let's just notice that the expression `n&1` is true (1) if `n` is an odd number, and false (0) if it's an even number.

```
class DraughtsBox(boards.Container, boards.GameCell):
  def __init__(self, board, i, j):
    if i+j & 1 == 0:
      box_colour = BOARD_WHITE
    else:
      box_colour = BOARD_BLACK
    self.init_container(['counter'])
    self.init_gamecell(board, i, j, fill_colour = box_colour)
```

So our DraughtsBox, each individual square on the board, is a kind of Container and GameCell, both of which are classes that come from the boards module. A Container is a special class that we provide to make sure that everything that we put on our 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. All we do is to pick the colour of the square and make sure that both classes are initialised.

Notice that we've got another naming convention here. We've given our class a name that has capital letters at the start of each word and no underlines separating them. Once again, Python doesn't care what we call the class, we're just doing this so that we can recognise what's what more easily.

Something happens in the call to init_container *here that might cause confusion later. In order to prevent the* Container *class requiring you do a lot of things in the right order, it secretly creates* self.counter *and sets it to* None. None *is a special Python object that is understood to mean "nothing here, guv." 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!*

All a board

As well as a class for our squares, we also need a class for the whole board. This will be a subclass of the SingleBoard class from the library, which as you might suspect from the name creates a single board. We will need to add a few features of our own in order to make the game work. To start with, type in the following:

```
class DraughtsBoard(boards.SingleBoard):
  def __init__(self, box_size, margin):
    self.init_singleboard((margin, margin), BOARD_SIZE, BOARD_SIZE,
                          box_size)
    self.create_directions()
    self.game_over = 0
    self.current_player = WHITE
    self.enable_cursor(0, 0)
```

This tells Python that DraughtsBoard is a kind of SingleBoard and records a few things that we will want to keep track of. In particular, we'll use a sneaky method of telling whose go it is — we'll keep a hold of the colour of counters that they use. This gives us a quick way of checking whether a player is allowed to move a given counter!

We also create_directions, a function in the library which puts together a list for each DraughtsBox called direction telling us which other square (if any) lies in any given direction. This will be important when we start trying to move pieces — it's always a good idea to know where you are expecting to move something.

Finally, the library can provide a special pointer for us on the screen that we call a "cursor". This puts a red outline around the square that we might want to move a counter from or to, and allows us to move this outline around by pressing the arrow keys. This will save us a lot of time and effort, so we ask the library to do this by calling enable_cursor.

Our DraughtsBoard class needs one more thing to make it the bare minimum that we need to start doing something. When we call init_singleboard, the class will go off and try to create the squares for the board. To do this it will call another method (*method* is just a fancy programmer's name for a function that is part of a class) called new_gamecell. The library does provide one, but we don't want to use it; it only creates a GameCell when we want a DraughtsBox. Instead, we have to write one of our own:

Line this up with the ''def __init__''

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

Not, perhaps, the most difficult of functions to write, but an important one none the less.

Test run

To make a complete Python program that does something, albeit not very much, we need to add two more lines to our file. We need to tell Python to create an instance of our `DraughtsBoard` class, and then do all the work of running the window for us.

```
draughts = DraughtsBoard(BOX_SIZE, MARGIN)
draughts.mainloop()
```

Short and sweet. If you now save your program and run it, you will see it open a new window on the screen containing an 8 by 8 grid of squares of alternating colours. One of the squares is outlined in red, and you can move the red outline around the grid by using the arrow keys.

Adding counters

Now that we've got the most basic bits and pieces sorted out, it's time to start adding our playing pieces to the board. That means working out what we want them to look like, and a few simple ideas of how they should behave.

For our first attempt, we will just draw a circle on the board of the appropriate colour. That's pretty easy — the library provides a class that draws circles already.

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

So far this is nothing that the `Circle` *class couldn't have done on its own. So why don't we use* `Circle` *directly instead of making our own class?*
The answer is that we're thinking ahead a bit, and seeing that our counters are going to get more complicated. We have yet to deal with crowning them, for instance!

Now we need to actually create some counters. We only need to do this once, when we make the board, so we could do all the work in our `DraughtsBoard`'s `__init__` routine. A line like

```
square.counter = Counter(...)
```

would be all that we need, but we're not going to do that!

It's a matter of good programming practice not to assume things that you don't need to assume, and not to let knowledge of the internals of one part of our program affect another. In this case, we would be allowing the `DraughtsBoard` object to know a little about the internals of a `DraughtsBox`, specifically that it has something called `counter` inside it and where it is on the screen. If we ever wanted to change the name of that variable, or change its behaviour, or even do some other admin while making a new counter like keeping track of how many there are, then we'd be in difficulties. It's a better idea to let `DraughtsBox` take care of its own internals.

This is all good advice for writing safe, clean programs. Like all advice, there are times when it's the wrong thing to do. If you are trying to squeeze more speed out of your system, for instance, then every function call slows you down a little bit, so you want to do things directly as much as possible. On the other hand, if speed is that much of an issue, you probably aren't writing your program in Python!

Since we're trying to teach you how to write safe, smart *object oriented* code, we'll do this the proper way. First, add this function to your `DraughtsBox` class:

Line this up with the other ‘‘def’’

```
    def new_counter(self, player):
      self.counter = Counter(self.board, self.screen_x + BOX_SIZE/2,
                             self.screen_y + BOX_SIZE/2, player)
```

This asks Python to create a new `Counter` object, telling it where to go on the screen (since the `screen_x` and `screen_y` bits of our `DraughtsBox` are automatically filled in for us to the corner of the box — that's why we have to add a bit in order to get the counter centred in the box). It also tells us what colour to make the counter — the colour of the `player` that it will belong to.

Then we can add the following lines to our `DraughtsBoard` `__init__` function:

Don't forget to line this up!

```
      self.grid[0][0].new_counter(WHITE)
      self.grid[2][0].new_counter(WHITE)
      self.grid[4][0].new_counter(WHITE)
      self.grid[6][0].new_counter(WHITE)
      self.grid[1][1].new_counter(WHITE)
      self.grid[3][1].new_counter(WHITE)
      self.grid[5][1].new_counter(WHITE)
      self.grid[7][1].new_counter(WHITE)
      self.grid[0][2].new_counter(WHITE)
      self.grid[2][2].new_counter(WHITE)
      self.grid[4][2].new_counter(WHITE)
      self.grid[6][2].new_counter(WHITE)
      self.grid[1][5].new_counter(BLACK)
      self.grid[3][5].new_counter(BLACK)
      self.grid[5][5].new_counter(BLACK)
      self.grid[7][5].new_counter(BLACK)
      self.grid[0][6].new_counter(BLACK)
      self.grid[2][6].new_counter(BLACK)
      self.grid[4][6].new_counter(BLACK)
      self.grid[6][6].new_counter(BLACK)
      self.grid[1][7].new_counter(BLACK)
      self.grid[3][7].new_counter(BLACK)
      self.grid[5][7].new_counter(BLACK)
      self.grid[7][7].new_counter(BLACK)
```

> That's rather a lot of typing. You can make this a lot shorter by using a `for` loop — see if you can figure out how.

Cursors!

Now that we have something on the board, we need to work out just how the player will tell us which counter should be moved where. The simplest thing from our point of view would be for the player to press some key when the cursor is over the counter that we want to move, and again when the it's over the place the player wants to move the counter to. We can use the same key for both purposes, both to select a piece and to actually move it, so we will be entirely arbitrary and pick the Space key.

We also need some way of telling which piece the player has selected, both on the screen and in our program. On the screen, the easiest way would be to change colour somewhere, say the colour of the square. In the program, we need another variable.

First, add the following line to our `DraughtsBoard` class `__init__` method:

Line it up correctly!

```
      self.selection = None
```

This creates the variable that we will want later. We will use it to point to the `DraughtsBox` that has been selected.

Now for the Space 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` in case we want to do anything. It includes a code in the call to say which key was pressed. In the case of the Space bar, the code is the constant `games.K_SPACE` that the library provides for us.

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

Indent to match __init__

```
def handle_keypress(self, key):
  if self.game_over:
    return

  if key == games.K_SPACE:
    if self.selection is None:
      self.selection = self.cursor.select(self.current_player)
```

`self.cursor` is a variable that the library handles for us. It gives us the `DraughtsBox` that the cursor is currently over, which is obviously the one that we want to select. We fob off the actual selection as such onto the `DraughtsBox`, since it will know better than `DraughtBoard` how to change its own colour and the like.

All that remains is to work out how to deal with selection. Notice that we've actually also fobbed off a bit of the decision making onto our `select` function as well — we don't really want to select a square that contains no counter, or that selects one belonging to the other player. This function, which goes in the `DraughtsBox` class, will do that for us:

Don't forget to align this

```
def select(self, player):
  if self.counter is None or not self.counter.is_player(player):
    return None
  self.set_colour(SELECTED)
  return self
```

Once again, in time-honoured fashion, we are fobbing off a bit of the work. This time we ask the counter who it belongs to rather than trying to work it out in `DraughtsBox`, which doesn't know directly. That means looking to see whether the colour we've been given (which is what `player` is) is the same as the colour of the counter.

> Try writing `is_player` yourself in the `Counter` class. All you need to know is that the function `self.get_colour()` will return the colour of any object.

Here Mousy, Mousy

So now we can race our cursor round the screen, selecting counters. Before we make life too complicated, we ought to consider how we can use the mouse instead, if that's what the players want to do. Fortunately, with the `games` library that's not hard to do.

Just like with key presses, when the library sees a mouse click that it doesn't understand (all of them!), it calls a function that we can rewrite to our heart's content. In this case we'll override the `mouse_up` function, which is called when the player lets go of a mouse button. The library gives us two parameters to this function, the position of the pointer on the screen and which button was released. For a game of Draughts we don't care which button was used, but in other games (Minesweeper for example) we might well want to do different actions for each button.

I hope you won't be surprised to find that our `mouse_up` function ends up looking a lot like `handle_keypress`. In fact they look so much alike that we really should separate out the bits that they have in common so that we don't have to write them twice. They are, after all, going to get more complicated!

Replace our previous `handle_keypress` *with this remembering to line it all up properly!*

```
def handle_keypress(self, key):
  if self.game_over:
    return
  if key == games.K_SPACE:
    self.action()

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.action()
```

> See if you can figure out for yourself what goes in the function `action`. It's not hard; you've already written it once!

More on selecting

So far we can select a counter, but not do anything else. Before we delve into the mystery of moving pieces, let's work on what to do if the player accidentally selects the wrong counter.

It would be nice if the player was allowed to change their mind before they committed to making a move. One simple way to indicate this would be to "select" the counter again, at which point the cursor can just remove the selection. To spot this in our program, we just need to check if the cursor is over our current selection. That only takes a couple more lines in the `action()` function:

Add this to the bottom, lining up with the ''if''

```
elif self.selection is self.cursor:
  self.cursor.deselect()
  self.selection = None
```

The we just need to write `deselect()` in the `DraughtsBox` class. All that has to do is to set the colour of the board back to white.

> No, really, it is that easy so we aren't going to write it out for you! Just remember that the board square colours are `BOARD_BLACK` and `BOARD_WHITE`, not `BLACK` and `WHITE`!

Making a Move

Working out where a counter can move to is a bit complicated. Ordinary white pieces can move diagonally downwards either to the left or right, ordinary black pieces can similarly move diagonally upwards either left or right, while crowned pieces of either colour can move in any diagonal direction. When our player presses the key or clicks on the mouse to make a move, we need to check that they have followed these rules.

For now, we are going to ignore crowned counters. That leaves us with a different pair of directions that a counter can move in depending on what colour it is. That suggests that we might use a dictionary like we do for `NEXT_PLAYER` to hold the list of directions for each player, and indeed we can do this to avoid typing lots of `if` statements!

Working from the top down, as usual, what we want to do is to try out the move from the `selection` square to the `cursor` square, and if it works we then do whatever we need to do at the end of each player's turn. In our program, that translates to a bit more at the end of the `action` function:

Don't forget to line it up!

```
    elif self.selection.try_move(self.cursor):
      self.end_turn()
```

This leaves us with two new functions to write. Let's do `end_turn` first, as it's simpler. All we have to do (for now) is to switch which player is the current one, as we discussed right at the start of the worksheet, and do a little tidying up. Add the following to the `DraughtsBoard` class:

Lined up properly!

```
  def end_turn(self):
    self.current_player = NEXT_PLAYER[self.current_player]
    self.selection = None
```

We'll come back to this later, but for now it's all we need.

Moving on to `try_move`, let's first break down what it has to do. First, it needs to see if the square that it's given is in one of the right directions from itself. If you recall, we mentioned earlier that the library provides a list called `direction` for each square that will help in this. If the `DraughtsBox` is in the right direction, it also needs to be an empty space (i.e. not to have a `Counter` in it). If either of these tests fail, we need to return a false value (`0`) so that `action` knows not to end the turn.

If all works properly, we need to tell the counter to shift itself on the screen, and then switch which `DraughtsBox` thinks it has the counter in it. Finally, we need to return a true value (`1` will do nicely) to tell `action` to give the other player a go.

All of this translates into the following function that you'll need to add to the `DraughtsBox` class:

Line up the defs

```
  def try_move(self, destination):
```

Do this first for convenience

```
    if destination.counter != None:
      return 0
```

Now look for our square

```
    dirs = self.counter.viable_directions()
    for d in dirs:
      if destination is self.direction[d]:
        self.counter.move_to(destination.screen_x + BOX_SIZE/2,
                             destination.screen_y + BOX_SIZE/2)
        self.counter.raise_object()
        destination.counter = self.counter
        self.counter = None
        self.deselect()
        return 1
    return 0
```

This looks a bit more complicated than it was described. You might hope that `viable_directions` *would return a list of the squares in the directions that our counter is allowed to move, but that's actually a bit more tedious to do that we really want.* `Counters` *don't really know about the innards of* `DraughtsBoxes` *after all, so rather than calling function after function to end up at the same place, we've simplified our life a bit and just dictated that* `viable_directions` *will return a list of the constant directions which we will then turn into squares ourselves.*

This leaves us with one last function to write in the `Counter` class:

Line up the defs!

```
  def viable_directions(self):
    return DIRECTIONS[self.get_colour()]
```

You might think that this looks a bit trivial to be bothering to write a whole separate function for. Normally you'd be right, but we secretly happen to know that things will get a bit more complicated when we add in crowned counters.

Finally you'll need to add the following to the constant at the top of the program:

```
DIRECTIONS = { BLACK : (boards.UP_LEFT, boards.UP_RIGHT),
               WHITE : (boards.DOWN_LEFT, boards.DOWN_RIGHT) }
```

Messages

It would be useful if the players had some way of telling who's 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 DraughtsBoard class. Add these lines to the __init__ function:

Don't forget to line up correctly!

```
    tx, ty = self.cell_to_coords(BOARD_SIZE/2, -1)
    self.status = games.Text(self, tx, ty, "White's turn",
                             TEXT_SIZE, colour.white)
```

This simply asks where the middle of the "-1"th row of squares is (as good a place as any to put our messages), 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.

> 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.

Once you've done that, we need to change the message whenever we change whose turn it is. The library gives us an easy way to do that: the function set_text changes the message of a Text object. Now all we have to do is work out what we want it to say, and for that we can use yet another one of those helpful dictionaries to turn the player's colour into his name.

First you need to add another new constant to the top of the program:

```
NAME = { WHITE : "White",
         BLACK : "Black" }
```

Then all you have to do is to concatenate the appropriate NAME with whatever message you want — check out Sheet S if you're not sure what that means. It all results in you adding the line:

Correctly lined up!

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

just after you change self.current_player in end_turn.

Taking Time

Now that we've worked out how to move pieces, it's time to work out how to take them. This is going to end up looking very much the same as moving, except that our destination should be two spaces diagonally away from where we start, and there should be a counter of the opposing colour in between.

The only thing that we need to be careful of is that the space in the middle really exists, rather than is somewhere off the board. We didn't have to worry about this in `try_move` because we knew that both the `selection` and the `destination` had to exist. While this is still true when we're taking a piece, we don't know which direction the square in the middle will be in, so we have to take care not to bother with squares that are actually off the board.

All this translates into a new method in our `DraughtsBox` class.

```
                                          Don't forget to line it up!
def try_take(self, destination, player):
                                          The first part looks just like try_move
  if destination.counter != None:
    return 0
  dirs = self.counter.viable_directions()
  for d in dirs:
                                          Look for destination two steps away
    middle = self.direction[d]
    if middle != None and middle.direction[d] is destination:
                                          Complain if there's no counter to take
                                          or if it's one of the player's own
      if (middle.counter is None or middle.counter.is_player(player):
        return 0
                                          The next bit should look familiar too
      self.counter.move_to(destination.screen_x + BOX_SIZE/2,
                           destination.screen_y + BOX_SIZE/2)
      self.counter.raise_object()
      destination.counter = self.counter
      self.counter = None
      self.deselect()
                                          Remove the counter that's been taken
      middle.counter.destroy()
      middle.counter = None
      return 1
                                          Line this up carefully!
  return 0
```

> You also need to add another `elif` clause to our `action` function. Since it looks almost exactly like the one we wrote earlier for `try_move`, we'll let you write it yourself. Just remember that `try_take` expects to be told what colour the player who's moving is, which we keep in `self.current_player` in the `DraughtsBoard` class.

> As we note above, the part of `try_take` involved in moving the counter is exactly the same as the part of `try_move` involved in moving the counter. You could extract that as a separate function if you want, making it easier if you ever want to alter exactly what happens or add some special effects.

Can Take, Must Take

As any draughts player will tell you, this isn't quite right. For example, when you take a piece, you are allowed to carry on jumping the same piece that you used in order to take lots of enemy pieces.

We will fix that, but first we're turning our attention to a different rule that we've ignored up until now. According to the rules, if a player is in a position where he can take a piece then he isn't allowed to make an ordinary (non-taking) move instead. If he can take several pieces, then it's his choice which one to take, but he has to take one and follow the chain along.

To deal with this, we need check at the start of each player's turn to see whether there are any valid captures for the player to make. Since we know that there can't be any captures on the very first turn of the game, we can equivalently check during `end_turn`, after switching who's turn it is.

What we hope to end up with is a list of squares which contain counters that can take one of the opposing counters, so we'd better set up such a thing in our `__init__` routine. This is something that belongs to the board itself, logically speaking, so add this line to the `DraughtsBoard` class `__init__` function:

Line up with everything else

```
    self.take_list = []
```

To do the actual work, we're going to use a special function that the library provides for us. `map_grid` is a function which takes another function as a parameter, then calls that function for each square on the board in turn. We're going to use this to check for each square whether or not we need to add it to the `take_list`, which means of course that we have to clear out anything previously in the list first.

All this comes to adding the following lines to our `end_turn` function:

Put at the end, lined up

```
    self.take_list = []
    self.map_grid(check_for_takes)
```

Then from our description, `check_for_takes` must be a separate function, not part of any class, and should look like this:

No spaces at the start of this line!

```
def check_for_takes(box):
  if box.can_take():
    box.add_to_take_list()
```

As usual, we've solved one problem by creating two more. Let's start with `can_take`. It has to do a lot of the same tests that `try_take` does, though with rather less assurance of having a valid destination place. When you break it down into steps, it's all quite straight forward. Just add this to your `DraughtsBox` class.

Line up the defs again

```
  def can_take(self):
    player = self.board.current_player
```

Does this square contain one of our counters?

```
    if self.counter == None or not self.counter.is_player(player):
      return 0
    dirs = self.counter.viable_directions()
    for d in dirs:
```

Does the next square in this direction exist...
...and does it have an enemy counter on it?

```
      middle = self.direction[d]
      if (middle != None and middle.counter != None and
          not middle.counter.is_player(player)):
```

Does the square after exist and have no counter?

```
        destination = middle.direction[d]
        if destination != None and destination.counter == None:
```

Yes, so we can perform a capture here

```
          return 1
    return 0
```

That just leaves us with `add_to_take_list`. If we were doing this properly, we would make our method in `DraughtsBox` call another method in `DraughtsBoard`, which is where the `take_list` lives. That would allow us to hide exactly how we record possible takes, in case we ever need to change them. In the interests of brevity, however, we'll do the stylistically wrong thing, and write this in our `DraughtsBox` class:

```
def add_to_take_list(self):
    self.board.take_list.append(self)
```

So far so good; we have a list of every possible capture that can happen on the board for the current player's turn. How do we use this information?

If you look above, we are told that if any takes are possible, we shouldn't allow ordinary moves. We can take that one stage further and be a bit more helpful to the player by not even allowing him to select counters that can't take something — after all we have a complete list of them!

All this means adding some checks of `take_list` into our `action` function. While we do the ones that we have to do, we can also add a couple more that we don't strictly speaking have to put in, but which will save a little time in the program. The end result should look something like this:

```
                                                        Don't forget to indent
def action(self):
  if self.selection is None:
                                                        We can try selecting this one if there are no
                                                        captures possible, or if this is one of them
    if self.take_list == [] or self.cursor in self.take_list:
      self.selection = self.cursor.select(self.current_player)
  elif self.selection is self.cursor:
    self.cursor.deselect()
    self.selection = None
                                                        Don't try ordinary moves if we must take something
  elif self.take_list == [] and self.selection.try_move(self.cursor):
    self.end_turn()
                                                        Don't try to capture if there aren't any
  elif (self.take_list != [] and
        self.selection.try_make(self.cursor, self.current_player)):
    self.end_turn()
```

Forced Captures

There is one last refinement we can add before we turn our attention back to multiple captures.

If our `take_list` consists of just one square, that's going to be the only square that we will allow our player to select. We can make life easier for the player by automatically selecting that square for him, and not letting him deselect it, forcing him to make the only move(s) available.

This is all pretty straightforward, really. First, let's make ourselves a flag so that we can quickly test whether or not we should be doing something. Add this line to the `DraughtsBoard` `__init__` function:

```
                                                        Indent it to match up
self.forcing = 0
```

We will set this to true if we must force the player into this move, and false otherwise. In other words, it will be true if there is only one square in `take_list`, and if it is true then we need to select the square automatically. This all sounds like more stuff for our `end_turn` function!

```
                                                        Add this after self.map_grid(check_for_takes)
self.forcing = (len(self.take_list) == 1)
if self.forcing:
  self.selection = self.take_list[0].select(self.current_player)
```

> We also said that if we are forcing the player's move, we won't allow him to deselect the square. Turning that round, we allow deselection only if we aren't `forcing`. Can you see how to change the `action` function to do this?

A Chain of Captures

The `forcing` mechanism that we've just written makes an ideal way of handling our earlier problem of handling multiple captures. All we have to do is to check to see if at the end of a capture, the piece we've just moved can do another capture (`can_take` will do this), and if so we force it's selection. That means adding this code to `try_take`:

Add this after `middle.counter = None`

```
        if destination.can_take():
          self.board.force(destination)
```

We also have to add the following function to the `DraughtsBoard` class, since we are doing this properly for once!

Align this with the other defs

```
  def force(self, box):
    self.forcing = 1
    self.selection = box.select(self.current_player)
    self.take_list = [ box ]
```

OK, that's not *quite* all we have to do. We also have to stop ourselves from switching players, which means changing `action` so that it doesn't always run `end_turn`. Fortunately, it's not too hard to test for this.

There are three cases to consider.

- At the end of a normal move or a take with no further options, `forcing` is false.
- At the end of a forced take with no further options, `forcing` is true, and `selection` will still be pointing to the original starting place of the counter.
- At the end of a forced or unforced take after which another capture can be made, we will have set `forcing` to true, but we will have changed the `selection` to be the same as our original destination, `cursor`.

It's only in the last case that we want to avoid ending the turn and switching players. That means changing `action` so that after trying out a take, we replace the call of `self.end_turn()` with this:

Align with where `end_turn` *was*

```
      if not self.forcing or self.selection is not self.cursor:
        self.end_turn()
```

And now you have all that you need to move and capture pieces!

Crown Him!

So far we've left quietly alone the question of what to do about crowning pieces, allowing them to move in all diagonal directions once they reach the row furthest away from where they started. It's time we rectified that.

We said earlier that we'd denote crowned pieces by putting a small red circle inside the black or white circle of the counter. That means that the big circle of the counter will *contain* the small circle of the crown, which is just what the library's `Container` class is designed for. You need to rewrite the class declaration and `__init__` so that they match what's printed below. Don't throw away any of the other methods in our original `Counter` class, they're fine!

```
class Counter(boards.Container, games.Circle):
  def __init__(self, board, x, y, player):
    self.init_container(['crown'])
    self.init_circle(board, x, y, COUNTER_SIZE, player)
```

All the other functions should be exactly the same

Once again, this makes an *element* of each counter called `crown` and sets it to `None`, which happens to be just what we want.

We then need to figure out how and when to crown a piece. From what we've said, a white piece gets to be crown when it reaches row 7, and a black piece when it reaches row 0. Adding the crown is just a matter of creating the circle in the same place on the screen as the counter is. All that translates to a new function for our new `Counter` class:

Align with the other defs

```
  def attempt_crowning(self, row):
    if self.crown != None:
      return
    if ((self.is_player(WHITE) and row == 7) or
        (self.is_player(BLACK) and row == 0)):
      self.crown = games.Circle(self.screen, self.xpos(), self.ypos(),
                                CROWN_SIZE, CROWN_COLOUR)
```

After that, all we need to do is to call our new function after every move or capture. In other words we add the line

```
destination.counter.attempt_crowning(destination.grid_y)
```

correctly indented, at a strategic point in each of `try_move` and `try_take`. You should be able to figure out exactly where without needing a leader to help you!

All that crowning really means is that the piece can now move in all diagonal directions. This is where our useless little function `viable_directions` suddenly becomes very useful indeed. We can use it to check if the piece has a crown, and if so we return a constant holding all the diagonal directions. In other words we add the following constant to the top of the program:

```
ALL_DIRECTIONS = [ boards.UP_LEFT, boards.UP_RIGHT,
                   boards.DOWN_LEFT, boards.DOWN_RIGHT ]
```

and rewrite `viable_directions` so that it looks like this:

Leave it lined up correctly!

```
  def viable_directions(self):
    if self.crown != None:
      return ALL_DIRECTIONS
    return DIRECTIONS[self.get_colour()]
```

That's all. This tells our move and take routines which directions to consider, and the `Container` class that we've built `Counter` from will keep the crown blob with the main circle whenever we move it.

Congratulations, you now have a fully working game of Draughts.

Ending the Game

The only thing that's currently missing from our game is for the computer to spot when the game is over. It would be nice if, when one or other player runs out of pieces, the program printed up a congratulatory message and stopped asking everyone to move!

We could count up the number of pieces of each colour at the end of each turn, but that's rather inefficient. We start off knowing the number of counters each player has — 12 — and we know each time one is taken off the board — whenever `try_take` returns a true value, so it's a lot less work for us to just keep count.

Let's start off by creating a couple of counters. We'll use a dictionary again because it makes it easy to look up either player's total, but this time we'll put the dictionary inside our `DraughtsBoard` since we aren't intending it to be a constant! Add this line to the end of our `__init__` (once again!):

Line up with everything else

```
    self.counters_left =  WHITE: 12, BLACK: 12
```

As we said, after a successful `try_take`, one of the opponent's pieces gets taken off the board. We should correspondingly take one away from the `counters_left` for the appropriate player — the one who isn't the `current_player`. That's easy, if a bit long to type:

```
next = NEXT_PLAYER[self.current_player]
self.counters_left[next] = self.counters_left[next] - 1
```

> Can you work out where in our `action` function to put these lines? Go on, it's not *that* hard!

Finally, at the end of the turn we need to check to see if the player whose turn it is about to become has any pieces left. The player whose turn it is must be OK, and we will always reach `end_turn` — we can't be `forcing` if there are no pieces left to take!

That translates to checking if `counters_left` for that player is greater than zero, and if not ending the game. That means we have to rearrange the start of `end_turn` to look like this:

Only change what you have to!

```
  def end_turn(self):
    next = NEXT_PLAYER[self.current_player]
    if self.counters_left[next] == 0:
      self.end_game()
      return
    self.current_player = next
```

...and the rest stays the same

To end the game, all we have to do is to set the `game_over` flag that we've been taking careful note of in our mouse and keyboard handler functions. We might as well also switch off the cursor, since the library lets us do that easily, and it's only polite to change the message to something appropriate. That's a very easy function to write:

Line up the defs

```
  def end_game(self):
    self.game_over = 1
    self.disable_cursor()
    self.status.set_text(NAME[self.current_player] + " wins the game!")
```

That's all you need for a proper, polite game of Draughts.

Bells and Whistles

If you've finished early, or you fancy a bit of a challenge, here are the few things that you could consider doing to make the game more interesting.

- We currently help the players by automatically selecting a piece if it's the only one that can move by capturing another.

We could be a bit more helpful and mark (say, by turning the squares light blue) where the pieces that can move are when the player has to capture something. This prevents players from sitting there moaning at your program because they want to move one piece, having not noticed the two other pieces that could make a capture!

- Possibly a bit more confusing for the players and therefore less useful for you to do, if there is only one possible move for a player then you could do it automatically, rather than just selecting the piece. Beware, though: all we check for currently is that only a single piece can take, not that it has only one possible piece to take!
- Currently, pieces move from one square to another all in one go. If you wanted a real challenge, you could make pieces move smoothly and slowly from one square to the next. You'll need to look up the `Mover` class in the library that some other games worksheets use!
- A very hard challenge indeed would be to turn this into a draughts *playing* program. You aren't going to be able to make it play well without a *lot* of work; it's probably best to start off having the computer select a random piece that can move and making a random move with it. You should be able to beat it easily, but at least the computer will be playing.
- You could introduce some "fairy draughts" pieces that move in special ways (for example, can only move by leaping over one of their own pieces). See how many variations you can think of, and which ones work.
- You could easily change the size of the board, or change the layout (introduce holes, for example, where pieces can't go). Think carefully about how many pieces you want on the board whenever you do something like that, though!
- If you want to be really silly, you could make pieces that are taken explode in a satisfyingly violent manner. Look up the bits of the library about `Animations`, or ask a leader for help.
- If there's anything else you can think of doing to your game, feel free!