Game: Chain Reaction

Rhodri James Revision 1.15, October 7, 2001

Credits

© Rhodri James. All rights reserved.

This document is part of the LiveWires Python Course. You may modify and/or distribute this document as long as you comply with the LiveWires Documentation Licence: you should have received a copy of the licence when you received this document.

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 version of a computer game called Chain Reaction.

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)
- Dictionaries (from Sheet D)
- Classes and Objects (from Sheet O)

> *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.*

What is Chain Reaction?

Chain Reaction is a two player game about exploding atoms. Each player takes turns putting counters on the board, trying to explode their atoms in order to capture their opponent's atoms.

When the game starts, the players see a blank grid. Each player then in turn places a piece on the grid, either on a blank space or on one already occupied by one of their pieces. When a space contains more pieces than it can hold, it explodes, scattering the pieces into neighbouring spaces. If this gives a space more counters than it can hold, then that space in turn explodes, and so on until no more spaces can explode. Any time that an explosion puts a piece into a square which contains the other player's pieces, it "captures" those pieces and makes them belong to the first player. Once all pieces of one colour are captured, the game is over.

Different squares can hold different numbers of counters. A corner space will explode when it gets two or more counters in it, while an edge can take three and others four before exploding.

What do we need?

So what things are there on the screen? From the description above, there doesn't seem to be a lot needed:

- a grid of squares,
- counters in two different colours for the two players,
- some way of telling how many counters are in a particular square,
- explosions, if we are feeling fancy.

The LiveWires games library provides us with the first of these, as well as shapes that we can put on the screen for the counters and numbers. These things are provided as classes, as you might expect from Sheet O. We will need to make sub-classes of these things in order to get them to do exactly what we want.

Boring bits to start with

Before we get to the interesting stuff, we need to tell Python to get all the bits and pieces together for us. In particular we need to tell it what library we want to use, but it's also a good idea to keep all the constants that you will want to refer to together at the beginning of the program. That way you can find them easily when you want to change them.

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
from random import randint
```

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. We will also want to use the function `randint` from the `random` module; this is in fact what we disguised as `random_between` in the beginners' library, so you can guess what we're going to do with that.

```
BOX_SIZE = 40
MARGIN = 60

COUNTER_SIZE = BOX_SIZE/2 - 3
NUMBER_SIZE = 24
TEXT_SIZE = 18
```

Then we set up `BOX_SIZE` to hold the height and width of each box on the screen, `MARGIN` to hold the gap between the boxes and the edge of the screen, and various other useful numbers. We did this so that we can easily change the number of boxes if we want to, just by changing the program at this point. If we decide that boxes of 40 units on each side are too small and that we want them twice as big, all we have to do is to change that line to read `BOX_SIZE = 80` and we can do everything automatically. If we had just used the number 40 everywhere we were referring to the height and width of our boxes, we would have to track down every time it was used and change them all individually, which would be a pain.

Worse, there may be times when our box size is used to determine another number, and we don't spot it. `COUNTER_SIZE` is a good example of that. We will use it as the radius of the circle that we will draw to represent a counter; clearly that needs to be a bit less than half the width of the square, so as to look neat. If we were just typing magic numbers all over the place, we would have replaced it with 17 (or whatever we thought looked neat), and wouldn't have thought twice about it when we changed the size of our boxes. This would look a bit silly. Since we're naming our constants, however, we can calculate the value we want from `BOX_SIZE` and not be caught out.

Another thing which we're doing here is following a convention for naming our variables. All of the variables that we have created here are ones that we don't intend to change during the program. We have given them names that are ALL IN

CAPITALS AND UNDERLINES to give us a hint that they are constants. Python *doesn't* force us to do this, but doing it makes it easier for someone reading our program to see what is going on. It also gives us a hint that we shouldn't be changing the variable after we have set it up the first time.

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

NEXT_PLAYER = { PLAYER1 : PLAYER2,
                PLAYER2 : PLAYER1 }

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

The last thing we set up is a bit sneaky. First we make a couple of constants that hold the colour we will use for each player. That means we don't have to remember that player 1 has the white counters and player 2 the black, the computer will do that for us. The sneaky bit is the Python dictionary `NEXT_PLAYER`. As the name suggests, we will use this to tell us which player's turn is next. There are many ways that we could have done this; in more traditional programming languages 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 to just 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 player 1's colour, `NEXT_PLAYER[PLAYER1]` gives us the answer `PLAYER2` and vice versa.

Python's dictionaries (called *hash tables* in some other languages) are very handy for this sort of thing. They make it easy to look up all sorts of things, and simplify all sorts of programming problems. Bear them in mind whenever you write new programs — you never know when they will save you some time and effort!

Making boxes

Now we get on to our boxes on the screen.

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

This says that our `Atom` 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 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've got another 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, 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 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!*

Lots and lots of boxes

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 make one board. Obviously we'll need a few extra things of our own. To start with, type in the following code:

```
class ChainReaction(boards.SingleBoard):
  def __init__(self, n_cols, n_rows):
    self.current_player = PLAYER1
    self.check = 0
    self.game_over = 0
    self.num_counters =  PLAYER1 : 0,
                         PLAYER2 : 0
    self.init_singleboard((MARGIN, MARGIN), n_cols, n_rows, BOX_SIZE)
    self.create_neighbours(orthogonal_only = 1)
    self.enable_cursor(0, 0)
```

This tells Python that `ChainReaction` 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 a sneaky little cheat that we'll need shortly — whether or not we should be checking to see if we've finished! Thinking ahead for a moment, our description said that the game ends when there's only one colour on the board. The obvious place to check for that is immediately after each player has his turn, which is when the exploding atoms might have eliminated the other player. If you think for a moment, you'll see that after the first player puts down his very first counter, he'll have one counter on the board while his opponent has none. If we check then and there, we'd have a very short game, so we need to hold off until both players have placed their first piece. We'll see exactly how to do that later, but for now we give our control flag `check` a false value (0, remember?).

Once that little bit of forward thinking is done with, we tell Python to set up the `SingleBoard` parts of our `ChainReaction`, 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 boxes and the edge of the window. We also call `create_neighbours`, a function that the library provides that creates a list called `neighbours` for each box listing which other boxes are next door to it. This will be important when we start exploding. Setting the oddly named parameter `orthogonal_only` tells the function that we only want the boxes that are left, right, up and down of each box, not the ones on the diagonals as well. That's what *orthogonal* means. All this computing and we give you free English lessons too!

Finally, the library can provide a special pointer for us that we call a "cursor". This puts a red outline around the box 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 `ChainReaction` class before it will run. When we call `init_singleboard`, the class will go off and try to create boxes, 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's boring and anyway creates a `GameCell`, not our `Atom`. We fix that by writing our own version inside the `ChainReaction` class:

Line this up with the previous def

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

OK, so that's not very exciting either, but it's *our* not very exciting function!

First run

We only need to add two more lines to have a complete Python program, albeit not a program that does much. All we need to do is to tell Python to create an instance of the `ChainReaction` class, and then to do all the work of running the screen for us.

```
chain_reaction = ChainReaction(8, 8)
chain_reaction.mainloop()
```

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, one of which is outlined in red. You can move the red outline around the grid using the arrow keys; try it out now.

Placing counters

Now that we've got the very basic bits and pieces sorted out, we can start to build the game proper. That means putting down counters and exploding them, or at least working out how to.

First, 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 the constant `games.K_RETURN` that the library provides for us.

This means that we need to add the following function to the `ChainReaction` 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 dealing with the counter.

Adding a counter to a box has two possibilities when you first look at it; either there's a counter already there, or there isn't. Let's look at the last case first, since when we start off there are no counters on the board at all.

It's usual when writing an object-oriented program like this to expect that our objects know best how to do things to themselves. In this case, we are going to write our program so that counters know how to display themselves. This makes adding a counter very easy for our Atoms; all we need to do is to tell it where on the screen we are (conveniently stashed away in the library as `self.screen_x` and `self.screen_y`) and what colour to make it — something we've carefully used already to determine which player we are!

Add this to the Atom class

```
  def add_counter(self, player):
    if self.counter == None:
      self.counter = Counter(self.board, self.screen_x + BOX_SIZE/2,
                             self.screen_y + BOX_SIZE/2, player)
```

The parameters may look a little odd, what with adding half the width and height of the box to the `x` and `y` respectively, but that's a case of thinking ahead and using one way of solving a problem to come. The `screen_x` and `screen_y` give the co-ordinates of the top left hand corner of the square (`y` counts the top of the screen as 0 and increases as you go downwards, just to be confusing), which is fine for most purposes. However, we will be using a circle to represent our counter, and for that we need to know where to put the centre of the circle!

Once again, we're nowhere near finished with this function, but first we need to define our `Counter` class a little more precisely.

A first attempt at Counters

As we mentioned above, for our first attempt at a counter we will just use a circle of the appropriate colour. That's nice and easy:

```
class Counter(games.Circle):
```

Remember, __init__ is the function that is called when you first make a Counter

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

That's it. Run your program now and see what it does.

We do *both* kinds of counter

Being able to put down white counters is all very well, but it would be nice if black had a chance too. It only takes us one line to swap between players in the `handle_keypress` function.

> Go on then! *Hint:* remember we were talking right at the beginning about how to work out which was the next player?

Unfortunately, there's a snag. If we swap players every time someone presses the Return key, then if one player hits the key while the cursor is over another piece, then they don't get their turn. Ignoring for the moment what's supposed to happen when you place a piece on top of one of your own, 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 nice and simple. All we have to do `return` a true value (1) if we actually drop a counter, and a false value (0) if we don't. All we then have to do is to make `add_counter` the condition of an `if` statement, and only swap players if it's true.

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 `ChainReaction` class. Add these lines to the `__init__` function:

Don't forget to line it up!

```
tx, ty = self.cell_to_coords(n_cols/2, -1)
self.status = games.Text(self, tx, ty, 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`.

Stacks of counters!

Back at the beginning, we said that we needed to stack counters on top of one another. Now it's time to start work on that.

The first thing that we need to think of is how this will all look to the players. There are lot of ways to do this, so we'll pick

a very simple one; we'll just put a number on top of the circle.

This is a bigger upheaval in our `Counter` class than we might have hoped; in the same way that our `Atom` class (the squares on the board) "contains" (i.e. has on top of it) a `Counter`, each `Counter` now has a number (a `Text` object) on top of it. That means we really need to make `Counter` a subclass of the `Container` class as well, otherwise the number will disappear unexpectedly some time when the library secretly moves things behind our backs.

So it's simplest to delete our tiny `Counter` class and replace it with this:

```
class Counter(boards.Container, games.Circle):
  def __init__(self, board, x, y, player):
    self.height = 1
    self.init_container(['number'])
    self.init_circle(board, x, y, COUNTER_SIZE, player, filled=1)
    self.number = games.Text(board, x, y, "1", NUMBER_SIZE, colour.yellow)
```

If you run this now, you will see the number "1" written on every counter that we place on the board. Again, you might want to nudge the position of the number a tiny bit if you want it in the exact centre of the counter, or you might choose a different colour to draw it in — just be sure that it shows up on all the counters!

By itself, that's not much of an improvement. Let's see about telling our counters to stack up a bit:

Don't forget to align this!

```
  def higher(self):
    self.height = self.height + 1
    self.number.set_text(str(self.height))
```

The `set_text` function in the library needs a string to set the text to, not a number. Fortunately, Python provides the `str` function to turn a number into a string, so all the work gets done for us.

Now all we have to do is to call `higher` at the right moment. If we go back to our `add_counter` routine, this means that we need an `else` to our `if` before we `return`. All we need to do in this `else` for the moment is to call `self.counter.higher`. It also means that we are doing something for the player on both sides of the `if`, so we should be returning `1` both times. This may seem a little pointless, but we will be making use of it again shortly, so keep it in.

> Go on then! Do remember that `return` ends the function there and then, so anything you add afterwards won't get executed. You might want to tidy things up by moving the `return 1` statements right to the end of the function; this will make things easier later.

If you run the program now, you'll notice a small problem. If the Black player drops a counter on top of a white one, it becomes a stack of two white counters. This is almost certainly not what Black had in mind at the time. We had better prevent each player from adding counters to the other's stacks, which means adding another `if` to the top of `add_counter`:

Remember, it's got to line up

```
    if self.counter <> None and self.counter.get_colour() <> player:
      return 0
```

Blowing things up

We've done almost everything else, now it's time to consider how our atoms explode.

If you re-read the description of the game, you might think that we need a big, complicated `if` statement to work out when to explode. Fortunately, if we notice one simple fact, we can make our life a lot easier.

According to the description, a corner square can take two counters before exploding, an edge square can take three, and any other square can take four. Notice, however, that a square in the corner of the board only has two neighbours if we

ignore the square on the diagonal. Similarly a square on the edge has three, and other squares have four. In other words, we explode when there are at least as many counters in the square as we have neighbours. The library helps us by providing the list `neighbours`, as we mentioned above, and we can use the function `len` to tell us how many there are, so that's easy.

Put this before the returns!

```
if self.counter.get_height() >= len(self.neighbours):
  self.explode()
```

> You should be able to write the `get_height` function for our `Counter` class on your own. All you have to do it to `return` how high the stack of counters is, and we're already keeping track of that.

For the explosion itself, we need to take off one counter for each neighbour we have, and put it on the neighbouring atom instead, something that we already know how to do. If this means that we have no counters left (often the case), then we have to take our counter off the screen entirely, for which we need the library again. The function `destroy` will get rid of our visible `Counter`, including the number as well as the circle.

But hang on a sec! Some of the neighbouring squares might contain counters of the other colour, and a little while ago we went to some trouble to stop that happening. How do we get around this now?

What we will do is to split our `add_counter` into two functions: one which doesn't allow us to drop a counter onto the opposition, and one which does. Edit your program so that all of this is now in our `Atom` class:

Don't forget to align the defs!

```
def add_counter(self, player):
  if self.counter <> None and self.counter.get_colour() <> player:
    return 0
  self.force_add_counter(player)
  return 1

def force_add_counter(self, player):
  if self.counter <> None:
    self.counter = Counter(self.board, self.screen_x + BOX_SIZE/2,
                           self.screen_y + BOX_SIZE/2, player)
  else:
    self.counter.higher()
  if self.counter.get_height() >= len(self.neighbours):
    self.explode()

def explode(self):
  player = self.counter.get_colour()
  if self.counter.get_height() == len(self.neighbours):
    self.counter.destroy()
    self.counter = None
  else:
    self.counter.lower(len(self.neighbours))
  for n in self.neighbours:
    n.force_add_counter(player)
```

That leaves us with one more function to write for our `Counter` class: `lower`. That's an easy one; it works just like `higher`, except that you are given a number to subtract from the height instead of adding one.

Those are my counters now

If you try out your program now, you'll find that there's still one small problem; when we explode onto a stack of the other player's colour, it stays belonging to the other player. According to the description, those counters should belong to the player whose counter exploded onto it.

Getting round this is quite simple. All we need to do is to tell our `higher` function which player we're increasing the stack for, i.e. add another parameter to it. We will then define it as

```
def higher(self, player):
```

and call it as

```
self.counter.higher(player)
```

(properly aligned, of course).

> You need to add one more line to `higher` to change the colour of the counter. If we tell you that the library method `set_colour` changes the colour of a `Circle`, can you write the line yourself?

One last thing

There is one rare problem that we don't yet cope with. When we explode a square, calling `force_add_counter` checks to see if a square exploded onto needs to explode itself, possibly calling `force_add_counter` again and again on different squares. However it is just possible, in a complicated and lengthy game, for a square to end up with two or more times the number of counters it needs to make it explode. Our program at the moment will explode it once, leaving it ready to explode again, but unless something else fortuitously explodes onto it again we'll never get around to exploding it the second time.

> Can you figure out how to solve this problem? *Hint:* Sheet L may help you. Beware though: your first attempt will probably go spectacularly wrong (well, mine did)!

Mouse hunt

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 Chain Reaction with. Fortunately, this is easy using the `games` library.

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 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!

Winning and losing

We nearly have a fully functional game of Chain Reaction. The only thing missing now is some way of ending the game.

According to the description, the game ends when one side or the other has all their counters wiped off the board. That means we need to keep track of how many counters of each colour there are. If you remember, we set up a dictionary called `num_counters` in the `ChainReaction` module to help us do this; now we have to keep it up to date.

First of all, every time we put a counter on the board, we add one to the number of counters that player has. It may cause a messy explosion, but that's something to deal with later. The `add_counter` function is the one that takes care of putting counters on the board for us, so it's there that we need to add our line. Just before we call `force_add_counter`, let's add the line:

```
self.board.num_counters[player] = self.board.num_counters[player] + 1
```

That was the easy bit. The harder bit is that when we explode onto a stack of counters of the other colour, we need to take the size of that stack away from one player and give it to the other. Thinking about it for a moment, the best place to do this is in our `force_add_counter` routine, so add the following lines just before we call `self.counter.higher`:

Line this up with self.counter.higher(player)

```
original = self.counter.get_colour()
if original <> player:
  height = self.counter.get_height()
  self.board.num_counters[original] =
    self.board.num_counters[original] - height
  self.board.num_counters[player] =
    self.board.num_counters[player] + height
```

> Now that we're keeping track of how many counters each player has, we could let the players know too. Can you figure out how to do this, using something like the method we use for telling the players whose turn it is?
> *Hint:* the co-ordinates that you use to make a `Text` object define where the *middle* of the text string will be. This might not be what you're expecting!

Finding the winner

Now that we have the numbers of counters in hand, we can work out if someone has won or not. We should do this check after each player adds a counter, except that if you remember our discussion some time ago we shouldn't check on the very first turn. This means adding the following line to our `do_counter` routine, just after we switch players and change the message:

Line up with self.current_player = ...

```
if self.check:
  if (self.num_counters[PLAYER1] == 0 or
      self.num_counters[PLAYER2] == 0:
    self.status.set_text("GAME OVER!")
    self.game_over = 1
    self.disable_cursor()
else:
  self.check = 1
```

Congratulations, you now have a fully working game of Chain Reaction!

Bells and whistles

There are still plenty of things that you can do to make your game more exciting. See if you can figure out how to do any of the following:

- Currently you can only play one game of Chain Reaction. 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 the function `map_grid` useful; the library provides this so that you can easily call a function on each box on the screen.
- If you are doing that, you might want to keep a score of how many games each player has won.
- Our explosions are a bit tame at the moment. There are several things you could do to fix this, but you'll need some help from a leader. You could make the exploding squares flash their colours (for which you'll need to the `Timer` class), or put up a graphic of an explosion (ask about the `Animation` of an explosion), or perhaps something else strikes your fancy.
- There's no reason to limit the game to just two players, or just the colours white and black. Try expanding the game to four players; you'll need to think hard about what the winning conditions are, and what to do about players who get knocked out. It might help to have a separate function to check for a win, otherwise your `handle_keypress` might get out of hand! You might also have to use different colours for the number on counters, just to keep it readable. If you do, think if a Python dictionary might help you.
- If more player is too much for you, how about less? Try writing a computer opponent, taking his turn at the end of `do_counter`. There are lots of easy algorithms to try out, like picking a square randomly, always going for a corner or edge if possible, or always exploding something if you can. If you can come up with a good strategy for the computer, let us know!
- You don't have to use a circle with a number on it to represent counters. You could change it to a number of circles, separate or stacked on one another, or different geometrical shapes, or even (not really recommended) different shades of colours for different numbers of counters.
- Anything else you can think of! Go on, surprise us.