# Game: Minesweeper

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 version of a classic computer game called Minesweeper.

## 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)

> *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 Minesweeper?

Minesweeper is a simple game played on a grid. Your job as the player is to find all the mines hidden in the grid without stepping on them. You may be familiar with versions of the game supplied free with some computers — now you'll be able to create your own version.

When the game starts, all you see is a blank grid. You have to pick a square to look at; if it's a mine, then it explodes and the game is over. If it isn't, then you find out how many mines there are in the surrounding squares. From this, you can eventually work out where the mines are, and you can mark them with flags.

Once you have flagged all the mines and cleared all the other squares, then you win the game.

To speed things up for the player, it is possible to clear out known squares more quickly. If, for example, you know that there is only one mine next to a particular square, and you already know where that mine is, then you can clear all the other squares around the square you know about in one go.

## What do we need?

So what things are there on the screen? From the brief description above, we can see that we will need the following things:

- a grid of squares,
- flags to mark where you think mines are,
- mines (when they explode, at least),
- some indication of how many mines are next to each square.

The LiveWires games library provides us with the first of these, as well as shapes that we can put on the screen to behave as mines, flags 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 know exactly what we can do.

## Boxes on the Screen

While the games library does provide a class to represent boxes on the screen, there are a few more things we would like our boxes to do for us. This means we are going to have to make a sub-class to get exactly what we want.

In fact, when you think about our squares on the screen, we actually have two distinct types of square that behave very differently some times. In particular, we have squares which have mines in them, which explode when we select them, and we have squares which don't have mines in them, which have to count up how many of their neighbours contain mines.

This suggests that we are going to want to make subclasses of our subclass! Fortunately, Python has no problem with that.

### In The Beginning

Before we get to the interesting stuff, we should tell Python that we are going to need the various libraries that we have mentioned. It's a good idea also to keep all the constants that you will want to refer to together at the beginning. 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 colour
from livewires import boards
from random import randint
```

First, we tell Python we wanted 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

MINE_SIZE = BOX_SIZE/2 - 3

NUMBER_SIZE = 24
TEXT_SIZE = 18

FLAG = ((BOX_SIZE/4, 2), (BOX_SIZE/2, BOX_SIZE-2), (3*BOX_SIZE/4, 2))
```

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. `MINE_SIZE` is a good example of that. We will use it as the radius of the circle that we will draw to represent a mine; 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.

### Making Boxes

Now we get on to our boxes on the screen.

```
class GridBox(boards.GameCell):
  def __init__(self, screen, i, j):
    self.init_gamecell(screen, i, j)
    self.flagged = 0
    self.shown = 0
    self.image = None
    self.flag = None
```

What this says is that our `GridBox` is a kind of `GameCell` from the `boards` module, and that we mostly use the same parameters to create one. We also set up some things that we will need later — variables to tell us if we have marked the box with a flag or selected it to show to the player, and variables to hold the flag shape on the screen itself and whatever else (a number or a mine) needs to be shown.

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.

### Mine, All Mine

Before we go much further, we should work out how to display a mine on the screen. This is something we ought to do at the end of the game so that the player can see where he was right and where he was wrong. This is simply a matter of drawing the mine on the board as a suitably sized circle. It would make it more obvious if we turned the rest of the box red, but that's just a detail. More importantly, if there is a flag in the box, we should make sure it is on top of everything else so that it can be seen. The following code will do the trick:

```
class Mine(boards.Container, GridBox):
  def __init(self, board, i, j):
    self.init_gridbox(board, i, j)
    self.init_container(['image', 'flag'])

  def reveal_mine(self):
    self.set_colour(colour.red)
    self.image = games.Circle(self.screen,
                              self.xpos() + BOX_SIZE/2,
                              self.ypos() + BOX_SIZE/2,
                              MINE_SIZE,
                              colour.black,
                              filled=1)
    if self.flagged:
      self.flag.raise_object()
```

All of this assumes that we keep `flagged` and `flag` up to date, so we'd better remember to do that!

Notice that while we've said that a `Mine` is a kind of `GridBox`, we've also declared it to be a kind of something called a `boards.Container`. This is a special class that the `boards` library provides to help when an object on the screen has things that will be displayed on top of it. Whenever our `Mine` is moved (which won't happen) or moved to the front of the screen (which the library will secretly do behind our backs), the `Container` class will make sure that everything in the box stays on top of the `Mine` itself. In our case that means the circle (`image`) representing the actual mine, and the shape (`flag`) that we'll use for a flag.

If this looks worryingly complicated, don't worry. The whole point of the `boards.Container` class is that it does all the work for you!

We aren't finished with the `Mine` class yet, but we'll put it aside for now. Instead, consider for a moment what an empty box would do if we asked it to show its mine. That's right, it wouldn't do anything!

> If you remember, Python's way of saying "do nothing" is the instruction `pass`. Armed with this, can you make a class called `EmptyBox` just like `Mine` above, but handling boxes which aren't mines?

### The Main Event

Finally, we need our class that represents the whole board, the collection of all the boxes and what to do with them. This will be a subclass of the `SingleBoard` class from the library, but it won't work in quite as simple a way as the library expects. Type in the following code to start with:

```
class Minesweeper(boards.SingleBoard):
  def __init__(self, n_cols, n_rows, mines_to_find):
    self.mines_to_find = mines_to_find
    self.flag_count = 0
    self.unknown_count = n_cols * n_rows
    self.game_over = 0

    self.init_singleboard((MARGIN, MARGIN), n_cols, n_rows, BOX_SIZE)
```

This just declares `Minesweeper` to be a kind of `SingleBoard`, records the things that we will need to keep track of, and initialises the `SingleBoard` parts of our new class. Once we have done that, we need to place the mines in the grid. This means that for each mine we need to keep picking boxes until we find one that doesn't already have a mine in it. Unfortunately, Python loops always carry out their test at the beginning of the loop when we really want to do the test at the end here. This leads to the following rather horrid piece of code, which you need to type immediately after what you have typed above.

*Remember to indent correctly*

```
    for m in xrange(mines_to_find):
      while 1:
        i = randint(0, n_cols - 1)
        j = randint(0, n_rows - 1)
        if self.grid[i][j] == None:
          break
      self.grid[i][j] = Mine(self, i, j)
```

What this says is "for each mine, loop forever picking a box and if it hasn't got anything in it break out of the loop." Not pretty, but it works.

Once we have done this, we need to go through the rest of the boxes in the grid turning them into EmptyBox objects. Once we have done that, there are two more useful things that we can do. First, the library can search through the grid for us making a list of which boxes are the neighbours of which and putting the results in a list called `neighbours` in each box. Second, the library will run a pointer for us that we call a "cursor." This puts a red outline around the box that we want to look at or put a flag on, and moves it around for us according to the left, right, up and down arrow keys.

*Align this with the previous ''for''*

```
    for i in xrange(n_cols):
      for j in xrange(n_rows):
        if self.grid[i][j] == None:
          self.grid[i][j] = EmptyBox(self, i, j)

    self.create_neighbours()
    self.enable_cursor(0, 0)
```

We still need to do a little work to get the libraries to do exactly what we want. When we first initialise the `SingleBoard`, it will try to create the boxes for itself. Since we want to create the boxes in our own peculiar way, we have to tell it not to do the work. We do this by overriding the function `new_gamecell`, which should normally returns a box of the type we want. In this case, `None` will do just fine, since we do the rest of the work later.

We now nearly have enough to run the program and see something happen. What we will do, just to see it run, is to pretend that we exploded a mine as soon as we started and end the game, showing all the mines. To do this we need to write one last function in the `Minesweeper` class and a separate useful function:

*Align with the ''def __init__''*

```
  def new_gamecell(self, i, j):
    return None

  def end_game(self):
    self.game_over = 1
    self.map_grid(reveal_all_mines)
    self.disable_cursor()

def reveal_all_mines(cell):
  cell.reveal_mine()

minefield = Minesweeper(8, 8, 10)
minefield.end_game()
```

Now run it and see what happens! The function `map_grid` that we call in `end_game` is a useful little thing in the library that calls the function that it is given on each box in the grid in turn.

## See What's There

So far we can blow up immediately after we start. Fun as that is, it's not much of a game. Let's see about searching boxes to see what is in them.

The first thing we need to do is to decide how we want the player to tell us to search a box. The easiest thing to do would be for the player to press some key when the cursor is over the box in question; arbitrarily we pick on the Return key.

When the `boards` module sees the 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, sending us a code to indicate which key has been pressed. For the Return key, the key code is the constant `games.K_RETURN` that the library provides for us.

Now that we know how to find out to look into a box, we then have to decide what to do about it. The simple answer is to ask the box to do it for us — after all, it knows what it is! What we will need to know is whether or not we just stepped on a mine, which is something that the box can return to us. If we did trigger a mine, we should end the game, and we already know how to do that!

All this comes down to adding the following function to the `Minesweeper` class:

*Indent to match the others*

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

  if key == K_RETURN:
    if self.cursor.reveal_box():
      self.end_game()
```

The `cursor` that is used above is something that the library keeps up for us. It gives us the box that the cursor is currently over on the screen, which we can then treat like any other box.

Having done this, the next thing to do is to write the `reveal_box` function that we have just asserted that the boxes will have. Clearly mines and empty boxes will behave differently, so we should write different functions for the `Mine` and `EmptyBox` classes.

For a mine, if it's stepped on we need to return a "true" value (i.e. anything that isn't zero). We would also like to explode the mine and make it visible, but just consider for a moment what is going to happen. When we return to the `handle_keypress` function that we've just written, it is going to end the game and show all the mines. If we take any special action to show our own mine, it will be done all over again, which is a bit of a waste. In fact, we can just let the `end_game` do all the work for us!

There is one last little wrinkle here; what do you do if there is already a flag on the mine? We could just ignore the flag and explode anyway, but putting a flag down normally means that the player thinks that there is a mine here. It's quite likely that the player pressed Return by mistake, and meant to do something else entirely. It's no trouble for us to be helpful and pretend that nothing happened if there is a flag there.

In other words, all we have to do is to return a true value if there is no flag (i.e. if `self.flagged` is false), and to return a false value otherwise.

> You should be now to be able to write the `reveal_box` for `Mine` yourself. Here's a hint that may make your function even shorter: if you use `not x` you get a value that is true when `x` is false, and false when `x` is true.

It's a little trickier to write `reveal_box` for an empty box. There are a few decisions we can make quickly.

- If the box has already been revealed, there is nothing to do.
- If there is a flag on the box, we should do nothing just like we do for a mine. Otherwise we are giving things away, and giving the player a particularly easy way to cheat.
- We should set `self.shown` to a true value, so that we will know if we try to reveal this box again.
- It would nice to change the background colour of our box if we do show it, just to make it more visible.
- When we are done, we need to return false (0) to show that we haven't revealed a mine.

A more difficult challenge is presented by the description of the game. According to that, we need to show how many of the surrounding squares contain a mine. Fortunately the library helps us here by providing a list of the surrounding boxes in `neighbours`. After that, all we need is some way of asking each neighbour whether or not it is a mine. We can write a simple function for that, which will allow us to count up the mines.

Once we have the number, we still need to display it. The library `Text` object provides a way to display a string, but it won't display numbers. Fortunately, the standard Python function `str` converts a number into a string for us.

*Type this in the EmptyBox class*
*Remember to align the defs!*

```
def reveal_box(self):
  if self.shown or self.flagged:
    return 0

  count = 0
  for n in self.neighbours:
    if n.is_mine():
      count = count + 1

  self.set_colour(colour.white)
  self.image = games.Text(self.screen,
                          self.xpos() + BOX_SIZE/2,
                          self.ypos() + BOX_SIZE/2,
                          str(count),
                          NUMBER_SIZE,
                          colour.blue)
  self.shown = 1
  self.board.revealed_box()
  if count == 0:
    for n in self.neighbours:
      n.reveal_box()
  return 0
```

> You still have to write the `is_mine` function for mines and empty boxes. That's up to you; just remember that `1` means true and `0` means false!

If you've been paying attention, you'll notice that we've sneaked two extra things into the code above without telling you. First, there is a call to `self.board.revealed_box`, which will be a function in the `Minesweeper` class that keeps track of how many boxes have been looked into. We'll need that later to work out when the game is finished.

The definition of `revealed_box` is very simple:

*Add this to the Minesweeper class*
*Don't forget to indent it!*

```
def revealed_box(self):
  self.unknown_count = self.unknown_count - 1
```

The second thing is just something to make life easier for players. If there are no mines in the surrounding boxes (i.e. if `count` is zero), then it is safe to look in to all of them. We do that on the player's behalf so that he doesn't have to do it manually for each of them.

There is only one thing we need to do to be able to run this now; we have to replace the `minefield.end_game()` instruction with something that will actually run our program:

```
minefield.mainloop()
```

There, that was easy!

### Mouse Hunt

While racing around the window using the keyboard is all very well, it might be easier to use the mouse to play Minesweeper with. Fortunately the libraries make this fairly easy for us.

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 the mouse button. The libraries give us two parameters to this function; first the coordinates of where on the screen the mouse was when we let go, and second the button that was let go. The buttons are numbered from 0 to 2, with 0 being

the left button, 1 the middle button, and 2 the right button.

If you think about how you normally use the mouse on the computer, you normally use the left button to select things. Whether it's a file to open, some text to cut or copy, or part of a picture to apply an effect to, the left button is the one we use to select it. In our game, the nearest idea we have to that is looking into a box to see what's there, the same thing that the Return key does. So what we need to do is to work out which box the mouse is pointing at (assuming it is pointing to any box at all), move the cursor there and do the same for button 0 as we do for the return key.

*Add this to the Minesweeper class*
*Get the indentation right!*

```
def mouse_up(self, (x, y), button):
  if self.game_over:
    return

  i,j = self.coords_to_cell(x, y)
  if not self.on_board(i, j):
    return
  self.move_cursor(i, j)
  if button == 0:
    if self.cursor.reveal_box():
      self.end_game()
```

## Flying the Flag

So far we can look into boxes for mines, but we can't plant flags on them so we still can't win the game. So what steps do we need to take to do that?

First, we need to extend our `handle_keypress` function to cope with an extra keypress to mean "plant a flag." Arbitrarily, we pick the space bar for this one, so we add the following immediately after the the `if` statement dealing with the Return key:

*Line up the ''elif'' with the ''if''*

```
elif key == games.K_SPACE:
  self.cursor.toggle_flag()
```

> You will want to do something very similar for the mouse too. Try adding making the right button (button 2) do the same as pressing the space bar.

Now, if the box has already been shown to the player, then there is no point in trying to put a flag on it because the player knows it isn't a mine. We can assume that he hit the button by mistake.

To put a flag on the box, we will need to draw the shape of the flag on the screen in the right place. We can get the coordinates of the box from the functions defined in the library, so that's easy. The hard bit is the flag shape, which is a set of coordinate pairs. We have defined something to be going on with in the constant `FLAG` a little earlier.

Thinking ahead a little, we are going to have to keep track of the number of flags that we have. Clearly no one box has any idea what the other boxes are doing, so we will need to palm this off on the `Minesweeper` class that contains all the boxes. For now, we'll just define what we're going to do later; in particular we will say that that we will write a function to add or subtract from a total number of flags displayed that we will have to keep somehow.

If there is already a flag in the box, we should remove it if we are asked to. The player may have put it down by mistake, after all. To do this we need to remove the flag object from the screen by calling its `destroy` method (a "method" is another name for a function that belongs to a class rather than existing on its own).

Notice that in all of this, it doesn't matter whether the box we are flagging is a mine or not. This suggests that our `toggle_flag` function should go into our `GridBox` class, since it applies to both `Mine` and `EmptyBox` types of boxes equally.

The following code will do all this for us. Be careful when you type it in that the `def` that it starts with lines up with the `def` for the `__init__` function in `GridBox`.

*Align with previous def!*

```
def toggle_flag(self):
  if self.shown:
    return
  if self.flagged:
    # We have already flagged here.  Remove it
    self.flag.destroy()
    self.flag = None
    self.screen.update_flag_count(-1)
  else:
    # Not yet flagged, so make one
    self.flag = games.Polygon(self.screen, self.xpos(), self.ypos(),
                              FLAG, colour.green, filled=1)
    self.screen.update_flag_count(1)
  self.flagged = not self.flagged
```

Updating the count of flags is in fact easy. Just type the following method into `Minesweeper`:

*Again, it must line up*

```
def update_flag_count(self, adjustment):
  self.flag_count = self.flag_count + adjustment
```

This will add one if we give it a parameter of `1`, and subtract one if we give it a `-1`, just like we want. Try it and see!

> The flag isn't a very pretty shape at present. Why don't you experiment with the coordinates we put in `FLAG` right at the beginning and see if you can make something that looks more appropriate?

## How To Win

We still have the problem that we don't know when the player has won. Since players like winning, we had better fix that.

Thinking about it, the player will have won if all the mines are correctly flagged and the player knows it. In other words, all the non-mine boxes must have been looked into, and all the other boxes must be flagged.

Looking back, we have been keeping track of two things in the `Minesweeper` class. First, we have kept track of the number of flags that are showing in the variable `flag_count`. Clearly, this needs to be the same as the number of mines we were looking for. The second thing that we have been watching is the number of boxes whose contents are unknown, `unknown_count`. Clearly again, the only unknown boxes should be those with mines in them. When both of these conditions are true, all the unknown boxes are flagged and contain mines, and there are no other mines (or we would have already lost).

This leads us to a nice simple function in `Minesweeper` for checking if we're done:

*Remember to align the defs!*

```
def check_for_win(self):
  if self.flag_count == self.mines_to_find == self.unknown_count:
    self.game_over = 1
```

Then we just need to call this function (as `self.check_for_win()`) every time that the player does something to change the board, i.e. after pressing a key or clicking on the mouse.

> Well, what are you waiting for?

## Telling The Player

Now we have a working game of Minesweeper. There's just one problem; the player doesn't know that he's won or lost, or how many mines he hasn't found yet. What we need is some way of getting messages out.

The games library provides us with a means of drawing text on the screen, as we saw when we were writing the number of mines surrounding a box earlier on. We can use `Text` objects to show our useful information as well, though we need to put them somewhere slightly off the board so that they don't get in the way of anything important.

We really need two places for messages, one at the top of the screen to tell the player when he has won or lost, and one at the bottom to keep track of how many mines have been put down. Both of them need to be added to the `Minesweeper` class `__init__` method. Here's the first one:

*Remember to align this*

```
x,y = self.cell_to_coords(n_cols/2, -1)
self.status = games.Text(self, x, y + BOX_SIZE/2,
                         "M I N E S W E E P E R",
                         TEXT_SIZE,
                         colour.white)
```

This puts the message off the top of the board area (in row "-1", if such a thing existed), with its middle point halfway across the board.

> Can you work out how to set up a message telling the player how many mines are yet to be found? Put it off the bottom of the grid (which has rows numbered from `0` to `n_rows-1`) and store it in a variable called `self.mine_count`.
> You'll need to turn the number of mines to be found (`mines_to_find`) into a string and append it to a few words of explanation — do you remember how to do that?

Once you have created the `mine_count` message, you will need to keep it up to date. We must change the number every time we add or remove a flag, but fortunately we already ensure that `update_flag_count` is called every time the number of flags changes. So all we need to do is to add the following line to that function:

*Remember to line it up*

```
self.mine_count.set_text('Mines remaining: ' +
                         str(self.mines_to_find - self.flag_count))
```

> We also need to change the status message to congratulate the player when he wins and commiserate when he loses. Can you work out how to do that?

## Making Things Quicker

We have a working game now, but there are a couple of things we can do to make life a little easier for the player. The main extra trick that other versions of Minesweeper have is the ability to reveal what's in surrounding boxes when the player has flagged all the surrounding mines, or at least thinks he has.

Let's pause a moment to work out what we need to do for this:

- If the player hasn't yet looked in the box, we should ignore any request to do this.
- We then need to count up the number of flags and the number of mines surrounding the box. If they aren't the same, we shouldn't go any further because the player hasn't counted properly.
- If we have got the right number of flags, we show the contents of each neighbour that isn't flagged. If it's a mine, the game is over because the player made a mistake.

When you look at this list, everything is done from the point of view of the box. It doesn't actually matter whether the box is empty or not, although obviously if it is a mine then it won't have been looked into yet so we will stop at the first hurdle. These things all suggest that we should do all the work in our `GridBox` class, the one that handles all boxes.

The first part is easy. Add this to the bottom of the `GridBox` class definition:

*Indent to line up with the other defs*

```
def reveal_known(self):
  if not self.shown:
    return 0
```

We're thinking ahead a little by returning 0 (false) here. If you look at our third bullet point above, we might have to end the game if the player was wrong, so we need to return an appropriate value to tell the rest of the program. The last time we did this, for `reveal_box`, we returned 1 for an exploded mine and 0 for a safe choice. For consistency's sake, we should do the same here.

Next, we need to go through all our neighbours and count things up. Remember how to do that? It's what the `neighbours` list is for:

*Line this up!*

```
number_of_flags = 0
number_of_mines = 0
for n in self.neighbours:
  if n.is_mine():
    number_of_mines = number_of_mines + 1
  if n.flagged:
    number_of_flags = number_of_flags + 1
```

Finally, if everything is to our liking, we need to run through the list of neighbours again and show them all.

*Indent again*

```
if number_of_flags == number_of_mines:
  for n in self.neighbours:
    if not (n.flagged or n.shown):
      if n.reveal_box():
        # Oh dear, a flag was in the wrong place
        return 1
                                              Line this up with the first 'if'
return 0
```

All you need to do now is to call our new function when an appropriate key is pressed, and end the game if it returns a true value.

> Go on then! Pick a key and add another `elif` clause to the function `handle_keypress`. You may need to ask a leader the name that the library uses for an appropriate key, but if you want an ordinary key then it's easy. The 'a' key is called `games.K_a`, 'b' is `games.K_b` and so on.

That's it! You now have a fully functioning game of Minesweeper.

## Bells and Whistles

There are a few things you could do if you wanted to improve your game. See if you can figure out how to do any of the following:

- Currently you can only play one game of Minesweeper. 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 player presses a given key after the end of the game. Hint: you need to `destroy` objects on the screen before they will disappear.

- Some versions of Minesweeper keep track of how long you take to win the game. See if you can work out how to do this; ask a leader about the `Timer` class in the games library.
- If you're doing both of the above, try keeping a high score table. Don't worry about saving it, just keep the table for the time that the program is running.