Game: The Fourteen-Fifteen Puzzle

Rhodri James Revision 1.10, 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 Sam Loyd's classic Fourteen-Fifteen Puzzle.

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)
- Lists (from Sheet A)
- 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 the Fourteen-Fifteen Puzzle?

American Sam Loyd (1841–1911) was one of the most prolific puzzle designers of his day. He started young, becoming the problem editor of *Chess Monthly* magazine at age 16, and went on to sell puzzles, problems and tricks to a variety of newspaper and magazine columns throughout his life.

In 1878 he came up with the 14-15 puzzle, the Rubik's Cube of its day. The puzzle is deceptively simple; the numbers 1 to 15 are laid out on tiles in a 4x4 grid as shown below, with the 14 and 15 swapped around and an empty space in the bottom right hand corner. Your task is to get the numbers in the right order (i.e. with the 14 and 15 the right way round) by sliding adjacent tiles into the empty space.

1	2	3	4
5	6	7	8
9	10	11	12
13	15	14	

Loyd offer a reward of £1000 for a solution to the puzzle, a very large amount of money for the time. He could do this perfectly safely because the puzzle is impossible!

We are going to turn this puzzle (or at least a possible version of it) into a game. What we will do is to randomly put fifteen square blocks into a four by four grid, leaving one empty space. The player's task is to put the blocks into the right order, by successively sliding neighbouring blocks into the empty space. We could simply slide around numbers, as Sam Loyd did in his original puzzle, but to make it more fun we'll use a picture of Jen.

Things on the screen

This description doesn't require us to display very much. All we need is:

- fifteen blocks with fragments of the picture on them
- an empty space
- a grid to keep all of them in

Thinking about it, we don't actually need the grid; we can just display the blocks in a window and leave a big enough border to look good.

Nuts and bolts

Before we get into the interesting bits of the program, there are some bits of housekeeping that ought to do. 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 random import randint
```

First, we tell Python we want to access the things in the LiveWires `games` module, 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.

```
FRAGMENT_SIZE = 150
MARGIN = 60
SCREENWIDTH = 4*FRAGMENT_SIZE + 2*MARGIN
SCREENHEIGHT = SCREENWIDTH
SPEED = 10
```

After that, we set up `SCREENWIDTH` to hold the width of our window on the screen, `SCREENHEIGHT` for the height, `FRAGMENT_SIZE` for the size of our picture fragments, `MARGIN` for the size of the margin in the window that we want to use, and `SPEED`, which we'll find a use for later. We do this here so that we can easily change the size of the screen or the size of the fragments (if we choose to use a different picture) just by changing the program at this point. If we decide to use a picture that has fragments of size 100, all we have to do is to alter one line to read `FRAGMENT_SIZE = 100` and we can arrange to do everything else automatically. If we had just written the number 150 everywhere that we were referring to the height and width of the fragments, we would have to track down every time it was used and change them all individually, which would be a bit of a pain.

Worse, there may be times when our fragment size is used to determine another number and we don't spot it. `SCREEN-WIDTH` is a good example of that — we could have done the calculation in our head and simply written `SCREENWIDTH = 720`, a magic number that doesn't give much of a clue as to how we derived it. If we changed the picture fragment size, this would leave us with uneven margins, and possibly a picture that disappears of the edge if we're really unlucky. Doing it the way that we can avoid that problem without us having to even think about it.

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.

One last little function and we'll be ready to start on the real program:

```
def get_coords(i, j):
    return (MARGIN + i*FRAGMENT_SIZE, MARGIN + j*FRAGMENT_SIZE)
```

All this function does is to convert from a position in our notional grid (the `(i, j)` coordinates) to the actual screen coordinates (usually referred to as `(x, y)` coordinates). We'll need to do this several times, so we might as well write a function to save us typing it all the time.

The Picture class

We need a window to display our picture fragments in, something that the library calls a `Screen`. Since we need to do a few different things in our window, we'll make ourselves a subclass to make life easier for ourselves.

```
class Picture(games.Screen):
  def __init__(self):
    self.init_screen(SCREENWIDTH, SCREENHEIGHT)
    self.image = []
```

This says that a `Picture` is a kind of `Screen` from the `games` module. We use the constants that we created earlier to start up our screen, and we also make ourselves an empty list that we will eventually use to hold our picture fragments according to their position on the screen. That list will actually be a list of lists — have a quick read of Sheet A if that doesn't sound familiar.

Now we need to load each fragment and put it in its place on the screen. The files are named *jen0.bmp* (which is in the top left corner), *jen1.bmp* and so on down to *jen14.bmp*. We could type all of these names in one by one, but that would be tedious. Instead, we'll let the program generate the names for us.

jen0.bmp i=0 j=0	*jen1.bmp* i=1 j=0	*jen2.bmp* i=2 j=0	*jen3.bmp* i=3 j=0
jen4.bmp i=0 j=1	*jen5.bmp* i=1 j=1	*jen6.bmp* i=2 j=1	*jen7.bmp* i=3 j=1
jen8.bmp i=0 j=2	*jen9.bmp* i=1 j=2	*jen10.bmp* i=2 j=2	*jen11.bmp* i=3 j=2
jen12.bmp i=0 j=3	*jen13.bmp* i=1 j=3	*jen14.bmp* i=2 j=3	

As you can see from the diagram above, we can easily calculate which number file to use from its position in the grid. All we then need to do is to concatenate the different bits together to make the name — check out Sheet S is you're not sure what that means. We do have to convert the number that we calculate into a string before we can do this, but Python handily

provides the function `str` to do exactly that for us.

Using a combination of the useful functions that the library provides and Sheet A to tell us how to add things to lists, we can do all this fairly easily. Add the following to the `__init__` method that you just wrote.

You must line this up!

```
for i in xrange(4):
  self.image.append([])
  for j in xrange(4):
    if i <> 3 or j <> 3:
      fragment = games.load_image("jen" + str(i+4*j) + ".bmp", transparent=0)
      (x, y) = get_coords(i, j)
      self.image[i].append(games.Sprite(self, x, y, fragment))
    else:
      self.image[3].append(None)
self.blank_i = 3
self.blank_j = 3
```

Notice that when we load each image, we have to say that it isn't transparent. Otherwise we would find that bits of each fragment had inexplicably disappeared, leaving a rather odd-looking picture. We also carefully don't bother loading the missing picture for the bottom right hand corner, which is where our empty square starts out. We still put something into our `self.image` list for the empty space, the special Python object `None`. We also set up the variables `self.blank_i` and `self.blank_j` so that we can keep track of where the empty square is.

Testing, testing

It may not look like much, but what we've written is almost enough to run a simple test of our program. It won't do much yet beyond displaying Jen's smiling face, but it's a start. All we have to do is to add the following two lines:

No spaces at the start of these lines

```
picture = Picture()
picture.mainloop()
```

That's it. Run the program and check that you haven't made any typos this far!

Shuffle and deal

The next step is to shuffle the blocks around on the screen so that the player has something to unpick. We'll use a simple shuffling technique which thoroughly muddles everything up without being too much of a pain to write.

Line this up with ''self.blank_j = 3''

```
# Randomise the blocks
for i in xrange(4):
  for j in xrange(4):
    (x, y) = get_coords(i, j)

    i1 = randint(0, 3)
    j1 = randint(0, 3)
    (x1, y1) = get_coords(i1, j1)

    temp = self.image[i][j]
    self.image[i][j] = self.image[i1][j1]
    self.image[i1][j1] = temp

    self.image[i][j].move_to(x, y)
    self.image[i1][j1].move_to(x1, y1)
```

What this does is to step through each block and swap it with a random other block. It uses a couple of `for` loops to march

through each block in turn, calls `randint` twice to select a random block to swap with, swaps the two blocks over in the `self.image` array, and moves them on the screen. As a method of randomising it's a little bit dubious, but it works well enough for our purposes.

If you try to run this now, you'll find that we've forgotten something. Python will complain that it has found an `AttributeError: 'NoneType' object has no attribute 'move_to'`. This means that we tried to tell Python to move `None`, the special object that we put in our blank slot. Oops!

> Go on, it's your turn. Make the program not try to call `move_to` on our empty space. Remember when you find the blank that you will need to change `self.blank_i` and `self.blank_j` to tell us where the blank space now is.

Moving blocks

Now we have a suitably jumbled picture, it's time to figure out how to move the blocks.

Since there is at most only one direction any block can move in at a time, the obvious way to tell the program that we want to move a block is to click on the block with the mouse. Handily, the library will let us know that a mouse click has happened by calling the `mouse_up` method of our `Picture` class.

> *Since we don't have a* `mouse_up` *function at the moment, you might expect that clicking on the mouse would cause our program to explode messily. In fact it doesn't do that, because the library provides a* `mouse_up` *function of its own, hidden away in the* `Screen` *class, that does nothing at all. What we are going to do now is to write a new* `mouse_up` *that will be called instead.*

Our `mouse_up` function is given two parameters in addition to the ubiquitous `self`: the `(x, y)` position of the mouse in our window, and the number of the button that has been pressed. We don't much care which button has been pressed, but we do need work out which block on the screen our `(x, y)` coordinates correspond to. This is the inverse of the `get_coords` function that we wrote a while back.

So add this to the bottom of your `Picture` class definition:

Line up with the other `def`

```
def mouse_up(self, pos, button):
  i = (pos[0] - MARGIN) / FRAGMENT_SIZE
  j - (pos[1] - MARGIN) / FRAGMENT_SIZE
```

The next thing to do is to figure out if this is a block which can move. This comes in two parts. First, we have to ignore anything outside our 4x4 grid (since the mouse can be clicked anywhere in our window). This is pretty easy to write the condition for.

Second, we need to see if the block clicked on is immediately to the left, right, above or below the empty space. That means that either the `i` coordinate is the same and the `j` differs by 1, or the `j` coordinate is the same and the `i` coordinate differs by 1.

We'll use a little trick to cut down on the typing here. As part of our condition we could say "`j == self.blank_j+1 or j == self.blank_j-1`", or equivalently "`j-self.blank_j == 1 or j-self.blank_j == -1`". We can make this a little shorter and faster by using the `abs` function, which takes a number and makes it positive by multiplying by `-1` if it has to. This is what mathematicians call the *absolute part* of the number, and it's very handy. In our case it makes our expression into `abs(j-self.blank_j) == 1`.

In fact there's another optimisation to be done here. We said that when `j` differed by 1, `i` had to be the same, and vice versa. In other words, whenever `abs(j-self.blank_j)` is 1, `abs(i-self.blank_i)` must be 0, and vice versa. So when you add the two together, they are always 1 when we have a block that we can move, and never 1 otherwise.

That leaves us with our two `if` statements looking like this:

```
                                                    First check it's on the grid!
if i >= 0 and j >= 0 and i < 4 and j < 4:
                                                    Then check it can be moved
  if abs(i - self.blank_i) + abs(j - self.blank_j) == 1:
```

So how do we move a block into the empty space? The first thing that we have to do is to tell our block to move itself to the position on the screen that is currently blank. We need the `(x, y)` coordinates of the blank space to do this properly, at which point we can use the library's `move_to` function to do all the hard work:

```
                                                    Remember to indent this properly!
    (x, y) = get_coords(self.blank_i, self.blank_j)
    self.image[i][j].move_to(x, y)
```

Once we've done that, we need to change the `self.image` array so that our block is now in the right place, and the coordinates of the empty block are changed accordingly.

```
                                                    Line it up!
    self.image[self.blank_i][self.blank_j] == self.image[i][j]
    self.image[i][j] = None
    self.blank_i = i
    self.blank_j = j
```

That's it. You now have a functioning game based on the 14-15 puzzle.

Parity error

If you're really good at the puzzle, or you paid attention at the start of this worksheet, you might have noticed something by now. Every now and then, the program comes up with an arrangement of blocks which you can't get back into order no matter how hard you try.

The reason for this, and the secret that allowed Sam Loyd to put up a very large prize knowing that no one could ever claim it, is that there are two classes of arrangements of the blocks in the grid: let's call them "odd" and "even" parity. You can slide the blocks to get from any odd arrangement to any other odd arrangement, and from any even arrangement to any other even arrangement, no amount of block shuffling will get you from any even arrangement to an odd arrangement or vice versa.

We could spend ages trying to work out an algorithm for determining which parity of arrangement we've got, but fortunately there's a simple rule that we can take advantage of. From any even parity arrangement, if you swap one pair of tiles you get an odd parity arrangement. Likewise from an odd parity arrangement you get an even parity arrangement by swapping a pair of tiles. All we have to do is to ensure that when we shuffle the picture, we do an even number of swaps. This means tweaking the shuffling part of our `__init__` routine.

At first sight this is trivial; we swap each of the sixteen grid spaces once, making a total of sixteen swaps, so we're done aren't we? Well, no, unfortunately not. There are two small problems.

The first and most obvious problem is that we can swap a block with any other block, *including itself.* Obviously if we swap a block with itself we haven't changed anything, so it doesn't count.

The second problem is a bit more invidious. Imagine that we have all the blocks in their desired position and just swap the `15` with the empty space. Since you can slide the `15` left to get back to the original position, this clearly hasn't changed the parity. On the other hand, if we swapped the `14` with the empty space then we could slide first the `15` then the `14` left and have Sam Loyd's original impossible puzzle, so we have changed the parity.

There are two basic approaches to solving the problem and leaving the player with a picture that can be put together properly. Either you can forbid any of the swaps that might cause trouble, or you can count up how many parity-altering swaps you have made (or equivalently how many non-parity-altering swaps you have made) and do an extra swap if needed. We aren't going to tell you how to do either of these things, but we will sketch it out for you.

Forbiddance

If you are going to force every swap to change parity, then you need to know a little more about Python's loops. You really want to be able to say the program equivalent of "do this thing (pick a block to swap with) *until* some condition (we pick a valid block)." Unfortunately Python's `while` loop doesn't work like that; it's more like saying "*while* this condition is true, do that." Python tests the condition first, which isn't what we want here.

Fortunately we can fudge up what we want using the `break` instruction. This tells Python to stop it's loop now, we want to get off, so our solution is to tell Python to loop forever and explicitly `break` out when we're happy.

The tweak that you'll need to make to the shuffling will therefore leave each individual block swap looking something like this:

```
while 1:
  pick a block to swap with
  if the swap changes parity:
    break
swap the blocks
```

> *Hint:* You can get away a loose condition here; swapping a block with the empty space might change parity or it might not, so if you exclude it you are definitely safe, even if you might have chucked out a perfectly valid possible swap.

Counting

For this option you'll need an extra variable to tell you if you've done an odd or even number of swaps. Then every time you do a swap that changes parity you flip your variable from true to false or vice versa. It's slightly less work (and therefore more elegant) to instead keep track of which swaps don't change parity, and therefore whether you need to add an extra swap at the end.

This will leave the shuffling looking something like this:

```
extra_swap_needed = 0
for each block:
  pick a block to swap with
  if the swap doesn't change parity:
    extra_swap_needed = not extra_swap_needed
  swap the block
if extra_swap_needed:
  swap a pair of blocks
```

> This time you really do need to properly work out whether swapping a block with the empty space changes parity. Here's a hint: consider how many swaps you need to make of your starting block with adjacent blocks until the block is next to the empty space, at which point you can just slide everything along to get effect you were after. Work out whether those swaps overall caused a change of parity, and see if you can figure out how to get the program to calculate that.

> One last hint: you might as well fix which pair of blocks you swap for the extra, as long as neither of them are the empty block! If one of them is empty, you can always fall back to a different (fixed) pair.

Sliding blocks

We can make our game a bit more visually interesting if we make our blocks slide into place rather than just popping across instantly. Fortunately the library makes this quite easy for us.

We currently make each block into a `Sprite`, the class that the `games` library provides for pictures like these. It also provides a class called `Mover` which as you might expect supports making objects move on the screen. We now need to define our own class that includes both of these classes, because we need things from both.

```
class Block(games.Sprite, games.Mover):
  def __init__(self, screen, x, y, image):
    self.init_sprite(screen, x, y, image)
    self.init_mover(0, 0)
    self.target_x = x
    self.target_y = y
```

This declares our `Block` as a kind of `Sprite` and a kind of `Mover`, then lets those classes sort themselves out by calling `init_sprite` and `init_mover` respectively. It also records the *target position* of the block, i.e. where we want it to end up, which for now is exactly where it is.

If you now change the line in the `Picture` class `__init__` function so that it `append`s a `Block` rather than a `games.Sprite` (conveniently with exactly the same parameters), you'll be able to run the program immediately. Of course, it won't do anything different yet.

Starting the slide

We initialise our `Block`s to be moving at a velocity of `(0, 0)` (that's what the call to `init_mover` means). To slide them from one place to another, we will need to call the library routine `set_velocity` to get them moving. That's pretty simple; just add the following to the `Block` class:

Line up the defs!

```
  def slide_to(self, x, y):
    self.target_x = x
    self.target_y = y
    (old_x, old_y) = self.pos()

    if old_x < x:
      dx = SPEED
    elif old_x > x:
      dx = -SPEED
    else:
      dx = 0

    if old_y < y:
      dy = SPEED
    elif old_y > y:
      dy = -SPEED
    else:
      dy = 0

    self.set_velocity(dx, dy)
```

This doesn't need a lot of explanation. First we update where we are aiming for, then we figure out whether we are going left or right, up or down, and change the value that we are going to put into `set_velocity` appropriately.

Now change the `move_to` in our `mouse_up` function to a `slide_to` and see what happens.

Brakes. We need brakes

Now we start our block sliding into position, but it doesn't yet stop when it gets there. We need some way of figuring out when it is in position, at which point we can set the velocity back to zero.

Fortunately the library again comes to our aid. After every movement that the block makes, the library will call a routine named `moved`. As usual, it provides a dummy version internally that does nothing, but we will want to write our own version.

We need to check to see if the block has reached the right place (i.e. if it is at `(target_x, target_y)`), and if so stop the block. That's simple enough:

Put this in the `Block` *class*

```
def moved(self):
  (x, y) = self.pos()
  if self.target_x == x and self.target_y == y:
    self.set_velocity(0, 0)
```

Try it now, and all seems to be working perfectly. However, there is a hidden problem.

Moving targets

The problem becomes more obvious if you slow the motion down. To do that, all you have to do is to change the value of `SPEED`, a constant that we set up right at the top of the program. A value of 10, which is what we started with, slides our blocks smoothly but rapidly into place. If you turn it to 1, blocks slowly sail across the window.

> *You can tune the value of* `SPEED` *so that the sliding looks right to you. Try experimenting with several different values, and you will see a problem with some values: the blocks again fail to stop in their correct spots and sail blithely off the edge of the window. Can you figure out why this happens?* Hint: *consider the relationship between the values of* `SPEED` *and the value of* `FRAGMENT_SIZE`.

Set `SPEED` to 1 for the following test. All you have to do is to juggle three blocks around, sliding each one into the empty space in turn. You can click on the next block to start it sliding before the first one has reached it's place.

So far so good, and indeed quite pretty. However at this speed it's entirely possible for you to click on a block to get it moving to its next position before it's finished getting to its current target. When that happens, disaster strikes: the block changes direction immediately, so never reaches the place it should be in. For example, if the block was sliding to the right and you click on it again to move it to the empty space below, it immediately starts moving downwards as well as right. A little plotting out the results on paper should show you that the block then won't be far enough down when it's far enough right, and will be too far right by the time it's far enough down, so our `moved` routine will never cut in and stop it.

To fix this, we need to go back to our `moved` routine and rewrite it almost completely. It's not enough to halt the block when it reaches the right spot, we have to deal with it being in only one of the right horizontal and vertical positions. That means separating out our `x` and `y` tests, and keeping track of how fast we were originally going.

Replace the current `moved` *with this*

```
def moved(self):
  (x, y) = self.pos()
  (dx, dy) = self.get_velocity()
  if self.target_x == x:
    dx = 0
  if self.target_y == y:
    dy = 0
  self.set_velocity(dx, dy)
```

Bells and whistles

You've now written a complete and attractive version of the 14-15 puzzle. There are a number of things you could do to improve it still; see if you can figure out how to do any of the following. You will probably need to read Sheet W for some of the details.

- Display a congratulatory message when the player manages to reassemble the whole picture. To do this you'll need to keep track of which image is supposed to be where — extra parameters to your `Block` initialiser — and create a `games.Text` object with your message in it when the last block is in place. You may also want to disable the mouse at this point.
- If you've done that, you can start a new game when the player next clicks the mouse.
- You could also display a timer so that the player knows how well he is doing. The library calls a method called `tick` in our `Picture` class every twentieth of a second, and you can use that to keep a track of elapsed seconds which you can display in a `games.Text` object.
- If you're feeling very adventurous, you can keep a high score table of the fastest times for solving the puzzle.
- It's trivial to change the program so that the blocks all slide into their shuffled positions, so why not do that?