Game: Tetris

Rhodri James

Revision 1.15, October 7, 2001

Credits

Introduction

This is a Python games worksheet. When you've finished it, you'll have written a version of a classic computer game called Tetris.

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 Tetris?

Tetris is a simple but addictive computer game. Differently shaped blocks fall down the screen. You have to rotate the blocks and move them to the left or right as they fall so that they form complete lines, which will then disappear, before the stack of blocks reaches the top of the screen and ends the game.

There are lots of versions of Tetris out there. This is the basic version with 4-block shapes. However once you've programmed that, there's nothing to stop you from working on 5-block shapes, special effects and so on.

What do we need?

So what things will there be on the screen? Fairly obviously there will be the screen itself, or at least a part of the screen for the blocks to fall down. On top of that we need the shapes themselves. All of our shapes are made up of four squares arranged so that each square shares an edge with at least one other. That's pretty much it, though, with the possible exception of some text so that we can keep score.

The LiveWires `games` library provides us with all of these things. They come as classes, as you might expect from Sheet O, and we will need to make sub-classes of them so that they do the things we want them to do.

I have a cunning plan, Milord

Before we start, let's take a moment to think about how Tetris will actually appear on the screen.

We have a number of connected squares, falling down the window. Eventually they will hit other squares and stop. When that happens, each row on the screen that consists entirely of squares with no empty spaces disappears, and all of the rest of the squares, no matter what shape they used to be in, are moved down one line too.

One thing to notice from this is that we need to think in terms of rows full (or not) of squares. We also need to think about columns of squares as well, because when we move blocks sideways we need to do so in units of a whole square's width.

We can most easily do this by pretending that the screen is in fact a grid of squares; our shapes start at the top, moving down one grid row at a time, and moving left or right one grid column at a time when we press the appropriate keys.

A grid of squares. Hmm. That sounds a bit like a chess board, and indeed the LiveWires `boards` library uses the `games` library to make chess boards and the like. We'll use that to make our lives easier!

Starting off

Before we get on to the interesting stuff, we need to tell Python the boring bits about which libraries we need. It's a good idea also to keep all the constants that you will want to refer to together at the top of your program. That way you can find them whenever 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 all at once).

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

First, we tell Python we want to access things in the LiveWires `games` and `boards` modules, and all the colour names in the `colour` module. We will also want to use various random number functions, which live in the standard library's `random` module. Don't worry, you've used them before; `random.randint` is the function we disguised as `random_between` in the beginner's library, for instance.

```
BOX_SIZE = 30
WIDTH = 8
HEIGHT = 16
MARGIN = 2*BOX_SIZE
INTERVAL = 40
TITLE_SIZE = 32
SCORE_SIZE = 24

EMPTY_COLOUR = colour.light_grey
BLOCK_COLOURS = [ colour.red, colour.green, colour.blue ]
```

Then we set up `BOX_SIZE`, which we'll use as the height and width of each square in our grid, `WIDTH` and `HEIGHT` to be the number of boxes we want across and down the screen respectively, and `MARGIN` as the gap between the boxes and the edge of the window, as well as some other useful things. We do this so that we can easily change the size of the boxes (for example) if we want to, just by changing the program at this point. If we decide that boxes of 30 units are too big for us, all we have to do is to change that line to read `BOX_SIZE = 20` and if we've written the rest of the program properly it will all just happen automatically. If we'd simply used the number 30 everywhere we were referring to the height and width of our boxes, we would have to track down every place we'd done that and change them all individually. That would be a pain, apart from the risk of us missing one or changing a 30 that was actually nothing to do with our box sizes.

Worse, there may be times when our box size if used to determine another number and we don't spot it. We do exactly that here when we set the `MARGIN` (the gap around the grid we're playing on) to be twice the size of any one box, but there are more interesting examples in other games that would look very silly if we missed them. If we're smart and calculate everything from `BOX_SIZE`, we won't 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's 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.

You'll notice that `BLOCK_COLOURS` is a list of different colours. What we're going to do is to give our different shapes different colours to brighten the game up a bit, so we'll pick a colour out of this list at random every time we make a new shape. Feel free to add more colours to the list — Sheet W has a list of all the colour names the `colour` module provides for you.

Little Boxes...

Now we get on to our boxes on the screen.

```
class TetrisBox(boards.GameCell):
  def __init__(self, board, i, j):
    self.init_gamecell(board, i, j)
    self.set_colour(EMPTY_COLOUR)
    self.full = 0
```

What this tells us is that `TetrisBox` is a kind of `GameCell` from the `boards` module, and that we mostly use the same parameters to create one. We also start the square off as not being a part of any shape, so we colour it in with `EMPTY_COLOUR` (since we handily defined that earlier) and we make ourselves a variable to tell us that the box isn't `full` of a shape!

Notice that we've got another couple of conventions here. We've given our class a name that has capital letters at the start of each word and no underlines separating them. We've also named our variable using no capital letters at all – we would have used underlines, but "full" is just one word. Once again, Python doesn't care what we call the class or whether variables have capital letters in them; it's all just a trick to make it easier for us to read our own programs. Trust us, these tricks make life a lot easier when you are trying to remember what your program did six months after you wrote it!

We haven't finished with the `TetrisBox` class yet, but first we'll look at something else.

The Board

We need a class that represents the entire grid of boxes that our shapes are going to fall down. The library proves the `SingleBoard` class to do just this, but that won't do all the things we need to do. The way to add more complicated bits to a class is to subclass it, just like we did to make our `TetrisBox` class, so let's do that. Type in the following code:

```
class TetrisBoard(boards.SingleBoard):
  def __init__(self, interval):
    self.init_singleboard((MARGIN, MARGIN), WIDTH, HEIGHT, BOX_SIZE)
    self.interval = interval
    self.tick_count = interval
    self.game_over = 0
    self.create_directions(orthogonal_only = 1)
```

This declares that `TetrisBoard` is a kind of `SingleBoard`, records some things that we will need to keep track of, and initialises the `SingleBoard` part of our new class. If all that sounds fairly familiar, don't be too surprised; a lot of computer programming is about finding similar ways of doing things!

The function `create_directions` is provided by the library to make a list of which cells (another word for squares or boxes) are next to one another. It builds up a list called `direction` that we'll use later to make our shapes fall and to move them left and right.

We say that two boxes are next to each other orthogonally *if touch each other on the sides, rather than just on the corners. In some ways it is the opposite of* diagonal. *So when we tell the library that we want* `orthogonal_only`*, we are saying that we want it to work out which boxes are neighbours on the left, right, top or bottom, but not diagonally to the top left or similar.*

Shapely Figures

As we said, we're going to be a little sneaky in how we make our falling shapes. We could draw a new `games.Polygon` on the screen, align it carefully with our grid and then reposition it when required, but why go to all that effort when we already have the `TetrisBox` class?

Instead, what we're going to do is to define that certain of our `TetrisBox` objects on the screen are in fact part of the falling shape, and when it moves we just change which boxes we consider to be part of the shape. To do that we're going to have to keep a list of which boxes belong in the shape, and change that when we move the shape. It may also need to do other things, like when we let the shape drop to the bottom of the window or when we rotate it. Is this beginning to sound like another class to you?

Put this near the top of your program, just after all the constants.

```
class Shape:
  def __init__(self, board):
    self.colour = random.choice(BLOCK_COLOURS)
    self.board = board
```

`random.choice` is a function from the standard Python library that takes a list of things and picks one at random. In our case it picks a colour, and records that as the colour we'll use for this shape.

You might be a bit surprised at that last line. Even if we secretly know that some time in the future we are going to have to make our shape do something to the game board (eliminate a full line, for instance), why do we want to keep another pointer to the board inside the `Shape` *class? Why not just use the* global variable *that we will have to create for the main program loop?*
The answer is more philosophical than practical. It is generally believed in writing computer programs that we should avoid global variables, ones that can be used from all over the program, wherever we can reasonably do so. There's nothing wrong with them as such, but if we ever need to change our program, global variables can make life a lot more difficult.
Suppose for example that we had multiple shapes falling down different grids on the screen all at the same time. Not very likely (and hard to work out how to control, for that matter), but it's possible to imagine an evil and twisted version of Tetris that did that. In that case, each shape would need to know which of the several possible grids it was on. The way that we've set things up here that would happen automatically, but if we had referred to a single global `board` *instead then we'd have a lot of work to do to change things round.*

Now we hit a problem. Our shape could look like any combination of four blocks that touch each other — a square, a line, a T, an L, and so on. Now we could record in our `Shape` class exactly which of these we've got, and write huge long lists of `if` statements in our movement functions saying "if it's a square do this, otherwise if it's a line do that," and so on. However, you should by now have recognised another possible solution to this problem. That's right, a sub-class.

We'll do a square shape first, since it's the simplest. Type this in after the (tiny) definition of the `Shape` class – Python needs to know what a `Shape` is before it'll let us define a subclass of it!

```
class SquareShape(Shape):
  def __init__(self, board):
    ∞                                      Uh-oh, here's trouble
    self.boxes = [ board.grid[WIDTH/2 - 1][0],
                   board.grid[WIDTH/2][0],
                   board.grid[WIDTH/2 - 1][1],
                   board.grid[WIDTH/2][1] ]
    self.place()
```

This makes a list of the boxes in the middle of the top two rows of the grid, calls `self.place()` (which we haven't yet written) to colour the grid in appropriately, and... does something else. What we'd like to do is the initialisation routine that we wrote earlier that does all the stuff that all shapes need to do. Normally that would be called automatically as the `__init__` function, but we've just written a new `__init__` function that hides the old one away. How can we get at it now?

There is a rather messy way to do it in Python, but it's a lot clearer if we take a leaf out of the library's book and make the `__init__` call something else to do all the work. Change the `Shape` definition so that it looks like this:

```
class Shape:
  def __init__(self, board):
    self.init_shape(board)                 This line is new

  def init_shape(self, board):             So is this one
    self.board = board
    self.colour = random.choice(BLOCK_COLOURS)
```

Now you work out what to type in place of the ∞ symbol.

We still need to write our `place` function. This is something that all sorts of shapes will need to do, so we put it in the `Shape` class so that they will all be able to get at it.

```
                                           Don't forget to line the ''defs'' up
  def place(self):
    for box in self.boxes:
      box.set_colour(self.colour)
```

Nice and short. This isn't quite the whole story, but don't worry about that yet. Notice in particular that we haven't used the `full` flag here — we're going to reserve that for shapes that have hit the bottom and stopped moving. That'll make it easier to tell later on whether or not we can move the shape a particular way.

Now that we've defined one kind of shape, we ought to create one when we start the game up. That means adding a line to our `TetrisBoard` class `__init__` function:

```
self.shape = SquareShape(self)             Line up with the other instructions
```

This isn't what we'll want to do eventually, but it'll do for now.

First Run

We nearly have everything that we need to be able to run the program and see something happen. We need one last function that the `boards` library asks us to provide in the `TetrisBoard` class:

```
                                           Align this with the ''def __init__''
  def new_gamecell(self, i, j):
    return TetrisBox(self, i, j)
```

Once we have that done, we can write the main program. In this sort of Python program, the "main program" is *very* short, because our classes and the library do all the work for us. In our case it takes a whole two lines!

```
tetris = TetrisBoard(INTERVAL)
tetris.mainloop()
```

That's it. Try running the program now and see what happens.

Falling Blocks

If you've done it right, you should find that a window appears showing a square block at the top which just sits there. Not wildly exciting, but at least it works. The next step is to start our blocks falling down the screen.

The library can help us here. It calls the function `tick` every fiftieth of a second (in theory – the library isn't always fast enough to keep up in practice), allowing us to make time-related changes to the game such as making our shape drop.

Hang on a sec, though. We haven't written a `tick` function yet, so why hasn't the library complained at us? The answer is that the `SingleBoard` class has a `tick` function itself, and the library is calling that since we haven't given it anything else to do. That function doesn't actually do anything, but its existence keeps the library happy.

So what do we need our `tick` function to do? All it needs do is to make our shape drop every so often. We don't really want the shape to move every fiftieth of a second, that would be far too fast, so we'll use the `interval` variable that we set up earlier. Let's make the shape drop every `interval` fiftieths of a second.

This goes in the `TetrisBoard` *class*

```
def tick(self):
  self.tick_count = self.tick_count - 1
  if self.tick_count == 0:
    self.tick_count = self.interval
    self.shape.drop()
```

How does a shape drop? It finds out which blocks are "down" from each of its current blocks, recolours its old blocks as empty and places itself in the new position. This is something that every shape does in exactly the same way, so we can make it part of the main `Shape` class.

This goes in the `Shape` *class*

```
def drop(self):
  new_boxes = []
  for box in self.boxes:
    box.set_colour(EMPTY_COLOUR)
    new_boxes.append(box.direction[boards.DOWN])
  self.boxes = new_boxes
  self.place()
```

Here `direction` is the list that the `create_directions` function made for us earlier, and `boards.DOWN` is a constant that the library provides to tell us which element of the list points to the box which is "down" from the box we're looking.

Dropping and Stopping

If you run the program now, you will see the square shape gracefully descend down to the bottom of the window. Unfortunately the program then explodes, complaining that " '`NoneType`' object has no attribute '`set_colour`' " or something similar. In other words, the block fell off the bottom of the window, where the boxes in the `boards.DOWN` direction are the special Python object `None` rather than a real box.

We need to stop the shape when it hits the bottom and start a new shape dropping. This is something to do at the beginning of the `drop` function; check each box that we are about to move to to make sure that it isn't `None`.

This goes after the "def drop"

```
      for box in self.boxes:
        next = box.direction[boards.DOWN]
        if next is None:
          for b in self.boxes:
            b.full = 1
          self.board.new_shape()
          return
```

Remember what we said about this?

> Can you work out what should be in our `new_shape` function? *Hint:* we did it already when we made our shape in the first place.
> In fact you should call `new_shape` in the `__init__` function rather than making it by hand. That way when you change it later (and you will, since we want a few shapes other than squares) then you won't have to remember to change that as well.

Stacking Blocks

This still isn't quite right. Our first square does fall down the window and stop at the bottom, starting a new square off at the top. Unfortunately it then falls down to the bottom again, squishing our original square. We need to make the squares stop when they come to an occupied box.

Fortunately we've just worked out how to stop the descending shape, and we've marked boxes as `full` when they have stopped moving. All we need to do now is stop the shape when the `next` box is `full` as well as when it's `None`.

> If you can't remember how to do this, look at Sheet C.

Once you've done that, everything looks fine. Squares fall down the screen, stacking on top of one another until they reach the top of the window. The only problem is that they keep stacking on top of each other once they've reached the top. What we should do is make the program check when it first puts the new shape into place that we aren't trying to sit on top of something that's already full.

This is something that needs to be done by the shape (since it's the only thing that knows where its boxes are), but needs to be done by all shapes when they are first created. Looking through our code, we find that the `Shape` class function `place` is the one called when the blocks are first placed on the screen, so it's there that we need to add something:

```
if box.full:
  self.board.game_over = 1
```

See if you can figure out where in `place` to put this. *Hint:* you need to make this check for every box.

Once we've done that, all we need to do is to stop any more blocks dropping down. We do this by changing the start of our `tick` routine, the one that makes everything move:

```
if self.game_over:
  return
```

That says "if the game is over, stop executing this function right now."

Not So Secret Messages

It would be nice if we told our player that the game was over. In fact it would be a good idea to work out how to tell the player things in general, because we'll need to do things like display the current score on the screen as well.

The library provides `Text` objects as a way of displaying messages. We'll use two `Text` objects, one for general status messages and one for the score. This means adding a few lines to our `TetrisBoard` class `__init__` function:

Put this at the end of `__init__`
Don't forget to line it up!

```
tx, ty = self.cell_to_coords(WIDTH/2, -1)
self.status_message = games.Text(self, tx, ty+BOX_SIZE/2,
                                 "TETRIS", TITLE_SIZE, colour.white)
tx, ty = self.cell_to_coords(WIDTH/2, HEIGHT)
self.score_message = game.Text(self, tx, ty+BOX_SIZE/2,
                               "Score: 0", SCORE_SIZE, colour.white)
self.score = 0
```

This asks the library where the middle of "row -1" (the row off the top of our grid) is on the screen, and puts the message "TETRIS" there. It then asks where the middle of "row HEIGHT" (the row off the bottom of the grid — remember that we count rows from number 0!) and puts the score message there. Finally, we start a score off at zero (otherwise we'd feel a little silly later). You may want to adjust the values of `TITLE_SIZE` and `SCORE_SIZE` to get the messages to be the right size for you.

Then we can make one small change in `place`. Instead of setting `self.board.game_over` we will instead call a function to make the board do the work for us, and change the status message at the same time.

Change `place` so that it reads:

```
if box.full:
  self.board.end_game()
```

Then we need to add this to the `TetrisBoard` class:

Align this properly!

```
   def end_game(self):
     self.game_over = 1
     self.status_message.set_text("GAME OVER")
```

Use whatever cheery message you feel like!

A Moving Experience

So far we have a game in which squares fall from the top of the screen, stack on top of one another and then end the game. Moderately pretty, but not very exciting as such. What we need now is some way of moving the squares right and left so that they don't all land in the same pile.

We will use the left and right arrow keys to make the block move left or right, which is all the moving that the game allows. What we have to work out is how to find out that those keys have been pressed.

Once again, the library comes to our rescue. It calls the function `keypress` in the `TetrisBoard` class whenever a key has been pressed, just like the name suggests. Like with `tick`, the `games` library provides a default version of `keypress` that doesn't do anything interesting. The `boards` library also provides its own version, but that doesn't do what we want either. We will have to write our own, as usual.

When the library calls `keypress`, it provides a *key code* which represents the key which has been pressed. The codes that we need for the left and right arrow keys are also defined in the library for us: `boards.K_LEFT` and `boards.K_RIGHT` respectively. We will use these codes to tell our shapes to move themselves.

Add this to the `TetrisBoard` *class*

```
  def keypress(self, key):
    if self.game_over:           # If we're finished
      return                     # don't do anything
    if key == boards.K_LEFT:
      self.shape.move(boards.LEFT)
    elif key == boards.K_RIGHT:
      self.shape.move(boards.RIGHT)
```

What does the shape need to do in order to move left or right? Something that looks a lot like what it has to do to drop down! The only difference is that when we try to move into a non-existent space or one which already contains a bit of a shape, we should just refuse to move rather than try to launch a new shape.

All `Shape`*s do this...*

```
def move(self, direction):
  for box in self.boxes:
    d = box.direction[direction]
    if d is None or d.full:
      return
  for b in range(4):
    box = self.boxes[b]
    box.set_colour(EMPTY_COLOUR)
    self.boxes[b] = box.direction[direction]
  self.place()
```

Going Down

One of the usual controls on a game of Tetris tells the computer to drop the current shape as far down the window as it will go. When you think about it, that's just a matter of calling `drop` until it doesn't drop any more but makes a new shape instead. But how do we know that `drop` has done all it can?

The simple answer is that we make `drop` tell us. We make it into a function that returns us a value rather than something that does its work without telling us anything. Ideally we would like `drop` to answer the question "Have you finished?" with a true or false response. If you remember from the beginners course, Python regards a zero value as false, and everything else as true – conventionally we use one to mean true, but anything non-zero would do.

> This is a very small change to your existing function: just change the `return` statement into a `return 1` (meaning "we finished"), and put a `return 0` ("we didn't finish") at the end of the function.

We then need to call our new improved function.

This also goes in `Shape`

```
def plummet(self):
  while not self.drop():
    pass
```

This says "while we haven't finished dropping down, do... er... nothing. Then stop." All we have to do now is to call our `plummet` function.

> If we tell you that the space bar produces the key code `games.K_SPACE`, can you add the necessary lines to `keypress` for it to invoke `self.shape.plummet`?

Rotation

The last movement control we have is to rotate our shape so that it can fit into the more awkward corners of the grid. We can tell that this is going to be a little different between different shapes because a square that has been rotated through 90 degree looks exactly the same as it did before.

Put this in the `SquareShape` *class*

```
def rotate(self):
  pass
```

> Change your `keypress` function again so that it rotates the object if the player hits the `RETURN` key, which has the key code `games.K_RETURN`

Exploding Rows

When our shapes stop moving and form a complete row, we are supposed to delete that row and let all of the rest of the grid blocks fall down by one. This is a little tricky to do. Let's write out what we need to do.

For each row, we need to check every box on the row. If all the boxes are `full` then the row contains no empty squares and we can delete it.

This is in TetrisBoard

```
def compress(self):
  for j in range(HEIGHT):
    for i in range(WIDTH):
      if not self.grid[i][j].full:
        break
    else:
      # More here later!
```

lines up with the second `for`!

You might be confused by this. `else` normally goes with an `if` statement, but on this occasion we are putting it with a `for` instead. This is an obscure but cunning feature of Python's `for` loops; if we finish a `for` loop without using `break`, then Python will execute the instructions after the `else` as well.

So how do we delete a line? We copy the line above it down over it, then the line above that, and the line above that, and so on. That means working from the current line (line `j`) back up to the line below the top (line `1`), then clearing out the top line (line `0`) How do we copy the line down? We'll ask each grid square in turn to copy what's in the cell above it.

Add this to compress

```
      # It's later, and this is more
      for k in range(j, 0, -1):
        for i in range(WIDTH):
          self.grid[i][k].copy_from_above()
      for i in range(WIDTH):
        self.grid[i][0].clear_contents()
```

> *You may not be familiar with the odd way we are using `range` here. `range` is actually a quite flexible function; normally we tell it when to end (actually the number after we want it to end) and assume (correctly) that it will start counting from 0 and count up 1 at a time. In fact we can tell `range` where to start from as well, and how much to count by each time through the loop. Here we are telling Python that we want to start our count at `j`, end at 0 and subtract 1 each time through the loop. So if `j` is 5, `k` takes the values 5, 4, 3, 2 and 1 in that order.*

Anyway, we have as usual palmed a bit of our problem off on something else. We've asked our `TetrisBox`es to grab the information from the box above them, so we'd better write the function to do that.

This goes in the `TetrisBox` class

```
def copy_from_above(self):
  # Don't call this if the box above doesn't exist
  # It'll whinge something rotten
  self.set_colour(self.direction[boards.UP].get_colour())
  self.full = self.direction[boards.UP].full
```

That's it. We ask the box in the `direction UP` from us what colour it is using `get_colour`, and set this box to that colour. Similarly we find out how `full` the box `UP` from us is and set our `full` to that. Easy.

> You also need to write `clear_contents` for your `TetrisBox`. This turns out to be easy too; it's pretty much exactly what you did when you initialised the `__init__` function except that you don't need to call `init_gamecell` again.

All we need to do then is to call `compress` when we've worked out that a shape cannot drop any further, just before we ask for a new shape. See if you can figure out where to insert this line:

```
self.board.compress()
```

Scoring

Traditionally in Tetris you score for filling up rows and getting the blocks deleted. Let's do the same; increase our score by 10 every time we delete a row. That's just a matter of adding two lines to our `compress` function; one to increase the score, and one to display it.

Put this in the ''else'' clause

```
        self.score = self.score + 10
        self.score_message.set_text("Score: " + str(self.score))
```

What's all this "str" business, you may ask? The answer is that we need to be clear what we're telling Python to do.
When we write `a+b` *and* `a` *and* `b` *are both numbers, it's pretty obvious that we want Python to work out the sum of the numbers. Similarly, you may remember from Sheet S that when* `a` *and* `b` *are both strings,* `a+b` *tells Python to* catenate *(glue together) the two strings. Unfortunately, if one of them is a string and the other is a number, there are several possible things we might mean for Python to do. We could want Python to hunt through the string for digits, turn them into a number and add that to the other number; or we could want Python to turn this number into a string and bolt them together. Rather than take a wild guess or insist that one meaning is more "obvious" than the other, Python takes the attitude "well don't do that, then." It insists that we either turn the number into a string or the string into a number, because then it really will be obvious what to do.*
`str` *is a built-in Python function that takes a number and turns it into a string.* `str(10)`*, for example, returns the string* `"10"`*. In our case we take the actual game score as a number, turn it into a string and then bolt it onto an explanatory message. Easy, huh?*

Common Code

As you may have noticed by now, our `drop` and `move` functions look very similar to each other. It's often a good idea to put "common code" as computer programmers call it into another function and have `move` and `drop` call it. That way if you ever discover some hideous problem with the way things move, you only need to change one place in your program. We may even have something else come up that needs very similar work done (and as it happens, we will), so in the long run it will save us typing.

So what do we do that's common to both functions? Essentially, both of them come up with a new set of boxes for our shape, check to see if we can use those boxes, and set them up as the new shape. If they can't be used, the two functions do different things — clearly that's something that must be handled separately, as is the choice of boxes. The other two steps, however, checking and updating, those are common.

```
                                          This goes in the Shape class
def try_place(self, new_boxes):
  for box in new_boxes:                   Check each new box
   if box is None or box.full:            Is it usable?
     return 0                             Stop with FALSE if not
  for box in self.boxes:                  Do this bit slightly differently
    box.set_colour(EMPTY_COLOUR)
  self.boxes = new_boxes
  self.place()
  return 1                                Return TRUE if we're happy
```

You'll notice we've changed a few things around here. Since we're requiring our `draw` and `move` to produce a list of new boxes in advance, it makes sense to let Python use that list directly as the new list of boxes. It's more efficient to tell Python to replace the whole list with a new one than to go through each element of the list one at a time.

Now we just need to call our new function correctly.

```
                                          Replace drop and move with these versions
def drop(self):
  new_boxes = []                          Start off with an empty list
  for box in self.boxes:
    new_boxes.append(box.direction[boards.DOWN])
  if not self.try_place(new_boxes):
    for box in self.boxes:
      box.full = 1
    self.board.compress()
    self.board.new_shape()
    return 1
  return 0

def move(self, direction):
  new_boxes = []
  for box in self.boxes:
    new_boxes.append(box.direction[direction])
  self.try_place(new_boxes)
```

Straight Lines

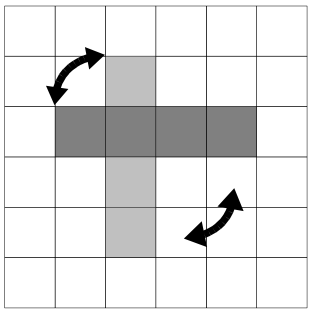
It's time to bite the bullet and try to work out how to deal with a shape other than a square. Let's start with the easiest of the rest, the straight line.

Obviously, this will be a new subclass of `Shape`, so it will need its own `__init__` and `rotate` methods, just like `SquareShape`, and the `__init__` will have to fill in a list called `boxes`. The difficulty that we have to cope with is that the line can be either horizontal or vertical; we should randomly pick one orientation at the start, and flip between them as we rotate.

```
class LineShape(Shape):
  def __init__(self, board):
    self.init_shape(board)                  Do the things all shapes must do
    # Randomly decide whether or not to be vertical
    self.vertical = random.randint(0, 1)
    if self.vertical:
      self.boxes = [ board.grid[WIDTH/2-1][0],
                     board.grid[WIDTH/2-1][1],
                     board.grid[WIDTH/2-1][2],
                     board.grid[WIDTH/2-1][3] ]
    else:
      self.boxes = [ board.grid[WIDTH/2-2][0],
                     board.grid[WIDTH/2-1][0],
                     board.grid[WIDTH/2][0],
                     board.grid[WIDTH/2+1][0] ]
    self.place()

  def rotate(self):
    pass                                    Temporarily!
```

For the purposes of testing, it's easier if we always have `LineShape`s rather than `SquareShape`s falling down the screen. It shouldn't take you long to figure out how to do that, and then you can make sure that the shapes fall, move and plummet as you'd expect.

Now, how are we going to rotate our shape? The simplest way to think of it is that we are going to pick one box of our line to pivot around and change where the boxes go. Just like moving, this will involve checking that our new boxes don't overlap anything already fixed in place before we change things.


Rotating a line

From the diagram, if we were horizontal then we pick on the second box (`self.boxes[1]`) and select the one above and the two below. Similarly if we were vertical, we pick the second box to pivot around and take the one to the left and the two to the right. We can use the `direction` list from our pivot point to find the boxes immediately up, down, left and right from it, and from there we can find the next one down or right from there.

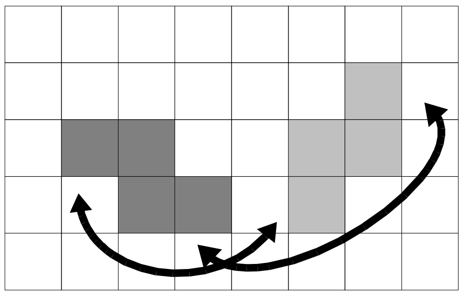
There is a problem with that. If our line is vertical and at the right hand edge of the board, then the box to our immediate right doesn't exist. If we try to take a `direction[boards.RIGHT]` from that, Python will complain that "None has no property called direction" or something like that. So we need to check that we have got a real box before we use its `direction` list.

Replace LineShape's rotate function with this

```
def rotate(self):
  if self.vertical:
    new_boxes = [ self.boxes[1].direction[boards.LEFT],
                  self.boxes[1],
                  self.boxes[1].direction[boards.RIGHT] ]
    if new_boxes[2] is None:
      return                                We can't rotate anyway!
    new_boxes.append(new_boxes[2].direction[boards.RIGHT])
  else:                                     Line this up with ``if self.vertical''
    new_boxes = [ self.boxes[1].direction[boards.UP],
                  self.boxes[1],
                  self.boxes[1].direction[boards.DOWN] ]
    if new_boxes[1] is None:
      return                                Again, the rotated line runs off the edge
    new_boxes.append(new_boxes[2].direction[boards.DOWN])
  if self.try_place(new_boxes):             If this works...
    self.vertical = not self.vertical       ...turn our flag upside-down
```

More Shapes

Well, that was straightforward enough. It turns out that there's another shape (or pair of shapes, actually) that are about as easy. The arrangement of blocks that looks like a Z (see the diagram below) also has only two ways up, horizontally (like a Z) or vertically (like an N). So we can make ourselves a subclass to handle this shape that looks almost the same as `LineShape`, again pivoting around the second box in the list.


Rotating a Z-shape

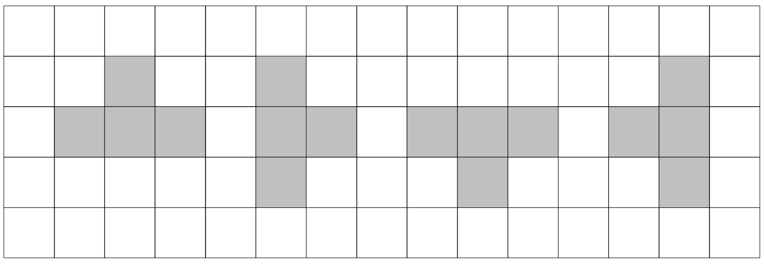
```
class ZShape(Shape):
  def __init__(self, board):
    self.init_shape(board)
    self.vertical = random.randint(0, 1)
    if self.vertical:
      self.boxes = [ board.grid[WIDTH/2][0],
                     board.grid[WIDTH/2][1],
                     board.grid[WIDTH/2-1][1],
                     board.grid[WIDTH/2-1][2] ]
    else:
      self.boxes = [ board.grid[WIDTH/2-2][0],
                     board.grid[WIDTH/2-1][0],
                     board.grid[WIDTH/2-1][1],
                     board.grid[WIDTH/2][1] ]
    self.place()

  def rotate(self):
    if self.vertical:
      new_boxes = [ self.boxes[1].direction[boards.LEFT],
                    self.boxes[1],
                    self.boxes[1].direction[boards.DOWN] ]
      if new_boxes[2] is None:
        return
      new_boxes.append(new_boxes[2].direction[boards.RIGHT])
    else:
      new_boxes = [ self.boxes[1].direction[boards.DOWN],
                    self.boxes[1],
                    self.boxes[1].direction[boards.RIGHT] ]
      if new_boxes[2] is None:
        return
      new_boxes.append(new_boxes[2].direction[boards.UP])
    if self.try_place(new_boxes):
      self.vertical = not self.vertical
```

Again, you'll find it useful to only produce `ZShapes` while you're testing this new class out.

> The reversed Z-shape, the one that looks like a backwards Z or an upside-down N depending on which way round it is, is exactly the same only with some of the positions of boxes changed. You should be able to write the `ReverseZShape` class yourself! Draw the different orientations of the shapes out on paper if you get confused.

Still More Shapes

The remaining shapes aren't quite so simple. So far we've got away with just knowing whether our shapes are vertically or horizontally oriented, but the remaining shapes have four different possible orientations. Let's look at the T-shape, the simplest of the remaining ones.



T-shapes and how to rotate them

We'll describe our different T-shapes by the direction the "stem" of the T is pointing, as shown in the diagram above. This means that instead of having an "am I vertical" flag like the other shapes have, we need a "direction" that can take up to four values. We might as well use the directions that the library supplies (`boards.UP` and the like), since we can use them to help us draw the shapes.

The library has a number of functions which may be useful to us at this point. First up is `random_direction`, which picks a random direction as you might expect. We'll need something like this to pick the starting orientation for our shape, but we'll have to make sure that it only picks orthogonal directions. (You do remember what *orthogonal* means, don't you?)

Once we have selected a direction for the stem of our "T", we need to know which directions are clockwise and anti-clockwise of it so that we can work out which boxes they correspond to. The library helps us here with the functions `turn_90_clockwise` and `turn_90_anticlockwise`, which take a direction and return the direction constants representing the direction 90 degrees clockwise or anticlockwise respectively.

There's one last consideration we need to make, and one arbitrary decision, before we can put the first T-shape on the screen. The consideration is that if the "T" is pointing down then we can have the crossbar of the "T" at the top of the grid, otherwise everything centres on the line below the top. The arbitrary decision is what order we'll put the boxes in our list. It doesn't matter what that order is as long as we're consistent about it. We'll take the middle of the crossbar of the "T" as our first box, since that will turn out to be fairly convenient, then the foot of the stem, the box that's clockwise of it, and finally the box that's anticlockwise of it.

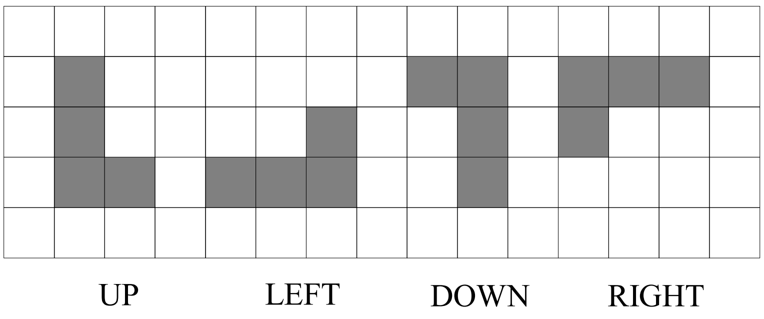
```
class TShape(Shape):
  def __init__(self, board):
    self.init_shape(board)
    self.direction = boards.random_direction(orthogonal_only = 1)
    if self.direction == boards.DOWN:
      self.boxes = [ board.grid[WIDTH/2-1][0] ]
    else:
      self.boxes = [ board.grid[WIDTH/2-1][1] ]
    self.boxes.append(self.boxes[0].direction[self.direction])
    self.boxes.append(self.boxes[0].direction[
      boards.turn_90_clockwise(self.direction) ]
    self.boxes.append(self.boxes[0].direction[
      boards.turn_90_anticlockwise(self.direction) ]
    self.place()
```

Obviously when we rotate our T-shape, we'll be rotating around the middle of the crossbar, again as in the diagram. Handily that's the box we made into the first box in the `boxes` list, so it's easy to find. For the sake of argument, we'll turn it clockwise.

> Can you figure out how to do this? Here's a hint: "turn" and "rotate" mean the same thing. Once you've figured out what your new `direction` is, all you have to do is pretty much what we've done already in the `__init__` function. Just don't change `self.direction` until you *know* that you have succeeded!

One (or Two) Last Wafer-Thin Shapes

The last two shapes we have to deal with are the L-shape and the reversed L-shape. These are a bit like both the T-shape (having four distinct orientations) and the Z-shape (having boxes that we need take directions from which might be off the grid). This means we'll have to write code that behaves a bit like a T-shape and a bit like a Z-shape too.



L-shapes and how to rotate them

First, let's arbitrarily decide to take as our "direction" the longest part of the L, as in the diagram, and pivot around the corner square. For an L, the shorter part of the shape is 90 degrees clockwise of the longer part (for a reversed L, it's obviously anticlockwise instead), so we can use that to avoid making our `__init__` function have to have a special case for every possible starting orientation. We do still have to take into account that our pivot point might have to be in a different place depending on which way up our shape is — obviously we don't want to have any bits of our shape disappearing off the grid right at the start!

For convenience, we'll decide right now how we want everything in our list of boxes to correspond to boxes on the screen. The only one we have to fix is our pivot point because we'll need to be able to find that whenever we rotate the shape, but if we're consistent about the others it will help us to understand what's going on if anything goes wrong.

We might as well make the pivot the first item in the list. It doesn't have to be — we could make it the second, like we did for our `LineShape` — but it's as good a place as any for it. We will then put the box for the short limb of the L in the second element of the list, the near box of the long limb third, and the far box last.

```
class LShape(Shape):
  def __init__(self, board):
    self.init_shape(board)
    self.direction = board.random_direction(orthogonal_only = 1)
    if self.direction == boards.UP:
      self.boxes = [ board.grid[WIDTH/2-1][2] ]
    else if self.direction == boards.LEFT:
      self.boxes = [ board.grid[WIDTH/2-1][1] ]
    else:
      self.boxes = [ board.grid[WIDTH/2-1][0] ]
    self.boxes.append(self.boxes[0].direction[
      boards.turn_90_clockwise(self.direction)])
    self.boxes.append(self.boxes[0].direction[self.direction])
    self.boxes.append(self.boxes[2].direction[self.direction])
    self.place()

  def rotate(self):
    direction = boards.turn_90_clockwise(self.direction)
    new_boxes = [ self.boxes[0],
                  self.boxes[0].direction[
                    boards.turn_90_clockwise(direction)],
                  self.boxes[0].direction[direction] ]
    if new_boxes[2] is None:
      return                                 We can't turn that way
    new_boxes.append(new_boxes[2].direction[direction])
    if self.try_place(new_boxes):
      self.direction = direction
```

> Figure out what changes you need to make in order to write the `ReverseLShape` class. If you start getting confused (as I did), try drawing out the possible orientations of the shapes.

Picking a Shape

So far we have been testing our Tetris program one shape at a time, changing our `new_shape` function to test out what we've just written. To play the game properly, we need to select a shape at random so that we don't know what's going to fall down at us next.

The Python standard library provides us with the function `random.choice` which allows us to pick an object out of a list, which sounds pretty much like what we want to do. All we need to do then is to make a list of shapes and select one. That sounds like `new_shape` should look something like this:

```
self.shape = random.choice([ SquareShape(self),
                             LineShape(self),
                             ZShape(self),
                             ReverseZShape(self),
                             TShape(self),
                             LShape(self),
                             ReverseLShape(self) ])
```

You just typed that in and it all went horribly wrong, didn't it? That's because when we write `SquareShape(self)` and the like, we are actually creating the square shape there and then and immediately displaying it in the grid. What our line of code has done is to create and display one of each shape and throw all bar one of them away. Unfortunately when we throw shapes away we don't take them off the screen (which is just as well, or we'd lose all the shapes that hit the bottom), so the result is a big mess.

We could do something more cunning such as separating out the creation of the shape from drawing it first time (a simple matter of not going `self.place()` in the `__init__` routines), but there is a better solution. Creating, initialising and displaying an example (or *instance* as programmers call them) of a shape is quite a lot of work, and it would be better if we could pick which *class* of shape we want to have and only create the one. This may sound a little odd, and it's very difficult to do in some programming languages. Python however treats classes in the same way that it treats any other objects. We can put a class in a variable, or a list, or whatever, and use it indirectly like that to make a new instance.

So what we actually want is this:

```
        self.shape = random.choice([ SquareShape,
                                     LineShape,
                                     ZShape,
                                     ReverseZShape,
                                     TShape,
                                     LShape,
                                     ReverseLShape ])(self)
```

This makes a list of the classes for shapes, picks one at random, and then creates a shape from it. And that's all we need to do!

Bells and Whistles

You've now written everything you need to write to make a game of Tetris. There are still plenty of things you could do, if you have time.

- You might think that some of shape rotations look a bit peculiar. L-shapes (and reverse L-shapes, of course) in particular don't necessarily behave the way you'd expect from other versions of Tetris. Should we be rotating anticlockwise rather than clockwise? Or should be using a different box as our pivot? Or both? Experiment a bit and see what you think. *Note:* this will involve quite a bit of messing about with the classes. It's probably a good idea for you to make a copy of the `LShape` and `ReverseLShape` classes and comment out the originals (put a '`#`' character at the start of each line) so that you can recover a version that works easily. Don't forget that you'll need to alter the `__init__` functions as much as the `rotate` functions!
- Lines and Z-shapes can look a bit odd as they rotate too. Since our pivot box can't be bang in the middle (there being only four boxes), we're actually changing the direction we rotate in each time we rotate. In other words while we're pretending that our lines and Z-shapes have only two orientations, in reality they have four because the pivot box could be the other side of the middle. See if you can change the program to behave that way, and decide for yourself if it makes the rotations of those shapes look more natural.
- At the moment, once the game of Tetris is over that's the end of things. You have to start the program all over again in order to play a new game. You could allow the player to start a new game so that he doesn't have to do all that tedious business. There is a worksheet that can help you with that — ask a leader to give you a copy.
- Some games of Tetris mark certain blocks as special for various different purposes. Perhaps the blocks score extra points when they compress off the screen, or perhaps they don't disappear when the rest of the line does, or perhaps they cause something else bizarre and unlikely to happen. See what you can think up and try it out. You'll need to randomly decide when to make a box in a shape have one of these special characteristics (make it rare!), and work out some way of showing this on the screen. Some of the other games worksheets may give you ideas for how to do this.
- Most games of Tetris get faster as time goes on. At the moment our game goes at a constant speed, moving the shape every `interval` ticks. Try making `interval` smaller after a particular (large) number of moves of the shapes, which will have the effect of speeding things up. Be careful not to make `interval` too small — if it ever gets less than 1, strange things will start happening!
- At the moment our lines compress in a quite boring manner. Most versions of Tetris cause disappearing blocks to explode prettily for a brief moment before continuing the game. See if you can figure out something similar. *Note:* this is hard. You will need to ask a leader for help!