Game: Snake

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, developed by Rhodri James, Daniel Watkins and Philip de Beger. When you've finished it, you'll have written a version of a classic computer game called Snake.

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)
- Dictionaries (from Sheet D)
- Class 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 Snake?

The game of Snake involves steering a snake around the screen, trying to eat up food as it appears without accidentally eating the snake itself. As the game progresses this gets more difficult, since every time the snake eats some food it gets longer. The longer your snake gets before it dies, the higher your score.

More advanced versions of the could involve avoiding poison, food going away if it isn't eaten for too long, wormholes or anything else you might imagine!

What do we need?

From the quick description above, obviously only a few things appear on the screen in the basic game.

- the snake itself, and

- the food it eats.

However, if you think about it a bit you'll see that there is a little more to it than that. We can make our life a lot easier by noticing that our snake actually travels along a grid, just like the robots did in Sheet 5. The LiveWires games library can handle grids for us automatically, which will make things a lot easier. As you will see, we can make things easier still with a little lateral thinking.

While the games library will handle grids for us, there are a few things that we would like our boxes to do beyond simply existing. This means that we are going to have to make a sub-class of what the library provides to get exactly what we want.

In The Beginning

Before we get on to the exciting bits, 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 boards
from livewires import colour
from random import randint
```

First, we tell Python we want to access the things in the LiveWires `games` and `boards` modules, and also the colour names supplied by the `colour` module. We will also want to use the function `randint` from the `random` module; this is in fact what we disguised as `random_between` in the beginners' library, so you can guess what we're going to do with that.

```
BOX_SIZE = 40
MARGIN = 60

SNAKE_COLOUR = colour.black
EMPTY_COLOUR = colour.light_grey
```

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. As it happens, we will do exactly that somewhat later, when we are drawing a blob of food on the screen. Obviously the blob will need to be smaller than a box, otherwise it will look messier than we intended. 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.

Little Boxes...

Now we get on to our boxes on the screen.

```
class SnakeBox(boards.GameCell):
  def __init__(self, board, i, j):
    self.init_gamecell(board, i, j)
    self.is_snake = 0
    self.is_food = 0
    self.food = None
```

What this tells us is that `SnakeBox` 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 this square on the board contains a snake or a blob of food. You will see later that we could find these things out in different ways, but it's convenient to have these *flags* (as they are called) as well.

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, while variables have no capital letters but do have underlines. 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 notice what is going on.

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

Making Boards

We need a class that represents an entire board full of boxes, and what to do with them. The library provides the `SingleBoard` class to do this, but it doesn't do all the things that we will need. The way to add more complicated bits to a class is to subclass it, just like we did above, so let's do that. Type in the following code for starters:

```
class SnakeBoard(boards.SingleBoard):
  def __init__(self, n_cols, n_rows, snake_size, interval):
    self.init_singleboard((MARGIN, MARGIN), n_cols, n_rows, BOX_SIZE)
    self.game_over = 0

    self.create_directions(orthogonal_only = 1, wrap = 1)
```

This just declares `SnakeBoard` to be a kind of `SingleBoard`, records some things that we will need to keep track of, and initialises the `SingleBoard` parts of our new class. If this looks familiar, don't be 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 are next to one another. It builds up a list called `direction` that we'll use later on to move our snake left, right, up and down. Since we want our snake to be able to run off one side of the board and back onto the opposite side, we tell the function that we want it to *wrap* around like that.

> *We say that two boxes are next to each other* orthogonally *if touch each other on the sides, not on the diagonals. 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.*

Enter the Dragon

...or the snake, at any rate. We could try to keep track of the snake just as boxes on the board, but we would find things getting complicated rather quickly. It's easier to have something separate that tells us which boxes the snake is in (rather than having to search through the board), and which direction it's going in. Notice here that since the snake will cover more than one box, we will need to keep a *list* of the boxes it is in. It will also need to move somehow, and change direction, and

perhaps do some other things. This is beginning to sound like a snake is a class all of its own!

```
class Snake:
  def __init__(self, board, box, direction):
    self.position = []
    self.position.append(box)
    self.direction = direction
    self.board = board
```

You might be surprised at the last line. Even if we secretly know that some time in the future we are going to have to make the snake do something to the game board (eat a piece of food, for example), why do we want to keep another pointer to the board inside the `Snake` *class? Why not just use the* global variable *that we will have to create for the main program loop?*
There is a general philosophy in writing computer programs that we should avoid global variables, ones that can be used from all over the program, wherever we can. There's nothing inherently wrong with them, but if we change our program to generalise something, global variables can make life more difficult.
Suppose for example that we had multiple snakes wandering around several different game boards. Not very likely, I'll grant you, but a different game might well need something like that. In such a case, each snake would need to know which of the several possible boards it was on. The way that we've set things up here, this would happen automatically, but if we had referred to a single global `board` *instead then we'd have a lot to do to change things round.*

Now that we've defined what a `Snake` is, we had better create one when we start the game up. That means adding some lines to our `SnakeBoard` class `__init__` function, which should look like this:

Line this up with the rest of __init__

```
    i = n_cols/2
    j = n_rows/2
    dir = boards.random_direction(orthogonal_only = 1)
    self.snake = Snake(self, self.grid[i][j], dir)
```

Passing the Buck

So far, our snake knows where it is, but it hasn't actually drawn anything on the screen. We could have the `Snake` class do this directly, finding out where its starting box is and colouring it in. This is messy though; the snake doesn't otherwise need to know where on the screen it is, just which box it's in. Instead, we will get the box to do the hard work for us.

Add the following line to the `Snake` class `__init__` function:

Remember to line this up!

```
    box.snakify()
```

The box then needs to flag up that it is part of a snake, and colour itself in appropriately. The following *method* (a posh programmer's name for a function that is part of a class) will do that in our `SnakeBox` class.

Line this up with the other defs

```
  def snakify(self):
    self.is_snake = 1
    self.set_colour(SNAKE_COLOUR)
```

> At some point we will need to tell a box that it isn't part of the snake any more. See if you can write a function to do that, and call it `unsnakify`. Hint: what other colours did we define earlier?

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 needs us to provide in the `SnakeBoard` class:

Align this with the ''def __init__''

```
  def new_gamecell(self, i, j):
    return SnakeBox(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 do all the work for us. In our case, it takes all of two lines:

Don't put any spaces at the start of this line!

```
snake = SnakeBoard(15, 15, 3, 10)
snake.mainloop()
```

Try running the program now and seeing what happens.

Running Run

What we have so far is not exactly wildly exciting. Our snake looks like a blob sitting still in the middle of the screen, and no amount of prodding with sharpened sticks will make it move. To do that, we need to add some code to do something after a particular amount of time.

Fortunately, the library can help us here. It calls the function `tick` in the `SnakeBoard` class 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 moving our snake.

Hang on a sec, though. We haven't written a `tick` function yet, so why hasn't the library complained at us and blown up? The answer is that the class that we made `SnakeBoard` from has a `tick` function, and the library is calling that since we haven't given it anything better 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? Well, it needs to make our snake move.

Indent this to line up with the other defs

```
  def tick(self):
    self.snake.move()
```

How does a snake move? If you break it down into its separate parts, there are two things to do. First we have to find the next box on from the head of the snake in the direction that it's going and add it to the snake. Once we have done that, we can drop the tail of the snake off and turn it back into ordinary board.

As you will remember from Sheet A, Python has quite a lot of functions for manipulating lists, which is mostly what our snake is. From the description, we are particularly interested in using `insert` to put a new box on the front (item 0) of the snake, and `pop` to remove the tail. In addition, the library will handle the directions stuff for us. It has built up a list called `direction` for us, and defines a bunch of useful constants, so the box to the left of where we currently are is held in `box.direction[boards.LEFT]`, for example.

Putting this all together, we get can write our `move` function for `Snake` like this:

Align this with the other defs

```
  def move(self):
    first = self.position[0].direction[self.direction]
    first.snakify()
    self.position.insert(0, first)

    last = self.position.pop()
    last.unsnakify()
```

> Since we can use return values from functions straight away, we can write the last two lines above a bit more compactly. Instead of putting the result from `pop` into a variable, we can use it directly like this:
> `self.position.pop().unsnakify()`
> It isn't worth getting rid of `first` in the same way because we use it twice.

Whoa!

Now our snake moves, but it hurtles across the screen much too quickly and it still doesn't look much like a snake!

We can fix the first problem by not calling the `move` function quite as often. If you remember, we have a parameter to the `SnakeBoard` class `__init__` called `interval`. We didn't tell you at the time, but we can use it to define how many ticks it takes before we move the snake. First, we will need to save it away and create another element of the class called `tick_count` which we want to start at zero. If you *really* can't work out how to do that, ask a leader. Prepare for sarcasm, though!

Once we have done that, our `tick` function needs to get more complicated.

Remember to align the defs properly

```
def tick(self):
  if self.game_over:
    return
  self.tick_count = self.tick_count + 1
  if self.tick_count == self.interval:
    self.tick_count = 0
    self.snake.move()
```

That should slow it down! You will have to adjust the interval parameter to taste, according to how fast your machine runs. A value of 20 is perhaps a little slow for some tastes, but is useful for testing.

The other problem we have is that our snake is still just a blob. To be a proper snake, it ought to be long and thin. We can make it a bit longer by adding more boxes to our snake, which will just have to do. Add the following to our `SnakeBoard __init__` function:

Line up with the rest of __init__

```
  snake_box = self.grid[i][j]
  dir = boards.turn_180(dir)
  for n in range(1, snake_size):
    snake_box = snake_box.direction[dir]
    self.snake.extend(snake_box)
```

This adds boxes to the snake backwards from the head, until we have as many as the `snake_size` parameter to our function requested. Of course, we now have to get our `Snake` to extend itself. All that means is adding the box to the end of the snake's list of boxes, and snakifying it.

> Try writing this one yourself. If you get stuck, look through Sheet A to see what functions can do what with lists.

Ouroboros

In mythology, the ouroboros was a snake that ate its own tail. Unfortunately for our snake, it's supposed to get indigestion if it does that. So how do we find out if our snake is about to eat itself?

If you recall, one of the things that our `SnakeBox` does when it is told to "snakify" itself is to set its flag `is_snake`. Since we know the box that we are about to go into, we can check to see if its flag is set when we try to move. If it is, then the game is over.

Unfortunately our `Snake` doesn't know what to do if the game is over, so it will have to tell its caller, the main `SnakeBoard`. This means that `move` now has to return a value in all circumstances — 1 if all is well, and 0 if the game is over. The results should look something like this:

Align with the other defs

```
def move(self):
  first = self.position[0].direction[self.direction]
  self.position.pop().unsnakify()

  if first.is_snake:
    return 0
  first.snakify()
  self.position.insert(0, first)
  return 1
```

Notice that we've moved up the line that pops the tail off the snake and tidied it up. This is to get around a potential problem that may bite us (if you'll pardon the pun) later on.
Imagine for a moment that the head of the snake is about to move into the box where its tail currently is. If we don't remove the tail first, then when we look at `first` *it will appear to be part of the snake, even though the tail would be out of the way by the time we actually finished the move. We would therefore mistakenly conclude that we had bitten ourselves, and end the game.*

Having made these changes, we need to change the behaviour of the `tick` function so that it pays attention when we tell it the game is over. We need to replace the call to `self.snake.move` with the following:

If the move wasn't OK...

```
if not self.snake.move():
```

...end the game

```
  self.game_over = 1
```

In case you didn't notice, earlier on we checked `game_over` before we did anything when a tick happened. Setting it to 1 like this will mean that we don't do anything when a tick happens, so the snake will stop, er, dead. Sorry about that.

A Matter of Control

Now we have a snake happily charging around the screen, ready to drop dead if it ever eats itself. Unfortunately for the herpetophobes out there, the snake is always moving in the same direction, so it's not going to kill itself. It's also not a very exciting game. What we need is some way of making the snake change direction.

The obvious way to do that is to use the cursor keys. We will stick to just the straight orthogonal directions: up, down, left and right. The library doesn't stop us from using other keys, or going off on diagonals, but we choose not to here. Apart from anything else, it can be very tricky working out if a snake has just crossed itself diagonally!

The `games` library calls the function `keypress` whenever it detects a key being pressed. Just like `tick`, the library provides a default version of `keypress` that doesn't do anything interesting. In this case, the `boards` library also provides a default version, but that one doesn't do what we want. We will need to write our own.

When the library calls `keypress`, it provides it with a *key code* which represents the key which has been pressed. We will need to turn these into a direction which the snake understands (which just happen to be the same as some constants that the library provides — how handy!). The easiest way to do this is to use a *dictionary*, which we will first need to set up. Add the following rather cryptic lines to the `SnakeBoard __init__` routine:

Remember to line it up!

```
self.key_movements = {
  boards.K_UP:    boards.UP,
  boards.K_DOWN:  boards.DOWN,
  boards.K_LEFT:  boards.LEFT,
  boards.K_RIGHT: boards.RIGHT
}
```

This makes a dictionary in which looking up the "up" cursor key (which the library calls `boards.K_UP`) will give you the "direction" `boards.UP`, and so on. Be careful when you type this in that you don't type too many commas, or Python will complain!

The `keypress` function in the `SnakeBoard` should then be pretty straightforward:

Remember to indent this properly

```
def keypress(self, key):
  if self.game_over:
    return
  direction = self.key_movements[key]
  self.snake.set_direction(direction)
```

All we then need is the function `set_direction` in the `Snake` class, which just needs to record the direction so that `move` will use it later. You should be able to write that yourself. Go on, then!

If At First You Don't Succeed...

This all appears to work fine. The snake ambles around the screen, changing direction when you press the cursor keys and dying if you reverse it onto itself. There is a hidden flaw, however; try pressing any key other than one of the cursor keys.

Bang! The program explodes, telling you that it has a `KeyError`. This means that it has tried to look the key code up in our dictionary, `key_movements`, but hasn't found it. This isn't very surprising really; we only put those four cursor keys into the dictionary.

There are various ways to fix this problem. We could put entries into the dictionary for all the other possible key codes, but this would be very tedious and make a very large, wasteful dictionary. We could give up on using the dictionary and write a chain of `if` statements instead; that isn't wasteful, but it's still pretty tedious. Fortunately, Python gives us a way of stopping half-expected errors from blowing up the program.

Line the "try" up where the "direction" used to be

```
try:
  direction = self.key_movements[key]
except:
  return
```

The `try` command effectively means "try to do the next block of program, but if it fails, give up and do the commands following `except`." In big, complicated programs, this is often used to recover from errors that you can't predict, such as a network being unplugged. We are using it to let us write shorter, neater programs, which is a perfectly good excuse.

The instructions above effectively mean "look up the `key` in `key_movements`. If you find it, put the result in `direction`. If you don't, leave the function since there is nothing more for us to do."

Food, Glorious Food

Now we have control of the snake, we need to think about feeding it. Every time the snake eats (runs over) a blob of food, it gets longer by one.

We will represent a food blob with a circle on the screen, and with a new class in the program. We don't have to make food into a class, but it might prove useful later if you decide to do something more fancy.

When we create a new food blob, it will be at a random place on the board. This suggests that "creating a new food blob", or at least, "working out where to create a new food blob" is a job for the `SnakeBoard` itself. The function itself looks a little peculiar; you may wish to reread Sheet L on loops before typing this in.

Remember to line everything up carefully

```
def new_food(self):
  while 1:
    i = randint(0, self._n_cols-1)
    j = randint(0, self._n_rows-1)
    if not self.grid[i][j].is_snake:
      break
  self.grid[i][j].foodify()
```

What we want to do is to pick a random box that doesn't already have a bit of the snake in it. To do that, we have to go round a loop, picking a box, until we find one that isn't snake. Since Python doesn't have a command for "do stuff until ready", we have to improvise with a `while` loop, using `break` to stop when we've found a usable box.

So, we have selected a box and told it to add some food. What does the box do? Not a lot, really.

Add this to SnakeBox

```
def foodify(self):
  self.is_food = 1
  self.food = Food(self.screen, self.screen_x, self.screen_y)
```

All the hard work is palmed off on the new `Food` class, which we haven't written yet. There's not even much of that to write — using a class for it is a bit of overkill at the moment!

```
class Food(games.Circle):
  def __init__(self, screen, x, y):
    self.init_circle(screen, x+BOX_SIZE/2, y+BOX_SIZE/2,
                     BOX_SIZE/2-2, colour.green)
```

Finally, we have to call `new_food` from the end of the `SnakeBoard` class `__init__` routine. That will put a blob of food on the screen, and we're ready to go!

Eating Your Dinner

We now have food on the board, but the snake doesn't know how to eat it yet. What we want to happen when the snake eats the food is for the food to disappear (and some new food to appear elsewhere), and for the snake to get longer (it's growing big and healthy because it's eating its greens). The easiest way to grow the snake is to not drop the tail off when we move it — that will have the effect of making the snake one box longer.

Change the `move` method so that instead of just popping off the end of the snake, it does this:

If our new box is edible...

```
  if first.is_food:
```

...tell the food to go away

```
    first.eat()
  else:
    self.position.pop().unsnakify()
```

Eating the food is just as straightforward. All the `SnakeBox` class has to do is to clear its flag, `destroy` the food blob (and set it back to None), and ask the `board` for a new blob.

> This is the last piece of the program, so see if you can write it yourself. The `eat` function is pretty short, really, but you can ask a leader if you get stuck.

Congratulations, you now have a fully working game of Snake!

Bells and Whistles

There are several things you could do to make your game even better. Here are a few ideas that you might want to work out how to do, or you can come up with your own improvements.

- The game just stops at the moment when the snake eats itself. You could show a "Game Over" message when the snake dies, so that the player knows what's going on. You will need to use the `Text` class from the library to put the message on the screen; see if you can work out how.
- If you've got a "Game Over" message, you could add a score to it as well. Give the player a score of one for each box in the snake. All the boxes in the snake are held in its `position` list; see if you can work out how long it is using Sheet A to help.
- You could have more than one food blob on the screen.
- As well as food, you could have poison pellets that kill the snake if it eats them.
- Food could disappear if you leave it for too long. You might find the `Timer` class in the library handy for this, with a little thought.
- Alternatively, food could turn poisonous if it is left alone for too long. If you do this, you probably want the poison to disappear after a while.
- The game could get faster (i.e. `interval` could get smaller) after a while, or after the snake eats a certain amount of food. If you do this, beware of one catch; we currently check for our `tick_count` being equal to `interval`, but if you're unlucky they may pass each other, `interval` being decreased as `tick_count` increases.
- Other specials could appear on the screen, such as wormholes (sorry about the joke, but they're really easy to do), turntables or bouncy walls that reverse the movement of the snake.