snakes, apples and python, oh my!

Carlo Zancanaro, 2013

Contents

Snake bot tutorial

The game

Snake[1] is a video game from around 1970, but it's more likely that you've played one of its more recent forms (either on an old Nokia phone, or in some other form). In this game you play the role of a snake moving around a board trying to eat as much food as possible without hitting yourself. As you eat food you grow longer and longer, making it harder and harder for you to avoid your tail.

In this workshop we won't be playing snake ourselves, instead we'll be writing a program to play the game snake for us! Rather than relying on our quick-thinking and dexterity we'll have to rely on our brains to build programs which are smart enough to play the game well.

Getting started

My first bot

To get started, we'll try making a very simple bot. To begin with let's just move in one direction: up.

```
def bot(board, position):
    return 'U'
```

Let's just ignore the board and position for a moment (we'll come to those later). All this function does is return the string `'U'`, for up. Each of the directions is represented by a different letter, all of which should be fairly obvious:

- to go up, return `'U'`
- to go down, return `'D'`
- to go left, return `'L'`
- to go right, return `'R'`

If you decide that you don't like up, feel free to swap out `'U'` for any of the other directions. My personal favourite is left/`'L'`.

Running my first bot

In order to run this bot you'll first need Peter Ward's snakegame[2]. Hopefully you were provided an easy script to run the snake game as a part of this workshop.

If you have just downloaded the code linked above, then you can run your bot most easily by adding the following code to the bottom of your bot file:

```
if __name__ == '__main__':
    from snakegame.engine import Engine
    from snakegame.viewers.pyglet import Viewer
    engine = Engine(10, 10, 25)
    engine.add_bot(bot)
    viewer = Viewer(engine)
    viewer.run()
```

[1] http://en.wikipedia.org/wiki/Snake_(video_game)
[2] http://hg.flowblok.id.au/snakegame

If your file is named `simple.py`, then you can now run it by running the following command:

```
$ python simple.py
```

You should see a screen pop up with your snake moving up continuously. When you've had enough of that hit escape and continue on to the next section!

Let's actually do something

In the last section we made a bot that moved in one direction forever. Now, this is pretty cool, but it's probably not the best strategy if you want to eat all the food on the board. What we really need is some variation in our movement.

An easy way to change the way we move is to just move around randomly! Have a go at writing your own bot to randomly choose between the four directions! (Hint: it might be worth looking at `random.choice()`[3])

... Something smarter, perhaps?

So, how did you go? Did you manage it? How does your bot go?

If it's anything like mine, it doesn't take very long at all for your bot to die. Why is that? Well, it doesn't know any better! As soon as our snake grows itself a tail it becomes its own worst enemy, because it might decide to move straight into it!

Our snake before eating some food:

```
....
.A*.
....
```

Our snake after eating some food (if it decides to move left it will run into it's tail and be in quite a bit of trouble):

```
....
.aA.
....
```

A bit of forethought

What we really need to do is make sure we don't run into our tail right away! In order to do this we'll need to take a look at the board to check if places are free. In order to do this we'll have to examine the two arguments to our bot function:

```
def bot(board, position):
```

The first argument, `board`, gives us a view of the board as it looks now. The second argument, `position` tells us where our head is at the moment. Using these two pieces of information we can determine whether we can move in each direction.

You can take a look at what these arguments are with a simple bot which just prints the arguments:

[3] http://docs.python.org/3.3/library/random.html#random.choice

```
def bot(board, position):
    print("board:", board)
    print("position:",position)
```

When you run the game with this bot you should get some sort of output like this (with a board size of 4x4, and 5 pieces of food):

```
board: [['.', '.', 'A', '.'], ['.', '.', '*', '.'], ['*'. '.', '.', '*'], ['.', '*', '*', '.']]
position: (2, 0)
```

(After the output there should be a few lines starting with `Exception in bot`, but you can safely ignore them - they're just because your bot isn't returning a valid direction).

Here you can see what the arguments look like: `board` is a list containing lists (each of which contains strings), and `position` is a pair of numbers indicating your position inside the `board`: the first number (usually called x) is how far in from the left you are, the second number (usually called y) is how far down from the top you are.

In order to get the character for our head we can do something like this:

```
x,y = position   # break the coordinates into two parts
head = board[y][x]
```

But now, by using different values for `position` we can get the character in different positions on the board!

So, we could try shifting our x by adding 1 to it to see what is in the square to our right:

```
x,y = position
value = board[y][x+1]
```

We can add and subtract from x and y to shift what square on the board we're inspecting at each point. So now we have enough to make a simple bot that will move right as long as it is able, and if it's blocked by something will move in another direction:

```
def bot(board, position):
    x,y = position
    value = board[y][x+1]
    if not value.isalpha():
        return 'R'
    return 'D'
```

Try running this bot, and see what happens. Did it run? Did it crash?

What you should have seen is that your bot starts, goes to the right, then gets to the edge of the board it crashes with an `IndexError` message. What this means is that we've tried to look at a square that's off the board. Our `x+1` has gone beyond the width of the board!

There are a few ways to solve this, but the easiest (and the best) is to use python's *modulo* operator (represented by a %).

Strengthen what remains

The *modulo*, or %, operator in python does one very simple operation: it returns the remainder of the first number after dividing by the second.

Let's take a look at a few examples:

```
10 % 3 = 1
7 % 4 = 3
100 % 10 = 0
```

This works quite similarly to how the minutes on clocks work. Try this: if it's been 25 minutes since 5:50, how many minutes past the hour is it?

Now, most of us can probably answer that without having to write down the calculation, but let's do it anyway: 5:50 + 25 = 5:75. But there are only 60 minutes in an hour, so it must be 6:15, so it's fifteen minutes past the hour.

That one was easy, what if I asked you to work out how many minutes past the hour it would be in 257 minutes (from 5:50)? Well, that's not so trivial. We could try to subtract each hour individually, over and over:

```
257 - 60 = 197
197 - 60 = 137
137 - 60 = 77
77 - 60 = 17
```

Then we could do 50 + 17 = 67, which is 7 minutes past the hour.

Instead of this process, though, we could use python's modulo operator instead. Then our work becomes (50 + 257) % 60. Much easier!

Back to snake

So, what does this have to do with snake? I'm glad you asked! Our board acts a bit like the minutes hand on a clock. When you go off one edge (past the 59th minute, in clock terms) you just start right back on the other edge (the beginning of the next hour). This means that we can use modular arithmetic to ensure we don't try to look outside the board's limits.

Give this new version of our earlier "right, with a down if blocked" bot:

```
def bot(board, position):
    height = len(board)
    width = len(board[0])
    x,y = position
    value = board[y][(x+1) % width]
    if not value.isalpha():
        return 'R'
    return 'D'
```

If you run this bot you should find that it will run off the right hand side of the board, but then continue back on from the left hand side. Success!

Now, have a go at improving your earlier "random" bot! See if you can make it so it doesn't run into its own tail (or, at least, not immediately). You should see it last much longer than it used to, but eventually it will succumb to its own tail.

Away from randomness

So far we've only been making decisions based on random chance. Slightly educated random chance, perhaps, but ultimately there's not very much "smarts" to our bot.

Instead of just randomly choosing a direction to go in, let's try to pick the direction which will take us on a path towards the closest piece of food available. In order to do this we'll try to find the full path to the closest piece of food, then we'll just move in the direction of the first step.

Finding a path

One of the easiest ways to find a path is by using an algorithm known as the Breadth First Search, or BFS.

In the BFS algorithm we maintain a list of squares for us to check, and we check them one by one. As we check them we do one of four things, depending on the square we're looking at:

1. If we've already looked at this square: move on to the next square in our list.
2. If the square we're looking at is food: we're done!
3. If the square we're looking at is empty: add the squares we can read from here to the end of our "to check" list.
4. Otherwise: move on to the next square in our list.

If we end up in a situation with an empty "to check" list then we know that we cannot reach any food at the moment. In this case we'll need some "fall back" option, which could be as simple as "just move down".

By following these steps our algorithm will look at each square on the grid, starting with the closest ones and moving outwards, until it either finds the food or runs out of squares to look at. When writing the code for this algorithm remember that you need to keep track of which direction you first moved in to get to each square, otherwise you won't know which direction you have to move in to get there!

Writing our bot

If you've understood everything that's happened up until here then you're doing well! Have a go at writing a bot that will always go for the closest piece of food.

Hopefully if you've made it to here you've got a working bot (or even a few) to play the game snake. That's great! How can you improve on your bot? Do you have any ideas for how your snake could be more effective? If so, now's the time to give them a go! Good luck!