Lab 4: A rudimentary game

This week we will create a rough draft of a working game and (hopefully) by the end of this session we will have our first (very basic) playable version.

We will do this by creating a `class` which will manage a grid of tile data for us. The core of the class looks like this:

```
class Game:
    def __init__(self):
        self.grid = [[None, None, None, None],
                     [None, None, None, None],
                     [None, None, None, None],
                     [None, None, None, None]]
```

Our class has an attribute `self.grid` which we will use to store the tile data.

Visualising the grid

Now we would like to see the grid data. We will create a draft game which is playable in the terminal. So the most sensible thing to do is to create a `__str__` method which allows us to simply print the game in order to see the grid state. Later on, we can add things like the score to this as well.

The `__str__` method must return a string. A simple approach is to just pass `self.grid` through the built in `str` function.

```
class Game:
    def __init__(self):
        self.grid = [[None, None, None, None],
                     [None, None, None, None],
                     [None, None, None, None],
                     [None, None, None, None]]

    def __str__(self):
        return str(self.grid)

if __name__ == "__main__":
    g = Game()
    print(g)
```

Running the above code will print out the grid data, but it comes out on one line.

```
$ python3 game.py
[[None, None, None, None], [None, None, None, None], [None, None, None, None],
[None, None, None, None]]
```

That's no good. We need to see the columns and rows. This means we need to insert newline characters (`\n`) in between each row of data so they each take up a line of their own.

So we can upgrade our `__str__` method as follows.

```
    def __str__(self):
        return "\n".join([str(row) for row in self.grid])
```

Here we are performing a list comprehension on the grid data, converting each row into a string. Then we take the resultant list and pass it into the `string.join` method to join the rows together using a newline character to place each row on a new line.

The result is much closer to what we need.

```
$ python3 game.py
[None, None, None, None]
[None, None, None, None]
[None, None, None, None]
[None, None, None, None]
```

Adding an extra line break to the beginning creates a space above the grid which helps with readability.

```
    def __str__(self):
        grid = "\n".join([str(row) for row in self.grid])
        return f"\n{grid}"
```

> We will extend this method a bit more next week.

Initialising the game

We have initialised the grid with all `None` values. The next thing our game needs is two random tiles to be set to the value `2`.

To do this, we first need to pick a random tile. We need to generate two random numbers from 0 to 3, one for the row and one for the column.

We will import the `randint` function from the `random` module to do this. We can use it like this.

```
from random import randint

class Game:
    def __init__(self):
        self.grid = [[None, None, None, None],
                     [None, None, None, None],
                     [None, None, None, None],
                     [None, None, None, None]]
        self.set_random_tile(2)
        self.set_random_tile(2)

    def __str__(self):
        grid = "\n".join([str(row) for row in self.grid])
        return f"\n{grid}"

    def set_random_tile(self, value):
        row = randint(0, 3)
        col = randint(0, 3)
        self.grid[row][col] = value

if __name__ == "__main__":
    g = Game()
    print(g)
```

The `set_random_tile` method will write the provided value into a random location on the grid. We call the new method twice in our `__init__` method to set two random tiles to `2`.

```
$ python3 game.py

[None, None, None, 2]
[None, None, None, None]
[None, None, None, None]
[None, None, None, 2]
```

However, there is a problem with this. In the rare case (one in sixteen, so not that rare) when the method randomly selects the same tile twice, we end up with only one `2` in the grid and the game will not be playable.

> Try running the code a few times, unless you are unlucky you should see the problem at least once within ~20 attempts.

To solve this, we need to upgrade the method with a bit more logic. We will check that the selected random tile contains `None` before we set the value.

```
from random import randint

class Game:
    def __init__(self):
        self.grid = [[None, None, None, None],
                     [None, None, None, None],
                     [None, None, None, None],
                     [None, None, None, None]]
        self.set_random_empty_tile(2)
        self.set_random_empty_tile(2)

    def __str__(self):
        grid = "\n".join([str(row) for row in self.grid])
        return f"\n{grid}"

    def set_random_empty_tile(self, value):
        while(True):
            row = randint(0, 3)
            col = randint(0, 3)
            if not self.grid[row][col]:
                break
        self.grid[row][col] = value


if __name__ == "__main__":
    g = Game()
    print(g)
```

The new method now enters an infinite loop and only breaks out of the loop if the randomly selected tile contains `None`. Otherwise, it tries again with another randomly selected tile.

Giving the player control

The final step this week is to give the player control of the game. We will do this with an infinite loop in which we ask the user for their next move.

To keep the code simple, we will split it across a few methods. The core moves, up, down, left and right will each be associated with a key. This mapping between keys and moves can be specified in the `__init__` method by creating an instance variable `moves`.

```
class Game:
    def __init__(self):
        self.grid = [[None, None, None, None],
                     [None, None, None, None],
                     [None, None, None, None],
                     [None, None, None, None]]
        self.set_random_empty_tile(2)
        self.set_random_empty_tile(2)

        self.moves = {
            "W": core.move_up,
            "A": core.move_left,
            "S": core.move_down,
            "D": core.move_right
        }
```

> This also requires us to import the `core` module at the top of the file (after we import from the random module).

The `process_command` method takes a character as an argument and executes the appropriate move.

```
    def process_command(self, command):
        self.grid = self.moves[command](self.grid)
```

This is a neat little trick, we are picking the appropriate function from our `self.moves` dictionary based on the provided command (which should be one of "W", "A", "S" and "D"). We then call the function with `self.grid` as an argument and assign the result to `self.grid`.

> If you prefer breaking down methods into more lines of code, this is exactly the same:
>
> ```
>     def process_command(self, command):
>         move_function = self.moves[command]
>         next_grid = move_function(self.grid)
>         self.grid = next_grid
> ```

Now we need to get user input to pass into the `process_command` method. We do this in the `next_move` method. It prints out the current grid and prompts the user with a set of valid keys, indicating the possible moves.

```
    def next_move(self):
        print(self)
        commands = input("move (W=Up, A=Left, S=Down, D=Right): ").upper()
        for c in commands:
            self.process_command(c)
```

The user input is converted to upper case and we are allowing the user to enter multiple commands in one go by looping over the resultant string and passing each character into the `process_command` method, one at a time.

Finally, the `play` method will set up an infinite loop. All it does (for now) is repeatedly call the `next_move` method.

```
    def play(self):
        while True:
            self.next_move()
```

This very basic game loop implementation allows the game to be played for the first time.

```
$ python3 game.py

[None, None, 2, None]
[None, None, None, None]
[None, None, None, None]
[2, None, None, None]

move (W=Up, A=Left, S=Down, D=Right): d

[None, None, None, 2]
[None, None, None, None]
[None, None, None, None]
[None, None, None, 2]

move (W=Up, A=Left, S=Down, D=Right): s

[None, None, None, None]
[None, None, None, None]
[None, None, None, None]
[None, None, None, 4]

move (W=Up, A=Left, S=Down, D=Right): was

[None, None, None, None]
[None, None, None, None]
[None, None, None, None]
[4, None, None, None]

move (W=Up, A=Left, S=Down, D=Right):
```

Exit the game with `ctrl+C` to raise a KeyboardInterrupt error.

The final code looks like this:

```
from random import randint

import core

class Game:
    def __init__(self):
        self.grid = [[None, None, None, None],
                     [None, None, None, None],
                     [None, None, None, None],
                     [None, None, None, None]]
        self.set_random_empty_tile(2)
        self.set_random_empty_tile(2)
        self.moves = {
            "W": core.move_up,
            "A": core.move_left,
            "S": core.move_down,
            "D": core.move_right
        }

    def __str__(self):
        grid = "\n".join([str(row) for row in self.grid])
        return f"\n{grid}"

    def set_random_empty_tile(self, value):
        while(True):
            row = randint(0, 3)
            col = randint(0, 3)
            if not self.grid[row][col]:
                break
        self.grid[row][col] = value

    def process_command(self, command):
        self.grid = self.moves[command](self.grid)

    def next_move(self):
        print(self)
        commands = input("\nmove (W=Up, A=Left, S=Down, D=Right): ").upper()
        for c in commands:
            self.process_command(c)

    def play(self):
        while True:
            self.next_move()


if __name__ == "__main__":
    g = Game()
    g.play()
```

Challenges

We wrote a lot of code this week. Make sure you take some time to review the whole `Game` class and see how it fits together. Think about what features are missing.

- What happens when the user enters an invalid command? Update the code to provide useful feedback.
- We need a way to exit the programme, implement a "Q" to quit.
- Try to upgrade the game to insert a `2` or `4` into a randomly selected empty tile after each move.