Lab 5: A fully working game

This week we will convert our basic playable system into a fully working implementation of the 2048 game. This requires us to add random tiles after each move and to detect the game over state where no more moves are possible.

There are a few wrinkles in this process so pay attention.

Upgrading the `__str__` method

First, we will upgrade our **str** method to center each tile in a four character string.

```
    def __str__(self):
        # nested list comprehension - converts all tiles into strings
        tiles = [[str(t or ".").center(4) for t in row] for row in self.grid]

        # join the tiles with a space and the rows with a newline character
        result = "\n".join([" ".join(row) for row in tiles])

        # add an extra newline character before and after the grid
        return f"\n{result}\n"
```

This is fairly complex because it involves the following nested list comprehension.

```
tiles = [[str(t or ".").center(4) for t in row] for row in self.grid]
```

The outer list comprehension (`for row in self.grid`) returns a new list for each row where the new list is the result of the inner list comprehension.

The inner list comprehension (`for t in row`) converts each tile using `str(t or ".").center(4)`. Tiles with numeric values will be converted to strings whilst `None` tiles will evaluates to `"."` (which is already a string). The resultant string is padded with spaces to become four-characters using `center()`.

Then we join all the strings using spaces between tiles and newline characters (`\n`) between rows. This code creates a single string representing the tile data.

```
result = "\n".join([" ".join(row) for row in tiles])
```

The string `join()` method converts a list into a single string, joined by the given character(s). In this case, we are converting each row into a string with a list comprehension. Then we join the resultant list with newline characters.

Once we add a few extra newline characters, the result looks like this:

```


  .    .    .    .
  .    .    2    2
  .    .    .    .
  .    .    .    .


```

Which is much nicer for seeing how the tiles line up in columns.

Handling invalid commands

Now, we will handle invalid user input. If we enter an invalid command such as 'k', the programme currently crashes.

```
$ python3 game.py


  .    .    .    .
  .    2    .    2
  .    .    .    .
  .    .    .    .


move (W=Up, A=Left, S=Down, D=Right, Q=Quit): k
Traceback (most recent call last):
  File "step_01/game.py", line 49, in <module>
    g.play()
  File "step_01/game.py", line 45, in play
    self.next_move()
  File "step_01/game.py", line 41, in next_move
    self.process_command(c)
  File "step_01/game.py", line 35, in process_command
    self.grid = self.moves[command](self.grid)
KeyError: 'K'
```

We need to handle this `KeyError` which occurs when any command other than "W", "A", "S", or "D" is entered. We will do this by using a `try except` construct. Rather than handling it where it occurs (on line 35 in `process_command` in my case, your error may differ) we will do this one level up by wrapping the call to `self.process_command` within in the `next_move` method (on line 41 in my case) in a `try` block. We do this mainly because this is where we handle the user input and it helps to keep our `process_command` method clean.

```
    def next_move(self):
        print(self)
        commands = input("\nmove (W=Up, A=Left, S=Down, D=Right): ").upper()
        for c in commands:
            try:
                self.process_command(c)
            except KeyError:
                print(f"\nInvalid command: {c}")
```

We have added an exception handler to respond to the case when the `process_command` method raises a `KeyError`. In which case, we print a simple message and allow the loop to continue.

Now the game will no longer crash if we provide an invalid command.

```
 2    .    .    .
 2    .    .    .
 .    .    .    .
 .    .    .    .


move (W=Up, A=Left, S=Down, D=Right, Q=Quit): k

Invalid command: K

 2    .    .    .
 2    .    .    .
 .    .    .    .
 .    .    .    .


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

Now, we also need to implement a way to exit the programme.

We will do this by modifying the `play` method.

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

We have created an attribute `self.playing` which must be `True` to keep the game loop running. Setting it to `False` will end the game cleanly and exit the programme.

We can do this within the exception handler, testing for the letter "Q".

```
    def next_move(self):
        print(self)
        commands = input("\nmove (W=Up, A=Left, S=Down, D=Right, Q=Quit):
").upper()
        for c in commands:
            try:
                self.process_command(c)
            except KeyError:
                if c == "Q":
                    self.playing = False
                else:
                    print(f"\nInvalid command: {c}")
```

So, if the user enters a "Q", the new `self.playing` attribute will be set to `False`.

Try it. You should now be able to enter any value without crashing the programme and should be able to exit with a "Q" command.

Inserting tiles

To complete the basic game rules, after each move, a randomly selected empty tile should be set to either 2 or 4. This change will begin to make the game truly playable because new merges will be possible each move and the numbers will be able to accumulate.

However, we need to be careful. A naive implementation might be something like this. Just adding a call to the existing `set_random_empty_tile` method to the end of the `process_command` method.

```
    def process_command(self, command):
        self.grid = self.moves[command](self.grid)
        self.set_random_empty_tile(2)
```

Obviously, one problem is that this will always add a 2 and never a 4. However, there is a deeper problem with this approach.

Try playing the game. It seems to work well. Until you hit a situation where a move makes no difference.

Consider this simple situation as an example.

```
.    .    .    .
2    .    .    .
2    .    .    .
.    .    .    .
```

Moving up, down, or right should move the tiles and cause the addition of a new tile. However, moving *left* in this situation should not cause the addition of a new tile. It should have no effect.

So, we need our code to ignore cases where a move has no effect. We can do this by comparing the result of the move with the original grid.

```python
    def process_command(self, command):
        next_grid = self.moves[command](self.grid)
        if next_grid != self.grid:
            self.grid = next_grid
            self.set_random_empty_tile(2)
```

Now we can play the game!

But let's not celebrate yet.

One final tweak. We need to occasionally insert a `4` instead of a `2`. We can do this by using the `choice` function from the `random` module.

Update the `import` statement at the top of the file.

```python
from random import randint, choice
```

and update the `process_command` method as follows.

```python
    def process_command(self, command):
        next_grid = self.moves[command](self.grid)
        if next_grid != self.grid:
            self.grid = next_grid
            new_tile = choice([2, 2, 2, 4])
            self.set_random_empty_tile(new_tile)
```

So we are now picking randomly from the list `[2, 2, 2, 4]`. Which will usually give us a `2` and less often, a `4`. Its easy enough to tweak this setting by adding more `2's` to the list.

Game over

The last major piece of game logic is to detect when the player can no longer make any legal moves and present the user with a game over message. This is very important because if we allow the game to reach this state without detecting it then the game will enter an infinite loop looking for an empty tile. So we need to exit the game when this happens.

The detection is divided into three parts which we will implement in our `core.py` module and test separately. Firstly, if there are **any** empty tiles, there are always legal moves available because tiles can move into the gaps. So we need a function that detects empty tiles.

```
def has_gaps(grid):
    for row in grid:
        if None in row:
            return True
    return False
```

The function simply loops over each row and checks to see if it contains `None`. If it does find a `None` then it returns `True` immediately. If no rows contain `None` then it returns `False`.

We can add a new test case to `test.py`.

```
class TestHasGaps(unittest.TestCase):

    def test_no_gaps(self):
        input  = [[ 1,  2,  3,  4],
                  [ 5,  6,  7,  8],
                  [ 9, 10, 11, 12],
                  [13, 14, 15, 16]]
        self.assertFalse(core.has_gaps(input))

    def test_one_gap(self):
        input  = [[ 1,  2,  3,  4],
                  [ 5,  6,  7,  8],
                  [ 9, 10, 11, 12],
                  [13, 14, 15, None]]
        self.assertTrue(core.has_gaps(input))
```

To detect game over, if there are no gaps, then we need to look for potential merges, i.e. similar tiles next to each other in the grid. We will need to look for both vertical and horizontal merges.

The two functions, `has_vertical_gaps` and `has_horizontal_gaps` are very similar. They both loop over the data and look for pairs of similar tiles. As soon as they find a pair they return `True`, if no pairs are found, they return `False`.

```python
def has_vertical_merges(data):
    for row in range(3):
        for col in range(4):
            if data[row][col] == data[row + 1][col]:
                return True
    return False

def has_horizontal_merges(data):
    for row in range(4):
        for col in range(3):
            if data[row][col] == data[row][col + 1]:
                return True
    return False
```

Similar tests are easy to design.

```python
class TestVerticalMerges(unittest.TestCase):

    def test_no_merges(self):
        input  = [[ 1,  2,  3,  4],
                  [ 5,  6,  7,  8],
                  [ 9, 10, 11, 12],
                  [13, 14, 15, 16]]
        self.assertFalse(core.has_vertical_merges(input))

    def test_one_merge(self):
        input  = [[ 1,  2,  3,  4],
                  [ 5,  6,  7,  8],
                  [ 9, 10, 11, 12],
                  [13, 10, 15, 16]]
        self.assertTrue(core.has_vertical_merges(input))

class TestHorizontalMerges(unittest.TestCase):

    def test_no_merges(self):
        input  = [[ 1,  2,  3,  4],
                  [ 5,  6,  7,  8],
                  [ 9, 10, 11, 12],
                  [13, 14, 15, 16]]
        self.assertFalse(core.has_horizontal_merges(input))

    def test_one_merge(self):
        input  = [[ 1,  2,  3,  4],
                  [ 5,  6,  7,  8],
                  [ 9, 10, 10, 12],
                  [13, 14, 15, 16]]
        self.assertTrue(core.has_horizontal_merges(input))
```

With these functions fully tested, we can implement a convenient function that checks to see if there are any valid moves available.

```python
def is_game_over(data):
    return not (
        has_gaps(data) or has_vertical_merges(data) or has_horizontal_merges(data)
    )
```

This then allows us to implement game over detection in our `game.py` module. First, in the main `play` method, we add a `self.game_over` boolean attribute. The code will break out of the game loop if this is set to `True` (or if the user quits).

```
    def play(self):
        self.game_over = False
        self.playing = True
        while self.playing and not self.game_over:
            self.next_move()
        print(self)
```

We also add an extra `print(self)` to the end, so we can show the user the end state of the game after we exit the loop.

Now, when we process each command, we need to check for *game over* using our new core function.

```
    def process_command(self, command):
        next_grid = self.moves[command](self.grid)
        if next_grid != self.grid:
            self.grid = next_grid
            new_tile = choice([2, 2, 2, 4])
            self.set_random_empty_tile(new_tile)
            self.game_over = core.is_game_over(self.grid)
```

This now works. But we should also upgrade our `__str__` method to provide the user with some feedback when the game ends. This can either be due to the user quitting, or due to the game being over.

```
    def __str__(self):
        tiles = [[str(t or ".").center(4) for t in row] for row in self.grid]
        result = "\n".join([" ".join(row) for row in tiles])
        msg = ""
        if not self.playing:
            msg = "\nYOU QUIT THE GAME\n"
        if self.game_over:
            msg = "\nGAME OVER\n"
        return f"\n{result}\n{msg}"
```

On quitting the game or achieving a game over state, we now present a message when the programme exits.

Challenges

The one final missing feature is scoring. Think about how we might calculate the score.

The best place to work out the score is within the merge algorithm. Consider how we might calculate and access the score from this method and how it would propagate through our existing code into a `self.score` attribute on the Game class.