> We interrupt this film to apologise for this unwarranted attack by the supporting feature.
>
> Luckily, we have been prepared for this eventuality, and are now taking steps to remedy it.

Sudoku Example

We are now going to have a look at an example that will incorporate list and file processing. This example will be of a Sudoku game stored in the file `sgame.txt`. The pretty-printed form of this game is `sgame.pdf`.

The first step is to write a function that takes the name of a file that contains a Sudoku game and returns a representation of the game. The data structure we use to store the game is a list of lists — the entries are rows and each entry in a row represents a column entry of the row. To do this we also write a function to turn a string representing a row into a row.

```
def row2list(row_string) :
    """Convert a string representing a row in a Sudoku game into a list.

    Parameters:
        row_string (string): Represents a single row in a Sudoku game.

    Return:
        list<string>: Containing the numbers in a row of a Sudoku game.

    Preconditions:
        Numbers in 'row_string' are single digits
        and there is a single space separating the numbers.
    """
    row = []
    for i in range(0, 18, 2) :
        row.append(row_string[i])
    return row
```

This function initialises an empty list for collecting the row entries. Then uses a `for` loop with `range` that goes through every second number, up to but not including 18. As each entry is a space apart in the string we only want every second character in that string. The character at each entry position is then appended into the `row` list.

```
def read_game(filename) :
    """Read the data for a Sudoku game from 'filename'.

    Parameters:
        filename (string): Name of the file from which to read the game data.

    Return:
        list<list<string>>: Representation of a Sudoku game as a matrix
                            (list of lists) of strings.

    Preconditions:
        The files can be opened for reading and writing.
        File contains 9 lines and each line represents one row of the game.
    """
    file = open(filename, 'r')
```

```
    game = []
    for line in file :
        game.append(row2list(line))
    file.close()
    return game
```

This function opens the file in universal read mode and then initialises an empty list representing the game data structure. Then for every line in the file the `row2list` function is called on the line to get the row list. That row list is then appended to the `game` list to create the full game.

Our code currently looks like `sudoku1.py`. A couple of tests of the functions are below.

```
>>> row2list('1 2 3 4 5 6 7 8 9')
['1', '2', '3', '4', '5', '6', '7', '8', '9']
>>> row2list('1 2   4 5 6 7 8  ')
['1', '2', ' ', '4', '5', '6', '7', '8', ' ']
>>> for i in read_game('sgame.txt'):
        print(i)

[' ', ' ', '1', ' ', ' ', ' ', ' ', '9', ' ']
['7', ' ', ' ', '1', ' ', '3', ' ', '5', ' ']
[' ', ' ', ' ', ' ', '9', '8', '1', ' ', '6']
[' ', '9', ' ', ' ', '4', ' ', ' ', ' ', ' ']
[' ', '8', '4', ' ', ' ', ' ', '6', '1', ' ']
[' ', ' ', ' ', ' ', '5', ' ', ' ', '2', ' ']
['6', ' ', '5', '9', '3', ' ', ' ', ' ', ' ']
[' ', '7', ' ', '4', ' ', '2', ' ', ' ', '1']
[' ', '2', ' ', ' ', ' ', ' ', '3', ' ', ' ']
>>>
```

Now we need functions to extract a given row, column or 3x3 block. Adding the following functions to our code will give us this functionality. Keep in mind that the game data structure can be thought of as a 2-dimensional matrix.

```
def get_row(row, game) :
    """Return the indicated 'row' from 'game'.

    Parameters:
        row (int): Index of the row to extract from 'game'.
        game (list<list<string>>): Matrix representation of the Sudoku game grid.

    Return:
        list<string>: The numbers in a row of a Sudoku game.

    Preconditions:
        0 <= 'row' <= 8
        'game' is a list representation of a Sudoku game.
    """
    return game[row]
```

This function simply needs to return the rowth index of game as game is a list of rows.

```
def get_column(col, game) :
    """Return the column indicated by 'col' from 'game'.
```

```
    Parameters:
        col (int): Index of the column to extract from 'game'.
        game (list<list<string>>): Matrix representation of the Sudoku game grid.

    Return:
        list<string>: The numbers in a column of a Sudoku game.

    Preconditions:
        0 <= 'col' <= 8
        'game' is a list representation of a Sudoku game.
    """
    column = []
    for row in game :
        column.append(row[col])
    return column
```

This is a bit more tricky than `get_row` as we do not have a list of columns. Therefore, the function needs to go through every row and collect the values at the column index of the row.

```
def get_block(row, col, game) :
    """Return the 3x3 block starting at index [row, col] from 'game'.

    Parameters:
        row (int): Row index of the start of the block to extract from 'game'.
        col (int): Column index of the start of the block to extract from 'game'.
        game (list<list<string>>): Matrix representation of the Sudoku game grid.

    Return:
        list<string>: The numbers in a 3x3 block of a Sudoku game.

    Preconditions:
        0 <= r < 3 and 0 <= c < 3
        'game' is a list representation of a Sudoku game.
    """
    block = []
    for block_row in range(3*row, 3*row+3) :
        block.extend(game[block_row][3*col:3*col+3])
    return block
```

This function is even more complicated. We need to be able to get a 3x3 block, therefore we need to get three adjacent rows and the three adjacent entries from each of those rows. The for loop gets the rows using range to generate indices that correspond to the block required. The `3*row` is used as the block rows are numbered 0, 1 or 2 down the game data structure. This moves the start index to the required starting row of the board that corresponds to the start of that block. The end row is simply 3 more rows down. As `game` is a list of lists representing rows it can be directly indexed to get the row of interest. Then slicing can be used on that row to get the 3 adjacent columns. This is done using the same system as obtaining the rows. The `extend` method of lists modifies the list it is called on by joining the list in its argument to the first list to form one list.

Our Sudoku code now looks like `sudoku2.py`. Here are some tests of the functions that we have just written.

```
>>> game = read_game('sgame.txt')
>>> get_row(1, game)
```

```
['7', ' ', ' ', '1', ' ', '3', ' ', '5', ' ']
>>> get_column(0, game)
[' ', '7', ' ', ' ', ' ', ' ', '6', ' ', ' ']
>>> get_block(1, 2, game)
[' ', ' ', ' ', '6', '1', ' ', ' ', '2', ' ']
>>>
```

To finish off this problem we look at the problem of determining what possible values can be put in an empty square. To do this, we need to determine what are the possibilities based on entries already in a given row, column or block. This can be done by finding the difference between the given entries and the valid entries. Below are the functions that will allow us to do this.

```
def list_diff(list1, list2) :
    """Return the list of entries in list1 that are not in list2 (list1 - list2).

    A general-purpose list function that works for lists of any type of elements.
    """
    diff = []
    for element in list1 :
        if element not in list2 :
            diff.append(element)
    return diff
```

This function creates an empty result list, `diff`. The function then goes through every `element` in `list1` and checks if it is in `list2`. If `element` is not in `list2` then `element` is appended to our result list `diff`.

```
# all the valid choices
ALL_CHOICES = ['1', '2', '3', '4', '5', '6', '7', '8', '9']

def choices(row, column, game) :
    """Return choices that are possible at position indicated by [row, column].

    Identify and return all the choices that are possible at position
    [row, column] in 'game' - i.e. each choice should not occur in the
    indicated 'row', 'column' or in the block containing this position.

    Parameters:
        row (int): Row index of the position to check.
        column (int): Column index of the position to check.
        game (list<list<string>>): Matrix representation of the Sudoku game grid.

    Return:
        list<string>: The valid choices at this position.

    Preconditions:
        0 <= row <= 8 and 0 <= column <= 8  game[row][column] == ' '
    """
    block_row = row / 3
    block_col = column / 3
    choices = list_diff(ALL_CHOICES, get_row(row, game))
    choices = list_diff(choices, get_column(column, game))
    choices = list_diff(choices, get_block(block_row, block_col, game))
    return choices
```

This function starts by finding the block that the coordinate [row, column] is in and

setting this to `block_row` and `block_col`. Then it finds all the choices for the row by calling `list_diff` with the list `ALL_CHOICES` and the list obtained from `get_row` as arguments. Then choices is updated by passing it into `list_diff` along with the result from `get_column` and assigning the result back into `choices`, this removes all the non-possible choices from the column from the `choices` list. This is again done with the block. This then leaves all the choices available for that square so `choices` is returned.

Finally, our sudoku code looks like `sudoku.py`. Here results of some tests of the last few function we wrote.

```
>>> list_diff(ALL_CHOICES, get_row(0, game))
['2', '3', '4', '5', '6', '7', '8']
>>> list_diff(ALL_CHOICES, get_column(0, game))
['1', '2', '3', '4', '5', '8', '9']
>>> list_diff(ALL_CHOICES, get_block(0, 0, game))
['2', '3', '4', '5', '6', '8', '9']
>>> list_intersection(['2', '3', '4', '5', '6', '7', '8'],
                      ['1', '2', '3', '4', '5', '8', '9'])
['2', '3', '4', '5', '8']
>>> choices(0, 0, game)
['2', '3', '4', '5', '8']
>>>
```