# Intelligent Systems

## Laboratory Documentation

Escuela Superior de Informática

2024

Carlos Ruiz García-Casarrubios

# TASK 1

This task consists of implementing the basics of the Sokoban game, in this case, in python. Sokoban is a logic video game that was designed in Japan in 1981 by Hiroyuki Imabayashi and first published in 1982. The object of the game is for the player, who controls a character called "Sokoban" (meaning "warehouse manager" in Japanese), to move boxes to their designated positions in a warehouse.

## Basic rules of the Sokoban game (context of the task):

1. Goal:
   - Move all boxes to designated target areas, usually marked on the floor with dots or squares.
2. Movement:
   - The player can move in four directions: up, down, left and right.
   - Only one box can be moved at a time by pushing it, i.e. the player must be behind the box to move it forward.
3. Restrictions:
   - Boxes and the player cannot go through walls or obstacles.
   - Boxes cannot be pushed outside the area bounded by walls.
4. Strategy:
   - The key to the game is to plan your moves carefully. Pushing a box into a corner or dead end can make it impossible to finish the level, forcing the player to restart it.
5. Levels:
   - The game consists of multiple levels, each with a different configuration of boxes, walls and target areas.
   - The complexity increases as you progress, requiring more planning and strategic thinking.

In the case of this task, the Sokoban board will be represented as a string of characters where each character has the following meaning:

Wall (#): It represents the board boundaries or barriers within the board…

Player (@): This is the character that the player controls to move the boxes.

Box ($): Element that the player must push towards the target.

Target (.): Place where the boxes must be placed to complete the level.

Box on a target (*): Indicates that a box has been correctly placed on a target.

Player on a target (+): Represents the player standing on a cell that is also a target.

Empty space (' '): Cells where there is no other element and through which the player can move.

Now having this to serve as a context, the goal of this specific task is to be able to get an output in this format, for example:

Taking this as the input:

*python sokoban.py T1 -l '###########\n####  .#  #\n#### # $. #\n####  $#  #\n#@$.   ## #\n#   ###   #\n#   ###   #\n###########'*

The output must be:

ID:C802519BD29D485361F28070E885B6DC

Rows:8

Colums:11

Walls:[(0,0),(0,1),(0,2),(0,3),(0,4),(0,5),(0,6),(0,7),(0,8),(0,9),(0,10),(1,0),(1,1),(1,2),(1,3),(1,7),(1,10),(2,0),(2,1),(2,2),(2,3),(2,5),(2,10),(3,0),(3,1),(3,2),(3,3),(3,7),(3,10),(4,0),(4,7),(4,8),(4,10),(5,0),(5,4),(5,5),(5,6),(5,10),(6,0),(6,4),(6,5),(6,6),(6,10),(7,0),(7,1),(7,2),(7,3),(7,4),(7,5),(7,6),(7,7),(7,8),(7,9),(7,10)]

Targets:[(1,6),(2,8),(4,3)]

Player: (4,1)

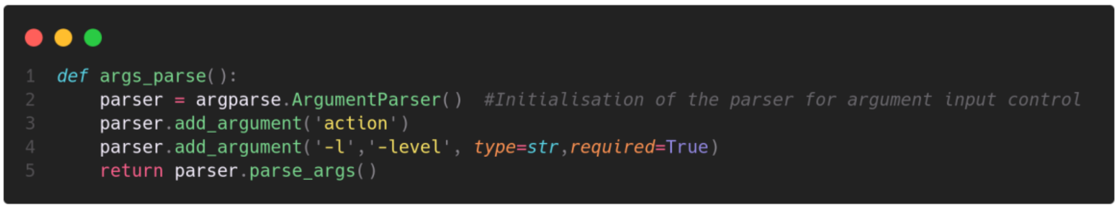
Boxes:[(2,7),(3,6),(4,2)]

This means that the format for a correct program usage command must be one of the following:

*python sokoban.py T1 -l <level>*

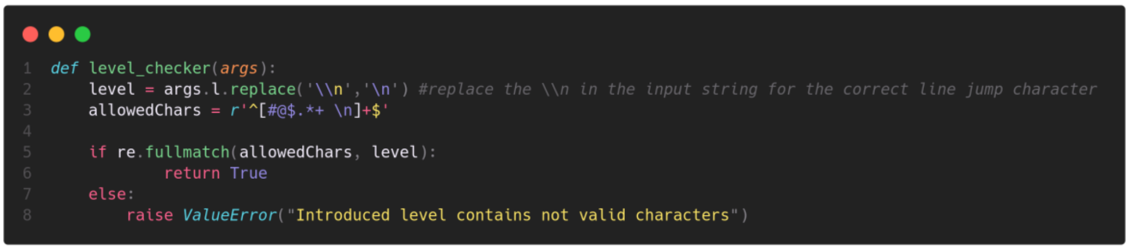
*python sokoban.py T1 -level <level>*

For getting this objective done the method *args_parse* is done, which objective is to parse the command line arguments and ensuring they are correctly used.



```python
def args_parse():
    parser = argparse.ArgumentParser()  #Initialisation of the parser for argument input control
    parser.add_argument('action')
    parser.add_argument('-l','-level', type=str,required=True)
    return parser.parse_args()
```

But even though the command line arguments are now controlled, in the level string there could be non-desired characters, making the introduced level invalid, to solve this problem a method *level_checker* is created, where the program controls if the characters introduced in the level string match the allowed characters, that we save on a variable *allowedChars.*

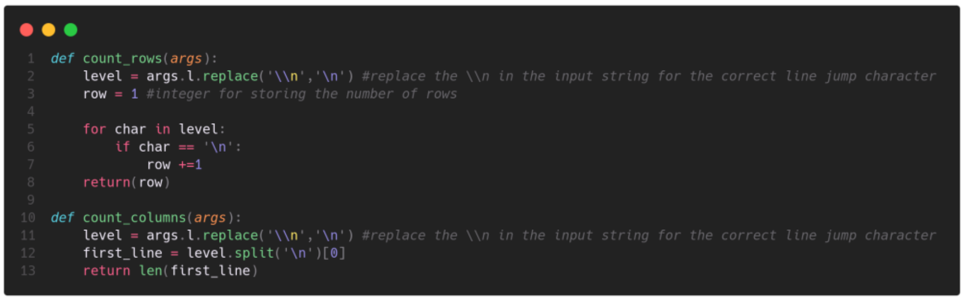


```python
def level_checker(args):
    level = args.l.replace('\\n','\n') #replace the \\n in the input string for the correct line jump character
    allowedChars = r'^[#@$.*+ \n]+$'

    if re.fullmatch(allowedChars, level):
            return True
    else:
        raise ValueError("Introduced level contains not valid characters")
```

Once we have the program running with a correct command input and level, the next step done is converting the level string into a bidimensional array, for manipulating this way the level, allowing us to travel it and get desired positions of level assets.

This is done at the method *create_level_matrix,* which takes as arguments the number of rows, columns the level has as well as the level string. The number of rows and columns are obtained by the methods *count_rows* and *count_columns* respectively.

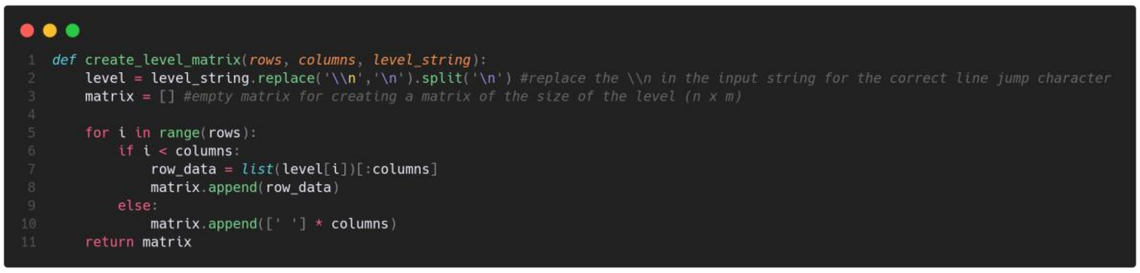


```python
def count_rows(args):
    level = args.l.replace('\\n','\n') #replace the \\n in the input string for the correct line jump character
    row = 1 #integer for storing the number of rows

    for char in level:
        if char == '\n':
            row +=1
    return(row)

def count_columns(args):
    level = args.l.replace('\\n','\n') #replace the \\n in the input string for the correct line jump character
    first_line = level.split('\n')[0]
    return len(first_line)
```

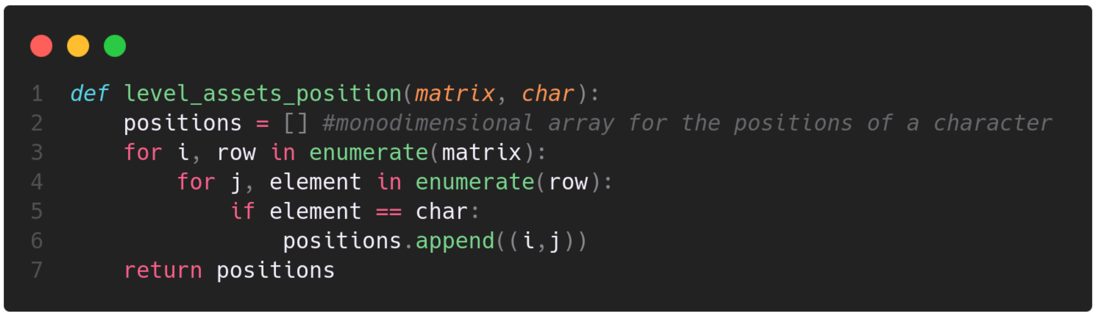
Matrix creation method is:



```python
def create_level_matrix(rows, columns, level_string):
    level = level_string.replace('\\n','\n').split('\n') #replace the \\n in the input string for the correct line jump character
    matrix = [] #empty matrix for creating a matrix of the size of the level (n x m)

    for i in range(rows):
        if i < columns:
            row_data = list(level[i])[:columns]
            matrix.append(row_data)
        else:
            matrix.append([' '] * columns)
    return matrix
```

Once the level is converted into a matrix, we can travel the matrix to obtain the positions of a desired level asset, this is done at a method called *level_assets_position*, that has a vector that fills with the position of a character each time that character is found in the matrix, it takes as arguments the level matrix and the character to be found.

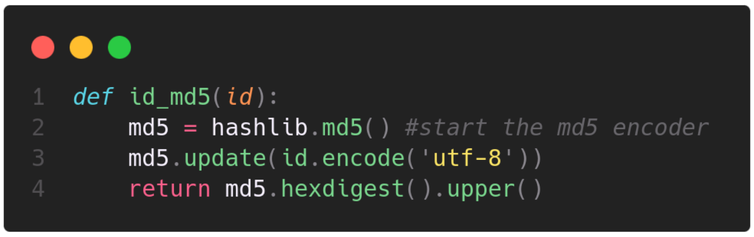


```python
def level_assets_position(matrix, char):
    positions = [] #monodimensional array for the positions of a character
    for i, row in enumerate(matrix):
        for j, element in enumerate(row):
            if element == char:
                positions.append((i,j))
    return positions
```

Now, with this done the only tasks remaining to achieve the objective are to get the md5 encoded id and to print in the correct format.

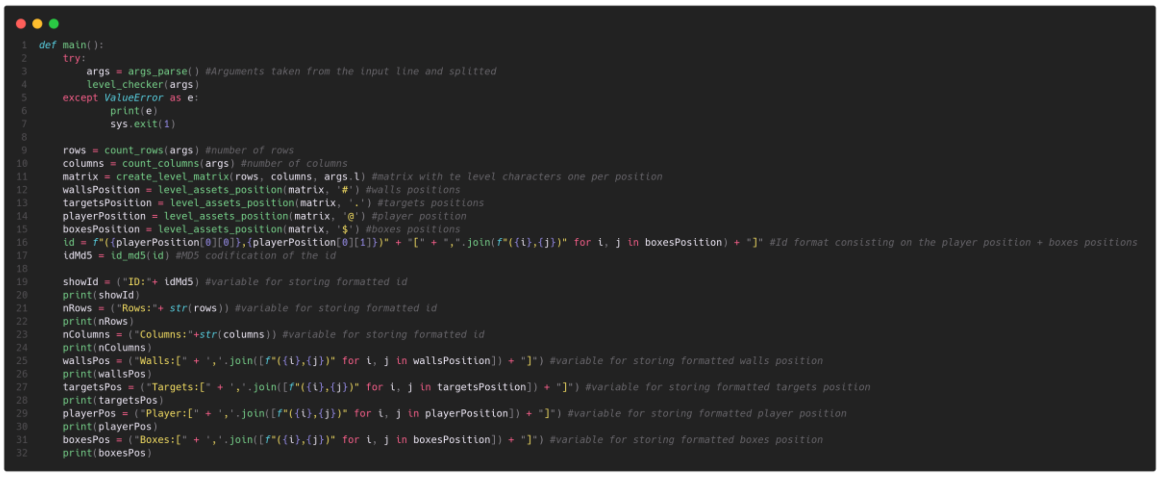
First, we must create the id, which is a combination of the player position followed by the boxes position, with following the example input shown before would be: (4,1)[(2,7),(3,6),(4,2)]

This message is created at the main method of our program and encoded to md5 in the *id_md5* method which takes as an input the id in a string format.



```python
def id_md5(id):
    md5 = hashlib.md5() #start the md5 encoder
    md5.update(id.encode('utf-8'))
    return md5.hexdigest().upper()
```

Now we just must use the main method to get the desired output, this method is:



```python
def main():
    try:
        args = args_parse() #Arguments taken from the input line and splitted
        level_checker(args)
    except ValueError as e:
            print(e)
            sys.exit(1)

    rows = count_rows(args) #number of rows
    columns = count_columns(args) #number of columns
    matrix = create_level_matrix(rows, columns, args.l) #matrix with te level characters one per position
    wallsPosition = level_assets_position(matrix, '#') #walls positions
    targetsPosition = level_assets_position(matrix, '.') #targets positions
    playerPosition = level_assets_position(matrix, '@') #player position
    boxesPosition = level_assets_position(matrix, '$') #boxes positions
    id = f"({playerPosition[0][0]},{playerPosition[0][1]})" + "[" + ",".join(f"({i},{j})" for i, j in boxesPosition) + "]" #Id format consisting on the player position + boxes positions
    idMd5 = id_md5(id) #MD5 codification of the id

    showId = ("ID:"+ idMd5) #variable for storing formatted id
    print(showId)
    nRows = ("Rows:"+ str(rows)) #variable for storing formatted id
    print(nRows)
    nColumns = ("Columns:"+str(columns)) #variable for storing formatted id
    print(nColumns)
    wallsPos = ("Walls:[" + ','.join([f"({i},{j})" for i, j in wallsPosition]) + "]") #variable for storing formatted walls position
    print(wallsPos)
    targetsPos = ("Targets:[" + ','.join([f"({i},{j})" for i, j in targetsPosition]) + "]") #variable for storing formatted targets position
    print(targetsPos)
    playerPos = ("Player:[" + ','.join([f"({i},{j})" for i, j in playerPosition]) + "]") #variable for storing formatted player position
    print(playerPos)
    boxesPos = ("Boxes:[" + ','.join([f"({i},{j})" for i, j in boxesPosition]) + "]") #variable for storing formatted boxes position
    print(boxesPos)
```