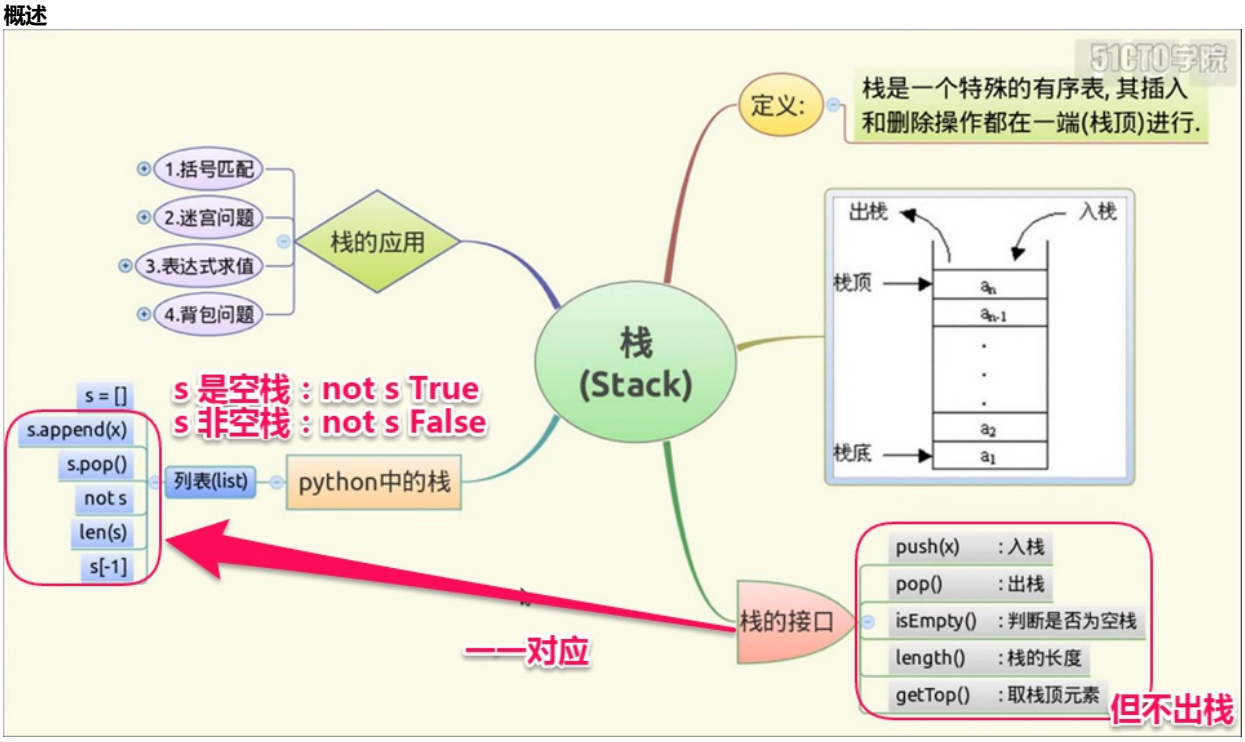
栈-1

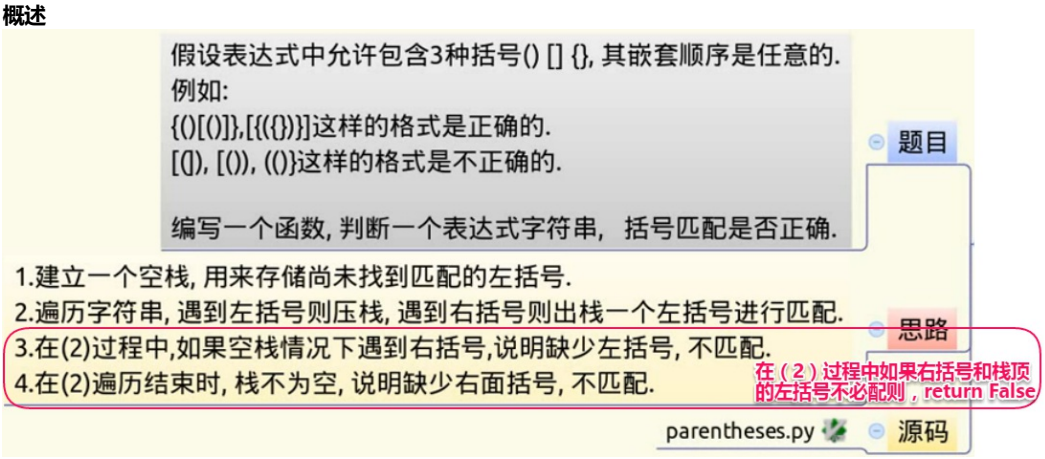
match.py

maze.py

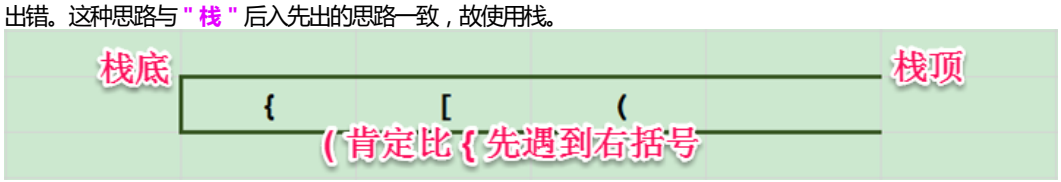
- **概述**



- **案例**
  - **案例-1：判断括号是否匹配**
    - **概述**



    - **思路**
      1. 大致思路？左括号进栈，用右括号消除栈中的左括号。
      2. 为什么使用 " 栈 " 这种数据结构？原因在于后入栈的左括号一定比先入栈的左括号先匹配上右括号，否则就会出错。这种思路与 " 栈 " 后入先出的思路一致，故使用栈。



      3. 三种错误？
         1. 匹配过程中，空栈遇到右括号，右括号多了。
         2. 匹配过程中，右括号与栈顶左括号匹配不上。
         3. 匹配结束，非空栈，左括号多了。
      4.

- **代码-1**：忽略非括号字符，测试用例只能是 '{(()))}'

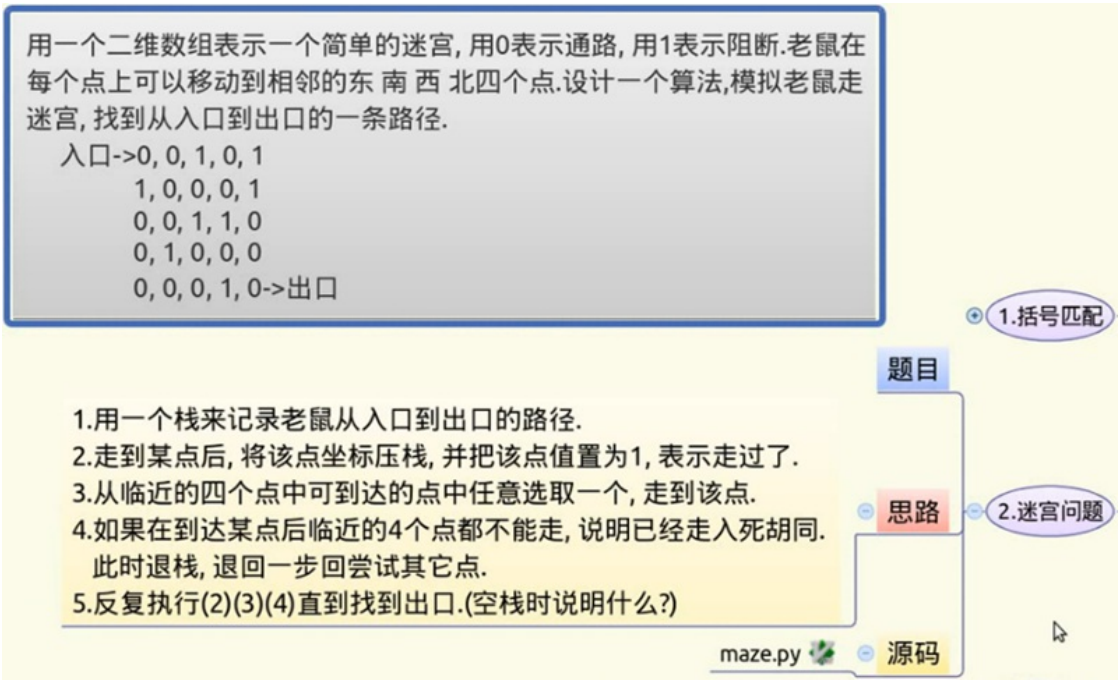
```
# coding:utf-8

def match(expr):

    # 定义左括号
    LEFT = {'{', '(', '['}
    # 定义右括号
    RIGH = {'}', ')', ']'}

    expr = filter(lambda char: True if char in LEFT | RIGH else False, expr)

    # 空列表用来存储待消除左括号
    stack = []

    for char in expr:
        # 如果是左括号 , 进栈
        if char in LEFT:
            stack.append(char)
        # 如果是右括号
        elif char in RIGH:
            # 如果此时已经是空栈 , 说明右括号多了 , 匹配出错
            if not stack:
                return u'匹配出错 , 右括号多'
            # 如果此时栈顶元素与右括号 , 匹配不上 , 匹配报错
            elif not 0 < ord(char) - ord(stack[-1]) <= 2:
                return u'匹配出错 , 匹配不上'
            # 匹配成功 , 栈顶元素出栈
            stack.pop()

    if not stack:
        return u'匹配成功'
    # 匹配结束 , 非空栈 , 说明左括号多了 , 匹配出错
    else:
        return u'匹配出错 , 左括号多'


if __name__ == '__main__':

    expr = '{{d(a[]a)}+}'

    print match(expr)
```

这个出栈操作一定要在右括号判断这个 elif 中，否则左括号会直接出栈，没有经过匹配。

- **代码-2**：过滤非括号字符，测试用例只能是 '{(a[b]@)d)++}'

```
# 并集：    " | " 符号是求 LEFT、 RIGH 两个集合的并集
# 行内 if：if char in LEFT | RIGH 为真返回 True 否则（else ） 返回 False
expr = filter (lambda char : True if char in LEFT | RIGH else False, expr)
```

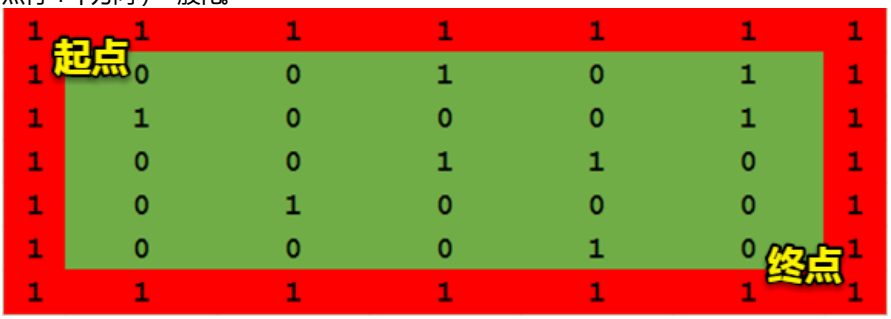
- **案例-2：迷宫路径问题**
  - **概述**



  - **思路**
    1. 为什么使用 " 栈 " 这种数据结构？大家都知道，至于迷宫的求解问题，可以用穷举法进行求解。那么什么是穷举法了，就是将每一种可能的情况都穷举完。而具体到迷宫的求解问题上，由于在求解过程中可能会遇到某一路径不可行的情况，此时我们就必须按**原路返回**，这时自然也就会想到栈的应用了，因为栈的一个很重要的特

性就是"**先进后出**"，可以用来记录每次所探索的路径，方便在原路返回的过程中，得到上一步所走路径，再按此方法，退回到可以走得通的位置上，继续探索下去，直到到达终点或者最终无法到达，正常退出程序为止。

2. 如何设计迷宫？如下这种方式（**5×5 的迷宫初始化成 7×7 的，最外层设置成 1**），之所以使用这种方式设计迷宫在于使得迷宫中所有点走一步都有四种方向选择（否则角上的点只有 2 个方向、边上的点有 3 个方向、内部点有 4 个方向）一般化。



3. 走过的位置都设置成 1，不管退不退回。**否则有可能出现死循环，走不通的路来回走。**

4. 栈顶的位置代表当前走到的位置。（当前位置）。
5. 空栈代表迷宫没有走出去的路径。

- **代码**

```
def path(maze, start, end):
    # i, j 代表起点的位置
    i, j = start
    # ei, ej 代表终点的位置
    ei, ej = end

    # 用栈来存储出迷宫的路径
    stack = []
    # " 人 " 在起点
    stack.append((i, j))
    # " 人 " 在起点 , 将走过的位置置为 1
    maze[i][j] = 1

    # 循环一次 , 相当于 " 人 "  走一步
    # 如果栈为空 , 则跳出循环 , 空栈代表迷宫走不通
    while stack:
        # 取栈顶元素 , 代表 " 人 " 当前位置
        i, j = stack[-1]
        # " 人 " 此时位置已经在终点 , 跳出循环
        if (i, j) == (ei, ej):
            break

        # 循环四次尝试能否向四个方向走
        # 用 di , dj 这种增量的形式表示 " 人 " 走一步的距离
        # 四种情况代表 4 个方向
        for di, dj in [(1, 0), (0, 1), (-1, 0), (0, -1)]:
            # 如果这个方向是 0 代表能走
            if maze[i+di][j+dj] == 0:
                # " 人 " 走到这个位置
                stack.append((i+di, (j+dj)))
                # 这个位置置为 1 , 代表走到过这个位置
                maze[i+di][j+dj] = 1
                break
        # 循环结束之后 , 没有跳出循环代表无路可走 , 后退一步
        else:
            stack.pop()

    return stack
```

"人"移动到起点位置

一直回退到起点以前

这里必须有 break 否则这个 for 循环就把能够走的路都堵上了（都置为 1 了）

- **And So On**