第7章 Minesweeper

前面几张已经介绍过Pygame模块的基本使用方法，本章将会把重点放在实现扫雷游戏的Python语法和算法。在第6章 Connect 4中，我们介绍了矩阵（二维数组）的结构和基本算法，在这章则会接触矩阵的基本搜素算法。 除此之外，本章将会介绍一些在应用方便的基础知识，比如应用广泛的JSON文件和正则表达式

我们在Minesweeper游戏将要涉及的几个功能如下:

- 元组
- Pygame 通过鼠标的人机交互
- 文件处理
- JSON文件
- 正则表达式 Regular Expression
- 二维数组的深度优先搜索

7.1 元组 Tuple

7.1.1 Tuple 基本语法

元组是一种常用的对象类型，很多高层编程语言都有它的身影。 在C++中他叫作pair或是set，在Java中它叫做Tuple。元组的作用其实很简单：表示一组**相关的**数据。 等等，表示一组数据？为什么这句话听起来与第6章学习的list如此相似？ 是的，Tuple这种数据类型的应用场景和方式与list的确有很多相似的地方。来看看语法：

```
fruit_tuple = ('apple', 'banana', 'orange') # a tuple of fruits
fruit_list = ['apple', 'banana', 'orange']  # a list of fruits
```

Python作为一种高级语言，一个元组所包含的元素类型可以不一样, 也可以有多种创建方法。 在创建元组的时候，Python允许我们使用()来表示这组数据的类型为元组，就像[]表示一组数据为列表一样。 与此同时，Python同样允许我们不写()，同样表示元组。 然而，作者极其不推荐这种写法，因为很多时候这种创建方式的表达意义并不明确，会带来很差的可读性。 要时刻记住，写代码的首要目的是让人能看懂，其次才是让计算机运行

```
tup1 = ('apple', 50, 'banana', 16.7)     # 包含不同类型的元素
tup2 = 'apple', 50, 'banana', 16.7       # 不是使用()创建
tup3 = ()   # 创建空元组
tup4 = ('delicious',)    # 创建只有一个元素的元组，有逗号
tup5 = ('delicious')     # 创建只有一个元素的元组，无逗号

print(tup1)
print(tup2)
print(tup3)
print(tup4)
print(tup5)
```

依次打印tup1-tup4, 看看这段代码的运行结果是什么。从运行结果可以看出，tup4和tup5的打印结果 有很大的区别。如果元素只有一个元素，请务必记住要在元素后加上都好','，否则Python解释器会把你的 意思理解为字符串，如同tup5的运行结果：

```
('apple', 50, 'banana', 16.7)
('apple', 50, 'banana', 16.7)
()
('delicious',)
delicious
```

在了解如何创建元组后，我们需要做的就是访问元组。 访问元组元素的方式与列表类似，通过objName[index]的方式来表示元组中的元素。

```
tup = ('apple', 'banana', 'orange', 'peach') # a tuple of fruits
print(tup)
print(tup[0])   # apple
print(tup[-2])  # orange
print(tup[1:])  # ('banana', 'orange', 'peach')

for fruit in tup:
    print(fruit)
```

运行结果如下：

```
('apple', 'banana', 'orange')
apple
orange
('banana', 'orange', 'peach')
apple
banana
orange
peach
```

除了简单的创建和访问之外，元组类还支持一些列的基础计算。 Python允许我们使用'+'操作符来合并两个元组，'*'操作符来复制元组的元素：

```
tup1 = (1, 'a', 2)
tup2 = ('b', 3, 'c')
tup3 = tup1 + tup2
print(tup3) # (1, 'a', 2, 'b', 3, 'c')

tup4 = ('abc',)*3
print(tup4) # ('abc', 'abc', 'abc')
```

元组最为Python的一个类，当然要有成员函数。 Python给元组类定义了五个最常用的成员函数：

- len(tuple)
- max(tuple)
- min(tuple)
- tuple(list)

```
tup1 = (1, 2, 3)
tup2 = (1, 2, 3)
list1 = [4, 5, 6]

print(len(tup1))    # 3
print(max(tup1))    # 3
print(min(tup1))    # 1
print(tuple(list1)) # (4, 5, 6)
```

7.1.2 Tuple与List的区别

1. 可更改与不可更改

通过第6章所学，我们知道如何对列表的元素进行更新。然而，同样的更改在元组是不可行的。 什么意思呢？这就是列表和元组的第一大区别。 在列表类型中，元素的值是可以被更改的；而在元组类型中，元素的之是不可更改的。 因此，我们可以使用元组最为字典的key值，而列表不行。 为了更加直观的体现，让我们用代码尝试一下吧:
列表：

```
fruits = ['apple', 'banana', 'orange', 'peach']
print(fruits)

fruits[1] = 'watermelon' # 更新fruits列表的第2位元素
print(fruits)
```

运行结果如下：

```
['apple', 'banana', 'orange', 'peach']
['apple', 'watermelon', 'orange', 'peach']
```

元组:

```
fruits = ('apple', 'banana', 'orange', 'peach')
fruits[1] = 'watermelon' # 尝试更新fruits元组的第2位元素
```

运行结果如下:

```
Traceback (most recent call last):
  File "try.py", line 4, in <module>
      fruits[1] = 'watermelon'
      TypeError: 'tuple' object does not support item assignment

      shell returned 1
```

很明显，Python解释器不允许我们这样做。它也给出了明显的提醒: TypeError: 'tuple' object does not support item assignment.

不光是元组的元素不可更改，它的长度同样也是不可更改的。 Tuple类甚至都没有给出增加/减少元素的函数:

```
# continues from the last code block
fruits.append('kiwi')
```

运行结果如下:

```
Traceback (most recent call last):
  File "try.py", line 3, in <module>
      fruits.append('kiwi')
      AttributeError: 'tuple' object has no attribute 'append'

      shell returned 1
```

2. 复制与引用

在第6章中，我们学习过函数传递值和传递引用的区别。 同样的思想也在列表与元组的区别中有所体现。 因为元组类型的不可更改性，当你使用tuple(tup_name)函数进行类型转换是，返回值是原元组的引用。 而列表具有可更改该性，list(list_name)的返回值是被复制产生的新列表。

```
fruit_tuple = ('apple', 'banana', 'orange') # a tuple of fruits
fruit_list = ['apple', 'banana', 'orange']  # a list of fruits

# test for tuple
tuple2 = tuple(fruit_tuple)
print(tuple2 is fruit_tuple)

# test for list
list2 = list(fruit_list)
print(list2 is fruit_list)
```

运行结果如下:

```
True
False
```

3. 空间占用

由于元组类型的不可能改性，Python解释器会为其分配更小的内存占用。 因此，当元组和列表包含通函的元素时，元组占用的空间会更小一些。

```
# continues from the last code block
print(fruit_tuple.__sizeof__())
print(fruit_list.__sizeof__())
```

运行结果如下:

```
48
64
```

综上所述，我们在编写程序时，根据需求来选择使用元组或者列表。 如果无需更改内容，或者需要使内容不可更改，我们会选择使用元组。 比如传递游戏屏幕上的坐标，使用一种颜色，等等。 而当我们需要对内容进行更改是，我们只能选择列表。 比如游戏棋盘。

7.2 Pygame 通过鼠标的人机交互

在上一章，我们学习了Pygame模块的基础功能，比如设置游戏屏幕，键盘交互，绘制图形等。 在这一章，因为扫雷不能用只用键盘来玩，我们将要学习如何通过鼠标点击与计算机交互。 在讨论通过鼠标的Pygame人机交互以前，我们先来认识一下常规鼠标的构造。

从图片可以看出，普通鼠标的正面（上面）有三个按键。从左到右，把它们标号为1,2,3。

键位	代号
左键	# 1
滚轮键	# 2
右键	# 3

为什么要定义这三个常用按键的代号呢？因为我们在Pygame中就是以这三个代号称呼他们的 在扫雷游戏中，我们需要用到鼠标的左键和右键，也就是1和3。在取得鼠标信息的过程中， 需要注意到的Pygame功能有2点:

- pygame.MOUSEBUTTONUP
- pygame.mouse.get_pos()

```
# 首先定义左键和右键，使得代码可读性更高
LEFT = 1
RIGHT = 3

# 引用第6章中获取状态的代码
for event in pygame.event.get():
    pos = pygame.mouse.get_pos()
    print(pos)  # 打印鼠标点击在游戏屏幕上的坐标
    if event.button == LEFT:
        print("Left Click")
    elif event.button == RIGHT:
        print("Right Click")
```

7.3 JSON

JSON全称叫做JavaScript Object Notation，是一种面对对象的通用表示方法。 JSON这种表示方法具备什么样的特性呢？

- 轻量级
  JSON文件本质是文本文件，占用极小的储存空间，读取时占用内存很小
- 易懂
  JSON文件格式可以很好的描述面对对象的程序设计理念，更加直观的说明程序设计API
- 独立于编程语言
  尽管JSON叫做JavaScript Object Notation，但其实并非只能用于JavaScript。 大多数流行的编程语言都有自己的JSON处理模块， 比如Python的json模块，C++的Json::Value库

由于这三个特性，JSON经常被用于软件的设置文件，比如笔者常用的Visual Studio Code和Microsoft Terminal;以及客户端与服务器之间的通讯，比如C++的jsoncpprpc库。

7.3.1 JSON语法

JSON文件有至少组key-value pair组成。 其中的key指一个对象的名字，它必须是字符串。 其中的value值对象的值，它可以使任何类型。 如果value是不同类型，我们直接写出它就行。 如果value是列表，我们需要把元素写在[]中，与Python一样。 如果value是一个对象，我们需要把它写在{}中。

如果JSON包含多组key-value pair的话，我们需要用','将他们分隔开。 最后一组key-value pair后面不能有','。下面会展示一个例子，使用JSON来描述一个课程。

```
// course.json
{
    "name": "Python Programming in Games",
    "id": "123456",
```

```
    "chapters": [
        // ...
        "chapter 6": "Connect 4",
        "chapter 7": "Minesweeper"
        // .. many more chapters
    ],
    "this chapter": {
        "number": 7,
        "name": "Minesweeper",
        "subsections": [
            "7.1": "Tuple",
            "7.2": "JSON",
            "7.3": "Regular Expression",
            "7.4": "DFS in 2D Array",
            "7.5": "Minesweeper Program"
        ]
    }
}
```

这段JSON所表示的对面在Python是如何体现的呢？

```
class Chapter:
    def __init__(self, number, name, subsections):
        # number: int, name: string, subsections: List[string]
        self.chapNumber = number
        self.name = name
        self.subsections = subsections;

class Course:
    def __init__(self, name, courseID, chapters, thisChapter):
        # name: string, courseID: string, chapters: List[string],
thisChapter: Chapter
        self.name = name
        self.courseID = courseID
        self.chapters = chapters
        self.thisChapter = thisChapter

thisChapter = Chapter(7, 'Minesweeper',
                      ['Tuple', 'JSON', 'Regular Expression', \
                      'DFS in 2D Array', 'Minesweeper Program'])

thisCourse = Course('Python Programming in Games', '123456', \
                      [..., 'Connect 4', 'Minesweeper', ...], \
                      thisChapter)
```

7.3.2 JSON在设置文件中的应用

在这个小结，我们将阅读Windows Terminal设置文件用的一个节选片段。 然后解释这个设置文件中每组key-value pair的意义。解释见代码注释。

```
{
    // Add custom color schemes to this array.
    // To learn more about color schemes, visit https://aka.ms/terminal-
color-schemes
    "scheme":
    [
        {
            "name": "Breeze",
            "black": "#12181d", // a color in rgb hexadecimal format
            "red": "#eb5b5b",
            "green": "#bfeea4",
            "yellow": "#fc8162",
            "blue": "#6eb7eb",
            "purple": "#9b59b6",
            "cyan": "#1abc9c",
            "white": "#eff0f1",
            "brightBlack": "#d8e3e4",
            "brightRed": "#c0392b",
            "brightGreen": "#1cdc9a",
            "brightYellow": "#fdbc4b",
            "brightBlue": "#3daee9",
            "brightPurple": "#57b4df",
            "brightCyan": "#62dd69",
            "brightWhite": "#fcfcfc",
            "background": "#31363b",
            "foreground": "#eff0f1"
        }
    ],

    // Add custom keybindings to this array.
    // To unbind a key combination from your defaults.json, set the command
to "unbound".
    // To learn more about keybindings, visit https://aka.ms/terminal-
keybindings
    "keybindings":
    [
        // Copy and paste are bound to Ctrl+Shift+C and Ctrl+Shift+V in
your defaults.json.
        // These two lines additionally bind them to Ctrl+C and Ctrl+V.
        // To learn more about selection, visit https://aka.ms/terminal-
selection
        { "command": {"action": "copy", "singleLine": false }, "keys":
"ctrl+c" },
        { "command": "paste", "keys": "ctrl+v" },

        // Press Ctrl+Shift+F to open the search box
        { "command": "find", "keys": "ctrl+shift+f" },

        // Press ctrl+shift+. to open a new pane.
        // - "split": "auto" makes this pane open in the direction that
provides the most surface area.
        // - "splitMode": "duplicate" makes the new pane use the focused
pane's profile.
```

```
        // To learn more about panes, visit https://aka.ms/terminal-panes
        { "command": "splitPane", "keys": "ctrl+shift+." },
        { "command": "closePane", "keys": "ctrl+shift+,"},
        { "command": "newTab", "keys": "ctrl+shift+a" },
        { "command": "closeTab", "keys": "ctrl+shift+z"},
        { "command": "nextTab", "keys": "ctrl+tab" }
    ]
}
```

7.3.3 编写Minesweeper游戏的JSON设置文件

了解完JSON文件的写法和作用后，我们就需要动手设计Minesweeper游戏的设置文件啦。 在开始编写之前，要首先搞清楚游戏的基础设计:

1. 游戏屏幕为一个正方向，我们需要size代表游戏屏幕的宽度和高度
2. 地雷数量。拥有两种模式:
    1. 如果地雷数量是一个整数，则设置这么多地雷
    2. 如果地雷数量是default，游戏将会根据size和level自行计算地雷数量
3. 因为2.2，我们需要level
4. 颜色设置
    - 游戏屏幕的背景颜色
    - 地雷被引爆后的颜色
    - 以证实为安全地区的颜色
    - 右键插的旗子颜色
    - 文字(数字)的颜色
5. 字体
    - 数字的字体
    - 游戏结束后弹出屏幕的字体
6. 玩家的昵称

在开始编写JSON配置文件之前，我们还需要知道Pygame模块都支持哪些字体。 我们可以通过指令行来获取这些字体的名称，在这里以Linux环境下bash举例:

```
$ python3 > pygameFonts.txt
Python 3.8.2 (default, Apr 27 2020, 15:53:34)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import pygame
pygame 1.9.6
Hello from the pygame community. https://www.pygame.org/contribute.html
>>> pygame.init()
>>> print(pygame.font.get_fonts())
```

然后查看pygameFonts.txt文件就可以知道哪些字体可以使用了。 接着根据获取到的字体编写JSON设置文件，setting.json:

```
{
    "size": 15,
    "number of mine": "default",
    "level": 2,
    "color": {
        "background": "#cd5c5c",
        "boom": "#e74c3c",
        "revealed_blank": "#eaeded",
        "flag": "#5b2c6f",
        "text": "#000000"
    },
    "font": {
        "number_font": "arialblack",
        "text_font": "lucidaconsole"
    },
    "player": "EEEH"
}
```

7.3.4 Python中的json模块

json模块的初步运用其实很简单，只需要遵循一下一个步骤:

1. 打开setting.json文件
2. 根据setting.json文件创建json对象
3. 根据key读取需要的value

```
# functions.py
import json

# load setting
settingFile = open("settings.json")
setting = json.load(settingFile)

# read values based on key-value pair
boomColor = setting["color"]["boom"]
blankColor = setting["color"]["revealed_blank"]
flagColor = setting["color"]["flag"]
numFont = setting["font"]["number_font"]
size = setting["size"]

# print variables out to see the result
print(boomColor)
print(blankColor)
print(flagColor)
print(numFont)
print(size)
```

运行结果:

```
#e74c3c
#eaeded
#5b2c6f
arialblack
15
```

7.4 正则表达式 Regular Expression

*正则表达式*是用于表示文本规律的一种符号标记。 正则表达式使用符号来描述语言，起到具有规律匹配的功能。 在很多常用的文本编辑器中，都存在使用正则表达式的搜索匹配方式。 很多常用的指令工具同样支持正则表达式 比如常用的ls(*list*)和grep(***g****lobal* ***r****gular* ***e****xpression* ***p****rint*) 多数常用的编程语言同样支持正则表达式的表示方法。 比如C++的<regex>库，Python的re模块，和Java的java.util.regex。 函数式编程语言Perl甚至内建了对正则表达式的支持。 不过正则表达式在这些不同编程语言的库中，语法可能略有不同。 所以在使用不同语言的正则库前最后先看一下官方文档。 在这章的讲解中，7.4.1 数据搜索将使用POSIX标准的正则表达式； 而7.4.2 re模块将以Python的官方文档为准。

7.4.1 正则表达式的用法: 数据搜索(grep)

注: 本小节以bash为标准
在指令行，最经常与正则表达式一起使用的软件是grep。 根据grep的全称global regular expression print，可以清晰看出它与正则表达式关系密切。 grep的功能就是根据正则表达式在一个或多个文件中进行搜索，然后以标准输出展示搜索结果。
根据grep的manual文档，看一看它的主要使用方法

```
$ man grep
GREP(1)                                                                 User
Commands

NAME
       grep, egrep, fgrep, rgrep - print lines that match patterns

SYNOPSIS
       grep [OPTION...] PATTERNS [FILE...]
        ...

DESCRIPTION
    ...
```

在Synopsis一栏的第一行写出了grep的主要用法。 这个PATTERNS代表的就是正则表达式。 正如前文所言，正则表达式起到具有规律匹配的功能。
在了解了grep程序后，是时候知道如何使用正则表达式与grep配合进行搜索了。 接下来，我们将会看看如何使用正则表达是的符号来代表文本

‘任意’字符

正则表达式中最常用的符号就是'.'符号，也称英文的句号，就是所谓的‘任意’字符。 '.'符合可以匹配ASCII中的任意字符。 假设有一个叫做"abc.txt"的文档, `grep ".abc" abc.txt`的作用就是在abc.txt这个文档中搜索所

用匹配正则表达式`.abc`的行。

```
$ cat abc.txt
1abc
2abc
3aac
4cca
5abc
6aabc
$ grep ".abc" acb.txt
1abc
2abc
5abc
6aabc
```

定性字符

正则表达是中使用'^'和'$'作为定性字符。 '^'表示这个正则表达式必须出现在一行的最开始。 '$'表示这个这则表达是必须出现在一行的最后。

```
$ echo "ccaaac" >> abc.txt
$ echo "bacaac" >> abc.txt
$ grep "abc$" abc.txt
1abc
2abc
5abc
6aabc
$ grep "^abc" abc.txt
```

可以看出，`grep "abc$" abc.txt`可以帮我们找出abc.txt中所有以"abc"结尾的行。 而在执行`grep "^abc" abc.txt`时，grep会搜索abc.txt中所以"abc"开始的行。 因为abc.txt文件中名没有以"abc"开始的行，所以没有标准输出的搜索结果。 如果我们查看一下这次运行grep的返回值，会发现是1，表示失败。

```
$ echo $?
1
```

括号表示法

我们也可以用这则表达是的括号表示法来达到匹配的目的。 在正则表达式中，一组[]表示一个字符。 而[]里面的内容表示这个括号可以有的选项。

```
$ grep "[ca$]" abc.txt
1abc
2abc
3aac
```

```
4cca
5abc
6aabc
ccaaac
bacaac
```

如你所见，`grep "[ca$]" abc.txt`可以找出abc.txt文件中所有以a字符或者c字符结尾的行。 我们再来看一个搜索范围更加精确的例子

```
$ grep "[bc]a$" abc.txt
4cca
```

这次的搜索只有一个匹配了。 因为`"[bc]a$"`仅可以匹配abc.txt中"ba"或者"ca"结尾的行。

字符类型

正则表达式同样支持按照字符类型进行匹配。 假设我们需要搜索abc.txt中所有以数字开头的行，按照普通的括号表示法该如何表示呢？ 如果只用普通的括号表示法，我们需要在[]中写出0-9所有的数字字符，来达到匹配数字的目的。

```
$ grep "^[1234567890]" abc.txt
1abc
2abc
3aac
4cca
5abc
6aabc
$ grep "^[1234567890]abc" abc.txt
1abc
2abc
5abc
```

但这样做实在是太麻烦了。 使用正则表达式的目的是快速方便的匹配内容，而不是给自己找麻烦。 因此我们可以使用正则表达式字符类型功能。 在正则表达式中，`[0-9]`代表数字类型，可以用来代替`[0123456789]`。 可以看出，这样的正则表达式效果是一样的。

```
$ grep "^[0-9]" abc.txt
1abc
2abc
3aac
4cca
5abc
6aabc
$ grep "^[0-9]abc" abc.txt
1abc
```

```
2abc
5abc
```

除了[0-9]表示全体数字字符之外，类型表达法还可以遵从自定义的区间。 比如[0-3]就可以表0,1,2,3这四个数字字符。

```
$ grep "^[0-3]abc" abc.txt
1abc
2abc
```

除了数字类型的字符可以用这种方式来表示外，字母类型也是可以的。 在正则表达式中，[a-z]用于表示小写字母。 [A-Z]用于表示大写字母。

```
$ cat 123.txt
a123
b123
c345
d567
lxyz
f555

A123
B123
C345
D567
Lxyz
F555
$ grep "^[a-z]" 123.txt
a123
b123
c345
d567
lxyz
f555
$ grep "^[A-Z]" 123.txt
A123
B123
C345
D567
Lxyz
F555
```

从例子中可以看出，^[a-z]可以比配123.txt中所有以小写字母开头的行。 而^[A-Z]则可以匹配123.txt中所有以大写字母开头的行。
现在我们知道如何用正则表达式来表示小写字母和大写字母了。 但感觉表示全体字母的时候还是有些麻烦。 有没有什么办法可以进行无视大小写的匹配呢?

```
$ grep "^[a-Z]" 123.txt
a123
b123
c345
d567
lxyz
f555
A123
B123
C345
D567
Lxyz
F555
```

答案是可以的。正则表达式支持通过`[a-Z]`来表示全体字母字符。 但是反过来`[A-z]`就不行了。

```
$ grep "^[A-z]" 123.txt
$ echo $?
2
```

量词

我们都知道什么是量词：用于表示数量的词语。 在任何一种语言的写作中，量词都是不可缺少的。 在正则表达式中也是存在量词的。并且，正则表达式存在多种量词的表示方法。

字符	意义 (匹配前一字符...)
?	0次或1次
*	0次或多次
+	1次或多次
{n}	n次
{n,m}	至少n次，最多m次
{n,}	至少n次
{,m}	最多m次

由此可见，通过正则表达式的语法，我们可以达成多种多样的自由搜索。 在处理文本数据的时候，正则表达式可以极高的提高我们的工作效率。 grep搜索也支持一种POSIX表示法。 POSIX式一种更高级的字符类型表示方法。 由于Python的re模块不支持POSIX，这里不做讲解。

7.4.2 Python的re模块

Python的re模块允许我们根据正则表达式来跟字符串进行比较。 re模块所使用的正则表达式语法跟7.4.1所讲的grep区别不大。 grep支持的正则表达式可以直接在Python的re模块使用。

findall(pattern: str, text: str) -> List[str]

`findall()`函数可以找到一个字符串中所有与正则表达式匹配的**子字符串**。 `findall()`可以找出text中所有匹配的子字符串，然后以列表形式返回搜索结果。 如果没有找到可以匹配的搜索结果，`findall()`会返回一个空的列表。

```
import re

text = "1abc 2abc 3aac 4cca 5abc 6aabc ccaaac bacaac"
res = re.findall("[0-9]abc", text)
print(res)
```

运行结果

```
['1abc', '2abc', '5abc']
```

如果我们稍微改动一点pattern，运行结果就会有所不同。 因为Python的`findall()`函数是对**子字符串**进行匹配，而非像grep按**行**进行匹配

```
res = re.findall("[0-9]a", text)
print(res)
```

运行结果

```
['1a', '2a', '3a', '5a', '6a']
```

如果要是找不到可以匹配的子字符串，`findall()`就会返回一个空列表。

```
res = re.findall("^[a-z]", text)
print(res)
```

运行结果

```
[]
```

search(pattern: str, text: str) -> match

`search()`函数可以找到text中的所有的匹配的**第一个**子字符串， 并把找到的结果作为一个 **Match object(匹配对象)** 返回。 虽然`search()`的功能看起来和`findall()`很像，但还是有本质去别的。

```
import re
text = "1abc 2abc 3aac 4cca 5abc 6aabc ccaaac bacaac"
res = re.search("[0-9]a", text)
print(res)
```

运行结果：

```
<re.Match object; span=(0, 2), match='1a'>
```

拿到这个搜索结果后，我们可以对它进行几个操作

```
print(res.span())
print(res.group())
print(res.string)
```

运行结果：

```
(0, 2)
1a
1abc 2abc 3aac 4cca 5abc 6aabc ccaaac bacaac
```

可以看出，`.span()`的返回值是一个元组，他代表搜索结果的(开始位置，结束位置)。 `.group()`则是可以取出这个匹配结果到底是什么子字符串，对应匹配对象的'match'。 `.string`的作用是查看原字符串。

split(pattern: str, text: str, maxsplit=0) -> List[str]

`split()`函数可以根据正则表达式，把text分裂成一个列表。

```
import re
text = "1abc 2abc 3aac 4cca 5abc 6aabc ccaaac bacaac"
res = re.split(" ", text)
print(res)
```

运行结果

```
['1abc', '2abc', '3aac', '4cca', '5abc', '6aabc', 'ccaaac', 'bacaac']
```

`re.split(" ", text)`把text这个字符串，从每个`" "`分开，以一个列表的形式返回。 如果我们不需要把字符串分割成这么多分怎么办呢？这时候就需要给`split()`在传递一个maxsplit参数了。 如果我们不传maxsplit参数的话，`split()`默认把字符串尽可能多的分割。 而如果我们传递这个参数了的话，`split()`就会按照我

们传递的参数进行分割了。 如果maxsplit = 1的话，意思式只在第一次搜索结果进行分割。 如果maxsplit = 2的话，意思是在第一次和第二次搜索结果的地方进行分割。其他数字以此类推。

```
res = re.split(" ", text, 1)
print(res)
res = re.split(" ", text, 2)
print(res)
res = re.split(" ", text, 3)
print(res)
```

运行结果

```
['1abc', '2abc 3aac 4cca 5abc 6aabc ccaaac bacaac']
['1abc', '2abc', '3aac 4cca 5abc 6aabc ccaaac bacaac']
['1abc', '2abc', '3aac', '4cca 5abc 6aabc ccaaac bacaac']
```

然后看看基于正则表达式的替换效果怎么样:

```
res = re.split("a[bc]", text)
print(res)
```

运行结果

```
['1', 'c 2', 'c 3a', ' 4cca 5', 'c 6a', 'c ccaa', ' b', 'a', '']
```

sub(patter: str, repl: str, text: str, count=0)

sub()函数用于替换文本内容。 这个函数会根据pattern实在text中搜索，并且把匹配项替换成repl的。 我们用sub()函数把文件中所有的" "替换成"-":

```
import re
text = "1abc 2abc 3aac 4cca 5abc 6aabc ccaaac bacaac"
res = re.sub(" ", "-", text)
print(text)
print(res)
```

运行结果

```
1abc 2abc 3aac 4cca 5abc 6aabc ccaaac bacaac
1abc-2abc-3aac-4cca-5abc-6aabc-ccaaac-bacaac
```

与`split()`类似，`sub()`也允许我们传入一个count参数。 count参数的作用式告诉`sub()`我们想要替换多少个匹配项，如果不穿这个参数的话，默认为替换所有的匹配项。

```
print(text)
res = re.sub(" ", "*", text, 1)
print(res)
res = re.sub(" ", "*", text, 2)
print(res)
res = re.sub(" ", "*", text, 4)
print(res)
```

运行结果

```
1abc 2abc 3aac 4cca 5abc 6aabc ccaaac bacaac
1abc*2abc 3aac 4cca 5abc 6aabc ccaaac bacaac
1abc*2abc*3aac 4cca 5abc 6aabc ccaaac bacaac
1abc*2abc*3aac*4cca*5abc 6aabc ccaaac bacaac
```

同样进行一次基于正则表达式的替换

```
text = "1abc 2abc 3aac 4cca 5abc 6aabc ccaaac bacaac"
res = re.sub("a[bc]", "*", text)
print(text)
print(res)
```

运行结果:

```
1abc 2abc 3aac 4cca 5abc 6aabc ccaaac bacaac
1*c 2*c 3a* 4cca 5*c 6a*c ccaa* b*a*
```

7.5 二维数组的深度优先搜索

7.6 Minesweeper 游戏编写