简介

对字符串进行操作的需求很多，只凭索引切片、对象方法等方式有时是不够用的（比如：判断一个字符串是否是有效的 email 地址），此时可以考虑使用正则表达式。

```
import re


pattern = r'([A-Za-z0-9]+[-_])*[A-Za-z0-9]+@[A-Za-z0-9-]+(\.[A-Z|a-z]{2,})+'
p = re.compile(pattern)

def isValid(email):
    if p.fullmatch(email):
        print("有效的email地址")
    else:
        print("无效的email地址")
```

正则表达式（Regular Expression），是对字符串操作的一种"逻辑公式"，它是由一些特定的字符组成的一个"匹配规则"，可以对要匹配的字符串指定该"规则"。

许多语言都支持利用正则表达式进行字符串操作，Python也不例外，通过内置的 re 模块实现。

字符匹配

匹配机制：对要匹配的字符串的元素挨个判断是否与 "规则" 匹配。

书写注意：正则表达式是字符串，所以书写时通常不要随意加空格。

反斜杠问题：因为正则表达式和Python字符串都使用反斜杠字符来转义，有时就需要使用双倍的反斜杠才能达到想要的效果，而这很麻烦，所以强烈建议大家在写正则表达式时，用原始字符串

正则表达式可以包含普通字符或者特殊字符（元字符）

普通字符

- 大多数字母和字符一般都会和自身匹配

```
import re



p = re.compile(r"test123")
print(p.search("atest123b"))
```

特殊字符（元字符）

- 有些字符它们和自身并不匹配，而是匹配一些与众不同的东西或者影响正则表达式的其他部分（对其重复或改变含义）
- 元字符：`.  ^  $  *  +  ?  {  }  [  ]  \  |  (  )`

`.`

- 匹配除了换行符以外的任意一个字符
- DOTALL 模式下，它将匹配包括换行符的任意一个字符

```
import re


p = re.compile(r".")
print(p.match("abc"))
print(p.match("9bc"))
print(p.match("@bc"))
print(p.match(".bc"))
print(p.match("\tbc"))
print(p.match("\nbc"))

p = re.compile(r".", flags=re.DOTALL)
print(p.match("\nbc"))
```

`^`

- 匹配字符串的开头
- MULTILINE 模式下，还会继续匹配换行后的开头

```
import re


p = re.compile(r"^ab")
print(p.findall("abcd\nabfg"))

p = re.compile(r"^ab", flags=re.MULTILINE)
print(p.findall("abcd\nabfg"))
```

`$`

- 匹配字符串的末尾 或者 匹配在字符串结尾的换行符之前的末尾
- MULTILINE 模式下，还会匹配换行符之前的末尾（换行符可以不在字符串末尾）

```
import re


p = re.compile(r"cd$")
```

```
print(p.findall("abcd\n"))


p = re.compile(r"cd$", flags=re.MULTILINE)
print(p.findall("abcd\nefcd"))


""" 会找到两个（空的）匹配：一个在换行符之前，一个在字符串的末尾 """
p = re.compile(r"$")
print(p.findall("abcd\n"))
```

*

- 对它前面的正则表达式匹配0到任意次重复， 尽量多的匹配（贪婪）

```
import re


p = re.compile(r"ab*")
print(p.search("a"))
print(p.search("ab"))
print(p.search("abb"))
print(p.search("abbbc"))
```

+

- 对它前面的正则表达式匹配1到任意次重复， 尽量多的匹配（贪婪）

```
import re


p = re.compile(r"ab+")
print(p.search("a"))
print(p.search("ab"))
print(p.search("abb"))
print(p.search("abbbc"))
```

`?`

- 对它前面的正则表达式匹配0到1次， 尽量多的匹配（贪婪）

```
import re


p = re.compile(r"ab?")
print(p.search("a"))
print(p.search("ab"))
print(p.search("abb"))
print(p.search("abbbc"))
```

`*?` `+?` `??`

- `*` `+` `?` 都是贪婪的，它们对字符串进行尽可能多的匹配，有时候并不需要这种行为，可以在之后添加 `?`，就可以以非贪婪的方式进行匹配，则尽可能少的字符将会被匹配

```
import re


p = re.compile(r'<.*>')
print(p.search('<a> b <c>'))

p = re.compile(r'<.*?>')
print(p.search('<a> b <c>'))

p = re.compile(r"ab+?")
print(p.search("abbbc"))

p = re.compile(r"ab??")
print(p.search("abc"))
```

`{m}`

- 对其之前的正则表达式指定匹配 m 个重复

```
import re


p = re.compile(r"ab{2}")
print(p.search("abc"))
print(p.search("abbc"))
print(p.search("abbbc"))
```

`{m,n}`

- 对其之前的正则表达式进行 *m* 到 *n* 次匹配，在 *m* 和 *n* 之间取尽量多（贪婪方式）
- 忽略 *m* 则下限默认为 0，忽略 *n* 则上限默认为无限次（逗号不能省略）

```
import re


p = re.compile(r"ab{2,4}")
print(p.search("abc"))
print(p.search("abbc"))
print(p.search("abbbc"))
print(p.search("abbbbc"))
print(p.search("abbbbbc"))

p = re.compile(r"ab{,4}")
print(p.search("ac"))
print(p.search("abc"))

p = re.compile(r"ab{2,}")
print(p.search("abbbbc"))
print(p.search("abbbbbc"))
```

`{m,n}?`

- 上面 `{m,n}` 的非贪婪模式

```
import re
```

```
p = re.compile(r"ab{2,4}?")
print(p.search("abc"))
print(p.search("abbc"))
print(p.search("abbbc"))
print(p.search("abbbbc"))
print(p.search("abbbbbc"))

p = re.compile(r"ab{,4}?")
print(p.search("ac"))
print(p.search("abc"))

p = re.compile(r"ab{2,}?")
print(p.search("abbbbc"))
print(p.search("abbbbbc"))
```

`|`

- 任意个正则表达式可以用 `|` 连接，比如 `A|B` 表示匹配正则表达式 A 或者 B，一旦有一个先匹配成功，另外的就不会再进行匹配，`|` 绝不贪婪

```
import re


p = re.compile(r"d|e|b")
print(p.search("abc"))
print(p.search("aebcd"))
```

`\`

- 转义特殊字符（元字符）

```
import re


# 只匹配*号
p = re.compile(r"\*")
print(p.fullmatch("*"))
```

```
# 只匹配+号
p = re.compile(r"\+")
print(p.fullmatch("+"))

# 只匹配?号
p = re.compile(r"\?")
print(p.fullmatch("?"))
```

- 用来表示一个特殊序列（由 `\`和一个字符组成的特殊序列）

  `\number`

  匹配数字代表的分组里面的内容（每个括号是一个子组，子组从1开始编号），在 `[` 和 `]` 字符集内，任何数字转义都被看作是字符

  ```
  import re


  """ \1匹配的内容和第1组一定一样 """
  p = re.compile(r"(.+) \1")
  print(p.search("ab abc"))
  print(p.search("5 5"))

  """ 两个组匹配的内容不一定一样 """
  p = re.compile(r"(.+) (.+)")
  print(p.search("ab abc"))
  print(p.search("5 5"))
  ```

  `\A`

  匹配字符串的开头，类似于 `^`，区别在于：MULTILINE模式下，`\A` 不识别换行

  ```
  import re


  p = re.compile(r"^ab")
  print(p.findall("abcd\nabfg"))

  p = re.compile(r"^ab", flags=re.MULTILINE)
  ```

```
print(p.findall("abcd\nabfg"))

p = re.compile(r"\Aab")
print(p.findall("abcd\nabfg"))

p = re.compile(r"\Aab", flags=re.MULTILINE)
print(p.findall("abcd\nabfg"))
```

`\b`

匹配空字符串，但只在单词开始或结尾的位置，即匹配一个单词边界

```
import re


p = re.compile(r"er\b")
print(p.search("never"))
print(p.search("verb"))

p = re.compile(r"\ba\b")
print(p.search("I have a dog"))
```

`\B`

匹配空字符串，但不能在单词开始或结尾的位置，即匹配非单词边界

```
import re


p = re.compile(r"er\B")
print(p.search("never"))
print(p.search("verb"))

p = re.compile(r"\Ba\B")
print(p.search("I have a dog"))
```

`\d`

匹配任何一个十进制数字，等价于 `[0-9]`

```
import re


p = re.compile(r"\d")
print(p.search("a1234b"))

p = re.compile(r"\d+")
print(p.search("a1234b"))
```

`\D`

匹配任何一个非数字字符，等价于 `[^0-9]`

```
import re


p = re.compile(r"\D")
print(p.search("ab1234c"))

p = re.compile(r"\D+")
print(p.search("ab1234c"))
```

`\s`

匹配任何一个空白字符

```
import re


p = re.compile(r"a\sb")
print(p.search("adb a bc"))
```

`\S`

匹配任何一个非空白字符

```
import re


p = re.compile(r"a\Sb")
print(p.search("adb a bc"))
```

`\w`

匹配一个字母或一个数字或一个下划线，等价于 `[a-zA-Z0-9_]`

```
import re


p = re.compile(r"a\wb")
print(p.findall("adba9ba_ba b"))
```

`\W`

匹配一个非字母非数字非下划线的字符，等价于 `[^a-zA-Z0-9_]`

```
import re


p = re.compile(r"a\Wb")
print(p.findall("adba9ba_ba b"))
```

`\Z`

只匹配字符串的末尾，且 MULTILINE 模式下，`\Z` 不识别换行

```
import re


p = re.compile(r"cd\Z")
print(p.findall("abcd"))

""" 结尾是'\n'，不是'cd' """
print(p.findall("abcd\n"))

""" MULTILINE 模式下，\Z 不识别换行 """
```

```
p2 = re.compile(r"cd\Z", flags=re.MULTILINE)
print(p2.findall("abcd\nef"))

""" 只会找到一个（空的）匹配 """
p = re.compile(r"\Z")
print(p.findall("abcd\n"))
```

`\n` `\t` `\\` `\'` `\"`

绝大部分Python的标准转义字符也被正则表达式分析器支持

```
import re


p = re.compile(r"\n")
print(p.findall("\n"))

p = re.compile(r"\t")
print(p.findall("\t"))

p = re.compile(r"\\")
print(p.findall("\\"))

p = re.compile(r"\'")
print(p.findall("'"))

p = re.compile(r"\"")
print(p.findall('"'))
```

`[]`

用于表示一个字符集：

- 字符可以单独列出，比如 `[amk]` 匹配 `'a'`， `'m'`， 或者 `'k'`

```
import re


p = re.compile(r"[amk]")
print(p.findall("I have a monkey"))
```

- 可以表示字符范围，通过用 `-` 将两个字符连起来

```
import re


p = re.compile(r"[a-y]")
print(p.findall("ahzyqAHZYQ"))

p = re.compile(r"[0-5][A-Y]")
print(p.findall("a0hzyq125A6HZYQ"))
```

- 特殊字符在字符集中，失去它的特殊含义

```
import re


p = re.compile(r"[.+]")
print(p.findall("abc"))

p = re.compile(r"[.+]")
print(p.findall("a.b+c.d+"))
```

- 特殊序列，如 `\d` `\s` `\w` 在集合内可以被接受

```
import re


p = re.compile(r"[\d]")
print(p.search("a1234b"))

p = re.compile(r"[\d+]")
```

```
print(p.findall("a1234b+"))

p = re.compile(r"[a\sb]")
print(p.findall("adb a bc"))

p = re.compile(r"[\w]")
print(p.findall("adb_a b!c"))
```

- 不在字符集范围内的字符可以通过取反来进行匹配，如果字符集首字符是 `^`，所有不在字符集内的字符将会被匹配， `^` 如果不在字符集首位，就没有特殊含义

```
import re


p = re.compile(r"[^5]")
print(p.findall("5a b512!5"))

p = re.compile(r"[^^]")
print(p.findall("5a^b512!5"))
```

- 如果要匹配 `'['` `']'`，可以在它之前加上反斜杠

```
import re


p = re.compile(r"[\[\]]")
print(p.findall("[]"))
```

`(...)`

- 捕获分组，匹配括号内的任意正则表达式，并标识出该分组的开始和结尾。
- 组从 0 开始编号，组 0 始终存在，它表示整个正则，所以Match的对象方法都将组 0 作为默认参数；子组从左到右编号，从 1 向上编号
- 分组匹配的内容可以在之后其他分组用 `\number` 进行再次引用
- 要匹配字符 `(` 或者 `)`，用 `\(` 或 `\)`，或者把它们包含在字符集里： `[(]`， `[)]`

```
```

```
import re


p = re.compile(r"b(.+)a(.+)e")
m = p.match("babacdefg")
print(m)
print(m.group(1), m.group(2))
print(m.groups())
print(m.span(1), m.span(2))
print(m.start(1), m.end(1))
print(m.start(2), m.end(2))

# 多个分组，返回元组列表
print(p.findall("babacdefg"))

# 引用第1组匹配的内容
p = re.compile(r"b(.+)a(\1)e")
print(p.findall("babaabefg"))
```

`(?:…)`

- 非捕获分组，并不创建新的组合，所匹配的子字符串不能在执行匹配后被获取或是之后在模式中被引用

```
import re


p = re.compile(r"b(?:.+)a(?:.+)e")
m = p.match("babacdefg")
print(m)
```

`(?=...)`

- 前向肯定界定符，并不创建新的组合，且括号里的正则表达式匹配成功时才能继续匹配，否则整个匹配失败

```python
import re

"""
第一步：    .+[.] 匹配成功
第二步：    前向肯定界定符匹配成功才继续第三步，否则匹配失败
第三步：    .+ 接着第一步继续匹配
最后结果为第一步和第三步匹配的结果 """
p = re.compile(r".+[.](?=exe$).+")
m = p.match("ab.exe")  # 文件名必须以exe为后缀
print(m)
```

`(?!...)`

- 前向否定界定符，并不创建新的组合，且括号里的正则表达式匹配失败时才能继续匹配，否则整个匹配失败

```python
import re

"""
第一步：    .+[.] 匹配成功
第二步：    前向否定界定符匹配失败才继续第三步，否则匹配失败
第三步：    .+ 接着第一步继续匹配
最后结果为第一步和第三步匹配的结果 """
p = re.compile(r".+[.](?!exe$|txt$).+")
m = p.match("ab.txt")  # 文件名不能以exe或txt为后缀
print(m)
```

使用正则表达式

编译正则表达式

re.compile(pattern, flags=0)

- pattern：正则表达式
- flags：标志，用于控制正则表达式的匹配方式
- 编译一个正则表达式，返回一个Pattern类的实例对象

```
import re


p = re.compile('ab*', flags=0)
print(p)
print(isinstance(p, re.Pattern))
```

执行匹配

编译正则表达式得到 Pattern 实例对象，然后通过调用该对象的不同方法来执行匹配

Pattern 实例对象支持以下方法：

Pattern.search(string[, pos[, endpos]])

- string：要匹配的字符串
- pos：匹配的起始位置，默认为0
- endpos：匹配的结束位置，默认为字符串长度
- 扫描整个字符串，寻找第一个成功的匹配，返回Match类的实例对象（该实例对象包含匹配相关的信息：起始和结束位置、匹配的子串等等）；如果没有匹配，则返回 None

```
import re


p = re.compile('og')
m = p.search("dog")
print(m)
print(isinstance(m, re.Match))

print(p.search("dog", 2))
print(p.search("dog", endpos=2))
```

Pattern.match(string[, pos[, endpos]])

- string：要匹配的字符串
- pos：匹配的起始位置，默认为0
- endpos：匹配的结束位置，默认为字符串长度
- 当字符串的起始位置匹配成功，返回Match类的实例对象（该实例对象包含匹配相关的信息：起始和结束位置、匹配的子串等等）；如果起始位置没有匹配，则返回 None

```
import re


p = re.compile('og')

print(p.match("dog"))
print(p.search("dog", 1))
```

Pattern.fullmatch(string[, pos[, endpos]])

- string：要匹配的字符串
- pos：匹配的起始位置，默认为0
- endpos：匹配的结束位置，默认为字符串长度
- 当整个字符串都匹配成功，返回Match类的实例对象（该实例对象包含匹配相关的信息：起始和结束位置、匹配的子串等等），否则返回 None

```
import re


p = re.compile('o[gh]')

print(p.fullmatch("ogh"))
print(p.fullmatch("og"))
print(p.fullmatch("oh"))
print(p.fullmatch("dog"))
print(p.fullmatch("dog", 1))
```

Pattern.findall(string[, pos[, endpos]])

- string：要匹配的字符串
- pos：匹配的起始位置，默认为0
- endpos：匹配的结束位置，默认为字符串长度
- 对字符串从左往右扫描，找到所有不重复匹配，以列表的形式返回（保存子串），如果有多个组（至少两个子组），则返回元组列表，如果没有找到匹配的，则返回空列表
- 会把空字符也参与到匹配中去

```
import re


p = re.compile(r'\d')
print(p.findall("Ten years ago, Three dogs"))
print(p.findall("10 years ago, 3 dogs"))

p = re.compile(r".+")
print(p.findall("abc"))

# ".*"可以表示空，所以会有空匹配
p = re.compile(r".*")
print(p.findall("abc"))

# 多个分组，返回元组列表
p = re.compile(r'(\d+)-(\D)')
print(p.findall("Ten-years ago, Three-dogs"))
print(p.findall("101-years ago, 3-dogs"))
```

```
# 当捕获分组被重复时，组号也重复了，所以后面组的结果会把前面组的结果覆盖
p = re.compile(r"(\d)(\d){2}")  # 等价于 (\d)(\d)(\d) 且后面两个组号都为2
print(p.findall("1234567890"))
```

Pattern.finditer(string[, pos[, endpos]])

- string：要匹配的字符串
- pos：匹配的起始位置，默认为0
- endpos：匹配的结束位置，默认为字符串长度
- 和findall类似，不同在于finditer以迭代器形式返回，保存的是Match类的实例对象

```
import re


p = re.compile(r'\d')
for i in p.finditer("10 years ago, 3 dogs"):
    print(i)
```

Pattern.split(string, maxsplit=0)

- string：要匹配的字符串
- maxsplit：最大分割次数，默认为 0，表示不限制次数
- 按照匹配的子串将字符串分割，以列表形式返回
- 如果有捕获分组，那么分组里匹配的内容也会包含在结果中
- 如果有捕获分组，并且匹配到字符串的开始，那么结果将会以一个空字符串开始。对于结尾也是一样

```
import re


p = re.compile(r"\W+")
print(p.split('Words, words, words.'))

p = re.compile(r"(\W+)")
print(p.split('Words, words, words.'))

print(p.split('...Words, words, words...'))
```

Pattern.sub(repl, string, count=0)

- repl：替换的字符串 或者 函数
- string：要被匹配后替换的字符串
- count：匹配后替换的最大次数，默认 0，表示替换所有的匹配

```
import re


p = re.compile(r'blue|white|red')
""" 把每一个从左开始非重叠匹配的字符串用其他字符串替换 """
print(p.sub('colour', 'blue socks and red shoes'))
print(p.sub('colour', 'blue socks and red shoes', count=1))


def func(matchobj):
    if matchobj.group() == '-':
        return ' '
    return '-'

""" 把每一个从左开始非重叠匹配的对象作为参数传入函数调用
这个函数只能有一个 匹配对象 参数，并返回一个替换字符串 """
p = re.compile(r'-{1,2}')
print(p.sub(func, 'pro----gram-files'))
```

Pattern.subn(repl, string, count=0)

- 行为与 `sub()` 相同，但是返回一个元组 `(字符串，替换次数)`

```
import re


p = re.compile(r'blue|white|red')
print(p.subn('colour', 'blue socks and red shoes'))
print(p.subn('colour', 'blue socks and red shoes', count=1))


def func(matchobj):
    if matchobj.group(0) == '-':
        return ' '
    else:
        return '-'

p = re.compile(r'-{1,2}')
print(p.subn(func, 'pro----gram-files'))
```

Match 实例对象支持以下方法：

Match.group([group1, ...])

- groupN：对应的组号，默认为 0，返回整个匹配结果
- 返回一个或者多个子组的匹配结果，如果有多个参数，结果就是一个元组

```
import re


p = re.compile(r"b(.+)a(.+)e")
m = p.match("babacdefg")
print(m)
print(m.group())
print(m.group(0))
print(m.group(1))
print(m.group(2))
print(m.group(2, 1, 0))
```

Match.groups(default=None)

- 返回一个元组，包含所有子组的匹配结果
- default 参数用于子组不参与匹配的情况

```
import re


p = re.compile(r"b(.+)a(.+)e")
m = p.match("babacdefg")
print(m)
print(m.groups())

# 后面两个组不参与匹配，返回default对应的值
p = re.compile(r"b(.+)(.+)?(.+)?")
m = p.match("babacdefg")
print(m)
print(m.groups())
print(m.groups("no"))
```

Match.start([group])

- 返回对应 group 匹配开始的位置，group默认为 0

Match.end([group])

- 返回对应 group 匹配结束的位置，group默认为 0

```
import re


p = re.compile(r"b(.+)a(.+)e")
m = p.match("babacdefg")
print(m)
print(m.start(), m.end())
print(m.start(1), m.end(1))
print(m.start(2), m.end(2))
```

Match.span([group])

- 返回一个元组，包含 (Match.start([group]), Match.end([group]))，group默认为 0

```
import re



p = re.compile(r"b(.+)a(.+)e")
m = p.match("babacdefg")
print(m)
print(m.span())
print(m.span(0))
print(m.span(1))
print(m.span(2))
```

模块级别函数

如果你不想创建Pattern实例对象并调用其方法，也可以直接使用re模块提供的函数：

re.search(pattern, string, flags=0)

re.match(pattern, string, flags=0)

re.fullmatch(pattern, string, flags=0)

re.findall(pattern, string, flags=0)

re.finditer(pattern, string, flags=0)

re.split(pattern, string, maxsplit=0, flags=0)

re.sub(pattern, repl, string, count=0, flags=0)

re.subn(pattern, repl, string, count=0, flags=0)

编译标志

编译标志可以修改正则表达式的一些匹配方式，它在 re 模块中有两个名称：全名 和 缩写。其中需要掌握的有：

re.I / re.IGNORECASE

- 进行忽略大小写匹配

```
import re


p = re.compile(r"[a-z]+", flags=re.IGNORECASE)
print(p.match("aAbBcC"))
```

re.M / re.MULTILINE

- 多行匹配，影响 ^ 和 $
- 设置以后，`^` 匹配字符串的开始，和每一行的开始；`$` 匹配字符串尾，和每一行的结尾

```
import re


p = re.compile(r"^ab", flags=re.MULTILINE)
print(p.findall("abcd\nabfg"))

p = re.compile(r"cd$", flags=re.MULTILINE)
print(p.findall("abcd\nefcd"))
```

re.S / re.DOTALL

- 使 `.` 匹配包括换行在内的所有字符，没有设置时 `.` 是不能匹配换行符的

```
import re


p1 = re.compile(r".")
p2 = re.compile(r".", flags=re.DOTALL)
print(p1.search("\nbc"))
print(p2.search("\nbc"))
```

re.X / re.VERBOSE

- 允许你编写更具可读性的正则表达式，主要体现在分段、添加注释、空白符号

```
import re


# 等价于 p = re.compile(r"\d+\.\d*")
p = re.compile(r"""
                \d +  # 匹配整数部分，re.X使得该空格不影响
                \.    # 匹配小数点，re.X使得可以分段写
                \d *  # 匹配小数部分，re.X使得该空格不影响
                """, re.X)

print(p.findall("1.2345.78"))
```