1.正则表达式

正则表达式面向什么样的问题？

1.判断一个字符串是否匹配给定的格式（判断是不是邮箱或者电话号码） 数据校验

2，从一个字符串里面根据指定规则提取信息（抓取页面中的链接或者其它信息） 数据提取

2. re模块

正则表达式写出来后需要使用

那么需要使用re模块进行使用，提取及验证等操作

re模块为内置模块

使用时需要导包 ----- import re

常用方法分为：findall， match， search

```
# re.findall() 是 Python 中 re 模块提供的一个函数，用于在字符串中查找所有满足指定正则表达式的子串，并返回一个列表。下面我将详细介绍 re.findall() 的使用方法及其相关参数。
"""
re.findall(pattern, string, flags=0)

pattern 是要匹配的正则表达式；
string 是要在其中进行匹配的字符串；
flags 参数可以指定正则表达式的匹配模式，如是否忽略大小写等。
"""

import re

# 定义正则表达式  '\d+'，它可以匹配一个或多个数字字符
pattern = r'\d+'

# 定义字符串
string = 'The price of the apple is 2 dollars, and the price of the orange is 1 dollar.'

# 使用 findall() 函数查找数字
result = re.findall(pattern, string)

# 输出结果
print(result)
# ['2', '1']

"""
# ====== findall =======
#  re.finall(目标数据，目标字符串)
```

```
#       在目标字符串中找出所有符合目标数据的数据，符合条件的数据放入列表中
#       没有就返回空列表
"""
```

1. re.findall()返回的是一个列表，列表中的每个元素都是字符串类型。如果正则表达式中包含分组，则返回的列表中同样包含分组捕获的内容。
2. 如果正则表达式中包含多个子表达式，则返回的列表中会按照整个正则表达式的优先级顺序排列子表达式的匹配结果。
3. 当正则表达式中包含重复字符集（如 * 或 +）时，返回的是一个包含所有匹配到的子串的列表。如果希望返回所有匹配到的重复字符集中单个重复的内容，可以使用非贪婪模式的量词（如 *? 和 +?）或分组语法。
4. 如果要精确匹配某个字符串，应该使用锚定字 ^ 和 $ 来限定匹配范围。否则可能会匹配到意想不到的内容。

```python
# re.match() 是 Python 中 re 模块提供的一个函数，用于在字符串的开头匹配正则表达式，并返回一个 Match 对象。下面我将详细介绍 re.match() 的使用方法及其相关参数。

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

pattern 是要匹配的正则表达式；
string 是要在其中进行匹配的字符串；
flags 参数可以指定正则表达式的匹配模式，如是否忽略大小写等。
"""

import re

# 定义正则表达式
pattern = r'\d+'

# 定义字符串
string = 'The price of the apple is 2 dollars.'

# 使用 match() 函数查找数字
match_result = re.match(pattern, string)

# 输出匹配结果
if match_result:
    print("匹配成功：", match_result.group())
else:
    print("匹配失败")

# 匹配失败

"""
# ====== match =======

# re.match(pattern, string, flags=0)
#   pattern     匹配的正则表达式
#   string      要匹配的字符串
#   flags       标志符指定，用于控制正则表达式的匹配方式，如：是否区分大小写，多行匹配等等
```

```
#          必须从字符串开头匹配！
#          match方法尝试从字符串的起始位置匹配一个模式，
#          如果不是起始位置匹配成功的话，match()就返回none
# 返回值为对象

# group()：查看匹配字符
# span：查看匹配数据的索引取值区间
"""
```

定义了一个正则表达式 `r'\d+'`，它可以匹配一个或多个数字字符。然后定义了一个字符串 `string`，需要在其中查找与正则表达式匹配的子串。最后使用 `re.match()` 函数在字符串开头查找符合正则表达式规则的子串，并返回一个 `Match` 对象。如果匹配成功，则输出匹配到的结果；否则输出“匹配失败”。

1. `re.match()` 只会匹配到字符串的开头。如果想要在整个字符串中匹配正则表达式，应该使用 `re.search()` 或 `re.findall()`。
2. 如果 `Match` 对象存在，则可以通过调用 `group()` 方法获取匹配到的子串；如果不存在，则说明匹配失败。
3. 在使用正则表达式时，需要根据具体情况考虑各种特殊字符和操作符的含义和使用方式，并进行适当的转义处理或括号分组。

```
# re.search() 是 Python 中 re 模块提供的一个函数，用于在字符串中搜索与正则表达式匹配的子串，并返回一个 Match 对象。
"""
re.search(pattern, string, flags=0)

其中，pattern 是要匹配的正则表达式；
string 是要在其中进行搜索的字符串；
flags 参数可以指定正则表达式的匹配模式，如是否忽略大小写等。
"""

import re

# 定义正则表达式
pattern = r'\d+'

# 定义字符串
string = 'The price of the apple is 2 dollars.'

# 使用 search() 函数查找数字
search_result = re.search(pattern, string)

# 输出匹配结果
if search_result:
    print("匹配成功：", search_result.group())
else:
    print("匹配失败")

# 匹配成功： 2
```

首先定义了一个正则表达式 `r'\d+'`，它可以匹配一个或多个数字字符。然后定义了一个字符串 `string`，需要在其中搜索符合正则表达式规则的子串。最后使用 `re.search()` 函数在字符串中搜索第一个符合正则表达式规则的子串，并返回一个 `Match` 对象。如果匹配成功，则输出匹配到的结果；否则输出“匹配失败”。

```
# group：查看匹配字符
# span：查看匹配数据的索引取值区间
```

1. `re.search()` 只会搜索到第一个符合正则表达式规则的子串，并返回一个 `Match` 对象。如果想要搜索所有符合规则的子串，则应该使用 `re.findall()`
2. 如果 `Match` 对象存在，则可以通过调用 `group()` 方法获取匹配到的子串；如果不存在，则说明匹配失败。
3. 在使用正则表达式时，需要根据具体情况考虑各种特殊字符和操作符的含义和使用方式，并进行适当的转义处理或括号分组。

3.修饰符

==修饰符(可选标志--flags)==

修饰符	描述
re.I	使匹配对大小写不敏感
re.L	做本地化识别（locale-aware）匹配
re.M	多行匹配，影响 ^ 和 $
re.S	使 . 匹配包括换行在内的所有字符
re.U	根据Unicode字符集解析字符。这个标志影响 \w, \W, \b, \B.
re.X	该标志通过给予你更灵活的格式以便你将正则表达式写得更易于理解。

==接下来具体学习一下==

```
# re.I 或 re.IGNORECASE：表示忽略大小写匹配
import re

# 定义正则表达式，使用忽略大小写（re.I）匹配模式
pattern = r'hello'

# 定义字符串
string = 'Hello, world!'

# 使用 search() 函数查找
search_result = re.search(pattern, string, re.I)

# 输出匹配结果
if search_result:
```

```
    print("匹配成功：", search_result.group())
else:
    print("匹配失败")
```

正则表达式 pattern 用于匹配字符串中的单词 'hello'，但是使用了大小写不敏感的匹配模式 re.I，因此可以匹配到大写的单词 'Hello'。

```
# re.M 或 re.MULTILINE：表示进行多行匹配。

import re

# 定义正则表达式，使用多行（re.M）匹配模式
pattern = r'^hello'  # 匹配以hello开头的

# 定义字符串
string = 'Hello\nhello, world!'   # 两行
# print(string)

# 使用 findall() 函数查找所有匹配项
result = re.findall(pattern, string, re.M)

# 输出结果
print(result)
```

正则表达式 pattern 用于匹配字符串中以单词 'hello' 开头的行，使用了多行匹配模式 re.M，因此可以匹配到两行中以 'hello' 开头的字符串。

```
# re.S 或 re.DOTALL：表示可以匹配任意字符，包括换行符。
import re

# 定义正则表达式，使用 . 匹配任意字符（含换行符）的模式
pattern = r'.*'

# 定义字符串
string = 'Hello\nworld!'

# 使用 search() 函数查找
search_result = re.search(pattern, string, re.S)

# 输出匹配结果
if search_result:
    print("匹配成功：", search_result.group())
else:
    print("匹配失败")
```

正则表达式 pattern 用于匹配字符串中的任何字符，包括换行符。由于使用了 re.S 修饰符，因此可以匹配到整个字符串。

```
# re.X 或 re.VERBOSE：表示进行可读性更好的正则表达式编写。

import re

# 定义正则表达式，使用换行和注释来分隔模式
pattern = r"""
    \d+       # 表示匹配一个或多个数字字符
    \s*       # 表示匹配零个或多个空格字符
    dollars   # 表示匹配单词 'dollars'
"""

# 定义字符串
string = 'The price is 2 dollars.'

# 使用 search() 函数查找
search_result = re.search(pattern, string, re.X)

# 输出匹配结果
if search_result:
    print("匹配成功：", search_result.group())
else:
    print("匹配失败")


pattern = r'\d+\s*dollars'

# 定义字符串
string = 'The price is 2 dollars.'

# 使用 search() 函数查找
search_result = re.search(pattern, string)
print(search_result)
print(search_result.group())
print(search_result.span())
```

正则表达式 `pattern` 用于匹配字符串中的价值和货币单位，使用了 `re.X` 修饰符来进行可读性更好的正则表达式编写。通过注释和换行等方式，可以将正则表达式分解为多个易于理解的部分，使得正则表达式变得更加清晰和易于维护。

```
# re.U 或 re.UNICODE：表示使用 Unicode 字符集进行匹配。

import re

# 定义 Unicode 字符串
unicode_str = u'Hello, 你好! '

# 定义 ASCII 字符串
ascii_str = 'Hello, world!'
```

```
# 定义正则表达式
pattern = r'\w+'  # 匹配单词字符  a-z，A-Z，0-9 ，_

# 使用 re.U 修饰符进行匹配
match_result1 = re.findall(pattern, unicode_str, re.U)
print("使用 re.U 修饰符的匹配结果：", match_result1)

# 不使用 re.U 修饰符进行匹配
match_result2 = re.findall(pattern, unicode_str)
print("不使用 re.U 修饰符的匹配结果：", match_result2)
```

3.元字符

3-1 字符匹配元字符

元字符	含义
`.`	匹配任意一个字符（换行符除外）
`^`	匹配字符串的开头
`$`	匹配字符串的结尾

```
import re

# 使用 . 匹配任意字符
string = "abc123"
pattern = r"a.c"
match_object = re.search(pattern, string)
print(match_object.group())  # 输出结果为 "abc"

# 使用 ^ 匹配字符串开头
string = "hello, world!"
pattern = r"^hello"
match_object = re.search(pattern, string)
print(match_object.group())  # 输出结果为 "hello"

# 使用 $ 匹配字符串结尾
string = "hello, world!"
```

```
pattern = r"world!$"
match_object = re.search(pattern, string)
print(match_object.group())  # 输出结果为 "world!"
```

3-2 重复次数限定元字符

元字符	含义
`*`	匹配前面的字符出现 0 次或多次
`+`	匹配前面的字符出现 1 次或多次
`?`	匹配前面的字符出现 0 次或 1 次
`{m}`	匹配前面的字符恰好出现 m 次
`{m,}`	匹配前面的字符至少出现 m 次
`{m,n}`	匹配前面的字符出现 m~n 次

```
import re

# 使用 * 匹配前面的字符出现 0 次或多次
string = "goood job"
pattern = r"go*d"
match_object = re.search(pattern, string)
print(match_object.group())  # 输出结果为 "goood"

# 使用 + 匹配前面的字符出现 1 次或多次
string = "good job"
pattern = r"go+d"
match_object = re.search(pattern, string)
print(match_object.group())  # 输出结果为 "good"

# 使用 ? 匹配前面的字符出现 0 次或 1 次
string1 = "color"
string2 = "colour"
pattern = r"colou?r"
match_object1 = re.search(pattern, string1)
match_object2 = re.search(pattern, string2)
print(match_object1.group())  # 输出结果为 "color"
print(match_object2.group())  # 输出结果为 "colour"

# 使用 {} 匹配前面的字符出现固定次数
string = "12345"
pattern = r"\d{3}"
match_object = re.search(pattern, string)
print(match_object.group())  # 输出结果为 "123"

# 使用 {} 匹配前面的字符出现一定范围内的次数
string = "oooo"
pattern = r"o{2,3}"   # {2,3} 不是左闭右开  只是出现前面字符的一个范围2 到3 次
match_object = re.search(pattern, string)
```

```
print(match_object.group())  # 输出结果为 "oo"
```

3-3 字符集合匹配元字符

元字符	含义	示例
`[]`	匹配方括号内的任意一个字符	`[abc]d` 可以匹配 "ad"、"bd"、"cd"，但不能匹配 "dd"
`[^]`	匹配不在方括号内的任意一个字符	`[^abc]d` 可以匹配 "dd"、"ed"，但不能匹配 "ad"、"bd"、"cd"

```
import re

# 使用 [] 匹配方括号内的任意一个字符
string = "abcd"
pattern = r"[abc]d"
match_object = re.search(pattern, string)
print(match_object.group())  # 输出结果为 "cd"

# 使用 [^] 匹配不在方括号内的任意一个字符
string = "abd"
pattern = r"[^afc]d"
match_object = re.search(pattern, string)
print(match_object.group())  # 输出结果为 "bd"
```

3-4 分组元字符

元字符	含义	示例
`()`	分组，匹配括号内的表达式	`(go)+` 可以匹配 "gogo"、"gogogo" 等字符串

```
import re

# 使用 () 进行分组
string = "abc123"
pattern = r"(abc)\d+"
match_object = re.search(pattern, string)
print(match_object.group(1))  # 输出结果为 "abc"
```

```
# 使用 (?P<name>) 对捕获的分组进行命名
string = "2023-05-11"
pattern = r"(?P<year>\d{4})-(?P<month>\d{2})-(?P<day>\d{2})"
match_object = re.search(pattern, string)
print(match_object.group("year"))  # 输出结果为 "2023"
print(match_object.group("month"))  # 输出结果为 "05"
print(match_object.group("day"))  # 输出结果为 "11"
```

3-5 边界匹配元字符

元字符	含义	示例
`\b`	匹配单词边界（空格、标点符号等）	`\bh\w*\b` 可以匹配 "hello"、"hi" 等以字母 h 开头的单词
`\B`	匹配非单词边界	`\Bh\w*\B` 可以匹配 "ahem"、"shah" 等以字母 h 开头的非单词字符串

```
import re

# 使用 \b 匹配单词边界
string = "hello, world! hello"
pattern = r"\bhello\b"
match_object = re.findall(pattern, string)
print(match_object)  # 输出结果为 ["hello", "hello"]

string = "hello, world! hello   hello.>?  hello  1hello1 1hello1"
pattern = r"\Bhello\B"
match_object = re.findall(pattern, string)
print(match_object)


# 使用 ^ 匹配字符串开头
string = "hello, world!"
pattern = r"^hello"
match_object = re.findall(pattern, string)
print(match_object)  # 输出结果为 ["hello"]

# 使用 $ 匹配字符串结尾
string = "hello, world"
pattern = r"world$"
match_object = re.findall(pattern, string)
print(match_object)  # 输出结果为 ["world"]
```

3-6 字符类别匹配元字符

元字符	含义	示例
`\d`	匹配数字	`\d{3}` 可以匹配 "123"，但不能匹配 "1a3"
`\D`	匹配非数字字符	`\D{3}` 可以匹配 "abc"，但不能匹配 "a1c"
`\s`	匹配任意空白字符（包括空格、制表符、换行符等）	`hello\sworld` 可以匹配 "hello world" 等包含空白符的字符串
`\S`	匹配任意非空白字符	`hello\Sworld` 可以匹配 "hello,world" 等不包含空白符的字符串
`\w`	匹配任意字母、数字或下划线	`\w+` 可以匹配 "hello123"、"world_2021" 等包含字母、数字和下划线的字符串
`\W`	匹配任意非字母、数字或下划线字符	`\W+` 可以匹配 ",!$" 等不包含字母、数字和下划线的字符串

```python
import re

# 使用 \d 匹配数字字符
string = "abc123"
pattern = r"\d+"
match_object = re.search(pattern, string)
print(match_object.group())  # 输出结果为 "123"

# 使用 \w 匹配字母、数字和下划线字符
string = "hello_world_123"
pattern = r"\w+"
match_object = re.findall(pattern, string)
print(match_object)  # 输出结果为 ["hello_world_123"]

# 使用 \s 匹配空白字符
string = "hello world"
pattern = r"\s+"
match_object = re.findall(pattern, string)
print(match_object)  # 输出结果为 [" "]
```

4.技巧

4-1 贪婪与非贪婪

```
# *、+ 和 ? 这几个操作符是贪婪匹配的，它们会尽量匹配更多的文本。

# 为了避免贪婪匹配，可以使用 *?、+? 和 ?? 这几个操作符，它们会尽量匹配更少的文本。
```

```
import re

# 贪婪匹配示例，输出结果为 "abccccccc"
string = "abccccccc"
pattern = r"abc+"
match_object = re.search(pattern, string)
print(match_object.group())

# 非贪婪匹配示例，输出结果为 "abc"
string = "abccccccc"
pattern = r"abc+?"
match_object = re.search(pattern, string)
print(match_object.group())
```

5.案例

```
import re

# 电话号码匹配示例，输出结果为 "13812345678"
string = "我的电话是13812345678，请给我打电话。"
pattern = r"1[3456789]\d{9}"
match_object = re.search(pattern, string)
print(match_object.group())

# 邮件地址匹配示例，输出结果为 "example@example.com"
string = "我的邮箱是example@example.com，请发邮件给我。"
pattern = r"[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}"
match_object = re.search(pattern, string, re.IGNORECASE)
print(match_object.group())

# HTML 标签替换示例
string = "<p>我是一段HTML文本。</p>"
pattern = r"<.*?>"
replacement = ""
new_string = re.sub(pattern, replacement, string)
print(new_string)  # 输出结果为 "我是一段HTML文本。"

# 元音字母相邻去重示例，输出结果为 "abbccdddeiouxwxz"
string = "aabbccdddeeiioouuxwxzz"
pattern = r"(a|e|i|o|u|x|w|z)\1+"
match_iter = re.finditer(pattern, string)
for match_object in match_iter:
    old_str = match_object.group()
    new_str = match_object.group(1)
    string = string.replace(old_str, new_str)
```

```
print(string)
```

圆括号用于创建一个捕获组，以便在后续的表达式中引用。其中，捕获组可以使用 "\1"、"\2" 等符号在表达式中引用到。例如，"(ab)\1" 匹配由两个 "ab" 组成的字符串，而 "(ab)(cd)\2\1" 则匹配由"abcdcdab"组成的字符串。

```
import re

# (hello) 表示创建一个捕获组，输出结果为 "hello"。
string = "hello, world!"
pattern = r"(hello)"
match_object = re.search(pattern, string)
print(match_object.group(1))

# (\d{4})-(\d{2})-(\d{2}) 创建三个捕获组分别用于匹配年、月和日，输出结果为 "2023", "05" 和"11"。
string = "今天是2023-05-11，天气晴朗。"
pattern = r"(\d{4})-(\d{2})-(\d{2})"
match_object = re.search(pattern, string)
print(match_object.group(1))
print(match_object.group(2))
print(match_object.group(3))

# (ab)\1 匹配由两个 "ab" 组成的字符串，输出结果为 "abab"。
string = "abababab"
pattern = r"(ab)\1"
match_object = re.search(pattern, string)
print(match_object.group())

# (ab)(cd)\2\1 匹配由 "abcdcdab"组成的字符串，输出结果为 "abcdcdab"。
string = "abcdcdab"
pattern = r"(ab)(cd)\2\1"
match_object = re.search(pattern, string)
print(match_object.group())
```