PYTHON程序设计

计算机学院 王纯

七 异常和正则表达式

七 异常和正则表达式

- 异常处理
- 正则表达式

异常处理：基本概念

异常（*Exception*），指的是程序中遇到的非致命的错误。

例如程序要打开一个不存的文件、网络连接中断、除零操作、操作数越界、装载一个不存在的类等情况。

常见异常

序号	名称	描述
1	ArithmeticError	算术错误，例如除零错ZeroDivisionError，结果太大时的溢出错误OverflowError等
2	AssertionError	当assert断言条件为假的时候抛出的异常
3	AttributeError	当访问的对象属性不存在的时候抛出的异常
4	LookupError	当映射或序列上使用的键或索引无效时引发的异常的基类。包括超出对象索引的范围时抛出的异常IndexError，在字典中查找一个不存在的key抛出的异常KeyError等。
5	MemoryError	当操作耗尽内存时引发的异常
6	NameError	访问一个不存在的变量时抛出的异常
7	OSError	与操作系统相关的错误会引发此异常，包括文件、目录、进程、连接、中断、权限等方面的错误。例如“找不到文件”、“磁盘已满”、“写入文件无权限”等。
8	RuntimeError	运行时错误，一般是在检测到不属于任何其它类别的错误时触发
9	SyntaxError	解析器遇到语法错误时引发
10	TypeError	类型错误，通常是不同类型之间的操作会出现此异常
11	ValueError	参数的类型正确但是值不在指定范围之内

异常处理：异常捕获

1）异常处理（某些时候也称为错误处理）功能，提供了处理程序运行时出现的任何意外或异常情况的方法。

2）异常处理使用 ***try、except***和 ***finally*** 关键字来尝试可能未成功的操作、处理失败，以及在事后清理资源。

异常处理示例

```
#异常的产生
age= int(input("Please enter your age: "))
print(f"your age is {age}")#用户输入数字，正常输出

#如果如输入字符串，比如“abc”
Traceback (most recent call last):
  File "E:/Python/例子/6-2异常处理例子1-1.py", line 1, in
<module>
    age= int(input("Please enter your age: "))
ValueError: invalid literal for int() with base 10: 'abc ‘

#在程序中增加异常处理
try:
    age= int(input("Please enter your age: "))
    print(f"your age is {age}")
except:#输入非数字后，就执行下面的异常处理逻辑
    print("It's not a number! Please input again!")
```

异常处理：捕获多个异常

可以将**多个异常**，分别写好对应的处理逻辑。

异常处理示例

```
#捕获多个异常
try:
    number1 = int(input("Please enter number1: "))
    number2 = int(input("Please enter number2: "))
    print(number1/number2)

except ValueError: #如果输入的不是数字
    print("It's not a number! Please input again!")

except ZeroDivisionError: #如果第二个数是零
    print("The second number can not be zero. Please input it again!")

#另一种写法如下
try:
    number1 = int(input("Please enter number1: "))
    number2 = int(input("Please enter number2: "))
    print(number1/number2)
except (ValueError, ZeroDivisionError)as err:
    print("Something is wrong.")
    print(err)
```

异常处理：ELSE子句和FINALLY子句

else子句（可选）。如果没有异常发生，可以使程序走到else分支，处理正常情况时的程序逻辑。注意else子句需要放在所有except子句之后。

finally 子句（可选）。如果使用了该子句，则无论什么情况，该子句都会被执行，通常用于**释放外部资源**

异常处理示例

```
#增加else子句的处理逻辑
while True: #循环直到输入正确为止
    try:
        number1 = int(input("Please enter number1: "))
        number2 = int(input("Please enter number2: "))
        answer = (number1/number2)
    except ValueError:
        print("It's not a number! Please input again!")
    except ZeroDivisionError:
        print("The second number can not be zero. Please input it again!")
    else:
    print(f"The answer is {answer}")
    break

#再增加finally子句的处理逻辑
try:
    f = open("demo.txt", encoding = 'utf-8')
    # 执行文件操作
except:
    # 处理各种异常
finally:
    f.close()
```

异常处理：主动抛出异常

系统默认的异常类型总是有限的，如果当程序运行时，产生的特殊数据并不在默认的异常类型之内时，就可以选择主动抛出异常，以便程序进行后续的异常捕捉处理。

例如当输入年龄时，输入的数据应该在0至100岁之间。如果输入是数字但不在此范围内，系统自带的异常类型是无法处理的，因为只要是数字都会认为是正确的输入。如果你的程序需要处理这种异常的输入，就可以通过主动抛出异常来进行后续处理。

异常处理示例

```
try:
    age= int(input("Please enter your age: "))
    if 0<= age <= 100:
        print(f"your age is {age}.")
    else:
        raise ValueError(f"{age} is not a valid age.")
        #此处主动抛出一个异常

except ValueError as err:
    print(f"You entered incorrect age.{err}")

#首先是自己抛出一个异常，然后对自己抛出的异常进行处理
```

异常处理：自定义异常类

1）异常在Python中是一个类（*class*），自定义异常的父类是***Exception***，所以我们自定义类也必须继承***Exception***。

2）定义好这个自定义的异常类之后，就可以通过***try-except***方式来使用它。

异常处理示例

```
#定义自己的异常类
class MyOwnException(Exception):
    pass

#使用自己的异常类
try:
    raise MyOwnException ("自定义异常")
except MyOwnException as err:
    print(err)
#raise关键字后面可以指定你要抛出的自定义异常类的实例，捕获一个异常就是捕获到该自定义类的一个实例。
```

异常处理：自定义异常类

```
#定义自己的异常类
class MyOwnException(Exception):
    def __init__(self,length,minlen):
        Exception.__init__(self)
        self.length = length
        self.minlen = minlen
while(True):
    try:
        str= input('Please enter a string:')
        if len(str)<6:
            #如果输入的字符串长度小于6,触发异常
            raise MyOwnException(len(str),6)
    except MyOwnException as err:
        print("MyOwnException:Your string length is {err.length}. Please input at least {err.minlen}")
    else:
        print("No exception!")
        break
```

正则表达式：基本概念

1）正则表达式，又称规则表达式。（**Regular Expression**，在代码中常简写为*regex*、*regexp*或*RE*），通常被用来检索、替换那些符合某个模式(规则)的文本。

2）正则表达式是对字符串操作的一种逻辑公式，就是用事先定义好的一些特定字符、及这些特定字符的组合，组成一个“规则字符串”，这个“规则字符串”用来表达对字符串的一种过滤逻辑。

常规示例（检测电话号码：xxx-xxx-xxxx）

```
def isPhoneNumber(text):
    if len(text) != 12: #先看是不是12位
        return False
    for i in range(0, 3):#看前3位是不是数字
        if not text[i].isdecimal():
            return False
        if text[3] != '-':#看第4个是不是分隔符
            return False
    for i in range(4, 7): #看5-7，是不是数字
        if not text[i].isdecimal():
            return False
        if text[7] != '-':#看第8位是不是分隔符
            return False
    for i in range(8, 12):#看9-12，是不是数字
        if not text[i].isdecimal():
            return False
    return True
print(isPhoneNumber('415-555-4242'))
#定义一个函数，经过一串逻辑，判断是不是电话号码
#代码冗长、功能有限，支持情况太少：其他分隔符、区号有没有括号、分机问题等。
```

正则表达式：示例

1）Python的re模块，提供了正则表达式处理函数相关函数。

2）使用步骤：

a）导入re模块。

b）创建正则表达式对象。

c）使用该对象的search方法进行搜索。

d）返回一个结果对象，调用结果对象的group方法获取检索结果。

正则表达式对比示例

```
import re #导入正则表达式模块
#调用compile方法，传入正则表达式字符串，生成正则表达式对象。\d代表1位数字。规则解读：3位数字跟1个“-”，再跟一个3位数字，跟一个“-”，最后是4位数字。因为解析规则中可能包括很多情况下的\转义字符，所以在最开始加了一个“r”，用来代表规则字符串不需要进行转义。
phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d')
#调用正则表达式对象的search方法，进行检索。检索不到的话，返回None
mo = phoneNumRegex.search('My number is 415-555-4242.')
#读取返回的Match对象的group方法，得到检索匹配出来的结果。
print('找到电话号码: ' + mo.group())

#三行代码即可完成。也可以通过将表达式直接传给search方法，简化成两行代码：
#mo = re.search(r'\d\d\d-\d\d\d-\d\d\d\d', 'My number is 415-555-4242.')
#还可以改为1行代码：
re.search(r'\d\d\d-\d\d\d-\d\d\d\d', 'My number is 415-555-4242.').group()
```

正则表达式：更多用法-分组匹配

1）利用括号进行分组。可以在正则表达式中，使用括号，对匹配结果进行分组。

如前例中，如果需要将前三位（区号）单独识别出来，可以用括号将正则表达式分组：(\d\d\d)-(\d\d\d-\d\d\d\d)。

2）用group方法读取结果时候： group(0)或group()，返回整个结果；group(1)返回区号；group(2)返回后面8位。

3）如果内容中有括号，比如区号本身就是括号括起来来的，就需要在正则表达式中，对于括号进行转义:(\(\d\d\d\))-(\d\d\d-\d\d\d\d)

正则表达式示例

```
import re #导入正则表达式模块

#调用compile方法，传入正则表达式字符串
phoneNumRegex = re.compile(r'(\d\d\d)-(\d\d\d-\d\d\d\d)')

#调用正则表达式对象的search方法，进行检索。
mo = phoneNumRegex.search('My number is 415-555-4242.')

#读取返回对象的group方法，得到检索匹配出来的结果。
print('找到电话号码: ' + mo.group())
#找到电话号码: 415-555-4242
print('区号是: ' + mo.group(1))
#区号是: 415
print('号码是: ' + mo.group(2))
#号码是: 555-4242
print(mo.groups())
#('415', '555-4242')，输出区号和号码构成的元组
```

正则表达式：更多用法-匹配多个条件

1）一次匹配多个条件。可以使用字符“|”，一次匹配多个表达式条件。如要匹配“|”字符，需要在正则表达式中进行转义“\|”。

2）如，正则表达式 r'Batman|Tina Fey'将匹配'Batman'或'Tina Fey'。如果 Batman 和 Tina Fey 都存在，将返回第一个。

3）可以通过结合括号，实现按前缀的匹配条件。如，r'Bat(man|mobile|copter|bat)')，可以匹配'Batman'、'Batmobile'、'Batcopter'和'Batbat'中的任意一个。

正则表达式示例

```
import re #导入正则表达式模块

#调用compile方法，传入正则表达式字符串
heroRegex = re.compile (r'Batman|Tina Fey')
mo1 = heroRegex.search('Batman and Tina Fey.')
print(mo1.group())#'Batman'，返回了第一个

mo2 = heroRegex.search('Tina Fey and Batman.')#颠倒顺序
print(mo2.group())#'Tina Fey'，返回了第一个

batRegex = re.compile(r'Bat(man|mobile|copter|bat)')
mo = batRegex.search('Batmobile lost a wheel')
print(mo.group())#'Batmobile',以‘Bat’为前缀，匹配到的第一个
print(mo.group(1))#'mobile',group()返回的是完整的串，group(1)返回的是括号中中识别到的部分
```

正则表达式：更多用法-问号/星号可选匹配

1）用问号实现可选匹配。字符?表明它前面的分组在这个模式中是可选的“匹配这个问号之前的分组零次或一次”。

2）用星号匹配零次或多次。

3）如果想匹配的内容中有问号或星号，写正则表达式的时候也要进行转义。

正则表达式示例

```
import re #导入正则表达式模块

#调用compile方法，传入正则表达式字符串
batRegex = re.compile(r'Bat(wo)?man')
mo1 = batRegex.search('The Adventures of Batman')
print(mo1.group())#'Batman'，此时“wo”没有出现

mo2 = batRegex.search('The Adventures of Batwoman')
print(mo2.group())#'Batwoman'，此时“wo”出现1次

#对于前面例子加入？号，r'(\d\d\d-)?\d\d\d-\d\d\d\d'），不管有没有区号，电话号码都会被识别出来

batRegex = re.compile(r'Bat(wo)*man')
mo1 = batRegex.search('The Adventures of Batman')
print(mo1.group())#'Batwoman',0个“wo”
mo2 = batRegex.search('The Adventures of Batwowowowoman')
print(mo2.group())#'Batwowowowoman',4个“wo”
```

正则表达式：更多用法-加号/花括号可选匹配

1）用加号匹配一次或多次。想匹配内容中的“+”号，也需要转义。

2）用花括号匹配特定次数。如(Ha){3}意思是匹配'Ha'3次，将匹配字符串'HaHaHa'，但不会匹配'HaHa'，因为后者只重复了(Ha)分组两次。

(Ha){3,}将匹配 3 次或更多次实例；

(Ha){,5}将匹配 0 到 5 次实例。

正则表达式示例

```
import re #导入正则表达式模块

#调用compile方法，传入正则表达式字符串
batRegex = re.compile(r'Bat(wo)+man')
mo1 = batRegex.search('The Adventures of Batman') print(mo1 is
None)#True，因为“wo”一次也没有出现

mo2 = batRegex.search('The Adventures of Batwowoman')
print(mo2.group())#'Batwowoman'，此时“wo”出现2次

haRegex = re.compile(r'(Ha){3}')
mo1 = haRegex.search('Ha')
print(mo1 is None)#True，因为“ha”不够3次
mo2 = haRegex.search('HaHaHa')
print(mo2.group())#'HaHaHa',3个“Ha”

#用该方法，可以简化前面电话号码的例子：
#原先：r'\d\d\d-\d\d\d-\d\d\d\d'
#现在：r'(\d){3}-(\d){3}-(\d){4}'
```

正则表达式：更多用法-贪心匹配

1）贪心匹配和非贪心匹配。Python 的正则表达式默认是“贪心”的，表示在有二义的情况下，会尽可能匹配最长的字符串。花括号的“非贪心”版本匹配尽可能最短的字符串，即在结束的花括号后跟着一个问号。

2）问号的含义。问号可以用来声明非贪心匹配或表示可选的分组，这两种含义是无关的。

正则表达式示例

```
import re #导入正则表达式模块

#调用compile方法，传入正则表达式字符串
greedyHaRegex = re.compile(r'(Ha){3,5}')
mo1 = greedyHaRegex.search('HaHaHaHaHa')
print(mo1.group())#'HaHaHaHaHa'，贪心模式，默认匹配符合条件的最长字符串

nongreedyHaRegex = re.compile(r'(Ha){3,5}?')
print(mo2.group())#'HaHaHa'，非贪心模式，匹配符合条件的最短字符串
```

正则表达式：更多用法-FINDALL

1）findall()方法。Regex对象还有一个findall()方法。

a）search()将返回一个Match对象。findall()返回的是一个字符串为对象的列表。

b）search()返回被查找字符串中的“第一次”匹配的文本，而 findall()方法将返回被查找字符串中的所有匹配。

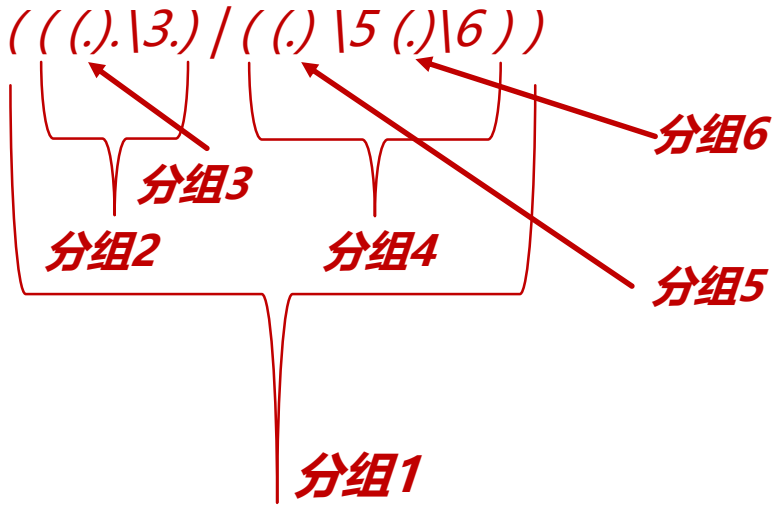
正则表达式示例

```
import re #导入正则表达式模块

#调用compile方法，传入正则表达式字符串
phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d')
mo = phoneNumRegex.search('Cell: 415-555-9999 Work: 212-555-0000')
print(mo1.group())#'415-555-9999'，search只返回了第一个

phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d')
#正则表达式的条件没有用括号进行分组的情况下
phoneNumRegex.findall('Cell: 415-555-9999 Work: 212-555-0000')
#['415-555-9999', '212-555-0000']，返回所有匹配到的字符串构成的列表
phoneNumRegex = re.compile(r'(\d\d\d)-(\d\d\d)-(\d\d\d\d)')#正则表达式的条件有分组的情况下
phoneNumRegex.findall('Cell: 415-555-9999 Work: 212-555-0000')
#[('415', '555', '9999'), ('212', '555', '0000')]，返回分组构成的元组作为元素的列表
```

正则表达式：更多用法-引用分组匹配的内容

1）\分组编号。正则表达式的分组有默认编号，可以引用该编号匹配重复出现的内容等。

示例中需要匹配ABAC或AABB形式的成语。



正则表达式示例

```
from re import findall

text = '''行尸走肉、金蝉脱壳、百里挑一、金玉满堂、
背水一战、霸王别姬、天上人间、不吐不快、海阔天空、
情非得已、满腹经纶、兵临城下、春暖花开、插翅难逃、
黄道吉日、天下无双、偷天换日、两小无猜、卧虎藏龙、
珠光宝气、簪缨世族、花花公子、绘声绘影、国色天香、
相亲相爱、八仙过海、金玉良缘、掌上明珠、皆大欢喜\
浩浩荡荡、平平安安、秀秀气气、斯斯文文、高高兴兴'''

pattern = r'(((.).\3.)|((.)\5(.)\6))'
for item in findall(pattern, text):
    print(item[0])
```

正则表达式：更多用法-字符分类

1）字符分类

a）\d，0到9的任何数字。

b）\D，除0到9的数字以外的任何字符。

c）\w，任何字母、数字或下划线字符（可以认为是匹配“单词”字符）。

d）\W，除字母、数字和下划线以外的任何字符。

e）\s 空格、制表符或换行符（可以认为是匹配“空白”字符）。

f）\S 除空格、制表符和换行符以外的任何字符。

2）自定义字符分类

正则表达式示例

```
import re #导入正则表达式模块

re.compile(r'[0-5]')#匹配数字0-5

re.compile(r'\d+\s\w+')
#匹配1个或多个数字，然后1个空白，然后1个或多个字符，如'10 lords', '9 ladies'

re.compile(r'[a-zA-Z0-9]')
#自定义分类，匹配所有小写字母、大写字母和数字
re.compile(r'[aeiouAEIOU]')
#自定义分类，匹配所有元音字符，不论大小写

re.compile(r'[^a-zA-Z0-9]')
#自定义分类，通过字符“^”，匹配所有不在“所有小写字母、大写字母和数字”分类中的内容

re.compile(r'[^aeiouAEIOU]')
#自定义分类，通过字符“^”，匹配所有不在“所有元音字符，不论大小写”中的内容
```

正则表达式：更多用法-头尾匹配

1）插入字符，约束起头；美元字符，约束结尾。

a）插入符号（^），表明匹配必须发生在被查找文本开始处。

b）正则表达式的末尾加上美元符号（$），表示该字符串必须以这个正则表达式的模式结束。

c）可以同时使用^和$，表明整个字符串必须匹配该模式。

正则表达式示例

```
import re #导入正则表达式模块

#待匹配的文本，必须从开始就有Hello，才能成功
beginsWithHello = re.compile(r'^Hello')
mo=beginsWithHello.search('Hello world!')
print(mo.group())#Hello，匹配成功
beginsWithHello.search('He said hello.') == None
#True，该文本虽然有Hello，但不在开头，所以匹配失败

#待匹配的文本，结尾有数字，才能成功
endsWithNumber = re.compile(r'\d$')

#待匹配的文本，从头到尾都是数字，才能成功。例如
'12345xyz67890'、'12 34567890'，都会匹配失败
wholeStringIsNum = re.compile(r'^\d+$')
```

正则表达式：更多用法-通配符

1）.（句点）字符称为“通配符”。匹配除了换行之外的所有字符。

2）点-星匹配所有字符。

正则表达式示例

```
import re #导入正则表达式模块

atRegex = re.compile(r'.at')
atRegex.findall('The cat in the hat sat on the flat mat.')
['cat', 'hat', 'sat', 'lat', 'mat']#匹配所有字符+at

nameRegex = re.compile(r'First Name: (.*) Last Name: (.*)')
mo = nameRegex.search('First Name: Al Last Name: Sweigart')
mo.group(1)#'Al'
mo.group(2)#'Sweigart'
#此逻辑，相当于匹配字符串'First Name:'，接下来是任意文本，接下来是'Last Name:'，然后又是任意文本。

#点-星使用“贪心”模式，总是匹配尽可能多的文本。要用“非贪心”模式匹配所有文本，就使用点-星和问号。
nongreedyRegex = re.compile(r'<.*?>')
mo = nongreedyRegex.search('<To serve man> for dinner.>')
mo.group()#'<To serve man>'。此处由于是非贪心模式，Python匹配截止到了man后面这个尖括号。如果是贪心模式，会匹配到句子末尾的尖括号。
```

正则表达式：更多用法-替换方法SUB

1）sub()方法替换字符串。正则表达式不仅能找到文本模式，而且能够用新的文本替换掉这些模式。

语法：***re.sub(pattern, repl, string, count=0, flags=0***)。

pattern，正则表达式。

repl，repl 可以是字符串或函数。

count，替换的计数次数。

flags，后面讲。

正则表达式示例

```
import re #导入正则表达式模块
#匹配'Agent '开头，后面跟着1个或多个字符
n_Re = re.compile(r'Agent \w+')
n_Re.sub('007','Agent Alice gave the secret documents to Agent Bob.')
#'007 gave the secret documents to 007.'
#也可以写为：re.sub(r'Agent \w+','007', 'Agent Alice gave the secret documents to Agent Bob.')
#repl,也可以使用匹配结果中的分组结果进行替换。 “\1”表示 将由分组1匹配的文本所替代，也就是正则表达式的(\w)分组。如果表达式中有多个分组，想用其他分组此处可以是 “\2”、“\3”等。
re.sub(r'Agent (\w)\w*',r'\1****', 'Agent Alice gave the secret documents to Agent Bob and  Agent Carol.',count=2)#替换2次
#'A**** gave the secret documents to B**** and Agent Carol.'

def dashrepl(matchobj):
    if matchobj.group(0) == '-': return ' '
    else:return '-'
re.sub('-{1,2}', dashrepl, 'pro----gram-files')
#'pro--gram files'。此处，dashrepl是一个函数，该函数只能接受matchobj对象作为参数，并返回替换字符串。总体效果：匹配文本中的1至2个-，1个则替换为空，两个则替换为1个。
```

正则表达式：更多用法-复杂表达式书写1

1）支持换行、忽略大小写、忽略空白和注释。

compile方法可以输入第2个参数：

a）*re.DOTALL*。可以让句点作为通配符的时候，也支持匹配换行符。

b）*re.IGNORECASE*。识别时候不区分表达式中字符的大小写。

c）*re.VERBOSE*。识别时候忽略正则表达式中的空白和注释。（见：管理复杂的表达式）

3）同时使用，用“|”将多个flags拼起来：

re.IGNORECASE | re.DOTALL | re.VERBOSE

正则表达式示例

```
import re #导入正则表达式模块
lineRegex = re.compile('.*', re.DOTALL)
lineRegex.search('Serve the public trust.\nProtect the innocent.\nUphold the law.').group()
#'Serve the public trust.\nProtect the innocent.\nUphold the law.'匹配了换行符。

robocop = re.compile(r'robocop', re.I)#re.I等同于re.IGNORECASE
robocop.search('RoboCop is part man, part machine, all cop.').group()
#'RoboCop'，识别时候没有区分大小写
```

正则表达式：更多用法-复杂表达式书写2

1）复杂的表达式。

如果要匹配的文本模式很简单，正则表达式比较好看懂。如果匹配复杂的文本模式，可能正则表达式很长、很难读懂。

可以*re.VERBOSE*，忽略正则表达式字符串中的空白符和注释，形成看上去“结构化”比较好懂的正则表达式。

正则表达式示例

```
import re #导入正则表达式模块

#用三个引号括起来长文本
phoneRegex = re.compile(r'''(
 (\d{3}|\(\d{3}\))? #3个数字，或带括号的3个数字；且是否有区号，是可选的，即可以没有区号只有后面的号码
 (\s|-|\.)? #空白、-、.等形成的分隔符；也是可选，可以没有
 \d{3} #3位数字
 (\s|-|\.) #又一个分隔符
 \d{4} #四位数字
 )''', re.VERBOSE)#忽略表达式中的空白和注释
```

正则表达式-常见模式示例

数字：
^[0-9]$* #起头和结尾都是数字

m-n位的数字：
^\d{m,n}$

非零开头的最多带两位小数的数字：
^([1-9][0-9])+(.[0-9]{1,2})?$*

英文和数字：
^[A-Za-z0-9]+$

长度为3-20的所有字符：
^.{3,20}$

由数字和26个英文字母组成的字符串：
^[A-Za-z0-9]+$

国内电话号码(0511-4405222、021-87888822)：
\d{3}-\d{8}|\d{4}-\d{7}

帐号是否合法(字母开头，允许5-16字节，允许字母数字下划线)：
^[a-zA-Z][a-zA-Z0-9_]{4,15}$

正则表达式：综合示例

1）解析一段文本中的所有中国的手机号码。

a）手机号码是11位的数字。

b）手机号码的第1位一定是1。

c）手机前面可能有国家码：86、+86、86-、+86-。

2）解析后输出一个列表，去掉所有前缀，只保留后面11位。

3）然后再将原文本进行替换，也都替换为只有该11位的号码。

```
.2/scratches/regexp_1.py', wdir='C:/Users/bryan/AppData/Roaming/JetBrains
/PyCharm2021.2/scratches')
['13988200993', '13091112234', '13300112233', '13766778854']
张三的电话是13988200993，李四的电话是13091112234，王五的电话是13300112233，赵六的电话是
13766778854
```

```python
import re

def find_phone_no(text_w_no:str)->tuple:
    r_list=[]
    pno_re=re.compile(r'''(
        (\+)? #识别有没有+号，需转移
        (86)? #识别有没有86前缀
        (-)?  #识别有没有分隔符
        ((\d){11})  #识别后面的11位号码
        )''', re.VERBOSE|re.DOTALL)
    #忽略空白注释，可以换行

        r_list=pno_re.findall(text_w_no)
    #得到的结果列表的元素是元组，其中第5个（正则式中第4部分确定的）是11位号码
        #再处理一下
        pno_list = [t[4] for t in r_list]
    new_text = pno_re.sub(r'\5', text_w_no)  #替换匹配到的字符串

        return (pno_list, new_text) #元组形式返回

pno_text="张三的电话是13988200993，李四的电话是86-13091112234，王五的电话是+8613300112233，赵六的电话是+86-13766778854"

found_pno=find_phone_no(pno_text)
number_list=found_pno[0]
text_subsitute=found_pno[1]

print(number_list)
print(text_subsitute)
```

七 异常和正则表达式

- 异常处理
- 正则表达式

谢谢