

3. PATTERN MATCHING WITH REGULAR EXPRESSIONS

This chapter includes, writing a program to find text patterns without using regular expressions, use regular expressions to make the code much less bloated, string substitution and creating your own character classes and finally write a program that can automatically extract phone numbers and email addresses from a block of text.

3.1 Finding Patterns of Text Without Regular Expressions

Say you want to find an American phone number in a string. You know the pattern: three numbers, a hyphen, three numbers, a hyphen, and four numbers. Example: 415-555-4242. Let's use a function named `isPhoneNumber()` to check whether a string matches this pattern, returning either True or False.

In []:

```
def isPhoneNumber(text):
    if len(text) != 12:
        return False
    for i in range(0, 3):
        if not text[i].isdecimal():
            return False
    if text[3] != '-':
        return False
    for i in range(4, 7):
        if not text[i].isdecimal():
            return False
    if text[7] != '-':
        return False
    for i in range(8, 12):
        if not text[i].isdecimal():
            return False
    return True
print('Is 415-555-4242 a phone number?')
print(isPhoneNumber('415-555-4242'))
print('Is Moshi moshi a phone number?')
print(isPhoneNumber('Moshi moshi'))
```

If you wanted to find a phone number within a larger string then modify the above program as follows

In []:

```
message = 'Call me at 415-555-1011 tomorrow. 415-555-9999 is my office.'

def isPhoneNumber(text):
    if len(text) != 12:
        return False
    for i in range(0, 3):
        if not text[i].isdecimal():
            return False
    if text[3] != '-':
        return False
    for i in range(4, 7):
        if not text[i].isdecimal():
            return False
    if text[7] != '-':
        return False
    for i in range(8, 12):
        if not text[i].isdecimal():
            return False
    return True
for i in range(len(message)):
    chunk = message[i:i+12]
    if isPhoneNumber(chunk):
        print('Phone number found: ' + chunk)
print('Done')
```

3.2 Finding Patterns of Text with Regular Expressions

A Regular Expression (RegEx) is a sequence of characters that defines a search pattern. For example. ^a...s\$
The above code defines a RegEx pattern. The pattern is: any five letter string starting with a and ending with s.

Regular expressions, called regexes for short, are descriptions for a pattern of text. For example, a \d in a regex stands for a digit character—that is, any single numeral from 0 to 9.

The regex \d\d\d-\d\d\d-\d\d\d\d is used by Python to match the same text pattern the previous isPhoneNumber() function did: a string of three numbers, a hyphen, three more numbers, another hyphen, and four numbers.

Any other string would not match the \d\d\d-\d\d\d-\d\d\d\d regex.

A much more sophisticated RegEx would be, example, adding a 3 in braces ({3}) after a pattern is like saying, “Match this pattern three times.”

So the slightly shorter regex \d{3}-\d{3}-\d{4} also matches the correct phone number format.

3.2.1 Creating Regex Objects

All the regex functions in Python are in the re module. To import this module type the following command

In [2]:

```
import re
```

To create a Regex object that matches the phone number pattern, enter the following

In []:

```
phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d')
```

3.2.2 Matching Regex Objects

A Regex object's search() method searches the string it is passed for any matches to the regex.

The search() method will return None if the regex pattern is not found in the string and if the pattern is found, the search() method returns a Match object which have a group() method that will return the actual matched text from the searched string.

In []:

```
phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d')
mo = phoneNumRegex.search('My number is 415-555-4242.')
print('Phone number found: ' + mo.group())
```

3.3 More Pattern Matching with Regular Expressions

To separate the area code from the rest of the phone number we can add parentheses that will create groups in the regex: `(\d\d\d)-(\d\d\d-\d\d\d\d)`.

Then use the group() match object method to grab the matching text from just one group.

The first set of parentheses in a regex string will be group 1. The second set will be group 2.

By passing the integer 1 or 2 to the group() match object method, the different parts of the matched text can be grabbed.

Passing 0 or nothing to the group() method will return the entire matched text.

In []:

```
phoneNumRegex = re.compile(r'(\d\d\d)-(\d\d\d-\d\d\d\d)')
mo = phoneNumRegex.search('My number is 415-555-4242.')
print(mo.group(1))
print(mo.group(2))
print(mo.group(0))
print(mo.group())
```

In []:

```
mo.groups()
```

In []:

```
areaCode, mainNumber = mo.groups()
print("areacode:" + areaCode)
print("mainNumber:" + mainNumber)
```

To match actual parentheses characters use (and) escape characters in the raw string passed to re.compile().

In regular expressions, the following characters have special meanings: . ^ \$ * + ? { } [] \ | ()

3.3.1 Matching Multiple Groups with the Pipe

The | character is called a pipe which can be used anywhere you want to match one of many expressions. For example, the regular expression r'Batman|Tina Fey' will match either 'Batman' or 'Tina Fey'. When both Batman and Tina Fey occur in the searched string, the first occurrence of matching text will be returned as the Match object.

In []:

```
heroRegex = re.compile(r'Batman|Tina Fey')
mo1 = heroRegex.search('Batman and Tina Fey')
print("mo1="+mo1.group())

mo2 = heroRegex.search('Tina Fey and Batman')
print("mo2="+mo2.group())
```

Pipe can also be used to match one of several patterns as part of regex. For example, say to match any of the strings 'Batman', 'Batmobile', 'Batcopter', and 'Batbat'. Since all these strings start with Bat, it would be nice if we specify the prefix only once. which can be done using parentheses as shown below.

In []:

```
batRegex = re.compile(r'Bat(man|mobile|copter|bat)')
mo = batRegex.search('Batmobile lost a wheel')
print("matched: "+mo.group())
print("matched group1:"+mo.group(1))
```

3.3.2 Optional Matching with the Question Mark

suppose there is a pattern that is to be matched only optionally i.e., the regex should find a match regardless of whether that bit of text is there. The ? character flags the group that precedes it as an optional part of the pattern. Example:

In []:

```
batRegex = re.compile(r'Bat(wo)?man')
mo1 = batRegex.search('The Adventures of Batman')
print(mo1.group())
mo2 = batRegex.search('The Adventures of Batwoman')
print(mo2.group())
```

3.3.3 Matching Zero or More with the Star

The * (called the star or asterisk) means “match zero or more”—the group that precedes the star can occur any number of times in the text. It can be completely absent or repeated over and over again.

Example:

In []:

```
batRegex = re.compile(r'Bat(wo)*man')
mo1 = batRegex.search('The Adventures of Batman')
print(mo1.group())
mo2 = batRegex.search('The Adventures of Batwoman')
print(mo2.group())
mo3 = batRegex.search('The Adventures of Batwowowowoman')
mo3.group()
```

3.3.4 Matching One or More with the Plus

The + (or plus) means “match one or more match, the group preceding a plus must appear at least once in matched string.

Example:

In []:

```
batRegex = re.compile(r'Bat(wo)+man')
mo1 = batRegex.search('The Adventures of Batwoman')
print(mo1.group())
mo2 = batRegex.search('The Adventures of Batwowowowoman')
print(mo2.group())
mo3 = batRegex.search('The Adventures of Batman')
print(mo3)
```

3.3.5 Matching Specific Repetitions with Braces

If there is group that has to repeat a specific number of times, then the group should be followed in the regex with a number in braces. For example, the regex (Ha){3} will match the string 'HaHaHa', but it will not match 'HaHa', since the latter has only two repeats of the (Ha) group.

Example:

In []:

```
haRegex = re.compile(r'(Ha){3,5}')
mo1 = haRegex.search('HaHaHa')
print(mo1.group())
mo2 = haRegex.search('HaHaHaHaHa')
mo2.group()
```

3.4 Greedy and Non-greedy Matching

Python's regular expressions are greedy by default, which means that in ambiguous situations they will match the longest string possible. The non-greedy (also called lazy) version of the braces, which matches the shortest string possible, has the closing brace followed by a question mark.

In []:

```
greedyHaRegex = re.compile(r'(Ha){3,5}')
mo1 = greedyHaRegex.search('HaHaHaHaHa')
print(mo1.group())
```

In []:

```
nongreedyHaRegex = re.compile(r'(Ha){3,5}?)'
mo2 = nongreedyHaRegex.search('HaHaHaHaHa')
mo2.group()
```

3.5 The.findall() Method

Unlike search() which will return a Match object of the first matched text in the searched string, the.findall() method will return the strings of every match in the searched string.

In []:

```
#Example Search():
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')
mo.group()
```

In []:

```
#example.findall()
phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d') # has no groups
phoneNumRegex.findall('Cell: 415-555-9999 Work: 212-555-0000')
```

If there are groups in the regular expression, then.findall() will return a list of tuples. Each tuple represents a found match, and its items are the matched strings for each group in the regex.

In []:

```
phoneNumRegex = re.compile(r'(\d\d\d)-(\d\d\d)-(\d\d\d\d)') # has groups
phoneNumRegex.findall('Cell: 415-555-9999 Work: 212-555-0000')
```

3.6 Character Classes

Shorthand character class	Represents
\d	Any numeric digit from 0 to 9.
\D	Any character that is <i>not</i> a numeric digit from 0 to 9.
\w	Any letter, numeric digit, or the underscore character. (Think of this as matching “word” characters.)
\W	Any character that is <i>not</i> a letter, numeric digit, or the underscore character.
\s	Any space, tab, or newline character. (Think of this as matching “space” characters.)
\S	Any character that is <i>not</i> a space, tab, or newline.

In []:

```
xmasRegex = re.compile(r'\d+\s\w+')
xmasRegex.findall('12 drummers, 11 pipers, 10 lords, 9 ladies, 8 maids, 7 swans, 6 geese, 5
```

3.7 Making Your Own Character Classes

User can define their own character class using square brackets.

For example, the character class [aeiouAEIOU] will match any vowel, both lowercase and uppercase.

In []:

```
vowelRegex = re.compile(r'[aeiouAEIOU]')
vowelRegex.findall('RoboCop eats baby food. BABY FOOD.')
```

User can also include ranges of letters or numbers by using a hyphen.

For example, the character class [a-zA-Z0-9] will match all lowercase letters, uppercase letters, and numbers.

By placing a caret character (^) just after the character class’s opening bracket, you can make a negative character class. A negative character class will match all the characters that are not in the character class.

```
#example: consonantRegex = re.compile(r'^[aeiouAEIOU]')
consonantRegex.findall('RoboCop eats baby food.
BABY FOOD.')
```

3.8 The Caret and Dollar Sign Characters

The caret symbol (^) at the start of a regex indicates that a match must occur at the beginning of the searched text. Likewise, dollarsign at the end of the regex indicates that the string must end with this regex pattern. And the ^ and \$ can be used together to indicate that the entire string must match the regex.

In [4]:

```
beginsWithHello = re.compile(r'^Hello')
beginsWithHello.search('Hello, world!')
```

Out[4]:

```
<re.Match object; span=(0, 5), match='Hello'>
```

In [5]:

```
beginsWithHello.search('He said hello.') == None
```

Out[5]:

```
True
```

In [10]:

```
#The r'\d$' regular expression string matches strings that end with a numeric character from 0-9
endsWithNumber = re.compile(r'\d$')
endsWithNumber.search('Your number is 42')
```

Out[10]:

```
<re.Match object; span=(16, 17), match='2'>
```

In [11]:

```
#The r'^\d+$' regular expression string matches strings that both begin and end with one or more digits
wholeStringIsNum = re.compile(r'^\d+$')
wholeStringIsNum.search('1234567890')
```

Out[11]:

```
<re.Match object; span=(0, 10), match='1234567890'>
```

In [12]:

```
wholeStringIsNum.search('12345xyz67890') == None
True
wholeStringIsNum.search('12 34567890') == None
```

Out[12]:

```
True
```

3.9 The Wildcard Character

The . (or dot) character in a regular expression is called a wildcard and will match any character except for a newline.

In [13]:

```
# Example
atRegex = re.compile(r'.at')
atRegex.findall('The cat in the hat sat on the flat mat.')
```

Out[13]:

```
['cat', 'hat', 'sat', 'lat', 'mat']
```

The dot character will match just one character, which is why the match for the text flat in the previous example matched only lat.

3.9.1 Matching Everything with Dot-Star

The dot-star (.) stand for “anything.” Remember that the dot character means “any single character except the newline,” and the star character means “zero or more of the preceding character.”

In [14]:

```
nameRegex = re.compile(r'First Name: (.*) Last Name: (.*)')
mo = nameRegex.search('First Name: Al Last Name: Sweigart')
mo.group(1)
```

Out[14]:

```
'Al'
```

In [15]:

```
mo.group(2)
```

Out[15]:

```
'Sweigart'
```

The dot-star uses greedy mode: It will always try to match as much text as possible. To match any and all text in a non-greedy fashion, use the dot, star, and question mark (.*?).

In [16]:

```
nongreedyRegex = re.compile(r'<.*?>')
mo = nongreedyRegex.search('<To serve man> for dinner.>')
mo.group()
```

Out[16]:

```
'<To serve man>'
```

In [17]:

```
greedyRegex = re.compile(r'<.*>')
mo = greedyRegex.search('<To serve man> for dinner.>')
mo.group()
```

Out[17]:

```
'<To serve man> for dinner.>'
```

3.9.2 Matching Newlines with the Dot Character

By passing `re.DOTALL` as the second argument to `re.compile()`, you can make the dot character match all characters, including the newline character.

In [18]:

```
noNewlineRegex = re.compile('.*')
noNewlineRegex.search('Serve the public trust.\nProtect the innocent.\nUphold the law.').group()
```

Out[18]:

```
'Serve the public trust.'
```

In [19]:

```
newlineRegex = re.compile('.*', re.DOTALL)
newlineRegex.search('Serve the public trust.\nProtect the innocent.\nUphold the law.').group()
```

Out[19]:

```
'Serve the public trust.\nProtect the innocent.\nUphold the law.'
```

3.10 Review of Regex Symbols

- The ? matches zero or one of the preceding group.
- The * matches zero or more of the preceding group.
- The + matches one or more of the preceding group.
- The {n} matches exactly n of the preceding group.
- The {n,} matches n or more of the preceding group.
- The {,m} matches 0 to m of the preceding group.
- The {n,m} matches at least n and at most m of the preceding group.
- {n,m}? or *? or +? performs a non-greedy match of the preceding group.
- ^spam means the string must begin with spam.
- spam\$ means the string must end with spam.
- The . matches any character, except newline characters.
- \d, \w, and \s match a digit, word, or space character, respectively.
- \D, \W, and \S match anything except a digit, word, or space character, respectively.
- [abc] matches any character between the brackets (such as a, b, or c).
- [^abc] matches any character that isn't between the brackets.

3.11 Case-Insensitive Matching

But sometimes you care only about matching the letters without worrying whether they're uppercase or lowercase. To make your regex case-insensitive, you can pass `re.IGNORECASE` or `re.I` as a second argument to `re.compile()`.

In [20]:

```
robocop = re.compile(r'robocop', re.I)
robocop.search('RoboCop is part man, part machine, all cop.')
group()
```

Out[20]:

```
'RoboCop'
```

3.12 Substituting Strings with the sub() Method

Regular expressions can not only find text patterns but can also substitute new text in place of those patterns. The `sub()` method for Regex objects is passed two arguments. The first argument is a string to replace any matches. The second is the string for the regular expression. The `sub()` method returns a string with the substitutions applied.

In [21]:

```
namesRegex = re.compile(r'Agent \w+')
namesRegex.sub('CENSORED', 'Agent Alice gave the secret documents to Agent Bob.')
```

Out[21]:

```
'CENSORED gave the secret documents to CENSORED.'
```

Sometimes the matched text itself can be used as part of the substitution. In the first argument to `sub()`, you can type `\1`, `\2`, `\3`, and so on, to mean “Enter the text of group 1, 2, 3, and so on, in the substitution.” Example to censor names of secret agents:

In [3]:

```
agentNamesRegex = re.compile(r'Agent (\w)\w*')
agentNamesRegex.sub(r'\1****', 'Agent Alice told Agent Carol that Agent Eve knew Agent Bob')
```

Out[3]:

```
'A**** told C**** that E**** knew B**** was a double agent.'
```

3.13 Managing Complex Regexes

Matching complicated text patterns might require long, convoluted regular expressions which can be mitigated by telling the `re.compile()` function to ignore whitespace and comments inside the regular expression string. This “verbose mode” can be enabled by passing the variable `re.VERBOSE` as the second argument to `re.compile()`.

Example: the regular expression can be spread over multiple lines with comments as shown below:

In [5]:

```
phoneRegex = re.compile(r'''(
    (\d{3})|(\d{3}(\d{3}))?          # area code
    (\s|-|\.)?                      # separator
    \d{3}                           # first 3 digits
    (\s|-|\.)                       # separator
    \d{4}                           # last 4 digits
    (\s*(ext|x|ext.)\s*\d{2,5})?    # extension
)''', re.VERBOSE)
```

3.14 Combining re.IGNORECASE, re.DOTALL, and re.VERBOSE

What if you want to use re.VERBOSE to write comments in your regular expression but also want to use re.IGNORECASE to ignore capitalization? Unfortunately, the re.compile() function takes only a single value as its second argument. You can get around this limitation by combining the re.IGNORECASE, re.DOTALL, and re.VERBOSE variables using the pipe character (|), which in this context is known as the bitwise or operator.

In [6]:

```
someRegexValue = re.compile('foo', re.IGNORECASE | re.DOTALL | re.VERBOSE)
```

3.15 Project: Phone Number and Email Address Extractor

In [14]:

```
pip install pyperclip
```

```
Collecting pyperclip
  Downloading pyperclip-1.8.0.tar.gz (16 kB)
Building wheels for collected packages: pyperclip
  Building wheel for pyperclip (setup.py): started
  Building wheel for pyperclip (setup.py): finished with status 'done'
  Created wheel for pyperclip: filename=pyperclip-1.8.0-py3-none-any.whl size=8696 sha256=c74be91443998054a31570745b9ec2555e208086897689b536266e1245e54218
  Stored in directory: c:\users\91984\appdata\local\pip\cache\wheels\5e\f7\441179ddf6ac56f36cb1d84d94f35beedd5da15986ce3d321d
Successfully built pyperclip
Installing collected packages: pyperclip
Successfully installed pyperclip-1.8.0
Note: you may need to restart the kernel to use updated packages.
```

In [3]:

```

import pyperclip,re

phoneRegex = re.compile(r'''(
    (\d{3}|\(\d{3}\))?
    (\s|-|.)?
    (\d{3})
    (\s|-|.)
    (\d{4})
    (\s*(ext|x|ext.)\s*(\d{2,5}))?
)''', re.VERBOSE)
# Create email regex.
emailRegex = re.compile(r'''(
    [a-zA-Z0-9._%+-]+      # username
    @                      # @ symbol
    [a-zA-Z0-9.-]+          # domain name
    (\.[a-zA-Z]{2,4})       # dot-something
)''', re.VERBOSE)
text = str(pyperclip.paste())

matches = []
for groups in phoneRegex.findall(text):
    phoneNum = '-'.join([groups[1], groups[3], groups[5]])
    if groups[8] != '':
        phoneNum += ' x' + groups[8]
    matches.append(phoneNum)
for groups in emailRegex.findall(text):
    matches.append(groups[0])
    # Copy results to the clipboard.
if len(matches) > 0:
    pyperclip.copy('\n'.join(matches))
    print('Copied to clipboard:')
    print('\n'.join(matches))
else:
    print('No phone numbers or email addresses found.')

```

Copied to clipboard:
 800-420-7240
 415-863-9900
 415-863-9950
 info@nostarch.com
 media@nostarch.com
 academic@nostarch.com
 info@nostarch.com

Assignment questions:

write programs to do the following tasks.

- 1) Date Detection
- 2) Strong Password Detection
- 3) Regex Versin of the strip() Method

In []:

