IT Scripting and Automation

Finding Patterns of Text with Regular Expressions in Python

Lecturer: Art Ó Coileáin

Creating *Regex* Objects

In this section we are going to look at Regular Expressions in the Python environment.

- Regular expressions, called ***regexes*** for short, are descriptions for a pattern of text.
- All the regex functions in Python are in the **re** module.

  >>> import re

- Passing a string value representing your regular expression to ***re.compile()*** returns a Regex pattern object.

For example:

To create a Regex object that matches the phone number pattern:

- A ***\d*** in a regex stands for a digit character that is, any single numeral 0 to 9.
- So to create the correct regular expression pattern to match phone number like 415-555-4242 in Python the regex ***\d\d\d-\d\d\d-\d\d\d\d*** is used.

Creating Regex Objects

Using the r Option when creating Regex objects:

Since regular expressions frequently contain backslashes it is convenient to pass raw strings to the **re.compile()** function instead of typing a lot of extra backslashes.

- *Remember that escape characters in Python use the backslash (\).*
- *That is, the string value **'\n'** represents a single newline character, not a backslash followed by a lowercase n.*

  *Therefore, you need to enter the escape characters \\ to print a single backslash. So **'\\\\n'** is the string that represents a backslash followed by a lowercase **n**.*

- We can mark the string as a raw (*which does not escape characters*) by using the "**r**" option; That is ***by putting an r before the first quote of the string value***.

For example:

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

- Typing **r'\d\d\d-\d\d\d-\d\d\d\d'** is much easier than typing:

  '\\\\d\\\\d\\\\d-\\\\d\\\\d\\\\d-\\\\d\\\\d\\\\d\\\\d'.

Matching Regex Objects

This first matching method examined is the ***search()*** method:

- 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.
  - If the pattern **is** found, the *search()* method returns a Match object.
- Match objects have a *group()* method that will return the actual matched text from the searched string. (groups will be explained shortly)

Example:

```
>>> 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())
Phone number found: 415-555-4242
```

Review of Regular Expression Matching

Summary: The steps in using regular expressions in Python are fairly simple:

1. Import the regex module with import **re**.
2. Create a Regex object with the **re.compile()** function. (***Remember to use a raw string***)
3. Pass the string you want to search into the Regex object's **search()** method. This returns a Match object.
4. Call the Match object's **group()** method to return a string of the actual matched text.

Web-based regular expression tester:

- http://regexpal.com/

Grouping with Parentheses

- If you want to separate the area code from the rest of the phone number.
  - Example: Adding parentheses will create groups in the regex:

    (\d\d\d)-(\d\d\d-\d\d\d\d).

- Then you can 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, you can grab different parts of the matched text. Passing 0 or nothing to the *group()* method will return the entire matched text.

Grouping with Parentheses

Examples:

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

>>> mo.group(2)
'555-4242'

>>> mo.group(0)
'415-555-4242'

>>> mo.group()
'415-555-4242'
```

Matching Multiple Groups with the Pipe

As you already know, the '|' character is called a pipe. In Regex's you can use it when 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.

Example:

```
>>> heroRegex= re.compile(r'Batman|Tina Fey')
>>> mo1 = heroRegex.search('Batman and Tina Fey.')
>>> mo1.group()
'Batman'
>>> mo2 = heroRegex.search('Tina Fey and Batman.')
>>> mo2.group()
'Tina Fey'
```

Matching Multiple Groups with the Pipe

- You can also use the pipe to match one of several patterns as part of your regex.

Example:

```
>>> batRegex= re.compile(r'Bat(man|mobile|copter|bat)')
>>> mo= batRegex.search('Batmobile lost a wheel')
>>> mo.group()
'Batmobile'

>>> mo.group(1)
'mobile'
```

Matching with the Question Mark

Sometimes there is a pattern that you want to match only optionally.

- That is, the regex should find a match whether or not that bit of text is there.
- The **?** character flags the group that precedes it as an optional part of the pattern.

Example:

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

Challenge: Can you make the regex look for phone numbers that do or do not have an area code?

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:

```
>>> batRegex= re.compile(r'Bat(wo)*man')
>>> mo1 = batRegex.search('The Adventures of Batman')
>>> mo1.group()
'Batman'

>>> mo2 = batRegex.search('The Adventures of Batwoman')
>>> mo2.group()
'Batwoman'

>>> mo3 = batRegex.search('The Adventures of Batwowowowoman')
>>> mo3.group()
'Batwowowowoman'
```

Matching One or More with the Plus

- While * means "match zero or more," the + (or plus) means "match **one or more**."
- Unlike the star, which does not require its group to appear in the matched string, **the group preceding a plus must appear at least once**. It is not optional.

Example:

```
>>> batRegex= re.compile(r'Bat(wo)+man')
>>> mo1 = batRegex.search('The Adventures of Batwoman')
>>> mo1.group()
'Batwoman‘

>>> mo2 = batRegex.search('The Adventures of Batwowowowoman')
>>> mo2.group()
'Batwowowowoman‘

>>> mo3 = batRegex.search('The Adventures of Batman')
>>> mo3 == None
True
```

Matching Specific Repetitions with {}

- If you have a group that you want to repeat a specific number of times, follow the group in your regex with a number in curly brackets.

<u>For example:</u>

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.

- Instead of one number, you can specify a range by writing a minimum, a comma, and a maximum in between the curly brackets.
  - For example, the regex **(Ha){3,5}** will match **'HaHaHa'**, **'HaHaHaHa'**, and **'HaHaHaHaHa'**.

Continues...

Matching Specific Repetitions with {}

- You can also leave out the first or second number in the curly brackets to leave the minimum or maximum unbounded.

For example:

- **(Ha){3,}** will match three or more instances of the **(Ha)** group,
- while **(Ha){,5}** will match zero to five instances.

- **Curly brackets can help make your regular expressions shorter**.

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.

For example:

- Since **(Ha){3,5}** can match three, four, or five instances of **Ha** in the string **'HaHaHaHaHa'**, you may wonder why the Match object’s call to **group()** in the next curly bracket example returns **'HaHaHaHaHa‘** instead of the shorter possibilities. After all, **'HaHaHa‘** and **'HaHaHaHa‘** are also valid matches of the regular expression **(Ha){3,5}**.
- The ***nongreedy*** version of the curly brackets, which matches the **shortest** string possible, has the closing curly bracket followed by a question mark (**?**).

Greedy and Nongreedy Matching

Examples:

Greedy:

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

Non-Greedy:

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

The *findall()* Method

- While ***search()*** 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.
- ***findall()*** will not return a Match object but a list of strings—*as long as there are no groups in the regular expression*.

Examples:

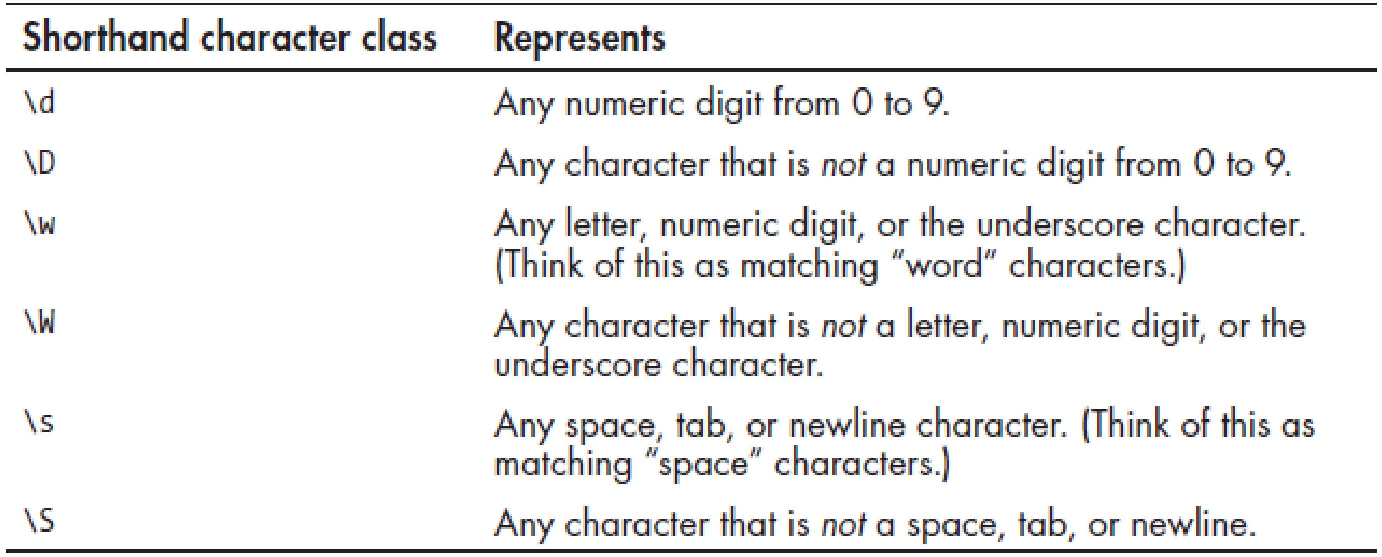
```
>>> phoneNumRegex= re.compile(r'\d\d\d-\d\d\d-\d\d\d\d')
>>> mo= phoneNumRegex.search('Cell: 415-555-9999 Work: 212-555-0000')
>>> mo.group()
'415-555-9999‘
>>> phoneNumRegex= re.compile(r'\d\d\d-\d\d\d-\d\d\d\d') # has no groups
>>> phoneNumRegex.findall('Cell: 415-555-9999 Work: 212-555-0000')
['415-555-9999', '212-555-0000']
```

Character Classes

Even though we have already used the \d character class we shall examine them a little further as they are very useful for shortening regular expressions:

Example:

```
>>> xmasRegex= re.compile(r'\d+\s\w+')
>>> xmasRegex.findall('12 drummers, 11 pipers, 10 lords, 9 ladies, 8 maids, 7
swans, 6 geese, 5 rings, 4 birds, 3 hens, 2 doves, 1 partridge')
['12 drummers', '11 pipers', '10 lords', '9 ladies', '8 maids', '7 swans', '6
geese', '5 rings', '4 birds', '3 hens', '2 doves', '1 partridge']
```



Shorthand character class	Represents
\d	Any numeric digit from 0 to 9.
\D	Any character that is *not* 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 *not* 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 *not* a space, tab, or newline.

Making Your Own Character Classes

There are times when you want to match a set of characters but the shorthand character classes (\d, \w, \s, and so on) are too broad. You can define your own character class using square brackets.

For example,

- The character class [0-5] will match only the numbers 0 to 5; this is much shorter than typing (0|1|2|3|4|5).
- And, the character class **[aeiouAEIOU]** will match any vowel, both lowercase and uppercase:

```
>>> vowelRegex= re.compile(r'[aeiouAEIOU]')
>>> vowelRegex.findall('RoboCopeats baby food. BABY FOOD.')
['o', 'o', 'o', 'e', 'a', 'a', 'o', 'o', 'A', 'O', 'O']
```

- 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. E.g.:

```
>>> consonantRegex= re.compile(r'[^aeiouAEIOU]')
```

The Caret and Dollar Sign Characters

- **Please note:** you can also use the caret symbol (**^**) at the start of a regex to indicate that a match must occur at the beginning of the searched text.
- Likewise, you can put a dollar sign (**$**) at the end of the regex to indicate the string must end with this regex pattern.
- You can use the **^** and **$** together to indicate that the entire string must match the regex.

Example:

```
>>> beginsWithHello = re.compile(r'^Hello')
>>> beginsWithHello.search('Hello world!')
<_sre.SRE_Matchobject; span=(0, 5), match='Hello'>
>>> beginsWithHello.search('He said hello.') == None
True
```

The Caret and Dollar Sign Characters

- The **r'\d$'** regular expression string matches strings that end with a numeric character from 0 to 9.

Example:

```
>>> endsWithNumber= re.compile(r'\d$')
>>> endsWithNumber.search('Your number is 42')
<_sre.SRE_Matchobject; span=(16, 17), match='2'>
>>> endsWithNumber.search('Your number is forty two.') == None
True
```

- *Question*:

  `What does the following regular expression match?

  r'^\d+$'

The Wildcard Character

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

Example:

```
>>> atRegex= re.compile(r'.at')
>>> atRegex.findall('The cat in the hat sat on the flat mat.')
['cat', 'hat', 'sat', 'lat', 'mat']
```

- Sometimes you will want to match everything and anything.

For example: if you want to match the string 'First Name:', followed by any and all text, followed by 'Last Name:', and then followed by anything again.

- You can use the dot-star (.*) to stand in for that “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.”

The Wildcard Character

Examples:

```
>>> nameRegex= re.compile(r'FirstName: (.*) Last Name: (.*)')
>>> mo= nameRegex.search('First Name: Al Last Name: Sweigart')
>>> mo.group(1)
'Al'
>>> mo.group(2)
'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 (**.*?**).
- Like with curly brackets, the question mark tells Python to match in a *non-greedy* way.

Continues…

The Wildcard Character

Examples:

Non-Greedy:

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

Greedy:

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

Case-Insensitive Matching

- To make your regex case-insensitive, you can pass **re.IGNORECASE** or **re.I** as a second argument to **re.compile()**.

Example:

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

>>> robocop.search('ROBOCOP protects the innocent.').group()
'ROBOCOP‘

>>> robocop.search('Al, why does your programming book talk about robocopso much?').group()
'robocop'
```

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.

Example:

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

Continues…

Substituting Strings with the sub() Method

Example:

```
>>> agentNamesRegex= re.compile(r'Agent (\w)\w*')
>>> agentNamesRegex.sub(r'\1****', 'Agent Alice told Agent Carol that Agent Eve knew Agent Bob was a double agent.')
A**** told C**** that E**** knew B**** was a double agent.‘
```

Managing Complex Regexes

Regular expressions are fine if the text pattern you need to match is simple.

But matching complicated text patterns might require long, convoluted regular expressions.

- You can instruct the **re.compile()** function to ignore whitespace and comments inside the regular expression string by using the "**verbose mode**"

- The "verbose mode" can be enabled by passing the variable **re.VERBOSE** as the second argument to **re.compile()**.

Continues…

Managing Complex Regexes

Example:

```
phoneRegex= re.compile(r'''(
(\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)
```

- Note how the previous example **uses the triple-quote syntax** (''') to create a multiline string so that you can spread the regular expression definition over many lines, making it much more legible.
- The comment rules inside the regular expression string are the same as regular Python code: The **#** symbol and everything after it to the end of the line are ignored.