ADVANCED REGULAR EXPRESSIONS

INTRODUCTION

In our introduction to regular expressions of our tutorial we have covered the basic principles of regular expressions. We have shown, what the simplest regular expression looks like. We have also learnt, how to use regular expressions in Python by using the search() and the match() methods of the re module. The concept of formulating and using character classes should be well known by now, as well as the predefined character classes like \d, \D, \s, \S, and so on. You should have learnt how to match the beginning and the end of a string with a regular expression. You should know the special meaning of the question mark to make items optional. We have also introduced the quantifiers to repeat characters and groups arbitrarily or in certain ranges.

You should also be familiar with the use of grouping and the syntax and usage of back references.

Furthermore, we had explained the match objects of the re module and the information they contain and how to retrieve this information by using the methods span(), start(), end(), and group().

The introduction ended with a comprehensive example in Python.

In this chapter we will continue with our explanations of the syntax of the regular expressions. We will also explain further methods of the Python module re. E.g. how to find all the matched substrings of a regular expression. A task which needs programming in other programming languages like Perl or Java, but can be dealt with the call of one method of the re module of Python. So far, we only know how to define a choice of characters with a character class. We will demonstrate in this chapter of our tutorial, how to formulate alternations of substrings.

FINDING ALL MATCHED SUBSTRINGS

The Python module re provides another great method, which other languages like Perl and Java don't provide. If you want to find all the substrings in a string, which match a regular expression, you have to use a loop in Perl and other languages, as can be seen in the following Perl snippet:

```
while ($string =~ m/regex/g) {
  print "Found '$&'.  Next attempt at character " . pos($string)+1
```

```
. "\n";
}
```

It's a lot easier in Python. No need to loop. We can just use the findall method of the re module:

```
re.findall(pattern, string[, flags])
```

findall returns all non-overlapping matches of pattern in string, as a list of strings. The string is scanned left-to-right, and matches are returned in the order in which they are found.

```
>>> t="A fat cat doesn't eat oat but a rat eats bats."
>>> mo = re.findall("[force]at", t)
>>> print(mo)
['fat', 'cat', 'eat', 'oat', 'rat', 'eat']
```

If one or more groups are present in the pattern, findall returns a list of groups. his will be a list of tuples if the pattern has more than one group. We demonstrate this in our next example. We have a long string with various Python training courses and their dates. With the first call to findall, we don't use any grouping and receive the complete string as a result. In the next call, we use grouping and findall returns a list of 2-tuples, each having the course name as the first component and the dates as the second component:

```
>>> import re
>>> courses = "Python Training Course for Beginners: 15/Aug/2011 -
19/Aug/2011;Python Training Course Intermediate: 12/Dec/2011 -
16/Dec/2011;Python Text Processing Course:31/Oct/2011 -
4/Nov/2011"
>>> items = re.findall("[^:]*:[^;]*;?", courses)
>>> items
['Python Training Course for Beginners: 15/Aug/2011 -
19/Aug/2011;', 'Python Training Course Intermediate: 12/Dec/2011 -
16/Dec/2011;', 'Python Text Processing Course:31/Oct/2011 -
4/Nov/2011']
>>> items = re.findall("([^:]*):([^;]*;?)", courses)
>>> items
[('Python Training Course for Beginners', ' 15/Aug/2011 -
19/Aug/2011;'), ('Python Training Course Intermediate', '
12/Dec/2011 - 16/Dec/2011;'), ('Python Text Processing Course',
'31/Oct/2011 - 4/Nov/2011')]
>>>
```

ALTERNATIONS

In our introduction to regular expressions we had introduced character classes. Character classes offer a choice out of a set of characters. Sometimes we need a choice between several regular expression. It's a logical "or" and that's why the symbol for this construct is the "|" symbol.

In the following example, we check, if one of the cities London, Paris, Zurich, Konstanz Bern or Strasbourg appear in a string preceded by the word "location":

```
>>> import re
>>> str = "Course location is London or Paris!"
>>> mo = re.search(r"location.*
(London|Paris|Zurich|Strasbourg)",str)
>>> if mo: print(mo.group())
...
location is London or Paris
>>>
```

If you consider the previous example as too artificial, here is another one. Let's assume, you want to filter your email. You want to find all the correspondence (conversations) between you and Guido van Rossum, the creator and designer of Python. The following regular expression is helpful for this purpose:

```
r"(^To:|^From:) (Guido|van Rossum)"
```

This expression matches all lines starting with either 'To:' or 'From:', followed by a space and then either by the first name 'Guido' or the surname 'van Rossum'.

COMPILING REGULAR EXPRESSIONS

If you want to use the same regexp more than once in a script, it might be a good idea to use a regular expression object, i.e. the regex is compiled.

The general syntax:

```
re.compile(pattern[, flags])
```

compile returns a regex object, which can be used later for searching and replacing. The expressions behaviour can be modified by specifying a flag value.

Abbreviation	Full name	Description
re.I	re.IGNORECASE	Makes the regular expression case-insensitive
re.L	re.LOCALE	The behaviour of some special sequences like \w, \W, \b,\s, \S will be made dependant on the current locale, i.e. the user's language, country aso.
re.M	re.MULTILINE	^ and $ will match at the beginning and at the end of each line and not just at the beginning and the end of the string
re.S	re.DOTALL	The dot "." will match every character **plus the newline**
re.U	re.UNICODE	Makes \w, \W, \b, \B, \d, \D, \s, \S dependent on Unicode character properties

re.X	re.VERBOSE	Allowing "verbose regular expressions", i.e. whitespace are ignored. This means that spaces, tabs, and carriage returns are not matched as such. If you want to match a space in a verbose regular expression, you'll need to escape it by escaping it with a backslash in front of it or include it in a character class. # are also ignored, except when in a character class or preceded by an non-escaped backslash. Everything following a "#" will be ignored until the end of the line, so this character can be used to start a comment.

Compiled regular objects usually are not saving much time, because Python internally compiles AND CACHES regexes whenever you use them with re.search() or re.match(). The only extra time a non-compiled regex takes is the time it needs to check the cache, which is a key lookup of a dictionary.

A good reason to use them is to separate the definition of a regex from its use.

EXAMPLE

We have already introduced a regular expression for matching a superset of UK postcodes in our introductory chapter:

```
r"[A-z]{1,2}[0-9R][0-9A-Z]? [0-9][ABD-HJLNP-UW-Z]{2}"
```

We demonstrate with this regular expression, how we can use the compile functionality of the module re in the following interactive session. The regular expression "regex" is compiled with re.compile(regex) and the compiled object is saved in the object compiled_re. Now we call the method search() of the object compiled_re:

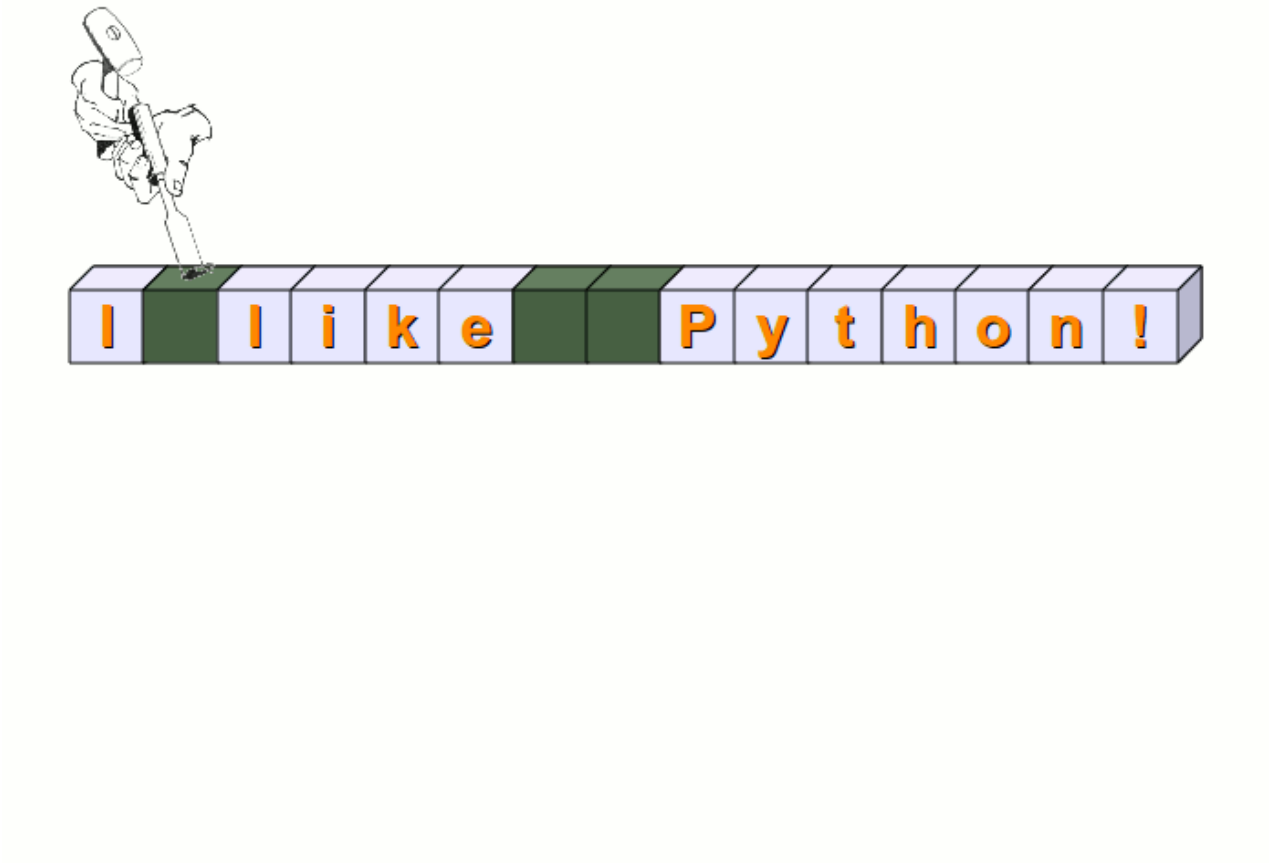
```
>>> import re
>>> regex = r"[A-z]{1,2}[0-9R][0-9A-Z]? [0-9][ABD-HJLNP-UW-Z]{2}"
>>> address = "BBC News Centre, London, W12 7RJ"
>>> compiled_re = re.compile(regex)
>>> res = compiled_re.search(address)
>>> print(res)
<_sre.SRE_Match object at 0x174e578>
>>>
```

SPLITTING A STRING WITH OR WITHOUT REGULAR EXPRESSIONS

There is a string method split, which can be used to split a string into a list of substrings.

```
str.split([sep[, maxsplit]])
```

As you can see, the method split has two optional parameters. If none is given (or is None) , a string will be separated into substring using whitespaces as delimiters, i.e. every substring consisting purely of whitespaces is used as a delimiter.



We demonstrate this behaviour with a famous quotation by Abraham Lincoln:

```
>>> law_courses = "Let reverence for the laws be breathed by every
American mother to the lisping babe that prattles on her lap. Let
it be taught in schools, in seminaries, and in colleges. Let it be
written in primers, spelling books, and in almanacs. Let it be
preached from the pulpit, proclaimed in legislative halls, and
enforced in the courts of justice. And, in short, let it become
the political religion of the nation."
>>> law_courses.split()
['Let', 'reverence', 'for', 'the', 'laws', 'be', 'breathed', 'by',
'every', 'American', 'mother', 'to', 'the', 'lisping', 'babe',
'that', 'prattles', 'on', 'her', 'lap.', 'Let', 'it', 'be',
'taught', 'in', 'schools,', 'in', 'seminaries,', 'and', 'in',
'colleges.', 'Let', 'it', 'be', 'written', 'in', 'primers,',
```

```
'spelling', 'books,', 'and', 'in', 'almanacs.', 'Let', 'it', 'be',
'preached', 'from', 'the', 'pulpit,', 'proclaimed', 'in',
'legislative', 'halls,', 'and', 'enforced', 'in', 'the', 'courts',
'of', 'justice.', 'And,', 'in', 'short,', 'let', 'it', 'become',
'the', 'political', 'religion', 'of', 'the', 'nation.']
>>>
```

Now we look at a string, which could stem from an Excel or an OpenOffice calc file. We have seen in our previous example that split takes whitespaces as default separators. We want to split the string in the following little example using semicolons as separators. The only thing we have to do is to use ";" as an argument of split():

```
>>> line = "James;Miller;teacher;Perl"
>>> line.split(";")
['James', 'Miller', 'teacher', 'Perl']
```

The method split() has another optional parameter: maxsplit.
If maxsplit is given, at most maxsplit splits are done. This means that the resulting list will have at most "maxsplit + 1" elements.
We will illustrate the mode of operation of maxsplit in the next example:

```
>>> mammon = "The god of the world's leading religion. The chief
temple is in the holy city of New York."
>>> mammon.split(" ",3)
['The', 'god', 'of', "the world's leading religion. The chief
temple is in the holy city of New York."]
```

We used a Blank as a delimiter string in the previous example, which can be a problem: If multiple blanks or whitespaces are connected, split() will split the string after every single blank, so that we will get empty strings and strings with only a tab inside ('\t') in our result list:

```
>>> mammon = "The god  \t of the world's leading religion. The
chief temple is in the holy city of New York."
>>> mammon.split(" ",5)
['The', 'god', '', '\t', 'of', "the world's leading religion. The
chief temple is in the holy city of New York."]
>>>
```

We can prevent the separation of empty strings by using None as the first argument. Now split will use the default behaviour, i.e. every substring consisting of connected whitespace characters will be taken as one separator:

```
>>> mammon.split(None,5)
['The', 'god', 'of', 'the', "world's", 'leading religion. The
chief temple is in the holy city of New York.']
```

REGULAR EXPRESSION SPLIT

The string method split() is the right tool in many cases, but what, if you want e.g. to get the bare words of a text, i.e. without any special characters and whitespaces. If we want this, we have to use the split function from the re module. We illustrate this method with a short text from the beginning of Metamorphoses by Ovid:

```
>>> import re
>>> metamorphoses = "OF bodies chang'd to various forms, I sing:
Ye Gods, from whom these miracles did spring, Inspire my numbers
with coelestial heat;"
>>> re.split("\W+",metamorphoses)
['OF', 'bodies', 'chang', 'd', 'to', 'various', 'forms', 'I',
'sing', 'Ye', 'Gods', 'from', 'whom', 'these', 'miracles', 'did',
'spring', 'Inspire', 'my', 'numbers', 'with', 'coelestial',
'heat', '']
```

The following example is a good case, where the regular expression is really superior to the string split. Let's assume that we have data lines with surnames, first names and professions of names. We want to clear the data line of the superfluous and redundant text descriptions, i.e. "surname: ", "prename: " and so on, so that we have solely the surname in the first column, the first name in the second column and the profession in the third column:

```
>>> import re
>>> lines = ["surname: Obama, prename: Barack, profession:
president", "surname: Merkel, prename: Angela, profession:
chancellor"]
>>> for line in lines:
...     re.split(",* *\w*: ", line)
...
['', 'Obama', 'Barack', 'president']
['', 'Merkel', 'Angela', 'chancellor']
>>>
```

We can easily improve the script by using a slice operator, so that we don't have the empty string as the first element of our result lists:

```
>>> import re
>>> lines = ["surname: Obama, prename: Barack, profession:
president", "surname: Merkel, prename: Angela, profession:
chancellor"]
>>> for line in lines:
...     re.split(",* *\w*: ", line)[1:]
...
['Obama', 'Barack', 'president']
['Merkel', 'Angela', 'chancellor']
>>>
```

And now for something completely different: There is a connection between Barack Obama and Python, or better Monty Python. John Cleese, one of the members of Monty Python told the Western Daily Press in April 2008: "I'm going to offer my services to him as a speech writer because I think he is a brilliant man"

SEARCH AND REPLACE WITH SUB

```
re.sub(regex, replacement, subject)
```

Every match of the regular expression regex in the string subject will be replaced by the string replacement.
Example:

```
>>> import re
>>> str = "yes I said yes I will Yes."
>>> res = re.sub("[yY]es","no", str)
>>> print(res)
no I said no I will no.
```