Regular Expressions in Python

Regular expressions are a special syntax for describing textual patterns. If you are familiar with the UNIX command line, you will have used a technique known as *globbing* in order to match file and directory names. For example, the UNIX command

```
ls *.py
```

matches all files in the current working directory that end with the `.py` extension. The wild card character (`*`) matches any number of characters, so the whole pattern '*.py' matches all names that end in '.py'.

It is easy to confuse regular expressions with globbing, because both provide a means of matching textual patterns, but the syntax of the two is quite different, so try to keep them separate in your mind — globbing and regular expressions are not the same thing.

There are two basic operations that regular expressions are used for: searching and matching. Searching involves moving through a string to locate a sub-string that matches a given pattern, and matching involves testing a string to see if it conforms to a pattern.

To illustrate the difference, imagine first that you are trying to locate a number in a line of text. This requires a search, because you do not require the line to conform to a particular pattern; instead, you want to scan through the line looking for a particular pattern of digits.

Now consider that you want to verify that a particular string conforms to some predefined format. For example, an example might be 'HI454NNN'. You want to confirm that the text begins with two letters, is followed by some digits, and finishes with three letters. This is an example of matching: you want to see if the string *matches* a given pattern.

Python regular expressions are handled by the `re` package. After you have imported it, you have access to functions for searching (`search`, `findall`), matching (`match`), substituting strings (`sub`), and splitting strings (`split`). We will address each of these in due course, but first you need to know the basics of the regular expression syntax. We'll begin with a table of the most important pattern matching characters.

Characters	Description	Example
*	Matches zero or more of the preceding expression.	`a*` matches '', 'a', 'aa', 'aaa', etc
+	Matches one or more of the preceding expression.	`a+` matches 'a', 'aa', 'aaa', etc
.	Matches any single character, except the new line. (You can change this behavior by passing `re.DOTALL`.)	`.` matches 'a', 'b', '2', '(' etc

Characters	Description	Example
?	Matches zero or one of the preceding expression.	`a?` matches '' or 'a'
$	Matches the end of a string, or just before a new line.	
^	Matches the start of a string, or just after a new line.	
{m}	Matches exactly m instances of the preceding expression.	`a{2}` matches only 'aa'
[]	Matches any character, or character set (*eg*, \d), that appears in the square brackets.	`[a3t_]` matches 'a', '3', 't', or '_'.
|	Matches if either the preceding expression, or the expression that follows, matches.	`a|b` matches 'a' or 'b'
\b	Matches the start or end of a word.	
\s	Matches a whitespace character, including new lines and tabs.	
\d	Matches any digit, *ie*, 0–9.	
\w	Matches any alphanumeric character, or the underscore.	
\n	Matches a new line character.	
()	Group together terms in a single expression.	

There are many more regular expression operators, but you can go a long way with just those listed in the table. We will now consider them in more detail.

One of the most used regular expression characters is `+`; it matches one or more instances of an expression. Let's take a look at an example that makes use of this regular expression character in the `match` function:

```
>>> import re
>>> re.match('a+', 'aaa')
<_sre.SRE_Match object at 0x68950>
```

The first argument to the `match` function is the regular expression, and the second is the string to be checked for matching. If the regular expression matches at the start of the string, a `Match` object will be returned; otherwise, `None` will be returned. In this example, the expression `a+` matches one or more of the letter 'a', so a `Match` object gets returned.

The `Match` object contains information about the range of characters in the string that matched. Most of the time you don't need this detail, and it is only important to know whether there was a match or not. In such cases, you can use an `if` statement to check for a match.

```
>>> if re.match('a+', 'aaa'):
...     print 'It matched!'
...
It matched!
```

To show that the regular expression need only match at the start of the string, consider this

```
>>> re.match('a+', 'aaabbb')
<_sre.SRE_Match object at 0x689f8>
```

The output shows that 'aaabbb' is also a match, even though the regular expression does not match the letter 'b'. 'bbbaaa', on the other hand, does not match, because the pattern does not match at the start of the string.

```
>>> print re.match('a+', 'bbbaaa')
None
```

The `search` function is similar to `match`, but the match can occur anywhere in the string. Using the same regular expression and string as in the preceding example, the `search` function returns a `Match` object, corresponding to the first substring that matches the pattern.

```
>>> print re.search('a+', 'bbbaaa')
<_sre.SRE_Match object at 0x68950>
```

These simple examples could give you the idea that regular expressions are as primitive as command line globbing, but nothing could be further from the truth. Regular expressions are very powerful, and much of their power comes from the way you can combine operators into complex pattern matching expressions. For example, suppose you were searching a file for a line of text like this

```
  Wavelength (cm-1) :: 734.45
```

A pattern that matches such a line is

```
\s*Wavelength.*::\s*[\d\.]+
```

Let's dissect this to try to understand it. The pattern begins with `\s*`. The `\s` matches a whitespace character, and the `*` matches zero or more of the preceding expression. Taken together, the expressions match zero or more whitespace characters.

Next in the pattern is the text `Wavelength`. You can enter literal text like this in a regular expression. It will only match if exactly the same text is found in the string.

The character combination `.*` follows. This is similar to the `\s*` above, except that it matches zero or more of any character, not just whitespace characters.

Next we have the literal text `::`, followed again by `\s*`, which — as we now know — matches zero or more whitespace characters.

Lastly, consider the expression `[\d\.]+`. Square brackets match any of the characters they enclose. We wish to match all positive real numbers, so we could use a pattern like this `[0123456789]` to match any digit, but Python gives us some abbreviations for this. You can use ranges, like `[0-9]`, but you can also use `\d`, which matches any digit.

That covers the digits, but what about the decimal point? The period character has special meaning in regular expressions — as we have already seen — so you have to *escape* that meaning by using a backslash, similar to how you use a backslash to escape special meaning of characters in strings. With this in mind, `[\d\.]` will match any digit, or a decimal point. `[\d\.]+` thus matches one or more digits and/or decimal points, which are the characters that make up a real number.

Putting this altogether, the regular expression would thus read something like this in English:

> A string that begins with zero or more whitespace characters, followed by the text 'Wavelength', followed by zero or more characters of any type, followed by two colons and zero or more whitespace characters, and concluding with one or more digits and/or periods.

It's a mouthful, but hopefully this gives you some insight into how regular expressions work. Once you understand the strange syntax, you should realize they are just a language for describing textual patterns.

It's useful to be able to search and match patterns of text, because it allows you to locate the proverbial needle in a haystack, but when you have found that elusive line of text, you still need to extract the values you are interested in. Regular expressions has a means of doing that two: *groups*.

A group is nothing more than a section of a regular expression that is enclosed in parentheses. When the expression matches, the value matched by the group will be stored for later retrieval.

To demonstrate this, we'll return to the previous example, and modify the regular expression to use groups to retrieve the numeric value from the data.

```
\s*Wavelength.*::\s*([\d\.]+)
```

The only difference between this regular expression, and the previous, is the parentheses around the part of the expression that matches digits and periods, *ie*, the part designed to match the real number. With this small change, whenever a match occurs, the sub-string that matches the pattern in the parentheses will be stored so that we can retrieve it afterwards. Here's how: the `Match` object returned by functions like `match` and `search` includes a method called `group`; if you pass an index corresponding to a group, it will return the string that matched.

```
>>> r = '\s*Wavelength.*::\s*([\d\.]+)'
>>> s = '  Wavelength (cm-1) :: 734.45'
>>> match = re.match(r, s)
>>> match.group(0)
'  Wavelength (cm-1) :: 734.45'
>>> match.group(1)
'734.45'
```

Note that the group with index 0 is the part of the string that matched the whole regular expression. Thereafter, the indexes correspond to the order of groups in the regular expression. In this example, group number 1 holds the value we are interested in.

A regular expression can have as many groups as you like, and they can even be embedded. Consider the following data by way of example:

```
X  2.45 -3454.4443
```

Here is an expression that will match the line, and extract the label and numerical values.

```
(\w+)\s+((+|-)?\d*\.?\d*)\s+((+|-)?\d*\.?\d*)
```

This is quite a convoluted expression, so let's rewrite it in *verbose* mode.

```
(\w+)                  # Match and store the label
\s+                    # Skip whitespace
((+|-)?\d*\.?\d*)      # Match a real number, with optional sign. Store group
\s+                    # More whitespace
((+|-)?\d*\.?\d*)      # Another real number
```

Verbose mode allows you to spread out your regular expression, and use comments and whitespace to make it more legible. All whitespace and comments are ignored, unless explicitly escaped with a backslash.

Here is how you use a verbose regular expression:

```
import re
from string import rjust

# Setup regular expression string.
# Use a raw string to prevent any substitutions.
regEx = r"""
(\w+)                   # Match and store the label
\s+                     # Skip whitespace
((\+|-)?(\d*)\.?\d*)    # Match a real number, with optional sign. Store group
\s+                     # More whitespace
((\+|-)?(\d*)\.?\d*)    # Another real number
"""

# Call function with verbose flag
m = re.match(regEx, 'X  2.45 -3454.4443', re.VERBOSE)

# Print results from groups
print rjust('Label:', 20), m.group(1)
print rjust('First Value:', 20), m.group(2)
print rjust('Sign:', 20), m.group(3)
print rjust('Integer part:', 20), m.group(4)
print rjust('Second Value:', 20), m.group(5)
print rjust('Sign:', 20), m.group(6)
print rjust('Integer part:', 20), m.group(7)
```

This script prints out

```
              Label: X
        First Value: 2.45
               Sign: None
       Integer part: 2
       Second Value: -3454.4443
               Sign: -
       Integer part: 3454
```

To use the verbose mode, you pass an extra argument, and set it equal to the `VERBOSE` variable from the `re` module.

A regular expression may contain many groups, and they can even be embedded within one another. Given this fact, how do you know which group corresponds to which index in the `Match` object? The easiest way to establish the index of a group is to count the opening parentheses: the group at index 1 is the one with the first opening parenthesis when reading from left-to-right; the group with index 2 is the one with the next opening parenthesis, and so forth.

To make this discussion more concrete, take a look at the various `print` statements, and try to match the group index for each with the value printed. In particular, note how various aspects of the numeric values can be extracted by careful embedding of groups, including the complete number, its sign, and the integer part of the real number.

The numeric values are each matched by an expression that looks like this

```
((\+|-)?(\d*)\.?\d*)
```

Each one has three groups in all. The first encloses the whole expression, and will thus take on the value of the complete number. The second, `(\+|-)`, matches either the plus symbol — which must be escaped due to its special meaning in regular expressions — or the negative symbol. If a sign is given in the string, its value will end up in the corresponding group; if no sign is given, the group will get the value `None`. The last group matches the integer value of the number, which appears before the decimal point.

This is an advanced example which hopefully conveys just how powerful regular expressions can be. A single expression can be used to carve up a textual string, extracting any useful information, and storing it in groups for later use.

We saw above that the regular expression functions support an optional third argument, which can be used to pass in flags like `re.VERBOSE`. These flags are particularly useful if you want to match strings that extend over several lines. Consider this data, for example:

```
Irreducible Representations, including subspecies
-------------------------------------------------
S
P:x  P:y  P:z
D:z2  D:x2-y2  D:xy  D:xz  D:yz
F:z3  F:z  F:xyz  F:z2x  F:z2y  F:x  F:y


Configuration of Valence Electrons
==================================

         Occupation Numbers
         -------------------------------------------------
S        1
P        0
D        0
F        0
         -------------------------------------------------
```

Now suppose you are interested in extracting the block of text that begins after the horizontal rule following 'Occupation Numbers'. This is clearly a multi-line piece of string. Here is how you could do it.

```
import re, sys

m = re.search(r'Occupation Numbers\s*-*(.*?)-', sys.stdin.read(),
    re.MULTILINE | re.DOTALL)
print m.group(1)
```

When run, and passed the data above via standard input, this script produces

```
S        1
```

```
P        0
D        0
F        0
```

There are a number of aspects of this short script that warrant discussion. First, a number of flags are passed via the third argument to the `search` function. You can pass multiple flags by combining them with the | operator. The `re.MULTILINE` option causes the `^` and `$` operators to match wherever new line characters are found, rather than just at the start and end of the string. This isn't strictly necessary in this particular case, because neither of these characters appear in the regular expression. But should the expression be altered in the future, the multiline behavior would be desirable, so it has been included anyway.

The `re.DOTALL` flag is more important: it causes the `.` operator, which is a single period, to match all characters *including* the new line. Usually, the `.` operator does not match the new line character, but in multiline matching it is useful for the new line to be treated just like any other character.

The regular expression also has some interesting aspects to it.

```
Occupation Numbers\s*-*(.*?)-
```

It begins with the text 'Occupation Numbers', followed by some whitespace (`\s*`) and zero or more hyphens(`-*`). Together, these expressions form a 'landmark': it is quite common when scripting to extract a small section of data from a large file. A way to do this is to look for a unique sequence of characters just before and just after the section of interest. These allow you to anchor your regular expression, and extract the desired text.

The terminal landmark in this case is the line of hyphens under the section of interest. A single hyphen has been included at the end of the regular expression, because once that has been found, we know that the block of text is finished.

A group has been used to capture the section of text we are interested in. It looks like this

```
(.*?)
```

As we have already seen, `.*` matches zero or more characters, but what role is the `?` playing in this case? The question mark actually modifies the behavior of the `*`, causing it to become *non-greedy*. Regular expressions usually try to match as much as possible — they are said to be *greedy*. If you want them to match the minimum possible, you need to make them non-greedy by using the `?` character.

What would happen if you didn't use the non-greedy operator in this case? `.*` will match zero or more of *any* character, since we are using the `re.DOTALL` flag, so it would simply match everything to the end of the string, including the line of hyphens, and anything else that might appear afterwards. This is clearly not the behavior we are looking for. We want to match as few characters as possible to get to the first of the trailing hyphens, and the non-greedy operator helps achieve this.

Thus far, we have looked at regular expressions that get used once, and then discarded. Sometimes you will need to use a regular expression repeatedly. To improve performance, and the need to duplicate the regular expression text, it is possible to *compile* an expression and store it in regular expression object. Compiling involves taking the string representation of the regular expression, and converting that into an internal form that can be applied much faster.

Here is an example of compiling and applying a regular expression object.

```
import re
from string import ljust

dateEx = re.compile(r'''
    ^([A-Z][a-z]{2})        # Match a month (eg Jan, Feb)
    \s+                     # Skip whitespace
    (\d{1,2})               # Match date (eg 1, 2, 10)
    ,\s*                    # Match comma, and optional whitespace
    (\d{4})$                # Match year (eg 1999, 2008)
    ''', re.VERBOSE)

dates = ['Jan 23, 1999', 'jan 23, 1999', '23 Jan, 1999', 'Jan 23, 99']
for l in dates:
    m = dateEx.match(l)
    print 40*'-'
    print l
    if m:
        print 'Correct Date Format'
        print ljust('Month',20), m.group(1)
        print ljust('Day',20), m.group(2)
        print ljust('Year',20), m.group(3)
    else:
        print 'Incorrect Date Format'
```

In this example, which checks the validity of a number of date strings, rather than passing the regular expression string directly to the `match` function, the `compile` function is used to create a regular expression object. `compile` takes both the regular expression string, and the flags (*eg* `VERBOSE`), as arguments. The script then calls the `match` method of the object, rather than the `match` function, to apply the regular expression to a given string.

The output of the script is

```
----------------------------------------
Jan 23, 1999
Correct Date Format
Month                Jan
Day                  23
Year                 1999
----------------------------------------
jan 23, 1999
Incorrect Date Format
----------------------------------------
23 Jan, 1999
Incorrect Date Format
----------------------------------------
Jan 23, 99
Incorrect Date Format
```

It identifies the first date as being correctly formatted, and extracts strings for the month, day, and year. The other dates are all incorrectly formatted.

(A small aside: the horizontal rules are generated by passing `40*'-'` to the print command. In Python, you can do such a 'multiplication' to repeat strings, in this case generating 40 hyphens.')

You can do a lot just with the `search` and `match` functions/methods, but there are a few other very useful functions in the `re` package. The first is `split`, which is similar to the `string` module `split` function, but more powerful. You use it to split up a string into components. For example, take this string:

```
XXX,36346, 6633.334, -1
```

This may seem trivial enough, but the `string` modules's `split` function would have trouble, because it can only work with either whitespace-delimited components, or components separated by a constant string. In this case, each component is separated by a comma and zero or more spaces.

With the `split` function from the `re` module, you can use a regular expression to define the separator, like this

```
>>> re.split(r',\s*', 'XXX,36346, 6633.334, -1')
['XXX', '36346', '6633.334', '-1']
```

The first argument is the regular expression, in this case matching a comma followed by zero or more whitespace characters. The second argument is the string to be split. The result is a list of the string components, just as you get when using `string.split`.

The `search` function allows you to locate a single sub-string matching a given regular expression, but what if you want to locate many such sub-strings? You could apply the `search` function repeatedly, each time passing in what remains of the string to be searched, but this is a bit clumsy. A better solution is to use the `findall` function, which locates all non-overlapping matches, and returns them in a list.

```
import re

data = """
Coordinates
  H 3.234 34.3 55.
  O 3.234 14.3 12.
  Zn 3.234 34.2 55.2

Other
  Sn 3.234 34.2 55.2
  Pd -3.23 34.2 55.2
"""

numPattern = r'\s+([\+\-]?\d*\.?\d*)'   # Matches a real number with leading space
regEx = re.compile(r'''
    ^\s*                            # Skip whitespace at start of line
    [A-Z][a-z]?                     # Match a chemical symbol
    %s%s%s                          # Three numbers, each preceded by whitespace
    \s*$                            # Optional whitespace, and end of line
    ''' % (numPattern, numPattern, numPattern),
    re.VERBOSE | re.MULTILINE)


for m in regEx.findall(data):
    print m
```

The output of this script is

```
('3.234', '34.3', '55.')
```

```
('3.234', '14.3', '12.')
('3.234', '34.2', '55.2')
('3.234', '34.2', '55.2')
('-3.23', '34.2', '55.2')
```

The script is designed to extract three coordinate values from any line in the data that matches a particular format, beginning with a chemical element symbol, and followed by three real numbers.

The regular expression is quite involved. Note how it has been simplified somewhat by extracting the real number pattern — which gets repeated — into a variable, and using string substitution to form the regular expression. Without this the expression would be less readable and maintainable. Consider doing this in your own scripts: reduce complexity by moving parts of your regular expressions into variables, and using string operators to combine them into a single string.

The return value of `findall` is a list. If there are multiple groups in the regular expression, such as is the case here, each entry in the list will be a tuple corresponding to a particular match, and each will contain the groups for the match. In the example above, each entry in the list is a tuple containing three strings, corresponding to the three coordinate values of the atoms in the data.

The last function that we will cover is `sub`. This allows you to search for, and replace, sequences of characters that match a given regular expression. For instance, imagine you have a program in which has many labels of the form `MT...`, such as `MTWaveFunction` and `MTOptimizer`. You wish to replace the `MT` in each label with `TM`. How can you do this swiftly and safely?

With the `sub` function, you can identify labels of the correct form, and transform them, like so

```
import re

code = """
waveFunc = MTWaveFunction()
waveFunc += 5.0
opt = MTOptimizer(waveFunc)
"""

print re.sub(r'\bMT(\w+)\b', r'TM\1', code)
```

The output is

```

waveFunc = TMWaveFunction()
waveFunc += 5.0
opt = TMOptimizer(waveFunc)

```

The `sub` function takes three arguments: the regular expression to replace; the string to replace it with, and the string to search through and modify. The regular expression in this example is quite straightforward:

```
\bMT(\w+)\b
```

This matches a word boundary (`\b`), followed by the letters `MT`, followed by one or more alphanumeric or underscore characters, and finishing with another word boundary (\b). This describes the labels we are trying to transform.

Parentheses have been added around the pattern that matches the second half of the label, following the `MT` prefix. This creates a group which stores the matched sub-string. The reason for doing this is that you can access any groups matched in the regular expression from within the substitution string. To do this, you simply supply a backslash, followed by the index of the group.

In the example above the substitution string is `TM\1`, which means 'replace the matched string with `TM` followed by the first group from the match'. The first group from the match was the text that followed `MT`, so the net effect is to swap `TM` for `MT`.

This is quite a simple example of substitution, but you can do some very powerful manipulations using regular expressions, and some astute use of grouping.

We will finish off this section on regular expressions with a warning, best encapsulated in the following quote:

> Some people, when confronted with a problem, think "I know, I'll use regular expressions." Now they have two problems.
>
> Jamie Zawinski

Regular expressions are powerful, but they are not suitable for every situation. Not only that, they can be difficult to write — even for experienced developers — and even more difficult to read. By all means use them, but don't use them where a better solution already exists. (For example, don't parse XML documents with regular expressions. Use a specialized XML parser, as described in the next sub-section.)

Exercise: Getting regular

Come up with regular expression to match the following date format:

```
17/05/2009 8:15
```

Test your regular expression using the `re.match` function on the string above.

Exercise: Groupies

Introduce groups in the regular expression from the script in the previous exercise, to extract the hour of the day, and the minutes. Use these values to calculate how many seconds have passed since midnight, and print out the answer.

Exercise: Needle in haystack

Consider the following data:

```
**********************
* T E C H N I C A L *
**********************
```

```
============================================================
P A R A L L E L I Z A T I O N  and  V E C T O R I Z A T I O N
============================================================

Nr of parallel processes:                                   1
Internal max. (compile-time) nr of processes:               8
Maximum vector length in NumInt loops:                    128



===============
I O  vs.  C P U  ***  (store numerical data on disk or recalculate)  ***
===============

Basis functions:   recalculate when needed
Fit functions:     recalculate when needed

IO buffersize (Mb):                                     64.000000



=====================
S C F   U P D A T E S
=====================

Max. nr. of cycles:                                100
Convergence criterion:                               0.0000000100
  secondary criterion:                               0.0000000100

Mix parameter (when DIIS does not apply):            0.2000000000
Special mix parameter for the first cycle:           1.0000000000
```

Write a script that uses regular expressions to extract and print the number given for the 'IO buffersize'.

Exercise: Pick up sticks

Write a script that uses the `re.findall` function to match and extract data values from lines of the following form:

```
XY19 : 23.4 -234.0 9854.0, 645.345 34453 34.3 b=b1
```

Your script should extract the label at the beginning, each of the numbers before the comma, each of the numbers after the comma, and the string on the right of the = sign (`b1` in above example). Test your script on this data:

```
XY19 : 23.4 -234.0 9854.0, 645.345 34453 34.3 b=b1
XY19 : 23.4 -234.0 9854.0, 645.345 34453 34.3 b=b1
XY19

Elevation
---------------
YY19 : 2.4 -234.0 984.0, 645.345 3445 34. b=b3
XY20 : 3.4 -24.0 9854.0, 65.345 3453 34.3 b=a1
----
```

Print out the extracted values, and confirm that they are correct.

Exercise: The splits

Use the `re.split` function with an appropriately formed regular expression to extract the numbers from the following line of data:

```
45, 3453 : 19, -1.e-10
```

Your script should be able to handle the case that any of the numbers are in exponential form (such as the last number shown above).

Exercise: No substitute for practice

Imagine you have a script that names variables with a leading underscore, like this: `_someVar`. You decide you want to remove the leading underscore, and use a trailing underscore instead, like this: `someVar_`. Write a short script that uses the `re.sub` function to achieve this transformation.

Come up with a small amount of trial data, and test your script on it.