Lecture 6.3 : Regular expressions

Introduction ¶

- Assume we have downloaded the CA117 classlist from the web. Further assume that the classlist is delivered to us as a single HTML document. We would like to e-mail all students in the class. Included in the HTML document is every student's email address. So far so good. Unfortunately, the document is "noisy": 95% of it is HTML mark-up that is of no interest to us and that obscures the e-mail addresses sprinkled throughout the document.
- What can we do? We could manually scroll through the document looking for e-mail addresses and copy them to a list. That would be a tedious and error-prone task however. Is there an easier way?
- What we would like to be able to do is specify a "pattern" or "template" for the information that is of interest to us and extract just that information. If we could specify a general pattern that every e-mail address follows and then extract everything in the document that matches that pattern then we would have a list of just the required e-mail addresses.

Regular expressions

- Regular expressions are used to specify patterns for entities we wish to locate and match in a larger string. Examples might be dates, times, e-mail addresses, names, credit card numbers, social security numbers, directory paths, file names, etc.
- Once we have defined a suitable regular expression we can ask questions such as the following:
  - Does a given string match the specified pattern?
  - Is there a match for this pattern anywhere in the given string?
- We can also efficiently find all substrings of a larger string that match the specified pattern. (This would seem ideal for our task: if we treat the HTML document as a single large string our task is to extract every substring from it that matches the pattern of an e-mail address.)

Defining patterns

- The simplest of patterns takes the form of an ordinary string. For example the pattern 'cat' can match any occurrence of the substring 'cat'. Let's try it:

```
>>> from re import findall # We first import the findall function from the re (
>>> s = 'A catatonic cat sat on the mat. Catastrophe!'
>>> p = r'cat' # Build a regular expression
>>> m = findall(p, s) # Try to match the pattern in the string s
>>> m # How many matches did we get?
['cat', 'cat']
```

- We defined a regular expression (`r'cat'`) to match occurrences of the pattern 'cat' and we called this regular expression `p` (for pattern). When defining a pattern we *always* precede it with 'r' in order to indicate to Python that this is a *raw string* (i.e. we do not want Python imposing its own interpretation on any special sequences that might arise in the pattern).

- We tried to match this pattern against the string `s` by calling `findall()`. The latter function returns a list of all substrings of `s` that match the defined pattern. Two matches are returned as we might expect.

Character classes

- We can define *character classes* to be matched against. The character class `[abc]` will match any *single* character 'a', or 'b' or 'c'. The character class `[a-z]` will match any *single* character 'a' through 'z'. The class `[a-zA-Z0-9]` will match any alphanumeric character.

- Let's use a character class to match instances of both 'cat' and 'Cat' in our example above:

```
>>> p = r'[Cc]at' # Match 'C' or 'c' followed by 'at'
>>> findall(p, s)
['cat', 'cat', 'Cat']
```

- We can negate character classes using the '^' symbol. For example:

```
>>> p = r'[^Cc]at' # Match anything but 'C' or 'c' followed by 'at'
>>> findall(p, s)
['tat', 'sat', 'mat']
```

- The above pattern matches any substring which consists of *a character which is neither 'c' nor 'C' followed by 'at'*.

Special characters and sequences

- Most characters simply match themselves. Exceptions to this rule are metacharacters. Metacharacters are special characters that do not match themselves but signal that something else should be matched. Here are three common examples:

Metacharacter	Matches
^	Matches the beginning of a string
$	Matches the end of a string
.	Matches any character (except a new line)

- In addition to defining our own character classes we can call upon a predefined set of character classes when constructing regular expressions. Such predefined classes are accessed using *special sequences*. Examples are given below:

Sequence	Matches
\d	Matches any decimal digit
\D	Matches anything not a digit
\s	Matches any whitespace character (e.g. space, tab, newline)
\S	Matches any non-whitespace character
\w	Matches any alphanumeric character
\W	Matches any non-alphanumeric character

Sequence	Matches
\b	Matches any word boundary (a word is an alphanumeric sequence of characters)

- Below are some examples of using these special sequences in regular expressions:

```
>>> p = r'\d' # Match one digit
>>> findall(p, '1')
['1']
>>> findall(p, 'a')
[]
>>> p = r'\D' # Match one non-digit
>>> findall(p, 'a')
['a']
>>> findall(p, '3')
[]
>>> p = r'\s' # Match one whitespace character
>>> findall(p, ' ')
[' ']
>>> findall(p, 'x')
[]
>>> findall(p, '\n')
['\n']
>>> p = r'\S' # Match one non-whitespace character
>>> findall(p, 'a')
['a']
>>> findall(p, ' ')
[]
>>> findall(p, '\t')
[]
>>> p = r'\w' # Match one alphanumeric character
>>> findall(p, 'a')
['a']
>>> findall(p, '2')
['2']
>>> findall(p, '') # The empty string is not a character
[]
>>> findall(p, '>')
[]
>>> p = r'\W' # Match one non-alphanumeric character
>>> findall(p, 'a')
[]
>>> findall(p, '2')
[]
>>> findall(p, '') # The empty string is not a character
[]
>>> findall(p, '>')
['>']
```

Repeating a pattern zero times or once

- We can match a pattern zero times or once with the '?' metacharacter. Thus we can use '?' to effectively make a pattern *optional*:

```
>>> p = r'do?g'
>>> findall(p, 'dg')
['dg']
>>> findall(p, 'dog')
['dog']
>>> findall(p, 'doog')
[]
```

Repeating a pattern a fixed number of times

- With regular expressions we can match portions of a pattern multiple times. We do so by specifying the number of required matches inside curly brackets. For example:

```
>>> p = r'a{3}' # Match 'a' 3 times
>>> findall(p, 'a')
[]
>>> findall(p, 'aa')
[]
>>> findall(p, 'aaa')
['aaa']
>>> findall(p, 'aaaa')
['aaa']
>>> p = r'cat{3}' # Match 'ca' followed by 3 't' characters
>>> findall(p, 'cat')
[]
>>> findall(p, 'catt')
[]
>>> findall(p, 'cattt')
['cattt']
>>> findall(p, 'catttt')
['cattt']
```

- If our pattern contains a *group* of characters that must be matched some number of times then we need to enclose the pattern with `(?:` on the left hand side and `)` on the right hand side. For example:

```
>>> p = r'(?:cat){3}' # Match 'cat' 3 times
>>> findall(p, 'dog')
[]
>>> findall(p, 'cat')
[]
>>> findall(p, 'catcat')
[]
>>> findall(p, 'catcatcat')
['catcatcat']
>>> findall(p, 'catcatcatcat')
catcatcat
>>> findall(p, 'catcatcatcatcatcat')
['catcatcat', 'catcatcat']
```

- If we need to match a pattern at *least* a number of times *m* and at *most* a number of times *n* then we write it `{m, n}`. For example:

```
>>> p = r'(?:cat){2,4}dog'
>>> findall(p, 'catdog')
[]
>>> findall(p, 'catcatdog')
['catcatdog']
>>> findall(p, 'catcatcatdog')
['catcatcatdog']
>>> findall(p, 'catcatcatcatdog')
['catcatcatcatdog']
>>> findall(p, 'catcatcatcatcatdog')
['catcatcatcatdog']
```

Repeating a pattern an arbitrary number of times

- Above we are specifying an exact number of required matches. Some *metacharacters* allow us to specify an *arbitrary* number of matches. One such metacharacter for specifying a repeated pattern is '*'. The '*' metacharacter signifies that the preceding pattern can be matched zero or more times (instead of exactly once).

- Here is an example of the '*' metacharacter in action:

```
>>> p = r'do*g' # Match 'd' followed by zero or more 'o's followed by 'g'
>>> findall(p, 'dg')
['dg']
>>> findall(p, 'dog')
['dog']
>>> findall(p, 'doog')
['doog']
```

- Another metacharacter for specifying a repeated pattern is '+'. It signifies that the preceding pattern must can be matched an arbitrary number of times but *must be matched at least once*. Note the difference between '*' and '+': with '*' the specified character may not be present at all while with '+' the specified character must be present at least once.

- Here is an example of the '+' metacharacter in action:

```
>>> p = r'do+g'
>>> findall(p, 'dg')
[]
>>> findall(p, 'dog')
['dog']
>>> findall(p, 'doog')
['doog']
```

Examples

- We have just scratched the surface with regular expressions. They are a mini-programming language in themselves. Even with what we have covered so far however there are some useful things we can do…

- Consider the contents of the file `matches.txt`:

```
$ cat matches.txt
Jimmy arrived in work at 3.45pm. His phone number is 087 4567890.
Or you can email him at jimmy.murphy@computing.dcu.ie. Mary arrived
at 9.12am. Her phone number is 085 2345678. She can be contacted at
mary.oneill2@rte.ie. Wendy arrived at 11:18am. Her email address is
wendy@google.com. Her phone number is 086 1234567. Jimmy earns 2.00
euro per hour. Mary earns 14.50 euro per hour. Wendy earns 36.00 euro
per hour. Some people like to include hyphens in their phone numbers,
087-6213344, for example. Valid phone numbers begin with 086 or 087
or 085. This is not a phone number 111 1234567. Examples of invalid
times include 3.71am, 30:19pm and 12.3pm and we do not want to match
these when looking for times. Examples of valid times are 1.59am and
12.00pm. We do not allow leading zeros in the hour part of the time
so 04.13am is invalid, rather it should be 4.13am. Long dates look
```

```
like 1 January 2014 and 18 March 2016. Is this a valid phone number:
123087 66543789920?
```

- Let's try to extract all of the mobile phone numbers from the file:

```
>>> from re import findall
>>> fin = open('matches.txt')
>>> s = fin.read()
>>> phone = r'\b\d{3}\s\d{7}\b'
>>> findall(phone, s)
['087 4567890', '085 2345678', '086 1234567', '111 1234567']
```

- We have two problems. Firstly we are failing to collect phone numbers that have a hyphen between their two components. Secondly we are collecting numbers that do not look like phone numbers.
- Let's first collect phone numbers that include hyphens:

```
>>> phone = r'\b\d{3}[-\s]\d{7}\b'
>>> findall(phone, s)
['087 4567890', '085 2345678', '086 1234567', '087-6213344', '111 1234567']
```

- Now let's insist that a phone number begin with one of *085*, *086* or *087*:

```
>>> phone = r'\b(?:085|086|087)[-\s]\d{7}\b'
>>> findall(phone, s)
['087 4567890', '085 2345678', '086 1234567', '087-6213344']
```

- Let's extract all the times from the file:

```
>>> time = r'\b\d{1,2}[.:]\d{2}[ap]m\b'
>>> findall(time, s)
['3.45pm', '9.12am', '11:18am', '3.71am', '30:19pm', '1.59am', '12.00pm', '04.1
```

- How can we exclude invalid times? We need a regular expression that matches only hours 1-12 and only minutes 00-59. We can do so as follows:

```
>>> time = r'\b(?:[1-9]|1[0-2])[.:](?:0[0-9]|[1-5][0-9])[ap]m\b'
>>> findall(time, s)
['3.45pm', '9.12am', '11:18am', '1.59am', '12.00pm', '4.13am']
```

- Let's extract all the rates of pay from the file:

```
>>> pay = r'\b\d{1,2}\.\d{2}\s\euro\b'
>>> findall(pay, s)
['2.00\neuro', '14.50 euro', '36.00 euro']
```

- Let's extract all the e-mail addresses from the file:

```
>>> email = r'\b(?:\w+\.)*\w+\@\w+\.\w+(?:\.\w+)*\b'
>>> findall(email, s)
```

```
['jimmy.murphy@computing.dcu.ie', 'mary.oneill2@rte.ie', 'wendy@google.com']
```

- Let’s extract all the long dates from the file:

```
>>> ldate = r'\b\d{1,2}\s(?:January|February|March|April|May|June|July|August|S
>>> findall(ldate, s)
['1 January 2014', '18 March 2016']
```