Lecture 6.1 : Regular expressions

Introduction

- Imagine we have downloaded the CA117 classlist from the web as an HTML document.
- We would like to e-mail all students in the class and 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 obscures the e-mail addresses scattered 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?
- It would be great if we could specify a "pattern" or "template" that matched and extracted just the information in the document that is of interest to us i.e. e-mail addresses.
- 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 allow us to do just that!

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: Is there a match for this pattern anywhere in the given string?
- Regular expressions also allow us to 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.
- Below we define a regular expression `r'cat'` to match occurrences of the pattern 'cat' and we call 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* (this prevents Python imposing its own interpretation on any special sequences that might arise in the pattern).
- We 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, as we might expect, are returned.

```python
# We want to find all matches so import the required function
from re import findall

# We will look for matches in here
s = 'A catatonic cat sat on the mat. Catastrophe!'

# Define our pattern in a raw string
p = r'cat'

# Match and print the result
print(findall(p, s))
```

```
['cat', 'cat']
```

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'.

```python
# We will look for matches in here
s = 'A catatonic cat sat on the mat. Catastrophe!'

# Match one of 'C' or 'c' followed by 'at'
p = r'[Cc]at'

# Match and print the result
print(findall(p, s))
```

```
['cat', 'cat', 'Cat']
```

Character class negation

- We can negate character classes by preceding them with the ^ symbol.

```python
# We will look for matches in here
s = 'A catatonic cat sat on the mat. Catastrophe!'

# Match anything but 'C' or 'c' followed by 'at'
p = r'[^Cc]at'

# Match and print the result
print(findall(p, s))
```

```
['tat', 'sat', 'mat']
```

Sequences

- 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*.

Sequence	Matches
\d	Matches any decimal digit
\D	Matches any non-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
\b	Matches any word boundary (a word is an alphanumeric sequence of characters)

```python
# Match one digit
p = r'\d'
print(findall(p, '1 and 2 and 34'))
print(findall(p, 'No digits here'))
```

```
['1', '2', '3', '4']
[]
```

```python
# Match one non-digit
p = r'\D'
print(findall(p, '1 and 2 and 34'))
print(findall(p, 'No digits here'))
```

```
[' ', 'a', 'n', 'd', ' ', ' ', 'a', 'n', 'd', ' ']
['N', 'o', ' ', 'd', 'i', 'g', 'i', 't', 's', ' ', 'h', 'e', 'r', 'e']
```

```python
# Match one whitespace character
p = r'\s'
print(findall(p, '1 and 2 and 34'))
print(findall(p, 'No digits here'))
print(findall(p, '1\n2\t3'))
```

```
[' ', ' ', ' ', ' ']
[' ', ' ']
['\n', '\t']
```

```python
# Match one non-whitespace character
p = r'\S'
print(findall(p, '1\n2\t3'))
```

```
['1', '2', '3']
```

```
# Match one alphanumeric character
p = r'\w'
print(findall(p, '1 and 2 and 34'))
print(findall(p, '1\n2\t3'))

# Match one non-alphanumeric character
p = r'\W'
print(findall(p, '1 and 2 and 34'))
print(findall(p, '1\n2\t3'))
print(findall(p, '1 < 3'))
```

```
['1', 'a', 'n', 'd', '2', 'a', 'n', 'd', '3', '4']
['1', '2', '3']
[' ', ' ', ' ', ' ']
['\n', '\t']
[' ', '<', ' ']
```

Metacharacters

- Most characters simply match themselves.
- Exceptions 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)

A pattern that occurs once or zero times

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

```
# Match US and IE spelling
p = r'colou?r'
print(findall(p, 'In America they spell it color'))
print(findall(p, 'Over here we spell it colour'))
```

```
['color']
['colour']
```

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.

```python
# Match a date
p = r'\d{2}[-/]\d{2}[-/]\d{2}'
print(findall(p, 'Christmas falls on 25-12-21'))
print(findall(p, "Valentine's Day is 14/02/21"))
```

```
['25-12-21']
['14/02/21']
```

Groups

- 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.

```python
# Match a date
p = r'(?:\d{2}[-/]){2}\d{2}'
print(findall(p, 'Christmas falls on 25-12-21'))
print(findall(p, "Valentine's Day is 14/02/21"))
```

```
['25-12-21']
['14/02/21']
```

Repeating a pattern at least M and at most N times

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

```python
# The more o's in your yahoo the happier you are
p = r'Yahoo{1,3}!'
print(findall(p, 'Yahoo!'))
print(findall(p, 'Yahooo!'))
print(findall(p, 'Yahoooo!'))
print(findall(p, 'Yahooooo!')) # Too happy to match
```

```
['Yahoo!']
['Yahooo!']
['Yahoooo!']
[]
```

Repeating a pattern zero or more times

- Some *metacharacters* allow us to handle 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.

```python
p = r'Yabba Dabba Do*!'
print(findall(p, 'Yabba Dabba Do!'))
print(findall(p, 'Yabba Dabba Doo!'))
print(findall(p, 'Yabba Dabba Doooooooooo!'))
print(findall(p, 'Yabba Dabba D!')) # We probably shouldn't match this
```

```
['Yabba Dabba Do!']
['Yabba Dabba Doo!']
['Yabba Dabba Doooooooooo!']
['Yabba Dabba D!']
```

Repeating a pattern one or more times

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

```python
p = r'Yabba Dabba Do+!'
print(findall(p, 'Yabba Dabba Do!'))
print(findall(p, 'Yabba Dabba Doo!'))
print(findall(p, 'Yabba Dabba Doooooooooo!'))
print(findall(p, 'Yabba Dabba D!')) # We no longer match this
```

```
['Yabba Dabba Do!']
['Yabba Dabba Doo!']
['Yabba Dabba Doooooooooo!']
[]
```

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…

```python
s = """
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 above string.

```
phone = r'\b\d{3}\s\d{7}\b'
print(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'
print(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'
print(findall(phone, s))
```

```
['087 4567890', '085 2345678', '086 1234567', '087-6213344']
```

- Let’s extract all the times.

```
time = r'\b\d{1,2}[.:]\d{2}[ap]m\b'
print(findall(time, s))
```

```
['3.45pm', '9.12am', '11:18am', '3.71am', '30:19pm', '1.59am', '12.00pm', '04.13am', '
```

- 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'
print(findall(time, s))
```

```
['3.45pm', '9.12am', '11:18am', '1.59am', '12.00pm', '4.13am']
```

- Let's extract all the rates of pay.

```
pay = r'\b\d{1,2}\.\d{2}\seuro\b'
print(findall(pay, s))
```

```
['2.00\neuro', '14.50 euro', '36.00 euro']
```

- Let's extract the e-mail addresses.

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

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

- Let's extract long dates.

```
ldate = r'\b\d{1,2}\s(?:January|February|March|April|May|June|July|August|September|
print(findall(ldate, s))
```

```
['1 January 2014', '18 March 2016']
```