# Script language: Python
## Regular Expressions


Cédric Saule

Technische Fakultät
Universität Bielefeld

12. Februar 2016

# Regular expressions

What we want to do WITHOUT using any text editor. (`CTRL + F`) is not a text search:

- Reading text files line per line.

# Regular expressions

What we want to do WITHOUT using any text editor. (`CTRL + F`) is not a text search:

- Reading text files line per line.
- Search lines for patterns.

# Regular expressions

What we want to do WITHOUT using any text editor. (`CTRL + F`) is not a text search:

- Reading text files line per line.
- Search lines for patterns.
- Sentences become subwords.

# Regular expressions

What we want to do WITHOUT using any text editor. (`CTRL + F`) is not a text search:

- Reading text files line per line.
- Search lines for patterns.
- Sentences become subwords.
- Replaces pieces of text.

# Regular expressions

- The simpliest Chomsky family languages.
- In Python: *Regular Expressions* (RE) are close to the Perl syntax.
- Regular Expressions in Python are significantly more powerful than regular languages.

# Regular expressions

We work with the module `re` that provides us the functionality RE available in Perl.

In Python, everything starts with an object `pattern` that provides the appropriate functions.

```
> import re

> story = 'In a hole in the ground there lived a boggit.'
> p = re.compile(r"in")

> m = p.match(story)      #Looks at the beginning of the string
> m                       #No result -> None
> m = p.search(story)     #Looks in the entire string
> m                       #Object match
<_sre.SRE_Match at 0x1042a9648>
```

# Object pattern

The pattern is built with `re.compile(<RE_STR>, <FLAGS>)`.

- The pattern begins with `r` followed by the string RE (eg: `re.compile(r"[a-z]*")`).
- Most frequently used *Modifier* (flag):

  re.IGNORECASE Ignore the letters' case.

From the above example follows:
`re.compile(r "[a-z]*", re.IGNORECASE)`.

# Object match

The following methods are defined in the objects `match`.

- group() Returned value: The matched string.
- start() Returned value: Start index of the match.
- end() Returned value: End index of the match.
- span() Returned value: Start-/End index as tuple.

# Object match

If a match is found, an object `match` is returned, otherwise `None`.

General procedure for RE processing in Python.

```
> p = re.compile(<PATTERN>)
> m = p.match( 'string␣goes␣here' )
> if m:
>     print 'Match␣found:␣', m.group(), '␣with␣indices␣', m.span()
> else:
>     print 'No␣match'
```

# Pattern – findall() & finditer()

To find all occurrences, use `findall()` and `finditer()`:

```
> p = re.compile('\d+')
> p.findall('12␣drummers␣drumming,␣11␣pipers␣piping,
␣␣10␣lords␣a-leaping')
['12', '11', '10'] #All patterns found are listed

> iterator = p.finditer('12␣drummers␣drumming,␣11␣...␣10␣...')
> iterator #Iterator on a match object.
<callable-iterator object at 0x...>
> for match in iterator:
>     print match.span()
(0, 2)
(22, 24)
(29, 31)
```

# sub() – Pattern replacement

With `sub(<REPL>, <STR>[, count=0])` (*Substitute*) Pattern can be replaced with `REPL`. The maximum number of replacements can be specified by `count`.

```
> t = '12 drummers drumming, 11 pipers piping, ...'
> p = re.compile('umm')

> p.sub("ift", t)
'12 drifters drifting, 11 pipers piping, ...'

> p.sub("rift", t, count=1)
'12 drifters drumming, 11 pipers piping, ...'
```

# Exercise – Search and replace

The text files listed below can be found in */vol/lehre/python/*. They are plays by Wayne Anthoney.

- How many lines of *romeo.txt* contains the word „Gold“?
- Give the respective index positions of hits per line.
- Replace in the text *eric.txt* the word „Estragon“ by „Basilic“ and „Vladimir“ by „Ilitch“.

# RegEx

- Alternative: `r"Huey|Dewey|Louie"`
- Grouping: `r"(Hu|Dew)ey|Louie"`
- Quantifiers:

```
r"ab?a" # aa, aba
r"ab*a" # aa, aba, abba, abbba, abbbba, ...
r"ab+a" # aba, abba, abbba, abbbba, ...
r"ab{3,6}a" # abbba, abbbba, abbbbba, abbbbbba
r"a(bab)+a" # ababa, ababbaba, ababbabbaba, ...
```

# RegEx

- Character classes

```
r"hello\s+world" # whitespace
r"es␣ist␣\d+␣Uhr" # digits
r"name:␣\w+" # letters (words)
```

- „Opposites“: `\S`, `\D`, `\W`
- Self-created character class

```
r"M[ea][iy]er" # Meier, Meyer, Maier, Mayer
r"[a-z]{2,8}" # Account name
r"[A-Z][^0-9]+"
```

- Fits all: `.`

# Anchor

Tie the pattern to a specific position:

- Start/ end of lines.

```
r"^LOCUS.+" # LOCUS line from GenBank file
r"\s+$" # all trailing whitespace
r"^\d+␣\d+␣\d+$" # 3d coord
```

- Space between words.

```
r"\bwith\b" # "not with me", "Come with me!"
r"\bmit\B" # "mittendrin", nicht "vermitteln"
```

# Exercise – Sentence extraction

- Read the file */etc/services* lines by lines.
  - Extract all the lines which correspond to the protocol *TCP*.
  - Extract all the lines which describe a service (So, there is no line of commentary).
  - Extract all the lines which contain a four or five digits port number.
- How could we extract all the lines from *romeo.txt*, in which the words „club“ or „clubs“ appear but not „clubroom“?

# Patterns Capture

- Up to now: patterns occur in the text.

# Patterns Capture

- Up to now: patterns occur in the text.
- But: What was the hit ? $\rightarrow$ `findall` only half the truth.

# Patterns Capture

- Up to now: patterns occur in the text.
- But: What was the hit ? $\rightarrow$ `findall` only half the truth.
- Select region of interest:

```
r"^LOCUS\s+(\S+)"
r"^VERSION\s+(\S+)\.(\d+)\s+GI:(\d+)$"
```

# Patterns Capture

- Up to now: patterns occur in the text.
- But: What was the hit ? $\rightarrow$ `findall` only half the truth.
- Select region of interest:

```
r"^LOCUS\s+(\S+)"
r"^VERSION\s+(\S+)\.(\d+)\s+GI:(\d+)$"
```

- Hits stand at `match` in `match` objects.

```
> import re
> p = re.compile(r'a(b((c)d))')
> m = p.match('abcd')
> m.group()     #Whole match -> m.group(0)
'abcd'
> m.groups()    #Selected groups
('bcd', 'cd', 'c')
> m.group(2)
'cd'
```

# Patterns Capture

- Up to now: patterns occur in the text.
- But: What was the hit ? $\rightarrow$ `findall` only half the truth.
- Select region of interest:

```
r"^LOCUS\s+(\S+)"
r"^VERSION\s+(\S+)\.(\d+)\s+GI:(\d+)$"
```

- Hits stand at `match` in `match` objects.

```
> import re
> p = re.compile(r'a(b((c)d))')
> m = p.match('abcd')
> m.group()     #Whole match -> m.group(0)
'abcd'
> m.groups()    #Selected groups
('bcd', 'cd', 'c')
> m.group(2)
'cd'
```

- Use quantifiers correctly: `(\w)+` != `(\w+)`

# Exercise – Romeo, oh Romeo...

In *romeo.txt* we find the scene of the „ROMEO enters“

Extract the names of the people who took the scene in this way. Use an appropriate data type to save the people only once.

# Pattern Capturing

- The pattern must fit completely:

```
> p = re.compile(r'a(b((c)d))')
> m = p.match('abcd')
> type(m)
NoneType
```

- Differences between *grouping* and *capturing*:

```
r"\d+(-\d+)*"       # -12345
r"\d+(?:-\d+)*"     # 12345
```

# Exercise – Service list as a Service

Read */etc/services*. Place the informations about the coloquial services

For the line `ftp 21/tcp` the output looks like:

```
Der Dienst "ftp"verwendet TCP auf Port 21
```

Any additional information (Name/alias or comments) should be ignored.

# Greedy Matches

- What happens when a pattern is not unambiguous ?

```
> t = "aaaaaaaaaaa"
> p = re.compile(r"(a+)(a+)")
> m = p.match(t)
> m.group()      #???
> m.groups()     #???
```

# Greedy Matches

- What happens when a pattern is not unambiguous ?

```
> t = "aaaaaaaaaaa"
> p = re.compile(r"(a+)(a+)")
> m = p.match(t)
> m.group()      #???
> m.groups()     #???
```

- Try out:

```
r"(a+)(a*)"
r"(a*)(a+)"
r"(a*)(a*)"
r"(a?)(a*)"
r"(a{2,4})(a*)"
```

# Greedy Matches

- What happens when a pattern is not unambiguous ?

```
> t = "aaaaaaaaaaa"
> p = re.compile(r"(a+)(a+)")
> m = p.match(t)
> m.group()      #???
> m.groups()     #???
```

- Try out:

```
r"(a+)(a*)"
r"(a*)(a+)"
r"(a*)(a*)"
r"(a?)(a*)"
r"(a{2,4})(a*)"
```

- Set behind the first quantifiers: `+?  *?  ??  {2,4}?`

# Text decomposition

- Sequence components to combine a string:

```
> l = ['b', 1, 'ffeeg']
> "".join(map(str, l))
'b1ffeeg'
```

- Opposite function with RE: `split(string[, count=0])`
- Separation with pattern:

```
> story = "In␣a␣hole␣in␣the␣ground␣there␣lived␣a␣hobbit."
> p = re.compile(r"\s")
> p.split(story)
['In', 'a', 'hole', 'in', 'the', 'ground', 'there', 'lived',
   'a', 'hobbit.']
```

# Exercise – Separate the sentences!

Split the sentences.

„In a hole in the ground there lived a hobbit.“

With the following patterns. Which words are built there?

```
r"␣"
r""
r"\s*"
r"\b"
r"\B"
```

How big are the results?