Regular expressions

Martin Kroon

Hacking the Humanities
8 October 2018

What are they?

Technically:
A description of a *regular language*

i.e. a set of strings accepted by a *regular grammar*

What are they?

Technically:
A description of a *regular language*

i.e. a set of strings accepted by a *regular grammar*

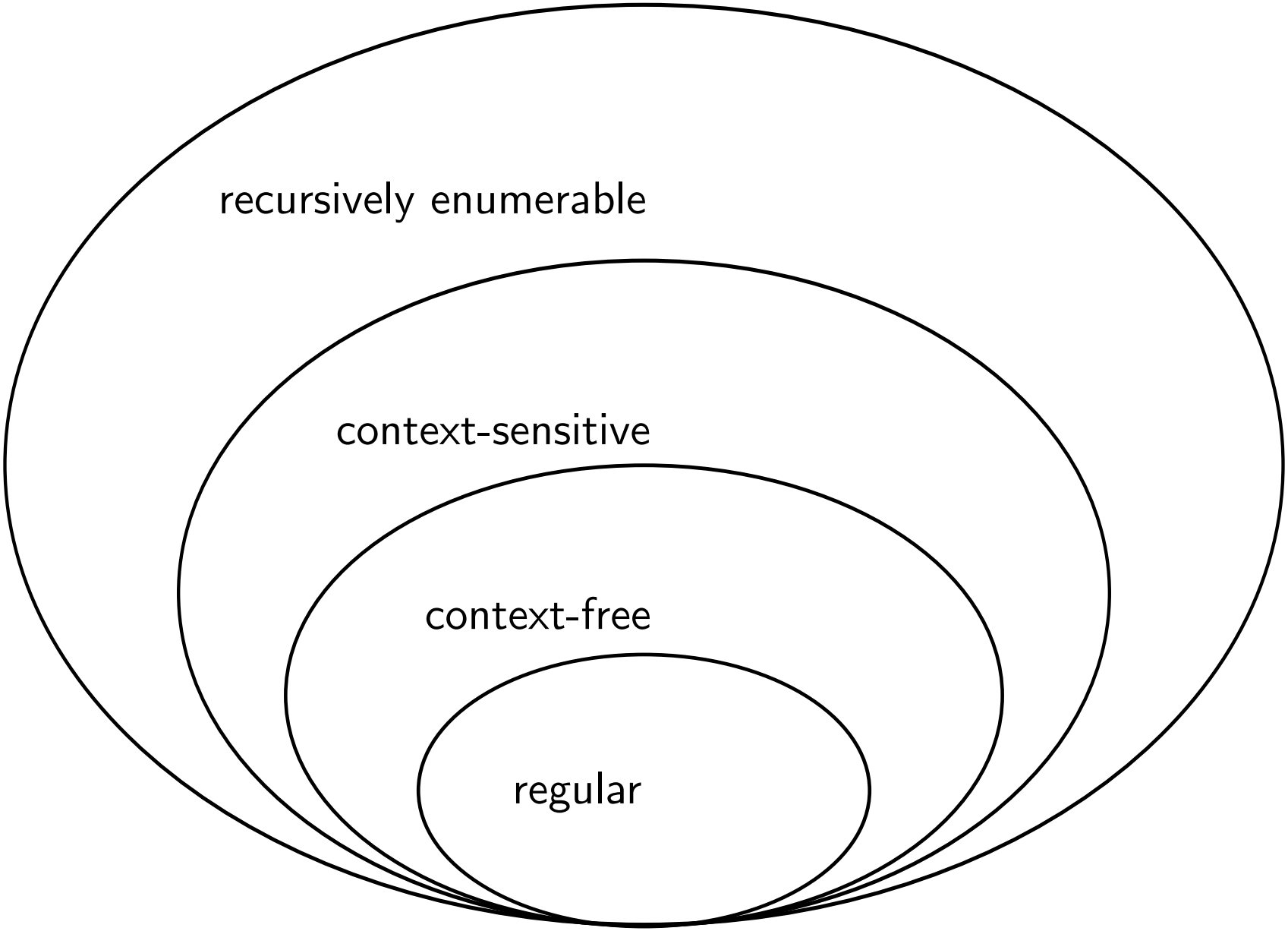
$$\left(\begin{array}{c} \text{or:} \\ \text{a pattern} \end{array} \right)$$

Formal languages

mathematical concept

a possibly infinite set of strings described by a set of rules: the grammar

Formal languages, cont'd



So?

What can we do with regexes?

used for search patterns and replacing things

matches all strings in the (possibly infinite) set of strings described by the regex

For example

British vs. American spelling:

- `sociali[sz]e` matches both *socialise* and *socialize*
- `honou?r` matches both *honor* and *honour*
- `thr(ough|u)` matches both *through* and *thru*

All (most) email addresses:

- `[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9.-]+`

They can be as complicated or as simple as you want!

First things first

```
import re
```

Matching

```
import re
myString = 'hello, how are you?'
result = re.match('hello', myString)
print(result)
```

returns:

```
<_sre.SRE_Match object; span=(0, 5), match='hello'>
```

Matching: note

`result = re.match('hello', myString)`
matches only at the beginning of the string!

`result = re.search('hello', myString)`
matches anywhere in the string but only returns the first hit

Groups and spans

```
print(result.group())
print(result.start())
print(result.end())
print(result.span())
```

returns:

```
'hello'
0
5
(0,5)
```

Notation: sets

`sociali[sz]e`

`[...]` means a set.
Matches any of the characters inside the set.

Notation: ranges

sets understand ranges:

`[a-z]` matches any lowercase letter
`[p-s]` matches p, q, r, and s
`[A-Z]` matches any uppercase letter
`[0-9]` matches any digit
`[A-Za-z0-9]` matches any letter (no diacritics!) or digit

Notation: set shorthands

some sets are predefined:

`\d` matches any digit (==`[0-9]`)
`\s` matches any whitespace character

- `\n` newline
- `\t` tab
- `\r` carriage return
- `\r\n` newline (Windows systems)

`\w` matches any alphanumeric character (does match diacritics! and underscores)

Notation: complementing sets

"anything but anything in the set"

`[^...]`
e.g. `[^sz]` matches anything as long as it isn't an s or a z

`\D` matches anything but a digit (`==[^0-9]`)
`\S` matches anything but a whitespace character
`\W` matches anything but an alphanumeric character

Anything

.

but not a `\n`!

anything anything: `(.|\n)`

Special characters

- \b matches a word boundary
- \B matches not a word boundary
- ^ matches the beginning of the string
- $ matches the end of the string

- * is the Kleene star: 0 or more times whatever comes before
  - a* matches an empty string (0 as), a, aa, aaa etc.
- ? matches 0 or 1 time whatever comes before
- + matches 1 or more times whatever comes before

Multiples

what if you don't want 0 or more, 1 or more or 0 or 1?

```
\d{3}
\d{3,}
\d{,3}
\d{1,3}
```

Greedy vs. non-greedy

`*`, `+` and `{,}` are greedy
they try to match as large a string as possible

adding a `?` after it, makes it non-greedy, matching the smallest possible

```
re.match('ae+?', 'aeeeeee')
```

Groups

you can group sequences

`(bla)+` matches `bla`, `blabla` etc.

Or

```
(cat|dog)
```

Recap

`[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9.-]+`

you can practice writing them on `regexone.com`

Findall, finditer

```
myString = "In 1954 I found 28 crumpets inside the
house at 11 County Road."
results = re.findall(r'\d', myString)
print(results)
results = re.finditer(r'\d', myString)
print(results)
```

returns

```
['1', '9', '5', '4', '2', '8', '1', '1']
<callable_iterator object at 0x008343D0>
```

Shortly on generators

```
results = re.finditer(r'\d', myString)
```

is a generator

- iterable that does not reserve memory but remembers where it was

access elements in it with:

```
for item in results:
   print(item)
```

you can make it into a list with `list(results)`

Groups

you can also use groups to return just parts of your match

```
someHtml = "<a href='www.google.com'>"
result = re.search(r"<a href='(.+)'>", someHtml)
print(result)
print(result.group(1))
```

returns

```
<_sre.SRE_Match object; span=(0, 25), match="<a
href='www.google.com'>">
'www.google.com'
```

Groups

you can also have multiple groups

```
myString = "It occured from pages 336-7."
result = re.search('(\d+)-(\d+)',myString)
print(result.group(1))
print(result.group(2))
```

returns

```
336
7
```

Sub

```
myString = "This has a      few    spaces."
myString = re.sub(r"\s+", " ", myString)
print(myString)
```

returns

```
'This has a few spaces.'
```

Sub with groups

```
myString = "123-456"
myString = re.sub(r"(\d+)-(\d+)", r"\2-\1", myString)
print(myString)
```

returns

```
'456-123'
```

Split

```
myString = "This has a      few   spaces."
myString = re.split(r"\W+", myString)
print(myString)
```

returns

```
['This', 'has', 'a', 'few', 'spaces', '']
```

Compile

if your regex is very complicated, you could compile it first to speed up your code (recommended in `for` loops)

```
myString = "m.s.kroon@hum.leidenuniv.nl"
pattern =
re.compile("[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9.-]+"
result = pattern.search(myString)
print(result)
```

Verbose

regexes can get very confusing and long
you can use the VERBOSE flag to make it clearer
ignores all whitespaces unless specifically written out, and allows for comments inside it

```
myString = "m.s.kroon@hum.leidenuniv.nl"
pattern = re.compile(
    r"""[a-zA-Z0-9_.+-]+  #the local-part
    @                     #the at-sign
    [a-zA-Z0-9-]+         #first part of the domain
    \.                    #dot
    [a-zA-Z0-9.-]+        #last part of the domain""",
    re.VERBOSE)
result = pattern.search(myString)
print(result)
```

Escaping characters

if you want to search for any character that means something in the syntax of a regex, you need to escape it with a backslash. For example:

- `.` matches any character, `\.` matches an actual period
- `[ab]` matches an *a* or a *b*, `\[ab\]` matches the string *[ab]*

other characters that need to be escaped are: `{ } ? * ^ $ + | ( )` and `\`

Raw strings

some characters or escaped characters actually mean something in Python syntax, such as \1
you can tell Python not to evaluate examples like these and have it read them as backslash-1 (which is necessary for `re` to understand the regex) by using a raw string. Some of the previous examples use them:

```
r"I, Python, will not evaluate this \1, but read it
as a \ and a 1 separately."
```

it is safest to always use raw strings, especially when your regex contains a backslash

Some puzzles

can you come up with a regex for:

- any number, either with or without fractional digits
- any phone number
- all words that start with *ge* and end in *en*
- any Roman numeral
- all occurrences of *the* but only if the next word or the word after that starts with an *s*