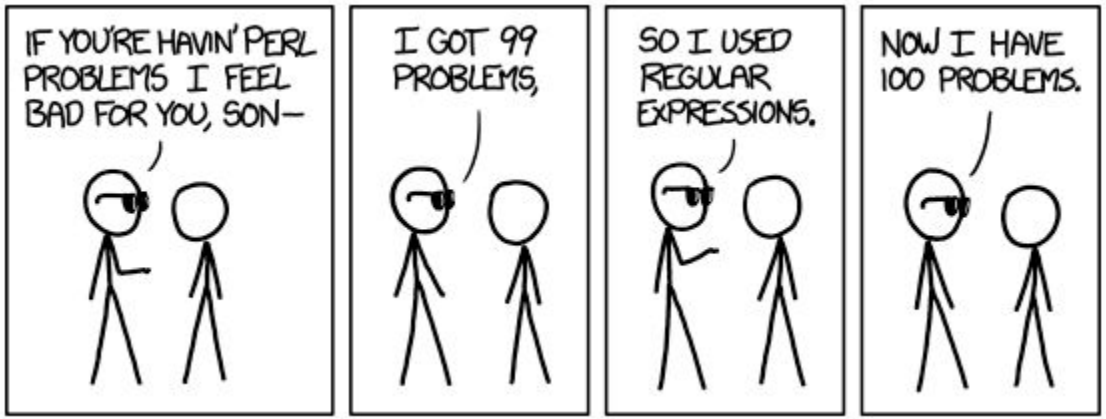
Regular Expressions



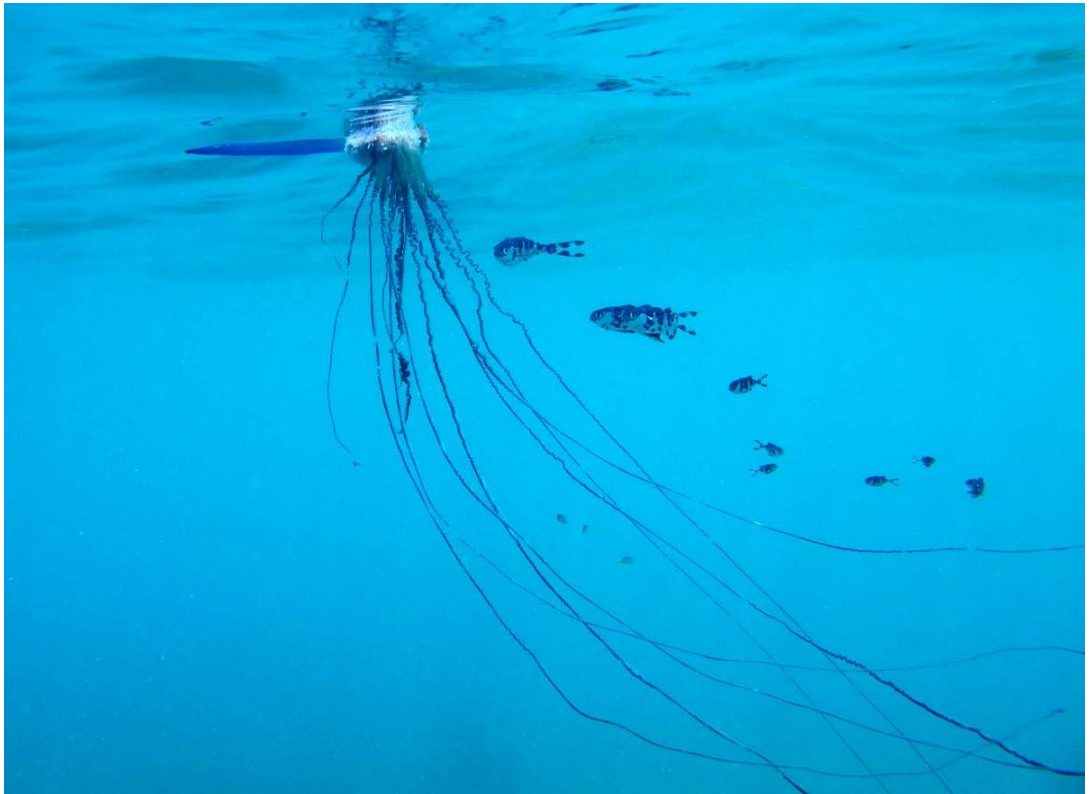
Alternative point of view

Regular Expressions

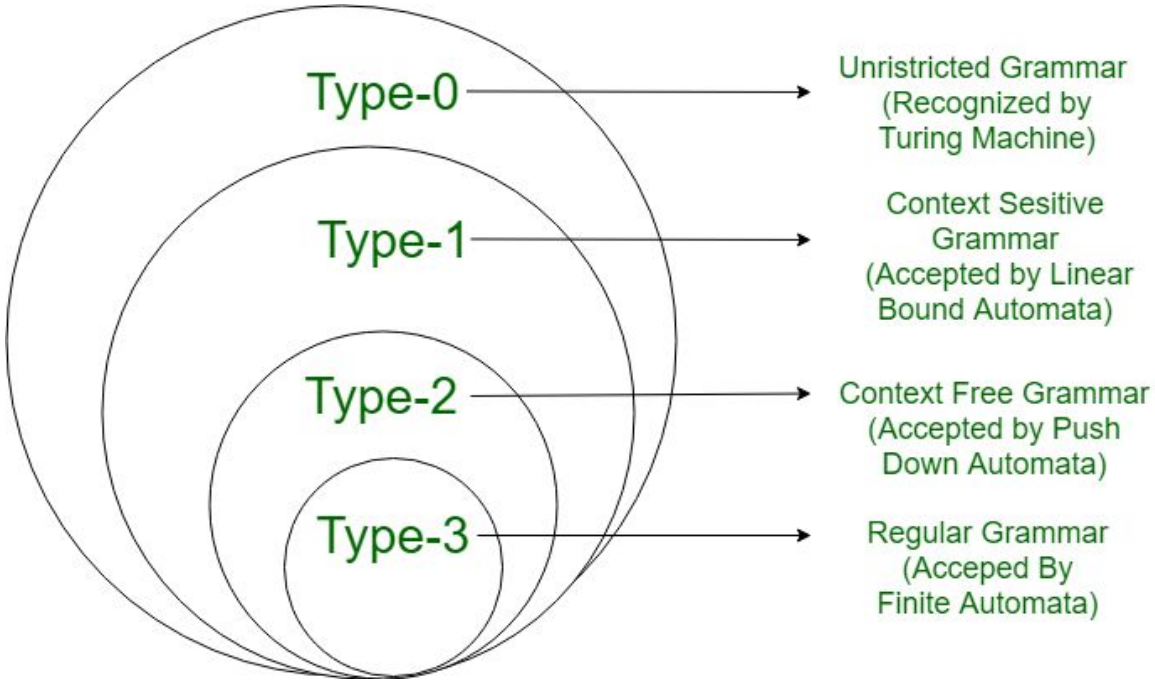
Regular expressions is a metalanguage which can be used to parse texts. It is linked to regular grammar - the simplest one from Chomsky hierarchy

1. Recursively enumerable - natural languages
2. Context-sensitive - xml, haskell
3. Context-free - most programming languages
4. Regular - serial patterns <- we are here

Some examples of regularity



Hierarchical grammar structure



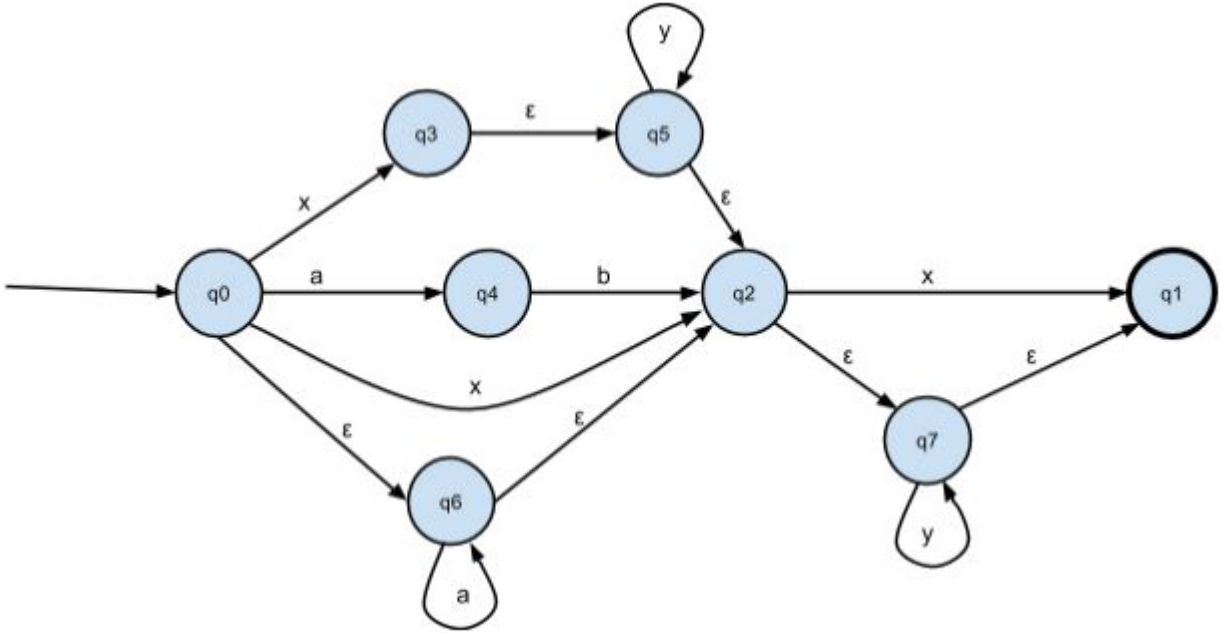
Don't use regular expressions for grammars harder than regular

4418

Locked. There are disputes about this answer's content being resolved at this time. It is not currently accepting new interactions.

You can't parse [X]HTML with regex. Because HTML can't be parsed by regex. Regex is not a tool that can be used to correctly parse HTML. As I have answered in HTML-and-regex questions here so many times before, the use of regex will not allow you to consume HTML. Regular expressions are a tool that is insufficiently sophisticated to understand the constructs employed by HTML. HTML is not a regular language and hence cannot be parsed by regular expressions. Regex queries are not equipped to break down HTML into its meaningful parts. so many times but it is not getting to transport a *programmer's consciousness i*nto a wor*l*d of ceaseless screaming, he comes~~, the pestilent sl~~ithy regex-infection will **devour your HT**ML parser, application and existence for all time like Visual Basic only worse *he comes he comes do not fi*ght h**e comes, hi**s unholy radiańcé destro͎ying all enl̈ightenment, HTML tags **leak̃ing ~~fr~~o̧m,yo̲ur eyes̸ ̸like liq**uid pain, the song of re̸gular expre~~ssion parsing~~ will ext*inguish the voices of mor***tal man from the sp***here I can see it can you see it it is beautiful t*he f `inal snuf` fing o*f the lie***s of Man ALL IS LOŞT ALL IS L**OST th*e pon̸y he comes* he c̶o̶m̶e̶s̶ ̶h̶e̶ ̶c̶o̶m̶e̶s̶ ***the ich****or permeates al*l MY FACE *MY FACE ᵒh god n***o NO NOO̼OO N**Θ stop *the an*gles `are` n**ot rè̑ał ZA̡LGΌ IS TO͠ NY THE PO̲NY HE COMES**

Finite automata



Re basics

Let's start with some denotations

- digits denotes digits
  - 1 will match 1
  - 23 will match 23
- letters denotes letters
  - a will match a
  - word will match word
- Space symbols match spacesymbols
  - \n, \t and so on

```
import re


# Define a pattern
pat = re.compile(r'wtf')
text = 'Wtf are you doing, wtf?!'


# Search text
ans = pat.search(text)
ans
<_sre.SRE_Match object; span=(19, 22), match='wtf'>
```

Match attributes

- `start()` - index of match start in string
- `end()` - index of match end in string
- `span()` - tuple with indices of match start and end in string
- `group()` - get match itself
- `re` - command for pattern creating
- `string` - string where we have searched

`match` searches pattern in string and returns first occurence

```
ans.start()
19

ans.end()
22

ans.span()
(19, 22)

ans.group()
'wtf'

ans.re
re.compile('wtf')
```

Non-single pattern occurrence in text

```
s = 'wtf re? wtf...'
ans = pat.search(s)
ans
<_sre.SRE_Match object; span=(0, 3), match='wtf'>
```

```
s = 'hmm, hm-hm-hm'
ans = pat.search(s)
print(ans)
None
```

Match

Checks whether the string starts with pattern

```
s = 'wtf, why are you still using this example?'
pat.match(s)
```

<_sre.SRE_Match object; span=(0, 3), match='wtf'>

```
s = 'Nonstarting wtf'
print(pat.match(s))
```

None

Findall

Finds all occurences and returns them as a list. Not very convenient in general

```
s = 'Multiple wtfs: wtf, wtf'
pat.findall(s)
['wtf', 'wtf', 'wtf']

s = ''
pat.findall(s)
[]
```

Finditer

Best method in my opinion. Returns all necessary information as an iterator

```
s = 'Multiple wtfs: wtf, wtf'
it = pat.finditer(s)

it = pat.finditer(s)
for i in it:
    print(i)
<_sre.SRE_Match object; span=(9, 12), match='wtf'>
<_sre.SRE_Match object; span=(15, 18), match='wtf'>
<_sre.SRE_Match object; span=(20, 23), match='wtf'>
```

re syntax

More about notation. Special characters

- `[]` - denotes group of characters
- `-` - used to define range of symbols inside `[]`
- `()` - used to separate groups of characters

```
pat = re.compile(r'[a-z]')
pat.findall('wtf is not wtf')
['w', 't', 'f', 'i', 's', 'n', 'o', 't', 'w', 't',
'f']


s = 'cat under the table, cat in the table, cat on
 the table'
pat = re.compile('cat [io]n the table')

pat.findall(s)
['cat in the table', 'cat on the table']
```

```
s = 'H: 1, Z: 4'
pat = re.compile('([a-zA-Z]): ([0-9])')

pat.findall(s)
[('H', '1'), ('Z', '4')]
```

For each match we now have a tuple, where each group is a separate element

```
for m in pat.finditer(s):
    print(m)
<_sre.SRE_Match object; span=(0, 4), match='H: 1'>
<_sre.SRE_Match object; span=(6, 10), match='Z: 4'>
```

```
for m in pat.finditer(s):
    print('available groups:', m.groups())
    print('whole match:', m.group(0))
    print('1st group:', m.group(1))
    print('2nd group:', m.group(2))

available groups: ('H', '1')
whole match: H: 1
1st group: H
2nd group: 1
available groups: ('Z', '4')
whole match: Z: 4
1st group: Z
2nd group: 4
```

Group attributes

So we have additional parameters

- `groups()` - tuple with all groups from pattern
- `group(n=0)` - select group from match
  - n = 0 - return whole match
  - n = 1, 2… - return content of corresponding group

More syntax

- `\d` - match all digits, equivalent to the class [0-9]
- `\D` - match everything except digits
- `\w` - match all alphanumeric characters (digits + letters), equivalent to the class `[a-zA-Z0-9_]`
- `\W` - match everything except alphanumeric characters
- `\s` - all space characters, equivalent to the class `[\t\n\r\f\v ]`
- `\S` - match everything except space characters
- `\b` - match the edge of the word
- `\B` - match not the edge of the word

```
s = '+7-921-742-33-67'
pat = re.compile(r'\d\d\d')
pat.findall(s)
['921', '742']

s = 'Pytho, Python, Pythono, Pythonum'
pat = re.compile(r'Python')
pat.findall(s)
['Python', 'Python', 'Python']


pat = re.compile(r'Python\b')
pat.findall(s)
['Python']
```

```
s = 'Pytho, Python, Pythono, Pythonum'
pat = re.compile(r'Python\B')
pat.findall(s)
['Python', 'Python']


s = 'My   \nmultispace \t\rline'
pat = re.compile(r'\S')
pat.findall(s)
['M', 'y', 'm', 'u', 'l', 't', 'i', 's', 'p', 'a',
 'c', 'e', 'l', 'i', 'n', 'e']
```

Quantificators

Used for specifying number of occurences

- `*` - Kleene star - match any number of previous element, including 0
- `+` - Kleene plus - match 1 and greater number of previous element
- `?` - match 0 or 1 time
- `{min,max}` - match `min` minimum number of previous element up to `max` number; `min`=0, `max`=infinite by default, they can be skipped

```
s = 'ab, abc, abccc, abcccc'
pat_any = re.compile(r'abc*')
pat_gr = re.compile(r'abc+')
pat_3 = re.compile(r'abc{3}')
pat_2h = re.compile(r'abc{2,}')

pat_any.findall(s)
['ab', 'abc', 'abccc', 'abccc']

pat_gr.findall(s)
['abc', 'abccc', 'abccc']

pat_3.findall(s)
['abccc', 'abccc']
pat_2h.findall(s)
['abccc', 'abcccc']
```

```
s = 'look, cook, lok, cok and 2 guns'
pat = re.compile(r'\woo?\w')
pat.findall(s)
['look', 'cook', 'lok', 'cok']

s = 'thorium - 2, uranium - 3, iridium - 7'
pat = re.compile(r'\w+')
pat.findall(s)
['thorium', '2', 'uranium', '3', 'iridium', '7']
```

```
s = 'H: 1, He: 4'
pat = re.compile(r'(\w{1,2}): (\d+)')
pat.findall(s)
[('H', '1'), ('He', '4')]


pat = re.compile(r'(\w+): (\d+)')
pat.findall(s)
[('H', '1'), ('He', '4')]
```

Alternative re invocation

Notation which we have used is similar to the other one

```
pat = re.compile(pattern)
pat.command(string)

re.command(pattern, string)
```

But the latter case less effective. Now this difference is neglegible if you have a moderate number of different patterns in a program

Non-greedy search

By default each quantifier tries to digest as much characters as possible.
Addition of `?` after it makes them non-greedy

```
re.findall(r'a+', 'aaaaaaaaaaa')
['aaaaaaaaaaa']

re.findall(r'a+?', 'aaaaaaaaaaa')
['a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a']
```

Important note

Such behaviour can slow down search greatly. Some patterns have exponential time complexity, so be careful and test them on small datasets. If search takes too long - probably your pattern is not optimal

Other stuff

- `.` - wildcard - match any symbol except newline
- `\` - escape character for special characters
- `^` - match start of line
- `$` - match end of line
- `[^]` - match everything except elements inside the set
- `|` - match element before pipe or after it

```
# Parse my phone
s = '+7-921-742-33-67'



# Parse site name
s =
'https://docs.python.org/3/library/re.html#match-objects'

# Parse date
s = '30/12/1994'
```

```
pat = re.compile(r'\+(\d)-(\d{3})-(\d{3})-(\d{2})-(\d{2})')
pat.findall(s)
[('7', '921', '742', '33', '67')]
```

```
pat = re.compile(r'https?:\/\/([^\/]+)')
pat.findall(s)
['docs.python.org']
```

```
pat = re.compile(r'\d{1,2}/\d{1,2}/\d{2,4}')
pat.findall(s)
['30/12/19']
```

Flags

Parameter to modify re behaviour

```
re.compile(pattern, flags)
```

`flags` accept number denoting flag, it is easier to provide them from the re module. Flags

- `re.I` - ignorecase, pattern will act without considering case
- `re.A` - ascii, make special characters match only ascii symbols
- `re.M` - multiline, pattern will act on each string independently
- `re.S` - dotall, . will match everythong including newline
- `re.X` - verbose, make it possible to split pattern across several lines and write comments

Named groups

It is possible to assign names for groups and it is good

```
(?P<elem>pattern)other_pattern
```

- `?P` - denotes named group
- `<name>` - name of your group
- `pattern, other_patter` - some re pattern

```
s = 'H: 1, He: 4'
pat = re.compile(r'''(?P<elem>\w+)  # get element name
                     :[ ]           # skip separator
                     (?P<mass>\d+)  # get element mass
                   ''', re.VERBOSE)

for m in pat.finditer(s):
    print(m.group('mass'), m.group('elem'))
1 H
4 He
```

Groups referring

Furthermore we can refer to previous groups

`(?P<name>pattern)some_pattern(?P=name)other_pattern`

- `?P` - denotes named group
- `<name>` - name of your group
- `=name` - reference to the match of `name` group
- `pattern, some_pattern, other_patter` - re patterns

Unnamed groups referring

Same can be done with ordinary groups. Instead of `?P=name` just write `(\index)` where `index` is an index of group to which you make a reference

```
s = '07/11/12 was in 2012 03/12.15'
pat =
re.compile(r'(\d\d)(?P<sep>[-/.])(\d\d)(?P=sep)(\d\d)')
pat.findall(s)
[('07', '/', '11', '12')]


pat = re.compile(r'(\d\d)([-/.])(\d\d)(\2)(\d\d)')
pat.findall(s)
[('07', '/', '11', '/', '12')]
```