Regular Expressions in Python Tutorial

Michelle Fullwood

PyLadies Meetup Aug 2013

Motivating regular expressions

Scenario: You're evaluating Acme Company's products for a hospital. You're given a text file containing purchase records from Acme. Looking through the text file, however, you see that purchases from other companies are included as well, with no mention of which ones come from which!

```
PCOD    QTY    DEPT      COST
A169    100    Micro     0.58
PDA1    1      Xray      600.00
X280    5      ER        199.99
...
```

Luckily, you know that Acme's product codes consist of one uppercase letter followed by three digits.

Motivating regular expressions

So you get to work with a script:

```
import string
for line in open('purchaserecords.txt','r'):
    if line[0] in string.uppercase and \
       line[1].isdigit() and \
       line[2].isdigit() and \
       line[3].isdigit():
        print line
    else:
        continue
```

It's a bit clunky, but it works.

Motivating regular expressions

The next step in your evaluation is to collate comments emailed to you by hospital staff. These were free text, so you need to extract the product codes from within them to know which evaluation refers to which product.

```
'...The X701 vacuum cleaner really sucked!...'
'...The gloves(P180) felt sticky...'
```

You might be able to think of ways to program this, but really...

It's time to bust out regular expressions.

Motivating regular expressions

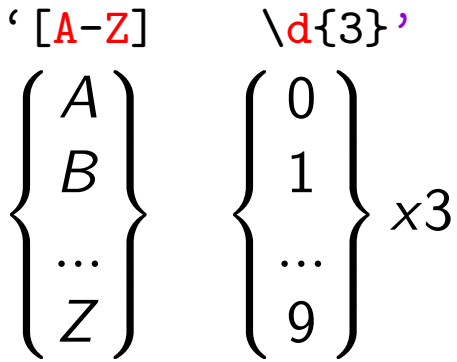
What we want is a way to simply search for "one uppercase letter followed by three digits". We can do this using (1) a regular expression and (2) the `search` function provided by Python's `re` module:

```
import re

re.search(r'[A-Z]\d{3}', mystring)
```

Just what are regular expressions, anyway?

Regular expressions are strings that describe other sets of strings.



What we need to know to write regular expressions:

- How to define sets of characters
  - Metacharacters `\w`, `\s`, `\d`
  - Character sets `[A-Z]`, `[AGCT]`, `[^AGCT]`
- How to define how many times to repeat them
  - A specific number of times `{3,5}`, `?`
  - An unlimited number of times `*`,`+`

Plan for today

We'll learn:

- The pattern language for regular expressions
- The Python `re` functions that allow us to work with regexes

Playing along

How to practise the code as we go along:

- Install `git` (if you attended last month's meeting, you already have it!)
- From the command line:

  ```
  git clone https://github.com/michelleful/RegexTutorial.git
  ```

- Run the exercises from the command line interface:
  - `cd exercises`
  - Open up `ex1.py` etc as we move along
  - Edit and run `python ex1.py` from the CLI
- OR if you have iPython:
  - Run `ipython notebook` from the command line
  - A browser window will automatically open
  - Select the only notebook, follow the instructions therein
- OR if you don't have `git`, play with the regexes online
  - Enter regexes and strings into `http://www.pythonregex.com/`

Aside: why r"?

All the regular expressions you'll see today will be within `r''`. What's the deal with the `r`?

`r` stands for raw strings. It means that backslashes are interpreted as backslashes, as opposed to regular strings, where they're interpreted as special characters.

You could define regexes using regular Python strings, but then you'd need to escape every backslash.

```
r'\d3' == '\\d3'
```

And if you're searching for backslashes, that would mean needing to write: `'\\\\'`, as opposed to just `r'\\'`.

Metacharacters

Metacharacters are pre-defined sets of characters.

- . matches ANY character except the newline character \n*
- \d matches digits 0 through 9
- \w matches alphanumeric characters and underscore _
- \s matches whitespace characters
  - Spaces
  - Tabs \t
  - Newlines \n\r

→ Exercise 1

Defining sets of characters

- List characters individually
  - `[AGCT]` matches one character A, G, C or T.
  - `[\s\d]` matches one whitespace character or digit
- Define a range of characters
  - `[A-T]` matches one character between A and T.
  - `[1-7]` matches one digit between 1 and 7.
  - Ranges as defined by ASCII or Unicode tables
  - You can combine ranges: `[a-cA-C]`

→ Exercise 2

Defining where a regular expression applies

We can say a regex has to be at the start or the end of the string, or at word boundaries, with more special characters.

- `^` – beginning of line
- `$` – end of line
- `\b` – word boundary

Examples: → Exercise 3

- `^Hallo$`
- `\bHallo\b`

Escaping characters

→ Exercise 4

Why didn't the regex(es) work? (Discuss.)

- When using characters that also have a special meaning, we have to escape them with a backslash
- `\^ \$ \. \\`
- Exception: within character sets [], metacharacters have their regular meaning, except backslashes.*

Defining complements of a set

Sometimes it's easier to define a set of characters as "everything other than X".

- `\S` – all non-whitespace characters
- `\W` – all non-alphanumeric characters (also excludes underscore)
- `\D` – all non-numeric characters
- `[^A-D]` – all characters other than A, B, C, D

→ Exercise 5

Repeating things

So far, whenever we've wanted to match, say, three digits, we've just been writing `\d\d\d`. This could easily get old if we want to match thirty or a thousand digits. It gets impossible when we want to match any arbitrary number of digits. Fortunately, regular expressions let us express this very succinctly.

Repeating things

- `*` means 0 or more times
- `+` means 1 or more times
- `?` means 0 or 1 times
- `{n}` means n times exactly
- `{m,}` means m or more times
- `{m,n}` means m–n times
- Example: `.*` matches anything
- Example: `[a-c]{3}` matches 'abc' etc

→ Exercise 6

Specifying alternatives

Sometimes you want to say match this OR that. You can do that with the | operator.

- `Alex|Bill|Conrad` matches any of these three names

Sometimes there can be confusion about what | refers to. In such cases, put brackets around the alternatives.

- `Jim and (Alex|Bill|Conrad)` matches 'Jim and Alex', 'Jim and Bill', etc

→ Exercise 7

Taking a step back

We've gone through a lot of the syntax for regular expressions.

1. Any questions?
2. Any challenging patterns you'd like to try as a group to express as a regex?

Python functions for dealing with regular expressions

- `re.find(regex, string)` – returns a match group if the string starts with the regex pattern, `None` otherwise.
- `re.search(regex, string)` – returns a match group if the string contains the regex pattern, `None` otherwise.

→ Exercise 8

Doing stuff with regular expressions

So far we've just been checking for matches with regexes. But we can do more!

We can:

- Capture the matches and do stuff with them
- Replace the match with something else
- Split a string whenever you match the regex (not covered today)

Capturing matches

We capture matches by putting brackets around the part we want to extract. Then access the matched bits via the match object returned by `re.find()` and `re.search()`

Example: We want to match strings that look like a first name followed by a last name, and extract them separately.

```
import re

name_regex = r'\b([A-Z][a-z]+)\b \b([A-Z][a-z]+)\b'
m = re.search(name_regex, '"Oliver Twist" was written
    by Charles Dickens')
if m:
    print m.groups()

# returns ('Oliver', 'Twist')
```

→ Exercise 9

Capturing multiple matches

Notice on the previous slide that we only captured the first instance of each name. In order to capture multiple matches, use `re.findall()`.

```
import re

name_regex = r'\b([A-Z][a-z]+)\b \b([A-Z][a-z]+)\b'
print re.findall(name_regex, '"Oliver Twist" was
    written by Charles Dickens')

# returns [('Oliver','Twist'), ('Charles','Dickens')]
```

→ Exercise 10

Nesting capturing parentheses

You can actually nest capturing parentheses. For instance, if we put another set of parentheses around the example in the preceding slide, we get this:

```
import re

name_regex = r'\b(([A-Z][a-z]+)\b (\b[A-Z][a-z]+))\b'
print re.findall(name_regex,
    '"Oliver Twist" was written by Charles Dickens')

# returns [('Oliver Twist', 'Oliver', 'Twist'),
#          ('Charles Dickens', 'Charles', 'Dickens')]
```

Non-greedy matching

By default, operators like * and + are "greedy". They match as much as they can. In order to make it match in a "non-greedy" way, add a ? after the operator.

```
import re

# Default greedy matching
m = re.match(r'(ab+)', 'abbbbbbbb')
print m.groups() # ('abbbbbbbb',)

# Non-greedy matching with ?
m = re.match(r'(ab+?)', 'abbbbbbbb')
print m.groups() # ('ab',)
```

Non-capturing parentheses

→ Exercise 11

Earlier, we used parentheses `()` to disambiguate what the `|` operator referred to. But now we're getting unwanted captures.

The solution here is to use non-capturing parentheses. Replace the regular parentheses `(...)` with `(?P:...)` whenever you need to use parentheses but don't want to capture the result.

Replacing matches

Here's another thing you can do with regexes – replace some matched segment with another. For example, you're writing a novel with the protagonist Edward, who has various nicknames. Halfway through you decide to rename him just Paul.

```
import re

# Syntax: re.sub(regex, replacement_text,
    string_to_replace_text_in)
print re.sub(r'\bEdward|Ned|Eddie|Ed\b', r'Paul',
    'Edward left. Ned came back in. Eddie asked, "Am I
    schizophrenic?" Ed said no.')

# 'Paul left. Paul came back in. Paul asked, "Am I
    schizophrenic?" Paul said no.'
```

(This is a bit of a stupid example because you could just use string replacement, but it's a first step.)

→ Exercise 12

Back references

Suppose you want to protect all the email addresses on your site from being slurped up by a robot. You want to replace '@' with ' (at) ', but only in valid email addresses and not, say, Twitter handles.

We want to reuse our reliable ol' regex:

```
r'w+@w+\.(?:com|org|edu|net)'
```

Problem: how do we replace the @ sign while leaving the stuff on either side intact? We can do this by capturing what's to the left and right using parentheses, then reproducing them in our replace string with **back references**.

Back references

Backreferences are referred to with `\n`. `\1` refers to the first match, `\2` to the second match, etc.

```
import re

email_at = r'(\w+)@(\w+\.(?:com|net|edu|gov))'
print re.sub(email_at, r'\1 (at) \2', 'amy@gmail.com')
# 'amy (at) gmail.com'
print re.sub(email_at, r' (at) ', '@amy')
# '@amy'
```

→ Exercise 13

Back references

You can also use back references within the regex you use to match to find multiple copies of the same string.

```
import re

print re.sub(r'(\b\S+\b) \1', r'\1',
    'This is is a string with doubled doubled words')
# 'This is a string with doubled words'
```

Advanced topics

- Compiling regular expressions
- Flags
- Named matches

Compiling regular expressions

Sometimes you might see code like this:

```
import re

product_code_regex = re.compile(r'[A-Z]\d{3}')
print type(product_code_regex)
# <type '_sre.SRE_Pattern'>

print product_code_regex.findall('A103 B296 Z999')
# ['A103', 'B296', 'Z999']

# This is equivalent to:
print re.findall(r'[A-Z]\d{3}', 'A103 B296 Z999')
# ['A103', 'B296', 'Z999']
```

Once you have compiled your regex, you can call all the `re` functions directly as methods of the compiled regex.

Compiling your regex is a good idea but not necessary for small programs because Python will compile and cache your last few regexes anyway.

Flags

Flags change the way metacharacters, character sets and spaces are interpreted.

Some of the flags:

- `re.IGNORECASE` or `re.I` – case-insensitive matching
- `re.MULTILINE` or `re.M` – makes `^` and `$` match beginning and end of each newline
- `re.DOTALL` or `re.S` – makes `.` match any character *including* `\n`
- `re.UNICODE` or `re.U` – `\w`,`\b` reinterpreted according to Unicode definition
- `re.VERBOSE` or `re.V` – ignores whitespace and lets you use comments within the regex

Flags

How to set flags:

On any `re` function, add the flags as the last parameter:

- `re.search(myregex, mystring, re.U)`
- `re.compile(myregex, re.MULTILINE | re.DOTALL)`

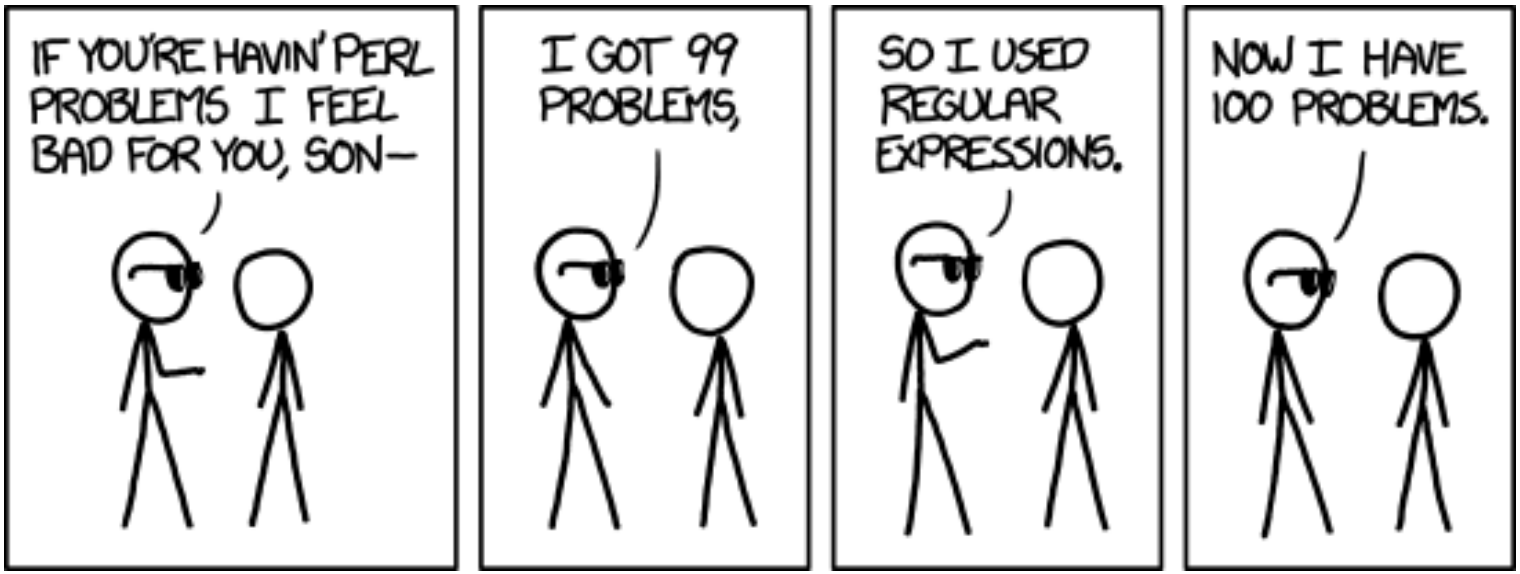
Named matches

It's possible to identify groups by names:

```
# code from the Python docs
m = re.match(r"(?P<first_name>\w+)
                (?P<last_name>\w+)",
                "Malcolm Reynolds")
m.group('first_name') # 'Malcolm'
m.group('last_name')  # 'Reynolds'
```

This can be helpful for keeping track of multiple matches.

A word of caution on your newfound power...



http://xkcd.com/1171/

When not to use regular expressions

Regular expressions are powerful, but you shouldn't reach for them whenever you need some string matching.

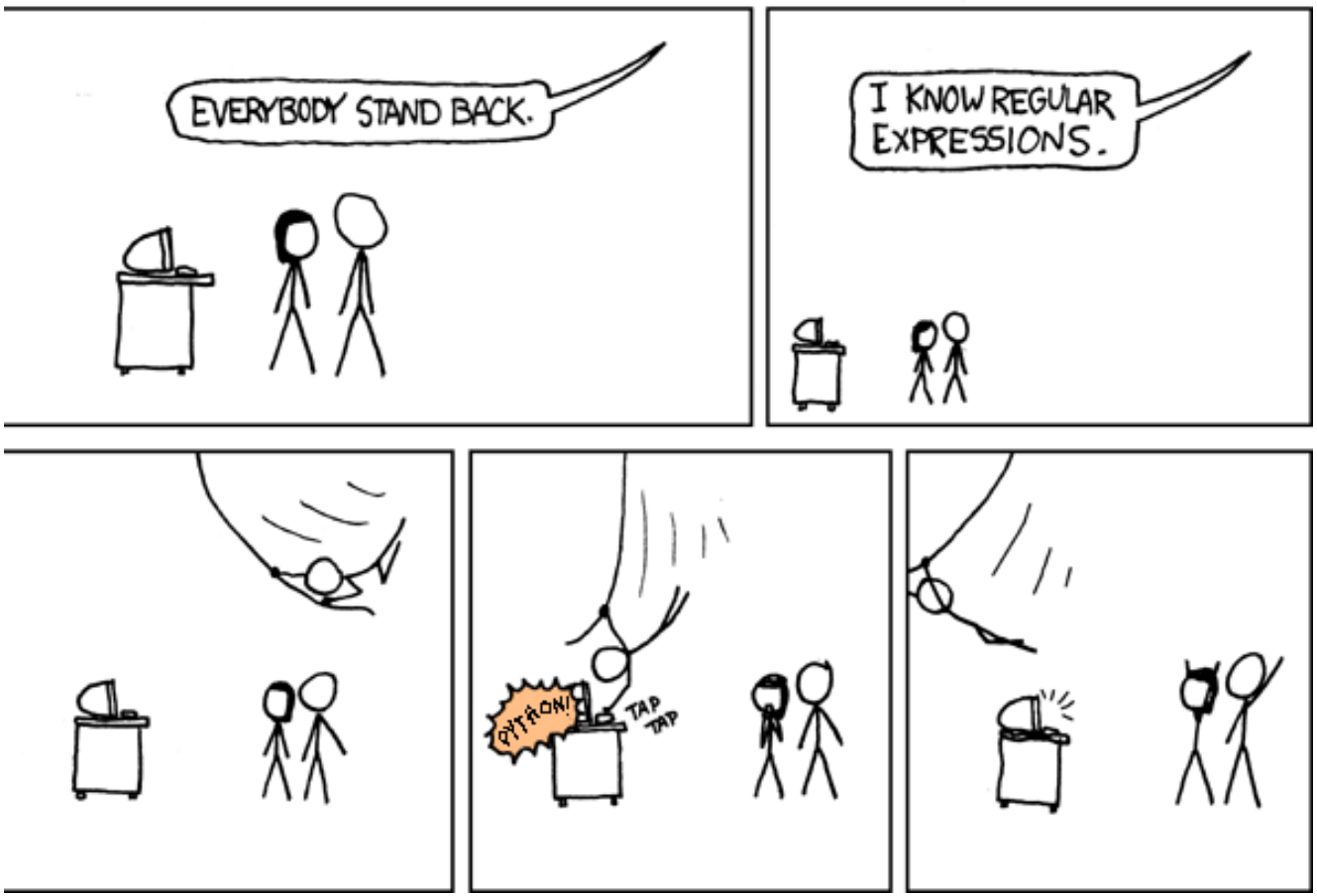
In particular, don't use regular expressions when:

- A simple string operation will do
  - `'abc' in mystring`
  - `mystring.replace('x','y')`
- You're parsing something that uses matching brackets, e.g. HTML and XML.
  - Use the `BeautifulSoup` or `lxml` libraries instead!

The end

That's all I have to teach you about regular expressions.

Now get out there and be a hero.



Slightly mangled https://xkcd.com/208/

Further resources

- Ask me a question now!

- The Python docs are great.
  http://docs.python.org/2/library/re.html
- Online tool for testing regexes: http://www.pythonregex.com/
- *Mastering Regular Expressions* by Jeffrey Friedel
- Learn Regex the Hard Way
  http://regex.learncodethehardway.org/book/