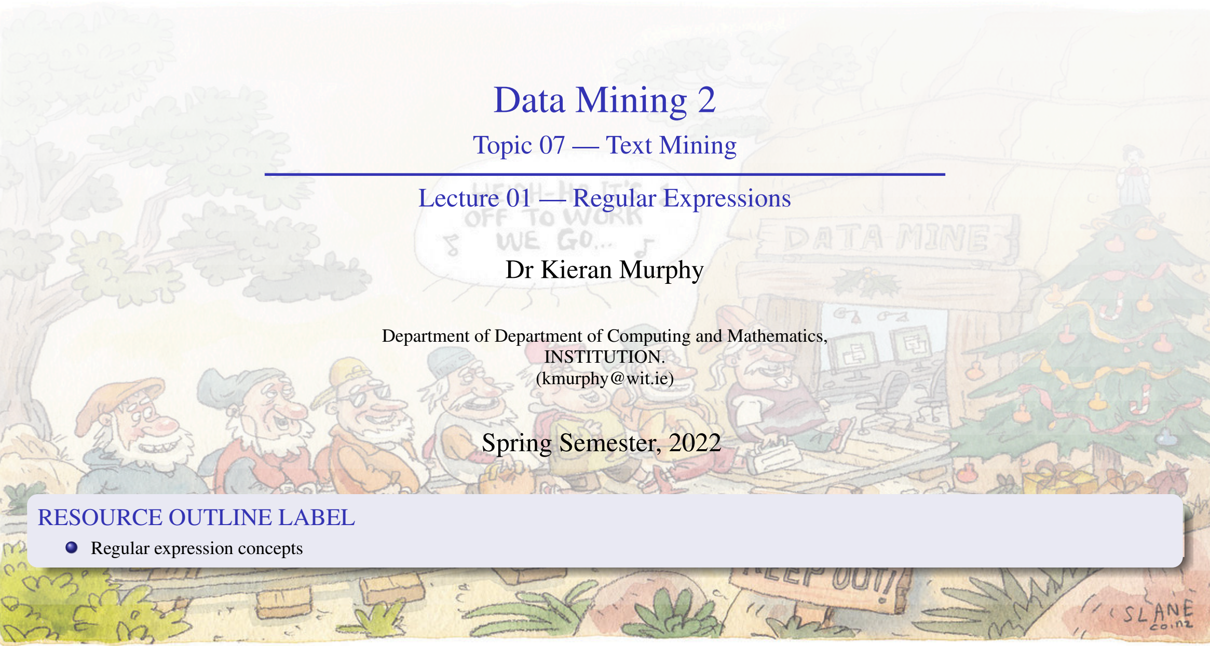


Data Mining 2

Topic 07 — Text Mining

Lecture 01 — Regular Expressions

Dr Kieran Murphy

Department of Department of Computing and Mathematics,
INSTITUTION.
(kmurphy@wit.ie)

Spring Semester, 2022

RESOURCE OUTLINE LABEL

- Regular expression concepts

Outline

Example 1 — DNA, Repressed Binding Sites

- DNA can be represented as a loooooong string sequence consisting only of characters "A", "C", "G", and "T".
- Want to find particular sub-sequences, but these sub-sequences can have multiple variations.
- A **repressed binding site** has the following sub-sequence variations*

⇒ Some possible† sub-sequences denoting a **repressed binding site** are

"AGGCATGTCCTAACATGCCT" "AGGCATGTTTAACATGCCT"
"GGACATGTCCTAACATGCCC" "GGACATGTCCTAACTTGTGC"

---

*I'm treating the patterns  and  as identical, and both mean that either "A" or "G" can appear with equal probability, etc. (reality is more complicated, but if Trump can ignore it , then so can I)

†Not to bring back horrible memories of *Discrete Mathematics*, but how many variations are possible?

Example 1 — Naive (=non-RE) Implementation

- Naive-naive python implementation — using only syntax common in other languages.

example_binding_sites.py

```
    for i in range(0,len(dna)-19):

        if ((dna[i] == "A" or dna[i] == "G") and
            (dna[i+1] == "A" or dna[i+1] == "G") and
            (dna[i+2] == "A" or dna[i+2] == "G") and
            (dna[i+3] == "C") and
            (dna[i+4] == "A") and
```

... and skip a few lines ...

```
            (dna[i+17] == "C" or dna[i+17] == "T") and
            (dna[i+18] == "C" or dna[i+18] == "G") and
            (dna[i+19] == "C" or dna[i+19] == "T")):
            print ("Sample %d (%s): match found at pos %s" % (k, dna, i))
            break
    else:
        print("Sample %d (%s): no match " % (k,dna))
```

Example 1 — Regular Expression Implementation

- Similar regular expression implementations in c/c++, java, javascript, haskell, perl . . . .

First, load regular expression module and create regular expression pattern to compare against . . .

example_binding_sites.py

```
33 import re
34 r = re.compile(r"[AG]{3}CATG[TC]{4}[AG]{2}C[AT]TG[CT][CG][TC]")
```

Then, search for matching sub-sequences . . .

example_binding_sites.py

```
39     m = r.search(dna)
40     if m != None:
41         print ("Sample %d (%s): match found at pos %s" % (k, dna, m.start()))
42     else:
43         print("Sample %d (%s): no match " % (k,dna))
```

Implementation needed two lines, line 34 to define the regular expression, and line 39 to test for a match.

Example 2— Date/time Identification/Parsing

Consider the problem of standardising date and time information from unstructured data source (i.e., free form text input).
For example, in medical notes — each one-line entry contains a (partial) date but the formatting was not consistent, and includes variation such as:

- month/day/year (with(out) leading zero, two or four digit year),

  e.g. "04/20/2009". "04/20/09"; "4/3/09"
- Mixture of delimiter symbols ("/", "\" or "-")
- Month (partially) written out (and order changing)

  "Mar-20-2009"; "Mar 20, 2009"; "20 March, 2009"; etc
- Use of ordinal numbers

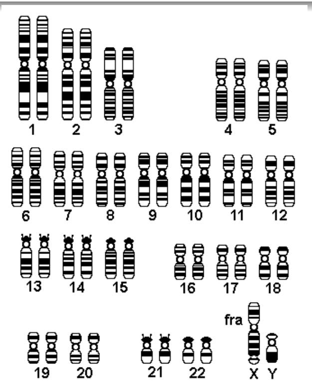
  "Mar 20th, 2009"; "Mar 21st, 2009"; etc
- Partial dates

  "Feb 2009"; "6/2008"; "2010"

This can be accomplished without regular expressions but is a pain.

Example 3 — DNA, Fragile X Syndrome

- Fragile X syndrome is a genetic condition that causes a range of developmental problems including learning disabilities and cognitive impairment. Usually, affecting males more severely than females.
- Within the FMR1 gene is a sub-string containing triplet repeats of "CGG" or "AGG", bracketed by "GCG" at the beginning and "CTG" at the end.
- Number of repeats is variable and is correlated to syndrome.



Regular Expression

expression GCG(CGG|AGG)$*$CTG

string . . . GCGGCGTGTGTGCGAGAGAGTGGGTTTAAAGCTGGCGC GGAG-GCGGCTGGCGCGGAGGCTG . . .

Summary (Before we start!)

> *"Some people, when confronted with a problem, think 'I know I'll use regular expressions.' Now they have two problems."*
>
> — *Jamie Zawinski (flame war on alt.religion.emacs)*

- ✔ Regular expressions can match arbitrary complex sub-strings.
- ✔ Simpler, shorter and less error-prone than using standard control statements.
- ✘ Can get very complicated and difficult to debug.

> Regular expressions is vital skill for any programmer, sys admin, data analyst, etc..
> Just use regular expressions up to the level of complexity that you feel comfortable with.

- Regular expressions are greedy (match as many characters as possible) and eager (stops as soon as a match is found).

How bad can it get?

The first half of the Perl RE for valid RFC822 emails is

```
(?:(?:\r\n)?[ \t])*(?:(?:(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)
?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))
|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[\t]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|
(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])
+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*|(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()
<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)*\<(?:(?:\r\n)?[ \t])*(?:@(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:
(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[\t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+
(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*(?:,@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\]
\000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@
,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*)*:(?:(?:\r\n)?[ \t])*)?
(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)
?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|
(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>
@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=
[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*\>(?:(?:\r\n)?[ \t])*)|(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])
+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)*:(?:(?:\r\n)?[ \t])*(?:(?:(?:[^()<>@,;:\\".\[\] \000-\0
+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@
".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[
*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[
(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*|(?:[^()<>@,;:\\"
\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)*\<(?:(?:\r\n)?[ \t
(?:@(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n
?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*(?:,@(?:
(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)
(?:\.(?:(?:\r\n)?[ \t/])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n
)?[ \t])*))*)*:(?:(?:\r\n)?[ \t])*)?(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[
]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(
?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r
\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:
(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\
```

www.ex-parrot.com/~pdw/Mail-RFC822-Address.html

Regular Expressions as a Classification Problem

When building regular expression you need to be mindful of two antagonistic aims:

- make expression permissive enough to match desired patterns
- make expression restrictive enough to exclude undesired patterns

---

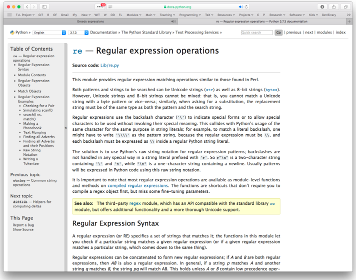
Or if you think in terms of errors — we have two kinds of errors

- Matching strings that we should not have matched
  False positives (Type I)
- Not matching things that we should have matched
  False negatives (Type II)

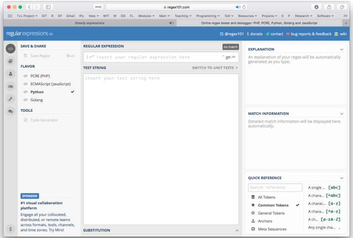
Hence can think of constructing a regular expression as a classification task — and reducing the error rate for an regular expression involves two efforts:

- Increasing precision, (minimising false positives)
- Increasing coverage, or recall, (minimising false negatives)

Resources



https://regex101.com



Outline

Literal Characters

- Letters and digits match themselves
- Normally case sensitive
- Watch out for punctuation characters — most of them have special meanings!

Example

Given text

“A␣Jack␣and␣Phil␣went␣up␣a␣hill. ”

then regular expressions

- `/hil/` matches

  A␣Jack␣and␣Phil␣went␣up␣a␣hill.

- `/a/` matches

  A␣Jack␣and␣Phil␣went␣up␣a␣hill.

- `/il./` matches

  A␣Jack␣and␣Phil␣went␣up␣a␣hill.

Literal Characters

- Letters and digits match themselves
- Normally case sensitive
- Watch out for punctuation characters — most of them have special meanings!

Example

Given text

"A␣Jack␣and␣Phil␣went␣up␣a␣hill. "

then regular expressions

- `/hil/` matches

A␣Jack␣and␣Phil␣went␣up␣a␣hill.

- `/a/` matches

A␣Jack␣and␣Phil␣went␣up␣a␣hill.

- `/il./` matches

A␣Jack␣and␣Phil␣went␣up␣a␣hill.

Literal Characters

- Letters and digits match themselves
- Normally case sensitive
- Watch out for punctuation characters — most of them have special meanings!

Example

Given text

“A␣Jack␣and␣Phil␣went␣up␣a␣hill. ”

then regular expressions

- `/hil/` matches

  A␣Jack␣and␣Phil␣went␣up␣a␣hill.

- `/a/` matches

  A␣Jack␣and␣Phil␣went␣up␣a␣hill.

- `/il./` matches

  A␣Jack␣and␣Phil␣went␣up␣a␣hill.

Literal Characters

- Letters and digits match themselves
- Normally case sensitive
- Watch out for punctuation characters — most of them have special meanings!

Example

Given text

“A␣Jack␣and␣Phil␣went␣up␣a␣hill. ”

then regular expressions

- `/hil/` matches

  A␣Jack␣and␣Phil␣went␣up␣a␣hill.

- `/a/` matches

  A␣Jack␣and␣Phil␣went␣up␣a␣hill.

- `/il./` matches

  A␣Jack␣and␣Phil␣went␣up␣a␣hill.

Metacharacters

`\ . * + - {} [] ^ $ | ? () : ! =`

- Characters with special meaning
  - escape — transform literal character to/from metacharacter

    `\`
  - wildcard operator (represents any character)

    `.`
  - set of characters

    `[]`
  - ranges of characters

    `-`
  - repeats — zero or more/one or more/range

    `* + {}`
  - start/end of string

    `^ $`
- Can have more than one meaning — depends on context.

Wildcard Metacharacter

.

- The wildcard represents any single character except the newline.
  - Original unix regex engines were line based tools.
- `/h.t/` matches

  The␣hot␣MAGA␣hat␣sat␣on␣the␣heated␣hob.
- Widest match character.
- Common mistake when matching numbers `/9.00/` matches

  9.00␣vs␣9500␣vs␣9:00

Escape Metacharacter

\

- Allows use of metacharacters as literal characters
- Match as period with `/\./`,for example `/9\.00/` matches

  9.00␣vs␣9500␣vs␣9:00
- Match backslash using `/\\/`
- Convert literal characters to metacharacter (if defined).
- Note quotation characters are not metacharacters.

Characters Set

`[]`

- The string of characters inside the braces specifies a disjunction (ORing) of characters to match.
- Order of characters does not matter — is a set

Examples

- `/[aeiou]/` matches any one vowel, i.e.,

  My␣queue␣is␣not␣a␣stack

- `/gr[ae]y/` matches

  Is␣the␣colour␣gray␣or␣grey?

- `/[01234567890]/` matches any one digit, i.e.,

  8␣out␣of␣10␣cats␣does␣countdown

Warning

In first and third example we only did repeated matching of SINGLE characters.

For example, we did not match “8” and “10”, but matched “8”, “1”, and “0”.

Characters Range

–

- Range metacharacter
  - Represent all characters between two characters (inclusive).
  - Only a metacharacter inside a character set, a literal dash otherwise.

Examples

- `/[0-9]/`matches any integer.
- `/[A-Za-z]/`matches any letter.
- `/[a-ek-ou-y]/`matches any letter from 'a' to 'e', from 'k' to 'o', and from 'u' to 'y' inclusive.
- Warning: `/[50-99]/`does not match all numbers from 50 to 99. It is the same as `/[0-9]/`or `/[0-99999]/`or ...

Negative Characters Sets

^

- Negative metacharacter.
  - Indicates not any one of serval characters.
  - Must be first character inside a character set — otherwise is treated as a literal character.
  - Resulting character set still represents a single character.

Examples

- `/[^aeiou]/` matches any non-vowel, i.e.,

  My␣queue␣is␣not␣a␣stack

  (*Note: I'm also matching the spaces.*)

- `/[ ^aeiou]/` matches space, any vowel, or the character "^", i.e.,

  My␣queue␣is␣not␣a␣stack

- `/[Ss]ee[^mn]/` matches

  The␣Seeker␣does␣see␣but␣does␣not␣seem␣to␣have␣seen.

  (*Note: Matched "see␣" because of the trailing spacex.*)

Metacharacters inside Character Sets

- Metacharacters inside character sets are already escaped.
  - Do not need to escape them again
  - `/h[abc.xyz]t/` matches

My␣hat␣is␣not␣hot␣but␣is␣h.t

- Exceptions

`]   -   ^   \`

- `/[[\]]/`matches

[]␣brackets␣rule!␣Down␣with␣parentheses!

However, to avoid a future warning (in python) you should also escape the opening square bracket also, .i.e., `/[\[\]]/`

Shorthand Characters Sets

Shorthand	Meaning	Equivalent
`\d`	Digit	`[0-9]`
`\w`	Word character	`[A-Za-z0-9_]`
`\s`	Whitespace	`[ \t\r\n]`
`\D`	Not a digit	`[^0-9]`
`\W`	Not a word character	`[^A-Za-z0-9_]`
`\S`	Not a whitespace	`[^ \t\r\n]`

- Underscore is a word character, but hyphen is not.
- `/\s\d\d\d\d\s/` matches
  Reading␣1984␣in␣1984␣was␣cool␣only␣in␣1984.
- `/\s\w\w\w\w\s/` matches
  Reading␣1984␣in␣1984␣was␣cool␣only␣in␣1984.
- Note: While `/[^\d]/` is same as `/[\D]/`, and `/[^\s]/` is same as `/[\S]/`, however `/[^\d\s]` is not the same as `/[\D\S]/`

Repetition Metacharacters

Metacharacter	Meaning
*	Preceding item (character or expression) zero or more times
+	Preceding item one or more times
?	Preceding item zero or one time

- `/apples*/` matches

  apple,␣apples,␣applesss,␣applesss5s

  (Note the greedy matching)

- `/apples+/` matches

  apple,␣apples,␣applesss,␣applesss5s

  (Again greedy matching)

- `/apples?/` matches

  apple,␣apples,␣applesss,␣applesss5s

- `/\d\d\d\d*/` matches

  1,␣12,␣123,␣1234,␣12345,␣123456

Quantified Repetition

Metacharacter	Meaning
`{n}`	*n* occurrences of previous item
`{m,n}`	From *m* to *n* occurrences of previous item
`{m,}`	At least *m* occurrences of previous item
`{,n}`	At most *n* occurrences of previous item

- `/\d{0,}/`is same as `/\d*/`
- `/\d{1,}/`is same as `/\d+/`
- `/A{1,2}/` matches one or two “A” so what happened here?

A␣bonds,␣AA␣bonds,␣AAA␣bonds

(In the python code that I used to generate examples I used finditer *which performs multiple matches and so the three “A” are matched by two “A” and one “A”. Remember regrex is greedy and eager)*

Greedy Expressions

- Standard repetition qualifiers greedy — tries to match the longest possible string.
- However, it priorities eager over greedy — so will be less greedy in order to make a successful match.
- Consider applying the regex `/.+\.jpg/` to

  filename.jpg

  - The regex `/.+/` matches

    filename.jpg and filename.png

  - The `/+/` is greedy but rewinds or backtracks so that ".jpg" is matched by rest of the expression.
- The regex gives back as little as possible to make match.
  For example, `/.*[0-9]+/` matches

  Page␣666

  where `/.*/` matches "Page␣66" and `/[0-9]+/` matches "6".

Lazy Expressions

?

- Lazy Expression Metacharacter
  - Switches the previous repetition operator from greedy to lazy strategy.

  Greedy — match as much as possible before giving control to next expression part.
  Lazy — match as little as possible before giving control to next expression part.

  - Both defer to overall match (i.e. backtrack/roll forward)
  - neither strategy is always faster/more efficient.
- Note `?` is also a repetition metacharacter (zero or one time). How do we know which role it is playing? — context.

Examples

- `/\w*?\d{3}/` matches

  AA12␣AA123␣BBB12D0000

- `/.{4,8}?\-.{4,8}/` matches

  AAAAAA-AAAA␣BBBB-BBBBBB

- `/.{4,8}\-.{4,8}?/` matches

  AAAAAA-AAAA␣BBBB-BBBBBB

Strategies for Efficient Regex Repetition

General principle

Efficient matching ⇒ less backtracking ⇒ faster results.

Stratagies

- Specify the quantity of repeated expressions — the more restrictive the better:
  - `/+/`is faster then `/*/`.
  - `/.{4}/`and
  - `/.{2.7}/`are even faster.
- Narrow scope of ranges — the narrower the better.
  - Replace `/.+/`with `/[A-Za-z]+/`
- Provide clearer starting and ending points
  - Replace `/<.+>/`with `/<[^>]+>/`

Grouping Metacharacter

`( )`

- Grouping metacharacters
  - Apply repetition operation to a group
  - Make expressions more readable (for humans)
  - Captures a group for use in matching and replacing
  - Cannot be used inside a character set.

Examples

- `/C(GT)+A/` matches

  CA␣CGTA␣CGTA␣CGTGTGTA␣CGTGTGA

- `/(in)?dependent/` matches

  independent␣or␣dependent

Alternation Metacharacter

|

- Alternation metacharacter — OR operator
  - Either match expression on the left or match expression on the right
  - Ordered, left most expression gets precedence
  - Multiple options can be daisy-chained
  - Can group (using parenthesis) alternation expressions to improve readability
  - Alternation expressions can be nested

Examples

- `/apples|oranges|grapes/` matches
  I␣like␣apples␣and␣oranges␣but␣not␣grapes
- `/(AA|BB|(CC|\d)){2,4}/` matches
  DAA3CCBBEECBB3AACCBB

  (First matched alternation does not affect the next matches.)

Start and End Anchors

Metacharacter	Meaning
`^`	Start of a string/line
`$`	End of a string/line
`\b`	Word boundary

- Anchors reference to a position not a character ⇒ have zero width
- Word boundary conditions
  - Before the first word character in the string
  - After the last word character in the string
  - Between a word character and a non-word character
  - Recall: word characters are `/[A-Za-z0-9_]/`
- `/\b\w+\b/` matches

Sticks␣and␣stones␣can␣break␣my␣bones

(Note spaces are not matched)

Backreferences

`\1 \2 ... \9`

- Group expressions are captured
- Store the matched text portion that corresponds to the group expression.
  - For example. `/(W\d{8})/` matches

    My␣id␣is␣W66600666

    and stores "W66600666" in `\1`.
  - Stored the matched text, not the expression.
- Can be used in same expression as the group
- Can be accessed after the match is complete (see regex object in python)
- Cannot be used inside character classes.
- Groups are captured automatically, but if you don't want[‡] to capture a group then start group with `?:` For example, replace `/(\w+)/`with `/(?:\w+)/`

[‡]Why? It is faster and you can only capture 9 (or 99 on some systems) groups.

Outline

Example 4

Problem 4

Find all instances of the word "the" in the text.

"The␣thesis␣title␣is␣'The␣tithe␣of␣the␣Theron.' "

- `/the/`

  The␣thesis␣title␣is␣'The␣tithe␣of␣the␣Theron.'

  Fails to match "The" and incorrectly matches parts of words.

- `/[Tt]he/`

  The␣thesis␣title␣is␣'The␣tithe␣of␣the␣Theron.'

  … getting there, but we are still matching parts of words …

- `/\b[Tt]he\b/`

  The␣thesis␣title␣is␣'The␣tithe␣of␣the␣Theron.'

  Done

Example 4

Problem 4

Find all instances of the word "the" in the text.

"The␣thesis␣title␣is␣'The␣tithe␣of␣the␣Theron.' "

- `/the/`

  The␣thesis␣title␣is␣'The␣tithe␣of␣the␣Theron.'

  Fails to match "The" and incorrectly matches parts of words.

- `/[Tt]he/`

  The␣thesis␣title␣is␣'The␣tithe␣of␣the␣Theron.'

  … getting there, but we are still matching parts of words …

- `/\b[Tt]he\b/`

  The␣thesis␣title␣is␣'The␣tithe␣of␣the␣Theron.'

  Done

Example 4

Problem 4

Find all instances of the word "the" in the text.

"The␣thesis␣title␣is␣'The␣tithe␣of␣the␣Theron.' "

- `/the/`

  The␣thesis␣title␣is␣'The␣tithe␣of␣the␣Theron.'

  Fails to match "The" and incorrectly matches parts of words.

- `/[Tt]he/`

  The␣thesis␣title␣is␣'The␣tithe␣of␣the␣Theron.'

  . . . getting there, but we are still matching parts of words . . .

- `/\b[Tt]he\b/`

  The␣thesis␣title␣is␣'The␣tithe␣of␣the␣Theron.'

  Done

Example 4

Problem 4

Find all instances of the word "the" in the text.

"The␣thesis␣title␣is␣'The␣tithe␣of␣the␣Theron.' "

- `/the/`

  The␣thesis␣title␣is␣'The␣tithe␣of␣the␣Theron.'

  Fails to match "The" and incorrectly matches parts of words.

- `/[Tt]he/`

  The␣thesis␣title␣is␣'The␣tithe␣of␣the␣Theron.'

  . . . getting there, but we are still matching parts of words . . .

- `/\b[Tt]he\b/`

  The␣thesis␣title␣is␣'The␣tithe␣of␣the␣Theron.'

  Done

Example 5 — Matching IP (IPV4) addresses

- `/\b\d+\.\d+\.\d+\.\d+\b/`

  192.168.1.1␣64.16.83.133␣0.0.0.0␣0..00␣999.0.0.0␣2545.0.0.0

  Need to limit number of digits
- `/\b\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}\b/`

  192.168.1.1␣64.16.83.133␣0.0.0.0␣0..00␣999.0.0.0␣2545.0.0.0

  What to do about numbers in range 256-299?
  - Match numbers in range 250–255 using `/25[0-5]/`
  - Match numbers in range 200–249 using `/2[0-4][0-9]/`
  - Match numbers in range 100–199 using `/1[0-9][0-9]/`
  - Match numbers in range 10–99 using `/[1-9][0-9]/`
  - Match numbers in range 0-9 using `/[0-9]/`
- If we just match a single number (to save space) we then have regex

  `/\b(25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9][0-9]|[0-9])\b/`

  192␣1␣24␣255␣256␣999␣2545

Exercise: shorten this expression!

Example 5 — Matching IP (IPV4) addresses

- `/\b\d+\.\d+\.\d+\.\d+\b/`

  192.168.1.1␣64.16.83.133␣0.0.0.0␣0..00␣999.0.0.0␣2545.0.0.0

  Need to limit number of digits

- `/\b\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}\b/`

  192.168.1.1␣64.16.83.133␣0.0.0.0␣0..00␣999.0.0.0␣2545.0.0.0

  What to do about numbers in range 256-299?

  - Match numbers in range 250–255 using `/25[0-5]/`
  - Match numbers in range 200–249 using `/2[0-4][0-9]/`
  - Match numbers in range 100–199 using `/1[0-9][0-9]/`
  - Match numbers in range 10–99 using `/[1-9][0-9]/`
  - Match numbers in range 0-9 using `/[0-9]/`

- If we just match a single number (to save space) we then have regex

  `/\b(25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9][0-9]|[0-9])\b/`

  192␣1␣24␣255␣256␣999␣2545

Exercise: shorten this expression!

Example 5 — Matching IP (IPV4) addresses

- `/\b\d+\.\d+\.\d+\.\d+\b/`

  192.168.1.1␣64.16.83.133␣0.0.0.0␣0..00␣999.0.0.0␣2545.0.0.0

  Need to limit number of digits
- `/\b\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}\b/`

  192.168.1.1␣64.16.83.133␣0.0.0.0␣0..00␣999.0.0.0␣2545.0.0.0

  What to do about numbers in range 256-299?
  - Match numbers in range 250–255 using `/25[0-5]/`
  - Match numbers in range 200–249 using `/2[0-4][0-9]/`
  - Match numbers in range 100–199 using `/1[0-9][0-9]/`
  - Match numbers in range 10–99 using `/[1-9][0-9]/`
  - Match numbers in range 0-9 using `/[0-9]/`
- If we just match a single number (to save space) we then have regex

  `/\b(25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9][0-9]|[0-9])\b/`

  192␣1␣24␣255␣256␣999␣2545

Exercise: shorten this expression!

Example 6 — Matching Binding Sites

Recall the DNA binding pattern:

A suitable regular expression is

```
/[AG]{3}CATG[TC]{4}[AG]{2}C[AT]TG[CT][CG][TC]/
```

Examples

- ...AACTAGGCATGTCCTAACATGCCTAACT...
- ...AACTGGACATGTCCTAACATGCCCAACT...
- ...AACTGGACATGTCCTAACTTGTGCAACT...
- ...AACTAGGCATGTCCTAACATGCCAACT...
- ...AACTCGGCATGTCCTAACATGCCTAACT...

Outline

The Python re Module

The python module, re, contains functions for manipulating regular expressions, and performing match and/or substitution. Main concepts:

- *Compiling Regular Expressions*
  A regular expression needs to be compiled into a more efficient representation before use. You can explicitly compile using the function re.compile, python keeps a cache of previously compiled expressions.
- *Regular expression object*
  Some re methods return an object on successful matching that not only contains the characters matched but also the start and end index, span, etc.
- *Storing regex as strings*
  Python does not use forward-slash character to delimit regular expressions. Instead just treats them as ordinary strings and uses single and double quotes. To avoid metacharacters being interpreted by python use prefix r as in `regex = r"[AG]3CATG[TC]4[AG]2C[AT]TG[CT][CG][TC]"`

Match vs Substitution

Matching

- match(regex, string)
  Tries to match a regular expression at the beginning of the string. If successful, return a regular-expression object. Otherwise, returns None.
- search(regex, string)
  Like match but not restricted to start of string.
- findall (regex, string)
  Find all substrings of the string that match the regular expression. The function returns a list of all matching substrings.
- finditer (regex, string)
  Find all substrings of the string that match the regular expression. The function returns a list of regular-expression objects.[§]

Substitution

- re.sub(regex, replacement, string)
  Replace parts of a string that match a regular expression.

[§] See script markup_re_examples.py.

Outline

Regular Expression Golf

— http://xkcd.com/1313

Example

Match elected US presidents but not opponents (unless they later won).

Match	obama, bush, clinton, regan, . . . washington.
Don't match:	romney, mccain, gore, ...

Solution

A solution — works up to but not including Trump (I'm still living in denial).

bu|[rn]t|[coy]e|[mtg]a|j|iso|n[hl]|[ae]d|lev|sh|[lnd]i|[po]o|ls

Illegally Screening a Job Candidate

```
" [First name]! and pre/2 [last name] w/7
   bush or gore or republican! or democrat! or charg!
or accus! or criticiz! or blam! or defend!
or iran contra or clinton or  spotted owl
or florida recount or sex! or controvers! or fraud!
or investigat! or bankrupt! or layoff! or downsiz!
or PNTR or NAFTA or outsourc! or indict! or enron
or kerry or iraq or wmd! or arrest! or intox! or fired
or racis! or intox! or slur! or controvers! or abortion!
or gay! or homosexual! or gun! or firearm! "
```

— LexisNexis search string used by Monica Goodling
to illegally screen candidates for DOJ positions

www.justice.gov/oig/special/s0807/final.pdf

(Not all Republican misdeeds were under Trump!)


WHENEVER I LEARN A NEW SKILL I CONCOCT ELABORATE FANTASY SCENARIOS WHERE IT LETS ME SAVE THE DAY.
OH NO! THE KILLER MUST HAVE FOLLOWED HER ON VACATION!
BUT TO FIND THEM WE'D HAVE TO SEARCH THROUGH 200 MB OF EMAILS LOOKING FOR SOMETHING FORMATTED LIKE AN ADDRESS!
IT'S HOPELESS!
EVERYBODY STAND BACK.
I KNOW REGULAR EXPRESSIONS.
PERL!
TAP TAP