Scraping unstructured formats

Before we begin

Introduction

So far, we have seen

- How to replicate a crawler and get the content of an HTTP request / of a webpage
- How to parse a mark-up language to extract content
- How to automate the collection

Introduction

Still...

\t\t\n Born on March, 26th 1924 \t\t\n
or
Site inspir� de perdu.com, pr�sent� � des fins p�dagogiques.

And this is not very easy to deal with...

Introduction

These problems emerge

- when scraping
- but also when dealing with classic data sets (encoding, format, etc)
- And you could work on raw text for many reasons (OCR,
- searching, etc)

Introduction

What we want to be able to do

“\t\t\n Marie was Born on April, 15th 1992 in Paris \t\t\n”
“\t\t\n John was born on March, 12th (?) 1991 in N.Y.\t\t”

Who	Month	Day	Year	Place
Marie	April	15	1992	Paris
John	March	12	1991	N.Y.

Introduction

The classic response to this is “Regular Expressions”
also known as “rational expressions”, or “RegEx”

```
/h[a4@](([c<]((k)|(\|<)))|((k)|(\|<))|(x))\s+\
((d)|([t\+]h))[3ea4@]\s+p[l1][a4@]n[3e][t\+]/i
```

Introduction

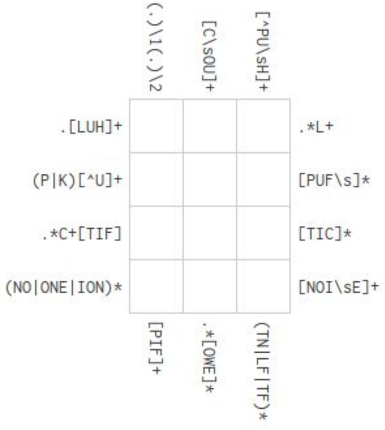
Regex: What are They?

- A powerful “Search and Replace” tool
  > Identify a pattern in the text, and do something to it

- They are central to working with unstructured data
  > Another tool for selection, on raw text this time

- A common tool among programmers

- An object of reverence and fascination



www.regexcrossword.com

Introduction

Regex: What are They?

And in Practice

- YAPL!
  > Yet Another Programming Language

- But for our purpose, you will only need a few basic commands

- And there is plenty of help online.

Introduction

Outline

Regex: Basic Syntax

- Basic query
- Multiple choices
- Jokers
- Context
- Quantifiers
- Special characters

And in Practice

- In R: Find
- In R: Replace
- Resources

Regex: Basic Syntax

Regex: Basic Syntax
Regex serve to do basic searches
The search pattern is *exact*

Regex	Will match	Will not match
“a”	“Laura” “Alexia”	“Roberto” “Alexi”
“dataf”	“dataframe”	“data”
“19”	“1930”	“139”
“c_1”	“abc_1997”	“abc-1997”

Regex: Basic Syntax

Regex: Basic Syntax

Multiple choice

There are a few operators that will help you

- [az8] = a or z or 8
- [a-u] = All letters from a to u
- a|b = a or b

Regex	Yes	No
“[Pp]aul”	“Pauline” “epaule”	“Pau”
“P[a-z4]ul”	“Paula”, “Pbula” “P4ula”, “Poula”	“PAula”
“P[a-zA-Z0-9]ul”	“Pbul”, “P0ul”	“P*ul”
“Dupont|Dupond”	“Dupont”, “DuponDupond”	“Tintin”

Regex: Basic Syntax

Regex: Basic Syntax

Jokers

They help you out when you are unsure.

- \d = Any digit
- \w = Any word, or digit, or _
- .   = Anything you want
- [^a1*] = Anything, except a, 1, or *

Regex: Basic Syntax

Regex: Basic Syntax

Context

They position your search pattern in the string

- ^ = indicates the beginning of a string
- $ = indicates the end of a string
- \b = indicates the boundary of a word (no sign after)

Regex	Yes	No
“^Paula”	“Paula likes John”	“John is liked by Paula”
“Paula$”	“John is liked by Paula”	“Paula likes John”
“Paul\b”	“Paul”	“Paula”

Regex: Basic Syntax

Regex: Basic Syntax

Quantifiers

How many of these patterns do you want?

- a{18} = a, 18 times in a row
- a{7,} = a, 7 or more times in a row
- a+ = a, once or more
- a* = a, zero times or more
- a? = 0 or 1 a (“is there an a?”)

Regex	Yes	No
“Pa+ul”	“Paul”, “Paaaul”	“Pul”
“Pa*ul”	“Pul”, “Paaaul”	“Pa#ul”
“\d{4}”	“1976”, “12345”	“123”
“a\d{3,}a”	“a123a”, “a1234a”	“a12a”

Regex: Basic Syntax

Regex: Basic Syntax
Special characters
Sometimes, two languages mix in unpredictable ways
. + * \ [] () ?

All of these are part of our common language (your string of character) and the regex language.

==> To capture them, you need to escape them (add \)

In R, escaping requires not one but two \\

Regex: Basic Syntax

Regex: Basic Syntax
Questions?

Regex: Basic Syntax

Regex: Basic Syntax
Question

What will this pattern capture?

```
.*
```

And what about this one?

```
\b[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,4}\b
```

In Practice

Search in R

grep(REGEX, VECTOR)

text ← c(“alice”, “Carole”, “Bob”)

grep(“a”, text) will return 1 – 2, ie. the index (the position) of the matching elements.

> Note this this is an “index vector”. In can then be used to select and subset.

grep(a, text, value = T) will return “alice” “Carole”
> This is useful to see that your regex works

In Practice

Replace in R

gsub(REGEX, REPLACEMENT, VECTOR)

text ← c(“ Alice ”, “ Carole ”, “B ob ”)

gsub(“ ”, “”, text) will return the first names, without space
gsub(“\\s”, “”, text) will also delete all spaces

gsub(“^\\s|\\s$”, “”, text) → What will this do?

In Practice

Replace in R

gsub(REGEX, REPLACEMENT, VECTOR)

text ← c(“on the year 1515”, “Party like its 1999”)

gsub(“\\d”, “”, text) will return only not-numbers

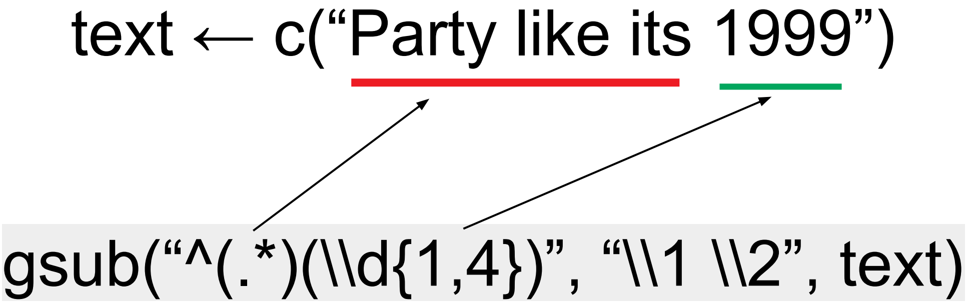
gsub(“\\D”, “”, text) will return only numbers

In Practice

Replace in R

```
gsub(REGEX, REPLACEMENT, VECTOR)
```

But this is a bad way to do extraction, and we can be more precise



```
text ← c("Party like its 1999")

gsub("^(.*)(\\d{1,4})", "\\1 \\2", text)
```

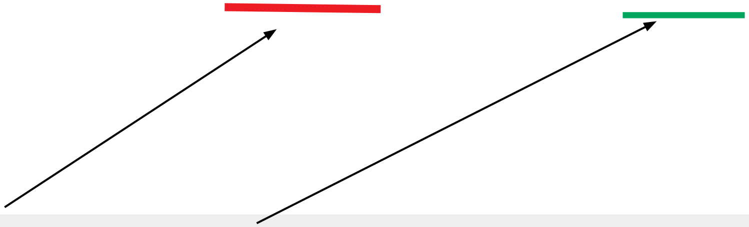
In Practice

Replace in R

```
gsub(REGEX, REPLACEMENT, VECTOR)
```

Of course, this gets more useful in certain circumstances

```
text ← c(“Bourdieu, Paris 1932 – Paris 2002”)
```



- `gsub(“.*(\\d{1,4}).*(\\d{1,4})”, “\\1 - \\2”, text)`

Will return 1932 - 2002

In Practice

Resources

Regex can be hard, but most of what you'll have to do it easy.

In addition to this, there are countless resources online

- http://regex101.com → Test your regex (and add another \ in R)
- https://www.rexegg.com/

Conclusion

8 h of lecture, 8h of labs, sweat, headaches, data & objects, tears and frustration... but in the end, it all boils down to this:

1. Get the source code
2. Select some elements
3. Store
4. Automate
5. Clean-up your data set

LOONEY TUNES
"That's all Folks!"