

GAPLEN #5

EL MARAVILLOSO MUNDO DE LAS REGEX

#GAPLEN (grupo de aprendizaje de procesamiento del lenguaje natural), una idea de Lingwars

Febrero 2018 @ Medialab-Prado

REGEX

Describen patrones en cadenas de caracteres o *strings*.



Buscar y reemplazar

Buscar

Reemplazar por

☐ Distinguir entre mayúsculas y minúsculas

☐ Solo palabras completas

Reemplazar

Buscar Reemplazar Buscar en archivos Marcar

Buscar: \t

Reemplazar con:

☐ En la selección

☐ Backward direction

☐ Solo palabras completas

☐ Coincidir MAYÚSCULAS/minúsculas

☒ Buscar en todo el documento

Modo de búsqueda

☐ Normal

☐ Extendido (\n, \r, \t, \x...)

☒ Expresión regular ☐ . se ajusta a línea

☒ Transparencia

☒ Al perder el foco

☐ Siempre

CARACTERES LITERALES Y METACARACTERES

Literales: corresponden con caracteres de la string tal cual.
Si ponemos `a` nos encuentra la primera `a` (*) de `Fulanita`.

(*) depende de las opciones del programa / la función que estemos usando al programar

Metacaracteres: tienen otros significados. Si queremos usarlos de forma literal hay que *escaparlos*. Son: `\`, `^`, `$`, `.`, `|`, `?`, `*`, `+`, `(`, `)`, `[` y `{`

Para encontrar `1+1=2` hay que poner: `1\+1=2`

SETS Y RANGOS DE CARACTERES

Con los corchetes cuadrados `[]` buscamos un carácter de entre los que metamos dentro de los corchetes.

Por ejemplo, `gr[ae]y` encontrará tanto `gray` como `grey`.

Con el guion `-` indicamos un rango. Muy útil junto a los corchetes:

1. Prólogo

`[0-9]\.`

1. Prólogo

2. Introducción

2. Introducción

SETS Y RANGOS DE CARACTERES

Se puede usar más de un rango a la vez en una misma regex; por ejemplo, podemos hacer `[a-z]` case-insensitive: `[a-zA-Z]`

Los corchetes también nos sirven, junto con el acento circunflejo `^`, a negar caracteres:

`q[^u]` ¿qué encuentra en: `qué` `quién`? `qué` `quién` `qé` `qién`

Si metemos más caracteres dentro de los corchetes, niega todos:

`q[^ui]` `qué` `quién` `qé` `qién`

`[^] [- -] [^]` de ninguna manera `a-dijo` “Señores” `-comenzó-` no vamos a...

REPETICIÓN Y ALTERNANCIA

Con `?` indicamos que el carácter anterior es opcional. Por ejemplo: `colou?r` encuentra `colour` y también `color`.

Con `+` buscamos que el carácter anterior salga una o más veces.

Ciudad del Cabo, una ciudad de 3,7 millones de habitantes

`+`

El `*` se puede entender como una mezcla de `?` y `+`: indicamos que es opcional, pero que, si sale, lo haga una o más veces.

`[«" ` '][a-zA-Z]*[' ´"»]` para cualquier palabra entrecomillada

REPETICIÓN Y ALTERNANCIA

Podemos indicar el número exacto de apariciones que nos interesan con las llaves, o un rango si separamos los números mediante comas. Por ejemplo, para buscar un número:

entre 1000 y 9999 `[1-9][0-9]{3}`

entre 100 y 99999 `[1-9][0-9]{2,4}`

La pleca `|` permite la alternancia entre dos opciones:

`cat|dog` encontrará: About `cats` and `dogs`

COMODINES

Ciertas letras normales se pueden escapar, como hacíamos con los metacaracteres, para usos especiales. Tienen la particularidad de que al ponerlos en mayúsculas, los negamos.

Con `\d` encontramos dígitos: `\d = [0-9]`; `\D = [^0-9]`

Con `\w` encontramos caracteres alfanuméricos y la barra baja:
`\w = [a-zA-Z0-9_]`; `\W = [^a-zA-Z0-9_]`

Con `\s` encontramos espacios, tabuladores y saltos de línea:
`\s = [\t\r\n]`; `\S = [^\t\r\n]`

CARACTERES INVISIBLES Y EL PUNTO

Con `\t` encontramos el tabulador.

Con `\r` encontramos el carácter de retorno de carro.

Con `\n` encontramos el carácter de nueva línea.

En Windows, por defecto, al crear un nuevo párrafo en los programas de procesamiento de texto, en realidad se están imprimiendo `\r` y `\n`. En Linux, se imprime solo `\n`.

El punto `.` encuentra casi todo: todo menos precisamente los saltos de línea (aunque esto es configurable). ¡Hay que tener mucho cuidado con el punto!

ANCLAS

Las anclas no se corresponden con ningún carácter, sino con posiciones. Su particularidad reside en que si lo que queremos es reemplazar una string por otra, no tenemos que ponerlos en la cadena meta.

`^` indica el principio de una línea y `$`, el final. Los límites de las palabras también los podemos encontrar, con `\b` (y usar su contrario, `\B`).

`\bt[aoe]s?\b` era `to` más parecido una de `tas` mejores obras de...

AVARICIA Y PEREZA

Los cuantificadores que hemos visto antes son, por defecto, avariciosos; eso quiere decir que abarcarán todo lo que puedan.

Las etiquetas HTML son ejemplos típicos: Hay una `primera cosa` que quiero enfatizar y también una `segunda`.

La regex `<.+>` a priori es muy succulenta para cazar cada etiqueta, pero va a hacer *match* con: Hay una `primera cosa` que quiero enfatizar y también una `segunda`.

Tampoco nos sirve `<[^]+>`; pillaría `segunda`

Con `?` la hacemos perezosa: `<.+?>` captura solo las etiquetas.

AGRUPACIÓN

Con `()` se pueden agrupar varios caracteres, útil para:

- poder aplicarle a toda esa cadena un mismo cuantificador
- capturar ese grupo, es decir, poder usarlo después (en la misma regex de búsqueda o en el reemplazo)

B: `([^-])[--]([^-])` R: `\1 - \2`

de ninguna maneraa-dijo

de ninguna manera - dijo

“Señores”-comenzó-no vamos a...

“Señores” - comenzó - ”no vamos a...

B: `\bt([aoe]s?)\b` R: `\1`

era to más parecido

era lo más parecido

una de tas mejores obras de...

una de las mejores obras de...

EJERCICIOS

Para encontrar códigos RGB de colores tipo `#63ffed` `#daffbb` `#ff787b`... ¿qué regex tendríamos que escribir?

```
#[0-9a-f]{6}
```

La regex para encontrar las 3 formas de referirse a las regex: `regex`, `regexp`, `regular expression` es...

```
reg(ular )?exp?(ression)?
```

Hay que encontrar las palabras de la columna de la izquierda, SIN encontrar las de la derecha (ej. 1-2 de [SketchEngine](#))

pit	pt	rap them	sap tray	aleht	tarreth
spot	Pot	tapeth	87ap9th	happy them	ddapdg
spate	peat	apth	apothecary	tarpth	apples
slap	two part	wrap/try		Apt	shape the
respite				peth	

```
(re)?s?(la)?p[ioa ]t(wo)?e?
```

```
[8w]?[rts7]?ap[ e\\/o9]?th?[er]*[myac]?[ya]?(ry)?
```

EN PYTHON

Librería re: <https://pymotw.com/2/re/>

```
>>> import re
>>> re.search(r'[a-z]', 'Fulanita')
<_sre.SRE_Match object at 0x6ffffdd85e0>
>>> re.search(r'[A-Z]', 'Fulanita')
<_sre.SRE_Match object at 0x6ffffdd8648>
>>> re.match(r'[a-z]', 'Fulanita')
>>> re.match(r'[A-Z]', 'Fulanita')
<_sre.SRE_Match object at 0x6ffffdd85e0>
```

`re.search()` busca en toda la string; `re.match()`, solo en el principio

Podemos usarlos como condiciones:

```
>>> result = re.match(r'[a-z]', 'Fulanita')
>>> print result
None
```

EN PYTHON

```
import re

# Detectar los saludos que son multipalabra
saludos = ["hola", "adios", "hasta luego", "nos vemos"]

def detectar_locuciones(lista):
    for elemento in lista:
        if re.search(r' ', elemento):
            print elemento

detectar_locuciones(saludos)
```

¿Qué pasa si ejecutamos esto?

EN PYTHON

```
# Crear los participios adecuados
infinitivos = ["comprometer", "abrir", "manejar", "absorber", "congelar",
"estudiar", "descubrir", "correr"]

def crear_participios(verbos):
    for verbo in verbos:
        if re.search(r'er$', verbo):
            participio = re.sub(r'er$', 'ido', verbo)
            print participio
        elif re.search(r'ar$', verbo):
            participio = re.sub(r'ar$', 'ado', verbo)
            print participio
        elif re.search(r'ir$', verbo):
            participio = re.sub(r'rir$', 'ierto', verbo)
            print participio

crear_participios(infinitivos)
```

¿Qué pasa si ejecutamos esto?

RECURSOS

Esta presentación es una adaptación de la guía rápida de regular-expressions.info

[Regex101](#): para comprobar si una regex hace lo que quieres.

[Regex Aracne](#): lista de regex útiles para corregir textos generados por OCR