Expresiones regulares

Procesamiento de Lenguaje Natural

Dra. Helena Gómez Adorno
helena.gomez@iimas.unam.mx

Dra. Gemma Bel
gbele@iingen.unam.mx

Correo del curso:
pln.cienciadedatos@gmail.com

Asistente:
Luis Ramón Casillas

Expresiones Regulares

- Patrones de texto que definen la forma que debe tener una cadena de texto
- Ejemplos de uso:
  - Verificar si un valor ingresado en un formulario HTML es una dirección válida de correo electrónico
  - Verificar si la palabra "color" o la palabra "colores" aparece en un documento con solo un escaneo
  - Extraer partes específicas de un texto: C.P.
  - Reemplazar porciones de texto: color por rojo
  - Divida un texto más grande en partes más pequeñas: signos de puntuación

Relevancia y propósito

- Las expresiones regulares son omnipresentes: ofimática, javaScript, lenguajes de programación.
- Todavía no están generalizadas en el kit de herramientas del programador moderno.
- Curva de aprendizaje difícil. Pueden ser difíciles de dominar y muy complejas de leer si no se escriben con cuidado.

> *"Some people, when confronted with a problem, think "I know, I'll use regular expressions." Now they have two problems."*
>
> *-Jamie Zawinski, 1997*

“Sabores” de expresiones regulares

- Muchas de las implementaciones en lenguajes de programación o herramientas se basan en expresiones regulares con “sabor a **Perl**” *(Perl flavor)*.
- El estándar de IEEE **POSIX** ha intentado estandarizar y dar mejor soporte Unicode a la sintaxis y los comportamientos de la expresión regular. El sabor **POSIX** (POSIX flavor).
- El módulo estándar de Python para expresiones regulares – `re` – solo admite expresiones regulares al estilo **Perl**.
- Hay un esfuerzo por escribir un nuevo módulo de expresiones regulares con mejor soporte de estilo POSIX en `https://pypi.python.org/pypi/regex`.
- En este curso, aprenderemos cómo aprovechar solo el módulo `re` estándar.

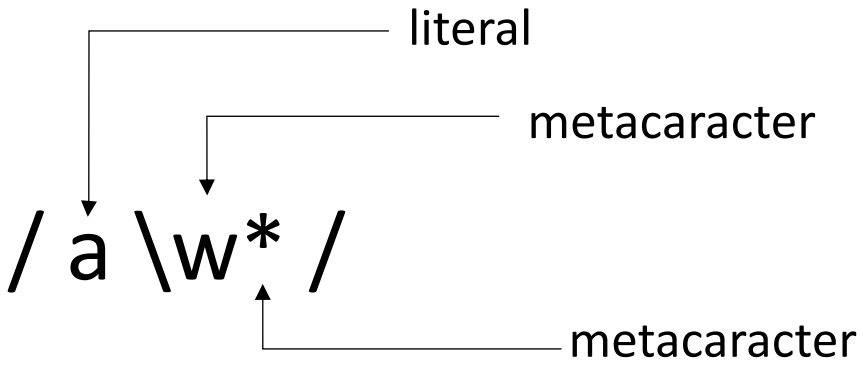
https://docs.python.org/3.5/library/re.html

Uso de la expresión regular

- Uso del asterisco (*) y el signo de interrogación (?) para encontrar patrones:
  - El **(?)** coincidirá con un solo carácter con cualquier valor en un nombre de archivo. Por ejemplo, un patrón como `archivo?.xml` coincidirá con `archivo1.xml`, `archivo2.xml`, y `archivo3.xml`, pero no coincidirá con `archivo99.xml`.
  - Cuando se usa **(*)**, se acepta cualquier número de caracteres con cualquier valor. En el caso del `archivo*.xml`, coincidirá con todo lo que comience con el archivo, seguido de cualquier número de caracteres de cualquier valor y termine con .xml.
- Dos tipos de componentes: **literales** (`archivo` y `.xml`) y **metacaracteres** (`?` O `*`).

Sintaxis de la expresión regular

- Es un patrón de texto que consiste en caracteres comunes (por ejemplo, letras **a** la **z** o números del 0 al 9) y caracteres especiales conocido como **metacaracteres**. Este patrón describe las cadenas que coincidirían cuando se aplica a un texto.
- Expresión regular que coincida con cualquier palabra que comience con **a**:



Literales

- La forma más simple de coincidencia de patrones en expresiones regulares. Tendrán éxito siempre que se encuentre ese literal.
- Si aplicamos la expresión regular `/ zorro /` para buscar la frase: *El **zorro** salta sobre el perro perezoso*, encontraremos **una** coincidencia.
- Sin embargo, también podemos obtener **varios** resultados en lugar de solo uno, si aplicamos la expresión regular `/ ser /` a la frase: ***ser**, o no **ser***:

Uso de metacaracteres como literales

Expresión regular: / `(esto está adentro)` /
Texto: *esto está afuera (esto está adentro)*,
Resultado: *esto está adentro*

- Esto pasa porque los paréntesis son **metacaracteres** y tienen un significado especial.
- Hay tres mecanismos para usar **metacaracteres** como si fueran literales:
  1. Precediendo los metacaracteres con una barra diagonal inversa: / `\(esto está adentro\)` /
  2. En python, usando el método `re.escape` para escapar de los caracteres no alfanuméricos que pueden aparecer en la expresión.
  3. Entrecomillar con `\Q` y `\E`. En los sabores que la soportan, ejemplo:
     /`\Q(\E esto esta dentro \Q)\E` /.

Metacaracteres

- Diagonal invertida \
- Acento circunflejo ^
- Signo de dólar $
- Punto .
- Símbolo de tubería |
- Signo de interrogación ?
- Asterisco *
- Signo más +
- Abrir paréntesis (
- Cerrar paréntesis)
- Apertura de corchete [
- Apertura de llave {

Clases de caracteres o conjunto de caracteres

- Permite definir un caracter que coincidirá si alguno de los caracteres definidos en el conjunto está presente.
- Para definir una clase de caracteres, debemos abrir corchete `[`, luego cualquier carácter aceptado, y finalmente cerrar corchete `]`.
  - Una expresión regular que coincida con las palabras “estimado” y “estimada”: **`/estimad[oa]/`**
- También es posible usar un rango de caracteres, usando el símbolo de guión `(-)` entre dos caracteres relacionados:
  - Para hacer coincidir cualquier letra minúscula, podemos usar `[a-z]`.
  - Para hacer coincidir cualquier dígito, podemos definir la clase de caracteres `[0-9]`.

Combinando clases de rangos de caracteres

- Si queremos hacer coincidir cualquier carácter alfanumérico en minúscula o mayúscula, podemos usar `[0-9a-zA-Z]`. Esto puede escribirse alternativamente utilizando el mecanismo de unión: `[0-9[a-z[A-Z]]]`.
- Ejemplos de rangos de caracteres:

```
/[a-z]/ letras minusculas
/[A-Z]/ letras mayusculas
/[0-9]/ numeros
/[,'¿!¡;:\.\?]/ caracteres de puntuacion
                -la barra invertida hace que
                no se consideren como comando
                ni en punto ni el interrogante
/[A-Za-z]/      letras del alfabeto (del ingles claro ;)
/[A-Za-z0-9]/   todos los caracteres alfanumericos habituales
                -sin los de puntuacion, claro-
/[^a-z]/        El simbolo ^ es el de negación. Esto es decir
                TODO MENOS las letras minusculas.
/[^0-9]/        Todo menos los numeros.
```

Clase de caracteres predefinidas

Elemento	Descripción
.	Este elemento coincide con cualquier carácter excepto el salto de línea `\n`
\d	Coincide con cualquier dígito decimal; esto es equivalente a la clase `[0-9]`
\D	Coincide con cualquier carácter que no sea un dígito; esto es equivalente a la clase `[^0-9]`
\s	Coincide con cualquier carácter de espacio en blanco; esto es equivalente a la clase `[\t\n\r\f\v]`
\S	Esto coincide con cualquier carácter que no sea de espacio en blanco; esto es equivalente a la clase `[^\t\n\r\f\v]`
\w	Esto coincide con cualquier caracter alfanumérico; esto es equivalente a la clase `[a-zA-Z0-9_]`
\W	Esto coincide con cualquier carácter no alfanumérico; esto es equivalente a `[^a-zA-Z0-9_]`

Alternancia

- Uso del símbolo de tubería | para coincidir con un conjunto de expresiones regulares
- Si queremos hacer coincidir si encontramos la palabra "sí" o la palabra "no”: `/ si|no /`
- Si queremos hacer coincidir si encontramos la palabra "sí”, "no” o “quizas”: `/ si|no|quizás /`
- La siguiente expresión con que coincide?: `/ Licencia: si|no /`
  - Licencia: si **o** Licencia: no
  - ✓ Licencia: si **o** no
- Usar paréntesis para definir grupos de alternancia :

```
/ Licencia: (si|no) /
```

Cuantificadores

- Mecanismos para definir cómo se puede repetir un caracter, metacaracter o conjunto de caracteres

Símbolo	Nombre	Cuantificación de carácter previo
?	Símbolo de interrogación	Opcional (0 o 1 repeticiones)
*	Asterisco	Cero o mas repeticiones
+	Signo mas	Una o mas repeticiones
{n,m}	Llaves	Entre *n* y *m* repeticiones -> {n},{n,},{,n}

- El símbolo `(?)` puede ser usado para coincidir la palabra `carro` y su plural `carros`: `/ carros? /`
- Encontrar un número telefónico en el formato:

555-555-555, 555 555 555, o 555555555

Cuantificadores codiciosos y reacios

- Si aplicamos un cuantificador como este `/".+"/` a un texto como el siguiente: *inglés "Hello", español "Hola"*
- Podemos esperar que coincida con ”Hello" y "Hola", pero realmente coincidirá con ”Hello", español "Hola”
- Este comportamiento se llama **codicioso** y es uno de los dos comportamientos posibles de los cuantificadores en Python: **codiciosos** y **no codiciosos** (también conocidos como **reacios**).
  - Por defecto se aplica al comportamiento **codicioso** en los cuantificadores. Un cuantificador **codicioso** intentará hacer coincidir tanto como sea posible para obtener el mayor resultado posible.
  - El comportamiento **no codicioso** puede solicitarse agregando un signo de interrogación adicional al cuantificador: `??`, `*?` o `+?`. Un cuantificador **reacio** se comportará como el opuesto exacto de los codiciosos. Intentará tener la menor coincidencia posible: `/".+?"/`

Comparadores de límites

- Se usan para hacer coincidir una línea completa, el inicio de una línea o incluso el final de la línea
- Los comparadores de límites son una serie de identificadores que corresponderán a una posición particular dentro de la entrada.

Comparador	Descripción
^	Coincide con el comienzo de una línea
$	Coincide con el final de una línea
\b	Coincide con un límite de palabra
\B	Coincide con lo opuesto de \b. Cualquier cosa que no sea un límite de palabras
\A	Coincide con el comienzo de la entrada
\Z	Coincide con el final de la entrada

Comparadores de límites

- Se comportan diferente en diferentes contextos. Por ejemplo: `\b` depende de la configuración local, ya que diferentes lenguajes pueden tener límites de palabras diferentes
- Ejemplo `(^)`: `/^Nombre/`
  - Si queremos asegurarnos de que después del nombre, solo haya caracteres alfabéticos o espacios hasta el final de la línea: `/^Nombre:[\sa-zA-Z]+$/`
- Ejemplo `(\b)`: `/\bpara\b/`
  - Es muy útil cuando queremos trabajar con palabras aisladas y no queremos crear conjuntos de caracteres con cada carácter que puede dividir nuestras palabras (espacios, comas, dos puntos, guiones, etc.).

Expresiones regulares con Python

- Las expresiones regulares son implementadas con el modulo `re`: https://docs.python.org/3/library/re.html
- Importar el módulo: `>>> import re`
- Para coincidir un patrón, se compila transformando en **bytecode**, este código luego se ejecuta por un motor escrito en C:

```
>>> pattern = re.compile(r'\bfoo\b')
<_sre.SRE_Match object; span=(0, 3), match='foo'>
```

- Si uno o mas caracteres coiniden con la expresión regular retorna un objeto `match`: https://docs.python.org/3/library/re.html#match-objects

Bloques de construcción para Python Regex

- En Python, hay dos objetos diferentes que tratan con Regex:
  - **RegexObject**: también se conoce como *Pattern Object*. Representa una expresión regular compilada.
  - **MatchObject**: representa el patrón de coincidencia.

RegexObject

- La compilación de una expresión regular produce un *Pattern Object* reutilizable
- Proporciona todas las operaciones que se pueden realizar, como la **coincidencia** y la **búsqueda** de todas las subcadenas que coinciden con una expresión regular determinada

RegexObject

- Dos formas de coincidir patrones y ejecutar las operaciones
- Si queremos volver a usar la expresión regular, podemos usar el siguiente código:

```
>>> pattern = re.compile("<HTML>")
>>> pattern.match("<HTML>")
<_sre.SRE_Match object; span=(0, 6), match='<HTML>’>
```

- Por otro lado, podemos realizar directamente la operación en el módulo usando la siguiente línea de código:

```
>>> re.match("<HTML>", "<HTML>")
<_sre.SRE_Match object; span=(0, 6), match='<HTML>'>
```

Operaciones de coincidencia y búsqueda

- **match(string[, pos[, endpos]]):** hace coincidir un patrón compilado solo al comienzo de la cadena. Retorna `none` si no hay coincidencia.

```
>>> pattern = re.compile("o")
>>> pattern.match("dog") #No coincide porque "o" no está al inicio de "dog".
>>> pattern.match("dog", 1) #Coincide porque "o" es el 2do caracter de "dog".
 <re.Match object; span=(1, 2), match='o'>
```

- **search(string[, pos[, endpos]]):** hace coincidir el patrón en cualquier ubicación de la cadena, no solo al inicio.

```
>>> pattern = re.compile("d")
>>> pattern.search("dog") #Coincide en el indice 0
<re.Match object; span=(0, 1), match='d'>
>>> pattern.search("dog", 1) #No coincide; la busqueda no incluye la "d"
```

Operaciones de coincidencia y búsqueda

- **findall(string[, pos[, endpos]]):** retorna una lista de todas las ocurrencias del patrón que no se superponen. Incluye cadena vacia.

```
>>> pattern = re.compile("\w+")
>>> pattern.findall("hello world")
['hello', 'world’]
```

- **finditer(string[, pos[, endpos]]):** Igual que el anterior, pero retorna un iterador en el que cada elemento es un `MatchObject`

```
>>> pattern = re.compile("(\w+) (\w+)")
>>> it = pattern.finditer("Hello world hola mundo")
>>> match = next(it)
>>> match.groups()
('Hello', 'world')
>>> match.span()
(0, 11)
```

Modificando una cadena

- **split(string, maxsplit=0):** permite dividir una cadena usando expresiones regulares.

```
>>> pattern = re.compile("\W")
>>> pattern.split("Beautiful is better than ugly", 2)
['Beautiful', 'is', 'better than ugly']
```

- **sub(repl, string, count=0):** devuelve la cadena resultante después de reemplazar el patrón coincidente en la cadena original con otra.

```
>>> pattern = re.compile("[0-9]+")
>>> pattern.sub("-", "order0 order1 order13")
'order- order- order-'
```

- **subn(repl, string, count=0):** Realiza la misma operación que `sub()`, pero devuelva una tupla `(nueva_cadena, numero_substituciones)`.

MatchObject

- Representa un patrón de coincidencia, que se obtiene cada vez que se ejecuta una operación: `match`, `search` o `finditer`
- Métodos importantes:
  - **group([group1, …]):** retorna subgrupos de la coincidencia, uno por uno.
  - **groups([default]):** retorna subgrupos de la coincidencia, en una tupla.
  - **groupdict([default]):** se utiliza en caso en que los grupos son nombrados.
  - **start([group]):** retorna el indice donde se encontró el inicio del patron.
  - **end([group]):** retorna el indice donde se encontró el fin del patron.
  - **span([group]):** retorna el indice de inicio y el fin.
  - **expand(template):** devuelve la cadena después de reemplazarla con referencias en la cadena de la plantilla.

Banderas de compilación

- Se usan para modificar el comportamiento estándar de los patrones.
- Ejemplo de banderas más usadas en Python 3

Bandera	Descripción
`re.I` `re.IGNORECASE`	Se usa para coincidir mayúsculas o minúsculas
`re.s` `re.DOTALL`	Hace que el metacaracter "." coincida con cualquier carácter, incluyendo el salto de línea
`re.A` `re.ASCII`	Hace que \w, \W, \b, \B, \d, \D, \s y \S realice una coincidencia ASCII solamente en lugar de una coincidencia Unicode completa.
`re.M` `re.MULTILINE`	^: ahora coincide al principio de la cadena y al comienzo de cada nueva línea. $: coincide al final de la cadena y al final de cada línea. Concretamente, coincide justo antes del carácter de nueva línea.

Caso de estudio:
Expresiones regulares para fechas

- Variaciones de fechas para 23 de Octubre de 2002

23-10-2002

23/10/2002

23/10/02

10/23/2002

23 Oct 2002

23 de Octubre de 2002

Oct 23, 2002

Octubre 23, 2002

Expresiones regulares para fechas (números)

23-10-2002
23/10/2002
23/10/02
10/23/2002

```
import re
dateStr = '23-10-
2002\n23/10/2002\n23/10/02\n10/23/2002\n23 Oct 2002\n23
de Octubre de 2002\nOct 23, 2002\nOctubre 23, 2002\n’

>>> re.findall('\d{2}[/-]\d{2}[/-]\d{4}', dateStr)
['23-10-2002', '23/10/2002', '10/23/2002’]

>>> re.findall('\d{2}[/-]\d{2}[/-]\d{2,4}', dateStr)
['23-10-2002', '23/10/2002', '23/10/02', '10/23/2002’]
```

Expresiones regulares para fechas (letras)

23 Oct 2002
23 de Octubre de 2002
Oct 23, 2002
Octubre 23, 2002

```
>>>re.findall('\d{2} (Ene|Feb|Mar|Abr|May|Jun|Jul|Ago|Sep|Oct|Nov|Dic) \d{4}',
dateStr)
```

['Oct']

```
>>>re.findall('\d{2} (?:Ene|Feb|Mar|Abr|May|Jun|Jul|Ago|Sep|Oct|Nov|Dic) \d{4}',
dateStr)
```

['23 Oct 2002']

```
>>>re.findall('\d{2} (?:de)?(?:Ene|Feb|Mar|Abr|May|Jun|Jul|Ago|Sep|Oct|Nov|Dic)[a-z]*
(?:de )?\d{4}', dateStr)
```

['23 Oct 2002', '23 de Octubre de 2002']

```
>>> re.findall(r'(?:\d{2} )?(?:de )?(?:
Ene|Feb|Mar|Abr|May|Jun|Jul|Ago|Sep|Oct|Nov|Dic)[a-z]* (?:de )?(?:\d{2}, )?\d{4}',
dateStr)
```

['23 Oct 2002', '23 de Octubre de 2002', 'Oct 23, 2002', 'Octubre 23, 2002']

Expresiones regulares para fechas

1-10-2002
1 Oct 2001

```
#ahora tenemos en el día un solo digito, que pasa con nuestras expresiones regulares?
>>> dateStr = '1-10-2001\n1 Oct 2001’
>>> re.findall('\d{2}[/-]\d{1,2}[/-]\d{2,4}', dateStr)
[]

>>> re.findall('\d{1,2}[/-]\d{1,2}[/-]\d{2,4}', dateStr)
['1-10-2001’]

>>> re.findall(r'(?:\d{1,2} )?(?:de
)?(?:Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec)[a-z]* (?:de )?(?:\d{1,2},
)?\d{4}', dateStr)
['1 Oct 2001']
```