Expresiones regulares

Universitat Politècnica de València

Índice

1. Introducción y objetivos **1**

2. Encodings, Unicode y su tratamiento en Python 3 **2**
2.1. Encodings y Unicode 2
2.2. Clases `str` y `bytes` en Python 3 2
2.3. Encoding y lectura/escritura de ficheros 4
2.4. Literales de `str` y de `bytes` en Python 3 4

3. Expresiones regulares **6**
3.1. Funciones y métodos del módulo `re` de Python 6
3.2. Sintaxis de las expresiones regulares 11
3.2.1. Metacaracteres 11
3.2.2. Conjuntos de caracteres 13
3.2.3. Disyunción 13
3.2.4. Cuantificadores 13
3.2.5. Grupos 16

1. Introducción y objetivos

- Adquirir conocimientos básicos sobre cómo se codifican los caracteres, qué es Unicode, qué son las codificaciones (o *encodings*) y cómo se tratan en Python 3.
- Ampliar tu conocimiento de la sintaxis de Python para expresar literales de cadenas (incluyendo las cadenas de tipo *raw*) y saber distinguir las cadenas y la clase `bytes`.
- Qué son las expresiones regulares.
- Aprender a utilizar las expresiones regulares en Python (con el módulo o biblioteca `re`).

Las expresiones regulares nos permiten especificar patrones de búsqueda mediante una secuencia de caracteres que sigue unas reglas determinadas. Aunque pueden existir variantes (o *flavours*) en la forma de indicar las expresiones regulares (ej: POSIX y Perl Compatible son dos formas ligeramente diferentes de escribir reglas de expresiones regulares), la parte que tienen

en común y el tipo de construcciones que ofrecen es básicamente similar. Muchos lenguajes de programación comparten la misma sintaxis para escribir las reglas e incluso comparten las bibliotecas o *"motores"* utilizados. Por ejemplo, en Python (al igual que en otros lenguajes) se utiliza prácticamente[1] el mismo formato que en el lenguaje Perl.
Antes de ver las reglas para construir estas expresiones, veamos primero unos conceptos sobre cadenas literales en Python y la diferencia entre las clases `str` y `bytes`.

2. Encodings, Unicode y su tratamiento en Python 3

2.1. Encodings y Unicode

Si tienes tiempo y ganas, el siguiente enlace[2] proporciona una introducción interesante sobre la existencia de diversas codificaciones y los problemas que eso conlleva.
A continuación vamos a resumir más o menos lo imprescindible. El estándar ASCII tiene su historia y razón de ser, es evolución de otros formatos previos, lo importante es saber que es de 7 bits y, por tanto, define los primeros 128 caracteres que se pueden guardar en un solo byte. Esta codificación se extendió aprovechando los 128 códigos restantes que es posible guardar en un byte. Lamentablemente, esta extensión se realizó de manera independiente en diversos paises o regiones, dando lugar a diversas codificaciones o *encodings* incompatibles entre sí. Por si fuera poco, algunos idiomas requieren un repertorio de símbolos mucho mayor a 256, con lo que no bastaría un solo byte para representarlos. Todo esto explica el origen del Unicode Consortium[3] cuyo objetivo (ambicioso) es crear una codificación de todos los sistemas de escritura.
Unicode no es un *encoding* propiamente dicho, sino una forma de estandarizar un conjunto de *code points*.[4] De todas formas, aunque puede haber muchas formas de codificar un carácter Unicode, el propio Unicode define varios *encodings*, entre ellos el UTF-8[5]. UTF-8 tiene, entre otras, estas características:

- es compatible hacia abajo con ASCII,
- es de longitud variable, de manera que los caracteres pueden ocupar entre 1 y 4 bytes.

2.2. Clases `str` y `bytes` en Python 3

Python 3 guarda todas las cadenas (la clase `str`) en Unicode. Es decir, una cadena es una secuencia (inmutable) de caracteres Unicode. Esto significa que, en principio, no debe importarnos cómo está representada internamente una cadena (o instancia de la clase `str`). Digamos que no necesitamos plantearnos temas de codificación cuando estamos manipulando cadenas en Python, pero sí nos vamos a tener que enfrentar al problema de las codificaciones (o del *encoding*) al leer o escribir a fichero, a un terminal, etc.

[1] `https://stackoverflow.com/questions/12022443/which-regular-expression-flavour-is-used-in-python`
[2] `https://www.joelonsoftware.com/2003/10/08/the-absolute-minimum-every-software-developer-absolutely-positively-must-know-about-unicode-and-character-sets-no-excuses`
[3] `https://en.wikipedia.org/wiki/Unicode`
[4] "Code point" es la forma de llamar en abstracto un carácter para distinguirlo así de la forma concreta en que se puede representar en pantalla (por ejemplo).
[5] `https://en.wikipedia.org/wiki/UTF-8`

Para poder trabajar con codificaciones o *encodings*, ya que a veces es necesario, Python 3 distingue entre:

str son las cadenas que has estado estudiando desde el curso pasado y, como hemos dicho, son secuencias de caracteres Unicode,

bytes (a veces llamado "raw bytes") las instancias de esta clase Python representan una secuencia de bytes. Es importante resaltar que dicho vector de bytes **no tiene ningún *encoding* asociado**.

Es posible pasar de `str` a `bytes` con el método `encode` y de `bytes` a `str` con el método `decode`, tal y como muestra el siguiente ejemplo (observa que los literales de tipo `bytes` llevan una letra `b` antes de las comillas):

```
>>> 'piña'.encode('utf-8') # pasamos de str a bytes
b'pi\xc3\xb1a' # un literal de la clase bytes
>>> 'piña'.encode('latin-1') # misma cadena a bytes con OTRO ENCODING
b'pi\xf1a' # otro literal de la clase bytes de longitud diferente
>>> type(b'pi\xc3\xb1a')
<class 'bytes'>
>>> b'pi\xf1a'.decode('latin-1')
'piña'
>>> b'pi\xf1a'.decode('utf-8') # intentamos un encoding diferente
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xf1 in position
   2: invalid continuation byte
```

Como se puede apreciar en el ejemplo anterior, estamos pasando un *encoding* como parámetro a los métodos `encode` y `decode`, si bien es posible no dar ninguno y, en ese caso, tomará un valor por defecto (`'utf-8'`). Tanto `encode` como `decode` reciben un segundo argumento sobre la forma de tratar los errores de codificación, no vamos a entrar en detalle sobre este argumento. También se puede observar que, si intentamos convertir a cadena una secuencia de `bytes` con un *encoding* que no es adecuado, podría dar error. Lamentablemente no siempre da error, ya que una misma secuencia de `bytes` puede convertirse en diversas cadenas válidas, aunque diferentes, según el *encoding*. Dicho de otra manera: podría ocurrir que se generase una cadena diferente a la original de manera totalmente inadvertida. Por este motivo, *tener una secuencia de bytes sin saber el* encoding *asociado no garantiza que podamos recuperar el contenido exacto.* En algunos casos el *encoding* se podría deducir por ensayo error, pero normalmente puede existir una ambiguedad insalvable y no ser capaces de saber lo que pone realmente.
Como ya hemos dicho, en Python 3 las cadenas son secuencias de Unicode y no tenemos que preocuparnos de si luego alguno de esos caracteres ocupa varios bytes en una codificación determinada. Sin embargo, una secuencia de `bytes` se accede byte a byte (observa que el en segundo bucle se imprimen 5 cosas en lugar de 4):

```
>>> for i in 'piña': print(i) # iteramos en una instancia str
...
p
i
```

```
ñ
a
>>> for i in 'piña'.encode('utf-8'): print(i) # en una instancia bytes
...
112
105
195
177
97
```

Python 3 no nos deja concatenar valores de tipo `str` y de tipo `bytes`:

```
>>> 'hola' + 'mundo'
'holamundo'
>>> 'hola' + b'mundo'
TypeError: must be str, not bytes
>>> b'hola' + b'mundo'
b'holamundo'
```

Se recomienda siempre trabajar con `str` y únicamente codificar y decodificar de cara al exterior (*unicode sandwich*, ver el siguiente enlace[6]).

2.3. Encoding y lectura/escritura de ficheros

Cuando abrimos un fichero, podemos elegir si es para leer o escribir, pero también si es para leer en modo texto o en modo binario:

- El modo **texto** usa el modificador "`t`" pero, al ser el modo por defecto, no es preciso ponerlo. En este modo las cadenas se convierten a *raw bytes* al escribir y se pasan de *raw bytes* a cadena de manera transparente usando el *encoding* proporcionado en el argumento opcional de la función `open` (tiene como valor por defecto lo que devuelva `locale.getpreferredencoding()`).

- El modo **binario** (que utiliza el modificador "`b`") únicamente trabaja con valores de la clase `bytes`, de modo que no se realiza *encoding* o *decoding* alguno (y tampoco se tratan de manera especial los cambios de línea).

2.4. Literales de `str` y de `bytes` en Python 3

En las secciones anteriores hemos visto que en Python se puede poner la letra "`b`" precediendo una cadena para indicar que se trata de un literal de la clase `bytes`:

```
>>> type( 'hola' )
<class 'str'>
>>> type( b'hola' )
<class 'bytes'>
```

[6] `https://nedbatchelder.com/text/unipain.html`

En esta sección hablaremos un poco más sobre los literales de cadenas y de `bytes`. Puedes encontrar información muy detallada en la documentación oficial[7] de Python 3. Básicamente, una cadena puede venir precedida por uno de estos prefijos:

```
"r"|"u"|"R"|"U"|"f"|"F"|"fr"|"Fr"|"fR"|"FR"|"rf"|"rF"|"Rf"
```

El "puede venir" de la frase anterior es porque el prefijo de cadena es opcional. En cambio, el prefijo de un literal de tipo `bytes` es obligatorio y ha de ser uno de estos:

```
"b"|"B"|"br"|"Br"|"bR"|"BR"|"rb"|"rB"|"Rb"|"RB"
```

Desglosándolo por letras:

- Si el prefijo contiene "`b`" indica que se trata de un literal de `bytes`. Los literales de tipo bytes solamente admiten caracteres ASCII:

```
>>> b'piña'
File "<stdin>", line 1
SyntaxError: bytes can only contain ASCII literal characters.
```

- Si contiene "`r`" o "`R`" es que es de tipo *raw*. Esto significa que los caracteres de escape (como la barra invertida "\") se interpretan literalmente y no escapan otros caracteres:

```
 >>> 'uno\ndos\u0065\N{new line}'
'uno\ndose\n'
>>> r'uno\ndos\u0065\N{new line}' # de tipo raw
'uno\\ndos\\u0065\\N{new line}'
```

- El prefijo `u` o `U` tiene que ver con compatibilidad hacia abajo con Python 2. Como siempre trabajamos con Python 3 vamos a ignorarlo completamente.

- El prefijo `f` tiene que ver con cadenas de formato, también llamadas *f-strings* o *string interpolation* que sin duda ya viste en la asignatura PRG. Aun así, puedes ver este enlace[8] o este otro[9] para más información:

```
>>> '2 + 2 = {2+2}'
'2 + 2 = {2+2}'
>>> f'2 + 2 = {2+2}'
'2 + 2 = 4'
```

De todo esto nos podemos quedar, para esta práctica, con la idea de que a veces resulta *muy conveniente* utilizar *raw strings* (prefijo "`r`") para expresar reglas o patrones de expresiones regulares, ya que estas reglas muchas veces incluyen caracteres como "\" que, en cadenas que no sean *raw*, necesitan ser escapadas (por ejemplo, escribir `'\\'` en lugar de `r'\'`).

[7] `https://docs.python.org/3/reference/lexical_analysis.html#literals`
[8] `https://docs.python.org/3/reference/lexical_analysis.html#f-strings`
[9] `https://realpython.com/python-f-strings`

3. Expresiones regulares

Las expresiones regulares nos permiten especificar patrones de búsqueda. Estos patrones pueden utilizarse normalmente para:

- Encontrar una cadena dentro de otra de maneras más complejas o sofisticadas que con el método `find` de la clase `str` (ejemplo: buscar direcciones de correo en un texto enorme).
- Validar o comprobar si una cadena tiene un formato determinado (ejemplo: si es una dirección de correo válida o no).
- Reemplazar texto más flexiblemente que con el método `replace` de la clase `str`.
- Romper una cadena en fragmentos de manera más flexible que con el método `split` de la clase `str`.

Veamos cómo utilizar el módulo `re` de Python para todas estas cosas.

3.1. Funciones y métodos del módulo `re` de Python

El módulo `re` se importa como de costumbre:

```
import re
```

y se puede utilizar de dos formas:

- Creando un objeto con `re.compile` y usándolo posteriormente (en el ejemplo, llamando al método `match` sobre `obj`):

  ```
  >>> obj = re.compile('casa')
  >>> type(obj)
  <class '_sre.SRE_Pattern'>
  >>> obj.match('casas')
  <_sre.SRE_Match object; span=(0, 4), match='casa'>
  >>> obj.match('escasa') # no sale nada porque devuelve None
  >>>
  ```

  En este primer ejemplo se ha creado un objeto `obj` de la clase `'_sre.SRE_Pattern'` con el patrón `'casa'` y, posteriormente, se ha utilizado `match`. Más abajo se explica qué significa crear una expresión regular con un texto (`'casa'` en este caso) y para qué sirve `match`. De momento puedes observar que el resultado de `match` puede ser un objeto (que tiene información `span=(0,4)` indicando dónde se ha hecho *matching*) o bien el valor `None`.

- Usando directamente funciones del módulo (`re.match` en el ejemplo):

  ```
  >>> # 'casa' es la expresión, se aplica a 'casas'
  >>> re.match('casa','casas')
  <_sre.SRE_Match object; span=(0, 4), match='casa'>
  >>> re.match('casa','escasa') # no sale nada porque devuelve None
  >>>
  ```

Nosotros utilizaremos indistintamente ambas aproximaciones para que veas que se pueden utilizar ambas. Cada una tiene sus pros y sus contras:

- Poner `re.match(patron,cadena)` puede resultar menos verboso y más ágil o cómodo de escribir si no vamos a reutilizar el patrón más de una vez y no queremos darle un nombre (a la variable) que indique su funcionalidad.
- Suele preferirse `re.compile` cuando vamos a utilizar el mismo patrón en muchas ocasiones y queremos darle un nombre que indique su funcionalidad. A veces se escucha que compilado puede ser algo más rápido pero realmente Python guarda las expresiones regulares en un diccionario y las reutiliza para ahorrar tiempo.[10]

En el ejemplo anterior hemos visto dos cosas:

1. Hemos utilizado una cadena (`'casa'` en el ejemplo) como patrón para crear la expresión regular. En este caso, todos los caracteres de `'casa'` se representan a sí mismos y ponerlos juntos hace que el patrón represente literalmente la cadena `'casa'`. Pronto veremos otras reglas y tipos de carácter para tener patrones que expresen más cosas.
2. Podemos observar que `'casa'` ha hecho *match* con `'casas'` pero no con `'escasa'` a pesar de que ambas cadenas contiene `'casa'` como subcadena. Compara con:

   ```
   >>> 'casa' in 'casas'
   True
   >>> 'casa' in 'escasa'
   True
   ```

   Lo que ocurre es que el método `re.match` intenta aplicar el patrón *al inicio* de la cadena, mientras que el operador `in` busca subcadenas (pero siempre literalmente, no permite crear reglas).

   `re.match` devuelve un objeto de tipo `SRE_Match`[11] o bien un `None` si el patrón no encaja:

   ```
   >>> type(re.match('casa','casas'))
   <class '_sre.SRE_Match'>
   >>> type(re.match('casa','escasa'))
   <class 'NoneType'>
   ```

La diferencia entre `match` y `search` es que `match` aplica el patrón al inicio de la cadena mientras que `search` lo busca en cualquier punto de la misma:

```
>>> re.match('casa','escasas')
# devuelve None, no encaja pq sólo busca al inicio
>>> re.search('casa','escasas')
<_sre.SRE_Match object; span=(2, 6), match='casa'>
```

Atención, `re.search` únicamente encuentra la **primera** ocurrencia:

```
>>> re.search('casa','la casaca está en casa')
<_sre.SRE_Match object; span=(3, 7), match='casa'>
```

[10] `https://stackoverflow.com/questions/12514157/how-does-pythons-regex-pattern-caching-work`
[11] Ver `https://docs.python.org/3/library/re.html#match-objects`

donde `span=(3,7)` corresponde a los índices de `'casa'` al inicio de la palabra `'casaca'`.
Los objetos de tipo `SRE_Match` (devueltos por métodos como `match`, `search`, `finditer`, `sub`, ...) admiten, entre otros, los siguientes métodos:

- `group()` devuelve la cadena que hizo *matching*.
- `start()` devuelve la posición inicial del *matching* en la cadena.
- `end()` devuelve la posición final.
- `span()` devuelve las posiciones inicial y final en una tupla.
- `groups()` devuelve los grupos si el patrón definía varios grupos (ver Sección 3.2.5).

Si queremos encontrar todas las ocurrencias de un determinado patrón o regla en una cadena podríamos utilizar[12] `re.findall(pattern,string)` o `re.finditer(pattern,string)`, pero únicamente buscarán las ocurrencias **que no solapen** tal y como muestra el siguiente ejemplo:

```
>>> re.findall(' hola.? ',' hola1 hola2 hola3 ')
[' hola1 ', ' hola3 ']
```

En este ejemplo acabamos de introducir un par de conceptos nuevos:

- El **metacarácter** punto ".". Los metacaracteres, también llamados **clases**, representan conjunto de caracteres. En este caso, el punto "." representa a cualquier carácter (excepto quizás el cambio de línea, como veremos a continuación).
- Un tipo de **cuantificador**, en este caso el símbolo "?" que significa que el elemento al que precede (en este caso el punto ".") puede aparecer 0 ó 1 veces.

Es decir, `' hola.? '` representa cadenas formadas por un espacio, la letra `h`, ..., la letra `a`, un carácter cualquiera (metacarácter ".") opcional (por el "?") y, finalmente, otro espacio.
Para poder elegir si queremos que el metacarácter "." incluya o no el cambio de línea debemos utilizar un modificador o *flag* tal y como se muestra en este ejemplo:

```
>>> re.findall(' hola.? ',' hola\n hola2 hola3 ')
[' hola2 ']
```

como ves en este ejemplo, no ha hecho matching con el primer `' hola\n '`, pero si usamos la opción o *flag* `re.DOTALL` entonces el "." también considera el cambio de línea:

```
>>> re.findall(' hola.? ',' hola\n hola2 hola3 ', re.DOTALL)
[' hola\n ', ' hola3 ']
```

estos *flags*[13] también se pueden utilizar en `re.compile` (es decir, que el sitio para poner el *flag* es al compilarla y no al aplicar el método):

[12] Alternativamente, aplicar los métodos `obj.findall(string)` y `obj.finditer(string)` sobre un objeto creado como `obj = re.compile(pattern)`.

[13] ver `https://docs.python.org/3/howto/regex.html#compilation-flags`.

```
>>> pattern = re.compile(' hola.? ', re.DOTALL)
>>> pattern.findall(' hola\n hola2 hola3 ')
[' hola\n ', ' hola3 ']
```

¿Qué consecuencias tiene que `re.search` busque patrones **no solapados**? Podemos observarlo cuando `re.findall('hola.? ','hola1 hola2 hola3 ')` encuentra `' hola1 '` y `' hola3 '` pero no `' hola2 '`. Esto sucede porque el espacio final de `'hola1 '` ya no es considerado (por formar parte ya de un *matching*) para utilizarse como espacio inicial de `' hola2 '`.
De todas formas, si quisiéramos buscar palabras no es buena idea añadir los espacios en el patrón de búsqueda. Hay palabras que no encontraríamos (por ejemplo, al inicio o final de una línea, separadas por signos de puntuación, etc.). La alternativa es utilizar el metacarácter "`\b`" (debe estar escapado o poniendo el patrón como una cadena de tipo *raw*) que no consume ningún carácter y que únicamente encaja (o hace *matching*) con el límite (inicio o final) de una palabra, donde por "palabra" se entiende cualquier secuencia de caracteres del conjunto definido por el metacarácter "`\w`". El metacarácter "`\w`" engloba letras, dígitos y el símbolo "_". Así, por ejemplo:

```
>>> re.findall(r'\bhola.?\b',' hola,hola2 hola3 holaaa4 ')
['hola', 'hola2', 'hola3']
```

(observa que `'holaaa4'` no encaja con el patrón `r'\bhola.?\b'`).
La diferencia entre los métodos `re.findall` y `re.finditer` es que el primero devuelve una lista y el segundo un *iterador* que va proporcionándonos objetos de tipo `SRE_Match` cuando iteramos sobre él como en el siguiente ejemplo:

```
>>> for i in re.finditer(r'\bhola.?\b',' hola1,hola2 hola3 holaaa4 '):
>>>     print(i)
<_sre.SRE_Match object; span=(1, 6), match='hola1'>
<_sre.SRE_Match object; span=(7, 12), match='hola2'>
<_sre.SRE_Match object; span=(13, 18), match='hola3'>
>>> for i in re.finditer(r'\bhola.?\b',' hola1,hola2 hola3 holaaa4 '):
>>>     print(i.group())
hola1
hola2
hola3
```

Antes de terminar esta sección veamos un par de funciones/métodos más de la biblioteca `re`.
Para entender mejor el próximo ejemplo conviene saber que:

- El metacarácter "`\s`" representa cualquier tipo de espacio (también tabulador, enter,...).
- El cuantificador "`+`" representa repetir "1 o más veces" lo que tenga antes.

Con `re.split(patron,cadena)` el patrón proporcionado sirve para determinar dónde cortar una cadena en trozos y devuelve una lista con dichos trozos.
Es fácil ver que `r'\s+'` representa uno o más espacios consecutivos (ej: un espacio seguido de un enter). Veamos cómo funciona este patrón cuando lo pasamos a `re.split`, es de esperar que corte la cadena por cualquier secuencia de uno o más espacios:

```
>>> re.split(r'\s+', 'uno dos    y\ntres\n')
['uno', 'dos', 'y', 'tres', '']
```

```
>>> re.split(r'\s+', '   uno dos    y\ntres  ')
['', 'uno', 'dos', 'y', 'tres', '']
>>> re.split(r'\s+', 'uno dos    y\ntres')
['uno', 'dos', 'y', 'tres']
```

Observa que el resultado de `re.split` puede incluir la cadena vacía al inicio y/o al final para indicar si el patrón aparece al inicio y/o al final, respectivamente.
Una característica de `re.split` es que si ponemos paréntesis en el patrón (concepto de "grupo" que veremos más adelante), además de devolver los trozos de la cadena separados por los delimitadores, devuelve los propios delimitadores intercalados entre las partes que ha separado:

```
>>> re.split(r'(\s+)', 'uno dos    y\ntres\n')
['uno', ' ', 'dos', '    ', 'y', '\n', 'tres', '\n', '']
>>> re.split(r'(\s+)', '   uno dos    y\ntres  ')
['', '   ', 'uno', ' ', 'dos', '    ', 'y', '\n', 'tres', '  ', '']
```

Terminamos esta sección sobre métodos de `re` con el método `re.sub()`, que sirve para sustituir partes de una cadena:

```
>>> re.sub(r'\s+',' <espacio> ','uno\ndos y     \t tres')
'uno <espacio> dos <espacio> y <espacio> tres'
```

Los argumentos son el patrón, el texto a sustituir y la cadena sobre la que aplicarlo. Cuanto utilizamos `re.compile` el patrón no aparece en la llamada al método:

```
>>> patron = re.compile(r'\s+')
>>> patron.sub(' <espacio> ','uno\ndos y     \t tres')
'uno <espacio> dos <espacio> y <espacio> tres'
```

También es posible utilizar una función en lugar de una cadena para indicar el reemplazamiento. La particularidad es que dicha función recibe un objeto de tipo `SRE_Match` en lugar de una cadena. Esto da más flexibilidad porque es posible que haya "agrupado" varias cosas. En este ejemplo:

```
>>> def reverse(match): return match.group()[::-1]
...
>>> re.sub(r'\b\w+\b', reverse, 'hola... ¿'123? casa!!!')
'aloh... ¿'321? asac!!!'
```

la función `reverse` recibe un objeto de tipo `SRE_Match`, extrae de él la cadena que hizo *matching* y la devuelve tras darle la vuelta (usando `[::-1]`). Al aplicarlo en `sub` con la regla o patrón `r"\b\w+\b"` (que busca una palabra o secuencia de uno o más caracteres de la clase `\w` entre separadores de palabra (metacarácter `\b`)) produce como resultado una cadena en la que todas las palabras son escritas al revés mientras el resto de caracteres permanecen inalterados.
En esta explicación nos hemos saltado muchísimos detalles que puedes consultar en varias fuentes (por ejemplo, en la documentación oficial[14]). Entre las cosas que no hemos visto:

- Cómo indicar que queremos insertar trozos que han encajado (*matching*) con algunos *"grupos"* del patrón (veremos los grupos en la siguiente sección).
- Aunque no lo vamos a utilizar, es interesante saber que es posible limitar el número de sustituciones hasta un tope máximo con el argumento opcional `count`:

[14] `https://docs.python.org/3/howto/regex.html`

```
>>> patron = re.compile(r'\s+')
>>> patron.sub(' <espacio> ',"uno\ndos y     \t tres", count=2)
'uno <espacio> dos <espacio> y     \t tres'
>>> patron.sub(' <espacio> ',"uno\ndos y     \t tres", 2)
'uno <espacio> dos <espacio> y     \t tres'
```

3.2. Sintaxis de las expresiones regulares

En la sección anterior nos hemos centrado en los métodos que proporciona el módulo `re`. Si bien allí hemos adelantado la existencia de algunos metacaracteres y cuantificadores, vamos a desarrollar los detalles de la sintaxis de las expresiones regulares en esta sección. Se recomienda no conformarse con este resumen y consultar el siguiente enlace[15] y los vídeos enlazados en la página de PoliformaT de esta práctica.

3.2.1. Metacaracteres

Ya hemos adelantado que muchos caracteres como las letras (`abc...z`, `ABC...Z`) o los dígitos (`0123456789`) solamente hacen *matching* con ellos mismos. Sin embargo, existen metacaracteres que representan grupos de caracteres, algunos de ellos son:

- "`.`" representa "cualquier carácter" (que se incluya o no el cambio de línea depende del *flag* `re.DOTALL` tal y como se vió en la sección anterior). Si quieres hacer *matching* únicamente con el símbolo "`.`" hay que escaparlo poniendo "`\.`".
- "`\s`" representa cualquier tipo de espacio (tabulador, enter, ...).
- "`\S`" representa el conjunto complementario[16] a `\s`.
- "`\w`" representa letras, dígitos y el símbolo "`_`".
- "`\W`" corresponde al complementario de "`\w`".
- "`\d`" el conjunto de los dígitos.
- "`\D`" es el complementario de "`\d`".

Los siguientes metacaracteres no corresponden a ningún símbolo sino a determinadas posiciones (*boundary matches*):

- "`\A`" hace *matching* con el inicio de la entrada.
- "`\Z`" hace *matching* con el final de la entrada.
- "`^`" hace *matching* con el inicio de la línea.
- "`$`" hace *matching* con el final de la línea.

No hay diferencia entre "entrada" y "línea" (la diferencia entre `\A` y `\Z`, por una parte, y `^` y `$`, por otra) salvo que apliquemos el *flag* para múltiples líneas (`re.M` o `re.MULTILINE`) tal y como se observa en los siguientes ejemplos:

[15] `https://docs.python.org/3/library/re.html#regular-expression-syntax`

[16] En general, un metacarácter en mayúscula es el complementario de su correspondiente en minúscula.

```
>>> p = re.compile(r'^A\w+') # empieza por A al inicio de entrada
>>> p.findall('Atila huno\nAsturias región\nAntártida continente')
['Atila']
>>> p = re.compile(r'^A\w+', re.M) # ahora al inicio de línea
>>> p.findall('Atila huno\nAsturias región\nAntártida continente')
['Atila', 'Asturias', 'Antártida']
```

mientras que con `\A` en lugar de `^` no hay diferencia al aplicar el *flag* `re.M`:

```
>>> p = re.compile(r'\AA\w+')
>>> p.findall('Atila huno\nAsturias región\nAntártida continente')
['Atila']
>>> p = re.compile(r'\AA\w+', re.M)
>>> p.findall('Atila huno\nAsturias región\nAntártida continente')
['Atila']
```

Existen otros metacaracteres que también son *boundary* pero que corresponden a encontrarse entre símbolos determinados:

- "`\b`" ya lo hemos visto en la sección anterior, representa una posición entre un carácter de "`\w`" y otro de "`\W`".
- "`\B`" representa cualquier posición entre un carácter "`\w`" al lado de otro de "`\w`" o uno de "`\W`" al lado de otro de "`\W`" (es decir, donde no se cumpla "`\b`"). Ejemplo (observa que `span=(19, 22)` corresponde a la posición de la palabra "`aunque`"):

  ```
  >>> re.search(r'\Bque','el dia que dije aunque')
  <_sre.SRE_Match object; span=(19, 22), match='que'>
  ```

3.2.2. Conjuntos de caracteres

Acabamos de ver metacaracteres predefinidos que hacen *matching* con conjuntos de caracteres. Veamos que también es posible definir nosotros mismos esos grupos de caracteres utilizando los corchetes y poniendo dentro de ellos los caracteres, por ejemplo:

```
>>> # estamos considerando el caso SIN ACENTOS
>>> vocales_seguidas = re.compile(r'[aeiouAEIOU]+')
>>> vocales_seguidas.findall('murcielago') # sin el acento
['u', 'ie', 'a', 'o']
```

También podemos incluir rangos separados por un guión "`-`". Por ejemplo, los dígitos se pueden representar con el metacarácter "`\d`", con el conjunto "`[0123456789]`" o con el rango "`[0-9]`". Cuando el primer carácter dentro de los `[]` es un "`^`", se representa el conjunto complementario. Así, por ejemplo, para representar cualquier cosa que no sean dígitos, además del metacarácter "`\D`" podemos utilizar "`[^0-9]`".

3.2.3. Disyunción

El símbolo "`|`" en una expresión tipo `A|B`, siendo `A` y `B` expresiones regulares, crea una expresión regular que acepta lo que acepte `A` o lo que acepte `B`. Por ejemplo:

```
>>> re.findall(r'[abc]+|[012]+','aaba00011xyz111acabaac')
['aaba', '00011', '111', 'acabaac']
```

En este otro ejemplo se intenta capturar algunos adverbios de modo:

```
>>> txt = 'aparcó bien el coche, salió silenciosamente'
>>> re.findall(r'bien|mal|despacio|lento|\w+mente', txt)
['bien', 'silenciosamente']
```

3.2.4. Cuantificadores

Hemos visto que "`?`" indica que el elemento anterior de la regla es opcional, y que "`+`" indica que se puede repetir 1 o más veces. Veamos estos y otros cuantificadores:

- "`?`" puede repetirse 0 ó 1 veces. Se realiza la captura en modo voraz o *greedy* (consumir lo más posible) tal y como muestra el siguiente ejemplo:

  ```
  >>> re.findall(r'abb?s', 'absoluto abbs')
  ['abs', 'abbs']
  >>> re.findall(r'abb?', 'absoluto abbs')
  ['ab', 'abb']
  ```

  En el primer caso *no le queda otra*, mientras que en el segundo podría haber hecho matching en "`abbs`" con "`ab`" o con "`abb`" y ha elegido la opción más larga.

  En cualquier caso, ten en cuenta que se avanza de izquierda a derecha sin solapamientos entre capturas distintas, por lo que en el siguiente ejemplo no se realiza la captura de `bac` de longitud 3 ya que la primera captura toma la `b` y ésta ya no se puede usar después:

  ```
  >>> re.findall(r'b?a.', 'abac')
  ['ab', 'ac']
  ```

- "`??`" puede repetirse 0 ó 1 veces como antes, pero si puede realiza la captura más corta posible:

  ```
  >>> re.findall(r'abb??s', 'absoluto abbs')
  ['abs', 'abbs']
  >>> re.findall(r'abb??', 'absoluto abbs')
  ['ab', 'ab']
  ```

  En el primer caso *no le queda otra* (como en el caso "`?`"), mientras que en el segundo elige la opción más corta de hacer matching con "`ab`" en la palabra "`abbs`".

- "`+`" se puede repetir 1 o más veces de modo voraz o *greedy* (consumir lo más posible), por ejemplo, el patrón `r'<.+>'` busca una cadena que empiece con "`<`" y termine con "`>`" (en medio cualquier cosa gracias a "`.+`"), veamos un ejemplo:

  ```
  >>> re.findall(r'<.+>', '<tag>hola</tag> <li>hola</li>')
  ['<tag>hola</tag> <li>hola</li>']
  ```

  Como se puede observar, `findall` realiza una única captura de todo el texto llegando hasta el último "`>`" del texto, ya que "`.+`" permite hacer matching con los "`>`" previos. Si quisiéramos capturar únicamente el "`<tag>`", podríamos crear un patrón que impida que aparezca el caracter de cierre en la parte afectada por el "`+`", como en el siguiente ejemplo:

```
>>> re.findall(r'<[^>]+?>', '<tag>hola</tag> <li>hola</li>')
['<tag>', '</tag>', '<li>', '</li>']
```

Que se tome la cadena más larga está sujeto a que se haga *matching* con la expresión regular. Por ejemplo, en este ejemplo la expresión `A+` hace matching únicamente con 2 de las 3 `A` para que el `.` pueda encajar con la última `A`:

```
>>> re.findall(r'(A+).', 'AAA')
['AA']
```

Este ejemplo muestra que si agrupas con paréntesis una parte de la expresión regular, el método `findall` usa toda la expresión regular para hacer el *matching* pero solamente devuelve como resultado la parte asociada a lo que está entre paréntesis (esto se ve en la siguiente sección sobre grupos). En el siguiente ejemplo la parte `(A+)` solamente captura una letra porque después aparecen dos puntos que hay que encajar:

```
>>> re.findall(r'(A+)..', 'AAA')
['A']
```

- "`+?`" se puede repetir 1 o más veces de modo *lazy* (consumir lo menos posible), por ejemplo, el patrón `r'<.+? >'` busca la cadena más corta que empiece por "`<`", tenga uno o más caracteres en medio y termine por "`>`", por ejemplo:

```
>>> re.findall(r'<.+?>', '<tag>hola</tag> <li>hola</li>')
['<tag>', '</tag>', '<li>', '</li>']
```

- "`*`" permite 0, 1 o más repeticiones en modo voraz o *greedy*, es similar a `+` pero permite también 0 repeticiones y consume la cadena más larga posible que haga matching.

- "`*?`" es la versión *lazy* de `*`. Es decir, permite 0, 1 o más repeticiones pero realiza la captura de menor longitud.

- "`{n}`" se debe repetir exactamente `n` veces. Por ejemplo, si queremos encontrar expresiones del estilo `</etiqueta>` con etiquetas de longitud 3 podríamos hacer:

```
>>> re.findall(r'</[^>]{3}>','<tag>hola</tag> <li>hola</li>')
['</tag>']
```

- `{m,n}` se debe repetir entre `m` y `n` veces (es "voraz": esto significa que, en caso de duda, acapara todo lo máximo que se pueda dentro de ese rango). Podemos omitir alguno de los dos valores:

  - `{,n}` se repite entre 0 y `n` veces.
  - `{m,}` se repite `m` o más veces.

- Podemos añadir un `?` al final de `{m,n}` como en el ejemplo:

```
>>> re.findall(r'a\d{3,5}', 'a12 a1234 a12345678')
['a1234', 'a12345']
>>> re.findall(r'a\d{3,5}?', 'a12 a1234 a12345678')
['a123', 'a123']
```

Donde la versión `\d{3,5}?` captura entre 3 y 5 dígitos pero, si puede, elige la opción de menor longitud.

Sin embargo, el concepto de *laziness* solamente funciona en el lado derecho porque la expresión regular se procesa de izquierda a derecha y para en cuanto encuentra un *matching*:

```
>>> re.findall(r'\d{3,5}a', '12a 1234a 12345678a')
['1234a', '45678a']
>>> re.findall(r'\d{3,5}?a', '12a 1234a 12345678a')
['1234a', '45678a'] # OPSSSSS 45678a ha pillado 5 dígitos
```

Recordatorio: *Poner un "`?`" después de "`+`", de "`*`" o de "`m,n`" hace que se comporten de manera* lazy*.*

Advertencia: *La diferencia* greedy/lazy *es bastante complicada/delicada porque hay que tener en cuenta que no se solapan* matchings*, que se procede de izquierda a derecha, etc.*

3.2.5. Grupos

Existe un repertorio enorme de opciones y variantes, en esta práctica introductoria nos limitaremos a comentar que una parte de una expresión regular encerrada entre paréntesis delimita un *grupo* que posteriormente puede recuperarse en diversos métodos o incluso en las cadenas de sustitución del método `re.sub`. Antes hemos visto que el método `re.split` también devolvía las cadenas de separación si el patrón tenía un grupo. En el siguiente ejemplo vemos que el método `re.findall` devuelve tuplas en lugar de cadenas si hay más de un grupo:

```
>>> txt = 'correo de perico@gmail.com dirigido a maria@hooya.com'
>>> re.findall(r'([^@\s]+)@([^@\s]+\.[^@\s]+)', txt)
[('perico', 'gmail.com'), ('maria', 'hooya.com')]
```

El patrón (algo burdo) para reconocer una dirección de correo está compuesto por:

- `([^@\s]+)` reconoce uno o más caracteres que no sean "`@`" o algún tipo de espacio.
- `@` reconoce únicamente ese símbolo (es un caracter normal, no es un metacarácter).
- `([^@\s]+\.[^@\s]+)` reconoce una secuencia que no contenga "`@`" o espacios, un punto (observa que está escapado para distinguirlo el metacarácter) y, finalmente, otra secuencia de 1 o más símbolos que tampoco sean "`@`" o espacios.

En este otro ejemplo vemos que a los objetos de tipo `SRE_Match`[17] les podemos pedir tanto la cadena completa como los distintos grupos (de haberlos):

```
>>> m = re.search(r'([^@\s]+)@([^@\s]+\.[^@\s]+)', txt)
>>> m
<_sre.SRE_Match object; span=(16, 32), match='perico@gmail.com'>
>>> m.groups()
('perico', 'gmail.com')
>>> m.group()
'perico@gmail.com'
```

[17] Ver `https://docs.python.org/3/library/re.html#match-objects`

Podemos dar nombres a los grupos con la sintaxis `(?P<nombre> ...)` como en este ejemplo:

```
>>> txt = 'correo de perico@gmail.com dirigido a maria@hooya.com'
>>> expr_reg = r'(?P<nombre>[^@\s]+)@(?P<dominio>[^@\s]+\.[^@\s]+)'
>>> m = re.search(expr_reg, txt)
>>> m.group('nombre')
'perico'
>>> m.group('dominio')
'gmail.com'
```

o en este otro:

```
>>> txt = 'correo de perico@gmail.com dirigido a maria@hooya.com'
>>> expr_reg = r'(?P<nombre>[^@\s]+)@(?P<dominio>[^@\s]+\.[^@\s]+)'
>>> for i in re.finditer(expr_reg, txt):
>>>     print(i.group('nombre'))
perico
maria
```

En la documentación de Python[18] verás que existen otros usos para los paréntesis. En esta práctica únicamente utilizaremos:

- `(`*algo*`)` sirve para agrupar *algo* y **capturarlo como grupo**.
- `(?P<nombre>`*algo*`)` sirve para agrupar *algo* y **capturarlo como grupo** y darle un nombre con el que recuperarlo después.
- `(?:`*algo*`)` sirve para agrupar *algo* pero **sin capturarlo como grupo**. ¿Qué interés puede tener eso? Muy sencillo: al igual que sucede con el uso de paréntesis en algunas expresiones aritméticas, puede ser necesario agrupar cosas para aplicarles disyunción u otras operaciones como sucede en el siguiente ejemplo:

  ```
  >>> re.search(r'uno|dos+','dosssssssss')
  <_sre.SRE_Match object; span=(0, 12), match='dosssssssss'>
  >>> re.search(r'uno|dos+','dosdosdos')
  <re.Match object; span=(0, 3), match='dos'>
  >>> re.search(r'(?:uno|dos)+','dosssssssss')
  <re.Match object; span=(0, 3), match='dos'>
  >>> re.search(r'(?:uno|dos)+','unounodosdosunodos')
  <_sre.SRE_Match object; span=(0, 18), match='unounodosdosunodos'>
  ```

  Observa la diferencia entre `(`*algo*`)` y `(?:`*algo*`)`:

  ```
  >>> re.findall(r'(uno|dos)+(tres|cuatro)+','unodosunotres')
  [('uno', 'tres')]
  >>> re.findall(r'(?:uno|dos)+(?:tres|cuatro)+','unodosunotres')
  ['unodosunotres']
  ```

[18] https://docs.python.org/3/library/re.html#regular-expression-syntax