Expresiones regulares

Una expresión regular es un patrón que describe una colección de cadenas de caracteres.

Por ejemplo, el patrón `a.u` representa todas las cadenas de caracteres que empiezan por la letra `a`, luego un carácter cualquiera (representado por el símbolo `.`) y luego la letra `u`. Más concretamente, el patrón `a.u` representa las cadenas `abu`, `agu` y `a-u`, pero no las cadenas `Abu` o `ave`.

Otro ejemplo: el patrón `"ab*"` representa las cadenas de caracteres que empiecen por la letra `a` seguida por cero o más caracteres `b`.

Y otro: el patrón `"qui(jote|jano|mera)"` representa las cadenas de caracteres que empiecen por `qui`, seguidas `jote`, por `jano` o por `mera`. Concretamente, las cadenas `"quijote"`, por `"quijano"` o por `quimera`.

La función `match()`

La función `match()` averigua si una cadena de caracteres contiene un patrón, *en su comienzo*.

Por ejemplo, como el símbolo `"."` representa exactamente un carácter cualquiera, el patrón `"a.u"` encaja al comienzo de las cadenas `"abuelo"`, `"avuelo"`, `"avutarda"` y `"aguja"`, pero no de las cadenas `"Abuelo"` y `"aaaaa"`.

In [1]:

```
import re

def comprobar_encaje(patron_str, cadena):
    patron = re.compile(patron_str)
    if re.match(patron, cadena):
        print("El patrón '" + patron_str + "' encaja en la cadena '" + cadena + "'")
    else:
        print("El patrón '" + patron_str + "' NO encaja en la cadena '" + cadena + "'")

comprobar_encaje("a.u", "abuelo")
comprobar_encaje("a.u", "Abuelo")
```

```
El patrón 'a.u' encaja en la cadena 'abuelo'
El patrón 'a.u' NO encaja en la cadena 'Abuelo'
```

In [2]:

```
def comprobar_encaje_varias_cadenas(patron_str, varias_cadenas):
    patron = re.compile(patron_str)
    for cadena in varias_cadenas:
        if re.match(patron, cadena):
            print("Ok: " + cadena)
        else:
            print("No: " + cadena)

cadenas = ["abuelo", "Abuelo", "avuelo", "avutarda", "aguja", "aaaaa"]
comprobar_encaje_varias_cadenas("a.u", cadenas)
```

```
Ok: abuelo
No: Abuelo
Ok: avuelo
Ok: avutarda
Ok: aguja
No: aaaaa
```

Otro comodín: `"*"`

Otro comodín es el símbolo `"*"`, que representa exactamente la repetición "ninguna o más veces", del carácter precedente. Así por ejemplo, el patrón `"ab*o"` representa el comienzo de las cadenas `"abono"` y `"abbbbonar"` o `"ao"`, pero no de las cadenas `"abuelo"` o `"aaaaa"`.

In [3]:

```
cadenas = ["abuelo", "Abuelo", "abono", "abbbbonar",
           "abuela", "abuelitos", "aaaaa", "ao"]

comprobar_encaje_varias_cadenas("ab*o", cadenas)
```

```
No: abuelo
No: Abuelo
Ok: abono
Ok: abbbbonar
No: abuela
No: abuelitos
No: aaaaa
Ok: ao
```

Algunos patrones básicos

El símbolo `"."` en el patrón representa un carácter cualquiera, y `"*"`, ninguna o más repeticiones del símbolo precedente. En los patrones se usan otros muchos símbolos. Veamos algunos ejemplos para empezar:

Algunos códigos especiales nos permiten identificar dígitos, caracteres no dígitos...

Patrón	Significado
"a*"	El carácter a, ninguna o más veces
"a+"	El carácter a, una o más veces
"[a-z]"	Una letra de la "a" a la "z"

Patrón	Significado
"[1-9]+"	Un dígito entre uno y nueve, una o más veces
"mi(.l..)o"	Empieza por "mi", luego uno o dos caracteres y luego una "o": mito, mico, mirlo, miedo
"^The.*Spain$"	Empieza por "The" y termina con "Spain"

In [4]:

```
cadenas = ["abuelo", "Abuelo", "abono", "abbbbonar",
           "abuela", "abuelitos", "aaaaa", "ao"]

comprobar_encaje_varias_cadenas("a.*b+", cadenas)

print(".....................................")

comprobar_encaje_varias_cadenas("a(b|v|g).*", cadenas)

print(".....................................")

comprobar_encaje_varias_cadenas("^En.*Mancha$",
                                ["En un lugar de la Mancha", "En la Mancha..."])
```

```
Ok: abuelo
No: Abuelo
Ok: abono
Ok: abbbbonar
Ok: abuela
Ok: abuelitos
No: aaaaa
No: ao
.....................................
Ok: abuelo
No: Abuelo
Ok: abono
Ok: abbbbonar
Ok: abuela
Ok: abuelitos
No: aaaaa
No: ao
.....................................
Ok: En un lugar de la Mancha
No: En la Mancha...
```

La función `search()`:

La función `search()` recorre una cadena y busca en ella algún fragmento que encaje con el patrón dado, y da el fragmento posible en caso de encontrarlo:

In [5]:

```
patron = re.compile("c...")

encaje = re.search(patron, "En un lugar...")
print(encaje)

encaje = re.search(patron, "... de la Mancha, de cuyo nombre no quiero acordarme...")
print(encaje)
print(encaje.start())
print(encaje.end())
```

```
None
<re.Match object; span=(13, 17), match='cha,'>
13
17
```

La función `findall()`:

Podríamos desear ver todos los encajes, y no sólo el primero: nuestra función es ahora `findall()`:

In [6]:

```
cadena = """En un lugar de la Mancha de cuyo nombre\
no quiero acordarme, había un hidalgo, de los de\
lanza en astillero, rocín flaco y galgo corredor..."""

print(re.findall("c...", cadena))

print(re.findall(". l..", cadena))
```

```
['cha ', 'cuyo', 'cord', 'cín ', 'co y', 'corr']
['n lug', 'e la ', 'e los']
```

Algunos patrones más

En los ejemplos anteriores hemos visto los patrones siguientes:

- El patron "ab*", que representa las cadenas de caracteres que empiezan por la letra `a` seguida de cero o más caracteres "b"
- El patrón ". l..", que representa las cadenas de cinco caracteres: uno cualquiera, un espacio, la letra "l" y dos caracteres cualesquiera más.

Pero hay otros patrones posibles, que se pueden definir con las siguientes convenciones. Vamos a verlos mediante ejemplos:

In [7]:

```
patrones = ['.ab*',       # un carácter, el carácter "a" seguido por cero o más caracteres
            '.ab+',       # un carácter, el carácter "a" seguido por uno o más caracteres "
            '.ab?',       # un carácter, el carácter "a" seguido por cero o un carácter "b"
            '.ab{2}',     # un carácter, el carácter "a" seguido por dos caracteres "b".
            '.ab{2,4}',   # un carácter, el carácter "a" seguido por 2, 3 o 4 caracteres "b
            '.[ab].',     # "[ab]" es el carácter "a" o el carácter "b".
            '.[ab]+',     # un carácter seguido de uno o más caracteres "a" o "b"
           ]

frase = "0a 1b 2ab 3aba 4abc 5aaaaaaa 6abbab 7abbbbb 6abababababab"

for p in patrones:
    print("Búsqueda con el patrón '" + p + "'")
    print(re.findall(p,frase))
```

```
Búsqueda con el patrón '.ab*'
['0a', '2ab', '3ab', '4ab', '5a', 'aa', 'aa', 'aa', '6abb', '7abbbbb', '6a
b', 'bab', 'bab']
Búsqueda con el patrón '.ab+'
['2ab', '3ab', '4ab', '6abb', '7abbbbb', '6ab', 'bab', 'bab']
Búsqueda con el patrón '.ab?'
['0a', '2ab', '3ab', '4ab', '5a', 'aa', 'aa', 'aa', '6ab', 'bab', '7ab', '6a
b', 'bab', 'bab']
Búsqueda con el patrón '.ab{2}'
['6abb', '7abb']
Búsqueda con el patrón '.ab{2,4}'
['6abb', '7abbbb']
Búsqueda con el patrón '.[ab].'
['0a ', '1b ', '2ab', '3ab', '4ab', '5aa', 'aaa', 'aa ', '6ab', 'bab', '7a
b', 'bbb', '6ab', 'aba', 'bab', 'aba']
Búsqueda con el patrón '.[ab]+'
['0a', '1b', '2ab', '3aba', '4ab', '5aaaaaaa', '6abbab', '7abbbbb', '6ababab
ababab']
```

Veamos algunos patrones más:

In [8]:

```
patrones = ["[^!.? ]+",   # "Un carácter que no es "!", ni ".", ni "?" ni " ".
            "[a-z]",      # rango de caracteres
            "[a-zA-Z]",   # una minúscula o una mayúscula
            "[A-Z][a-z]", # una minúscula seguida de una mayúscula
           ]

frase = "¡Qué lindos ojos! ¿Puedes decirme tu nombre?"

for p in patrones:
    print("Búsqueda con el patrón " + p)
    print(re.findall(p,frase))
```

```
Búsqueda con el patrón [^!.? ]+
['¡Qué', 'lindos', 'ojos', '¿Puedes', 'decirme', 'tu', 'nombre']
Búsqueda con el patrón [a-z]
['u', 'l', 'i', 'n', 'd', 'o', 's', 'o', 'j', 'o', 's', 'u', 'e', 'd', 'e',
's', 'd', 'e', 'c', 'i', 'r', 'm', 'e', 't', 'u', 'n', 'o', 'm', 'b', 'r',
'e']
Búsqueda con el patrón [a-zA-Z]
['Q', 'u', 'l', 'i', 'n', 'd', 'o', 's', 'o', 'j', 'o', 's', 'P', 'u', 'e',
'd', 'e', 's', 'd', 'e', 'c', 'i', 'r', 'm', 'e', 't', 'u', 'n', 'o', 'm',
'b', 'r', 'e']
Búsqueda con el patrón [A-Z][a-z]
['Qu', 'Pu']
```

La función `split()`:

Separa una cadena según un patrón:

In [9]:

```
cadena = """En un lugar de la Mancha de cuyo nombre\
no quiero acordarme, había un hidalgo, de los de\
lanza en astillero, rocín flaco y galgo corredor..."""

separada = re.split("c...", cadena)
print(separada)

print("....................................")

separadores_de_frase = "[.;:]"
frases = re.split(separadores_de_frase, "Estoy de acuerdo. Pero no del todo; otro día lo di
print(frases)
```

```
['En un lugar de la Man', 'de ', ' nombreno quiero a', 'arme, había un hidal
go, de los delanza en astillero, ro', 'fla', ' galgo ', 'edor...']
....................................
['Estoy de acuerdo', ' Pero no del todo', ' otro día lo discutimos', ' ahora
no puedo', '']
```

La función `sub()`:

Sustituye las apariciones de un patrón por otra cosa, en una cadena de carateres:

In [10]:

```
cadena = """En un lugar de la Mancha de cuyo nombre\
no quiero acordarme, había un hidalgo, de los de\
lanza en astillero, rocín flaco y galgo corredor..."""

separada = re.sub("c...", "----", cadena)
print(separada)
```

```
En un lugar de la Man----de ---- nombreno quiero a----arme, había un hidalg
o, de los delanza en astillero, ro----fla---- galgo ----edor...
```

La función `group()`:

Dentro de un fragmento que encaja con un patrón, podemos distinguir partes. La función group permite definir estas partes y localizarlas por separado:

In [11]:

```
cadena = """En un lugar de la Mancha de cuyo nombre\
no quiero acordarme, había un hidalgo, de los de\
lanza en astillero, rocín flaco y galgo corredor..."""

patron = re.compile("c....")
encaje = re.search(patron, cadena)
print(encaje.group())

patron = re.compile("c((..)(..))")
encaje = re.search(patron, cadena)
print(encaje.group(0))
print(encaje.group(1))
print(encaje.group(2))
print(encaje.group(3))

print("....................................")

# Buscamos un número entero en una cadena:

patr_ent = re.compile("[^0-9]*([0-9]+)[^0-9].*")
cadena = "el número de orden es 17, el nombre es Calixto y su novia es Melibea."
encaje = re.search(patr_ent, cadena)
print(encaje.group(1))

print("....................................")

# Buscamos un número entero y un nombre propio una cadena:

patr_ent_np = re.compile("[^0-9]*([0-9]+)[^0-9][^A-Z]*([A-Z][a-z]*).*")
cadena = "el número de orden es 17, el nombre es Calixto y su novia es Melibea."
encaje = re.search(patr_ent_np, cadena)
print(encaje.group(1))
print(encaje.group(2))
```

```
cha d
cha d
ha d
ha
 d
....................................
17
....................................
17
Calixto
```

Secuencias de escape

Algunos códigos especiales nos permiten identificar dígitos, caracteres no dígitos...

Código	Significado
"\d"	un dígito
"\D"	un carácter no dígito
"\s"	espacio en blanco o tabulador o nueva línea, etc.)
"\S"	no espacio en blanco o...

Código	Significado
"\w"	carácter alfaumérico
"\W"	no carácter alfaumérico

In [12]:

```
patrones = ["\d+", # Secuencia de uno o más dígitos
            "\D+", # Secuencia de no dígitos
            "\s+", # Secuencia de espacios en blanco
            "\S+", # Secuencia de no espacios en blanco
            "\w+", # Secuencia de caracteres alfanuméricos
            "\W+", # Secuencia de no caracteres alfanuméricos
           ]

frase = "¡El número del anticristo es 666, el número de la bestia!"

for p in patrones:
    print("Búsqueda con el patrón " + p)
    print(re.findall(p,frase))
```

```
Búsqueda con el patrón \d+
['666']
Búsqueda con el patrón \D+
['¡El número del anticristo es ', ', el número de la bestia!']
Búsqueda con el patrón \s+
[' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']
Búsqueda con el patrón \S+
['¡El', 'número', 'del', 'anticristo', 'es', '666,', 'el', 'número', 'de',
'la', 'bestia!']
Búsqueda con el patrón \w+
['El', 'número', 'del', 'anticristo', 'es', '666', 'el', 'número', 'de', 'l
a', 'bestia']
Búsqueda con el patrón \W+
['¡', ' ', ' ', ' ', ' ', ' ', ', ', ' ', ' ', ' ', ' ', '!']
```

Ejemplos adicionales

In [13]:

```
# Patrón para identificar una dirección de email:

patron_email = r"([a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+)"
frase = "Mi email es cpareja@ucm.es, no c.pareja@sip.ucm.es ni c_pareja@SIP.ucm.es.es.es"
print(re.findall(patron_email,frase))
```

```
['cpareja@ucm.es', 'c.pareja@sip.ucm.es', 'c_pareja@SIP.ucm.es.es.es']
```

In [14]:

```
# Patrón para identificar una fecha:

fecha_re = re.compile('\d{2}/\d{2}/\d{4}')

linea = '[26/11/1962 00:01:35] <font color="#00ff00"¡Sorpresa!</font>>'
encaje = fecha_re.search(linea)
print(encaje)
print(encaje.group(0))

linea_sin_fecha = "En tiempos de Ahrun al Rashid..."
encaje = fecha_re.search(linea_sin_fecha)
print(encaje)
```

```
<re.Match object; span=(1, 11), match='26/11/1962'>
26/11/1962
None
```

In [15]:

```
# Patrón para identificar una fecha y una hora:

linea = '[26/11/1962 00:01:35] <font color="#00ff00"¡Sorpresa!</font>>'

# Ahora definimos dos grupos, con los paréntesis:
fecha_con_hora = re.compile('(\d{2}/\d{2}/\d{4}) (\d{2}:\d{2}:\d{2})')

encaje = fecha_con_hora.search(linea)
print(encaje)
print(encaje.group(0))
print(encaje.group(1))
print(encaje.group(2))
```

```
<re.Match object; span=(1, 20), match='26/11/1962 00:01:35'>
26/11/1962 00:01:35
26/11/1962
00:01:35
```

In [16]:

```
# Para reconvertir las cadenas de caracteres de  fechas en fechas:

from datetime import datetime
fecha = datetime.strptime(encaje.group(0), "%d/%m/%Y %H:%M:%S")
fecha

# https://docs.python.org/3.6/library/datetime.html#strftime-strptime-behavior
```

Out[16]:

```
datetime.datetime(1962, 11, 26, 0, 1, 35)
```

Brevísimo comentario final

Las expresiones regulares se pueden manejar en Python mediante la librería "`re`", que se ha de importar para poder usar las funciones para manejar los patrones y las funciones "`match()`", "`search()`", "`findall()`", etc., con las que manejaremos las expresiones regulares.

Pero el tema en sí es bastante más amplio en el mundo de la Computación, pues involucra conceptos de autómatas finitos deterministas y no deterministas, compiladores, etc., que no consideramos aquí obviamente.

Entre las muchas referencias que pueden consultarse, he aquí dos que me gustan a mí:

- https://www.w3schools.com/python/python_regex.asp
- https://docs.python.org/3/howto/regex.html

Finalmente, os dejo el siguiente link, que permite probar y depurar expresiones regulares:

```
https://regex101.com/
```

Cuenta con ayuda en tiempo real y guía de uso de las expresiones disponibles.

In [ ]: