Programación en Python II

Escuela Técnica Superior de Ingeniería de Telecomunicación
Universidad Rey Juan Carlos

gsyc-profes (arroba) gsyc.urjc.es

Enero de 2018

format

En python 2.6 y superiores las cadenas cuentan con el método `format()`

- Dentro de una cadena, podemos indicar, entre llaves, qué campos se mostrarán y con qué formato.
  Format tiene un microlenguaje para esto
- Los argumentos de `format()` serán los campos

Ejemplo: Indicar qué campo mostrar, a partir del ordinal

```
#!/usr/bin/python -tt
# -*- coding: utf-8 -*-

name="Juan"
surname="García"
print "Se llama {0} y se apellida {1}".format(name,surname)
print "Se llama {} y se apellida {}".format(name,surname)

persona=["Juan","García"]
print "Se llama {0[0]} y se apellida {0[1]}".format(persona)

persona={"name":"Juan", "surname":"García"}
print "Se llama {0[name]} y se apellida {0[surname]}".format(persona)
```

Resultado:

```
Se llama Juan y se apellida García
Se llama Juan y se apellida García
Se llama Juan y se apellida García
Se llama Juan y se apellida García
```

Después de indicar qué campo mostrar, separado por el carácter dos puntos, podemos especificar cuántos caracteres debe ocupar la salida, y si estará alineada a la derecha (signo de mayor), a la izquierda (signo de menor o ningún signo) o al centro (acento circunflejo)
Ejemplo: mostrar una palabra, ocupando siempre 12 caracteres

```
#!/usr/bin/python -tt
# -*- coding: utf-8 -*-

print("{0:>12}{1:>12}".format("sota","caballo"))
print("{0:<12}{1:<12}".format("sota","caballo"))
print("{0:12}{1:12}".format("sota","caballo"))
print("{0:^12}{1:^12}".format("sota","caballo"))
```

Resultado:

```
        sota     caballo
sota         caballo
sota         caballo
    sota       caballo
```

- Si solo hay un campo, podemos omitir el 0 a la izquierda del carácter dos puntos
- Con el carácter `d` podemos indicar que el campo contiene un número entero. En este caso, la alineación por omisión es a la derecha
- Con el carácter `f` indicamos que el campo es un número real Podemos especificar cuántos decimales representar. Por ejemplo 4: `.4f`

```
print("{:<6d} metros".format(592))
print("{:>6d} metros".format(592))
print("{0:6d} metros".format(592))
print("Pi vale {:.4f}".format(3.14159265358979))
```

Resultado:

```
592    metros
   592 metros
   592 metros
Pi vale 3.1416
```

Naturalmente, la cadena no tiene por qué ser una constante, puede ser una variable

```
#!/usr/bin/python -tt
# -*- coding: utf-8 -*-

x=12.3
y=0.345
z=34000
template="{:8},{:8},{:8}"
msg=template.format(x,y,z)
print(msg) #    12.3,   0.345,   34000
```

Format también permite usar parámetros con nombre

```
#!/usr/bin/python -tt
# -*- coding: utf-8 -*-

d={"equis": 12.3, "y_griega":0.345, "zeta":34000}

template="{x:8},{y:8},{z:8}"
msg=template.format(z=d["zeta"], y=d["y_griega"], x=d["equis"])
print(msg)
#     12.3,   0.345,   34000
```

optparse

`optparse` es una librería de python para procesar las opciones y argumentos con los que se llama a un script

```
  orden  opciones           argumentos
-----------------------------------------------
   cp     -r  -v      directorio1    directorio2
```

En un buen interfaz

- Las opciones deben ser opcionales. (El programa debe hacer algo útil sin ninguna opción)
- Las opciones proporcionan flexibilidad, pero demasiadas introducen complejidad
- Los parámetros fundamentales e imprescindibles deben ser argumentos

- Creamos una instancia de la clase `OptionParser`, pasando como argumento la cadena `usage` (que se mostrará al usuario cuando use mal el script, o cuando lo llame con `-h` o `--help`

```
usage = "Uso: %prog [opciones] origen destino"
parser = OptionParser(usage)
```

- Para añadir opciones invocamos al método `add_option`

```
parser.add_option("-v", "--verbose",
                  action="store_true", dest="verbose",
                  help="Informe detallado")
```

- Invocamos a `parse_args()`, que devuelve las opciones ya procesadas y los argumentos

```
(opciones, argumentos) = parser.parse_args()
```

```
#!/usr/bin/python -tt
# -*- coding: utf-8 -*-
import sys

from optparse import OptionParser
def main():
    usage = "%prog [opciones] origen destino"
    parser = OptionParser(usage)
    parser.add_option("-e", "--energy",
                      action="store", dest="energy",
                      help="Tipo de energia a usar en la copia ",
                      default='eolic')
    parser.add_option("-v", "--verbose",
                      action="store_true", dest="verbose",
                      help="Informe detallado")
    parser.add_option("-q", "--quiet",
                      action="store_false", dest="verbose",
                      help="Informe silencioso")
    (opciones, argumentos) = parser.parse_args()
    if len(argumentos) != 2:
        parser.error("Número incorrecto de argumentos")
    print "Tipo de energia:"+opciones.energy
    print "Origen:",argumentos[0]
    print "Destino:",argumentos[1]
    if opciones.verbose:
        print "mucho blablabla "

if __name__ == "__main__":
    main()
```

add_option

```
parser.add_option("-e", "--energy",
                  action="store", dest="energy",
                  help="Tipo de energia a usar en la copia ", default='eolic')
```

- Cada opción puede invocarse con una única letra (p.e. `-v`) o con una palabra (p.e. `--verbose`)
- Con el atributo *help* se construye el mensaje que se mostrará al usuario cuando invoque el programa con `-h` o `--help` [1]
- La opción puede
  - Limitarse a activar o desactivar un flag.
    `action="store_true"` `action="store_false"`
  - Indicar un valor
    `action="store"`

  En ambos casos, la información se almacena en un atributo que se llama como indique el parámetro `dest`

[1] En el ubuntu actual salta un error si usamos caracteres españoles en el mensaje

```
parser.add_option("-d", "--discount",
                  action="store", dest="discount", type="float",
                  help="Coeficiente de descuento")
```

- Por omisión el tipo de la opción es un string, pero también acepta `string`, `int`, `long`, `choice`, `float` y `complex`

```
koji@mazinger:~/python$ ./cp_ecologico.py
Usage: cp_ecologico.py [opciones] origen destino

cp_ecologico.py: error: Número incorrecto de argumentos

koji@mazinger:~/python$ ./cp_ecologico.py -h
Usage: cp_ecologico.py [opciones] origen destino

Options:
  -h, --help            show this help message and exit
  -e ENERGY, --energy=ENERGY
                        Tipo de energia a usar en la copia
  -v, --verbose         Informe detallado
  -q, --quiet           Informe silencioso
  -d DISCOUNT, --discount=DISCOUNT
                        Coeficiente de descuento

koji@mazinger:~/python$ ./cp_ecologico.py -v  -d 0.15  mi_origen  mi_destino
Tipo de energia:eolic
Coeficiente de descuento:0.15
Origen: mi_origen
Destino: mi_destino
mucho blablabla
```

Módulos

Un módulo es un fichero que contiene definiciones y sentencias, que pueden ser usados desde otro fichero

mi_modulo.py

```
#!/usr/bin/python -tt
# -*- coding: utf-8 -*-
a=3
def f(x):
    return x+1
```

test.py

```
#!/usr/bin/python -tt
# -*- coding: utf-8 -*-
import mi_modulo

print mi_modulo.a     # 3
print mi_modulo.f(0)  # 1
```

También se pueden importar los objetos por separado, de forma que luego se puede usar sin indicar explícitamente el módulo

```
#!/usr/bin/python -tt
# -*- coding: utf-8 -*-
from mi_modulo import f
from mi_modulo import a

print f(0)  # 1
print a     # 3
```

Es posible importar todos los objetos de un módulo

```
#!/usr/bin/python -tt
# -*- coding: utf-8 -*-
from mi_modulo import *

print f(0)  # 1
print a     # 3
```

Pero esto es una mala práctica, porque cuando el número de módulos aumenta, es difícil saber en qué módulo está cada objeto

Búsqueda de los módulos

El intérprete busca los módulos en el siguiente orden

1. En el directorio del script
2. En cada directorio indicado en la variable de entorno PYTHONPATH
3. En el directorio por omisión
   - En Unix y Linux suele estar en `/usr/lib`
     Por ejemplo
     `/usr/lib/python2.7`
     `/usr/lib/python3.4`

Ficheros .pyc

Cuando se importa un módulo, si el intérprete tiene permisos, guarda en el mismo directorio un fichero con extensión .pyc que contiene el script compilado en bytecodes

- Este fichero ahorra tiempo la segunda vez que se ejecuta el módulo
- No es dependiente de la arquitectura pero sí de la versión exacta del intérprete. Si no existe o no es adecuado, se genera uno nuevo automáticamente
- Permite borrar el fuente .py si no queremos distribuirlo

Objetos en módulos

- Usar objetos globales es peligroso, muchas metodologías lo prohíben
- Pero usar algún objeto global, en un módulo compartido por otros módulos, en algunas ocasiones puede ser una práctica aceptable y conveniente

mis_globales.py

```
#!/usr/bin/python -tt
a=3
```

modulo1.py

```
#!/usr/bin/python -tt
import mis_globales
def f():
    return mis_globales.a
```

test.py

```
#!/usr/bin/python -tt
import mis_globales, modulo1

print modulo1.f() #3
mis_globales.a=5
print modulo1.f() #5
```

Un fichero puede ser un script y un módulo simultáneamente, si añadimos una función main() y la sentencia

```
if __name__== "__main__":
    main()
```

De esta manera,

- Si el fichero se ejecuta como un script, el intérprete ejecutará la función main()
- Si el fichero se usa como módulo, importando sus funciones desde otro script, la función main() no será ejecutada

modulo1.py

```
#!/usr/bin/python -tt
# -*- coding: utf-8 -*-
def f(x):
    return x+1

def main():
    print "probando f", f(2)

if __name__== "__main__":
    main()
```

test.py

```
#!/usr/bin/python -tt
# -*- coding: utf-8 -*-
import modulo1
print modulo1.f(0) #1  No se ejecuta main()
```

Expresiones regulares. Introducción

- Las *expresione regulares* son expresiones que definen un conjunto de cadenas de texto
- Pertenecen a la disciplinas de teoría de autómatas y lenguajes formales. Las bases las sienta Stephen Cole Kleene en la década de 1950. Se desarrollan en los años 60 y se popularizan en los años 80
- Se denominan abreviadamente *re*, *regex* o *regexp*
  También *patrón*
- Las regex son una herramienta muy potente para procesar texto automáticamente. Especialmente texto plano, no son muy apropiadas para HTML o XML

- Las regex pueden manejarse desde
  - Herramientas clásicas como grep, sed, awk
  - Editores de texto
  - Librerías para lenguajes de programación clásicos como C o Pascal
  - Librerías nativas en cualquier lenguaje moderno: perl, python, java, ruby, c#, etc
- Entre las distintinas versiones hay similitudes y diferencias
  - Las regex *tradicionales* (grep, sed, awk) se parecen bastante entre sí.
  - Las regex *modernas* se parecen entre sí. Son una derivación de las tradicionales. Su uso resulta más sencillo
- Es una materia que puede llegar a resultar bastante compleja, conocerlas a fondo es difícil. Pero manejar sus fundamentos resulta de gran utilidad para prácticamente cualquier programador en cualquier entorno

Algunas definiciones

Decimos que una regex y una cadena de texto *encajan* o *no encajan*. [2]
Ejemplo. Patrón/regex

```
[Aa]na [Pp].rez
```

- La cadena `Ana Pérez` encaja
- También encajan las cadenas `ana perez`, `ana pérez`, `ana porez`, `Ana pÑrez`, etc
- La cadena `ANA PEREZ` no encaja

[2] O también *se corresponde*, *se ajusta a*. En inglés, *match*

- Decimos que un carácter [3]
  - Se usa como literal si representa a ese carácter.
  - Se usa como metacarácter (o *comodín*) si tiene un significado especial, si representa algo distinto al propio carácter

  Ejemplo: el punto usado como literal, representa un punto. Usado como metacarácter, representa cualquier carácter
- Normalmente, cuando un carácter puede tomarse como metacarácter o como literal
  - Por omisión se toma como metacarácter
  - Para interpretarlo como literal, hay que escaparlo. Típicamente anteponiendo una barra invertida o incluyendolo entre comillas, rectas o dobles. Ejemplo: `\.`

[3] la palabra *carácter* es llana y lleva tilde, no es aguda. El plural es caracteres, también es llana

Metacaracteres clásicos

```
^       Principio de cadena (principio de línea)
$       Fin de cadena (fin de línea)
.       Cualquier carácter
*       La regex precedente puede aparecer 0 o más veces
?       La regex precedente puede aparecer o no aparecer
[]      Clase de caracteres: uno cualquiera de los caracteres entre
        corchetes
[^]     Complementario de la clase de caracteres: cualquiera menos
        los incluidos entre corchetes
[a-f]   Caracteres de la 'a' hasta la 'f'
{2,3}   La regex precedente se puede repetir entre 2 y 3 veces
{2,}    La regex precedente se repite 2 o más veces
{,3}    La regex precedente se repite entre 0 y 3 veces
{4}     La regex precedente se repite 4 veces
()      Permite agrupar una regex
\2      El segundo grupo de regex
r1|r2   Una regex u otra

\<      Inicio de palabra
\>      Fin de palabra
```

Ejemplos

```
[a-z][a-z0-9_]*     letra minúscula seguida de cero o
                    más letras minúsculas, números o barras bajas
Señora?             Señor o Señora
Serg[eé][iy]? Ra(j|ch|h|kh)m[aá]n[ij]no(v|ff|w)
                    Sergéi / Sergei / Sergey / Serge
                    Rajmáninov / Rachmaninoff / Rahmanjnov ...
```

Dentro una clase de caracteres, cada carácter siempre se toma literalmente, no se escapa ningún posible metacarácter (excepto el cierre de corchetes)

```
[0-9.]          # Un dígito o un punto. (Aquí el punto representa
                  un punto, no "cualquier carácter")
```

Atención: algunos metacaracteres de bash coinciden, otros tienen un significado distinto

```
?     En bash, cualquier carácter
*     En bash, cualquier carácter 0 o más veces
```

Fin de línea

El fin de línea se representa de diferentas maneras

- En MS-DOS/Windows y otros, el fin de línea se representa con `CRLF`
- En Unix, se representa con `LF`

Esto es una fuente tradicional de problemas

- En Windows, un fichero para Unix se verá como una única línea
- En Unix, un fichero para Windows tendrá un `^M` al final de cada línea

Algunos editores son lo bastante *listos* como para mostrar correctamente un fichero con un formato distinto

- Pero ocultar el problema a veces es contraproducente: puede suceder que la apariencia sea correcta, pero el compilador no lo acepte y muestre un error muy confuso

Nombre ASCII	Abreviatura	Decimal	Hexa	Caret Notation	Notación C
Carriage Return	CR	13	OD	`^M`	`\r`
Line Feed	LF	10	0A	`^J`	`\n`

- *Caret notation* es una método empleado en ASCII para representar caracteres no imprimibles. (Caret: acento circunflejo). Normalmente, se puede usar la tecla `control` para generar estos caracteres
- *Notación C*: Notación del lenguaje C, que después han seguido muchos otros como python

Obsérvese que nada de esto se refiere directamente a las expresiones regulares: Cuando en una cadena escribimos `\n`, se entiende que es un avance de línea (excepto si lo escapamos con otra barra adicional, o con una cadena cruda de python)

```
\n suele representar LF, excepto en MacOS, donde suele representar CR.
En java o en .net sobre cualquier SO, siempre representa LF
```

Python emplea *universal newlines*:
En la E/S de ficheros, por omision:

- Sea cual sea el criterio de la entrada, lo convierte a `\n`
- A la salida, escribe el formato propio de la plataforma

Este comportamieto puede cambiarse si es necesario (consultar PEP 278 y PEP 3116)
Para cadenas que no provengan de un fichero, se puede emplear el método *splitlines()* de las cadenas, que:

- Trocea una cadena con el mismo enfoque (soporta todos los criterios), y elimina el fin de linea (sea el que sea)
- A menos que se invoque *splitlines(true)*, entonces conserve el fin de linea, inalterado

Otra fuente típica de problemas: ¿El fin de línea es un terminador o un separador?

- Algunas herramientas/aplicaciones/sistemas operativos entienden que es un separador, y por tanto la última línea no acaba en `\n` sino en fin de fichero
- Otras consideran que es un terminador, por tanto la última línea sí acaba en `\n` (P.e. Unix)

Todo esto son cuestiones que puede ser necesario considerar procesando texto. Pero si lo único que queremos es convertir ficheros entre Windows y Unix, no hace falta usar regex

```
sed -e 's/$/\r/' inputfile > outputfile     # Unix a Windows
sed -e 's/\r$//' inputfile > outputfile     # Windows a Unix
```

El metacarácter $ de las regex no se corresponde exactamente con CR ni con LF. Su significado exacto depende de la plataforma. Normalmente encaja tanto con el fin de cadena como con la posición inmediatamente antes de LF/CR/CRLF

Metacaracteres modernos

El lenguaje perl es el *padre* de las regex modernas. Incluye los metacaracteres clásicos y añade otros nuevos. Lenguajes como python copian las regex de perl

Metac.		Clase equivalente	
\d	Dígito	[0-9]	
\s	Espacio en blanco, tab...	[\ \t\r\n\f]	(*)
\w	Carácter de palabra (alfanumético o barra baja)	[0-9a-zA-Z_]	
\D	Cualquiera menos \d	[^0-9]	
\S	Cualquiera menos \s	[^\s]	
\W	Cualquiera menos \w;	[^\w]	
\b	Limite de palabra. (Secuencia de alfanuméricos o barra baja)		

(*) \t: Tab
\f: Form Feed, salto de página

Observaciones

- El único metacarácter que cambia entre regex clásicas y modernas es el límite de palabra, se usa \b y no \< \>
- Las locales no siembre están bien definidas, en tal caso para definir una palabra tal vez haya que incluir explicitamente las letras españolas (si procede)

Regexp en python

- Para operaciones sencillas con cadenas, como búsquedas y sustituciones sin metacaracteres, es más eficiente emplear los métodos de las cadenas, como `find` y `replace`
- El módulo `re` tiene funciones a la que se puede pasar directamente una cadena regexp

```
>>> import re
>>> m=re.search('[0-9]+' , 'abc98521zzz')
>>> m.group(0)
'98521'
```

Pero aquí usaremos objetos regex, más potentes

Regexp en python

- Para usar regexp, importamos el módulo `re`

  ```
  import re
  ```

- Una regex es un objeto que construimos con la función `compile`

  ```
  regex=re.compile("a+")
  ```

- Para buscar el patrón en una cadena tenemos los métodos
  - `match()`, que comprueba si el principio de la cadena encaja en la regex
  - `search()`, que comprueba si alguna parte de la cadena encaja en la regex

  Ambos métodos devuelven
  - Un objeto `SRE_Match` si han tenido éxito
  - `None` si han fracasado

```
#!/usr/bin/python -tt
# -*- coding: utf-8 -*-
import re
regex=re.compile("aa+")

m=regex.match("taartamudo")
print m    # None

m=regex.search("taartamudo")
print m    # Cierto

m=regex.match("aaahora")
print m    # Cierto
```

Casi siempre hay más de una regex posible. Ejemplo: Capturar una dirección IP

Estas sentencias son equivalentes

```
direccion_ip=re.compile(r"""\d\d?\d?\.\d\d?\d?\.\d\d?\d?\.\d\d?\d?""")
direccion_ip=re.compile(r"""(\d\d?\d?\.){3}\d\d?\d?""")
direccion_ip=re.compile(r"""\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}""")
direccion_ip=re.compile(r"""(\d{1,3}\.){3}\d{1,3}""")
```

- Es necesario *escapar* el punto
- Obsérvese que esta regex no se corresponde exactamente con una dirección IP. Por ejemplo admitiría `315.15.256.715`
- Suele ser conveniente definir la regex con *cadenas crudas* de python (`r"""cadena"""`)
  Esto evita tener que escapar las barras invertidas para que se tomen como literales.
  También permite, por ejemplo, que la secuencia `\n` se tome como como una barra invertida y una ene. (Y no como un salto de línea carro)

Comentarios en las regex

El flag `re.VERBOSE` es muy útil. Al activarlo se ignoran

- Los espacios (no *escapados*)
- Las almohadillas y todo el texto posterior, hasta fin de línea

```
ip = re.compile(r"""
         (\d{1,3}\.){3}  # de 1 a 3 digitos y punto, repetido 3 veces
         \d{1,3}         # de 1 a 3 digitos
         """, re.VERBOSE)
```

Otros flags

- `re.VERBOSE`
  `re.X`
  Permite comentarios dentro de la regex
- `re.IGNORECASE`
  `re.I`
  No distingue entre mayúsculas y minúsculas
- `re.LOCALE`
  `re.L`
  Hace que `\w`, `\W`, `\b`, `\B`, `\s` y `\S` tengan en cuenta las *locales*

Para combinar más de un flag, se usa la barra vertical (`'|'`), que es el operador *or* a nivel de bit.

Grupos

Un objeto `SRE_Match` devuelve en el atributo `group` las partes de la cadena que han encajado en la regex
`group[0]` es el texto que ha encajado en la regex completa

```
#!/usr/bin/python -tt
# -*- coding: utf-8 -*-
import re
ip = re.compile(r"""
                (\d{1,3}\.){3}  # de 1 a 3 digitos y punto, repetido 3 veces
                \d{1,3}         # de 1 a 3 digitos
                """, re.VERBOSE)
texto=r"""Mi correo es j.perez@alumnos.urjc.es
          y mi dirección IP, 192.168.1.27"""

for linea in texto.split('\n'):
    m=ip.search(linea)
    if m:
        print m.group(0)
```

Ejecución:

```
koji@mazinger:~$ ./ejemplo_regex.py
192.168.1.27
```

Los paréntesis

- Como hemos visto, definen el ámbito y precedencia de los demás operadores
- Además, definen grupos. El resultado de cada búsqueda devuelve en `group[n]` el grupo n-ésimo

```
#!/usr/bin/python -tt
# -*- coding: utf-8 -*-
import re
correo_alumno = re.compile(r"""
(
\b                  # Límite de palabra
[\w.]+              # 1 o más caracteres de palabra o punto
\b                  # límite de palabra
)                   # Hasta aquí el grupo 1
@
(alumnos\.urjc\.es) # Grupo 2
""", re.VERBOSE)

texto=r"""Llegó un correo de j.perez@alumnos.urjc.es preguntando
          si hay clase mañana"""

for linea in texto.split('\n'):
    m=correo_alumno.search(linea)
    if m:
        print "Alumno: "+m.group(1)    # j.perez
        print "Dominio: "+m.group(2)   # alumnos.urjc.es
```

Dentro de una regex, podemos hacer referencia a un grupo

```
#!/usr/bin/python -tt
# -*- coding: utf-8 -*-
import re

regex=re.compile(r"""(\b\w+\b)  # Una palabra
                 \s+            # Espacios
                 \1             # Grupo 1: la misma palabra
                 """, re.VERBOSE)

texto=r"""Buscando palabras repetidas repetidas"""

for linea in texto.split('\n'):
    m=regex.search(linea)
    if m:
        print m.group(1)  # Devuelve "repetidas"
```

Ejemplo de definición explícita de palabra española

```
#!/usr/bin/python -tt
# -*- coding: utf-8 -*-
import re

regex=re.compile(r"""
              (\b                    # Límite de palabra
              [\wáéíóúÁÉÍÓÚñÑüÜ]+ # Palabra, incluyendo letras españolas
              \b)
              \s*                    # Espacios, opcionalmente
              $                      # Fin de línea
              """, re.VERBOSE)

texto=r"""Buscando la última palabra de la línea  """

for linea in texto.split('\n'):
    m=regex.search(linea)
    if m:
        print m.group(1)  # Devuelve "línea"
```

Sustituciones

Además de `search` y `match`, los objetos regex tienen el método `sub(reemplazo,cadena)` que

- Busca el patrón en la cadena
- Si lo encuentra, reempleza el texto que ha encajado por `reemplazo`
  Dentro de `reemplazo` se pueden usar referencias a grupos
- Devuelve el texto resultante

```
#!/usr/bin/python -tt
# -*- coding: utf-8 -*-
import re
# reemplazamos los correos  login@urjc.es por
# [Correo de login en la URJC ]

correo_urjc = re.compile(r"""
(
\b                # Límite de palabra
[\w.]+            # 1 o más caracteres de palabra o punto
\b                # límite de palabra
)
@urjc\.es
""", re.VERBOSE)

texto="Si es necesario, escribe a j.perez@urjc.es"
for linea in texto.split('\n'):
    print correo_urjc.sub(r"""[Correo de \1 en la URJC]""",linea)
```

Resultado de la ejecución

```
koji@mazinger:~/python$ ./test.py
Si es necesario, escribe a [Correo de j.perez en la URJC]
```

Regex multilínea

Hasta ahora hemos procesado cada línea de forma independiente de las demás, lo cual es bastante frecuente
En este caso

- El metacarácter '^' representa el principio de cadena, lo que equivale al principio de línea
- El metacarácter '$' representa el fin de cadena, lo que equivale al fin de línea
- El metacarácter '.' no encaja en el fin de línea

Pero en otras ocasiones querremos aplicar la regex a más de una línea. Esto generalmente requiere de algunos *flags* adicionales

- `re.DOTALL`
  `re.S`
  Hace que el metacarácter `'.'` encaje en el fin de línea
- `re.MULTILINE`
  `re.M`
  Hace que el metacarácter `'^'` represente el principio de línea
  El metacarácter `'$'` representa el fin de línea

```
#!/usr/bin/python -tt
# -*- coding: utf-8 -*-
import re
regex=re.compile(r"""
                 ^                    # Principio de línea
                 (\b
                 [\wáéíóúÁÉÍÓÚñÑ]+    # Palabra
                 \b)                  #
                 .*                   # Resto de la línea
                 ^                    # Comienzo de línea
                 \1                   # La misma palabra
                 """, re.VERBOSE|re.MULTILINE|re.DOTALL)

texto=r"""En este ejemplo estamos
buscando dos líneas que comiencen igual
buscando líneas con primera palabra
coincidente
"""
m=regex.search(texto)
if m:
    print m.group(1)  # Devuelve "buscando"
```

Split con regex

Se puede trocear una cadena indicando con una regex cuál es el separador
Ejemplo: queremos una lista con todos los unos consecutivos, separados por ceros

```
>>> import re
>>> miregex=re.compile(r'0+')
>>> miregex.split('1001110001111000 1')
['1', '111', '1111', '1']
```

Atención: el separador, por definición, está entre dos elementos. No antes del primero ni después del último.
En el siguiente ejemplo los ceros no se comportan como separadores, por lo que el resultado no es exactamente el deseado (aunque se acerca mucho)

```
>>> miregex.split('00100111000111100010')
['', '1', '111', '1111', '1', '']
```

Referencias

- The Python Standard Library
- *Mastering Regular Expressions*. Jeffrey E. F. Friedl. Ed. O'Reilly, 2006