12. LIBRERÍAS DE PYTHON

12.1. Librería sys

libreria sys

- Argumentos de linea de órdenes

sys.argv devuelve una lista con los argumentos pasados al script python desde la shell

```
#!/usr/bin/env python3
# argumentos.py
import sys
print(sys.argv[:])
```

```
koji@mazinger:~$ ./argumentos.py un_argumento otro_argumento
['./argumentos.py', 'un_argumento', 'otro_argumento']
```

(El argumento cero es el nombre del programa)

Escribir en stderr

```
#!/usr/bin/env python3
# error_estandar.py
import sys
sys.stderr.write('Error: \n')
```

Leer desde stdin, escribir en stdout

```
#!/usr/bin/env python3
# cat.py
import sys
for linea in sys.stdin.readlines():
    sys.stdout.write(linea)
```

12.2. Librería subprocess

subprocess

- subprocess.check_output() permite ejecutar una orden de la shell en un subproceso externo

- Aunque puede ser muy útil, el script deja de ser portable entre sistemas operativos diferentes
- Su primer argumento es una lista con los argumentos de la orden a ejecutar
- Devuelve la salida estándar del subproceso pero como bytes que habrá que convertir a cadena (unicode) con decode
    - En caso contrario obtendremos un error
      TypeError: a bytes-like object is required, not 'str'
- Si se produce algún error, eleva la excepción subprocess.CalledProcess-Error

```
#!/usr/bin/env python3
# subproceso01.py
import subprocess,sys
mandato="ps -ef"
mandato_troceado=mandato.split()
try:
    salida=subprocess.check_output(mandato_troceado)
except subprocess.CalledProcessError:
    sys.stderr.write("La orden ha producido un error\n")
    raise SystemExit
salida=salida.decode("utf-8")      # De bytes a string
lineas=salida.split("\n")      # troceamos la salida línea a línea
lineas.pop(0)      # quitamos la primera línea, la cabecera del ps
for linea in lineas:
    if len(linea) != 0:
        campos_linea=linea.split()
        usuario = campos_linea[0]
        proceso = campos_linea[7]
        print("Usuario:{} Proceso:{}".format(usuario,proceso))
```

Si necesitamos más control sobre los posibles errores

- Para redirigir la salida de error del subproceso a la salida estándar, pasamos el parámetro stderr=subprocess.STDOUT
- El atributo returncode de CalledProcessError contiene el código del error

```
#!/usr/bin/env python3
# subproceso02.py
import subprocess,sys
```

```
mandato="ls inexistente"
mandato_troceado=mandato.split()
try:
    salida=subprocess.check_output(mandato_troceado,
                                   stderr=subprocess.STDOUT)
except subprocess.CalledProcessError as e:
    plantilla = "{}Código:{}\n"
    msj_subproceso = e.output
    msj_subproceso = msj_subproceso.decode()      # De bytes a string
    codigo_subproceso = e.returncode
    msj = plantilla.format(msj_subproceso, codigo_subproceso)
    sys.stderr.write(msj)
```

12.3. Librerías os, shutil

os.path

- Las funciones os.path.join() y os.path.split() unen y separan nombres de fichero con directorios
  - Son compatibles con cualquier S.O.
  - No importa si el path acaba en barra o no
- os.path.exists() devuelve un boolean indicando si un fichero existe

```
#!/usr/bin/env python3 # path01.py
# path01.py
import    os
ejemplo=os.path.join("/etc/apt","sources.list")
print(ejemplo)       # /etc/apt/sources.list
print(os.path.split(ejemplo))       # ('/etc/apt', 'sources.list')

print(os.path.exists(ejemplo))       # True
print(os.path.exists("/usr/local/noexiste"))      # False
```

Enlazar, borrar

```
#!/usr/bin/env python
# enlazar_borrar.py
import    os
if not os.path.exists("/tmp/test"):
    os.mkdir("/tmp/test")
```

```
os.chdir("/tmp/test")                          # cd /tmp/aa
os.symlink("/etc/hosts","enlace_hosts")        # crea enlace simbólico
os.remove("enlace_hosts")                      # borra el fichero
os.rmdir("/tmp/test")                          # borra directorio (vacio)
```

copiar, copiar y borrar recursivamente

```
#!/usr/bin/env python3
# recursivo.py
import   shutil,os
shutil.copytree("/home/koji/.gnome", "/tmp/probando")
     # copia recursivamente. El destino no debe existir

shutil.copy("/etc/hosts", "/tmp/probando")
     # copia 1 fichero (como el cp de bash)

shutil.move("/tmp/probando/hosts", "/tmp/probando/mi_hosts")

shutil.rmtree("/tmp/probando")          # borra directorio lleno
```

os.walk

- Recorre recursivamente un directorio
- Por cada directorio devuelve una 3-tupla
    - Directorio
    - Subdirectorios
    - Ficheros

```
#!/usr/bin/env python3
# walk.py
import os
directorio_inicial=os.getcwd()   # current working directory
os.chdir("/tmp/musica")          # cd

for x in os.walk("."):
```

```
    print(x)

os.chdir(directorio_inicial)
```

```
/tmp/musica
|-- listado.txt
|-- jazz
`-- pop
    |-- sabina
    |   |-- pirata_cojo.mp3
    |   `-- princesa.mp3
    `-- serrat
        |-- curro_el_palmo.mp3
        `-- penelope.mp3

('.', ['jazz', 'pop'], ['listado.txt'])
('./jazz', [], [])
('./pop', ['serrat', 'sabina'], [])
('./pop/serrat', [], ['curro_el_palmo.mp3', 'penelope.mp3'])
('./pop/sabina', [], ['princesa.mp3', 'pirata_cojo.mp3'])
```

12.4. Librerías pickle: Persistencia

Persistencia

Persistencia en Python:

Una librería de persistencia permite *serializar objetos*, esto es, convertirlos en una secuencia binaria o de texto que se pueda transmitir fuera de python, almacenar en disco, base de datos, etc

- Librería *Pickle*
  - Formato propio de Python, binario, muy eficiente
  - Soporta cualquire clase, predefinida en Python o por el usuario
- Librería *json*
  - Formato estándar de internet, modo texto, legible
  - Solo soporta cadenas, números, booleanos, diccionarios y listas

Persistencia con pickle

```
#!/usr/bin/env python3
# conserva.py
import pickle

cp={28:'madrid',8:'barcelona',33:'asturias'}
fich=open('prueba.pick','wb')              # Apertura modo binario
pickle.dump(cp,fich)
fich.close()

fich=open('prueba.pick','rb')              # Lectura modo binario
codigos_postales=pickle.load(fich)
fich.close()

for x in codigos_postales.keys():
    print(x,codigos_postales[x])
```

Persistencia con json

```
#!/usr/bin/env python3
# ejemplo_json.py
import json

# Objeto python ordinario
diccionario = { 'MAD':'Madrid Barajas', 'MCV':'Madrid Cuatro Vientos'}
print(type(diccionario))        # <class 'dict'>

# Lo convertimos en cadena json
cadena_json = json.dumps(diccionario)
print(type(cadena_json))    # <class 'str'>
print(cadena_json)
# {"MAD": "Madrid Barajas", "MCV": "Madrid Cuatro Vientos"}

# Volvemos a convertir la cadena en objeto python
diccionario = json.loads(cadena_json)
print(type(diccionario))        # <class 'dict'>
```

12.5. Acceso a las variables de entorno

Variables de entorno

```
#!/usr/bin/env python3
# entorno.py
import os, sys
mi_variable=os.getenv("MI_VARIABLE")
if mi_variable==None:
        msg="ERROR: variable de entorno MI_VARIABLE no definida"
        sys.stderr.write(msg+'\n')
        raise SystemExit
```

Atención: Cuando la shell crea un proceso (p.e. el intérprete de python), puede no pasarle todas las variables de entorno. Por tanto, las variables visibles desde la shell serán distintas a las visibles desde python

12.6. Módulos

Módulos

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

mi_modulo.py

```
#!/usr/bin/env python3
a=3
def f(x):
    return x+1
```

test.py

```
#!/usr/bin/env python3
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/env python3
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/env python3
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/python3.9

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/env python3
a=3
```

modulo1.py

```
#!/usr/bin/env python3
import mis_globales
def f():
    return mis_globales.a
```

test.py

```
#!/usr/bin/env python3
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/env python3
def f(x):
    return x+1

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

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

test.py

```
#!/usr/bin/env python3
import modulo1
print(modulo1.f(0))   #1   No se ejecuta main()
```

12.7. optparse

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/env python3
# opciones01.py
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
- 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

```
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
```

12.8. Bots de telegram

Bots de telegram

Telegram es una aplicación de mensajería instantánea muy similar a WhatsApp, con la que tiene las siguientes diferencias:

- Telegram es muy popular pero no tiene tantos usuarios como WhatsApp
- Telegram no solo tiene clientes para iOS y Android (como WhatsApp) , también tiene clientes independientes del telefono para Windows, Linux y macOS
- El cliente es software libre (el servidor,no)
- Fácilmente accesible mediante API

En python hay muchas librerías para manejar telegram, aqúı veremos *telepot*

12.8.1. Creación de un bot de telegram

Para poder enviar y recibir mensajes, hay que crear un tipo de usuario especial llamado *bot*

Para crear un bot, tenemos que usar otro bot, llamado *BotFather*

1. Desde nuestro cliente de telegram, enviamos un mensaje con el texto /newbot al usuario @BotFather
2. Un diálogo nos irá preguntando todo lo necesario
3. Recibiremos un *token* que nos permitirá maneja el bot via API

De la misma forma, @BotFather nos permite cambiar el nombre de nuestro bot (comando /newbot), su foto (/setuserpic), eliminarlo (deletebot), etc

Para que un bot de telegram pueda enviar un mensaje a un usuario, hace falta:

1. Que el usuario le envíe al bot un mensaje cualquiera (un mensaje en la vida es suficiente)
2. Que el usuario facilite al programador del bot su propio id de usuario

- Es un número de unos 9 dígitos, no confundir con el nombre de usuario (p.e. @JuanPerez)
- El usuario puede averiguar su propio id enviando un mensaje cualquiera al bot @userinfobot

12.8.2. Uso de la librería telepot

Uso de la librería telepot

Para instalar telepot

- Si tenemos privilegios de root y queremos instalarlo para todos los usuarios del sistema, ejecutamos desde la shell
  pip3 install telepot
- En otro caso, podemos instalarlo solo para nuestro usuario:
  pip3 install --user telepot

Enviar un mensaje a un usuario

```
#!/usr/bin/env python3
import telepot

TOKEN = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
bot = telepot.Bot(TOKEN)

def main():
    id_usuario = "999999999"
    bot.sendMessage(id_usuario, "Hola")

    return

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

Contestar a los mensajes de un usuario

```
#!/usr/bin/env python3
import telepot
import telepot.namedtuple
import time
from telepot.loop import MessageLoop

TOKEN = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
bot = telepot.Bot(TOKEN)

def handle(msg):

    chat_id = msg["chat"]["id"]
    texto = msg["text"]

    print("Recibiendo mensaje:")
    print(msg)

    respuesta = "Me has dicho " + texto
    bot.sendMessage(chat_id, respuesta)
    return
```

```
def main():
    MessageLoop(bot, handle).run_as_thread()
    print "Escuchando..."

    while 1:
        time.sleep(10)
    return

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

Ejemplo:

Escuchando...
Recibiendo mensaje:
{u'date': 1542706429, u'text': u'Hola', u'from': {u'username': u'Juan_perez', u'first_name': u'Juan', u'last_name': u'Perez', u'is_bot': False, u'language_code': u'es', u'id': 154195197}, u'message_id': 2704, u'chat':
{u'username': u'Juan_perez', u'first_name': u'Juan', u'last_name': u'Perez', u'type': u'private', u'id': 154196127}}

- En los ejemplos anteriores, para preparar un ejemplo mínimo hemos escrito el token en el código fuente

- Por motivos de seguridad, en cualquier programa que no sea una prueba de concepto, el token (o cualquier otra contraseña similar) debería ir en un fichero aparte

    - Al leer el token desde un fichero, será necesario que suprimas el salto de línea final
    - Puedes usar el método strip(), que devuelve una copia de la cadena eliminando tabuladores, espacios y saltos de línea al princio y al final de una cadena

```
>>> a = ' hola '
>>> a.strip()
'hola'
```

12.9. Expresiones Regulares en python

12.9.1. Introducción
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*. [8]

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

- Decimos que un carácter[9]

    - 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: \.

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

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

12.9.2. Metacaracteres

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. (Aqúı 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)

12.9.3. Regexp en python

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 aqúı 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 (que se evalúa como *cierto*)

```
#!/usr/bin/env python3
# regex01.py
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/env python3
# regex02.py
import re
ip = re.compile(r"""
        (\d{1,3}\.){3}          # 1,2 o 3 digitos y punto, repetido 3 veces
        \d{1,3}                 # 1,2 o 3 dígitos
        """, 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:

```
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/env python3
# regex03.py
import re
correo_alumno = re.compile(r"""
(
\b                      # Límite de palabra
[\w.]+                  # 1 o más caracteres de palabra o punto
\b                      # límite de palabra

(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: {}".format(m.group(1)))        # j.perez
        print("Dominio: {}".format(m.group(2)))      # alumnos.urjc.es
```

Dentro de una regex, podemos hacer referencia a un grupo

```
#!/usr/bin/env   python3 #
regex04.py
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/env   python3
# regex05.py
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/env  python3
# regex06.py
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 aj.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/env  python3
# regex07.py
import re
regex=re.compile(r"""
          ^                    # Principio de línea
          (\b
          [\wáéíóú´A´E´I´O´U~n~N]+    # 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('100111000111110001')
['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('001001110000111100010')
['', '1', '111', '1111', '1', '']
```