Python para la administración de sistemas GNU/Linux. Shell scripting introduction in Python for Linux Systems administration.

Book · October 2018

CITATIONS
0

READS
573

1 author:

Eugenia Bahit
Hackers & Developers Press
18 PUBLICATIONS **0** CITATIONS

SEE PROFILE

EUGENIA BAHIT

PYTHON PARA LA ADMINISTRACIÓN DE SISTEMAS GNU/LINUX

MATERIAL DE ESTUDIO

Informes e inscripción:
Curso: http://python.eugeniabahit.com | **Certificaciones**: http://python.laeci.org

SUMARIO

Unidad 1: Manejo y manipulación de archivos....4
Modos de Apertura de un archivo....4
Algunos métodos del Objeto File....6
Acceso a archivos mediante la estructura with y la función open....6
Ejercicios....7
Unidad 2: Manipulación básica de cadenas de texto y diccionarios....10
Métodos del objeto string....10
Manejo de diccionarios....14
Ejercicios....16
Unidad 3: Generación de registros de sistema....18
Principales elementos del módulo logging....18
Ejercicio de generación de registros....21
Obtención de argumentos por línea de comandos con argv....22
Captura básica de excepciones con try y except....22
Unidad 4: Manipulación avanzada de cadenas de texto....27
Expresiones regulares en Python....28
Ejercicio....29
Unidad 5: Creación de un menú de opciones con argparse....32
Paso 1: Importación del módulo....32
Paso 2: Construcción de un objeto ArgumentParser....32
Paso 3: Agregado de argumentos y configuración....33
Paso 4: Generación del análisis (parsing) de argumentos....35
Ejercicio....35
Unidad 6: Módulos del sistema (os, sys y subprocess)....39
El módulo OS....39
Variables de entorno: os.environ....41
Ejecución de comandos del sistema mediante subprocess y shlex....41
Capturar la salida estándar y los errores....42
Emplear la salida de un comando como entrada de otro....43
Variables y funciones del módulo sys....44
Ejercicios....47
Unidad 7: Conexiones remotas (HTTP, FTP y SSH)....51
Conexiones remotas vía HTTP y HTTPS....51
Conexiones remotas vía FTP....53

Solicitando la contraseña con getpass.....54
Conexiones SSH con Paramiko.....55
Requisitos previos.....55
Uso de Paramiko.....56
Ejercicio integrador.....58
Unidad 8: Librerías para el Manejo avanzado de archivos, en sistemas GNU/Linux.....62
Compresión y descompresión de archivos con las librerías tarfile y zipfile....62
La librería tarfile.....62
La Librería zipfile.....63
Manejo de archivos temporales con la librería tempfile.....64
Lectoescritura de archivos temporales.....64
Búsqueda de archivos con las librerías glob y fnmatch.....65
Ejercicio integrador.....67

UNIDAD 1: MANEJO Y MANIPULACIÓN DE ARCHIVOS

Python permite trabajar en dos niveles diferentes con respecto al sistema de archivos y directorios.

Uno de ellos, es a través del módulo *os*, que facilita el trabajo con todo el sistema de archivos y directorios, a nivel del propios Sistema Operativo.

El segundo nivel, es el que permite trabajar con archivos manipulando su lectura y escritura desde la propia aplicación o *script*, tratando a cada archivo como un objeto.

MODOS DE APERTURA DE UN ARCHIVO

El **modo de apertura de un archivo**, está relacionado con el objetivo final que responde a la pregunta ***"¿para qué se está abriendo este archivo?"***. Las respuestas pueden ser varias: para leer, para escribir, o para leer y escribir.

Cada vez que se "abre" un archivo se está creando un ***puntero en memoria***.

Este puntero posicionará un ***cursor*** (o *punto de acceso*) en un lugar específico de la memoria (dicho de modo más simple, posicionará el cursor en un *byte* determinado del contenido del archivo).

Este cursor se moverá dentro del archivo, a medida que se lea o escriba en dicho archivo.

Cuando un archivo se abre en modo lectura, el cursor se posiciona en el *byte 0* del archivo (es decir, al comienzo del archivo). Una vez leído el archivo, el cursor pasa al *byte final* del archivo (equivalente a cantidad total de *bytes* del

archivo). Lo mismo sucede cuando se abre en modo escritura. El cursor se moverá a medida que se va escribiendo.

Cuando se desea escribir al final de un archivo no nulo, se utiliza el modo *append* (agregar). De esta forma, el archivo se abre con el cursor al final del archivo.

El símbolo + como sufijo de un modo, agrega el modo contrario al de apertura una vez se ejecute la acción de apertura. Por ejemplo, el modo r (read) con el sufijo + (r+), abre el archivo para lectura, y tras la lectura, vuelve el cursor al *byte* 0.

La siguiente tabla muestra los diferentes modos de apertura de un archivo:

Indicador	Modo de apertura	Ubicación del puntero
`r`	Solo lectura	Al inicio del archivo
`rb`	Solo lectura en modo binario	Al inicio del archivo
`r+`	Lectura y escritura	Al inicio del archivo
`rb+`	Lectura y escritura en modo binario	Al inicio del archivo
`w`	Solo escritura. Sobreescribe el archivo si existe. Crea el archivo si no existe.	Al inicio del archivo
`wb`	Solo escritura en modo binario. Sobreescribe el archivo si existe. Crea el archivo si no existe.	Al inicio del archivo
`w+`	Escritura y lectura. Sobreescribe el archivo si existe. Crea el archivo si no existe.	Al inicio del archivo
`wb+`	Escritura y lectura en modo binario. Sobreescribe el archivo si existe. Crea el archivo si no existe.	Al inicio del archivo
`a`	Añadido (agregar contenido). Crea el archivo si éste no existe.	Si el archivo existe, al final de éste. Si el archivo no existe, al comienzo.
`ab`	Añadido en modo binario (agregar contenido).	Si el archivo existe, al final de éste.

	Crea el archivo si éste no existe.	Si el archivo no existe, al comienzo.
a+	Añadido (agregar contenido) y lectura. Crea el archivo si éste no existe.	Si el archivo existe, al final de éste. Si el archivo no existe, al comienzo.
ab+	Añadido (agregar contenido) y lectura en modo binario. Crea el archivo si éste no existe.	Si el archivo existe, al final de éste. Si el archivo no existe, al comienzo.

ALGUNOS MÉTODOS DEL OBJETO FILE

El objeto file, entre sus métodos dispone de los siguientes:

Método	Descripción
read([bytes])	Lee todo el contenido de un archivo. Si se le pasa la longitud de bytes, leerá solo el contenido hasta la longitud indicada.
readlines()	Lee todas las líneas de un archivo
write(cadena)	Escribe *cadena* dentro del archivo
writelines(secuencia)	Secuencia será cualquier iterable cuyos elementos serán escritos uno por línea

ACCESO A ARCHIVOS MEDIANTE LA ESTRUCTURA WITH Y LA FUNCIÓN OPEN

Con la estructura with y la función open(), puede abrirse un archivo en cualquier modo y trabajar con él, sin necesidad de cerrarlo o destruir el puntero, ya que de esto se encarga la estructura with.

Leer un archivo:

```
with open("archivo.txt", "r") as archivo:
    contenido = archivo.read()
```

Escribir en un archivo:

```
contenido = """
    Este será el contenido del nuevo archivo.
    El archivo tendrá varias líneas.
"""

with open("archivo.txt", "r") as archivo:
    archivo.write(contenido)
```

EJERCICIOS

1) Leer un archivo PDF, JPG, GIF o PNG, en modo binario, y hacer el contenido leído se escriba en un nuevo archivo (siempre utilizando la estructura with y open).

2) Leer un archivo de texto plano, línea por línea, y recuperar en una variable, solo la última línea, y en otra variable, todas las líneas menos la primera.

3) Si se siente curiosidad por trabajar con archivos de datos CSV, puede leerse el apartado de «Manejo de archivos CSV» del libro del curso de «Introducción a la Ciencia de Datos con Python».

UNIDAD 2: MANIPULACIÓN BÁSICA DE CADENAS DE TEXTO Y DICCIONARIOS

En Python, toda variable se considera un *objeto*. Sobre cada objeto, pueden realizarse diferentes tipos de acciones denominadas *métodos*. Los métodos son funciones pero que se desprenden de una variable. Por ello, se accede a estas funciones mediante la sintaxis:

```
variable.funcion()
```

En algunos casos, estos métodos (funciones de un objeto), aceptarán parámetros como cualquier otra función.

```
variable.funcion(parametro)
```

MÉTODOS DEL OBJETO STRING

A continuación, se verán algunos de los principales métodos que pueden aplicarse sobre una cadena de texto. Para una lista más extensa, organizada por categorías, se recomienda ver las páginas 5 a 13 del libro del curso de **«Introducción a la Ciencia de Datos con Python»** disponible en https://www.safecreative.org/search?q=1809018247679.

Contar cantidad de apariciones de una subcadena

```
Método: count("cadena"[, posicion_inicio, posicion_fin])
Retorna: un entero representando la cantidad de apariciones de subcadena dentro
de cadena

>>> cadena = "bienvenido a mi aplicación".capitalize()
>>> cadena.count("a")
3
```

Buscar una subcadena dentro de una cadena

```
Método: find("subcadena"[, posicion_inicio, posicion_fin])
Retorna: un entero representando la posición donde inicia la subcadena dentro de
cadena. Si no la encuentra, retorna -1
>>> cadena = "bienvenido a mi aplicación"
>>> cadena.find("mi")
13
>>> cadena.find("mi", 0, 10)
-1
```

Saber si una cadena comienza con una subcadena determinada

```
Método: startswith("subcadena"[, posicion_inicio, posicion_fin])
Retorna: True o False
>>> cadena = "bienvenido a mi aplicación"
>>> cadena.startswith("bienvenido")
True
>>> cadena.startswith("aplicación")
False
>>> cadena.startswith("aplicación", 16)
True
```

Saber si una cadena finaliza con una subcadena determinada

```
Método: endswith("subcadena"[, posicion_inicio, posicion_fin])
Retorna: True o False
>>> cadena = "bienvenido a mi aplicación"
>>> cadena.endswith("aplicación")
True
>>> cadena.endswith("bienvenido")
False
>>> cadena.endswith("bienvenido", 0, 10)
True
```

Dar formato a una cadena, sustituyendo texto dinámicamente

```
Método: format(*args, **kwargs)
Retorna: la cadena formateada
>>> cadena = "bienvenido a mi aplicación {0}"
>>> cadena.format("en Python")
bienvenido a mi aplicación en Python

>>> cadena = "Importe bruto: ${0} + IVA: ${1} = Importe neto: {2}"
>>> cadena.format(100, 21, 121)
Importe bruto: $100 + IVA: $21 = Importe neto: 121

>>> cadena = "Importe bruto: ${bruto} + IVA: ${iva} = Importe neto: {neto}"
>>> cadena.format(bruto=100, iva=21, neto=121)
Importe bruto: $100 + IVA: $21 = Importe neto: 121

>>> cadena.format(bruto=100, iva=100 * 21 / 100, neto=100 * 21 / 100 + 100)
Importe bruto: $100 + IVA: $21 = Importe neto: 121
```

Reemplazar texto en una cadena

```
Método: replace("subcadena a buscar", "subcadena por la cual reemplazar")
Retorna: la cadena reemplazada
>>> buscar = "nombre apellido"
>>> reemplazar_por = "Juan Pérez"
>>> "Estimado Sr. nombre apellido:".replace(buscar, reemplazar_por)
Estimado Sr. Juan Pérez:
```

Unir una cadena de forma iterativa

```
Método: join(iterable)
Retorna: la cadena unida con el iterable (la cadena es separada por cada uno de
los elementos del iterable)
>>> formato_numero_factura = ("Nº 0000-0", "-0000 (ID: ", ")")
>>> numero = "275"
>>> numero_factura = numero.join(formato_numero_factura)
>>> numero_factura
Nº 0000-0275-0000 (ID: 275)
```

Partir una cadena en varias partes, utilizando un separador

```
Método: split("separador")
Retorna: una lista con todos elementos encontrados al dividir la cadena por un
separador
>>> keywords = "python, guia, curso, tutorial".split(", ")
>>> keywords
['python', 'guia', 'curso', 'tutorial']
```

Partir una cadena en en líneas

```
Método: splitlines()
Retorna: una lista donde cada elemento es una fracción de la cadena divida en
líneas
>>> texto = """Linea 1
Linea 2
Linea 3
Linea 4
"""
>>> texto.splitlines()
['Linea 1', 'Linea 2', 'Linea 3', 'Linea 4']
>>> texto = "Linea 1\nLinea 2\nLinea 3"
>>> texto.splitlines()
['Linea 1', 'Linea 2', 'Linea 3']
```

MANEJO DE DICCIONARIOS

Varios son los métodos provistos por el objeto dict. En el curso de «Introducción a la Ciencia de Datos con Python», se hace un recorrido exhaustivo por estos métodos. Esta unidad se centrará solo en los métodos **get** e **items** del objeto dict, empleados para acceder a una clave en particular o iterar sobre un diccionario.

Obtener el valor de una variable (clave del diccionario)

```
Método: get(clave[, "valor x defecto si la clave no existe"])
>>> environ.get("PWD")
'/home/eugenia/cursos'

>>> environ.get("DOCUMENT_ROOT", "Entorno incorrecto")
'Entorno incorrecto'
```

Saber si la clave existe en el diccionario

Instrucción: 'clave' in diccionario

```
>>> if 'DOCUMENT_ROOT' in environ:
>>>     echo("Entorno Web")
```

Obtener las claves y valores de un diccionario

Método: items()

```
diccionario = {'color': 'rosa', 'marca': 'Zara', 'talle': 'U'}

for clave, valor in diccionario.items():
    clave, valor
```

Salida:

```
('color', 'rosa')
('marca', 'Zara')
('talle', 'U')
```

En Python 2 existía iteritems():

```
>>> a = dict(a=1, b=2)
>>> a.iteritems()
```

En Python 3 ya no existe:

```
>>> a.iteritems()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'dict' object has no attribute 'iteritems'
```

Debe emplearse items() para generar código híbrido. No obstante, tener en cuenta que los objetos retornados se verán de forma diferente en ambas versiones:

Python 3:

```
>>> a.items()
dict_items([('a', 1), ('b', 9)])
```

Python 2:

```
>>> a.items()
[('a', 1), ('b', 9)]
```

Sin embargo, se itera igual en las dos:

```
for tupla in a.items():
    tupla

('a', 1)
('b', 9)
```

EJERCICIOS

1) Escribir el siguiente texto en un archivo:

```
Nombres: John Juan
Apellidos: Perez Doe
Nacionalidad: colombiana
Año de nacimiento: 1912
Edad: 75
```

2) Leer el archivo para obtener año de nacimiento y edad

3) Con los datos obtenidos en el paso 2, calcular el año de fallecimiento y agregar el resultado al archivo generado en el paso 1

4) Leer nuevamente el archivo modificado, y parsear el contenido a fin de obtener un diccionario, cuyas claves sean los datos ubicados a la izquierda de los dos puntos (y los valores, cada uno de los datos ubicados a la derecha)

UNIDAD 3: GENERACIÓN DE REGISTROS DE SISTEMA

Si se necesita que un programa o script del sistema guarde un registro, puede emplearse el módulo `logging`.

El módulo `logging` provee cinco niveles de registros:

NIVEL		Utilizado generalmente cuando se desea...
DEBUG	10	**monitorizar el funcionamiento de un programa** permitiendo depurar un programa durante su ejecución normal, a fin de obtener la información deseada para efectuar un diagnóstico determinado.
INFO	20	**registrar eventos afirmativos** es decir, mantener un registro detallado, de ciertas acciones ejecutadas en la aplicación, de forma satisfactoria.
WARNING	30	**emitir una alerta sobre un evento determinado** permitiendo grabar en el archivo de registros, información que, sin representar un error o momento crítico de fallo, podría ser indicativa de un posible fallo, error, o acción no deseada. Generalmente útil en advertencias de seguridad.
ERROR	40	**registrar un error** cuando el programa no logra llevar a cabo una acción esperada
CRITICAL	50	**registrar un error que frene la ejecución normal del programa.** Suele emplearse cuando errores fatales son capturados, y la ejecución normal del programa se ve impedida.

El **nivel por defecto** es **`WARNING`**, por lo que si se desea grabar (o mostrar) registros de niveles inferiores como **`INFO`** o **`DEBUG`**, deberá modificarse el nivel de registro por defecto.

Los registros pueden mostrarse en pantalla o grabarse en un archivo, tal y como se hará en lo sucesivo.

PRINCIPALES ELEMENTOS DEL MÓDULO LOGGING

Constantes: representan los distintos niveles de registro. Estas son:

```
INFO, DEBUG, WARNING, ERROR, CRITICAL
```

Clase basicConfig: utilizada para inicializar un registro, configurar el nivel de registro por defecto, y opcionalmente, establecer la ruta del archivo de registro y el modo de escritura.

```
from logging import basicConfig, INFO
basicConfig(
    filename='/var/log/programa.log',
    filemode='a',
    level=INFO
)
```

Los parámetros compartidos en ambas ramas del lenguaje, para `basicConfig` con los siguientes:

- **filename:** ruta del archivo
- **filemode:** modo de apertura (comúnmente 'a' [append, valor por defecto] o 'w' [escritura])
- **format:** establece el formato en el que se generarán los registros
- **datefmt:** formato de fecha y hora que se utilizará en los registros
- **level:** nivel de registro (cualquiera de las 5 constantes)
- **stream** (esta opción no será abarcada en el curso)

Algunas de las variables admitidas como parte del valor del parámetro **`format`**, son las siguientes:

```
asctime         %(asctime)s
created         %(created)f
filename        %(filename)s
funcName        %(funcName)s
levelname       %(levelname)s
levelno         %(levelno)s
lineno          %(lineno)d
module          %(module)s
msecs           %(msecs)d
message         %(message)s
name            %(name)s
pathname        %(pathname)s
```

```
process           %(process)d
processName       %(processName)s
relativeCreated   %(relativeCreated)d
thread            %(thread)d
threadName        %(threadName)s
```

Para una descripción detallada, ver la sección «LogRecords Attributes» en la documentación oficial de Python 2.7[1] y Python 3.6[2]. Ambas ramas conservan las mismas variables.

```
'[%(asctime)s] [%(levelname)s] [pid %(process)d] MYAPP MyErrorLevel Alert: %
(message)s'
```

El ejemplo anterior, producirá un registro similar al siguiente:

```
[2018-04-20 00:34:42,803] [WARNING] [pid 12318] MYAPP MyErrorLevel Alert:
Posible violación de seguridad
```

Para establecer el formato que tendrá la fecha, mediante el parámetro **`datefmt`** se pueden emplear las siguientes directivas:

DIRECTIVA	SIGNIFICADO
%A	Nombre del día de la semana
%b	Abreviatura del nombre del mes
%B	Nombre del mes completo
%d	Número del día del mes [01,31]
%H	Hora en formato de 24 horas [00,23]
%I	Hora en formato de 12 horas [00,12]
%m	Número del mes [01,12].
%M	Minutos [00,59].
%p	AM / PM.
%S	Segundos [00,59].
%w	Número del día de la semana [0,6]
%y	Año en formato YY [00,99]

1 https://docs.python.org/2.7/library/logging.html#logrecord-attributes
2 https://docs.python.org/3.6/library/logging.html#logrecord-attributes

DIRECTIVA	SIGNIFICADO
%Y	Año
%Z	Zona horaria

1. Tabla obtenida de la documentación oficial de Python:
https://docs.python.org/3.6/library/time.html#time.strftime

Funciones de registro: utilizadas para mostrar o grabar los diferentes mensajes de registro. Estas son:

```
info(), debug(), warning(), error(), critical()
```

A estas funciones, se le debe pasar como parámetro, el mensaje que se desea almacenar en el registro:

```
funcion("mensaje a grabar")
```

También es posible emplear variables como parte del mensaje, utilizando modificadores formato en la cadena, y pasando las variables como argumentos:

```
funcion("Mensaje %s %i", variable_string, variable_entero)
```

EJERCICIO DE GENERACIÓN DE REGISTROS

Replicar el siguiente código y ejecutarlo repetidas veces con modificaciones, tanto de configuración, como de niveles de registro y mensajes:

```
#-*- coding: utf-8 -*-
from logging import basicConfig, error, info, INFO
from sys import argv

basicConfig(
    filename='ejemplo_logging.log',
    filemode='a',
    level=INFO,
    format='[%(asctime)s] [%(levelname)s] [pid %(process)d] %(message)s',
    datefmt="%d/%m/%Y %H:%M"
)
```

```
try:
    with open(argv[1], "a") as f:
        f.write(argv[2])

    info("Agregado el texto %s al archivo %s", argv[2], argv[1])
except:
    error("Se produjo un error al intentar escribir en el archivo %s", argv[1])

try:
    with open("/var/log/foo.log", "a") as f:
        f.write("Mensaje de prueba")
except (Exception) as problema:
    error(problema)
```

OBTENCIÓN DE ARGUMENTOS POR LÍNEA DE COMANDOS CON ARGV

`argv`, una lista del módulo `system`, almacena los argumentos pasados al script por línea de comandos, siendo la ruta del archivo o nombre del ejecutable, el primer elemento de la lista.

CAPTURA BÁSICA DE EXCEPCIONES CON TRY Y EXCEPT

La estructura `try / except` permite capturar excepciones que de otro modo provocarían la finalización abrupta del script, cuando una excepción es lanzada.

Cuando una instrucción o algoritmo tiene la posibilidad de fallar (normalmente, cuando depende de valores obtenidos al vuelo), puede colarse el código, dentro de la estructura `try`, y utilizar excep para ejecutar una acción en caso de que el intento de ejecución de código del `try`, falle. Su sintaxis podría interpretarse como la siguiente:

```
intentar:
   ejecutar esto
si falla:
   hacer esto otro
```

Pasado a lenguaje Python:

```
try:
    # instrucción que puede fallar
except:
    # instrucción a ejecutar en caso de que el código del try, falle
```

El tipo de excepción lanzada, también es posible capturarlo:

```
try:
    # instrucción que puede fallar
except (TipoDeExcepción1):
    # instrucción a ejecutar en caso de que se produzca una excepción de
    # tipo TipoDeExcepcion1
except (TipoDeExcepción2):
    # instrucción a ejecutar en caso de que se produzca una excepción de
    # tipo TipoDeExcepcion2
```

También es admisible capturar más de un tipo de excepción de forma simultánea:

```
try:
    # instrucción que puede fallar
except (TipoDeExcepción1, TipoDeExcepción2):
    # instrucción a ejecutar en caso de que se produzca una excepción de
    # tipo TipoDeExcepcion1 o TipoDeExcepcion2
```

E incluso, puedo capturarse una descripción del error, aunque no se conozca el tipo de excepción:

```
try:
    # instrucción que puede fallar
except (Exception) as descripcion_del_problema:
    # instrucción a ejecutar en caso de que se produzca una excepción de
    # tipo TipoDeExcepcion1 o TipoDeExcepcion2
```

Los diferentes **tipos de excepciones**, pueden estudiarse en la documentación oficial de Python 2[3] y de Python 3[4]. No obstante, debe tenerse en cuenta que los tipos de excepciones difieren en ambas ramas del lenguaje. Estos aspectos son tratados en cursos de nivel avanzado. **Para un nivel inicial, se recomienda trabajar solo con `except`.**

3 https://docs.python.org/2/library/exceptions.html
4 https://docs.python.org/3.7/library/exceptions.html

UNIDAD 4: MANIPULACIÓN AVANZADA DE CADENAS DE TEXTO

Python provee de soporte nativo para búsquedas mediante expresiones regulares, de forma similar a Perl.

Una **expresión regular** es un patrón de caracteres de reconocimiento, que aplicado sobre una cadena de texto, permite encontrar fragmentos que coincidan con dicha expresión.

Para definir los patrones se utilizan **caracteres** de forma simbólica (es decir, que cada carácter posee un significado particular en el patrón). Por ejemplo, el patrón "^ho" significa «cadena que comienza por las letras ho», y "la$", significa «cadena que finaliza por las letras la». Mientras que el acento circunflejo ^ simboliza los comienzos de cadenas, el signo dólar, simboliza los finales. Los caracteres simbólicos se listan a continuación.

Caracteres de posición			
^	Inicio de cadena	**$**	Final de cadena
Cuantificadores			
?	Cero o uno	*	Cero o más
+	Uno o más	**{n}**	n veces
{n,}	n o más veces	**{,m}**	Entre 0 y n veces
{n,m}	Entre n y m veces		
Agrupamiento			
(...)	Grupo exacto	**[...]**	Caracteres opcionales y rangos
|	Operador lógico «or» (A|B)	-	Usado para expresar un rango [a-z]
Caracteres de formato			
****	Caracter de escape para expresar literales: \. (literal del carácter punto)	**\d**	Dígito[NOTA]
.	Cualquier carácter excepto el salto de línea	**\n**	Salto de línea
\s	Espacio en blanco[NOTA]	**\w**	Palabra[NOTA]

NOTA: En mayúsculas significa lo contrario. Por ejemplo, \S simboliza cualquier carácter que no sea un espacio en blanco.

EXPRESIONES REGULARES EN PYTHON

Para realizar búsquedas mediante expresiones regulares en Python, se utiliza el **módulo re**. La función **search** de este módulo, permite realizar una búsqueda mediante la sintaxis:

```
search(expresión, cadena)
```

Una búsqueda mediante la función search, en caso de encontrar al menos una coincidencia, retornará un objeto SRE_Match. Se accede a cada grupo de coincidencias mediante el método **group(índice)**.

```
from re import search
cadena = "hola mundo"
ser = search("a\sm", cadena)
ser.group(0)
'a m'
```

En la administración de sistemas GNU/Linux, el uso del constructor with para la apertura de archivos, combinado con métodos de del objeto string y expresiones regulares, se puede emplear en el análisis de registros (*logs*) del sistema.

Se toma como ejemplo el archivo de autenticación, /var/log/auth.log

Las siguientes líneas, representan un intento de autenticación fallida del usuario eugenia como root del sistema:

```
Nov 13 13:34:23 bella su[25375]: pam_unix(su:auth): authentication failure;
logname= uid=1000 euid=0 tty=/dev/pts/0 ruser=eugenia rhost=  user=root
Nov 13 13:34:25 bella su[25375]: pam_authenticate: Authentication failure
Nov 13 13:34:25 bella su[25375]: FAILED su for root by eugenia
```

La última línea puede utilizarse como patrón, para por ejemplo, obtener una lista de autenticaciones fallidas similares:

```
from re import search


with open("/var/log/auth.log", "r") as f:
    log = f.read()


regex = "(.)+: FAILED su for root by [a-z]+\n"
ser = search(regex, log)

>>> ser.group(0)
'Nov 13 13:34:25 bella su[25375]: FAILED su for root by eugenia\n'
```

En la expresión anterior:

(.)+ Indica cualquier carácter una o más veces. Esto coincidirá con la fecha del registro, comando e ID del proceso: **Nov 13 13:34:25 bella su[25375]**

La cadena que sigue, es un literal.

[a-z]+ Coincide con el nombre de usuario ya que indica «cualquier letra entre la a y la z, repetidas una o más veces: **eugenia**

\n El salto de línea coincidiría con el final del registro

El mismo sistema puede emplearse para analizar registros de servicios, sistema, etc., entre ellos, el de Apache, el syslog y otros.

EJERCICIO

¿Cuántas veces ha intentado autenticarse como root, desde su usuario habitual, y ha fallado? Modifique el último ejemplo para lograr obtener este resultado (no olvidar los temas pasados).

UNIDAD 5: CREACIÓN DE UN MENÚ DE OPCIONES CON ARGPARSE

En el *scripting*, puede resultar útil, dar al usuario un menú de opciones y hacer que el script, actúe según la opción elegida por el usuario. En el libro del curso de «Introducción a la Ciencia de Datos con Python», se muestra un truco para resolver un menú de forma simple e ingeniosa. En este curso, se verá cómo parsear argumentos, pasado al script, por línea de comendos, mediante el módulo **argparse**.

PASO 1: IMPORTACIÓN DEL MÓDULO

Se debe importar la clase ArgumentParser del módulo argparse:

```
from argparse import ArgumentParser
```

PASO 2: CONSTRUCCIÓN DE UN OBJETO ARGUMENTPARSER

Se construye un objeto `ArgumentParser` a fin de establecer cuáles serán los argumentos que el programa recibirá.

Los parámetros aceptados por el método constructor del objeto `ArgumentParser` (función `__init__`) son todos opcionales. Entre otros, admite los siguientes parámetros:

- **prog:** el nombre del programa (por defecto, toma el nombre del ejecutable)
- **description:** descripción del programa que se mostrará en la ayuda
- **epilog:** texto que se mostrará al final de la ayuda

```
#!/usr/bin/env python
from argparse import ArgumentParser

argp = ArgumentParser(
    description='Descripción breve del programa',
    epilog='Copyright 2018 Autor bajo licencia GPL v3.0'
)
```

PASO 3: AGREGADO DE ARGUMENTOS Y CONFIGURACIÓN

Para agregar un argumento puede emplearse el método **`add_argument`**.

Existen dos tipos de argumentos que pueden declararse:

- Argumentos posicionales: por defecto, todos aquellos que sean declarados con un nombre en vez de emplear una bandera
- Opciones (banderas / flags): todos aquellos que empleen el prefijo de opción -

De esta forma, un argumento definido como 'foo' será posicional, mientras que si se lo define como '-f' o '--foo', será una opción:

```
argp.add_argument('foo')    # argumento posicional
argp.add_argument('--foo')  # opción foo
argp.add_argument('-f')     # opción f
```

add_argumento puede recibir, por lo tanto, solo un nombre de argumento posicional, o una lista de banderas de opción (flags). En el siguiente ejemplo, si se ejecutara el programa sin pasar ningún argumento, el fallo se produciría por la ausencia del argumento posicional «directorio», pero no, por la ausencia de las opción -f o --foo.

```
argp.add_argument('directorio')   # Solo un nombre posicional
argp.add_argument('-f', '--foo')  # Una lista de banderas de opción
```

Configuración de argumentos

El método `add_argument`, además del nombre del argumento posicional u opción, puede recibir de forma no obligatoria, algunos parámetros que establecen la forma en la que el nombre de argumento o bandera de opción, será tratado. Todos estos parámetros opcionales, se definen a continuación:

Parámetro	Descripción	Valores posibles*	Valor por defecto
action	Acción a realizar con el parámetro	**store**: almacenar el valor **append**: agregarlo a una lista	store
nargs	Cantidad de valores admitidos	**?** : cero o uno * : cero o más + : uno o más valor entero	1
default	Valor por defecto para el argumento	cualquier valor es admisible	
type	Tipo de datos	cualquier tipo soportado por Python	None
choices	Lista de valores posibles	Una lista	None
required	Indica si el argumento es obligatorio	True o False	False
help	Texto de ayuda a mostrar para el argumento	Una string	None
metavar	El nombre del argumento que se empleará en la ayuda	Una string	el nombre del argumento o flag
dest	Nombre de la variable en la que será almacenado el argumento	Una string	El nombre del argumento o bandera

(*) Para una lista completa, remitirse a la documentación oficial o al artículo **«Shell Scripting: Análisis de argumentos por línea de comandos»** en el siguiente enlace: http://fileserver.laeci.org/Art%c3%adculos%20t%c3%a9cnicos/Python/ArgParse.pdf

```
argp.add_argument(
    'vocal',                    # Argumento posicional
    nargs='+',                  # Admite uno o más valores
    choices=['a', 'e', 'o'],    # Valores posibles
    metavar='VOCAL',            # Nombre de variable a mostrar en la ayuda
    help='Vocal abierta',       # Texto a mostrar como ayuda
)
```

Siguiendo el ejemplo anterior, si el programa se ejecutara sin argumentos:

```
usuario@host:~$ ./ejemplo.py
```

daría el siguiente error:

```
usage: ejemplo [-h] VOCAL [VOCAL ...]
curl: ejemplo: the following arguments are required: VOCAL
```

Y si se ejecutase con la bandera de opción -h:

```
usuario@host:~$ ./ejemplo.py -h
```

arrojaría la siguiente ayuda:

```
usage: ejemplo [-h] VOCAL [VOCAL ...]

Descripción del programa

positional arguments:
  VOCAL           Vocal abierta

optional arguments:
  -h, --help      show this help message and exit
```

PASO 4: GENERACIÓN DEL ANÁLISIS (PARSING) DE ARGUMENTOS

```
argumentos = argp.parse_args()
```

El método `parse_args` es el encargado de generar un objeto cuyas propiedades serán los argumentos recibidos por línea de comandos. A cada argumento se accederá mediante la sintaxis:

```
objeto_generado.nombre_del_argumento
```

Por ejemplo:

```
argumentos.foo
```

EJERCICIO

Para comprender mejor el funcionamiento de argparse, se propone el siguiente ejemplo, el cual, se recomienda replicar a modo de ejercicio y ejecutarlo repetidas veces con modificaciones.

El siguiente ejemplo, se trata de un menú basado en el programa `curl`.

Ejecutar `man curl` si se desea poner en contexto el ejemplo.

A modo de ejercicio, se recomienda intentar imprimir los valores obtenidos mediante `argumentos.parametro`.

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

from argparse import ArgumentParser

descripcion_del_programa = "{}, {}".format(
    "Herramienta para transferir datos desde o hacia un servidor",
    "utilizando uno de los protocolos compatibles"
)

argp = ArgumentParser(
    prog='curl',
    description=descripcion_del_programa,
)

argp.add_argument(
    '-H', '--header',     # Banderas
    action='append',      # Lista de valores
    nargs='+',            # Admite uno o más valores
    type=str,             # Convertir los valores a string
    metavar='LINE',       # Nombre de opción a mostrar en la ayuda
    help='Cabecera adicional a incluir en la solicitud HTTP a enviar'
)

argp.add_argument(
    '-d', '--data',
    action='append',
    nargs='+',
    type=str,
    help='envía los datos especificados en una solicitud post, al servidor http'
)

argp.add_argument(
    'url',
    type=str,
    metavar='URL',
    help='URL a la cual realizar la solicitud. La sintaxis es dependiente del
protocolo (RFC 3986)'
)

argumentos = argp.parse_args()
```

Observaciones:

1. Recordar del curso de «Introducción al Lenguaje Python», que la especificación de la codificación de caracteres UTF-8, si bien en Python 3 no es necesaria, se utiliza a fin de hacerlo compatible también con Python 2.

2. Cuando se realice el ejercicio, utilizar el hack de retrocompatiblidad para imprimir, aprendido en el curso de «Introducción al Lenguaje Python».

UNIDAD 6: MÓDULOS DEL SISTEMA (OS, SYS Y SUBPROCESS)

Los módulos os y subprocess permiten manejar **funcionalidades del sistema operativo**, y **procesos del sistema**, respectivamente, mientras que el módulo sys, provee acceso a **variables del intérprete** del lenguaje.

A diferencia de **`shutil`**, un módulo de Python que permite manejar archivos a alto nivel, el módulo os provee funciones que operan a bajo nivel. Por este motivo, no se abarca el módulo `shutil`, sino, `os`.

EL MÓDULO OS

Este módulo permite operar a bajo nivel con funcionalidades del sistema operativo. Algunas de ellas, se listan a continuación.

Acción	Comando GNU/Linux	Método
ACCESO A ARCHIVOS Y DIRECTORIOS		
Obtener directorio actual	`pwd`	**`getcwd`**`()`
Cambiar el directorio de trabajo	`cd ruta`	**`chdir`**`(path)`
Mover el directorio de trabajo a la raíz	`cd`	**`chroot`**`()`
Modificar permisos	`chmod`	**`chmod`**`(path, permisos)`
Cambiar el propietario de un archivo o directorio	`chown`	**`chown`**`(path, permisos)`
Crear un directorio	`mkdir`	**`mkdir`**`(path[, modo])`
Crear directorios recursivamente	`mkdir -p`	**`mkdirs`**`(path[, modo])`
Eliminar un archivo	`rm`	**`remove`**`(path)`
Eliminar un directorio	`rmdir`	**`rmdir`**`(path)`
Renombrar un archivo	`mv`	**`rename`**`(actual, nuevo)`
Crear un enlace simbólico	`ln -s`	**`symlink`**`(origen, destino)`

Acción	Comando GNU/Linux	Método
Establecer máscara de creación de ficheros	`umask`	**umask**(máscara)
Obtener listado de archivos y directorios	`ls -a`	**listdir**(path)
Obtener el estado de un fichero	`stat`	**stat**(path)
EVALUACIÓN DE ARCHIVOS Y DIRECTORIOS (Módulo `os.path`)		
Obtener ruta absoluta	-	path.**abspath**(path)
Obtener directorio base	-	path.**basename**(path)
Saber si un directorio existe	-	path.**exists**(path)
Conocer último acceso a un directorio	-	path.**getatime**(path)
Conocer tamaño del directorio	-	path.**getsize**(path)
Saber si una ruta es:	-	
absoluta	-	path.**isabs**(path)
un archivo	-	path.**isfile**(path)
un directorio	-	path.**isdir**(path)
un enlace simbólico	-	path.**islink**(path)
un punto de montaje	-	path.**ismount**(path)
FUNCIONALIDADES DEL SISTEMA OPERATIVO		
Obtener el valor de una variable de entorno	`$VARIABLE`	**getenv**(variable)
Obtener los datos del sistema operativo	`uname -a`	**uname**()
Obtener UID	`id -u`	**getuid**()
Obtener ID del proceso	`pgrep`	**getpid**()
Crear variable de entorno (del sistema)	`export $VARIABLE`	**putenv**(variable, valor)
Forzar la escritura del caché al disco	`sync`	**sync**()
Matar un proceso	`kill`	**kill**(pid, señal)

Para una definición completa y detallada, referirse a la documentación oficial del lenguaje: https://docs.python.org/[2|3]/library/os.html

VARIABLES DE ENTORNO: OS.ENVIRON

Environ es un diccionario del módulo os que provee variables de entorno. Esto significa que las variables disponibles en `environ` (como claves del diccionario), varían de acuerdo al entorno en el que se esté ejecutando el programa o *script*. Por ejemplo, no serán las mismas variables si se ejecuta un *script* en la *shell*, que si se llama a `environ` desde una aplicación Web. Mientras que en el primer caso, las variables disponibles serán las de la *shell* de GNU Bash, en el segundo (y suponiendo que el servidor sea Apache), las de Apache.

Para ver las variables disponibles en la *shell*, ejecutar en ambas versiones del lenguaje:

```
from os import environ

for variable, valor in environ.items():
    variable, valor
```

EJECUCIÓN DE COMANDOS DEL SISTEMA MEDIANTE SUBPROCESS Y SHLEX

A través de la clase **Popen** del módulo `subprocess`, es posible ejecutar comandos directamente sobre el sistema operativo, y manipular tanto la E/S estándar como los errores. La función `split` del módulo `shlex`, puede emplearse como complemento de `Popen`, para el *parsing* de cadenas de texto como lista de comandos y argumentos.

La clase `Popen` (*Process open - proceso abierto*), abre un nuevo proceso en el sistema, permitiendo emplear tuberías para manejar la E/S estándar y los errores:

```
from subprocess import Popen, PIPE
proceso = Popen(<comando/argumentos>, stdout=PIPE, stderr=PIPE)
```

La captura y manejo de la E/S estándar y los errores, se trata más adelante.

El primer argumento pasado a Popen, debe ser una lista. Dicha lista, deberá contener cada uno de los comandos, listas de opciones (banderas) y cada uno de los argumentos, como un elemento. Esto significa que para ejecutar el comando:

```
ls -la /home/usuario/Documentos
```

los elementos serán 3 ya que existe:

- 1 argumento (ls)
- 1 lista de opciones (-la)
- 1 argumento

Total: 3 elementos

```
lista = ['ls', '-la', '/home/usuario/Documentos']
proceso = Popen(lista)
```

Sin embargo, la instrucción completa podría escribirse en una cadena de texto, y emplear la función split del módulo shlex, para generar la lista necesaria para Popen:

```
from shlex import split
from subprocess import Popen

comando = 'ls -la /home/usuario/Documentos'
proceso = Popen(split(comando))
```

CAPTURAR LA SALIDA ESTÁNDAR Y LOS ERRORES

Para capturar la salida estándar y los errores, puede emplearse una tubería:

```
from shlex import split
from subprocess import Popen, PIPE

comando = 'ls -la /home/usuario/Documentos'
proceso = Popen(split(comando), stdout=PIPE, stderr=PIPE)
salida = proceso.stdout.read()
```

```
errores = proceso.stderr.read()

if not errores:
    acción a realizar si no hubo errores
else:
    acción a realizar si hubo errores
```

El siguiente es un ejemplo del código anterior, ejecutado en la *shell* de Python y con errores:

```
>>> from shlex import split
>>> from subprocess import Popen, PIPE
>>>
>>> comando = 'ls -la /home/usuario/Documentos'
>>> proceso = Popen(split(comando), stdout=PIPE, stderr=PIPE)
>>> salida = proceso.stdout.read()
>>> errores = proceso.stderr.read()
>>> errores
"ls: no se puede acceder a '/home/usuario/Documentos': No existe el fichero o el
directorio\n"
>>> salida
''
```

Y a continuación, el mismo ejemplo pero con una ruta accesible:

```
>>> comando = 'ls -la /home/eugenia/borrador/python'
>>> proceso = Popen(split(comando), stdout=PIPE, stderr=PIPE)
>>> salida = proceso.stdout.read()
>>> errores = proceso.stderr.read()
>>> errores
''
>>> salida
'total 20\ndrwxr-xr-x  3 eugenia eugenia  4096 nov 16 21:17 .\ndrwxr-xr-x 43
eugenia eugenia 12288 nov 16 20:03 ..\ndrwxr-xr-x  5 eugenia eugenia  4096 nov
16 20:06 a\n'
```

EMPLEAR LA SALIDA DE UN COMANDO COMO ENTRADA DE OTRO

En la línea de comandos, se emplea la salida de un comando como entrada de otro, al utilizar el símbolo | (pipe). El siguiente es ejemplo de ello:

```
ls -la /home/eugenia/borrador/python | grep '20:06'
```

En el ejemplo anterior, se utiliza la salida del comando *ls* como entrada del comando *grep*.

Al emplear Popen, la salida de un comando se encuentra disponible en:

```
proceso_creado.stdout
```

Esta salida, puede utilizarse como valor del argumento stdin del segundo proceso creado con Popen:

```
from shlex import split
from subprocess import Popen, PIPE

# Comandos necesarios
ls_command = "ls -la /home/eugenia/borrador/python"
grep_command = "grep '20:06'"

# Procesos
ls_process = Popen(split(ls_command), stdout=PIPE, stderr=PIPE)

grep_process = Popen(
    split(grep_command),
    stdin=ls_process.stdout,  # Salida del proceso anterior como entrada
    stdout=PIPE,
    stderr=PIPE
)
```

A continuación, la ejecución del *script* anterior, y la salida del *stdout*, en la *shell* de Python:

```
>>> from shlex import split
>>> from subprocess import Popen, PIPE
>>> ls_command = "ls -la /home/eugenia/borrador/python"
>>> grep_command = "grep '20:06'"
>>> ls_process = Popen(split(ls_command), stdout=PIPE, stderr=PIPE)
>>> grep_process = Popen(
...     split(grep_command),
...     stdin=ls_process.stdout,
...     stdout=PIPE,
...     stderr=PIPE
... )
>>>
>>> grep_process.stdout.read()
'drwxr-xr-x  5 eugenia eugenia  4096 nov 16 20:06 a\n'
```

VARIABLES Y FUNCIONES DEL MÓDULO SYS

Algunas de las variables disponibles en este módulo, son las siguientes:

Variable	Descripción

sys.argv	Retorna una lista con todos los argumentos pasados por línea de comandos, incluyendo como primero elemento, el del archivo ejecutado.
	
	Al ejecutar:
	`python modulo.py arg1 arg2`
	
	`sys.arg` retornará una lista:
	`['modulo.py', 'arg1', 'arg2']`

sys.executable	Retorna el path absoluto del binario ejecutable del intérprete de Python
sys.path	Retorna una lista con las rutas empleadas por el intérprete para buscar los archivos
sys.platform	Retorna la plataforma sobre la cuál se está ejecutando el intérprete
sys.version	Retorna el número de versión de Python con información adicional

Entre las funciones del módulo, **`exit()`** se emplea para finalizar un programa o *script* de forma abrupta:

```
from shlex import split
from subprocess import Popen, PIPE

ls_command = "ls -la /home/no-existe"
grep_command = "grep '20:06'"

ls_process = Popen(split(ls_command), stdout=PIPE, stderr=PIPE)

if ls_process.stderr.read():
    exit("Terminación abrupta tras error en comando ls")

grep_process = Popen(
    split(grep_command),
    stdin=ls_process.stdout,
    stdout=PIPE,
    stderr=PIPE
)
```

El resultado de la ejecución anterior en una shell:

```
>>> from shlex import split
>>> from subprocess import Popen, PIPE
>>>
>>> ls_command = "ls -la /home/no-existe"
>>> grep_command = "grep '20:06'"
>>>
>>> ls_process = Popen(split(ls_command), stdout=PIPE, stderr=PIPE)
```

```
>>>
>>> if ls_process.stderr.read():
...     exit("Terminación abrupta tras error en comando ls")
...
Terminación abrupta tras error en comando ls
eugenia@bella:~/borrador/python$
```

Se debe notar que:

1. El mensaje pasado a la función exit() es opcional.
2. La función exit() puede recibir un entero representativo del motivo de salida (cero es el valor por defecto, e indica una salida normal)
3. La función exit() del módulo sys tiene un propósito similar a la constante incorporada exit, sin embargo, ambos elementos responden de manera no exactamente idénticas:

```
eugenia@bella:~/borrador/python$ python
...
Type "help", "copyright", "credits" or "license" for more information.
>>> print exit  # constante inforporada
Use exit() or Ctrl-D (i.e. EOF) to exit
>>> import sys
>>> print sys.exit
<built-in function exit>
>>> print sys.exit()
eugenia@bella:~/borrador/python$
```

EJERCICIOS

1) Repasar los temas de esta unidad, y empleando algunas de las funciones y métodos de las librerías `os` y `subprocess`, escribir una herramienta que reciba como parámetro las rutas de dos directorios:

a) Un origen para crear de él una copia de respaldo comprimida en formato tar.xz (`tar -cjvf /destino/tar.xz /origen`)

b) Un destino para almacenar dicha copia

2) Guardar la herramienta de forma tal que pueda ser ejecutada como un comando (ver el primer tema del curso anterior, «Introducción al Lenguaje Python»)

3) Manualmente, agregue una tarea al `crontab` para así tener siempre, una copia de respaldo de los archivos que le interesen, mediante una herramienta hecha por sus propias manos ☺

UNIDAD 7: CONEXIONES REMOTAS (HTTP, FTP Y SSH)

Python provee dos librerías, **`http`** y **`ftplib`**, para efectuar conexiones mediante los protocolos HTTP/HTTPS y FTP, respectivamente. Sin embargo, para realizar conexiones mediante el protocolo SSH, se empleará la librería **Paramiko**[5], creada por Robey Pointer[6].

CONEXIONES REMOTAS VÍA HTTP Y HTTPS

Pueden efectuarse con el módulo **`client`** de la librería http de Python.

Para crear la **conexión** se utilizan las clases `HTTPConnection` y `HTTPSConnection`:

```
from http.client import HTTPConnection
http = HTTPConnection('algun.host.com', port=80, timeout=10)
```

El número de puerto y el tiempo de espera, son dos parámetros opcionales, y son admitidos, junto al parámetro posicional `host`, por ambas clases.

Las **solicitudes** se realizan mediante el método **`request`** que requiere de dos parámetros posicionales:

1. El método HTTP
2. El recurso HTTP

```
http.request("GET", "/foo/bar")
```

Adicionalmente, admite otros parámetros como `headers` (un diccionario con campos de cabecera) y `body` (una cadena de texto), útiles sobre todo, para

5 http://www.paramiko.org
6 https://robey.lag.net/

peticiones que requieren el envío de información, como por ejemplo, **envío de datos por POST**:

```
parametros = "nombre=Juan&apellido=Perez"
cabeceras = {"Content-Type": "application/x-www-form-urlencoded"}
http.request("POST", "/foo/bar", headers=cabeceras, body=parametros)
```

La **respuesta** recibida, se obtiene mediante el método **`getresponse`**, que retorna un objeto `HTTPResponse`, el cual, entre sus propiedades, posee `status` (el código de respuesta HTTP) y `reason` (la descripción de la respuesta), y entre sus métodos, `read`, que retorna el cuerpo de la respuesta:

```
respuesta = http.getresponse()
codigo = respuesta.status
descripcion = respuesta.reason
body = respuesta.read()
```

El **cierre** de una conexión HTTP, se efectúa mediante el método **`close`**:

```
http.close()
```

El siguiente ejemplo, realiza una petición POST a un *host* local que como respuesta, imprime el mensaje «Gracias <nombres>!»:

```
>>> from http.client import HTTPConnection
>>> http = HTTPConnection('juanproyecto.local', port=80, timeout=30)
>>> parametros = "nombre=Juan&apellido=Perez"
>>> cabeceras = {"Content-Type": "application/x-www-form-urlencoded"}
>>> http.request("POST", "/foo/bar", headers=cabeceras, body=parametros)
>>> respuesta = http.getresponse()
>>> codigo = respuesta.status
>>> descripcion = respuesta.reason
>>> body = respuesta.read()
>>> body
'Gracias Juan'
>>> codigo
200
>>> descripcion
'OK'
```

CONEXIONES REMOTAS VÍA FTP

La librería **`ftplib`** permite conexiones a través del protocolo FTP.

Para **crear una instancia FTP**, dispone de las clases **`FTP`** y **`FTP_TLS`**, la segunda, con soporte del protocolo `TLS` (evolución de SSL). Si bien estas clases, como parámetros opcionales, el host, el usuario y la clave (entre otros), a fin de obtener un mejor control sobre las operaciones, estos datos serán enviado mediante los métodos `connect` y `login`, que serán abarcados adelante.

```
from ftplib import FTP
ftp = FTP()
```

Para **abrir la conexión** se emplea el método **`connect`**, que como parámetros admite, entre otros, el host y el puerto:

```
ftp.connect('algunhost.com', 21)
```

De ser necesario establecer el **modo pasivo**, se dispone del método **`set_pasv`**:

```
ftp.set_pasv(True)
```

La **autenticación** se realiza mediante el método **`login`**, quien recibe por parámetros, usuario y contraseña, respectivamente:

```
ftp.login('algunusuario', 'clave')
```

Para **cerrar** una conexión puede utilizarse el método **`quit`**. Esto cierra la conexión de ambos lados siempre que el servidor lo soporte y no retorne un

error. En caso de que así sea, se llamará al método **close**, el cual cierra la conexión unilateralmente.

```
ftp.quit()
```

Otros **métodos** disponibles se citan a continuación:

Acción	Método
Directorios	
Listar directorios	`dir()` `dir('ruta/a/listar')`
Crear un directorio	`mkd('ruta/a/nuevo-dir')`
Moverse a un directorio	`cwd('ruta/a/algun-dir')`
Eliminar un directorio	`rmd('ruta/a/dir-a-borrar')`
Obtener directorio actual	`pwd()`
Archivos	

Recuperar un archivo remoto	`retrbinary(`
	`'RETR origen',`
	`open('/ruta/destino', 'w').write)`

Enviar un archivo local	`storbinary(`
	`'STOR destino/remoto.txt',`
	`open('/origen/local.txt', 'r'))`

Eliminar un archivo	`delete('archivo/a/eliminar')`
Renombrar (mover) un archivo	`rename('origen', 'destino')`

Cuando los modos «**w**» y «**r**» aparecen en negritas, significa que para archivos binarios debe agregarse «**b**» al modo.

SOLICITANDO LA CONTRASEÑA CON GETPASS

La librería **getpass** permite solicitar mediante un input, una contraseña al estilo GNU/Linux, para evitar tener que trabajar con la contraseña en crudo en el código fuente:

```
from getpass import getpass
clave = getpass('Ingresar clave: ')
```

La función getpass puede utilizarse en forma conjunta con el método login, y así se evita escribir la clave en crudo dentro del código fuente:

```
from ftplib import FTP
from getpass import getpass

ftp = FTP()
ftp.connect('algunhost.com', 21)
ftp.login('algunusuario', getpass('Clave FTP: '))
```

CONEXIONES SSH CON PARAMIKO

La librería `paramiko` debe instalarse de forma adicional, ya que no forma parte de las librerías de Python, ni está mantenida por Python. Se trata de una librería de terceros, que puede instalarse a través de PyPI (el gestor de paquetes de Python). En Debian 9, también es posible instalarla desde apt (tanto para Python 2 como 3). Sin embargo, se instalará mediante PyPI, ya que es la opción recomendada por el fabricante[7].

REQUISITOS PREVIOS

Para poder instalar un paquete desde **PyPI**, se necesita la herramienta **`pip`** de Python. En Debian 9, el gestor de paquetes de Python se instala mediante *apt*:

```
apt install python-pip   # para Python 2, y
apt install python3-pip  # para Python 3
```

Una vez instalado el gestor de paquetes de Python, las instalaciones para Python 2 y para Python 3, se manejarán de forma independiente, por lo cual, habrá que instalar **Paramiko** en ambas versiones. Esto se hará como **`root`**:

```
pip install paramiko   # para Python 2
pip3 install paramiko  # Para Python 3
```

7 http://www.paramiko.org/installing.html

USO DE PARAMIKO

Una conexión SSH se inicializa con la **creación de un objeto `SSHClient`**:

```
from paramiko import SSHClient, AutoAddPolicy

ssh = SSHClient()
```

Al igual que con la librería FTP, tanto la conexión como la autenticación, se realizarán de forma separada (y no al construir el objeto), a fin de tener un mayor control sobre las mismas.

Para la **autenticación mediante llave** pública (en vez de uso de contraseñas), se empleará **`set_missing_host_key_policy`**, a fin de localizar las llaves y facilitar el intercambio de las mismas:

```
ssh.set_missing_host_key_policy(AutoAddPolicy())
```

Normalmente, el uso de este método no debería ser necesario, y bastaría con emplear **`load_system_host_keys`**:

```
ssh.load_system_host_keys()
```

Sin embargo, utilizar set_missing... resuelve el problema anticipadamente, evitando algoritmos complejos para captura y tratamiento de errores.

La **conexión** al servidor se hará mediante el método **`connect`**, quien recibe como parámetros, entre otros, el host o IP del servidor, el puerto de conexión y el nombre de usuario:

```
ssh.connect('123.45.67.89', 22, 'usuario')
```

Cuando se requiera **autenticación por contraseña**, como cuarto parámetro del método **connect**, puede pasarse la clave:

```
ssh.connect('123.45.67.89', 22, 'usuario', 'clave')
```

La **ejecución de comandos** en el servidor, se realiza mediante el método **exec_command** a quien se le debe pasar una cadena con la instrucción que se desea ejecutar. Este método retorna tres objetos, de E/S estándar y errores:

```
entrada, salida, error = ssh.exec_command('ls -la')
```

Los objetos de salida y error, pueden ser leídos mediante el método read:

```
salida.read()
error.read()
```

Mientras que la entrada, puede ser escrita mediante write:

```
entrada.write('entrada que espera el comando\n')
salida.read()
```

Finalmente, para **cerrar** la conexión, se utiliza el método **close**:

```
ssh.close()
```

EJERCICIO INTEGRADOR

1) Escribir un *script* que como primer parámetro reciba una URL completa y como segundo, una ruta local. Que lea la URL recibida y guarde la salida en la ruta indicada como primer parámetro.

2) Escribir un *script* que permita enviar archivos por FTP a un servidor local o remoto.

3) Escribir un *script* que se conecte por SSH a un servidor, obtenga los datos del sistema (uname -a), y los muestre en pantalla luego de cerrar la conexión.

UNIDAD 8: LIBRERÍAS PARA EL MANEJO AVANZADO DE ARCHIVOS, EN SISTEMAS GNU/LINUX

COMPRESIÓN Y DESCOMPRESIÓN DE ARCHIVOS CON LAS LIBRERÍAS TARFILE Y ZIPFILE

La librería `tarfile` puede utilizarse para leer, comprimir y descomprimir archivos `.tar, .tar.gz. tar.bz2` y `tar.xz`, mientras que la librería `zipfile`, se utiliza para los archivos `.zip`

LA LIBRERÍA TARFILE

Bien sea para leer un archivo comprimido, o bien para comprimir o descomprimir, un objeto `TarFile`, se crea mediante la función open del módulo.

A diferencia de una apertura estándar, los modos lectura y escritura, se acompañan del formato deseado, mediante la sintaxis **`<modo>:<formato>`**, donde `<modo>` puede ser `r` (lectura) o `w` (escritura), y `<formato>`, **`gz`** (*gzip*), **`bz2`** (*bzip2*) o, solo en Python 3, **`xz`** (*lzma*).

Modo de apertura	Comando tar
`[r|w]:gz`	`tar -[c|x]z`
`[r|w]:bz2`	`tar -[c|x]j`
`[r|w]:xz`	*`tar -[c|x]J`*

2. Tabla: Modos de apertura y equivalencias con el comando tar.
El formato lzma (xz) solo está disponible en la rama 3 del lenguaje.

Descomprimir archivos

```
from tarfile import open as tar_open

with tar_open("origen.tar.bz2", "r:bz2") as tar:
    tar.extractall('carpeta/destino')
```

Comprimir archivos

```
from tarfile import open as tar_open

with tar_open("carpeta/destino.tar.gz", "w:gz") as tar:
    tar.add('foo.txt')
    tar.add('bar.txt')
    tar.add('baz.txt')
```

Observaciones generales sobre el código

Se utiliza un alias, para que el método open de la librería `tarfile`, no sobrescriba la función incorporada open. Se emplea la estructura `with`, para no utilizar el método `close`.

Observaciones de seguridad

No deben descomprimirse archivos sin verificar el nombre de los mismos. Un nombre de archivo, podría contener una `/` o `..`, que provocarían que los archivos se almacenasen en un directorio no esperado.

Observaciones de compatibilidad entre versiones

El formato *lzma* solo está disponible en la rama 3 del lenguaje. Para que un *script* o herramienta sea compatible con ambas versiones, la única opción es utilizar los formatos *gzip* o *bzip2*.

LA LIBRERÍA ZIPFILE

La extracción y compresión de archivos *zip* se realiza de la siguiente forma:

```
from zipfile import ZipFile

# Escritura de archivos zip
with ZipFile('carpeta/destino.zip', 'w') as z:
    z.write('foo.txt')
    z.write('bar.txt')
    z.write('baz.txt')
```

```
# Lectura de archivos zip
with ZipFile('carpeta/origen.zip', 'r') as z:
    z.extractall('carpeta/destino', pwd='contraseñaSecreta')
    # el parámetro pwd (contraseña) es opcional
```

Para los archivos *zip*, aplican las mismas observaciones de seguridad que para los archivos *tar*.

MANEJO DE ARCHIVOS TEMPORALES CON LA LIBRERÍA TEMPFILE

Cuando sea necesario que un *script*, guarde temporalmente archivos, no es una buena práctica que el mismo *script* los guarde y luego los elimine, ni que intente escribir directamente en el directorio `/tmp`. Para este caso, se debe emplear la librería `tempfile`.

LECTOESCRITURA DE ARCHIVOS TEMPORALES

Cuando se crean objetos de archivos temporales mediante la clase **`TemporaryFile`** del módulo `tempfile`, los mismos se crean y destruyen en tiempo de ejecución. La destrucción se lleva a cabo al cerrar el archivo. Esto implica que si se trabaja con la estructura `with`, al finalizar dicha estructura, el archivo se habrá eliminado.

```
from tempfile import TemporaryFile

with TemporaryFile() as tmp:
    # aquí el archivo existe

# Aquí el archivo ya no existe
```

Todo archivo temporal escrito, para ser escrito, requiere que el **contenido** se pase como un **objeto tipo bytes** (y no una cadena). Este requerimiento es exigencia de Python 3, sin embargo, en Python 2 está perfectamente

soportado. Para que una cadena sea convertido a bytes, solo basta con especificar su tipo:

```
from tempfile import TemporaryFile

with TemporaryFile() as tmp:
    tmp.write(b"Cadena de texto que será pasada a bytes")
```

Finalmente, se debe tener en cuenta que una vez escrito, el cursor estará al final del archivo, por lo que si se lo quiere leer, retornará una cadena nula. Por lo tanto, habrá que mover el cursor al byte 0 a fin de poder leerlo:

```
from tempfile import TemporaryFile

with TemporaryFile() as tmp:
    tmp.write(b"Cadena de texto que será pasada a bytes")
    # … acciones intermedias
    tmp.seek(0)  # Se mueve el cursor al byte 0
    contenido = tmp.read()
```

Observaciones importantes: es necesario aclarar que los archivos temporales creados con `TemporaryFile`, no son archivos persistentes en memoria, sino en disco. De hecho, se almacenan en el directorio temporal del sistema, independientemente de la plataforma. Es posible conocer este directorio invocando a la función `gettempdir()`:

```
from tempfile import TemporaryFile, gettempdir

with TemporaryFile() as tmp:
    tmp.write(b"Cadena de texto")
    tmp_dir = gettempdir()
```

BÚSQUEDA DE ARCHIVOS CON LAS LIBRERÍAS GLOB Y FNMATCH

Estas librerías permiten buscar archivos que coincidan con un patrón, con el mismo estilo empleado en sistemas **nix*. Mientras que el módulo `glob` busca

archivos que coincidan con un patrón, `fnmatch` verifica si un patrón coincide con el nombre de un archivo.

Símbolos interpretados

Símbolo	Significado
*	Cualquier coincidencia
?	Coincidencia con un único carácter
`[secuencia]`	Coincidencia con cualquier carácter de la secuencia
`[!secuencia]`	Coincidencia con cualquier carácter, excepto los de la secuencia

Uso de glob

```
>>> from glob import glob
>>> glob('*.txt')
['foo.txt', 'baz.txt', 'bar.txt']
>>> glob('*[!of].txt')
['baz.txt', 'bar.txt']
```

Uso de fnmatch con os.listdir

```
>>> from os import listdir
>>> from fnmatch import fnmatch
>>> for archivo in listdir('.'):
...     archivo, fnmatch(archivo, '*[!0-9].txt')
...
('3.r', False)
('foo.gif', False)
('.bar', False)
('carpeta', False)
('foo.txt', True)
('baz.txt', True)
('origen.tar.xz', False)
('.foo', False)
('2.r', False)
('bar.txt', True)
('.baz', False)
('origen.tar.bz2', False)
('a.r', False)
('1.r', False)
('origen.tar.gz', False)
```

EJERCICIO INTEGRADOR

Busque en su carpeta de Imágenes, mediante `glob` o `fnmatch` con `listdir`, archivos con extensión png, y vaya guardando mediante `TemporaryFile`, los nombres de dichos archivos, en un archivo temporal. Luego, cree un *tarball* (archivo `.tar.gz`) con todos los archivos encontrados, recuperando los nombres desde el archivo temporal guardado.

CERTIFÍCATE

Demuestra cuánto has aprendido!

Si llegaste al final del curso puedes obtener una triple certificación:

- Certificado de **asistencia** (emitido por la escuela **Eugenia Bahit**)
- Certificado de **aprovechamiento** (emitido por **CLA Instituto Linux**)
- Certificación **aprobación** (emitido por **LAECI - UK**)

Informáte con tu docente o visita la Web de certificaciones en http://python.eugeniabahit.com.

Si necesitas preparar tu examen, puedes inscribirte en el
Curso de Python para la Adminstración de Sistemas GNU/Linux, en la
Escuela de Informática Eugenia Bahit
https://python.eugeniabahit.com