Python Scripting: Manipular Archivos de Configuración Mediante ConfigParser

El módulo ConfigParser de Python resultará de una gran ayuda para desarrollar scripts que nos permitan automatizar tareas generalmente relacionadas con la instalación de aplicaciones en las cuales, tanto la configuración como la optimización y aseguramiento de las mismas, cumpla un papel prácticamente protagónico.

Desde herramientas como fail2ban hasta aplicaciones como MySQL y PHP, utilizan archivos de configuración basados en secciones y, nada más usual, que la necesidad de crear *scripts* automatizados que ayuden a los administradores de sistemas o especialistas en seguridad (o Hackers Éticos si eres de os que disfruta los eufemismos *marketineros*) a configurar de forma óptima estos archivos.

Generalmente, como programadores, resolvemos esta necesidad haciendo "malabares", utilizando expresiones regulares, reemplazando archivos y un sinfín de alternativas que si bien solucionan el problema, claramente nos dan hartos dolores de cabeza.

Sin embargo, desde hace muchísimo tiempo, Python nos provee en su librería estándar un módulo cuya responsabilidad, es justamente, la de permitirnos la manipulación de archivos de configuración basados en secciones. Se trata del módulo `ConfigParser` (renombrado a `configparser` en Python 3).

El módulo ConfigParser permite tanto la lectura como edición de archivos de configuración basados en secciones. Como bien comenté al inicio, ejemplo de este tipo de archivos, podrían ser `fail2ban.conf` de fail2ban, `my.cnf` de MySQL o `php.ini` de PHP, entre otros tantos.

No obstante, `ConfigParser` no será únicamente un módulo destinado solo a un público relacionado a la administración del sistema; también puede ser una buena alternativa, para que los programadores implementen sistemas de configuración a sus propias aplicaciones. Incluso hasta puede ser sumamente útil para implementar en aplicaciones Web en las cuáles se requieran interfaces en múltiples idiomas.

En estos casos, los archivos de configuración podrían servir para definir textos estáticos en los diversos idiomas que la aplicación requiera y toda alternativa de personalización que se le quiera permitir al usuario, podría realizarse de forma sencilla, evitando el consumo de recursos producido por la implementación de bases de datos. Por ejemplo, suponiendo una aplicación Web en español, inglés y catalán, podrían tenerse tres archivos:

```
textos.es
textos.ca
textos.en
```

Y cada uno de ellos, contar con las mismas secciones y variables (reconocidas por `ConfigParser` como "opciones"), pero por supuesto, diferentes valores:

```
; Archivo textos.es

[modulos]
clientes = Clientes
proveedores = Proveedores

[acciones]
administrar = Administrar
editar = Modificar
eliminar = Eliminar


; Archivo textos.en

[modulos]
clientes = Clients
proveedores = Suppliers

[acciones]
administrar = Manage
editar = Edit
eliminar = Delete
```

Como se puede ver, la finalidad de `ConfigParser` es directamente proporcional a la cantidad de usos que se le puede dar a los archivos de configuración basados en secciones y el único límite, está en la imaginación y creatividad del programador.

Crear archivos de configuración con ConfigParser

Cuando se trata de efectuar tareas de administración y/o de seguridad, difícilmente se necesiten crear los archivos de configuración de las aplicaciones que se están optimizando. Generalmente, **la creación de estos archivos, será útil cuando se los requiera para aplicaciones propias.**

> ***La no inclusión de archivos de configuración*** *por defecto en aplicaciones propias, puede utilizarse como* ***parámetro condicional***

Por supuesto, que estos archivos podrían incluirse por defecto en la aplicación, pero la no inclusión, podría utilizarse como parámetro deductivo en la lógica de instalación: **si el archivo no existe, la instalación no se ha concluido**.

ConfigParser fue renombrado a configparser desde Python 3. La herramienta 2to3 efectúa el cambio de nombre automático en la importación del módulo, al momento de convertir el código fuente.

Un ejemplo básico de cómo crear un archivo de configuración desde `ConfigParser` aplicando una lógica deductiva, podría verse como el siguiente:

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

from ConfigParser import RawConfigParser
from os import path


# Si el archivo de configuración no existe
if path.exists('myapp.conf') is False:
     # Pregunto si se lo desea crear
     print 'La aplicación aún no se ha configurado correctamente'
     continuar = raw_input('¿Desea configurarla ahora? (Y/n) ')
     # Creo el archivo de configuración
     if continuar.lower() == 'y':
         cparser = RawConfigParser()  # Se crea el objeto ConfigParser
         with open('myapp.conf', 'wb') as archivo:
            cparser.write(archivo)    # Se escribe el archivo de configuración
     else:
        exit()  # Finalizo la aplicación si el usuario eligió no concluir la instalación

print 'Continúo con la ejecución...'
```

`RawConfigParser` es el objeto de configuración básico que utiliza `ConfigParser`. Sin embargo, el objeto **`SafeConfigParser`** también se encuentra disponible:

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

from ConfigParser import SafeConfigParser

#...

     if continuar.lower() == 'y':
         cparser = SafeConfigParser()
         with open('myapp.conf', 'wb') as archivo:
            cparser.write(archivo)
```

A simple vista, no se puede notar ninguna **diferencia entre `RawConfigParser` y `SafeConfigParser`**, sin embargo, este último cuenta con una característica que lo convierte en "mágico": permite la interpolación de variables en los archivos de configuración.

> *A diferencia de `RawConfigParser`,* ***`SafeConfigParser`*** ***permite la interpolación de variables*** *en los archivos de configuración*

```
; Archivo de configuración
; Ejemplo de interpolación de variables

[miseccion]
carga_maxima_permitida = 100 %(unidad_de_medida)s
unidad_de_medida = MB

; el valor de carga_maxima_permitida será 100 concatenado al valor de unidad_de_medida
; es decir: 100 MB
```

Se debe tener en cuenta que la interpolación anterior, solo será posible si se utiliza `SafeConfigParser`.

En el ejemplo anterior, nos limitamos a crear un archivo de configuración en blanco, lo cual claramente, no tendría mucho sentido. Sin embargo, completar el archivo, no es nada difícil. Basta con **agregar las secciones** con **`add_section`**`('nombre_de_la_seccion')` e **incluir variables** y valores con **`set`**`('seccion', 'variable', 'valor')`. A continuación, un sencillo ejemplo que lo aclarará todo:

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

from ConfigParser import SafeConfigParser
from os import path


if path.exists('myapp.conf') is False:
     print 'La aplicación aún no se ha configurado correctamente'
     continuar = raw_input('¿Desea configurarla ahora? (Y/n) ')
     if continuar.lower() == 'y':
         cparser = SafeConfigParser()
         cparser.add_section('Accesos')
         cparser.set('Accesos', 'servidor', '123.456.78.90')
         cparser.set('Accesos', 'usuario', 'pepegrillo')
         cparser.set('Accesos', 'puerto', '1001')
         with open('myapp.conf', 'wb') as archivo:
            cparser.write(archivo)
     else:
        exit()

# ...
```

Lo anterior, generará el siguiente archivo de configuración:

```
[Accesos]
servidor = 123.456.78.90
usuario = pepegrillo
puerto = 1001
```

Vale aclarar que una forma de crear el mismo archivo de forma personalizada, podría haber sido la utilización de `raw_input` para asignar los valores de cada variable:

```
        cparser.add_section('Accesos')
        cparser.set('Accesos', 'servidor', raw_input('Servidor: '))
        cparser.set('Accesos', 'usuario', raw_input('Usuario: '))
        cparser.set('Accesos', 'puerto', raw_input('Puerto: '))
```

Obteniendo valores

Supongo que a esta altura, existiendo un método `set()`, la obviedad de que seguramente se dispone de un método `get()`, no será necesaria mencionarla. Pues como siempre sucede en Python, la lógica, el sentido común, la intuición y por sobre todo, la prolijidad y la coherencia, van de la mano.

Antes de poder obtener un valor concreto es necesario leer el archivo con el método read

Lo único que se necesita ANTES de invocar al método **`get('sección', 'variable')`** (ya sea de `RawConfigParser` como de `SafeConfigParser`) es "leer" el archivo de configuración. Y para ello, disponemos de un método **`read()`**:

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


from ConfigParser import SafeConfigParser
from os import path


cparser = SafeConfigParser()

if path.exists('myapp.conf') is False:
     print 'La aplicación aún no se ha configurado correctamente'
     continuar = raw_input('¿Desea configurarla ahora? (Y/n) ')
     if continuar.lower() == 'y':
         cparser.add_section('Accesos')
         cparser.set('Accesos', 'servidor', '123.456.78.90')
         cparser.set('Accesos', 'usuario', 'pepegrillo')
         cparser.set('Accesos', 'puerto', '1001')
         with open('myapp.conf', 'wb') as archivo:
            cparser.write(archivo)
     else:
        exit()
else:
```

```
    cparser.read('myapp.conf')
    print cparser.get('Accesos', 'usuario')
```

Lo anterior (siguiendo los mismos ejemplos que al crear el archivo), arrojará `pepegrillo` como salida en caso que el archivo de configuración ya exista.

Otras opciones para obtener valores mediante `ConfigParser` son los métodos `sections()` y `options()` que retornan la lista de secciones y variables (opciones) del archivo de configuración leído, respectivamente:

```
from ConfigParser import SafeConfigParser

cparser = SafeConfigParser()
cparser.read('myapp.conf')
    for seccion in cparser.sections():
        print seccion
        print cparser.options(seccion)
```

Lo anterior, retornaría:

```
Accesos
['servidor', 'usuario', 'puerto']
```

Sin embargo, es muy probable que se desee no solo obtener las variables `(options)` sino también sus valores. Una forma de lograrlo, es mediante el método `items()`, similar a `options()` pero que agrega los valores al elemento de retorno, en forma de tuplas:

```
from ConfigParser import SafeConfigParser

cparser = SafeConfigParser()
cparser.read('myapp.conf')
    for seccion in cparser.sections():
        print seccion
        print cparser.items(seccion)


# La salida de lo anterior será:
# Accesos
# [('servidor', '123.45.678.90'), ('usuario', 'pepegrillo'), ('puerto', '1001')]
```

Finalmente, es posible que se necesite verificar las variables obteniendo errores si no se cumple con el tipo de datos esperado. Para ello, 3 métodos se encuentran disponibles: `getint()`, `getfloat()` y `getboolean()`, aunque es importante entender que **el uso de estos tres métodos, solo se justificará si se desea obtener error en caso de que el elemento solicitado no se corresponda al tipo de datos pedido**:

```
cparser.getboolean('Accesos', 'puerto')  # ValueError: Not a boolean: 1001
```

Si bien los métodos generan error cuando el tipo de datos no es el esperado, no fallan cuando la conversión previa es posible:

```
cparser.getfloat('Accesos', 'puerto')  # 1001.0
```

Agregar secciones y/u opciones

Cuando se desea agregar una o más secciones a un archivo de configuración, se realiza exactamente de la misma forma en la que se ha efectuado el agregado al momento de la creación del archivo. Esta regla aplica también, como es de esperarse, cuando lo que se desea agregar son opciones a las secciones que se están incorporando.

Sin embargo, si se quieren agregar opciones a secciones existentes, la invocación al método `read()` se hará necesaria:

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


from ConfigParser import SafeConfigParser
from os import path


cparser = SafeConfigParser()

if path.exists('myapp.conf') is False:
    print 'La aplicación aún no se ha configurado correctamente'
    continuar = raw_input('¿Desea configurarla ahora? (Y/n) ')
    if continuar.lower() == 'y':
        # Se agrega nueva sección y nuevas opciones
        cparser.add_section('Accesos')
        cparser.set('Accesos', 'servidor', '123.456.78.90')
        cparser.set('Accesos', 'usuario', 'pepegrillo')
        cparser.set('Accesos', 'puerto', '1001')
    else:
        exit()
else:
    # Para agregar nuevas opciones a secciones existentes
    # primero se lee el archivo y luego se realiza la agregación
    cparser.read('myapp.conf')
    cparser.set('Accesos', 'rsa_file_path', raw_input('Ruta llave RSA: '))

with open('myapp.conf', 'wb') as archivo:
    cparser.write(archivo)
```

Analizar o evaluar el contenido de un archivo de configuración

Algo tan simple como verificar si una determinada sección u opción se encuentra presente en un archivo de configuración antes de ejecutar una instrucción concreta, es posible gracias a los métodos **has_section** y **has_option**:

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


from ConfigParser import SafeConfigParser
from os import path


cparser = SafeConfigParser()

if path.exists('myapp.conf') is False:
    print 'La aplicación aún no se ha configurado correctamente'
    continuar = raw_input('¿Desea configurarla ahora? (Y/n) ')
    if continuar.lower() == 'y':
        cparser.add_section('Accesos')
        cparser.set('Accesos', 'servidor', '123.456.78.90')
        cparser.set('Accesos', 'usuario', 'pepegrillo')
        cparser.set('Accesos', 'puerto', '1001')
    else:
        exit()
else:
    cparser.read('myapp.conf')
    # Antes de agregar una nueva opción, verifica si la sección existe:
    if cparser.has_section('Accesos'):
    # Y que la opción a agregar no se encuentre ya presente
        if not cparser.has_option('Accesos', 'rsa_file_path'):
            cparser.set('Accesos', 'rsa_file_path', raw_input('Ruta llave RSA: '))

with open('myapp.conf', 'wb') as archivo:
    cparser.write(archivo)
```

Modificar valores de opciones existentes

En este caso, con tan solo "mezclar un poco de `read()` con algo de `set()`" será suficiente. Siempre que se desee modificar un valor puntual, debe primero leerse el archivo y recurrir al método `set()` para modificar el valor de una variable. Esto puede hacernos llegar a la conclusión que:

> **al invocar al método set si la opción pasada por parámetro existe, modifica su valor; en caso contrario, la crea.**

En el siguiente ejemplo, si la sección "Accesos" existe, se modificará el valor de la variable `rsa_file_path` en caso que exista o se creará la opción, en caso contrario:

```
#...
    if cparser.has_section('Accesos'):
        cparser.set('Accesos', 'rsa_file_path', raw_input('Ruta llave RSA: '))
#...
```

Eliminar secciones u opciones

Finalmente, lo que nos faltaba para poder manipular archivos de configuración de forma completa, era una

forma de eliminar secciones y/u opciones.

Para lograrlo, primero necesitaremos leer el archivo de configuración, luego, invocar a los métodos **`remove_section`**`('sección')` o **`remove_option`**`('sección', 'variable')` para eliminar una sección u opción respectivamente y por último, guardar el archivo:

```
# Elimina toda la sección Accesos
cparser.remove_section('Accesos')

# Elimina solo la opción rsa_file_path de la sección Accesos
cparser.remove_option('Accesos', 'rsa_file_path')
```

Es muy importante saber que si la sección u opción pasada a remove_section o remove_option no existe, ninguno de los dos métodos arrojará un error.

Resumen de ConfigParser

```
# Importación del módulo en Python 2.x
from ConfigParser import SafeConfigParser   # Para soporte de interpolación
from ConfigParser import RawConfigParser    # Sin soporte de interpolación

# Importación del módulo en Python 3.x
from configparser import SafeConfigParser   # Para soporte de interpolación
from configparser import RawConfigParser    # Sin soporte de interpolación


# Inicializar el parser
cparser = SafeConfigParser()  # Con soporte de interpolación
cparser = RawConfigParser()   # Sin soporte de interpolación

# Leer el archivo EXCEPTO si se lo va a crear
cparser.read('/ruta/al/archivo')

# Agregar secciones
cparser.add_section('Sección')

# Agregar o modificar opciones
cparser.set('Sección', 'Variable', 'Valor')


# Obtener valores:
#   Obtener una opción:
cparser.get('Sección', 'Variable')

#   Obtener una opción especificando el tipo de datos para ser convertido o
#   fallando en caso de error
cparser.getint('Seccion', 'variable')
cparser.getfloat('Seccion', 'variable')
cparser.getboolean('Seccion', 'variable')

#   Obtener una lista con todas las secciones
cparser.sections()
```

```
#   Obtener una lista con todas las opciones de una sección
cparser.options('Sección')

#   Obtener una lista con las opciones y valores de una sección, dentro de tuplas
cparser.items('Sección')

# Saber si una sección existe
cparser.has_section('Sección')

# Saber si una opción existe
cparser.has_option('Sección', 'variable')

# Guardar un archivo de configuración
with open('/ruta/al/archivo', 'wb') as archivo:
    cparser.write(archivo)

# Eliminar una sección
cparser.remove_section('Sección')

# Eliminar una opción
cparser.remove_option('Sección', 'variable')
```

Formato de los archivos de configuración

```
; comentarios iniciados por punto y coma
# también son ignoradas líneas iniciadas con el signo numeral

[Nombre de la Sección]
opción = valor tipo string
entero = 1000
flotante = 19.5
booleano = True

; opciones interpoladas
; requieren SafeConfigParser
variable = Se hicieron %(horas)i horas extras
horas = 12
; el valor de variable será: Se hicieron 12 horas extras
```