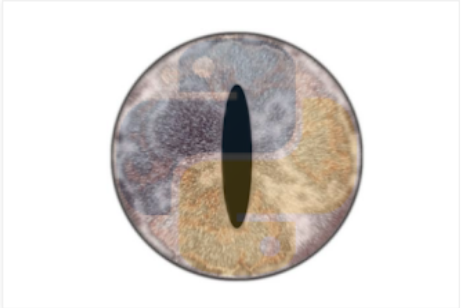
Python 3 para impacientes

"Simple es mejor que complejo" (Tim Peters)

sábado, 1 de agosto de 2015

Buscar, extraer y depurar con Pysaurio



Pysaurio es un módulo para Python 3.x que desarrollamos para el caso práctico del artículo "Empaquetado y distribución de un proyecto Python" publicado en este blog.

Pysaurio permite buscar, extraer y depurar información de un conjunto de archivos del mismo tipo; y crear con toda la información seleccionada un archivo, por ejemplo, del tipo CSV.

La información en los archivos a explorar puede estar dispuesta en varias filas:

PC01.txt:
User=ms123
Name=Mayra Sanz
OS=GNU/Linux
IP=10.226.140.1

Y también puede estar organizada en varias columnas:

PC01.txt:
User: ms123 Name: Mayra Sanz
OS: GNU/Linux IP: 10.226.140.1

En resumen, **Pysaurio**, a partir de los datos de los siguientes archivos:

PC01.txt:
User=ms123
Name=Mayra Sanz
OS=GNU/Linux
IP=10.226.140.1

PC02.txt:
User=lt001
Name=Luis Toribio
OS=GNU/Linux
IP=10.226.140.2

PC03.txt:
User=co205
Name=Clara Osto
OS=Win
IP=10.226.140.3

...puede construir un único archivo CSV con el siguiente contenido:

User,Name,OS,IP
MS123,Mayra Sanz,GNU/Linux,10.226.140.1
LT001,Luis Toribio,GNU/Linux,10.226.140.2
CO205,Clara Osto,Win,10.226.140.3

En el ejemplo el archivo generado está organizado en columnas con un encabezado y se podría abrir con programas que trabajan con hojas de cálculo como **Calc**, **GNumeric** y **Excel**. Este proceso de consolidación de la información puede facilitar el análisis de datos que se almacenan

Buscar

Buscar

Python para impacientes

Python
IPython
EasyGUI
Tkinter
JupyterLab
Numpy

Anexos

Guía urgente de MySQL
Guía rápida de SQLite3

Entradas + populares

Dar color a las salidas en la consola
En Python para dar color a las salidas en la consola (o en la terminal de texto) existen varias posibilidades. Hay un método basado ...

Instalación de Python, paso a paso
Instalación de Python 3.6 A finales de 2016 se produjo el lanzamiento de Python 3.6 . El propósito de esta entrada es mostrar, pas...

Añadir, consultar, modificar y suprimir elementos en Numpy
Acceder a los elementos de un array. [], [,], ... Acceder a un elemento de un array. Para acceder a un elemento se utiliz...

Variables de control en Tkinter
Variables de control Las variables de control son objetos especiales que se asocian a los widgets para almacenar sus valore...

Cálculo con arrays Numpy
Numpy ofrece todo lo necesario para obtener un buen rendimiento cuando se trata de hacer cálculos con arrays. Por como está concebid...

Tkinter: interfaces gráficas en Python
Introducción Con Python hay muchas posibilidades para programar una interfaz gráfica de usuario (GUI) pero Tkinter es fácil d...

Operaciones con fechas y horas. Calendarios
Los módulos datetime y calendar amplían las posibilidades del módulo time que provee funciones para manipular expresiones de ti...

Convertir, copiar, ordenar, unir y dividir arrays Numpy
Esta entrada trata sobre algunos métodos que se utilizan en Numpy para convertir listas en arrays y viceversa; para copiar arrays d...

Tkinter: Tipos de ventanas
Ventanas de aplicación y de diálogos En la entrada anterior tratamos los distintos gestores de geometría que se utilizan para di...

Threading: programación con hilos (I)
En programación, la técnica que permite que una aplicación ejecute

en muchos archivos independientes.

Para instalar **Pysaurio** con **pip**:

$ pip3 install pysaurio

Plantillas .rap

Para procesar archivos con **Pysaurio** es necesario crear una **plantilla .rap** que tiene una estructura similar a un archivo **INI**. Dicha plantilla contendrá una descripción del proceso de búsqueda, extracción y depuración que tendrá que ejecutarse cuando la plantilla se abra. **Pysaurio** incorpora métodos para crear y abrir las plantillas.

Para el ejemplo que hemos comentado la **plantilla .rap** tendría el siguiente contenido:

```
[General]
description = Obtener lista de usuarios
extension = txt
prefix = PC
output_folder = txt
input_folder = txt
output_file = users.csv
delimiter = ,
quotechar = "
include_header = 1
include_file = 0
search_multiple = 0

[Fields]
user = User=
name = Name=
os = OS=
ip = IP=

[Rules]
rule1 = ('user', 'UPPER')
```

Secciones de una plantilla .rap

Un archivo **.rap** se divide en las siguientes secciones:

[General]

Esta sección es obligatoria y puede contener los siguientes atributos:

- **description**: descripción breve de la plantilla.
- **extension**: extensión de los archivos a rastrear (.txt, .log, ...).
- **prefix**: cadena de comienzo de los nombres de los archivos.
- **input_folder**: directorio donde se encuentran los archivos a rastrear.
- **output_folder**: directorio donde se guardará el archivo de salida.
- **output_file**: nombre del archivo de salida (.csv, .html, .txt, ...).
- **delimiter**: carácter utilizado como separador de campos (, ; . | ...).
- **quotechar**: carácter utilizado para indicar el inicio y fin de un campo cuando contenga el carácter expresado como separador de campos (' " ...).
- **include_header**: establece que la salida tenga o no encabezado ('0' o '1').
- **include_file**: establece que la salida incluya o no un campo con el nombre del archivo de donde se extrajeron los datos ('0' o '1').
- **search_multiple**: establece que en una línea puede aparecer o no más de un campo ('0' or '1').

[Fields]

Esta sección es obligatoria y contendrá una línea por cada campo del fichero de salida.

Cada campo se expresará como una asignación: a la izquierda del signo **'='** se escribirá el nombre del campo y a la derecha la cadena que tendrá que buscarse en los ficheros para localizar la información:

nombre_campo = cadena_de_búsqueda

Ejemplo:

user = user=

Pyrsaurio inicialmente leerá todo lo que haya a la derecha de la cadena buscada, hasta alcanzar el final de línea. Y con posterioridad tendremos la posibilidad de depurar la información capturada con reglas: obteniendo una subcadena de una longitud determinada, buscando o sustituyendo caracteres, borrando información desde/hasta un lugar determinado, etc.

simultáneamente varias operaciones en el mismo espacio de proceso se...

Archivo

agosto 2015 (2)

python.org

pypi.org

Sitios

- ActivePython
- Anaconda
- Bpython
- Django
- Flask
- Ipython
- IronPython
- Matplotlib
- MicroPython
- Numpy
- Pandas
- Pillow
- PortablePython
- PyBrain
- PyCharm
- PyDev
- PyGame
- Pypi
- PyPy
- Pyramid
- Python.org
- PyTorch
- SciPy.org
- Spyder
- Tensorflow
- TurboGears

Si en la sección **[General]** el atributo **search_multiple** tiene el valor '**1**' el sistema podrá encontrar más de un campo en una misma línea. Esto implica que después de la captura será necesario aplicar alguna regla para adaptar la información encontrada.

Si el atributo **search_multiple** tiene el valor '**0**' cuando se encuentre un campo en una línea ya no buscará el resto de campos y así **Pysaurio** funcionará más rápido.

[Rules]

Esta sección es opcional y si está presente contendrá las reglas que se aplicarán a los datos encontrados, pudiendo haber más de una regla por campo y pudiendo no aparecer campos a los que no sean necesarios aplicarle ninguna.

Como se ha comentado con anterioridad el uso de reglas será imprescindible cuando se busquen datos de varios campos en una misma línea. En una línea multicampo si un primer campo buscado se encuentra al comienzo de la línea, como dentro de la información capturada estará el resto de la línea, si hubiera más campos, estarían incluidos en ella. Con posterioridad, se aplicarían las reglas necesarias para extraer la información requerida, eliminar la innecesaria, etc., adaptando la información a cada necesidad.

Cada regla se expresará como una asignación: a la izquierda del signo '**=**' se escribirá el identificador de la regla (rule1, rule2, etc) y a la derecha una tupla que contendrá el nombre del campo al que se aplica, la función que se aplica y los argumentos que sean necesarios:

rule1 = ('user', 'UPPER')

Funciones para reglas (rules)

- **SUBSTR**: Extrae del dato una subcadena desde una posición y con una longitud determinada. rule1 = (nombre_campo, 'SUBSTR', posición_inicial, longitud)
- **REPLACE**: Busca una cadena en el dato y la sustituye por otra. rule1 = (nombre_campo, 'REPLACE', cadena_búsqueda, cadena_sustitución)
- **UPPER**: Convierte a mayúsculas el dato. rule1 = (nombre_campo, 'UPPER')
- **LOWER**: Convierte a minúsculas el dato. rule1 = (nombre_campo, 'LOWER')
- **REVERSE**: Invierte el orden de los caracteres que aparecen en el dato. rule1 = (nombre_campo, 'REVERSE')
- **REMOVE**: Borra el dato. rule1 = (nombre_campo, 'REMOVE')
- **FIELDISDATA**: Cambia el dato encontrado por el nombre del campo. rule1 = (nombre_campo, 'FIELDISDATA')
- **REMOVEFROM**: Borra del dato toda la información desde la cadena indicada. rule1 = (nombre_campo, 'REMOVEFROM', cadena)
- **REMOVETO**: Borra del dato toda la información hasta la cadena indicada. rule1 = (nombre_campo, 'REMOVETO', cadena)

Métodos de la clase Raptor

- __init__(self): Inicializa un objeto plantilla .rap

```
>>> from pysaurio import Raptor
>>> rap1 = Raptor()
```

- Save(self, name): Guarda una plantilla .rap

name -- nombre del archivo .rap

>>> rap1.Save('template.rap')

- Open(self, name): Abre una plantilla .rap

name -- nombre del archivo .rap

>>> rap1.Open('template.rap')

- ApplyRules(self, rec): Aplica reglas a los datos de los campos

rec -- Diccionario Python con nombres de campos y valores

Retorna diccionario Python con nombres de campos y valores actualizados

>>> rap1.ApplyRules(Dict)

- BuildHeader(self): Construye encabezado

Retorna lista con nombre de campos

>>> rap1.BuildHeader()

- BuildRow(self, row): Construye el registro de un archivo de entrada

row -- nombre del archivo

Retorna diccionario Python con nombres de campos y valores.

>>> rap1.BuildRow('filename')

- ShowError(self): Muestra errores

Retorna una cadena con los errores producidos al abrir o guardar una plantilla

>>> rap1.ShowError()

Generar archivo .CSV (campos en filas)

En el siguiente ejemplo se muestra el modo de crear y después abrir una plantilla **.rap** para generar el archivo **CSV** que comentamos al principio de este artículo. La información está ordenada en filas y los archivos a procesar se encontrarían en el directorio '**txt**' al mismo nivel que el **script create_csv.py**:

create_csv.py

```
from pysaurio import Raptor
import csv

def main():

 # Crear y guardar una plantilla .rap

 rap1 = Raptor()
 rap1.description = 'Obtener lista de usuarios'
 rap1.extension = 'txt'
 rap1.prefix = 'PC'
 rap1.input_folder = 'txt'
 rap1.output_folder = 'txt'
 rap1.output_file = 'users.csv'
 rap1.delimiter = ','
 rap1.quotechar = '"'
 rap1.include_header = '1'
 rap1.include_file = '0'
 rap1.search_multiple = '0'
 rap1.fields['user'] = 'User='
 rap1.fields['name'] = 'Name='
 rap1.fields['os'] = 'OS='
 rap1.fields['ip'] = 'IP='
 rap1.rules.append(('user', 'UPPER'))
 rap1.Save("users.rap")
 del rap1

 # Abrir la plantilla .rap y generar el archivo .csv

 rap2 = Raptor()
 rap2.Open('users.rap')
 if rap2.number_errors == 0:
  file_csv = open(rap2.output_file, 'w', newline='')
  csv_output = csv.writer(file_csv,
     delimiter=rap2.delimiter,
     quotechar=rap2.quotechar,
     quoting=csv.QUOTE_MINIMAL)
  if rap2.include_header == '1':
   fields_list = rap2.BuildHeader()
   print(fields_list)
   csv_output.writerow(fields_list)

  for row in rap2.list_files:
   new_record = rap2.ApplyRules(rap2.BuildRow(row))
   new_record = list(new_record.values())
   print(new_record)
   csv_output.writerow(new_record)
  file_csv.close()
 else:
  print(rap2.ShowError())
 del rap2
 return 0

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

Además de los atributos de la sección **[General]** en un programa se podrían consultar también los siguientes:

- **fields**: diccionario Python que contendrá nombre de campos y cadenas de búsqueda, es decir, lo declarado en la sección **[Fields]** de un archivo **.rap**.
- **record**: cuando se procesa una lista de archivos, contendrá para un archivo determinado, la lista de campos encontrados con sus respectivos valores.
- **rules**: lista de reglas declarada en la sección **[Rules]** de un archivo **.rap**.
- **list_files**: lista de archivos a procesar.
- **errors**: lista con mensajes de errores al abrir o guardar una plantilla **.rap**.
- **number_errors**: número de errores producidos al abrir o guardar una plantilla **.rap**.

Generar archivo .CSV (campos en columnas)

En el siguiente ejemplo se muestra el modo de crear un archivo **CSV** a partir de ficheros con los campos distribuidos en columnas. Los archivos a procesar serían los siguientes:

PC01.log:
User=ms123 Name=Mayra Sanz
OS=GNU/Linux IP=10.226.140.1

PC02.log:
User=lt001 Name=Luis Toribio
OS=GNU/Linux IP=10.226.140.2

PC03.log:
User=co205 Name=Clara Osto
OS=Win IP=10.226.140.3

create_csv_from_columns.py

```
from pysaurio import Raptor
import csv

def main():

 # Crear y guardar una nueva plantilla .rap

 rap1 = Raptor()
 rap1.description = 'Obtener lista de usuarios con datos en columnas'
 rap1.extension = 'log'
 rap1.prefix = 'PC'
 rap1.input_folder = 'txt'
 rap1.output_folder = 'txt'
 rap1.output_file = 'users_from_columns.csv'
 rap1.delimiter = ','
 rap1.quotechar = '"'
 rap1.include_header = '1'
 rap1.include_file = '0'
 rap1.search_multiple = '1'
 rap1.fields['user'] = 'User='
 rap1.fields['name'] = 'Name='
 rap1.fields['os'] = 'OS='
 rap1.fields['ip'] = 'IP='
 rap1.rules.append(('user', 'UPPER'))
 rap1.rules.append(('user', 'SUBSTR', 1, 6))
 rap1.rules.append(('os', 'REMOVEFROM', ' '))
 rap1.Save("users_columns.rap")
 del rap1

 # Abrir la plantilla .rap y generar el archivo .csv

 rap2 = Raptor()
 rap2.Open('users_columns.rap')
 if rap2.number_errors == 0:
  file_csv = open(rap2.output_file, 'w', newline='')
  csv_output = csv.writer(file_csv,
     delimiter=rap2.delimiter,
     quotechar=rap2.quotechar,
     quoting=csv.QUOTE_MINIMAL)
  if rap2.include_header == '1':
   fields_list = rap2.BuildHeader()
   print(fields_list)
   csv_output.writerow(fields_list)

  for row in rap2.list_files:
   new_record = rap2.ApplyRules(rap2.BuildRow(row))
   new_record = list(new_record.values())
   print(new_record)
```

```
   csv_output.writerow(new_record)
  file_csv.close()
 else:
  print(rap2.ShowError())
 del rap2
 return 0

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

Generar varios archivos .csv utilizando varias plantillas .rap

Para finalizar, un ejemplo en el que se procesan las plantillas **.rap** de una lista. Posiblemente, lo más interesante de este código está en ver cómo se crean dinámicamente objetos en Python.

create_csv_run_list_templates.py

```
from pysaurio import Raptor
import csv

def main():

 # Declarar lista con plantillas .rap

 rap_templates = ['users.rap',
   'users_columns.rap']

 # Declarar objetos dinámicamente

 objects = [Raptor() for index in range(len(rap_templates))]

 # Recorrer todos los objetos

 for number_object in range(len(objects)):
  current_rap = rap_templates[number_object]
  current_object = objects[number_object]
  print('Template:', current_rap)

  # Abrir plantilla .rap y generar archivo .csv

  current_object.Open(current_rap)
  if current_object.number_errors == 0:
   file_csv = open(current_object.output_file, 'w', newline='')
   csv_output = csv.writer(file_csv,
      delimiter=current_object.delimiter,
      quotechar=current_object.quotechar,
      quoting=csv.QUOTE_MINIMAL)
   if current_object.include_header == '1':
    fields_list = current_object.BuildHeader()
    print(fields_list)
    csv_output.writerow(fields_list)

   for row in current_object.list_files:
    new_record=current_object.ApplyRules(current_object.BuildRow(row))
    new_record=list(new_record.values())
    print(new_record)
    csv_output.writerow(new_record)
   file_csv.close()
  else:
   print(current_object.ShowError())
  del current_object
 return 0

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

Ir al índice del tutorial de Python

Publicado por Pherkad en 16:49

Etiquetas: Python3

Entrada más reciente Inicio Entrada antigua