Corso di **Python**

Lezione 11

Fare script con Python

Editor: Davide Brunato
Scuola Internazionale Superiore di Studi Avanzati di Trieste

Definire comandi

- La definizione di comandi di sistema è la tipica attività del sistemista Unix:
  - Ridurre l'onere di ricordare sequenze lunghe di comandi
  - Per definire comandi da eseguire come batch
  - CLI più *elegante ed efficiente* della GUI
- In campo Unix storicamente si usano i linguaggi di shell (**bash**, ksh)
- Successivamente si è affermato anche l'uso di Perl per fare script più elaborati
- Perché Python allora?

Python VS Bash ??

- Non è da intendere come un *versus* ma come un aiuto
- Bash infatti pur avendo un suo set di istruzioni e funzioni builtin già si deve forzatamente avvalere di comandi extra per molte elaborazioni (uno per tutti: *grep*)
- Gli script Bash possono diventare facilmente troppo complicati e per poter successivamente migliorarli o mantenerli
- Python può essere usato sia in sostituzione per compiti elaborati che per definire comandi extra per Bash
- Python dispone di una serie di librerie che in Bash mancano e l'interfacciamento mediante programmi esterni non sempre è agevole

Interazione Bash e Python

- Conveniente usare Python per realizzare comandi per i quali risulterebbe decisamente più complicato farli in Bash
- Ad esempio avendo la necessità di contare le occorrenze da un elenco di nomi si può costruire un comando:

```
#!/usr/bin/env python
import sys
if __name__ == "__main__":
    names = {}
    for name in sys.stdin.readlines():
        name = name.strip()
        if name in names:
            names[name] += 1
        else:
            names[name] = 1
    for name, count in names.iteritems():
        sys.stdout.write("%d %s\n" % (count, name))
```

- Il nuovo comando può essere usato con la pipe insieme ad altri comandi Unix:

```
$ cat names.log | contanomi.py | sort -rn
```

Librerie Python

- Alcune librerie utili per definire comandi:
  - **platform**: accesso alla piattaforma HW e SW
  - **optparse**: parser per gli argomenti/opzioni
  - **configparser**: parser per i file di configurazione
  - **logging**: servizio di logging
  - **datetime**: date e tempo

La libreria **platform**

- Per avere velocemente informazioni più dettagliate sulla piattaforma HW/SW si può usare la libreria **platform**
- Questa libreria fornisce informazioni sull'architettura, processore, tipo si sistema operativo, versione
- Esempi:

```
>>> import platform
>>> platform.linux_distribution()
('Fedora', '22', 'Twenty Two')
>>> platform.libc_ver()
('glibc', '2.2.5')
>>> platform.processor()
'x86_64'
>>> platform.platform()
'Linux-3.10.0-229.11.1.el7.x86_64-x86_64-with-centos-7.1.1503-Core'
>>> platform.python_version()
'2.7.5'
```

- Documentazione: https://docs.python.org/library/platform.html

Link tra librerie

- Platform è un esempio di come in Python le librerie sono connesse tra loro:

```
>>> import platform
>>> platform.sys
<module 'sys' (built-in)>
>>> sys
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'sys' is not defined
>>> platform.sys.version
'3.4.3 (default, Jun 29 2015, 12:16:01) \n[GCC 5.1.1
20150618 (Red Hat 5.1.1-4)]'
```

- In effetti succede ogni volta che carichiamo un modulo che usa una libreria ...

Parse delle opzioni

- Per il parse della linea di comando si può scegliere tre le librerie **argparse** o **optparse**:
  - **optparse** è disponibile dalla v. 2.3 ma è più limitata e non verrà più sviluppata
  - **argparse** è disponibile dalla v.2.7
- Cos'ha in più argparse rispetto ad optparse:
  - Gestione argomenti posizionali
  - Supporto di sotto-comandi
  - Possibilità di diversificare i prefissi delle opzioni
  - Gestione argomenti multipli zero-or-more e one-or-more
  - Fornisce messaggi d'uso più informativi
  - Rende più semplice la customizzazioni del tipo e delle azioni
- Usare la libreria optparse se si vogliono script compatibili con le versioni 2.4- 2.6, ancora diffuse sulle distro Linux di lungo supporto

Come usare optparse

- Esempio di comando con 2 opzioni:

```
from optparse import OptionParser

parser = OptionParser()
parser.add_option("-f", "--file", dest="filename",
      help="write report to FILE", metavar="FILE")
parser.add_option("-q", "--quiet", action="store_false",
      dest="verbose", default=True, help="don't"
      "print status messages to stdout")
(options, args) = parser.parse_args()
```

- ***options*** conterrà le opzioni dichiarate come attributi
- ***args*** invece conterrà gli argomenti posizionali residui, ossia gli argomenti non associati alle opzioni

Come si farebbe in Bash?

- Gli argomenti sono in variabili predefinite:
  - `$0`, `$1–$9`, `${10}`, `$#`, `$*`
- Modalità manuale per il parse delle opzioni:

```
if [ "$1" = "-f"]; then
    optfile = $2
    shift 2
fi
```

  - Problemi con l'ordine e il numero degli argomenti
- È preferibile usare il comando ***getopts*** (o la variante esterna *getopt*, peralto disponibile anche nella libreria di Python), che prevede prima una verifica degli argomenti e poi in un'analisi in un ciclo iterativo

Uso di **getopts** in Bash

```
#!/bin/bash
usage() { echo "Usage: $0 [-s 0..100] [-p <string>]" 1>&2; exit 1; }
while getopts ":s:p:" o; do
    case "${o}" in
        s)
            s=${OPTARG}
            ((s >= 00 && s <= 100)) || usage
            ;;
        p)
            p=${OPTARG}
            ;;
        *)
            usage
            ;;
    esac
done
shift $((OPTIND-1))
if [ -z "${s}" ] || [ -z "${p}" ]; then usage fi
echo "s = ${s}, p = ${p}"
```

Libreria **argparse**

- Utile per comandi complessi che richiedono opzioni più elaborate
- Analizza anche gli argomenti oltre che le opzioni, permettendo anche azioni su opzioni multiple (optparse prende l'ultima specificata ed ignora le altre)
- Esempio (tratto dal tutorial):

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
                    help="display a square of a given number")
parser.add_argument("-v", "--verbosity", action="count", default=0,
                    help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2
if args.verbosity >= 2:
    print("the square of {} equals {}".format(args.square, answer))
elif args.verbosity >= 1:
    print("{}^2 == {}".format(args.square, answer))
else:
    print answer
```

File di configurazione

- Non sempre le opzioni sono sufficienti a definire bene il comportamento degli script
- Alcuni parametri di default è meglio metterli in file di configurazione invece che nel codice
- Questo evita di dover sempre passare tutto per le opzioni o di stabilire default troppo rigidi

Opzioni in Python per i file di configurazione

- Direttamente in modulo dedicato (es. *settings.py*) che si trova in nello spazio utente
  - Definire un modo per accedere ai settaggi che non si porti dentro codice non desiderato
- File di configurazione in formato .INI like con parser definito nella libreria **configparser**
- In altri formati usabili in Python per definire file di configurazione:
  - **JSON** con libreria standard `json` (dalla versione 2.6, prima era disponibile come libreria esterna)
  - **YAML** con libreria esterna `PyYAML`
  - **XML** con libreria standard `xml.etree.ElementTree` (dalla versione 2.5, prima si usavano API DOM o SAX2) oppure con libreria esterna **`lxml`** (il miglior parser XML disponibile in Python, che effettua anche la validazione sullo schema)

File di configurazione con libreria **configparser**

- Si possono gestire file di configurazione con struttura simile ai file *INI* di MS Windows
  - Limitazione a 2 livelli: sezioni e opzioni
- In Python 2 il modulo sarebbe *ConfigParser*, ridenominato in Python 3 in *configparser* (2to3 converte il codice ...).
- Per fare codice compatibile 2.x-3.x si può usare un import con un'eccezione:

```
try:
    import configparser
except ImportError:
    # Fall back for Python 2.x
    import ConfigParser as configparser
```

Esempio di configurazione

```
# Configuration file example (save as 'testconfig.cfg')
; This is a comment line
[main] ; This is a comment to a section
logdir : /var/log
fromaddr = root@localhost
smtpserv = /usr/sbin/sendmail -t

[report]
template_dir = /etc/templates
title: $localhost system events: $localtime
html_template= %(template_dir)s/report_template.html
text_template= ${template_dir}/report_template.txt   ;
another_option: ' option with spaces on the sides   '
long_option: this is an option defined on ; This is a comment
   two lines

[mail]
method = mail
formats = plain, csv
mailto = root
include_rawlogs = no
rawlogs_limit = 200
```

Leggere la configurazione

```
#!/usr/bin/env python
# Test di lettura di una configurazione per Python 2/3 (salvare come .py)
try:
    import configparser
except ImportError:
    import ConfigParser as configparser

parser = configparser.ConfigParser()
parser.read('testconfig.cfg')
for sect in parser.sections():
    print("Configuration section [%s]" % sect)
    for opt in parser.options(sect):
        print('Option "%s": "%s"' %(opt, parser.get(sect, opt)))
    print("")

# Togliere i commendi per visualizzare alla fine la struttura del parser
# print("*** Struttura del parser ***")
# print('dir(parser): {0}\n'.format(dir(parser)))
# print('parser.__dict__: {0}'.format(parser.__dict__))
```

Opzioni per leggere una configurazione

- La lettura da file con funzione *read* può comprendere contemporaneamente la lettura da più file di configurazione:

```
import configparser, os
config = configparser.ConfigParser()
config.read_file(open('defaults.cfg'))
config.read(['site.cfg',
    os.path.expanduser('~/.myapp.cfg')])
```

- `ConfigParser.read` ritorna la lista di file effettivamente letti e dai quali si è estratta la configurazione:

```
>>> parser.read('testconfig.cfg')
['testconfig.cfg']
```

- Dalla versione 3.2 c'è la possibilità di leggerla da sorgenti che non sono file:

```
configparser.ConfigParser.read_string(string, source='<string>')
configparser.ConfigParser.read_dict(dictionary, source='<dict>')
```

Il parametro *source* specifica un nome per identificare l'oggetto dal quale la configurazione è stata letta.

Nel caso di lettura da stringa il formato della stringa deve essere identico a quello del file. Nel caso di dizionario: le sezioni sono le chiavi di primo livello, le opzioni quelle di secondo livello.

Modificare e scrivere una configurazione

- API per alterare le sezioni ed opzioni:

```
configparser.ConfigParser.set(section, option, value)
configparser.ConfigParser.add_section(section)
configparser.ConfigParser.remove_section(section)
configparser.ConfigParser.remove_option(section, option)
```

- Le modifiche vengono poi scritte con il metodo `ConfigParser.write`:

```
>>> from configparser import ConfigParser
>>> from io import StringIO
>>> parser = ConfigParser()
>>> parser.read('testconfig.cfg')
>>> f = StringIO()
>>> parser.set('report', 'title', "Sezione principale")
>>> parser.write(f)
>>> print(f.getvalue())
```

Interpolazione dei valori

- Il parser gestisce un'interpolazione sui valori delle opzioni con la sintassi `%(`*`opzione`*`)s`
  - Vincolo: la sostituzione è limitata alla sezione
- Un'interpolazione più estesa è disponibile in Python 3:
  - Utilizza la sintassi `${opzione}` per definire le stringhe da sostituire
  - È possibile effettuare la sostituzione a livello globale di configurazione, usando la sintassi `${sezione:opzione}`
- Per usare la forma estesa bisogna creare il parser con una classe di mixin diversa da quella base:

```
from configparser import ConfigParser, ExtendedInterpolation
parser = ConfigParser(interpolation=ExtendedInterpolation())
```

Interpolazione a posteriori

- L'interpolazione sulle opzioni di configurazione si può fare anche a posteriori usando delle *template string* e il metodo `string.Template`*:*
  - PEP 292: https://www.python.org/dev/peps/pep-0292/

- Esempio:

```
>>> from string import Template
>>> from configparser import ConfigParser
>>> parser = ConfigParser()
>>> parser.read('testconfig.cfg')
>>> title = parser.get('report', 'title')
>>> Template(title).substitute(localhost='ubik', localtime='12:22')
'ubik system events: 12:22'
```

- In questo caso è meglio usare una struttura ad-hoc (es. dizionario) per mappare le configurazioni e poi alterarle, oppure fare una sottoclasse di ConfigParser per implementare un differente metodo di get dei valori che effettui le sostituzioni con variabili del proprio ambiente

Logging dei comandi

- Un metodo fondamentale per controllare e correggere gli errori nei comandi è quello di produrre dei log:
  - Gli echo/print sono utili ma costa gestirli
  - Meglio un feedback che possa essere regolato
- Python 2.3+ dispone della libreria **logging**
- In Bash si usa in genere il comando esterno **logger**, per collegare lo script con i log di sistema:

```
exec 1> >(logger -s -t $(basename $0)) 2>&1
```

  - Con l'opzione `-p`/`--priority` si imposta il livello, però si coinvolgono anche i file stdout e stderr, che vengono rediretti se si vogliono i log anche in console

Logging in Python

- È indipendente dalla gestione di stdout/stderr
- Si possono impostare più *handler*, anche di tipo diverso, ad esempio:
  - ***StreamHandler***: che redirige su stream di output (come sys.stdout)
  - ***FileHandler***: per il logging su specifico file
- Si possono definire delle formattazioni specifiche per ogni handler
- Si possono definire più logger e condividerli tra i moduli
- Può essere impostato un livello generale e uno per handler:

`logging.CRITICAL`	`50`
`logging.ERROR`	`40`
`logging.WARNING`	`30`
`logging.INFO`	`20`
`logging.DEBUG`	`10`
`logging.NOTSET`	`0`

Definire un logger

- Si può lavorare con il logger fornito dal modulo della libreria:

```
>>> import logging
>>> logging.basicConfig(filename='example.log',level=logging.DEBUG)
>>> logging.debug('Messaggio per il debug')
>>> logging.warning('Messaggio di warning')
>>> logging.info('Messaggio di livello informativo')
```

- Il logger della libreria può andare bene per comandi limitati ad un solo modulo
- Quando invece serve definire un logger comune tra più moduli si crea un'istanza nominale:

```
>>> import logging
>>> logger = logging.getLogger('miologger')
>>> logger.setLevel(logging.INFO)
>>> logger.warning("Messaggio di warning")
>>> logger.addHandler(logging.StreamHandler())
>>> logger.warning("Messaggio di warning")
Messaggio di warning
>>> logger.debug("Messaggio per il debug")
>>> logger.info("Messaggio informativo")
Messaggio informativo
```

Libreria **time**

- Fornisce alcune funzioni varie per valori legati al tempo
- Alcune API:

  `time.`**`clock`**`()`: frequenza del processore

  `time.`**`sleep`**`(`*`seconds`*`)`: sospende l'esecuzione del thread

  `time.`**`time`**`()`: ritorna il tempo come numero floating point con numero di secondi da *epoch*

  `time.`**`timezone`**: offset in secondi della timezone locale rispetto al riferimento UTC

  `time.`**`strptime`**`(`*`string`*`[, `*`format`*`])`: effettua un parse della stringa secondo un formato data specificato e ritorna struttura che rappresenta il tempo indicato (*struct_time*)

  `time.`**`strftime`**`(`*`format`*`[, `*`t`*`])`: converte una oggetto *struct_time* in stringa, secondo il formato specificato

Libreria **calendar**

- Fornisce classi di utilità per la gestione delle date

  `calendar.`**`TextCalendar`**`([`*`firstweekday=0`*`])`

  - classe per generare calendari in formato testo

  `calendar.`**`HTMLCalendar`**`([`*`firstweekday`*`])`

  - Classe per generare calendari HTML

  `calendar.`**`LocaleTextCalendar`**`([`*`firstweekday`*`[, `*`locale`*`]])`:

  `calendar.`**`LocaleHTMLCalendar`**`([`*`firstweekday`*`[, `*`locale`*`]])`:

  - Derivate dalle corrispondenti classi ma con output codificato secondo le convenzioni della linguaggio selezionato

- Le classi includono metodi per ottenere le date formattate secondo il layout testuale scelto e iteratori per create cicli su settimane e mesi

Libreria **datetime**

- È una libreria fondamentale per definire oggetti che rappresentano il tempo
- Fornisce 5 classi:

  `datetime.date()`:

  - Rappresentazione delle date secondo il calendario Gregoriano, con 3 attributi: *year*, *month* e *day*.

  `datetime.time()`:

  - Rappresentazione del tempo giornaliero ideale (24*60*60 seconds). Possiede 5 attributi: *hour*, *minute*, *second*, *microsecond, tzinfo*.

  `datetime.datetime()`:

  - Combinazione delle precedenti, con attributi *year*, *month*, *day*, *hour*, *minute*, *second*, *microsecond*, *tzinfo*.

  `datetime.timedelta()`:

  - per rappresentare differenze in microsecondi tra istanze delle precendeti classi.

  `datetime.tzinfo()`:

  - classe per rappresentare informazioni sulle time zone

La classe **datetime**

- Per gli script è sicuramente la classe fondamentale
- Metodi principali:

`datetime.`**`datetime`**`(`*`year`*`, `*`month`*`, `*`day`*`[, `*`hour`*`[, `*`minute`*`[, `*`second`*`[, `*`microsecond`*`[, `*`tzinfo`*`]]]]])`

  - Costruttore di istanza, necessario specificare anno, mese e giorno

`datetime.`**`fromtimestamp`**`(`*`timestamp`*`[, `*`tz`*`])`

  - Ritorna un oggetto *datetime* partendo dal timestamp relativo al numero di secondi da *epoch*

`datetime.`**`today`**`()`

`datetime.`**`now`**`([`*`tz`*`])`

  - ritorna un oggetto datetime relativo al tempo corrente
  - Equivale a: `datetime.fromtimestamp(time.time())`

`datetime.`**`strptime`**`(`*`date_string`*`, `*`format`*`)`

  - ritorna un oggetto *datetime* costruendolo dalla stringa in argomento, con interpretazione definita dal formato

`datetime.`**`strftime`**`(`*`format`*`)`

  - ritorna la rappresentazione dell'oggetto datetime come stringa, secondo la formattazione indicata

Operatori su oggetti **datetime** e **timedelta**

Operazione	Descrizione
`datetime2 = datetime1 + timedelta`	Somma un *timedelta* ad un *datetime* ottenendo un oggetto che rappresenta un tempo precedente o successivo, a seconda che il delta sia positivo o negativo
`datetime2 = datetime1 - timedelta`	Come la somma, ma la il risultato rappresenterà un tempo successivo se il timedelta è negativo
`timedelta = datetime1 - datetime2`	Effettua una differenza tra due datetime, e il risultato è un delta temporale, che può essere positivo se datetime1 rappresenta un tempo successivo rispettoa a *datetime2*
`datetime1 < datetime2` `datetime1 == datetime2` `datetime1 > datetime2`	Operatori di confronto tra due oggetti *datetime*