Generazione di documenti LaTeX con Python e Jinja

Giacomo Mazzamuto

Sommario Questo articolo illustra come creare documenti LaTeX con Jinja, un motore di *template* per Python, mettendo in evidenza le potenzialità fornite dall'avere a disposizione un linguaggio semplice e ricco come Python all'interno di un documento LaTeX. L'articolo è composto da un *tutorial* che dimostra le funzionalità di base di Jinja e da un esempio di configurazione più avanzata finalizzato a rendere più gradevole la sintassi di Jinja all'interno di un documento LaTeX. Infine, è mostrato l'esempio della generazione di una fattura.

Abstract This short article describes how to use Jinja, a template engine for Python, to create LaTeX documents harnessing the power of a simple and rich language such as Python. The article begins with a tutorial showing the basic usage of Jinja, followed by a more advanced configuration aiming at making Jinja's syntax more LaTeX-friendly. Finally, an example involving the generation of an invoice is shown.

1. Introduzione

Jinja[1] è un motore per modelli (o *template engine*) per Python[2]. Come molti *template engine*, Jinja è nato in ambito web dove viene usato principalmente per creare pagine HTML in modo dinamico da parte di un server di *backend*. La sua sintassi si ispira a quella del motore di template di Django, un popolare *web framework* anch'esso scritto in Python.

In generale, un *template engine* non è altro che un sistema per generare *file* di testo in base a un modello di partenza — il *template* appunto — e trova quindi applicazione nella generazione dinamica di documenti di testo di qualsiasi tipo, siano essi pagine HTML, documenti XML oppure sorgenti LaTeX. Per un'introduzione più dettagliata sul concetto di *template* si veda Giacomelli e Pignalberi (2015). Personalmente mi è capitato di utilizzare Jinja persino per la generazione in serie di documenti SVG, il diffuso formato di grafica vettoriale che, infatti, è un formato testuale, essendo basato su XML. Tutto quello che si può fare con Jinja e LaTeX si può ottenere, per esempio, anche con LuaLaTeX grazie al linguaggio di *scripting* Lua. Tuttavia, date la popolarità e la semplicità di Python, l'ampiezza della sua libreria standard e la vastità di pacchetti disponibili, vale la pena di esplorare le potenzialità offerte dall'unione di questi due mondi: Python e LaTeX. Sarà infatti molto facile creare documenti LaTeX con dati prelevati dinamicamente da un *database* o da Internet, o semplicemente si potrà utilizzare Python come linguaggio di *scripting* all'interno di documenti LaTeX.

In questo articolo vedremo come utilizzare Jinja per generare dei semplici sorgenti LaTeX, ed esploreremo alcune opzioni di configurazione per renderlo più "amichevole" in quest'ambito

1. https://palletsprojects.com/p/jinja/
2. https://www.python.org

d'uso. Si presuppone che l'utente abbia già installato un interprete Python sul proprio sistema, mentre per installare Jinja è sufficiente eseguire questo comando:

```
pip install -U Jinja2
```

2. Guida all'uso

Consideriamo il seguente "modello" di documento LaTeX:

```
\documentclass{article}
\begin{document}
L'autore di {{myvar.title}} è
{{myvar.author}}.
\end{document}
```

Nell'esempio qui sopra, si può facilmente riconoscere che si tratta di un modello perché alcune porzioni di esso (`{{myvar.title}}` e `{{myvar.author}}`) sono dei segnaposti che verranno sostituiti dalle corrispondenti variabili quando il *template* sarà "compilato". Utilizziamo il seguente *script* Python per compilare il *template*:

```
from jinja2.nativetypes import \
    (NativeEnvironment, Environment)
from jinja2.loaders import \
    FileSystemLoader

env = Environment(
    loader=FileSystemLoader('.'),
)

mydict = {
    'author': 'James Joyce',
    'book': 'Dubliners',
}

templ = env.get_template('template.tex')
templ = templ.stream(myvar=mydict)
templ.dump('output.tex')
```

Commentiamo il codice passo passo. Dopo aver importato i pacchetti e le classi necessarie, alle righe 6–8 si inizializza Jinja configurandolo affinché carichi i modelli cercandoli nel *filesystem*, in particolare nella cartella corrente (`'.'`). Alle righe 10–13 si definisce un *dizionario* con due chiavi (`author` e `book`). Alla riga 15 si carica il *template*, supponendo di aver salvato il documento LaTeX di cui sopra in un file chiamato `template.tex`. Infine, alla riga 16 si "compila" il template avendo cura di passare tutte le variabili di cui abbiamo bisogno (in questo caso, soltanto `myvar=mydict`), e il risultato della compilazione sarà salvato in un file chiamato `output.tex` (riga 17). Dopo aver eseguito questo *script*, il file `output.tex` così generato conterrà il seguente documento LaTeX:

```
\documentclass{article}
\begin{document}
L'autore di Dubliners è James Joyce.
\end{document}
```

Come si può vedere, i due segnaposti sono stati sostituiti con i corrispondenti valori del *dizionario* `mydict` che abbiamo definito nello *script* Python e che abbiamo reso disponibile al *template* come variabile chiamata `myvar`. Questo è un esempio illustrativo, dove per motivi di semplicità i valori da sostituire sono definiti (*hard-coded*) direttamente nello script; più probabilmente, in un caso reale di utilizzazione, i dati sarebbero prelevati da un qualche tipo di banca dati come un *database* MySQL, o anche semplicemente letti da un file. Le doppie parentesi graffe racchiudono un'*espressione*, al loro interno sono cioè ammesse semplici espressioni Python come mostrato in questo esempio:

```
\begin{document}
Il cubo di 4 è {{4**3}}
La mia stringa in maiuscolo:
{{'prova'.upper()}}.
\end{document}
```

che diventa:

```
\begin{document}
Il cubo di 4 è 64.
La mia stringa in maiuscolo: PROVA.
\end{document}
```

Complichiamo leggermente le cose passando al motore di *template*, anziché un singolo *dizionario* come fatto sopra, una *lista* (`books`) contenente due *dizionari*:

```
books = [
    {'author': 'James Joyce',
     'title': 'Dubliners'},
    {'author': 'Donald Knuth',
     'title': 'The Art of Computer '
              'Programming'}
]
templ = env.get_template('template.tex')
templ = templ.stream(books=books)
templ.dump('output.tex')
```

Modifichiamo inoltre il file `template.tex` in questo modo:

```
\begin{document}                                    1
{% for b in books %}                                2
L'autore di {{b.title}} è {{b.author}}.             3
{% endfor %}                                        4
\end{document}                                      5
```

Mentre come abbiamo già visto le doppie parentesi graffe delimitano un'*espressione*, la parentesi graffa seguita dal segno di percentuale delimita un *blocco* che racchiude uno *statement* — ad esempio: un ciclo `for` come in questo caso, oppure un costrutto condizionale (`if`). Dopo aver compilato il *template*, il risultato sarà:

```
\begin{document}

L'autore di Dubliners è James Joyce.

L'autore di The Art of Computer
Programming è Donald Knuth.

\end{document}
```

Si notino le linee vuote che vengono inserite nel documento finale. Poiché in LaTeX la linea vuota ha il ben preciso significato di iniziare un nuovo capoverso, può essere necessario fare attenzione a questo comportamento che in ogni caso è personalizzabile agendo su specifiche opzioni di configurazione di Jinja, oppure si possono semplicemente rimuovere gli accapo alle righe 2–4 del *template* (cioè scrivendo tutto su una o due righe anziché su tre come nell'esempio).

Concludiamo questa breve introduzione vedendo come chiamare funzioni Python direttamente dal *template*. Supponiamo di avere una funzione `cubed(x)` che restituisce il cubo di un numero. Occorre innanzitutto "registrare" la funzione presso Jinja prima di compilare il template:

```
def cubed(x):
    return x**3

env.globals.update(cubed=cubed)
```

In questo modo, il seguente *template*:

```
\begin{document}
Il cubo di 4 è {{cubed(4)}}.
\end{document}
```

diventerà:

```
\begin{document}
Il cubo di 4 è 64.
\end{document}
```

Ovviamente il documento così prodotto deve poi essere compilato con `pdflatex`, ma non è difficile modificare lo *script* Python in modo che, oltre a generare il documento `.tex` (magari in una cartella temporanea), si occupi anche di compilare il documento finale in PDF, ad esempio usando il modulo `subprocess` della libreria standard di Python per eseguire `pdflatex` oppure `latexmk -pdf`. Uno *script* così fatto sarebbe perfetto come *microservizio*[3] nel *backend*

3. Nelle moderne architetture *cloud*, i microservizi sono un modo di strutturare il *backend* in una rete di piccoli servizi indipendenti, ciascuno dedicato ad assolvere un compito specifico e scritto nel linguaggio di programmazione più appropriato per quel compito.

di un sito web che deve generare documenti PDF dinamicamente in base ai dati degli utenti, come ad esempio un attestato o una fattura.

3. Personalizzazione di Jinja

Gli esempi mostrati fino ad ora sono scritti nella sintassi predefinita di Jinja. Tuttavia, è possibile configurare Jinja cambiandone la sintassi in modo che questa si amalgami meglio con quella di LaTeX. Il mescolamento delle sintassi di LaTeX e di Jinja, infatti, può non risultare gradevole e inoltre non è ben gestito dagli *editor* di testo che supportano la colorazione della sintassi. Ad esempio, nella sezione precedente abbiamo visto che un *blocco* è delimitato dalla coppia di caratteri costituita da una parentesi graffa e da un segno di percentuale. Siccome in LaTeX il segno di percentuale introduce un commento, il nostro *editor* riconoscerebbe lo *statement* come un commento ma solo a partire dal segno `%`, escludendo quindi il primo carattere (la parentesi graffa aperta); è per questo motivo che, anche in questo articolo, la parentesi graffa dello *statement* è mostrata in verde mentre il resto del rigo è in grigio come se fosse un commento. Per quanto riguarda le *espressioni*, la doppia parentesi graffa non è dissonante con la sintassi di LaTeX, ma si può pensare di renderla comunque più esplicita.

Fortunatamente, Jinja offre molte opzioni di configurazione e personalizzazione, sfruttando le quali è ad esempio possibile ridefinire alcuni elementi della sua sintassi in stile LaTeX:

```
1 env = Environment(
2     loader=FileSystemLoader('.'),
3     variable_start_string='\PYEXP{',
4     variable_end_string='}',
5     block_start_string='\PYBLOCK{',
6     block_end_string='}',
7     line_statement_prefix='%%',
8 )
```

Con questa configurazione, l'espressione `{{4**3}}` si scriverebbe come `\PYEXP{4**3}`, camuffandola quindi come un comando LaTeX. Lo stesso si può dire per lo *statement* che adesso si scriverebbe come:

```
\PYBLOCK{for b in books}
L'autore di \PYEXP{b.title} è
\PYEXP{b.author}.
\PYBLOCK{endfor}
```

C'è però un modo ancora più conciso e gradevole per scrivere uno *statement* in un *template* LaTeX. Jinja consente infatti di considerare una riga come uno *statement* se questa inizia con un determinato prefisso, che nella configurazione predefinita è il carattere cancelletto. Nella configurazione di cui sopra, alla riga 7 abbiamo specificato che intendiamo usare un doppio segno di percentuale come prefisso per i cosiddetti *line statement*. In questo modo non c'è ambiguità tra un normale commento LaTeX, che inizia con un singolo carattere `%`, e uno *statement*, che invece comincia con `%%` ma che viene comunque interpretato come commento dall'*editor* di testo come mostrato nell'esempio seguente:

```
\begin{document}
%% for b in books
L'autore di \PYEXP{b.title} è
\PYEXP{b.author}.
%% endfor
% normale commento LaTeX
\end{document}
```

che diventa:

```
\begin{document}
L'autore di Dubliners è James Joyce.
L'autore di The Art of Computer
Programming è Donald Knuth.
% normale commento LaTeX
\end{document}
```

Si noti che, quando si usano i *line statement* come in questo esempio, nel documento finale non vengono inserite le righe vuote prima e dopo il *blocco*, a differenza di quanto visto nella sezione precedente.

Nella pagina seguente è riportato il codice completo di un piccolo esempio che raccoglie quanto visto fino ad ora, il quale per ragioni di brevità non rende giustizia alle potenzialità offerte dal poter accedere, direttamente in un documento LaTeX, a elaborazioni comunque complesse scritte in un linguaggio agevole come Python. Il listato 1 mostra sia il *template* sorgente (in alto) sia il risultato della compilazione tramite lo *script* riportato nel listato 2 (in basso). Si noti, in aggiunta al già discusso ciclo `for`, l'uso del condizionale `if`.

Nella sezione che segue vedremo invece un esempio un po' più articolato incentrato sulla generazione di una fattura, un tipo di documento che per sua natura potrebbe richiedere un'elevata complessità algoritmica, per meglio mettere in luce quali vantaggi si possano ottenere dall'usare insieme Jinja e LaTeX in un contesto pratico.

4. Un esempio: generazione di una fattura

In questa sezione vediamo un esempio relativamente più complicato: la generazione di una fattura. Cercheremo comunque di mantenere un livello piuttosto semplice, focalizzandoci unicamente sul *template* e sull'interazione con Python, senza porre in questa sede particolare attenzione alla resa tipografica del documento.

In un contesto reale è facile immaginare che i dati necessari per compilare la fattura sarebbero estratti da un *database* tramite una *query* opportuna. Ai fini di questa dimostrazione, non ha importanza soffermarci su come lo *script* Python recuperi i dati di interesse (ad esempio, se tramite una connessione diretta al database, oppure tramite una richiesta HTTP a un server, o ancora prelevandoli da un foglio di calcolo in formato Excel o OpenDocument usando il pacchetto `pyexcel`). Fortunatamente, la ricchezza della libreria standard di Python e l'esistenza di librerie dedicate facilita non di poco questo compito. Per questo esempio,

Template sorgente:

```
\documentclass{article}

\begin{document}
Il cubo di 4 è \PYEXP{cubed(4)}.
%% if myvar == 42
Hai vinto.
%% else
Hai perso.
%% endif

%% for item in mylist
L'autore di \PYEXP{item.title} è
\PYEXP{item.author}.
%% endfor

% normale commento LaTeX
\end{document}
```

Template "compilato":

```
\documentclass{article}

\begin{document}
Il cubo di 4 è 64.
Hai vinto.
L'autore di Dubliners è James Joyce.
L'autore di The Art of Computer
Programming è Donald Knuth.
% normale commento LaTeX
\end{document}
```

Listato 1. In alto: *template* sorgente. In basso: *template* compilato con il codice del listato 2.

supponiamo che i dati siano memorizzati sul disco in un *file* in formato YAML[4] come mostrato nel listato 3. Una volta caricato il *file* usando il pacchetto `PyYAML`, i dati saranno accessibili in Python sotto forma di un *dizionario*.

Per questo esempio useremo il *template* mostrato nel listato 4 e lo *script* mostrato nel listato 5. Il documento finale compilato con LaTeX è mostrato in figura 1. Come si può vedere, il *template* rappresenta una lettera con diversi elementi che sono riempiti dinamicamente tra cui: il numero e la data della fattura (riga 6), il nome e l'indirizzo del destinatario (righe 9–10),

4. YAML, acronimo ricorsivo per "YAML Ain't Markup Language", è un formato per la serializzazione dei dati pensato per essere facilmente leggibile e modificabile dagli esseri umani. Più probabilmente, nelle comunicazioni tra differenti servizi, quindi tra macchine, i dati sarebbero serializzati in formato JSON: JavaScript Object Notation.

```
from jinja2.nativetypes import \
    (NativeEnvironment, Environment)
from jinja2.loaders import \
    FileSystemLoader

env = Environment(
    loader=FileSystemLoader('.'),
    variable_start_string='\PYEXP{',
    variable_end_string='}',
    line_statement_prefix='%%',
)

def cubed(a):
    return a**3

env.globals.update(cubed=cubed)

mylist = [
    {
    'author': 'James Joyce',
    'book': 'Dubliners',
    },
    {
    'author': 'Donald Knuth',
    'book': 'The Art of Computer '
            'Programming',
    },
]
myvar = 42

templ = env.get_template('template.tex')
templ = templ.stream(
    mylist=mylist,myvar=myvar)
templ.dump('output.tex')
```

Listato 2. Esempio di *script* Python per compilare il *template* Jinja mostrato nel listato 1.

le righe della tabella contenenti il dettaglio dei libri acquistati (righe 18–24). Si possono notare gli *statement* `for` e `if` rispettivamente alle righe 18 e 21, evidenziate in azzurro. In particolare il condizionale `if` viene usato per aggiungere alla tabella una voce riguardante l'eventuale confezione regalo. All'interno delle espressioni, demarcate da `\PYEXP{}`, figurano chiamate a funzioni Python (riga 10) e operazioni matematiche (riga 20).

Passando adesso allo *script* Python, si possono notare alcuni blocchi di codice già incontrati in precedenza, come la personalizzazione di Jinja alle righe 15–20. Inoltre:

```yaml
invoice:
  number: 42
  date: 2021-03-28
customer:
  name: Mario Rossi
  address: |
    Viale delle Idee, 1
    50100 Firenze
books:
  - title: The Art of Computer Programming
    author: Donald Knuth
    price: 65.49
    gift: false
    quantity: 2
  - title: 'Just for Fun: The Story of an Accidental Revolutionary'
    author: Linus Torvalds
    price: 14.64
    gift: true
    quantity: 1
  - title: The Cathedral and the Bazaar
    author: Eric S. Raymond
    price: 9.24
    gift: false
    quantity: 1
```

Listato 3. Un esempio di documento YAML con i dati per la generazione di una fattura.

- alla riga 7 si imposta il *locale* italiano.
- alle righe 9–13 si definiscono due funzioni ausiliarie `P()` e `D()`, rispettivamente per la formattazione dei prezzi e delle date. Come abbiamo già visto, alla riga 30 queste vengono "registrate" nell'ambiente Jinja per poter essere usate direttamente nel *template*.
- alle righe 22–23 si carica il documento YAML mostrato nel listato 3 e contenente i dati per la fatturazione.
- alle righe 25–28 si calcola il costo totale della fattura, sommando i prezzi dei singoli articoli e aggiungendo le commissioni per le confezioni regalo.
- alla righe 32–35 si definisce un *dizionario* di variabili aggiuntive che vogliamo rendere disponibili nel *template*.
- alla riga 38 si "compila" il template, passandogli tutte le variabili di cui abbiamo bisogno. Si notino i doppi asterischi con cui si fa il cosiddetto *unpacking* dei due *dizionari* `data` e `my_vars`. In questo modo le tre *chiavi* del *dizionario* corrispondente ai dati del listato 3 (`invoice`, `customer` e `books`), insieme alle due *chiavi* del *dizionario* `my_vars`, sono direttamente disponibili nel *template* come se fossero variabili singole.

```
\documentclass[DIV=13]{scrletter}
\usepackage{eurosym} \let\texteuro\euro
\date{}
\setkomavar{fromname}{The GuIT Bookshop}
\setkomavar{fromaddress}{Via Claudio 21 \\ 80125 Napoli}
\setkomavar{title}{Fattura n. \PYEXP{invoice.number} del
  \PYEXP{D(invoice.date)}}

\begin{document}
  \begin{letter}{\PYEXP{customer.name} \\
  \PYEXP{customer.address.replace('\n', ' \\\\ ')}}
    \opening{}
    \footnotesize
    \begin{tabular}{lrrr}
      \hline
      \textbf{Titolo} & \textbf{Prezzo unitario} & \textbf{Quantità}
      & \textbf{Totale} \\
      \hline
      %% for book in books
        \PYEXP{book.title} & \PYEXP{P(book.price)} &
        \PYEXP{book.quantity} & \PYEXP{P(book.price * book.quantity)}\\
        %% if book.gift
          \hfill\textit{confezione regalo} & & &
            \PYEXP{P(gift_wrap_price)} \\
        %% endif
      %% endfor
      \hline
      \textbf{Totale} & & & \textbf{\PYEXP{P(total)}} \\
      \hline
    \end{tabular}
  \end{letter}
\end{document}
```

Listato 4. Un semplice *template* per una fattura.

5. Conclusioni

In questo articolo abbiamo visto come usare Jinja per la generazione dinamica di documenti LaTeX. Jinja è un motore ricco di funzionalità per le quali si rimanda alla sua documentazione ufficiale che è molto ben scritta. Per quanto riguarda la sintassi da usare all'interno dei *template* si può fare riferimento alla *Template Designer Documentation*[5]; per quanto riguarda l'interfaccia di programmazione Python si può fare riferimento alla *API Documentation*[6].

5. https://jinja.palletsprojects.com/en/2.11.x/templates/
6. https://jinja.palletsprojects.com/en/2.11.x/api/

```python
import locale
import yaml

from jinja2.nativetypes import (NativeEnvironment, Environment)
from jinja2.loaders import FileSystemLoader

locale.setlocale(locale.LC_ALL, 'it_IT.utf8')

def P(x):
    return locale.currency(x)

def D(x):
    return x.strftime('%d %B %Y')

env = Environment(
    loader=FileSystemLoader('.'),
    variable_start_string='\PYEXP{',
    variable_end_string='}',
    line_statement_prefix='%%',
)

with open('db.yaml', 'r') as f:
    data = yaml.safe_load(f)

gift_wrap_price = 2

total = sum([book['price'] * book['quantity'] for book in data['books']])
total += sum([book['gift'] for book in data['books']]) * gift_wrap_price

env.globals.update(P=P, D=D)

my_vars = {
    'total': total,
    'gift_wrap_price': gift_wrap_price,
}

template = env.get_template('invoice.tex')
template.stream(**data, **my_vars).dump('output.tex')
```

Listato 5. *Script* per compilare il *template* mostrato nel listato 4.

The GuIT Bookshop
Via Claudio 21
80125 Napoli

The GuIT Bookshop, Via Claudio 21 , 80125 Napoli

Mario Rossi
Viale delle Idee, 1
50100 Firenze

Fattura n. 42 del 28 marzo 2021

Titolo	Prezzo unitario	Quantità	Totale
The Art of Computer Programming	€ 65,49	2	€ 130,98
Just for Fun: The Story of an Accidental Revolutionary	€ 14,64	1	€ 14,64
confezione regalo			€ 2,00
The Cathedral and the Bazaar	€ 9,24	1	€ 9,24
Totale			**€ 156,86**

Figura 1. Esempio di fattura generata con Jinja.

Riferimenti bibliografici

Giacomelli, Roberto e Gianluca Pignalberi (2015). «Generare documenti LaTeX con diversi linguaggi di programmazione». *ArsTEXnica*, p. 40.

Giacomo Mazzamuto
Consiglio Nazionale delle Ricerche – Istituto Nazionale di Ottica
(CNR-INO)
g.mazzamuto+ctan@gmail.com