Universidade do Minho
Escola de Engenharia

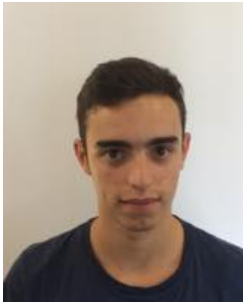
Introdução ao Processamento de Linguagem Natural

Mestrado Integrado em Engenharia Informática

4º ANO, 1º Semestre, 2019/2020

Estudo da Biblioteca Jinja2

Segundo trabalho prático



Diogo Sobral
A82523

Maria Dias
A81611

November 30, 2019

Índice

1 Introdução

Vivemos numa era de puro desenvolvimento em que o mundo não para. Todos os dias surgem novos desafios e novas soluções prontas a inovar e auxiliar tudo o que já existe. Na área tecnológica, principalmente, esta tendência faz-se notar muito intensamente.

O Jinja2 é uma biblioteca moderna, desenvolvida em Python e destinada à construção de templates de forma rápida e cómoda para os seus utilizadores. Esta biblioteca é baseada nos templates do framework Django e pode ser facilmente integrada em projetos de desenvolvimento web. O Jinja é usado sobretudo para a construção de templates HTML e XML.

Neste trabalho prático iremos não só estudar algumas das ferramentas disponibilizadas por esta biblioteca, como também a sua estrutura e possíveis formas de a utilizar no contexto do processamento de linguagem natural.

2 A ferramenta

O Jinja é uma linguagem de templates e, como todas as linguagens, existe uma sintaxe própria que deve ser dominada de modo a conseguir tirar partido de todo o potencial da biblioteca. Esta secção será focada em compreender como é que podemos utilizar as funcionalidades disponíveis.

Esta linguagem recorre a *delimiters* específicos nos ficheiros onde é utilizada para conseguir identificar o objetivo pretendidos. Estes *delimiters* podem ser:

- **{%....%}**: utilizado para escrever um *statement*.
- **{{exp}}**: utilizado para imprimir o valor da expressão no output to template.
- **{#....#}**: utilizado para escrever comentários que não serão incluídos no output.

Ao utilizar as expressões acima descritas, conseguimos criar ficheiros dinâmicos com os valores que pretendemos. Para isto, juntamente com o template a usar, é necessário utilizar um dicionário com o valor pretendido para cada expressão. De seguida iremos ver como é que os *delimiters* são utilizados.

2.1 Expressões

Como visto na secção anterior, uma expressão serve para imprimir o valor no output do template. Abaixo temos um pequeno exemplo de como tudo isto funciona.

```
>>> tpl = Template(u'A variável i tem o valor {{i}}')
>>> tpl.render({'i':10})
'A variável i tem o valor 10'
```

Figura 1: Exemplo simples para uma expressão

É de notar que o valor a substituir pode ser qualquer coisa desde que possa ser introduzido no dicionário a passar à função render.

Caso o valor passado no dicionário seja um objeto é ainda possível imprimir no template os atributos do mesmo utilizando 2 nomenclaturas distinta. A primeira recorrendo ao uso do **.** (i.val) ou i['val'].

```
>>> t = Test()
>>> t.val
1
>>> tpl = Template(u'A variável i tem o valor {{i.val}}')
>>> tpl.render({'i':t})
'A variável i tem o valor 1'
```

Figura 2: Exemplo para substituir pelo atributo do objeto.

2.2 Statements

Conseguir utilizar e calcular o valor de expressões é essencial para um template, todavia disponibilizar apenas expressões seria muito limitador. Para combater este problema, existem os statements que nos permitem fazer inúmeras coisas como ciclos *for*, *while*, *if* ou, até mesmo, utilizar hierarquias para reutilizar templates na criação de novos.

2.2.1 For

```
>>> tpl = Template(u'{% for k in seq %} <a>{{k}}<a/> {% endfor %}')
>>> seq = [1,2,3,4,5]
>>> tpl.render({'seq':seq})
' <a>1<a/>  <a>2<a/>  <a>3<a/>  <a>4<a/>  <a>5<a/> '
```

Figura 3: Exemplo da utilização do for para templates.

A figura acima mostra um exemplo simples para a utilização do bloco *for*. Existe um conjunto de primitivas que podem ser utilizadas em conjunto com este bloco e que visam tirar proveito de algumas propriedades de um ciclo. As primitivas podem ser consultadas na documentação.

2.2.2 If

```
>>> t = Test()
>>> t.val
1
>>> t.inc()
>>> t.val
2
>>> tpc = Template(u'Valor de t = {{t.val}}\\n {% if t.val == 1 %} {{t.val}} {% else %} None {% endif %}')
>>> tpc.render({'t':t})
'Valor de t = 2\\n  None '
```

Figura 4: Exemplo da utilização do if para templates.

O Exemplo abaixo mostra como é que podemos usar as duas primitivas de modo gerar uma página HTML para mostrar uma lista de utilizadores.

```
<!DOCTYPE html>
<html>
<body>
    <p>Tabela de Utilizadores</p>
    <table>
        <tr>
        {% for col in columns %} <td>{{col}}</td>{% endfor %}
        </tr>
        {% for user in users %}
        <tr>
            <td>{{user.id}}</td> <td>{{user.name}}</td>
            {% if user.sex == 'M' %}<td>Male</td>{% else %}<td>Female</td>{% endif %}
        </tr>
        {% endfor %}
      </table>
</body>
</html>
```

Listing 1: Template Input

```
<!DOCTYPE html>
<html>
<body>
    <p>Tabela de Utilizadores</p>
    <table>
        <tr>
         <td>Id</td> <td>Nome</td> <td>Sexo</td>
        </tr>

        <tr>
            <td>1</td> <td>Banshee</td>
            <td>Female</td>
        </tr>

        <tr>
            <td>2</td> <td>Addams</td>
            <td>Female</td>
        </tr>

        <tr>
            <td>3</td> <td>Golias</td>
            <td>Male</td>
        </tr>

      </table>
</body>
</html>
```

Listing 2: Output gerado

Tabela de Utilizadores

Id	Nome	Sexo
1	Banshee	Female
2	Addams	Female
3	Golias	Male

Figura 5: Output html produzido.

2.2.3 Extend

Outra das grandes funcionalidades do jinja é permitir uma hierarquia parecida com a hierarquia de classes. Um template base marca alguns blocos de forma especial. Os templates que implementam estes templates bases podem escolher se querem substituir os blocos marcados ou adicionar

informação aos mesmos. Para isto utilizamos as tags {% block %} e {% extends "base.html" %}. No caso de querermos renderizar o conteúdo de blocos definidos no template "pai", usa-se a expressão {{ super() }}.

```
<!DOCTYPE html>
<html>
<head>
    <title>{% block title %} Base {% endblock %}</title>
</head>
<body>
    {% block tabname %}
    <p>Tabela de Utilizadores</p>
    {% endblock %}
    <table>
        <tr>
        {% for col in columns %} <td>{{col}}</td>{% endfor %}
        </tr>
        {% for user in users %}
        <tr>
            <td>{{user.id}}</td> <td>{{user.name}}</td>
            {% if user.sex == 'M' %}<td>Male</td>{% else %}<td>Female</td>{% endif %}
        </tr>
        {% endfor %}
      </table>
</body>
</html>
```

Listing 3: Template base utilizado - base.html

```
{% extends "base.html" %}
{% block title %}
    {{super()}} - Sou o filho
{% endblock %}
{% block tabname %}
    <h2>Template do Filho</h2>
{% endblock %}
```

Listing 4: Template que implementa o base

```
<!DOCTYPE html>
<html>
<head>
    <title>Base  - Sou o filho</title>
</head>
<body>
    <h2>Template do Filho</h2>
    <table>
        <tr>
         <td>Id</td> <td>Nome</td> <td>Sexo</td>
        </tr>
        <tr>
            <td>1</td> <td>Banshee</td>
            <td>Female</td>
        </tr>
        <tr>
            <td>2</td> <td>Addams</td>
            <td>Female</td>
        </tr>
        <tr>
            <td>3</td> <td>Golias</td>
            <td>Male</td>
        </tr>
      </table>
</body>
</html>
```

Listing 5: Output Gerado

Template do Filho

Id	Nome	Sexo
1	Banshee	Female
2	Addams	Female
3	Golias	Male

Figura 6: Output produzido

2.2.4 Include

O último bloco que vamos falar é o include. Este tipo de bloco permite a integração de templates já construídos noutros, quer sejam estáticos ou possam receber argumentos. Isto permite a reutilização de trabalho previamente desenvolvido e acelera, muitas vezes, o processo de desenvolvimento. Para o utilizar, basta incluir a tag {% *include* nome_do_template %} no sítio onde queremos que esse template apareça. É de notar que esse template tem acesso ao conjunto de variáveis passadas no render. Em conjunto com este, podemos também usar o statement *with* de forma a introduzir valores de variáveis num template. Segue um exemplo desta aplicação, em que um template para um *form* recebe dois argumentos que lhe são passados

```
<form role="form" method="POST" action="/">
    <div class="form-group">
      <input type="text" name="url" class="form-control" placeholder="{{ placeholder }}"
          style="max-width: 400px;">
    </div>
    <button type="submit" class="btn btn-default">{{ button }}</button>
</form>
```

Listing 6: Template form.html

```
(...)
{% with button="Calcular", placeholder="Inserir URL" %}
    {% include "form.html" %}
{% endwith %}
(...)
```

Listing 7: Template que inclui form.html

3 Estrutura de um projeto Jinja

O *Jinja2* implementa todas as funcionalidades acima descritas e muitas mais segundo duas API fornecidas. A *High Level API* é a que devemos utilizar caso apenas pretendamos criar templates e desenha-los. Caso o nosso objetivo seja aprofundar mais os nossos conhecimentos ou a construção de extensões devemos utilizar a *Low Level API*. Neste trabalho apenas estudamos a *High Level API*.

3.1 Classe Environment

A classe *Environment* é classe core do jinja. As instâncias desta classe são responsável por guardar toda a informação necessária que é utilizada para encontrar e carregar os templates do sistema de ficheiros para memória. A maior parte das aplicações, geralmente, criam uma instâncias deste objeto na inicialização das mesmas. Este tipo de objetos pode ser inicializado, na pasta em que nos encontramos, da seguinte forma:

```
env = jinja2.Environment(
    loader=jinja2.FileSystemLoader(searchpath="./"),
    autoescape=jinja2.select_autoescape(['html', 'xml'])
)
```

Após termos uma instância desta classe para obtermos um template no path que indicamos basta usar o método *get_template(nome do template)*.

3.2 Classe Template

A classe template é utilizada para gerar toda a informação que pretendemos. Esta classe pode ser inicializada com uma simples *string* ou através de um ficheiro usando a classe *Environment*. Como vimos anteriormente, é necessário um dicionário com todas a variáveis para conseguir transformar o template no produto final. Assim, a classe template possuí um método chamado *render* que é responsável por, dado um determinado dicionário, fazer todas as alterações necessárias no template.

4 Implementação no contexto de NLP

Após fazer um estudo da biblioteca *Jinja2*, passamos a investigar formas de a utilizar no contexto do processamento de linguagem natural.

À partida, não vemos uma ligação direta entre esta ferramenta e o processamento de texto. No entanto, sabemos que *Jinja2* é maioritariamente utilizado em conjunto com a framework *Flask*, representando o seu mecanismo de templates padrão. Tirando partido disso, decidimos implementar uma aplicação web *Flask* que conta as dez palavras que mais se repetem num dado site. Para inovar e expandir os nossos conhecimentos também no que toca ao processamento de texto, decidimos experimenta o framework *NLTK* para separar texto em palavras e fazer a contagem da frequência com que surgem.

4.1 Guia de Utilização

Para correr a aplicação, é necessário primeiro instalar alguns pacotes. O comando abaixo instala automaticamente todas as dependências que possam existir no projeto.

```
$ pip3 install -r requirements.txt
```

De seguida, na shell do Python, é necessário correr os comandos que se seguem.

```
>>> import nltk
>>> nltk.download()
```

Na janela do downloader NLTK que surge, no separador *Models*, seleciona-se e instala-se a ferramenta *punkt*. Posto isto, estamos prontos para correr a aplicação. Para isso, basta usar o comando

```
$ python3 contapal.py
```

e de seguida aceder ao endereço http://127.0.0.1:5000/ num novo separador. Nesta página surgirá então a nossa aplicação, na qual se pode inserir o URL de qualquer página da internet e obter a contagem das palavras mais frequentes.

4.2 Aplicação *Conta Palavras*

A aplicação surge com uma interface bastante minimalista, surgindo uma caixa de texto onde se pode inserir o URL do site para o qual se pretende fazer a contagem de palavras. Na segunda figura vemos já o resultado obtido para um site específico. A aplicação apresenta os 10 termos mais referidos na página em questão.

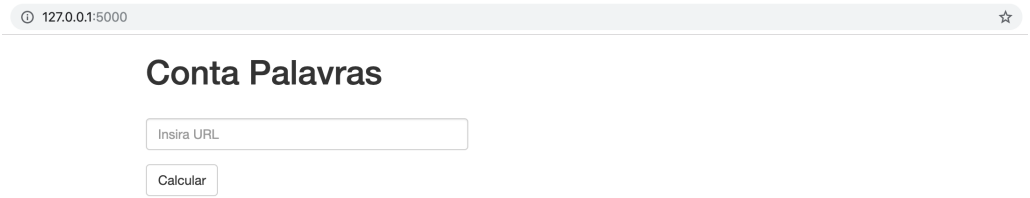


Figura 7: Aplicação Conta Palavras

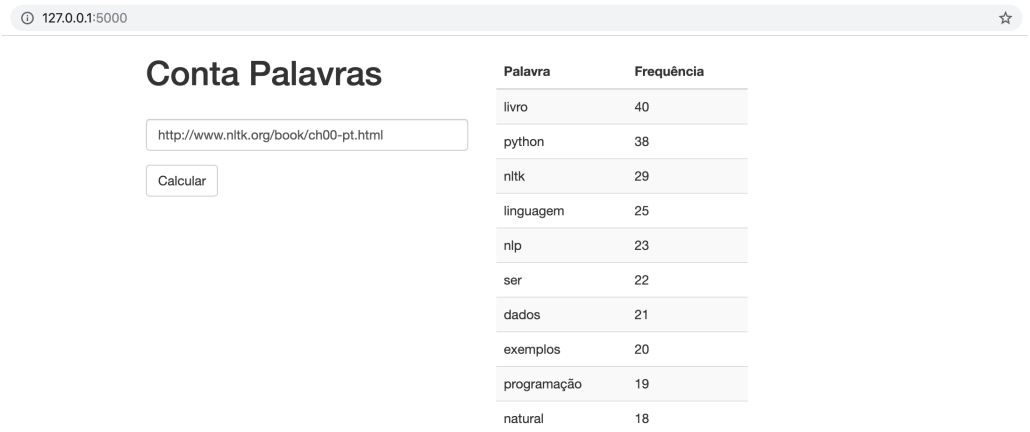


Figura 8: Exemplo de utilização da aplicação

4.3 Implementação

4.3.1 Flask

Uma vez que estamos a usar *Flask*, o *engine* do Jinja2 é configurado automaticamente. Isto altera um pouco aquilo que seria inicialmente a estrutura do nosso projeto, pelo que para usar um dado template basta apenas chamar o método `render_template()`, fornecendo como argumentos o nome do template e, opcionalmente, outras variáveis que seja necessário passar ao mesmo.

Em termos da aplicação propriamente dita, o nosso programa consiste apenas de uma função *index* que trata do comportamento de toda a app, uma vez que temos apenas de uma página html. A função aceita os métodos GET e POST do HTML, pois temos um *form* no qual inserimos dados e retiramos informação. A aplicação inicia-se com a inserção de um URL no *form*. A partir deste endereço conseguimos aceder aos conteúdos do site e fazer o parse do formato HTML para texto com a ajuda da biblioteca BeautifulSoup. Tendo extraído o texto do site, passamos então ao seu processamento, que é feito com NLTK.

4.3.2 NLTK

A biblioteca de código aberto NLTK (Natural Language Toolkit) surge com várias funcionalidades que facilitam o processamento de texto. Na nossa aplicação, usamos o NLTK para dividir o texto cru em palavras e, posteriormente, retirar as *stopwords* ou, em português, palavras vazias - expressão que designa palavras usadas, por exemplo, como conectores de frase e que não têm conteúdo significativo.

Na lista resultante é feita a contagem das ocorrências de cada palavra, que fica guardada num dicionário palavra → num_ocorrências.

4.3.3 Jinja

No que toca à apresentação em formato html, a nossa aplicação divide-se em três templates, `base.html`, `index.html` e `form.html`. O primeiro é hierarquicamente o template "pai", pois apresenta o esqueleto de qualquer página que faça parte da aplicação. Nesse template, existe um *block statement*, que corresponde ao conteúdo que varia de página para página.

No ficheiro `index.html`, que é uma extensão do anterior, surge o corpo da página da nossa aplicação, no qual se faz a inclusão do ficheiro `form.html`, esqueleto do *form*, passando-lhe as strings que ficam no botão de submissão e no placeholder do mesmo. Este template inclui também um *for statement* que itera pelos resultados (se existirem - implica aqui um *if statement*) para desenhar a tabela da contagem das ocorrências das palavras.

Como forma de experimentar o processamento de texto dentro da própria ferramenta *Jinja*, decidimos passar parte do processamento que originalmente era feito na aplicação para os templates. Para isto, fizemos proveito de alguns filtros embutidos no *Jinja*, entre os quais `dictsort` e `reverse`, os quais podem ser usados, respetivamente, para ordenar o dicionário resultado pelo número de ocorrências da palavra e por ordem descendente. Para fazer este processamento podemos seguir um de dois formatos diferentes:

```
{% for result in results|dictsort(by='value', reverse=true) %}
    (...)

ou

{% for result in results|dictsort(by='value')|reverse %}
    (...)
```

5 Conclusão

A realização deste projeto trouxe a descoberta de ferramentas de Python com os propósitos mais variados. Não só pudemos aprender o funcionamento da biblioteca Jinja2, como também nos foi despertada a curiosidade para várias das outras ferramentas que foram distribuídas pelos restantes colegas, pelo que nos esforçámos por integrar neste projeto algumas delas.

A partir do breve contacto que tivemos com a nossa ferramenta, foi-nos possível construir uma aplicação que, apesar de simples, demonstra o seu potencial no desenvolvimento de programas e no auxílio do processamento de linguagem natural, apesar de inicialmente nos ter parecido que em pouco ou nada seria útil no contexto de NLP.

De uma forma geral, terminamos este trabalho prático confiantes de que o seu objetivo foi cumprido e que todas as alíneas do enunciado foram resolvidas de forma pertinente. Para além disso, consideramos que este trabalho permitiu explorar o lado criativo do grupo, sendo que, a partir do assunto base fornecido, o grupo esforçou-se em imaginar diferentes formas de implementar o NLP no projeto e de experimentar a ferramenta atribuída.