Jinja the SQL way of the Ninja

Using Jinja templates and Python to generate re-usable SQL queries

Julien Kervizic Follow
Oct 9 · 5 min read ★

Photo by Anton Danilov on Unsplash

There is many use for dynamically generating SQL queries such as increased code legibility or re-use. Jinja is a Python templating engine that can be used for just that purpose.

Introduction to Jinja Templates

What is a Jinja Template

Jinja is a templating engine for Python similar to twig, or Django templates. Jinja templates are traditionally used in HTML/web development for the creation of views using Flask as the web-framework. Other Python software such as Ansible also leverage Jinja as templating engine.

Templating engines are often used to help with the separation of concerns within MVC (Model View Controller) to separate the data layer, business logic and the presentation layer of the information.

What features are available

Jinja templates offer some basic programming functionalities, such as Variable substitutions, for loops, functions calls, filters, as well as the ability to extend base components.

Variable substitution: Variables can be substituted within templates by using a double curly brace around the variable name `{{ myVariable }}`

Tags: ForLoops and control flow (if/else)can be included within the template using tags. Tags come in the following syntax `{% tag %}` , the for loop is for instance implemented in the following way: `{% for i in myList %} {{i}} {% endfor %}`

Filters: filters can be used to perform some operations on variables, such as defining default value: `{{ myVariable|default('undefined') }}` , lower or uppercasing: `{{ myVariable|lower }}` , trim or truncate: `{{ myVariable | truncate(10)}}` , or join a list of string `{{ myList | join(',' }}`.

Function calls: It is possible to call functions defined elsewhere in python call within Jinja templates. This can be done in multiple different ways, either by registering the function as a filter or a global template variable, in Flask using a `context_processor` , or simply passing the function as an argument in the Template rending step.

Extension: Jinja templates supports template inheritance, which allows templates to be extended. This is particularly useful for web development where a base template, usually containing a header and footer needs to be included in every rendering.

Jinja Templates in Data Engineering

How they can be used in data engineering

Jinja template can help in implementing *Dynamic SQL*. Dynamic SQL helps generalize the SQL code that is being created. In Python, there are

alternative to Jinja templates such as use building string statements to achieve this goal, but there are a few advantages of using Jinja templates over built in string interpolation and standard SQL queries:

Increased legibility: Having the code used used for the generation of the sql query in a template, increase legibility compared to building string statements, which might require different functions to generate part of the SQL statements. With a template these different components are rendered inline based on the initial variables input. In complex query situation, the use of loops and other types of abstraction with Jinja can increase the legibility of the queries over typing/copy pasting multiple times over like you would do in a normal SQL statement.

Easily importable: The template can easily be imported as a single file that once rendered will provide a SQL query to run. It allows for clear separation of code in a MVC like manners.

Web developer on-boarding: Since the template style is heavily used in web development, using the same pattern to build Dynamic SQL statement can make it easier to onboard those with a background of web development to the project.

Base Example

The following Jinja template, represents a basic SQL aggregation query:

```
from jinja2 import Template
sql_template = Template("""
SELECT
        {{query_id}} AS query_id
{% for column in columns %}
    {% for key in column %}
        ,a.{{key}} AS {{column[key]}}
    {% endfor %}
{% endfor %}
        ,COUNT(1) AS _count
FROM {{table_name}} a
GROUP BY
        1,
{% for column in columns %}
    {% for key in column %}
        ,a.{{key}}
    {% endfor %}
{% endfor %}
""")
```

jinja_template_sql_query.py hosted with ❤️ by GitHub view raw

It requires a list of columns in a dictionary containing the name of the column as key and its alias as value, as well as the name of the table on which to group by and the query id.

In order to generate the output of template, we need to invoke the render function of the template and provide these parameters:

```
columns = [
    {'id': 'user_id'},
    {'date': 'ds'},
    {'gender': 'gender'},
    {'firstName': 'first_name'},
    {'lastName': 'last_name'}
]

output_template = sql_template.render(
    query_id=102,
    table_name='test_02',
    columns=columns
)
```

jinja_template_render.py hosted with ❤ by GitHub view raw

Ignoring blank lines, the output of the template would be the following SQL query:

```
SELECT
        102 AS query_id
        ,a.id AS user_id
        ,a.date AS ds
        ,a.gender AS gender
        ,a.firstName AS first_name
        ,a.lastName AS last_name
```

```
        ,a.lastName AS last_name
        ,COUNT(1) AS _count
FROM test_02 a
GROUP BY
        1,
        ,a.id
        ,a.date
        ,a.gender
        ,a.firstName
        ,a.lastName
```

jinja_sql_output.sql hosted with ❤️ by GitHub view raw

Reading from a file

Jinja templates allows for a clear separation of the template code from the data fetching. You can setup the Jinja templates as separate piece of code and initialize them in the following manner:

```
with open('template.sql.j2') as f:
    sql_template = Template(f.read())
```

jinja_from_file.py hosted with ❤️ by GitHub view raw

Leveraging Jinja in Full

Joins and custom filters: in the previous example, I used a for loop to generate the set of columns to use in the select and group by clause. This

required a dummy column to be added (query_id). Using the join filter on a list makes this unnecessary. This can be used with custom filters to give the intended output:

```
from jinja2 import Template

def alias_column(eval_ctx, value=None, indent=None):
    return [k + " AS " + v for k,v in eval_ctx.items()][0]

def name_column(eval_ctx, value=None, indent=None):
    return [k for k,v in eval_ctx.items()][0]

sql_template = Template("""
SELECT
    {{ columns |map('alias_column')| join(',\n') }}

FROM {{table_name}} a

GROUP BY
{{ columns | map('name_column') | join(',\n') }}
""")

columns = [
    {'id': 'user_id'},
    {'date': 'ds'},
    {'gender': 'gender'},
    {'firstName': 'first_name'},
    {'lastName': 'last_name'}
]

sql_template.environment.filters['alias_column'] = alias_column
```

```
sql_template.environment.filters['name_column'] = name_column
output_template = sql_template.render(
    table_name='test_02',
    columns=columns
)
```

jinja_join_and_filters.py hosted with ❤️ by GitHub view raw

Generating the following output:

```
SELECT
    id AS user_id,
    date AS ds,
    gender AS gender,
    firstName AS first_name,
    lastName AS last_name
FROM test_02 a
GROUP BY
    id,
    date,
    gender,
    firstName,
    lastName
```

There are alternative implementations that can achieve the same result, such as the use of `loop.first` or `loop.last` keywords.

Conditional Statements: Jinja allows for the use of conditional statements such as if else, that allows for the inclusion of certain pieces of codes when certain conditions are met.

```
sql_template = Template("""
SELECT
    {{ columns |map('alias_column')| join(',\n') }}

FROM {{table_name}} a
{% if where != None %}
WHERE
    {{where}}
{% endif %}

GROUP BY
{{ columns | map('name_column') | join(',\n') }}
""")

sql_template.environment.filters['alias_column'] = alias_column
sql_template.environment.filters['name_column'] = name_column
output_template = sql_template.render(
    table_name='test_02',
    columns=columns,
    where="category = 'test'"
)
```

jinja_where.py hosted with ❤️ by GitHub view raw

These can be particularly useful when generating SQL queries that may use specific SQL clauses such as in the example above, in which the WHERE clause is optional but will be generated if a where condition is provided.

Other features of Jinja such as template inheritance can further be used to generate certain of aspect of SQL such as query hints.

JinjaSQL

JinjaSQL is a library specifically designed to leverage Jinja templates to generate SQL queries. The library offers two particular feature of interests, *prepared queries* generation and a set of custom filters specifically designed for SQL.

Prepared query: JinjaSql allows to leverage Jinja templates as prepared query statements, the library is able to decompose the template and its arguments into a query and parameters variables that then can be passed to a SQL connection engine.

```
from jinjasql import JinjaSql
j = JinjaSql()

j.env.filters['alias_column'] = alias_column
j.env.filters['name_column'] = name_column

```

```
template = """
SELECT
    {{ columns |map('alias_column')| join(',\n') }}

FROM {{table_name}} a
{% if where != None %}
WHERE
    {{where}}
{% endif %}

GROUP BY
{{ columns | map('name_column') | join(',\n') }}
"""

data = {
    "table_name": 'test_02',
    "columns":columns,
    "where":"category = 'test'"
}
query, bind_params = j.prepare_query(template, data)
```

jinja_sql_prepared_statement.py hosted with ❤️ by GitHub view raw

Custom filters: Another of the feature offered by the JinjaSQL libary is the use of specific custom filters relevant to SQL queries such as the `inclause` or the `sqlsafe` filter.

Summary

Jinja templates is a powerful tool to generate dynamic sql queries, it can enable code re-use, legibility and a clearer separation of concern between the definition of the data jobs and the rest of the application.

. . .

More from me on Hacking Analytics:

- ON the evolution of Data Engineering
- Overview of the different approaches to putting Machine Learning (ML) models in production
- Setting up Airflow on Azure & connecting to MS SQL Server
- Overview of efficiency concepts in Big Data Engineering

Programming Data Engineering Data Big Data Data Science