一、jinja2语法（包括基础格式和函数）

1. {{ ... }} 包含表达式或变量。
2. {# ... #} 则用于注释。
3. {% ... %} 包含语句。
4. 条件语句为：{% if 条件 %} ... {% elif 条件 %} ... {% else %} ... {% endif %}。
5. 循环语句：{% for 元素 in 序列 %} ... {% endfor %}。同时获取键和值：{% for key, value in my\_dict.items() %}。Jinja2 在 for 循环内部提供了一系列有用的循环变量，通过特殊的 loop 对象访问，例如：loop.index（当前迭代次数，从1开始）、loop.first（是否第一次迭代）、loop.last（是否最后一次迭代）等
6. 循环中先使用过滤条件过滤：{% for item in items if item.show %}。
7. 输出模板语法本身（不被解析）：{% raw %}...{% endraw %}。
8. {% set %}用于在模板内部给变量赋值等。
9. 模板继承允许我们定义一个包含通用结构的基础模板，然后由子模板继承并填充差异部分，这个差异部分的层次必须是块，也就是说同名的块会被替换，块就像类里面的函数。{% extends "base\_prompt.jinja" %}。
10. Jinja2 提供了特殊变量self，允许在模板内调用自身的块，以便在同一模板中重复输出块内容，例如 {{ self.user\_message() }} 可在模板内再次渲染自己的某块。
11. 子模板中可以使用 {{ super() }} 调用父模板对同一块的原始内容，这在希望在覆盖块时保留父模板部分内容很有用，可以取得增量修改的效果。
12. 宏相当于模板中的格式化函数，对模板片段进行服用，可以传入默认参数和可变参数，就和python函数一样，内部可以使用控制流、变量等模板语法。
13. format\_message(role, content)定义了函数名和参数，\*\*{{ role }}:\*\* {{ content }}定义了参数的处理逻辑，这块是对参数进行了格式化。

```jaml

{% macro format\_message(role, content) -%}

\*\*{{ role }}:\*\* {{ content }}

{%- endmacro %}

```

1. 利用其它模板的宏，需先把这个模板导入。导入方式：{% import 'components.jinja' as comp %}

引用：{{ comp.button('Help', '/help') }}。

1. include是插入模板内容，单与插入的模板不具有父子关系，插入的模板共享这个模板的上下文。

```yaml

{% include 'header.jinja' %}

... （主体提示内容） ...

{% include 'footer.jinja' %}

```

1. Environment 初始化时使用 StrictUndefined，使未定义变量在任何情况下都抛出错误。环境是在python代码中创建的，先创建环境，王环境中加入模板，才能对这个模板进行渲染。
2. 用点号 (.) 访问变量的属性或字典键，也可以使用下标语法 ([]) 访问，变量在传入的上下文字典中定义。在渲染模板的时候会传入模板需要的上下文字典。
3. 过滤器就是函数可以对变量值进行格式转换和处理。用管道符号 | 将过滤器附加在变量或表达式之后多个过滤器可以链式书写，前一个过滤器的输出将作为下一个过滤器的输入。
4. 带参数的过滤器可以在其后附加括号传入参数，例如：{{ list\_values|join(', ') }} 会用逗号和空格将列表连接成字符串，Jinja2 内置了丰富的过滤器（如default, length, jsonify等）。
5. 自定义过滤器：过滤器本质上是接受输入值并返回新值的Python函数。第一个参数接收管道左侧的值，其余参数来自过滤器调用时提供的参数，定义函数后，需要将其注册到环境的过滤器字典中。
6. 过滤器的定义，就是python函数：def emphasize(value, mark='\*\*'):
7. 过滤器的注册：

```yaml

from jinja2 import Environment

env = Environment()

def emphasize(value, mark='\*\*'):

# 将文本用指定标记包围

return f"{mark}{value}{mark}"

env.filters['emphasize'] = emphasize #注册

```

1. 过滤器的使用：{{ "hello"|emphasize('$$') }}。
2. 自定义全局函数，注册进环境的全局变量中。

```yaml

def calculate\_risk\_level(score):

return "high" if score > 80 else "low"

env.globals['risk\_level'] = calculate\_risk\_level

```

1. 自定义全局函数的使用：Risk level: {{ risk\_level(user\_score) }}。
2. 这种方式对非纯文本处理或需要执行复杂计算/数据库查询的功能特别有用。不过要注意，Jinja2 模板引擎本身并不限制 global 函数的行为，它可以调用任意Python代码。

二、Jinja2 与 Python 的集成

1. 第一步：创建环境
   1. Environment是Jinja2的核心对象，包含配置、已注册的过滤器/函数、以及模板加载器等。通常在应用初始化时创建一个Environment实例。
   2. 这段代码在初始化环境的时候，传入FileSystemLoader('templates')，指明了需要加载的模板所在的目录。

```yaml

from jinja2 import Environment, FileSystemLoader

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

autoescape=False)

```

1. 第二步：加载模板,使用get\_template函数

```yaml

template = env.get\_template("prompt.txt")

```

1. 第三步：渲染模板，在渲染模板的时候，传入模板所需的上下文

```yaml

output\_text = template.render(user\_name="Alice", score=95)

print(output\_text)

```

1. render方法接收的参数既可以是显式的关键字参数，也可以传入单个字典（如template.render(context\_dict)）。渲染返回的是生成的完整字符串，后续可以将其发送给LLM接口。
2. 除了模板文件加载器，还有字典加载器，就可以直接写在pythonb代码里面了。
3. DictLoader（字典加载器）：从一个Python字典中加载模板，使用时需传入一个字典映射，键是模板名称，值是模板源字符串。

```yaml

templates\_dict = {

"greeting.txt": "Hello, {{name}}!",

"farewell.txt": "Goodbye, {{name}}."

}

env = Environment(loader=DictLoader(templates\_dict))

```