

参考学习资料

Python 网页与多轮对话

面向有少量 Python 基础的同学

概念讲解 · 独立示例 · 练习与自测

01 数据与函数

组织信息，理解变量与状态

02 API 与网页

理解请求、表单和函数调用

03 会话与验证

设计自己的程序，检查实际行为

目录

点击目录可跳转到对应章节；也可使用 PDF 阅读器的书签展开各节。

阅读指南	3
01 · Python 数据与函数	5
02 · 理解 API 与单次请求	10
03 · 网页表单与 Flask	14
04 · 会话与上下文的概念	18
05 · 运行环境与排错	21
06 · 自测与学习练习	25
官方资料索引	28

使用方式

先跑通独立的小例子，再思考各部分怎样连接。基础语法可以查第 01 章；遇到环境和运行问题可直接翻到第 05 章。

本册由七份 Markdown 学习文档整理而成。正文保留示例、说明和练习；章节导航已改为 PDF 跳转。代码框中的缩进属于代码的一部分。

导读

阅读指南

适用对象：已经接触过 Python 变量、简单计算和基本输入输出，希望进一步完成网页多轮对话作业的同学。

本资料解释完成作业需要的知识，并提供彼此独立的小例子。你需要自己决定程序如何组织、数据怎样保存、不同部分如何连接。这里没有完整聊天程序，也没有只需填空的作业代码。

阅读顺序

文档	主要问题	学完后应能做什么
01 · Python 数据与函数	怎样组织、读取和更新数据？	使用列表、字典、循环和函数处理小型数据
02 · 理解 API 与单次请求	Python 怎样向模型提问？	分清请求、响应、JSON 和 API Key，读懂单次调用
03 · 网页表单与 Flask	浏览器输入如何到达 Python？	跑通一个独立的小表单，知道 GET、POST、路由和返回页面的作用
04 · 会话与上下文的概念	程序记住了什么，模型又看到了什么？	解释历史、会话、请求上下文的区别，并自行设计会话管理
05 · 运行环境与排错	代码为什么没有运行，网页为什么打不开？	检查解释器、依赖、端口、环境变量和常见错误
06 · 自测与学习练习	如何证明自己理解并完成了功能？	用可观察的现象检查程序，并整理简洁说明

建议先完成第 01、03 章中不需要 API 的小练习，再尝试第 02 章的单次请求。最后阅读第 04 章，自己考虑这些知识怎样应用到作业中。遇到运行问题时随时查第 05 章。

如果某一章已经熟悉，可以先回答章末问题；能解释清楚再跳过。

哪些知识需要掌握

应理解的核心：字符串、列表、字典、普通循环、条件判断、函数参数和返回值、简单异常处理；HTTP 请求与响应；当前会话和上下文的含义。

可以先当作固定库用法学习：Flask 程序对象、路由注册、请求对象、服务启动，以及模型接口规定的字段名。能够解释它们的输入和输出即可，不必先学习框架内部实现。

不作为本资料的前置条件：自定义类、装饰器、异步编程、前端框架、数据库、登录系统、流式输出和精美样式。资料中的朴素网页仍有少量 HTML，因为浏览器需要这些标记来显示表单和文字。

本资料不增加作业要求或评分条目。正式要求以另附的作业描述及课程通知为准。尤其应区分“运行期间保留历史”和“重启后仍然保留历史”，并如实说明自己实现的范围。

怎样使用代码例子

- 除非正文说明需要承接上一段，代码块应独立阅读。不要把不同例子全部粘到同一个文件中运行。
- 标为 python 的代码在 Python 文件中运行；标为 bash 的命令在终端运行。二者不要混用。
- 第 02 章的真实 API 示例需要你自己的有效配置，可能消耗额度；离线 JSON 练习不需要网络。
- 示例中的人名、数据和返回结果都是教学内容。不要把自己的 API Key、私有聊天或个人信息写入提交文件。
- 遇到不懂的一行，先说明它涉及哪个变量、接收什么数据、产生什么结果，再查文档或询问 AI。

用 AI 辅助学习

可以要求 AI 解释一个报错、说明某一行的输入输出，或为一个小函数提出检查用例。得到建议后，要自己运行并核对。

例如：“请解释为什么这两个列表一起变化，并用另一个例子说明”；比“给我一个更复杂的完整项目”更有助于理解当前问题。提交说明中如实记录 AI 的用途和自己的验证。

技术资料核对日期：2026-09-18。接口参数、模型可用性和环境配置以课程实际提供的信息及各章链接的官方文档为准。

01

Python 数据与函数

本章帮助你把“看得懂几行 Python”推进到“能组织一段程序”。先单独运行每个示例，再修改一个输入，观察结果。所有示例都只依赖 Python 自带功能，可以分别保存为 `practice.py`，使用 `python practice.py` 运行。

1. 字符串、数字与转换

字符串用于保存文字，数字用于计算。它们看起来相似，但用途不同：“12”是字符串，12 是整数。从输入框或文件读到的内容经常是字符串，需要转换后才能计算。

PYTHON

```
raw_text = " 12 "  
clean_text = raw_text.strip()  
number = int(clean_text)  
next_number = number + 1  
print("原始内容: ", raw_text)  
print("处理结果: ", next_number)
```

`strip()` 返回去掉首尾空白后的新字符串；`int()` 将合适的内容转换为整数。`float()` 可转换小数，`str()` 可把数字转换成文字。`print()` 的多个参数用逗号隔开，就不必先拼成一条字符串。

检查点：把输入改成 "12.5"，需要换用什么转换函数？

2. 列表：按顺序保存多项内容

列表写在方括号 `[]` 中，位置编号从 0 开始。`append()` 在末尾加入一项，`len()` 返回元素个数。

PYTHON

```
scores = [3, 5]  
scores.append(4)  
first_score = scores[0]  
print(scores)          # [3, 5, 4]  
print(first_score)     # 3  
print(len(scores))     # 3
```

三个元素的合法非负下标是 0、1、2。访问不存在的位置会报错。`append()` 会修改原列表，返回值是 `None`，因此不要写 `scores = scores.append(4)`：这会使 `scores` 不再指向原列表。

3. 字典与嵌套：给数据加上含义

字典用 `{}` 表示，通过“键”找到对应的“值”。键通常是有意义的名称，例如实验编号、单位和测量值。字典的值也可以是列表，于是能表达稍复杂的数据。

PYTHON

```
observation = {
    "name": "温度观测",
    "unit": "摄氏度",
    "values": [20.1, 20.4, 20.2]
}
name = observation["name"]
values = observation["values"]
first_value = values[0]
print(name)
print(first_value) # 20.1
```

这里先从字典中取出列表，再从列表中取出数字。把嵌套取值拆成几个步骤，可以更容易看出每个变量保存什么。字典键名必须一致，"values" 和 "value" 是两个不同的键；缺少键时，直接取值会引发 `KeyError`。

4. 判断与循环：让程序逐步处理数据

`if/else` 根据条件选择操作，`for` 依次处理列表中的元素。冒号后属于该语句的代码必须缩进，建议统一使用四个空格。`=` 用于赋值，`==` 用于比较是否相等。

PYTHON

```
scores = [2, 5, 3, 4]
recommended = []
for score in scores:
    if score >= 4:
        recommended.append(score)
    else:
        print("暂不推荐，评分：", score)
print(recommended) # [5, 4]
```

循环变量 `score` 每次代表一个元素。`recommended = []` 放在循环外，才能积累结果；如果每次循环都创建空列表，之前加入的内容就会丢失。

5. 两个变量不一定代表两份数据

赋值一个列表给另一个变量，不会自动复制数据。两个变量可能指向同一份列表，这通常称为“别名”。

PYTHON

```
original = [10, 20]
other_name = original
other_name.append(30)
print(original) # [10, 20, 30]
```

如果需要两个独立的空列表，就分别创建它们：

PYTHON

```
first_group = []
second_group = []
first_group.append(10)
print(first_group)  # [10]
print(second_group) # []
```

如果需要复制已有的简单数字列表，可以逐项加入新列表：

PYTHON

```
original = [10, 20]
new_values = []
for value in original:
    new_values.append(value)
new_values.append(30)
print(original)  # [10, 20]
print(new_values) # [10, 20, 30]
```

注意：最后一种写法只创建了新的外层列表。若元素本身是列表或字典，加入的仍是同一个内部对象，修改它仍可能影响两边。不要把“新建了列表”理解成“所有内部数据都独立了”。本章练习只使用数字，先掌握外层关系即可。

6. 函数：接收参数，返回结果

函数把一段有明确用途的操作命名。def 定义函数，调用时传入参数，return 把结果交给调用者，并结束本次函数执行。

PYTHON

```
def area_of_rectangle(width, height):
    area = width * height
    return area

result = area_of_rectangle(3, 4)
print(result) # 12
```

width、height 和 area 是这次函数内部使用的名字。print() 负责显示，return 负责交回数据；如果函数只打印却没有返回结果，调用者得到的值是 None。定义函数不会立即执行函数体，执行发生在调用时。

传入列表时也适用上一节的别名规则：函数中对参数调用 append()，会影响调用者传入的那份列表。写函数前先决定它只计算结果，还是还需要修改输入。

7. 初始化位置：数据为什么有时能留下来

函数内部每次执行 records = []，都会新建一个空列表。把某份记录在函数外初始化一次，再让函数修改它，效果不同：

PYTHON

```
measurements = []

def add_measurement(value):
    measurements.append(value)

add_measurement(20.1)
add_measurement(20.4)
print(measurements) # [20.1, 20.4]
```

运行到文件顶部时创建列表，随后两次函数调用修改的是同一份列表。这里只演示一个进程里的数据积累；程序重新启动、重新执行初始化代码，数据仍会清空。

`measurements.append(value)` 是修改现有列表；在函数内部写 `measurements = []` 则是在那里给名字重新赋值，通常会创建局部变量，不能当作同一种操作。初学阶段不必急着使用 `global`，但应清楚区分“建立数据”“读取数据”“修改对象”和“重新赋值”。

你也可以让函数返回计算结果，由调用者决定何时保存。具体采用哪种组织方式，需要自己判断数据希望保留多久，以及哪些函数应有权修改它。

8. 导入库：使用别人已经写好的功能

模块可以理解为一组已封装好的功能。`import` 后使用 模块名.功能名 调用：

PYTHON

```
import math

value = math.sqrt(9)
print(value) # 3.0
```

`math` 是 Python 自带模块，无需另行安装。网页框架等第三方库通常需要安装到当前使用的 Python 环境中。文件名不要取成 `math.py`、`flask.py` 等库名，否则 Python 可能导入自己的同名文件。

9. 异常处理：遇到预期问题时给出反馈

用户输入不一定合法。可以将可能失败的小段操作放进 `try`，使用 `except` 处理明确的异常类型。

PYTHON

```
text = "不是数字"
try:
    value = float(text)
except ValueError:
    print("请输入一个数字。")
else:
    print("转换成功：", value)
```

这段示例会正常结束，并输出提示。将 `text` 改成 `"2.5"`，就会进入 `else`。不要把整个程序都包进一个捕获所有错误的结构；写错变量名等程序问题应保留错误信息，以便定位。与外部服务交互时，失败后也应明确哪些数据已经改变、哪些尚未改变。

动手练习与检查点

1. **评分筛选。** 使用列表 [1, 4, 5, 2], 逐项选出不少于 4 分的评分。检查: 结果应为 [4, 5], 原列表保持不变; 输入空列表时应得到空列表。
2. **观测数据读取。** 自行创建包含实验名称、单位和三个测量值的字典, 分别输出名称、单位和第二个测量值。检查: 能解释每一步取值后变量的类型, 改变一个测量值后输出也相应改变。
3. **函数与非法输入。** 定义将分钟转换为秒的函数, 再处理数字字符串的转换失败。检查: 2.5 分钟对应 150 秒; 非数字文字有清晰提示; 函数返回的结果能继续参与计算。

完成后, 尝试不看代码解释: 列表和字典各适合保存什么? 赋值为什么可能影响同一份数据? 函数返回与打印有什么区别?

官方资料

遇到不熟悉的语法时, 可查阅 Python 官方教程: [数据结构](#)、[控制流与函数](#)、[错误和异常](#)。本章不要求学习其中所有高级写法。

02

理解 API 与单次请求

本章只讨论一次请求，不实现网页或多轮会话。阅读前应能从列表和字典中取出数据。

1. API 是程序之间约定的交流方式

在网页中点击按钮，是人和程序交互；使用 API，是一个程序按规定格式向另一个服务发送数据。

一次模型请求可以分成以下部分：

部分	用途	容易混淆的地方
URL	请求发送到哪里	API 地址不一定等于服务商官网首页
HTTP 方法	进行哪类请求	生成回答通常使用 POST
请求头 headers	携带认证等附加信息	API Key 不是问题正文
请求体 body	说明模型、消息等参数	需要符合接口规定的字段和类型
响应 response	服务端返回的状态与数据	请求成功不代表模型回答一定正确

HTTP 状态码常见含义：200 表示请求成功处理；400 常见于请求参数有误；401 常见于认证失败；429 常见于限流或额度限制；5xx 表示服务端错误。具体含义以所用接口说明为准。

API Key 类似调用服务的凭据。把它发给模型作为“提示词”，不会完成身份认证。

2. Python 字典与 JSON 不是同一种东西

Python 中的字典是内存中的对象；JSON 是传输或保存数据时常用的文本格式。看起来相似，不代表可以完全混用。

下面是一个**离线练习**：

PYTHON

```
import json

measurement = {}
measurement["name"] = "水样 A"
measurement["temperature"] = 23.5
measurement["checked"] = True

text = json.dumps(measurement, ensure_ascii=False)
print(text)

restored = json.loads(text)
print(restored["temperature"])
```

观察打印结果：Python 的 True 在 JSON 文本中变成 true。不要直接把含有 true、null 的 JSON 当作普通 Python 字典代码粘贴。

使用 Requests 时，json=data 可以把 Python 对象编码为 JSON 请求体；response.json() 则尝试把响应正文解析为 Python 对象。它不是“直接取出 AI 的回答”。[Requests 官方快速入门](#)

3. 认识消息格式

本资料以常见的 Chat Completions 接口为例。一条文本消息通常有两项：

PYTHON

```
message = {}
message["role"] = "user"
message["content"] = "请用一句话说明什么是测量误差。"

print(message["role"])
print(message["content"])
```

- role 表示消息的角色，例如 user、assistant、system。
- content 是消息正文。
- 请求中的 messages 是消息列表，即使只问一句话，也不是直接传一个裸字符串。
- model 是服务商提供的模型标识，不能随意起名。
- stream=False 表示等待完整响应。本次基础学习可先不处理流式输出。

不同服务的接口不完全相同，不能只改网站域名就假定兼容。下面只展示 DeepSeek 官方接口的一次文本请求；课程网关若有不同地址或参数，应查课程说明。[DeepSeek 单次调用说明](#)

4. 一个独立的单次请求示例

先阅读第 05 章，设置 DEEPSEEK_API_KEY 和 LLM_MODEL。将下面代码保存为 single_request.py。它只负责发出一条问题，没有网页、历史管理或新建会话。

PYTHON

```
import os
import requests

api_key = os.environ.get("DEEPSEEK_API_KEY", "")
model_name = os.environ.get("LLM_MODEL", "")

if api_key == "" or model_name == "":
    raise ValueError("请先设置 API Key 和模型名称")

headers = {}
headers["Authorization"] = "Bearer " + api_key

question = {}
question["role"] = "user"
question["content"] = "请用一句话介绍量纲分析。"

data = {}
data["model"] = model_name
data["messages"] = [question]
data["stream"] = False

response = requests.post(
    "https://api.deepseek.com/chat/completions",
    headers=headers,
    json=data,
    timeout=30
)
response.raise_for_status()

result = response.json()
choices = result["choices"]
first_choice = choices[0]
message = first_choice["message"]
answer = message["content"]
print(answer)
```

这里有两组不同的数据：data 是发出的请求，result 是收到的响应。字段名由接口约定，变量名则由你决定。

raise_for_status() 在 HTTP 错误状态下抛出异常；timeout 用于限制网络等待，不是模型思考轮数，也不能保证模型一定在这个时间内完成整个请求。上面的例子保留异常，便于先观察真实错误；面向用户的程序应给出可理解的失败提示。[Requests 响应状态与超时](#)

本例不包含重试逻辑。没有确认上一次是否成功前，不宜盲目反复点击或自动重复发送。

5. 不用 API 也能练习响应解析

下面是人为构造的**简化响应**，不是某次真实调用记录：

PYTHON

```
result = {
    "choices": [
        {"message": {"role": "assistant", "content": "先比较各量的单位。"}}
    ]
}

choices = result["choices"]
first_choice = choices[0]
message = first_choice["message"]
answer = message["content"]

print(answer)
```

方括号中的字符串表示字典的键，数字表示列表位置。逐步打印 `type(result)`、`type(choices)`、`type(message)`，比只记住一串下标更有用。

“服务器返回 JSON” 仍可能包含错误信息，不能假定任何 JSON 都有 `choices`。正常业务代码还要考虑缺少字段、空列表、空回答和请求失败。

6. 小练习

1. 为离线测量数据加入一个单位字段，编码成 JSON，再读取回来。检查数值和字符串的类型有没有变。
2. 在简化响应中加入第二个候选回答，分别打印两个回答。哪些地方使用列表下标，哪些使用字典键？
3. 单次请求成功后，只更改问题正文再试一次。说明哪个字段影响问题、哪个字段影响模型选择。
4. 不提交真实请求，先用 `print(data)` 检查自己构造的数据。请求头含有密钥，不能把 `headers` 一起打印或截图提交。

完成检查：你能说清“发到哪里、身份如何认证、问题放在哪里、回答从哪里读出”，而不只是复制一串调用代码。

03

网页表单与 Flask

本章学习把一个普通 Python 函数放到网页后面：浏览器收集输入，Python 处理输入，再把结果交给浏览器。练习只做“输入姓名，显示问候”，先理解网页交互本身。

1. 浏览器和 Python 分别做什么

在终端程序中，`input()` 等待输入，`print()` 显示结果。网页程序的入口不同：用户打开地址或提交表单，浏览器发出 **HTTP 请求**，服务器上的程序处理请求并返回 **HTTP 响应**。

说明 / 输出

浏览器：填写姓名，点击提交

↓ HTTP 请求

Python：读取姓名，产生问候语

↓ HTTP 响应

浏览器：显示新的页面

浏览器负责显示 HTML。HTML 是描述标题、段落、表单等内容的标记，不是 Python 语法。Python 可以把一段 HTML 字符串作为结果返回，因此练习不需要单独编写 JavaScript，也不需要设计样式。

网页程序不需要用 `while True` 反复询问用户是否输入了内容。服务器启动后，等待请求的工作由 Flask 负责；你写的函数只处理当前这一次请求。打开页面会调用一次函数，点击提交又会触发另一次调用，函数中的局部变量也会重新创建。页面仍然显示着文字，不代表 Python 函数还停在那里等待。

本章用到两种请求方法：

方法	本章用途	数据通常在哪里
GET	打开页面、查看结果	地址中的查询参数，如 <code>?name=小林</code>
POST	提交表单，让程序处理输入	请求正文中的表单字段

GET 适合读取信息；会改变程序状态的操作应使用 POST 等合适的方法。表单如何指定方法和发送字段，可查 [MDN 表单数据说明](#)。

2. 看懂最小表单

HTML

```
<form action="/greet" method="post">
  <label>姓名: <input name="name"></label>
  <button type="submit">提交</button>
</form>
```

只需先认识四个约定：

- form 包住一次提交的内容。
- action="/greet" 指定交给当前网站的哪个地址处理。
- method="post" 指定使用 POST。
- name="name" 给输入值起一个字段名，Python 据此查找它。按钮上的“提交”只是界面文字。

需要多行输入时，可把 input 换成 <textarea name="note"></textarea>。相应地，Python 就要读取 note 字段。name 的拼写需要两边一致；只有 id 而没有 name，不能替代表单字段名。

3. 一个可以独立运行的小程序

把下面代码保存为 form_demo.py。不要把文件命名为 flask.py，避免与安装的 Flask 包重名。

PYTHON

```
from flask import Flask, request, redirect, url_for
from html import escape

app = Flask(__name__)

def page():
    name = request.args.get("name", "")
    message = "请填写姓名。"
    if name != "":
        message = "你好, " + name + "! "
    html = '<!doctype html><html lang="zh-CN"><head><meta charset="utf-8">'
    html = html + '<title>问候练习</title></head><body><h1>问候练习</h1>'
    html = html + "<p>" + escape(message) + "</p>"
    html = html + '<form action="/greet" method="post">'
    html = html + '<label>姓名: <input name="name"></label>'
    html = html + '<button type="submit">提交</button></form></body></html>'
    return html

def greet():
    name = request.form.get("name", "").strip()
    if name == "":
        error = "姓名不能为空，请返回填写。"
        return "<p>" + escape(error) + '</p><a href="/">返回</a>', 400
    return redirect(url_for("page", name=name), code=303)

app.add_url_rule("/", view_func=page, methods=["GET"])
app.add_url_rule("/greet", view_func=greet, methods=["POST"])

if __name__ == "__main__":
    app.run(host="127.0.0.1", port=8500, debug=False)
```

先在**自己电脑上已安装 Flask 的环境**中运行：

说明 / 输出

python form_demo.py

再用这台电脑的浏览器访问 `http://127.0.0.1:8500/`。终端保持运行，按 `Ctrl+C` 停止。若 Python 运行在远程服务器，这个地址指的是浏览器所在的电脑，不能直接代替服务器地址；请按 [第 05 章](#) 配置端口转发。这个启动方式用于练习和个人验证。

4. 框架代码不必一次全部吃透

`app = Flask(__name__)` 创建网页应用。可以先把它当成固定的初始化写法，不必先学会编写类。

`app.add_url_rule()` 把路径、允许的请求方法和处理函数对应起来。这里的 `/` 交给 `page`，`/greet` 交给 `greet`。`view_func=page` 没有括号，是把函数交给 Flask；写成 `page()` 则会立即执行它。注册方式见 [Flask add_url_rule 文档](#)。

路径 `/greet` 是你给处理入口起的名字，不表示磁盘上需要有一个 `greet` 文件。文件名、函数名、网页路径可以不同。本例先定义函数，再注册规则，最后启动服务；这样收到请求时，应用已经知道该调用谁。可以试着访问一个没有注册的路径，观察 404；直接在地址栏打开只允许 POST 的 `/greet`，则会得到 405，因为地址栏访问使用 GET。

两个函数都由 Flask 在收到相应请求时调用。`return html` 把字符串送给浏览器；`print(html)` 只会写到运行 Python 的终端，不会把页面交给用户。

重点读懂两种取值：

写法	读取位置	找不到时
<code>request.args.get("name", "")</code>	地址 ? 后面的查询参数	返回空字符串
<code>request.form.get("name", "")</code>	提交的表单正文	返回空字符串

读取位置取决于数据放在哪里，不能把两种写法随意替换。一个 POST 请求的地址也可以带查询参数，此时仍用 `request.args` 读查询参数，用 `request.form` 读表单字段。字段名只是字符串键，浏览器不会直接修改 Python 中的同名变量，需要程序自己取值和赋值。

`request` 表示正在处理的这次请求，因此在处理函数内部读它。表单值是字符串，做数值计算前需要转换类型。空输入先用 `strip()` 去除两端空白，再判断是否为空。示例错误分支返回的 400 是“请求有问题”的 HTTP 状态码。请求取值和返回字符串的约定见 [Flask Quickstart](#)。

5. 文本要作为文本显示

试着提交姓名 `<b>小林</b>`。本例应显示这些字符本身，不应把“小林”变成粗体。这是 `escape(message)` 的作用。

`html.escape()` 将 `<`、`>`、`&` 等字符转成适合放入 HTML 的形式，默认也处理引号。它改变的是显示用的字符串，不会原地修改原值。用字符串拼网页时，用户输入、外部返回的文字，以及可能包含输入的错误提示，都应先转义，再放进 HTML；自己写的 `<p>` 等结构标记不用转义。[Python html.escape 文档](#)

6. 为什么提交后还要跳转

成功提交的流程是：

说明 / 输出

POST /greet → 处理姓名 → 返回 303 跳转 → GET /?name=小林

`url_for("page", name=name)` 根据函数对应的地址生成链接，并处理中文、空格等参数编码。

`redirect(..., code=303)` 告诉浏览器接着用 GET 打开结果页。这样在结果页刷新时，重复的是读取页面，不是刚才的 POST。303 的语义可查 [MDN 说明](#)。

这个问候例子即使重复处理一次影响也很小；如果处理会增加一条记录，重复提交就可能产生重复数据。跳转能处理“成功后刷新”这一常见情况，但不能单独阻止用户快速连点两次提交。本例也不保存姓名，只是通过查询参数把结果带到下一页。查询参数会出现在地址栏中，所以这个办法适合本章的公开练习信息，不适合密码等私密内容。注意，跳转本身不会替你保存数据，它只是让浏览器再访问另一个地址。

7. 自测与小练习

先验证，再修改，每次只改一项：

1. 分别提交中文姓名、空字符串、仅有空格、`<b>小林</b>`，观察结果是否符合预期。
2. 成功提交后刷新，观察地址和终端请求记录。能说明为什么最终是 GET 吗？
3. 将字段名改为 `nickname`，找出必须同步改动的位置。
4. 独立做一个“厘米换成米”的表单：考虑空输入、非数字输入和结果显示。先在普通 Python 函数中验证计算，再接入页面。

能解释“一次输入如何从表单到达函数、结果又如何显示”，就可以继续学习 [第 04 章](#)。无需背下所有框架参数。

04

会话与上下文的概念

网页能显示回答以后，还需要弄清“连续对话”到底依赖什么。本章解释数据之间的关系，不指定完整的数据结构或函数设计。

1. 三件看起来相近的事

概念	指的是什么	不能由它直接推出什么
页面显示的聊天记录	用户在浏览器里能看到的文字	不代表这些文字已经发给模型
程序保留的历史	Python 或存储文件中保留的数据	不代表下一次请求一定选中了正确历史
本次请求的上下文	模型这次实际收到的信息	不代表模型一定正确使用了所有信息

一个页面可以显示十条旧消息，但程序发给模型的只有最新一句话；也可能界面已经清空，程序却仍在发送此前的数据。判断功能是否正确，需要检查这些部分之间是否一致。

本资料讨论需要显式传入 messages 的 Chat Completions 用法。这个接口不会自动读取你的网页或 Python 变量来恢复历史。多轮交流所需的信息需要体现在请求中；其他带服务端会话状态的接口应另查其约定。

[DeepSeek 多轮对话说明](#)

2. 模型为什么会“记得前文”

设想你把几张纸交给一个刚加入讨论的人：

- 第一张写了问题的条件。
- 第二张记录了先前的回答。
- 第三张写着“按照刚才的条件继续”。

如果只交第三张，对方可能不知道“刚才”指什么。模型在一次请求中也只能依据实际收到的信息及它原有的能力作答。

“当前问题引用前文”与“在一个网页里问了很多问题”不是同一判断标准。可以想一组后续问题必须引用早先信息的对话，再观察自己的程序究竟传输了什么。

消息的角色和先后顺序也有意义。用户说了什么、模型答了什么，不应无区别地都标成用户输入。系统提示可以用于说明助手的一般行为，但不能替代用户提出的具体条件。

3. 一段会话需要什么身份

在纸质实验记录中，两组独立实验可以分别放进标有 A、B 的文件夹。记录内容相似，也不意味着可以混放。

程序同样需要某种办法区分“当前正在查看的记录”和“其他记录”。它可以使用编号、名称或其他标识；选择哪种形式，需要你自己决定。

先在纸上回答：

- 什么信息能唯一找到一段历史？这个标识与页面上显示的标题是否相同？
- 用户切换页面时，程序从哪里得知选择了哪段记录？
- 新建时，是创建独立的数据，还是只是改变了一个标题？
- 用户重新打开旧会话时，能否找到它原来的内容？
- 两个浏览器窗口是否可能操作同一份数据？

最后一个问题不是要求实现多用户系统，而是提醒你理解程序的使用范围。单人演示与公共聊天服务的需求不同。

4. 保存多久，是另一个维度

存储位置	通常能保留多久	学习时需要知道的特点
Python 变量	当前进程仍在运行期间	程序重启、崩溃或重新加载时通常丢失
本地文件	文件仍存在期间	要考虑读写格式、位置和失败情况
数据库	数据仍保存期间	需要额外学习，不是理解基础会话管理的前提
浏览器中的存储	取决于浏览器和存储方式	不会自动成为服务器的历史或模型上下文

刷新网页通常只是发起另一次页面请求，不等于重启服务器上的 Python 程序。但程序也可能在处理请求时主动重置数据，所以需要实际检查。

“保留历史记录”与“必须使用数据库”没有必然关系。保存范围以课程要求和你的设计为准，并在说明中写清楚；本资料不额外要求持久化。

5. 什么时候更新记录

设想程序要向远程服务登记一次测量，但网络突然中断：

- 在收到结果以前，输入只是一次待完成操作。
- 收到正常结果以后，才能确认这次操作的结果。
- 失败时，应该能区分原来已经成功的记录与这次没有完成的操作。

这个区别同样适用于聊天程序。需要想清楚：错误发生后，历史是否会留下一个没有对应结果的条目？重试会不会让同一条内容出现两遍？页面显示了什么，实际保存了什么？

不用一开始学习事务或复杂的并发控制，也应该能解释这些现象。先用小数据观察变量如何变化，通常比接入真实模型后猜测更有效。

6. 不把模型的正确率当作唯一证据

下面几种情况都可能发生：

- 没有发送历史，但模型凭常识碰巧答对。
- 发送了完整历史，但模型计算错了。
- 某条信息在最近的模型回答里被复述，即使更早的消息没发送，也仍能答对。
- 两段独立会话用了相同问题，得到相同回答，看起来像“串会话”，实际并未混用历史。

因此，应结合两类证据：**程序实际选择的数据与模型产生的回答**。检查消息正文或数量时不要同时打印带密钥的请求头。

7. 纸面练习与思考

1. 准备两个实验主题：主题 A 是摆长，主题 B 是温度。分别写下三条记录，再画出它们在你的程序里怎样区分。只画数据关系，不急着写完整应用。
2. 假设 A 的问题失败了一次，B 已成功一次。描述此时哪些信息应已保存、哪些只应作为待重试输入。
3. 做出一个能依赖早先条件的后续问题，并解释为什么只看最后一句难以可靠回答。避免把答案或全部条件重复写进后续问题。
4. 写出自己对历史保存范围的决定：刷新、切换、新建、关闭浏览器、重启程序，分别可能发生什么？

检查自己：你应能解释设计如何满足会话隔离，而不仅是“我有一个历史列表”。数据结构、函数分工和界面连接仍需要你自行完成。

05

运行环境与排错

本章用于查问题，不需要一次记住所有命令。先分清你正在操作哪台电脑、哪个终端、哪个 Python。

1. 三个容易混淆的位置

位置	可能做的事情
自己电脑的终端	建立 SSH 连接或端口转发
服务器上的终端	运行服务器里的 Python 文件，设置服务器进程的环境变量
自己电脑的浏览器	访问本地页面或通过转发访问服务器页面

在 SSH 终端里运行 Python，程序就在远端服务器执行。你在本机设置一个环境变量，不会使已经运行的远端程序自动获得它。

浏览器中的 127.0.0.1 指浏览器所在电脑自身；Python 服务中的 127.0.0.1 指运行该服务的那台机器。

2. 确认运行的是哪个 Python

把下面代码保存为 check_python.py，在你准备运行作业的环境中执行：

```
PYTHON
import sys

print(sys.executable)
print(sys.version)
```

终端命令：

```
终端 • BASH
python check_python.py
```

如果课程服务器提供 /opt/miniconda/envs/pubpy/bin/python，也可直接指定它：

```
终端 • BASH
/opt/miniconda/envs/pubpy/bin/python check_python.py
```

路径是环境位置，不是必须背诵的 Python 语法。你分配到的服务器若采用不同路径，按课程环境说明调整。

如果遇到 ModuleNotFoundError，先确认解释器，再检查依赖。可以用同一个解释器执行：

终端 • BASH

```
/opt/miniconda/envs/pubpy/bin/python -c "import flask; import requests; print('依赖可导入')"
```

上面是终端命令，`-c` 后面引号里的才是 Python 代码。不要为了一个导入错误就直接修改公共环境；先核实是不是选错解释器。如果使用自己的环境，可按需要执行 `python -m pip install Flask requests`，使安装工具与解释器对应。

文件也不要命名为 `flask.py`、`requests.py` 或 `json.py`，否则可能遮蔽同名库。

3. 设置 API 配置，不把密钥写进代码

下面命令适用于**服务器上的 Bash 终端**。它们不是 Python 代码，也不是 Windows PowerShell 命令。

把模型名称替换为课程或服务商提供的实际名称：

终端 • BASH

```
export LLM_MODEL="课程提供的实际模型名称"
read -r -s -p '请输入 API Key: ' DEEPSEEK_API_KEY
export DEEPSEEK_API_KEY
printf '\n'
```

输入 Key 时不显示字符。后续从这个终端启动的 Python 程序可以读取这两个环境变量。第 02 章就是这种方式。

检查是否设置成功时，只打印“有没有”，不要打印值：

PYTHON

```
import os

key_is_set = os.environ.get("DEEPSEEK_API_KEY", "") != ""
model_is_set = os.environ.get("LLM_MODEL", "") != ""
print("已设置密钥: ", key_is_set)
print("已设置模型: ", model_is_set)
```

环境变量一般不会由你修改终端后自动同步到已经启动的 Python 进程，需要重新启动该进程。不要把 Key 放进聊天问题、网页源码、README 或截图。环境变量的读取方式见 [Python os.environ 文档](#)。

4. 运行普通脚本与网页程序

普通脚本通常执行完就退出，例如打印几行数据。网页程序启动后会一直等待请求，因此终端没有回到命令提示符，可能是**正常运行**。

第 03 章的小例子保存为 `form_demo.py`。先切换到文件所在目录，再运行：

终端 • BASH

```
/opt/miniconda/envs/pubpy/bin/python form_demo.py
```

用 `Ctrl+C` 停止当前终端前台运行的程序。修改源码后，按示例的启动方式重新运行才会加载新代码；不要把“网页刷新”当成“重新执行 Python 源文件”。

一些文件末尾会有：

PYTHON

```
if __name__ == "__main__":  
    print("这是直接运行文件时执行的代码")
```

这里的作用是区分“直接运行文件”和“被别的文件导入”。初学阶段可以先理解用途，不必马上研究模块系统的全部细节。[Python 主程序入口说明](#)

5. 在自己的浏览器访问服务器网页

假设远端程序监听 127.0.0.1:8500。在**自己电脑的新终端**中运行下面的转发命令，其中 你的SSH别名 应替换为本机已经能正常连接的服务器别名：

终端 • BASH

```
ssh -N -L 127.0.0.1:8500:127.0.0.1:8500 你的SSH别名
```

然后在本机浏览器打开 <http://127.0.0.1:8500/>。

这条命令各部分的含义：

- 第一个 127.0.0.1:8500：在本机开放一个仅本机可访问的入口。
- 后面的 127.0.0.1:8500：由 SSH 服务器一侧连接远端网页程序。
- -N：只做转发，不进入远端命令行。

网页程序和 SSH 转发都要保持运行。转发命令不打印信息、一直占着终端也可能是正常状态。没有 SSH 别名时，可按课程提供的账号、主机和端口使用 -p 参数连接。[OpenSSH 本地端口转发说明](#)

同一服务器上，不同同学的程序不能同时占用同一个监听地址和端口。若改用了远端 8501，本机转发命令中的远端端口也要一起改。VS Code 的“端口”面板也可以完成转发，但最终应核对它给出的本地访问地址。

6. 按层定位，而不是不断重装

现象	先检查什么
SyntaxError、IndentationError	冒号、引号、括号和缩进；是否把终端命令粘进 Python 文件
NameError	变量是否先赋值；变量名是否拼错
TypeError	是否把字符串和数字直接相加；当前对象到底是什么类型
KeyError、IndexError	实际响应有哪些字段，列表有多长；是否假设一定存在答案
ModuleNotFoundError	正在使用哪个解释器；该环境是否有对应库
Address already in use	端口已被占用，可能是自己上一次的程序；选可用端口并同步转发
浏览器 “无法连接”	Python 是否仍运行；地址和端口是否正确；SSH 转发是否仍运行
页面打开但发送报 405	表单提交的方法是否与路由允许的方法一致
API 返回 401	Key 是否有效、是否在运行进程中设置、是否对应当前服务
API 返回 429 或超时	是否频繁请求、额度是否足够、网络和服务是否正常
网页看到旧消息，模型却不知前文	检查本次实际请求的数据，而不只看页面

阅读 Python 报错时，先看最后一行的异常类型和描述，再找自己文件中的行号。只改一个可能原因，然后重新运行，观察现象有没有变化。

如果需要求助，提供“在哪运行、执行了什么、预期什么、实际什么、完整错误类型及相关几行代码”。删除 Key 和其他敏感信息。不要仅说“运行不了”，也不要全部终端历史直接发出去。

自测与学习练习

这些练习帮助你理解并检查功能，不是额外的作业要求或评分标准。先使用小数据和离线例子，确认基础逻辑后再调用模型。

1. 能否独立完成四个小练习

小练习	需要用到的知识	可以观察的结果
记录三次温度观测，并打印每次的名称和数值	列表、字典、循环	三条记录按原顺序输出
从一份人为构造的嵌套响应中读出文字	列表下标、字典键	能解释每一步取出的是什么类型
做一个姓名问候或单位换算的小表单	HTML、Flask、GET/POST	浏览器输入确实到达 Python，并显示相应结果
让一次操作在输入错误时给出提示	条件判断、try/except	错误可解释，之前的有效数据仍能使用

这些小程序不需要与作业采用相同的文件结构，也不需要整合成聊天程序。完成后，用自己的话解释其中一个变量从哪里产生、在哪变化、最后用到了哪里。

2. 检查实际作业时可以观察什么

场景	观察问题
初次打开页面	能否找到输入与发送的位置？空输入会发生什么？
同一会话连续提问	后续问题是否确实依赖前文？实际请求是否包含需要的信息？
新建会话	页面和程序是否都选择了新会话？原会话是否仍可找到？
切回旧会话	能否查看之前内容？继续提问是否仍属于这段会话？
请求失败	有没有提示？当前输入和已完成历史分别怎样变化？
刷新网页	会不会无意中再次调用 API？原来显示的数据是否符合你的设计？
停止并重启程序	历史是否保留？现象是否与你的说明一致？
输入尖括号等文字	是否按普通文字显示，而不是破坏页面结构？

不必把全部检查都写成自动测试。手工操作、观察实际请求和记录结果，也能提供有用证据。无法从界面确认的事情，不要仅凭“看起来正常”下结论。

故障可以通过临时替换**自己程序中的调用函数**，让它报告失败来模拟；也可以使用不发出网络请求的小练习。不需要真的切断公共服务器网络，也不需要修改其他同学的环境。

3. 自己设计一个上下文例子

选择一个简单、可以独立核验的场景：约定的符号含义、实验条件、单位或一组参数。设计后续问题时，让它需要引用先前信息。

思考：

- 如果只看到最后一句，回答者还缺少什么信息？
- 如果模型答对了，有没有可能是猜测或常识，而不是程序发送了前文？
- 如果新会话碰巧给出相同答案，怎样检查是否真的发生了历史混用？

这里不提供固定测试问题和标准回答。你应能解释自己的例子在检验什么，不能只复制一张看起来成功的聊天截图。

4. 把发现的问题写成可复现记录

可以用以下格式记一两次有代表性的调试过程：

说明 / 输出

问题现象：

触发操作：

我预期的行为：

我检查了哪些变量或请求信息：

发现的原因：

修改后怎样重新验证：

例如，“我增加了一句 `print`”不是结果；“发现实际拿到的是字符串而不是数字”才是能够支持后续修改的观察。记录应来自真实操作，不需要把每个小错误都写成报告。

5. README 可以回答哪些问题

简短说明应使别人知道如何运行并理解你的实现，例如：

- 使用什么 Python 环境和依赖，从哪个文件启动？
- API 地址、模型名称、Key 如何配置？不应提供真实 Key。
- 网页运行在哪里，自己电脑的浏览器如何访问？
- 怎样区分会话，历史保存到什么程度？
- 做了哪些验证，发现或保留了哪些限制？
- AI 帮助了哪些环节，哪些结果经过自己运行确认？

不要求写冗长文档。重点是说明与实际代码相符，避免声称未实现的功能。提交范围仍按正式作业描述执行。

6. 最后问自己

尝试在不查看 AI 回答的情况下解释：

1. Python 字典、JSON 文本和网页中的文字有什么区别？
2. 为什么浏览器能显示历史，不代表模型收到了历史？
3. 哪些变量属于这次请求，哪些数据希望保留更久？
4. 如果新建会话时只是换了标题，可能发生什么？
5. 为什么请求失败后需要检查历史有没有变化？
6. 如果换一台电脑运行，哪些是代码，哪些是环境配置？

能解释这些问题，是理解程序的重要证据；答不出来的部分，就是下一轮学习应回到的章节。

附录

官方资料索引

以下链接对应正文引用的官方文档。可点击访问，也可按标题查找；阅读范围以各章说明为准。

1. 数据结构

<https://docs.python.org/zh-cn/3/tutorial/datastructures.html>

2. 控制流与函数

<https://docs.python.org/zh-cn/3/tutorial/controlflow.html>

3. 错误和异常

<https://docs.python.org/zh-cn/3/tutorial/errors.html>

4. Requests 官方快速入门

<https://requests.readthedocs.io/en/latest/user/quickstart/#more-complicated-post-requests>

5. DeepSeek 单次调用说明

<https://api-docs.deepseek.com/>

6. Requests 响应状态与超时

<https://requests.readthedocs.io/en/latest/user/quickstart/#errors-and-exceptions>

7. MDN 表单数据说明

https://developer.mozilla.org/en-US/docs/Learn_web_development/Extensions/Forms/Sending_and_retrieving_form_data

8. Flask add_url_rule 文档

https://flask.palletsprojects.com/en/stable/api/#flask.Flask.add_url_rule

9. Flask Quickstart

<https://flask.palletsprojects.com/en/stable/quickstart/#accessing-request-data>

10. Python html.escape 文档

<https://docs.python.org/3/library/html.html#html.escape>

11. MDN 说明

<https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Status/303>

12. DeepSeek 多轮对话说明

https://api-docs.deepseek.com/guides/multi_round_chat/

13. Python os.environ 文档

<https://docs.python.org/zh-cn/3.13/library/os.html#os.environ>

14. Python 主程序入口说明

https://docs.python.org/zh-cn/3.13/library/__main__.html

15. OpenSSH 本地端口转发说明

<https://man.openbsd.org/ssh#L>