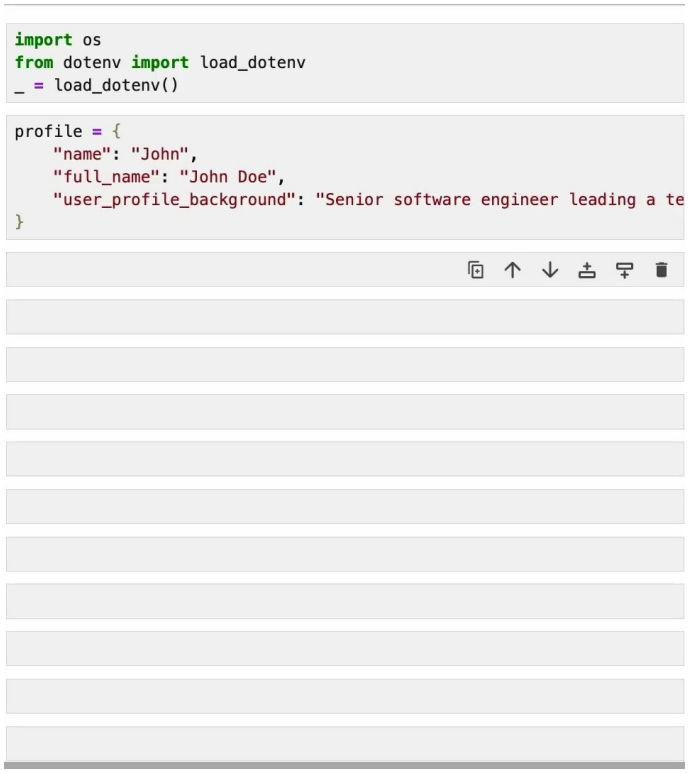
一、基础邮件助手 *00:01*

1. 创建基础邮件助手 00:09

1）设置邮件助手用户资料 00:16



```
import os
from dotenv import load_dotenv
_ = load_dotenv()
```

```
profile = {
    "name": "John",
    "full_name": "John Doe",
    "user_profile_background": "Senior software engineer leading a te
}
```

-
- **基础信息**: 包含用户姓名和背景信息，用于邮件助手的个性化设置
- **可定制性**: 可根据实际使用者的需求自由修改资料内容
- **用途**: 作为邮件响应时的身份背景参考

2）定义提示指令 00:42

```
import os
from dotenv import load_dotenv
_ = load_dotenv()
```

```
profile = {
    "name": "John",
    "full_name": "John Doe",
    "user_profile_background": "Senior software engineer leading a te
}
```

```
prompt_instructions = {
    "triage_rules": {
        "ignore": "Marketing newsletters, spam emails, mass company a
        "notify": "Team member out sick, build system notifications,
        "respond": "Direct questions from team members, meeting reque
    },
    "agent_instructions": "Use these tools when appropriate to help m
}
```

-
- **模块化设计**: 将提示指令分为两部分：分类规则和主代理指令
- **分类规则**: 定义三种邮件处理类型：忽略(ignore)、通知(notify)和响应(respond)
- **主代理指令**: 包含当需要响应邮件时的具体操作指南
- **设计优势**: 提高代码可读性和未来支持记忆功能的扩展性

3）定义示例邮件 02:04

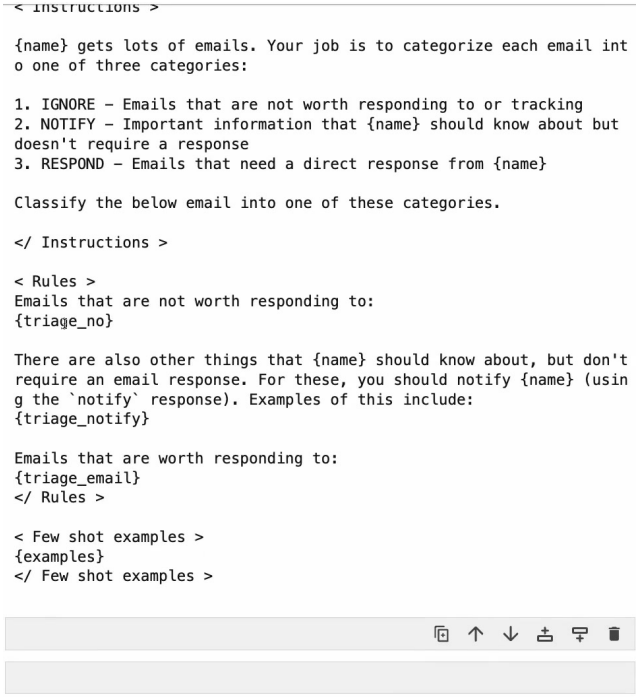
- **四要素**: 包含发件人(from)、收件人(to)、主题(subject)和正文(body)
- **测试用途**: 用于验证代理各环节功能是否正常
- **可替换性**: 可根据实际需求修改示例邮件内容

4）定义路由器 02:31

- 导入 02:37

- **核心组件**: 使用Pydantic的BaseModel定义路由器的输出模式
- **字段定义**:
  - reasoning: 包含分类决策的逐步推理过程
  - classification: 限定为ignore/respond/notify三种值

- 格式化系统提示 04:55



```
< Instructions >

{name} gets lots of emails. Your job is to categorize each email int
o one of three categories:

1. IGNORE - Emails that are not worth responding to or tracking
2. NOTIFY - Important information that {name} should know about but
doesn't require a response
3. RESPOND - Emails that need a direct response from {name}

Classify the below email into one of these categories.

</ Instructions >

< Rules >
Emails that are not worth responding to:
{triage_no}

There are also other things that {name} should know about, but don't
require an email response. For these, you should notify {name} (usin
g the `notify` response). Examples of this include:
{triage_notify}

Emails that are worth responding to:
{triage_email}
</ Rules >

< Few shot examples >
{examples}
</ Few shot examples >
```

- **结构组成**: 包含角色(Role)、背景(Background)、指令(Instructions)、规则(Rules)和示例(Few shot examples)五个部分
- **模板变量**: 使用${variable}形式预留填充位置
- **动态填充**: 运行时将用户资料和提示指令注入模板

- 格式化用户提示 05:14

- **简洁设计**: 仅包含邮件的基本信息格式化
- **四要素**: 格式化发件人、收件人、主题和邮件正文

- 测试路由器 05:24

- **调用方式**: 传递系统消息和用户消息给llm_router
- **输出解析**: 结果包含reasoning(推理过程)和classification(分类结果)
- **调试建议**: 可尝试修改邮件内容或指令观察分类变化

5）定义工具 06:07

- 邮件工具 06:24

```python
from langchain_core.tools import tool
```

```python
@tool
def write_email(to: str, subject: str, content: str) -> str:
    """Write and send an email."""
    # Placeholder response - in real app would send email
    return f"Email sent to {to} with subject '{subject}'"
```

- **参数**: 收件人(to)、主题(subject)和内容(content)
- **占位实现**: 当前仅返回确认信息，实际应用需集成邮件API

- 安排会议工具 06:40

```python
from langchain_core.tools import tool
```

```python
@tool
def write_email(to: str, subject: str, content: str) -> str:
    """Write and send an email."""
    # Placeholder response - in real app would send email
    return f"Email sent to {to} with subject '{subject}'"
```

```python
@tool
def schedule_meeting(
    attendees: list[str],
    subject: str,
    duration_minutes: int,
    preferred_day: str
) -> str:
    """Schedule a calendar meeting."""
    # Placeholder response - in real app would check calendar and sch
    return f"Meeting '{subject}' scheduled for {preferred_day} with {
```

- **参数**: 参会者列表(attendees)、主题(subject)、时长(duration_minutes)和偏好日期(preferred_day)
- **模拟实现**: 返回预定确认信息，实际需连接日历API

- 检查日历工具 07:06

```python
from prompts import agent_system_prompt
def create_prompt(state):
    return [
        {
            "role": "system",
            "content": agent_system_prompt.format(
                instructions=prompt_instructions["agent_instructions"
                **profile
                )
        }
    ] + state['messages']
```

- **固定返回**: 始终返回9:00 AM、2:00 PM和4:00 PM三个时间段
- **实际应用**: 应替换为真实日历系统查询逻辑

6）创建代理提示 07:19

```python
from prompts import agent_system_prompt
def create_prompt(state):
    return [
        {
            "role": "system",
            "content": agent_system_prompt.format(
                instructions=prompt_instructions["agent_instructions"
                **profile
                )
        }
    ] + state['messages']
```

```python
print(agent_system_prompt)
```

```
< Role >
You are {full_name}'s executive assistant. You are a top-notch execu
tive assistant who cares about {name} performing as well as possibl
e.
</ Role >

< Tools >
You have access to the following tools to help manage {name}'s commu
nications and schedule:

1. write_email(to, subject, content) - Send emails to specified reci
pients
2. schedule_meeting(attendees, subject, duration_minutes, preferred_
day) - Schedule calendar meetings
3. check_calendar_availability(day) - Check available time slots for
a given day
</ Tools >

< Instructions >
{instructions}
```

- **动态生成**: 函数接收状态并返回消息列表
- **组成**: 系统消息(包含角色、工具列表和自定义指令) + 状态中的历史消息
- **指令来源**: 从预先定义的prompt_instructions中获取

7）构建代理人 08:07

```python
from langgraph.prebuilt import create_react_agent
```

```python
tools=[write_email, schedule_meeting, check_calendar_availability]
agent = create_react_agent(
    "openai:gpt-4o",
    tools=tools,
    prompt=create_prompt,
)
```

-
- **三要素配置**:
  - 模型选择(如openai:gpt-4o)
  - 工具列表(邮件、会议、日历工具)
  - 提示生成函数
- **响应测试**: 可通过询问"周二可用时间"等简单请求验证功能

8）创建整体代理人 09:27

- 定义代理人的状态 09:32

```python
from langgraph.graph import add_messages

class State(TypedDict):
    email_input: dict
    messages: Annotated[list, add_messages]
```

  -
  - **双组件结构**:
    - email_input: 包含待处理邮件的完整信息

- messages: 记录代理处理过程中的消息流

- 定义节点进行分类 10:00
  - 实现逻辑 10:33

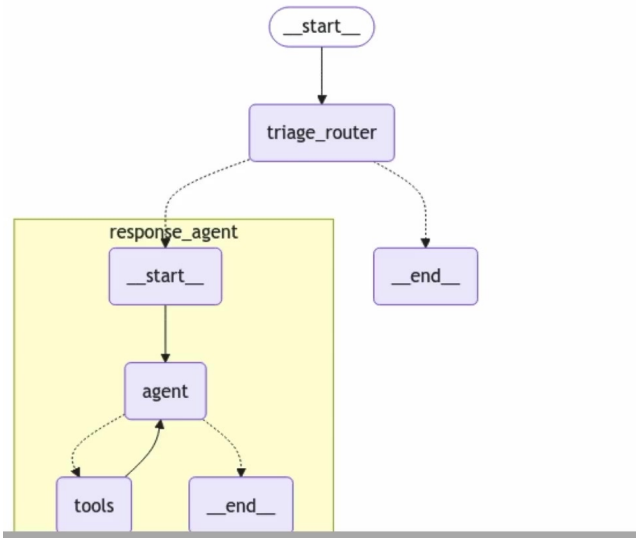
```python
def triage_router(state: State) -> Command[
    Literal["response_agent", "__end__"]
]:
    author = state['email_input']['author']
    to = state['email_input']['to']
    subject = state['email_input']['subject']
    email_thread = state['email_input']['email_thread']

    system_prompt = triage_system_prompt.format(
        full_name=profile["full_name"],
        name=profile["name"],
        user_profile_background=profile["user_profile_background"],
        triage_no=prompt_instructions["triage_rules"]["ignore"],
        triage_notify=prompt_instructions["triage_rules"]["notify"],
        triage_email=prompt_instructions["triage_rules"]["respond"],
        examples=None
    )
    user_prompt = triage_user_prompt.format(
        author=author,
        to=to,
        subject=subject,
        email_thread=email_thread
    )
```

  - **流程步骤**:
    - 提取邮件要素(发件人、收件人等)
    - 格式化系统和用户提示
    - 调用llm_router获取分类
    - 根据分类结果返回不同命令
  - **命令结构**: 包含goto(下一节点)和update(状态更新)两个字段
- 定义电子邮件代理人 13:16

```python
email_agent = StateGraph(State)
email_agent = email_agent.add_node(triage_router)
email_agent = email_agent.add_node("response_agent", agent)
email_agent = email_agent.add_edge(START, "triage_router")
email_agent = email_agent.compile()
```

```python
display(Image(email_agent.get_graph(xray=True).draw_mermaid_png()))
```



  - **双节点设计**:
    - triage_router: 初始分类节点

        - response_agent: 响应处理子代理
    - **边定义**: 从START到triage_router的初始边
- 绘制代理人图表 14:17

    - **图形展示**: 使用mermaid图展示代理流程
    - **子代理细节**: 通过xray=true参数可查看response_agent内部的REACT循环
- 应用案例 14:38
    - 例题:邮件分类处理

```
        subject=subject,
        email_thread=email_thread
    )
    result = llm_router.invoke(
        [
            {"role": "system", "content": system_prompt},
            {"role": "user", "content": user_prompt},
        ]
    )
    if result.classification == "respond":
        print("📧 Classification: RESPOND - This email requires a res
        goto = "response_agent"
        update = {
            "messages": [
                {
                    "role": "user",
                    "content": f"Respond to the email {state['email_i
                }
            ]
        }
    elif result.classification == "ignore":

    elif result.classification == "notify":

    else:
        raise ValueError(f"Invalid classification: {result.classifica
    return Command(goto=goto, update=update)
```

        - **处理结果**: 被正确分类为ignore(忽略)
        - **验证方式**: 观察打印的分类日志信息
    - 例题:邮件响应处理 15:04

```
    email_thread=email_thread
)
result = llm_router.invoke(
    [
        {"role": "system", "content": system_prompt},
        {"role": "user", "content": user_prompt},
    ]
)
if result.classification == "respond":
    print("📧 Classification: RESPOND - This email requires a respo
    goto = "response_agent"
    update = {
        "messages": [
            {
                "role": "user",
                "content": f"Respond to the email {state['email_inp
            }
        ]
    }
elif result.classification == "ignore":
    print("🚫 Classification: IGNORE - This email can be safely ign
    update = None
    goto = END
elif result.classification == "notify":
    # If real life, this would do something else
    print("🔔 Classification: NOTIFY - This email contains importan
    update = None
    goto = END
else:
    raise ValueError(f"Invalid classification: {result.classificati
return Command(goto=goto, update=update)
```

-
- **处理流程**:
  - 分类为respond(需要响应)
  - 生成包含详细解答的回复邮件
  - 调用write_email工具发送回复
- **消息跟踪**: 可通过检查messages列表观察完整处理过程

二、知识小结

知识点	核心内容	易混淆点/关键细节	难度系数
基础邮件助手搭建	创建无记忆的初始邮件助手框架，包含环境变量配置、用户资料设置（姓名/背景）	用户资料需匹配实际使用者的身份	★★
提示指令设计	分两部分：- **分类规则**（忽略/通知/回复三类邮件的判定标准）；- **主代理指令**（回复邮件时的操作规范）	分类规则与主代理指令的边界划分	★★★
邮件数据结构	包含四个字段：- 发件人地址; - 收件人地址; - 邮件主题; - 邮件正文	字段完整性对代理处理的影响	★★
分类代理实现	使用OpenAI GPT模型进行邮件分类：- 输出含推理过程及分类结果（ignore/notify/respond）；- 结构化输出绑定	**分类逻辑**与**实际业务需求**的匹配度	★★★★
工具链集成	模拟三类工具：- 邮件发送工具（收件人/主题/内容）；- 会议安排工具（参会人/主题/时长/日期）；- 日历查询工具（返回固定时间段）	工具接口设计与真实API的差异	★★★
主代理构建	基于ReAct架构的代理：- 动态生成系统提示（角色定义/工	提示模板的变量填充逻辑	★★★★

	具列表/自定义指令）；- 处理消息状态（含历史消息）		
工作流整合	状态图控制流程： - 分类节点→响应代理或终止;- 响应代理内嵌ReAct循环	子代理（响应代理）与主流程的协同	★★★★
测试验证	通过示例邮件验证： - 垃圾邮件→ignore;- 需回复邮件→调用工具生成响应	**分类准确性**与**工具调用合理性**的验证	★★★