项目名称：Python 与股票技术分析入门：知其然，知其所以然

1. 你的任务 (Your Mission)

欢迎来到你的第一个 Python 金融项目！你的任务是构建一个小型的股票分析工具。本项目与其他教程最大的不同是：我们不仅告诉你**“做什么”**，更要让你明白**“为什么这么做”**。

请把这个过程想象成一次学徒之旅。每完成一个任务，你不仅获得一项技能，更收获其背后的工程智慧。

核心目标:

1. **知其然 (The What)**: 掌握 Python 核心语法，完成一个能工作的项目。
2. **知其所以然 (The Why)**: 理解现代软件开发的最佳实践和设计思想。
3. **建立工程思维**: 学习如何像专业人士一样思考和组织代码。

---

2. 我们的思考过程：如何从零到一设计项目 (Our Thinking Process)

在开始写任何代码之前，让我们先学习最重要的技能：如何思考。我们采用两种强大的思维模型来设计这个项目。

第一步：以终为始 (Start with the End in Mind)

我们首先定义最终要达成的目标 (The End)：

> “我想要一个能自动获取某只股票的数据，计算它的20日移动平均线，然后把价格和均线画在一张图上，并把结果保存下来的工具。”

这个清晰的目标就是我们的终点。它告诉我们项目“需要做什么 (What)”。

第二步：运用第一性原理进行分解 (Breakdown with First Principles)

接下来，我们将这个最终目标，分解成一系列最基础、最核心、不可再分的功能“积木”(First Principles)。一个软件工程师会问：“要实现那个最终目标，我最少需要哪些基本功能？”

1. **需要一个“东西”来获取数据** -> 我们称之为 **数据获取器 (Data Fetcher)**。
2. **需要一个“东西”来做数学计算** -> 我们称之为 **指标计算器 (Indicator Calculator)**。
3. **需要一个“东西”来画图** -> 我们称之为 **可视化工具 (Visualizer)**。
4. **需要一个“东西”来保存数据** -> 我们称之为 **文件处理器 (File Handler)**。
5. **需要一个地方存放股票代码、日期等易变信息** -> 我们称之为 **配置 (Config)**。
6. **最后，需要一个“总指挥”来按顺序调用这些“东西”** -> 我们称之为 **主入口 (Main Entrypoint)**。
7. **（最重要的一步）需要一个“东西”来验证以上所有“东西”都工作正常** -> 我们称之为 **测试 (Tests)**。

第三步：将功能映射到代码结构 (Map Functions to Code Structure)

看！上面分解出的7个核心功能，完美地、一一对应地映射到了我们项目的模块化文件结构：

- 数据获取器 -> `data_fetcher.py`
- 指标计算器 -> `indicator_calculator.py`
- 可视化工具 -> `visualizer.py`
- 文件处理器 -> `file_handler.py`
- 配置 -> `config.py`
- 主入口 -> `main.py`
- 测试 -> `tests/`

这就是我们设计这个项目文件结构的**根本原因**。它不是随意的，而是我们"以终为始，第一性原理分解"思考过程的直接产物。通过这种方式，我们构建的系统逻辑清晰、高度解耦、易于维护。

现在，你将要亲手搭建的，正是这个专业思考过程的结晶。

---

3. 项目任务清单 (Project Checklist)

阶段一：项目初始化与环境搭建 (Phase 1: Initialization & Setup)

- [ ] **1. 初始化项目**:
  - **操作**: 创建 `openbb_project` 目录, `cd` 进入, 运行 `git init`。
  - **> 为什么?** 从第一天起就用 `git` 进行版本控制，是专业开发者的基本功。
- [ ] **2. 创建 `.gitignore` 文件**:
  - **操作**: 创建 `.gitignore` 文件，并添加 `.venv/`, `__pycache__/`, `*.pyc`, `data/` 和 `.pytest_cache/` 目录。
  - **> 为什么?** 保持代码仓库干净，只追踪我们的**源代码**，忽略自动生成的、本地的数据文件和测试缓存。
- [ ] **3. 创建并激活 Python 虚拟环境**:
  - **操作**: 运行 `python3 -m venv .venv` 并激活它。
  - **> 为什么?** 为项目创建独立的"房间"，保证依赖隔离和环境的可复现性。
- [ ] **4. 安装依赖并生成 `requirements.txt`**:
  - **操作**: `pip install openbb pandas matplotlib`, 然后 `pip freeze > requirements.txt`。
  - **> 为什么?** `requirements.txt` 像一张"购物清单"，让任何人都可用一条命令完美复现你的开发环境。
- [ ] **5. 创建项目文件结构**:
  - **操作**:
    1. 创建 `data/` 文件夹。
    2. 创建 `tests/` 文件夹。
    3. 创建空的 Python 文件: `main.py`, `config.py`, `data_fetcher.py`, `indicator_calculator.py`, `visualizer.py`, `file_handler.py`。
    4. 在 `tests/` 文件夹下创建空的 `test_indicator_calculator.py`。

- **> 为什么？ 模块化**思想。每个文件/目录职责单一：`.py` 文件放代码逻辑，`data/` 目录放数据，`tests/` 目录放测试。这让项目结构清晰，易于维护。

完成此阶段后，你的项目目录看起来应该像这样：

```
openbb_project/
├── data/                       # 存放我们生成的数据文件
├── tests/                      # 存放我们的测试代码
│   └── test_indicator_calculator.py
├── .venv/
├── .gitignore
├── config.py
├── data_fetcher.py
├── file_handler.py
├── indicator_calculator.py
├── main.py
├── requirements.txt
└── visualizer.py
```

阶段二：编写代码模块 (Phase 2: Coding the Modules)

- [ ] **6. 配置你的分析 (`config.py`)**:
  - **操作**: 在 `config.py` 中定义 `TICKER`, `START_DATE`, `SMA_WINDOW` 等变量。
  - **> 为什么？ 配置与逻辑分离**，让程序更灵活。
- [ ] **7. 编写数据获取器 (`data_fetcher.py`)**:
  - **操作**: 定义 `fetch_stock_data(ticker, start_date)` 函数。根据 OpenBB 官方文档，推荐使用其 Python SDK 的 `obb.equity.price.historical()` 方法来获取历史数据。你需要先 `import openbb as obb`，然后在函数内部调用 `obb.equity.price.historical(symbol=ticker, start_date=start_date, provider="yfinance").to_df()` 来获取数据并返回 `DataFrame`。
  - **> 为什么？** 将数据获取逻辑**封装**成一个独立的、可复用的功能模块。我们这里使用了 `obb.equity.price.historical()`，这是 OpenBB 官方文档中推荐的、用于获取历史股票数据的标准方法。它明确指定了数据提供者（如 `yfinance`），并直接返回一个易于使用的 `pandas DataFrame`，这比旧版或通用的 `openbb.stocks.load()` 更精确、更可靠。
- [ ] **8. 编写指标计算器 (`indicator_calculator.py`)**:
  - **操作**: 定义 `add_sma(df, window_size)` 函数，计算 SMA 并返回更新后的 `DataFrame`。
  - **> 为什么？** 同样是**封装**。计算指标的逻辑是独立的，应该放在它自己的模块里。
- [ ] **9. 编写文件处理器 (`file_handler.py`)**
  - **操作**:
    1. 打开 `file_handler.py`。
    2. 导入 `pandas` 和 `pathlib` (`from pathlib import Path`)。
    3. 定义一个 `save_to_csv(df, ticker)` 函数。

4. 在函数内部，使用 `df.to_csv(f"data/{ticker}_data.csv")` 将 `DataFrame` 保存到 `data` 目录下。

- **> 为什么这么做?**

> 这就是**数据持久化 (Data Persistence)**。从网络获取数据可能很慢，而且有次数限制。将处理好的数据保存到本地文件（如 CSV），我们就不需要每次运行都重新获取了。这相当于为数据创建了一个**本地缓存**，极大地提高了后续分析的效率。CSV 是一种通用的、人类可读的表格文件格式，非常适合初学者。

- [ ] **10. 编写可视化工具 (`visualizer.py`):**
  - **操作**: 定义 `plot_data(df, ticker)` 函数，使用 `matplotlib` 绘制图表。
  - **> 为什么?** 将所有与"画图"相关的代码集中在一起，符合**关注点分离**原则。

阶段三：组装并运行你的应用 (Phase 3: Assembling and Running)

- [ ] **11. 编写主入口 (`main.py`):**
  - **操作**:
    1. 导入所有需要的模块和配置。
    2. 定义 `main()` 函数。
    3. 在 `main()` 函数中，按以下顺序编排逻辑：
       a. 调用 `data_fetcher.fetch_stock_data()` 获取数据。
       b. 调用 `indicator_calculator.add_sma()` 计算指标。
       c. 调用 `file_handler.save_to_csv()` 将处理后的数据保存到文件。
       d. 调用 `visualizer.plot_data()` 显示图表。
    4. 在文件末尾使用 `if __name__ == "__main__":` 调用 `main()`。
  - **> 为什么这么做?**

> `main.py` 是项目的**总指挥**。它清晰地定义了我们应用的**工作流 (Workflow)**：`获取数据 -> 处理数据 -> 保存结果 -> 展示结果`。
> `if __name__ == "__main__":` 结构则保证了我们的脚本既可以被直接执行，也可以作为模块被其他程序安全地导入，是 Python **代码复用性**和**模块化设计**的关键。

- [ ] **12. 运行你的项目!**
  - **操作**: 在终端运行 `python main.py`。
  - **> 检查**:
    1. 程序运行后，是否弹出了图表?
    2. 同时，查看你的 `data` 文件夹，里面是否出现了一个新的 `.csv` 文件（如 `AAPL_data.csv`）？你可以用文本编辑器或 Excel 打开它，看看里面的内容。

---

4. 恭喜与进阶：成为一名真正的工程师 (Congratulations & Next Steps)

恭喜！你已经完成了一个包含数据获取、处理、持久化和可视化的完整流程。但要成为一名真正的软件工程师，我们还差最后，也是最关键的一步：**为你的代码编写测试**。

终极挑战：为你的计算器编写单元测试 (Final Mission: Unit Test Your Calculator)

> 为什么需要测试?

> 如果没有测试，你怎么能100%确定你的 `add_sma` 函数在所有情况下计算都正确？当你未来修改了它，你怎么保证没有意外破坏它原有的功能？测试就是你代码的“质量保证书”和“安全网”。它能给你自信，让你的软件产品坚如磐石。

我们将为最核心的逻辑——`indicator_calculator.py`——编写一个**单元测试**。

- [ ] **1. 安装测试框架**:
  - **操作**: 激活你的虚拟环境，运行 `pip install pytest`。然后，不要忘记更新你的依赖清单：`pip freeze > requirements.txt`。
  - **> 为什么?** `pytest` 是 Python 社区最流行、最强大的测试框架。它能让编写测试变得简单而直观。
- [ ] **2. 编写第一个测试用例**:
  - **操作**: 在 `tests/test_indicator_calculator.py` 中，写入以下代码：

```
import pandas as pd
import numpy as np
import sys
import os

# 为了让测试能找到我们的模块，需要将项目根目录添加到 Python 路径
sys.path.insert(0, os.path.abspath(os.path.join(os.path.dirname(__file__),
'..')))

from indicator_calculator import add_sma

def test_add_sma_basic():
    # 1. 准备 (Arrange): 创建一个简单、可预测的测试数据
    data = {'close': [10, 11, 12, 13, 14]}
    df = pd.DataFrame(data)
    window_size = 3

    # 2. 执行 (Act): 调用我们要测试的函数
    result_df = add_sma(df, window_size)

    # 3. 断言 (Assert): 验证结果是否符合预期
    expected_sma = [np.nan, np.nan, 11.0, 12.0, 13.0] # 手动计算出的正确结果

    assert 'SMA_3' in result_df.columns # 检查新列是否存在
    pd.testing.assert_series_equal(result_df['SMA_3'], pd.Series(expected_sma,
name='SMA_3'))
    print("\n测试成功：SMA 计算结果符合预期！")
```

- **> 为什么这么做?**

> 这就是经典的 **Arrange-Act-Assert (3A)** 测试模式。我们首先准备好输入和预期的输出，然后执行函数，最后用 `assert`（断言）来检查实际输出是否和预期完全一致。如果一致，测试通过；否则，`pytest` 会告诉你哪里出了问题。

- [ ] **3. 运行测试**:
  - **操作**: 在你的项目根目录（`openbb_project/`）打开终端，直接运行 `pytest` 命令。
  - **> 检查**：`pytest` 是否找到了你的测试并显示 `1 passed`?

完成了这一步，你才真正完成了这个项目。你现在拥有了一个**经过验证、质量可靠**的软件模块。

进阶挑战 (Bonus Mission):

- **读取本地数据**: 修改 `main.py` 的逻辑。在获取数据前，先检查 `data` 文件夹中是否已存在对应的 CSV 文件。如果存在，就直接从本地文件读取数据 (`pd.read_csv()`)，跳过网络请求。这会真正实现“缓存”功能，让你的应用快如闪电!