

LMCC例卷机试-解析

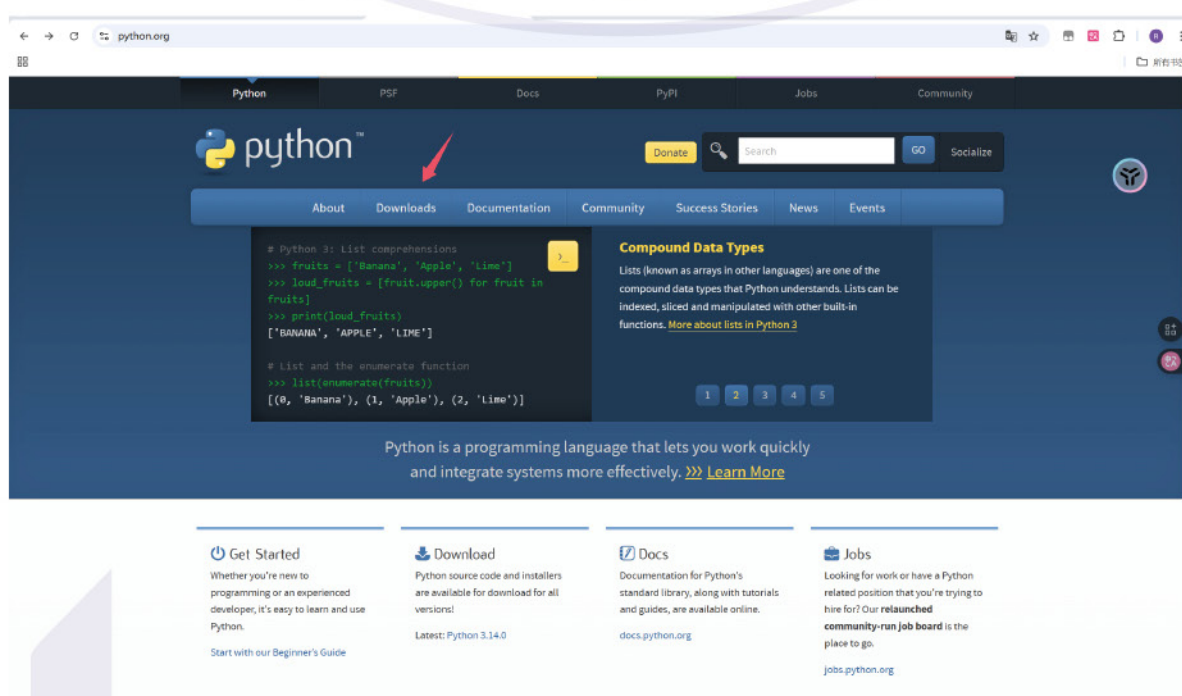
1. 环境配置

1. Windows系统的环境配置

Step 1 Python环境准备:

注: 电脑上有python环境直接跳过这一步

访问 <https://www.python.org/> 网页, 在 **Download** 页面找到 **Windows** 进入, 选择 **Windows installer(64-bit)** 下载 (Python版本建议在3.10 - 3.13, 下图示以3.11.9为例)



python.org/downloads/

Release schedules

- Python 3.15 Release Schedule
- Python 3.14 Release Schedule
- Python 3.13 Release Schedule
- Python 3.12 Release Schedule
- Python 3.11 Release Schedule
- Python 3.10 Release Schedule
- Python 3.9 Release Schedule
- Python 3.8 Release Schedule

See Status of Python versions for an overview of all versions, including unsupported.

Information about specific ports, and developer info

- Windows
- macOS
- Android
- Other platforms
- Source
- Python Developer's Guide
- Python Issue Tracker

How to verify your downloaded files are genuine

Sigstore verification

Starting with the Python 3.11.0, Python 3.10.7, and Python 3.9.14 releases, CPython release artifacts are signed with Sigstore. See our dedicated [Sigstore Information page](#) for how it works.

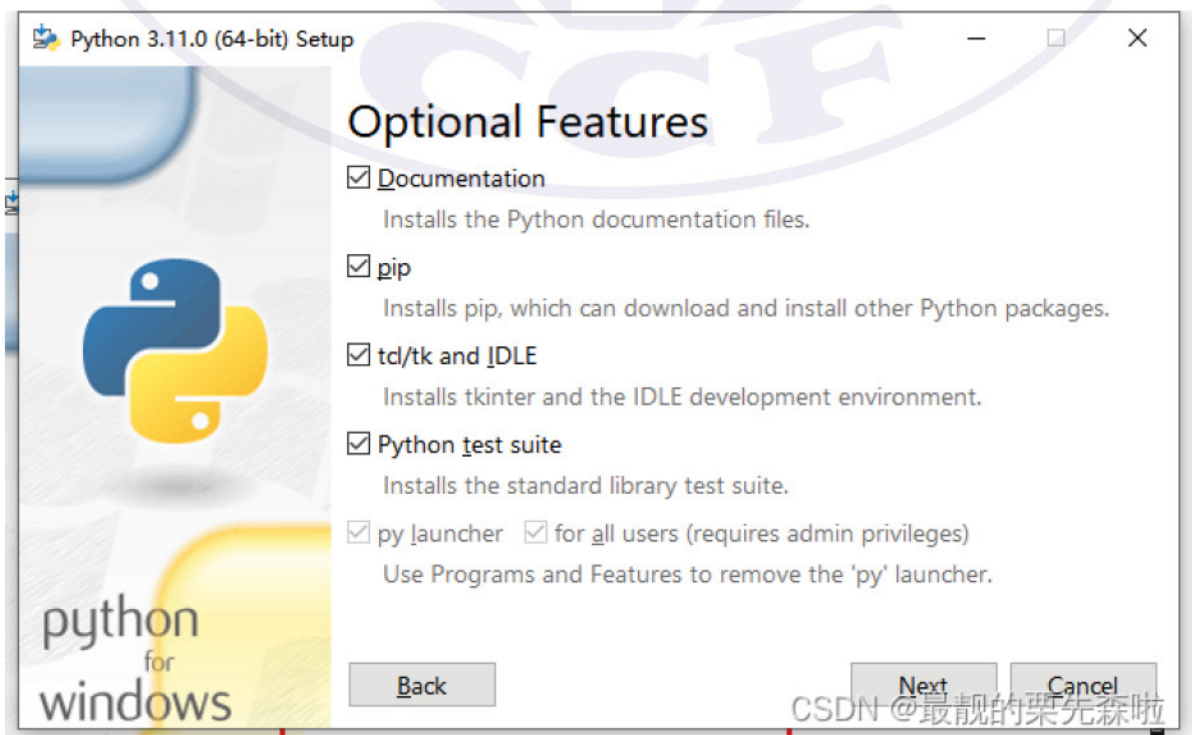
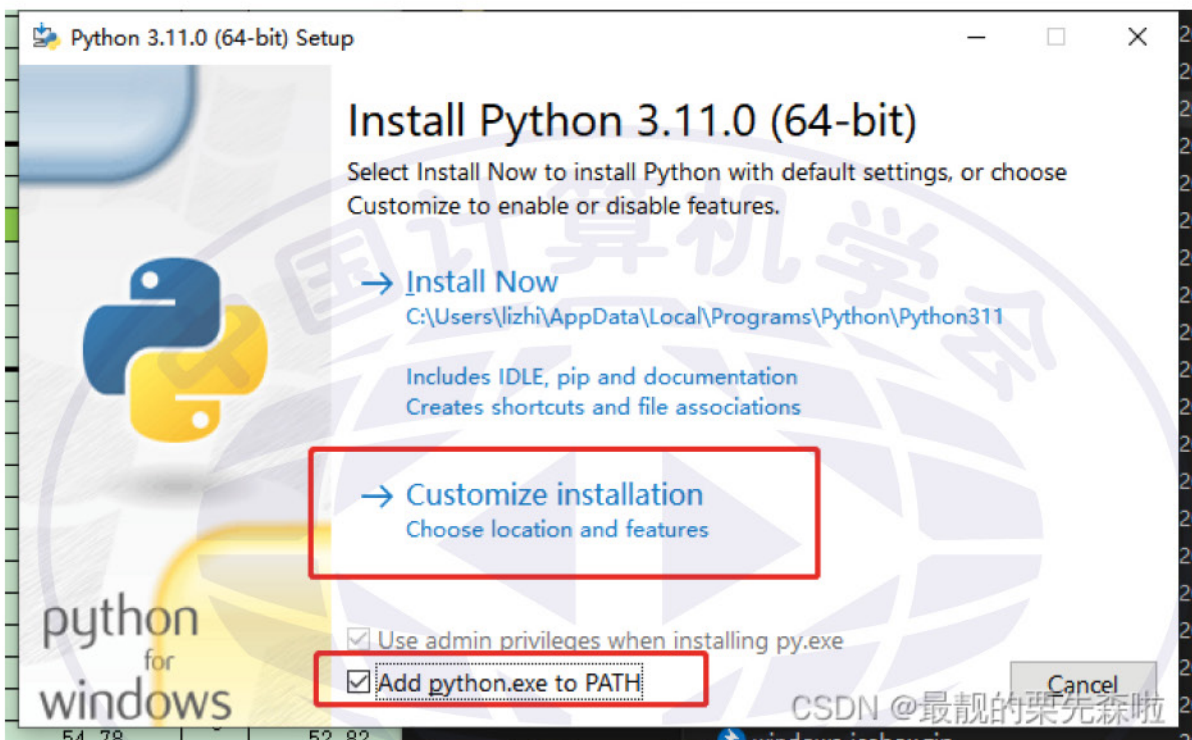
OpenPGP verification

Python versions before 3.14 are also signed using OpenPGP private keys of the respective release manager. In this case, verification through the release manager's public key

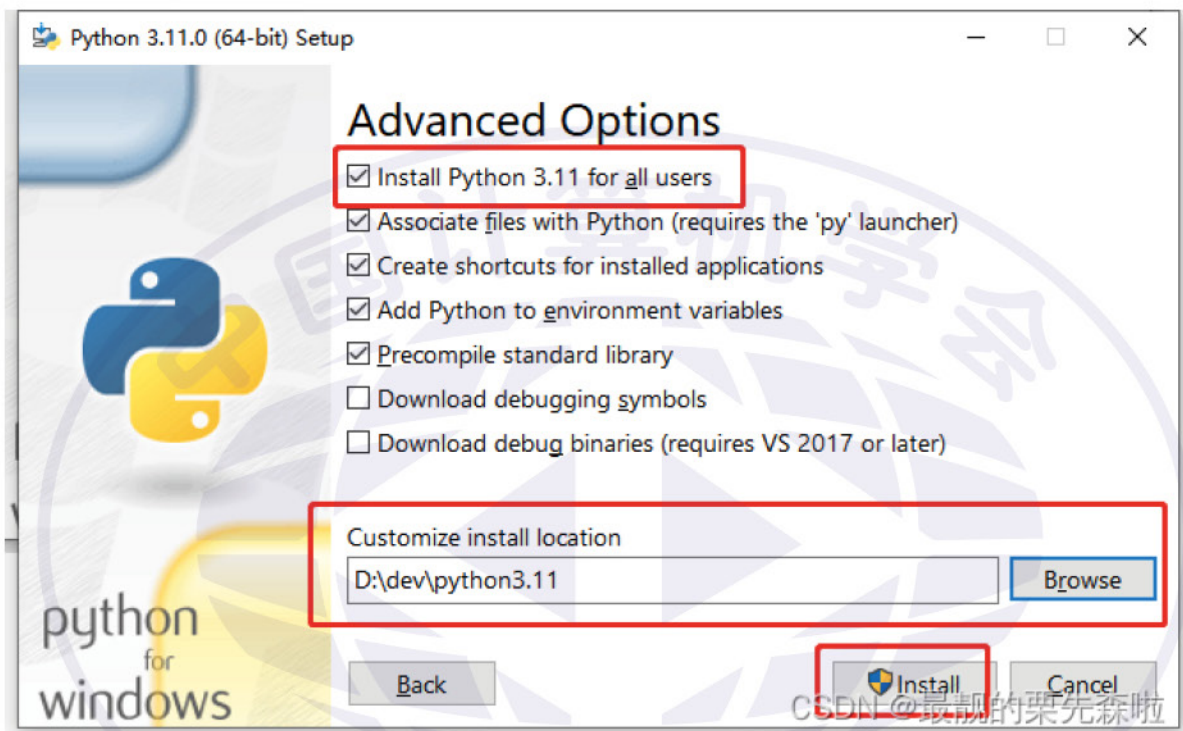
python.org/downloads/windows/

- Download Windows embeddable package (32-bit)
- Download Windows embeddable package (ARM64)
- Python 3.12.3 - April 9, 2024
 - Note that Python 3.12.3 cannot be used on Windows 7 or earlier.
 - Download Windows installer (64-bit)
 - Download Windows installer (32-bit)
 - Download Windows installer (ARM64)
 - Download Windows embeddable package (64-bit)
 - Download Windows embeddable package (32-bit)
 - Download Windows embeddable package (ARM64)
- Python 3.11.9 - April 2, 2024
 - Note that Python 3.11.9 cannot be used on Windows 7 or earlier.
 - Download Windows installer (64-bit)
 - Download Windows installer (32-bit)
 - Download Windows installer (ARM64)
 - Download Windows embeddable package (64-bit)
 - Download Windows embeddable package (32-bit)
 - Download Windows embeddable package (ARM64)
- Python 3.11.8 - Feb. 6, 2024
 - Note that Python 3.11.8 cannot be used on Windows 7 or earlier.
 - Download Windows installer (64-bit)
 - Download Windows installer (32-bit)
 - Download Windows installer (ARM64)
 - Download Windows embeddable package (64-bit)
 - Download Windows embeddable package (32-bit)
 - Download Windows embeddable package (ARM64)
- Python 3.12.2 - Feb. 6, 2024
 - Note that Python 3.12.2 cannot be used on Windows 7 or earlier.
 - Download Windows installer (64-bit)
- Python 3.13.0b3 - June 27, 2024
 - Download Windows installer (64-bit)
 - Download Windows installer (32-bit)
 - Download Windows installer (ARM64)
 - Download Windows embeddable package (64-bit)
 - Download Windows embeddable package (32-bit)
 - Download Windows embeddable package (ARM64)
- Python 3.13.0b2 - June 5, 2024
 - Download Windows installer (64-bit)
 - Download Windows installer (32-bit)
 - Download Windows installer (ARM64)
 - Download Windows embeddable package (64-bit)
 - Download Windows embeddable package (32-bit)
 - Download Windows embeddable package (ARM64)
- Python 3.13.0b1 - May 8, 2024
 - Download Windows installer (64-bit)
 - Download Windows installer (32-bit)
 - Download Windows installer (ARM64)
 - Download Windows embeddable package (64-bit)
 - Download Windows embeddable package (32-bit)
 - Download Windows embeddable package (ARM64)
- Python 3.13.0a6 - April 9, 2024
 - Download Windows installer (64-bit)
 - Download Windows installer (32-bit)
 - Download Windows installer (ARM64)
 - Download Windows embeddable package (64-bit)
 - Download Windows embeddable package (32-bit)
 - Download Windows embeddable package (ARM64)
- Python 3.13.0a5 - March 12, 2024
 - Download Windows installer (64-bit)
 - Download Windows installer (32-bit)

打开下载好的.exe文件进行安装，勾选 **Add python.exe to PATH** 和 **Use admin privileges when installing py.exe**，选择 **Customize installation** 进行下一步。



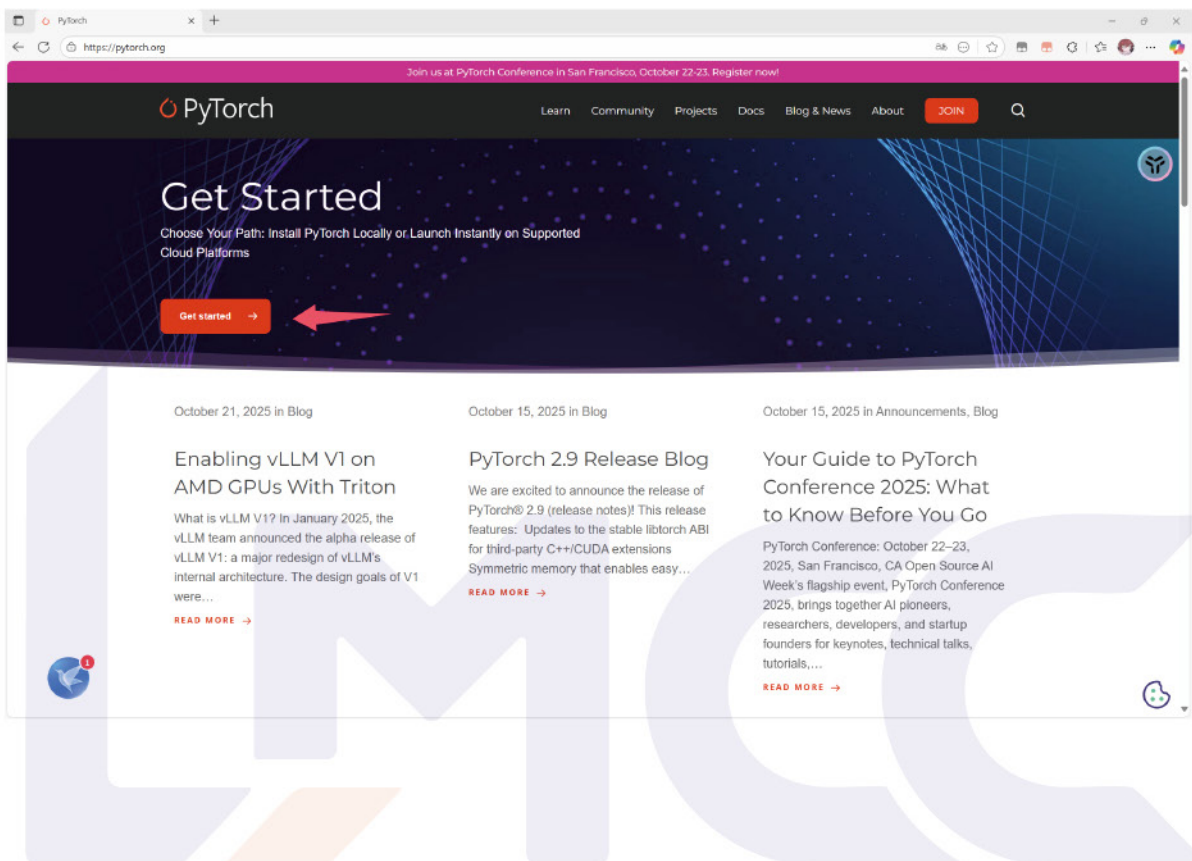
勾选如下图所示，并选择想要安装python的位置进行安装。



验证安装是否成功：win+R 输入 cmd 后回车，输入python，回车，如果出现安装的python版本，即为安装成功。

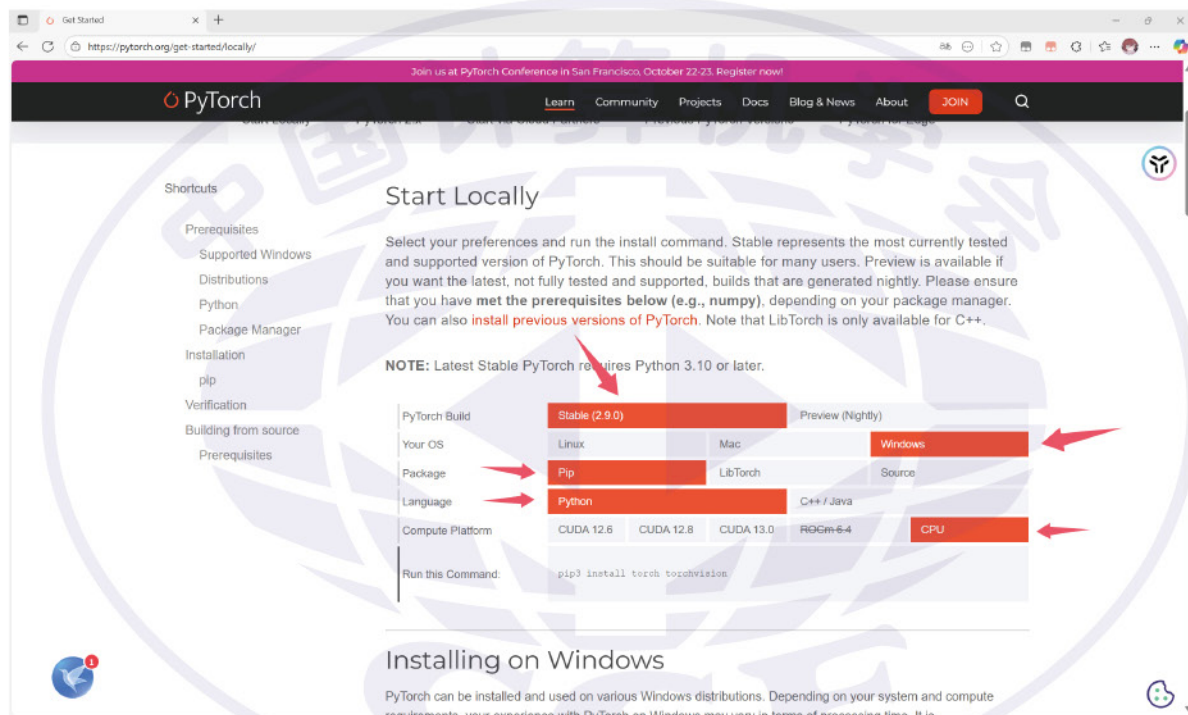
Step 2: pytorch安装

访问 <https://pytorch.org/> 网页，点击 **Get started**。



2. 无显卡的Windows系统:

勾选下图红色箭头指示的部分后复制 **Run this Command** 后面的代码。



win+R 输入powershell, 回车, 将刚才复制的代码通过右键黏贴后, 回车。

```
(LMCC) PS C:\Users\27682> pip3 install torch torchvision
Collecting torch
  Downloading torch-2.9.0-cp311-cp311-win_and64.whl.metadata (30 kB)
Collecting torchvision
  Downloading torchvision-0.24.0-cp311-cp311-win_and64.whl.metadata (5.9 kB)
Requirement already satisfied: filelock in d:\anaconda\envs\lmcc\lib\site-packages (from torch) (3.19.1)
Requirement already satisfied: typing-extensions>=4.10.0 in d:\anaconda\envs\lmcc\lib\site-packages (from torch) (4.15.0)
Requirement already satisfied: sympy<=1.13.3 in d:\anaconda\envs\lmcc\lib\site-packages (from torch) (1.14.0)
Requirement already satisfied: networkx>=2.5.1 in d:\anaconda\envs\lmcc\lib\site-packages (from torch) (3.5)
Requirement already satisfied: jinja2 in d:\anaconda\envs\lmcc\lib\site-packages (from torch) (3.1.6)
Requirement already satisfied: fsspec>=0.8.5 in d:\anaconda\envs\lmcc\lib\site-packages (from torch) (2025.9.0)
Requirement already satisfied: numpy in d:\anaconda\envs\lmcc\lib\site-packages (from torchvision) (2.3.3)
Requirement already satisfied: pillow<9.3.0, >=6.3.0 in d:\anaconda\envs\lmcc\lib\site-packages (from torchvision) (11.3.0)
Requirement already satisfied: mpmath<1.4, >=1.2.0 in d:\anaconda\envs\lmcc\lib\site-packages (from sympy<=1.13.3->torch) (1.3.0)
Requirement already satisfied: MarkupSafe>=2.0 in d:\anaconda\envs\lmcc\lib\site-packages (from jinja2->torch) (2.1.5)
Downloading torch-2.9.0-cp311-cp311-win_and64.whl (109.3 MB)
109.3/109.3 MB 12.0 MB/s 0:00:09
Downloading torchvision-0.24.0-cp311-cp311-win_and64.whl (4.0 MB)
4.0/4.0 MB 80.1 MB/s 0:00:00
Installing collected packages: torch, torchvision
Successfully installed torch-2.9.0 torchvision-0.24.0
(LMCC) PS C:\Users\27682>
```

Step3: transformers下载

win+R 输入powershell, 回车, 复制下面的代码, 右键黏贴后, 回车:

```
python -m pip install -U transformers
```

```
(LMCC) PS D:\LMCC-例卷-机试\T2> python -m pip install -U transformers
Collecting transformers
  Using cached transformers-4.57.1-py3-none-any.whl.metadata (43 kB)
Requirement already satisfied: filelock in d:\anaconda\envs\lmcc\lib\site-packages (from transformers) (3.19.1)
Collecting huggingface-hub<1.0,>=0.34.0 (from transformers)
  Using cached huggingface_hub-0.35.3-py3-none-any.whl.metadata (14 kB)
Requirement already satisfied: numpy>=1.17 in d:\anaconda\envs\lmcc\lib\site-packages (from transformers) (2.3.3)
Collecting packaging>=20.0 (from transformers)
  Using cached packaging-25.0-py3-none-any.whl.metadata (3.3 kB)
Collecting pyyaml>=5.1 (from transformers)
  Using cached pyyaml-6.0.3-cp311-cp311-win_and64.whl.metadata (2.4 kB)
Collecting regex<=2019.12.17 (from transformers)
  Downloading regex-2025.10.23-cp311-cp311-win_and64.whl.metadata (41 kB)
Collecting requests (from transformers)
  Using cached requests-2.32.5-py3-none-any.whl.metadata (4.9 kB)
Collecting tokenizers<=0.23.0,>=0.22.0 (from transformers)
  Using cached tokenizers-0.22.1-cp39-abi3-win_and64.whl.metadata (6.9 kB)
Collecting safetensors>=0.4.3 (from transformers)
  Using cached safetensors-0.6.2-cp38-abi3-win_and64.whl.metadata (4.1 kB)
Collecting tqdm>=4.27 (from transformers)
  Using cached tqdm-4.67.1-py3-none-any.whl.metadata (57 kB)
Requirement already satisfied: fsspec>=2023.5.0 in d:\anaconda\envs\lmcc\lib\site-packages (from huggingface-hub<1.0,>=0.34.0->transformers) (2025.9.0)
Requirement already satisfied: typing-extensions>=3.7.4.3 in d:\anaconda\envs\lmcc\lib\site-packages (from huggingface-hub<1.0,>=0.34.0->transformers) (4.15.0)
Collecting colorama (from tqdm>=4.27->transformers)
  Using cached colorama-0.4.6-py2.py3-none-any.whl.metadata (17 kB)
Collecting charset-normalizer<4,>=2 (from requests->transformers)
  Downloading charset-normalizer-3.4.4-cp311-cp311-win_and64.whl.metadata (38 kB)
Collecting idna<4,>=2.5 (from requests->transformers)
  Downloading idna-3.11-py3-none-any.whl.metadata (8.4 kB)
Collecting urllib3<3,>=1.21.1 (from requests->transformers)
  Using cached urllib3-2.5.0-py3-none-any.whl.metadata (6.5 kB)
Collecting certifi>=2017.4.17 (from requests->transformers)
  Using cached certifi-2025.10.5-py3-none-any.whl.metadata (2.5 kB)
Using cached transformers-4.57.1-py3-none-any.whl (12.0 MB)
Using cached huggingface_hub-0.35.3-py3-none-any.whl (564 kB)
Using cached tokenizers-0.22.1-cp39-abi3-win_and64.whl (2.7 MB)
Using cached packaging-25.0-py3-none-any.whl (66 kB)
Using cached pyyaml-6.0.3-cp311-cp311-win_and64.whl (158 kB)
Downloading regex-2025.10.23-cp311-cp311-win_and64.whl (277 kB)
Using cached safetensors-0.6.2-cp38-abi3-win_and64.whl (320 kB)
Using cached tqdm-4.67.1-py3-none-any.whl (78 kB)
Using cached colorama-0.4.6-py2.py3-none-any.whl (25 kB)
Using cached requests-2.32.5-py3-none-any.whl (64 kB)
Downloading charset-normalizer-3.4.4-cp311-cp311-win_and64.whl (106 kB)
Downloading idna-3.11-py3-none-any.whl (71 kB)
Using cached urllib3-2.5.0-py3-none-any.whl (129 kB)
Using cached certifi-2025.10.5-py3-none-any.whl (163 kB)
Installing collected packages: urllib3, safetensors, regex, pyyaml, packaging, idna, colorama, charset-normalizer, certifi, tqdm, requests, huggingface-hub, tokenizers, transformers
Successfully installed certifi-2025.10.5 charset-normalizer-3.4.4 colorama-0.4.6 huggingface-hub-0.35.3 idna-3.11 packaging-25.0 pyyaml-6.0.3 regex-2025.10.23 requests-2.32.5 safetensors-0.6.2 tokenizers-0.22.1 tqdm-4.67.1 transformers-4.57.1 urllib3-2.5.0
```

2. Linux系统的环境配置

Step1：Python环境准备

在 linux 终端按顺序执行如下命令：

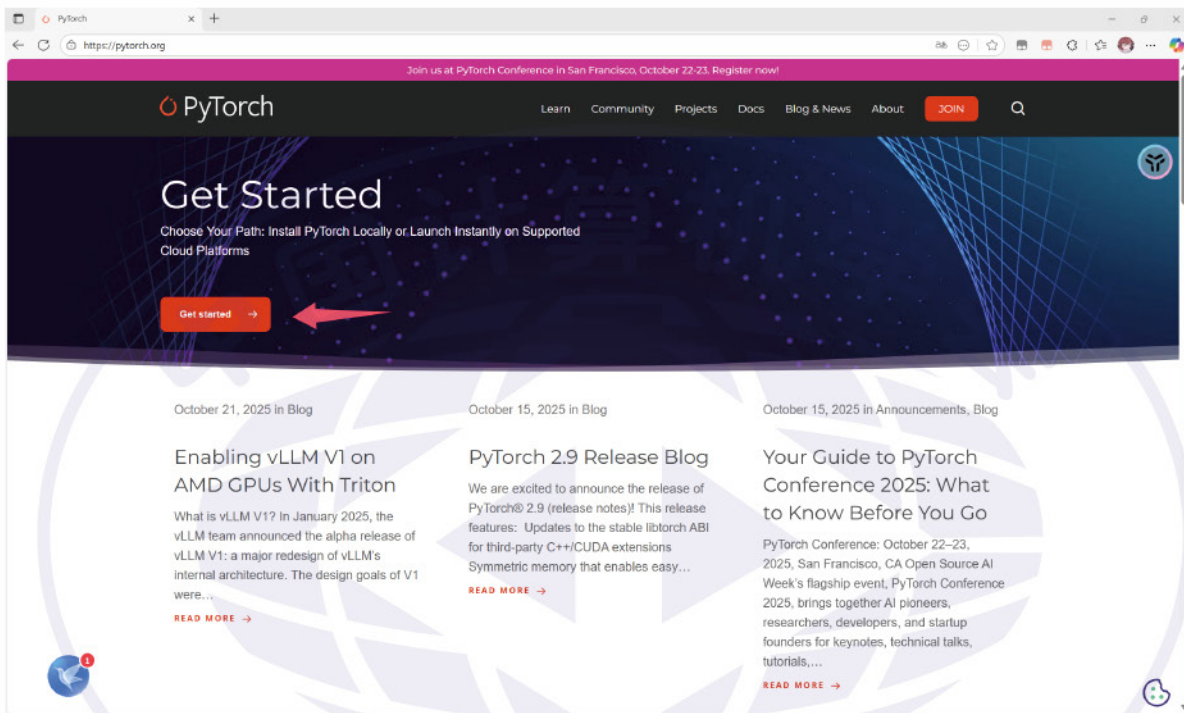
```
sudo apt update
```

```
sudo apt install -y python3 python3-venv python3-pip build-essential python3-dev
git curl libssl-dev libffi-dev
```

```
python3 -m pip install -U pip setuptools wheel
```

Step2: pytorch安装

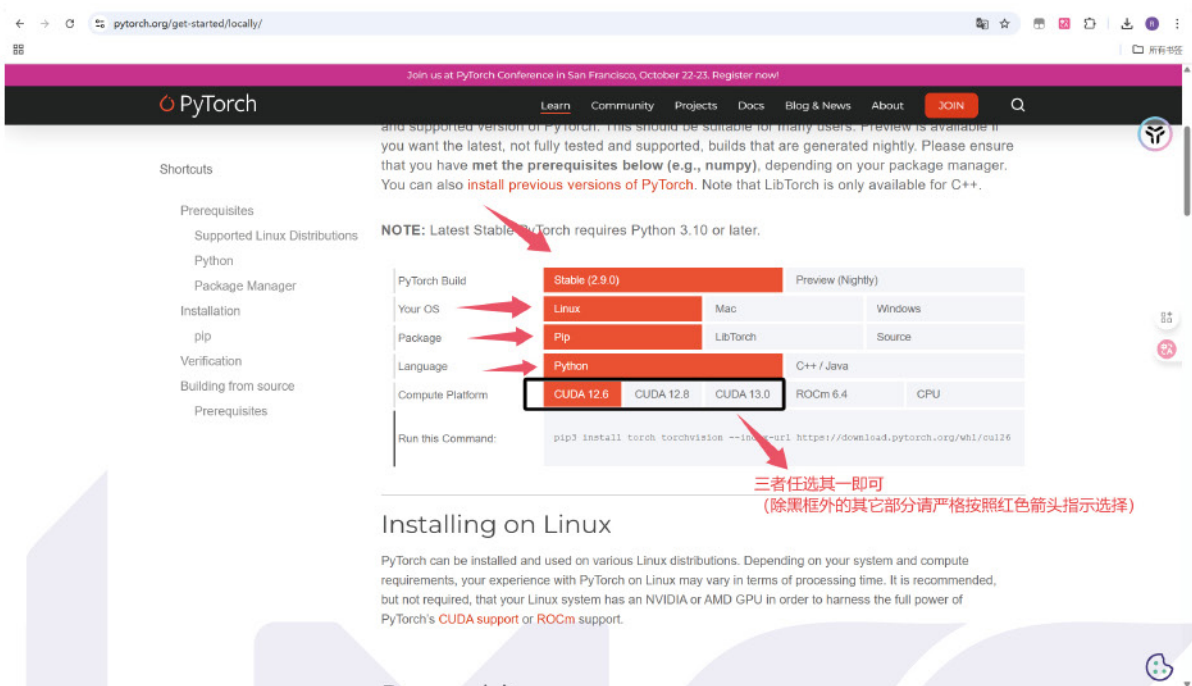
访问 <https://pytorch.org/> 网页，点击 **Get started**。



接下来请根据实际电脑的不同情况按照下面的指示进行安装：

1. 有显卡的Linux系统：

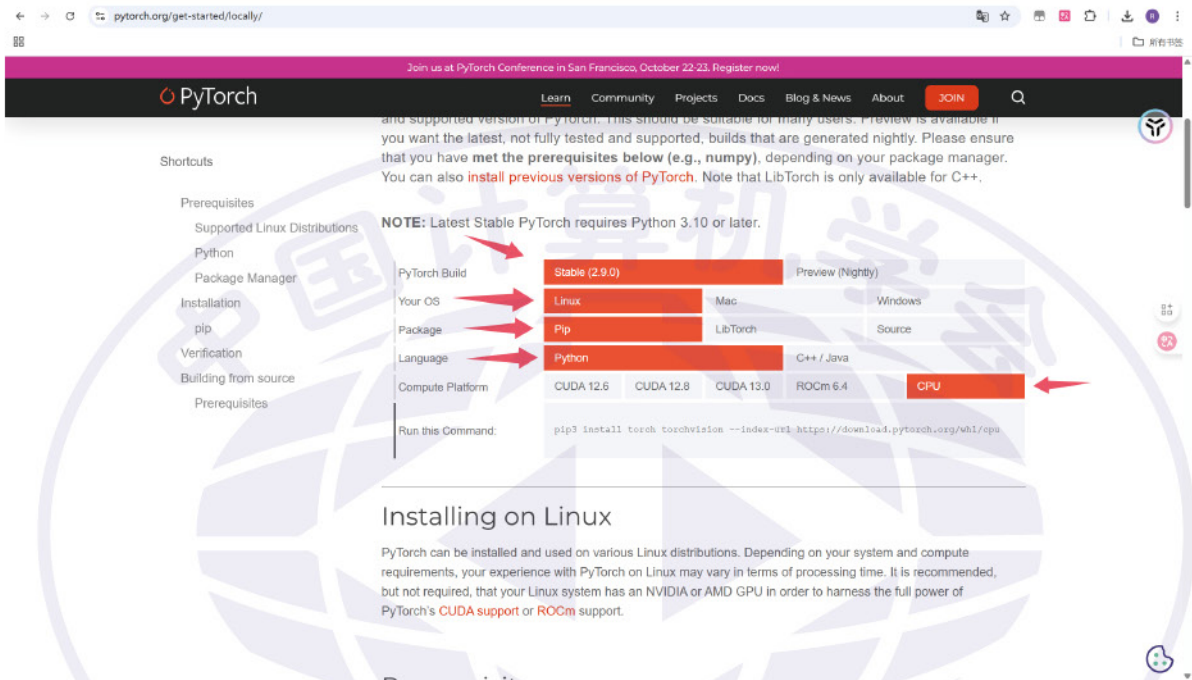
勾选下图红色箭头指示的部分后复制 **Run this Command** 后面的代码。



将刚才复制的代码通过右键黏贴到终端后，回车。

2. 无显卡的Linux系统：

勾选下图红色箭头指示的部分后复制 **Run this Command** 后面的代码。



将刚才复制的代码通过右键黏贴到终端后，回车。

Step 3: transformers下载：

复制下面的代码，右键黏贴到终端后，回车：

```
python -m pip install -U transformers
```

2. 代码解析

1. T1 代码解析：

```
# -*- coding: utf-8 -*-  
"""
```

仅此文件允许考生修改：

- 请在下列函数的函数体内完成实现。
- 不要改动函数名与参数签名。
- 你可以新增少量辅助函数。

```
"""
```

主要流程包含三部分：

- # 1) 构造 **system prompt**;
- # 2) 通过 **tokenizer** 的 **chat** 模板拼装单条对话输入;
- # 3) 基于 **HuggingFace Transformers** 的 **generate** 接口实现单条/批量推理。

```
from typing import List # 引入 List 用于类型注解（批量文本列表等）  
import torch # 引入 PyTorch，用于张量与设备迁移
```

```
from transformers import AutoTokenizer, AutoModelForCausalLM # 引入 HF 的分词器与
自回归语言模型抽象
```

```
# =====
# 第一部分: Prompt 定义
# =====
def build_system_prompt() -> str:
    """
    考生实现: 定义 system prompt
    - 返回一个 system prompt, 要求模型以 "[Answer]: " 的单行格式给出最终数值。
    """
    # ===== 考生实现区域 (可修改) =====
    # 下面注释掉的 prompt 是“非思考模式”的一个备选示例; 保留以供参考 (不参与实际返回)。
    # 用于非思考模式
    #     return """你是一个精确的数学计算助手。请仔细计算并输出最终数值。
    #
    # 【输出格式】
    # - 使用格式: [Answer]: 数值
    # - 例如: [Answer]: 42
    # - 整数直接输出, 确保计算准确
    # - **千万不要输出任何计算过程**
    #
    # """
    # 实际返回的 system prompt (启用分步说明): 引导模型
    # 1) 先进行符号标准化;
    # 2) 再进行分步计算;
    # 3) 最终仅用 “[Answer]: 数值” 输出答案。
    # 注意: 虽然提示词中描述了中间步骤, 但评测端通常只解析最终答案行。
    return """你是一个精确的数学计算助手。请按照以下步骤严格计算并返回准确的最终数值。
```

【计算步骤】

步骤1 - 符号转换:

- 将问题中的所有符号转换为标准数学符号
- +, $\boldsymbol{+}$, $\oplus$, 加 $\rightarrow$ +
- -, 减 $\rightarrow$ -
- $\times$, *, 乘 $\rightarrow$ $\times$
- $\div$, /, 除 $\rightarrow$ $\div$
- $\oplus$ 不是异或运算, 就是普通加法 +
- 输出转换后的标准计算公式

步骤2 - 分步计算:

- 按照运算优先级逐步计算
- 每步只写出计算结果, 不要过多解释

步骤3 - 输出答案:

- 使用格式: [Answer]: 数值
- 例如: [Answer]: 42
- 整数直接输出, 确保计算准确

```
"""
# ===== 考生实现区域 (可修改) =====
```

```
# =====
# 第二部分: 模板拼装
# =====
```

```

def apply_chat_template_single(
    tokenizer: AutoTokenizer,
    system_prompt: str,
    problem: str,
) -> str:
    """
    考生实现：将单个问题转换为模型输入文本
    - 使用 tokenizer.apply_chat_template
    - 返回拼装好的文本字符串
    """
    # ===== 考生实现区域（可修改） =====

    # 构造符合 chat 模板的消息列表：
    # - system: 提供系统级指令（约束输出格式与思路）
    # - user: 提供用户的具体题目文本
    messages = [
        {"role": "system", "content": system_prompt},
        {"role": "user", "content": problem},
    ]

    # 通过 tokenizer.apply_chat_template 将消息渲染为模型可消费的串联文本
    # - tokenize=False 表示返回字符串而非 token id
    # - add_generation_prompt=True 会在末尾追加生成起始标记（因不同模型模板而异）
    # - enable_thinking=True（若 tokenizer 支持）可启用“思考”相关特殊标记/格式
    rendered = tokenizer.apply_chat_template(
        messages,
        tokenize=False,
        add_generation_prompt=True,
        enable_thinking=True,
    )

    return rendered # 返回拼装好的原始文本，供后续 tokenize/生成使用
    # ===== 考生实现区域（可修改） =====

# =====
# 第三部分：核心推理实现
# =====

def generate_single(
    model: AutoModelForCausalLM,
    tokenizer: AutoTokenizer,
    rendered_text: str,
    max_new_tokens: int,
    do_sample: bool,
) -> torch.Tensor:
    # ===== 考生实现区域（可修改） =====

    # 1) 将渲染好的文本进行 tokenize:
    # - return_tensors="pt" 返回 PyTorch 张量
    # - padding=True 在批量场景中对齐序列长度（这里单条输入也可安全启用）
    # - .to(model.device) 将输入迁移到与模型相同的设备（CPU/GPU）
    inputs = tokenizer(rendered_text, return_tensors="pt",
padding=True).to(model.device)

    # 2) 使用自回归 generate 生成输出:
    # - input_ids / attention_mask 来自上一步的编码结果
    # - max_new_tokens 控制新增生成长度
    # - do_sample 控制是否采样（True 为采样，False 为贪心/束搜索等）
    # - eos_token_id 指定终止 token（避免生成无限延伸）

```

```

# - pad_token_id 指定补齐 token (便于批量对齐)
outputs = model.generate(
    input_ids=inputs["input_ids"],
    attention_mask=inputs.get("attention_mask"),
    max_new_tokens=max_new_tokens,
    do_sample=do_sample,
    eos_token_id=tokenizer.eos_token_id,
    pad_token_id=tokenizer.pad_token_id,
)

return outputs # 返回模型生成的 token 序列 (包含输入与生成段, 视模型而定)
# ===== 考生实现区域 (可修改) =====

def generate_batch(
    model: AutoModelForCausalLM,
    tokenizer: AutoTokenizer,
    rendered_texts: List[str],
    max_new_tokens: int,
    do_sample: bool,
) -> List[torch.Tensor]:
    # ===== 考生实现区域 (可修改) =====

    all_outputs = [] # 用于收集每个批次的生成结果
    batch_size: int = 8 # 设定批量大小 (可根据显存调整)

    # 分批处理所有输入文本, 避免一次性送入超出显存
    for i in range(0, len(rendered_texts), batch_size):
        batch_texts = rendered_texts[i:i + batch_size] # 切片取当前批次文本

        # 1) 批量 tokenize:
        # - padding=True: 对齐序列长度
        # - truncation=True: 超长截断以适配模型的最大长度限制
        # - 返回张量并迁移到模型设备
        batch_inputs = tokenizer(
            batch_texts,
            return_tensors="pt",
            padding=True,
            truncation=True,
        ).to(model.device)

        # 2) 批量生成:
        # - 与单条生成参数一致, 只是输入改为批量
        batch_outputs = model.generate(
            input_ids=batch_inputs["input_ids"],
            attention_mask=batch_inputs.get("attention_mask"),
            max_new_tokens=max_new_tokens,
            do_sample=do_sample,
            eos_token_id=tokenizer.eos_token_id,
            pad_token_id=tokenizer.pad_token_id,
        )

        all_outputs.append(batch_outputs) # 将当前批次结果加入列表

    # 返回所有批次的输出列表 (每个元素通常是形如 [batch, seq_len] 的张量)
    return all_outputs

# ===== 考生实现区域 (可修改) =====

```

2. T2 代码解析:

```
# -*- coding: utf-8 -*-
"""
仅此文件允许考生修改：
- 请在下列函数的函数体内完成实现。
- 不要改动函数名与参数签名。
- 你可以新增少量辅助函数。
"""

import torch # 引入 PyTorch 主包，用于张量运算与设备管理
import torch.nn.functional as F # 引入常用函数式 API，这里用于向量归一化
from transformers import AutoTokenizer, AutoModel # 引入 HF 的自动分词器与自动模型基
类

def _last_token_pool(last_hidden_states: torch.Tensor, attention_mask:
torch.Tensor) -> torch.Tensor:
    """
    Last-token pooling

    参数：
        last_hidden_states: 模型输出的最后一层隐藏状态 (batch_size, seq_len,
hidden_dim)
        attention_mask: 注意力掩码 (batch_size, seq_len)

    返回：
        句向量 (batch_size, hidden_dim)
    """
    # 判断是否使用了“左侧 padding”
    # 逻辑：如果 attention_mask 的最后一列（序列末尾位置）对每个样本都为 1，
    # 则说明有效 token 延伸到序列末尾，padding（若有）在左侧；反之多为右侧 padding。
    left_padding = (attention_mask[:, -1].sum() == attention_mask.shape[0])

    if left_padding:
        # 左侧 padding 情况：序列末尾必为有效 token，直接取最后一个位置的隐状态作为句向量
        return last_hidden_states[:, -1]
    else:
        # 右侧 padding 情况：需要定位每个样本“最后一个有效 token”的索引
        # attention_mask 为 1 的位置是有效 token，sum(dim=1) 得到每个样本有效长度
        # 减 1 即得到最后一个有效 token 在序列维度上的索引
        seq_lens = attention_mask.sum(dim=1) - 1 # 形状 (batch_size,)
        batch_size = last_hidden_states.shape[0] # 当前批次大小
        # 使用高级索引，按样本维度逐一取对应的最后有效 token 向量
        return last_hidden_states[torch.arange(batch_size,
device=last_hidden_states.device), seq_lens]

def compute_similarity(text1: str, text2: str, model: AutoModel, tokenizer:
AutoTokenizer) -> float:
    """
    参考实现：计算两个文本之间的相似度
    """
```

参数:

text1: 第一个文本字符串
text2: 第二个文本字符串
model: 预加载的 `AutoModel`
tokenizer: 预加载的 `AutoTokenizer`

返回:

相似度值 (0.0 到 1.0 之间的浮点数)

步骤:

1. 对文本进行分词和编码
2. 获取模型输出
3. 使用 `last-token pooling` 提取句向量
4. L2 归一化
5. 计算余弦相似度 (点积)

"""

取模型当前所在设备 (CPU/GPU), 用于把输入张量迁移到相同设备

`device = next(model.parameters()).device`

1. 对文本进行分词和编码

```
inputs = tokenizer(  
    [text1, text2],          # 将两个文本组成一个批次, 便于一次前向  
    padding=True,           # 对齐到批次中最长序列 (自动 padding)  
    truncation=True,        # 超过最大长度则截断  
    max_length=8192,        # 针对可支持的长上下文模型 (如 Qwen3-Embedding) 的上限  
    return_tensors="pt",    # 返回 PyTorch 张量格式  
)
```

将编码后的各项 (`input_ids`, `attention_mask` 等) 迁移到与模型相同的设备

`inputs = {k: v.to(device) for k, v in inputs.items()}`

2. 获取模型输出 (推理时关闭梯度以节省显存与加速)

`with torch.no_grad():`

`outputs = model(**inputs)` # 典型返回值包含 `last_hidden_state` 等字段

3. 使用 `last-token pooling` 提取句向量

`last_hidden_states = outputs.last_hidden_state` # 形状 (batch=2, seq_len, hidden_dim)

`sentence_embeddings = _last_token_pool(last_hidden_states, inputs["attention_mask"])` # 形状 (2, hidden_dim)

4. L2 归一化: 把每个向量缩放为单位范数 (长度为 1), 便于后续用点积表示余弦相似度

`sentence_embeddings = F.normalize(sentence_embeddings, p=2, dim=1)`

5. 计算余弦相似度: 归一化后两个向量的点积即为 `cos` 相似度, 范围 `[-1, 1]`

由于文本嵌入常为非负或分布集中, 在多数场景该值位于 `[0, 1]` 更直观

`similarity = (sentence_embeddings[0] @ sentence_embeddings[1]).item()`

返回标量相似度 (float)

`return similarity`

T2第二部分的随机数学问题生成方式多种多样, 只需在随机生成后再经由同第一部分相似运算, 保证总数据集平均相似度不超过0.5即可, 下面仅列出一种可能的实现方式:

```

import torch # 引入 PyTorch 主包, 用于张量运算与设备管理
import torch.nn.functional as F # 引入常用函数式 API, 这里用于向量归一化
from transformers import AutoTokenizer, AutoModel # 引入 HF 的自动分词器与自动模型基
类
import random, json # 引入随机数与 JSON 序列化库 (本文件主要用于构建/落盘数据集)

def generate_dataset(outfile="dataset.jsonl"):
    """
    生成一个满足“答案为三位数”的四则运算数据集, 并以 JSONL 形式落盘。

    参数:
        outfile (str): 输出文件名 (位于脚本同级目录下的 data/ 子目录中)
    返回:
        dataset (list[dict]): 内存中的数据列表 (每行包含 problem 与 answer)
    """
    import os # 局部引入 os, 避免污染外部命名空间

    target_count = 1024 # 目标样本数量

    # 选择推理设备: 优先使用 GPU (cuda), 否则回退到 CPU
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

    # 加载嵌入模型与分词器 (Qwen3-Embedding-0.6B 变体)
    # trust_remote_code=True 以支持模型仓库中的自定义代码
    tokenizer = AutoTokenizer.from_pretrained("Qwen/Qwen3-Embedding-0.6B",
trust_remote_code=True)
    model = AutoModel.from_pretrained("Qwen/Qwen3-Embedding-0.6B",
trust_remote_code=True).to(device)
    model.eval() # 切换到推理模式, 禁用 dropout 等

    ops = ['+', '-', '*', '/'] # 支持的四则运算符集合

    def safe_calc(a, op, b):
        """
        安全计算二元表达式 a (op) b:
        - 加减乘按常规返回整数
        - 除法仅在 b!=0 且整除时返回整数 (否则返回 None)

        返回:
            int 或 None
        """
        if op == '+':
            return a + b
        if op == '-':
            return a - b
        if op == '*':
            return a * b
        # 下方处理除法: 若除数为 0 或不能整除, 判定为无效
        if b == 0 or a % b != 0:
            return None
        return a // b # 整除结果

    def build_binary():
        """
        随机构造一个二元 (两个操作数) 的四则运算表达式, 要求结果在 [100, 999]。

```

策略:

- 按不同运算符设置不同的采样范围（便于更高概率命中三位数结果）
- 最多尝试 200 次，若均未命中则返回 None

返回：

```
tuple[str, str] 或 None
e.g. ("123 + 456", "579")
"""
for _ in range(200):
    op = random.choice(ops) # 随机选取运算符
    if op == '+':
        # 加法：两个中等范围数，提升得到三位数的概率
        a = random.randint(10, 550)
        b = random.randint(10, 550)
    elif op == '-':
        # 减法：使被减数足够大、减数不至过大，便于结果落在三位数
        a = random.randint(200, 1400)
        b = random.randint(10, a - 100)
    elif op == '*':
        # 乘法：两位数相乘更容易得到三位数
        a = random.randint(10, 99)
        b = random.randint(10, 99)
    else:
        # 除法：先定结果与除数，再反推被除数，确保整除且结果为三位数
        result = random.randint(100, 999)
        b = random.randint(2, 12)
        a = result * b
    # 进行安全计算（对除法进行严格约束）
    result = safe_calc(a, op, b)
    # 若无效或不在三位数范围，则继续尝试
    if result is None or result < 100 or result > 999:
        continue
    # 命中则返回表达式与字符串形式的结果
    return f"{a} {op} {b}", str(result)
return None # 超过尝试次数未命中
```

def build_ternary():

"""

随机构造一个三元（带一个括号）四则运算表达式，要求结果在 [100, 999]。

策略：

- 表达式形如 "(a op1 b) op2 c" 或 "a op1 (b op2 c)"，两种括号位置随机
- 对每个中间步骤使用 safe_calc 过滤非法/非整除情况
- 最多尝试 400 次，若均未命中则返回 None

返回：

```
tuple[str, str] 或 None
e.g. ("(20 * 5) + 12", "112")
"""
for _ in range(400):
    op1, op2 = random.choice(ops), random.choice(ops) # 两个运算符
    nums = [random.randint(10, 120) for _ in range(3)] # 三个操作数
    if random.choice([True, False]):
        # 形式一：先算 (a op1 b)，再与 c 进行 op2
        first = safe_calc(nums[0], op1, nums[1])
        if first is None:
            continue
        result = safe_calc(first, op2, nums[2])
        expr = f"({nums[0]} {op1} {nums[1]}) {op2} {nums[2]}"
```

```

else:
    # 形式二: 先算 (b op2 c), 再与 a 进行 op1
    second = safe_calc(nums[1], op2, nums[2])
    if second is None:
        continue
    result = safe_calc(nums[0], op1, second)
    expr = f"{nums[0]} {op1} ({nums[1]} {op2} {nums[2]})"
    # 检查结果有效且为三位数
    if result is None or result < 100 or result > 999:
        continue
    # 命中则返回表达式与字符串形式的结果
    return expr, str(result)
return None # 超过尝试次数未命中

# 序列化函数: 将条目转为 JSON 字符串 (ensure_ascii=True 输出纯 ASCII, 中文会转义)
serialize = lambda entry: json.dumps(entry, ensure_ascii=True)

dataset, dataset_texts = [], [] # dataset 存结构化样本; dataset_texts 存其 JSON
文本
used_problems = set() # 已使用的题目表达式 (去重用)
rejected_problems = set() # 已拒绝的题目表达式 (若有相似度筛失败等, 可放入此处; 当前逻辑未使用)
max_attempts = target_count * 500 # 最大尝试轮数, 避免极端情况下的无限循环
attempts = 0 # 记录已尝试次数

emb = [] # 保存已接受样本的嵌入向量 (list[Tensor], 每个是一维单位向量)

# 主循环: 不断采样候选题目, 直至达到目标数量或超过尝试上限
while len(dataset) < target_count and attempts < max_attempts:
    attempts += 1
    # 以 40% 概率生成三元表达式, 60% 概率生成二元表达式
    builder = build_ternary if random.random() < 0.4 else build_binary
    candidate = builder()
    if candidate is None:
        # 构造失败 (未命中三位数范围等), 继续下一轮
        continue
    problem, answer = candidate
    # 去重: 若题目已被使用或已被标记为拒绝, 则跳过
    if problem in used_problems or problem in rejected_problems:
        continue

    # 组装条目并序列化为 JSON 文本, 用于计算文本嵌入 (加入上下文有助于去重)
    entry = {"problem": problem, "answer": answer}
    entry_text = serialize(entry)

    # 对 entry_text 编码为模型输入 (padding/truncation 适配模型上下文长度)
    inputs = tokenizer(
        entry_text,
        padding=True, # 单条也可 padding (与批处理一致)
        truncation=True, # 超长截断, 避免超过模型上限
        max_length=8192, # 模型最大长度 (与嵌入模型匹配)
        return_tensors="pt", # 返回 PyTorch 张量
    )
    # 将输入迁移到推理设备 (CPU/GPU)
    inputs = {k: v.to(device) for k, v in inputs.items()}
    # 前向计算 (关闭梯度, 节省显存/加速)
    with torch.no_grad():
        outputs = model(**inputs)

```

```

# 取最后一层隐藏状态（形状：[batch=1, seq_len, hidden]）
last_hidden_states = outputs.last_hidden_state
# 使用外部定义的 last-token pooling（需在其他文件中提供该函数）
sentence_embeddings = _last_token_pool(last_hidden_states,
inputs["attention_mask"])
# L2 归一化后得到单位向量；squeeze(0) 去掉 batch 维；detach+cpu 便于后续处理与存
储
entry_vec = F.normalize(sentence_embeddings, p=2,
dim=1).squeeze(0).detach().cpu() # (hidden,)

# 与已有样本做余弦相似度筛查：若与任意已存样本 > 0.5，则视为过于相似
too_similar = False
for text_emb in emb: # text_emb 也是一维单位向量
    similarity = torch.dot(entry_vec, text_emb).item() # 归一化后 dot 即余
弦相似度
    if similarity > 0.5:
        too_similar = True
        break

if not too_similar:
    # 若不过于相似，则将该嵌入加入库；用作后续样本的相似度对比
    emb.append(entry_vec) # 存一维向量（不合并为矩阵以简化逻辑）

# 注意：无论是否“too_similar”，下方仍会把该题目加入 used_problems 与 dataset
# 这意味着相似度筛查只控制 embedding 库的更新，而非严格过滤样本进入数据集
used_problems.add(problem) # 标记该题目已使用（避免重复）
dataset.append(entry) # 加入数据列表
dataset_texts.append(entry_text) # 记录其 JSON 文本

# 若最终数量不足目标值，则抛出异常（提示需放宽约束或增加尝试次数）
if len(dataset) < target_count:
    raise RuntimeError("Unable to generate dataset that satisfies similarity
constraints.")

# 组织输出路径：写入当前脚本目录下的 data/ 子目录
data_dir = os.path.join(os.path.dirname(__file__), "data")
os.makedirs(data_dir, exist_ok=True) # 确保目录存在
dataset_path = os.path.join(data_dir, outfile)
# 逐行写入 JSONL 文件（ensure_ascii=True: 统一 ASCII 编码，便于跨环境处理）
with open(dataset_path, "w", encoding="utf-8") as f:
    for item in dataset:
        f.write(json.dumps(item, ensure_ascii=True) + "\n")

return dataset # 返回内存中的数据副本（方便调用方复用）

# 直接执行脚本时，调用生成函数并使用默认输出文件名
generate_dataset()

```

3. 评测

从 <https://huggingface.co/> 网页下载 Qwen3-Embedding-0.6B 与 Qwen3-0.6B 到本地，然后将下载好的Qwen\Qwen3-0.6B 放在T1的submission.py同级目录下，将Qwen\Qwen3-Embedding-0.6B放在T2的submission.py同级目录下。

在下面三条代码中任选一种运行即可进行本地评测

```
python evaluate.py
python evaluate.py --mode demo
python evaluate.py --mode grading
```

```
(LMCC) PS D:\LMCC-例卷-机试\T2\T2> python evaluate.py --mode grading
```

```
=====
评分模式 - T2 多样性数据生成与相似度计算
=====
```

```
正在加载考生模型 (transformers) ...
```

```
✅ 考生模型加载完成
```

```
==== 第一部分：相似度计算准确性 ====
```

```
[1/10] ✓ (完全相同)
```

```
[2/10] ✓ (符号不同)
```

```
[3/10] ✓ (顺序不同)
```

```
[4/10] ✓ (文字表达)
```

```
[5/10] ✓ (带前缀)
```

```
[6/10] ✓ (运算不同)
```

```
[7/10] ✓ (数字不同)
```

```
[8/10] ✓ (完全不同)
```

```
[9/10] ✓ (复杂表达式)
```

```
[10/10] ✓ (幂运算)
```

```
✅ 第一部分满分，继续第二部分
```

```
==== 第二部分：数据多样性 ====
```

```
正在计算平均相似度...
```

```
计算完成，耗时：2.46 秒
```

```
=====
===== 最终评分 =====
=====
```

```
【第一部分：相似度计算准确性】
```

```
通过测试用例：10 / 10
```

```
得分：50 / 50 (每组5分，≥8组满分)
```

