

## 实验报告四 4

24 计算机科学与技术吴虹霖 学号: 24020007135

2025 年 9 月 19 日

# 一、程序性能与调试

## 实例 1.1 line profiler 比较插入快速排序性能

操作: 使用 kernprof 命令运行 line profiler, 分析插入排序和快速排序的性能差异。

```
PS C:\Users\吴虹霖\Desktop\Codes\2025小学用codes> kernprof -l -v sorts.py
sorts.py:54: UserWarning: Adding a function with a '._wrapped__' attribute. You may want to profile the wrapped function by adding 'insertionsort.__wrapped__' instead.
  ad.
  profiler.add_function(insertionsort)
sorts.py:55: UserWarning: Adding a function with a '._wrapped__' attribute. You may want to profile the wrapped function by adding 'quicksort.__wrapped__' instead.
  profiler.add_function(quicksort)
sorts.py:56: UserWarning: Adding a function with a '._wrapped__' attribute. You may want to profile the wrapped function by adding 'quicksort_inplace.__wrapped__' instead.
  profiler.add_function(quicksort_inplace)
Testing insertionsort...
Testing quicksort...
Testing quicksort_inplace...
Timer unit: 1e-07 s

Total time: 0.0063083 s
File: D:\python\lib\site-packages\line_profiler\profiler_mixin.py
Function: @CountProfilerMixin.wrap_function.<locals>.wrapper at line 410

=====
Line #      Hits         Time  Per Hit   % Time  Line Contents
=====
410                                     @functools.wraps(func)
411                                     def wrapper(*args, **kwargs):
412                                     self.enable_by_count()
413                                     try:
414                                     result = func(*args, **kwargs)
415                                     finally:
416                                     self.disable_by_count()
417                                     return result

Wrote profile results to 'sorts.py.lprof'
Timer unit: 1e-06 s

Total time: 0.0026848 s
File: sorts.py
Function: insertionsort at line 7

=====
Line #      Hits         Time  Per Hit   % Time  Line Contents
=====
7                                     @profile
8                                     def insertionsort(array):
9                                     for i in range(len(array)):
10                                    j = i-1
11                                    v = array[i]
12                                    while j >= 0 and v < array[j]:
13                                    array[j+1] = array[j]
14                                    array[j+1] = v
15                                    j -= 1
16                                    return array
```

图 1: line profiler 比较插入快速排序性能

## 实例 1.2 cprofile 比较插入快速排序性能

操作: 使用 cProfile 运行性能分析, 获取函数调用统计信息。

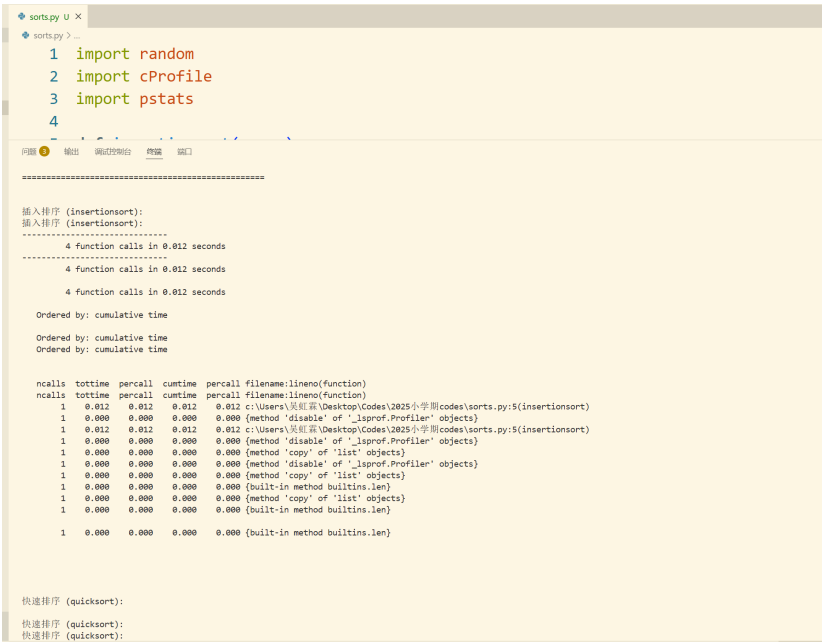


图 2: shellcheck 检查脚本

实例 1.3 shellcheck 检查脚本

操作： 运行 shellcheck 命令检查脚本，获取改进建议。

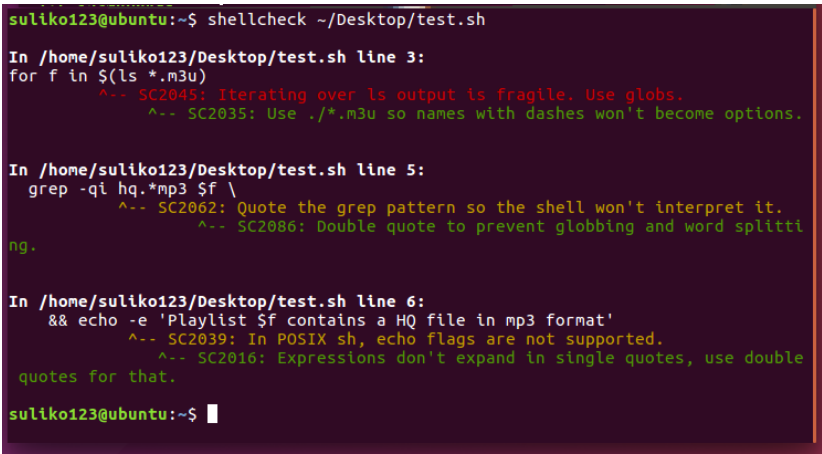


图 3: shellcheck 检查脚本

实例 1.4 snakeviz 可视化程序性能

操作： 运行 cProfile 生成性能数据，使用 snakeviz 进行可视化分析。

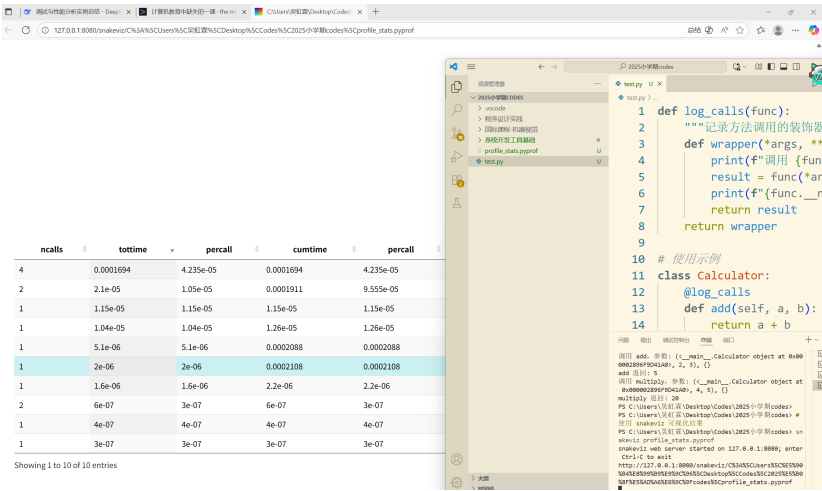


图 4: snakeviz 可视化程序性能

实例 1.5 调试查看日志

操作: 使用 journalctl 命令查看特定时间范围或特定用户的系统日志。

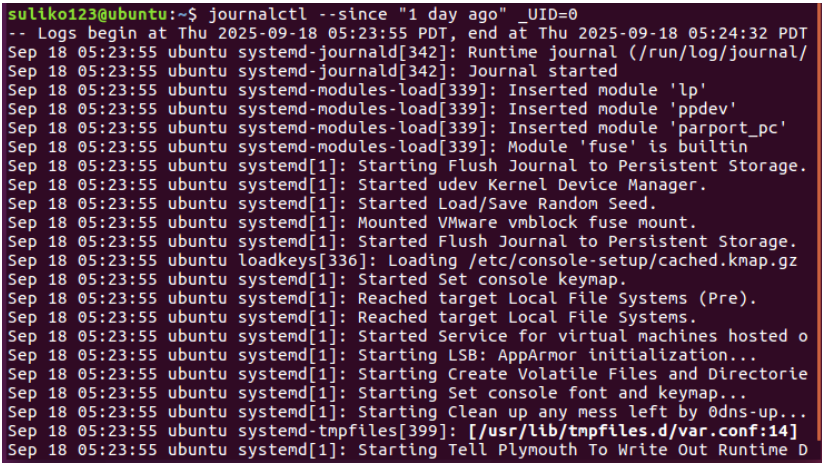


图 5: 调试查看日志

2 二、元编程

实例 2.1 type() 动态创建类

操作: 使用 type() 函数传入类名、基类和属性字典来动态创建类。

```
test.py U x
test.py > ...
1 # 使用 type() 函数动态创建一个类
2 def __init__(self, name):
3     self.name = name
4
5 def say_hello(self):
6     return f"Hello, {self.name}!"
7
8 # 使用 type() 创建类
9 MyDynamicClass = type('MyDynamicClass', (), {
10     '__init__': __init__,
11     'say_hello': say_hello
12 })
13
14 # 测试动态创建的类
15 obj = MyDynamicClass("World")
16 print(obj.say_hello())# Output: Hello, World!
```

问题 输出 调试控制台 终端 窗口

PS C:\Users\吴虹霖\Desktop\Codes\2025小学期codes> & D:/python/python.exe c:/Users/吴虹霖/Desktop/Codes/2025小学期codes/test.py  
Hello, World!  
PS C:\Users\吴虹霖\Desktop\Codes\2025小学期codes>

图 6: type() 动态创建类

## 实例 2.2 注册元类

**操作:** 定义元类并在创建类时自动注册到全局注册表中。

```
test.py > ...
10
11 class BaseModel(metaclass=RegistryMeta):
12     """所有模型类的基类"""
13     def __init__(self, name):
14         self.name = name
15
16     def describe(self):
17         return f"{self.__class__.__name__}: {self.name}"
18
19 class User(BaseModel):
20     pass
21
22 class Product(BaseModel):
23     pass
24
25 class Order(BaseModel):
```

问题 输出 调试控制台 终端 窗口

PS C:\Users\吴虹霖\Desktop\Codes\2025小学期codes> & D:/python/python.exe c:/Users/吴虹霖/Desktop/Codes/2025小学期codes/test.py  
User: Alice  
Product: Laptop  
注册的类: dict\_keys(['User', 'Product', 'Order'])  
PS C:\Users\吴虹霖\Desktop\Codes\2025小学期codes>

图 7: type() 动态创建类

## 实例 2.3 动态函数创建

**操作:** 使用字符串拼接函数代码，通过 exec() 函数执行并创建函数对象。

```
test.py > ...
1 # 使用 exec() 动态创建函数
2 def create_function(name, operation):
3     func_code = f"""
4     def {name}(a, b):
5         return a {operation} b
6     """
7     exec(func_code, globals())
8     return globals()[name]
9
10 # 测试
11 add_func = create_function('dynamic_add', '+')
12 multiply_func = create_function('dynamic_multiply', '*')
13
14 print(add_func(5, 3))
15 print(multiply_func(5, 3))
```

问题 输出 调试控制台 终端 窗口

PS C:\Users\吴虹霖\Desktop\Codes\2025小学期codes> & D:/python/python.exe c:/Users/吴虹霖/Desktop/Codes/2025小学期codes/test.py

8

15

PS C:\Users\吴虹霖\Desktop\Codes\2025小学期codes>

图 8: 动态函数创建

实例 2.4 动态属性访问

操作： 重写特殊方法实现属性的动态获取和设置。

```
test.py U X
test.py > ...
1 class DynamicAttributes:
6     def __getattr__(self, name):
15         return getter
16         raise AttributeError(f"{self.__class__.__name__} object has no attribute '{name}'")
17
18     def __setattr__(self, name, value):
19         """设置属性"""
20         if name == '_data':
21             super().__setattr__(name, value)
22         else:
23             self._data[name] = value
24
25     def __dir__(self):
26         """返回可用属性列表"""
27         return super().__dir__() + list(self._data.keys())
28
29
```

问题 输出 调试控制台 终端 窗口

PS C:\Users\吴虹霖\Desktop\Codes\2025小学期codes> & D:/python/python.exe c:/Users/吴虹霖/Desktop/Codes/2025小学期codes/test.py

Dynamic Object

42

Dynamic Object

所有属性: ['\_\_class\_\_', '\_\_delattr\_\_', '\_\_dict\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_firstlineno\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattr\_\_', '\_\_getattribute\_\_', '\_\_getstate\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_init\_subclass\_\_', '\_\_le\_\_', '\_\_lt\_\_', '\_\_module\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', '\_\_weakref\_\_', '\_data', 'name', 'value']

PS C:\Users\吴虹霖\Desktop\Codes\2025小学期codes>

图 9: 动态属性访问

实例 2.5 方法调用日志装饰器

操作： 定义装饰器函数，包装目标方法并添加日志记录功能。

```
test.py U x
test.py > ...
1 def log_calls(func):
2     """记录方法调用的装饰器"""
3     def wrapper(*args, **kwargs):
4         print(f"调用 {func.__name__}, 参数: {args}, {kwargs}")
5         result = func(*args, **kwargs)
6         print(f"{func.__name__} 返回: {result}")
7         return result
8     return wrapper
9
10 # 使用示例
11 class Calculator:
12     @log_calls
13     def add(self, a, b):
14         return a + b
15
16     @log_calls
17     def multiply(self, a, b):
18         return a * b
19
20 # 测试
21 obj = Calculator()
22 print(obj.add(2, 3))
23 print(obj.multiply(4, 5))
```

问题 输出 调试控制台 终端 窗口

PS C:\Users\吴虹霖\Desktop\Codes\2025小学期codes> & D:/python/python.exe c:/Users/吴虹霖/Desktop/Codes/2025小学期codes/test.py  
调用 add, 参数: (<\_\_main\_\_.Calculator object at 0x0000020FFAB66F90>, 2, 3), {}  
add 返回: 5  
调用 multiply, 参数: (<\_\_main\_\_.Calculator object at 0x0000020FFAB66F90>, 4, 5), {}  
multiply 返回: 20  
PS C:\Users\吴虹霖\Desktop\Codes\2025小学期codes>

图 10: 方法调用日志装饰器

## 实例 2.6 属性拦截

操作: 重写 `getattr` 方法, 对特定模式的属性访问进行特殊处理。

```
test.py > ...
1 # 创建一个类, 拦截未定义的属性访问
2 class AttributeInterceptor:
3     def __getattr__(self, name):
4         if name.startswith('get_'):
5             attr_name = name[4:]
6             return lambda: f"Value of {attr_name} is not defined"
7         return f"Attribute {name} does not exist"
8
9 # 测试
10 obj = AttributeInterceptor()
11 print(obj.get_test())
12 print(obj.non_existent)
```

问题 输出 调试控制台 终端 窗口

PS C:\Users\吴虹霖\Desktop\Codes\2025小学期codes> & D:/python/python.exe c:/Users/吴虹霖/Desktop/Codes/2025小学期codes/test.py  
Value of test is not defined  
Attribute non\_existent does not exist  
PS C:\Users\吴虹霖\Desktop\Codes\2025小学期codes>

图 11: 属性拦截

## 实例 2.7 原编程装饰器

操作: 定义装饰器函数, 在函数执行前后记录时间并计算耗时。

```
test.py > timer
1 # 创建一个计时装饰器，测量函数执行时间
2 import time
3
4 def timer(func):
5     def wrapper(*args, **kwargs):
6         start = time.time()
7         result = func(*args, **kwargs)
8         end = time.time()
9         print(f"func.__name__ executed in {end - start:.4f} seconds")
10        return result
11    return wrapper
12
13 # 测试你的装饰器
14 @timer
15 def example_function(n):
16     return sum(range(n))
17
```

PS C:\Users\吴虹霖\Desktop\Codes\2025小学期codes> & D:/python/python.exe c:/Users/吴虹霖/Desktop/Codes/2025小学期codes/test.py  
example\_function executed in 0.0078 seconds  
0.007800000  
PS C:\Users\吴虹霖\Desktop\Codes\2025小学期codes>

图 12: 原编程装饰器

3 三、PyTorch

实例 3.1 PyTorch 张量操作

操作： 使用 torch 模块创建张量，进行逐元素运算和属性访问。

```
test.py > --
1 import torch
2
3 # 设置数据类型和设备
4 dtype = torch.float # 张量数据类型为浮点型
5 device = torch.device("cpu") # 本次计算在 CPU 上进行
6
7 # 创建并打印两个随机张量 a 和 b
8 a = torch.randn(2, 3, device=device, dtype=dtype) # 创建一个 2x3 的随机张量
9 b = torch.randn(2, 3, device=device, dtype=dtype) # 创建另一个 2x3 的随机张量
10
11 print("张量 a:")
12 print(a)
13
14 print("张量 b:")
15 print(b)
16
17 # 逐元素相乘并输出结果
```

PS C:\Users\吴虹霖\Desktop\Codes\2025小学期codes> & D:/python/python.exe c:/Users/吴虹霖/Desktop/Codes/2025小学期codes/test.py  
张量 a:  
tensor([[ 1.0385, -1.1261, -0.9968],  
 [ 0.0052, -0.5329, 0.2076]])  
张量 b:  
tensor([[ 0.1700, -0.2768, 0.6485],  
 [ 0.2375, 0.6744, -0.5768]])  
a 和 b 的逐元素乘积:  
tensor([[ 0.1752, 0.3117, -0.6459],  
 [ 0.0012, -0.3594, -0.1197]])  
张量 a 所有元素的总和:  
tensor(-1.4117)  
张量 a 第 2 行第 3 列的元素:  
tensor(0.2076)  
张量 a 中的最大值:  
tensor(1.0385)  
PS C:\Users\吴虹霖\Desktop\Codes\2025小学期codes>

图 13: PyTorch 张量操作

实例 3.2 PyTorch 激活函数

操作： 使用 torch.nn.functional 模块调用不同的激活函数处理张量。

```
test.py U x
test.py > ...
1 import torch
2 import torch.nn.functional as F
3
4 # 首先创建一个示例输入张量
5 input_tensor = torch.tensor([-1.0, 0.0, 1.0, 2.0])
6
7 # ReLU 激活
8 output_relu = F.relu(input_tensor)
9 print("ReLU output:", output_relu)
10
11 # Sigmoid 激活
12 output_sigmoid = torch.sigmoid(input_tensor)
13 print("Sigmoid output:", output_sigmoid)
14
15 # Tanh 激活
16 output_tanh = torch.tanh(input_tensor)
17 print("Tanh output:", output_tanh)
```

问题 输出 调试控制台 终端 窗口

PS C:\Users\吴虹霖\Desktop\Codes\2025小学期codes> & D:/python/python.exe c:/Users/吴虹霖/Desktop/Codes/2025小学期codes/test.py  
ReLU output: tensor([0., 0., 1., 2.])  
Sigmoid output: tensor([0.2689, 0.5000, 0.7311, 0.8808])  
Tanh output: tensor([-0.7616, 0.0000, 0.7616, 0.9640])  
PS C:\Users\吴虹霖\Desktop\Codes\2025小学期codes>

图 14: PyTorch 激活函数

实例 3.3 PyTorch 简单的神经网络

操作： 继承 nn.Module 类，定义网络结构和前向传播过程。

```
test.py > ...
1 import torch
2 import torch.nn as nn
3
4 # 定义一个简单的神经网络模型
5 class SimpleNN(nn.Module):
6     def __init__(self):
7         super(SimpleNN, self).__init__()
8         # 定义一个输入层到隐藏层的全连接层
9         self.fc1 = nn.Linear(2, 2) # 输入 2 个特征, 输出 2 个特征
10        # 定义一个隐藏层到输出层的全连接层
11        self.fc2 = nn.Linear(2, 1) # 输入 2 个特征, 输出 1 个预测值
12
13    def forward(self, x):
14        # 前向传播过程
15        x = torch.relu(self.fc1(x)) # 使用 ReLU 激活函数
16        x = self.fc2(x) # 输出层
17        return x
```

问题 输出 调试控制台 终端 窗口

PS C:\Users\吴虹霖\Desktop\Codes\2025小学期codes> & D:/python/python.exe c:/Users/吴虹霖/Desktop/Codes/2025小学期codes/test.py  
SimpleNN(  
 (fc1): Linear(in\_features=2, out\_features=2, bias=True)  
 (fc2): Linear(in\_features=2, out\_features=1, bias=True)  
)  
PS C:\Users\吴虹霖\Desktop\Codes\2025小学期codes>

图 15: PyTorch 简单的神经网络

实例 3.4 PyTorch 损失函数

操作： 使用 nn 模块中的损失函数类，计算预测值与真实值之间的差异。



```
7
8 # 创建目标张量 (真实标签)
9 target = torch.tensor([0.0, 0.5, 1.0, 1.5]) # 对于回归任务
10 target_class = torch.tensor([0, 1, 0, 1]) # 对于分类任务 (类别索引)
11 target_probs = torch.tensor([0.0, 1.0, 0.0, 1.0]) # 对于二分类概率
12
13 # 创建模型输出 (预测值)
14 output = torch.tensor([-0.5, 0.5, 1.5, 2.0]) # 回归预测
15 output_logits = torch.tensor([[1.0, -1.0], [0.5, 0.5], [-1.0, 1.0], [0.0, 2.0]]) # 分
16 output_sigmoid = torch.sigmoid(torch.tensor([-0.5, 1.0, 0.0, 2.0])) # 二分类概率
17
18 # 均方误差损失 (用于回归任务)
19 mse_loss = nn.MSELoss()
20 loss_mse = mse_loss(output, target)
21 print(f"MSE Loss: {loss_mse.item()}")
22
23 # 交叉熵损失 (用于分类任务)
```

图 16: PyTorch 损失函数

实例 3.5 PyTorch 优化器

操作: 使用 optim 模块创建优化器, 设置学习率和模型参数。

```
6 class SimpleModel(nn.Module):
10     def forward(self, x):
11         return self.fc(x)
12
13 # 实例化模型
14 model = SimpleModel()
15
16 # 然后才能使用优化器
17 # 使用 SGD 优化器
18 optimizer_sgd = optim.SGD(model.parameters(), lr=0.01)
19
20 # 使用 Adam 优化器
21 optimizer_adam = optim.Adam(model.parameters(), lr=0.001)
22
23 print("优化器创建成功!")
```

图 17: PyTorch 优化器

实例 3.6 PyTorch 训练循环的基本结构

操作: 设置训练循环, 在每个 epoch 中执行完整的前向和反向传播过程。

```
test.py U x
test.py > ...
23 optimizer = optim.Adam(model.parameters(), lr=learning_rate)
24
25 # 生成训练数据
26 X = torch.randn(10, 2) # 10 个样本，每个样本有 2 个特征
27 Y = torch.randn(10, 1) # 10 个目标标签
28
29 # 训练过程
30 for epoch in range(num_epochs):
31     model.train() # 设置模型为训练模式
32     optimizer.zero_grad() # 清除梯度
33     output = model(X) # 前向传播
34     loss = criterion(output, Y) # 计算损失
35     loss.backward() # 反向传播
36     optimizer.step() # 更新权重
37
38     if (epoch + 1) % 10 == 0: # 每 10 轮输出一损失
39         print(f'Epoch [{epoch + 1}/{num_epochs}], Loss: {loss.item():.4f}')
```

问题 输出 调试控制台 终端 窗口

PS C:\Users\吴虹霖\Desktop\Codes\2025小学期codes> & D:/python/python.exe c:/Users/吴虹霖/Desktop/Codes/2025小学期codes/test.py

Epoch [10/100], Loss: 0.7467  
Epoch [20/100], Loss: 0.6489  
Epoch [30/100], Loss: 0.5820  
Epoch [40/100], Loss: 0.5358  
Epoch [50/100], Loss: 0.5034  
Epoch [60/100], Loss: 0.4803  
Epoch [70/100], Loss: 0.4636  
Epoch [80/100], Loss: 0.4512  
Epoch [90/100], Loss: 0.4418  
Epoch [100/100], Loss: 0.4346  
Final Test Loss: 0.4340

PS C:\Users\吴虹霖\Desktop\Codes\2025小学期codes>

图 18: PyTorch 训练循环的基本结构

实例 3.7 简单的线性关系的数据集

操作： 使用 torch 生成随机数据，并添加线性关系和噪声。

```
test.py U x
test.py > ...
1 import torch
2 import numpy as np
3 import matplotlib.pyplot as plt
4
5 # 随机种子，确保每次运行结果一致
6 torch.manual_seed(42)
7
8 # 生成训练数据
9 X = torch.randn(100, 2) # 100 个样本，每个样本 2 个特征
10 true_w = torch.tensor([2.0, 3.0]) # 假设真实权重
11 true_b = 4.0 # 偏置项
12 Y = X @ true_w + true_b + torch.randn(100) * 0.1 # 加入一些噪声
13
14 # 打印部分数据
15 print(X[:5])
16 print(Y[:5])
```

问题 输出 调试控制台 终端 窗口

PS C:\Users\吴虹霖\Desktop\Codes\2025小学期codes> & D:/python/python.exe c:/Users/吴虹霖/Desktop/Codes/2025小学期codes/test.py

tensor([[ 1.9269, 1.4873],  
 [ 0.9007, -2.1055],  
 [ 0.6784, -1.2345],  
 [-0.0431, -1.6047],  
 [-0.7521, 1.6487]])

tensor([12.4460, -0.4663, 1.7666, -0.9357, 7.4781])

PS C:\Users\吴虹霖\Desktop\Codes\2025小学期codes>

图 19: 简单的线性关系的数据集

实例 3.8 自定义数据集类 MyDataset

操作： 实现 `__len__` 和 `__getitem__`

```
test.py U X
test.py > ...
5 class MyDataset(Dataset):
15     def __getitem__(self, idx):
16         # 按索引返回数据和标签
17         sample = self.data[idx]
18         label = self.labels[idx]
19         return sample, label
20
21 # 生成示例数据
22 data = torch.randn(100, 5) # 100 个样本, 每个样本有 5 个特征
23 labels = torch.randint(0, 2, (100,)) # 100 个标签, 取值为 0 或 1
24
25 # 实例化数据集
26 dataset = MyDataset(data, labels)
27
28 # 测试数据集
29 print("数据集大小:", len(dataset))
30 print("第 0 个样本:", dataset[0])

问题 输出 调试控制台 终端 窗口
PS C:\Users\吴虹霖\Desktop\Codes\2025小学期codes> & D:/python/python.exe c:/Users/吴虹霖/Desktop/Codes/2025小学期codes/test.py
数据集大小: 100
第 0 个样本: (tensor([-0.5372, -1.4751,  0.2092, -0.8100,  0.7753]), tensor(1))
PS C:\Users\吴虹霖\Desktop\Codes\2025小学期codes>
```

图 20: 自定义数据集类 MyDataset

个人心得

通过本次实验，我深入学习了程序性能分析与调试、元编程以及 PyTorch 深度学习框架的使用。在程序性能与调试部分，掌握了使用 line profiler、cProfile 和 snakeviz 等工具分析代码性能的方法，以及使用 shellcheck 和 journalctl 进行脚本检查和系统调试的技巧。这些工具的使用大大提高了代码优化和问题排查的效率。

在元编程部分，我学习了 Python 中强大的元编程能力，包括使用 type() 动态创建类、使用装饰器增强函数功能、以及通过特殊方法实现动态属性访问等。这些技术使得代码更加灵活和可扩展，能够应对更复杂的编程场景。

在 PyTorch 部分，我从张量操作基础开始，逐步学习了神经网络构建、损失函数、优化器以及训练循环的实现。通过创建简单的线性数据集和自定义数据集类，我加深了对深度学习数据流程的理解。这些知识为后续进行更复杂的深度学习项目打下了坚实基础。

实验过程中，我遇到了一些挑战，如性能分析工具的输出解读、元编程概念的理解以及 PyTorch 模型训练中的参数调整等。通过查阅文档、调试代码和反复实验，我逐步解决了这些问题，增强了解决问题的能力。

本次实验不仅巩固了 Python 高级编程技巧，也为我后续从事系统开发和机器学习项目提供了重要支持。未来我将进一步探索性能优化技术、元编程高级应用以及 PyTorch 高级特性，提升开发效率和项目质量。