实验报告 4

姓名: 郭千纯

学号：23020007033

课程：系统开发工具基础

2024 年 9 月 11 日

目录

1 github 链接

2 练习内容

调试及性能分析
元编程演示实验
PyTorch 编程

3 实例及结果

3.1 实例 1

使用 print() 进行调试, 以追踪变量的值和程序执行流程。

```
def calculate_sum(a, b):
    print(f"Calculating sum of {a} and {b}")
    return a + b

result = calculate_sum(5, 10)
print(f"The result is: {result}")
```

```
Calculating sum of 5 and 10
The result is: 15
```

图 1: 实例 1

3.2 实例 2

用 import logging 和 logging.basicConfig(level=logging.DEBUG) 设置日志记录。logging 模块可以更好地控制输出内容的级别（如 DEBUG, INFO, WARNING），并能方便地将日志信息保存到文件中。

```
import logging

logging.basicConfig(level=logging.DEBUG)

def multiply(a, b):
    logging.debug(f"Multiplying {a} by {b}")
    return a * b

result = multiply(3, 4)
logging.info(f"The result is: {result}")
```

```
DEBUG:root:Multiplying 3 by 4
INFO:root:The result is: 12
```

图 2: 实例 2

3.3 实例 3

在代码中插入 import pdb; pdb.set_trace()。逐步执行代码、检查变量值、设置断点等，帮助定位错误。

3.4 实例 4

timeit 可以精确测量小段代码的执行时间，适合比较不同实现的性能。

```python
def factorial(n):
    import pdb; pdb.set_trace()  # 启动调试器
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)

result = factorial(5)
print(f"The result is: {result}")
```

```
> c:\users\lenovo\appdata\local\temp\ipykernel_4480\2129114471.py(3)factorial()

ipdb>
```

图 3: 实例 3

```python
import timeit

execution_time = timeit.timeit('sum(range(100))', number=1000)
print(f"Execution time: {execution_time} seconds")
```

```
Execution time: 0.0011580999999978303 seconds
```

图 4: 实例 4

3.5 实例 5

cProfile 显示函数调用的时间和次数，有助于识别性能瓶颈。

3.6 实例 6

使用 type 函数动态生成类。

3.7 实例 7

定义一个装饰器来修改函数的输入或输出。

```python
import cProfile

def example_function():
    total = 0
    for i in range(10000):
        total += i
    return total

cProfile.run('example_function()')
```

```
         4 function calls in 0.001 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.001    0.001    0.001    0.001 507436236.py:3(example_function)
        1    0.000    0.000    0.001    0.001 <string>:1(<module>)
        1    0.000    0.000    0.001    0.001 {built-in method builtins.exec}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

图 5: 实例 5

```python
MyClass = type('MyClass', (object,), {})
obj = MyClass()
print(f"Created an instance of: {type(obj).__name__}")
```

```
Created an instance of: MyClass
```

图 6: 实例 6

3.8 实例 8

定义一个类装饰器以修改类的行为。

3.9 实例 9

创建一个元类并在类定义中使用它。

3.10 实例 10

使用 eval 执行表达式，或使用 exec 执行多行代码。

```python
def my_decorator(func):
    def wrapper(*args, **kwargs):
        print("Before calling the function")
        result = func(*args, **kwargs)
        print("After calling the function")
        return result
    return wrapper

@my_decorator
def say_hello():
    print("Hello!")

say_hello()
```

```
Before calling the function
Hello!
After calling the function
```

图 7: 实例 7

3.11 实例 11

使用 torch.tensor() 创建张量。

3.12 实例 12

创建张量并进行基本运算。

3.13 实例 13

使用 requires_grad=True 追踪张量的梯度。

```python
def class_decorator(cls):
    cls.new_method = lambda self: print("New method added!")
    return cls

@class_decorator
class MyClass:
    pass

obj = MyClass()
obj.new_method()
```

```
New method added!
```

图 8: 实例 8

```python
class MyMeta(type):
    def __new__(cls, name, bases, attrs):
        attrs['greet'] = lambda self: print("Hello from the meta class!")
        return super().__new__(cls, name, bases, attrs)

class MyClass(metaclass=MyMeta):
    pass

obj = MyClass()
obj.greet()
```

```
Hello from the meta class!
```

图 9: 实例 9

3.14 实例 14

定义一个神经网络类。

3.15 实例 15

将输入传递给模型，获取输出。

```
result = eval('2 + 2')
print(f"Result of eval: {result}")

exec('for i in range(3): print(i)')
```

```
Result of eval: 4
0
1
2
```

图 10: 实例 10

```
import torch

tensor = torch.tensor([[1, 2], [3, 4]])
print(tensor)
```

图 11: 实例 11

3.16 实例 16

使用 setattr 动态设置对象属性。

3.17 实例 17

使用 getattr 动态访问对象属性。

```
a = torch.tensor([1, 2])
b = torch.tensor([3, 4])
c = a + b
print(c)  # 输出: tensor([4, 6])
```

```
tensor([4, 6])
```

图 12: 实例 12

```
x = torch.ones(2, 2, requires_grad=True)
y = x + 2
z = y.mean()
z.backward()  # 计算梯度
print(x.grad)  # 输出: tensor([[0.25, 0.25], [0.25, 0.25]])
```

```
tensor([[0.2500, 0.2500],
        [0.2500, 0.2500]])
```

图 13: 实例 13

3.18 实例 18

使用 ___call___ 使对象可调用。

3.19 实例 19

使用装饰器为函数添加缓存功能。

```python
import torch.nn as nn

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(2, 2)
        self.fc2 = nn.Linear(2, 1)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x

model = Net()
print(model)
```

```
Net(
  (fc1): Linear(in_features=2, out_features=2, bias=True)
  (fc2): Linear(in_features=2, out_features=1, bias=True)
)
```

图 14: 实例 14

```python
input_data = torch.tensor([[1.0, 2.0]])
output = model
```

图 15: 实例 15

3.20 实例 20

使用 classmethod 创建工厂方法。

```python
class Person:
    pass

person = Person()
setattr(person, 'name', 'Alice')
setattr(person, 'age', 30)

print(person.name)  # 输出: Alice
print(person.age)   # 输出: 30
```

```
Alice
30
```

图 16: 实例 16

4 练习感悟

调试和性能分析是编程过程中不可或缺的部分。通过 ‘print()‘、‘logging‘ 和 ‘pdb‘ 等工具，我们可以更好地理解代码的执行过程，快速定位问题。使用 ‘cProfile‘ 和 ‘timeit‘ 进行性能分析，使我们能够识别性能瓶颈，从而优化代码，提升程序效率。元编程为我们提供了强大的工具，允许在运行时动态创建和修改类、函数和属性。通过实例化类、使用装饰器和元类等技术，我们可以实现更灵活和动态的编程风格。这种灵活性可以让我们的代码更加模块化和可重用。通过 PyTorch 的基础学习，我体会到其强大的深度学习功能和友好的接口设计。使用张量运算和自动微分，我们可以轻松构建

```python
class Person:
    def __init__(self, name):
        self.name = name

person = Person('Bob')
attribute = 'name'
print(getattr(person, attribute))  # 输出: Bob
```

```
Bob
```

图 17: 实例 17

和训练神经网络，快速实现各种机器学习模型。PyTorch 的动态计算图特性，使得模型的调试和修改变得更加直观。通过实践不同的实例，我对调试、元编程和 PyTorch 的理解加深了。每一个代码示例不仅是对理论知识的应用，更是对实际问题的解决方案。动手实验能帮助我更好地掌握这些技术。编程是一个不断探索和学习的过程。通过这些练习，我意识到还有许多高级特性和技术等待我去发现。保持好奇心和学习热情，将对我的编程能力和项目实践产生积极的影响。总之，本次练习让我在调试、性能分析、元编程和 PyTorch 的应用上获得了新的视角和深刻的理解。未来我会继续深入研究这些领域，并将所学知识应用到实际项目中，以提高自己的编程水平和解决问题的能力。

```python
class Adder:
    def __init__(self, x):
        self.x = x

    def __call__(self, y):
        return self.x + y

add_five = Adder(5)
print(add_five(10))  # 输出: 15
```

```
15
```

图 18: 实例 18

```python
def cache(func):
    memo = {}
    def wrapper(*args):
        if args not in memo:
            memo[args] = func(*args)
        return memo[args]
    return wrapper

@cache
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

print(fibonacci(10))  # 输出: 55
```

```
55
```

图 19: 实例 19

```python
class Circle:
    def __init__(self, radius):
        self.radius = radius

    @classmethod
    def from_diameter(cls, diameter):
        return cls(diameter / 2)

circle = Circle.from_diameter(10)
print(circle.radius)  # 输出: 5.0
```

```
5.0
```

图 20: 实例 20