实验报告

程传哲

2024 年 9 月 13 日

1 学习成果

1.1 调试与性能分析

1. 打印调试

   - 在python中我们可以使用print输出变量查看，得到信息以便调试内容
   - 同理在c++中可以使用cout或者printf

2. 日志调试

   - 可以将日志写入文件，也可以根据严重等级过滤日志

```
$ python logger.py
# Raw output as with just prints
$ python logger.py log
# Log formatted output
$ python logger.py log ERROR
# Print only ERROR levels and above
$ python logger.py color
# Color formatted output
```

3. 调试器pdb

- 调试器可以允许我们在程序运行过程中交互

```python
import pdb
def bubble_sort(arr):
    n = len(arr)
    for i in range(n-1):
        pdb.set_trace()
        j = i
        for j in range(n-1):
            if arr[j] > arr[j+1]:
                temp = arr[j]
                arr[j] = arr[j+1]

                arr[j+1] = temp
    return arr

print(bubble_sort([4, 2, 1, 8, 7, 6]))
```

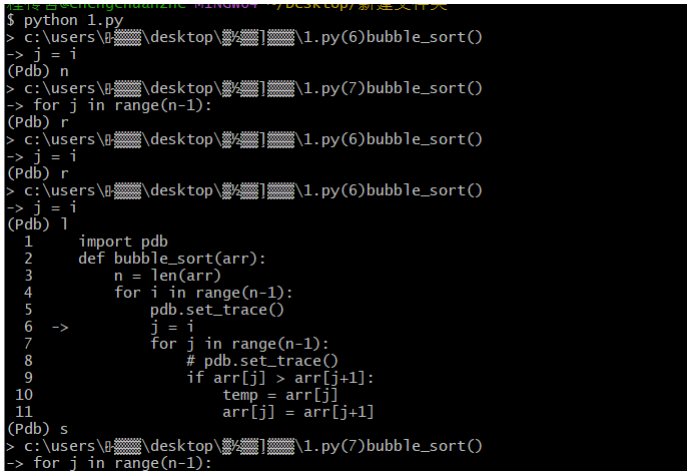

图 1: pdb调试器

-l	显示当前行附近的 11 行或继续执行之前的显示
-s	执行当前行，并在第一个可能的地方停止
-n	继续执行直到当前函数的下一条语句或者 return 语句
-b	设置断点（基于传入的参数）
-p	在当前上下文对表达式求值并打印结果
-r	继续执行直到当前函数返回
-q	退出调试器

4. 静态分析

- 静态分析的优势在于不用执行程序便可以发现代码的问题所在

```python
import time

def foo():
    return 42

for foo in range(5):
    print(foo)
bar = 1
bar *= 0.2
time.sleep(60)
print(baz) # baz is not bar ,it is wrong
```

```
程传哲@chengchuanzhe MINGW64 ~/Desktop/新建文件夹
$ pyflakes 3.py
3.py:6:5: redefinition of unused 'foo' from line 3
3.py:11:7: undefined name 'baz'
```

图 2: 静态分析

5. 计时

记录执行时间

```
import time, random
n = random.randint(1, 10) * 100

start = time.time()

print("Sleeping for {} ms".format(n))
time.sleep(n/1000)

print(time.time() - start)
```

运行结果

```
Sleeping for 800 ms
0.8006622791290283
```

6. 性能分析工具

源代码

```
import sys, re
def grep(pattern, file):
    with open(file, 'r') as f:
        print(file)
        for i, line in enumerate(f.readlines()):
            pattern = re.compile(pattern)
            match = pattern.search(line)
            if match is not None:
                print("{}: {}".format(i, line), end=""
                    )
if __name__ == '__main__':
    times = int(sys.argv[1])
    pattern = sys.argv[2]
    for i in range(times):
        for file in sys.argv[3:]:
            grep(pattern, file)
```

执行语句

```
$ python -m cProfile -s tottime grep.py 1000 '^(import
    |\s*def)[^,]*$' *.py
```

7. 可视化

   - 使用述分析器的时候会输出大量信息，因此使用可视化输出结果能很好的帮助我们调试分析

1.2 元编程

1. makefile构建系统

   - 一个构建系统包括构建目标，相关依赖和规则都需要在makefile中定义

例如：

```
paper.pdf: paper.tex plot-data.png
    pdflatex paper.tex
plot-%.png: %.dat plot.py
    ./plot.py -i $*.dat -o $@
```

2. make使用

简单示例：

```
print("hello world")
```

上面写入一个a.py的文件

makefile:

```
1_make:a.py
    python a.py -i
```

执行：

```
make
```

输出结果

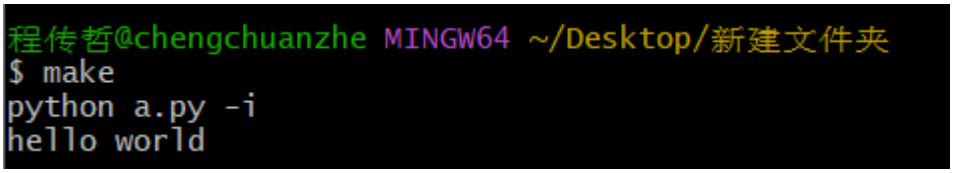

图 3: make

3. 依赖来源

   - 项目的依赖可能是本身也可能是其他项目，现在大多数的依赖可以通过某些软件仓库来获取

4. 依赖API

   - 项目的依赖的项目在每次发布的时候都会创建一个版本号，类似python3.5，python3.7,他的格式是主版本号 .次版本号.补丁号
   - 版本号相关规则：
     - 如果新的版本没有改变 API，请将补丁号递增
     - 如果您添加了 API 并且该改动是向后兼容的，请将次版本号递增
     - 如果您修改了 API 但是它并不向后兼容，请将主版本号递增

1.3 大杂烩

1. Markdown

   - Markdown是一个轻量化的标记语言

Markdown的使用方式较为简单：

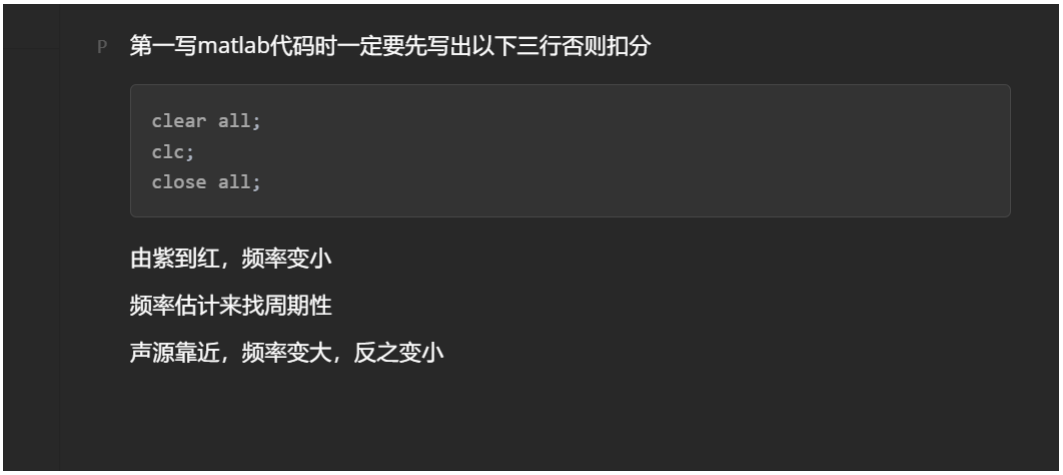

图 4: Markdown

2. 修改键位映射

   - 推荐修改键位是因为键盘上很多不常用的功能却占用了比较方便的按键位置，例如CapsLock
   - 推荐Windows在控制面板中修改

3. API

   - API 通俗易懂的来讲就是封装操作，应用程序编程接口

4. VPN

   - 发出的流量看上去来源于 VPN 供应商的网络而不是你的“真实”地址，而你实际接入的网络只能看到加密的流量

1.4 Pytorch

1. tensor初始化

   - 不同方式初始化tensor

```
import torch
import numpy as np
```

```
3
4  data = [[1, 2],[3, 4]]
5  x_data = torch.tensor(data)
6
7  np_array = np.array(data)
8  x_np = torch.from_numpy(np_array)
9
10 x_ones = torch.ones_like(x_data)
11 print(f"Ones Tensor: \n {x_ones} \n")
12
13 x_rand = torch.rand_like(x_data, dtype=torch.float)
14 print(f"Random Tensor: \n {x_rand} \n")
15
16 shape = (2,3,)
17 rand_tensor = torch.rand(shape)
18 ones_tensor = torch.ones(shape)
19 zeros_tensor = torch.zeros(shape)
20
21 print(f"Random Tensor: \n {rand_tensor} \n")
22 print(f"Ones Tensor: \n {ones_tensor} \n")
23 print(f"Zeros Tensor: \n {zeros_tensor}")
```

[language = python]

2. tensor属性

```
1  tensor = torch.rand(3,4)
2
3  print(f"Shape of tensor: {tensor.shape}")
4  print(f"Datatype of tensor: {tensor.dtype}")
5  print(f"Device tensor is stored on: {tensor.device}")
```

- 三行四列
- float32浮点数

- CPU

tensor 剩下的操作很多使用的情况类似numpy的array数组，算数，线代，矩阵算式等等

3. 加载可视化数据集

```
import torch
from torch.utils.data import Dataset
from torchvision import datasets
from torchvision.transforms import ToTensor
import matplotlib.pyplot as plt


training_data = datasets.FashionMNIST(
    root="data",
    train=True,
    download=True,
    transform=ToTensor()
)

test_data = datasets.FashionMNIST(
    root="data",
    train=False,
    download=True,
    transform=ToTensor()
)

labels_map = {
    0: "T-Shirt",
    1: "Trouser",
    2: "Pullover",
    3: "Dress",
    4: "Coat",
```

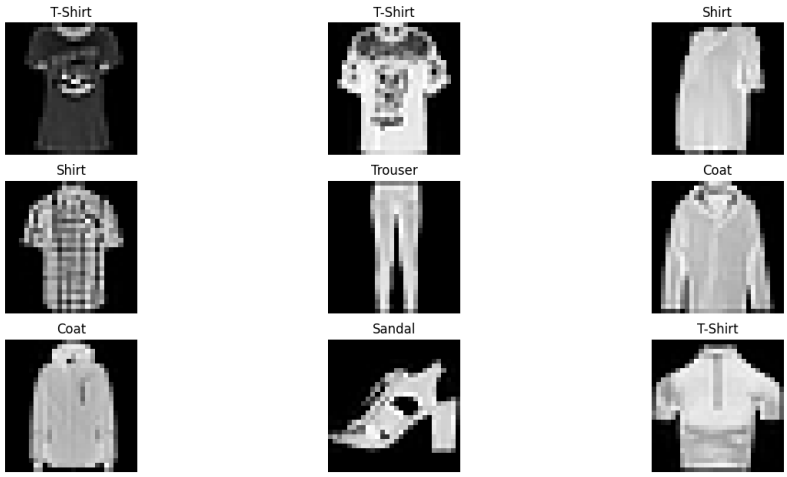

图 5: 数据集

```
    5: "Sandal",
    6: "Shirt",
    7: "Sneaker",
    8: "Bag",
    9: "Ankle Boot",
}
figure = plt.figure(figsize=(8, 8))
cols, rows = 3, 3
for i in range(1, cols * rows + 1):
    sample_idx = torch.randint(len(training_data),
        size=(1,)).item()
    img, label = training_data[sample_idx]
    figure.add_subplot(rows, cols, i)
    plt.title(labels_map[label])
    plt.axis("off")
    plt.imshow(img.squeeze(), cmap="gray")
plt.show()
```

4. 转换 数据并不总是以所需的最终处理形式出现,需要转换

```
import torch
from torchvision import datasets
from torchvision.transforms import ToTensor, Lambda

ds = datasets.FashionMNIST(
    root="data",
    train=True,
    download=True,
    transform=ToTensor(),
    target_transform=Lambda(lambda y: torch.zeros(10,
       dtype=torch.float).scatter_(0, torch.tensor(y),
        value=1))
)
```

5. 自定义运算符

   - 必须通过 Python torch.library 文档或 C++ API 向 PyTorch 注册自定义操作

2 Github链接

https://github.com/Chengchuanzhe/-.git

3 学习感悟

元编程是编程的编程，它允许我们在更高层次上操作代码本身，如修改代码结构、生成代码等。学习元编程让我深刻体会到编程的灵活性和抽象能力的极限。Python中的装饰器、元类以及反射机制是元编程的强有力工具。掌握这些工具后，我发现自己能够更优雅地解决一些复杂的设计问题，比如动态生成类、运行时修改函数行为等。元编程是进阶编程技能的关键一环，值得深入探索。Markdown以其简洁的语法和强大的功能，迅速

成为了我写作和记录的首选工具。它不仅易于学习，而且能够高效地生成结构清晰、格式美观的文档。从笔记整理到项目报告，Markdown都能轻松应对。最让我喜欢的是Markdown与Git、GitHub等版本控制系统的完美结合，使得文档的编写、分享和协作变得更加便捷。调试是编程过程中不可或缺的一环，它教会了我如何系统地定位和解决问题。从最初的盲目试错，到后来的使用调试工具（如Python的pdb）、打印日志、分段执行等策略，我的调试技能有了显著提升。更重要的是，我学会了如何设计易于调试的代码，比如遵循良好的命名规范、编写清晰的注释、使用断言来验证假设等。PyTorch作为一款流行的深度学习框架，以其灵活性和易用性著称。学习PyTorch让我深入理解了深度学习的基本原理，并亲手实践了多个经典模型，如卷积神经网络（CNN）、循环神经网络（RNN）等。PyTorch的动态图特性使得模型的开发和调试变得异常简单，极大地提高了我的开发效率。同时，PyTorch丰富的社区资源也让我在遇到问题时能够快速找到解决方案。