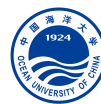


海纳百川
取则行远



中国海洋大学
OCEAN UNIVERSITY OF CHINA

Week4

左昊天

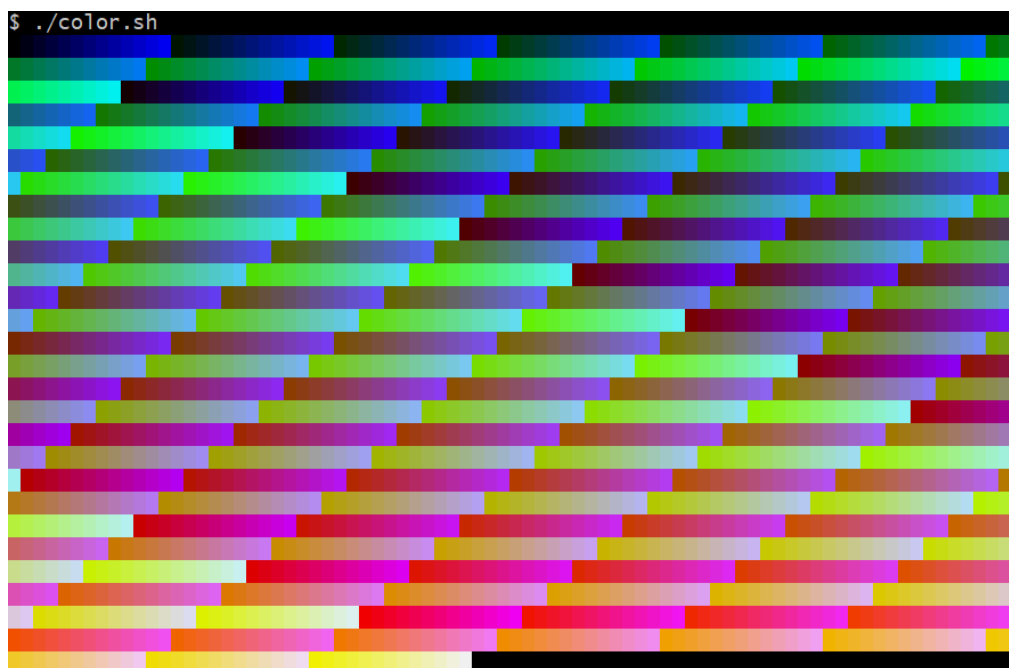
2024-09-13

Github 仓库地址

1 调试与性能分析

1.1 打印颜色

```
1 #!/usr/bin/env bash
2 for R in $(seq 0 20 255); do
3     for G in $(seq 0 20 255); do
4         for B in $(seq 0 20 255); do
5             printf "\e[38;2;${R};${G};${B}m \e[0m";
6         done
7     done
8 done
```



1.2 使用 Linux 上的 `journalctl` 或 macOS 上的 `log show` 命令来获取最近一天中超级用户的登录信息及其所执行的指令。如果找不到相关信息，您可以执行一些无害的命令，例如 `sudo ls` 然后再次查看。

```
eric@eric-virtual-machine:~/Desktop$ journalctl
9月 06 11:57:35 eric-virtual-machine kernel: Linux version 6.8.0-40-generic (bu
9月 06 11:57:35 eric-virtual-machine kernel: Command line: BOOT_IMAGE=/boot/vml
9月 06 11:57:35 eric-virtual-machine kernel: KERNEL supported cpus:
9月 06 11:57:35 eric-virtual-machine kernel: Intel GenuineIntel
9月 06 11:57:35 eric-virtual-machine kernel: AMD AuthenticAMD
9月 06 11:57:35 eric-virtual-machine kernel: Hygon HygonGenuine
9月 06 11:57:35 eric-virtual-machine kernel: Centaur CentaurHauls
9月 06 11:57:35 eric-virtual-machine kernel: zhaoxin Shanghai
9月 06 11:57:35 eric-virtual-machine kernel: BIOS-provided physical RAM map:
9月 06 11:57:35 eric-virtual-machine kernel: BIOS-e820: [mem 0x0000000000000000>
9月 06 11:57:35 eric-virtual-machine kernel: BIOS-e820: [mem 0x0000000000000e80>
9月 06 11:57:35 eric-virtual-machine kernel: BIOS-e820: [mem 0x000000000000dc00>
9月 06 11:57:35 eric-virtual-machine kernel: BIOS-e820: [mem 0x0000000000100000>
9月 06 11:57:35 eric-virtual-machine kernel: BIOS-e820: [mem 0x00000000bfe0000>
9月 06 11:57:35 eric-virtual-machine kernel: BIOS-e820: [mem 0x00000000bfeff000>
9月 06 11:57:35 eric-virtual-machine kernel: BIOS-e820: [mem 0x00000000bff00000>
9月 06 11:57:35 eric-virtual-machine kernel: BIOS-e820: [mem 0x00000000f0000000>
9月 06 11:57:35 eric-virtual-machine kernel: BIOS-e820: [mem 0x00000000fec00000>
9月 06 11:57:35 eric-virtual-machine kernel: BIOS-e820: [mem 0x00000000fee00000>
9月 06 11:57:35 eric-virtual-machine kernel: BIOS-e820: [mem 0x00000000fffe0000>
9月 06 11:57:35 eric-virtual-machine kernel: BIOS-e820: [mem 0x0000000100000000>
```

1.3 pdb 调试器

- l(ist) - 显示当前行附近的 11 行或继续执行之前的显示；
- s(tep) - 执行当前行，并在第一个可能的地方停止；
- n(ext) - 继续执行直到当前函数的下一条语句或者 `return` 语句；
- b(reak) - 设置断点（基于传入的参数）；
- p(rint) - 在当前上下文对表达式求值并打印结果。还有一个命令是 `pp`，它使用 `pprint` 打印；
- r(eturn) - 继续执行直到当前函数返回；
- q(uit) - 退出调试器。

这里使用的是 `ipdb`，它是 `pdb` 的增强版本，相较于 `pdb` 有直观的界面显示。

```
> c:\users\19355\desktop\大二暑假\系统开发工具基础\1\tool_class_2024_sum\week4\bug.py(1)<module>()
----> 1 def bubble_sort(arr):
      2     n = len(arr)
      3     for i in range(n):
ipdb> c
```

`c` 命令代表继续执行，直到出现错误。

```
Traceback (most recent call last):
  File "C:\Users\19355\AppData\Local\Programs\Python\Python310\lib\site-packages\ipdb\__main__.py", line 323, in main
    pdb._runscript(mainpyfile)
  File "C:\Users\19355\AppData\Local\Programs\Python\Python310\lib\pdb.py", line 1592, in _runscript
    self.run(statement)
  File "C:\Users\19355\AppData\Local\Programs\Python\Python310\lib\bdb.py", line 597, in run
    exec(cmd, globals, locals)
  File "<string>", line 1, in <module>
  File "c:\users\19355\desktop\大二暑假\系统开发工具基础\1\tool_class_2024_sum\week4\bug.py", line 10, in <module>
    print(bubble_sort([4, 2, 1, 8, 7, 6]))
  File "c:\users\19355\desktop\大二暑假\系统开发工具基础\1\tool_class_2024_sum\week4\bug.py", line 5, in bubble_sort
    if arr[j] > arr[j+1]:
IndexError: list index out of range
Uncaught exception. Entering post mortem debugging
Running 'cont' or 'step' will restart the program
> c:\users\19355\desktop\大二暑假\系统开发工具基础\1\tool_class_2024_sum\week4\bug.py(5)bubble_sort()
----> 4         for j in range(n):
       5             if arr[j] > arr[j+1]:
       6                 arr[j] = arr[j+1]
```

使用 `p locals()` 命令，可以查看此时所有变量的值，用此进行错误分析。

```
ipdb> p locals()
{'arr': [2, 1, 1, 7, 6, 6], 'n': 6, 'i': 0, 'j': 5}
ipdb>
```

1.4 计时

使用 `time` 库中的 `time` 函数

```
1 import time, random
2 n = random.randint(1, 10) * 100
3
4 # 获取当前时间
5 start = time.time()
6
7 # 执行一些操作
8 print("Sleeping for {} ms".format(n))
9 time.sleep(n/1000)
10
11 # 比较当前时间和起始时间
12 print(time.time() - start)
```

```
Sleeping for 800 ms
0.813072919845581
```

1.5 memory-profiler

使用 `memory-profiler` 模块对内存进行监控。

实例代码：

```
1 @profile
2 def my_func():
3     a = [1] * (10 ** 6)
```

```
4     b = [2] * (2 * 10 ** 7)
5     del b
6     return a
7
8 if __name__ == '__main__':
9     my_func()
```

```
PS C:\Users\19355\Desktop\大二暑假\tools\1\tool_class_2024_sum\week4\python> python -m memory_profiler 内存.py
Filename: 内存.py
Line #    Mem usage    Increment    Occurrences    Line Contents
=====
1      45.453 MiB      45.453 MiB          1      @profile
2
3      53.090 MiB       7.637 MiB          1      def my_func():
4      205.680 MiB    152.590 MiB          1          a = [1] * (10 ** 6)
5      53.090 MiB    -152.590 MiB          1          b = [2] * (2 * 10 ** 7)
6      53.090 MiB       0.000 MiB          1          del b
7      53.090 MiB       0.000 MiB          1          return a
```

1.6 shellcheck

shellcheck 是一个静态的脚本分析工具，它可以得到程序错误的原因。

```
1     shellcheck 文件名
```

```
$ shellcheck test.sh
In test.sh line 1:
echo "Hello World"
^-- SC2148 (error): Tips depend on target shell and yours is unknown. Add a shebang or a 'shell' directive.
```

此代码的问题是开头没有声明使用的 shell 类型。需加上#!/bin/bash

1.7 这里有一些用于计算斐波那契数列 Python 代码，它为计算每个数字都定义了一个函数。将代码拷贝到文件中使其变为一个可执行的程序。首先安装 pycallgraph 和 graphviz(如果您能够执行 dot, 则说明已经安装了 GraphViz.)。并使用 pycallgraph graphviz - ./fib.py 来执行代码并查看 pycallgraph.png 这个文件。fibN 被调用了多少次？我们可以通过记忆法来对其进行优化。将注释掉的部分放开，然后重新生成图片。这回每个 fibN 函数被调用了多少次？

```
1  #!/usr/bin/env python
2  def fib0(): return 0
3
4  def fib1(): return 1
```

```
5
6 s = """def fib{}(): return fib{}() + fib{}()"""
7
8 if __name__ == '__main__':
9
10     for n in range(2, 10):
11         exec(s.format(n, n-1, n-2))
12     # from functools import lru_cache
13     # for n in range(10):
14     #     exec("fib{} = lru_cache(1)(fib{})".format(n, n
15             ))
16     print(eval("fib9()"))
```

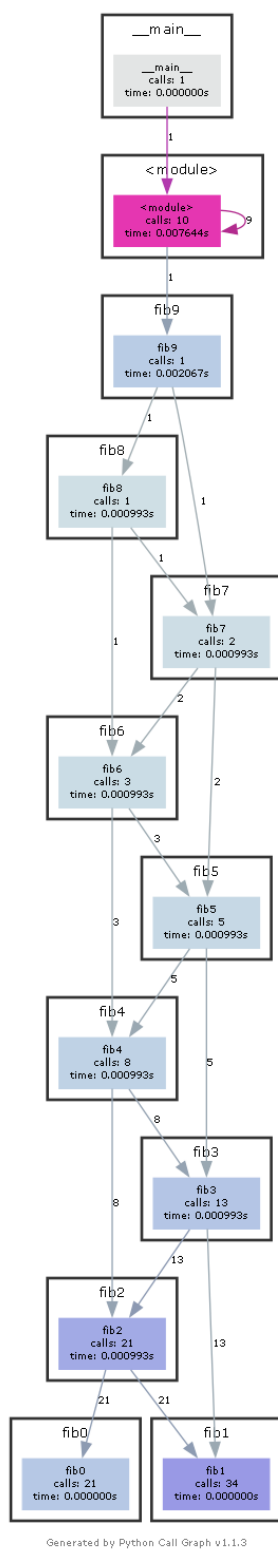


图 1: 放开注释前
被调用 101 次

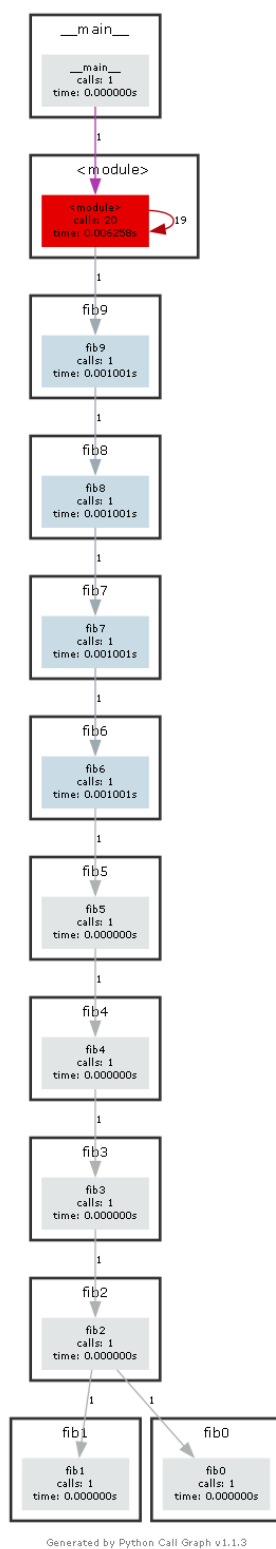


图 2: 放开注释后
被调用 10 次

1.8 遇到的问题：无法安装 pycallgraph

一开始根据提示，尝试回退 setuptools 版本，发现没有用。在网上不断搜索，找到解决方法，安装时用 `pip install pycallgraph2` 命令就可以安装了。

2 元编程

2.1 makefile 的写法

```
1 paper.pdf: paper.tex plot-data.png
2 pdflatex paper.tex
3
4 plot-%.png: %.dat plot.py
5             plot.py -i $.dat -o $@
```

冒号左侧的是构建目标，冒号右侧的是构建它所需的依赖。缩进的部分是构建时用到的命令

2.2 生成图片的规则，和 latex 的写法

2.2.1 python（用于生成图片）

```
1 import matplotlib
2 import matplotlib.pyplot as plt
3 import numpy as np
4 import argparse
5
6 parser = argparse.ArgumentParser()
7 parser.add_argument('-i', type=argparse.FileType('r'))
8 parser.add_argument('-o')
9 args = parser.parse_args()
10
11 data = np.loadtxt(args.i)
12 plt.plot(data[:, 0], data[:, 1])
13 plt.savefig(args.o)
```

2.2.2 latex（将生成的图片放入 latex 中）

```
1 \documentclass{article}
```

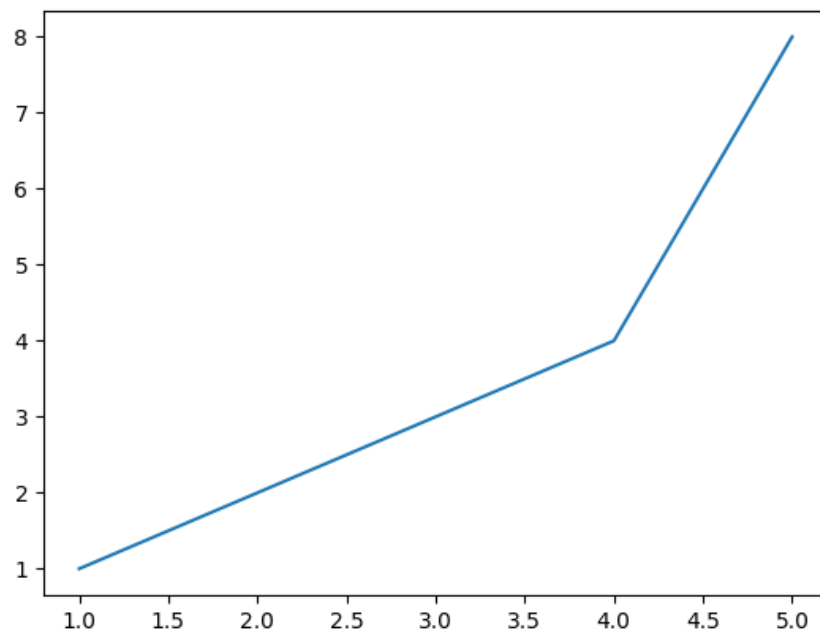
```
2 \usepackage{graphicx}
3 \begin{document}
4 \includegraphics[scale=0.65]{plot-data.png}
5 \end{document}
```

2.2.3 .dat 数据文件

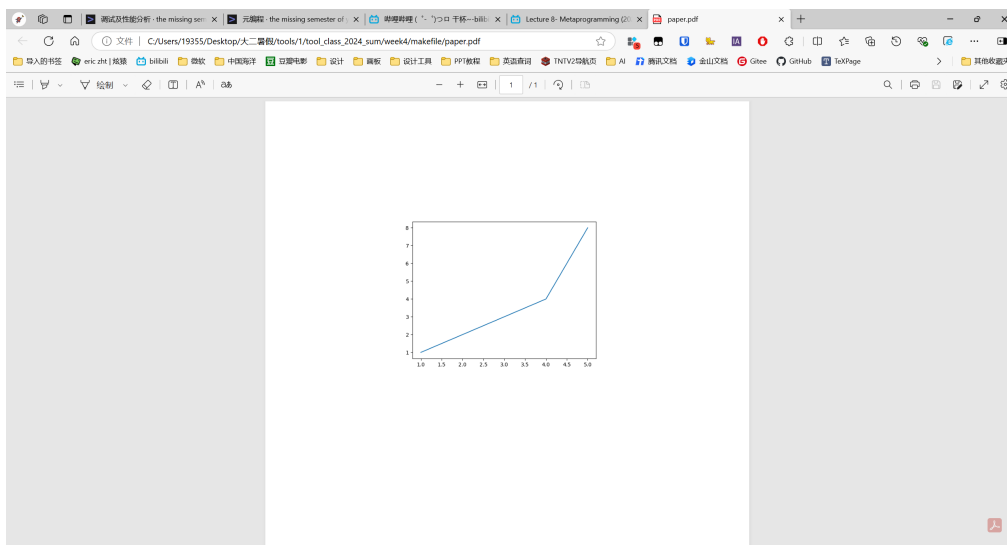
```
1 1 1
2 2 2
3 3 3
4 4 4
5 5 8
```

2.3 make 语句

make 后不加参数将会构建 plot-data.png 和 paper.pdf 文件；但，如果 make 后加参数，比如 make plot-data.png 则只会构建这一个文件。当然，只有在图片文件存在的情况下，才可以单独构建 paper.pdf 文件。生成的图片：



latex 生成的 pdf:



2.4 遇到的问题

一开始一直报错，试了很多办法都没有解决。最开始使用的是 windows shell，所以没有察觉出路径有什么问题后来先去做别的，再返回来做这个时发现，在 git bash 中会出现无法显示的内容，考虑是中文引起的。将所有路径的中文改成英文后可以运行了。同时 makefile 里其实是可以相对路径的，但由于一开始使用的是绝对路径，才导致中文路径造成的影响。经测试，用相对路径时，路径中的中文不影响代码运行。

```
C:\Users\19355\Desktop\大二暑假\tools\1\tool_class_2024_sum\week4\makefile\plot.py -i data.dat -o plot-data.png
Makefile:5: recipe for target 'plot-data.png' failed
process_begin: CreateProcess(NULL, C:\Users\19355\Desktop\大二暑假\tools\1\tool_class_2024_sum\week4\makefile\plot.py -i data.dat -o plot-data.png, ...) failed.
make (e=2): 错误 2 [plot-data.png]
make: *** [plot-data.png] Error 2
```

2.5 大多数的 makefiles 都提供了一个名为 clean 的构建目标，这并不是说我们会生成一个名为 clean 的文件，而是我们可以使用它清理文件，让 make 重新构建。您可以理解为它的作用是“撤销”所有构建步骤。在上面的 makefile 中为 paper.pdf 实现一个 clean 目标。您需要构建 phony。

修改后的代码:

```

1  paper.pdf:  paper.tex  plot-data.png
2              pdflatex paper.tex
3
4  plot-%.png:  %.dat  plot.py

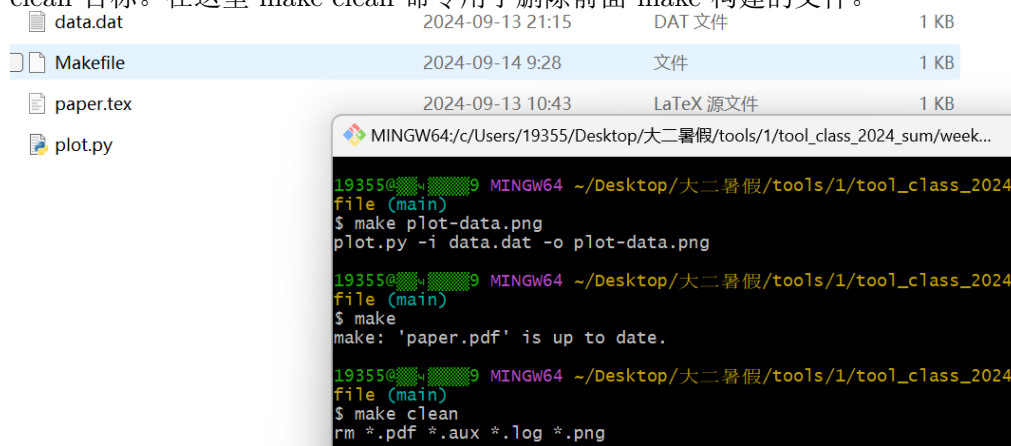
```

```
5      ./plot.py -i $.dat -o $@
6
7  .PHONY: clean
8  clean:
9      rm *.pdf *.aux *.log *.png
```

当文件存在，并且数据没有任何更新时，输入 make 命令并不会重新构建文件。

```
$ make
make: 'paper.pdf' is up to date.
```

这时就体现 .PHONY 的作用了，.PHONY 是用于声明伪目标的关键字。伪目标不是文件。如果目标与实际文件同名，make 可能会认为目标是已经存在的，不需要重新执行。如果存在名为 clean 的文件，make clean 将不会执行 clean 目标，因为 make 认为 clean 已经是一个文件。使用 .PHONY: clean，make 会忽略文件的存在，强制执行 clean 目标。在这里 make clean 命令用于删除前面 make 构建的文件。



2.6 版本号

主版本号. 次版本号. 补丁号

关于版本号的规则：

如果新的版本没有改变 API，递增补丁号；

如果添加了 API 并且该改动是向后兼容的，递增次版本号；

如果修改了 API 但是它并不向后兼容，递增主版本号。

3 pytorch

3.1 引入库

```
1 from __future__ import print_function
```

```
2 import torch
```

注：以下代码均省略引入库的代码。

3.2 随机生成 $a \times b$ 的矩阵

```
1 rand_x = torch.rand(a, b)
2 print(rand_x)
```

```
tensor([[0.6003, 0.8039, 0.1552],
        [0.3492, 0.8825, 0.4327],
        [0.4235, 0.1198, 0.3892],
        [0.0445, 0.6159, 0.7562],
        [0.2491, 0.7200, 0.9900]])
```

(此为 5×3 的矩阵)

3.3 创建数值都是 0, 类型为 long 的矩阵

```
1 zero_x = torch.zeros(a, b, dtype=torch.long)
2 print(zero_x)
```

```
tensor([[0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0]])
```

3.4 保留相同的尺寸大小, 并修改数据类型

```
1 tensor1 = torch.zeros(3, 4, dtype=torch.long)
2 tensor2 = torch.randn_like(tensor1, dtype=torch.float)
3 print('tensor1: ', tensor1)
4 print('tensor2: ', tensor2)
```

```
tensor1: tensor([[0, 0, 0, 0],
                 [0, 0, 0, 0],
                 [0, 0, 0, 0]])
tensor2: tensor([[-0.6876, -0.6814,  0.5686, -0.2144],
                 [-1.5013,  0.3748,  2.0203,  0.2432],
                 [-0.1230,  0.6541,  0.6294, -0.1987]])
```

3.5 修改 tensor 的尺寸

```
1 x = torch.randn(3, 6)
2 y = x.view(18)
3 # -1 表示除给定维度外的其余维度的乘积
4 z = x.view(-1, 9)
5 print(x.size(), y.size(), z.size())
```

解释: z 矩阵的列计算方式:

x 是 4×4 的, 所以总共有 18 个。z 的列乘以行数 (9) 等于 18, 所以 z 的列数为 2。

```
torch.Size([3, 6]) torch.Size([18]) torch.Size([2, 9])
```

3.6 Tensor 的加法

三种方式:

1、+

2、torch.add(tensor1, tensor2)

3、tensor1.add_(tensor2), 该方法会直接修改 tensor1 变量的值

```
tensor1= tensor([[ -1.9233,  0.6145, -0.4592,  0.3885],
 [ 1.0400, -0.0471,  1.8407, -0.8701],
 [-0.5764,  0.2761,  0.5713,  1.2667]])
tensor2= tensor([[ 0.7943,  0.2186,  0.0189,  0.5150],
 [ 0.8489,  0.2613,  0.5728,  0.9321],
 [ 0.7139,  0.9534,  0.1504,  0.1818]])
tensor1 + tensor2= tensor([[ -1.1290,  0.8331, -0.4402,  0.9035],
 [ 1.8889,  0.2141,  2.4136,  0.0619],
 [ 0.1375,  1.2295,  0.7218,  1.4485]])
tensor1 + tensor2= tensor([[ -1.1290,  0.8331, -0.4402,  0.9035],
 [ 1.8889,  0.2141,  2.4136,  0.0619],
 [ 0.1375,  1.2295,  0.7218,  1.4485]])
add result= tensor([[ -1.1290,  0.8331, -0.4402,  0.9035],
 [ 1.8889,  0.2141,  2.4136,  0.0619],
 [ 0.1375,  1.2295,  0.7218,  1.4485]])
tensor1= tensor([[ -1.1290,  0.8331, -0.4402,  0.9035],
 [ 1.8889,  0.2141,  2.4136,  0.0619],
 [ 0.1375,  1.2295,  0.7218,  1.4485]])
tensor2= tensor([[ 0.7943,  0.2186,  0.0189,  0.5150],
 [ 0.8489,  0.2613,  0.5728,  0.9321],
 [ 0.7139,  0.9534,  0.1504,  0.1818]])
```

3.7 Tensor 转换为 Numpy 数组

```
1 a = torch.ones(5)
2 print(a)
3 b = a.numpy()
4 print(b)
```

```
tensor([1., 1., 1., 1., 1., 1.])  
[1. 1. 1. 1. 1. 1.]
```

3.8 Numpy 数组转换为 Tensor

```
1 import numpy as np  
2 a = np.ones(5)  
3 b = torch.from_numpy(a)  
4 np.add(a, 1, out=a)  
5 print(a)  
6 print(b)
```

```
[2. 2. 2. 2. 2.]  
tensor([2., 2., 2., 2., 2.], dtype=torch.float64)
```

3.9 CUDA 张量

```
1 x = torch.randn(4, 4)  
2  
3 # 当 CUDA 可用的时候, 可以使用下方这段代码, 采用 torch.  
   device() 方法来改变 tensors 是否在 GPU 上进行计算操作  
4 if torch.cuda.is_available():  
5     device = torch.device("cuda")           # 定义一个  
       CUDA 设备对象  
6     y = torch.ones_like(x, device=device)   # 创建一个在  
       GPU 上的 tensor  
7     x = x.to(device)                         # 将 x 移动到  
       GPU  
8     z = x + y  
9     print(z)  
10    print(z.to("cpu", torch.double))        # 将 z 移动到  
       CPU 并改变数值类型  
11 else:  
12    print("CUDA is not available")
```

```
tensor([[ 1.1667, -0.2021,  2.3083,  1.7255],
        [ 0.7949,  0.0281, -0.4448, -0.2990],
        [ 2.3527,  0.0554,  1.7378,  2.1257],
        [ 0.5861,  0.4349,  0.6697,  0.3967]], device='cuda:0')
tensor([[ 1.1667, -0.2021,  2.3083,  1.7255],
        [ 0.7949,  0.0281, -0.4448, -0.2990],
        [ 2.3527,  0.0554,  1.7378,  2.1257],
        [ 0.5861,  0.4349,  0.6697,  0.3967]], dtype=torch.float64)
```

第一个结果是在 gpu 上运行的，第二个结果是在 cpu 上运行的。

3.10 遇到的问题：

一直报错，说缺少某 dll 文件，无法运行 pytorch，但可以引入 torch 库。但打开这个文件夹，里面的确是有这个 dll 文件的。在网上寻找解决方案，下载了新的 dll 文件放入文件夹内也依然报错。也考虑过是环境没配置好，重新下载安装了 cuda，配置了 python 的路径等方法都无效。网上的各种方案都试了一下，都没用。最后尝试重装 pytorch，再次运行后发现居然可以了。

可能导致问题的原因：第一次下载的时候看到进度条到100%就关闭了，导致东西虽然是下完了，但是配置还没完成。

4 关于本课的感想

这门课我认为是除了编程课之外最实用的一门课，虽然它没有教我们什么“硬知识”，也只有 0.5 学分，但它让我们知道了 git、latex 的存在，让我们学习了 shell、命令行、vim 的使用，ipdb 的调试等等。老师上课提点一些相关内容，并让我们自己尝试，完成实验、报告的形式也让我们提高了自学能力、独立思考能力。在发现问题并解决问题后，会产生许多的满足感。并且在《程序设计基础实践》这门课中，我们小组已经全流程使用 git 进行协同开发了，在开发中我们也深刻体会到了这些工具的重要性。感谢老师和助教，让 missing semester 不再 missing。

