DA-Python入门与实战文档

作者：李翔宇 丁俊杰 梁远心 张博仕

指导老师：白玉琦

目录

L1-Python下载与配置

这份教程适合 **零基础同学**，帮助你完全从零开始配置 Python 环境，并使用 VS Code 来编写和运行 Python 代码。

教程内容

1. 下载并安装 Python
2. 配置环境变量
3. 安装 VS Code 并配置 Python 插件

4. 在 VS Code 中运行 Python 程序

1. 下载并安装 Python

1. 打开 Python官网
2. 点击 **Download Python 3.x.x** （推荐下载较新的稳定版本）
3. 运行安装包，在安装界面勾选：
    - ✅ `Add Python 3.x to PATH`
    - ✅ `Install Now`

安装完成后，在命令行中输入：

```
python --version
```

如果无报错且能显示版本号，说明安装成功。

2. 配置环境变量（如果安装时忘记勾选 PATH ， 或下载后版本显示存在问题）

1. 打开 **开始菜单** → 输入 `环境变量` → 点击 `编辑系统环境变量`
2. 在弹出的窗口中点击 **环境变量**
3. 在 **系统变量** 中找到 `Path` → 点击编辑
4. 新建一条记录，填入 Python 的安装路径，例如：
    - `C:\\Users\\你的用户名\\AppData\\Local\\Programs\\Python\\Python312`
5. 保存后重新打开命令行，再次输入：

```
python --version
```

3. 安装 VS Code 并配置 Python 插件

1. 打开 VS Code官网
2. 下载并安装 **Windows 用户选择 User Installer**
3. 打开 VS Code 后，点击左侧 **扩展（Extensions）**
4. 搜索并安装插件：
    - `Python`
    - `Jupyter`

这样就能在 VS Code 中编写和运行 Python 代码

4. 在 VS Code 中运行 Python 程序

1. 打开 VS Code，新建一个文件，保存为 `trail.py`

2. 输入以下代码：

```
print("Hello, Python!")
```

3. 点击右上角的 ▶️（运行按钮），或在终端输入：

```
python trail.py
```

你就能看到输出：

```
Hello, Python!
```

恭喜！你已经成功运行第一个 Python 程序！

```
# 在 Jupyter Notebook 中同样可以运行 Python 代码
print("Hello, Python in Jupyter!")
```

L2-Python语法基础

预备知识

- Python 中的语句可以直接执行
- 用 `#` 表示从此开始到行尾都是注释
- 段落注释采用''' (三引号中间的是注释掉的部分) '''
- 变量赋值不用事先声明、不用写类型
    - 数组都是 `object*[]` 类型，所以可以混着存任意的对象
- 用 `type(x)` 函数获取对象类型，用 `dir(x)` 函数获取对象的方法，用 `id(x)` 函数获取对象的 ID，（其实是对象的地址）

```
例子：
a = 43  # 在内存中申请一个空间存放 43，再用 a 指向它，id(a) 为 6
b = a   # 将 b 指向 a 指向的东西，id(b) 为 6
```

- `print(x)` 函数用于输出指定内容，例如 `print(123)`，也可以传入多个值，如 `print(1, 2, 3)`

1. 常见数据类型

类型	名称	例子
int	整数	`-100`
complex	复数	`1 + 2j`
bool	布尔型	只有 `True` 和 `False`
float	浮点数	`3.1416`
str	字符串	`'hello', "abc"`
list	列表	`[1, 0.3, 'hello']`
tuple	元组	`('hello', 10)`
set	集合	`{1, 2, 3}`
dict	字典	`{'dogs': 5, 'pigs': 3}`
NoneType		只有 `None`

注意：

- int 没有大小限制，可以做任意大的整数运算
- float 是 64 位浮点数，相当于 C/C++ 中的 double 类型
- bool 类型只有首字母大写的 `True` 和 `False`
- NoneType 是特殊的类型，它只有 `None` 这一个值/对象实例，通常被用来表示空值

1.1 数值运算

```
# 基本算术运算
1 + 2         # 3
1 - 2         # -1
2 * 3         # 6
4 / 2         # 2.0 注意：无论结果在数学上是否是整数，一定是 float 类型
5 // 2        # 2 向下取整除法
5 % 2         # 1 取余
2**10         # 1024 幂次

# 原地运算
a = 1
a += 2        # a 变为 3

# 逻辑运算
True and False     # False
True or False      # True
not True           # False

# 数学函数
abs(-3)            # 3 绝对值
```

```
min(3, 4)          # 3
max(3, 4)          # 4
int(-3.9)          # -3 向 0 取整
round(-3.9)        # -4 就近取整

# 科学计数法、进制表示
1e-6, 0xFF, 0o67, 0b1110    # (1e-06, 255, 55, 14)
```

1.2 比较操作

```
1 == 2          # False 等于
1 != 2          # True 不等于
1 < 2           # True 小于
1 <= 2          # True 小于等于
1 > 2           # False 大于
1 >= 2          # False 大于等于
1 == 1.0        # True
[1, 2, 3] == [1.0, 2.0, 3.0]  # True
```

`==` 和 `!=` 是基于值做判断的；`is` 是基于 `id` 判断的，即判断两个东西是不是同一个对象

```
a = 1
b = 1.0
a == b          # True
a is b          # False

a = [1, 2, 3]
b = [1, 2, 3]
a == b          # True
a is b          # False
```

注意："不是" 一般写作 `a is not b`

注意：关于 `None` 的判断用 `is` 和 `is not` 而不是 `==` 和 `!=`，理由是 `None` 这个对象是唯一的，我们只需要根据 `id` 判等，而不是根据值判等

```
a = None
b = 1
c = None
a is b          # False
a is c          # True
```

python 中的比较操作可以串着写

```
1 < 2 < 3       # True, 等价于 1 < 2 and 2 < 3

# 可以串任意比较操作、任意多个
1 < 2 != 3 < 4 > 0 == 0  # True
# 等价于 1 < 2 and 2 != 3 and 3 < 4 and 4 > 0 and 0 == 0
```

1.3 字符串

可以写成单引号或者双引号：`'abc'`、`"abc"`

特殊字符需要转义操作`\`：`'123\n456'`

单引号字符串中的单引号、双引号字符串中的双引号，也需要转义操作：`'123\'456'`

但单引号字符串中的双引号、双引号字符串中的单引号不需要转义操作，所以一般来说写成下面这样：`"123'456"`

使用 3 个单/双引号的字符串可以跨行：

```
'''第一行
第二行
第三行'''
```

字符串操作：**(注意：字符串是不可变的，任何操作都会返回新的字符串对象)**

- 拼接：`'ab' + 'cd' -> 'abcd'`
- 重复：`'ab' * 3 -> 'ababab'`
- 分割：`a.split(b)`，以字符串 b 为界分割 a

```
"1 2 3".split(" ")       # ['1', '2', '3']
```

- 连接：`a.join(b)`，以字符串 a 来将字符串序列 b 中元素连接起来

```
" ".join(["1", "2", "3"])    # '1 2 3'
```

- 替换：`a.replace(b,c)`，将字符串 a 中的 b 都替换为 c

```
"1,2,3".replace(",", "@")    # '1@2@3'
```

- 大小写转化：upper()、lower()

```
"abcd123ABCD".upper()        # 'ABCD123ABCD'
"abcd123ABCD".lower()        # 'abcd123abcd'
```

字符串转换：

- `str(a)`: 将 a 转化为字符串
- `hex(a)`,`oct(a)`,`bin(a)`: 将整数 a 转化为 16、8、2 进制字符串
- `int(str,b)`: 将字符串 str 转化为 b 进制整数
- `float(str)`: 将字符串 str 转化为浮点数

```
str(3.4), hex(234), int("FA", 16), float("5.78")
# ('3.4', '0xea', 250, 5.78)
```

Raw 字符串：在引号前面加 `r`，表示不将 `\` 视为转义。

```
r"C:\abc\def"    # 'C:\\abc\\def'
```

模板字符串：在前面加 `f`，在需要用到变量值时非常方便，只需要用 `{}` 括起来

```
a = 1
b = 2
f"a 的值是 {a}, b 的值是 {b}"    # 'a 的值是 1, b 的值是 2'

# 除了变量名，也可以是其他表达式
f"{2**10 + a}"    # '1025'
```

可以用类似 C 中 printf 的语法对浮点数格式化：

```
a = 1.2345678
f"a ≈ {a:.3f}"    # 'a ≈ 1.235' 3位小数
```

如果需要用到单纯的 `{` 或者 `}` 字符，需要写两遍表示转义：

```
a = "b"
f"{{ a }} vs {a}"    # '{ a } vs b'
```

1.4 索引和切片

索引：
对于一个有序序列，可以通过索引的方法来访问对应位置的值。字符串便是一个有序序列的例子，使用[]来对有序序列进行索引。
索引是从 0 开始的，索引 0 对应与序列的第 1 个元素，当索引值大于最大值时，就会报错。
python 中还有负索引值，其为从后向前计数。

```
str = "abcdefghijk"
print(str[8], str[-2])    # i j
```

切片：
可以从序列中取出想要的子序列，其用法为：
`var[lower:upper:step]`
左闭右开，即包括 `lower`，不包括 `upper`； `step` 表示取值的步长。

这三个参数可以省略一部分，例如

- `a[:]`：从头到尾，即创建副本
- `a[1:]`：从 1 到结尾
- `a[:3]`、`a[:-2]`：从 0 到 3、从 0 到倒数第二
- `a[::-1]`：逆序

```
str = "abcdefghijk"
print(str[0:4], str[0:7:2])    # abcd aceg
```

1.5 列表

列表在Python中非常有用，列表是一个有序的序列。列表用一对 `[]` 生成，中间的元素用 `,` 隔开，其中的元素不需要是同一类型，同时列表的长度也不固定。也可以利用 `list()` 或 `[]` 来生成空的列表

```
a = [1, 2.0, "abc"]
a, type(a), list()    # ([1, 2.0, 'abc'], list, [])
```

列表的操作同字符串类似，如下：

- `len()`：输出列表的长度
- `+`：拼接
- `*`：重复

```
a = [1, 2]
b = [4, 5]
a + b, len(a + b), a * 3    # ([1, 2, 4, 5], 4, [1, 2, 1, 2, 1, 2])
```

数组的索引和切片可以**读取**也可以**赋值**

```
a = [1, 2, 3, 4, 5, 6, 7, 8, 9]
a[0], a[7], a[3:6], a[::3]    # (1, 8, [4, 5, 6], [1, 4, 7])

# 切片赋值
a = [1, 2, 3, 4, 5, 6, 7, 8, 9]
a[5:7] = ["a", "b", "c", "d"]
print(a)    # [1, 2, 3, 4, 5, 'a', 'b', 'c', 'd', 8, 9]
a = [1, 2, 3, 4, 5, 6, 7, 8, 9]
a[5:7] = []
print(a)    # [1, 2, 3, 4, 5, 8, 9]
```

还可以用 `del` 删除

```
a = [1, 2, 3, 4, 5, 6, 7, 8, 9]
del a[0:3]
print(a)    # [4, 5, 6, 7, 8, 9]
a = [1, 2, 3, 4, 5, 6, 7, 8, 9]
del a[::3]
print(a)    # [2, 3, 5, 6, 8, 9]
```

- 测试从属关系：`in`
- 计数和索引：`l.count(a)` 返回列表 l 中 a 的个数；`l.index(a)` 返回列表 l 中 a 第一次出现的索引位置，不存在会报错

- 添加，插入和移除：
  - `l.append(a)` 在列表最后加入 a
  - `l.extend(a)` 如果 a 是一个序列，相当于将序列中的每个元素依次 `append`
  - `l.insert(idx,a)` 在列表的 idx 索引处插入 a
  - `l.remove(a)` 移除第一次出现的 a，不存在会报错
- 排序和反向: `a.sort()` 对 a 排序，但是改变 a 中的值；`sorted()` 返回排序的索引，不改变原有值；`reverse()` 列表反向

1.6 元组

- 与列表相似，元组也是个有序序列，可以索引和切片，但是其是不可变的
- 空元组：`()` 或 `tuple()`
- 注意：单元素元组应该写为 `(1,)`，以避免和普通括号表达式产生歧义

```
t = (1, 3, 4, 6, 7, 56, 83, 3)
print(t[0], t[2:5], t[::2])    # 1 (4, 6, 7) (1, 4, 7, 83)

type((12,)), type((12))        # (tuple, int)
```

一般的，元组可以用来作为函数的多返回值，同时也可以用来对多变量进行赋值：

```
w, x, y, z = 1, 2, 3, 4
x = 1, 2, 3, 4
print(x, type(x))              # (1, 2, 3, 4) <class 'tuple'>
```

1.7 字典

字典，是一种由键值对组成的数据结构，常用于处理一对多的函数映射关系。

基本操作

创建字典: 可以通过 `{}` 或者 `dict()` 来创建空字典，同样也可以在创建时使用 `key: value` 这样的结构来初始化。

```
a = {}
b = dict()
c = {"1": "a1", "2": "a2"}
print(type(a), type(b), c["1"])    # <class 'dict'> <class 'dict'> a1
b[1] = "b1"
b["2"] = "b2"
print(b)                           # {1: 'b1', '2': 'b2'}
```

注意：

- 字典的键必须是不可变的类型，比如整数，字符串，元组等，而值可以是任意的 python 对象
- 一般不使用浮点数来作为键，因为其存在存储精度问题

字典方法

- 取值: `d.get(key)`，相比于使用 `d[key]`，其在字典中键不存在是不会报错，而是返回 `None`。
- 删除元素: `d.pop(key)`，删除并返回键为 `key` 的键值对，如果没有返回 `None`。同样可以使用 `del d[key]` 来进行删除。
- 更新字典: `d.update(d')`,将字典 `d'` 中元素更新到 `d` 中。
- 查询字典: `a in d`, 查询键 `a` 是否在字典 `d` 中。

注意："不在" 一般写作 `a not in d` 而不是 `not a in d`

- keys 方法：`d.keys()`, 返回所有键构成的序列。
- values 方法：`d.values()`, 返回所有值构成的序列。
- items 方法：`d.items()`, 返回所有键值对元组构成的序列。

1.8 集合

集合 set 是一种无序的序列，故当其中有两个相同的元素，只会保留一个；并且为了保证其中不包含相同元素，放入集合中元素只能是确定性对象。
集合的生成可以通过 `{}` 或者 `set()` 进行创建，但是在创建空集合，只能通过 `set()` 创建，因为 `{}` 表示空字典。

```
a = set()
b = {}
c = set([1, 2, 3, 2])  # 自动去除相同的元素
d = {1, 2, 3}
print(type(a), type(b), c, d)    # <class 'set'> <class 'dict'> {1, 2, 3} {1, 2, 3}
```

集合操作：

- 并: `a.union(b)` 或者 `a | b`
- 交: `a.intersection(b)` 或者 `a & b`
- 差: `a.difference(b)` 或者 `a - b`；在 `a` 中不在 `b` 中的元素
- 对称差: `a.symmetric_difference(b)` 或者 `a ^ b`；在 `a` 或 `b` 中，但是不同时在 `a`， `b` 中的元素

2. 流程控制

2.1 if

- 基本结构是 `if <条件>:`
- 不像 C 语言一样需要 `{}`，而是用缩进来区分层级
- `elif` 是 else if 的含义
- False，None，0，空字符串，空列表，空字典，空集合，都会被当做 `False`

```
a = 1
if a < 0:
    if a < -1:
        print("a < -1")
    elif a == -1:
        print("a = -1")
    else:
        print("-1 < a < 0")
else:
    print("a >= 0")    # 输出：a >= 0
```

```
# 因为缩进是必须的，而缩进后的内容又不能为空，所以我们通常写个 pass，它不会做任何事
# 在编写代码时，想要暂时先不写某个分支，可以用 pass 临时占位
if True:
    pass
```

2.2 while

- 基本格式是 `while <条件>:`
- 可以用 `break` 跳出，用 `continue` 进入下一次循环

```
i = 0
total = 0
while i < 1000000:
    total += i
    i += 1
print(total)    # 499999500000
```

2.3 for 和 range / zip / enumerate

- 基本格式是`for <变量> in <可迭代对象>`，可迭代对象如列表、字符串、`range`、打开的文件等
- range 用于创建一串等差数列
    - `range(3)` -> 0, 1, 2
    - `range(1, 5)` -> 1, 2, 3, 4
    - `range(1, 10, 2)` -> 1, 3, 5, 7, 9，从 1 开始到 10，步长为 2
- 同样可以用 `break` 和 `continue`
- 可选：可以接 `else`，当循环正常退出而非 `break` 退出时会执行

```
a = [1, 2, 3]
for i in a:
    if i == 1:
        print("a 里有 1")
        break
else:
    print("a 里没有 1")    # 不会执行，因为break了
```

python 中的一个编程习惯是用 `_` 变量表示不使用的值，比如我们想循环 3 次但不需要循环变量：

```
for _ in range(3):
    print(233)
```

列表推导式

比 for-append 更快地创建列表

```
a = []
for i in range(10):
    a.append(i * i)

# 等价于
a = [i * i for i in range(10)]
```

可以有 `if`

```
a = []
for i in range(10):
    if i % 2 == 0:
        a.append(i * i)

# 等价于
a = [i * i for i in range(10) if i % 2 == 0]
```

可以有多重循环

```
a = []
for i in range(10):
    for j in range(i, 10):
        a.append(i + j)

# 等价于
a = [i + j for i in range(10) for j in range(i, 10)]
```

zip

用于同时迭代多个可迭代对象，迭代次数以最短的那个为准

```
for x, y, z in zip("1234567", "abcdefg", "ABCDE"):
    print(x, y, z)
# 输出:
# 1 a A
# 2 b B
# 3 c C
# 4 d D
# 5 e E
```

enumerate

用于同时迭代下标和值

```
a = "abcdefg"
for i, j in enumerate(a):
    print(i, j)
# 输出:
# 0 a
# 1 b
# 2 c
# 3 d
# 4 e
# 5 f
# 6 g
```

3. 函数

- 格式：`def 函数名(参数列表):`
- 参数列表直接写变量名称，同样不用写类型
- 用 `=` 定义关键字参数，类似 C++ 中的默认参数
- 用 `return` 返回返回值，可以是任意类型

```
# 什么都不做的函数
def f():
    pass

# 加法
def add(x, y):
    a = x + y
    return a

add(1, 2)    # 3
```

python 不会检查传入的参数类型，这样既有灵活性，又有隐患：

```
add("abc", "def")    # 'abcdef'
add("abc", 123)      # 报错：TypeError
```

两种传参模式：

- 位置模式：按照位置传入参数
- 关键字模式：使用 `a=123` 这样显式的指定参数名

可以混合这两种模式，但是位置模式必须在关键字模式之前。

```
print(add(x=2, y=3))    # 5
print(add(y="foo", x="bar"))    # barfoo
print(add(2, y=3))      # 5
```

设定参数默认值

```
def quad(x, a=1, b=0, c=0):
    return a * x**2 + b * x + c

print(quad(2.0))        # 4.0
print(quad(2.0, b=3))   # 10.0
```

接收不定长参数

`*args` 表示将多余的位置参数作为元组传给 `args`

```
def add(x, *args):
    total = x
    for arg in args:
        total += arg
    return total

print(add(1, 2, 3, 4))  # 10
print(add(1, 2))        # 3
```

`**kwargs` 表示将多余的关键字参数作为字典传给 `kwargs`

```
def add(x, **kwargs):
    total = x
    for arg, value in kwargs.items():
        print("adding ", arg)
        total += value
    return total

print(add(10, y=11, z=12, w=13))  # 46
```

返回多个值

```
def f():
    return 1, 2, 3

a, b, c = f()
print(a, b, c)    # 1 2 3
```

4. 输入输出

用 `input` 函数获取输入：

```
a = input("请输入：")  # 在 vs code 里运行时会在上方弹出一个输入框
```

输出用 print：

```
print(123)
a = 123
print("a 的值是：", a)    # a 的值是： 123
```

更复杂的输出可以利用模板字符串：

```
a = 3
b = 3.1415926535
print(f"a 的值是 {a:03d}, b 的值是 {b:.2f}")    # a 的值是 003, b 的值是 3.14
```

5. 文件操作

```
# 写文本文件
with open("123.txt", "w", encoding="utf8") as f:
    f.write("这是一行\n")
    f.write("这是另一行\n")

with open("123.txt", "a", encoding="utf8") as f:
    f.write("再给你加一行\n")

# 读取文本文件
with open("123.txt", "r", encoding="utf8") as f:
    for i in f:  # 迭代每一行
        print(i)

with open("123.txt", "r", encoding="utf8") as f:
    a = f.read()  # 返回一个字符串

with open("123.txt", "r", encoding="utf8") as f:
    a = f.readlines()  # 返回每一行的字符串的列表
```

6. 包

类似 C/C++ 中的 `#include` 语句，我们可以用 `import` 导入我们希望用的包

```
import math
math.sin(math.pi / 4)    # 0.7071067811865475

# 只导入部分函数
from math import sin
sin(1)    # 0.8414709848078965

# 导入多个函数
from math import sin, cos
```

```
cos(1)    # 0.5403023058681398

# 给导入的函数换个名字
from math import sin as a_ba_a_ba
a_ba_a_ba(1)    # 0.8414709848078965

# 导入包中的全部内容（不推荐）
from math import *
sin(pi / 2)    # 1.0
```

7. 类

基本格式如下

```
class A:
    def __init__(self):
        self.a = 1

    def f(self, x):
        self.b = 1
        return self.a + x
```

- python 中以 `__` 开头和结尾的是特殊的函数或变量
- 这里 `__init__` 函数会在对象创建时执行，可以理解为构造函数
- 不以 `_` 或 `__` 开头的函数都是 `public` 的
- 非静态类成员函数在被调用时会将类对象作为第一个参数传入，一般约定用 `self` 作为参数名
- 在任何函数中，都可以通过 `self.xxx = xxx` 的方式给对象添加新的属性

如果需要继承，基本格式如下

```
class B(A):
    def __init__(self):
        super().__init__()
        ...
```

这里 `super()` 会返回它所继承的父类（即 `A`），`super().__init__()` 表示调用父类的 `__init__` 函数。

8. 异常处理

基本格式如下

```
try:
    ...
except ...:
    ...
else:  # 可省略
    ...
finally:  # 可省略
    ...
```

- `try` 后面接可能会出错的代码
- `except` 后面接要捕获的错误类型，不接等价于 `except Exception`，能捕获大部分错误类型；里面写出错后要执行的代码
- `else` 后面是不出错时执行的代码
- `finally` 后面是无论出不出错都会执行的代码
- 用 `raise` 抛出一个指定类型的错误
- 用 `assert` 在断言为假时抛出一个 `AssertionError` 类型的错误

L3-数据处理及可视化

1. pandas库是什么?

Python的pandas库是数据分析领域不可或缺的工具，它提供了高效的数据结构和数据分析功能，使得处理和操作数据变得简单易行。

2. pandas库的安装与导入

在终端输入如下指令进行安装（如果是vscode按下 `ctrl` + `~` 即可打开终端）：

```
pip install pandas
```

在`.py`文件开头导入：

```
import pandas as pd
```

3. pandas库中的数据结构

3.1 Series：类似一维数组，由一组数据和相应的索引组成

每个元素有一个唯一的索引（称为index），默认从0开始编号。

```
import pandas as pd

# 创建Series
s= pd.Series(['a','b','c','d','e'])  #index=[0,1,2,3,4]
print(s)
```

也可以为每个元素重新设定唯一的索引（index）：

```
import pandas as pd

# 创建Series
s= pd.Series(['a','b','c','d','e'], index=[1,2,3,4,5])
print(s)
```

3.2 DataFrame：类似二维表格，由多组数据和相应的索引值组成（由多个Series组成）

Series越多，列越多
Series中的数据越多，行越多

```
import pandas as pd

# 创建DataFrame
df = pd.DataFrame({
    'name':['Zhangsan','Lisi','Wangwu'],
    'age':[18,26,35],
    'occupation':['student','teacher','doctor']
})
print (df)
```

4. 查看与操作数据

4.1 查看数据

示例数据集：

```python
import pandas as pd

data = {
    "Name": ["Alice", "Bob", "Charlie", "David", "Eve", "Frank"],
    "Age": [20, 21, 19, 22, 20, 21],
    "Score": [85, 90, 78, 92, 88, 84],
    "Grade": ["A", "A", "B", "A", "B", "B"],
    "Passed": [True, True, True, True, True, False]
}

df = pd.DataFrame(data)
print("Original DataFrame:")
print(df)
```

1. `df.head()` —— 查看前几行数据

默认显示前 5 行，可以传入参数 `n` 来指定行数。

```python
print("\nFirst 3 rows:")
print(df.head(3))  # 只显示前 3 行
```

2. `df.tail()` —— 查看后几行数据

默认显示后 5 行，可以传入参数 `n` 来指定行数。

```python
print("\nLast 2 rows:")
print(df.tail(2))  # 只显示最后 2 行
```

3. `df.info()` —— 显示 DataFrame 的基本信息

- **列名**（Columns）
- **数据类型**（Dtype）
- **非空值数量**（Non-Null Count）
- **内存占用**（Memory Usage）

```python
print("\nDataFrame Info:")
print(df.info())
```

4. `df.describe()` —— 显示数值列的统计信息

- **计数**（count）
- **平均值**（mean）
- **标准差**（std）
- **最小值**（min）
- **25%、50%、75% 分位数**（percentiles）

- **最大值**（max）

```
print("\nDescriptive Statistics:")
print(df.describe())
```

4.2 选择数据

1. 选择单列数据

使用方括号 `[]` 选择单列：

```
print("\n选择 name 列:")
print(df['name'])
```

2. 选择多列数据

使用双括号 `[[]]` 选择多列：

```
print("\n选择 age 和 occupation 列:")
print(df[['age', 'occupation']])
```

3. 选择单行数据

使用 `iloc` 按索引位置选择行：

```
print("\n选择第 1 行数据:")
print(df.iloc[0])
```

4. 选择多行数据

使用 `iloc` 切片选择多行：

```
print("\n选择第 2-3 行数据:")
print(df.iloc[1:3])
```

5. 根据条件选择数据

使用布尔条件筛选数据：

```
print("\n选择年龄大于 30 的人员:")
print(df[df['age'] > 30])
```

4.3 添加与删除与更改数据

1. 添加列数据

直接对新列赋值即可添加列：

```
# 添加gender列
df['gender'] = ['male', 'female', 'male']
print("\n添加gender列后的数据集:")
print(df)
```

2. 添加行数据

使用`loc`索引器添加新行：

```
# 添加第4行数据
df.loc[3] = ['Xiaoqi', 20, 'student', 'female']
print("\n添加行后的数据集:")
print(df)
```

3. 删除列数据

使用`drop()`方法删除列：

```
# 删除gender列
df = df.drop('gender', axis=1)
print("\n删除gender列后的数据集:")
print(df)
```

4. 删除行数据

通过索引删除行：

```
# 删除索引为3的行
df = df.drop(3)
print("\n删除行后的数据集:")
print(df)
```

5. 修改数据

```
# 修改单个值
df.loc[1, 'age'] = 36
print("\n修改年龄后的数据集:")
print(df)

# 整列替换
df['occupation'] = ['undergraduate', 'professor', 'surgeon']
print("\n替换职业列后的数据集:")
print(df)
```

4.4 数据清洗

1. 处理重复数据

使用`drop_duplicates()`删除完全重复的行：

```
# 删除重复行（保留第一条）
df_clean = df.drop_duplicates()
print("\n去重后的数据集:")
print(df_clean)
```

2. 处理缺失值

```
# 删除含有NaN的行
df_dropna = df.dropna()
print("\n删除缺失值后的数据集:")
print(df_dropna)

# 用0填充NaN
df_fillna = df.fillna(0)
print("\n填充缺失值后的数据集:")
print(df_fillna)
```

3. 处理异常值

修正超出合理范围的分数（0-100）：

```
# 修正异常分数
score_cols = ['math', 'chinese', 'english']
df[score_cols] = df[score_cols].clip(0, 100)
print("\n修正异常值后的数据集:")
print(df)
```

4. 计算统计量

添加平均分和中位数列：

```
# 计算统计量
df['mean'] = df[['math', 'chinese', 'english']].mean(axis=1)
df['median'] = df[['math', 'chinese', 'english']].median(axis=1)
print("\n添加统计量后的数据集:")
print(df)
```

5. 数据类型转换

转换年龄列为整数类型：

```
# 安全转换数据类型
df['age'] = pd.to_numeric(df['age'], errors='coerce').astype('Int64')
print("\n转换数据类型后的数据集:")
print(df.dtypes)
```

6. 重置索引

清理后的索引重整：

```
# 重置索引（不保留旧索引）
df = df.reset_index(drop=True)
print("\n最终清洗后的数据集:")
print(df)
```

4.5 数据分析

1. 数据分组统计

```
# 按班级分组计算平均分
grouped_mean = grades_df.groupby('class')['score'].mean()
print("\n各班级平均分:")
print(grouped_mean)

# 按班级和学科分组计算统计量
grouped_multi = grades_df.groupby(['class', 'subject'])['score'].agg(['mean', 'max',
'count'])
print("\n班级-学科多维统计:")
print(grouped_multi)
```

2. 数据提取转换

```
# 提取用户名和域名
extract_result = email_df['email'].str.extract(r'(\w+)\.(\w+)@(\w+)\.(\w+)')
extract_result.columns = ['first_name', 'last_name', 'domain', 'suffix']
print("\n邮箱成分提取结果:")
print(extract_result)

# 提取数学成绩高于80的记录
high_math = grades_df[(grades_df['subject'] == 'math') & (grades_df['score'] > 80)]
print("\n高分数学成绩记录:")
print(high_math)
```

3. 数据合并连接

```
# 纵向合并
combined_df = pd.concat([grades_df, new_students], ignore_index=True)
print("\n合并后的数据集:")
print(combined_df.tail(3))

# 横向关联合并
merged_df = pd.merge(combined_df, info_df, on='class')
print("\n关联班级信息后的数据:")
print(merged_df.head(3))
```

5. 读取文件

1. 读取CSV文件

```
import pandas as pd

# 基础读取
df_csv = pd.read_csv('data.csv')
print("CSV数据前3行:")
print(df_csv.head(3))

# 带参数读取
df_csv_param = pd.read_csv(
    'data.csv',
    sep=',',            # 分隔符
    header=0,           # 使用第一行作为列名
    encoding='utf-8',   # 编码格式
    na_values=['NA']    # 自定义缺失值标识
)
```

2. 读取Excel文件

```
# 读取单个sheet
df_excel = pd.read_excel('data.xlsx', sheet_name='Sheet1')
print("\nExcel数据前3行:")
print(df_excel.head(3))

# 读取多个sheet
with pd.ExcelFile('data.xlsx') as xls:
    df_sheet1 = pd.read_excel(xls, 'Sheet1')
    df_sheet2 = pd.read_excel(xls, 'Sheet2')

# 读取所有sheet
excel_all = pd.read_excel('data.xlsx', sheet_name=None)
```

3. 读取JSON文件

```
# 标准JSON读取
df_json = pd.read_json('data.json')
print("\nJSON数据:")
print(df_json)

# 处理嵌套JSON
df_json_nested = pd.json_normalize(
    pd.read_json('nested.json'),
    record_path=['嵌套字段'],
    meta=['元信息字段']
)

# JSON字符串解析
json_str = '{"name":"John", "age":30}'
```

```
df_from_str = pd.read_json(json_str, orient='index').T
```

4. 读取SQL数据库

```
from sqlalchemy import create_engine

# 创建数据库连接
engine = create_engine('sqlite:///example.db')

# 读取单表
df_sql = pd.read_sql('SELECT * FROM table_name', engine)

# 读取表结构信息
table_info = pd.read_sql_table('table_name', engine)
```

5. 读取HTML表格

```
# 读取网页表格
url = 'http://example.com/tables.html'
dfs_html = pd.read_html(url)

# 读取本地HTML
dfs_local = pd.read_html('table.html')

# 带参数解析
df_table = pd.read_html(
    url,
    attrs={'id': 'table1'},  # 指定表格属性
    header=0,                # 表头行
    parse_dates=['日期列']    # 日期解析
)[0]  # 返回列表中的第一个表格
```

总结

pandas是Python数据分析的核心库，提供高效的数据结构和数据处理功能，支持从数据读取、清洗、转换到分析的全流程操作。

核心功能：

数据结构创建

- `pd.Series()` - 创建一维数组结构
- `pd.DataFrame()` - 创建二维表格结构

数据查看

- `head(n)` - 查看前n行数据
- `tail(n)` - 查看后n行数据

- `info()` - 显示数据基本信息（列名、类型、非空值等）
- `describe()` - 显示数值列统计信息（计数、均值、标准差等）

数据选择

- `df['col']` - 选择单列
- `df[['col1','col2']]` - 选择多列
- `iloc[]` - 按位置索引选择行
- `loc[]` - 按标签选择数据

数据操作

- `drop()` - 删除指定行或列
- `fillna()` - 填充缺失值
- `dropna()` - 删除缺失值
- `drop_duplicates()` - 删除重复行
- `clip()` - 限制数值范围
- `astype()` - 转换数据类型
- `reset_index()` - 重置索引

数据分析

- `groupby()` - 数据分组
- `mean()` - 计算平均值
- `max()` - 计算最大值
- `min()` - 计算最小值
- `count()` - 计数
- `agg()` - 多函数聚合计算

文件读写

- `read_csv()` - 读取CSV文件
- `read_excel()` - 读取Excel文件
- `read_json()` - 读取JSON文件
- `read_sql()` - 读取SQL数据库
- `read_html()` - 读取HTML表格

数据合并

- `concat()` - 数据纵向合并
- `merge()` - 数据横向关联合并

L4-Torch使用

1. Torch库是什么?

`torch` 是一个广泛使用的开源机器学习库，它提供了丰富的功能来支持深度学习研究和开发。

`torch` 提供了灵活的张量操作、自动求导机制以及GPU加速等功能。它使得开发者能够快速构建和训练复杂的神经网络模型。

2. 安装Torch库

2.1 Anaconda安装

Anaconda是python的包管理器，可以很方便的管理不同项目的python环境，解决不同项目python包的环境冲突问题。

我们做python项目时要养成良好的习惯，不同的python项目要采取不同的python虚拟环境，不同的虚拟环境之间相互隔离，python版本和包均不共用。python虚拟环境的创建和管理常用Anaconda。

安装Anaconda步骤：

官网下载安装包：https://www.anaconda.com/distribution/ ，随后运行并选择安装路径，等待安装完成。(要记得勾选 Add Anaconda to the system PATH environment variable，是为了将Anaconda添加到环境变量中。是的它显示不建议你这样做，但我建议你这样做，要不然还要自己手动把他添加到环境变量里)。

安装完毕，查看是否安装成功。
在cmd(按住win+R)中输入conda

```
C:\Users>conda
```

回车，如果出现如下信息则说明安装成功。

```
C:\Users>conda
usage: conda-script.py [-h][-V] command ...
conda is a tool for managing and deploying applications, environments and packages.
Options :
positional arguments :
...
```

2.2 CUDA与cuDNN安装

（如果没有GPU跳过此步骤）
这里建议在CSDN上搜索教程安装。需要检查电脑所支持的CUDA版本，不要安装错版本。

2.3 Pytorch安装

（1）配置torch环境

打开Anaconda Prompt(anaconda)，然后输入

```
conda env list
```

命令行中会显示出当前已存在的python虚拟环境，如果是刚安装anaconda,应该只有一个base基环境。

下面我们新建立一个python虚拟环境(命名为new)

```
conda create -n new python=3.12
```

这里为了在创建环境的时候指定了python解释器的版本，避免疏漏

然后激活环境。在anaconda prompt中输入：

```
conda activate new
```

可以看到命令行中base变成了new，说明成功了。

```
(new) C:\Users\李翔宇>
```

此时再进行python包的安装就是对这个虚拟环境操作，比如我们输入

```
pip install numpy
```

或者

```
conda install numpy
```

再输入

```
conda list
```

就可以看到new这个虚拟环境里面已经有numpy这个包了。说明numpy安装成功。

（2）pytorch的安装

打开pytorch官网：https://pytorch.org/get-started/previous-versions/ 找到自己所对应版本的使用conda命令安装即可

至此pytorch应该就安装完成了。下面介绍torch库。

3. Torch库简介

3.1 张量tensor的创建

张量可以理解成多维数组。
以下是torch中有关函数和代码示例：
`torch.tensor` 从数据创建张量

```
import torch

# 从数据创建
x = torch.tensor([1, 2, 3])           # 标量或列表转张量
x = torch.tensor([[1, 2], [3, 4]])    # 2D张量（矩阵）
```

`torch.arange` 顺序创建张量

```
# 顺序创建
y = torch.arange(5)               # tensor([0, 1, 2, 3, 4]),默认start=0, step=1
y = torch.arange(1, 4)            # tensor([1, 2, 3]),指定start和end
y = torch.arange(1, 2.5, 0.5)     # tensor([1.0, 1.5, 2.0]),支持浮点数步长
```

`torch.zeros` 全0张量

```
zeros = torch.zeros(2, 3)             # 全0张量
#tensor([[0, 0, 0], [0, 0, 0]])
```

`torch.ones` 全1张量

```
ones = torch.ones(2, 3)               # 全1张量
#tensor([[1, 1, 1], [1, 1, 1]])
```

`torch.rand` 均匀随机张量

```
rand = torch.rand(2, 3)               # 均匀随机张量（0~1）
#tensor([[0.1234, 0.5678, 0.9012], [0.3456, 0.7890, 0.2345]])
```

`torch.randn` 标准正态分布张量

```
randn = torch.randn(2, 3)             # 标准正态分布张量
#tensor([[ 0.1234, -0.5678,  1.2345], [-0.3456,  0.7890, -1.2345]])
```

`torch.eye` 单位矩阵

```
eye = torch.eye(3)                    # 单位矩阵
#tensor([[1., 0., 0.],
#        [0., 1., 0.],
#        [0., 0., 1.]])
```

`torch.zeros_like` 形状同另一个张量的全0张量

```
x_like = torch.zeros_like(rand)       # 形状同rand的全0张量
```

3.2 张量tensor的形状操作

`view` 调整形状,要求张量的内存必须是连续的
`reshape` 调整形状,但不要求输入张量是连续的
`flatten` 展平为1维张量
`T` 转置
`permute` 维度交换
`unsqueezed` 增加维度
`squeezed` 压缩维度

```
import torch

x = torch.tensor([[1, 2, 3], [4, 5, 6]])

# 调整形状
view = x.view(3, 2)                  # tensor([[1, 2], [3, 4], [5, 6]])
reshaped = x.reshape(3, 2)           # tensor([[1, 2], [3, 4], [5, 6]])
flattened = x.flatten()              # tensor([1, 2, 3, 4, 5, 6])

# 转置
transposed = x.T                     # tensor([[1, 4], [2, 5], [3, 6]])
                                     # 等价于 x.transpose(0, 1)

# 维度交换
permuted = x.permute(1, 0)           # tensor([[1, 4], [2, 5], [3, 6]])
                                     # 效果与transpose相同

# 增加/压缩维度
unsqueezed = x.unsqueeze(0)          # tensor([[[1, 2, 3], [4, 5, 6]]])
                                     # shape: (1, 2, 3)
squeezed = unsqueezed.squeeze(0)     # tensor([[1, 2, 3], [4, 5, 6]])
                                     # 恢复原shape: (2, 3)
```

3.3 数学运算

`+` 加
`-` 减
`*` 乘
`/` 除
`sqrt` 平方根
`matmul` 矩阵乘法
`sum` 求和
`mean` 均值
`max` 最大值及索引

```
import torch

a = torch.tensor([[1, 2, 3], [4, 5, 6]])
b = torch.tensor([[1, 3, 5], [2, 4, 6]])

# 逐元素运算
```

```
add = a + b                             # tensor([[2, 5, 8], [6, 9, 12]])
mul = a * b                             # tensor([[1, 6, 15], [8, 20, 36]])
div = a / b                             # tensor([[1.0000, 0.6667, 0.6000],
                                        #         [2.0000, 1.2500, 1.0000]])
sqrt = torch.sqrt(a)                    # tensor([[1.0000, 1.4142, 1.7321],
                                        #         [2.0000, 2.2361, 2.4495]])

# 正确维度的矩阵乘法示例
mat_a = torch.tensor([[1, 2, 3],
                      [4, 5, 6]])
mat_b = torch.tensor([[1, 3],
                      [5, 2],
                      [4, 6]])

matmul = torch.matmul(mat_a, mat_b)  # 或 mat_a @ mat_b
                                     #tensor([[23, 25],
                                     #        [53, 58]])

# 归约运算
sum_all = a.sum()                       # tensor(21) (1+2+...+6)
sum_dim = a.sum(dim=0)                  # tensor([5, 7, 9]) (列求和)
mean = a.mean()                         # tensor(3.5000) (21/6)
max_val, max_idx = a.max(dim=1)         # (tensor([3, 6]), tensor([2, 2]))
                                        # 每行最大值及其索引
```

3.4 索引与切片

一般都是使用start : end : step来表示获取的数据范围，范围是前闭后开，step表示步长。Pytorch不支持负步长，但是支持负索引，-1指的是该维度下的最后一个索引。

```
import torch
x = torch.arange(12).reshape(3, 4)  # 3行4列
print(x)
# tensor([[ 0,  1,  2,  3],
#         [ 4,  5,  6,  7],
#         [ 8,  9, 10, 11]])

# 切片操作
row_slice = x[1, :]      # 第2行所有列 → [4, 5, 6, 7]
col_slice = x[:, 2]      # 所有行第3列 → [2, 6, 10]
block = x[0:2, 1:3]      # 第1-2行，第2-3列 → [[1, 2], [5, 6]]
step_slice = x[::2, ::3] # 隔行(步长2)隔列(步长3) → [[0, 3], [8, 11]]

# 布尔掩码
mask = x > 0.5
selected = x[mask]                      # 返回1D张量
```

3.5 类型与设备转换

```
x = torch.rand(2, 3)

# 数据类型转换
x_float = x.float()                    # 转为float32
x_double = x.double()                  # 转为float64

# 设备转移（CPU/GPU）
if torch.cuda.is_available():
    x_gpu = x.to('cuda')               # 转移到GPU
    x_cpu = x_gpu.to('cpu')            # 转回CPU
```

3.6 广播机制

广播的意思是自动扩展维度以匹配操作；广播机制允许不同形状的张量在某些维度自动扩展以实现数学运算。

核心规则如下：

1. 从后往前比对维度：
2. 两维相等，或其中一维为1，可以扩展
3. 否则报错

示例：

```
#标量广播
a = torch.tensor([[1, 2], [3, 4]])
b = torch.tensor(10)
a + b     # 等效于 [[11, 12], [13, 14]]

#维度自动扩展
a = torch.tensor([[1], [2], [3]])   # (3,1)
b = torch.tensor([10, 20])          # (2,)
a + b   # 结果为(3,2)，广播为 [[11, 21], [12, 22], [13, 23]]

#批量矩阵加偏置
x = torch.randn(32, 100)            # batch_size = 32
bias = torch.randn(100)             # 对每个样本添加同样的偏置
out = x + bias
```

广播操作无须显式扩展维度，节省内存，是深度学习模型中常见的操作模式

3.7 张量拼接与分割

```python
a, b = torch.rand(2, 3), torch.rand(2, 3)

# 拼接
cat = torch.cat([a, b], dim=0)        # 沿0维拼接 → (4, 3)
stack = torch.stack([a, b], dim=0)    # 新维堆叠 → (2, 2, 3)

# 分割
chunks = torch.chunk(cat, 2, dim=0)  # 将张量沿指定维度 dim=0 均匀分成2块（尽可能均分）
split = torch.split(cat, 2, dim=0)   # 每块大小2

# 选择
dim_0_0 = torch.select(cat, 0, 0)     # 沿维度 0 选择索引为 0 的子张量 → (1, 3)
dim_0_1 = torch.select(cat, 0, 1)     # 沿维度 0 选择索引为 1 的子张量 → (1, 3)
```

3.8 其他使用操作

```python
x = torch.tensor([1, 2, 3])

# 克隆（避免共享内存）
y = x.clone()                         # 独立副本

# 原地操作（避免内存分配）
x.add_(1)                             # 等价于 x += 1

# 条件赋值
y = torch.where(x > 1, x, torch.zeros_like(x))  # >1保留，否则置0
```

总结

PyTorch是一个广泛使用的开源机器学习库，提供灵活的张量操作、自动求导机制和GPU加速功能，支持深度学习研究和开发。

核心功能

张量创建

- `torch.tensor()` - 从数据创建张量
- `torch.arange()` - 创建顺序张量
- `torch.zeros()` - 创建全0张量
- `torch.ones()` - 创建全1张量
- `torch.rand()` - 创建均匀随机张量
- `torch.randn()` - 创建正态分布张量
- `torch.eye()` - 创建单位矩阵

形状操作

- `view()` - 调整张量形状（需连续内存）

- `reshape()` - 调整张量形状（无需连续内存）
- `flatten()` - 展平为1维张量
- `T` - 张量转置
- `permute()` - 维度交换
- `unsqueeze()` - 增加维度
- `squeeze()` - 压缩维度

数学运算

- `+`, `-`, `*`, `/` - 基本算术运算
- `torch.sqrt()` - 平方根计算
- `torch.matmul()` - 矩阵乘法
- `sum()` - 求和计算
- `mean()` - 均值计算
- `max()` - 最大值及索引查找

索引切片

- `x[i, j]` - 基本索引
- `x[start:end:step]` - 切片操作
- `x[mask]` - 布尔掩码索引

类型设备转换

- `.float()` - 转换为float32类型
- `.double()` - 转换为float64类型
- `.to('cuda')` - 转移到GPU设备
- `.to('cpu')` - 转移回CPU设备

广播机制

- 自动维度扩展 - 不同形状张量的运算支持
- 标量广播 - 标量与张量的自动运算
- 维度匹配 - 按规则自动扩展维度

张量操作

- `torch.cat()` - 张量拼接
- `torch.stack()` - 张量堆叠
- `torch.chunk()` - 张量分块
- `torch.split()` - 按大小分割
- `torch.select()` - 维度选择

高级操作

- `.clone()` - 张量克隆（避免共享内存）
- `add_()` - 原地操作（节省内存）
- `torch.where()` - 条件赋值操作

GPU支持

- `torch.cuda.is_available()` - 检测GPU可用性
- CUDA版本匹配 - 确保GPU驱动兼容
- 设备转移 - CPU/GPU间数据迁移

L5-手写RNN、CNN、Transformer等经典架构

RNN

工作原理：
神经网络是由相互连接的节点组成的网络，每个节点都会对输入数据进行某种形式的处理。在传统的神经网络中，所有的输入都是独立处理的，这意味着网络无法在处理当前输入时考虑到之前的输入。而RNN的设计就是为了解决这个问题。

传统的神经网络数据流只在一个方向上流动，即从输入到输出。这些网络在处理与时间或序列无关的问题时表现良好，例如图像识别。但是对于需要考虑数据的时间序列信息的任务（如语言模型），这种一次性处理模式就略显劣势。

RNN的核心思想是使用循环，使得网络能够将信息从一个步骤传递到下一个步骤。这种循环结构使得网络能够保留某种状态，即网络在处理当前输入时，同时考虑之前的输入。在RNN中，每个序列元素都会更新网络的隐藏状态。这个隐藏状态是网络记忆之前信息的关键，它可以被视为网络的“记忆”。

为了处理序列中的每个元素，RNN会对每个输入执行相同的任务，但每一步都会有一些小的改变，因为它包含了之前步骤的信息。这种结构使得RNN在处理序列数据时非常有效，例如，在文本中，当前单词的含义可能取决于之前的单词。

例子：基于 RNN 的正弦波序列预测实现

Setup

```
import torch
import torch.nn as nn
import numpy as np
import matplotlib.pyplot as plt
```

1、定义简单的RNN模型

这里forward函数表述RNN的前向传播。其中参数x表示输入张量，形状为 (batch_size, seq_len, input_size)，返回值output为输出张量，形状 (batch_size, seq_len, output_size)，而hidden则代表最后一个时间步的隐藏状态。

```python
class SimpleRNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(SimpleRNN, self).__init__()

        self.hidden_size = hidden_size

        # 输入到隐藏层的权重
        self.Wxh = nn.Linear(input_size, hidden_size)
        # 隐藏层到隐藏层的权重
        self.Whh = nn.Linear(hidden_size, hidden_size)
        # 隐藏层到输出层的权重
        self.Why = nn.Linear(hidden_size, output_size)

        # 初始化隐藏状态
        self.hidden = self.init_hidden()

    def init_hidden(self, batch_size=1):
        return torch.zeros(batch_size, self.hidden_size)

    def forward(self, x):
        batch_size, seq_len, input_size = x.size()

        # 重置隐藏状态
        self.hidden = self.init_hidden(batch_size)

        # 存储每个时间步的输出
        outputs = []

        # 遍历序列中的每个时间步
        for t in range(seq_len):
            # 获取当前时间步的输入
            x_t = x[:, t, :]

            # 计算隐藏状态: h_t = tanh(Wxh * x_t + Whh * h_{t-1})
            hidden_t = torch.tanh(self.Wxh(x_t) + self.Whh(self.hidden))

            # 计算输出: y_t = Why * h_t
            output_t = self.Why(hidden_t)

            # 更新隐藏状态和输出列表
            self.hidden = hidden_t
            outputs.append(output_t)

        # 将输出列表转换为张量并调整形状
        outputs = torch.stack(outputs, dim=1)
        return outputs, self.hidden
```

2、生成测试数据

```python
def generate_sine_data(seq_length=20, num_samples=1000):
    # 生成时间步
```

```
    time = np.linspace(0, 30, num_samples + seq_length)

    # 生成正弦波
    data = np.sin(time)

    # 创建输入输出序列
    X, y = [], []
    for i in range(len(data) - seq_length):
        X.append(data[i:i+seq_length])
        y.append(data[i+1:i+seq_length+1])

    # 转换为PyTorch张量并添加维度 (样本数, 序列长度, 特征数)
    X = torch.FloatTensor(X).unsqueeze(2)
    y = torch.FloatTensor(y).unsqueeze(2)

    return X, y
```

3、训练模型

```
def train_rnn():
    # 超参数
    input_size = 1        # 输入特征数（正弦波值）
    hidden_size = 32      # 隐藏层大小
    output_size = 1       # 输出特征数
    seq_length = 20       # 序列长度
    num_epochs = 100      # 训练轮数
    learning_rate = 0.01 # 学习率

    # 生成数据
    X, y = generate_sine_data(seq_length=seq_length)

    # 划分训练集和测试集
    split_idx = int(0.8 * len(X))
    X_train, y_train = X[:split_idx], y[:split_idx]
    X_test, y_test = X[split_idx:], y[split_idx:]

    # 创建模型、损失函数和优化器
    model = SimpleRNN(input_size, hidden_size, output_size)
    criterion = nn.MSELoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

    # 训练循环
    for epoch in range(num_epochs):
        # 前向传播
        outputs, _ = model(X_train)
        loss = criterion(outputs, y_train)

        # 反向传播和优化
        optimizer.zero_grad()  # 清零梯度
        loss.backward()        # 反向传播
        optimizer.step()       # 更新参数
```

```
        # 打印训练进度
        if (epoch + 1) % 10 == 0:
            print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.6f}')

    # 在测试集上评估
    model.eval()
    with torch.no_grad():
        test_outputs, _ = model(X_test)
        test_loss = criterion(test_outputs, y_test)
        print(f'Test Loss: {test_loss.item():.6f}')

    # 可视化预测结果
    plt.figure(figsize=(12, 6))

    # 绘制一个测试样本的预测结果
    sample_idx = 0
    plt.plot(y_test[sample_idx, :, 0], label='真实值')
    plt.plot(test_outputs[sample_idx, :, 0], label='预测值', linestyle='--')
    plt.title('RNN正弦波序列预测')
    plt.xlabel('时间步')
    plt.ylabel('值')
    plt.legend()
    plt.show()

    return model
```

4、运行训练

```
if __name__ == "__main__":
    model = train_rnn()
```

CNN

工作原理： CNN，即Convolutional Neural Network，卷积神经网络。它是一种专为处理具有类似网格结构的数据（如图像、音频、时序信号）而设计的深度神经网络。其核心思想是通过卷积操作自动提取局部特征，实现空间不变性和参数高效性。它的主要结构包括：

卷积层（Convolutional Layer）：通过卷积核（filter/kernel）滑动提取局部特征。

激活层（Activation Layer）：常用ReLU等非线性函数。

池化层（Pooling Layer）：如最大池化（Max Pooling）、平均池化（Average Pooling），实现下采样和特征压缩。

全连接层（Fully Connected Layer, FC）：用于整合高层语义特征，输出分类或回归结果。

数学表述：

1、卷积操作

设输入特征图为 X，卷积核为 W，偏置为 b，输出特征图为 Y，则二维卷积可表示为：
$$Y{i,j}^{(k)} = f \left(\sum{m=1}^{M} \sum{n=1}^{N} W{m,n}^{(k)} \cdot X_{i+m-1, j+n-1} + b^{(k)} \right)$$
其中 f 为激活函数，k 表示第 k 个卷积核。

2、池化操作

以最大池化为例，P 为池化窗口：
$$Y{i,j} = \max{(m,n) \in P} X_{i+m, j+n}$$

3、前向传播流程

假设网络有 L 层卷积/池化，最后接全连接层，最终输出为 $\hat{y}$：

$$
\begin{aligned}
a^{(0)} &= X \\
a^{(l)} &= f^{(l)}\left(\mathrm{Conv/Pool}(a^{(l-1)})\right), \quad l = 1, 2, \ldots, L \\
\hat{y} &= \mathrm{Softmax}\left(W^{(fc)} a^{(L)} + b^{(fc)}\right)
\end{aligned}
$$

例子：用CNN实现MNIST手写数字分类

Setup

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import matplotlib.pyplot as plt
from torchvision import datasets, transforms
```

1、数据加载与预处理

```
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])
trainset = datasets.MNIST(root='./data', train=True, download=True, transform=transform)
testset = datasets.MNIST(root='./data', train=False, download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=64, shuffle=True)
testloader = torch.utils.data.DataLoader(testset, batch_size=1000, shuffle=False)
```

2、定义CNN模型

```
class SimpleCNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 16, kernel_size=3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, padding=1)
        self.fc1 = nn.Linear(32 * 7 * 7, 128)
        self.fc2 = nn.Linear(128, 10)
    def forward(self, x):
```

```
        x = self.pool(F.relu(self.conv1(x)))  # 28x28 -> 14x14
        x = self.pool(F.relu(self.conv2(x)))  # 14x14 -> 7x7
        x = x.view(-1, 32 * 7 * 7)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x

model = SimpleCNN()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
criterion = nn.CrossEntropyLoss()
```

3、训练模型并记录损失

```
losses = []
epochs = 5
for epoch in range(epochs):
    running_loss = 0.0
    for images, labels in trainloader:
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
    avg_loss = running_loss / len(trainloader)
    losses.append(avg_loss)
    print(f"Epoch {epoch+1}/{epochs}, Loss: {avg_loss:.4f}")
```

4、可视化训练损失曲线

```
plt.figure(figsize=(7, 4))
plt.plot(range(1, epochs+1), losses, marker='o')
plt.title('Training Loss Curve (CNN on MNIST)')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.grid(True)
plt.tight_layout()
plt.show()
```

5、测试集准确率

```
model.eval()
correct = 0
total = 0
with torch.no_grad():
    for images, labels in testloader:
        outputs = model(images)
        _, predicted = torch.max(outputs, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
print(f"Test Accuracy: {100 * correct / total:.2f}%")
```

6、可视化部分测试样本及预测结果

```
examples = enumerate(testloader)
batch_idx, (example_data, example_targets) = next(examples)
output = model(example_data)
_, preds = torch.max(output, 1)

plt.figure(figsize=(10, 3))
for i in range(10):
    plt.subplot(2, 5, i+1)
    plt.imshow(example_data[i][0].cpu().numpy(), cmap='gray')
    plt.title(f"Label: {example_targets[i]}\nPred: {preds[i].item()}")
    plt.axis('off')
plt.suptitle('CNN Predictions on MNIST Test Samples')
plt.tight_layout(rect=[0, 0, 1, 0.92])
plt.show()
```

Transformer

Transformer是一种在自然语言处理任务中广泛使用的模型，它基于注意力机制，特别是自注意力机制和多头注意力机制，来捕捉输入序列中的长距离依赖关系。Transformer模型的主要优点是它可以并行处理整个输入序列，而不是像循环神经网络（RNN）那样逐个处理序列中的元素。这使得Transformer模型在处理长序列时具有更高的效率和更好的性能。值得一提的是，后面产生的大语言模型很多都是基于Transformer结构的，如Bert、Gpt。

下面我们来分别编写几个Transformer重要部分。

Setup

```
import torch
import torch.nn as nn
import math
```

1、PositionalEncoding

```
class PositionalEncoding(nn.Module):
    def __init__(self, d_model, max_len=5000):
        super().__init__()
```

```
        # 创建位置编码矩阵：shape [max_len, d_model]
        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2) * (-math.log(10000.0) / d_model))

        # 偶数位置编码使用sin，奇数位置使用cos
        pe[:, 0::2] = torch.sin(position * div_term)  # even
        pe[:, 1::2] = torch.cos(position * div_term)  # odd

        # 添加 batch 维度：shape [1, max_len, d_model]
        pe = pe.unsqueeze(0)
        self.register_buffer('pe', pe)

    def forward(self, x):
        # x: [batch_size, seq_len, d_model]
        x = x + self.pe[:, :x.size(1), :]
        return x
```

2、Multi-Head Attention

```
class MultiHeadAttention(nn.Module):
    def __init__(self, d_model, n_heads):
        super().__init__()
        assert d_model % n_heads == 0

        self.n_heads = n_heads
        self.head_dim = d_model // n_heads

        self.wq = nn.Linear(d_model, d_model)
        self.wk = nn.Linear(d_model, d_model)
        self.wv = nn.Linear(d_model, d_model)
        self.fc_out = nn.Linear(d_model, d_model)

    def forward(self, x, mask=None):
        B, S, D = x.size()
        Q = self.wq(x)
        K = self.wk(x)
        V = self.wv(x)

        # 拆成多头
        Q = Q.view(B, S, self.n_heads, self.head_dim).transpose(1, 2)
        K = K.view(B, S, self.n_heads, self.head_dim).transpose(1, 2)
        V = V.view(B, S, self.n_heads, self.head_dim).transpose(1, 2)

        # 注意力
        scores = torch.matmul(Q, K.transpose(-2, -1)) / math.sqrt(self.head_dim)
        if mask is not None:
            scores = scores.masked_fill(mask == 0, float('-inf'))
        attention = torch.softmax(scores, dim=-1)
        out = torch.matmul(attention, V)
```

```
        # 拼接多头
        out = out.transpose(1, 2).contiguous().view(B, S, D)
        return self.fc_out(out)
```

3、EncoderLayer

```
class TransformerEncoderLayer(nn.Module):
    def __init__(self, d_model, n_heads, dim_feedforward, dropout):
        super().__init__()
        self.self_attn = MultiHeadAttention(d_model, n_heads)
        self.linear1 = nn.Linear(d_model, dim_feedforward)
        self.linear2 = nn.Linear(dim_feedforward, d_model)
        self.dropout = nn.Dropout(dropout)
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)
        self.activation = nn.ReLU()

    def forward(self, src, src_mask=None):
        # 多头注意力 + 残差连接 + LayerNorm
        attn_output = self.self_attn(src, src_mask)
        src = self.norm1(src + self.dropout(attn_output))

        # FFN + 残差连接 + LayerNorm
        ffn_output = self.linear2(self.dropout(self.activation(self.linear1(src))))
        src = self.norm2(src + self.dropout(ffn_output))

        return src
```

4、Encoder

```
class TransformerEncoder(nn.Module):
    def __init__(self, num_layers, d_model, n_heads, dim_feedforward, dropout,
max_len=5000):
        super().__init__()
        self.pos_encoder = PositionalEncoding(d_model, max_len)
        self.layers = nn.ModuleList([
            TransformerEncoderLayer(d_model, n_heads, dim_feedforward, dropout)
            for _ in range(num_layers)
        ])
        self.dropout = nn.Dropout(dropout)

    def forward(self, src, src_mask=None):
        # 添加位置编码
        src = self.pos_encoder(src)
        src = self.dropout(src)

        # 逐层传递
        for layer in self.layers:
            src = layer(src, src_mask)
        return src
```

下面是一个实例：

```
d_model = 512
n_heads = 8
dim_ff = 2048
dropout = 0.1
num_layers = 6

model = TransformerEncoder(num_layers, d_model, n_heads, dim_ff, dropout)

src = torch.rand(32, 100, d_model)
output = model(src)
```

总结

本教程以Python手写实现为核心，完成了RNN、CNN、Transformer三大经典架构的代码实现，深化对深度学习本质的理解。深度学习的核心是让模型从数据中自主学习特征，而非依赖人工设计，这一点在实操中得以体现出来：RNN通过循环结构捕捉时序特征，CNN凭卷积与池化提取空间特征，Transformer以自注意力建立全局关联，三者分别适配不同场景，印证了架构为任务服务的设计逻辑。

参考文献：

https://blog.csdn.net/Iconicdusk/article/details/136520154

https://blog.csdn.net/ai_aijiang/article/details/149255318

https://blog.csdn.net/weixin_64110589/article/details/149603632

L6-实战训练

本项目演示如何用 PyTorch + torchvision 实现 **图像二分类任务**（如猫狗分类、自定义两类数据集），并通过迁移学习快速训练出一个高准确率的模型。

为起到实战锻炼效果，本部分开始需要同学自己下载数据集合，自己调优模型，自己评测自己的模型效果。

---

1. 数据准备

1.1 文件结构

准备一个目录 `data/`，并放置如下结构的图像数据（支持任意两类，示例为 Cat 与 Dog）：

```
data/
  train/
    Cat/  xxx.jpg
          yyy.jpg
    Dog/  aaa.jpg
          bbb.jpg
  val/
    Cat/  ccc.jpg
          ddd.jpg
    Dog/  eee.jpg
          fff.jpg
```

- `train/`：训练集（建议每类 ≥100 张图像）
- `val/`：验证集（每类 ≥20 张图像）

1.2 数据来源

- 可从 **Kaggle Cats vs Dogs** 数据集下载，手动划分 `train/` 与 `val/`。
- 或者准备自己的两类图片（比如“口罩 vs 非口罩”、“有缺陷 vs 正常”）。

---

2. 环境配置

```
conda create -n resnet python=3.12 -y
conda activate resnet

# 安装 PyTorch 与 torchvision（根据你的 CUDA 版本替换）
pip install torch torchvision matplotlib scikit-learn
```

验证安装是否成功：

```
python -c "import torch; print(torch.__version__); print(torch.cuda.is_available())"
```

---

3. 训练代码

保存为 `finetune_resnet.py`：

```
import argparse, time
import torch, torch.nn as nn
from torchvision import datasets, transforms, models
from torch.utils.data import DataLoader
import copy, os

def get_loaders(data_dir, batch_size=32):
    # 数据增强 & 预处理
    train_t = transforms.Compose([
        transforms.Resize((224,224)),
```

```
        transforms.RandomHorizontalFlip(),
        transforms.ColorJitter(0.2,0.2,0.2,0.1),
        transforms.ToTensor()
    ])
    val_t = transforms.Compose([
        transforms.Resize((224,224)),
        transforms.ToTensor()
    ])
    train_ds = datasets.ImageFolder(os.path.join(data_dir, "train"), transform=train_t)
    val_ds   = datasets.ImageFolder(os.path.join(data_dir, "val"),   transform=val_t)
    return (DataLoader(train_ds, batch_size=batch_size, shuffle=True, num_workers=2),
            DataLoader(val_ds,   batch_size=batch_size, shuffle=False, num_workers=2),
            train_ds.classes)

def train(args):
    device = "cuda" if torch.cuda.is_available() else "cpu"
    train_dl, val_dl, classes = get_loaders(args.data_dir, args.batch_size)

    # 加载预训练 ResNet18
    model = models.resnet18(weights=models.ResNet18_Weights.DEFAULT)
    for p in model.parameters():  # 冻结特征层
        p.requires_grad = False
    in_f = model.fc.in_features
    model.fc = nn.Linear(in_f, len(classes))  # 替换最后分类层
    model.to(device)

    crit = nn.CrossEntropyLoss()
    optim = torch.optim.Adam(model.fc.parameters(), lr=args.lr)

    best_w, best_acc = copy.deepcopy(model.state_dict()), 0.0
    for ep in range(1, args.epochs+1):
        # 训练
        model.train()
        tr_loss, tr_correct, tr_total = 0.0, 0, 0
        t0 = time.time()
        for x,y in train_dl:
            x, y = x.to(device), y.to(device)
            optim.zero_grad()
            logit = model(x)
            loss = crit(logit, y)
            loss.backward()
            optim.step()
            tr_loss += loss.item() * x.size(0)
            tr_correct += (logit.argmax(1) == y).sum().item()
            tr_total += x.size(0)
        tr_acc = tr_correct / tr_total

        # 验证
        model.eval()
        va_loss, va_correct, va_total = 0.0, 0, 0
        with torch.no_grad():
            for x,y in val_dl:
```

```
                x, y = x.to(device), y.to(device)
                logit = model(x)
                loss = crit(logit, y)
                va_loss += loss.item() * x.size(0)
                va_correct += (logit.argmax(1) == y).sum().item()
                va_total += x.size(0)
        va_acc = va_correct / va_total

        print(f"Epoch {ep}/{args.epochs} | "
              f"train acc {tr_acc:.3f} | val acc {va_acc:.3f} | "
              f"time {time.time()-t0:.1f}s")

        if va_acc > best_acc:
            best_acc = va_acc
            best_w = copy.deepcopy(model.state_dict())

    model.load_state_dict(best_w)
    torch.save({"state_dict": model.state_dict(), "classes": classes}, "best_resnet18.pt")
    print(f"保存最优模型：best_resnet18.pt（val acc={best_acc:.3f}）")

if __name__ == "__main__":
    ap = argparse.ArgumentParser()
    ap.add_argument("--data_dir", type=str, required=True)
    ap.add_argument("--epochs", type=int, default=5)
    ap.add_argument("--batch_size", type=int, default=32)
    ap.add_argument("--lr", type=float, default=3e-4)
    args = ap.parse_args()
    train(args)
```

运行训练：

```
python finetune_resnet.py --data_dir data --epochs 5 --batch_size 32 --lr 3e-4
```

---

4. 推理代码

保存为 `infer.py`：

```
import argparse, torch
from PIL import Image
from torchvision import transforms, models
import torch.nn as nn

def load_model(ckpt_path, labels=None):
    ckpt = torch.load(ckpt_path, map_location="cpu")
    classes = ckpt.get("classes", None)
    if labels:
        classes = labels.split(",")
    model = models.resnet18()
    model.fc = nn.Linear(model.fc.in_features, len(classes))
```

```
    model.load_state_dict(ckpt["state_dict"])
    model.eval()
    return model, classes

if __name__ == "__main__":
    ap = argparse.ArgumentParser()
    ap.add_argument("--ckpt", required=True)
    ap.add_argument("--img", required=True)
    ap.add_argument("--labels", default=None, help="Comma separated class names if not
saved in ckpt.")
    args = ap.parse_args()

    model, classes = load_model(args.ckpt, args.labels)
    t = transforms.Compose([transforms.Resize((224,224)), transforms.ToTensor()])
    x = t(Image.open(args.img).convert("RGB")).unsqueeze(0)
    with torch.no_grad():
        prob = torch.softmax(model(x), dim=1)[0]
    idx = prob.argmax().item()
    print(f"预测：{classes[idx]} (概率 {prob[idx].item():.3f}) ")
```

推理示例：

```
python infer.py --ckpt best_resnet18.pt --img data/val/Cat/cat001.jpg --labels "Cat,Dog"
```

---

5. 模型评估与可视化

训练完成后，可以进一步分析模型表现：

5.1 混淆矩阵

```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
import matplotlib.pyplot as plt
import torch

# 假设 val_dl, model, classes 已定义
y_true, y_pred = [], []
device = "cuda" if torch.cuda.is_available() else "cpu"

model.eval()
with torch.no_grad():
    for x,y in val_dl:
        x, y = x.to(device), y.to(device)
        logit = model(x)
        preds = logit.argmax(1)
        y_true.extend(y.cpu().numpy())
        y_pred.extend(preds.cpu().numpy())

cm = confusion_matrix(y_true, y_pred, labels=range(len(classes)))
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=classes)
```

```
disp.plot(cmap="Blues")
plt.title("Confusion Matrix")
plt.show()
```

5.2 可视化预测结果

```
import matplotlib.pyplot as plt

model.eval()
images, labels = next(iter(val_dl))
with torch.no_grad():
    outputs = model(images.to(device))
    preds = outputs.argmax(1).cpu()

plt.figure(figsize=(10,5))
for i in range(8):
    plt.subplot(2,4,i+1)
    plt.imshow(images[i].permute(1,2,0).numpy())
    plt.title(f"T:{classes[labels[i]]}
P:{classes[preds[i]]}")
    plt.axis("off")
plt.tight_layout()
plt.show()
```

---

6. 进阶挑战

- 解冻 ResNet 的最后几层，进行 **微调**（fine-tuning），提升精度。
- 增加 **学习率调度器**（如 `CosineAnnealingLR`）。
- 保存/加载 **整个模型**（不仅是 state_dict）。
- 使用 `torch.onnx.export` 导出 ONNX 模型，方便部署。

总结

本文档提供了从Python基础到实战的完整学习路径：

1. **Python下载与配置**: Say Hello to Python
2. **Python基础**: 从环境配置到语法掌握，为后续学习打下坚实基础
3. **数据处理与可视化**: 使用pandas进行数据分析和处理，掌握数据科学基础技能
4. **Torch库** 学习PyTorch张量操作，为深度学习模型实现做准备
5. **手写经典架构**: 详细讲解RNN，Transformer等经典架构
6. **实战演练**: 通过实战学习项目，让读者体验到完成某类特定任务所需要的步骤

通过系统学习本文档，读者可以从Python零基础成长为一位能基本使用Python的开发者。每个部分都包含了理论讲解、代码实现和实战案例，确保理论与实践相结合，为进一步的研究和应用打下坚实基础。

希望这份文档能够帮助更多的DAer熟练掌握Python！