Python数据杂谈

1. **Python开发环境**

**1.1 python解释器**

Python作为动词，它是解释器，是一种能够执行Python代码的程序。它**读取和解释Python源代码，并将其转换为计算机可以执行的指令**。Python解释器可以理解和执行Python语法和语义，从而实现程序的运行。

Python解释器有多种不同的实现，其中最常见的是CPython，它是官方的Python解释器。CPython是用C语言编写的，它解释和执行Python代码，并**将其转换为字节码形式**，然后通过解释器**循环逐条执行字节码**。

除了CPython之外，还有其他的Python解释器，如Jython、IronPython和PyPy等。这些解释器在实现上有所不同，但它们都能够解释和执行Python代码。

Python解释器的存在使得开发人员能够快速编写和调试代码，并提供交互式编程的能力。开发人员可以通过命令行界面或集成开发环境（IDE）来使用Python解释器。无论使用哪种Python解释器，Python的语法和语义在不同的解释器之间是一致的，因此代码可以在不同的解释器中进行移植和执行。

**1.1.1 Python解释器的下载和安装**



上图的python解释器的路径：C:\Python312\python.exe，一台机器可以安装多个不同版本的Python解释器，比如，我可以安装Python2.7也可以安装Python3.10。

注：如果在控制台输入python命令后，显示找不到命令或者不是内部命令等情况，大部分的原因是因为在第一步时没有勾选上Add Python 3.X to PATH，这个时候我们需要手动的添加环境变量：

右键单击桌面 我的电脑/此电脑-->属性-->找到高级系统设置-->环境变量-->下面的系统变量-->找到 Path

添加python安装的路径（C:\Python312）和Scripts路径（C:\Python312\Scripts）即可。



**1.1.2 Python环境**

当你安装好了一个**特定版本的Python解释器**，你就建立了一个**基础**的Python环境（base interpreter，包括Python解释器以及相关的库和工具），下一步你可以在这个基础环境下安装依赖于这个Python解释器的其他第三方包来执行脚本完成特定的项目任务。

但是在日常的工作中，我们经常会遇见这样的场景：

**1、各个项目使用的python版本不相同**

由于Python的解释器版本众多，各版本之间差异非常大。特别是python2和python3，互不兼容。有些项目可能用的python2.7，有些项目可能用的是python3.6，有些则使用的3.8等，但是它们却需要运行在同一个服务器环境中。（docker除外，docker容器可以隔离不同的项目环境。）

**2、系统依赖自带的解释器**

系统的一些服务组件一般也会依赖Python环境。不同的Linux发行版自带的Python也不同。如ubuntu16自带2.7和3.5版本, Centos7依赖python2.7。而系统很多组件都依赖自带的解释器，比如yum等，你不能轻易删除这个版本，一旦删除或者更改都可能造成系统出问题。

**3、依赖默认的解释器路径冲突**

比如Centos7系统自带的python是2.7，系统很多组件比如yum依赖的都是2.7这个版本，我们发现这些工具开头使用的都是：#!/usr/bin/python。而一些新的使用python开发的服务组件，它们依赖的却是python3.6以上的版本，但是它们一些代码开头用的也是这个引用：#!/usr/bin/python。它们都是用python这一个引用，却没有使用python2、python3这样分开，这就很容易导致它们的一些python引用冲突。

**4、依赖冲突（最常见）**

我们都知道python的软件包依赖经常是个很头疼的问题，经常因为这个问题导致到家在安装一些python环境或者服务组件时失败。而不同的python解释器版本，对软件包依赖库的管理也是个问题。比如sqlalchemy这个包，有些项目使用的python2.7版本，它需要依赖这个库，有些项目使用的python3.6版本，它也需要依赖这个库，有些项目使用的python3.8版本，它同样也需要依赖这个库，但是头疼的是，这三者它们依赖的这个包版本还不一致。sqlalchemy从0.1-2.0有众多版本。这时候如果你在系统上直接使用pip install sqlalchemy的话，它只能选择安装一个版本，但是这样其他两个项目是无法使用这个版本，就会出现依赖冲突的问题。由于 Python 的依赖库管理是中心化的，而且大版本上的不兼容且长期并行，就出现了这么一个独特的话题。

**1.1.3 Python环境管理**

针对上述问题，我们可以基于这个版本的Python解释器（base ）创建新的环境（虚拟环境）来隔离不同的项目需求，即每一个项目基于base interpreter创建一个python环境（虚拟环境）。这里需要用到的两个工具：

* Venv：

python3.3版本之后自带的模块，只支持3.3版本之后的，不支持2.x。

* Virtualenv：

需要自己安装：pip3 install virtualenv，同时支持 python2 和 python3，并可以选择继承基础版本的包。

终端下：

（1）进入到项目目录下D:\myProject

- Python自3.3版本之后，官方自带了用于创建虚拟环境的venv模块

- python -m venv [虚拟环境名称]

- Python自3.3版本之前，

- virtualenv [虚拟环境名称]

- 创建完虚拟环境后，会在项目文件夹下生成一个虚拟环境文件夹D:\myProject\venv，解释器的位置D:\myProject\venv\Scripts\python.exe

* 激活虚拟环境：D:\myProject\venv\Scripts下运行activate.bat
* 安装依赖：pip install <package\_name>，安装的库存在D:\myProject\venv\Lib\site-packages

**当然，也可以在Pycharm中直接为项目创建一个基于**base interpreter的python虚拟环境**。**

两者共同存在的最大问题：

Venv/virtualenv可以为每个虚拟环境指定 python 解释器，但是指定的只能是当前系统已经安装好的python解释器，无法使用其他版本的python解释器。

**1.1.3 pip常用命令**

1.在线安装包pip install <包名> -i

- 阿里云 http://mirrors.aliyun.com/pypi/simple/

- 中国科技大学 https://pypi.mirrors.ustc.edu.cn/simple/

- 豆瓣(douban) http://pypi.douban.com/simple/

- 清华大学 https://pypi.tuna.tsinghua.edu.cn/simple/

- 中国科学技术大学 <http://pypi.mirrors.ustc.edu.cn/simple/>

2. 设置全局镜像源避免每次-i（清华源为例）:

pip config set global.index-url https://pypi.tuna.tsinghua.edu.cn/simple

2. 查看某个包可用的版本/安装指定版本的包：pip install <包名>==

3. 升级包：pip install -U <包名>

4. 卸载包：pip uninstall <包名>

5. 本地安装方式：pip install ～/Downloads/a.whl

6. pip导出安装的库到requirements.txt：pip freeze > requirements.txt

注意：只会导出你使用[pip安装](https://so.csdn.net/so/search?q=pip%E5%AE%89%E8%A3%85&spm=1001.2101.3001.7020)的依赖包

7. pip导入requirements.txt中列出的库到新机：pip install -r requirements.txt

注意：有一些安装包名和导入包名（文件夹名）不一样，比如yaml包，使用时import yaml，但是安装时候是pip install pyyaml

**1.2 anaconda**

尽管Python解释器本身就可以执行Python代码，Anaconda是一个用于科学计算和数据科学的Python发行版，它包含了Python解释器以及大量的科学计算和数据科学相关的库、工具和软件包。Anaconda的目标是简化Python环境的设置和管理，并提供一个集成的平台，使得科学计算和数据科学的工作更加方便和高效。

Anaconda的主要特点包括：

（1）管理Python环境：Anaconda提供了一个命令行工具conda，可以用于创建、管理和切换**不同的Python环境。这使得开发人员可以在同一台机器上同时使用不同版本的Python和库，而不会相互干扰**。

（2）预安装科学计算库：Anaconda预先安装了许多常用的科学计算和数据科学库，如NumPy、SciPy、Pandas和Matplotlib等。这些库对于数据处理、数值计算、统计分析和可视化等任务非常有用，使用Anaconda可以避免手动安装和配置这些库的麻烦。

（3）跨平台支持：**Anaconda可在多个操作系统上运行，包括Windows、macOS和Linux。这使得开发人员可以在不同的平台上保持一致的开发环境**，并轻松共享代码和项目。

（4）社区支持和扩展：Anaconda拥有庞大的用户社区和生态系统，用户可以从社区中获取支持、分享经验和获取扩展库和工具。Anaconda还提供了一个集成的包管理系统，可以方便地安装、更新和管理库和软件包。

**1.2.1 anaconda的下载与安装**

**1.2.2 conda环境**

Conda环境是指由Anaconda或Miniconda（Anaconda的轻量级版本，只包含最基本的组件）软件包管理器创建和管理的**独立Python环境**。

Conda环境允许你在同一台机器上同时管理和使用多个独立的Python环境。每个环境都可以具有不同的Python版本和不同的包集合，这使得你可以更好地管理和组织你的Python项目，确保项目之间的隔离性和依赖关系的一致性。

通过创建Conda环境，你可以：

（1）管理Python版本：你可以创建一个新的Conda环境，并在其中安装特定版本的Python解释器。这使得你可以在不同的项目中使用不同的Python版本而不会相互干扰。

（2）管理包依赖关系：每个Conda环境都可以拥有自己独立的包集合，你可以在每个环境中安装所需的软件包和库，而不会与其他环境产生冲突。这样可以确保每个项目使用的库版本是一致的，避免了包之间的冲突和版本不兼容的问题。

（3）共享环境配置：你可以通过创建和共享environment.yml文件来轻松重现和共享Conda环境的配置。这个文件记录了环境中安装的所有包及其版本，使得其他人可以轻松地创建相同的环境。

当安装好anaconda之后，你就拥有了一个基础conda环境（一种特殊的Python环境），目录通常在D:\software\Anaconda3\python.exe，下一步你可以借助conda命令工具在这个环境下安装这个conda环境需要的其他包，同时，你也可以**指定python版本**建立一个新的conda环境，这个环境的目录通常在D:\software\Anaconda3\envs\env\_name\python.exe来隔离不同项目的需求。在这个python版本解释器下，也可以基于这个python版本的解释器进一步为项目创建虚拟环境，在项目文件夹下生成一个虚拟环境文件夹D:\myProject\venv，这个虚拟环境不继承conda，只继承了python解释器的版本。

**1.2.3 conda环境管理**

Anaconda Prompt 是 Anaconda 发行版中提供的一个命令行界面，用于管理和运行 Python 环境。它基于 Windows 的命令提示符（Command Prompt）或者 macOS/Linux 的终端，提供了一些额外的功能和命令，方便用户在 Anaconda 环境中进行 Python 开发和数据科学工作。

使用 Anaconda Prompt，你可以执行以下操作：

（1）创建和管理 Python 环境：Anaconda Prompt 允许你创建独立的 Python 环境，每个环境可以拥有不同的 Python 版本和依赖包，方便项目之间的隔离和管理。

（2）安装和更新依赖包：通过使用 conda 命令，你可以方便地安装、更新和删除 Python 包和依赖项。conda 是 Anaconda 的包管理工具，类似于 pip，但它可以处理更复杂的环境和依赖关系。

（3）启动 Jupyter Notebook：Anaconda Prompt 提供了命令来启动 Jupyter Notebook，这是一个流行的交互式开发环境，用于编写和运行 Python 代码、创建数据分析报告等。

（4）管理 Anaconda 发行版：你可以使用 Anaconda Prompt 来管理 Anaconda 发行版本身，包括更新 Anaconda、安装额外的包和工具、配置环境变量等。

**谈一谈几种环境的区别（如何创建和使用）：**

1. **Virtualenv Environment**



* [通过 Virtualenv 方式创建虚拟环境](https://blog.csdn.net/mukes/article/details/115905766#_Virtualenv__36)

**b．Conda Environment**



* [通过 Conda 方式创建虚拟环境](https://blog.csdn.net/mukes/article/details/115905766#_Conda__148)

**c．System Interpreter**



**系统解释器**

**d. pipenv Environment**



pip install pipenv

* [通过 Pipenv 方式创建虚拟环境](https://blog.csdn.net/mukes/article/details/115905766#_Pipenv__99)

**e. Poetry Environment**



**通过Poetry方式创建虚拟环境**

注意：Pycharm版本过低可能无法识别高版本的python。报错pycharm unsupported Python 3.1**。**

**1.2.4 conda命令**

1.创建环境

- conda create -n py37 python=3.7

2.查看已有的环境

- conda env list

- conda info -e

- conda info –envs

3.激活某个环境

- conda activate env\_name

4.删除环境/删除环境中的某个包

- conda remove -n env\_name -–all

- conda remove -n env\_name package\_name

5.查看环境下的包

- conda list

- 执行 conda list ，用pip安装的包显示的build项目为pypi。用conda 安装的包显示的是其他

6.在线安装conda install package\_name或pip install package\_name

- conda install xxx：这种方式安装的库会放在anaconda3/pkgs目录下，这样的好处就是，当在某个环境下已经下载好了某个库，再在另一个环境中还需要这个库时，就可以直接从anaconda3/pkgs目录下将该库复制至新环境而不用重复下载。

- pip install xxx：

- 如果当前conda环境的python是conda安装的，和系统的不一样，那么xxx会被安装到anaconda3/envs/current\_env/lib/site-packages文件夹中，

- 如果当前conda环境用的是系统的python，那么xxx会通常会被安装到python12/lib/site-packages文件夹中

7.本地安装方式

- conda install --use-local ~/Downloads/a.tar.bz2

8.在环境下卸载包

- conda uninstall package\_name

9.查看包的版本信息

- anaconda show <USER/PACKAGE>

10.查看anaconda环境下的python版本

- python --version

11.查看conda环境下各个包的版本：conda list

12.查看指定包版本（以matplotlib为例）： conda list matplotlib

13. 查看当前下载源conda config --show-sources

14. 添加下载源

conda config --add channels https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/free/

conda config --add channels https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/main/

15. 升级conda自身

conda update -n base -c conda-forge conda

16. 升级指定环境中所有包的命令

conda update -n 环境名称 --all

17. 升级指定环境中所有包的命令（第二种方法）

conda update -p 环境文件夹绝对路径 --all

18. 导出当前conda env

conda env export > environment.yaml(也可以为\*.yaml文件的绝对路径)

注意：这可能只会导出conda命令直接安装的包，用pip安装在Anaconda的lib和site-package里的包还要用导出pip的方法。

19. 根据conda env yaml文件创建conda env

conda env create -f environment.yaml（也可以为\*.yaml文件的绝对路径）安装时，需要注意修改\*.yaml文件中的 prefix: 用于指定待创建env的专属文件夹。

声明：conda 安装包指定版本时中间是 一个 = 、pip 中间是两个 =

**1.2.5 conda和pip的区别**

- conda可以管理非python包，pip只能管理python包。

- conda自己可以用来创建环境，pip不能，需要依赖virtualenv之类的。

- conda安装的包是编译好的二进制文件，安装包文件过程中会自动安装依赖包；pip安装的包是wheel或源码，装过程中不会去支持python语言之外的依赖项。

- conda安装的包会统一下载到一个目录文件中，当环境B需要下载的包，之前其他环境安装过，就只需要把之间下载的文件复制到环境B中，下载一次多次安装。pip是直接下载到对应环境中。

- conda只能在conda管理的环境中使用比如conda所创建的虚环境中使用。pip可以在任何环境中使用，在conda创建的环境 中使用pip命令，需要先安装pip（conda install pip ），然后可以环境中使用pip。

- conda 安装的包，pip可以卸载，但不能卸载依赖包，pip安装的包，只能用pip卸载。

**1.3 运行python代码的方式**

运行Python程序常见的方式包括交互式运行（Python解释器运行）、命令行运行（脚本运行）、集成开发环境运行。

**1.3.1交互式运行**

Python 是一门解释型语言，在交互式环境中，可以逐行或逐个代码块地执行Python代码，并立即查看结果。这种方式适用于实验、调试和快速尝试代码片段等情况。在命令行中直接运行 python 命令，进入Python解释器的交互式模式。然后，你可以**逐行输入和执行**代码。这种方式适用于简单的代码片段和交互式开发。

**Python console（交互式的Python解释器环境）**

**1.3.2命令行运行**

在命令行终端中，通过运行Python解释器并指定**代码文件**来执行Python代码。这种方式适用于运行较大的代码文件或批处理任务。这里的代码文件通常是指脚本文件。接下来说一说，同样都是python的代码文件，**脚本与模块的有区别？如何区分？**

**脚本（Script）：**

* 脚本是一段可执行的代码，通常用于完成特定的任务或解决特定的问题。
* 脚本通常以 .py 扩展名保存，并且可以在命令行或脚本解释器中直接执行。
* 脚本可以包含一系列的语句和函数调用，用于实现特定的功能。
* 脚本可以独立运行，并且可以接受命令行参数作为输入。
* 脚本通常被用于自动化任务、批处理处理、快速原型开发等场景。

**模块（Module）：**

* 模块是一个包含了一组相关函数、类和变量的文件，它可以被其他程序导入和使用。
* 模块通常以 .py 扩展名保存，并且可以在其他Python程序中通过 import 语句导入。
* 模块可以提供可重用的代码，可以被多个程序共享和调用。
* 模块可以定义函数、类、变量等，并且可以通过命名空间进行访问。
* 模块可以被组织成包（Package），形成更大的代码组织结构。

脚本和模块之间并没有严格的界限，一个脚本文件也可以作为一个模块被导入和使用。这取决于文件的用途和使用方式。可以通过 if \_\_name\_\_ == "\_\_main\_\_": 判断语句，我们可以确定代码是作为脚本执行还是作为模块导入。

在Python中，每个模块都有一个特殊的全局变量\_\_name\_\_，它存储了模块的名称。当模块直接运行时，\_\_name\_\_的值为\_\_main\_\_；当模块被导入时，\_\_name\_\_的值为模块的名称。

**Terminal（终端命令行控制台）**

**1.3.3IDE / 代码编辑器运行**

**1.3.3.1 Pycharm**

PyCharm是一款由JetBrains开发的集成开发环境（IDE），专门用于Python开发。它提供了丰富的功能和工具，以提高开发效率并简化Python**项目的管理**。

**1、新建一个项目**

一个项目就是一个文件夹，下面有一些子文件夹（比如自定义的包）

**2、两种运行模式**

在PyCharm中，你可以选择两种运行模式：Run模式和Debug模式。这两种模式在运行和调试代码时有一些区别。

**Run模式**：

Run模式是默认的运行模式。当你选择Run模式时，PyCharm会执行你的代码，并在控制台或输出窗口显示结果。

在Run模式下，你可以通过设置断点来暂停代码的执行，以便检查变量的值和程序的状态。

在Run模式下，你不能逐行调试代码，也不能查看代码的执行流程。

**Debug模式**：

Debug模式用于逐行调试代码，以便更详细地检查代码的执行过程和变量的值。

在Debug模式下，你可以设置断点，并逐步执行代码，观察每一行的执行结果。

在Debug模式下，你可以查看变量的值、观察表达式的结果，并使用调试工具来分析代码的执行流程。

Debug模式通常用于解决代码中的错误、调试复杂的逻辑或查找程序的性能问题。

**5、常用的快捷键**

**1.3.3.2 Jupyter Notebook**

Jupyter 是一个开源的交互式计算环境，它支持多种编程语言，包括 Python、R、Julia 等。Jupyter通过 Jupyter Notebook 提供了一个基于 Web 的界面，可以创建、编辑和运行 Notebooks。Notebooks是一个包含可执行代码、富文本元素（如说明文档、图表和公式）以及展示结果的交互式环境，你可以在 Jupyter Notebook 中编写和运行 Python 代码。

**启动 Jupyter Notebook**

**修改路径**

**快捷键**

**导出PDF**

1. **Python高阶语法**

Python作为名词，是一种高级编程语言，由Guido van Rossum于1991年创建。它被设计成易于阅读和理解的语言，具有清晰简洁的语法和强大的功能。Python具有广泛的应用领域，包括软件开发、数据分析、人工智能、科学计算、网络编程等。

Python具有许多特点，其中包括：

（1）简洁易读：Python的语法简单明了，使用空格缩进来表示代码块，使得代码易于阅读和理解。

（2）跨平台性：Python可以在多个操作系统上运行，包括Windows、macOS和Linux等。

（3）大量的库和框架：Python拥有丰富的标准库和第三方库，提供了各种功能和工具，使开发人员能够更快速地开发应用程序。

（4）动态类型：Python是一种动态类型语言，不需要显式声明变量的类型，可以根据赋值自动推断类型。

（5）面向对象编程：Python支持面向对象编程，可以使用类和对象来组织和管理代码。

（6）可扩展性：Python可以通过C/C++编写扩展模块，提高性能并与其他语言进行交互。

**2.1 数据类型**

**2.1.1 基本数据类型**

Python常用的基本数据类型有7种：数字(Number)、布尔(Booleans)、字符串(String)、列表(List)、元组(Tuple)、字典(Dictionary)、集合(Set)。

**2.1.1.1数字(Numbers)**

在Python中，有几种常见的数字数据类型，包括：

* 整数（int）：整数是没有小数部分的数字，可以是正数、负数或零。例如：-5、0、10等。
* 浮点数（float）：浮点数是带有小数部分的数字，可以是正数、负数或零。浮点数使用小数点来表示。例如：3.14、-0.5、1.0等。
* 复数（complex）：复数是由实数部分和虚数部分组成的数字。复数的虚数部分用 "j" 或 "J" 表示。例如：2+3j、-1+2j等。

下面是一些使用数字数据类型的示例：

1. # 整数
2. x = 10
3. y = -5
4. z = 0
6. # 浮点数
7. a = 3.14
8. b = -0.5
9. c = 1.0
11. # 复数
12. d = 2 + 3j
13. e = -1 + 2j
15. # 数值计算
16. sum = x + y  # 加法
17. difference = x - y  # 减法
18. product = x \* y  # 乘法
19. quotient = x / y  # 除法
20. power = x \*\* 2  # 幂运算
22. # 内置数学函数
23. import math
25. sqrt = math.sqrt(x)  # 平方根
26. abs\_value = abs(y)  # 绝对值
27. ceil\_value = math.ceil(a)  # 向上取整
28. floor\_value = math.floor(a)  # 向下取整

这些数字数据类型在Python中是内置的，你可以直接使用它们进行数值计算和操作。Python还提供了丰富的数学运算和函数库，可以对数字进行各种数值计算、转换和处理。

**2.1.1.2布尔(Booleans)**

在Python中，布尔（Booleans）是一种表示真（True）或假（False）的数据类型。布尔值用于条件判断和逻辑运算。

在Python中，布尔类型有两个预定义的值：True（真）和False（假）。注意，首字母必须大写，小写的true和false是无效的。

下面是一些使用布尔类型的示例：

1. x = True
2. y = False
4. print(x)  # 输出：True
5. print(y)  # 输出：False
7. # 布尔运算
8. a = 5
9. b = 10
11. print(a > b)  # 输出：False，因为5不大于10
12. print(a < b)  # 输出：True，因为5小于10
14. # 逻辑运算
15. p = True
16. q = False
18. print(p and q)  # 输出：False，因为p和q都是False
19. print(p or q)  # 输出：True，因为p是True
20. print(not p)  # 输出：False，因为p是True，取反后为False

**2.1.1.3元组(Tuples)**

在Python中，元组（tuple）是一种**有序、不可变**的数据类型。**元组允许重复元素，每个元素可以是任意类型**。元组一般使用圆括号 () 将元素括起来，并用逗号分隔。

有序：元组的元素可以通过索引访问，索引从0开始。也可以使用切片（slice）操作获取元组的子集

不可变：不能对元组的元素进行赋值操作，不能动态地添加、删除元素。

**创建方法**

1. # 创建一个空元组
2. empty\_tuple = ()
4. # 创建一个包含多个元素的元组
5. fruits = ('apple', 'banana', 'orange')
7. # 创建一个包含不同类型元素的元组
8. person = ('John', 25, 'USA')
10. # 创建只包含一个元素的元组
11. single\_element\_tuple = (42,)
13. # 创建元组时可以省略括号
14. coordinates = 1, 2, 3

**常规操作**

1. fruits = ('apple', 'banana', 'orange')
3. # 访问元组的元素
4. print(fruits[0])  # 输出：'apple'
5. print(fruits[1])  # 输出：'banana'
7. # 使用切片获取元组的子集
8. print(fruits[1:])  # 输出：('banana', 'orange')
10. # 尝试修改元组的元素（会引发错误）
11. fruits[0] = 'pear'  # 引发 TypeError: 'tuple' object does not support item assignment

应用场景

* 当你希望创建一个不可变的序列时，可以使用元组。
* 当你希望将多个值作为一个整体传递给函数时，可以使用元组。

**2.1.1.4集合(Sets)**

在Python中，集合（Sets）是一种**无序、可变**的数据集合。它是由一组唯一的元素组成（不可重复）。**集合中的元素必须是可哈希的，也就是说，集合中的元素必须是不可变的类型。**可以进行集合运算，如并集、交集、差集等**。**集合一般使用大括号{}将元素括起来，并用逗号分隔。

**创建方法**

**S = set()**

**常规操作**

**应用场景**

**2.1.1.5列表(list)**

在Python中，列表（list）是一种**有序、可变**的数据集合。**列表允许重复元素，可以包含任意类型的元素。**创建列表可以使用方括号 [] 或者使用 list() 函数。

**创建方法**

**空列表：**

L = list() # 通过内置函数

L = []

非空列表：

**常规操作**

1、列表每个元素出现的次数，返回一个字典：

from collections import Counter

counter = Counter(L)

2、计算列表某个值出现的次数，返回int：

val\_count = nums.count(val)

元素插入：

1、将指定对象（obj）插入列表的指定位置（index），无返回值

list.insert(index, obj)

2、在列表的末尾添加一个元素（可以是列表对象，**列表作为一整个元素插入**），无返回值

list.append(obj)

3、在列表的末尾添加多个元素（可以是列表对象，**将列表中的元素取出逐个添加**），无返回值

list.extend()

元素移除：

1、移除列表中的一个元素（默认最后一个元素），返回该元素的值：

Val = list.pop([index=-1])

2、移除列表中的指定对象（obj），无返回值

list.remove(obj)

**应用场景**

**2.1.1.6字典(Dictionaries)**

在Python中，字典（dictionary）是一种**无序的键值对**数据结构，用于存储和管理**具有唯一键的元素**。字典中的每个元素由一个键（key）和一个对应的值（value）组成。字典中的键必须是不可变的类型，通常使用字符串作为键。值可以是任意类型，包括数字、字符串、列表、字典等。创建字典可以使用花括号 {key:value} 或者使用 dict() 函数。

创建方法

创建空字典：

Dic = dict() # 内置函数声明一个字典

Dic = {}

创建非空字典：

常规操作

遍历key：

for key in dict.keys():

遍历value：

For value in dict.values():

同时遍历key和value：

For key, value in dict.items():

应用场景

**字典与json的区别：**

- python中，json和dict非常类似，都是key-value的形式，而且json和dict也可以非常方便的通过dumps和loads互转，但是它们之间还是有区别的：

- json：是一种数据格式，是纯字符串。可以被解析成python的dict或其他形式。

- dict：是一个完整的数据结构，是对hash table这一数据结构的实现，是一套从存储到提取都封装好的方案。它使用内置的hash函数来规划key对应的value的存储位置，从而获得O(1)的数据读取速度。

**2.1.1.7字符串(Strings)**

在Python中，字符串是由字符组成的序列，可以包含字母、数字、符号和空格等字符，是一种表示文本数据的数据类型。创建字符串可以使用单引号 ' ' 或双引号 " " 将文本括起来。字符串也可以使用三重引号 ''' ''' 或 """ """ 来创建多行字符串。

创建方法

S = str()

常规操作

1. 字符串分割：用空格（默认）或固定字符分割字符串（**相当于str—>list**）

s\_1 = s.split()

‘LOVE’ -> [‘L’, ‘O’, ‘V’, ‘E’]

2.字符串拼接：**相当于list—> str**

S = ‘’.join(s\_l)

[‘L’, ‘O’, ‘V’, ‘E’] -> ‘LOVE’

3.字符串反转

original\_string = "hello"

reversed\_string = original\_string[::-1]

print(reversed\_string) # 输出: "olleh"

string.capitalize()

把字符串的第一个字符大写

string.count(str, beg=0, end=len(string))

返回 str 在 string 里面出现的次数，如果 beg 或者 end 指定则返回指定范围内 str 出现的次数

string.endswith(obj, beg=0, end=len(string))

检查字符串是否以 obj 结束，如果beg 或者 end 指定则检查指定的范围内是否以 obj 结束，如果是，返回 True,否则返回 False.

string.find(str, beg=0, end=len(string))

检测 str 是否包含在 string 中，如果 beg 和 end 指定范围，则检查是否包含在指定范围内，如果是返回开始的索引值，否则返回-1

string.index(str, beg=0, end=len(string))

跟find()方法一样，只不过如果str不在 string中会报一个异常.

string.isalnum()

如果 string 至少有一个字符并且所有字符都是字母或数字则返回 True,否则返回 False

string.isalpha()

如果 string 至少有一个字符并且所有字符都是字母则返回 True,否则返回 False

string.isdecimal()

如果 string 只包含十进制数字则返回 True 否则返回 False.

string.isdigit()

如果 string 只包含数字则返回 True 否则返回 False.

string.islower()

如果 string 中包含至少一个区分大小写的字符，并且所有这些(区分大小写的)字符都是小写，则返回 True，否则返回 False

string.isnumeric()

如果 string 中只包含数字字符，则返回 True，否则返回 False

string.isspace()

如果 string 中只包含空格，则返回 True，否则返回 False.

string.istitle()

如果 string 是标题化的(见 title())则返回 True，否则返回 False

string.isupper()

如果 string 中包含至少一个区分大小写的字符，并且所有这些(区分大小写的)字符都是大写，则返回 True，否则返回 False

string.join(seq)

以 string 作为分隔符，将 seq 中所有的元素(的字符串表示)合并为一个新的字符串

string.lower()

转换 string 中所有大写字符为小写.

string.lstrip()

截掉 string 左边的空格

max(str)

返回字符串 str 中最大的字母。

min(str)

返回字符串 str 中最小的字母。

string.replace(str1, str2, num=string.count(str1))

把 string 中的 str1 替换成 str2,如果 num 指定，则替换不超过 num 次.

string.split(str="", num=string.count(str))

以 str 为分隔符切片 string，如果 num 有指定值，则仅分隔 num+ 个子字符串

string.startswith(obj, beg=0,end=len(string))

检查字符串是否是以 obj 开头，是则返回 True，否则返回 False。如果beg 和 end 指定值，则在指定范围内检查.

string.strip([obj])

在 string 上执行 lstrip()和 rstrip()

string.swapcase()

翻转 string 中的大小写

string.title()

返回"标题化"的 string,就是说所有单词都是以大写开始，其余字母均为小写(见 istitle())

string.translate(str, del="")

根据 str 给出的表(包含 256 个字符)转换 string 的字符,要过滤掉的字符放到 del 参数中

string.upper()

转换 string 中的小写字母为大写

应用场景

可哈希性：可哈希性是一个对象的特性，并且与对象的值相关。指一个对象具有哈希值（hash value），并且在其生命周期中保持不变。哈希值是一个整数，用于唯一标识一个对象。**可哈希的对象是不可变的**，常见的可哈希对象包括整数、浮点数、字符串、元组等。这些对象都具有唯一的哈希值，且在其生命周期中不会发生变化，两个具有相同值的字符串会有相同的哈希值。**不可哈希的对象是可变的**，不可哈希的对象包括列表（list）、字典（dictionary）和集合（set）等可变对象。这些对象在其生命周期中可以发生变化，因此它们没有固定的哈希值，两个具有相同值的列表会有不同的哈希值。

数据类型转换：

**2.1.2引申数据类型**

**2.1.2.1索引序列(Series)**

- 由index和value组成

- index是索引，是一个一维列表

- value是值，是一个一维列表

**2.1.2.2数据框(Dataframe)**

- 由index（索引）、colums（字段）、values（值）三部分组成

- index和colums是一个一维列表，values是一个二维列表

**2.1.2.3数组(np.array)**

一维数组：（3， ）表示一维数组，3个元素

二维数组：（3， 3）表示二维数组，3行3列

（None， 3）是一个二维数组的形状（shape）。在这种表示法中，第一个维度的大小是 **None**，这通常意味着这个维度的大小是未定的或可变的，可以在运行时被确定。而第二个维度的大小是 3。

这种形状会出现在神经网络的结构中，有时候输入的形状是不固定的，比如批量大小是不固定的

**2.1.2.4张量（tensor）**

**2.2 魔法函数**

魔法函数（Magic Methods），也被称为特殊方法或双下划线方法，**是Python中一些特殊命名的函数**，它们以双下划线开头和结尾。魔法属性和方法是Python内置的一些属性和方法，有着特殊的含义。命名时前后加上两个下划线，在执行系统特定操作时，会自动调用。用于定义对象的行为和特性。它们允许你在自定义类中重写和定义特定的行为，使你的类具有类似内置类型的行为，例如迭代、比较、字符串表示等。那么接下来，我们就列举一些常见的魔法属性和方法。

**2.3 内置函数**

内置函数（Built-in functions）是 Python 语言提供的一组预定义的函数，可以直接在代码中使用，而无需导入其他模块。这些函数可以完成各种常见的操作，例如类型转换、数学计算、字符串处理、文件操作等。例如，len()、str()、print()、type() 等都是内置函数。

魔法属性和魔法方法是用于**自定义类行为**的特殊方法。它们是与对象相关的，而内置函数是独立于对象的通用函数。需要注意的是，有些内置函数可能与魔法方法的名称相同，例如len()函数与\_\_len\_\_魔法方法。但它们的作用和用法是不同的。内置函数是全局可用的函数，而魔法方法是在类定义中定义的特殊方法。



1、bool()、dict()、float()、int()、list()、object()、set()、str()、tuple()这几个和数据类型相关

2、

any()函数

all() 函数

zip() 函数

l1 = ['a', 'b', 'c', 'd']

l2 = ['1', '2', '3', '4']

res = zip(l1, l2)压缩变成

res = [('a', '1'), ('b', '2'), ('c', '3'), ('d', '4')]

org = zip(\*res)解压缩变成

org = [('a', 'b', 'c', 'd'), ('1', '2', '3', '4')]

enumerate() 函数

sorted()函数

filter()函数: 用于过滤序列，过滤掉不符合条件的元素，返回由符合条件元素组成的新列表。该接收两个参数，第一个为函数，第二个为序列，序列的每个元素作为参数传递给函数进行判断，然后返回 True 或 False，最后将返回 True 的元素放到新列表中。

语法

filter(function, iterable)

参数

function -- 判断函数。

iterable -- 可迭代对象。

返回值

返回列表。

property()函数：把方法包装成属性，让方法可以以属性的形式被访问和调用。

super()函数

print()函数

- 输入单个打印值默认自动换行

- 输入多个打印值要换行设置 sep = ‘\n’

Print(“A”,”B”,”C”,sep=’\n’)

与

Print(‘A’)

Print(‘B’)

Print(‘C’)

等价

**2.4 匿名函数**

**lambda 函数特点：**

* lambda 函数是匿名的，它们没有函数名称，只能通过赋值给变量或作为参数传递给其他函数来使用。
* lambda 函数通常只包含一行代码，这使得它们适用于编写简单的函数。

**lambda 语法格式：**

lambda arguments: expression

* lambda是 Python 的关键字，用于定义 lambda 函数。
* arguments 是参数列表，可以包含零个或多个参数，但必须在冒号(:)前指定。
* expression 是一个表达式，用于计算并返回函数的结果。

以下实例使用 lambda 创建匿名函数，函数参数 a 与 b 相乘，并返回结果：

实例

x = lambda a, b : a \* b

print(x(5, 6))

输出结果为：

30

以下实例使用 lambda 创建匿名函数，函数参数 a、b 与 c 相加，并返回结果：

实例

x = lambda a, b, c : a + b + c

print(x(5, 6, 2))

输出结果为：

13

lambda 函数通常与内置函数如 map()、filter() 和 reduce() 一起使用，以便在集合上执行操作。例如：

实例

numbers = [1, 2, 3, 4, 5]

squared = list(map(lambda x: x\*\*2, numbers))

print(squared) # 输出: [1, 4, 9, 16, 25]

输出结果为：

[1, 4, 9, 16, 25]

**2.4 标准库**

**2.4.1 os**

OS（Operating System）是Python标准库中的一个模块，它提供了一种与**操作系统**进行交互的方式，可以进行**文件和目录操作、进程管理、环境变量访**问等。

**2.4.2 sys**

“sys”即“system”，“系统”之意。sys 模块主要负责与 **Python 解释器进行交互**，该模块提供了一系列用于控制 Python 运行环境的函数和变量。该模块提供了一些接口，用于访问 Python 解释器自身使用和维护的变量，同时模块中还提供了一部分函数，可以与解释器进行比较深度的交互。

**2.4.3 re**

Python的标准库re（正则表达式）模块提供了处理正则表达式的功能。它允许您在文本中进行模式匹配、搜索、替换和分割等操作。

下面是re模块中一些常用函数和方法的概述：

1. re.compile(pattern, flags=0)：将正则表达式模式编译为正则表达式对象，以便在后续的匹配操作中重复使用。
2. re.match(pattern, string, flags=0)：尝试从字符串的起始位置进行匹配。如果匹配成功，返回**一个**匹配对象；否则返回None。
3. re.search(pattern, string, flags=0)：在整个字符串中搜索，返回第一个符合要求的匹配。如果匹配成功，返回一个匹配对象；否则返回None。
4. re.findall(pattern, string, flags=0)：在整个字符串中搜索，并以列表形式返回满足要求的所有匹配。
5. re.sub(pattern, repl, string, count=0, flags=0)：使用替换字符串替换字符串中匹配模式的非重叠部分。
6. re.split(pattern, string, maxsplit=0, flags=0)：根据模式的匹配来分割字符串，并返回分割后的子串列表。

**- 案例**

PATTERN=r'(?:https?:\/\/)?(?:www\.)?(?:github|gitee|gitlab)\.com\/[\w\-]+\/[\w\-]+\/(?:pulls|pull|merge\_requests)\/[0-9]+'

+ ：匹配前面的字符一次或多次

?：匹配前面的字符出现0次或1次

\*：匹配前面的字符出现0次或多次

\w：匹配任意一个字母或数字字符

\s : 匹配任何空白字符

\d ： 匹配任何数字字符

(..)：匹配括号中的字符串

(?:) ：是一个非捕获组，表示匹配但不捕获括号中的内容

[...]：匹配方括号中的任意一个字符

|：或操作符

**2.5 高阶用法**

**2.5.1 类的继承**

首先提一下，经典类和新式类。在Python2中，如果定义类的方式是 class MyClass: 那么该类叫做经典类，如果定义类的方式为class MyClass(object): 那么该类为新式类。在Python3中，上面两种方式定义出来的类都叫新式类。本文是基于新式类来进行讲解的。

**单继承**

如果子类没有重写父类的方法，当调用该方法的时候，会调用父类的方法，当子类重写了父类的方法，默认是调用自身的方法。如果子类Son重写了父类Father的方法，如果想调用父类的action方法，可以利用super()

**代码示例：**

1. class FooParent(object):
2. def \_\_init\_\_(self):
3. self.parent = 'I\'m the parent.'
4. print ('Parent')
6. def bar(self,message):
7. print ("%s from Parent" % message)
9. class FooChild(FooParent):
10. def \_\_init\_\_(self):
11. super(FooChild,self).\_\_init\_\_()
12. print ('Child')
14. def bar(self,message):
15. super(FooChild, self).bar(message)
16. print ('Child bar fuction')
17. print (self.parent)
19. if \_\_name\_\_ == '\_\_main\_\_':
20. fooChild = FooChild()
21. fooChild.bar('HelloWorld')

**代码解读：**



**super()** 是用于调用父类（或父类的父类...）方法的 函数，主要用于单继承或者多继承。

super(FooChild,self) 首先找到 FooChild 的父类（就是类 FooParent），然后把FooChild 的类对象转换为FooParent 的类对象，可以调用FooParent类的方法和属性。

**多继承**

super() 和父类没有实质性的关联。如果想要搞懂super() 函数的运行原理，那一定要先搞懂 \_\_mro\_\_ 属性, \_\_mro\_\_ 是Method Resolution Order，中文方法解析顺序。单继承中super() 函数使用比较简单的原因 也是因为 \_\_mro\_\_ 比较简单，多继承的\_\_mro\_\_就稍微复杂了，总之 \_\_mro\_\_的目的就是按照一定顺序，保证父类的函数只调用一次。

1. #实例一：
2. **class** **A**(**object**):
3. **def** **\_\_init\_\_**(self):
4. print("class ---- A ----")
6. **class** **B**(**A**):
7. **def** **\_\_init\_\_**(self):
8. print("class ---- B ----")
9. super(B, self).\_\_init\_\_()
11. **class** **C**(**A**):
12. **def** **\_\_init\_\_**(self):
13. print("class ---- C ----")
14. super(C, self).\_\_init\_\_()
16. **class** **D**(B, C):
17. **def** **\_\_init\_\_**(self):
18. print("class ---- D ----")
19. super(D, self).\_\_init\_\_()
21. d = D()
22. '''
23. #输出结果：
24. class ---- D ----
25. class ---- B ----
26. class ---- C ----
27. class ---- A ----
28. '''



类D的\_\_mro\_\_顺序是 D -> B -> C -> A -> object。

新式类的MRO方法是采用 从左至右的深度优先遍历 的算法，**重复留后者。**

按照深度遍历，其顺序为 [D, B, A, object, C, A, object]，重复者只保留**后面**一个，因此变为 [D, B, C, A, object]。

根据\_\_mro\_\_顺序开始查找，实例一中类D中 调用super()函数时 传入的第一个参数是D，那么super() 函数就会在\_\_mro\_\_ 里从D的上一级开始查找，它的上一级是B, 那么super(D, self).\_\_init\_\_() 就调用B的\_\_init\_\_()函数，B的\_\_init\_\_()函数里又调用了B的super()函数，super(B, self).\_\_init\_\_()， 那就从B的上一级再开始查找，B的上一级是C, 以此类推，然后是A，最后是object。

#实例二： 更改一下类D的super函数：

**class** **D**(B, C):

def **\_\_init\_\_**(self):

**print**("class ---- D ----")

super(B, self).**\_\_init\_\_**()

d = **D**()

'''

#输出结果：

class ---- D ----

class ---- C ----

class ---- A ----

'''

实例二中 类D的super() 方法第一个参数传入的是 B ,那么根据\_\_mro\_\_顺序开始查找，B的上一级是C，C的上一级是A，所以实例二的打印顺序是 D - > C -> A

# 实例三： 再更改一下类D的super函数：

**class** **D**(B, C):

def **\_\_init\_\_**(self):

**print**("class ---- D ----")

super(C, self).**\_\_init\_\_**()

d = **D**()

'''

# 输出结果：

class ---- D ----

class ---- A ----

'''

实例三读者可以自行理解。

**类的私有属性、公有属性、私有方法、公有方法**

类的公有属性

public\_attrs：能在类的外部被使用或直接访问。在类内部的方法中使用时 public\_attrs\_attrs，在类的外部class\_name.public\_attrs。

类的私有属性

\_\_private\_attrs：两个下划线开头，声明该属性为私有，不能在类的外部被使用或直接访问。在类内部的方法中使用时 self.\_\_private\_attrs。

类的(公有)方法

在类的内部，使用 def 关键字来定义一个方法，与一般函数定义不同，类方法必须包含参数 self，且为第一个参数，self 代表的是类的实例。

self 的名字并不是规定死的，也可以使用 this，但是最好还是按照约定是用 self。

类的私有方法

\_\_private\_method：两个下划线开头，声明该方法为私有方法，只能在类的内部调用 ，不能在类的外部调用。self.\_\_private\_methods。

**2.5.2 装饰器**

Python装饰器（Decorators）是一种函数或类，用于修改其他函数或类的行为。装饰器提供了一种简洁的方式来对函数或类进行包装、扩展或修改，而无需修改它们的源代码。

在Python中，装饰器使用@符号紧跟在函数或类定义之前，并将装饰器函数或类应用于目标函数或类。装饰器可以接受目标函数或类作为参数，并返回一个新的函数或类，或者在内部修改目标函数或类。

（1）假设我们有一个原函数（目标函数）

（2）定义一个装饰器（是一个嵌套函数），使得原函数能够有处理异常的功能

（3）使用@符号+装饰器的名字来使用装饰器，可以在不改变原函数（divide）的情况下增加异常处理的功能

1. # 假设我有一个函数（原函数），可以实现两个数相除
2. def divide(x, y):
3. return x / y
5. res = divide(10, 2)
6. print(res)
8. # 定义一个装饰器，使得原函数能够有处理异常的功能
9. def handle\_exceptions(func):    # 外层函数：接受一个函数（原函数）作为参数，并返回一个函数（新函数）。
10. def wrapper(\*args, \*\*kwargs):  # 里层函数：接受和原函数相同的参数
11. try:
12. # 调用被装饰的函数
13. return func(\*args, \*\*kwargs)     # 将原来的函数包装在自己功能中，同时添加新的功能
14. except Exception as e:
15. # 处理异常的逻辑
16. # 返回异常信息
17. return e
18. return wrapper   # 新函数

21. # 使用@符号+装饰器的名字来使用装饰器，可以在不改变原函数（divide）的情况下增加异常处理的功能
22. @handle\_exceptions
23. def divide(x, y):    # 这是想要装饰的函数（原函数），即实现两个数相除
24. return x / y
26. # 也可以使用divide = handle\_exceptions(divide)来使用装饰器，可以在不改变原函数（divide）的情况下增加异常处理的功能
27. # divide = handle\_exceptions(divide)
29. res = divide(10, 0)
30. print(res)

**2.5.3 并发编程**

**2.5.3.1 引言**

多任务是指在同一时间内执行多个任务。例如: 现在电脑安装的操作系统都是多任务操作系统，你可以同时运行着多个软件，一边用浏览器上网，一边在听MP3，一边在用Word赶作业。**在代码里面，一个程序函数或方法就是一个任务，平时我们写的程序几乎都是单任务的，也就是说一个函数或者方法执行完成 , 另外一个函数或者方法才能执行。多任务就是多个程序函数或方法同时执行**。多任务计算密集型任务与I/O密集型任务：

（1）IO密集型，涉及到网络、磁盘IO的任务都是IO密集型任务，这类任务的特点是CPU消耗很少，任务的大部分时间都在等待IO操作完成（因为IO的速度远远低于CPU和内存的速度）。对于IO密集型任务，任务越多，CPU效率越高，但也有一个限度。常见的大部分任务都是IO密集型任务。

（2）CPU密集型任务的特点是要进行大量的计算，消耗CPU资源，比如计算圆周率、对视频进行高清解码等等，全靠CPU的运算能力。这种计算密集型任务虽然也可以用多任务完成，但是任务越多，花在任务切换的时间就越多，CPU执行任务的效率就越低，所以，要最高效地利用CPU，计算密集型任务同时进行的数量应当等于CPU的核心数。

**2.5.3.2多任务的两种表现形式：并行和并发**

**并行处理**是指计算机系统中能同时执行两个或多个任务的计算方法，并行处理可同时工作于同一程序的不同方面。

**并发处理**是同一时间段内有几个程序都在一个cpu中处于运行状态，但任一时刻只有一个程序在cpu上运行。

并发的重点在于有处理多个任务的能力，不一定要同时，交替执行（伪并行）；而并行的重点在于就是有同时处理多个任务的能力。并行是并发的子集。

以上所说的是相对于所有语言来说的，Python的特殊之处在于Python有一把GIL锁，这把锁限制了同一时间内一个进程只能有一个线程能使用cpu。

**2.5.3.3多任务的实现方式：并发编程**

**多任务协同处理**

多任务在代码里面，就是一个程序有多个地方同时执行，多任务分为并行和并发，并行（真的多任务）,比如：有两个任务要同时进行，而cpu内核是4核的时候，这两个任务就会真正的同时分配给其中两个内核来完成。当任务数量大于内核数量的时候就不能这么干了，因为cpu内核为n时最多同时运行n个任务。这时候就变成了并发（假的多任务）由于cpu处理的速度很快，它采用一种轮转的方式。一个任务执行一段时间后扔一边又去执行另一个任务。这样来回转，雨露均沾。以达到类似于同时进行的效果。

并发编程是实现多任务协同处理，改善系统性能的方式。Python中实现并发编程主要依靠一下三种方式：

**进程(Process)：**进程是计算机中的程序关于某数据集合的一次运行实例，是操作系统进行资源分配的最小单位，是操作系统进行资源分配和调度运行的基本单位，通俗理解：一个正在运行的程序就是一个进程。例如:正在运行的qq , 微信等 他们都是一个进程。一个程序运行后至少有一个进程。

**线程(Thread)：**线程被包含在进程之中，是操作系统进行程序调度执行的最小单位。实际上进程只负责分配资源 , 而利用这些资源执行程序的是线程 , 也就说进程是线程的容器 , 一个进程中最少有一个线程来负责执行程序 。同时线程自己不拥有系统资源，只需要一点儿在运行中必不可少的资源，但它可与同属一个进程的其它线程共享进程所拥有的全部资源 。这就像通过一个QQ软件(一个进程)打开两个窗口(两个线程)跟两个人聊天一样 , 实现多任务的同时也节省了资源。具体行为是由内核操作的，什么时候运行什么部分，这个部分运行多长时间，都不受用户控制。

**协程(Coroutine)：**协程是在程序层面操作的，可以控制遇到阻塞的时候程序该干什么，是用户态执行的轻量级编程模型，由单一线程内部发出控制信号进行调度，而非受到操作系统管理。协程拥有自己的寄存器上下文和栈。协程调度切换时，将寄存器上下文和栈保存到其他地方，在切回来的时候，恢复先前保存的寄存器上下文和栈，直接操作栈则基本没有内核切换的开销，可以不加锁的访问全局变量，所以上下文的切换非常快。

一个进程可以包含多个线程，一个线程可以包含多个协程。一个进程中最少有一个线程来负责执行程序，GIL锁限制了同一时间内一个进程只能有一个线程能使用cpu，因此python中的多线程不能真正同时运行，更适合处理 I/O 密集的场景。而相较于线程，协程没有切换开销和同步锁机制，处理 I/O 密集任务更优。

协程和线程更适合处理 I/O 密集的场景；而对于 CPU 密集的场景来说，多进程、多机器、多处理器才能提高程序的运行速度。

**2.5.3.4并发（编程）与异步（编程）**

**并发强调同时（或交替同时）执行多个任务的能力和机制**，并发编程是指在同一时间段内执行多个任务，这些任务可以是同时执行的（并行），也可以是交替执行的（并发），一般可以通过多线程、多进程来实现。并发编程狭义上指通过多进程和多线程方式实现多任务，广义上包括异步编程，也就是协程。

**异步强调多个任务执行的顺序**，指的是一个任务的完成不会阻塞其他任务的执行。异步编程是一种编程模型，它允许在等待某个操作完成的同时执行其他任务，而不会阻塞程序的执行。通常使用回调、协程、事件循环等机制来实现，它可以在等待 I/O 操作、计算任务等时释放 CPU 的控制权，执行其他任务。异步编程通常是并发编程的一种实现方式，通过异步操作可以实现并发执行多个任务的效果。

以代码实现下载 url\_list 中的3张图片为例：

* 同步编程，按照顺序逐一排队执行，如果图片下载时间为2分钟，那么全部执行完则需要6分钟。
* 异步编程，几乎同时发出了3个下载任务的请求（遇到IO请求自动切换去发送其他任务请求），如果图片下载时间为2分钟，那么全部执行完毕也大概需要2分钟左右就可以了。

**2.5.3.5 协程：**

* 协程：本质是一个异步函数。
* 协程对象：调用异步函数所返回的对象。
* 利用协程在IO等待时间就去切换执行其他任务，当IO操作结束后再自动回调，那么就会大大节省资源并提供性能，从而实现异步编程（不等待任务结束就可以去执行其他代码），因此协程在处理IO密集型任务方面非常高效。
* **asyncio库：**asyncio即Asynchronous I/O是python一个用来处理并发(concurrent)事件的包，是很多python异步架构的基础，多用于处理高并发网络请求方面的问题。简单来说，asyncio解决的是：IO阻塞导致cpu利用率降低的问题。
  + async：用于声明一个函数为异步函数，即该函数内部可能会使用到异步操作。
  + await：用于等待一个异步操作完成。当在一个异步函数中使用await关键字时，该函数会暂停执行，直到等待的异步操作完成并返回结果后，才继续执行后续代码。在特殊函数内部，凡是阻塞操作前都必须使用await进行修饰。异步程序执行到某一步时需要等待很长时间，就将此挂起，去执行其他的异步程序。
  + asyncio.run() ：运行协程

（1） 创建任务

task：任务，对协程对象的进一步封装,包含任务的各个状态。

task = asyncio.create\_task(func(参数列表))

（2） 创建多任务

tasks = [asyncio.create\_task(func(1)),

asyncio.create\_task(func(2)),

asyncio.create\_task(func(3))]

将协程当做任务添加到事件循环的任务列表，然后事件循环检测列表中的协程是否 已准备就绪（默认可理解为就绪状态），如果准备就绪则执行其内部代码。当前协程（任务）挂起过程中事件循环可以去执行其他的协程（任务），当前协程IO处理完成时，可以再次切换回来执行await之后的代码。

（3）并发执行多个协程（coroutine）的函数

asyncio.gather

tasks = [task1,task2,task3]

results = await asyncio.gather(\*tasks)

results = await asyncio.gather(task1,task2,task3)

asyncio.gather接收一个协程列表作为参数，并返回一个新的协程。当所有传入的协程都完成时，这个新的协程也会完成。

asyncio.gather返回一个新的协程，可以通过调用其result()方法获取所有协程的结果。（似乎直接返回所有已经完成的协程的结果）

asyncio.gather会等待所有传入的协程都完成，如果其中任何一个协程抛出异常，那么整个asyncio.gather协程都会抛出异常。

asyncio.wait

await asyncio.wait(tasks) #对协成tasks阻塞的结果收集，得到一个对象

#获取每个任务对象的结果值

for task in tasks:

print(task.done(),task.result())

asyncio.wait接收一个或多个协程以及一个超时时间作为参数。它会阻塞主线程，直到至少有一个协程完成或者超时。

asyncio.wait返回一个包含已完成协程对象的集合，可以通过遍历这个集合来获取每个协程的结果。

asyncio.wait只会等待至少有一个协程完成，如果其中任何一个协程抛出异常，那么asyncio.wait会立即返回已完成的协程对象集合，而不会继续等待其他协程完成。

总结来说，asyncio.gather适用于需要等待所有协程都完成的场景，而asyncio.wait适用于只需要等待至少一个协程完成的场景。

**import** asyncio

**import** time

# 子协程：用于执行一些耗时的操作，例如网络请求、IO 操作等。

**async** **def** **func**(i):

print(f"任务{i}启动")

**await** asyncio.sleep(i) # 模拟阻塞操作

print(f"任务{i}结束")

**return** i \* i # 返回每个任务的平方值

# 主协程：程序的起点，可以创建和管理其他子协程

**async** **def** **main\_1**():

start = time.time()

# 显式的创建协程对象任务，并进行传参

tasks = [asyncio.create\_task(func(1)),

asyncio.create\_task(func(2)),

asyncio.create\_task(func(3))]

res = **await** asyncio.gather(\*tasks) #使用gather直接收集协成对象的结果

print("cost timer:", time.time() - start) # 总耗时

print(res)

# 主协程：程序的起点，可以创建和管理其他子协程

**async** **def** **main\_2**():

start = time.time()

# 显式的创建协程对象任务，并进行传参

tasks = [asyncio.create\_task(func(1)),

asyncio.create\_task(func(2)),

asyncio.create\_task(func(3))]

**await** asyncio.wait(tasks)

print("cost timer:", time.time() - start) # 总耗时

# 获取每个任务对象的结果值

**for** task **in** tasks:

print(task.done(), task.result())

**if** \_\_name\_\_ == '\_\_main\_\_':

asyncio.run(main\_1())

asyncio.run(main\_2())

main\_1()执行结果如下，三个阻塞任务异步并发执行，执行时间约等于阻塞时长最长的那个任务。



main\_2()执行结果如下，三个阻塞任务异步并发执行，执行时间约等于阻塞时长最长的那个任务。



**2.5.3.2 Python中的线程**

在Python中，我们可以使用threading模块创建和管理线程。以下是一个创建线程的例子：

**import** threading

def **print\_numbers**():

**for** i **in** **range**(10):

**print**(i)

def **print\_letters**():

**for** letter **in** 'abcdefghij':

**print**(letter)

thread1 = threading.**Thread**(target=print\_numbers)

thread2 = threading.**Thread**(target=print\_letters)

thread1.**start**()

thread2.**start**()

thread1.**join**()

thread2.**join**()

在这个例子中，我们创建了两个线程：thread1和thread2。thread1的任务是打印数字0到9，thread2的任务是打印字母'a'到'j'。 我们使用threading.Thread类创建了两个线程对象，并指定了每个线程的目标函数。然后，我们调用了start方法来启动线程。 join方法用于等待线程结束。这是阻塞调用，也就是说，调用join方法的线程（在这个例子中是主线程）会被阻塞，直到被调用join方法的线程（在这个例子中是thread1和thread2）结束。

在多线程环境中，多个线程可能需要共享一些资源。例如，多个线程可能需要写入同一个文件或修改同一个内存位置。这可能会引发所谓的竞态条件，其中一个线程的操作可能会影响其他线程的结果。 为了避免竞态条件，我们需要进行线程同步。Python的threading模块提供了多种线程同步的工具，包括锁（Lock）、条件（Condition）、事件（Event）、信号量（Semaphore）等。以下是一个使用锁进行线程同步的例子：

**import** threading

lock = threading.Lock()

counter = 0

**def** **increase\_counter**():

**global** counter

**with** lock:

counter += 1

print(counter)

threads = []

**for** \_ **in** range(10):

thread = threading.Thread(target=increase\_counter)

threads.append(thread)

thread.start()

**for** thread **in** threads:

thread.join()

在这个例子中，我们创建了10个线程，每个线程都会增加共享变量counter的值。为了避免竞态条件，我们使用了一个锁来保护对counter的修改。只有获得了锁的线程才能修改counter，其他试图修改counter的线程将被阻塞，直到锁被释放。 with lock:语句用于自动获取和释放锁。当进入with语句时，锁被自动获取；当退出with语句时，锁被自动释放。这确保了即使在发生异常的情况下，锁也能被正确释放。

需要注意的是，由于Python的全局解释器锁（GIL）的存在，Python的线程并不能实现真正的并行计算。全局解释器锁是Python解释器级别的锁，用于同步Python字节码的执行。由于这个锁的存在，同一时间只有一个线程能够执行Python字节码。这意味着在计算密集型任务中，多线程可能并不会带来速度的提升。 然而，在I/O密集型任务中，多线程仍然可以有效地提高程序的性能。当一个线程在等待I/O操作完成（例如读取文件或网络请求）时，其他的线程可以继续执行，从而提高了程序的整体效率。

Python的多线程是一种强大的工具，可以帮助我们编写出更高效的程序。然而，与所有并发编程一样，多线程编程也有其复杂性。为了避免竞态条件和其他并发问题，我们需要仔细地设计我们的程序，并正确地使用线程同步工具。

**2.6 常用的语法糖**

Python 中的语法糖是指一些特殊的语法结构，它们并没有引入新的功能，但能够使代码更加简洁易读。

**2.6.1with 语句：**

with open("file.txt", "r") as file:

    data = file.read()

这个语法糖可以自动管理文件等资源的打开和关闭，无需手动调用 file.close()。

上下文管理器类：

class MyContextManager:

    def \_\_enter\_\_(self):

        # 在进入 with 代码块前执行的操作

        return self

    def \_\_exit\_\_(self, exc\_type, exc\_val, exc\_tb):

        # 在离开 with 代码块后执行的操作

        pass

with MyContextManager() as obj:

    # 在 with 代码块中可以安全地使用 obj 对象

    pass

这个语法糖可以通过定义 \_\_enter\_\_() 和 \_\_exit\_\_() 方法，实现自己的上下文管理器，用于资源的安全管理。

with 语句的扩展：

from contextlib import contextmanager

@contextmanager

def open\_file(filename):

file = open(filename)

yield file

file.close()

with open\_file("file.txt") as file:

data = file.read()

这个语法糖可以通过使用 contextlib 模块中的 @contextmanager 装饰器将函数转换为上下文管理器，用于创建自定义的上下文管理器。

**2.6.2try-except 块与异常处理：**

try:

    # 可能引发异常的代码块

    pass

except SomeException:

    # 处理某个特定异常

    pass

except AnotherException:

    # 处理另一个特定异常

    pass

else:

    # 当没有引发任何异常时执行

    pass

finally:

    # 无论是否引发异常都会执行

    pass

**2.6.3 可变参数与关键字参数**

def func(\*args, \*\*kwargs):

    # 处理可变位置参数 args

    # 处理可变关键字参数 kwargs

    pass

func(1, 2, name="Alice", age=25)

这个语法糖可以在定义函数时接受可变数量的位置参数和关键字参数，方便灵活的函数调用。

具体用途，是看用\*或者\*\*的场合是在**函数定义**时还是**函数调用**时。

如果在定义函数时，\*代表收集参数，\*\*代表收集关键字参数

def test1(x,\*args):   
      print("x: ",x)  
 print(args)

  for i in args:

        print("i=",i)

test1(1,2,3,4,5,6,7)  

x:  1  
(2, 3, 4, 5, 6, 7)  
i= 2  
i= 3  
i= 4  
i= 5  
i= 6  
i= 7

def test2(x,\*\*kwargs):  
    print("x: ",x)  
    print(kwargs)

test2("Information",name="Peter",age=20, score=100)

x:  Information  
{'name': 'Peter', 'age': 20, 'score': 100}

如果在调用函数时，\*和\*\*都是分配参数用的，\*表示分配元组中的元素，元组中元素的个数和函数参数的个数要匹配，\*\*表示分配字典中的元素，字典中的键名和函数的参数名字要保持一致。

例子：

def test5(x,y):  
    print("x: ", x)  
    print("y: ", y)  
调用：

paramters = (1,2)    
test5(\*paramters)  
输出：

x:  1  
y:  2

def test6(x,y):  
    print("x: ", x)  
    print("y: ", y)

调用：

paramters = {"x":10,"y":20}  
test6(\*\*paramters)  
输出：

x:  10  
y:  20

**2.7 编码规范**

- 每行代码不要超过79个字符，可以使用括号或反斜杠来换行。

- 在二元运算符两侧、逗号后面、冒号后面等需要的地方添加空格，但是在括号内部不需要添加空格。当'='用于指示关键字参数或默认参数值时，不要在其两侧使用空格。函数的参数列表中，逗号之后要有空格。

- **常量名**：常量命名使用大写字母，下划线分隔。

- **函数名**：应该小写，如果有多个单词，应该用下划线连接。私有函数在函数前加一个下划线。

- **类名**：所有单词首字母大写，多个单词时，每个单词开头字母都要大写，私有类可用一个下划线开头。（例如MyClass）。

- **模块名**：应该短小，并且全部小写，如果有多个单词，应该用下划线连接。

- 函数和类的定义应该用两个空行隔开，而类的方法定义应该用一个空行隔开。

- 在注释中，应该遵循一定的格式，例如在#号后面添加一个空格。

- 如果使用了import语句，应该将标准库模块放在第一组，第三方库模块放在第二组，自己编写的模块放在第三组。

- 尽量避免使用单字符变量名，除非是在循环中使用。

Python类型注解必备利器：typing模块

typing模块为Python带来了类型提示和类型检查的能力。它允许开发者在代码中添加类型注解，提高代码的可读性和可维护性。尽管Python是一种动态类型语言，但类型注解能让开发者更清晰地了解函数和变量的预期类型。

**typing模块的作用:**

类型检查，防止运行时出现参数和返回值类型不符合。

作为开发文档附加说明，方便使用者调用时传入和返回参数类型。

该模块加入后并不会影响程序的运行，不会报正式的错误，只有提醒。

1. **class** **Solution**:
2. **def** **twoSum**(self, nums: List[int], target: int) -> List[int]:
3. hashtable = dict()
4. **for** i, num **in** enumerate(nums):
5. **if** target - num **in** hashtable:
6. **return** [hashtable[target-num], i]
7. hashtable[num] = i
8. **return** []
9. **Python数据分析**

**3.1 数据概述**

**3.1.1什么是数据？**

入门数据学科，首先第一步要认识数据什么，可能大多数人都无法对数据做一个准确的定义，在我们印象中，提到数据首先头脑浮现的是数据表格，是一堆堆数字，那么数据就是数字吗？显然二者不能完全划等号，我们来看一下度娘对数据定义的解读。

数据（data）是事实或观察的结果，是对客观事物的逻辑归纳，是用于表示客观事物的未经加工的原始素材。

数据是信息的表现形式和载体，可以是文字、符号、数字、语音、图像、视频等。数据和信息是不可分离的，数据是信息的表达，信息是数据的内涵，数据本身没有意义，数据只有对实体行为产生影响时才成为信息。

数据可以是连续的值，比如声音、图像，称为模拟数据（连续数据）。也可以是离散的，如符号、文字，称为数字数据（离散数据）。

在计算机系统中，数据以二进制信息单元0、1的形式表示。

通过以上解读，我们可以发现，我们印象中的数字只是数据的一种形式，广义的数据包含的内容很多，文字，语音，图像，视频都是数据定义的范畴，因为无论哪种形式，它们到计算机里面都是0和1，只是编码的形式有差别，这样说来，数据挖据，计算机视觉，自然语言处理等都是数据科学的范畴。

**3.1.2什么是数据类型？**

在学任何一门编程语言的开始，我们都要学习相关的数据类型，那么数据类型究竟又是如何定义的呢？

数据类型在数据结构中的定义是一组性质相同的值的集合以及定义在这个值集合上的一组操作的总称。

变量是用来存储值的所在处，它们有名字和数据类型。变量的数据类型决定了如何将代表这些值的位存储到计算机的内存中。在声明变量时也可指定它的数据类型，所有变量都具有数据类型，以决定能够存储哪种数据。

在学python语言时，有一个问题一直困扰着我，列表元组这些到底是数据类型还是数据结构呢？后来读了一些博客，发现数据类型主要包括原子类型和结构类型：

原子类型是指一种值的集合以及定义在值上的一组操作，比如在python中，有整数型(int)，浮点数型(float)，布尔型(bool)，字符串(str)等，这是我们印象中的常规表现形式；

结构类型（复合类型）是指一种数据结构以及定义在结构上的一种操作，是数据类型的另一种表现形式，是通过原子类型封装的更复杂的数据类型，比如在面向对象的编程语言python中，列表(List)，字典(Dict)，元组(Tuple)等都是这种类型。

**3.1.3什么是数据结构？**

从我的困惑可以看出，数据类型和数据结构之间存在联系但也有一定的区别。

数据结构是指相互之间存在着一种或多种关系的数据元素的集合和该集合中数据元素之间的关系组成 。包括逻辑结构和物理结构。

逻辑结构描述数据之间的相互关系，按照逻辑结构划分，主要分为线性结构和非线性结构，线性结构数据元素是一对一的关系，非线性结构数据元素是一对多和多对多的关系。

常见的线性结构有数组(Array)、链表(Linked List)、栈(Stack)、队列(Queue)，常见的非线性结构有树(Tree)，堆(Heap)，图(Graph)，散列表(Hash)。

物理结构描述数据具体在内存中的存储（如：顺序结构、链式结构、索引结构、哈希结构）等。

最后需要指出的是，数据类型和数据结构是基于某种编程语言的，不同的计算机语言（C、python、C#等）它们的数据类型和数据结构略有不同，但本质差异不大。

入门数据科学，第一步就是认识数据，了解数据类型和数据结构，数据类型和数据结构是依赖于编程语言的，对数据类型和数据结构有充分的认识，对以后数据分析和挖掘非常有帮助。

**3.2 数据读取**

**3.2.1爬虫技术**

通过编写程序模拟人类浏览网页的行为，从网页上抓取数据。爬虫技术需要一定的编程能力和网络知识，但可以采集各种类型的数据，包括结构化数据和非结构化数据。

**3.2.2API接口调用**

许多网站和应用程序提供API接口，可以通过编程调用API接口获取数据。API接口调用需要一定的编程能力和对API接口的理解，但可以获得高质量的数据，且通常比爬虫技术更加稳定和可靠。

**3.2.3数据库查询**

许多数据存储在数据库中，可以通过查询数据库获取数据。数据库查询需要一定的SQL语言知识和对数据库的理解，但可以获得高质量的结构化数据。

**3.2.4文件导入**

许多数据存储在各种类型的文件中，可以通过导入文件的方式获取数据。文件导入需要对文件格式的理解和相应的工具支持，但可以获得各种类型的数据，包括结构化数据和非结构化数据。

导入数据

pd.read\_csv(filename)：从CSV文件导入数据

pd.read\_table(filename)：从限定分隔符的文本文件导入数据

pd.read\_excel(filename)：从Excel文件导入数据

pd.read\_sql(query, connection\_object)：从SQL表/库导入数据

pd.read\_json(json\_string)：从JSON格式的字符串导入数据

pd.read\_html(url)：解析URL、字符串或者HTML文件，抽取其中的tables表格

pd.read\_clipboard()：从你的粘贴板获取内容，并传给read\_table()

导出数据

df.to\_csv(filename)：导出数据到CSV文件

df.to\_excel(filename)：导出数据到Excel文件

df.to\_sql(table\_name, connection\_object)：导出数据到SQL表

df.to\_json(filename)：以Json格式导出数据到文本文件

**3.2.5传感器技术**

许多物理量和环境数据可以通过传感器等硬件设备采集。传感器技术需要相应的硬件设备和技术支持，但可以获得实时的、高精度的数据。

不同的数据采集方式适用于不同的数据来源和采集目的。在选择数据采集方式时，需要考虑数据来源、数据类型、数据质量、采集效率和采集成本等因素。

**3.1 数据创建与插入**

创建dataframe的几种方式

**3.2 数据删除**

**3.2.1 按列条件删除行**

**3.3 数据遍历**

**3.3.1 逐行遍历**

**3.3.2 逐列遍历**

**3.3 数据修改**

**3.3.1 替换**

**3.3.2 索引重置**

**df = df.reset\_index(drop=True)**

**3.3.3 调整字段的顺序**

**3.3.4 修改字段名称**

**3.3 数据排序**

**3.3 数据去重**

**3.3 数据扩充**

**3.3.1 行扩充**

**3.3.2 列扩充**

**3.4 数据筛选（行）**

**3.4.1 切片**

**3.4.2 精确匹配**

**3.4.3 模糊匹配**

**3.6 数据分组聚合**

**3.6.1 分组求和**

**3.6.2 分组计数**

**3.7 数据合并**

包含pandas对象的数据可以通过多种方式是连接在一起，主要分为两类：

1）第一类：将两个pandas表根据一个或者多个键（列）值进行连接。这种操作类似关系数据库中sql语句的连接操作。这一类操作在使用pandas的merge、join操作来实现。

2）第二类：将两个pandas表在轴向上（水平、或者垂直方向上）进行粘合或者堆叠。这一类操作在使用pandas的concat、append操作来实现。

**3.7.1 Merge**

**3.7.2 Join**

**3.7.3 Concat**

**3.7.4 Append**

**3.8 数据可视化**

1. **Python特征工程**

特征工程是一门将原始数据转换为有意义的变量并输入模型的艺术。它涉及选择、转换和创建新特征，为学习算法提供相关信息。领域知识在此过程中起着至关重要的作用，因为它指导最能指示目标变量的特征的选择。

**4.1 缺失值处理**

数据缺失的典型机制：

* 完全随机丢失（MCAR，Missing Completely at Random）：某个变量是否缺失与它自身的值无关，也与其他任何变量的值无关。例如，由于测量设备出故障导致某些值缺失。
* 随机丢失（MAR，Missing at Random）：在控制了其他变量已观测到的值后，某个变量是否缺失与它自身的值无关。例如，人们是否透露收入可能与性别、教育程度、职业等因素有关系。
* 非随机丢失（MNAR，Missing not at Random）：数据的缺失与不完全变量自身的取值有关。分为两种情况：缺失值取决于其假设值（例如，高收入人群通常不希望在调查中透露他们的收入）；或者，缺失值取决于其他变量值（假设女性通常不想透露她们的年龄，则这里年龄变量缺失值受性别变量的影响）。

在前两种情况下可以根据其出现情况删除缺失值的数据，同时，随机缺失可以通过已知变量对缺失值进行估计。在第三种情况下，删除包含缺失值的数据可能会导致模型出现偏差，同时，对数据进行填充也需要格外谨慎。

缺失值的处理方法共分为三种：删除、插补、不做处理。其中删除实现比较简单，很多模型对于缺失值有容忍度或灵活的处理方法，因此在预处理阶段可以不做处理。插补的方法有很多，可以分为统计学法和模型法。

统计法：

* 具体值填补（如0或其他常数）：直接用一个常数替换所有的缺失值，适用于缺失值不多且对具体值有确切预期的情况。
* 均值填补：用列的均值来填补缺失值，适用于数值型数据且数据分布较为对称的情况。
* 中位数填补：用列的中位数来填补缺失值，适合处理有偏的数据集。
* 众数填补：用列中最频繁出现的值来填补缺失值，适用于分类数据。
* 前向填补（Forward Fill）：用前一个非空值来填补缺失值，适用于时间序列数据或有序数据。
* 后向填补（Backward Fill）：用后一个非空值来填补缺失值，同样适用于时间序列数据或有序数据。

模型法：

* 聚类填充：最为典型的代表是K均值（K-means clustering），先根据欧式距离或相关分析来确定距离具有缺失数据样本最近的K个样本，将这K个值加权平均来估计该样本的缺失数据。
* 监督学习模型预测填充

df.isnull().sum()

df.isnull().any()

np.isnan(df).any()

1.删除

删除所有缺失值的行 df.dropna()

删除所有缺失值的列 df.dropna(axis = ‘columns’)/df.dropna(axis = 1)

删除带有nan的行df.dropna(how = ‘nan’)

删除所有值都缺失的行 df.dropna(how = ‘all’)

删除至少有两个缺失值的行 df.dropna(thresh = 2)

删除指定的列范围 df.dropna(subset = [‘B’,‘D’])

删除指定列的缺失值 df.col.dropna()

使删除的结果生效 df.dropna(inplace = True)

2.填充

填充固定值（常数） df.fillna(100)

填充固定值（字典） df.fillna({0:10,1:20,2:30})

使填充的结果生效 df.fillna(0,inplace=True)

用前一个非缺失值填充 df.fillna(method=‘ffill/pad’)

用后一个非缺失值填充 df.fillna(method=‘bfill/backfill’)

指定填充两个个数值 df.fillna(method=‘bfill’, limit=2)

按行填充（axis=1）一个值 df.fillna(method=“ffill”, limit=1, axis=1)

4.2 无穷值处理

train\_inf = np.isinf(df)

train[train\_inf] = 0

**4.2 异常值处理**

**4.3 重复值处理**

**4.4 标准化**

**4.5 特征筛选（特征重要性分析）**

**4.6 特征构造**

1. **Python机器学习**

机器学习（Machine Learning）是一种人工智能（Artificial Intelligence）的基础，它研究和开发使计算机能够自动学习和改进的算法和模型。机器学习算法通过从大量的训练数据中学习，自动发现数据中的模式和规律，并使用这些学习到的知识来进行预测和决策。机器学习算法的分类有多种，本文将其分为以下几个主要类型，并对该类型下常见的算法做一个简单的应用展示。

机器学习是对研究问题进行模型假设，利用计算机从训练数据中学习得到模型参数，并最终对数据进行预测和分析的一门学科。

研究问题---模型假设（函数）---训练得到参数---预测分析



sklearn安装：

pip install scikit-learn

**5.1 有监督模型**

有监督学习万能模板V1.0



有监督学习万能模板V2.0



有监督学习万能模板V3.0



**5.1.1 单模型**

**5.1.1.1线性模型**

- 线性回归（只能回归任务）

- 逻辑回归（只能分类任务）

- Lasso

- Ridge

- LDA

**5.1.1.2决策树模型**

用来学习用于决策的树，是一种逼近离散值目标函数的方法，学习到的函数被表示为一颗决策树。学习过程是递归过程，关键在于如何选择最佳划分属性。决策树的目的就是为了构造一棵泛化能力强，即处理待测样本能力强的决策树。具体算法采用遵循自顶向下、分而治之的策略。

**5.1.1.3支持向量机模型**

样本空间，寻找最大边缘超平面，作为决策边界

**5.1.1.4 K近邻模型**

给定一个待测样本，根据近邻索引算法，找出训练集中与之最靠近的k个样本，基于这K个样本的后验概率来预测待测样本的类标记

**5.1.1.5朴素贝叶斯（Naive Bayes）：只能用于分类任务**

概率计算，贝叶斯定理

**5.1.1.5多层感知机模型（放在下一章解释）**

**5.1.2 集成学习模型**

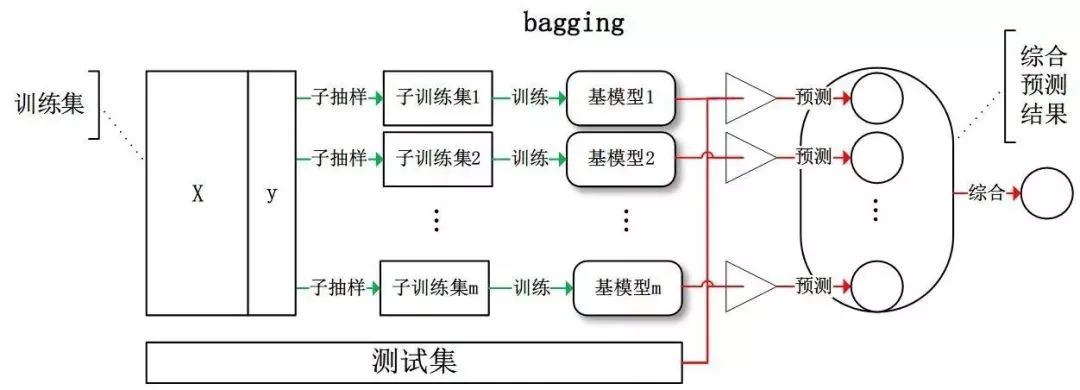


* Boosting模型
  + AdaBoost
    - 基于KNN
    - 基于决策树
  + GBDT
    - Xgbboost
    - LightGBM
* Bagging模型
  + 随机森林

集成学习先生成好而不同的的个体学习器，采用某种策略结合生成的这些个体学习器的结果

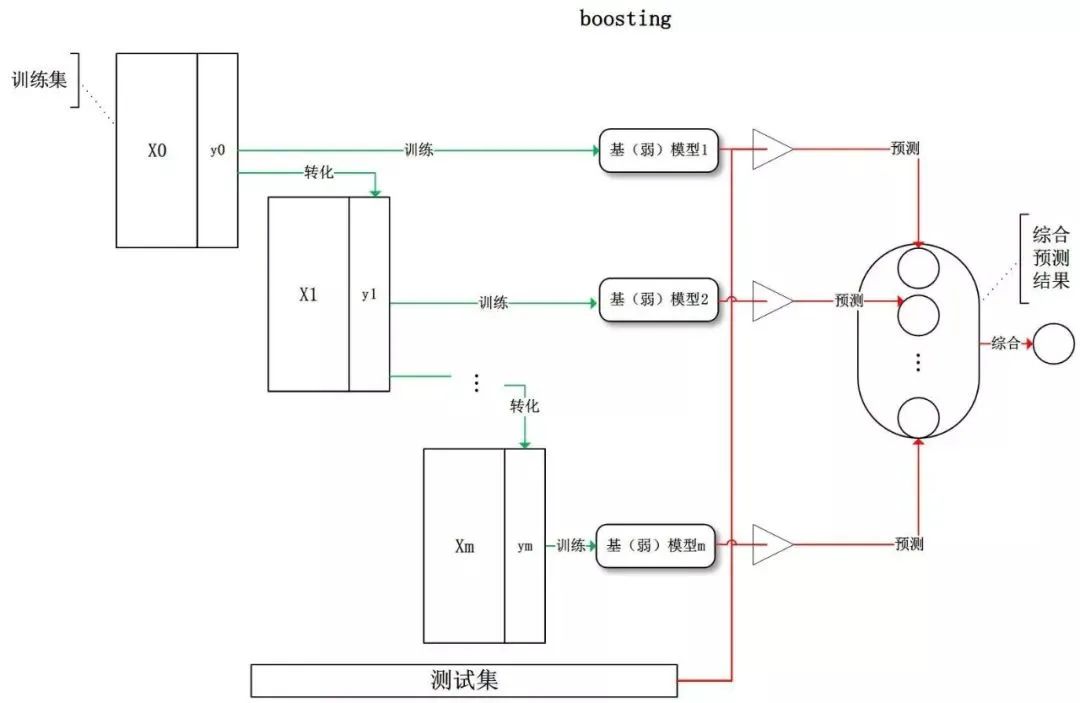
**1.1 Bagging**

Bagging 全称叫 Bootstrap aggregating，看到 Bootstrap 我们立刻想到著名的开源前端框架（抖个机灵，是 Bootstrap 抽样方法） ，每个基学习器都会对训练集进行有放回抽样得到子训练集，比较著名的采样法为 0.632 自助法。每个基学习器基于不同子训练集进行训练，并综合所有基学习器的预测值得到最终的预测结果。Bagging 常用的综合方法是投票法，票数最多的类别为预测类别。



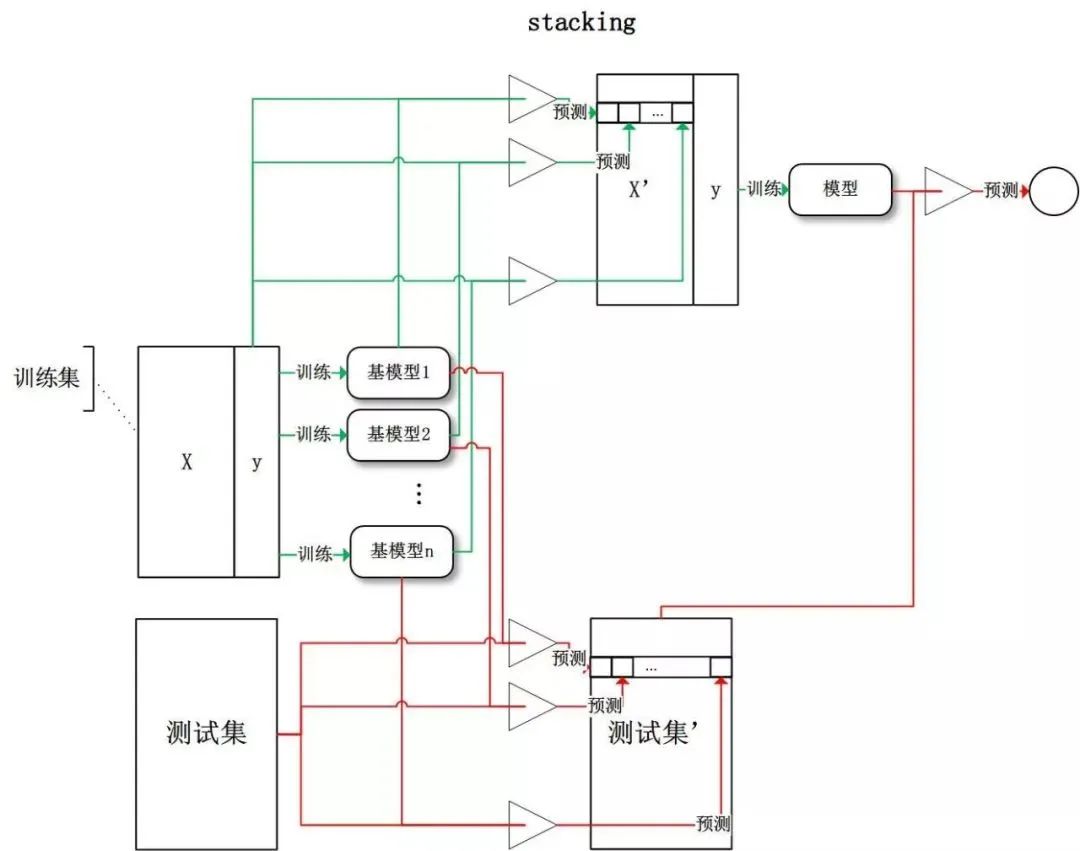
**1.2 Boosting**

Boosting 训练过程为阶梯状，基模型的训练是有顺序的，每个基模型都会在前一个基模型学习的基础上进行学习，最终综合所有基模型的预测值产生最终的预测结果，用的比较多的综合方式为加权法。



**1.3 Stacking**

Stacking 是先用全部数据训练好基模型，然后每个基模型都对每个训练样本进行的预测，其预测值将作为训练样本的特征值，最终会得到新的训练样本，然后基于新的训练样本进行训练得到模型，然后得到最终预测结果。



那么，为什么集成学习会好于单个学习器呢？原因可能有三：

1. 训练样本可能无法选择出最好的单个学习器，由于没法选择出最好的学习器，所以干脆结合起来一起用；
2. 假设能找到最好的学习器，但由于算法运算的限制无法找到最优解，只能找到次优解，采用集成学习可以弥补算法的不足；
3. 可能算法无法得到最优解，而集成学习能够得到近似解。比如说最优解是一条对角线，而单个决策树得到的结果只能是平行于坐标轴的，但是集成学习可以去拟合这条对角线。

**5.2 无监督模型**

**5.3 概率模型**

1. **Python深度学习**

**6.1 深度学习模型软件体系架构**



* 模型

模型包括结构和计算过程（学习范式）两个内容，训练完成，本质是一个文件

* 训练优化库

deepspeed作为一个大模型训练加速库，位于模型训练框架和模型之间，用来提升训练、推理等。

* 训练框架

tensorflow、pytorch，用来构建模型结构和配置模型计算过程

* 训练基础设施

训练平台，比如本地电脑，云服务器、

* 软硬件支撑（操作系统、显卡）

内存、显卡、显存

**6.1 深度学习训练框架：Tensorflow、Pytorch、Keras**

1、Tensorflow更倾向于工业应用领域，适合深度学习和人工智能领域的开发者进行使用，具有强大的移植性。

pip install tensorflow

import tensorflow as tf

2、Keras因为是在Tensorflow的基础上再次封装的，所以运行速度肯定是没有Tensorflow快的；但其代码更容易理解，容易上手，用户友好性较强。目前Keras框架已经被集成到Tensorflow里了，在TensorFlow 2.0及其之后的版本中，Keras已经成为TensorFlow的默认高级API，使得用户可以更加方便地使用Keras构建、训练和评估深度学习模型。

import keras

在tensorflow2.6.0版本以上的话已经封装好了keras库，只需要对应好python的版本即可，2.6.0以上的只支持3.7版本以上的python。同时封装了CPU和GPU版本，使用起来十分方便。

3、Pytorch更倾向于科研领域，语法相对简便，利用动态图计算，开发周期通常会比Tensorflow短一些。

import torch

4、Tensorflow、Pytorch GPU版本

pip/conda install tensorflow-gpu=版本号，在安装 TensorFlow 2.x 及更新版本时，您不再需要单独安装 CPU 版本或 GPU 版本的 TensorFlow。现在，安装 TensorFlow 时会同时安装 CPU 和 GPU 支持，但最终使用哪种版本取决于您的系统环境和 TensorFlow 库中是否存在 GPU 支持的驱动程序。

如果您的系统中有适当的 NVIDIA GPU 并安装了 CUDA 和 cuDNN，TensorFlow 将自动检测到 GPU，并在必要时使用 GPU 进行加速。否则，它将在 CPU 上运行。

**在 Windows 原生平台上，GPU 不支持 2.10 以上的版本。也就是说，你在windows上安装2.10以上的tensorflow是没办法用GPU的。所以要想在windows上使用GPU，必须安装tensorflow的版本在2.10以及以下。**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | CUDA | CUDNN | **Python** | **tensorflow/** tensorflow-gpu | **Keras** |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

检查 TensorFlow 是否在使用 GPU 进行加速，可以执行以下代码：

import tensorflow as tf  
print(tf.test.is\_built\_with\_cuda())  
print(tf.config.list\_physical\_devices('GPU'))



确定 PyTorch、CUDA 和显卡驱动的版本并确保它们兼容，可以按照以下步骤进行：

**「确定显卡驱动版本」**：

* 在终端中执行 nvidia-smi 命令。这个命令会显示当前系统上的 NVIDIA 显卡驱动版本以及相关信息。
* 首先，你需要确定你的计算机上安装了哪个版本的 NVIDIA 显卡驱动。你可以使用以下方法来查看：
* 记下显示的 NVIDIA 驱动版本号。例如，版本号可能类似于 465.19.01。

**「确定 CUDA 版本」**：

* 通常，NVIDIA 显卡驱动与 CUDA 版本一起安装。所以，你可以通过查看 CUDA 的版本来确定。
* 在终端中执行以下命令来查看 CUDA 版本：nvcc --version
* 记下显示的 CUDA 版本号。例如，版本号可能类似于 11.1。

**「确定 PyTorch 版本」**：

* 使用以下 Python 代码来查看 PyTorch 的版本：import torchprint(torch.\_\_version\_\_)
* 记下显示的 PyTorch 版本号。例如，版本号可能类似于 1.8.1。

**「检查兼容性」**：

* 一旦你确定了各个组件的版本号，你可以查阅 PyTorch 的官方文档，了解哪个版本的 PyTorch 与哪个版本的 CUDA 和显卡驱动兼容。通常，PyTorch 的文档会明确说明支持的 CUDA 版本范围。
* 如果你的 PyTorch 版本与你的 CUDA 版本和显卡驱动版本不兼容，你可能需要升级或降级其中一个或多个组件，以确保它们能够良好地协同工作。

❝往往我们在实际项目时，起始首先确定的是PyTorch的版本，进而确定CUDA的版本，再根据CUDA的版本去查看自己平台的驱动是否支持。❞

具体步骤：

* 更新并确认GPU的显卡驱动
* 根据驱动版本，确认需要下载的CUDA版本
* 下载并安装CUDA
* 根据CUDA版本，确认需要下载的cudnn版本
* 将下载的cudnn文件解压，将每个文件夹中的所有文件复制到CUDA安装目录的对应位置即可。默认是：C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v11.0\
* 安装GPU版的Pytorch，前往Pytorch主页（https://pytorch.org/），选择相应的平台和CUDA版本，然后复制命令，下载并安装Pytorch即可。

如果你使用的是 CUDA 11.7 版本。如果你安装的是 PyTorch 2.3.0 并且它是带有 CUDA 11.8（即 **torch2.3.0+cu118**）的版本，那么理论上是有可能存在一些不兼容的问题，因为你的本地 CUDA 版本是 11.7，而你安装的 PyTorch 是绑定到 CUDA 11.8 的。

但从实践的角度出发，PyTorch 包含了 CUDA 的二进制文件（即 **cu118**），因此在大部分情况下，你不一定需要依赖本地安装的 CUDA 版本。PyTorch 会使用它自己打包的 CUDA 版本，只要驱动程序满足要求，就可以正常工作。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | CUDA | CUDNN | **Python** | pytorch | torchvision |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

import torch

print(torch.\_\_version\_\_)

print(torch.version.cuda) *# 查看对应CUDA的版本号*

print(torch.cuda.is\_available()) *# 查看pytorch是否支持CUDA*

如果你使用的是 PyTorch 1.8.0，官方文档可能会明确指出支持 CUDA 11.1，因此你需要安装 CUDA 11.1 或兼容版本的 CUDA 驱动来与 PyTorch 1.8.0 一起使用。

打开cmd看看自己安装的cuda版本

nvcc --version

**6.1 深度学习模型**

**6.1 模型架构**

**编码器和解码器架构（**Encoder- Decoder**）**

- 编码器（Encoder）：负责将输入（Input）转化为特征（Feature）

- 解码器（Decoder）：负责将特征（Feature）转化为目标（Target）

1 编码器和解码器同时接受输入：transform架构





2 仅编码器接收输入，解码器不接收输入：CNN架构



如果不把softmax当成decoder，CNN就是一个编码器（encoder-only）架构

要理解神经网络，就必须理解层的结构和运算（输入-输出），它是神经网络模型的基础模块，比如常见的层有全连接层、卷积层、池化层、LSTM层、RNN层等，不同类型的层模块对应着不同的运算方式，比如全连接层是一种计算方式，卷积层是一个计算方式。

**6.1 多层感知机**

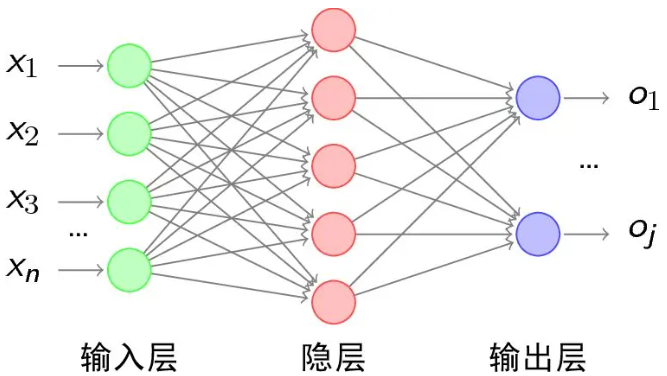
模型原理：深度神经网络（DNN）是一种构建于多层隐藏层之上的神经网络。每一层都扮演着信息的传递者和加工者的角色，通过非线性激活函数将输入数据转换为更具表现力的特征表示。正是这些连续的非线性变换，使得DNN能够捕捉到输入数据的深层次、复杂特征。

模型训练：DNN的权重更新主要依赖于反向传播算法和梯度下降优化算法。在训练过程中，通过计算损失函数关于权重的梯度，再利用梯度下降或其他优化策略，逐步调整权重值，以达到最小化损失函数的目的。

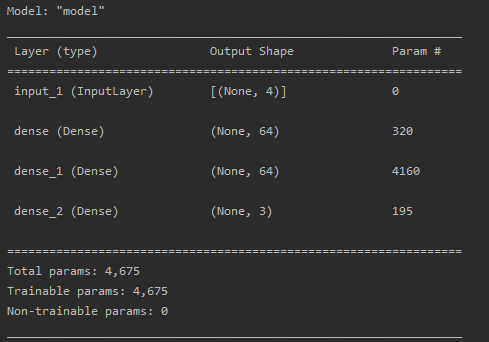
优点：DNN凭借其强大的特征学习和表示能力，能够有效学习输入数据的复杂特征，并精确捕捉非线性关系，使其在各种任务中表现出色。

缺点：然而，随着网络层数的增加，梯度消失问题逐渐凸显，这可能导致训练过程的不稳定。此外，DNN容易陷入局部最小值，从而限制了其性能，通常需要复杂的初始化策略和正则化技术来应对这些问题。

使用场景：DNN在多个领域有着广泛的应用，包括图像分类、语音识别、自然语言处理以及推荐系统等。



input = Input(shape=(4,))  
hidden1 = Dense(units=64, activation='relu')(input)  
hidden2 = Dense(units=64, activation='relu')(hidden1)  
output = Dense(units=3, activation='softmax')(hidden2)  
model = Model(inputs=input, outputs=output)



结构如下：

输入：4个神经元

第一个隐含层：64个神经元

第二个隐含层：64个神经元

输出层：3个神经元

第一个隐含层是一个全连接层，参数数量：

320=units\_now \* (input\_dim+1)=64\*(4+1)

第二个隐含层是一个全连接层，参数数量：

4160= units\_now \* (units\_previous+1)=64\*(64+1)

输出层是一个全连接层，参数数量：

195=units\_now \* (units\_previous +1)=3\*(64+1)

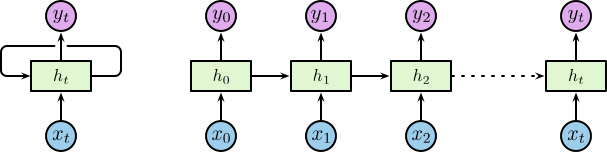
**案例：结构化表格数据分类**

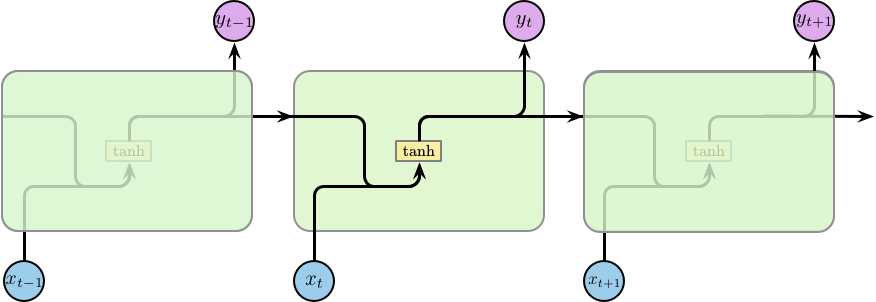
**6.2 卷积神经网络CNN**

**案例：图像识别**

**6.3 循环神经网络RNN/LSTM**

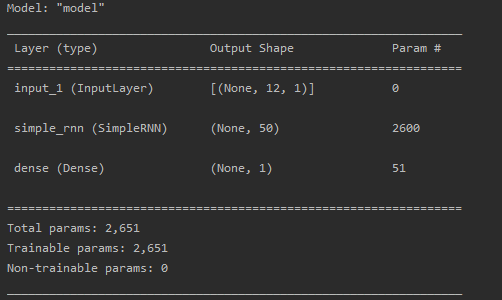
RNN：





模型架构代码实现：

input = Input(shape=(window\_size, 1))  
lstm = SimpleRNN(units=50)(input)  
output = Dense(units=1)(lstm)  
model = Model(inputs=input, outputs=output)  
model.summary()



从参数角度去理解模型的架构和计算过程：

第一层是一个LSTM层，参数数量：input\_units \* output\_units + biases

2600=50\*（50+1+1）=units\_now\*（units\_now+input\_dim+1）

= units\_now\*units\_now+ units\_now \*(input\_dim+1）

这部分权重参数分为两部分，一部分是当前输入到当前隐含层的权重（是一个全连接：units\_now \*(input\_dim+1）），一部分是上一时刻隐含层到当前时刻隐含层的权重（是一个全连接：units\_now\*units\_now）。

输出层是一个全连接层，参数数量：

51=1\*(50+1)= units\_now \*( units\_previous +1）

在处理序列数据时，传统的循环神经网络（RNN）面临着梯度消失和模型退化等问题，这限制了网络的深度和性能。为了解决这些问题，LSTM被提出。

模型原理：

LSTM借助创新的“门控”机制，巧妙地调控信息的流动，从而攻克了梯度消失和模型退化这两大难题。具体而言，LSTM拥有三个核心门控机制：输入门、遗忘门和输出门。输入门负责筛选并接纳新信息，遗忘门则决定哪些旧信息应当被丢弃，而输出门则掌控着最终输出的信息流。正是这些精巧的门控机制，让LSTM在应对长期依赖问题时展现出了卓越的性能。

模型训练：

LSTM的训练过程通常采用反向传播算法和优化算法（如随机梯度下降）相结合的方式。训练过程中，算法会精确计算损失函数关于权重的梯度，并利用优化算法不断调整权重，以最小化损失函数。为了进一步提升训练效率和模型的泛化能力，还可以考虑采用正则化技术、集成学习等高级策略。

优点：

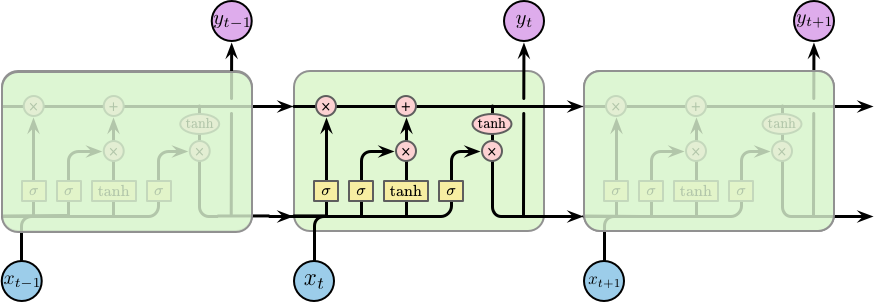
1. 攻克梯度消失和模型退化：通过引入门控机制，LSTM在解决长期依赖问题上表现卓越，有效避免了梯度消失和模型退化的问题。
2. 构建深邃网络结构：得益于对梯度消失和模型退化的处理，LSTM能够构建深度庞大的网络结构，从而充分发掘数据的内在规律，提升模型性能。
3. 多任务表现出色：LSTM在文本生成、语音识别、机器翻译等多个任务中均展现了出色的性能，证明了其强大的特征学习和表示能力。

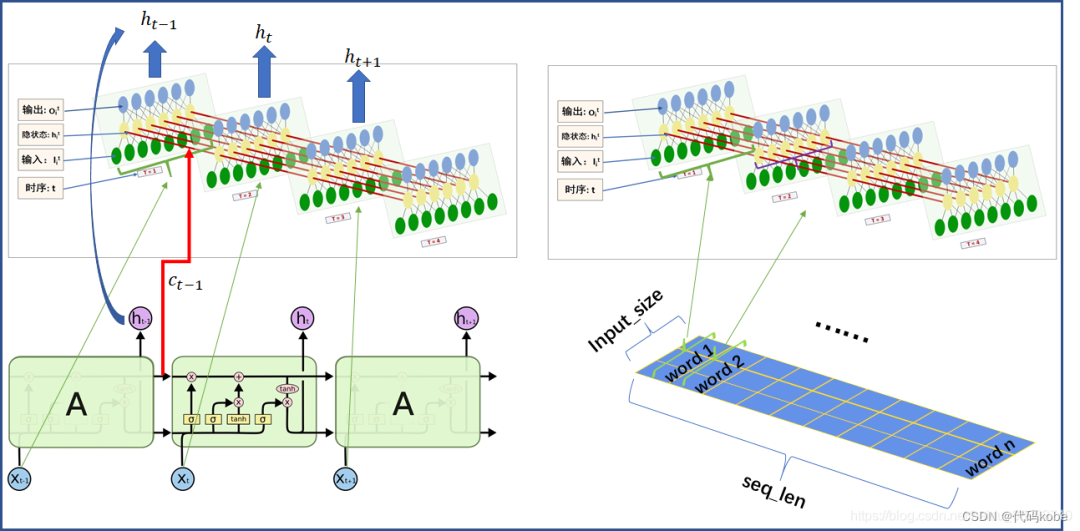
缺点：

1. 参数调优挑战大：LSTM涉及大量参数，调优过程繁琐，需要投入大量时间和精力进行超参数选择和调整。
2. 对初始化敏感：LSTM对权重的初始化极为敏感，不合适的初始化可能导致训练不稳定或出现过拟合问题。
3. 计算量大：由于LSTM通常构建深度网络结构，计算量庞大，对计算资源和训练时间要求较高。

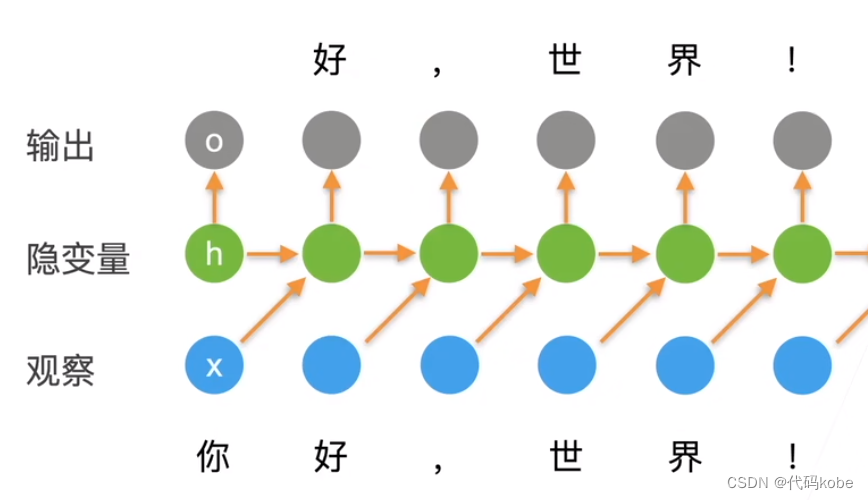
使用场景：

在自然语言处理领域，LSTM凭借其独特的优势在文本生成、机器翻译、语音识别等任务中广泛应用。此外，LSTM在时间序列分析、推荐系统等领域也展现出了巨大的潜力。





input = Input(shape=(12, 1))  
lstm = LSTM(units=50, activation='relu')(input)  
output = Dense(units=1)(lstm)  
model = Model(inputs=input, outputs=output)  
model.summary()



LSTM层参数数量计算:

* + LSTM 有若干个权重矩阵，这些矩阵分别用于输入门、输出门、遗忘门和单元状态的计算。
  + 参数总数可以通过公式计算：4 \* (units \* units + units\*（input\_dim + biases))，其中 input\_dim 是输入特征的数目，units 是 LSTM 单元的数目，biases 是偏置。

对于这个模型：

* + units = 50
  + input\_dim = 1
  + biases = 1

参数数量 = 4 \* (50 \* (50 + 1 + 1)) = 4 \* (50 \* 52) = 4 \* 2600 = 10400

Dense层参数数量计算:

* + Dense层权重矩阵参数数量 = output\_units \* （input\_units + biases）

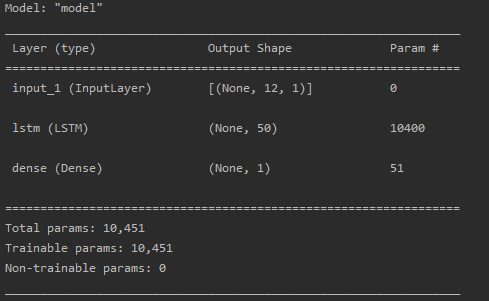
对于这个模型：

* + input\_units = 50 (LSTM层输出的单元数)
  + output\_units = 1
  + biases = 1

参数数量 = 1\*（50 + 1） = 51

模型摘要

* 输入形状是 **(12, 1)**。
* LSTM层有50个单元，使用ReLU激活函数。
* LSTM层共有10400个参数。
* Dense层有1个单元，共有51个参数。
* 模型总参数数量为10451。



**案例：时序数据预测**

在现实世界的案例中，我们主要有两种类型的时间序列分析：

* **单变量时间序列**

对于单变量时间序列数据，我们将使用单列进行预测。只有一列，因此即将到来的未来值将仅取决于它之前的值。

* **多元时间序列**

在多元时间序列数据的情况下，将有不同类型的特征值并且目标数据将依赖于这些特征。，目标数据不仅取决于它以前的值，还取决于其他特征。因此，要预测即将到来的目标数据值，我们必须考虑包括目标列在内的所有列来对目标值进行预测。

**6.4 生成对抗网络GAN**

**案例：图像生成**

**6.5 深度残差网络**

**6.5 transformer**

**6.5 模型的构建**

Tensorflow2.x通过接口创建模型的方式主要有三种：

1. Sequential API，顺序模型；

顺序API是layer-by-layer的方式，适用于简单的层堆栈，如单输入单输出，但对于构建多输入、多输出的模型难以实现。

1. Function API，函数式模型；

函数式API能很好的处理非线性拓扑、共享层、具有多输入多输出的模型。且模型通常都是层的有向无环图（DAG），因此函数式API是构建层计算图的一种方式。

1. Subclassing API，子类化模型；

子类化API是通过继承tf.keras.layers.Layer类或tf.keras.Model类的自定义层和自定义模型。它们与函数式API并不冲突，特别是自定义层---创建自己的层来扩展API，很方便的与函数式API结合构建模型。

除了通过以上三种调用函数接口的方式，你也可以采用最原生的方法，通过自己写原生代码搭建神经网络模型。

**6.5 模型的训练**

几个概念：

* Epoch（周期）：一个Epoch就是将所有训练样本训练一次的过程。然而，当一个Epoch的样本（也就是所有的训练样本）数量可能太过庞大（对于计算机而言），就需要把它分成多个小块，也就是就是分成多个Batch 来进行训练。
* Batch（批 / 一批样本）：将整个训练样本分成若干个Batch。
* Batch\_Size（批大小）：每批样本的大小。即1次迭代所使用的样本量。
* Iteration（一次迭代）：训练一个Batch就是一次Iteration（这个概念跟程序语言中的迭代器相似）每次迭代更新1次网络结构的参数
* step（一步）：训练一个样本就是一个step。有时候step和Iteration的概念的含义是一致的，及step在有些文献或者代码里面就是表示一次迭代。

       比如我有1000个训练样本，bachsize设置为10，则数据分成了100个batch，所有训练样本训练一次即一个epoch需要100个iteration，训练一个batch就是一次iteration。

损失函数

优化器（优化算法）

优化是指通过调整模型参数来最小化损失函数的过程；损失函数是用来衡量模型预测值与真实值之间差异的函数，优化算法的目标是找到使损失函数最小的模型参数。

常见的优化算法包括批量梯度下降（Batch Gradient Descent，BGD）,随机梯度下降（Stochastic Gradient Descent，SGD），自适应梯度下降（Adagrad），RMSprop，Adam等.

- fit函数接口

训练（fit）之前需要编译（compile）

- 原生方式

**for** epoch **in** range(200):

**for** step, (x, y) **in** enumerate(train\_db):

**with** tf.GradientTape() **as** tape: # 梯度记录器

out = model(x) # 通过网络获得输出

loss = tf.reduce\_mean(keras.losses.MSE(y, out)) # 计算MSE

mae\_loss = tf.reduce\_mean(keras.losses.MAE(y, out)) # 计算MAE

**if** step % 10 == 0:

print(epoch, step, float(loss))

# 更新梯度信息

grads = tape.gradient(loss, model.trainable\_variables)

optimizer.apply\_gradients(zip(grads, model.trainable\_variables))

train\_mae\_losses.append(float(mae\_loss))

out = model(tf.constant(normed\_test\_data.values))

test\_mae\_losses.append(tf.reduce\_mean(keras.losses.MAE(test\_lables, out)))

原生方式包含了训练和编译（compile）两部分，不需要model.compile()

# seed\_value = 42  
# np.random.seed(seed\_value)  
# tf.random.set\_seed(seed\_value)  
# random.seed(seed\_value)

**6.5 模型的保存（持久化）**

- 完整保存

- 仅保存权重

- 仅保存架构

**6.5 模型的部署**

- 直接部署

- Docker容器化部署

**6.5 模型提供服务的三种方式**

- 命令行界面

- 通过 命令行界面(Command Line Interface, CLI) 进行默认交互

- 可视化界面

- 通过 Gradio 提供基于 Web 的 GUI

- API 服务器

- 可以使用 FastAPI 包装代理并将其用作 API 端点

**7、Python大语言模型**

**7.1 大语言模型基础知识**

**7.2 从头训练一个大语言模型（预训练模型、基座模型）**

**7.3 大语言模型微调（行业大模型、监督微调模型）**

**7.3.1 监督微调**

根据特定任务进行指令微调，使其更适合执行具体的自然语言处理任务。与预训练不同，监督微调需要**\*\*特定的标记数据集\*\***，其中包含了指令和相应的期望输出。

指令微调的关键在于微调数据集的构建。**\*\*指令数据集中的每个实例一般由三个元素组成\*\***:

- instruction: 指令，它是指定任务的自然语言文本序列，是对模型的语言提示

- input: 输入：上下文提供补充信息的可选输入

-output: 基于指令和输入的预期输出

- 数据格式：整体的结构采用{instruction，input，output}，数据集可以以文本文件（如CSV、JSON等）或数据库的形式存储。每个实例包含输入提示和预期的输出。

QA和Instruction的区别与联系

QA是指一问一答的形式，通常是用户提问，模型给出回答。而Instruction则源自于Prompt Engineering，将问题拆分为两个部分：Instruction用于描述任务，Input用于描述待处理的对象。**问答（QA）格式的训练数据**通常用于训练模型回答基于知识的问题，而**指令（Instruction）格式的训练数据**更适用于训练模型执行具体任务。例如，对于问题"请解释VC银翘片和双黄连口服液之间的区别"

* 问答（QA）格式:

指令（Instruction）：

输入（Input）：VC银翘片和双黄连口服液之间的区别是什么？

* + 1
  + 2
* 指令（Instruction）格式:

指令（Instruction）：请解释下面两个药品之间的区别。

输入（Input）：VC银翘片和双黄连口服液。

指令的形式可能使模型具有更好的泛化能力，因为它强调了任务的性质，而不仅仅是特定的输入。通常指令格式和问答格式可以相互转化。

正如市面上好多针对微调机器学习超参数的第三方库一样，

现在已经出现了很多针对不同模型不同方法的微调框架（开源项目），比如针对chatglm的微调项目**liucongg/ChatGLM-Finetuning，**我们只需要准备好指令微调数据集借助这些框架便可以快速对某个大模型的某种微调。

**7.3.1.1 全量微调**

**7.3.1.2 高效参数微调**

Prompt Tuning

Prefix Tuning

P-Tuning

LoRA

QLoRA

**7.3.2 对齐微调**

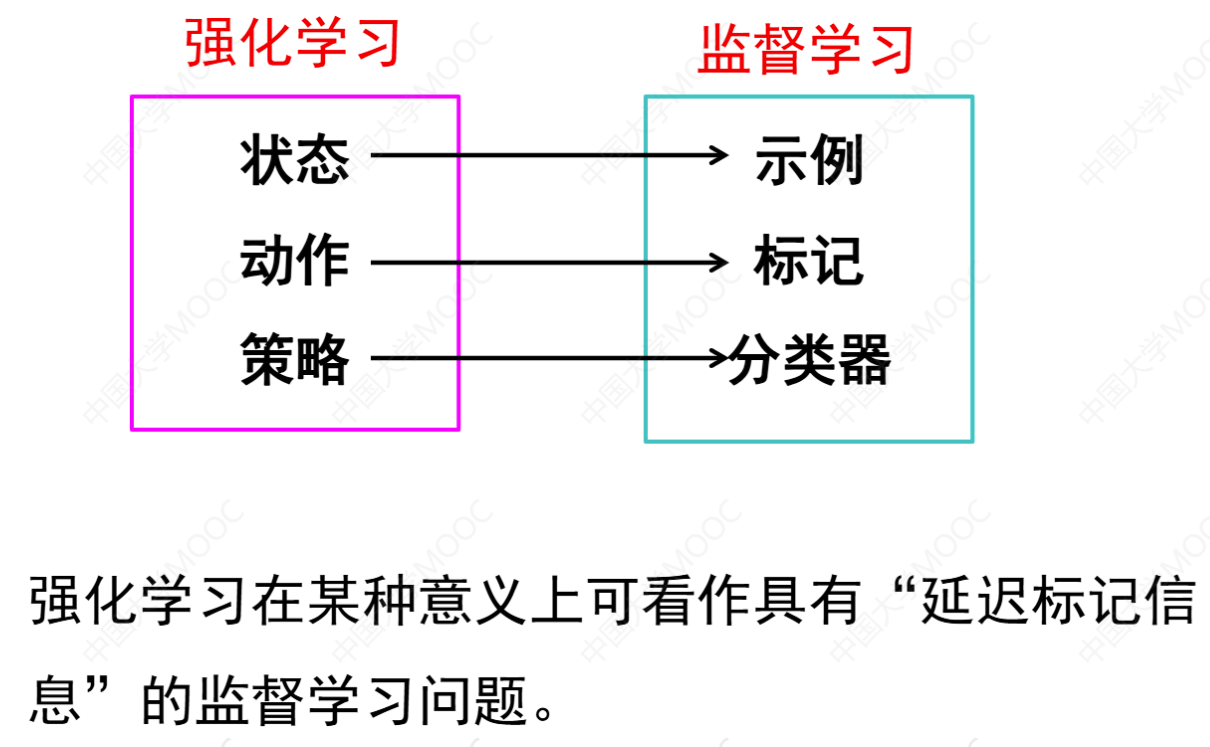
目标是将LLM的输出与人类偏好对齐，以进一步提高模型的性能、帮助性和安全性。这一阶段常常涉及强化学习与人类反馈（RLHF）的应用。RLHF是对齐阶段的核心，它为模型提供了一个持续学习和改进的机制。在RLHF中，模型生成回答（response或answer），并接收人类提供的反馈信息。

**7.3.2.1 奖励模型**

对SFT训练得到的模型喂入一些prompt，使它输出多个response，由人类进行打分排序，所得到的包含\*\*(prompt, response)排序\*\*的数据集进行训练得到RM模型

**7.3.2.2 强化学习模型**

将SFT训练的LMM模型作为策略（Policy），将RM模型视为环境（Environment），采用PPO的RL方法训练LMM模型，得到最终模型。



强化学习在某种意义上可看作具有“延迟标记信息”的监督学习问题。

强化学习基本模型：

在强化学习中，Agent选择一个动作a作用于环境；环境接收该动作后发生变化，同时产生一个强化信号反馈给Agent；Agent再根据强化信号和环境的当前状态s再选择下一个动作，选择的原则是使受到奖赏值的概率增大。

强化学习的目的就是寻找一个最优策略，使得Agent在运行中所获得的累计期望回报最大。

**7.3.3 RAG**

**7.4 基于大语言模型的应用开发（Agent开发）**

7.4.1 单智能体

7.4.2 多智能体

**8、Python项目学习**

**8.1 学习别人的优秀GitHub项目**

7.1.1认识项目中的一些配置文件

7.1.2下载软件包（项目已经发布到Pypi了）

这种方式比较简单，只需要pip install 就可以跑这个项目，实现一些功能

7.1.1 git下载源文件

这种方法需要创建环境，然后安装项目需要的环境，

Requirement.txt，认识**Setup.py**

修改一些配置文件就可以运行

7.1.3 下载镜像文件（项目已经打包成docker镜像）

通过docker跑这个项目，认识**Dockerfile**

**8.2 打造自己的优秀GitHub项目**