

Python数据科学基础：常用库与数据处理技巧

Python数据科学基础：常用库与数据处理技巧

Python数据科学基础：常用库与数据处理技巧

本章将深入探讨使用Python进行数据科学的基础知识，重点介绍核心库及其在数据处理中的应用。我们将从环境搭建开始，逐步学习NumPy、Pandas等库的使用，掌握数据清洗、预处理、探索与可视化等关键技巧，并了解一些高级处理方法和最佳实践。

1. Python数据科学环境搭建与基础

Python因其简洁的语法、强大的生态系统和广泛的社区支持，成为数据科学领域的首选语言。搭建一个稳定、高效的数据科学环境是入门的第一步。

环境搭建：

推荐使用Anaconda或Miniconda。它们是Python的发行版，集成了众多科学计算所需的库，并提供了强大的包管理和环境管理工具（conda）。

1. **下载与安装：** 从Anaconda或Miniconda官网下载对应操作系统的安装包并进行安装。
2. **创建虚拟环境：** 为了隔离不同项目所需的库版本，强烈建议为每个项目创建独立的虚拟环境。

```
conda create -n my_datascience_env python=3.9
```

这里创建了一个名为 my_datascience_env 、使用Python 3.9的虚拟环境。

3. **激活虚拟环境：**

```
conda activate my_datascience_env
```

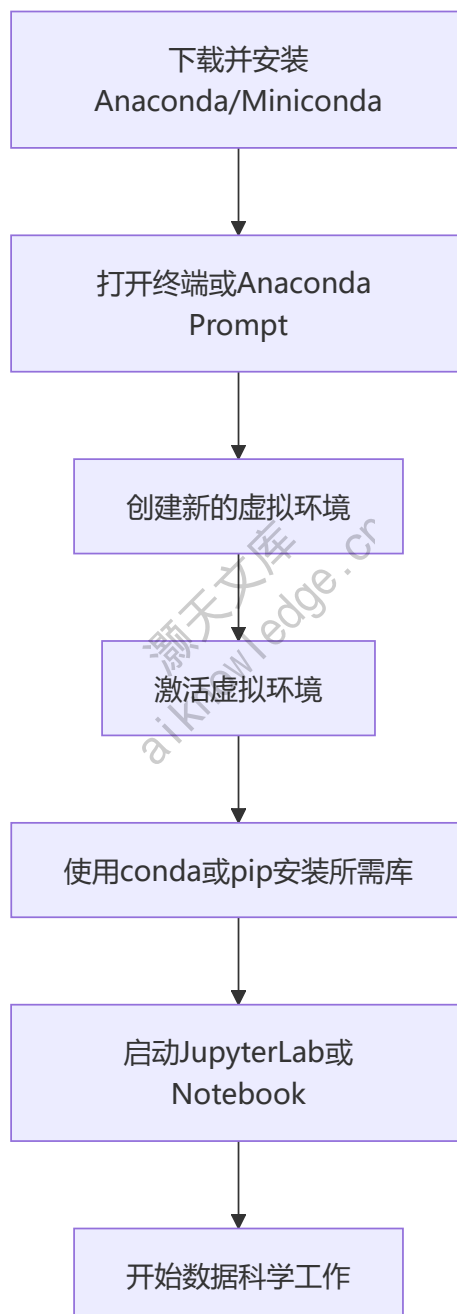
4. **安装常用库：** 在激活的环境中安装本章及后续学习所需的库。

```
conda install numpy pandas matplotlib seaborn jupyterlab scikit-learn
# 或使用 pip
# pip install numpy pandas matplotlib seaborn jupyterlab scikit-learn
```

5. **启动JupyterLab/Jupyter Notebook：** 这是进行交互式数据探索和代码编写的常用工具。

```
jupyter lab  
# 或  
# jupyter notebook
```

图示：数据科学环境搭建流程



Python基础回顾（简要）：

- **数据类型：** 整型 int、浮点型 float、字符串 str、布尔型 bool、列表 list、元组 tuple、字典 dict、集合 set。

- **变量与赋值:** `variable_name = value`
- **控制流:** `if/elif/else` 条件判断, `for` 循环, `while` 循环。
- **函数:** 使用 `def` 定义函数。
- **模块与包:** 使用 `import` 导入库或模块。

这些基础知识是进行数据处理的前提。

2. NumPy: 科学计算基础

NumPy (Numerical Python) 是Python中用于科学计算的核心库。它提供了高性能的多维数组对象 `ndarray`, 以及用于处理这些数组的工具。几乎所有其他科学计算和数据科学库 (如 Pandas、SciPy、Scikit-learn) 都构建在NumPy之上。

核心概念: ndarray

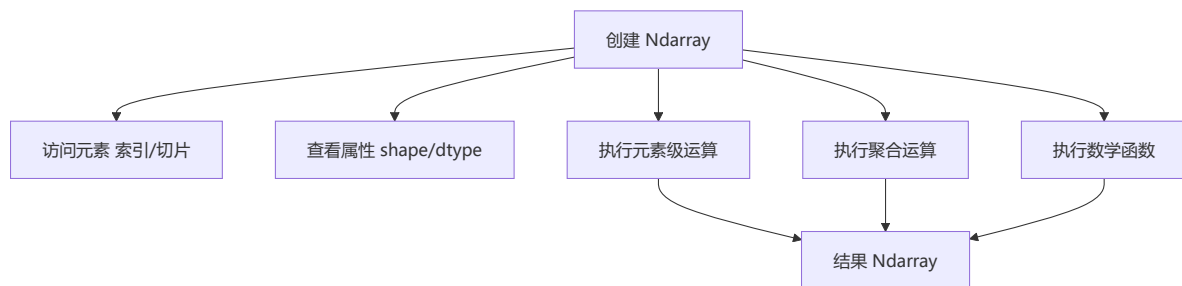
- `ndarray` 是一个同质的、多维的数组。这意味着数组中所有元素的类型必须相同。
- 与Python列表相比, `ndarray` 在存储效率和运算速度上具有显著优势, 特别是在处理大量数值数据时。

主要功能:

- **数组创建:**
 - 从Python列表创建: `np.array([1, 2, 3])`
 - 创建特定值的数组: `np.zeros((2, 3))`, `np.ones((3, 2))`, `np.full((2, 2), 7)`
 - 创建序列: `np.arange(0, 10, 2)`, `np.linspace(0, 1, 5)`
 - 创建随机数组: `np.random.rand(2, 2)`, `np.random.randn(3, 3)`
- **数组属性:** `shape` (维度), `dtype` (元素类型), `ndim` (轴数), `size` (元素总数)。
- **索引与切片:** 与Python列表类似, 支持基本索引、切片、花式索引 (Fancy Indexing) 和布尔索引。
 - `arr[0]` (获取第一个元素)
 - `arr[1:3]` (切片)
 - `arr[row_indices, col_indices]` (花式索引)
 - `arr[arr > 5]` (布尔索引)
- **数组运算:** NumPy支持元素级运算 (`+`, `-`, `*`, `/`等), 矩阵运算 (`@` 或 `np.dot`), 聚合函数 (`sum`, `mean`, `std`, `max`, `min` 等), 以及各种数学函数 (`np.sin`, `np.cos`, `np.exp` 等)。

- **广播 (Broadcasting)** : NumPy能够自动处理形状不同的数组之间的运算, 前提是它们的形状满足广播规则。

图示: NumPy Nddarray 核心操作



NumPy是进行高效数值计算的基石, 理解其 `ndarray` 对象和广播机制对于后续学习Pandas和机器学习库至关重要。

3. Pandas: 数据处理核心

Pandas是Python中最流行的数据处理和分析库。它提供了高性能、易于使用的数据结构 `Series` 和 `DataFrame`, 使得处理结构化 (表格化) 数据变得非常便捷。Pandas构建在NumPy之上, 并与Matplotlib等库紧密集成。

核心概念: Series 和 DataFrame

- **Series**: 一维带标签的数组。可以看作是带索引的NumPy数组或单列的电子表格。
- **DataFrame**: 二维带标签的数据结构, 由按列组织的 `Series` 组成。可以看作是电子表格或SQL表。

主要功能:

- **数据加载与保存**: 支持读取多种数据格式, 如CSV (`pd.read_csv`), Excel (`pd.read_excel`), SQL数据库 (`pd.read_sql`), JSON等。同样支持将DataFrame保存到这些格式。
- **数据选择与索引**:
 - 列选择: `df['column_name']`, `df[['col1', 'col2']]`
 - 行选择: 使用标签 (`.loc`) 或整数位置 (`.iloc`)。
 - `df.loc[label]` (按标签选择行)
 - `df.iloc[index]` (按位置选择行)
 - `df.loc[row_labels, col_labels]` (按标签选择行和列)
 - `df.iloc[row_indices, col_indices]` (按位置选择行和列)
 - 布尔索引: `df[df['column'] > value]`

- **数据操作:**

- 排序: `df.sort_values(by='column')`, `df.sort_index()`
- 过滤: 使用布尔索引。
- 新增/修改/删除列: 直接赋值 `df['new_col'] = ...`, 使用 `df.drop()`。

- **处理缺失数据:**

- 检测缺失值: `df.isnull()`, `df.notnull()`
- 删除缺失值: `df.dropna()` (删除包含缺失值的行或列)
- 填充缺失值: `df.fillna(value)` (用指定值、均值、中位数等填充)

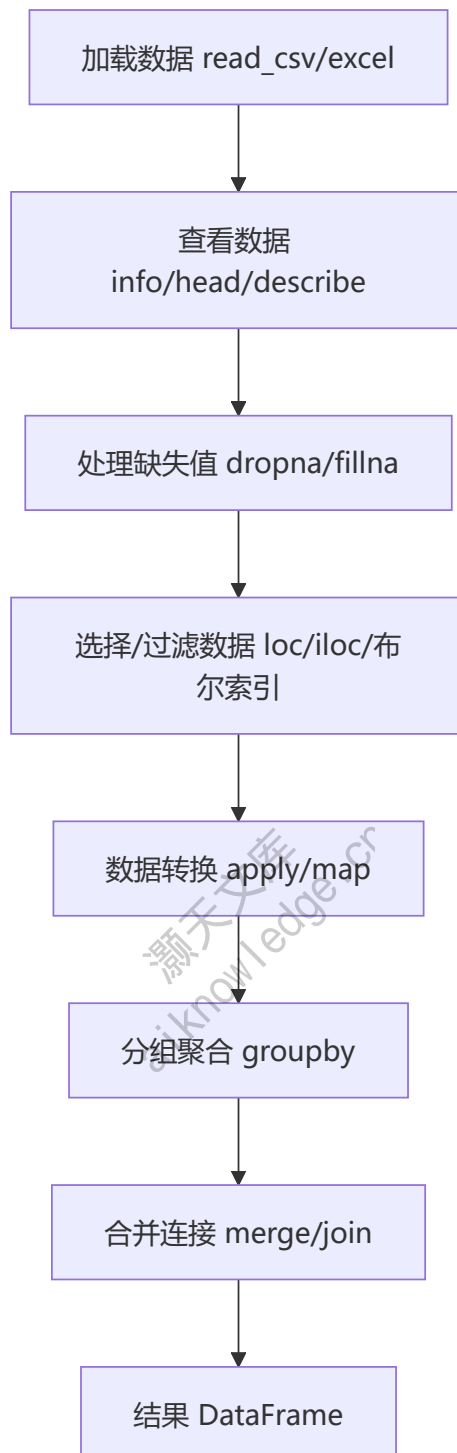
- **分组与聚合:** `df.groupby('column').agg({'col1': 'mean', 'col2': 'sum'})`

- **数据合并与连接:** `pd.merge()`, `df.join()`, `pd.concat()`

- **数据转换:** `df.apply()`, `df.map()`, `df.transform()`

图示: Pandas DataFrame 常用操作

灏天文库
aiknowledge.cn



Pandas是数据科学家日常工作中不可或缺的工具，熟练掌握其各种操作是高效处理数据的关键。

4. 数据清洗与预处理

真实世界的数据往往不完美，包含各种错误、不一致和缺失值。数据清洗（Data Cleaning）和预处理（Data Preprocessing）是数据分析流程中至关重要的一步，直接影响后续分析和模型的质量。

常见数据问题：

- **缺失值**：数据记录不完整。
- **重复值**：完全相同或部分相同的记录。
- **异常值 (Outliers)**：显著偏离大多数数据的观测值。
- **数据类型不一致**：例如，数字存储为字符串。
- **格式错误**：日期格式不统一，字符串包含无关字符等。
- **不一致的记录**：同一个实体有不同的表示（例如，"USA" vs "United States"）。

清洗与预处理技术：

1. 处理缺失值：

- **识别**：使用 `df.isnull().sum()` 查看每列的缺失值数量。
- **删除**：如果缺失值数量很少或整行/列不重要，可以使用 `df.dropna()` 删除。
- **填充 (Imputation)**：使用均值、中位数、众数、前向填充 (`ffill`)、后向填充 (`bfill`) 或基于模型的预测值填充。 `df.fillna()` 是主要方法。

2. 处理重复值：

- **识别**：`df.duplicated()` 返回布尔Series，指示哪些行是重复的。
- **删除**：`df.drop_duplicates()` 删除重复行。

3. 处理异常值：

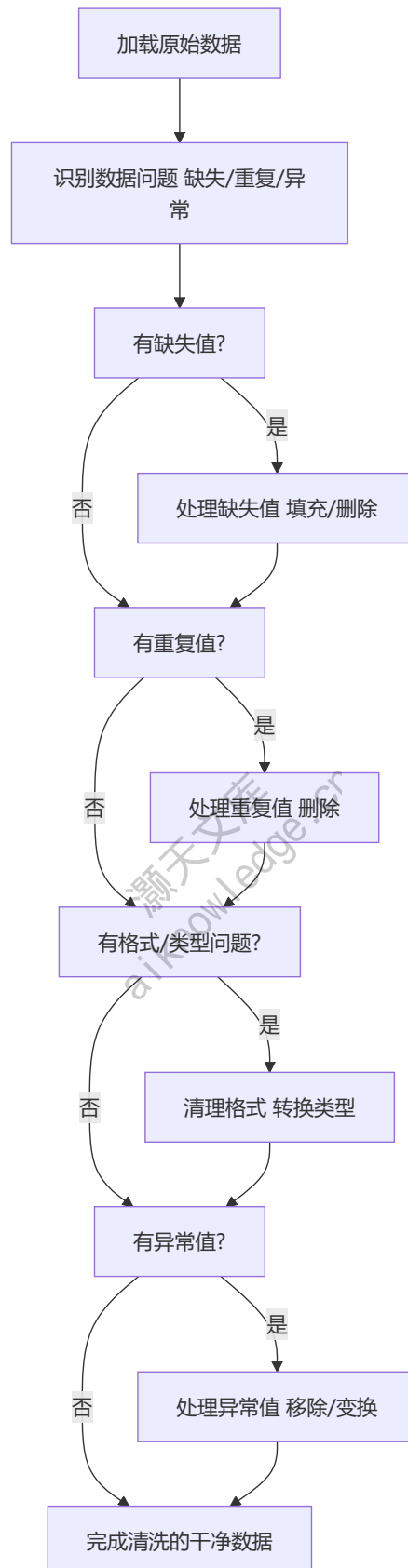
- **识别**：通过可视化（箱线图、散点图）、统计方法（Z-score、IQR方法）检测。
- **处理**：删除异常值行、替换为特定值（如分位数）、使用变换（如了对数变换）减少异常值的影响。

4. 数据类型转换：使用 `df['column'].astype(new_dtype)` 或 `pd.to_datetime()` , `pd.to_numeric()` 等函数将列转换为正确的类型。

5. 格式清理：使用字符串方法 (`.str` 访问器) 清理文本数据，如去除空格 (`.str.strip()`)、转换为大小写 (`.str.lower()`)、替换字符 (`.str.replace()`)。

6. 数据标准化/归一化 (Scaling/Normalization)：在构建机器学习模型前常用，将数据缩放到特定范围或使其符合特定分布，以消除特征间的量纲差异。常用的有Min-Max Scaling和Standardization (Z-score scaling)。通常使用Scikit-learn的 `preprocessing` 模块实现。

图示：典型数据清洗流程



数据清洗是耗时但至关重要的一步，通常占据数据科学家大量工作时间。

5. 数据探索与可视化

数据探索性分析 (EDA - Exploratory Data Analysis) 是理解数据特征、发现模式、检验假设和识别问题的过程。数据可视化是EDA的核心工具，能够直观地展示数据分布、关系和趋势。

数据探索技术：

- **描述性统计：** 使用 `df.describe()` 获取数值列的统计摘要（计数、均值、标准差、最小值、最大值、分位数）。使用 `df.info()` 获取列的类型和非空值数量。使用 `df['column'].value_counts()` 查看分类列的频次分布。
- **数据分组与聚合：** 使用 `groupby()` 对数据进行分组，然后应用聚合函数（如 `mean()`，`sum()`，`count()`）来比较不同组的特征。
- **相关性分析：** 使用 `df.corr()` 计算列之间的相关系数矩阵，帮助理解变量间的线性关系。

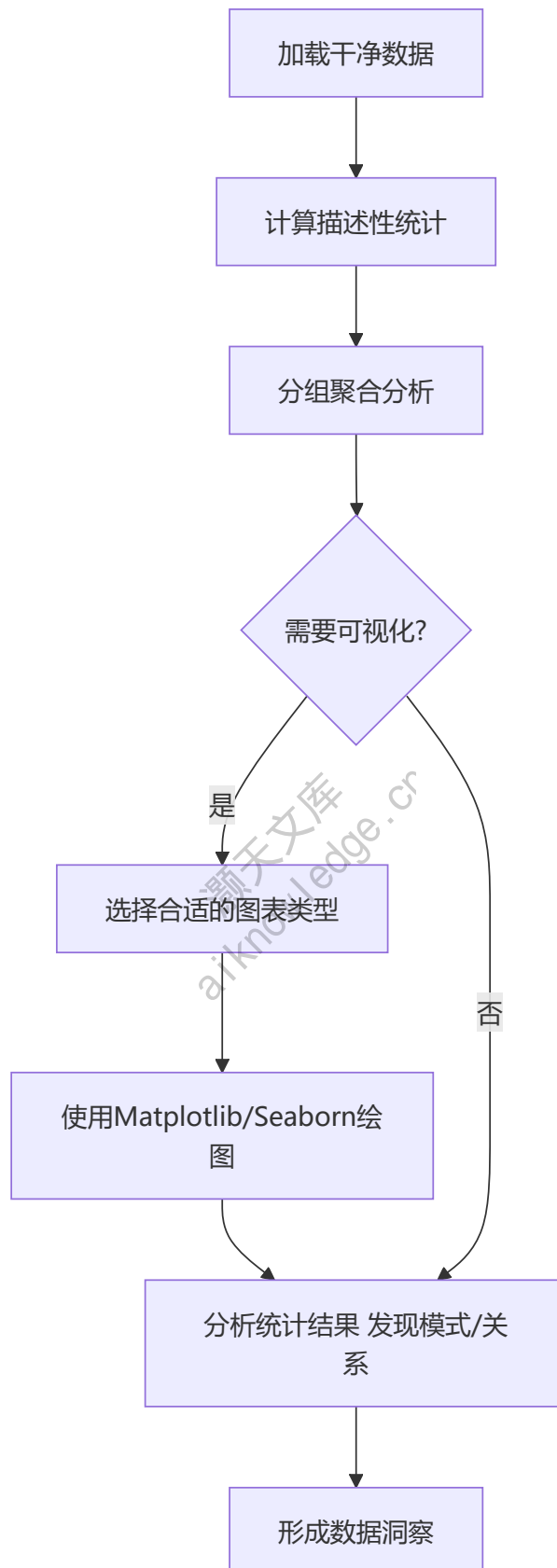
数据可视化库：

- **Matplotlib：** Python中最基础和广泛使用的绘图库，提供了丰富的图表类型和高度的定制能力。通常作为其他高级可视化库的底层。
- **Seaborn：** 基于Matplotlib，提供了更美观的默认风格和更高级的统计图表类型（如箱线图、小提琴图、热力图、散点图矩阵等），特别适合用于数据探索。

常用图表类型及用途：

- **直方图 (Histogram)：** 显示单个数值变量的分布。
- **箱线图 (Box Plot)：** 显示单个数值变量的分布摘要（中位数、四分位数、异常值），或比较不同类别下数值变量的分布。
- **散点图 (Scatter Plot)：** 显示两个数值变量之间的关系。
- **条形图 (Bar Plot)：** 显示分类变量的频次，或比较不同类别下数值变量的总和/均值等。
- **折线图 (Line Plot)：** 显示数据随时间或其他有序变量的变化趋势。
- **热力图 (Heatmap)：** 用颜色强度表示矩阵中数值的大小，常用于展示相关性矩阵或二维数据的分布。

图示：数据探索与可视化流程



EDA和可视化是理解数据、生成假设并指导后续建模的关键步骤。

6. 高级数据处理技巧与最佳实践

掌握基础操作后，了解一些高级技巧和遵循最佳实践可以显著提高数据处理效率和代码质量。

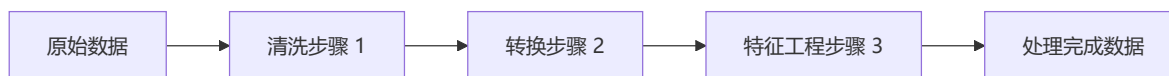
高级数据处理技巧：

- **向量化操作：** 优先使用NumPy和Pandas内置的向量化函数和方法，而不是Python循环。向量化操作通常在底层使用C或其他编译语言实现，效率远高于Python循环。
 - 例如：`df['col1'] + df['col2']` 比 `for index, row in df.iterrows():` 中相加快得多。
- **apply, map, applymap：**
 - `apply()`：沿DataFrame的轴（行或列）应用函数。常用于对行或列进行复杂转换。
 - `map()`：仅用于Series，将函数或字典映射到Series的每个元素上。
 - `applymap()`：用于DataFrame，将函数应用到DataFrame的每个元素上（在较新版本中推荐使用 `df.map()` 配合 `na_action=None` 或 `df.apply(lambda x: x.map())`）。
- **Multindex（多级索引）：** 处理具有层次结构的数据时非常有用。允许在轴上拥有多个索引层。
- **Categorical Data（分类数据）：** 对于具有有限且重复值的字符串列，将其转换为Pandas的 `category` 数据类型可以显著减少内存使用并提高某些操作的速度。
- **时间序列数据处理：** Pandas对时间序列数据提供了强大的支持，包括时间索引、重采样（resampling）、频率转换、移动窗口计算等。
- **高效读取大型数据：** 对于非常大的文件，可以考虑分块读取（`pd.read_csv(..., chunksize=...)`），或者使用专门处理大数据的库如Dask或PySpark。

最佳实践：

- **代码可读性：** 使用有意义的变量名，添加注释，保持代码整洁。遵循PEP 8编码规范。
- **模块化：** 将重复的数据处理逻辑封装成函数或类，提高代码的复用性和可维护性。
- **创建数据处理管道（Pipeline）：** 将一系列数据清洗和预处理步骤组织成一个流程。这使得流程清晰、易于管理和重现。Scikit-learn的 `Pipeline` 类是一个很好的工具，尽管主要用于建模流程，但其思想也适用于数据预处理。
- **版本控制：** 使用Git等工具管理代码版本，方便追踪修改、协作和回滚。
- **单元测试：** 对关键的数据处理函数编写单元测试，确保代码的正确性，尤其是在处理复杂逻辑时。
- **文档：** 记录数据处理的步骤、假设和决策，方便自己和他人理解。
- **效率考虑：** 在处理大型数据集时，时刻关注内存使用和计算效率，选择最合适的方法。

图示：数据处理管道概念



通过结合高级技巧和遵循最佳实践，可以构建出高效、健壮和易于维护的数据处理流程。

本章涵盖了Python数据科学的基础，从环境搭建到NumPy和Pandas的核心使用，再到数据清洗、探索、可视化以及一些高级技巧和最佳实践。这些知识是进行任何数据分析、机器学习或数据挖掘项目的基础。在后续章节中，我们将基于这些基础进一步深入学习更高级的数据科学主题。

1. Python数据科学环境搭建与基础

1. Python数据科学环境搭建与基础

欢迎来到Python数据科学的世界！本章作为整个学习旅程的起点，将引导您完成必要的准备工作，并回顾或学习Python中最核心、最常用于数据处理和分析的基础知识。一个稳定且配置良好的环境是高效工作的前提，而扎实的Python基础则是驾驭各种数据科学工具的基石。

1.1 为何选择Python进行数据科学？

在深入环境搭建之前，我们先简要探讨为何Python成为数据科学领域的主流语言之一。

- **生态系统丰富：** Python拥有庞大且活跃的数据科学库生态，例如NumPy、Pandas、Matplotlib、Scikit-learn等，几乎涵盖了数据获取、清洗、转换、分析、建模、可视化等所有环节。
- **易学易用：** Python语法清晰简洁，接近自然语言，入门门槛相对较低，使得数据科学家可以更专注于问题本身而非语言细节。
- **社区支持强大：** 遇到问题时，可以轻松在社区找到解决方案、教程和代码示例。
- **通用性：** Python不仅用于数据科学，还可以用于Web开发、自动化脚本、机器学习工程部署等，这意味着数据科学项目可以更容易地集成到更广泛的系统中。
- **性能：** 虽然Python本身是解释型语言，但其核心数据科学库（如NumPy、Pandas）底层通常使用C、C++或Fortran编写，执行效率很高，可以处理大规模数据集。

1.2 Python数据科学环境搭建

数据科学工作通常需要安装特定版本的Python以及大量的第三方库。直接在系统Python环境中安装可能会导致版本冲突或混乱。因此，推荐使用专门的数据科学发行版或虚拟环境来管理依赖。

1.2.1 推荐的数据科学发行版：Anaconda与Miniconda

Anaconda和Miniconda是专门为科学计算和数据科学设计的Python发行版。它们预装了许多常用的科学计算库，并提供了强大的包管理和环境管理工具 `conda`。

- **Anaconda:** 功能齐全，包含了Python解释器、conda包管理器以及数百个常用的科学计算库（如NumPy、Pandas、SciPy、Matplotlib、Scikit-learn等）和一些集成开发环境（如Spyder）。安装包较大。
- **Miniconda:** 是Anaconda的精简版，只包含Python解释器、conda包管理器以及少数核心库。用户可以根据需要使用conda安装其他库。安装包较小，更灵活。

对于初学者，Anaconda提供了一个开箱即用的环境，非常方便。对于希望节省磁盘空间或更精细控制环境的用户，Miniconda是更好的选择。

1.2.2 安装步骤概述

无论选择Anaconda还是Miniconda，安装过程大致如下：

1. 访问官方网站（Anaconda或Miniconda）。
2. 下载适合您操作系统（Windows、macOS、Linux）和系统架构（64位或32位）的安装包。推荐下载最新版本。
3. 运行安装包。在安装过程中，请注意以下几点：
 - **安装路径:** 可以选择默认路径或自定义路径。确保路径中不包含中文或特殊字符。
 - **添加到系统PATH:** 对于初学者，通常建议勾选"Add Anaconda to my PATH environment variable"或类似选项（非强制，但方便命令行使用）。如果不勾选，则需要在Anaconda Prompt或终端中使用conda命令。
 - **注册为默认Python:** 如果您不希望Anaconda的Python成为系统默认Python，可以取消勾选此选项。使用虚拟环境可以更好地管理多个Python版本。
4. 完成安装。

1.2.3 验证安装

安装完成后，打开命令行终端（Windows用户可以使用Anaconda Prompt，macOS/Linux用户使用系统终端）。输入以下命令验证Python和conda是否安装成功：

```
python --version
conda --version
```

如果能正确显示Python和conda的版本信息，说明安装成功。

1.2.4 理解并使用虚拟环境

虚拟环境是Python开发中的一项重要技术，尤其在数据科学中不可或缺。

为什么需要虚拟环境?

- **隔离性:** 不同项目可能依赖于同一库的不同版本。虚拟环境可以为每个项目创建一个独立的Python解释器和库集合, 避免项目间的依赖冲突。
- **管理性:** 轻松创建、激活、切换和删除特定于项目的环境。
- **可重复性:** 可以导出环境的依赖列表, 方便在其他机器上或与团队成员共享完全相同的环境。

使用conda创建和管理虚拟环境:

- **创建新环境:**

```
conda create --name my_data_env python=3.9
```

这将创建一个名为 `my_data_env` 的新环境, 并指定Python版本为3.9。您可以指定其他Python版本。

- **激活环境:**

- Windows: `conda activate my_data_env`

- macOS/Linux: `conda activate my_data_env`

激活后, 命令行提示符前会显示当前环境的名称 (例如 `(my_data_env)`) 。

- **在环境中安装库:** 激活环境后, 使用 `conda install` 或 `pip install` 安装所需的库。

```
conda install numpy pandas matplotlib scikit-learn
# 或者使用 pip
# pip install numpy pandas matplotlib scikit-learn
```

conda通常更适合安装科学计算库, 因为它能更好地处理二进制依赖。

- **查看已安装的库:** 激活环境后, 使用 `conda list` 或 `pip list` 。
- **查看所有环境:** `conda env list`
- **导出环境配置:** 激活环境后, `conda env export > environment.yml` 。这将生成一个 `environment.yml` 文件, 记录环境名称和所有依赖库及其版本。
- **从配置文件创建环境:** `conda env create -f environment.yml`
- **停用环境:** `conda deactivate`
- **删除环境:** `conda env remove --name my_data_env`

使用虚拟环境是数据科学工作流程中的标准实践，强烈建议您养成习惯。

1.2.5 常用的开发工具 (IDE与编辑器)

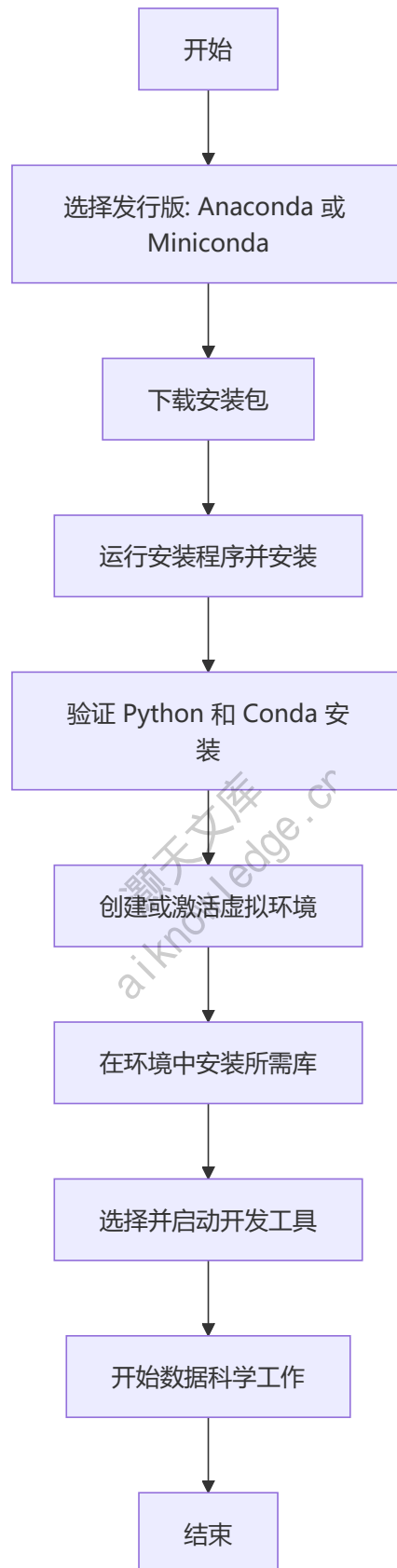
选择一个合适的开发工具可以极大地提高效率。

- **Jupyter Notebook/Jupyter Lab**: 这是数据科学领域最受欢迎的工具之一。它以基于Web的交互式笔记本形式工作，允许您将代码、输出（包括图表）、解释文本和数学公式结合在一起。非常适合探索性数据分析、数据可视化和文档编写。Jupyter Lab是Notebook的下一代版本，提供了更丰富的功能和更灵活的界面。通常随Anaconda安装。
- **VS Code (Visual Studio Code)**: 一个轻量级但功能强大的源代码编辑器，通过安装Python扩展，可以获得代码高亮、智能补全、调试、集成终端等功能，并且原生支持Jupyter Notebook。
- **PyCharm**: JetBrains开发的专业Python IDE，功能强大，提供了高级的代码分析、重构、调试工具，适合大型项目开发。社区版是免费的。

对于初学者和进行数据探索，Jupyter Notebook/Lab是绝佳选择。随着项目复杂度的增加，VS Code或PyCharm会变得更加有用。

1.2.6 环境搭建流程概览

我们可以用一个流程图来概览环境搭建的主要流程：



- **说明:**

- A : 流程的开始。

- B : 决定使用哪个Python发行版。
- C : 获取安装文件。
- D : 执行安装过程。
- E : 确认安装成功。
- F : 建立或进入一个隔离的工作空间。
- G : 获取数据科学必需的工具库。
- H : 打开编写和运行代码的软件。
- I : 进行实际的数据分析等任务。
- J : 流程结束。

1.3 Python基础知识回顾

在环境准备就绪后，我们需要回顾或学习一些基本的Python概念，这些是进行数据处理和分析的基础。

1.3.1 基本数据类型

Python内置了几种基本数据类型：

- **整型 (int)**：表示整数，如 `-2`，`0`，`100`。
- **浮点型 (float)**：表示带小数的数字，如 `3.14`，`-0.01`，`2.0`。
- **字符串 (str)**：表示文本序列，用单引号或双引号括起来，如 `'hello'`，`"World"`。
- **布尔型 (bool)**：表示真或假，只有两个值：`True` 和 `False`。常用于条件判断。

```
**示例**
age = 30
pi = 3.14159
name = "Alice"
is_student = True
print(type(age))
print(type(pi))
print(type(name))
print(type(is_student))
```

1.3.2 核心数据结构

Python提供了几种内置的数据结构，用于组织和存储数据集合。

- **列表 (list)**：有序、可变的数据集合。用方括号 `[]` 表示，元素之间用逗号分隔。可以包含

不同类型的元素。

```
# 示例
numbers = [1, 2, 3, 4, 5]
fruits = ['apple', 'banana', 'cherry']
mixed_list = [1, 'hello', True, 3.14]
# 访问元素（索引从0开始）
print(numbers[0]) # 输出: 1
print(fruits[-1]) # 输出: cherry（负数索引从末尾开始）
# 切片（获取子列表）
print(numbers[1:4]) # 输出: [2, 3, 4]（从索引1到3，不包含4）
print(fruits[:2]) # 输出: ['apple', 'banana']（从开始到索引1）
print(mixed_list[2:]) # 输出: [True, 3.14]（从索引2到末尾）
# 修改元素
fruits[1] = 'blueberry'
print(fruits) # 输出: ['apple', 'blueberry', 'cherry']
# 添加元素
numbers.append(6)
print(numbers) # 输出: [1, 2, 3, 4, 5, 6]
# 列表推导式（List Comprehensions） - 高效创建列表
squares = [x**2 for x in range(1, 6)] # [1, 4, 9, 16, 25]
```

列表在数据科学中常用于存储序列数据或作为临时容器。

- **元组 (tuple):** 有序、不可变的数据集合。用圆括号 `()` 表示，元素之间用逗号分隔。一旦创建，元组的元素不能被修改、添加或删除。

```
# 示例
coordinates = (10.0, 20.0)
rgb_color = (255, 0, 0)
# 访问元素（与列表类似）
print(coordinates[0]) # 输出: 10.0
# 尝试修改会报错
# rgb_color[0] = 200 # TypeError
```

元组常用于表示固定集合的数据，例如坐标、日期等，也常作为字典的键（因为不可变）。

- **字典 (dict):** 无序（在Python 3.7+中保持插入顺序）、可变的键-值对集合。用花括号 `{}` 表示，键和值之间用冒号 `:` 分隔，键值对之间用逗号分隔。键必须是不可变的类型（如字符串、数字、元组）。

```
# 示例
person = {'name': 'Alice', 'age': 30, 'city': 'New York'}
student_grades = {'Math': 90, 'Science': 85, 'History': 92}

# 访问值
print(person['name']) # 输出: Alice
print(student_grades.get('Art', 'N/A')) # 使用get方法, 如果键不存在返回默认值

# 修改值
person['age'] = 31
print(person) # 输出: {'name': 'Alice', 'age': 31, 'city': 'New York'}

# 添加新的键值对
person['occupation'] = 'Engineer'
print(person) # 输出: {'name': 'Alice', 'age': 31, 'city': 'New York',
'occupation': 'Engineer'}

# 遍历字典
for key, value in person.items():
    print(f"{key}: {value}")
```

字典在数据科学中常用于存储具有结构的数据, 例如表示记录、配置信息或查找表。

- **集合 (set):** 无序、不重复的元素集合。用花括号 {} 表示 (空集合必须用 set() 表示), 元素之间用逗号分隔。主要用于成员资格测试和消除重复元素。

```
# 示例
unique_numbers = {1, 2, 3, 3, 4, 5, 5}
print(unique_numbers) # 输出: {1, 2, 3, 4, 5} (顺序可能不同)

# 添加元素
unique_numbers.add(6)
print(unique_numbers) # 输出: {1, 2, 3, 4, 5, 6}

# 成员资格测试
print(3 in unique_numbers) # 输出: True
print(7 in unique_numbers) # 输出: False

# 集合运算 (交集、并集、差集等)
set1 = {1, 2, 3, 4}
set2 = {3, 4, 5, 6}
print(set1.intersection(set2)) # 输出: {3, 4}
print(set1.union(set2)) # 输出: {1, 2, 3, 4, 5, 6}
print(set1.difference(set2)) # 输出: {1, 2}
```

集合在数据清洗和预处理中非常有用, 例如查找唯一值、比较数据集的异同等。

1.3.3 控制流

控制流语句用于决定程序执行的顺序。

- **条件判断 (if, elif, else):** 根据条件的真假执行不同的代码块。

```
# 示例
score = 85
if score >= 90:
    print("优秀")
elif score >= 80:
    print("良好")
elif score >= 60:
    print("及格")
else:
    print("不及格")
```

- **循环 (for, while):** 重复执行一段代码。

- **for** 循环常用于遍历序列（如列表、元组、字符串）或可迭代对象。

```
# 示例: 遍历列表
for item in fruits:
    print(item)

# 示例: 遍历数字范围
for i in range(5): # range(5) 生成 0, 1, 2, 3, 4
    print(i)

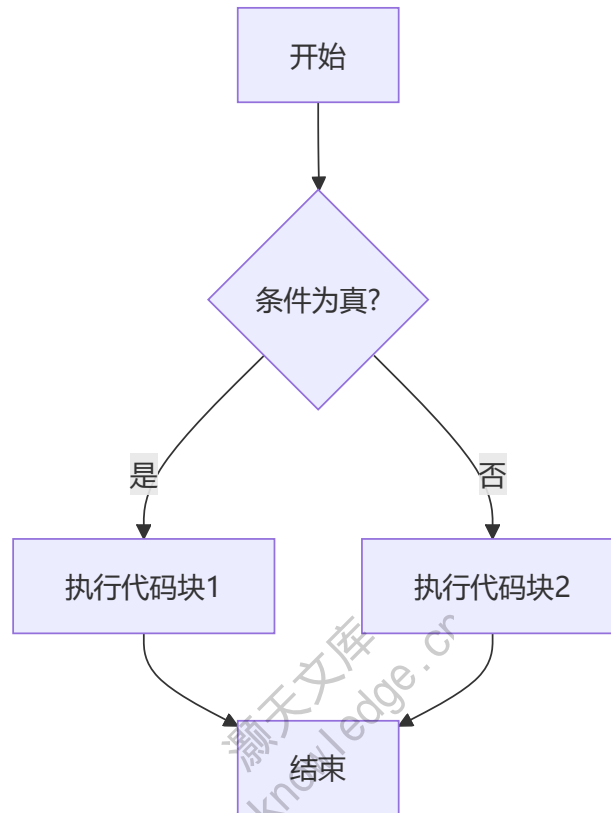
# 示例: 遍历字典
for key in person: # 默认遍历键
    print(key, person[key])
for value in person.values(): # 遍历值
    print(value)
for key, value in person.items(): # 遍历键值对
    print(f"{key}: {value}")
```

- **while** 循环在条件为真时重复执行。

```
# 示例
count = 0
while count < 5:
```

```
print(count)
count += 1
```

我们可以用一个流程图展示一个简单的条件判断结构：



• **说明：**

- A：流程开始。
- B：判断一个条件是否成立。
- C：如果条件为真，执行这部分代码。
- D：如果条件为假，执行这部分代码。
- E：流程结束。

1.3.4 函数

函数是一段可重用的代码块，用于执行特定任务。定义函数可以提高代码的模块化和可读性。

```
**定义函数**
def greet(name):
    """这是一个打招呼的函数."""
    print(f"Hello, {name}!")
def add_numbers(a, b):
```

```
"""计算两个数的和."""  
    return a + b # 使用 return 返回结果  
**调用函数**  
greet("Alice")  
sum_result = add_numbers(5, 3)  
print(sum_result) # 输出: 8
```

函数在数据科学中非常重要，您会经常定义自己的函数来处理数据，或者调用库中提供的函数。

1.3.5 导入模块

Python的强大之处在于其模块（Module）和包（Package）系统。模块是包含Python代码的文件（.py），包是组织模块的文件夹。导入模块可以访问其中定义的函数、类和变量。

```
**导入整个模块**  
import math  
print(math.sqrt(16)) # 调用 math 模块中的 sqrt 函数  
**导入模块并起别名（常用方式，尤其是数据科学库）**  
import numpy as np # numpy 是科学计算的核心库  
import pandas as pd # pandas 是数据处理和分析的核心库  
**导入模块中的特定部分**  
from random import randint  
print(randint(1, 10)) # 直接调用 randint 函数  
**导入模块中的所有内容（不推荐，容易引起命名冲突）**  
**from math import ***  
**print(sqrt(25))**
```

在数据科学中，您会频繁使用 `import numpy as np` 和 `import pandas as pd` 来加载这两个最基础也是最重要的库。

1.4 总结

本章我们首先了解了Python在数据科学领域的优势，然后详细介绍了如何搭建一个稳定高效的数据科学环境，包括选择合适的发行版（Anaconda/Miniconda）、安装过程、虚拟环境的使用以及常用的开发工具（Jupyter Notebook/Lab、VS Code）。最后，我们回顾了Python中最基础但至关重要的概念：基本数据类型、核心数据结构（列表、元组、字典、集合）、控制流（条件判断、循环）和函数定义与模块导入。

掌握了这些基础知识并搭建好环境，您就已经为后续深入学习Python数据科学库和数据处理技巧打下了坚实的基础。在接下来的章节中，我们将开始探索NumPy和Pandas等强大的库，它们将极大地提升您的数据处理能力。

2. NumPy：科学计算基础

2. NumPy: 科学计算基础

在Python数据科学领域，NumPy (Numerical Python) 是不可或缺的基石。它提供了一个高性能的多维数组对象以及用于处理这些数组的各种工具。几乎所有基于Python的数据科学库，如Pandas、SciPy、Scikit-learn等，都构建在NumPy之上。本章将深入探讨NumPy的核心概念和常用功能。

2.1 为什么选择NumPy?

Python原生列表 (list) 功能强大且灵活，但在处理大量数值数据时效率较低，尤其是在执行数学运算时。NumPy通过引入 `ndarray` 对象解决了这个问题。

`ndarray` (n-dimensional array) 是一个多维度的同质数据容器。同质意味着数组中的所有元素必须是相同的数据类型 (如整数、浮点数等)。这种特性使得NumPy能够：

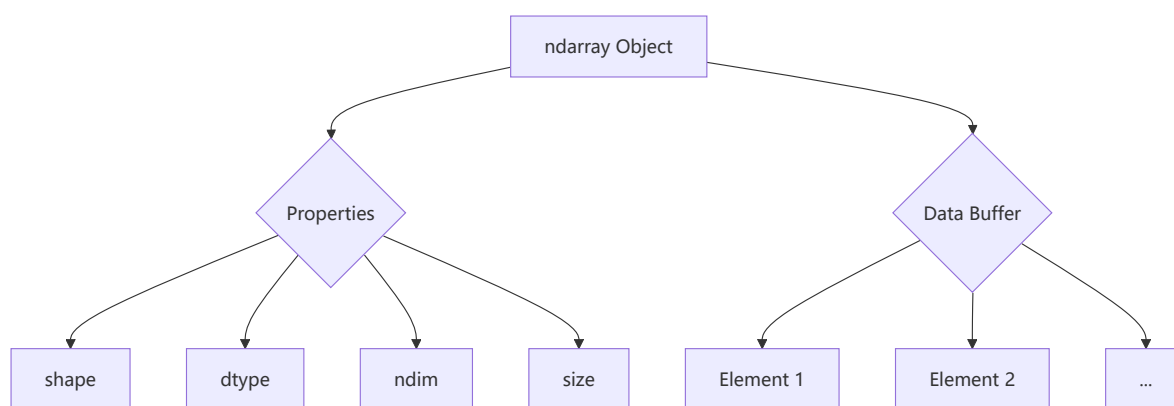
1. **显著提升计算速度：** NumPy的许多操作是在底层C或Fortran实现，远快于Python的纯解释器循环。
2. **节省内存：** 存储同质数据比异质数据更紧凑高效。
3. **提供强大的数学函数库：** 支持向量化操作和广播功能，简化了复杂的数学表达式。

2.2 `ndarray` 对象：NumPy的核心

`ndarray` 是NumPy中最重要的数据结构。一个 `ndarray` 包含：

- **数据：** 存储在连续的内存块中的元素。
- **元数据：** 描述数据的结构，包括：
 - `shape`：一个整数元组，表示数组在每个维度上的大小。例如，一个 `(3, 4)` 的 `shape` 表示一个3行4列的二维数组。
 - `dtype`：描述数组中元素的数据类型 (如 `int64`, `float32`, `bool` 等)。
 - `ndim`：数组的维度数量 (轴的数量)。
 - `size`：数组中元素的总数量。

下面是一个简单的 `ndarray` 结构示意图：



2.3 创建 ndarray

NumPy提供了多种创建数组的方法：

- **从Python列表或元组创建：** 使用 `np.array()` 函数。

```
import numpy as np
arr1d = np.array([1, 2, 3, 4, 5])
arr2d = np.array([[1, 2, 3], [4, 5, 6]])
```

- **创建全零或全一数组：** 使用 `np.zeros()` 和 `np.ones()`。需要指定数组的形状。

```
zeros_arr = np.zeros((2, 3)) # 创建一个2x3的全零数组
ones_arr = np.ones((4,))    # 创建一个长度为4的一维全一数组
```

- **创建空数组：** 使用 `np.empty()`。其初始内容是随机的，取决于内存状态，速度最快。

```
empty_arr = np.empty((3, 2)) # 创建一个3x2的空数组
```

- **创建指定范围的数组：** 使用 `np.arange()`，类似于Python的 `range()`。

```
range_arr = np.arange(0, 10, 2) # 创建从0到10，步长为2的数组 [0 2 4 6 8]
```

- **创建等差数列：** 使用 `np.linspace()`。指定起始值、终止值和元素个数。

```
linspace_arr = np.linspace(0, 1, 5) # 创建从0到1，包含5个元素的等差数列 [0.
0.25 0.5 0.75 1. ]
```


- **创建随机数组：** 使用 `np.random` 模块。

```
rand_arr = np.random.rand(2, 2)    # 创建一个2x2的0到1之间的随机浮点数数组
randn_arr = np.random.randn(3, 3) # 创建一个3x3的标准正态分布随机数数组
randint_arr = np.random.randint(0, 10, size=(2, 4)) # 创建一个2x4的0到10之间的
随机整数数组
```

2.4 数组索引和切片

NumPy数组的索引和切片与Python列表类似，但支持多维。

- **一维数组：** 与列表相同。

```
arr = np.array([10, 20, 30, 40, 50])
print(arr[0])    # 输出 10
print(arr[1:4])  # 输出 [20 30 40]
print(arr[::-2]) # 输出 [10 30 50]
```

- **多维数组：** 使用逗号分隔每个维度的索引或切片。

```
arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(arr2d[0, 0])    # 输出 1 (第一行第一列)
print(arr2d[1, :])    # 输出 [4 5 6] (第二行所有列)
print(arr2d[:, 2])    # 输出 [3 6 9] (所有行的第三列)
print(arr2d[0:2, 1:3]) # 输出 [[2 3] [5 6]] (前两行, 第2到第3列)
```

- **布尔索引：** 使用布尔数组来选择元素。

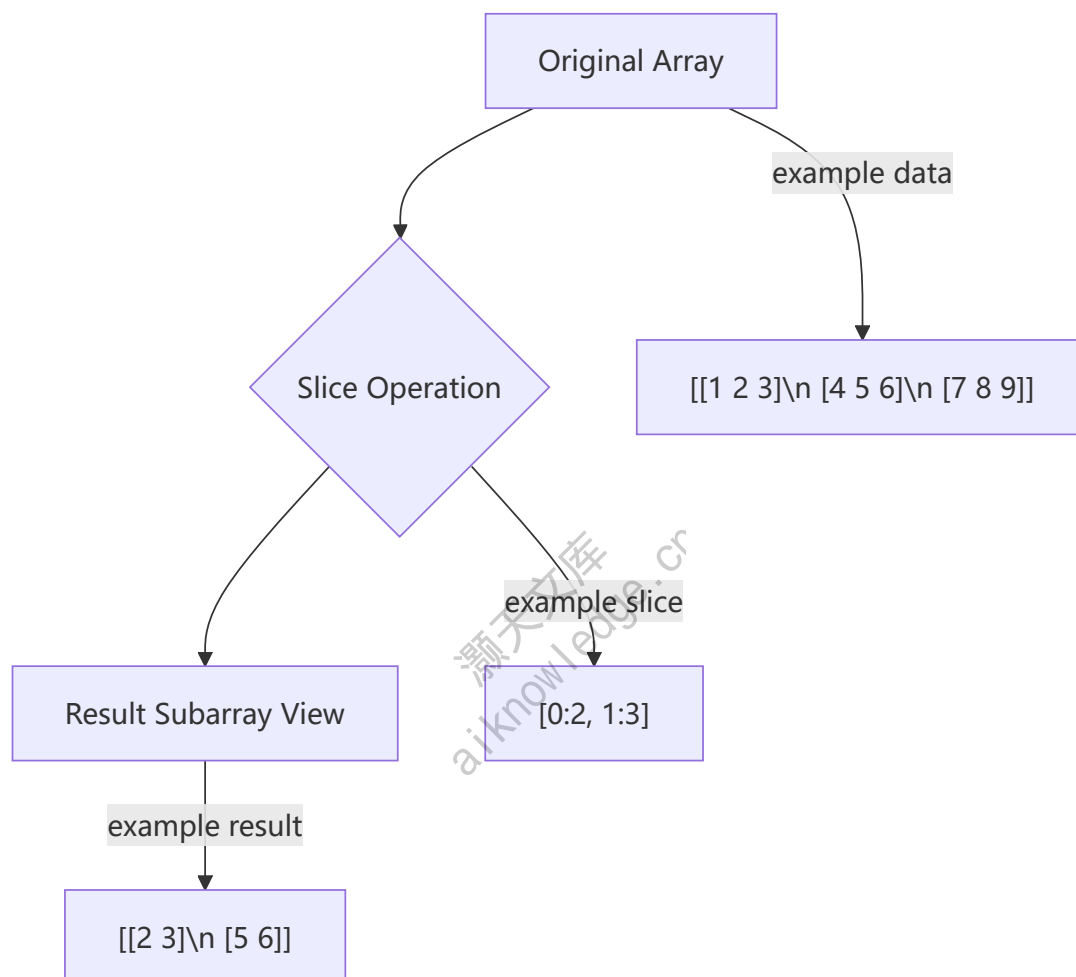
```
arr = np.array([1, 5, 2, 8, 3, 9])
bool_arr = arr > 4
print(bool_arr)    # 输出 [False  True False  True False  True]
print(arr[bool_arr]) # 输出 [5 8 9] (选择对应位置为True的元素)
print(arr[arr % 2 == 0]) # 输出 [2 8] (选择偶数)
```

- **花式索引 (Fancy Indexing)：** 使用整数数组来选择元素或行/列。返回的是原数组的副本。

```
arr = np.arange(10) # [0 1 2 3 4 5 6 7 8 9]
indices = [1, 3, 8]
```

```
print(arr[indices]) # 输出 [1 3 8]
arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(arr2d[[0, 2]]) # 输出 [[1 2 3] [7 8 9]] (选择第0行和第2行)
print(arr2d[[2, 0], [1, 2]]) # 输出 [8 3] (选择 arr2d[2, 1] 和 arr2d[0, 2])
```

下面是一个简单的切片示意图：



注意：NumPy切片通常返回的是原数组的“视图”（view），而不是副本（copy）。修改视图会修改原数组。花式索引返回的是副本。

2.5 数组运算：向量化与广播

NumPy的核心优势在于其高效的向量化运算。这意味着你无需编写显式的循环，就可以对数组中的所有元素执行操作。这些操作由NumPy在底层优化执行。

- **元素级运算：** 算术运算符（+、-、*、/ 等）默认执行元素级运算。

```
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
```

```
print(arr1 + arr2) # 输出 [5 7 9]
print(arr1 * 2)    # 输出 [2 4 6]
print(np.sqrt(arr1)) # 输出 [1.          1.41421356 1.73205081]
```

`np.sqrt` 是一个典型的通用函数 (Universal Function, ufunc)，NumPy提供了大量的 ufunc，包括三角函数、指数函数、对数函数等，它们都能对数组进行元素级操作。

- **矩阵乘法：** 使用 `@` 运算符或 `np.dot()` 函数。

```
mat1 = np.array([[1, 2], [3, 4]])
mat2 = np.array([[5, 6], [7, 8]])
print(mat1 @ mat2) # 矩阵乘法 [[19 22] [43 50]]
print(np.dot(mat1, mat2)) # 同样是矩阵乘法
```

- **广播 (Broadcasting)：** 当对形状不同的数组执行二元运算时，NumPy会尝试使用广播规则将较小的数组“广播”到较大数组的形状，以便进行元素级运算。

广播遵循一系列规则：

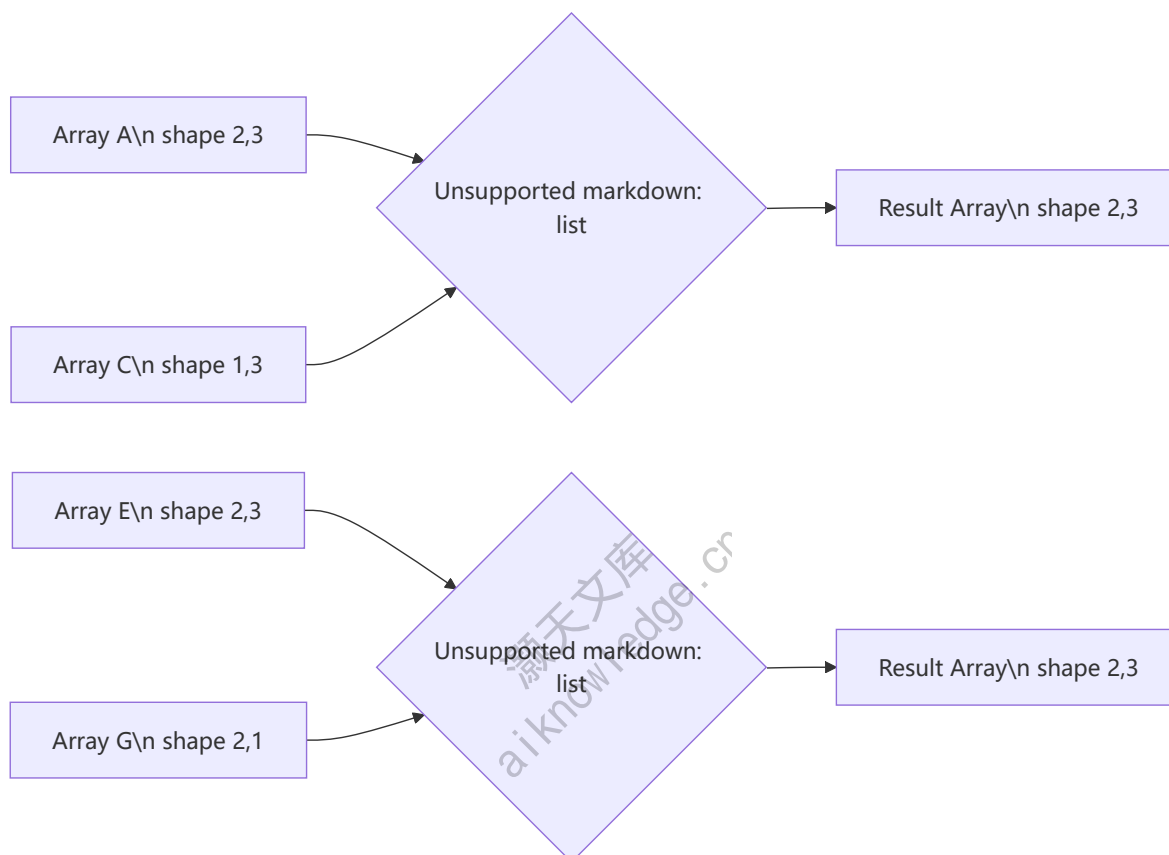
1. 如果两个数组的维度数不同，维度较少的数组会在其前导维度上被填充1。
2. 如果两个数组在某个维度上的大小一致，或其中一个数组在该维度上的大小为1，则它们在该维度上是兼容的。
3. 如果两个数组在任一维度上都不兼容，则会引发错误。

如果维度兼容，大小为1的维度会被拉伸以匹配另一个数组的大小。

```
arr = np.array([[1, 2, 3], [4, 5, 6]]) # 形状 (2, 3)
scalar = 10                               # 形状 (), 广播到 (2, 3)
print(arr + scalar)
# 输出:
# [[11 12 13]
#  [14 15 16]]
arr2d = np.array([[1, 2, 3], [4, 5, 6]]) # 形状 (2, 3)
row_vec = np.array([100, 200, 300])      # 形状 (3,), 广播到 (1, 3), 再广播到 (2, 3)
print(arr2d + row_vec)
# 输出:
# [[101 202 303]
#  [104 205 306]]
col_vec = np.array([[10], [20]])          # 形状 (2, 1), 广播到 (2, 3)
```

```
print(arr2d + col_vec)
# 输出:
# [[11 12 13]
#  [24 25 26]]
```

下面是一个广播的示意图:



2.6 数组形状操作

改变数组的形状是常见需求。

- **重塑 (Reshape):** 使用 `reshape()` 方法或函数。新形状必须与原数组的元素总数兼容。可以使用 `-1` 表示在该维度上自动计算大小。

```
arr = np.arange(12) # [0 1 2 3 4 5 6 7 8 9 10 11]
reshaped_arr = arr.reshape((3, 4)) # 重塑为3x4的二维数组
print(reshaped_arr)
# 输出:
# [[ 0  1  2  3]
#  [ 4  5  6  7]
```

```
# [ 8  9 10 11]]
print(arr.reshape((2, -1))) # 重塑为2行, 列数自动计算 (2, 6)
```

- **展平 (Flatten/Ravel):** 将多维数组展平为一维数组。 `flatten()` 返回副本, `ravel()` 返回视图 (可能)。

```
arr2d = np.array([[1, 2], [3, 4]])
print(arr2d.flatten()) # 输出 [1 2 3 4]
print(arr2d.ravel())   # 输出 [1 2 3 4]
```

- **转置 (Transpose):** 使用 `.T` 属性或 `np.transpose()` 函数。

```
arr2d = np.array([[1, 2], [3, 4]])
print(arr2d.T)
# 输出:
# [[1 3]
#  [2 4]]
```

- **轴交换 (Swapaxes):** 交换两个指定的轴。

```
arr3d = np.arange(24).reshape((2, 3, 4)) # 形状 (2, 3, 4)
# 交换轴 1 和 2 (原来的行和列)
print(arr3d.swapaxes(1, 2).shape) # 输出 (2, 4, 3)
```

2.7 数组组合与分割

- **连接 (Concatenate):** 使用 `np.concatenate()` 沿现有轴连接一系列数组。

```
arr1 = np.array([[1, 2], [3, 4]]) # (2, 2)
arr2 = np.array([[5, 6]])         # (1, 2)
# 沿轴 0 连接 (垂直堆叠)
print(np.concatenate((arr1, arr2), axis=0))
# 输出:
# [[1 2]
#  [3 4]
#  [5 6]]
arr3 = np.array([[7], [8]])       # (2, 1)
# 沿轴 1 连接 (水平堆叠)
```

```
print(np.concatenate((arr1, arr3), axis=1))  
# 输出:  
# [[1 2 7]  
#   [3 4 8]]
```

- **堆叠 (Stacking):** `np.vstack()` (垂直堆叠) 和 `np.hstack()` (水平堆叠) 是 `np.concatenate()` 的便捷函数。 `np.stack()` 沿新轴连接数组。

```
arr1 = np.array([1, 2])  
arr2 = np.array([3, 4])  
print(np.vstack((arr1, arr2))) # 输出 [[1 2] [3 4]] (形状 (2, 2))  
print(np.hstack((arr1, arr2))) # 输出 [1 2 3 4] (形状 (4,))  
print(np.stack((arr1, arr2), axis=0)) # 输出 [[1 2] [3 4]] (形状 (2, 2))  
print(np.stack((arr1, arr2), axis=1)) # 输出 [[1 3] [2 4]] (形状 (2, 2))
```

- **分割 (Splitting):** 使用 `np.split()`, `np.vsplit()` 和 `np.hsplit()`。

```
arr = np.arange(12).reshape((4, 3))  
# 垂直分割成两部分  
arr_split = np.vsplit(arr, 2)  
print(arr_split)  
# 输出:  
# [array([[0, 1, 2],  
#         [3, 4, 5]]),  
#   array([[ 6,  7,  8],  
#         [ 9, 10, 11]])]  
# 水平分割成三部分  
arr_hsplit = np.hsplit(arr, 3)  
print(arr_hsplit)  
# 输出:  
# [array([[0], [3], [6], [9]]),  
#   array([[ 1], [ 4], [ 7], [10]]),  
#   array([[ 2], [ 5], [ 8], [11]])]
```

2.8 统计方法

NumPy数组有许多内置的统计方法，它们会忽略 `NaN` 值（除非特别指定）。

```
arr = np.array([[1, 2, 3], [4, 5, 6]])
print(arr.mean())      # 计算所有元素的平均值 (3.5)
print(arr.mean(axis=0)) # 计算每列的平均值 [2.5 3.5 4.5]
print(arr.sum())        # 计算所有元素的总和 (21)
print(arr.sum(axis=1))  # 计算每行的总和 [ 6 15]
print(arr.std())        # 标准差
print(arr.var())        # 方差
print(arr.min())        # 最小值
print(arr.max(axis=0))  # 每列的最大值
print(arr.argmin())     # 最小值的索引 (展平后的)
print(arr.argmax(axis=1)) # 每行的最大值索引
```

2.9 文件输入/输出

NumPy可以方便地读写磁盘上的数组数据。

- **文本文件：** `np.loadtxt()` 和 `np.savetxt()`。

```
# 假设 data.txt 内容是：
# 1 2 3
# 4 5 6
# data = np.loadtxt('data.txt')
# np.savetxt('output.txt', arr, fmt='%.2f') # 保存为文本，保留两位小数
```

- **二进制文件：** `np.save()` 和 `np.load()`。这是NumPy特有的 `.npy` 格式，保存和加载速度更快，且能保留数组的 `shape` 和 `dtype`。

```
arr = np.arange(10)
np.save('my_array.npy', arr)
loaded_arr = np.load('my_array.npy')
print(loaded_arr) # 输出 [0 1 2 3 4 5 6 7 8 9]
```

2.10 总结

NumPy是Python科学计算和数据处理的基石。其核心 `ndarray` 对象提供高效的多维数组操作能力。通过向量化、广播机制和丰富的函数库，NumPy极大地简化了数值计算任务，并提供了卓越的性能。掌握NumPy是学习Pandas、SciPy、Scikit-learn等后续数据科学库的基础。熟练运用NumPy将为后续的数据分析和建模工作打下坚实的基础。

3. Pandas：数据处理核心

3. Pandas：数据处理核心

在Python数据科学领域，Pandas库是进行数据清洗、转换、分析和可视化的基石。它构建在NumPy之上，提供了高性能、易于使用的数据结构和数据分析工具。本章将深入探讨Pandas的核心概念、主要数据结构以及常用的数据处理技巧。

3.1 Pandas 简介与重要性

Pandas是Python社区广泛使用的数据分析库，其核心在于提供了两种强大的数据结构：Series（一维带标签数组）和DataFrame（二维带标签表格）。这些结构能够方便地处理各种类型的数据，包括时间序列数据、表格数据等。

Pandas的重要性体现在以下几个方面：

- **高效性**：底层使用优化的C/Cython实现，处理大规模数据时性能优异。
- **易用性**：提供了直观的API，使得数据操作变得简单快捷。
- **功能丰富**：涵盖了数据加载、清洗、转换、聚合、合并、切片、索引等数据处理的绝大部分常用功能。
- **兼容性**：与NumPy、SciPy、Matplotlib等其他科学计算库无缝集成。

可以说，掌握Pandas是进行高效数据科学工作的基础。

3.2 Pandas核心数据结构：Series与DataFrame

Pandas主要提供了两种核心数据结构：

3.2.1 Series

Series 是一种一维的带标签数组，可以存储任意数据类型（整数、浮点数、字符串、Python对象等）。它由两部分组成：数据（values）和与之关联的索引（index）。索引用于唯一标识数据中的每个元素。

```
import pandas as pd
import numpy as np
**从列表创建Series，索引默认为0到N-1**
s = pd.Series([1, 3, 5, np.nan, 6, 8])
print("Series 示例（默认索引）:")
print(s)
**从字典创建Series，键成为索引**
s2 = pd.Series({'a': 10, 'b': 20, 'c': 30})
```



```
print("\nSeries 示例 (自定义索引):")
print(s2)
```

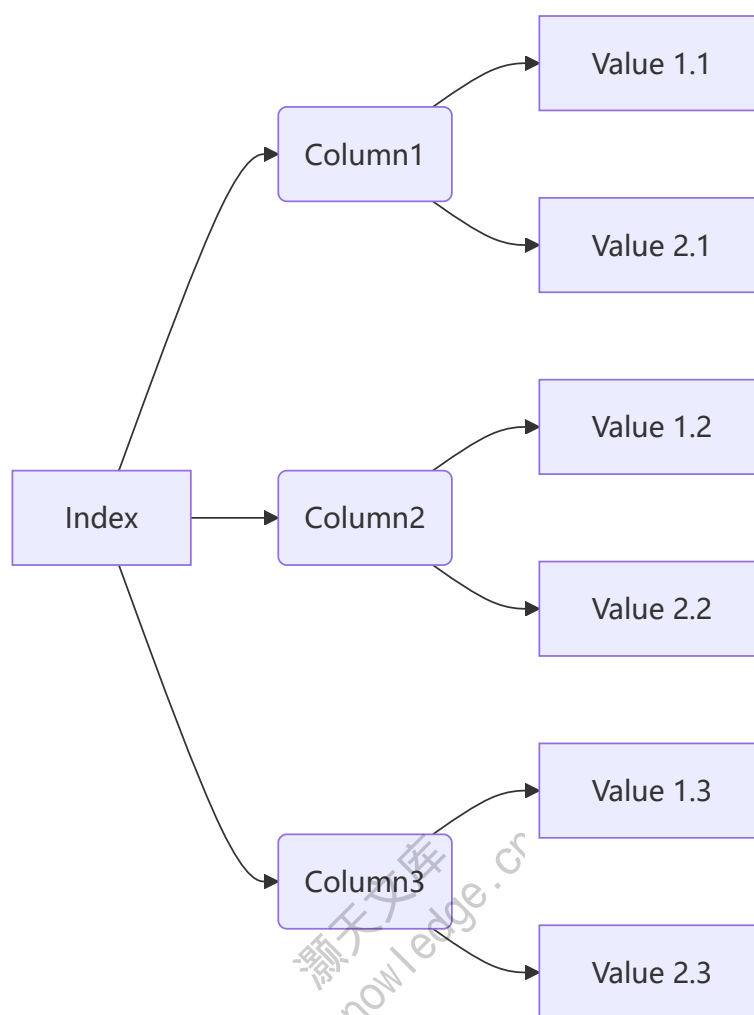
Series 的索引是其强大之处，它允许通过标签快速访问数据。

3.2.2 DataFrame

DataFrame 是一种二维的带标签数据结构，可以看作是由多个 Series 共享同一个索引组成的字典。它具有行索引 (index) 和列标签 (columns)。DataFrame 是Pandas中最常用的数据结构，非常适合表示真实的表格数据 (如Excel表格、数据库表等)。

```
**从字典创建DataFrame**
data = {
    'col1': [1, 2, 3, 4],
    'col2': [5, 6, 7, 8],
    'col3': ['A', 'B', 'C', 'D']
}
df = pd.DataFrame(data)
print("DataFrame 示例 (默认索引):")
print(df)
**从字典创建DataFrame, 并指定索引**
data2 = {
    '城市': ['北京', '上海', '广州', '深圳'],
    '人口': [2154, 2428, 1530, 1303], # 百万人
    '面积': [16410, 6340, 7434, 1997] # 平方千米
}
df2 = pd.DataFrame(data2, index=['BJ', 'SH', 'GZ', 'SZ'])
print("\nDataFrame 示例 (自定义索引):")
print(df2)
```

DataFrame的结构可以通过图简单表示:



这个图展示了DataFrame中，一个共享的Index关联着多个Columns，每个Column可以被视为一个Series，存储对应的数据值。

3.3 数据加载与初步检查

在进行数据分析之前，通常需要从文件加载数据，并进行初步检查以了解数据的概况。

3.3.1 数据加载

Pandas支持多种文件格式的数据加载，最常用的是CSV和Excel。

```
**假设当前目录下有一个名为 'data.csv' 的文件**
**df_csv = pd.read_csv('data.csv')**
**print("\n从CSV文件加载数据:")**
**print(df_csv.head()) # 显示前5行**
**假设当前目录下有一个名为 'data.xlsx' 的文件**
**df_excel = pd.read_excel('data.xlsx', sheet_name='Sheet1')**
**print("\n从Excel文件加载数据:")**
**print(df_excel.head()) # 显示前5行**
```

`read_csv` 和 `read_excel` 函数提供了丰富的参数来控制加载行为，例如指定分隔符(`sep`)、是否有表头(`header`)、指定索引列(`index_col`)、指定数据类型(`dtype`)等。

3.3.2 数据初步检查

加载数据后，需要快速了解其结构、内容和基本统计信息。常用的方法有：

- `.head(n)` / `.tail(n)` : 查看前n行或后n行数据（默认为5）。
- `.info()` : 打印DataFrame的简洁摘要，包括索引类型、列名、非空值数量和数据类型。
- `.describe()` : 生成描述性统计信息，包括计数、均值、标准差、最小值、最大值以及分位数（仅对数值列有效）。
- `.shape` : 返回DataFrame的维度（行数, 列数）。
- `.dtypes` : 返回各列的数据类型。
- `.columns` : 返回列标签列表。
- `.index` : 返回行索引。

****使用之前创建的df DataFrame进行示例****

```
print("\nDataFrame 初步检查:")
```

```
print("头部数据:")
```

```
print(df.head(2))
```

```
print("\n信息概览:")
```

```
df.info()
```

```
print("\n描述性统计:")
```

```
print(df.describe())
```

```
print("\n维度:")
```

```
print(df.shape)
```

```
print("\n数据类型:")
```

```
print(df.dtypes)
```

```
print("\n列名:")
```

```
print(df.columns)
```

```
print("\n索引:")
```

```
print(df.index)
```

这些初步检查步骤对于理解数据集的结构和内容至关重要，有助于规划后续的数据处理步骤。

3.4 索引与选择数据

从DataFrame中选择特定的行、列或单元格是数据处理中最基本的操作之一。Pandas提供了多种灵活的方式来实现这一点。

3.4.1 列选择

选择单列或多列非常直观，类似于字典操作。

```
**选择单列, 返回一个Series**
col1_data = df['col1']
print("\n选择单列 'col1':")
print(col1_data)
**选择多列, 返回一个DataFrame**
cols_data = df[['col1', 'col3']]
print("\n选择多列 ['col1', 'col3']:")
print(cols_data)
```

3.4.2 行选择: loc 和 iloc

Pandas提供了两种主要的基于索引选择行的方法:

- `.loc[]`: **基于标签**进行索引和切片。它使用行索引标签和列标签。
- `.iloc[]`: **基于整数位置**进行索引和切片。它使用行和列的整数位置 (从0开始)。

理解 `loc` 和 `iloc` 的区别非常重要。

```
**假设使用 df2, 其索引是 ['BJ', 'SH', 'GZ', 'SZ']**
print("\n使用 .loc[] 进行标签选择:")
**选择索引为 'SH' 的行**
print("选择行 'SH':")
print(df2.loc['SH'])
**选择索引为 'SH' 和 'GZ' 的行**
print("\n选择行 ['SH', 'GZ']:")
print(df2.loc[['SH', 'GZ']])
**选择索引从 'SH' 到 'SZ' 的行 (包含 'SZ')**
print("\n选择行切片 'SH':'SZ' (包含):")
print(df2.loc['SH':'SZ'])
**选择索引为 'SH' 的行, 以及 '人口' 列**
print("\n选择行 'SH', 列 '人口':")
print(df2.loc['SH', '人口'])
**选择索引为 'SH' 和 'GZ' 的行, 以及 '人口' 和 '面积' 列**
print("\n选择行 ['SH', 'GZ'], 列 ['人口', '面积']:")
print(df2.loc[['SH', 'GZ'], ['人口', '面积']])
print("\n使用 .iloc[] 进行位置选择:")
**选择位置为 1 的行 (对应索引 'SH')**
print("选择位置为 1 的行:")
print(df2.iloc[1])
**选择位置为 1 和 3 的行 (对应索引 'SH', 'SZ')**
print("\n选择位置为 [1, 3] 的行:")
print(df2.iloc[[1, 3]])
**选择位置从 1 到 3 的行 (不包含位置 3, 对应索引 'SH', 'GZ')**
print("\n选择位置切片 1:3 (不包含 3):")
print(df2.iloc[1:3])
```

```
**选择位置为 1 的行, 以及位置为 0 的列 (对应 'SH', '城市')**
print("\n选择位置为 1 的行, 位置为 0 的列:")
print(df2.iloc[1, 0])
**选择位置为 1 和 3 的行, 以及位置为 0 和 2 的列 (对应 ['SH', 'SZ'], ['城市', '面积'])**
print("\n选择位置为 [1, 3] 的行, 位置为 [0, 2] 的列:")
print(df2.iloc[[1, 3], [0, 2]])
```

3.4.3 布尔索引 (条件过滤)

布尔索引是根据某一列或多列的值来选择行的强大方法。它返回一个与DataFrame行数相同的布尔型Series, True表示保留该行, False表示丢弃。

```
**选择人口大于 1500 万的城市**
high_pop_cities = df2[df2['人口'] > 1500]
print("\n人口大于 1500 万的城市:")
print(high_pop_cities)
**选择人口大于 1500 万且面积小于 10000 平方千米的城市**
**注意使用 & 表示逻辑与, | 表示逻辑或**
condition = (df2['人口'] > 1500) & (df2['面积'] < 10000)
filtered_cities = df2[condition]
print("\n人口大于 1500 万且面积小于 10000 平方千米的城市:")
print(filtered_cities)
**使用 .loc[] 结合布尔索引**
**选择人口大于 1500 万的城市的所有列**
filtered_cities_loc = df2.loc[df2['人口'] > 1500, :]
print("\n使用 .loc[] 结合布尔索引选择行:")
print(filtered_cities_loc)
**选择人口大于 1500 万的城市 '城市' 和 '人口' 列**
filtered_cities_loc_subset = df2.loc[df2['人口'] > 1500, ['城市', '人口']]
print("\n使用 .loc[] 结合布尔索引选择行和指定列:")
print(filtered_cities_loc_subset)
```

布尔索引是数据过滤和子集选择最常用的技术之一。

3.5 处理缺失数据

真实世界的数据往往是不完整的, 包含缺失值 (通常用 NaN 表示)。Pandas提供了灵活的方法来检测、删除或填充缺失值。

3.5.1 检测缺失值

isnull() 和 notnull() 方法返回布尔型DataFrame, 指示每个位置是否为缺失值。

```
**创建一个包含缺失值的DataFrame**
df_missing = pd.DataFrame({
```

```
'A': [1, 2, np.nan, 4],
'B': [5, np.nan, np.nan, 8],
'C': ['a', 'b', 'c', 'd']
})
print("\n原始 DataFrame (含缺失值):")
print(df_missing)
print("\n缺失值检测 (.isnull()):")
print(df_missing.isnull())
print("\n非缺失值检测 (.notnull()):")
print(df_missing.notnull())
**统计每列的缺失值数量**
print("\n每列缺失值数量:")
print(df_missing.isnull().sum())
```

3.5.2 删除缺失值

`dropna()` 方法可以删除包含缺失值的行或列。

```
**删除包含任何缺失值的行 (默认 axis=0)**
df_dropped_rows = df_missing.dropna()
print("\n删除含缺失值的行:")
print(df_dropped_rows)
**删除包含任何缺失值的列 (axis=1)**
df_dropped_cols = df_missing.dropna(axis=1)
print("\n删除含缺失值的列:")
print(df_dropped_cols)
**删除所有值都是缺失值的行 (how='all')**
df_dropped_all_na_rows = df_missing.dropna(how='all')
print("\n删除所有值都缺失的行:")
print(df_dropped_all_na_rows)
```

`dropna` 的 `inplace=True` 参数可以直接修改原DataFrame，但通常建议创建新的DataFrame以避免意外修改。

3.5.3 填充缺失值

`fillna()` 方法可以用指定的值或方法填充缺失值。

```
**用固定值填充缺失值**
df_filled_value = df_missing.fillna(0)
print("\n用固定值 0 填充缺失值:")
print(df_filled_value)
**用每列的均值填充缺失值 (仅对数值列有效)**
df_filled_mean = df_missing.fillna(df_missing.mean(numeric_only=True)) #
numeric_only=True 避免对非数值列计算均值报错
print("\n用列均值填充缺失值:")
```

```
print(df_filled_mean)
**使用前向填充 (ffill 或 pad): 用前一个非缺失值填充**
df_filled_ffill = df_missing.fillna(method='ffill')
print("\n使用前向填充 (ffill):")
print(df_filled_ffill)
**使用后向填充 (bfill 或 backfill): 用后一个非缺失值填充**
df_filled_bfill = df_missing.fillna(method='bfill')
print("\n使用后向填充 (bfill):")
print(df_filled_bfill)
```

选择哪种填充方法取决于数据的性质和分析的需求。

3.6 基本数据操作

Pandas提供了许多方便的方法来对数据进行转换和操作。

3.6.1 添加、删除列/行

```
**添加新列**
df_copy = df.copy() # 创建副本, 避免修改原df
df_copy['col4'] = [9, 10, 11, 12]
print("\n添加新列 'col4':")
print(df_copy)
**基于现有列计算新列**
df_copy['col1_plus_col2'] = df_copy['col1'] + df_copy['col2']
print("\n添加计算列 'col1_plus_col2':")
print(df_copy)
**删除列**
df_dropped_col4 = df_copy.drop('col4', axis=1)
print("\n删除列 'col4':")
print(df_dropped_col4)
**删除行 (通过索引或位置)**
**删除索引为 0 的行**
df_dropped_row0 = df.drop(0, axis=0)
print("\n删除索引为 0 的行:")
print(df_dropped_row0)
**删除索引为 [1, 3] 的行**
df_dropped_rows_1_3 = df.drop([1, 3], axis=0)
print("\n删除索引为 [1, 3] 的行:")
print(df_dropped_rows_1_3)
```

3.6.2 应用函数

`apply()` 方法允许将函数应用于 Series 或 DataFrame 的行/列。

```
**定义一个函数，将数值转换为字符串并添加前缀**
def format_value(x):
    return f"Value_{x}"
**将函数应用于单列 (Series)**
df_copy['col1_formatted'] = df_copy['col1'].apply(format_value)
print("\n对 'col1' 应用函数:")
print(df_copy)
**将函数应用于 DataFrame 的每一行 (axis=1)**
**假设函数对行进行某种计算或转换**
def row_sum_and_label(row):
    return f"{row['col3']}_{row['col1'] + row['col2']}"
df_copy['row_summary'] = df_copy.apply(row_sum_and_label, axis=1)
print("\n对每行应用函数:")
print(df_copy)
```

3.6.3 排序

可以按值或按索引对 DataFrame 进行排序。

```
**按 'col1' 列的值升序排序**
df_sorted_col1 = df.sort_values(by='col1')
print("\n按 'col1' 升序排序:")
print(df_sorted_col1)
**按 'col2' 列的值降序排序**
df_sorted_col2_desc = df.sort_values(by='col2', ascending=False)
print("\n按 'col2' 降序排序:")
print(df_sorted_col2_desc)
**按索引排序**
df_sorted_index = df2.sort_index()
print("\n按索引升序排序:")
print(df_sorted_index)
df_sorted_index_desc = df2.sort_index(ascending=False)
print("\n按索引降序排序:")
print(df_sorted_index_desc)
```

3.6.4 唯一值与计数

了解列中有哪些唯一值以及每个唯一值出现的频率非常有用。

```
**获取 'col3' 列的唯一值**
unique_col3 = df['col3'].unique()
print("\n'col3' 列的唯一值:")
print(unique_col3)
**获取 'col3' 列每个唯一值出现的次数**
```



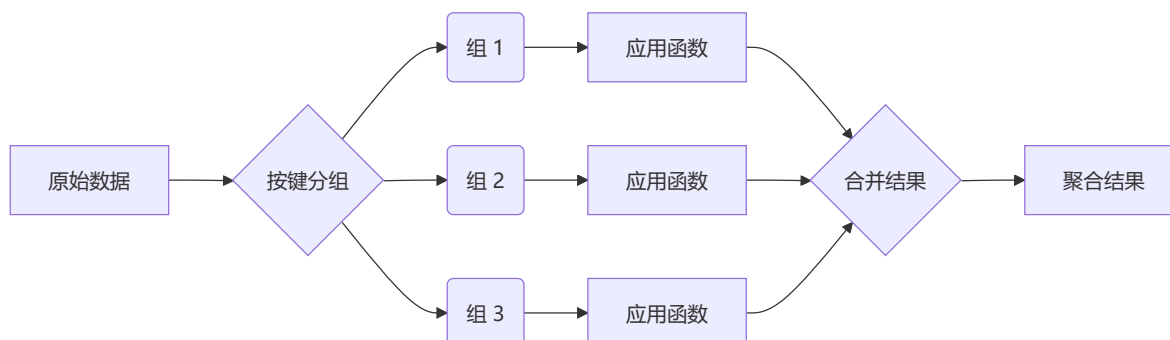
```
value_counts_col3 = df['col3'].value_counts()
print("\n'col3' 列的值计数:")
print(value_counts_col3)
```

3.7 分组与聚合

分组 (Grouping) 是Pandas中一个非常强大和常用的操作，它允许你根据一个或多个键将 DataFrame拆分成多个组，然后对每个组独立地应用函数（聚合、转换或过滤），最后将结果合并。这遵循“split-apply-combine”（拆分-应用-合并）的范式。

```
**创建一个用于分组的DataFrame**
data_group = {
    '类别': ['A', 'B', 'A', 'C', 'B', 'A', 'C'],
    '值1': [10, 15, 12, 18, 20, 11, 22],
    '值2': [1, 2, 3, 4, 5, 6, 7]
}
df_group = pd.DataFrame(data_group)
print("\n用于分组的 DataFrame:")
print(df_group)
**按 '类别' 列进行分组**
grouped_by_category = df_group.groupby('类别')
**对每个组计算 '值1' 的均值**
mean_value1_by_category = grouped_by_category['值1'].mean()
print("\n按 '类别' 分组后计算 '值1' 的均值:")
print(mean_value1_by_category)
**对每个组计算多个聚合函数**
aggregated_stats = grouped_by_category.agg({
    '值1': ['sum', 'mean', 'count'],
    '值2': ['min', 'max']
})
print("\n按 '类别' 分组后计算多个聚合统计量:")
print(aggregated_stats)
```

分组聚合的流程可以用图表示：



这个图清晰地展示了数据如何先被拆分到不同的组，然后对每个组独立执行操作，最后将结果重新组合起来。

3.8 合并与连接数据

在实际应用中，数据通常分散在多个文件或表中。Pandas提供了多种方法来合并或连接 DataFrame，类似于数据库中的JOIN操作。最常用的是 `pd.merge()` 函数。

```
**创建两个示例 DataFrame**
df_left = pd.DataFrame({
    'key': ['K0', 'K1', 'K2', 'K3'],
    'A': ['A0', 'A1', 'A2', 'A3'],
    'B': ['B0', 'B1', 'B2', 'B3']
})
df_right = pd.DataFrame({
    'key': ['K0', 'K1', 'K2', 'K4'],
    'C': ['C0', 'C1', 'C2', 'C4'],
    'D': ['D0', 'D1', 'D2', 'D4']
})
print("\n左 DataFrame:")
print(df_left)
print("\n右 DataFrame:")
print(df_right)
**基于 'key' 列进行内连接 (inner join)**
**只保留两个 DataFrame 中 'key' 列都存在的行**
merged_df_inner = pd.merge(df_left, df_right, on='key', how='inner')
print("\n内连接 (inner join):")
print(merged_df_inner)
**基于 'key' 列进行左连接 (left join)**
**保留左 DataFrame 的所有行，右 DataFrame 中没有匹配的用 NaN 填充**
merged_df_left = pd.merge(df_left, df_right, on='key', how='left')
print("\n左连接 (left join):")
print(merged_df_left)
```

`pd.merge()` 函数通过 `on` 参数指定用于连接的列，`how` 参数指定连接类型 ('inner', 'left', 'right', 'outer')。

3.9 数据保存

完成数据处理后，通常需要将结果保存到文件。

```
**将 DataFrame 保存为 CSV 文件**
**merged_df_inner.to_csv('merged_data.csv', index=False) # index=False 不保存索引**
**将 DataFrame 保存为 Excel 文件**
```

```
**merged_df_left.to_excel('merged_data.xlsx', index=False,  
sheet_name='Sheet1')**
```

`to_csv` 和 `to_excel` 也提供了多种参数来控制输出格式。

3.10 总结

本章详细介绍了Pandas库作为Python数据科学核心工具的关键方面。我们学习了其基本数据结构 Series 和 DataFrame，掌握了如何加载和初步检查数据，理解了基于标签 (loc) 和位置 (iloc) 以及布尔索引进行数据选择的方法。此外，我们还探讨了如何处理缺失值、执行基本的数据操作（添加/删除列、应用函数、排序）以及强大的分组聚合功能。最后，我们简要介绍了合并 DataFrame和保存数据的方法。

掌握这些Pandas的核心概念和技巧，将为你后续进行更复杂的数据分析、特征工程和建模工作打下坚实的基础。实践是最好的学习方法，建议读者多动手尝试使用Pandas处理实际数据集。

4. 数据清洗与预处理

4. 数据清洗与预处理

在任何数据科学项目中，原始数据往往是混乱、不完整、不一致且包含噪声的。直接使用这样的数据进行分析或构建模型，轻则导致结果偏差，重则模型失效。因此，数据清洗与预处理是获取高质量数据的必经之路，它为后续的数据探索、特征工程、建模和评估奠定坚实的基础。

本章将深入探讨使用Python及其核心数据科学库（如Pandas、NumPy和Scikit-learn）进行数据清洗和预处理的常用技术和方法。

4.1 理解数据：初步检查与探索

在开始清洗之前，首先需要对数据有一个全面的了解。这一步包括检查数据的基本结构、数据类型、是否存在缺失值、重复值以及初步的数据分布。

常用的Pandas方法包括：

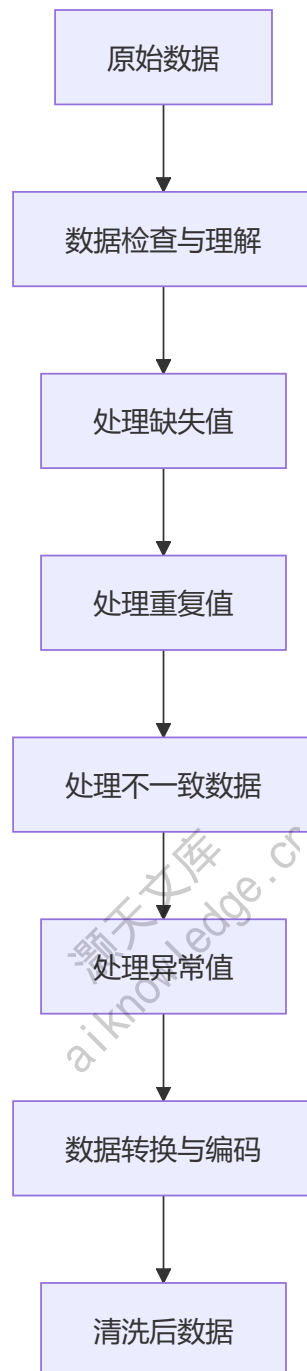
- `df.head()` / `df.tail()` : 查看数据集的前几行或后几行，快速了解数据外观。
- `df.info()` : 获取数据集的摘要信息，包括列名、非空值数量、数据类型以及内存使用情况。这是识别缺失值和数据类型问题的关键一步。
- `df.describe()` : 生成描述性统计信息，如计数、均值、标准差、最小值、最大值以及四分位数。这有助于发现潜在的异常值或数据分布问题（仅适用于数值列）。
- `df.isnull().sum()` : 计算每列的缺失值数量。这是量化缺失情况最直接的方法。
- `df.duplicated().sum()` : 计算重复行的数量。
- `df.dtypes` : 查看每列的数据类型。

- `df['column'].value_counts()` : 查看某一列（特别是类别型列）的唯一值及其出现频率。
这有助于发现类别不一致或输入错误。

通过这些初步检查，我们可以对数据的“健康状况”有一个初步判断，并规划后续的清洗步骤。

```
import pandas as pd
import numpy as np
**创建一个示例DataFrame**
data = {
    'ID': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
    'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eve', 'Alice', 'Frank',
            'Grace', 'Heidi', 'Ivy'],
    'Age': [25, 30, np.nan, 35, 28, 25, 40, 32, np.nan, 29],
    'City': ['New York', 'London', 'Paris', 'Tokyo', 'Seoul', 'New York',
            'London', 'Paris', 'Tokyo', 'Seoul'],
    'Salary': [50000, 60000, 70000, 80000, 55000, 50000, 90000, 65000, 75000,
              60000],
    'JoinDate': ['2020-01-15', '2019-05-20', '2021-11-10', '2018-08-01', '2022-
03-25', '2020-01-15', '2017-06-30', '2021-02-18', '2019-09-05', '2022-07-12'],
    'Rating': [4.5, 3.9, 4.1, 4.8, 4.0, 4.5, 4.9, 4.2, 4.3, 4.1]
}
df = pd.DataFrame(data)
print("--- 数据概览 ---")
print(df.head())
print("\n--- 数据信息 ---")
df.info()
print("\n--- 描述性统计 ---")
print(df.describe())
print("\n--- 每列缺失值数量 ---")
print(df.isnull().sum())
print("\n--- 重复行数量 ---")
print(df.duplicated().sum())
print("\n--- City列唯一值及其计数 ---")
print(df['City'].value_counts())
```

图示：数据清洗与预处理基本流程



4.2 处理缺失值 (Missing Values)

缺失值是实际数据集中最常见的问题之一。处理缺失值的方法取决于缺失的类型、数量以及对分析或模型的影响。

4.2.1 识别缺失值

前面已经介绍了 `df.isnull().sum()`，此外，还可以使用 `df.isnull()` 生成一个布尔型 DataFrame，结合 `.any()` 或 `.all()` 来查找包含缺失值的行或列。

```
**查找包含缺失值的行**
rows_with_nan = df[df.isnull().any(axis=1)]
print("\n--- 包含缺失值的行 ---")
print(rows_with_nan)
```

4.2.2 处理策略

主要策略包括：

1. 删除 (Dropping):

- **删除包含缺失值的行:** 如果缺失值占比较小，或者这些行对分析不重要，可以直接删除。使用 `df.dropna()`。
- **删除包含缺失值的列:** 如果某一列缺失值过多，或者该列对分析不重要，可以删除整列。使用 `df.dropna(axis=1)`。

```
# 删除包含任何缺失值的行
df_dropped_rows = df.dropna()
print("\n--- 删除包含缺失值的行后的数据 ---")
print(df_dropped_rows)

# 删除缺失值比例过高的列（例如，假设 Age 列缺失值过多）
# 这里 Age 只有两个缺失，不适合删除列，仅作参考
# df_dropped_cols = df.dropna(axis=1, thresh=len(df)*0.8) # thresh参数指定非缺失值的最小数量
```

注意: 删除数据可能导致信息丢失，特别是当缺失不是随机分布时，可能引入偏差。

2. 填充 (Imputation):

- **使用固定值填充:** 用一个常数（如0，未知）填充缺失值。
- **使用统计量填充:** 用列的均值、中位数或众数填充。适用于数值型数据。中位数对异常值不敏感，通常是比均值更稳健的选择。
- **使用前后值填充:** 使用前一个非缺失值（`ffill` 或 `pad`）或后一个非缺失值（`bfill` 或 `backfill`）填充。适用于时间序列数据或有序数据。
- **基于模型预测填充:** 使用其他列的数据构建模型来预测缺失值。这更复杂，但可能更准确。
- **插值填充:** 基于相邻值进行线性或其他方式的插值。

```
# 使用均值填充 'Age' 列的缺失值
df['Age_mean_filled'] = df['Age'].fillna(df['Age'].mean())
print("\n--- 使用均值填充 Age 列 ---")
print(df[['Age', 'Age_mean_filled']])

# 使用中位数填充 'Age' 列的缺失值
df['Age_median_filled'] = df['Age'].fillna(df['Age'].median())
print("\n--- 使用中位数填充 Age 列 ---")
print(df[['Age', 'Age_median_filled']])

# 使用特定值填充（例如，用 'Unknown' 填充缺失的类别）
# 假设 City 列可能有缺失值，这里只是演示
df['City_filled'] = df['City'].fillna('Unknown')

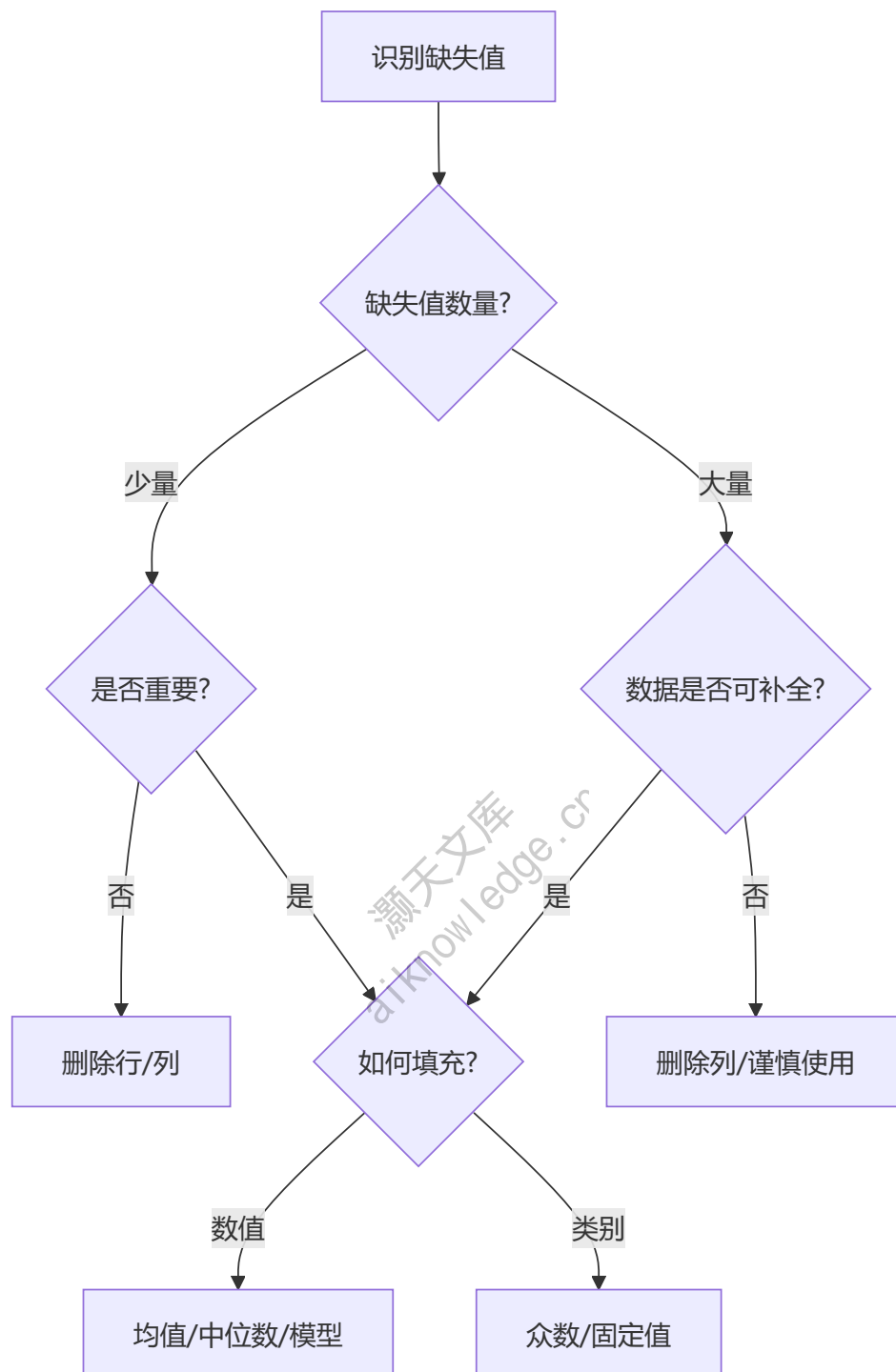
# 使用 ffill 填充（假设数据按时间排序）
df['Value'].fillna(method='ffill', inplace=True)
```

使用 Scikit-learn 的 Imputer:

Scikit-learn 提供了 `SimpleImputer` 类，可以方便地使用均值、中位数、众数或常数进行填充，并且可以处理多种数据类型。这在构建机器学习管线时特别有用。

```
from sklearn.impute import SimpleImputer
# 实例化 Imputer，使用中位数策略
imputer = SimpleImputer(missing_values=np.nan, strategy='median')
# 提取需要填充的列（通常是数值列）
age_data = df[['Age']] # SimpleImputer 需要二维数组
# 训练 Imputer 并转换数据
df['Age_sklearn_filled'] = imputer.fit_transform(age_data)
print("\n--- 使用 sklearn SimpleImputer (中位数) 填充 Age 列 ---")
print(df[['Age', 'Age_sklearn_filled']])
```

图示：缺失值处理决策流程



4.3 处理重复值 (Duplicate Values)

重复的行可能源于数据采集错误、合并数据集等。它们会夸大某些观察结果的权重，导致统计分析或模型训练产生偏差。

4.3.1 识别重复值

`df.duplicated()` 方法返回一个布尔型 Series，指示每一行是否是前一行（默认）的重复。

`df.duplicated().sum()` 计算总的重复行数。

可以通过 `subset` 参数指定在哪些列上检查重复, 通过 `keep` 参数指定保留哪个重复项 (`first`, `last`, `False` 表示标记所有重复项)。

```
print("\n--- 标记重复行 ---")
print(df.duplicated())
print("\n--- 基于 Name 和 Age 列检查重复 ---")
print(df.duplicated(subset=['Name', 'Age']))
```

4.3.2 删除重复值

使用 `df.drop_duplicates()` 方法删除重复行。同样可以使用 `subset` 和 `keep` 参数。

```
**删除所有完全重复的行, 保留第一次出现的行**
df_no_duplicates = df.drop_duplicates()
print("\n--- 删除重复行后的数据 ---")
print(df_no_duplicates)
**删除基于 Name 和 Age 列重复的行, 保留第一次出现的行**
df_no_duplicates_subset = df.drop_duplicates(subset=['Name', 'Age'])
print("\n--- 删除基于 Name 和 Age 重复的行后的数据 ---")
print(df_no_duplicates_subset)
```

4.4 处理不一致数据 (Inconsistent Data)

数据不一致可能表现为数据类型错误、格式不统一、类别名称拼写错误等。

4.4.1 数据类型转换

`df.info()` 已经帮助我们识别了数据类型。如果某一列的数据类型不正确 (例如, 表示数字的列被存储为字符串, 或者日期被存储为对象), 需要进行转换。

使用 `df['column'].astype('desired_type')` 进行类型转换。对于日期, 最好使用 `pd.to_datetime()`。

```
print("\n--- 原始 JoinDate 类型 ---")
print(df['JoinDate'].dtype)
**将 JoinDate 列转换为 datetime 类型**
df['JoinDate'] = pd.to_datetime(df['JoinDate'])
print("\n--- 转换后 JoinDate 类型 ---")
print(df['JoinDate'].dtype)
**示例: 将一个可能包含字符串的数字列转换为数值 (errors='coerce' 会将无法转换的值变为 NaN)**
**df['NumericColumn'] = pd.to_numeric(df['NumericColumn'], errors='coerce')**
```

4.4.2 字符串处理与格式统一

类别型数据或自由文本输入的数据经常存在格式不一致问题，如大小写混用、前后空格、拼写错误或同义词。

使用 Pandas 的 `.str` 访问器进行字符串操作：

- `.str.lower()` / `.str.upper()` : 转换为小写或大写。
- `.str.strip()` : 移除前后空格。
- `.str.replace(old, new)` : 替换子字符串。
- `.str.contains(pattern)` : 检查是否包含模式。
- `.str.extract(pattern)` : 提取符合模式的部分。

```
**示例: City 列可能存在大小写不一致或空格**
df['City'] = df['City'].str.lower().str.strip()
print("\n--- 清洗后的 City 列唯一值及其计数 ---")
print(df['City'].value_counts())
**示例: 替换拼写错误或同义词**
**df['City'] = df['City'].replace({'ny': 'new york', 'la': 'los angeles'})**
```

4.4.3 类别数据标准化

检查类别列的唯一值 (`.value_counts()`)，手动或通过映射、替换来统一不一致的类别名称。

```
**假设发现 'new york' 和 'newyork' 都是指同一个地方**
**df['City'] = df['City'].replace({'newyork': 'new york'})**
**print("\n--- 替换后的 City 列唯一值及其计数 ---")**
**print(df['City'].value_counts())**
```

4.5 处理异常值 (Outliers)

异常值是数据集中与其他观测值显著不同的数据点。它们可能是数据录入错误，也可能代表了真实的极端情况。异常值会影响统计量的计算（如均值、标准差）以及某些对异常值敏感的模式（如线性回归）。

4.5.1 识别异常值

- **可视化方法:** 箱线图 (`boxplot`)、直方图 (`hist`)、散点图 (`scatter plot`) 是识别异常值的有效工具。
- **统计方法:**
 - **Z-score:** 对于近似正态分布的数据，计算每个数据点与均值的标准差距离。Z-score 超过某个阈值（如 3）的点被视为异常值。

- **IQR (Interquartile Range):** 对于非正态分布数据更稳健。计算第三四分位数 (Q3) 和第一四分位数 (Q1) 之间的差值 ($IQR = Q3 - Q1$)。落在 $[Q1 - 1.5 \times IQR, Q3 + 1.5 \times IQR]$ 范围之外的点被视为异常值。

```
import matplotlib.pyplot as plt
import seaborn as sns
**绘制 Age 列的箱线图**
plt.figure(figsize=(6, 4))
sns.boxplot(x=df['Age_median_filled']) # 使用填充后的 Age 列进行演示
plt.title('Boxplot of Age')
plt.show()
**绘制 Salary 列的箱线图**
plt.figure(figsize=(6, 4))
sns.boxplot(x=df['Salary'])
plt.title('Boxplot of Salary')
plt.show()
**使用 IQR 方法识别 Salary 列的异常值**
Q1 = df['Salary'].quantile(0.25)
Q3 = df['Salary'].quantile(0.75)
IQR = Q3 - Q1
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR
outliers_iqr = df[(df['Salary'] < lower_bound) | (df['Salary'] > upper_bound)]
print("\n--- 使用 IQR 方法识别的 Salary 异常值 ---")
print(outliers_iqr)
**使用 Z-score 方法识别异常值 (适用于近似正态分布)**
from scipy.stats import zscore
**假设 Age 列近似正态分布**
**z_scores = np.abs(zscore(df['Age'].dropna())) # 计算 Z-score, 忽略 NaN**
**threshold = 3**
**outliers_zscore = df[np.abs(zscore(df['Age_median_filled'])) > threshold] # 在填充后的列上计算**
**print("\n--- 使用 Z-score 方法识别的 Age 异常值 ---")**
**print(outliers_zscore) # 在这个小样本中可能没有 Z-score 异常值**
```

4.5.2 处理策略

处理异常值的方法取决于其性质和对分析的影响:

1. **删除:** 如果异常值是明显的错误且数量很少, 可以直接删除包含异常值的行。
2. **转换:** 对数据进行数学转换 (如取对数、平方根转换) 可以减小异常值的影响, 使数据分布更接近正态。
3. **封顶/封底 (Capping/Flooring):** 将超出上下限的异常值替换为上下限的阈值 (例如, 将所有高于 $Q3 + 1.5 \times IQR$ 的值替换为 $Q3 + 1.5 \times IQR$)。

4. **视为缺失值:** 将异常值标记为缺失值, 然后使用缺失值处理方法进行处理。
5. **保留:** 如果异常值代表了真实的极端情况且对分析有意义, 可以选择保留它们, 但可能需要选择对异常值不敏感的模型 (如基于树的模型)。

```
**示例: 使用封顶/封底处理 Salary 异常值 (基于 IQR)**
df['Salary_capped'] = df['Salary'].clip(lower=lower_bound, upper=upper_bound)
print("\n--- 使用封顶/封底处理 Salary 异常值 ---")
print(df[['Salary', 'Salary_capped']])
**示例: 删除包含 Salary 异常值的行**
df_no_outliers = df[(df['Salary'] >= lower_bound) & (df['Salary'] <=
upper_bound)]**
**print("\n--- 删除 Salary 异常值后的数据 ---")**
**print(df_no_outliers)**
```

4.6 数据转换与编码 (Data Transformation and Encoding)

在数据清洗的最后阶段, 可能需要对数据进行转换, 使其更适合模型训练。这包括特征缩放 (如果模型对特征尺度敏感) 和类别特征编码。虽然特征缩放更常被视为特征工程的一部分, 但类别编码是数据预处理中处理非数值数据的必要步骤。

4.6.1 类别特征编码 (Categorical Feature Encoding)

大多数机器学习模型只能处理数值数据, 因此需要将类别型特征转换为数值表示。

1. **标签编码 (Label Encoding):** 将每个唯一的类别映射为一个整数。适用于**有序类别** (如“小”、“中”、“大”)。如果用于无序类别, 模型可能会错误地认为整数值之间存在顺序关系。

```
from sklearn.preprocessing import LabelEncoder
# 假设 City 是有序类别 (虽然实际不是, 仅作参考)
# le = LabelEncoder()
# df['City_LabelEncoded'] = le.fit_transform(df['City'])
# print("\n--- Label Encoding 示例 ---")
# print(df[['City', 'City_LabelEncoded']])
```

2. **独热编码 (One-Hot Encoding):** 为每个唯一的类别创建一个新的二进制 (0或1) 列。适用于**无序类别**。这是处理类别特征最常用的方法之一, 避免了模型对类别之间顺序的误解。Pandas 的 `get_dummies()` 函数是实现独热编码的便捷方式。

```
# 对 City 列进行独热编码
df_one_hot = pd.get_dummies(df, columns=['City'], prefix='City')
print("\n--- One-Hot Encoding 示例 ---")
```

```
print(df_one_hot.head())
# 使用 drop_first=True 避免多重共线性 (dummy variable trap)
df_one_hot_dropped = pd.get_dummies(df, columns=['City'], prefix='City',
drop_first=True)
print("\n--- One-Hot Encoding (drop_first=True) 示例 ---")
print(df_one_hot_dropped.head())
```

注意: `get_dummies` 会自动处理非数值列, 将它们转换为独热编码。

4.6.2 特征缩放 (Feature Scaling)

对于一些对特征尺度敏感的模型 (如支持向量机、KNN、线性回归、神经网络), 需要将不同尺度的数值特征缩放到相似的范围。常见方法有标准化 (Standardization, Z-score scaling) 和归一化 (Normalization, Min-Max scaling)。这部分更偏向特征工程, 但在预处理阶段也常进行。

- **标准化:** 将数据缩放到均值为 0, 标准差为 1 的分布。使用

`sklearn.preprocessing.StandardScaler`。

- **归一化:** 将数据缩放到一个固定范围, 通常是 [0, 1]。使用

`sklearn.preprocessing.MinMaxScaler`。

```
from sklearn.preprocessing import StandardScaler, MinMaxScaler
**示例: 对 Salary 列进行标准化**
scaler_std = StandardScaler()
df['Salary_Standardized'] = scaler_std.fit_transform(df[['Salary']]) # 需要二维输入
print("\n--- Salary 标准化示例 ---")
print(df[['Salary', 'Salary_Standardized']].head())
**示例: 对 Salary 列进行归一化**
scaler_minmax = MinMaxScaler()
df['Salary_MinMaxScaled'] = scaler_minmax.fit_transform(df[['Salary']]) # 需要二维输入
print("\n--- Salary 归一化示例 ---")
print(df[['Salary', 'Salary_MinMaxScaled']].head())
```

4.7 清洗流程的整合与自动化

在实际项目中, 数据清洗通常是一个迭代的过程, 并且需要将多个步骤组合起来。为了提高效率和可重复性, 可以将清洗步骤整合成函数或使用 Scikit-learn 的 `Pipeline`。`Pipeline` 特别适合将预处理步骤和模型训练连接起来。

```
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
**定义不同列的预处理步骤**
```

```

**对于数值列，使用中位数填充和标准化**
numerical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler())
])
**对于类别列，使用众数填充和独热编码**
categorical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='most_frequent')),
    ('onehot', OneHotEncoder(handle_unknown='ignore')) # handle_unknown='ignore'
    处理训练集中未出现的类别
])
**获取数值列和类别列的列表**
numerical_features = ['Age', 'Salary', 'Rating']
categorical_features = ['City'] # 排除 ID, Name, JoinDate (可能需要单独处理或删除)
**创建 ColumnTransformer 来并行处理不同类型的列**
preprocessor = ColumnTransformer(
    transformers=[
        ('num', numerical_transformer, numerical_features),
        ('cat', categorical_transformer, categorical_features)
    ],
    remainder='passthrough' # 保留未处理的列 (如 ID, Name, JoinDate)
)
**将预处理器集成到 Pipeline 中 (例如，与一个模型结合)**
**from sklearn.linear_model import LogisticRegression**
**model = Pipeline(steps=[('preprocessor', preprocessor),**
**('classifier', LogisticRegression())])****
**示例：仅应用预处理**
**注意：ColumnTransformer 输出的是 Numpy 数组，需要转换回 DataFrame**
**df_processed_array = preprocessor.fit_transform(df)**
**print("\n--- 使用 ColumnTransformer 进行预处理 ---")**
**print(df_processed_array[:5]) # 打印前5行数组结果**
**如果需要将结果转回 DataFrame (更复杂，需要保留列名)**
**processed_columns = numerical_features +
list(preprocessor.named_transformers_['cat']['onehot'].get_feature_names_in()) #
示例获取列名**
**df_processed = pd.DataFrame(df_processed_array, columns=processed_columns) #
列名可能需要更复杂的处理来匹配 OneHotEncoder 输出**
**print(df_processed.head())**

```

使用 `ColumnTransformer` 和 `Pipeline` 可以确保在训练集上 `fit` 预处理器，然后在训练集和测试集上 `transform`，避免数据泄露。

4.8 总结

数据清洗与预处理是数据科学流程中耗时但至关重要的一步。它不仅仅是机械地执行代码，更需要结合对业务背景和数据本身的理解，审慎地选择合适的处理策略。本章详细介绍了识别和处理缺失值、重复值、不一致数据和异常值的常用技术，以及类别特征编码等基本的数据转换方法。

掌握这些技能，是构建可靠、高效数据科学解决方案的基础。记住，“Garbage In, Garbage Out”——高质量的数据是高质量结果的前提。

5. 数据探索与可视化

5. 数据探索与可视化

5.1 引言：为什么需要数据探索与可视化？

在数据科学项目中，获取和清洗数据仅仅是第一步。一旦数据准备就绪，我们需要深入了解数据的内在结构、模式、趋势以及潜在的问题。这个过程就是**数据探索性分析 (Exploratory Data Analysis, EDA)**。EDA 的核心在于利用统计摘要和可视化技术来揭示数据的特征，而无需事先对数据做任何假设。

数据探索的主要目标包括：

1. **理解数据结构 and 内容：** 数据有多少行、多少列？每列的数据类型是什么？是否存在缺失值或异常值？
2. **识别数据质量问题：** 缺失值、重复值、不一致的数据格式、异常值等。
3. **发现变量之间的关系：** 两个变量是正相关、负相关还是没有明显关系？某个变量是否会影响另一个变量的分布？
4. **识别重要特征：** 哪些特征可能对后续的建模任务（如预测、分类）更重要？
5. **检验假设：** 数据是否符合某些统计假设？
6. **指导特征工程和模型选择：** 通过对数据的深入了解，我们可以更好地进行特征转换、创建新特征，并选择合适的模型算法。

数据可视化是 EDA 中最强大、最直观的工具。“一张图胜过千言万语”，通过将数据以图形形式展现出来，我们可以更容易地发现隐藏在大量数字中的模式、趋势和异常，这些信息往往难以从原始数据表格或简单的统计摘要中直接获取。可视化不仅帮助我们理解数据，也是向他人传达发现和洞察的有效方式。

本章将重点介绍如何使用 Python 中最常用的数据科学库——**Pandas** 进行数据探索，以及如何使用 **Matplotlib** 和 **Seaborn** 进行数据可视化。

5.2 数据探索基础：使用 Pandas

Pandas 库提供了高效的数据结构（DataFrame 和 Series）以及丰富的数据操作和分析功能，是进行数据探索的首选工具。

假设我们已经将数据加载到一个 Pandas DataFrame 中，通常命名为 `df`。

5.2.1 查看数据概览

开始 EDA 的第一步通常是快速浏览数据的基本信息。

- **查看前几行/后几行:**

```
import pandas as pd
# 假设df是你的DataFrame
# df = pd.read_csv('your_data.csv') # 如果数据还未加载
print("数据前5行:")
print(df.head())
print("\n数据后5行:")
print(df.tail())
```

`df.head(n)` 和 `df.tail(n)` 默认显示前/后5行, 可以指定参数 `n` 显示任意行数。这能让你对数据的外观、列名和数据格式有一个初步印象。

- **查看数据维度:**

```
print("\n数据维度 (行数, 列数):")
print(df.shape)
```

`df.shape` 返回一个元组, 表示 DataFrame 的行数和列数。

- **查看数据信息:**

```
print("\n数据信息概览:")
df.info()
```

`df.info()` 提供了非常重要的信息, 包括:

- 每列的名称。
- 非空值的数量 (`non-null count`), 这有助于快速发现哪些列存在缺失值。
- 每列的数据类型 (`dtype`), 如 `int64`, `float64`, `object` (通常是字符串), `datetime64` 等。数据类型是否正确对于后续分析至关重要。
- 内存使用情况。

- **查看描述性统计:**

```
print("\n数值列描述性统计:")
print(df.describe())
```



```
print("\n非数值列描述性统计:")
print(df.describe(include='object')) # 或者 include='all' 包含所有列
```

`df.describe()` 默认计算数值列的描述性统计信息, 包括: 计数 (`count`)、均值 (`mean`)、标准差 (`std`)、最小值 (`min`)、25%分位数 (`25%`)、中位数 (`50%`)、75%分位数 (`75%`) 和最大值 (`max`)。这些统计量能帮助我们了解数据的中心趋势、离散程度和分布范围。

`df.describe(include='object')` 则会为非数值列 (通常是字符串或类别) 计算不同的统计量, 如计数 (`count`)、唯一值数量 (`unique`)、出现次数最多的值 (`top`) 以及其出现频率 (`freq`)。这对于理解类别型数据的分布非常有用。

5.2.2 处理缺失值

缺失值是实际数据中常见的问题。在 EDA 阶段, 我们需要识别缺失值的位置和数量, 并决定如何处理。

- **检测缺失值:**

```
print("\n每列的缺失值数量:")
print(df.isnull().sum())
print("\n总缺失值数量:")
print(df.isnull().sum().sum())
# 查看哪些行包含任何缺失值
# print("\n包含缺失值的行:")
# print(df[df.isnull().any(axis=1)])
```

`df.isnull()` 返回一个布尔型的 DataFrame, 其中缺失值为 `True`。`.sum()` 方法在列上应用时会计算每列 `True` 的数量, 即缺失值的数量。再次 `.sum()` 则计算总的缺失值数量。

- **可视化缺失值:** 使用 Seaborn 的 `heatmap` 可以直观地展示缺失值的分布模式。

```
import seaborn as sns
import matplotlib.pyplot as plt
plt.figure(figsize=(10, 6))
sns.heatmap(df.isnull(), cbar=False, cmap='viridis')
plt.title('Missing Values Heatmap')
plt.show()
```

这个热力图的每一行代表一个数据点, 每一列代表一个特征。颜色表示是否存在缺失值 (通常一种颜色表示非缺失, 另一种表示缺失)。通过观察热力图, 我们可以看到哪些列缺失值多, 以及缺失值是否集中在某些行。

- **处理缺失值策略 (EDA 阶段的初步考虑):**

- **删除:** 如果某行或某列缺失值过多, 或者缺失值的存在本身就是一种信息, 可以考虑删除。

```
# 删除包含任何缺失值的行
# df_dropped_rows = df.dropna(axis=0)
# 删除包含任何缺失值的列
# df_dropped_cols = df.dropna(axis=1)
# 删除某列缺失值超过阈值的行
# threshold = len(df) * 0.8 # 例如, 保留至少80%非空值的行
# df_threshold = df.dropna(thresh=threshold, axis=0)
```

注意: 删除操作会减少数据量, 可能丢失信息。在 EDA 阶段, 通常是先识别问题, 具体的处理策略可能留到数据预处理阶段。

- **填充 (Imputation):** 用某个值 (如均值、中位数、众数、固定值) 或通过插值来填充缺失值。

```
# 用均值填充数值列的缺失值
# df['数值列'].fillna(df['数值列'].mean(), inplace=True)
# 用众数填充类别列的缺失值
# df['类别列'].fillna(df['类别列'].mode()[0], inplace=True)
# 用前一个非空值填充
# df.fillna(method='ffill', inplace=True)
# 用后一个非空值填充
# df.fillna(method='bfill', inplace=True)
```

填充策略的选择取决于数据的类型和缺失的原因。EDA 帮助我们理解缺失值的性质, 从而选择合适的填充方法。

5.2.3 处理重复值

重复的行可能是数据录入错误、合并数据时的重复等。

- **检测重复行:**

```
print("\n重复的行数:")
print(df.duplicated().sum())
```

`df.duplicated()` 返回一个布尔型的 Series，表示每一行是否是其之前行的重复。`.sum()` 计算重复行的数量。

- **查看重复行:**

```
# print("\n重复的行:")
# print(df[df.duplicated(keep=False)]) # keep='first'/'last'/'False'
```

`keep=False` 会标记所有重复的行（包括第一次出现的）。

- **删除重复行:**

```
# df_no_duplicates = df.drop_duplicates()
# print(f"\n删除重复行后, 数据剩余 {df_no_duplicates.shape[0]} 行")
```

`df.drop_duplicates()` 默认保留第一次出现的重复行。

5.2.4 理解唯一值和类别分布

对于类别型特征或标识符特征，了解其唯一值的数量和每个类别的分布非常重要。

- **查看唯一值数量:**

```
print("\n每列的唯一值数量:")
print(df.nunique())
```

`df.nunique()` 计算每列的唯一非空值数量。对于高基数（唯一值数量多）的列，可能需要特别处理（如特征哈希、编码等）。

- **查看特定列的唯一值:**

```
# print("\n列 '类别列' 的唯一值:")
# print(df['类别列'].unique())
```

- **查看类别分布 (频率):**

```
# print("\n列 '类别列' 的值计数:")
# print(df['类别列'].value_counts())
# 查看归一化的频率
```

```
# print("\n列 '类别列' 的值频率:")
# print(df['类别列'].value_counts(normalize=True))
```

`df['column'].value_counts()` 返回一个 Series，显示指定列中每个唯一值出现的次数（频率），按频率降序排列。`normalize=True` 参数可以显示频率的百分比。这对于理解类别型特征的分布、是否存在不平衡类别等非常有用。

5.2.5 数据分组与聚合

通过对数据进行分组并计算组的聚合统计量，可以发现不同组别之间的差异。

- 按列分组并计算均值：

```
# print("\n按 '类别列' 分组，计算 '数值列' 的均值:")
# print(df.groupby('类别列')['数值列'].mean())
```

`df.groupby('column')` 创建一个 GroupBy 对象，可以对其应用各种聚合函数（`mean()`，`sum()`，`count()`，`size()`，`min()`，`max()`，`std()`，`var()`，`agg()` 等）。

- 同时计算多个聚合统计量：

```
# print("\n按 '类别列' 分组，计算 '数值列' 的多个统计量:")
# print(df.groupby('类别列')['数值列'].agg(['mean', 'median', 'std']))
```

使用 `agg()` 方法可以方便地计算多个聚合统计量。

5.2.6 相关性分析

对于数值型特征，相关性分析可以帮助我们理解它们之间的线性关系强度和方向。

- 计算相关性矩阵：

```
# print("\n数值列相关性矩阵:")
# print(df.corr(numeric_only=True)) # numeric_only=True 避免警告
```

`df.corr()` 计算 DataFrame 中所有数值列之间的 Pearson 相关系数。结果是一个相关性矩阵，对角线是1（变量与自身的完全相关），其他位置的值介于-1和1之间。接近1表示强正相关，接近-1表示强负相关，接近0表示弱相关或无线性相关。

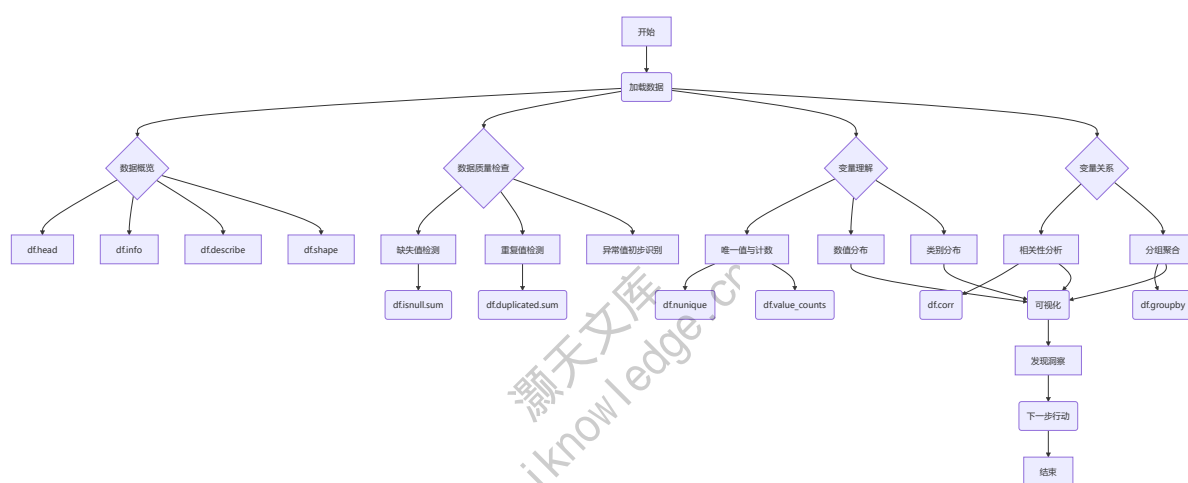
- 可视化相关性矩阵：使用 Seaborn 的 `heatmap` 是展示相关性矩阵的常用方法。

```
# plt.figure(figsize=(10, 8))
# sns.heatmap(df.corr(numeric_only=True), annot=True, cmap='coolwarm',
#             fmt=".2f")
# plt.title('Correlation Matrix')
# plt.show()
```

`annot=True` 在热力图上显示相关系数值, `cmap` 设置颜色方案, `fmt=".2f"` 设置数值格式。相关性热力图能直观地展示哪些变量之间存在较强的线性关系。

5.2.7 EDA 流程示意图

下面是一个简化的 EDA 流程示意图, 展示了使用 Pandas 进行数据探索的一些典型步骤。



这个流程图并非严格顺序, EDA 通常是一个迭代的过程。通过这些步骤, 我们可以对数据有一个全面的了解, 为后续的可视化和分析打下基础。

5.3 数据可视化: 使用 Matplotlib 和 Seaborn

数据可视化是将数据探索的发现以图形方式呈现出来, 帮助我们更直观地理解数据。Matplotlib 是 Python 最基础的绘图库, 提供了高度的灵活性; Seaborn 构建在 Matplotlib 之上, 提供了更美观的默认样式和更多专门用于统计数据可视化的图表类型, 通常更适合数据科学任务。两者经常结合使用。

5.3.1 Matplotlib 基础

Matplotlib 的核心概念是 `Figure` (图) 和 `Axes` (轴/子图)。一个 `Figure` 可以包含一个或多个 `Axes`。

```
**导入 Matplotlib**
import matplotlib.pyplot as plt
**创建一个 Figure 和一个 Axes**
fig, ax = plt.subplots(figsize=(8, 6)) # figsize设置图的大小
```

```
**在 Axes 上绘制数据（例如，一条简单的线）**  
x = [1, 2, 3, 4, 5]  
y = [2, 4, 5, 4, 5]  
ax.plot(x, y)  
**设置标题和轴标签**  
ax.set_title('Simple Line Plot')  
ax.set_xlabel('X-axis Label')  
ax.set_ylabel('Y-axis Label')  
**显示图表**  
plt.show()
```

这是 Matplotlib 的面向对象 API 的常用方式，推荐使用，因为它提供了对图表各个元素的精细控制。plt.pyplot（通常导入为 plt）也提供了快捷函数，可以直接绘制到当前的 Axes 上，对于简单的图表很方便。

5.3.2 Seaborn 基础

Seaborn 简化了许多常见统计图的绘制，并提供了更好的默认视觉效果。

```
**导入 Seaborn**  
import seaborn as sns  
**设置 Seaborn 风格（可选，但推荐）**  
sns.set_theme(style="whitegrid") # 或 "darkgrid", "ticks", etc.  
**使用 Seaborn 绘制图表（例如，一个简单的散点图）**  
**假设我们有一个 DataFrame df_sample 包含 'x' 和 'y' 列**  
**df_sample = pd.DataFrame({'x': [1, 2, 3, 4, 5], 'y': [2, 4, 5, 4, 5]})**  
**sns.scatterplot(data=df_sample, x='x', y='y')**  
**大多数 Seaborn 函数可以直接接受 DataFrame**  
**例如，使用之前加载的 df 绘制某个数值列的直方图**  
**sns.histplot(data=df, x='数值列')**  
**plt.title('Seaborn Scatter Plot Example') # Seaborn 函数通常返回 Axes 对象，也可以用 Matplotlib 设置标题等**  
**plt.show()**
```

Seaborn 函数通常接收 data 参数（一个 DataFrame）和 x, y 等参数（列名），这使得从 DataFrame 绘图非常方便。Seaborn 函数内部会调用 Matplotlib，所以可以结合使用 Matplotlib 的函数（如 plt.title(), plt.xlabel(), plt.show()）来进一步定制图表。

5.3.3 常用图表类型及应用

选择合适的图表类型取决于你想展示的数据类型（数值、类别、时间序列等）以及你想传达的信息（分布、关系、比较、构成等）。

- **单变量分布图 (Univariate Distribution Plots):** 用于展示单个变量的分布情况。

- **直方图 (Histogram):** 显示数值变量的频率分布, 将数据范围分成若干个 bin, 统计每个 bin 中的数据点数量。

```
# plt.figure(figsize=(8, 5))
# sns.histplot(data=df, x='数值列', bins=30, kde=True) # bins设置分组数量,
# kde=True叠加核密度估计
# plt.title('Distribution of 数值列')
# plt.xlabel('数值列')
# plt.ylabel('Frequency')
# plt.show()
```

直方图能帮助我们看出数据的中心位置、分散程度、偏度 (skewness) 和峰度 (kurtosis), 以及是否存在多个峰 (多模态分布)。

- **箱线图 (Box Plot):** 显示数值变量的五数概括 (最小值、第一四分位数 Q1、中位数 Q2、第三四分位数 Q3、最大值) 以及异常值。

```
# plt.figure(figsize=(6, 8))
# sns.boxplot(data=df, y='数值列') # y='数值列' 绘制垂直箱线图
# plt.title('Box Plot of 数值列')
# plt.ylabel('数值列')
# plt.show()
```

箱线图特别适合比较多个组的分布, 或者快速识别单个变量的异常值。

- **核密度估计图 (Kernel Density Estimate Plot - KDE Plot):** 对直方图进行平滑处理, 显示数值变量的概率密度分布。

```
# plt.figure(figsize=(8, 5))
# sns.kdeplot(data=df, x='数值列', fill=True) # fill=True 填充曲线下区域
# plt.title('Density Plot of 数值列')
# plt.xlabel('数值列')
# plt.ylabel('Density')
# plt.show()
```

KDE 图比直方图更能清晰地展示分布的形状, 尤其是在数据量较少或需要比较不同组的分布时。

- **计数图 (Count Plot):** 用于显示类别变量中每个类别的观测数量。

```
# plt.figure(figsize=(10, 6))
# sns.countplot(data=df, x='类别列') # x='类别列' 绘制垂直柱状图
# plt.title('Count of each Category')
# plt.xlabel('类别列')
# plt.ylabel('Count')
# plt.xticks(rotation=45, ha='right') # 如果类别名称较长, 旋转x轴标签
# plt.tight_layout() # 自动调整布局, 防止标签重叠
# plt.show()
```

计数图是 `value_counts()` 的可视化版本, 直观地展示了类别变量的分布是否均衡。

- **双变量关系图 (Bivariate Relationship Plots):** 用于展示两个变量之间的关系。
- **散点图 (Scatter Plot):** 显示两个数值变量之间的关系。每个点代表一个观测, 其位置由两个变量的值决定。

```
# plt.figure(figsize=(8, 6))
# sns.scatterplot(data=df, x='数值列1', y='数值列2', hue='类别列', size='数值列3') # hue按类别着色, size按数值大小调整点的大小
# plt.title('Scatter Plot of 数值列1 vs 数值列2')
# plt.xlabel('数值列1')
# plt.ylabel('数值列2')
# plt.show()
```

散点图可以显示变量之间的线性或非线性关系、数据点的聚集情况以及潜在的异常值。通过 `hue` 或 `size` 参数, 可以引入第三个变量的信息。

- **线图 (Line Plot):** 显示数据点随某个有序变量 (通常是时间) 的变化趋势。

```
# # 假设数据按时间排序, 并且有一列是时间/日期
# # df['时间列'] = pd.to_datetime(df['时间列']) # 确保时间列是datetime类型
# # df = df.sort_values('时间列')
# plt.figure(figsize=(12, 6))
# sns.lineplot(data=df, x='时间列', y='数值列')
# plt.title('Trend of 数值列 Over Time')
# plt.xlabel('时间')
# plt.ylabel('数值列')
# plt.show()
```


线图非常适合展示时间序列数据的趋势、周期性和季节性。

- **条形图 (Bar Plot):** 用于比较不同类别或组的数值变量的平均值、总和或其他聚合统计量。

```
# plt.figure(figsize=(10, 6))
# # 默认计算均值及置信区间
# sns.barplot(data=df, x='类别列', y='数值列', estimator='mean') #
estimator='sum' 计算总和
# plt.title('Mean of 数值列 by 类别列')
# plt.xlabel('类别列')
# plt.ylabel('Mean of 数值列')
# plt.xticks(rotation=45, ha='right')
# plt.tight_layout()
# plt.show()
```

条形图适用于比较离散组之间的数值度量。与计数图不同，条形图的纵轴是某个数值变量的统计量，而不是观测数量。

- **多变量关系图 (Multivariate Relationship Plots):** 用于同时展示多个变量之间的关系。
- **热力图 (Heatmap):** 使用颜色强度来表示二维矩阵中的值。常用于可视化相关性矩阵或展示两个类别变量的交叉频率。

```
# # 绘制相关性矩阵热力图（前面已展示）
# plt.figure(figsize=(10, 8))
# sns.heatmap(df.corr(numeric_only=True), annot=True, cmap='coolwarm',
fmt=".2f")
# plt.title('Correlation Matrix')
# plt.show()
# # 绘制两个类别变量的交叉表热力图
# # cross_tab = pd.crosstab(df['类别列1'], df['类别列2'])
# # plt.figure(figsize=(8, 6))
# # sns.heatmap(cross_tab, annot=True, fmt='d', cmap='Blues') # fmt='d'
# # 格式化为整数
# # plt.title('Crosstab Heatmap of 类别列1 vs 类别列2')
# # plt.xlabel('类别列2')
# # plt.ylabel('类别列1')
# # plt.show()
```

热力图是理解变量之间相互作用模式的强大工具。

- **Pair Plot (散点图矩阵):** 绘制 DataFrame 中所有数值变量两两之间的散点图。对角线默认绘制单变量分布图 (如直方图或 KDE 图)。

```
# # 选择一部分数值列进行 pair plot, 因为如果列太多会很拥挤
# # numerical_cols = df.select_dtypes(include=['number']).columns[:5] #
# # 选择前5个数值列
# # sns.pairplot(df[numerical_cols], diag_kind='kde', hue='类别列') #
# # diag_kind控制对角线图类型, hue按类别着色
# # plt.suptitle('Pair Plot of Numerical Features', y=1.02) # 设置整个图
# # 的标题
# # plt.show()
```

Pair Plot 提供了一个快速概览数据集中数值变量之间所有双变量关系的视图, 非常适合在 EDA 初期使用。

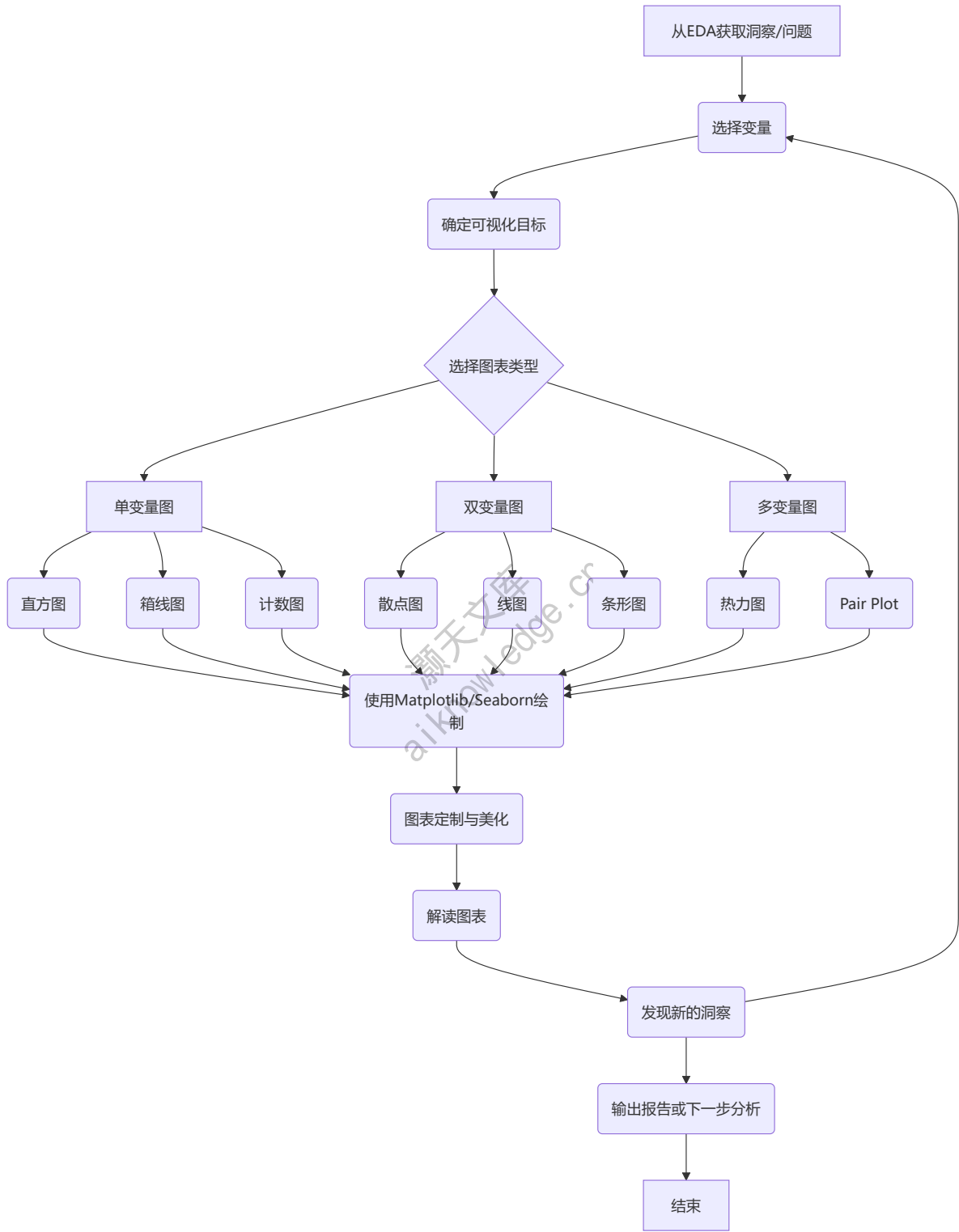
5.3.4 图表定制与美化

Matplotlib 和 Seaborn 提供了丰富的选项来定制图表的外观, 使其更具信息量和吸引力。

- **设置图表尺寸:** `plt.figure(figsize=(width, height))` 或在 Seaborn 函数中通过 `height` 和 `aspect` 参数调整。
- **设置标题和轴标签:** `ax.set_title()`, `ax.set_xlabel()`, `ax.set_ylabel()` 或 `plt.title()`, `plt.xlabel()`, `plt.ylabel()`。
- **添加图例:** `ax.legend()` 或 Seaborn 函数通常会自动生成图例。
- **调整轴刻度:** `ax.set_xticks()`, `ax.set_yticks()`, `ax.set_xticklabels()`, `ax.set_yticklabels()`。
- **旋转标签:** `plt.xticks(rotation=...)` 或 `plt.yticks(rotation=...)`。
- **调整布局:** `plt.tight_layout()` 自动调整子图参数, 使之填充整个 figure 区域并防止标签重叠。
- **添加文本注释:** `ax.text(x, y, 'text', ...)` 或 `plt.annotate('text', xy=(x, y), xytext=(x_text, y_text), arrowprops=...)`。
- **修改颜色、线条样式、标记:** 大多数绘图函数都有参数可以控制这些属性 (`color`, `linestyle`, `marker`, `cmap`, `palette` 等)。
- **使用风格:** `plt.style.use('ggplot')` 或 `sns.set_theme()` 可以快速改变图表的整体外观。

5.3.5 可视化流程示意图

下面是一个简化的数据可视化流程示意图。



这个流程同样是迭代的。可视化可以帮助我们验证 EDA 中的发现，也可以揭示新的模式，从而指导进一步的数据探索或后续分析。

5.4 连接探索与可视化

数据探索 (EDA) 和可视化是紧密相连、相互促进的过程。

1. **EDA 指导可视化：**通过 Pandas 的统计摘要，我们可以发现哪些列是数值型、哪些是类别型，是否存在缺失值、异常值，变量的分布大概是什么样子，变量之间是否存在初步的相关性。这些发现直接指导我们选择合适的图表类型。例如，发现某个数值列偏度很大，就应该绘制直方图或 KDE 图来查看其分布形状；发现两个数值列相关系数较高，就应该绘制散点图来观察具体的关系模式。
2. **可视化揭示新的探索方向：**可视化图表往往能揭示统计摘要难以捕捉的模式。例如，散点图可能显示出非线性的关系或数据点的聚类，这是相关系数无法直接体现的；箱线图能清晰地展示异常值，促使我们进一步探索这些异常值的原因；直方图可能显示出多模态分布，提示数据可能来自不同的子群体，需要进一步分组探索。
3. **迭代过程：**通常的流程是：初步 EDA -> 基于发现进行可视化 -> 从可视化中获得新的洞察 -> 回到 EDA 进行更深入的统计分析或数据筛选 -> 再次可视化验证或展示发现。这个循环会持续进行，直到对数据有了足够的理解。

5.5 总结

数据探索与可视化是数据科学流程中不可或缺的早期步骤。通过本章的学习，我们了解了如何利用 Pandas 库进行基本的数据概览、缺失值和重复值处理、唯一值和类别分布分析、分组聚合以及相关性分析。同时，我们也学习了如何使用 Matplotlib 和 Seaborn 库绘制各种常用的统计图表，包括单变量分布图（直方图、箱线图、KDE 图、计数图）、双变量关系图（散点图、线图、条形图）和多变量关系图（热力图、Pair Plot）。

掌握这些工具和技巧，能够帮助我们：

- 快速理解数据集的特征和结构。
- 识别并处理数据质量问题。
- 发现变量之间的潜在关系和模式。
- 为后续的特征工程、模型选择和结果解释提供重要的指导和依据。

有效的 EDA 和可视化不仅能提升数据分析的效率，更能帮助我们挖掘出有价值的洞察，为解决实际问题提供支持。在实际项目中，请务必投入足够的时间和精力进行充分的数据探索和可视化。

6. 高级数据处理技巧与最佳实践

6. 高级数据处理技巧与最佳实践

在掌握了 Python 数据科学基础库（如 Pandas、NumPy）的基本数据加载、清洗和转换技能后，面对更复杂、更大规模或特定类型的数据集时，我们需要运用更高级的处理技巧和遵循一系列最

佳实践，以提高效率、确保数据质量并构建可维护的工作流程。本章将深入探讨这些高级方面。

6.1 处理大规模数据：分块读取与内存优化

当数据集的大小超出计算机内存时，直接加载整个文件会导致内存错误。这时，分块（Chunking）读取成为必要手段。

分块读取

Pandas的 `read_csv` 等函数提供了 `chunksize` 参数，允许我们将大文件分割成小的块进行迭代处理，而不是一次性加载。

```
import pandas as pd
**假设有一个很大的CSV文件 'large_data.csv'**
**设置 chunksize 参数，例如每次读取 10000 行**
chunk_size = 10000
chunks = pd.read_csv('large_data.csv', chunksize=chunk_size)
**遍历每个数据块并进行处理（例如，计算总和）**
total_sum = 0
for chunk in chunks:
    # 对当前数据块进行处理
    # 例如，计算某一列 'value' 的总和
    total_sum += chunk['value'].sum()
    # 可以在此处进行其他基于块的处理
print(f"Total sum calculated from chunks: {total_sum}")
```

通过迭代处理数据块，我们可以在不将整个数据集载入内存的情况下完成计算或转换。

内存优化

即使数据能载入内存，优化数据类型也能显著减少内存占用。

- **选择合适的数据类型：** 对于整数列，如果范围不大，可以使用 `int8` , `int16` , `int32` 代替默认的 `int64` 。对于浮点数列，`float32` 通常足够精度且小于 `float64` 。对于包含有限个唯一值的字符串列，可以转换为 `category` 类型。
- **使用 `df.info(memory_usage='deep')` 检查内存使用。**
- **利用 `df.astype()` 进行类型转换。**

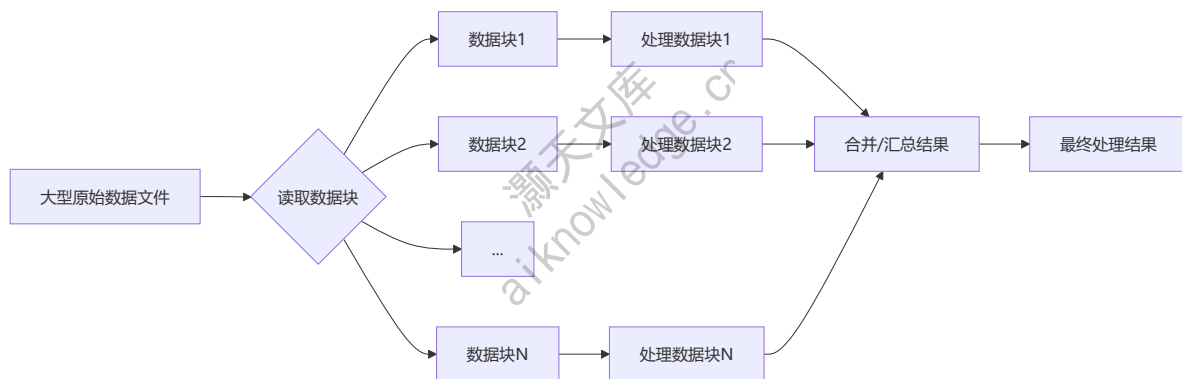
```
**示例：优化数据类型**
df = pd.read_csv('data.csv')
**检查初始内存使用**
print(df.info(memory_usage='deep'))
**尝试优化**
for col in df.columns:
    dtype = df[col].dtype
```

```

if dtype == 'object': # 字符串列
    if df[col].nunique() / len(df[col]) < 0.5: # 如果唯一值比例较低
        df[col] = df[col].astype('category')
elif dtype == 'int64':
    # 尝试转换为更小的整数类型
    min_val = df[col].min()
    max_val = df[col].max()
    if min_val >= -128 and max_val <= 127:
        df[col] = df[col].astype('int8')
    elif min_val >= -32768 and max_val <= 32767:
        df[col] = df[col].astype('int16')
    # ... 其他整数类型
elif dtype == 'float64':
    # 可以考虑转换为 float32
    df[col] = df[col].astype('float32')
**再次检查优化后的内存使用**
print(df.info(memory_usage='deep'))

```

分块处理流程图:



6.2 高级特征工程

特征工程是将原始数据转换为更能代表潜在问题、提高模型性能的特征的过程。高级特征工程涉及更复杂的转换和组合。

多项式特征与交互项

- **多项式特征**: 基于现有特征创建其多项式组合 (如 x^2, x^3) 。
- **交互项**: 组合两个或多个特征, 捕捉它们之间的相互作用 (如 $x \times y$) 。

```

from sklearn.preprocessing import PolynomialFeatures
import numpy as np
**示例数据**
data = {'feature1': [1, 2, 3, 4], 'feature2': [5, 6, 7, 8]}
df = pd.DataFrame(data)
**创建多项式特征和交互项 (degree=2)**

```

```
**include_bias=False 避免生成全为1的偏置项**
poly = PolynomialFeatures(degree=2, include_bias=False)
poly_features = poly.fit_transform(df)
**poly_features 现在是一个NumPy数组**
**对应的特征是 [feature1, feature2, feature1^2, feature1*feature2, feature2^2]**
print(poly_features)
**可以将其转换回DataFrame, 并查看生成的特征名称**
poly_feature_names = poly.get_feature_names_out(df.columns)
df_poly = pd.DataFrame(poly_features, columns=poly_feature_names)
print(df_poly)
```

基于日期/时间的特征

从日期时间戳中提取有意义的信息是时间序列数据中的常见高级特征工程。

```
**示例日期时间数据**
df['timestamp'] = pd.to_datetime(['2023-01-15 10:30:00', '2023-01-16 14:00:00',
'2023-02-20 09:15:00'])
**提取日期时间组件作为新特征**
df['year'] = df['timestamp'].dt.year
df['month'] = df['timestamp'].dt.month
df['day'] = df['timestamp'].dt.day
df['day_of_week'] = df['timestamp'].dt.dayofweek # Monday=0, Sunday=6
df['hour'] = df['timestamp'].dt.hour
df['minute'] = df['timestamp'].dt.minute
df['is_weekend'] = (df['timestamp'].dt.dayofweek >= 5).astype(int) # 判断是否周末
print(df)
```

6.3 时间序列数据处理

时间序列数据具有时间依赖性，需要特定的处理技术。

重采样 (Resampling)

重采样是将时间序列从一个频率转换到另一个频率的过程。例如，将每分钟的数据聚合到每小时或每天。

```
**创建一个示例时间序列 DataFrame**
rng = pd.date_range('2023-01-01', periods=100, freq='T') # 每分钟一个数据点
ts_df = pd.DataFrame({'value': np.random.rand(100)}, index=rng)
print("原始数据 (每分钟):")
print(ts_df.head())
**重采样到每小时, 计算平均值**
hourly_ts = ts_df.resample('H').mean()
print("\n重采样到每小时 (平均值):")
print(hourly_ts.head())
```

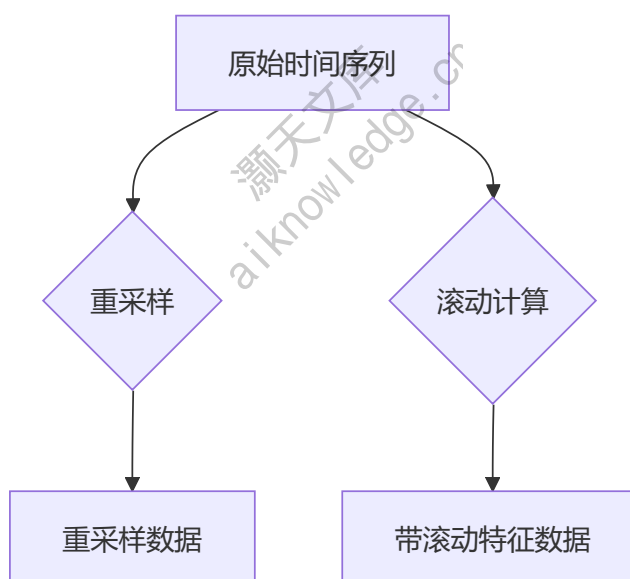
```
**重采样到每天，计算总和**  
daily_ts = ts_df.resample('D').sum()  
print("\n重采样到每天（总和）:")  
print(daily_ts.head())
```

滚动窗口 (Rolling Windows)

滚动窗口计算在数据点的滑动窗口上的聚合统计量，如移动平均、移动总和等，常用于平滑数据或捕捉局部趋势。

```
**在原始时间序列数据上计算5分钟的滚动平均**  
ts_df['rolling_mean_5min'] = ts_df['value'].rolling(window='5T').mean()  
**在原始时间序列数据上计算10分钟的滚动总和**  
ts_df['rolling_sum_10min'] = ts_df['value'].rolling(window='10T').sum()  
print("\n带有滚动统计量的数据:")  
print(ts_df.head(10))
```

时间序列处理流程简图：



6.4 数据验证与质量检查

在数据处理流程中加入数据验证步骤至关重要，可以确保数据的质量符合预期，避免后续分析或模型训练出现隐蔽的错误。

断言 (Assertions)

使用 `assert` 语句检查关键的数据属性是否满足条件。如果条件不满足，`assert` 会引发 `AssertionError`，中断程序执行，从而快速定位问题。


```
**示例 DataFrame**
data = {'ID': [1, 2, 3, 4, 5],
        'Value': [10, 25, -5, 30, 15],
        'Category': ['A', 'B', 'A', 'C', 'B']}
df = pd.DataFrame(data)
**验证 Value 列是否都大于 0**
try:
    assert (df['Value'] >= 0).all(), "Value 列包含负值!"
    print("Value 列验证通过: 所有值非负。")
except AssertionError as e:
    print(f"验证失败: {e}")
**验证 Category 列是否只包含预期值**
allowed_categories = ['A', 'B', 'C', 'D']
try:
    assert df['Category'].isin(allowed_categories).all(), "Category 列包含未预期的类别!"
    print("Category 列验证通过: 所有类别都在允许范围内。")
except AssertionError as e:
    print(f"验证失败: {e}")
**验证 DataFrame 行数**
expected_rows = 5
try:
    assert len(df) == expected_rows, f"DataFrame 行数不正确, 预期 {expected_rows} 行, 实际 {len(df)} 行!"
    print("DataFrame 行数验证通过。")
except AssertionError as e:
    print(f"验证失败: {e}")
```

除了 `assert`，还可以结合 Pandas 方法进行更复杂的检查：

- `df.isnull().sum()` 检查缺失值数量。
- `df.duplicated().sum()` 检查重复行数量。
- `df['column'].dtype` 检查数据类型。
- `df['column'].between(lower, upper).all()` 检查数值范围。
- `df['column'].value_counts()` 查看类别分布，检查是否存在异常值。

将这些检查集成到数据处理流程的关键步骤中，可以建立强大的数据质量门控。

6.5 提高处理效率：向量化操作

Python数据科学的核心库 NumPy 和 Pandas 的设计理念是利用底层优化的 C 或 Fortran 代码进行向量化操作。应尽量避免使用 Python 原生的循环（`for` 或 `while`）来处理 DataFrame 或 Series 的元素，因为这通常效率低下。

避免显式循环

```
**低效的循环方式**
**df['Value_squared_loop'] = 0**
**for index, row in df.iterrows():**
**df.loc[index, 'Value_squared_loop'] = row['Value'] ** 2**
**高效的向量化方式**
df['Value_squared_vectorized'] = df['Value'] ** 2
print(df)
```

向量化操作直接在整个 Series 或 DataFrame 上执行，速度远快于逐行或逐元素的 Python 循环。

使用内置函数和方法

Pandas 和 NumPy 提供了大量优化的内置函数和方法，如 `.sum()`，`.mean()`，`.max()`，`.min()`，`.apply()`（在适当的场景下，特别是 `axis=0` 或用于元素级函数时），`.transform()`，`.agg()`，`.clip()`，`.fillna()`，`.dropna()` 等。优先使用这些内置方法而不是自己编写循环。

```
**示例：使用内置方法进行条件替换（比循环快）**
df['Value_clipped'] = df['Value'].clip(lower=0) # 将小于0的值替换为0
**示例：使用 apply（axis=0 或用于 map/applymap 类型的元素操作）**
**注意：对于简单的元素操作，向量化更好；apply(axis=1) 通常效率较低**
**假设有一个函数 func，作用于 Value 列的每个元素**
**df['Value_processed'] = df['Value'].apply(func)**
```

6.6 构建可复用的数据处理管道

将一系列数据处理步骤组织成一个有序的管道（Pipeline）是高级数据处理的重要实践。管道提高了代码的可读性、可维护性和可复用性，并有助于防止数据泄露（特别是在机器学习任务中）。

Pandas 方法链与 `.pipe()`

Pandas 允许通过方法链（method chaining）的方式将多个操作连接起来，使代码更紧凑和易读。`.pipe()` 方法允许在链中插入自定义函数。

```
**示例：使用方法链和 .pipe()**
def standardize_column(df_in, col):
    df = df_in.copy()
    df[col] = (df[col] - df[col].mean()) / df[col].std()
    return df
def rename_columns_lower(df_in):
    df = df_in.copy()
    df.columns = df.columns.str.lower()
```

```
    return df
**原始数据**
data = {'FeatureA': [10, 20, 30], 'FeatureB': [100, 200, 300]}
df = pd.DataFrame(data)
**构建处理管道**
processed_df = (df.copy() # 从原始数据副本开始
                .pipe(rename_columns_lower) # 应用自定义函数1
                .pipe(standardize_column, col='featurea') # 应用自定义函数2, 带参数
                .rename(columns={'featureb': 'feature_b_renamed'})) # 应用Pandas内置方法

print("原始 DataFrame:")
print(df)
print("\n处理后的 DataFrame:")
print(processed_df)
```

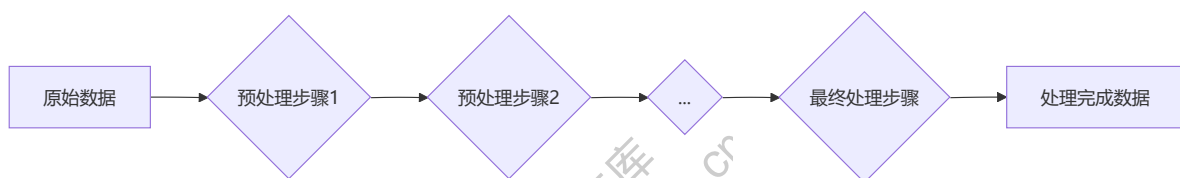
Scikit-learn 的 Pipeline

在机器学习工作流中，通常使用 Scikit-learn 的 Pipeline 来封装预处理步骤（如填充缺失值、特征缩放、特征编码等）和最终的模型。这确保了在训练和预测时应用完全相同的预处理流程，避免数据泄露问题。

```
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.linear_model import LogisticRegression
import numpy as np
**示例数据**
data = {'Numerical1': [1, 2, np.nan, 4],
        'Numerical2': [10, 20, 30, 40],
        'Categorical': ['A', 'B', 'A', 'C'],
        'Target': [0, 1, 0, 1]}
df = pd.DataFrame(data)
**分离特征和目标变量**
X = df[['Numerical1', 'Numerical2', 'Categorical']]
y = df['Target']
**创建预处理步骤**
**数值特征：填充缺失值 -> 标准化**
numerical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='mean')),
    ('scaler', StandardScaler())
])
**类别特征：独热编码**
categorical_transformer = Pipeline(steps=[
    ('onehot', OneHotEncoder(handle_unknown='ignore'))
```

```
])
**使用 ColumnTransformer 对不同列应用不同的预处理**
preprocessor = ColumnTransformer(
    transformers=[
        ('num', numerical_transformer, ['Numerical1', 'Numerical2']),
        ('cat', categorical_transformer, ['Categorical'])
    ])
**创建完整的机器学习管道: 预处理 -> 模型**
model_pipeline = Pipeline(steps=[('preprocessor', preprocessor),
                                   ('classifier', LogisticRegression())])
**现在可以直接在原始数据上训练管道**
model_pipeline.fit(X, y)
**预测新数据时, 管道会自动应用相同的预处理**
**new_data = pd.DataFrame(...)**
**predictions = model_pipeline.predict(new_data)**
```

数据处理管道流程图:



6.7 最佳实践总结

除了上述技术，以下是数据处理过程中的一些通用最佳实践：

1. **理解业务背景和数据来源：** 深入了解数据的含义、收集方式和潜在偏差是进行有效处理的基础。
2. **保持原始数据不变：** 在处理过程中始终操作数据的副本，保留原始数据以便回溯和验证。
3. **增量式开发与验证：** 每完成一个处理步骤，就检查结果是否符合预期，而不是一次性写完所有代码再调试。
4. **文档化：** 详细记录每个处理步骤的目的、实现方式以及关键决策（如如何处理缺失值、异常值）。代码中的注释和外部文档都很重要。
5. **版本控制：** 使用 Git 等工具管理数据处理代码，方便追踪修改、协作和回滚。
6. **模块化：** 将复杂的数据处理逻辑分解为小的、可测试的函数或类，提高代码的可读性和复用性。
7. **考虑可扩展性：** 在设计处理流程时，考虑未来数据量增加或需求变化的可能性。

结论

高级数据处理技巧是数据科学家工具箱中不可或缺的部分。掌握分块处理大规模数据、运用高级特征工程技术、处理时间序列数据、实施严格的数据验证、优化代码性能以及构建可复用的处理

管道，将使你能够更有效地应对真实世界中复杂多变的数据挑战，为后续的数据分析、可视化和模型构建奠定坚实的基础。将这些高级技术与良好的编程和协作实践相结合，才能构建健壮、高效且可靠的数据科学工作流。

灏天文库
aiknowledge.cn



您手中的这份学习资料，源自[灏天文库](https://aiknowledge.cn)。我们深耕人工智能与泛技术领域，致力于为您提供深度解析的技术内容，助您不仅知其然，更知其所以然。在这里，前沿理论与实战经验并重，复杂概念被化繁为简，助您直抵技术核心，把握时代机遇。我们不仅是信息的传递者，更是技术的深度解析者和价值的挖掘者。以人工智能为核心，灏天文库辐射计算机视觉、深度学习、自然语言处理等多个前沿领域，构建起全面而精深的泛技术知识体系。我们紧密追踪全球前沿动态，精选顶会论文进行深度解读，洞察行业趋势，并持续追踪大模型、AIGC等技术热点，助您抢占先机。

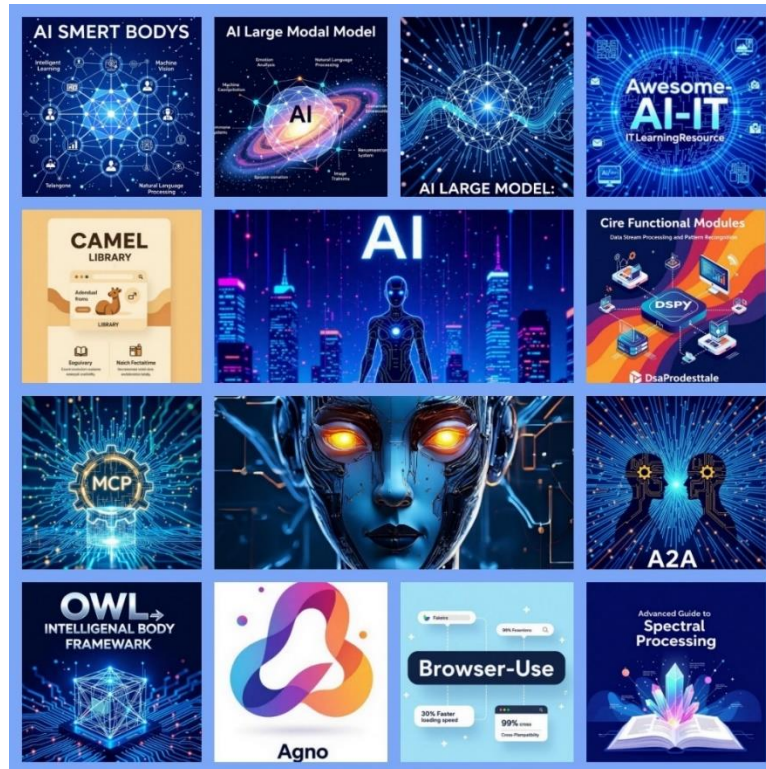
此外，我们致力于构建一个开放、互动的技术社区，鼓励知识分享、经验交流，让技术在碰撞中迸发火花。灏天文库，不仅仅是一个内容平台，更是一个技术人的精神家园，是我们共同深耕技术沃土，共建繁荣技术生态的知识引擎——秉持“技术驱动·深度解析·生态共建”的理念，助您在技术道路上不断精进，成就卓越。

欢迎访问灏天文库网站：<http://www.aiknowledge.cn>



微信公众号：灏天文库

在这里，您能探索瞬息万变的科技世界。我们聚焦前沿技术与创新应用，从 AI 大型模型到智能体技术，再到各种优秀 IT 资源，为您呈现最新进展。同时，我们也注重编程语言与开发工具教学，涵盖 PHP、HTML5、Linux 等众多方向。无论您是新手还是资深开发者，都能在这里找到有价值的学习资源与实践经验。



学习不止，成长不息！

感谢您阅读这份资料，希望它能为您带来启发。


如果您渴望在技术领域持续精进，深入理解前沿知识，拓展技能边界，那么请务必访问灏天文库平台：

 **灏天文库 (<http://www.aiknowledge.cn>)**

在这里，我们以人工智能为核心，深耕计算机视觉、深度学习、自然语言处理、语音识别、时间序列分析、无线通信等多个高精尖技术方向。

灏天文库，致力于为您打造一个：

- **专业深入的技术解析平台：** 助您洞悉技术原理，挖掘底层算法。
 - **前沿动态的追踪者：** 紧跟技术脉搏，把握行业趋势。
 - **互动共建的知识社区：** 与志同道合者交流，共同成长。

 **福利放送！** 我们深知优质资料对成长的价值。为了助力您的技术旅程，灏天文库定期精选并分享各类优质学习资源、技术干货，免费提供学习路线图！

访问 aiknowledge.cn，即刻加入灏天文库，获取更多免费学习资料，让我们共同见证您的技术飞跃！

灏天文库：技术驱动 · 深度解析 · 生态共建

您的终身学习资源库





• 您的免费技术宝库

限时开放！海量前沿技术资料免费下载，助您在AI、数据科学、开发等领域
极速成长！

海量资料

覆盖人工智能、计算机视觉、深度学习、NLP、编程语言、Web/移动开发、DevOps、数据库等。

专业深度

从概念入门到核心算法精讲，从原理到实战，满足不同阶段学习需求。

极速成长

精选优质内容，助您系统构建知识体系，应对技术挑战，提升竞争力。

为什么选择灏天文库？



体系化学习路径

告别碎片化知识，我们为你精心构建从入门到进阶的系统学习地图，助你步步为营，扎实提升。



高质量精选资源

万里挑一，只为你呈现真正有深度、有价值的顶尖教程、书籍与开源项目，避免踩坑。



可视化知识图谱

通过思维导图、饼图、流程图等，清晰展现知识结构与资源关联，全局掌握学习进度。



持续更新与拓展

紧跟技术前沿，内容不断迭代与丰富，覆盖AI、大模型、Web3等热门领域，永不落伍。



优化阅读体验

在灏天文库平台，享受更流畅、无干扰的文档阅读体验，让学习成为一种享受。



活跃技术社区

与志同道合者交流、讨论、共同成长，学习不再孤单，问题迎刃而解。