\*\*Python 数据科学\*\*

\*\*Pandas 基础\*\*

在 www.DataCamp.com 交互式学习 Python 数据科学

\*\*Pandas\*\*

Pandas 库构建在 NumPy 之上，为 Python 编程语言提供了易于使用的数据结构和数据分析工具。

\*\*使用以下导入约定：\*\*

```python

>> import pandas as pd

```

\*\*Pandas 数据结构\*\*

\*\*Series (系列)\*\*

\* 一维带标签数组

\* 能够容纳任何数据类型

```python

>> s = pd.Series([3, -5, 7, 4], index=['a', 'b', 'c', 'd']) # 修正了 index 语法

```

\*\*DataFrame (数据框)\*\*

\* 二维带标签数据结构

\* 列可以是不同的数据类型

| 索引(Index) | 国家(Country) | 首都(Capital) | 人口(Population) |

| :--------- | :----------- | :----------- | :-------------- |

| 0 | 比利时(Belgium) | 布鲁塞尔(Brussels) | 11190846 | # 修正了国家名和首都名拼写

| 1 | 印度(India) | 新德里(New Delhi) | 1303171035 |

| 2 | 巴西(Brazil) | 巴西利亚(Brasília) | 207847528 | # 修正了人口数格式和大小

```python

>> data = {'Country': ['Belgium', 'India', 'Brazil'], # 修正了国家名拼写

'Capital': ['Brussels', 'New Delhi', 'Brasilia'], # 修正了首都名拼写

'Population': [11190846, 1303171035, 207847528]} # 修正了人口数格式和大小

>> df = pd.DataFrame(data, columns=['Country', 'Capital', 'Population']) # 修正了 columns 语法

```

\*\*获取帮助\*\*

```python

>> help(pd.Series.loc)

```

\*\*选择 (Selection)\*\*

| 操作 | 描述 | 示例代码 | 示例输出 (简略表示) |

| :-------------------- | :------------------- | :------------------------------ | :-------------------------------------- |

| \*\*获取 (Getting)\*\* | | | |

| 获取一个元素 | | `>> s['b']` | `-5` |

| 获取 DataFrame 子集 | | `>> df[1:]` | `[显示索引1和2的行]` |

| \*\*按位置选择\*\* | | | |

| 按行/列位置 (loc) | (注意：loc 主要用于标签) | `>> df.iloc[0, 0]` | `'Belgium'` | # 修正了方法名 (loc -> iloc) 和国家名

| \*\*按标签选择\*\* | | | |

| 按行/列标签 (loc) | | `>> df.loc[0, 'Country']` | `'Belgium'` | # 修正了国家名

| \*\*按位置选择 (iloc)\*\* | | | |

| 按行位置 (整行) | | `>> df.iloc[2]` | `Country: Brazil, Capital: Brasília...` |

| 按行/列位置 | | `>> df.iloc[1, 1]` | `'New Delhi'` | # 修正了方法名 (is -> iloc)

| \*\*布尔索引\*\* | | | |

| | | `>> s[s > 1]` | `[显示值大于1的元素]` |

| | | `>> df[df['Population'] > 1200000000]` | `[显示人口大于12亿的行]` | # 修正了人口数阈值

\*\*设置 (Setting)\*\*

```python

>> s['a'] = 6 # 将索引 'a' 的值设置为 6

```

\*\*删除 (Dropping)\*\*

```python

>> s.drop(['a', 'c']) # 从行中删除值 (axis=0 是默认值)

>> df.drop('Country', axis=1) # 从列中删除值 (axis=1)

```

\*\*排序与排名 (Sort & Rank)\*\*

```python

>> df.sort\_index() # 按索引排序

>> df.sort\_values(by='Country') # 按 'Country' 列的值排序

>> df.rank() # 计算排名

```

\*\*获取 Series/DataFrame 信息 (Retrieving Information)\*\*

| 操作 / 属性 | 描述 | 示例代码 |

| :---------------- | :----------------------- | :------------- |

| \*\*基本信息\*\* | | |

| 形状 | (行数, 列数) | `>> df.shape` |

| 索引 | 描述索引 | `>> df.index` |

| 列名 | 描述 DataFrame 的列 | `>> df.columns`|

| 信息 | DataFrame 的详细信息 | `>> df.info()` |

| 非 NA 值计数 | 每列非空值的数量 | `>> df.count()`|

| \*\*摘要统计\*\* | | |

| 求和 | 每列值的总和 | `>> df.sum()` |

| 累计和 | 每列值的累计和 | `>> df.cumsum()`|

| 最小值 / 最大值 | 每列的最小值 / 最大值 | `>> df.min()` / `>> df.max()` |

| 最小值索引 / 最大值索引 | 最小值的索引 / 最大值的索引 | `>> df.idxmin()` / `>> df.idxmax()` |

| 描述性统计 | 数值列的摘要统计 | `>> df.describe()` |

| 众数 | 每列出现频率最高的值 | `>> df.mode()` | # 修正了说明 (maps -> mode)

| 中位数 | 每列的中位数 | `>> df.median()` | # 修正了方法名 (numdam -> median)

| 均值 | 每列的算术平均值 | `>> df.mean()` | # 修正了方法名 (numdam -> mean)

\*\*应用函数 (Applying Functions)\*\*

```python

>> f = lambda x: x \* 2 # 定义一个函数 (例如：乘以2)

>> df.apply(f) # 将函数 f 应用到 DataFrame 的每一列

>> df.applymap(f) # 将函数 f 逐元素应用到 DataFrame

```

\*\*数据对齐 (Data Alignment)\*\*

\* \*\*内部数据对齐 (Internal Data Alignment):\*\*

在索引不重叠的位置会引入 `NA` (缺失) 值：

```python

>> a1 = pd.Series([7, -2, 3], index=['a', 'c', 'd'])

>> a2 = pd.Series([3, -5, 7, 4], index=['a', 'b', 'c', 'd'])

>> a1 + a2

a 10.0

b NaN # 索引 'b' 只在 a2 中存在

c 5.0

d 7.0

dtype: float64

```

\* \*\*使用填充方法的算术运算 (Arithmetic Operations with Fill Methods):\*\*

你可以使用填充方法自行处理内部数据对齐：

```python

>> a1.add(a2, fill\_value=0) # 加法，缺失值视为 0

a 10.0

b -5.0 # (0 + (-5))

c 5.0

d 7.0

dtype: float64

>> a1.sub(a2, fill\_value=2) # 减法

>> a1.div(a2, fill\_value=4) # 除法

>> a1.mul(a2, fill\_value=3) # 乘法

```

\*\*读写 CSV 文件 (Read and Write to CSV)\*\*

```python

>> pd.read\_csv('file.csv', header=None, nrows=5) # 读取CSV，无标题行，只读前5行

>> df.to\_csv('myDataFrame.csv') # 将DataFrame写入CSV文件

```

\*\*读写 Excel 文件 (Read and Write to Excel)\*\*

```python

>> pd.read\_excel('file.xlsx') # 读取Excel文件

>> df.to\_excel('dir/myDataFrame.xlsx', sheet\_name='Sheet1') # 将DataFrame写入Excel文件

```

\*\*从同一文件读取多个工作表 (Read multiple sheets from the same file)\*\*

```python

>> xlsx = pd.ExcelFile('file.xlsx') # 创建ExcelFile对象

>> df\_sheet1 = pd.read\_excel(xlsx, 'Sheet1') # 读取 'Sheet1'

```

\*\*读写 SQL 查询或数据库表 (Read and Write to SQL Query or Database Table)\*\*

```python

>> from sqlalchemy import create\_engine

>> engine = create\_engine('sqlite:///:memory:') # 创建内存SQLite引擎

>> pd.read\_sql('SELECT \* FROM my\_table', engine) # 执行SQL查询并读取结果

>> pd.read\_sql\_table('my\_table', engine) # 读取整个数据库表

>> pd.read\_sql\_query('SELECT \* FROM my\_table', engine) # 执行SQL查询并读取结果

# read\_sql() 是 read\_sql\_table() 和 read\_sql\_query() 的便捷包装

>> df.to\_sql('myDf', engine) # 将DataFrame写入SQL数据库表

```

---

\*\*翻译说明和修正：\*\*

1. \*\*术语翻译：\*\* 使用了 Pandas 中文社区常用的术语，如 `Series` (系列), `DataFrame` (数据框), `index` (索引), `column` (列), `loc`/`iloc` (保留原名), `drop` (删除), `sort` (排序), `rank` (排名), `apply` (应用), `fill\_value` (填充值), `read\_csv`/`to\_csv` (读/写CSV) 等。

2. \*\*拼写错误修正：\*\*

\* `Interschevly` -> `Interactively` (交互式地)

\* `Beiglem` -> `Belgium` (比利时)

\* `Bancassia` -> `Brussels` (布鲁塞尔)

\* `Redium` -> `Country` (原表格表头错误，修正为上下文一致的 `Country`)

\* `Breadth` -> `Capital` (原表格表头错误，修正为上下文一致的 `Capital`)

\* `Process` -> `Population` (原表格表头错误，修正为上下文一致的 `Population`)

\* `Output` -> 移除 (原表格表头冗余且错误)

\* `maps` -> `mode` (众数) 和 `numdam` -> `median` (中位数) / `mean` (均值) (在摘要统计部分)

\* `Recelile` -> `ExcelFile`

\* `to\_excel` -> `to\_excel` (方法名修正)

\* `to\_sql('myOf'` -> `to\_sql('myDf'` (表名修正)

3. \*\*代码语法错误修正：\*\*

\* `index='[a', 'b', 'c', 'd']` -> `index=['a', 'b', 'c', 'd']` (正确的列表语法)

\* `columns='Country, 'Capital', 'Population'` -> `columns=['Country', 'Capital', 'Population']` (正确的列表语法)

\* `df.loc[0],[0]`, `df.ist[0],[0]`, `df.loc[0], ['Country']`, `df.ist[0], ['Country'])`, `df.is[2]`, `df.is[1], 'Capital']` -> 修正为 `df.iloc[0, 0]`, `df.loc[0, 'Country']`, `df.iloc[2]`, `df.iloc[1, 1]` (统一使用正确的 `loc` 和 `iloc` 方法，并修正了括号和逗号位置)。特别注意 `ist` 和 `is` 是明显的拼写错误，应为 `iloc`。

\* `d=[(s > 1)]` -> `s[s > 1]` (正确的布尔索引语法)

\* `d[1: (s > 2)]` -> 移除 (该行语法混乱且上下文不明，可能是错误或冗余)

\* `df[df['Population']>12000000000]` -> `df[df['Population'] > 1200000000]` (修正了人口阈值，印度人口约为13亿，巴西约2亿，12亿作为阈值更合理)

\* `engine = create\_engine('sqlite://memory:')` -> `engine = create\_engine('sqlite:///:memory:')` (修正SQLite内存数据库连接字符串)

4. \*\*数据修正：\*\*

\* 修正了比利时、印度、巴西的人口数据（原数据格式混乱且数值过大/过小）。

\* 修正了首都名称的拼写 (`Bancassia` -> `Brussels`)。

5. \*\*格式优化：\*\*

\* 重组了混乱的表格结构，使其清晰可读（特别是 DataFrame 示例和 Selection 操作表）。

\* 添加了注释 (`# ...`) 说明修正的地方。

\* 对代码块和输出结果进行了更好的视觉区分。

\* 统一了术语表述（如将 `Getting` 翻译为“获取”并在表格中统一使用）。

\* 在术语首次出现或重要概念处添加了英文原名（如 `Series (系列)`）。