1. Pandas库简介

1.1 Pandas是什么?

Pandas是一个开源的、用于数据处理和分析的Python库，特别适合处理表格类数据。它建立在NumPy数组之上，提供了高效的数据结构和数据分析工具，使得数据操作变得更加简单、便捷和高效。

1.2 核心

1.2.1 数据结构

1. **Series**：一维数组，可以存储任何数据类型（整数、字符串、浮点数等），每个元素都有一个与之对应的标签（索引）。
2. **DataFrame**：二维表格型数据结构，可以视为多个 Series 对象的集合，每一列都是一个 Series。每列可以有不同的数据类型，并且有行和列的标签。

1.2.2 数据操作

- **读取和保存数据**：支持多种数据格式，如 CSV、Excel、SQL 数据库、JSON 等。
- **数据选择和过滤**：提供灵活的索引和条件筛选功能，方便数据的提取和过滤。
- **数据清洗**：提供了处理缺失数据、重复数据、异常值等数据清洗功能。
- **数据转换**：通过 `apply()`，`map()`，`replace()` 等方法进行数据转换。
- **数据合并**：使用 `concat()`，`merge()`，`join()` 等方法进行数据的横向和纵向合并。
- **聚合和分组**：使用 `groupby()` 结合 `agg()`，`transform()` 等方法进行数据的分组和聚合。

1.3 主要特点

1. **数据结构**：Pandas提供了两种主要的数据结构：Series（一维数组）和DataFrame（二维表格）。
2. **数据操作**：支持数据的增、删、改、查等操作，以及复杂的数据转换和清洗。
3. **数据分析**：提供丰富的数据分析方法，如聚合、分组、透视等。
4. **文件读取与写入**：支持多种文件格式（如CSV、Excel、SQL等）的读取和写入。
5. **与其他库集成良好**：Pandas 与许多其他三方库（如 NumPy、Matplotlib、Scikit-learn等）无缝集成，形成了一个强大的数据科学生态系统。
6. **强大的社区支持**：Pandas 拥有庞大的开发者社区，提供丰富的资源和学习材料。

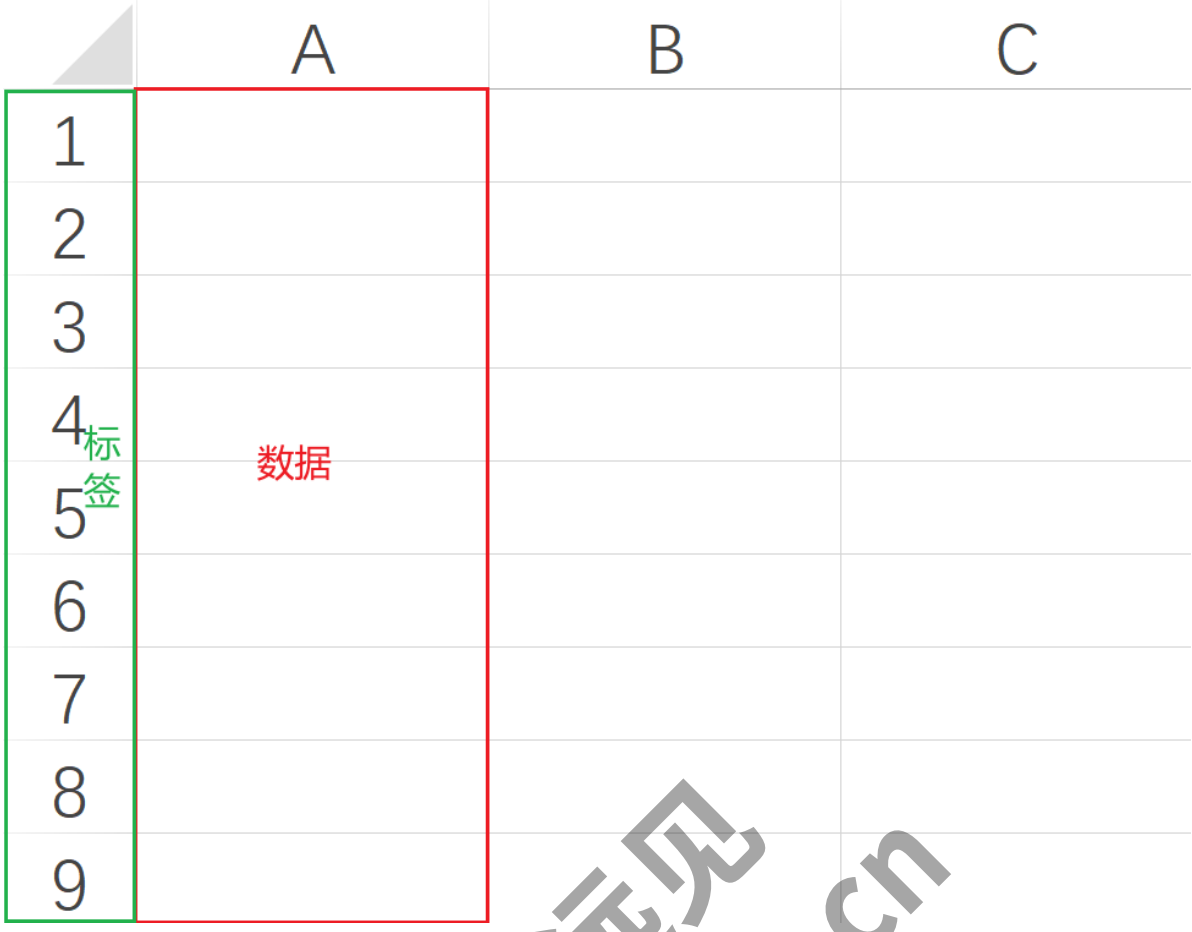
2. Pandas的安装

Pandas是一个第三方库，并不属于Python的系统库，需要进行安装，可输入以下命令进行安装：

```
pip install pandas
```

3. 一维数组Series

一维数组的结构可以理解为excel中的一列数据：



3.1 Series的创建

在Pandas中，一维数组的创建离不开Pandas库中的Series类。

```
class pandas.Series(data=None, index=None, dtype=None, name=None, copy=None,
fastpath=False)
```

- **data**：可以是以下几种类型的数据：
  - 标量值，如整数或字符串
  - Python列表或元组
  - Python字典
  - 1d-Ndarray
- **index**：数组或列表，用于定义Series的索引。如果未提供，则默认为从0开始的整数索引。
- **dtype**：指定Series的数据类型。
- **name**：给Series一个名字，用于后续的索引和操作。
- **copy**：布尔值，默认为False。如果为True，则复制数据；如果为False，则尽可能避免复制数据，仅影响Ndarray输入。
- **fastpath**：布尔值，默认为False，通常不需要用户指定。它是Pandas库内部使用的一个优化标志，当设置为`True`时，允许`Series`构造函数绕过一些检查和验证步骤，加快Series的创建速度。但由于这个参数跳过了某些安全检查，因此在正常使用中，如果在创建Series时设置了`fastpath=True`，而传入的数据又不符合预期，则可能会导致不可预测的行为或错误。

3.1.1 使用标量创建Series

```
import pandas as pd

# 定义一个变量data，并赋值为0
data = 0

# 使用pandas库的Series函数创建一个Series对象
# 第一个参数是要作为Series数据的值，这里是0
# 第二个参数是一个列表，用于指定Series的索引，这里指定了索引为['a', 'b', 'c']
series = pd.Series(data, index=['a', 'b', 'c'])

# 打印输出创建好的Series对象
print(series)
```

3.1.2 使用列表或元组创建Series

```
import pandas as pd

# 定义一个列表data1，其中包含了整数1到5
data1 = [1, 2, 3, 4, 5]

# 定义一个元组data2，其中包含了整数5到1
data2 = (5, 4, 3, 2, 1)

# 使用pandas库的Series函数创建第一个Series对象series1
# 第一个参数data1是要作为Series1数据的值，即列表[1, 2, 3, 4, 5]
# 第二个参数是一个列表['a', 'b', 'c', 'd', 'e']，用于指定Series1的索引
series1 = pd.Series(data1, index=['a', 'b', 'c', 'd', 'e'])

# 使用pandas库的Series函数创建第二个Series对象series2
# 第一个参数data2是要作为Series2数据的值，即元组(5, 4, 3, 2, 1)
# 第二个参数是一个列表['a', 'b', 'c', 'd', 'e']，用于指定Series2的索引
series2 = pd.Series(data2, index=['a', 'b', 'c', 'd', 'e'])

# 打印输出创建好的第一个Series对象series1
print(series1)

# 打印输出创建好的第二个Series对象series2
print(series2)
```

3.1.3 使用字典创建Series

使用字典创建Series时，字典的键就是索引，字典的值就是该索引对应的值。如果使用字典创建Series，并且指定了与字典的键不同的index参数，那么生成的Series数组中的数据就是以index参数的值为索引，但索引所对应的值是NaN。

在Pandas中，`NaN`（Not a Number）是一个特殊的浮点数，用于表示缺失数据或无效数据。`NaN` 是IEEE 浮点标准的一部分，Pandas 使用 `NaN` 来表示数据集中缺失或未定义的值。

```
import pandas as pd

# 定义一个字典data，其中包含了三个键值对，键分别为'a'、'b'、'c'，对应的值分别为1、2、3
data = {'a': 1, 'b': 2, 'c': 3}

# 使用pandas库的Series函数创建一个Series对象。
# 当传入一个字典作为参数时，字典的键会自动成为Series的索引，字典的值会成为对应索引下的数值。
series = pd.Series(data)

# 打印输出创建好的Series对象
print(series)
```

3.1.4 使用Ndarray创建Series

```
import pandas as pd
import numpy as np

# 使用numpy的array函数创建一个一维数组data，数组中包含了整数1到5。
data = np.array([1, 2, 3, 4, 5])

# 使用pandas库的Series函数创建一个Series对象。
# 传入刚刚创建的一维numpy数组data作为参数，此时会将该数组的数据依次作为Series的数值，
# 并且会自动生成一个默认的整数索引（从0开始，依次递增，与数组元素的下标对应）。
series = pd.Series(data)

# 打印输出创建好的Series对象，以便查看其具体内容，包括索引和对应的数据值等信息。
print(series)
```

3.2 Series的属性

3.2.1 index

返回Series中的索引。

```
import pandas as pd

# 创建Series数组
series = pd.Series([1, 2, 3], index=['a', 'b', 'c'])

# 打印输出该Series对象的索引。
# Series对象的index属性可以获取到它的索引信息，这里会输出索引['a', 'b', 'c']。
print(series.index)

# 重新给Series对象的索引赋值。
# 将原来的索引['a', 'b', 'c']修改为新的索引['e', 'f', 'g']，
# 这会改变Series对象中每个数据元素对应的索引标识。
series.index = ['e', 'f', 'g']

# 再次打印输出修改索引后的Series对象，
print(series)
```

3.2.2 values

用于返回Series中的数据，返回的数据将以Ndarray数组的形式存在。

```
import pandas as pd

# 创建Series数组
series = pd.Series([1, 2, 3], index=['a', 'b', 'c'])

# 打印输出该Series对象的数据。
print(series.values)
print(type(series.values))

# # values无法通过直接赋值的方式去修改
# series.values = ['e', 'f', 'g']
# print(series)
```

3.2.3 name

用于返回Series的名称，如果创建时指定了name参数，那么该属性的返回值就是name参数，如果没有指定则为None。

```
import pandas as pd

# 创建Series数组
series = pd.Series([1, 2, 3], index=['a', 'b', 'c'])

# 打印输出该Series对象的name属性值。
print(series.name)

# 给Series对象的name属性赋值为'test'。
# 这相当于给这个Series对象起了一个名字，方便在后续处理或展示数据时进行识别和区分。
series.name = 'test'

# 再次打印输出该Series对象的name属性值，
print(series.name)
```


3.2.4 dtype和dtypes

对于Series来说，dtype和dtypes的作用是一样的，都是用来返回Series对象的数据类型。

需要注意的是：该属性是只读属性，不可以通过直接赋值的方式去修改数据类型。

```
import pandas as pd

# 创建Series数组
series = pd.Series([1, 2, 3], index=['a', 'b', 'c'])

# 打印输出该Series对象的数据类型。
# Series对象的dtype属性可以获取到其数据的类型，这里创建的Series数据是整数类型，所以会输出'int64'。
print(series.dtype)

# Series对象的dtype属性是只读属性，不可以直接通过赋值的方式来改变数据类型。
# 要改变Series对象的数据类型，需要使用astype等合适的方法来进行转换操作。
```

```
# series.dtype = 'float32'
#
# print(series.dtype)
```

3.2.5 shape

用于描述Series的形状。

```
import pandas as pd

# 创建Series数组
series = pd.Series([1, 2, 3], index=['a', 'b', 'c'])

# 打印数组的形状
print(series.shape)
```

3.2.6 size

用于返回Series的元素数量，该返回值是一个整数。

```
import pandas as pd

# 创建Series数组
series = pd.Series([1, 2, 3], index=['a', 'b', 'c'])

# 打印数组元素的元素数量
print(series.size)
```

3.2.7 empty

用来表示Series数组是否为空，返回值一个布尔值，如果数组里一个元素都没有就返回True，否则返回False。

```
import pandas as pd

# 创建Series数组
series = pd.Series()

# 判断数组是否为空
print(series.empty)
```

3.2.8 hasnans

用于返回数组中是否包含NaN值，如果数组中存在NaN，那么返回True，否则返回False。

```
import pandas as pd
import numpy as np

# 创建Series数组
series = pd.Series([1, 2, 3, np.nan], index=['a', 'b', 'c', 'd'])

# 判断数组是否存在NaN值
print(series.hasnans)
```

3.2.9 is_unique

用于返回数组中的元素是否为独一无二的，如果所有的元素都是独一无二的，即数组中没有重复元素，那么就返回True，否则返回False。

```
import pandas as pd

# 创建Series数组
series = pd.Series(['a', 'b', 'c'])

# 判断数组中是否存在重复的元素
print(series.is_unique)
```

3.2.10 nbytes

用于返回该Series对象中所有数据占用的总字节数。

```
import pandas as pd

# 创建Series数组
series = pd.Series([1, 2, 3, 4, 5], dtype='int64')

# 获取数组元素所占用的内存大小
print(series.nbytes)
```

3.2.11 axes

用于返回series对象行轴标签的列表。

```
import pandas as pd

# 创建Series数组
series = pd.Series([1, 2, 3, 4, 5], index=['a', 'b', 'c', 'd', 'e'],
dtype='int64')

# 获取数组的行轴标签
print(series.axes)
```

3.2.12 ndim

返回Series数组的维度，对于Series数组来说，它的维度始终为1。

```
import pandas as pd

# 创建Series数组
series = pd.Series([1, 2, 3, 4, 5], index=['a', 'b', 'c', 'd', 'e'],
dtype='int64')

# 获取数组的索引
print(series.ndim)
```

3.2.13 array

用于返回Series的底层数组，包括数组的元素、数组的长度及数组元素的数据类型。

```
import pandas as pd
import numpy as np

# 创建一个NumPy数组
data = np.array([1, 2, 3, 4, 5])

# 使用NumPy数组创建一个Pandas Series
series = pd.Series(data)

# 打印Series的底层数据作为一个Pandas的Array对象
print(series.array)

# 打印上述Array对象的类型
print(type(series.array))
```

3.2.14 attrs

返回series的自定义属性，可以用来存储额外的说明性数据。

```
import pandas as pd

# 创建一个包含整数1到5的Pandas Series
series = pd.Series([1, 2, 3, 4, 5])

# 打印额外的属性
print(series.attrs)

# 给Series添加额外的属性，这里添加了来源和时间信息
series.attrs = {'source': 'file1', 'time': '19:27:27'}

# 打印Series本身，将显示其数据和索引
print(series)

# 打印Series的额外属性
print('额外属性', series.attrs)
```


3.2.15 is_monotonic_decreasing

返回一个布尔值，表示Series是否按降序排列。

```
import pandas as pd

# 创建一个Pandas Series，其值从5递减到1
series = pd.Series([5, 4, 3, 2, 1])

# 打印检查Series是否单调递减的结果
# 由于Series中的值是递减的，所以这个表达式将返回True
print(series.is_monotonic_decreasing)
```

3.2.16 is_monotonic_increasing

返回一个布尔值，表示Series 是否按升序排列。

```
import pandas as pd

# 创建一个Pandas Series，其值从1递增到5
series = pd.Series([1, 2, 3, 4, 5])

# 打印检查Series是否单调递增的结果
# 由于Series中的值是递增的，所以这个表达式将返回True
print(series.is_monotonic_increasing)
```

3.3 Series中元素的索引与访问

3.3.1 位置索引

可以使用整数索引来访问Series中的元素，就像访问列表一样。

```
import pandas as pd

# 创建一个Series
series = pd.Series([10, 20, 30, 40, 50])

# 通过位置索引获取元素
print(series[0])
print(series[2])
```

3.3.2 标签索引

除了使用位置索引之外，还可以使用标签进行索引，与访问字典中的元素类似。

```
import pandas as pd

# 创建一个带有标签的Series
series = pd.Series([10, 20, 30, 40, 50], index=['a', 'b', 'c', 'd', 'e'])

# 通过标签索引获取元素
print(series['a'])
print(series['c'])
```

3.3.3 切片索引

Series对象的切片方式有两种，第一种是使用位置切片，其使用方法与列表的切片类似；第二种是使用标签切片，其语法与位置切片类似，都是`start:stop`，且开始值与终止值可以省略，但与位置切片不同的是，标签切片的范围是左右都闭合，即既包含start，又包含stop，而位置切片是左闭右开，只包含start，不包含stop。

```
import pandas as pd

# 创建一个带有标签的Series
series = pd.Series([10, 20, 30, 40, 50], index=['a', 'b', 'c', 'd', 'e'])

# 通过位置切片
print(series[:])

# 通过标签切片
print(series['b':'d'])
```

3.3.4 loc与iloc

loc与iloc也是Series对象的属性，它们的作用就是用来访问Series中的元素，loc是基于标签的索引，iloc是基于位置的索引。

```
import pandas as pd

# 创建一个带有标签的Series
series = pd.Series([10, 20, 30, 40, 50], index=['a', 'b', 'c', 'd', 'e'])

# 使用.loc通过标签索引
print(series.loc['a'])

# 使用.iloc通过位置索引
print(series.iloc[0:2])
print(series.iloc[2])
```

3.3.5 at与iat

at与iat也是Series对象的属性，可以用来访问元素，at是基于标签的索引，iat是基于位置的索引。

```
import pandas as pd

# 创建一个带有标签的Series
series = pd.Series([10, 20, 30, 40, 50], index=['a', 'b', 'c', 'd', 'e'])

# 使用.at通过标签索引
print(series.at['a'])

# 使用.iat通过位置索引
print(series.iat[0])
print(series.iat[2])
```

3.3.6 head

head是Series对象的方法，用于快速查看`Series`数据的开头部分内容。

```
series.head(n=None)
```

- `n`：是可选参数，用于指定要返回的行数。如果不提供该参数，默认值为`5`。

```
import pandas as pd

# 创建一个Series对象
data = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
index = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']
series = pd.Series(data, index=index)

# 使用head函数查看前5行数据
print(series.head())
```

3.3.7 tail

tail的用法与head类似，但不同的是，它用于快速查看`Series`数据的末尾部分内容。

```
series.tail(n=None)
```

- `n`：是可选参数，用于指定要返回的行数。若不提供该参数，默认值为`5`。

```
import pandas as pd

# 创建一个Series对象
data = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
index = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']
series = pd.Series(data, index=index)

# 使用tail函数查看后5行数据
print(series.tail())
```

3.3.8 isin

该函数用于判断`Series`中的每个元素是否在指定的一组值中，它会返回一个与原`Series`长度相同的布尔型`Series`，其中对应位置为`True`表示该位置的元素在指定的值集合中，`False`则表示不在。

```
series.isin(values)
```

- `values`：是一个可迭代对象（如列表、元组、集合等），用于指定要进行判断的一组值。

```
import pandas as pd

# 创建一个Series对象
data = [10, 20, 30, 40, 50]
series = pd.Series(data)

# 指定要判断的一组值
values_to_check = [20, 40]

# 使用isin函数进行判断
result = series.isin(values_to_check)

print(result)
```

3.3.9 get

`Series.get` 方法用于通过标签来获取Series中的元素。

```
Series.get(key, default=None)
```

- `key`：你想要获取的元素的标签。
- `default`：可选参数，如果`key`不在标签中，返回这个默认值。如果没有指定，默认为`None`。

```
import pandas as pd

# 创建一个示例Series
s = pd.Series(['apple', 'banana', 'cherry'], index=[1, 2, 3])

# 使用get方法获取元素
print(s.get(2))
print(s.get(4, 'Not Found'))
```

3.4 数据操作

3.4.1 数据清洗

数据清洗指对Series对象中的某些值进行删除、修改等操作，分别使用以下方法：

1. dropna()：删除包含NaN值的行。

```
series.dropna(axis=0, inplace=False)
```

- `axis`：可选参数，用于指定按哪个轴删除缺失值。对于`Series`对象，因为它是一维数据结构，只有一个轴，所以此参数默认值为`0`，且一般不需要修改这个参数（在处理`DataFrame`时该参数才有更多实际意义，如`axis = 0`表示按行删除，`axis = 1`表示按列删除）。
- `inplace`：可选参数，用于指定是否在原`Series`对象上进行操作。如果`inplace = True`，则会直接在原`Series`上删除缺失值，原`Series`会被修改；如果`inplace = False`（默认值），则会返回一个删除了缺失值的新`Series`，原`Series`保持不变。

```
import pandas as pd
import numpy as np

# 创建一个包含缺失值的Series对象
data = [1, np.nan, 3, 4, np.nan]
series = pd.Series(data)

# 使用dropna函数删除缺失值，返回新的Series
new_series = series.dropna()

print(new_series)
```

2. fillna()：填充NaN值。

```
Series.fillna(value=None, method=None, axis=None, inplace=False, limit=None,
downcast=None)
```

- `value`：用于填充缺失值的标量值或字典。如果传递的是字典，则字典的键应该是要填充的标签，值是用于填充的值。
- `method`：字符串，表示填充的方法。可选值包括：
  - `pad` / `ffill`：用前一个非缺失值去填充缺失值。
  - `bfill` / `backfill`：用后一个非缺失值去填充缺失值。
- `axis`：填充的轴，对于 `Series` 对象来说，这个参数通常不需要指定，因为 `Series` 是一维的。
- `inplace`：布尔值，表示是否在原地修改数据。如果为 `True`，则直接在原 `Series` 上修改，不返回新的对象。
- `limit`：整数，表示最大填充量。如果指定，则只填充前 `limit` 个缺失值。
- `downcast`：字典，用于向下转换数据类型。例如，可以将 `float64` 转换为 `float32`。

```
import pandas as pd
import numpy as np

# 创建一个包含缺失值的 Series
s = pd.Series([1, np.nan, 3, np.nan, 5])

# 使用标量值填充
filled_with_scalar = s.fillna(0)
print(filled_with_scalar)


# 使用前向填充
filled_with_ffill = s.fillna(method='ffill')
print(filled_with_ffill)


# 使用后向填充
filled_with_bfill = s.fillna(method='bfill')
print(filled_with_bfill)


# 使用 limit 参数
filled_with_limit = s.fillna(value=0, limit=1)
print(filled_with_limit)
```


3. isnull()：检测Series对象中的缺失值，它会返回一个布尔型Series，其中每个元素表示原Series对应位置的值是否为缺失值（NaN）。

```
Series.isnull()
```

```
import pandas as pd
import numpy as np

# 创建一个包含缺失值的Series
s = pd.Series([1, 2, np.nan, 4, np.nan])

# 使用isnull()方法检测缺失值
missing_values = s.isnull()

print(missing_values)
```

4. drop_duplicates()：用于去除Series对象中的重复项。

```
Series.drop_duplicates(keep='first', inplace=False, ignore_index=False)
```

- keep：可选参数，决定了如何处理重复项。有三个选项：
  - `'first'`：默认值，保留第一次出现的重复项。
  - `'last'`：保留最后一次出现的重复项。
  - `False`：不保留任何重复项，即删除所有重复项。
- `inplace`：布尔值，默认为`False`。如果设置为`True`，则直接在原始Series上进行操作，返回值为`None`。如果设置为`False`，则返回一个新的Series，不修改原始Series。
- `ignore_index`：布尔值，默认为`False`。如果设置为`True`，则结果的索引将被重新设置，以反映删除重复项后的新顺序。如果设置为`False`，则保留原始索引。

```
import pandas as pd

series = pd.Series(['a', 'b', 'b', 'c', 'c', 'c', 'd'])
series_unique = series.drop_duplicates(keep='first')
print(series_unique)
```

3.4.2 数据转换

1. replace()：替换特定的值、一系列值或者使用字典映射进行替换。

```
Series.replace(to_replace=None, value=None, inplace=False, limit=None,
regex=False, method='pad')
```

- `to_replace`：要替换的值，可以是以下类型：
  - 标量：单个值。
  - 列表：一系列值。
  - 字典：键是要替换的值，值是替换后的新值。
  - 正则表达式：如果`regex=True`，则可以使用正则表达式匹配要替换的值。
- `value`：替换后的新值，可以是标量或字典。如果`to_replace`是列表，则`value`也应该是相同长度的列表。
- `inplace`：布尔值，表示是否在原地修改数据。如果为`True`，则直接在原`Series`上修改，不返回新的对象。
- `limit`：整数，表示最大替换量。如果指定，则只替换前`limit`个匹配的值。
- `regex`：布尔值，表示是否将`to_replace`解释为正则表达式。
- `method`：字符串，表示填充的方法，在`to_replace`参数是一个标量、列表或元组，同时`value`参数设置为`None`时，可以使用`method`参数来指定填充缺失值（NaN）的方式。可选值包括：
  - `pad` / `ffill`：用前一个非缺失值去填充缺失值。
  - `bfill`：用后一个非缺失值去填充缺失值。

```
import pandas as pd

# 创建一个 Series
s = pd.Series([1, 2, 3, 4, 5])

# 进行替换操作
replaced = s.replace(to_replace=2, value=20)
print(replaced)
```

2. astype()：用于将 `Series` 的数据类型（dtype）转换或转换为另一种类型。

```
Series.astype(dtype, copy=True, errors='raise')
```

- **dtype**：你希望将 `Series` 转换成的数据类型。
- **copy**：布尔值，默认为 `True`。如果为 `False`，则转换数据类型时不会复制底层数据（如果可能的话）。
- **errors**：默认为 'raise'，控制当转换失败时的行为。如果设置为 'raise'，则在转换失败时会抛出异常；如果设置为 'ignore'，转换失败后则返回原始 `Series`，不做任何修改。

```
import pandas as pd

s = pd.Series([1, 2, 3, 4, 5])

# 将整数转换为字符串
s_str = s.astype(float)
print(s_str)
```

3. transform()：用于对Series中的数据进行转换操作，并返回与原始Series具有相同索引的新Series。

```
Series.transform(func, axis=0, *args, **kwargs)
```

- `func`：应用于Series的函数。这个函数可以是内置函数，或者自定义的函数。
- `axis`：对于Series来说，这个参数不起作用，因为Series是一维的。在DataFrame上使用时，`axis=0`（默认）表示按列应用函数，`axis=1`表示按行应用函数。
- `*args, **kwargs`：这些参数会被传递给`func`函数。

```
import pandas as pd


# 自定义一个函数，返回每个元素的平方
def square(x):
    return x ** 2


# 创建一个Series
s = pd.Series([1, 2, 3, 4, 5])


# 使用自定义函数对Series进行平方变换
transformed_series = s.transform(square)

print(transformed_series)
```

3.4.3 数据排序

1. sort_values()：按照值对 `Series` 进行排序。

```
Series.sort_values(axis=0, ascending=True, inplace=False, kind='quicksort',
na_position='last', ignore_index=False, key=None)
```

- **axis**：默认为 0。对于 Series，这个参数不起作用，因为 Series 是一维的，而 `sort_values` 总是在 axis=0 上操作。
- **ascending**：布尔值，默认为 `True`。如果是 `True`，则按照升序排列；如果是 `False`，则按照降序排列。
- **inplace**：布尔值，默认为 `False`。如果为 `True`，则排序将直接在原始 Series 上进行，不返回新的 Series。
- **kind**：排序算法，{'quicksort', 'mergesort', 'heapsort', 'stable'}，默认为 'quicksort'。决定了使用的排序算法。
- **na_position**：{'first', 'last'}，默认为 'last'。这决定了 NaN 值的放置位置。
- **ignore_index**：布尔值，默认为 `False`。如果为 `True`，则排序后的 Series 将重置索引，使其成为默认的整数索引。
- **key**：函数，默认为 `None`。如果指定，则这个函数将在排序之前应用于每个值，并且排序将基于这些函数返回的值。

```
import pandas as pd
import numpy as np


def square(x):
    return x ** 2

s = pd.Series([-3, 1, 4, 1, np.nan, 9], index=['a', 'b', 'c', 'd', 'e', 'f'])

# 排序并重置索引
sorted_s = s.sort_values(ignore_index=True, key=square)
print(sorted_s)
```

2. sort_index()：按照索引的顺序对数据进行排序。

```
Series.sort_index(axis=0, level=None, ascending=True, inplace=False,
kind='quicksort', na_position='last', sort_remaining=True, ignore_index=False,
key=None)
```

- **axis** : 默认为0，对于 Series，这个参数不起作用。
- **level** : 默认为 None，如果索引是多级索引（也称为层次化索引或 MultiIndex），则可以指定要排序的级别。
- **ascending** : 默认为 True，如果为 True，则按升序排序；如果为 False，则按降序排序。对于多级索引，可以传递一个布尔值列表，以指定每个级别的排序顺序。
- **inplace** : 默认为 False，如果为 True，则直接在原对象上进行修改，不会返回一个新的对象。
- **kind** : {'quicksort', 'mergesort', 'heapsort'}，默认为 'quicksort'，指定排序算法。'quicksort' 是最快的通用排序算法，但不是稳定的；'mergesort' 是稳定的，但可能比 'quicksort' 慢；'heapsort' 是原地排序算法，但通常比其他两个选项慢。
- **na_position** : {'first', 'last'}, 默认为 'last'，指定 NaN 值应该排在排序结果的开头还是结尾。
- **sort_remaining** : 默认为 True，对于多级索引，如果为 True，在该level排序后，在排序的基础上对剩下的级别的元素还会排序。

- **ignore_index**：默认为 False，如果为 True，则排序后的结果将不再保留原始索引，而是使用默认的整数索引。
- **key**：函数，默认为 `None`。如果指定，则这个函数将在排序之前应用于每个值，并且排序将基于这些函数返回的值。

```
import pandas as pd
import numpy as np

arrays = [np.array(['qux', 'qux', 'foo', 'foo',
                    'baz', 'baz', 'bar', 'bar']),
          np.array(['two', 'one', 'two', 'one',
                    'two', 'one', 'two', 'one'])]


s = pd.Series([1, 2, 3, 4, 5, 6, 7, 8], index=arrays)
print(s)
res = s.sort_index(level=1, ascending=True, sort_remaining=True,
ignore_index=False)
print(res)
```

3.4.4 数据筛选

可以使用一个布尔数组来选择满足条件的元素。

```
import pandas as pd

# 创建一个带有标签的Series
series = pd.Series([10, 20, 30, 40, 50], index=['a', 'b', 'c', 'd', 'e'])

# 选择大于30的元素
print(series[series > 30])
```

3.4.5 数据拼接

concat()：用于将多个Pandas对象（如Series或DataFrame）沿着一个轴连接起来的函数。

```
pandas.concat(objs, *, axis=0, join='outer', ignore_index=False, keys=None,
levels=None, names=None, verify_integrity=False, sort=False, copy=None)
```

- `objs`：参与连接的Pandas对象的列表或元组。例如，可以是多个Series或DataFrame。
- `axis`：{0或'index', 1或'columns'}，表示连接的轴，默认为0。0表示沿着行方向连接（索引轴），1表示沿着列方向连接（列轴）。
- `join`：{'inner', 'outer'}，默认为'outer'。如何处理其他轴上的索引。'outer'表示并集，保留所有索引；'inner'表示交集，只保留所有对象共有的索引。
- `ignore_index`：布尔值，默认为False。如果为True，则不保留原索引，而是创建一个新索引，可以避免重复的索引。
- `keys`：序列，默认为None。用于创建分层索引的键。如果提供了keys，则生成的DataFrame或Series将具有分层索引。
- `levels`：序列列表，默认为None。用于构造分层索引的特定级别，如果设置了keys，则默认为keys。
- `names`：列表，默认为None。生成的分层索引中的级别名称。如果提供了keys，表示使用keys作为索引名称。

- `verify_integrity`：布尔值，默认为False。如果为True，则检查新连接的轴是否包含重复的索引，如果发现重复，则引发ValueError。这在确保数据没有重复时很有用。
- `sort`：布尔值，默认为False。在连接之前是否对非连接轴上的索引进行排序。这在连接多个DataFrame时很有用，可以确保索引是有序的。
- `copy`：布尔值，默认为None。如果为True，则不管是否需要，都会复制数据。如果为False，则尽量避免复制数据，除非必要。None表示自动选择。

```
import pandas as pd

# 创建三个Series
s1 = pd.Series([1, 2, 3], index=['a', 'b', 'c'])
s2 = pd.Series([4, 5, 6], index=['c', 'd', 'f'])
s3 = pd.Series([7, 8, 9], index=['e', 'f', 'g'])

# 使用concat连接Series
result = pd.concat([s1, s2, s3])

print(result)
```

3.5 统计计算

3.5.1 count

用于计算`Series`中非NaN（非空）值的数量。

```
import pandas as pd

# 创建一个包含空值的Series
s = pd.Series([1, 2, None, 4, None])

# 使用count()函数计算非空值的数量
count_non_na = s.count()
print(count_non_na)
```

3.5.2 sum

`sum()` 函数会计算所有值的总和。

```
Series.sum(axis=None, skipna=True, numeric_only=None, min_count=0)
```

- `axis`：对于`Series`对象来说，这个参数通常不起作用，因为`Series`是一维的。它主要在`DataFrame`对象中用于指定操作的轴（0表示按列求和，1表示按行求和）。
- `skipna`：布尔值，默认为`True`。如果为`True`，则在计算总和时会忽略`NaN`值，如果为False，则返回NaN。
- `numeric_only`：布尔值，默认为`None`。如果为`True`，则只对数字类型的数据进行计算，只针对DataFrame。
- `min_count`：int值，默认为0。表示在计算总和之前，至少需要多少个非`NaN`值。如果非`NaN`值的数量小于`min_count`，则结果为`NaN`。

```
import pandas as pd

s = pd.Series([1, 2, None, 4, 5])
total = s.sum(min_count=5)
print(total)
```

3.5.3 mean

`mean()` 函数会计算所有值的平均值。

```
Series.mean(axis=None, skipna=True, numeric_only=None)
```

- `axis`：对于`Series`对象来说，这个参数通常不起作用，因为`Series`是一维的。它主要在`DataFrame`对象中用于指定操作的轴（0表示按列计算平均值，1表示按行计算平均值）。
- `skipna`：布尔值，默认为`True`。如果为`True`，则在计算平均值时会忽略`NaN`值，如果为False，则返回NaN。
- `numeric_only`：布尔值，默认为`None`。如果为`True`，则只对数字类型的数据进行计算，只针对DataFrame。

```
import pandas as pd

s = pd.Series([1, 2, 3, 4, 5])
average = s.mean()
print(average)
```

3.5.4 median

`median()` 函数用于计算DataFrame或Series中的中位数。

```
Series.median(axis=0, skipna=True, numeric_only=False)
```

- `axis`：对于`Series`对象来说，这个参数通常不起作用，因为`Series`是一维的。它主要在`DataFrame`对象中用于指定操作的轴（0表示按列计算中位数，1表示按行计算中位数）。
- `skipna`：布尔值，默认为True。如果为True，则在计算中位数之前会排除NaN值，如果为False，则返回NaN。
- `numeric_only`：布尔值，默认为`None`。如果为`True`，则只对数字类型的数据进行计算，只针对DataFrame。

```
import pandas as pd

s = pd.Series([1, 2, 3, 4, 5])
median_value = s.median()
print(median_value)
```

3.5.5 min和max

`Series.min()` 函数用于计算Series对象中的最小值，`Series.max()` 函数用于计算Series对象中的最大值。

```
Series.min(axis=0, skipna=True, numeric_only=False)
```

```
Series.max(axis=0, skipna=True, numeric_only=False)
```

- `axis`：对于`Series`对象来说，这个参数通常不起作用，因为`Series`是一维的。它主要在`DataFrame`对象中用于指定操作的轴（0表示按列计算最小值或最大值，1表示按行计算最小值或最大值）。
- `skipna`：布尔值，默认为True。如果为True，则在计算最值之前会排除NaN值，如果为False，则返回NaN。
- `numeric_only`：布尔值，默认为`None`。如果为`True`，则只对数字类型的数据进行计算，只针对DataFrame。

```
import pandas as pd

s = pd.Series([1, 2, 3, 4, 5])
min_value = s.min()
max_value = s.max()
print('最小值是：', min_value)
print('最大值是：', max_value)
```

3.5.6 var

`Series.var()`函数用于计算Series对象的方差。

```
Series.var(axis=None, skipna=True, ddof=1, numeric_only=False)
```

- `axis`：对于`Series`对象，这个参数不会产生任何效果，因为`Series`是一维的。在`DataFrame`中，`axis`用于指定沿着哪个轴计算方差。
- `skipna`：布尔值，默认为`True`。如果为`True`，则在计算方差之前会忽略`NaN`值。如果设置为`False`，计算方差时会包括`NaN`值，通常会导致结果也是`NaN`。
- `ddof`：整数，默认为`1`。Delta Degrees of Freedom，用于贝塞尔校正，以得到样本方差的估计。对于无偏估计（样本方差），`ddof`通常设置为`1`。如果计算总体方差，应该将`ddof`设置为`0`。
- `numeric_only`：布尔值，默认为`False`。如果为`True`，则只对数字类型的数据进行方差计算，忽略非数字类型的数据。

```
import pandas as pd
import numpy as np

s = pd.Series([1, 2, np.nan, 4, 5])
variance = s.var()
print('方差是：', variance)
```

3.5.7 std

`Series.std()`函数用于计算Series对象的标准差。

```
Series.std(axis=None, skipna=True, ddof=1, numeric_only=False, **kwargs)
```

- `axis`：对于Series来说，这个参数不会产生任何效果，因为Series是一维的。它主要用于DataFrame，以指定沿着哪个轴计算标准差。
- `skipna`：布尔值，默认为True。如果为True，则在计算标准差之前会忽略NaN值，如果为False，则返回NaN。

- `ddof`：整数，默认为1。Delta Degrees of Freedom，是贝塞尔校正的参数。对于无偏估计（样本标准差），ddof通常设置为1。
- `numeric_only`：布尔值，默认为False。如果为True，则只对数字类型的数据进行计算，只针对DataFrame。
- `**kwargs`：其他关键字参数。

```
import pandas as pd
import numpy as np

s = pd.Series([1, 2, np.nan, 4, 5])
std_dev = s.std()
print('标准差是：', std_dev)
```

3.5.8 quantile

`Series.quantile()` 方法用于计算Series中数值的分位数。

```
Series.quantile(q=0.5, interpolation='linear')
```

- `q`：这个参数是必需的，它表示要计算的分位数值。可以是单一的数值，也可以是一个数值列表。例如，`q=0.5` 表示计算中位数（50%分位数）。
- `interpolation`：这个参数决定了当所需的分位数位于两个数据点之间时，应该如何插值。默认值是 `'linear'`，表示线性插值。其他选项包括 `'nearest'`（最近的值）、`'lower'`（选择较小的值）、`'higher'`（选择较大的值）、`'midpoint'`（两个值的中间点）。

其计算过程为：

1. 首先将Series中的所有非NaN元素按升序排序。
2. 假设Series的长度为n，我们需要计算第q个分位数的位置下标index：

$$index = q \times (n-1)$$

3. 如果index是一个整数，则直接取该位置上的值；如果index不是整数，则需要进行插值计算。
4. 根据不同的插值方法，确定最终的分位数值：
   - `linear（默认）`：线性插值。假设 `index = i + f`，其中 `i` 是整数部分，`f` 是小数部分。那么分位数为：$Q = x_i + f \times (x_{i+1} - x_i)$，其中$x_i$和$x_{i+1}$是排序后的数据中第 `i` 和 `i+1` 个元素。
   - `lower`：选择较小的那个值。即取x_i。
   - `higher`：选择较大的那个值。即取x_{i+1}。
   - `nearest`：选择最近的那个值。如果 `f <= 0.5`，取x_i，否则取x_{i+1}。
   - `midpoint`：取这两个值的平均数，即：$Q = \frac{x_i + x_{i+1}}{2}$。

```
import pandas as pd

# 创建一个示例Series
s = pd.Series([1, 2, 3, 4, 5])

# 计算0.5分位数
median_value = s.quantile(q=0.5, interpolation='linear')

print(median_value)
```

3.5.9 cummax

该方法用于计算Series中元素的累积最大值，返回一个相同长度的Series，其中每个位置上的值表示从Series开始到当前位置（包括当前位置）的最大值。

```
Series.cummax(axis=None, skipna=True, *args, **kwargs)
```

- `axis`：对于Series来说，这个参数不起作用，因为Series是一维的。在DataFrame上使用时，`axis=0`（默认）表示按列计算，`axis=1`表示按行计算。
- `skipna`：布尔值，默认为True。如果为True，则在计算累积最大值时会忽略NaN值；如果为False，则任何NaN值都会传播，并且会导致结果在该位置及之后的值都为NaN。
- `*args, **kwargs`：这些参数用于兼容性，通常不需要使用。

```
import pandas as pd

# 创建一个Series
s = pd.Series([2, 1, 3, 5, 4, 6])

# 计算累积最大值
cummax_series = s.cummax()

print(cummax_series)
```

3.5.10 cummin

该方法用于计算Series中元素的累积最小值，返回一个相同长度的Series，其中每个位置上的值表示从Series开始到当前位置（包括当前位置）的最小值。

```
Series.cummin(axis=None, skipna=True, *args, **kwargs)
```

- `axis`：对于Series来说，这个参数不起作用，因为Series是一维的。在DataFrame上使用时，`axis=0`（默认）表示按列计算，`axis=1`表示按行计算。
- `skipna`：布尔值，默认为True。如果为True，则在计算累积最小值时会忽略NaN值；如果为False，则任何NaN值都会传播，并且会导致结果在该位置及之后的值都为NaN。
- `*args, **kwargs`：这些参数用于兼容性，通常不需要使用。

```
import pandas as pd

# 创建一个Series
s = pd.Series([2, 1, 3, 5, 4, 6])

# 计算累积最小值
cummin_series = s.cummin()

print(cummin_series)
```

3.5.11 cumsum

用于计算Series中元素的累积和。该方法返回一个相同长度的Series，其中每个位置上的值表示从Series开始到当前位置（包括当前位置）的所有元素的累加和。

```
Series.cumsum(axis=None, skipna=True, *args, **kwargs)
```

- `axis`：对于Series来说，这个参数不起作用，因为Series是一维的。在DataFrame上使用时，`axis=0`（默认）表示按列计算，`axis=1`表示按行计算。
- `skipna`：布尔值，默认为True。如果为True，则在计算累积和时会忽略NaN值；如果为False，则任何NaN值都会传播，并且会导致结果在该位置及之后的值都为NaN。
- `*args, **kwargs`：这些参数用于兼容性，通常不需要使用。

```
import pandas as pd

# 创建一个Series
s = pd.Series([1, 2, NaN, 4, 5])

# 计算累积和
cumsum_series = s.cumsum()

print(cumsum_series)
```

3.5.12 cumprod

用于计算Series中元素的累积乘积。该方法返回一个相同长度的Series，其中每个位置上的值表示从Series开始到当前位置（包括当前位置）的所有元素的累积乘积。

```
Series.cumprod(axis=None, skipna=True, *args, **kwargs)
```

- `axis`：对于Series来说，这个参数不起作用，因为Series是一维的。在DataFrame上使用时，`axis=0`（默认）表示按列计算，`axis=1`表示按行计算。
- `skipna`：布尔值，默认为True。如果为True，则在计算累积乘积时会忽略NaN值；如果为False，则任何NaN值都会导致结果在该位置及之后的值都为NaN。
- `*args, **kwargs`：这些参数用于兼容性，通常不需要使用。

```
import pandas as pd

# 创建一个Series
s = pd.Series([1, 2, NaN, 4, 5])

# 计算累积乘积
cumprod_series = s.cumprod()

print(cumprod_series)
```

3.6 分组和聚合

3.6.1 groupby

用于将Series中的数据分组，并允许你对这些分组进行操作，比如计算每个组的总和、平均值、最大值、最小值等。

```
Series.groupby(by=None, axis=0, level=None, as_index=True, sort=True,
group_keys=True, observed=False, dropna=True)
```

- `by`：确定分组依据，如果`by`是一个函数，它会在对象索引的每个值上调用。如果传递了字典或Series，将使用这些对象的值来确定组。如果传递了长度等于所选轴的列表或 ndarray，则直接使用这些值来确定组。

- `axis`：用于分组的轴。对于Series，这个参数通常设置为0（默认值），因为Series是一维数据结构。
- `level`：如果索引是多级索引（MultiIndex），则此参数用于指定分组所依据的级别，by和level同时只能存在一个，且必须存在一个。
- `as_index`：是否将分组键作为结果的索引，默认值为 `True`。仅与DataFrame输入相关。
- `sort`：是否对结果进行排序。默认值为 `True`。
- `group_keys`：是否在结果中包含分组键。默认值为 `True`。
- `observed`：是否仅包含实际观察到的分类值。默认值为 `False`。
- `dropna`：是否从结果中排除包含 NaN 的组。默认值为 `True`。

```
import pandas as pd

# 创建一个Series对象
data = [10, 20, 10, 30, 20, 10]
series = pd.Series(data)

# 根据Series的值进行分组，并计算每组的计数
grouped = series.groupby(series).count()

print(grouped)
```

```
import pandas as pd

# 创建一个Series对象
data = [10, 20, 30, 40, 50, 60]
index = ['a', 'b', 'a', 'b', 'c', 'c']
series = pd.Series(data, index=index)

# 根据Series的索引进行分组，并计算每组的均值
grouped = series.groupby(series.index).sum()

print(grouped)
```

```
import pandas as pd
import numpy as np

arrays = [np.array(['qux', 'qux', 'foo', 'foo',
                    'baz', 'baz', 'bar', 'bar']),
          np.array(['two', 'one', 'two', 'one',
                    'two', 'one', 'two', 'one'])]


s = pd.Series([1, 2, 3, 4, 5, 6, 7, 8], index=arrays)
res = s.groupby(level=0).count()
print(res)
```

3.6.2 agg

`Series.agg()` 方法用于对Series中的数据进行聚合操作。

```
Series.agg(func=None, axis=0, *args, **kwargs)
```

- `func`：聚合函数或函数列表/字典。可以是一个函数名称（字符串），也可以是实际的函数对象，或者是这些的列表或字典。如果是字典，则键将是输出列的名称，值应该是应用于Series的函数。
- `axis`：整数或字符串，默认为0。由于Series是一维数据结构，这个参数实际上在Series的上下文中不起作用。在DataFrame的上下文中，`axis=0` 表示按列进行聚合，`axis=1` 表示按行进行聚合。
- `*args`：位置参数，可以传递给 `func`。
- `**kwargs`：关键字参数，可以传递给 `func`。

```
import pandas as pd

s = pd.Series([1, 2, 3, 4, 5])

# 计算平均值
result = s.agg('mean')
print(result)

# 计算最大值和最小值
result = s.agg(['max', 'min'])
print(result)

# 使用字典为聚合函数命名
result = s.agg({'Maximum': 'max', 'Minimum': 'min'})
print(result)


# 使用自定义函数并传递额外的参数
def custom_agg(x, power):
    return (x ** power).sum()

result = s.agg(custom_agg, power=2)
print(result)
```

3.7 数据可视化

3.7.1 plot

`Series.plot` 方法是用来绘制Series数据的可视化图表的，该方法提供了灵活的接口，允许用户通过不同的参数来定制图表的类型、样式、布局等，其用法与Matplotlib中的plot相同。

```
Series.plot(*args, **kwargs)
```

主要参数：

- `kind`：图表类型，可以是以下之一：
  - `'line'`：折线图（默认）
  - `'bar'`：柱状图
  - `'barh'`：水平柱状图
  - `'hist'`：直方图
  - `'box'`：箱线图
  - `'kde'`：核密度估计图
  - `'area'`：面积图
  - `'pie'`：饼图
  - `'scatter'`：散点图

    - `'hexbin'`: 六边形箱图
- `ax`: Matplotlib 轴对象，用于在指定的轴上绘制图表。如果不提供，则创建新的轴对象。
- `figsize`: 图表的尺寸，格式为 `(width, height)`，单位为英寸。
- `use_index`: 是否使用 Series 的索引作为 x 轴标签。默认为 `True`。
- `title`: 图表的标题。
- `grid`: 是否显示网格线。默认为 `False`。
- `legend`: 是否显示图例。默认为 `False`。
- `xticks`: x 轴的刻度位置。
- `yticks`: y 轴的刻度位置。
- `xlim`: x 轴的范围，格式为 `(min, max)`。
- `ylim`: y 轴的范围，格式为 `(min, max)`。
- `color`: 绘制颜色，可以是单个颜色或颜色列表。
- `label`: 图例标签。

```
import pandas as pd
import matplotlib.pyplot as plt

s = pd.Series([1, 3, 2, 4, 5], index=['a', 'b', 'c', 'd', 'e'])
s.plot(kind='line', title='Line Plot', grid=True, figsize=(8, 4), style='r--',
linewidth=2)
plt.show()
```

3.7.2 hist

用于绘制Series数据直方图的方法。这个方法提供了多种参数来定制直方图的外观和样式。

```
Series.hist(by=None, ax=None, grid=True, xlabelsize=None, xrot=None,
ylabelsize=None, yrot=None, figsize=None, bins=10, backend=None, legend=False,
**kwargs)
```

- `by`: 如果不是None，则将数据分组并分别绘制每个组的直方图。
- `ax`: matplotlib的Axes对象，如果指定了，则直方图将绘制在该Axes上。
- `grid`: 布尔值，默认为True，表示是否在直方图上显示网格线。
- `xlabelsize`: int或str，用于设置x轴标签的字体大小。
- `xrot`: int或float，用于设置x轴标签的旋转角度。
- `ylabelsize`: int或str，用于设置y轴标签的字体大小。
- `yrot`: int或float，用于设置y轴标签的旋转角度。
- `figsize`: 元组，用于设置直方图的大小，格式为 (width, height)。
- `bins`: int或序列，用于设置直方图的柱子数量或具体的边界。
- `backend`: 用于指定绘图后端，通常Pandas会使用matplotlib。
- `legend`: 布尔值，默认为False，表示是否在直方图上显示图例。
- `**kwargs`: 其他关键字参数，将被传递给matplotlib的 `hist` 函数。

```
import pandas as pd
import numpy as np
from matplotlib import pyplot as plt

# 创建一个包含随机数据的Series
```

```
s = pd.Series(np.random.randn(1000))

# 绘制直方图
s.hist(bins=30, figsize=(8, 6), color='blue', edgecolor='black')

# 设置标题和轴标签
plt.title('Histogram of a Random Sample')
plt.xlabel('Value')
plt.ylabel('Frequency')

# 显示图表
plt.show()
```

3.8 其他常用方法

3.8.1 unique

该函数用于返回 Series 中的唯一值。这个方法返回一个数组，其中包含了 Series 中所有唯一的值。数组中的值是按照它们在原始 Series 中首次出现的顺序排列的。

```
Series.unique()
```

```
import pandas as pd

# 创建一个 Series
s = pd.Series(['apple', 'banana', 'apple', 'orange', 'banana', 'banana'])

# 获取唯一值
unique_values = s.unique()

# 输出唯一值
print(unique_values)
```


3.8.2 nunique

该函数用于计算 Series 中唯一值的数量。这个方法返回一个整数，表示 Series 中唯一值的数量。

```
Series.nunique(dropna=True)
```

- **dropna**: 布尔值，默认为 `True`。如果为 `True`，则在计算唯一值数量之前，会先从 Series 中排除 NaN 值。

```
import pandas as pd

# 创建一个 Series
s = pd.Series(['apple', 'banana', 'apple', 'orange', 'banana', 'banana'])

# 计算唯一值的数量（排除 NaN）
unique_count = s.nunique(dropna=True)

# 输出唯一值的数量
print(unique_count)
```

3.8.3 value_counts

该方法用于计算 Series 中每个值的出现次数。这个方法返回一个包含每个唯一值及其对应出现次数的 Series。

```
Series.value_counts(normalize=False, sort=True, ascending=False, bins=None, dropna=True)
```

- **normalize**: 布尔值或 'all'，默认为 `False`。如果为 `True`，返回每个值的相对频率；如果为 'all'，则返回所有值的相对频率之和为 1。
- **sort**: 布尔值，默认为 `True`。如果为 `True`，结果将按计数值降序排序。
- **ascending**: 布尔值，默认为 `False`。如果为 `True`，结果将按计数值升序排序。
- **bins**: 用于离散化连续数据，可以是整数或分位数数组。如果指定了 `bins`，则结果将是每个 bin 的计数。
- **dropna**: 布尔值，默认为 `True`。如果为 `True`，则排除 NaN 值。

```
import pandas as pd

# 创建一个 Series
s = pd.Series(['apple', 'banana', 'apple', 'orange', 'banana', 'banana'])

# 计算每个值的出现次数
value_counts = s.value_counts()

# 输出每个值的出现次数
print(value_counts)
```

3.8.4 describe

该方法用于生成描述性统计信息。这个方法返回一个包含计数、均值、标准差、最小值、25% 分位数、中位数、75% 分位数和最大值的 Series。

```
Series.describe(percentiles=None, include=None, exclude=None)
```

- **percentiles**: 数值列表或数值元组，默认为 `[.25, .5, .75]`，表示要包含在输出中的分位数。
- **include**: 字符串或类型列表，用于指定要包括在结果中的数据类型。默认为 `None`，即包括所有数字类型。
- **exclude**: 字符串或类型列表，用于指定要从结果中排除的数据类型。默认为 `None`。

```
import pandas as pd

# 创建一个 Series
s = pd.Series([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

# 生成描述性统计信息
description = s.describe()

# 输出描述性统计信息
print(description)
```

3.8.5 copy

该函数用于创建 Series 对象的一个副本。

```
Series.copy(deep=True)
```

- **deep**: 布尔值，默认为 `True`。当 `deep=True` 时，会进行深拷贝，即复制数据和索引的副本，而不是仅仅复制引用。当 `deep=False` 时，会进行浅拷贝，即只复制数据或索引的引用。

```
import pandas as pd

# 创建一个 Series
original_series = pd.Series([1, 2, 3, 4, 5])

# 创建一个深拷贝
deep_copied_series = original_series.copy()

# 创建一个浅拷贝
shallow_copied_series = original_series.copy(deep=False)

# 修改深拷贝中的数据
deep_copied_series[0] = 999

# 输出原始 Series 和深拷贝后的 Series
print("Original Series:\n", original_series)
print("Deep Copied Series:\n", deep_copied_series)

# 修改浅拷贝中的数据
shallow_copied_series[1] = 888

# 输出原始 Series 和浅拷贝后的 Series
print("Original Series after shallow copy modification:\n", original_series)
print("Shallow Copied Series:\n", shallow_copied_series)
```

3.8.6 reset_index

该函数用于重置 Series 的索引，将原来的索引转换为一个列，并将一个新的默认整数索引赋给 Series。

```
Series.reset_index(level=None, *, drop=False, name=no_default, inplace=False,
allow_duplicates=False)
```

- **level**: int 或 level 名，可选。如果 Series 是多级索引（MultiIndex），则只移除指定的级别。默认为 None，移除所有级别。
- **drop**: 布尔值，默认为 False。如果为 True，则不将旧索引添加为新列，直接丢弃。
- **name**: 字符串，可选。用于新列的名称，默认为 no_default。如果未指定，并且索引有名字，则使用索引的名字。
- **inplace**: 布尔值，默认为 False。如果为 True，则在原地修改 Series，不返回新的对象。
- **allow_duplicates**: 布尔值，默认为 False。如果为 True，则允许在重置索引后出现重复的索引值。默认情况下，如果出现重复的索引值，Pandas 会抛出错误，对Series无用。

```
import pandas as pd
import numpy as np

# 创建一个简单的Series对象
```

```
data = pd.Series(np.random.randint(1, 100, 5), index=['c', 'a', 'e', 'b', 'd'])

# 对Series进行排序
sorted_series = data.sort_values()

# 重置索引
reset_indexed_series = sorted_series.reset_index(drop=True)
print("排序后的Series：")
print(sorted_series)
print("重置索引后的Series：")
print(reset_indexed_series)
```

3.8.7 info

用于显示Series的概要信息的方法。这个方法提供了关于Series的元数据，包括数据类型、非空值的数量、内存使用情况等。

```
Series.info(verbose=None, buf=None, max_cols=None, memory_usage=None,
show_counts=True)
```

- `verbose`：布尔值或None，默认为None。如果为True，则输出更详细的信息。
- `buf`：一个文件-like对象，如果提供，则将输出写入这个对象而不是标准输出。
- `max_cols`：int，用于显示的最大列数。如果列数超过这个值，则显示“...”。
- `memory_usage`：布尔值或str，默认为None。如果为True，则显示内存使用情况。如果设置为'deep'，则会计算列的内存使用情况，这可能非常慢。
- `show_counts`：布尔值，默认为True。如果为True，则显示非空值的数量。

```
import numpy as np
import pandas as pd

# 创建一个示例Series
s = pd.Series([1, 2, np.nan, 4, 5], name='example_series', index=['a', 'b', 'c',
'd', 'e'])

# 显示Series的概要信息
s.info()
```

3.8.8 apply

对 `Series` 中的每个元素应用一个函数，并返回一个结果 `Series`。

```
Series.apply(func, convert_dtype=True, args=(), **kwargs)
```

- `func`：一个函数，它将被应用到 `Series` 的每一个元素上。这个函数可以是 Python 的内置函数，也可以是用户自定义的函数。
- `convert_dtype`：布尔值，默认为 `True`。如果为 `True`，则在可能的情况下，Pandas 会尝试将结果转换为适合的数据类型。
- `args`：一个元组，包含传递给 `func` 的位置参数。
- `**kwargs`：一个字典，包含传递给 `func` 的关键字参数。

```
import pandas as pd

# 创建一个Series
series = pd.Series([1, 2, 3, 4, 5])

# 使用 apply 方法结合 lambda 函数，对 series 中的每个元素执行平方操作
res = series.apply(lambda x: x ** 2)

# 打印结果，输出每个元素的平方值
print(res)
```

3.8.9 map

对 `Series` 中的每个元素应用一个映射，它允许你将一个函数应用到 `Series` 的每个元素上，或者将一个字典或 `Series` 映射到 `Series` 的值上。

```
Series.map(arg, na_action=None)
```

- `arg`：这可以是以下几种类型：
  - 函数：将此函数应用于 `Series` 的每个元素。
  - 字典：将 `Series` 中的值映射到字典的键上，返回对应的值。
  - Series：使用另一个 `Series` 的索引来映射当前 `Series` 的值。
- **na_action**：默认为 None。如果设置为 'ignore'，并且 `arg` 是一个函数，那么将忽略 `NaN` 值，并保留它们不变。

```
import pandas as pd

# 创建一个 Series
s = pd.Series([1, 2, 3, 4, 5])

# 定义一个函数，用于将值翻倍
def double(x):
    return x * 2

# 使用 map 方法应用这个函数
s_doubled = s.map(double)
print(s_doubled)
```

```
import pandas as pd

# 创建一个映射字典
grade_mapping = {
    90: 'A',
    80: 'B',
    70: 'C',
    60: 'D',
    0: 'F'
}

# 创建一个成绩的 Series
grades = pd.Series([80, 92, 77, 59, 100])

# 使用 map 方法应用这个字典
```

```
grades_mapped = grades.map(grade_mapping)
print(grades_mapped)
```

```
import pandas as pd

# 创建一个名为series1的Series对象
series1 = pd.Series([50, 60, 70, 80, 90], index=['a', 'b', 'c', 'd', 'e'])

# 创建一个名为grades的Series对象，代表成绩数据
grades = pd.Series([80, 92, 77, 59, 100], index=[0, 1, 2, 3, 4])

# 使用grades的map方法，将grades中的每个值作为键，去series1中查找对应的键值并返回
res = grades.map(series1)

# 打印输出新生成的Series对象res
print(res)
```

4. 二维数组DataFrame

二维数组则是可以理解为excel表格：

4.1 DataFrame的创建

在Pandas中，使用DataFrame来创建二维数组DataFrame：

```
class pandas.DataFrame(data=None, index=None, columns=None, dtype=None,
copy=None)
```

- `data`：数据源，可以是以下几种形式：
  - 列表，其中每个元素是一行数据。
  - 字典，其中键是列名，值是列值（列表或数组）。
  - 2d-Ndarray。
  - `Series`对象，每个`Series`成为一列。
- `index`：行标签，如果没有指定，默认是整数索引`[0, ..., n-1]`，其中`n`是数据中的行数。

- `columns`：列标签，如果没有指定，则列标签从数据源中推断。
- `dtype`：指定某列的数据类型。如果指定，则所有列都将转换为指定的数据类型。
- `copy`：布尔值，默认为`None`。如果为`True`，则复制数据；如果为`False`，则尽可能避免复制数据。

4.1.1 使用列表创建

```
import pandas as pd

# 创建一个包含学生信息的列表，每个元素代表一个学生的姓名
data_list = ['小明', '小红', '小刚']

# 定义列名
columns = ['姓名']

# 使用pandas库创建一个DataFrame，将数据列表和列名作为参数传入
df = pd.DataFrame(data_list, columns=columns)

# 打印DataFrame以查看数据
print(df)
```

```
import pandas as pd

# 创建一个包含学生信息的嵌套列表，每个子列表代表一个学生的姓名、年龄和成绩
data_list = [
    ['小明', 20, 85],
    ['小红', 18, 90],
    ['小刚', 22, 88]
]

# 定义列名，分别对应姓名、年龄和成绩
columns = ['姓名', '年龄', '成绩']

# 使用pandas库创建一个DataFrame，将数据列表和列名作为参数传入
df = pd.DataFrame(data_list, columns=columns)

# 打印DataFrame以查看数据
print(df)
```


4.1.2 使用字典创建

可以使用一个字典来创建DataFrame，其中字典的键将作为列名，字典的值可以是列表、数组等可迭代对象，它们的长度要一致，代表每一列的数据。

```
import pandas as pd

# 定义一个字典，其中包含两组数据：姓名和年龄
data = {
    'Name': ['Tom', 'Nick', 'John'],  # 'Name' 键对应一个包含姓名的列
    'Age': [20, 21, 19]               # 'Age' 键对应一个包含年龄的列
}

# 使用pd.DataFrame()函数将字典转换为DataFrame对象
# 这里，data字典中的键自动成为DataFrame的列名，值成为列的数据
```

```
df = pd.DataFrame(data)

# 打印DataFrame对象，查看其内容
print(df)
```

4.1.3 使用Ndarray数组创建

```
import pandas as pd
import numpy as np

# 定义一个二维Ndarray数组，其中包含两组数据：姓名和年龄
data_array = np.array([
    ['Tom', 20],
    ['Nick', 21],
    ['John', 19]
])

# 使用pd.DataFrame()函数将二维数组转换为DataFrame对象
df = pd.DataFrame(data_array, columns=['Name', 'Age'])

# 打印
print(df)
```

4.1.4 使用Series创建

如果有多个Series对象，也可以将它们组合成一个DataFrame。

```
import pandas as pd

# 创建三个pandas Series对象
s1 = pd.Series(['小明', '小红', '小刚'], name='姓名')
s2 = pd.Series([20, 18, 22], name='年龄')
s3 = pd.Series([85, 90, 88], name='成绩')

# 将Series对象组合成一个字典，键是Series的名称，值是Series本身
# 然后将这个字典传递给DataFrame构造函数来创建一个DataFrame
df = pd.DataFrame({s1.name: s1, s2.name: s2, s3.name: s3})

# 打印DataFrame对象，查看其内容
print(df)
```

```
import pandas as pd

# 创建三个pandas Series对象
s1 = pd.Series(['小明', '小红', '小刚'], name='姓名')
s2 = pd.Series([20, 18, 22, 0], name='年龄')
s3 = pd.Series([85, 90, 88], name='成绩')

# 使用concat拼接，并指定轴为1
df = pd.concat([s1, s2, s3], axis=1)

# 打印DataFrame对象，查看其内容
print(df)
```

4.2 DataFrame的属性

4.2.1 index

返回DataFrame的行索引。

```
import pandas as pd

data = {
    '姓名': ['小明', '小红', '小刚'],
    '年龄': [20, 18, 22],
    '成绩': [85, 90, 88]
}

df = pd.DataFrame(data, index=['a', 'b', 'c'])
print(df)

print(df.index)
```

4.2.2 columns

返回DataFrame的列名。

```
import pandas as pd

data = {
    '姓名': ['小明', '小红', '小刚'],
    '年龄': [20, 18, 22],
    '成绩': [85, 90, 88]
}

df = pd.DataFrame(data, index=['a', 'b', 'c'])
print(df)

print(df.columns)
```


4.2.3 values

返回DataFrame中数据的Ndarray表示

```
import pandas as pd

data = {
    '姓名': ['小明', '小红', '小刚'],
    '年龄': [20, 18, 22],
    '成绩': [85, 90, 88]
}

df = pd.DataFrame(data, index=['a', 'b', 'c'])
print(df)

print(df.values)
```

4.2.4 dtypes

返回每列的数据类型。

```
import pandas as pd

data = {
    '姓名': ['小明', '小红', '小刚'],
    '年龄': [20, 18, 22],
    '成绩': [85, 90, 88]
}

df = pd.DataFrame(data, index=['a', 'b', 'c'])
print(df)

print(df.dtypes)
```

4.2.5 shape

返回DataFrame的形状（行数，列数）。

```
import pandas as pd

data = {
    '姓名': ['小明', '小红', '小刚'],
    '年龄': [20, 18, 22],
    '成绩': [85, 90, 88]
}

df = pd.DataFrame(data, index=['a', 'b', 'c'])
print(df)

print(df.shape)
```


4.2.6 size

返回DataFrame中的元素数量。

```
import pandas as pd

data = {
    '姓名': ['小明', '小红', '小刚'],
    '年龄': [20, 18, 22],
    '成绩': [85, 90, 88]
}

df = pd.DataFrame(data, index=['a', 'b', 'c'])
print(df)

print(df.size)
```

4.2.7 empty

返回DataFrame是否为空。

```
import pandas as pd

data = {
    '姓名': ['小明', '小红', '小刚'],
    '年龄': [20, 18, 22],
    '成绩': [85, 90, 88]
}

df = pd.DataFrame(data, index=['a', 'b', 'c'])
print(df)

print(df.empty)
```

4.2.8 T

返回DataFrame的转置。

```
import pandas as pd

data = {
    '姓名': ['小明', '小红', '小刚'],
    '年龄': [20, 18, 22],
    '成绩': [85, 90, 88]
}

df = pd.DataFrame(data, index=['a', 'b', 'c'])
print(df)

res = df.T
print(res)
```


4.2.9 axes

返回行轴和列轴的列表。

```
import pandas as pd

data = {
    '姓名': ['小明', '小红', '小刚'],
    '年龄': [20, 18, 22],
    '成绩': [85, 90, 88]
}

df = pd.DataFrame(data, index=['a', 'b', 'c'])
print(df)

print(df.axes)
```

4.2.10 ndim

返回DataFrame的维度数。对于标准的二维DataFrame，这个值通常是2。

```
import pandas as pd

data = {
    '姓名': ['小明', '小红', '小刚'],
    '年龄': [20, 18, 22],
    '成绩': [85, 90, 88]
}

df = pd.DataFrame(data, index=['a', 'b', 'c'])
print(df)

print(df.ndim)
```

4.2.11 attrs

允许用户存储DataFrame的元数据，它是一个字典，可以用来存储任意与DataFrame相关的额外信息。

```
import pandas as pd

data = {
    '姓名': ['小明', '小红', '小刚'],
    '年龄': [20, 18, 22],
    '成绩': [85, 90, 88]
}

df = pd.DataFrame(data, index=['a', 'b', 'c'])
print(df)

df.attrs['creator'] = 'xxxxx'
df.attrs['created_at'] = '2024-13-32'
print(df.attrs)
```


4.3 DataFrame中元素的索引与访问

4.3.1 使用列名访问

对于DataFrame来说，可以直接使用列名来访问某一列的数据，返回的是一个Series对象。

```
import pandas as pd

data = {
    '姓名': ['小明', '小红', '小刚'],
    '年龄': [20, 18, 22],
    '成绩': [85, 90, 88]
}

df = pd.DataFrame(data)
print(df)
print(df['姓名'])
print(df[['姓名', '年龄']])
```

4.3.2 loc和iloc

可以使用`loc`与`iloc`属性访问单个或多个数据，其语法为：

```
df.loc[row_label, column_label]
```

```
import pandas as pd

data = {
    '姓名': ['小明', '小红', '小刚'],
    '年龄': [20, 18, 22],
    '成绩': [85, 90, 88]
}

df = pd.DataFrame(data)
print(df)
print(df.loc[0, '姓名'])
print(df.loc[0:2, '姓名':'成绩'])
print(df.iloc[0, 0])
print(df.iloc[0:1, 0:1])
```

4.3.3 at和iat

使用`at`和`iat`属性来访问单个数据。

```
import pandas as pd

data = {
    '姓名': ['小明', '小红', '小刚'],
    '年龄': [20, 18, 22],
    '成绩': [85, 90, 88]
}

df = pd.DataFrame(data)
print(df)
print(df.at[0, '姓名'])
print(df.iat[0, 0])
```


4.3.4 head

该方法用于获取DataFrame的前 `n` 行。默认情况下，如果不指定 `n` 的值，它会返回前5行。

```
DataFrame.head(n=5)
```

- `n`: 整数，表示要返回的行数。默认值为5。

```
import pandas as pd

# 创建一个示例DataFrame
data = {
    'A': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
    'B': [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
}
df = pd.DataFrame(data)

# 使用head方法获取前5行
print(df.head())
```

4.3.5 tail

该方法用于获取DataFrame的最后 `n` 行。与 `DataFrame.head` 方法类似，如果不指定 `n` 的值，它会默认返回最后5行。

```
DataFrame.tail(n=5)
```

- `n`: 整数，表示要返回的行数。默认值为5。

```
import pandas as pd

# 创建一个示例DataFrame
data = {
    'A': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
    'B': [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
}
df = pd.DataFrame(data)

# 使用tail方法获取最后5行
print(df.tail())
```

4.3.6 isin

`DataFrame.isin(values)` 方法用于检查DataFrame中的元素是否包含在指定的集合 `values` 中。这个方法会返回一个布尔型DataFrame，其中的每个元素都表示原始DataFrame中对应元素是否在 `values` 中。

```
DataFrame.isin(values)
```

- `values`: 可以是单个值、列表、元组、集合或另一个DataFrame/Series。如果 `values` 是一个字典，键是列名，值是列表或元组，表示该列中要检查的值。

```
import pandas as pd

# 创建一个示例DataFrame
data = {
    'A': [1, 2, 3, 4, 5],
    'B': ['a', 'b', 'c', 'd', 'e']
}
df = pd.DataFrame(data)

# 检查DataFrame中的元素是否包含在指定的值集合中
values_to_check = [2, 4, 'c']
print(df.isin(values_to_check))
```

4.3.7 get

用于从DataFrame中获取列，它类似于直接使用 `df[key]` 来访问列，但是当列不存在时，`get` 方法提供了一个更安全的方式来处理这种情况，你可以指定一个默认值，而不是引发一个错误。

```
DataFrame.get(key, default=None)
```

- `key`：你想获取的列的名称。
- `default`：如果列不存在时返回的默认值。默认是 `None`。

```
import pandas as pd

data = {
    '姓名': ['小明', '小红', '小刚'],
    '年龄': [20, 18, 22],
    '成绩': [85, 90, 88]
}

df = pd.DataFrame(data)

# 获取'成绩'列
scores = df.get('成绩')

# 尝试获取不存在的列，返回指定值
non_existent_column = df.get('体重', default='Not Found')

print(scores)
print(non_existent_column)
```

4.4 数据操作

4.4.1 数据清洗

1. isnull()：用于检测 DataFrame 中的缺失值，它会返回一个相同形状的布尔型 DataFrame，其中每个元素表示原始 DataFrame 中相应位置的元素是否是缺失值。

```
import pandas as pd
import numpy as np

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

```
    'A': [1, 2, np.nan],
    'B': [4, np.nan, 6],
    'C': [7, 8, 9]
})

# 打印原始DataFrame
print(df)

# 使用 isnull() 方法检测缺失值
missing_values = df.isnull()

print(missing_values)
```

2. dropna()：用于删除 DataFrame 中的缺失值。

```
DataFrame.dropna(axis=0, how=any, thresh=_NoDefault.no_default, subset=None, inplace=False, ignore_index=False)
```

- `axis`：{0 或 'index'，1 或 'columns'}，默认为 0。0表示按行删除，1表示按列删除
- `how`：{'any'，'all'}，默认为 'any'。确定删除缺失值的逻辑：
  - 'any'：如果行或列中的任意一个值是 NaN，就删除该行或列。
  - 'all'：如果行或列中的所有值都是 NaN，才删除该行或列。
- `thresh`：指定每行或每列至少需要有多少个非缺失值才能保留。如果设置此参数，`how` 参数将被忽略。
- `subset`：指定在哪些列中搜索缺失值。如果未指定，则在所有列中搜索。
- `inplace`：是否修改 DataFrame 而不是创建新的 DataFrame。
- `ignore_index`：布尔值，默认为 False。如果为 True，则不保留原始 DataFrame 的索引。

```
import pandas as pd
import numpy as np

# 创建一个包含缺失值的 DataFrame
df = pd.DataFrame({
    'A': [1, 2, np.nan],
    'B': [4, np.nan, 6],
    'C': [7, 8, 9]
})

# 打印原始DataFrame
print(df)

# 删除任何含有 NaN 值的行
df_cleaned = df.dropna()

print(df_cleaned)
```

3. fillna()：用于填充 DataFrame 中的缺失值。

```
DataFrame.fillna(value=None, *, method=None, axis=0, inplace=False, limit=None)
```

- `value`：填充值，可以是单个值，也可以是字典（对不同的列填充不同的值），或者一个 Series。
- `method`：{'bfill'，'ffill'}，默认为无默认值。指定填充方法：

    - 'bfill' 或 'backfill'：使用下一个有效观测值填充。
    - 'ffill' 或 'pad'：使用前一个有效观测值填充。
- `axis`：{0 或 'index', 1 或 'columns'}，默认为0。
- `inplace`：布尔值，默认为 False。如果为 True，则在原地修改 DataFrame 而不返回新的 DataFrame。
- `limit`：int，默认为无默认值。如果指定了 `method`，则该参数限制连续填充的数量。

```
import pandas as pd
import numpy as np

# 创建一个包含缺失值的 DataFrame
df = pd.DataFrame({
    'A': [1, 2, np.nan],
    'B': [np.nan, np.nan, 6],
    'C': [7, np.nan, 9]
})

# 打印原始DataFrame
print(df)

# 使用固定值填充缺失值
df_filled_value = df.fillna(value=0)
print(df_filled_value)

# 使用字典填充
data = {
    'A': 'a',
    'B': 'b',
    'C': 'c'
}
df_filled_dict = df.fillna(value=data)
print(df_filled_dict)

# 使用Series填充
data_series = pd.Series(['a', 'b', 'c'], ['A', 'B', 'C'])
df_filled_series = df.fillna(value=data_series)
print(df_filled_series)

# 使用前一个有效观测值填充缺失值
df_filled_ffill = df.fillna(method='ffill')
print(df_filled_ffill)

# 使用后一个有效观测值填充缺失值
df_filled_bfill = df.fillna(method='bfill')
print(df_filled_bfill)
```


4. drop_duplicates()：用于删除 DataFrame 中的重复行。

```
DataFrame.drop_duplicates(subset=None, keep='first', inplace=False,
ignore_index=False)
```

- `subset`：指定要检查重复的列名或列名列表，默认值为 `None`，表示检查所有列。
- `keep`：{'first', 'last', False}，默认为 'first'。确定如何处理重复项：
    - 'first'：保留第一次出现的重复项。

    - 'last'：保留最后一次出现的重复项。
    - False：删除所有重复项。
- `inplace`：是否修改 DataFrame 而不是创建新的 DataFrame。
- `ignore_index`：是否重置索引值。

```
import pandas as pd

# 创建一个包含重复行的 DataFrame
df = pd.DataFrame({
    'A': [1, 1, 2, 2, 3, 3],
    'B': [1, 1, 2, 2, 3, 3],
    'C': [1, 1, 2, 2, 3, 3]
})

# 打印原始DataFrame
print(df)

# 删除重复行，保留第一次出现的重复项
df_dedup_first = df.drop_duplicates()
print(df_dedup_first)

# 根据指定列删除重复行
df_dedup_column = df.drop_duplicates(subset=['A'])
print(df_dedup_column)

# 删除重复行，保留最后一次出现的重复项
df_dedup_last = df.drop_duplicates(keep='last')
print(df_dedup_last)

# 删除所有重复行
df_dedup_all = df.drop_duplicates(keep=False)
print(df_dedup_all)
```

4.4.2 数据转换

1. replace()：用于替换 DataFrame 中的值。

```
DataFrame.replace(to_replace=None, value=_NoDefault.no_default, inplace=False,
limit=None, regex=False, method=_NoDefault.no_default)
```

- `to_replace`：被替换的内容，可以是 scalar, list, dict, regex。如果是字典，则键是要替换的值，值是相应的替换值。
- `value`：替换后的值。可以是单个值、列表或数组，与 `to_replace` 长度相同。
- `inplace`：是否在原地修改 `DataFrame`。
- `limit`：限制替换的数量。可以是整数，表示最多替换多少个值。
- `regex`：是否使用正则表达式进行匹配。默认值为 `False`。
- `method`：替换方法。可以是：
    - `'pad'` 或 `'ffill'`：使用前面的数据向后填充。
    - `'backfill'` 或 `'bfill'`：使用后面的数据向前填充。

```
import pandas as pd
```

```
# 创建一个 DataFrame
df = pd.DataFrame({
    'A': [1, 2, 3, 4, 5],
    'B': ['a', 'b', 'a', 'b', 'a']
})

# 打印原始DataFrame
print(df)

# 用数字 100 替换所有的 1
df_replaced = df.replace(to_replace=1, value=100)

# 用字符串 'z' 替换所有的 'a'
df_replaced = df_replaced.replace(to_replace='a', value='z')

# 使用字典替换多个值
df_replaced = df.replace({
    2: 200,
    'b': 'y'
})

# 使用正则表达式替换
df_replaced = df.replace(to_replace=r'^a$', value='z', regex=True)

print(df_replaced)
```

2. pivot()：用于改变表格形状格式。

```
DataFrame.pivot(columns, index=typing.Literal[<no_default>],
values=typing.Literal[<no_default>])
```

- `columns`：作为新 `DataFrame` 的行索引的列名。可以是单个列名或列名列表。
- `index`：作为新 `DataFrame` 的列标签的列名。可以是单个列名或列名列表。
- `values`：作为新 `DataFrame` 的值的列名。可以是单个列名或列名列表。

```
import pandas as pd

# 创建一个DataFrame
df = pd.DataFrame({
    'foo': ['one', 'one', 'one', 'two', 'two', 'two'],
    'bar': ['A', 'B', 'C', 'A', 'B', 'C'],
    'baz': [1, 2, 3, 4, 5, 6],
    'zoo': ['x', 'y', 'z', 'q', 'w', 't']
})

# 打印原始DataFrame
print(df)

# 使用pivot方法对DataFrame进行重塑，其中foo作为行索引，bar作为列索引，baz作为值
res1 = df.pivot(index='foo', columns='bar', values='baz')

# 打印重塑后的DataFrame
print(res1)

# 使用pivot方法对DataFrame进行重塑，其中foo作为行索引，bar作为列索引，baz、zoo作为值
```

```
res2 = df.pivot(index='foo', columns='bar', values=['baz', 'zoo'])
# 打印重塑后的DataFrame
print(res2)
```

3. melt()：用于改变表格形状格式。

```
DataFrame.melt(id_vars=None, value_vars=None, var_name=None, value_name='value',
col_level=None, ignore_index=True)
```

- `id_vars`：保持不变的列名或列名列表。
- `value_vars`：字符串或字符串列表，可选。要重塑的列名或列名列表。这些列的值将被展平到新的行中。
- `var_name`：新的列名，用于存储原来列的名称。默认值为 `None`，表示使用默认名称。
- `value_name`：字符串。新的列名，用于存储原来列的值。默认值为 `'value'`。
- `col_level`：整数或列标签，可选。如果 `DataFrame` 的列是多级索引，指定要使用的级别。默认值为 `None`，表示使用所有级别。
- `ignore_index`：是否忽略原来的索引，重新生成一个新的默认整数索引。默认值为 `True`。

```
import pandas as pd

# 创建一个DataFrame
df = pd.DataFrame({
    'A': {0: 'a', 1: 'b', 2: 'c'},
    'B': {0: 1, 1: 3, 2: 5},
    'C': {0: 2, 1: 4, 2: 6}
})

# 打印原始DataFrame
print(df)

# 使用melt方法对DataFrame进行重塑
res1 = df.melt(id_vars=['A'], value_vars=['B'], var_name='myVarname',
value_name='myValnames')

# 打印重塑后的DataFrame
print(res1)
```


4. pivot_table()：用于生成一个指定格式的数据透视表。

```
DataFrame.pivot_table(values=None, index=None, columns=None, aggfunc='mean',
fill_value=None, margins=False, dropna=True, margins_name='All', observed=False,
sort=True)
```

- `values`：要聚合的列名或列名列表。如果未指定，则使用所有数值列。
- `index`：作为新 `DataFrame` 的行索引的列名或列名列表。
- `columns`：作为新 `DataFrame` 的列标签的列名或列名列表。
- `aggfunc`：聚合函数，可以是：
  - 单个函数（如 `'mean'`、`'sum'`、`'count'` 等）。
  - 函数列表（如 `['mean', 'sum']`）。
  - 字典，键是列名，值是聚合函数。
- `fill_value`：用于填充缺失值的值。默认值为 `None`。

- `margins`：是否添加总计行和总计列。默认值为 `False`。
- `dropna`：是否从结果中删除包含缺失值的行。默认值为 `True`。
- `margins_name`：总计行和总计列的名称。默认值为 `'All'`。
- `observed`：是否仅显示已观察到的类别。默认值为 `False`。
- `sort`：是否对结果进行排序。默认值为 `True`。

```
import numpy as np
import pandas as pd

# 创建一个DataFrame
df = pd.DataFrame({
    "A": ["foo", "foo", "foo", "foo", "foo",
          "bar", "bar", "bar", "bar"],
    "B": ["one", "one", "one", "two", "two",
          "one", "one", "two", "two"],
    "C": ["small", "large", "large", "small",
          "small", "large", "small", "small",
          "large"],
    "D": [1, 2, 2, 3, 3, 4, 5, 6, 7],
    "E": [2, 4, 5, 5, 6, 6, 8, 9, 9]
})

# 打印原始DataFrame
print(df)

# 使用pivot_table方法创建一个数据透视表
table = df.pivot_table(values='D', index=['A', 'B'], columns=['C'],
aggfunc=np.sum)

# 打印数据透视表
print(table)
```

5. astype()：用于转换 DataFrame 中指定列的数据类型。

```
DataFrame.astype(dtype, copy=None, errors='raise')
```

- `dtype`：新的数据类型，可以是字典或数据类型。如果是字典，则键是列名，值是要转换为的数据类型。如果指定为单一数据类型，则所有列都将转换为该类型。
- `copy`：布尔值，默认为 `None`。如果为 `True`，则在转换数据之前创建数据的副本。如果为 `False`，则尽可能地避免复制，但这可能会影响到输入数据的原始 DataFrame。如果为 `None`（默认值），则仅在需要时复制数据。
- `errors`：{'raise', 'ignore'}，默认为 'raise'。控制当转换失败时的行为。如果为 'raise'，则在无法转换数据时抛出异常；如果为 'ignore'，则在无法转换数据时保持原始数据类型不变。

```
import pandas as pd

# 创建一个示例 DataFrame
df = pd.DataFrame({
    'A': [1, 2, 3],
    'B': [4.5, 5.5, 6.5],
    'C': ['7', '8', '9']
})
```

```
# 打印原始DataFrame
print(df)

# 将列 'A' 转换为浮点数类型
df['A'] = df['A'].astype(float)

# 使用字典将多列转换为不同的数据类型
# 将列 'B' 转换为整数类型，列 'C' 也转换为整数类型
df = df.astype({
    'B': int,
    'C': int
})

# 打印转换后的DataFrame
print(df)

# 打印DataFrame中各列的数据类型
print(df.dtypes)
```

4.4.3 数据排序

1. sort_values()：用于根据一个或多个列的值对 DataFrame 进行排序。

```
DataFrame.sort_values(by, axis=0, ascending=True, inplace=False,
kind='quicksort', na_position='last', ignore_index=False, key=None)
```

- `by`：用于排序的列名或列名列表。
- `axis`：{0 or 'index', 1 or 'columns'}，默认为 0。沿着哪个轴进行排序。
- `ascending`：排序的方向，True表示升序，False表示降序，默认为True。
- `inplace`：是否在原地修改 `DataFrame`。
- `kind`：{'quicksort', 'mergesort', 'heapsort', 'stable'}，默认为 'quicksort'。排序算法。
- `na_position`：{'first', 'last'}，默认为 'last'。缺失值的放置位置。
- `ignore_index`：布尔值，默认为 `False`。是否忽略原来的索引，重新生成一个新的默认整数索引。
- `key`：函数，默认为 `None`。应用于 `by` 中每个列的函数，排序将基于函数的返回值。

```
import numpy as np
import pandas as pd

# 创建一个示例 DataFrame
df = pd.DataFrame({
    'col1': ['A', 'A', 'B', np.nan, 'D', 'C'],
    'col2': [2, 1, 9, 8, 7, 4],
    'col3': [0, 1, 9, 4, 2, 3],
    'col4': ['a', 'B', 'c', 'D', 'e', 'F']
})

# 打印原始DataFrame
print(df)

# 根据 'col1' 列对DataFrame进行排序
res1 = df.sort_values(by=['col1'])
# 打印排序后的DataFrame
print(res1)
```

```
# 根据 'col1' 和 'col2' 列对DataFrame进行排序
res2 = df.sort_values(by=['col1', 'col2'])
# 打印排序后的DataFrame
print(res2)
```

2. sort_index()：用于根据索引对 DataFrame 进行排序。

```
DataFrame.sort_index(axis=0, level=None, ascending=True, inplace=False,
kind='quicksort', na_position='last', sort_remaining=True, ignore_index=False,
key=None)
```

- `axis`：{0 or 'index', 1 or 'columns'}，默认为 0。表示沿着哪个轴进行排序。0按照行标签排序，1按照列标签排序。
- `level`：如果索引是多级索引，指定要排序的级别。可以是整数或整数列表。
- `ascending`：默认为 `True`。表示排序是升序还是降序。
- `inplace`：是否在原地修改 `DataFrame`。
- `kind`：{'quicksort', 'mergesort', 'heapsort', 'stable'}，默认为 'quicksort'。排序算法。
- `na_position`：{'first', 'last'}，默认为 'last'。缺失值的放置位置。
- `sort_remaining`：是否对剩余的级别进行排序。仅在多级索引时有效。默认值为 `True`。
- `ignore_index`：是否忽略原来的索引，重新生成一个新的默认整数索引。默认值为 `False`。
- `key`：函数，默认为 `None`。应用于索引的函数，排序将基于函数的返回值。

```
import pandas as pd
import numpy as np

# 创建一个多级索引的DataFrame
arrays = [np.array(['qux', 'qux', 'foo', 'foo']),
          np.array(['two', 'one', 'two', 'one'])]
df = pd.DataFrame({'A': [1, 2, 3, 4], 'B': [4, 3, 2, 1]}, index=arrays)

print(df)
# 按第一层索引升序排序
df_sorted_by_first_level = df.sort_index(level=0)
print(df_sorted_by_first_level)

# 按第二层索引降序排序
df_sorted_by_second_level_desc = df.sort_index(level=1, ascending=False)
print(df_sorted_by_second_level_desc)

# 按整个索引升序排序
df_sorted_by_full_index = df.sort_index(ascending=True)
print(df_sorted_by_full_index)
```

4.4.4 数据筛选

可以使用布尔数组进行索引，选择满足条件的数据。

```
import pandas as pd

data = {
    '姓名': ['小明', '小红', '小刚'],
    '年龄': [20, 18, 22],
    '成绩': [85, 90, 88]
}
```

```
df = pd.DataFrame(data)

# 使用布尔索引选择成绩大于或等于90的学生
high_scores = df[df['成绩'] >= 90]

print(high_scores)
```

4.4.5 数据拼接

1. concat()：用于沿一个轴将多个 pandas 对象连接在一起。

```
pandas.concat(objs, axis=0, join='outer', ignore_index=False, keys=None,
levels=None, names=None, verify_integrity=False, sort=False, copy=True)
```

- `objs`：要连接的对象列表。
- `axis`：{0, 1, 'index', 'columns'}，默认为 0。
- `join`：连接方式。可以是：
  - `'outer'`：取所有索引的并集。
  - `'inner'`：取所有索引的交集。
- `ignore_index`：是否忽略原来的索引，重新生成一个新的默认整数索引。默认值为 `False`。
- `keys`：用于生成多级索引的键列表。每个键对应一个对象。
- `levels`：用于多级索引的级别列表。通常与 `keys` 一起使用。
- `names`：用于多级索引的名称列表。通常与 `keys` 一起使用。
- `verify_integrity`：是否验证最终的 `DataFrame` 是否有重复的索引。默认值为 `False`。
- `sort`：是否对结果按照列名进行升序排序。默认值为 `False`。
- `copy`：是否复制数据。

```
import pandas as pd

# 创建两个 DataFrame
df1 = pd.DataFrame({'A': ['A0', 'A1', 'A2', 'A3'],
                    'B': ['B0', 'B1', 'B2', 'B3'],
                    'C': ['C0', 'C1', 'C2', 'C3'],
                    'D': ['D0', 'D1', 'D2', 'D3']},
                   index=[0, 1, 2, 3])

df2 = pd.DataFrame({'A': ['A4', 'A5', 'A6', 'A7'],
                    'B': ['B4', 'B5', 'B6', 'B7'],
                    'C': ['C4', 'C5', 'C6', 'C7'],
                    'F': ['F4', 'F5', 'F6', 'F7']},
                   index=[4, 5, 6, 7])

# 沿着竖直方向拼接两个DataFrame
result = pd.concat([df1, df2], axis=0, ignore_index=True)

print(result)
```

2. merge()：用于根据一个或多个键将两个 DataFrame 对象连接起来。

```
DataFrame.merge(right, how='inner', on=None, left_on=None, right_on=None,
left_index=False, right_index=False, sort=False, suffixes=('_x', '_y'),
copy=None, indicator=False, validate=None)
```

- `right`：另一个 DataFrame 对象。
- `how`：{'left', 'right', 'outer', 'inner', 'cross'}, 默认为 'inner'。确定连接的类型：
  - 'left': 使用左侧（调用 merge的 DataFrame）的索引进行左连接。
  - 'right': 使用右侧（参数 `right` 中的 DataFrame）的索引进行右连接。
  - 'outer': 使用两个DataFrame的并集连接。
  - 'inner': 使用两个DataFrame的交集连接。
- `on`：用于合并的列名。如果 `left_on` 和 `right_on` 都没有指定，则使用 `on`。
- `left_on`：左侧 `DataFrame` 中用于合并的列名。不与on同时使用。
- `right_on`：右侧 `DataFrame` 中用于合并的列名。不与on同时使用。
- `left_index`：是否使用左侧 `DataFrame` 的索引作为合并键。默认值为 `False`。不与on同时使用。
- `right_index`：是否使用右侧 `DataFrame` 的索引作为合并键。默认值为 `False`。不与on同时使用。
- `sort`：是否对结果进行排序。默认值为 `False`。
- `suffixes`：用于重命名重复列的后缀。默认值为 `('_x', '_y')`。
- `copy`：是否复制数据。默认值为 `None`，表示根据需要自动决定是否复制。
- `indicator`：是否添加一个指示器列，显示每行来自哪个 `DataFrame`。默认值为 `False`。
- `validate`：检查合并键。可以是：
  - `'one_to_one'`：检查合并键在两者中是否唯一。
  - `'one_to_many'`：检查合并键在左侧是否唯一。
  - `'many_to_one'`：检查合并键在右侧是否唯一。
  - `'many_to_many'`：不检查。

```
import pandas as pd

# 创建两个 DataFrame
df1 = pd.DataFrame({'key': ['A', 'B', 'C', 'D'],
                    'value': [1, 2, 3, 4]}, index=['a', 'b', 'c', 'd'])
df2 = pd.DataFrame({'key': ['B', 'D', 'D', 'E'],
                    'value': [5, 6, 7, 8]}, index=['a', 'c', 'e', 'f'])

# 使用内连接（inner join）合并两个 DataFrame
result = df1.merge(df2, on='key', how='inner', suffixes=('_left', '_right'))

print(result)
```

3. join()：用于将两个对象的列连接起来。

```
DataFrame.join(other, on=None, how='left', lsuffix='', rsuffix='', sort=False,
validate=None)
```

- `other`：另一个 DataFrame 对象。
- `on`：用于连接的列名。

- `how`: {'left', 'right', 'outer', 'inner'}, 默认为 'left'。确定连接的类型：
  - 'left': 使用左侧（调用 join 的 DataFrame）的索引进行左连接。
  - 'right': 使用右侧（参数 `other` 中的 DataFrame）的索引进行右连接。
  - 'outer': 使用两个 DataFrame 的索引的并集进行全外连接。
  - 'inner': 使用两个 DataFrame 的索引的交集进行内连接。
- `lsuffix`: 用于重命名重复列的左后缀。默认值为空字符串 `''`。
- `rsuffix`: 用于重命名重复列的右后缀。默认值为空字符串 `''`。
- `sort`: 是否对结果进行排序。默认值为 `False`。
- `validate`: 检查合并键。可以是：
  - `'one_to_one'`：检查合并键在两者中是否唯一。
  - `'one_to_many'`：检查合并键在左侧是否唯一。
  - `'many_to_one'`：检查合并键在右侧是否唯一。
  - `'many_to_many'`：不检查。

```
import pandas as pd

# 创建两个 DataFrame
df1 = pd.DataFrame({'key': ['A', 'B', 'C', 'D'],
                    'value': [1, 2, 3, 4]},
                   index=['k0', 'k1', 'k2', 'k3'])
df2 = pd.DataFrame({'value2': [5, 6, 7, 8]},
                   index=['k1', 'k2', 'k3', 'k4'])

# 使用左连接（left join）根据索引合并两个 DataFrame
result = df1.join(df2, how='left')

print(result)
```

4.5 统计

4.5.1 count

用于计算 DataFrame 中非 NaN 值的数量。

```
DataFrame.count(axis=0, numeric_only=False)
```

- `axis`: {0 或 'index', 1 或 'columns'}，决定统计的方向。
  - 如果 `axis=0` 或 `axis='index'`，则对每列进行计数，返回一个 Series，其索引为列名，值为每列非 NaN 值的数量。
  - 如果 `axis=1` 或 `axis='columns'`，则对每行进行计数，返回一个 Series，其索引为行索引，值为每行非 NaN 值的数量。
- `numeric_only`: 是否只计算数值列中的非 NaN 值的数量，忽略非数值列，默认为 False。

```
import pandas as pd
import numpy as np

# 创建一个包含 NaN 值的 DataFrame
df = pd.DataFrame({
    'A': [1, 2, np.nan, 4],
    'B': [5, np.nan, np.nan, 8],
    'C': ['foo', 'bar', 'baz', np.nan]
```

```
})

# 计算每列非 NaN 值的数量
count_per_column = df.count()
print("Count per column:")
print(count_per_column)

# 计算每行非 NaN 值的数量
count_per_row = df.count(axis='columns')
print("\nCount per row:")
print(count_per_row)

# 只计算数值列的非 NaN 值的数量
count_numeric_only = df.count(numeric_only=True)
print("\nCount numeric only:")
print(count_numeric_only)
```

4.5.2 sum

用于计算 DataFrame 中数值的总和。

```
DataFrame.sum(axis=0, skipna=True, numeric_only=False, min_count=0, **kwargs)
```

- `axis`: {0 或 'index', 1 或 'columns', None}, 默认为 0。这个参数决定了求和是在哪个轴上进行：
  - 如果 `axis=0` 或 `axis='index'`，则对每列进行求和，返回一个 Series，其索引为列名，值为每列的总和。
  - 如果 `axis=1` 或 `axis='columns'`，则对每行进行求和，返回一个 Series，其索引为行索引，值为每行的总和。
- `skipna`: 布尔值，默认为 True。如果为 True，则在计算总和时会忽略 NaN 值。
- `numeric_only`: 布尔值，默认为 False。如果为 True，则只对数值列进行求和，忽略非数值列。
- `min_count`: int，默认为 0。这个参数指定了在计算总和之前，至少需要非 NaN 值的最小数量。如果某个分组中的非 NaN 值的数量小于 `min_count`，则结果为 NaN。
- `**kwargs`: 其他关键字参数。这些参数通常用于兼容性或特殊用途，通常不需要。

```
import pandas as pd
import numpy as np

# 创建一个包含 NaN 值的 DataFrame
df = pd.DataFrame({
    'A': [1, 2, np.nan, 4],
    'B': [5, np.nan, np.nan, 8],
    'C': ['foo', 'bar', 'baz', 'qux']
})

# 计算每列的总和
sum_per_column = df.sum()
print("Sum per column:")
print(sum_per_column)

# 计算每行的总和
sum_per_row = df.sum(axis='columns')
print("\nSum per row:")
print(sum_per_row)
```

```
# 只计算数值列的总和
sum_numeric_only = df.sum(numeric_only=True)
print("\nSum numeric only:")
print(sum_numeric_only)

# 使用 min_count 参数
sum_with_min_count = df.sum(min_count=2)
print("\nSum with min_count=2:")
print(sum_with_min_count)
```

4.5.3 mean

用于计算 DataFrame 中数值的平均值。

```
DataFrame.mean(axis=0, skipna=True, numeric_only=False, **kwargs)
```

- `axis`: {0 或 'index', 1 或 'columns', None}, 默认为 0。这个参数决定了计算平均值是在哪个轴上进行:
  - 如果 `axis=0` 或 `axis='index'`，则对每列进行计算，返回一个 Series，其索引为列名，值为每列的平均值。
  - 如果 `axis=1` 或 `axis='columns'`，则对每行进行计算，返回一个 Series，其索引为行索引，值为每行的平均值。
- `skipna`: 布尔值，默认为 True。如果为 True，则在计算平均值时会忽略 NaN 值。
- `numeric_only`: 布尔值，默认为 False。如果为 True，则只对数值列进行计算，忽略非数值列。
- `**kwargs`: 其他关键字参数。这些参数通常用于兼容性或特殊用途，通常不需要。

```
import pandas as pd
import numpy as np

# 创建一个包含 NaN 值的 DataFrame
df = pd.DataFrame({
    'A': [1, 2, np.nan, 4],
    'B': [5, np.nan, np.nan, 8],
    'C': ['foo', 'bar', 'baz', 'qux']  # 非数值列
})

# 计算每列的平均值
mean_per_column = df.mean()
print("Mean per column:")
print(mean_per_column)

# 计算每行的平均值
mean_per_row = df.mean(axis='columns')
print("\nMean per row:")
print(mean_per_row)

# 只计算数值列的平均值
mean_numeric_only = df.mean(numeric_only=True)
print("\nMean numeric only:")
print(mean_numeric_only)
```

4.5.4 median

用于计算 DataFrame 中数值的中位数。

```
DataFrame.median(axis=0, skipna=True, numeric_only=False, **kwargs)
```

- `axis`: {0 或 'index', 1 或 'columns'}, 默认为 0。这个参数决定了计算中位数是在哪个轴上进行:
  - 如果 `axis=0` 或 `axis='index'`，则对每列进行计算，返回一个 Series，其索引为列名，值为每列的中位数。
  - 如果 `axis=1` 或 `axis='columns'`，则对每行进行计算，返回一个 Series，其索引为行索引，值为每行的中位数。
- `skipna`: 布尔值，默认为 True。如果为 True，则在计算中位数时会忽略 NaN 值。
- `numeric_only`: 布尔值，默认为 False。如果为 True，则只对数值列进行计算，忽略非数值列。
- `**kwargs`: 其他关键字参数。这些参数通常用于兼容性或特殊用途，通常不需要。

```
import pandas as pd
import numpy as np

# 创建一个包含 NaN 值的 DataFrame
df = pd.DataFrame({
    'A': [1, 2, np.nan, 4],
    'B': [5, np.nan, 7, 8],
    'C': ['foo', 'bar', 'baz', 'qux']  # 非数值列
})

# 计算每列的中位数
median_per_column = df.median()
print("Median per column:")
print(median_per_column)

# 计算每行的中位数
median_per_row = df.median(axis='columns')
print("\nMedian per row:")
print(median_per_row)

# 只计算数值列的中位数
median_numeric_only = df.median(numeric_only=True)
print("\nMedian numeric only:")
print(median_numeric_only)
```


4.5.5 min

用于计算 DataFrame 中数值的最小值。

```
DataFrame.min(axis=0, skipna=True, numeric_only=False, **kwargs)
```

- `axis`: {0 或 'index', 1 或 'columns'}, 默认为 0。这个参数决定了计算最小值是在哪个轴上进行:
  - 如果 `axis=0` 或 `axis='index'`，则对每列进行计算，返回一个 Series，其索引为列名，值为每列的最小值。
  - 如果 `axis=1` 或 `axis='columns'`，则对每行进行计算，返回一个 Series，其索引为行索引，值为每行的最小值。
- `skipna`: 布尔值，默认为 True。如果为 True，则在计算最小值时会忽略 NaN 值。

- `numeric_only`：布尔值，默认为 False。如果为 True，则只对数值列进行计算，忽略非数值列。
- `**kwargs`：其他关键字参数。这些参数通常用于兼容性或特殊用途，通常不需要。

```
import pandas as pd
import numpy as np

# 创建一个包含 NaN 值的 DataFrame
df = pd.DataFrame({
    'A': [1, 2, np.nan, 4],
    'B': [5, np.nan, 7, 8],
    'C': ['foo', 'bar', 'baz', 'qux']  # 非数值列
})

# 计算每列的最小值
min_per_column = df.min()
print("Min per column:")
print(min_per_column)

# 计算每行的最小值
min_per_row = df.min(axis='columns')
print("\nMin per row:")
print(min_per_row)

# 只计算数值列的最小值
min_numeric_only = df.min(numeric_only=True)
print("\nMin numeric only:")
print(min_numeric_only)
```

4.5.6 max

用于计算 DataFrame 中数值的最大值。

```
DataFrame.max(axis=0, skipna=True, numeric_only=False, **kwargs)
```

- `axis`：{0 或 'index', 1 或 'columns'}, 默认为 0。这个参数决定了计算最大值是在哪个轴上进行：
  - 如果 `axis=0` 或 `axis='index'`，则对每列进行计算，返回一个 Series，其索引为列名，值为每列的最大值。
  - 如果 `axis=1` 或 `axis='columns'`，则对每行进行计算，返回一个 Series，其索引为行索引，值为每行的最大值。
- `skipna`：布尔值，默认为 True。如果为 True，则在计算最大值时会忽略 NaN 值。
- `numeric_only`：布尔值，默认为 False。如果为 True，则只对数值列进行计算，忽略非数值列。
- `**kwargs`：其他关键字参数。这些参数通常用于兼容性或特殊用途，通常不需要。

```
import pandas as pd
import numpy as np

# 创建一个包含 NaN 值的 DataFrame
df = pd.DataFrame({
    'A': [1, 2, np.nan, 4],
    'B': [5, np.nan, 7, 8],
    'C': ['foo', 'bar', 'baz', 'qux']  # 非数值列
})
```

```
# 计算每列的最大值
max_per_column = df.max()
print("Max per column:")
print(max_per_column)

# 计算每行的最大值
max_per_row = df.max(axis='columns')
print("\nMax per row:")
print(max_per_row)

# 只计算数值列的最大值
max_numeric_only = df.max(numeric_only=True)
print("\nMax numeric only:")
print(max_numeric_only)
```

4.5.7 var

用于计算 DataFrame 中数值的方差。

```
DataFrame.var(axis=0, skipna=True, ddof=1, numeric_only=False, **kwargs)
```

- `axis`: {0 或 'index', 1 或 'columns', None}, 默认为0。这个参数决定了计算方差是在哪个轴上进行:
  - 如果 `axis=0` 或 `axis='index'`，则对每列进行计算，返回一个 Series，其索引为列名，值为每列的方差。
  - 如果 `axis=1` 或 `axis='columns'`，则对每行进行计算，返回一个 Series，其索引为行索引，值为每行的方差。
- `skipna`: 布尔值，默认为 True。如果为 True，则在计算方差时会忽略 NaN 值。
- `ddof`: 整数，默认为 1。Delta Degrees of Freedom，计算样本方差时使用的无偏估计的自由度修正。对于整个群体的方差，`ddof` 应该设置为 0。
- `numeric_only`: 布尔值，默认为 False。如果为 True，则只对数值列进行计算，忽略非数值列。
- `**kwargs`: 其他关键字参数。这些参数通常用于兼容性或特殊用途，通常不需要。

```
import pandas as pd
import numpy as np

# 创建一个包含 NaN 值的 DataFrame
df = pd.DataFrame({
    'A': [1, 2, np.nan, 4],
    'B': [5, np.nan, 7, 8],
    'C': ['foo', 'bar', 'baz', 'qux']  # 非数值列
})

# 计算每列的方差
var_per_column = df.var()
print("Variance per column:")
print(var_per_column)

# 计算每行的方差
var_per_row = df.var(axis='columns')
print("\nVariance per row:")
print(var_per_row)
```

```
# 只计算数值列的方差，并且使用无偏估计（ddof=1）
var_numeric_only = df.var(numeric_only=True, ddof=1)
print("\nVariance numeric only with ddof=1:")
print(var_numeric_only)
```

4.5.8 std

用于计算 DataFrame 中数值的标准差。

```
DataFrame.std(axis=0, skipna=True, ddof=1, numeric_only=False, **kwargs)
```

- `axis`: {0 或 'index', 1 或 'columns', None}, 默认为 0。这个参数决定了计算标准差是在哪个轴上进行:
  - 如果 `axis=0` 或 `axis='index'`，则对每列进行计算，返回一个 Series，其索引为列名，值为每列的标准差。
  - 如果 `axis=1` 或 `axis='columns'`，则对每行进行计算，返回一个 Series，其索引为行索引，值为每行的标准差。
- `skipna`: 布尔值，默认为 True。如果为 True，则在计算标准差时会忽略 NaN 值。
- `ddof`: 整数，默认为 1。Delta Degrees of Freedom，计算样本标准差时使用的无偏估计的自由度修正。对于整个群体的标准差，`ddof` 应该设置为 0。
- `numeric_only`: 布尔值，默认为 False。如果为 True，则只对数值列进行计算，忽略非数值列。
- `**kwargs`: 其他关键字参数。这些参数通常用于兼容性或特殊用途，通常不需要。

```
import pandas as pd
import numpy as np

# 创建一个包含 NaN 值的 DataFrame
df = pd.DataFrame({
    'A': [1, 2, np.nan, 4],
    'B': [5, np.nan, 7, 8],
    'C': ['foo', 'bar', 'baz', 'qux']  # 非数值列
})

# 计算每列的标准差
std_per_column = df.std()
print("Standard deviation per column:")
print(std_per_column)

# 计算每行的标准差
std_per_row = df.std(axis='columns')
print("\nStandard deviation per row:")
print(std_per_row)

# 只计算数值列的标准差，并且使用无偏估计（ddof=1）
std_numeric_only = df.std(numeric_only=True, ddof=1)
print("\nStandard deviation numeric only with ddof=1:")
print(std_numeric_only)
```

4.5.9 quantile

用于计算 DataFrame 中数值的分位数。

```
DataFrame.quantile(q=0.5, axis=0, numeric_only=False, interpolation='linear', method='single')
```

- `q`: 可以是单个浮点数或浮点数列表，默认为 0.5。要计算的的分位数，应该在 0 到 1 之间。例如，q=0.5 表示中位数。
- `axis`: {0 或 'index', 1 或 'columns'}, 默认为 0。这个参数决定了计算分位数是在哪个轴上进行：
  - 如果 `axis=0` 或 `axis='index'`，则对每列进行计算，返回一个 Series 或 DataFrame，其索引为列名，值为每列的分位数。
  - 如果 `axis=1` 或 `axis='columns'`，则对每行进行计算，返回一个 Series 或 DataFrame，其索引为行索引，值为每行的分位数。
- `numeric_only`: 布尔值，默认为 False。如果为 True，则只对数值列进行计算，忽略非数值列。
- `interpolation`: {'linear', 'lower', 'higher', 'midpoint', 'nearest'}, 默认为 'linear'。这个参数决定了分位数在数据不包含精确分位数值时的插值方法：
  - 'linear': 线性插值。
  - 'lower': 选择小于分位数的最大值。
  - 'higher': 选择大于分位数的最小值。
  - 'midpoint': 选择两个相邻数据的中间值。
  - 'nearest': 选择最接近分位数的值。
- `method`: {'single', 'table'}, 默认为 'single'。这个参数决定了计算分位数的方法：
  - 'single': 对每列或每行单独计算分位数。
  - 'table': 使用整个表的分位数。选择table时，插值方法只能是higher、lower、nearest之一。

```
import pandas as pd

# 创建一个DataFrame
data = {
    'col1': [10, 20, 30, 40, 50],
    'col2': [15, 25, 35, 45, 55],
    'col3': [20, 30, 40, 50, 60]
}
df = pd.DataFrame(data)

# 按列方向计算0.5分位数（中位数）
col_median = df.quantile(0.5, axis=0)
print("按列方向的0.5分位数（中位数）:\n", col_median)

# 按行方向计算0.5分位数（中位数）
row_median = df.quantile(0.5, axis=1)
print("按行方向的0.5分位数（中位数）:\n", row_median)
```

4.5.10 cummax

用于计算 DataFrame 中数值的累积最大值。

```
DataFrame.cummax(axis=0, skipna=True, *args, **kwargs)
```

- `axis`: {0 或 'index', 1 或 'columns', None}, 默认为 0。这个参数决定了计算累积最大值是在哪个轴上进行：
  - 如果 `axis=0` 或 `axis='index'`，则对每列进行计算，返回一个 DataFrame，其索引保持不变，每一列的值是自上而下（从第一行到当前行）的最大值。
  - 如果 `axis=1` 或 `axis='columns'`，则对每行进行计算，返回一个 DataFrame，其列保持不变，每一行的值是自左向右（从第一列到当前列）的最大值。
- `skipna`: 布尔值，默认为 True。如果为 True，则在计算累积最大值时忽略 NaN 值；如果为 False，则包含 NaN 值，可能导致结果中包含 NaN。
- `*args` 和 `**kwargs`：这些参数用于传递给底层方法的其他参数，通常不需要使用。

```
import pandas as pd
import numpy as np

# 创建一个 DataFrame
df = pd.DataFrame({
    'A': [1, 2, np.nan, 4, 5],
    'B': [5, np.nan, 3, 2, 1],
})

# 计算每列的累积最大值
cummax_per_column = df.cummax(axis=0)
print("Cumulative max per column:")
print(cummax_per_column)

# 计算每行的累积最大值
cummax_per_row = df.cummax(axis=1)
print("\nCumulative max per row:")
print(cummax_per_row)
```

4.5.11 cummin

用于计算 DataFrame 中数值的累积最小值。

```
DataFrame.cummin(axis=0, skipna=True, *args, **kwargs)
```

- `axis`: {0 或 'index', 1 或 'columns', None}, 默认为 0。这个参数决定了计算累积最小值是在哪个轴上进行：
  - 如果 `axis=0` 或 `axis='index'`，则对每列进行计算，返回一个 DataFrame，其索引保持不变，每一列的值是自上而下（从第一行到当前行）的最小值。
  - 如果 `axis=1` 或 `axis='columns'`，则对每行进行计算，返回一个 DataFrame，其列保持不变，每一行的值是自左向右（从第一列到当前列）的最小值。
- `skipna`: 布尔值，默认为 True。如果为 True，则在计算累积最小值时忽略 NaN 值；如果为 False，则包含 NaN 值，可能导致结果中包含 NaN。
- `*args` 和 `**kwargs`: 这些参数用于传递给底层方法的其他参数，通常不需要使用。

```
import pandas as pd

# 创建一个 DataFrame
df = pd.DataFrame({
    'A': [3, 2, NaN, 4, 1],
    'B': [5, NaN, 3, 2, 6],
    'C': ['foo', 'bar', 'baz', 'qux', 'quux']  # 非数值列
```

```
})

# 计算每列的累积最小值
cummin_per_column = df.cummin(axis=0)
print("Cumulative min per column:")
print(cummin_per_column)

# 计算每行的累积最小值
cummin_per_row = df.cummin(axis=1)
print("\nCumulative min per row:")
print(cummin_per_row)
```

4.5.12 cumsum

用于计算 DataFrame 中数值的累积和。

```
DataFrame.cumsum(axis=0, skipna=True, *args, **kwargs)
```

- `axis`: {0 或 'index', 1 或 'columns', None}, 默认为 0。这个参数决定了计算累积和是在哪个轴上进行:
  - 如果 `axis=0` 或 `axis='index'`，则对每列进行计算，返回一个 DataFrame，其索引保持不变，每一列的值是自上而下（从第一行到当前行）的累积和。
  - 如果 `axis=1` 或 `axis='columns'`，则对每行进行计算，返回一个 DataFrame，其列保持不变，每一行的值是自左向右（从第一列到当前列）的累积和。
- `skipna`: 布尔值，默认为 True。如果为 True，则在计算累积和时忽略 NaN 值；如果为 False，则包含 NaN 值，可能导致结果中包含 NaN。
- `*args` 和 `**kwargs`: 这些参数用于传递给底层方法的其他参数，通常不需要使用。

```
import pandas as pd

# 创建一个 DataFrame
df = pd.DataFrame({
    'A': [1, 2, NaN, 4, 5],
    'B': [5, NaN, 3, 2, 6],
    'C': ['foo', 'bar', 'baz', 'qux', 'quux']  # 非数值列
})

# 计算每列的累积和
cumsum_per_column = df.cumsum(axis=0)
print("Cumulative sum per column:")
print(cumsum_per_column)

# 计算每行的累积和
cumsum_per_row = df.cumsum(axis=1)
print("\nCumulative sum per row:")
print(cumsum_per_row)
```

4.5.13 cumprod

用于计算 DataFrame 中数值的累积乘积。

```
DataFrame.cumprod(axis=0, skipna=True, *args, **kwargs)
```

- `axis`: {0 或 'index', 1 或 'columns', None}, 默认为 0。这个参数决定了计算累积乘积是在哪个轴上进行:
  - 如果 `axis=0` 或 `axis='index'`，则对每列进行计算，返回一个 DataFrame，其索引保持不变，每一列的值是自上而下（从第一行到当前行）的累积乘积。
  - 如果 `axis=1` 或 `axis='columns'`，则对每行进行计算，返回一个 DataFrame，其列保持不变，每一行的值是自左向右（从第一列到当前列）的累积乘积。
- `skipna`: 布尔值，默认为 True。如果为 True，则在计算累积乘积时忽略 NaN 值；如果为 False，则包含 NaN 值，可能导致结果中包含 NaN。
- `*args` 和 `**kwargs`: 这些参数用于传递给底层方法的其他参数，通常不需要使用。

```
import pandas as pd

# 创建一个 DataFrame
df = pd.DataFrame({
    'A': [1, 2, NaN, 4, 5],
    'B': [5, NaN, 3, 2, 6],
    'C': ['foo', 'bar', 'baz', 'qux', 'quux']  # 非数值列
})

# 计算每列的累积乘积
cumprod_per_column = df.cumprod(axis=0)
print("Cumulative product per column:")
print(cumprod_per_column)

# 计算每行的累积乘积
cumprod_per_row = df.cumprod(axis=1)
print("\nCumulative product per row:")
print(cumprod_per_row)
```

4.6 分组与聚合

4.6.1 groupby

用于将 DataFrame 分割成多个组，以便可以对这些组应用聚合函数或执行其他操作。

```
DataFrame.groupby(by=None, axis=0, level=None, as_index=True, sort=True,
group_keys=True, observed=False, dropna=True)
```

- `by`: 用于确定分组依据。
- `axis`: {0 或 'index', 1 或 'columns'}，默认为 0。这个参数决定了分组是在哪个轴上进行。
- `level`: 如果轴是多索引，则按指定的索引级别分组。
- `as_index`: 布尔值，默认为 True。当为 True 时，分组名称将作为结果的索引；如果为 False，则结果会保持原有的 DataFrame 结构，分组名称会作为一个普通列出现。
- `sort`: 布尔值，默认为 True。如果为 True，则对分组键进行排序；如果为 False，则不排序，保持原有的分组顺序。
- `group_keys`: 布尔值，默认为 True。如果为 True，则将组键添加到聚合后的结果中；如果为 False，则不添加。
- `observed`: 布尔值，默认为 False。仅当分组依据是多索引且包含分类数据时适用，如果为 True，则仅显示分类数据中的观察值。
- `dropna`: 布尔值，默认为 True。如果为 True，并且 `by` 是一个列表，则排除任何含有 NaN 的组。

```
import pandas as pd

# 创建一个 DataFrame
df = pd.DataFrame({
    'A': ['foo', 'bar', 'foo', 'bar', 'foo', 'bar', 'foo', 'foo'],
    'B': ['one', 'one', 'two', 'three', 'two', 'two', 'one', 'three'],
    'C': [1, 3, 2, 5, 4, 1, 2, 3],
    'D': [2, 5, 3, 7, 6, 2, 4, 6]
})

# 按 'A' 和 'B' 列分组，并计算每组的平均值
grouped = df.groupby(['A', 'B']).mean()

print(grouped)
```

```
import pandas as pd

# 创建两个列表，它们将用作多级索引的级别
arrays = [['Falcon', 'Falcon', 'Parrot', 'Parrot'],  # 第一级索引的值
          ['Captive', 'Wild', 'Captive', 'Wild']]   # 第二级索引的值

# 使用arrays列表创建一个DataFrame，'Max Speed'列包含对应于多级索引的数据
df = pd.DataFrame({'Max Speed': [390., 350., 30., 20.]},  # 数据
                  index=arrays)  # 使用arrays列表作为多级索引

# 打印原始DataFrame，以查看其结构和数据
print(df)

# 使用groupby方法按第一级索引（即'arrays'列表的第一个元素）分组
# level=0表示按照多级索引的第一级进行分组
res = df.groupby(level=0).mean()  # 计算每个组（即每个不同的第一级索引值）的平均速度

# 打印分组后的平均值，显示每个动物的Max Speed的平均值
print(res)
```

4.6.2 transform

对 DataFrame 或其中一个列应用一个函数，并返回一个与原始 DataFrame 具有相同索引的对象。此方法不会更改原始 DataFrame，而是返回一个新的 DataFrame，其中包含应用函数后的结果。

```
DataFrame.transform(func, axis=0, *args, **kwargs)
```

- `func`：函数，应用于每个组或列。这个函数可以是一个内置函数，比如 `numpy.mean`，也可以是一个自定义函数。
- `axis`：{0 或 'index', 1 或 'columns'}，默认为 0。这个参数决定了函数是在哪个轴上应用：
  - 如果 `axis=0` 或 `axis='index'`，则函数按列应用。
  - 如果 `axis=1` 或 `axis='columns'`，则函数按行应用。
- `*args`：位置参数，将传递给 `func`。
- `**kwargs`：关键字参数，将传递给 `func`。

```
import pandas as pd
import numpy as np
```

```
# 创建一个 DataFrame
df = pd.DataFrame({
    'A': [1, 2, 3, 4, 5],
    'B': [10, 20, 30, 40, 50],
    'C': [100, 200, 300, 400, 500]
})


def square(x):
    return x ** 2


# 调用transform函数，并将自定义函数带入
transformed_df = df.transform(square)

print(transformed_df)
```

4.6.3 agg

它允许你应用一个或多个聚合函数到 DataFrame 的列或行上，并返回聚合后的结果。

```
DataFrame.agg(func=None, axis=0, *args, **kwargs)
```

- `func`：函数或函数列表/字典，应用于 DataFrame 的列或行。如果传递一个函数，它将应用于所有列。如果传递一个函数列表或字典，则可以分别对不同的列应用不同的函数。可以使用的函数包括 NumPy 的统计函数、Python 的内置函数、Pandas 的自定义聚合函数，或者自定义的 lambda 函数。
- `axis`：{0 或 'index', 1 或 'columns'}，默认为 0。这个参数决定了聚合操作是在哪个轴上执行：
  - 如果 `axis=0` 或 `axis='index'`，则函数按列应用（逐行操作）。
  - 如果 `axis=1` 或 `axis='columns'`，则函数按行应用（逐列操作）。
- `*args`：位置参数，将传递给 `func`。
- `**kwargs`：关键字参数，将传递给 `func`。

```
import pandas as pd

# 创建一个 DataFrame
df = pd.DataFrame({
    'A': [1, 2, 3, 4, 5],
    'B': [10, 20, 30, 40, 50]
})

# 应用单个聚合函数，计算每列的平均值
result = df.agg('mean')
print(result)

# 对列 'A' 应用 'sum' 函数，对列 'B' 应用 'mean' 函数
result = df.agg({'A': 'sum', 'B': 'mean'})
print(result)

# 对所有列应用多个聚合函数
result = df.agg(['min', 'max', 'sum'])
print(result)

# 定义一个自定义函数
```

```
def custom_agg(x):
    return (x.max() - x.min()) / x.std()

# 使用自定义函数进行聚合
result = df.agg(custom_agg)
print(result)
```

4.7 数据可视化

4.7.1 plot

用于绘制 DataFrame 数据图形，它允许用户直接从 DataFrame 创建各种类型的图表，而不需要使用其他绘图库（底层实际上使用了 Matplotlib）。

```
DataFrame.plot(*args, **kwargs)
```

以下是一些常见的 `DataFrame.plot` 方法的关键字参数：

- `kind`：{'line', 'bar', 'barh', 'hist', 'box', 'kde', 'density', 'area', 'pie', 'scatter', 'hexbin'}，默认为 'line'。这个参数决定了要绘制哪种类型的图表。
- `ax`：Matplotlib axes 对象，可选。如果提供，则绘图将在指定的 axes 对象上进行。
- `subplots`：布尔值，默认为 False。如果为 True，则每个数值列将绘制在其自己的子图中。
- `sharex`：布尔值，默认为 False。如果为 True，并且 `subplots` 为 True，则所有子图共享相同的 x 轴。
- `sharey`：布尔值，默认为 False。如果为 True，并且 `subplots` 为 True，则所有子图共享相同的 y 轴。
- `layout`：元组（行数，列数），可选。用于安排子图的布局。
- `figsize`：元组（宽度，高度），可选。用于设置图表的大小。
- `use_index`：布尔值，默认为 True。如果为 True，则将 DataFrame 的索引用作 x 轴。
- `title`：字符串，可选。图表的标题。
- `grid`：布尔值，默认为 None。如果为 True，则在图表上显示网格线。

```
import pandas as pd
import matplotlib.pyplot as plt


# 创建一个 DataFrame
df = pd.DataFrame({
    'A': [1, 2, 3, 4, 5],
    'B': [10, 20, 30, 40, 50]
})

# 绘制线图
df.plot(kind='line')

# 显示图像
plt.show()
```

4.7.2 hist

用于绘制 DataFrame 中数据的直方图。

```
DataFrame.hist(column=None, by=None, grid=True, xlabelsize=None, xrot=None,
ylabelsize=None, yrot=None, ax=None, sharex=False, sharey=False, figsize=None,
layout=None, bins=10, backend=None, legend=False, **kwargs)
```

- `column`：字符串或字符串列表，可选。用于指定要绘制直方图的列。如果未指定，则默认绘制所有数值列的直方图。
- `by`：字符串或字符串列表，可选。用于指定分组变量，以便为每个组绘制直方图。
- `grid`：布尔值，默认为 True。是否在直方图上显示网格线。
- `xlabelsize`：整数，可选。用于设置 x 轴标签的大小。
- `xrot`：浮点数，可选。用于设置 x 轴标签的旋转角度。
- `ylabelsize`：整数，可选。用于设置 y 轴标签的大小。
- `yrot`：浮点数，可选。用于设置 y 轴标签的旋转角度。
- `ax`：Matplotlib axes 对象或数组，可选。如果提供，则在指定的 axes 对象上绘制直方图。
- `sharex`：布尔值，默认为 False。如果为 True，并且绘制多个直方图，则所有直方图共享相同的 x 轴。
- `sharey`：布尔值，默认为 False。如果为 True，并且绘制多个直方图，则所有直方图共享相同的 y 轴。
- `figsize`：元组（宽度，高度），可选。用于设置整个图形的大小。
- `layout`：元组（行数，列数），可选。用于指定直方图的布局。
- `bins`：整数或序列，默认为 10。用于设置直方图的柱子数量或边界。
- `backend`：字符串，可选。用于指定绘图的后端。默认情况下，Pandas 使用 Matplotlib。
- `legend`：布尔值，默认为 False。如果为 True，并且指定了 `by` 参数，则在直方图上显示图例。
- `**kwargs`：关键字参数，这些参数会被传递给 Matplotlib 的 `hist` 函数，用于进一步自定义直方图。

```
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np

# 创建一个包含正态分布数据的 DataFrame
df = pd.DataFrame(np.random.randn(100, 3), columns=['A', 'B', 'C'])

# 绘制 A 列的直方图，设置 20 个柱子，并指定图形大小
df['A'].hist(bins=20, figsize=(8, 4))

# 绘制所有列的直方图，每个直方图在不同的子图中
df.hist(figsize=(10, 7), bins=30)

# 使用 by 参数按列 B 的值分组绘制列 A 的直方图
df.hist(column='A', by='B', bins=15, figsize=(10, 5))

# 显示图像
plt.show()
```

4.8 其他常用方法

4.8.1 nunique

用于计算DataFrame中唯一值的数量。

```
DataFrame.nunique(axis=0, dropna=True)
```

- `axis=0`：默认值，表示按列进行操作。如果设置为`axis=1`，则表示按行进行操作。
- `dropna=True`：默认值，表示在计算唯一值之前先排除掉缺失值（NaN）。如果设置为`False`，则缺失值也会被计算在内。

```
import pandas as pd
import numpy as np


# 创建一个DataFrame
df = pd.DataFrame({
    'A': [1, 2, 2, 3, 3, 3],
    'B': ['a', 'b', 'b', 'c', 'c', 'c'],
    'C': [np.nan, np.nan, 1, 2, 2, 2]
})

# 使用nunique()方法计算每列的唯一值数量
unique_counts = df.nunique(axis=0, dropna=True)

print(unique_counts)
```

4.8.2 value_counts

用于计算DataFrame中各个值出现的频率的一个方法。

```
DataFrame.value_counts(subset=None, normalize=False, sort=True, ascending=False,
dropna=True)
```

- `subset=None`：可选参数，用于指定要进行计算操作的列名列表。如果未指定，则对整个DataFrame的所有列进行操作。
- `normalize=False`：布尔值，默认为False。如果设置为True，则返回每个值的相对频率，而不是计数。
- `sort=True`：布尔值，默认为True。如果设置为True，则结果将按计数值降序排序。
- `ascending=False`：布尔值，默认为False。当`sort=True`时，此参数指定排序顺序。如果设置为True，则结果将按计数值升序排序。
- `dropna=True`：布尔值，默认为True。如果设置为True，则在计数之前排除缺失值。

```
import pandas as pd
import numpy as np

# 创建一个DataFrame
df = pd.DataFrame({
    'A': [1, 2, 2, 3, 3, 3],
    'B': ['a', 'b', 'b', 'c', 'c', 'c'],
    'C': [np.nan, np.nan, 1, 2, 2, 2]
})

# 使用value_counts()方法计算值的频率
```

```
value_counts = df.value_counts(subset=['A', 'B'], normalize=False, sort=True,
ascending=False, dropna=True)

print(value_counts)
```

4.8.3 describe

用于生成DataFrame中数值列的统计摘要，会返回一个DataFrame，其中包含以下统计信息：

- `count`：非NA值的数量
- `mean`：平均值
- `std`：标准差
- `min`：最小值
- `分位数`：可指定
- `max`：最大值

```
DataFrame.describe(percentiles=None, include=None, exclude=None)
```

- `percentiles=None`：一个列表形式的数值，用于指定要计算的分位数。默认情况下，会计算25%，50%，和75%这三个分位数。如果你想要不同的分位数，可以在这里指定。
- `include=None`：一个列表形式的字符串或字符串序列，用于指定要包含在描述统计中的数据类型。默认情况下，只包括数值类型的数据。
- `exclude=None`：一个列表形式的字符串或字符串序列，用于指定要从描述统计中排除的数据类型。

```
import pandas as pd

# 创建一个DataFrame
df = pd.DataFrame({
    'A': [1, 2, 2, 3, 3, 3],
    'B': [10, 20, 20, 30, 30, 30],
    'C': ['foo', 'bar', 'bar', 'baz', 'baz', 'baz']
})

# 使用describe()方法生成统计摘要
description = df.describe(percentiles=[0.5, 0.75, 0.9], include=['number'],
exclude=['object'])

print(description)
```

4.8.4 copy

用于创建DataFrame的一个副本。

```
DataFrame.copy(deep=True)
```

- `deep=True`：默认为True，表示创建一个深拷贝。在深拷贝中，DataFrame中的所有数据都会被复制到新的DataFrame中，因此原始DataFrame中的数据与副本DataFrame中的数据是完全独立的。如果你对副本进行修改，原始DataFrame不会受到影响，反之亦然。

```
import pandas as pd

# 创建一个DataFrame
```

```
df = pd.DataFrame({
    'A': [1, 2, 3],
    'B': [4, 5, 6]
})

# 创建深拷贝
df_deep_copy = df.copy()

# 创建浅拷贝
df_shallow_copy = df.copy(deep=False)

# 修改深拷贝中的数据
df_deep_copy['A'][0] = 999

# 修改浅拷贝中的数据
df_shallow_copy['B'][0] = 888

# 打印原始DataFrame和副本
print("Original DataFrame:")
print(df)
print("\nDeep Copy:")
print(df_deep_copy)
print("\nShallow Copy:")
print(df_shallow_copy)
```

4.8.5 reset_index

用于重置DataFrame的索引。

```
DataFrame.reset_index(level=None, *, drop=False, inplace=False, col_level=0,
col_fill='', allow_duplicates=_NoDefault.no_default, names=None)
```

- `level=None`：整数或索引名称，用于指定要重置的索引级别。对于多级索引的DataFrame，可以指定一个级别或级别列表。如果为None，则重置所有级别。
- `drop=False`：布尔值，默认为False。如果为True，则重置索引后，原始索引将不会作为列添加到DataFrame中。如果为False，则原始索引将作为新列添加到DataFrame中。
- `inplace=False`：布尔值，默认为False。如果为True，则直接在原始DataFrame上进行操作，不返回新的DataFrame。
- `col_level=0`：如果原始索引是多级索引，则指定新列的索引级别。
- `col_fill=''`：如果原始索引是多级索引，并且`col_level`比原始索引的级别多，则使用此值填充缺失的级别。
- `allow_duplicates=_NoDefault.no_default`：允许索引列中出现重复值。如果设置为False，则在尝试插入重复列时会引发ValueError。
- `names=None`：列表或元组，用于为新列指定名称。默认情况下，如果`drop`为False，则使用原始索引的名称。

```
import pandas as pd

# 创建两个列表，它们将用作多级索引的级别
arrays = [['Falcon', 'Falcon', 'Parrot', 'Parrot'],  # 第一级索引的值
          ['Captive', 'Wild', 'Captive', 'Wild']]   # 第二级索引的值

# 使用arrays列表创建一个DataFrame，'Max Speed'列包含对应于多级索引的数据
df = pd.DataFrame({'Max Speed': [390., 350., 30., 20.]},  # 数据
```

```
                    index=arrays)  # 使用arrays列表作为多级索引

print(df)
# 重置所有索引级别，并将它们作为列添加到DataFrame中
df_reset = df.reset_index()
print(df_reset)


# 只重置第二级索引，并将它作为列添加到DataFrame中
df_reset_level_1 = df.reset_index(level=1)
print(df_reset_level_1)

# 重置所有索引级别，但不将它们作为列添加到DataFrame中
df_reset_drop = df.reset_index(drop=True)
print(df_reset_drop)
```

4.8.6 info

用于显示DataFrame的概要信息的一个便捷方法，它提供了关于DataFrame的列、非空值数量、数据类型以及内存使用情况的信息。

```
DataFrame.info(verbose=None, buf=None, max_cols=None, memory_usage=None,
show_counts=None)
```

- `verbose=None`：控制输出信息的详细程度。
- `buf=None`：一个打开的文件对象或类似文件的对象。如果提供，则输出将被写入这个缓冲区而不是标准输出。
- `max_cols=None`：要显示的最大列数。如果DataFrame的列数超过这个值，则只会显示部分列的信息。
- `memory_usage=None`：控制是否显示内存使用情况。可以设置为True或False，或者是一个字符串，'deep'表示深度计算内存使用情况，考虑对象内部数据。
- `show_counts=None`：控制是否显示非空值的数量。如果设置为True，则会在每个列旁边显示非空值的数量。

```
import pandas as pd

# 创建一个DataFrame
df = pd.DataFrame({
    'A': [1, 2, 3, None],
    'B': ['a', 'b', 'c', 'd'],
    'C': [1.1, 2.2, 3.3, 4.4]
})

# 显示DataFrame的概要信息
df.info(verbose=True, memory_usage='deep', show_counts=True)
```

4.8.7 apply

允许你对DataFrame中的每个元素、行或列应用一个函数。

```
DataFrame.apply(func, axis=0, raw=False, result_type=None, args=(), **kwargs)
```

- `func`：函数，应用于每个元素、行或列。这个函数需要你自定义，或者使用内置的函数。

- `axis=0`：指定应用函数的轴。
- `raw=False`：布尔值，默认为False，则每行或每列将作为Series使用，如果为True，则将作为Ndarray数组，性能会更好。
- `result_type=None`：控制返回类型。可以是以下之一：
  - ‘reduce’：如果可能的话，如果应用函数返回一个列表，这个列表会被转换成一个 Series。
  - ‘broadcast’：结果会被广播到原始 DataFrame 的形状，保留原始的索引和列。
  - ‘expand’：如果应用函数返回一个列表（或类似列表的结构），则这个列表会被转换为多个列。这意味着每个列表元素都会变成 DataFrame 的一列。
- `args=()`：元组，包含传递给函数的位置参数。
- `**kwargs`：关键字参数，将被传递给函数。

```
import pandas as pd

# 创建一个DataFrame
df = pd.DataFrame({
    'A': [1, 2, 3],
    'B': [4, 5, 6],
    'C': [7, 8, 9]
})

# 定义一个函数，计算每行的平均值
def mean_row(row):
    return row.mean()

# 应用函数到每行
result = df.apply(mean_row, axis=1)

print(result)
```

5. 保存与读取

5.1 to_csv

用于将DataFrame对象保存为CSV（逗号分隔值）文件的方法。

```
DataFrame.to_csv(path_or_buf=None, sep=',', na_rep='', float_format=None,
columns=None, header=True, index=True, mode='w', encoding=None, quoting=None,
quotechar='"',  **kwargs)
```

- `path_or_buf`：指定输出文件的路径或文件对象。
- `sep`：字段分隔符，通常使用逗号`,`或制表符`\t`。
- `na_rep`：缺失值的表示方式，默认为空字符串`''`。
- `float_format`：浮点数的格式化方式，例如`'%.2f'`用于格式化为两位小数。
- `columns`：要写入的列的子集，默认为None，表示写入所有列。
- `header`：是否写入列名，通常设置为True。
- `index`：是否写入行索引，通常设置为True或False，取决于是否需要索引。
- `mode`：写入模式，通常使用‘w’（写入，覆盖原文件）或‘a’（追加到文件末尾）。
- `encoding`：文件编码，特别是在处理非ASCII字符时很重要
- `quoting`：控制字段引用的行为，通常用于确保字段中的分隔符被正确处理。
- `quotechar`：用于包围字段的字符，默认为双引号`"`。
- `**kwargs`：其他关键字参数。

```
import pandas as pd

# 创建一个简单的DataFrame
data = {
    '姓名': ['张三', '李四', '王五'],
    '年龄': [28, 34, 29],
    '城市': ['北京', '上海', '广州']
}

df = pd.DataFrame(data)
# 将DataFrame保存为CSV文件
df.to_csv('人员信息.csv', index=False, encoding='utf_8_sig')
```

5.2 to_excel

在操作excel表格之前，需要安装一个库：`openpyxl`，因为`pandas`库本身并不包含写入Excel文件的直接支持，如果没有安装的话将无法操作excel。

其安装命令为：

```
pip install openpyxl
```

Pandas中的`to_excel`用于将DataFrame保存为Excel文件。

```
DataFrame.to_excel(excel_writer, sheet_name='Sheet1', na_rep='',
float_format=None, columns=None, header=True, index=True, index_label=None,
startrow=0, startcol=0, engine=None, merge_cells=True, inf_rep='inf',
freeze_panes=None, storage_options=None)
```

- `excel_writer`：字符串或ExcelWriter对象，指定输出文件的路径或文件对象。
- `sheet_name='Sheet1'`：要写入的工作表名称。
- `na_rep=''`：指定缺失值的表示方式。
- `float_format=None`：浮点数的格式化方式，例如'%.2f'。
- `columns=None`：要写入的列的子集，默认为None，表示写入所有列。
- `header=True`：是否写入列名，默认为True。
- `index=True`：是否写入行索引，默认为True。
- `index_label=None`：指定行索引列的列名，如果为None，并且`header`为True，则使用索引名。
- `startrow=0`：写入DataFrame的起始行位置，默认为0。
- `startcol=0`：写入DataFrame的起始列位置，默认为0。
- `engine=None`：指定用于写入文件的引擎，可以是'openpyxl'（默认）或'xlsxwriter'。
- `merge_cells=True`：是否合并单元格，这在有合并单元格的Header时很有用。
- `inf_rep='inf'`：指定无限大的表示方式。
- `freeze_panes=None`：指定冻结窗口的单元格范围，例如'A2'。
- `storage_options=None`：指定存储连接的参数，例如认证凭据。

```
import pandas as pd

# 创建一个简单的DataFrame
data = {
    '姓名': ['张三', '李四', '王五'],
    '年龄': [28, 34, 29],
    '城市': ['北京', '上海', '广州']
}

df = pd.DataFrame(data)
# 将DataFrame保存为Excel文件
df.to_excel('人员信息.xlsx', index=False)
```

5.3 read_csv

`pandas.read_csv` 是一个非常强大的函数，用于从文件、URL、文件-like对象等读取逗号分隔值（CSV）文件。这个函数有很多参数，允许你以多种方式自定义数据加载过程。

```
pandas.read_csv(filepath_or_buffer, sep, header, usercols, na_values,
parse_dates, skiprows, nrows)
```

- `filepath_or_buffer`：指定要读取的 CSV 文件的路径或文件对象。可以是一个字符串，表示文件的绝对路径或相对路径；也可以是一个已经打开的文件对象（例如通过 `open()` 函数打开的文件）。
- `sep`：字符串，用于分隔字段的字符。默认是逗号 `,`，但可以是任何字符，例如 `';'` 或 `'\t'`（制表符）。
- `header`：整数或整数列表，用于指定行号作为列名，或者没有列名（例如 `header=None`）。默认为 `'infer'`，表示自动检测列名。
- `usecols`：列表或 callable，用于指定要读取的列。可以是列名的列表，也可以是列号的列表。
- `na_values`：字符串、列表或字典，用于指定哪些其他值应该被视为 `NA` / `NaN`。
- `parse_dates`：列表或字典，用于指定将哪些列解析为日期。
- `skiprows`：整数或列表，用于指定要跳过的行号或条件。
- `nrows`：整数，用于指定要读取的行数。

5.4 read_excel

`pandas.read_excel` 是pandas库中用于读取Excel文件（`.xls` 或 `.xlsx`）的函数。它可以将Excel文件中的数据读取为DataFrame对象，便于进行数据分析和处理。

```
pandas.read_excel(io, sheet_name=0, header=0, index_col=None, usecols=None,
squeeze=False, dtype=None, skiprows=None, nrows=None, na_values=None,
keep_default_na=True, parse_dates=False, date_parser=None, skipfooter=0,
convert_float=True, **kwds)
```

- `io`：文件路径或文件对象。这是唯一必需的参数，用于指定要读取的Excel文件。
- `sheet_name=0`：要读取的表名或表的索引号。默认为0，表示读取第一个工作表。可以指定工作表名或索引号，如果指定多个，将返回一个字典，键为工作表名，值为对应的DataFrame。
- `header=0`：用作列名的行号，默认为0，即第一行作为列名。如果没有标题行，可以设置为None。
- `index_col=None`：用作行索引的列号或列名。默认为None，表示不使用任何列作为索引。可以是一个整数、字符串或列名的列表。

- `usecols=None`：要读取的列。默认为None，表示读取所有列。可以是一个整数列表、字符串列表或Excel列的位置（如`[0, 1, 2]`）或字母标记（如`['A', 'B', 'C']`）。
- `squeeze=False`：如果读取的数据只有一列，当设置为True时，返回一个Series而不是DataFrame。
- `dtype=None`：指定某列的数据类型。默认为None，表示自动推断。可以是一个字典，键为列名，值为NumPy数据类型。
- `skiprows=None`：要跳过的行号或行号列表。默认为None，表示不跳过任何行。可以是整数或整数列表。
- `nrows=None`：读取的行数，从文件头开始。默认为None，表示读取所有行。
- `na_values=None`：将指定的值替换为NaN。默认为None，表示不替换。可以是一个值或值的列表。
- `keep_default_na=True`：如果为True（默认），则除了通过`na_values`指定的值外，还将默认的NaN值视为NaN。
- `parse_dates=False`：是否尝试将列解析为日期。默认为False。可以是一个布尔值、列名列表或列号的列表。
- `date_parser=None`：用于解析日期的函数。默认为None，表示使用pandas默认的日期解析器。
- `skipfooter=0`：要跳过的文件底部的行数。默认为0，表示不跳过任何底部的行。
- `convert_float=True`：是否将所有浮点数转换为64位浮点数。默认为True，以避免数据类型推断问题。
- `**kwds`：允许用户传递其他关键字参数，这些参数可能会被引擎特定的读取器所识别。

6.案例

使用Pandas读取准备好的学生成绩表，计算每个学生的最终成绩，最终成绩是平时成绩的百分之三十加上考试成绩的百分之七十，判断是否及格（60），并将最终成绩和及格率填到表格里，并保存，然后使用Matplotlib进行显示最终成绩和及格率。

```
import numpy as np  # 导入NumPy库，用于数学计算
import pandas as pd  # 导入Pandas库，用于数据处理
import matplotlib.pyplot as plt  # 导入Matplotlib库，用于绘制图表

# 从Excel文件中读取数据到DataFrame
df = pd.read_excel('./source.xlsx')

# 使用fillna方法将DataFrame中的缺失值替换为0
df = df.fillna(0)

# 从DataFrame中提取考试分数列的值
exam_data = df['exam'].values
# 从DataFrame中提取出勤分数列的值
attendance_data = df['attendance'].values

# 使用NumPy的round函数计算最终成绩，考试分数占70%，出勤分数占30%
finally_data = np.round(exam_data * 0.7 + attendance_data * 0.3)

# 将计算得到的最终成绩添加到DataFrame的新列'finally'中
df['finally'] = finally_data

# 使用apply函数根据最终成绩判断是否通过（60分及以上），并创建新列'pass'
df['pass'] = df['finally'].apply(lambda x: 'yes' if x >= 60 else 'no')

# 设置直方图的区间边界，从0到110，步长为10
bins = np.arange(0, 111, 10)
# 使用np.histogram函数计算每个区间的学生人数
```

```
hist, bin_edges = np.histogram(df['finally'], bins=bins)

# 创建一个图表实例
fig = plt.figure()
# 计算条形图的宽度
bar_width = (bin_edges[1] - bin_edges[0])
# 绘制条形图，x轴是分数区间，y轴是学生人数
plt.bar(bin_edges[:-1], hist, width=bar_width, align='edge')

# 在每个条形图上添加文本，显示该区间内的学生人数
for i in range(len(hist)):
    if hist[i]:  # 如果该区间有学生，则添加文本
        plt.text(bin_edges[i] + bar_width / 2, hist[i] + 0.1, str(hist[i]),
ha='center')

# 设置图表的标题和坐标轴标签
plt.title('finally')
plt.xlabel('score')
plt.ylabel('number')

# 设置x轴的刻度，显示分数区间的边界
plt.xticks(bin_edges[:-1])

# 显示条形图
plt.show()

# 创建一个新的图表实例
fig = plt.figure()

# 使用value_counts方法统计通过和未通过的学生数量
pass_count = df['pass'].value_counts()

# 绘制饼图，显示通过和未通过的比例
plt.pie(pass_count, labels=pass_count.index, autopct='%1.1f%%')
# 设置饼图的标题
plt.title('Distribution of Passing Status')
# 显示饼图
plt.show()
```