pandas-intro

2023 年 8 月 28 日

[207]:
```
print("hi")
```

```
hi
```

1 快速开始

[208]:
```
import numpy as np
import pandas as pd
```

2 Series (系列/列表)

- Series 是一个一维的数组, 里面可以是任意的数据类型, 整数, 字符串等等....
- 其常用的创建格式如下

```
import pandas as pd
s = pd.Series(data,index=index)
```

其中 `index` 是一个列表, 存放着各个列的标签如果 `data` 是个存放 n 个元素的列表, 那么 `index` 的长度应和 `data` 一样, 也是 n

2.1 从数组创建系列 (Series)

[209]:
```
s= pd.Series(np.random.randn(5),index=['a','b','c','d','e'])
s
```

[209]:
```
a    0.074406
b   -0.198687
c    0.358383
d    0.808076
e   -0.404227
dtype: float64
```

[210]:
```
s.index
```

[210]:
```
Index(['a', 'b', 'c', 'd', 'e'], dtype='object')
```

在 *pandas* 中，索引值可以不唯一，即可以存在重复的索引值。这种设计允许用户在数据集中使用相同的索引值来标识不同的数据行。然而，某些操作可能要求索引值是唯一的，例如合并数据集或执行聚合操作等。如果在这些操作中尝试使用不支持重复索引值的方法，将会在操作时引发一个异常。这个异常是 *pandas* 提供的一种错误提示机制，用于指示当前操作不兼容当前的索引设置。

2.2 从字典创建系列

[211]:
```
d = {
    'b':1,
    'a':0,
    'c':2
}
pd.Series(d)
```

[211]:
```
b    1
a    0
c    2
dtype: int64
```

- 在将字典 `dict` 转换为 `Pandas` 的 `Series` 对象时，可以传递一个索引 `index` 给 `Series` 构造函数。索引是一组标签，用于标识数据中的每个值。如果传递了一个索引，那么在转换过程中，将会从数据 `data` 中提取与索引标签对应的值，并按照索引的顺序创建 `Series` 对象。

[212]:
```
pd.Series(d,index=['b','c','d','a'])
```

[212]:
```
b    1.0
c    2.0
```

```
d    NaN
a    0.0
dtype: float64
```

NaN (not a number) is the standard missing data marker used in pandas.

2.3 从常数创建系列

-可以从常数创建一个 Series, 其创建后的长度取决于 index 的长度

[213]:
```
pd.Series(5,index=['a','b','c','d','e'])
```

[213]:
```
a    5
b    5
c    5
d    5
e    5
dtype: int64
```

2.4 系列 (Series) 和多维数组 (ndarray) 很像

ndarray 是 NumPy 库中的一个重要数据结构，它代表多维数组（n-dimensional array）。

ndarray 是 NumPy(Numerical Python) 库的核心对象之一，它是一个多维、固定大小的数组，其中的元素必须是相同类型的。它可以是一维、二维或更高维度的数组，可以包含整数、浮点数、布尔值、字符串等各种数据类型。

ndarray 在数据分析和科学计算中被广泛使用，它提供了高效的存储和处理多维数据的能力。通过使用 NumPy 的函数和方法，可以对 ndarray 进行各种数学运算、统计分析、数组操作和广播等操作。

ndarray 不同于 Python 内置的列表 list 数据结构，它具有更高的性能和更丰富的操作功能，特别适合处理大规模数据和进行数值计算。Series 的操作方法和 ndarray 很相似, 以及在绝大多数时候都能被作为参数传入 numpy

[214]:
```
s[0]
```

[214]:
```
0.07440638791818803
```

[215]:
```
# 切片方法
s[:3]
```

[215]:
```
a    0.074406
b   -0.198687
c    0.358383
dtype: float64
```

[216]:
```
#numpy 方法
np.exp(s)
```

[216]:
```
a    1.077244
b    0.819806
c    1.431014
d    2.243587
e    0.667492
dtype: float64
```

具体而言，np.exp 函数用于计算自然指数函数，即 e 的 x 次幂。这里的 e 是自然对数的底数，约等于 2.71828

-与 numpy 数组一样,Series 也具有 dtype(DataType 数据类型)

[217]:
```
s.dtype
```

[217]:
```
dtype('float64')
```

-想要返回一个不含索引的数组, 可以使用.array 方法

[218]:
```
s.array
```

[218]:
```
<PandasArray>
[ 0.07440638791818803, -0.19868708482466543,   0.3583831444457036,
   0.8080758166300783, -0.40422715197258907]
Length: 5, dtype: float64
```

-也可以变成一个 numpy 数组

[219]:
```
s.to_numpy()
```

[219]:
```
array([ 0.07440639, -0.19868708,  0.35838314,  0.80807582, -0.40422715])
```

2.5 系列 (Series) 和字典 (dict) 很像

-支持通过标签的值定位数据

[220]:
```
s['e']
```

[220]:
```
-0.40422715197258907
```

-支持判断标签是否存在于系列中

[221]:
```
'e' in s
```

[221]:
```
True
```

-遇到不存在的索引会报错

[222]:
```
s['f']
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
File E:\pythonProject4\venv\lib\site-packages\pandas\core\indexes\base.py:3653,
 ↪in Index.get_loc(self, key)
   3652 try:
-> 3653     return self._engine.get_loc(casted_key)
   3654 except KeyError as err:

File E:\pythonProject4\venv\lib\site-packages\pandas\_libs\index.pyx:147, in␣
 ↪pandas._libs.index.IndexEngine.get_loc()

File E:\pythonProject4\venv\lib\site-packages\pandas\_libs\index.pyx:176, in␣
 ↪pandas._libs.index.IndexEngine.get_loc()

File pandas\_libs\hashtable_class_helper.pxi:7080, in pandas._libs.hashtable.
 ↪PyObjectHashTable.get_item()
```

```
File pandas\_libs\hashtable_class_helper.pxi:7088, in pandas._libs.hashtable.
 ↪PyObjectHashTable.get_item()

KeyError: 'f'

The above exception was the direct cause of the following exception:

KeyError                                  Traceback (most recent call last)
Cell In[222], line 1
----> 1 s['f']

File E:\pythonProject4\venv\lib\site-packages\pandas\core\series.py:1007, in␣
 ↪Series.__getitem__(self, key)
   1004     return self._values[key]
   1006 elif key_is_scalar:
-> 1007     return self._get_value(key)
   1009 if is_hashable(key):
   1010     # Otherwise index.get_value will raise InvalidIndexError
   1011     try:
   1012         # For labels that don't resolve as scalars like tuples and␣
 ↪frozensets

File E:\pythonProject4\venv\lib\site-packages\pandas\core\series.py:1116, in␣
 ↪Series._get_value(self, label, takeable)
   1113     return self._values[label]
   1115 # Similar to Index.get_value, but we do not fall back to positional
-> 1116 loc = self.index.get_loc(label)
   1118 if is_integer(loc):
   1119     return self._values[loc]

File E:\pythonProject4\venv\lib\site-packages\pandas\core\indexes\base.py:3655,
 ↪in Index.get_loc(self, key)
   3653     return self._engine.get_loc(casted_key)
   3654 except KeyError as err:
-> 3655     raise KeyError(key) from err
   3656 except TypeError:
   3657     # If we have a listlike key, _check_indexing_error will raise
```

```
   3658     #  InvalidIndexError. Otherwise we fall through and re-raise
   3659     #  the TypeError.
   3660     self._check_indexing_error(key)

KeyError: 'f'
```

-可以使用 Series.get() 来替换缺失标签对应的返回值

[ ]:
```
s.get('f',np.nan)
```

2.6 Series 对象的矢量化操作和标签对齐功能

[ ]:
```
s+s
```

s*2

-提供自动对齐操作无需考虑参与矢量运算的两个系列是否具有相同的标签

[ ]:
```
s[1:]+s[:-1]
# 具体而言，[:-1] 表示从序列的开头（索引 0）开始，直到倒数第二个元素（索引为-2）为止，不包括最后一个元素（索引为-1）。换句话说，它会返回一个新的序列，包含原序列中除了最后一个元素之外的所有元素。
```

当进行操作时，Pandas 会自动根据标签将两个 Series 对象的元素进行对齐，即使两个 Series 的长度或索引标签不完全相同。这种自动对齐功能大大简化了数据操作的编程过程，无需手动处理对齐和循环操作。同时，如果某个标签在一个 Series 中存在而在另一个 Series 中不存在，Pandas 会将对应位置的结果标记为缺失值 NaN，以便后续处理。通常情况下，我们选择在不同索引对象之间进行操作时，默认结果是索引的并集，以避免丢失信息。即使数据缺失，拥有索引标签通常仍然是计算中的重要信息。当然，您可以通过使用 dropna 函数来删除具有缺失数据的标签，以满足特定需求。

2.7 名称属性 (Name attribute)

[ ]:
```
s = pd.Series(np.random.randn(5), name="something")
s
```

[ ]:
```
s.name
```

[ ]:
```
# 改名
s2=s.rename('different')
s2.name
```

在 Pandas 中，很多数据结构都可以通过 name 属性来设置和获取名称。该属性是一个字符串，用于标识数据结构的名称或标签。通过为数据结构设置名称，可以提供更具有描述性和可识别性的标识，方便在数据分析和处理过程中进行引用、区分和识别。

3 DataFrame

DataFrame 是一种二维标记数据结构，其中包含了可能具有不同类型的列。可以将它想象成一个电子表格或 SQL 表，或者是一组 Series 对象的字典。在创建 DataFrame 时，你可以选择性地传递行参数: 索引 (index) 和列参数:(columns)，以确保生成的 DataFrame 具有指定的索引和列。如果没有传递这些参数，pandas 将根据输入数据自动推断并构建标签。另外，如果你传递了一个具有特定索引的 Series 字典，那么不与传递的索引匹配的数据将被丢弃。## 从系列字典, 或者字典到 DataFrame 的变换生成的 DataFrame 的索引将是各个 Series 索引的并集。如果你传递了多个 Series 对象作为数据源，那么生成的 DataFrame 的索引将包括这些 Series 对象的所有索引，并且不会有重复的索引值。如果传递的数据中存在嵌套的字典 (dict)，那么这些嵌套的字典将首先被转换为 Series 对象。在创建 DataFrame 之前，嵌套的字典会被展开为对应的 Series 对象，使得 DataFrame 的结构更加扁平化。如果没有传递列参数（列标签），那么生成的 DataFrame 的列将是字典键的有序列表。换句话说，如果你没有明确指定列标签，那么 DataFrame 的列将根据字典的键来确定，并按照字典键的顺序进行排序。

[223]:
```
d={
    "one":pd.Series([1.0,2.0,3.0],index = ['a','b','c']),
    "two":pd.Series([1.0,2.0,3.0,4.0],index=['a','b','c','d'])
}
#one 和 two 变成列标签, 行标签由 index 决定
df = pd.DataFrame(d)
df
```

[223]:

	one	two
a	1.0	1.0
b	2.0	2.0
c	3.0	3.0
d	NaN	4.0

[224]:
```
pd.DataFrame(d,index=['d','b','a'])
# 由于'one'一列并没有'd'索引，所以在 one[d] 处得到的是 NAN
```

[224]:

	one	two
d	NaN	4.0
b	2.0	2.0
a	1.0	1.0

[225]:
```
pd.DataFrame(d,index=['d','b','a'],columns=['two','three'])
# 由于原先的字典系列中没有'three'这一列 所以这一列都是 NAN
```

[225]:

	two	three
d	4.0	NaN
b	2.0	NaN
a	1.0	NaN

通过访问索引（index）和列（columns）属性来获取 DataFrame 的行标签和列标签。

具体解释如下：

1. “The row and column labels can be accessed respectively by accessing the index and columns attributes.”：行标签和列标签可以通过分别访问索引（index）和列（columns）属性来获取。

这意味着在 DataFrame 对象上，可以通过以下两种方式来获取行标签和列标签：

- 行标签：通过访问 DataFrame 的 `index` 属性来获取行标签。例如，如果 `df` 是一个 DataFrame 对象，可以使用 `df.index` 来获取行标签。
- 列标签：通过访问 DataFrame 的 `columns` 属性来获取列标签。例如，如果 `df` 是一个 DataFrame 对象，可以使用 `df.columns` 来获取列标签。

通过访问这两个属性，你可以获取 DataFrame 对象的行标签和列标签，从而进行相应的操作和分析。

当在创建 DataFrame 时同时传递了一组特定的列和一个数据字典时，传递的列会覆盖字典中的键。

具体解释如下：

- 当你传递了一组特定的列和一个数据字典来创建 DataFrame 时，DataFrame 的列将由传递的列来确定。传递的列标签将覆盖数据字典中的键作为 DataFrame 的列标签。

- 换句话说，如果你在创建 DataFrame 时指定了一组特定的列，并且数据字典中的键与这些列标签不匹配，那么传递的列标签将覆盖数据字典中的键，成为最终 DataFrame 的列标签。

这个行为可以帮助你控制 DataFrame 的列标签，确保 DataFrame 的列按照你指定的顺序和命名来创建。通过传递特定的列，你可以覆盖数据字典中的键，从而自定义 DataFrame 的列标签。

[226]:
```
df.index
```

[226]:
```
Index(['a', 'b', 'c', 'd'], dtype='object')
```

[227]:
```
df.columns
```

[227]:
```
Index(['one', 'two'], dtype='object')
```

3.1 从包含 n 维数组的字典/列表到 DataFrame

请注意: 字典中的每个 n 维数组长度必须相同, 且如果传入了 `index` 参数, 则 `index` 参数的索引长度也必须要相同, 如果没有传入则默认是 `range(n)`

[228]:
```
d={
    'one':[1.0,2.0,3.0,4.0],
    'two':[4.0,3.0,2.0,1.0]
}
pd.DataFrame(d)
```

[228]:

	one	two
0	1.0	4.0
1	2.0	3.0
2	3.0	2.0
3	4.0	1.0

[229]:
```
pd.DataFrame(d,index=['a','b','c','d'])
```

[229]:

	one	two
a	1.0	4.0
b	2.0	3.0
c	3.0	2.0
d	4.0	1.0

3.2 从结构化数组 (structed array) 或者记录数组 (record array) 到 DataFrame

在 Python 中，“structured array”（结构化数组）或”record array”（记录数组）是 NumPy 库中的一个概念。

结构化数组是一种具有复合数据类型的多维数组，其中每个元素可以包含多个字段（field）或属性。每个字段可以具有不同的数据类型，例如整数、浮点数、字符串等。这使得结构化数组可以表示和处理具有表格结构的数据，类似于数据库表或电子表格。

结构化数组可以通过 NumPy 库中的 `numpy.array` 函数创建，并使用 `dtype` 参数指定每个字段的名称和数据类型。例如，下面是一个创建结构化数组的示例：

```
import numpy as np

# 定义结构化数据类型
dtype = np.dtype([('name', 'S20'), ('age', int), ('height', float)])

# 创建结构化数组
data = np.array([('John', 25, 1.75), ('Alice', 30, 1.60)], dtype=dtype)

# 访问结构化数组的字段
print(data['name'])    # 输出: ['John', 'Alice']
print(data['age'])     # 输出: [25, 30]
print(data['height'])  # 输出: [1.75, 1.6]
```

记录数组是结构化数组的一种特殊情况，其中每个字段可以被视为一个记录。记录数组可以看作是表格数据的一行，其中每个字段表示该行的一个属性或特征。

总之，结构化数组和记录数组是 NumPy 中用于处理和操作具有结构化数据的强大工具。它们提供了一种灵活和高效的方式来表示和处理多个字段的数据。

[230]:
```
data = np.zeros((2,), dtype=[("A", "i4"), ("B", "f4"), ("C", "unicode_")])
# 这段代码创建了一个包含两个元素的结构化数组（structured array），其中每个元素包含三个字段：A、B 和 C。
#
# 具体解释如下：
#
```

```
# np.zeros((2,), dtype=[("A", "i4"), ("B", "f4"), ("C", "a10")]): 这行代码使用
NumPy 库中的 zeros 函数创建一个由两个元素组成的数组，并指定数据类型为结构化数组。通
过参数 dtype，我们定义了每个元素的字段名称和数据类型。
# 其中 (2,) 指的是有 2 行但是列数未知的意思
# [("A", "i4"), ("B", "f4"), ("C", "a10")]: 这是一个 Python 列表，其中包含了结构化
数组的字段信息。每个元素都是一个二元组，第一个元素表示字段的名称，第二个元素表示字段
的数据类型。
#
# 在这段代码中，结构化数组的每个元素包含三个字段：
#
# "A": 字段名为"A"，数据类型为"i4"，表示 4 字节的整数（32 位整数）。
# "B": 字段名为"B"，数据类型为"f4"，表示 4 字节的浮点数（32 位浮点数）。
# "C": 字段名为"C"，数据类型为"a10"，表示长度为 10 的字符串（10 个字节）。
# 因此，通过这段代码创建的结构化数组 data 是一个包含两个元素的数组，每个元素都有字
段"A"、"B" 和"C"，分别对应整数、浮点数和字符串类型的数据。初始值通过 np.zeros 函数设
定为 0。你可以根据需要修改初始值或者将其用于存储和操作结构化数据。
data
```

[230]:
```
array([(0, 0., ''), (0, 0., '')],
      dtype=[('A', '<i4'), ('B', '<f4'), ('C', '<U')])
```

在 Python 中，NumPy 的 dtype 参数可以使用多种参数来指定数据类型。以下是一些常见的 dtype 参数及其对应的数据类型：

- "bool"：布尔型数据。
- "int8"、"int16"、"int32"、"int64"：带符号整数型数据，分别为 8 位、16 位、32 位和 64 位。
- "uint8"、"uint16"、"uint32"、"uint64"：无符号整数型数据，分别为 8 位、16 位、32 位和 64 位。
- "float16"、"float32"、"float64"：浮点数型数据，分别为 16 位、32 位和 64 位。
- "complex64"、"complex128"：复数型数据，分别为 64 位和 128 位。
- "string_"、"unicode_"：字符串型数据，其中 "string_" 表示字节字符串，而 "unicode_" 表示 Unicode 字符串。
- "datetime64"、"timedelta64"：日期时间型数据和时间差型数据。

除了上述常见的数据类型参数外，还可以使用其他参数来指定自定义的数据类型。例如：

- np.dtype([('name', 'S10'), ('age', 'i4'), ('height', 'f4')])：自定义结构化数据

类型，包含名为 "name"、"age" 和 "height" 的字段。

此外，还可以使用 dtype 的对象属性来指定数据类型，例如：

- np.int32：32 位整数型数据。
- np.float64：64 位浮点数型数据。
- np.bool：布尔型数据。

这只是一些常见的 dtype 参数和对应的数据类型示例。NumPy 还提供了更多的 dtype 参数选项，以满足不同的数据类型需求。你可以参考 NumPy 官方文档以获取更详细的信息：https://numpy.org/doc/stable/reference/arrays.dtypes.html

[231]:
```
data[:] = [(1, 2.0, 'hello'), (2, 3.0, "World")]
pd.DataFrame(data)
```

[231]:

	A	B	C
0	1	2.0	
1	2	3.0	

字符串那一列（第三列）多出的 b 标记实际上是由于使用了 NumPy 的结构化数组，并且将其转换为 Pandas 的 DataFrame 时引起的。

Pandas 的 DataFrame 对于结构化数组的字符串字段，会给其添加一个 b 前缀来表示该字段的数据类型为字节字符串（bytes string）。这是因为 Pandas 的 DataFrame 无法直接处理 NumPy 的字节字符串类型。

要想消除这个 b 的标记, 可以考虑如下放方式

[232]:
```
data1 = np.zeros((2,), dtype=[("A", "i4"), ("B", "f4"), ("C", "a10")])
data1[:] = [(1, 2.0, 'hello'), (2, 3.0, "World")]
df_=pd.DataFrame(data1)
df_['C']=df_['C'].str.decode('utf-8')
df_
```

[232]:

	A	B	C
0	1	2.0	hello
1	2	3.0	World

[233]:
```
# 指定行标:
pd.DataFrame(data,index=['first','second'])
```

[233]:
```
         A    B C
first    1  2.0
second   2  3.0
```

[234]:
```
# 指定列标
pd.DataFrame(data, columns=["C", "A", "B"])
```

[234]:
```
   C  A    B
0     1  2.0
1     2  3.0
```

4 从列表字典到 DataFrame

[235]:
```
data2 = [{'a':1,'b':2},{'a':5,'b':10,"c":20}]
pd.DataFrame(data2)
```

[235]:
```
    a   b     c
0   1   2   NaN
1   5  10  20.0
```

[236]:
```
pd.DataFrame(data2,index=['first','second']) # 指定列标签
```

[236]:
```
         a   b     c
first    1   2   NaN
second   5  10  20.0
```

[237]:
```
pd.DataFrame(data2, columns=["a", "b"])
```

[237]:
```
    a   b
0   1   2
1   5  10
```

5 从字典元祖到 DataFrame

[238]:
```
pd.DataFrame(
    {
        ("a", "b"): {("A", "B"): 1, ("A", "C"): 2},
        ("a", "a"): {("A", "C"): 3, ("A", "B"): 4},
        ("a", "c"): {("A", "B"): 5, ("A", "C"): 6},
        ("b", "a"): {("A", "C"): 7, ("A", "B"): 8},
        ("b", "b"): {("A", "D"): 9, ("A", "B"): 10},
    }
)
```

[238]:

		a			b	
		b	a	c	a	b
A	B	1.0	4.0	5.0	8.0	10.0
	C	2.0	3.0	6.0	7.0	NaN
	D	NaN	NaN	NaN	NaN	9.0

这段代码创建了一个 Pandas DataFrame，其中包含一个多级列索引。DataFrame 的数据由一个字典提供，字典的键是多级列索引，字典的值是相应的数据。

让我们逐步解释这段代码的输出：这段代码中的字典有 5 个键值对，每个键值对对应一个多级列索引和相应的数据。让我们逐个解释每个键值对：

1. `("a", "b")`: 键 `("a", "b")` 对应的值是一个字典 `{("A", "B"): 1, ("A", "C"): 2}`。这表示 DataFrame 的第一列有两个子列标签，分别是 `"a"` 和 `"b"`。在子列标签 `"a"` 下，行索引为 `("A", "B")` 的单元格的值为 1，在行索引为 `("A", "C")` 的单元格的值为 2。

2. `("a", "a")`: 键 `("a", "a")` 对应的值是一个字典 `{("A", "C"): 3, ("A", "B"): 4}`。这表示 DataFrame 的第二列有两个子列标签，分别是 `"a"` 和 `"a"`。在子列标签 `"a"` 下，行索引为 `("A", "C")` 的单元格的值为 3，在行索引为 `("A", "B")` 的单元格的值为 4。

3. `("a", "c")`: 键 `("a", "c")` 对应的值是一个字典 `{("A", "B"): 5, ("A", "C"): 6}`。这表示 DataFrame 的第三列有两个子列标签，分别是 `"a"` 和 `"c"`。在子列标签 `"a"` 下，行索引为 `("A", "B")` 的单元格的值为 5，在行索引为 `("A", "C")` 的单元格的值为 6。

4. `("b", "a")`: 键 `("b", "a")` 对应的值是一个字典 `{("A", "C"): 7, ("A", "B"): 8}`。这表示 DataFrame 的第四列有两个子列标签，分别是 `"b"` 和 `"a"`。在子列标签 `"b"` 下，行索引为 `("A", "C")` 的单元格的值为 7，在行索引为 `("A", "B")` 的单元格的值为 8。

5. ("b", "b"): 键 ("b", "b") 对应的值是一个字典 {("A", "D"): 9, ("A", "B"): 10}。这表示 DataFrame 的第五列有两个子列标签，分别是 "b" 和 "b"。在子列标签 "b" 下，行索引为 ("A", "D") 的单元格的值为 9，在行索引为 ("A", "B") 的单元格的值为 10。

因此，这段代码创建的 DataFrame 具有多级列索引，并且每个单元格的值由相应的行索引和列索引决定。**需要注意的是，这一切都是自动完成的**

5.1 从系列 Series 到 DataFrame

当将一个 Series 转换为 DataFrame 时，结果 DataFrame 将具有与原始 Series 相同的索引。如果没有提供其他列名，则结果 DataFrame 将只有一个列，该列的名称将是原始 Series 的名称

[239]:
```
ser = pd.Series(range(3),index=list('abc'),name='str')
pd.DataFrame(ser)
```

[239]:

	str
a	0
b	1
c	2

5.2 从具名元祖 (namedtuple) 到 DataFrame

namedtuple 是 Python 中的一个数据类型，它是一个具名元组（named tuple）。元组是一个不可变的序列，而具名元组是为元组中的每个元素赋予了名称，使得可以通过名称来访问元组的元素，而不仅仅是通过索引。具名元组的定义使用 collections 模块中的 namedtuple 函数。它接受两个参数：元组的名称和元素的名称列表。例如，下面是一个具名元组的定义：

```
from collections import namedtuple
Person = namedtuple('Person', ['name', 'age', 'gender'])
```

上面的代码定义了一个名为 Person 的具名元组，它有三个字段：name、age 和 gender。可以使用这个具名元组创建新的实例：

```
person = Person('Alice', 25, 'Female')
```

可以通过字段名来访问具名元组的元素：

```
print(person.name)    # 输出: 'Alice'
print(person.age)     # 输出: 25
print(person.gender)  # 输出: 'Female'
```

具名元组提供了一个更加可读和易于理解的方式来处理元组数据。它们可以替代使用索引访问元组元素的方式，使代码更具可读性和可维护性。此外，具名元组的字段名是不可变的，因此可以确保数据的一致性。需要注意的是，具名元组实际上是普通的元组的子类，因此它们保留了元组的不可变性和性能优势。同时，具名元组也继承了元组的方法，如索引、切片和迭代等。总结起来，具名元组是一种带有字段名的不可变数据结构，用于创建具有命名字段的元组。它们提供了一种更加可读和易于理解的方式来处理元组数据，同时保留了元组的性能优势。**使用具名元组列表创建 DataFrame 的规则：第一个具名元组的字段名确定了列名，后续的具名元组会按行填充 DataFrame，如果某个具名元组比第一个具名元组短，则后面的列将被标记为缺失值；如果某个具名元组比第一个具名元组长，则会引发 ValueError 错误。**

[240]:
```
from collections import namedtuple
Point = namedtuple('Point',['x','y'])
pd.DataFrame([Point(0,0),Point(0,3),Point(2,3)])
```

[240]:

	x	y
0	0	0
1	0	3
2	2	3

[241]:
```
# 标记缺失值的范例
Point3D=namedtuple('Point3D',['x','y','z'])
pd.DataFrame([Point3D(0,0,0),Point3D(0,3,5),Point(2,3)])
```

[241]:

	x	y	z
0	0	0	0.0
1	0	3	5.0
2	2	3	NaN

5.3 从 dataclasses 到 DataFrame

[242]:
```
from dataclasses import make_dataclass
Point = make_dataclass('Point',[('x',int),('y',int)])
pd.DataFrame([Point(0,0),Point(0,3),Point(2,3)])
```

[242]:

	x	y
0	0	0
1	0	3

```
2  2  3
```

5.4 别的构造器

DataFrame.from_dict()

DataFrame.from_dict() 方法接受一个字典的字典（dict of dicts）或者一个字典的数组样式序列（dict of array-like sequences），并返回一个 DataFrame。它的用法类似于 DataFrame 构造函数，除了 orient 参数的默认值是 'columns'，但可以设置为 'index'，以使用字典的键作为行标签。

[243]:
```
pd.DataFrame.from_dict(dict([("A", [1, 2, 3]), ("B", [4, 5, 6])]))
```

[243]:
```
   A  B
0  1  4
1  2  5
2  3  6
```

如果传入了 orient = "index", 那么字典的键会被作为行标签, 否则默认是列标签

[244]:
```
pd.DataFrame.from_dict(
    dict(
        [
            ('A',[1,2,3]),
            ('B',[4,5,6]),
        ]
    ),
    orient="index",
    columns=['one','two','three'],
)
```

[244]:
```
   one  two  three
A    1    2      3
B    4    5      6
```

DataFrame.from_records DataFrame.from_records() 方法接受一个元组列表或带有结构化数据类型（structured dtype）的 ndarray，并返回一个 DataFrame。它的使用方式类似于普通的 DataFrame 构造函数，但不同之处在于生成的 DataFrame 的索引可以是结构化数据类型的特定字段。

[245]:
```
data
```

[245]:
```
array([(1, 2., ''), (2, 3., '')],
      dtype=[('A', '<i4'), ('B', '<f4'), ('C', '<U')])
```

[246]:
```
pd.DataFrame.from_records(data,index='C')
```

[246]:
```
   A    B
C
   1  2.0
   2  3.0
```

5.5 选择列, 增加列, 删除列

在语义上，你可以将 DataFrame 看作是一个类似于以相同索引为键的 Series 对象的字典。获取、设置和删除列的操作可以使用与类似的字典操作相同的语法。

[247]:
```
df['one']
```

[247]:
```
a    1.0
b    2.0
c    3.0
d    NaN
Name: one, dtype: float64
```

[248]:
```
df['three']=df["one"]*df['two']
df['flag']=df['one']>2
df
```

[248]:
```
   one  two  three   flag
a  1.0  1.0    1.0  False
b  2.0  2.0    4.0  False
c  3.0  3.0    9.0   True
d  NaN  4.0    NaN  False
```

可以像字典一样删除或者弹出元素

[249]:
```
del df['two']
three = df.pop('three')
df
```

[249]:

	one	flag
a	1.0	False
b	2.0	False
c	3.0	True
d	NaN	False

插入标量时候, 默认会填充全部的行

[250]:
```
df['foo']='bar'
df
```

[250]:

	one	flag	foo
a	1.0	False	bar
b	2.0	False	bar
c	3.0	True	bar
d	NaN	False	bar

当插入一个索引与 DataFrame 不相同的 Series 时，Series 的索引会被调整为与 DataFrame 的索引相匹配。

[251]:
```
df["one_trunc"] = df["one"][:2]
df
```

[251]:

	one	flag	foo	one_trunc
a	1.0	False	bar	1.0
b	2.0	False	bar	2.0
c	3.0	True	bar	NaN
d	NaN	False	bar	NaN

df.insert() column: 必需参数，表示要插入的列的名称。新的列将被插入到该名称指定的位置。value: 必需参数，表示要插入的列的值。可以是单个值、列表、数组或 Series 对象。如果值是单个值，则该值将被广播到整个列。如果值是列表、数组或 Series 对象，则其长度必须与 DataFrame 的行数相匹配。loc: 必需参数，表示要插入新列的位置。可以指定列的索引位置（整数值），也可以指定列的名称。

[252]:
```
df.insert(column='bar',value=df['one'],loc=1)
df
```

[252]:

	one	bar	flag	foo	one_trunc
a	1.0	1.0	False	bar	1.0
b	2.0	2.0	False	bar	2.0
c	3.0	3.0	True	bar	NaN
d	NaN	NaN	False	bar	NaN

5.6 索引/选择

操作	语法	返回值
选择列	df[col]	系列 Series
用标签选择行	df.loc(label)	系列 Series
用整数选择行	df.iloc[loc]	系列 Series
为行切片	df[5:10]	DataFrame
用布尔向量选择行	df[bool_vec]	DataFrame

5.7 数据对齐和算术运算

数据对齐确保正在操作的数据结构具有匹配的标签或索引，以便正确执行操作。当执行加法、减法、乘法或除法等算术运算时，数据对齐允许操作应用于基于标签或索引相匹配的元素

[253]:
```
df=pd.DataFrame(np.random.randn(10,4),columns=['A','B','C','D'])
df2 = pd.DataFrame(np.random.randn(7,3),columns=['A','B','C'])
df+df2
# 这段代码创建了两个 DataFrame 对象`df`和`df2`，并对它们进行了加法运算。
#
# 首先，通过`pd.DataFrame()`函数创建了一个包含 10 行 4 列的 DataFrame 对象`df`，其中的数据是从标准正态分布中随机生成的。每一列被命名为'A'、'B'、'C'和'D'。
#
# 接下来，使用`pd.DataFrame()`函数创建了一个包含 7 行 3 列的 DataFrame 对象`df2`，其中的数据也是从标准正态分布中随机生成的。每一列被命名为'A'、'B'和'C'。
#
```

```
# 然后，代码执行了`df + df2`的加法运算。在这个运算中，pandas 会自动对齐两个 DataFrame 对象的数据。由于`df`和`df2`具有相同的列名（'A'、'B'和'C'），它们的对应列将进行元素级别的相加运算。对于没有匹配的列（'D'和没有对应的列），相应的结果将为 NaN。
#
# 最终的结果是一个新的 DataFrame 对象，它具有与原始 DataFrame 对象相同的行数和列数，其中的值是根据对应元素相加的结果。如果某个位置在原始 DataFrame 中或者 df2 中没有对应的值，则结果中相应位置的值将为 NaN。
#
# 简而言之，这段代码演示了如何对两个 DataFrame 对象进行加法运算，并展示了数据对齐的结果。只有具有相同列名的列才会相互对齐并进行相加运算，其他列则被视为不匹配的列并产生 NaN 值。
```

[253]:

	A	B	C	D
0	-1.134323	-0.295077	0.461890	NaN
1	-0.267033	2.564926	-0.396469	NaN
2	-0.429033	-0.876337	0.920716	NaN
3	0.023357	1.755694	1.001540	NaN
4	-1.101023	-0.474365	-2.143047	NaN
5	-0.954898	1.684914	1.437906	NaN
6	1.950669	2.170750	-0.087492	NaN
7	NaN	NaN	NaN	NaN
8	NaN	NaN	NaN	NaN
9	NaN	NaN	NaN	NaN

当在 DataFrame 和 Series 之间进行操作时，pandas 会根据索引对齐数据。对于 Series 对象，它具有自己的索引，而 DataFrame 对象具有列索引。在默认情况下，pandas 会将 Series 的索引与 DataFrame 的列进行对齐，以确保操作的正确性。

具体而言，对于每个 Series 的索引，pandas 会查找与之对应的 DataFrame 的列索引。然后，将 Series 的值按行广播到对应的 DataFrame 行上。这意味着 Series 的值将被复制到 DataFrame 的每一行，以便与对应的列进行运算。

这种行为称为广播（broadcasting），因为 Series 的值会被广播到整个 DataFrame 的行上。通过这种方式，可以在 DataFrame 和 Series 之间进行基于行的操作，而不需要显式地编写循环或逐行操作。

[254]:
```
df
```

[254]:

	A	B	C	D
0	-0.272496	-0.352829	0.463889	0.712193
1	2.084525	2.585520	-0.168173	-1.564562
2	0.202141	-0.338019	-0.178463	0.998497
3	-1.254657	0.450336	1.593341	0.151471
4	-1.694684	-0.084137	-0.978055	-0.212781
5	-0.448056	0.069133	1.131621	-1.555782
6	0.405340	0.726877	-0.421899	-0.345312
7	1.565012	0.010758	-0.257429	-0.157207
8	-0.137183	1.312388	0.726485	0.390261
9	0.911358	0.915824	-0.137987	-1.159776

[255]:
```
df.iloc[0]
```

[255]:
```
A   -0.272496
B   -0.352829
C    0.463889
D    0.712193
Name: 0, dtype: float64
```

[256]:
```
df-df.iloc[0]
```

[256]:

	A	B	C	D
0	0.000000	0.000000	0.000000	0.000000
1	2.357021	2.938349	-0.632061	-2.276755
2	0.474637	0.014810	-0.642352	0.286304
3	-0.982162	0.803164	1.129453	-0.560722
4	-1.422188	0.268692	-1.441943	-0.924975
5	-0.175560	0.421962	0.667732	-2.267976
6	0.677836	1.079706	-0.885788	-1.057505
7	1.837508	0.363587	-0.721318	-0.869400
8	0.135313	1.665217	0.262596	-0.321932
9	1.183854	1.268653	-0.601876	-1.871969

布尔类型的元素也是可以参与运算的

[257]:
```
df1 = pd.DataFrame({"a": [1, 0, 1], "b": [0, 1, 1]}, dtype=bool)
```

```
df2 = pd.DataFrame({"a": [0, 1, 1], "b": [1, 1, 0]}, dtype=bool)

df1 & df2
```

[257]:

	a	b
0	False	False
1	False	True
2	True	False

[258]:
```
df1|df2
```

[258]:

	a	b
0	True	True
1	True	True
2	True	True

5.8 转置

[259]:
```
df
```

[259]:

	A	B	C	D
0	-0.272496	-0.352829	0.463889	0.712193
1	2.084525	2.585520	-0.168173	-1.564562
2	0.202141	-0.338019	-0.178463	0.998497
3	-1.254657	0.450336	1.593341	0.151471
4	-1.694684	-0.084137	-0.978055	-0.212781
5	-0.448056	0.069133	1.131621	-1.555782
6	0.405340	0.726877	-0.421899	-0.345312
7	1.565012	0.010758	-0.257429	-0.157207
8	-0.137183	1.312388	0.726485	0.390261
9	0.911358	0.915824	-0.137987	-1.159776

[260]:
```
df[:5].T
```

[260]:

	0	1	2	3	4
A	-0.272496	2.084525	0.202141	-1.254657	-1.694684
B	-0.352829	2.585520	-0.338019	0.450336	-0.084137

```
C  0.463889 -0.168173 -0.178463  1.593341 -0.978055
D  0.712193 -1.564562  0.998497  0.151471 -0.212781
```

5.9 和 numpy 的函数互相操作

[261]:
```
np.exp(df)
```

[261]:

	A	B	C	D
0	0.761477	0.702697	1.590246	2.038457
1	8.040772	13.270190	0.845208	0.209180
2	1.224021	0.713182	0.836555	2.714199
3	0.285174	1.568839	4.920161	1.163545
4	0.183657	0.919305	0.376042	0.808333
5	0.638869	1.071579	3.100678	0.211024
6	1.499812	2.068611	0.655800	0.708000
7	4.782732	1.010817	0.773036	0.854527
8	0.871811	3.715034	2.067799	1.477367
9	2.487699	2.498833	0.871110	0.313556

[262]:
```
#np.asarray() 是 NumPy 库中的一个函数，用于将输入（input）转换为 NumPy 数组。
np.asarray(df)
```

[262]:
```
array([[-0.27249582, -0.35282883,  0.46388876,  0.71219336],
       [ 2.08452509,  2.58552015, -0.1681726 , -1.56456177],
       [ 0.20214141, -0.33801926, -0.17846347,  0.99849688],
       [-1.25465733,  0.45033567,  1.59334131,  0.15147109],
       [-1.6946836 , -0.08413731, -0.97805467, -0.21278134],
       [-0.4480559 ,  0.06913306,  1.13162068, -1.55578245],
       [ 0.40533997,  0.72687723, -0.42189944, -0.34531171],
       [ 1.56501202,  0.01075842, -0.25742944, -0.15720679],
       [-0.13718306,  1.31238779,  0.72648476,  0.39026138],
       [ 0.9113582 ,  0.91582391, -0.1379874 , -1.15977592]])
```

DataFrame 不是作为 ndarray 的替代品而设计的，因为它的索引语义和数据模型在某些方面与 n 维数组有很大的区别。这意味着 DataFrame 和 ndarray 之间存在一些功能和行为上的差异。

Series 实现了 __array_ufunc__，这使得它可以与 NumPy 的通用函数（universal

functions--ufunc）一起使用。通用函数是一组对数组执行元素级操作的函数，例如加法、乘法、平方根等。

在 Series 中，ufunc 被应用于底层的数组。也就是说，Series 对象中的数据存储在一个数组中，而 ufunc 将直接作用于这个数组上。

[263]:
```
ser = pd.Series([1,2,3,4])
ser
```

[263]:
```
0    1
1    2
2    3
3    4
dtype: int64
```

[264]:
```
np.exp(ser)
```

[264]:
```
0     2.718282
1     7.389056
2    20.085537
3    54.598150
dtype: float64
```

[265]:
```
ser# 可见并不会影响其本身
```

[265]:
```
0    1
1    2
2    3
3    4
dtype: int64
```

当多个 Series 对象被传递给一个 ufunc 函数时，在执行操作之前会进行对齐（alignment）。

类似于库中的其他部分，pandas 会自动对标签化的输入进行对齐，作为 ufunc 函数的多个输入的一部分。例如，在两个具有不同排序标签的 Series 上使用 numpy.remainder() 函数时，它们会在执行操作之前进行对齐。**注：** numpy.remainder() 是 NumPy 库中的一个函数，用于计算两个数组的元素级别取模（求余）运算。

该函数接受两个参数，分别是被除数数组 x1 和除数数组 x2。它返回一个新的数组，其中的每个元素都是将 x1 中对应元素除以 x2 中对应元素后的余数。

以下是 numpy.remainder() 函数的示例：

```
import numpy as np

x1 = np.array([10, 20, 30])
x2 = np.array([3, 7, 4])
result = np.remainder(x1, x2)

print(result)
```

输出：

```
[1 6 2]
```

在上面的示例中，我们将两个 NumPy 数组 x1 和 x2 传递给 numpy.remainder() 函数，它返回一个新的数组 result，其中包含 x1 对应元素除以 x2 对应元素后的余数。因此，10 除以 3 的余数为 1，20 除以 7 的余数为 6，30 除以 4 的余数为 2。

numpy.remainder() 函数在数值计算中经常用于执行取模运算。它可以应用于整数数组、浮点数数组以及其他支持元素级别计算的 NumPy 数组类型。需要注意的是，当除数为 0 时，函数将返回 0 作为结果，而不会引发除以零的异常。

[266]:
```
ser1 = pd.Series([1, 2, 3], index=["a", "b", "c"])
ser2 = pd.Series([1, 3, 5], index=["b", "a", "c"])
ser1
```

[266]:
```
a    1
b    2
c    3
dtype: int64
```

[267]:
```
ser2
```

[267]:
```
b    1
a    3
c    5
dtype: int64
```

[268]:
```
np.remainder(ser1,ser2)
```

[268]:
```
a    1
b    0
c    3
dtype: int64
```

通常情况下，将两个索引的并集取出，并且对于不重叠的值，使用缺失值进行填充。

[269]:
```
ser3 = pd.Series([2, 4, 6], index=["b", "c", "d"])
ser3
```

[269]:
```
b    2
c    4
d    6
dtype: int64
```

[270]:
```
np.remainder(ser1,ser3)
```

[270]:
```
a    NaN
b    0.0
c    3.0
d    NaN
dtype: float64
```

当将一个二元 ufunc（二元通用函数, 如: 加法, 减法）应用于一个 Series 和一个 Index 时，Series 的实现方式优先，并且返回一个 Series 对象。

[271]:
```
ser = pd.Series([1, 2, 3])
idx = pd.Index([4,5,6])#pd.Index 是 pandas 库中用于创建索引对象的函数。它返回一个 Index 对象，用于表示一维标签化的索引。
np.maximum(ser,idx)
```

[271]:
```
0    4
1    5
2    6
dtype: int64
```

输出结果是一个新的 Series 对象，其中的每个元素是 ser 和 idx **对应位置**元素的最大值。在这个例子中，比较的结果是 1 和 4 的最大值（即 4），2 和 5 的最大值（即 5），以及 3 和 6 的最大值（即 6）。

5.10 控制台显示

简单的说, 对于过大的数据, 你可以使用.info() 函数来查看其概要 ## DataFrame 列属性访问 DataFrame 列属性访问（DataFrame column attribute access）：指的是使用属性访问方式来访问 DataFrame 中的列。在 pandas 中，DataFrame 是一个二维的带标签的数据结构，每个列都有一个唯一的标签或名称。通过列属性访问，可以使用点符号（.）和列名来访问 DataFrame 中的列数据。

[272]:
```
df = pd.DataFrame({"foo1": np.random.randn(5), "foo2": np.random.randn(5)})
df
```

[272]:
```
       foo1      foo2
0  0.604855  1.568471
1  0.567668 -1.111407
2 -0.575184  0.908364
3 -1.083471  0.519498
4 -0.430405 -0.427078
```

[273]:
```
df.foo1
```

[273]:
```
0    0.604855
1    0.567668
2   -0.575184
3   -1.083471
4   -0.430405
Name: foo1, dtype: float64
```

[273]:
```

```