< 在Pandas中操作数据 | 目录 | 层次化的索引 >

Open in Colab

Handling Missing Data

处理缺失数据

> The difference between data found in many tutorials and data in the real world is that real-world data is rarely clean and homogeneous. In particular, many interesting datasets will have some amount of data missing. To make matters even more complicated, different data sources may indicate missing data in different ways.

我们在许多教程里面看到的数据和真实的数据的区别就是真实的数据很少是干净和同质的。更寻常的情况是，很多有意思的数据集都有很多的数据缺失。更复杂的是，不同的数据源可能有着不同指代缺失数据的方式。

> In this section, we will discuss some general considerations for missing data, discuss how Pandas chooses to represent it, and demonstrate some built-in Pandas tools for handling missing data in Python. Here and throughout the book, we'll refer to missing data in general as *null*, *NaN*, or *NA* values.

在本节中，我们会讨论一些对于缺失数据的通用处理方式，介绍Pandas如何选择和表示这些数据，展示Pandas中用来处理缺失数据的內建工具。本节和本书其他部分，我们会将这些缺失数据标示为*null*、*NaN*或*NA*。

Trade-Offs in Missing Data Conventions

缺失数据约定的权衡

> There are a number of schemes that have been developed to indicate the presence of missing data in a table or DataFrame. Generally, they revolve around one of two strategies: using a *mask* that globally indicates missing values, or choosing a *sentinel value* that indicates a missing entry.

用来在数据表或DataFrame中指定和标示缺失数据的方案有很多种。通常来说，会有两种主要的策略：使用一个全局的*遮盖*来标示缺失数据，或者选择使用*哨兵值*来标示缺失的元素。

> In the masking approach, the mask might be an entirely separate Boolean array, or it may involve appropriation of one bit in the data representation to locally indicate the null status of a value.

在遮盖方案中，遮盖层可以是一整个独立的布尔数组，又或者可以在数据中使用一个比特标示空值。

> In the sentinel approach, the sentinel value could be some data-specific convention, such as indicating a missing integer value with -9999 or some rare bit pattern, or it could be a more global convention, such as indicating a missing floating-point value with NaN (Not a Number), a special value which is part of the IEEE floating-point specification.

在哨兵值的情况下，哨兵值是某种数据特定的约定值，例如用-9999标示一个缺失的整数或者其他罕见的数值，又或者使用更加通用的方式，比方说标示一个缺失的浮点数为NaN（非数字），NaN是IEEE浮点数标准中的一部分。

> None of these approaches is without trade-offs: use of a separate mask array requires allocation of an additional Boolean array, which adds overhead in both storage and computation. A sentinel value reduces the range of valid values that can be represented, and may require extra (often non-optimized) logic in CPU and GPU arithmetic. Common special values like NaN are not available for all data types.

以上解决方案都是有所取舍的：独立的遮盖数组需要更多的内存空间用于存储布尔数组；普通的哨兵值会缩小正确数据的取值范围，而且需要额外的（通常是未优化的）CPU和GPU运算；通用的特殊值如NaN又无法应用于所有的数据类型上。

> As in most cases where no universally optimal choice exists, different languages and systems use different conventions. For example, the R language uses reserved bit patterns within each data type as sentinel values indicating missing data, while the SciDB system uses an extra byte attached to every cell which indicates a NA state.

因为在大多数情况下并不存在普适的优选策略，因此不同的语言和系统都会选择使用不同的约定。例如，R语言用户在每个数据类型中保留一个比特位作为哨兵值来标示缺失数据，而SciDB系统的用户使用一个额外的字节绑定在每个元素值上用于标示不可用的情况。

Missing Data in Pandas

Pandas中的缺失值

> The way in which Pandas handles missing values is constrained by its reliance on the NumPy package, which does not have a built-in notion of NA values for non-floating-point data types.

Pandas中用来处理缺失值的方式取决于它依赖的NumPy包，因此对于非浮点数类型不存在內建的缺失值标志。

> Pandas could have followed R's lead in specifying bit patterns for each individual data type to indicate nullness, but this approach turns out to be rather unwieldy. While R contains four basic data types, NumPy supports *far* more than this: for example, while R has a single integer type, NumPy supports *fourteen* basic integer types once you account for available precisions, signedness, and endianness of the encoding. Reserving a specific bit pattern in all available NumPy types would lead to an unwieldy amount of overhead in special-casing various operations for various types, likely even requiring a new fork of the NumPy package. Further, for the smaller data types (such as 8-bit integers), sacrificing a bit to use as a mask will significantly reduce the range of values it can represent.

Pandas可以采用R语言的方式，即在数据值中指定一个比特位为缺失值标志，但是这种方案实现起来显得很笨重。因为R只有4中基本数据类型，而NumPy支持的类型却*远超*这个数：例如，R只有1中整数类型，NumPy却支持*14*种不同精度、是否带符号、大小尾编码的整数类型。保留一个比特位作为缺失值的标志，会影响到NumPy的所有类型的很多不同的操作，基本上等同于需要一整套新的NumPy包来支持新的操作。并且在数据类型比较小的情况下（例如8比特的整数），这种做法会严重缩小数据类型可以表达的数值的范围。

> NumPy does have support for masked arrays – that is, arrays that have a separate Boolean mask array attached for marking data as "good" or "bad." Pandas could have derived from this, but the overhead in both storage, computation, and code maintenance makes that an unattractive choice.

NumPy当然支持遮盖数组，即一个数组包含着分散的布尔数值用来标示数据是“好的”还是“坏的”。Pandas当然也继承了这一点，但是存储、计算和代码维护方面的额外需求也使得这种方案不是特别吸引人。

> With these constraints in mind, Pandas chose to use sentinels for missing data, and further chose to use two already-existing Python null values: the special floating-point `NaN` value, and the Python `None` object. This choice has some side effects, as we will see, but in practice ends up being a good compromise in most cases of interest.

因此，Pandas选择了最后一种方案，即通用哨兵值标示缺失值。更进一步说就是，使用两个已经存在的Python空值：`NaN`代表特殊的浮点数值和Python的`None`对象。这种做法当然也有一些副作用，我们后面也会看到，但是在实践中它被证明在大多数情况下都是一个较好的折中方案。

`None`: Pythonic missing data

`None`：Python的缺失值

> The first sentinel value used by Pandas is `None`, a Python singleton object that is often used for missing data in Python code. Because it is a Python object, `None` cannot be used in any arbitrary NumPy/Pandas array, but only in arrays with data type `'object'` (i.e., arrays of Python objects):

第一个被Pandas使用的缺失哨兵值是`None`，它是一个Python的单例对象，很多情况下它都作为Python代码中缺失值的标志。因为这是一个Python对象，`None`不能在任意的NumPy或Pandas数组中使用，它只能在数组的数据类型是`object`的情况下使用（例如，Python对象组成的数组）：

In [1]:

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

In [2]:

```
vals1 = np.array([1, None, 3, 4])
vals1
```

Out[2]:

```
array([1, None, 3, 4], dtype=object)
```

> This `dtype=object` means that the best common type representation NumPy could infer for the contents of the array is that they are Python objects. While this kind of object array is useful for some purposes, any operations on the data will be done at the Python level, with much more overhead than the typically fast operations seen for arrays with native types:

这里的`dtype=object`表示这个NumPy数组的元素类型是Python的对象。虽然这种类型的对象数组在某些场景中很有用，任何数据的操作都会在Python层面进行，这会比NumPy其他基础类型进行的快速操作消耗更多的执行时间：

In [3]:

```
for dtype in ['object', 'int']:
    print("dtype =", dtype)
    %timeit np.arange(1E6, dtype=dtype).sum()
    print()
```

```
dtype = object
106 ms ± 645 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)

dtype = int
2.55 ms ± 8.77 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

> The use of Python objects in an array also means that if you perform aggregations like `sum()` or `min()` across an array with a `None` value, you will generally get an error:

而且使用Python对象作为数组数据类型的话，当使用聚合操作如`sum()`或`min()`的时候，如果碰到了`None`值，那就会产生一个错误：

In [4]:

```
vals1.sum()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-4-30a3fc8c6726> in <module>
----> 1 vals1.sum()

~/anaconda3/lib/python3.7/site-packages/numpy/core/_methods.py in _sum(a, axis, dtype, out, keepdims,
 initial)
     34 def _sum(a, axis=None, dtype=None, out=None, keepdims=False,
     35          initial=_NoValue):
---> 36     return umr_sum(a, axis, dtype, out, keepdims, initial)
     37 
     38 def _prod(a, axis=None, dtype=None, out=None, keepdims=False,

TypeError: unsupported operand type(s) for +: 'int' and 'NoneType'
```

> This reflects the fact that addition between an integer and `None` is undefined.

错误的原因是整数和`None`对象之间进行加法运算是未定义的。

`NaN`: Missing numerical data

`NaN`：缺失的数值类型数据

> The other missing data representation, `NaN` (acronym for *Not a Number*), is different; it is a special floating-point value recognized by all systems that use the standard IEEE floating-point representation:

另外一个缺失的数据表现形式`NaN`（*非数字*的缩写），能被所有支持IEEE浮点数标准的系统所识别：

In [5]:

```
vals2 = np.array([1, np.nan, 3, 4])
vals2.dtype
```

Out[5]:

```
dtype('float64')
```

> Notice that NumPy chose a native floating-point type for this array: this means that unlike the object array from before, this array supports fast operations pushed into compiled code. You should be aware that `NaN` is a bit like a data virus–it infects any other object it touches. Regardless of the operation, the result of arithmetic with `NaN` will be another `NaN`:

NumPy使用原始的浮点类型来存储这个数组：这意味着不像前面的对象数组，这个数组支持使用编译代码来进行快速运算。你应该了解到`NaN`就像一个数据的病毒，它会传染到任何接触到的数据。不论运算是哪种类型，`NaN`参与的算术运算的结果都会是另一个`NaN`：

In [6]:

```
1 + np.nan
```

Out[6]:

```
nan
```

In [7]:

```
0 *  np.nan
```

Out[7]:

```
nan
```

> Note that this means that aggregates over the values are well defined (i.e., they don't result in an error) but not always useful:

因此对于这个数组进行的聚合操作是良好定义的（意思是不会发生错误），但是却并不十分有意义：

In [8]:

```
vals2.sum(), vals2.min(), vals2.max()
```

Out[8]:

```
(nan, nan, nan)
```

> NumPy does provide some special aggregations that will ignore these missing values:

NumPy还提供了一些特殊的聚合函数可以用来忽略这些缺失值：

In [9]:

```
np.nansum(vals2), np.nanmin(vals2), np.nanmax(vals2)
```

Out[9]:

```
(8.0, 1.0, 4.0)
```

> Keep in mind that `NaN` is specifically a floating-point value; there is no equivalent NaN value for integers, strings, or other types.

请记住`NaN`是一个特殊的浮点数值；对于整数、字符串或者其他类型来说都没有对应的值。

NaN and None in Pandas

Pandas中的NaN和None

> `NaN` and `None` both have their place, and Pandas is built to handle the two of them nearly interchangeably, converting between them where appropriate:

`NaN`和`None`在Pandas都可以使用，而且Pandas基本上将两者进行等同处理，可以在合适的情况下互相转换：

In [10]:

```
pd.Series([1, np.nan, 2, None])
```

Out[10]:

```
0    1.0
1    NaN
2    2.0
3    NaN
dtype: float64
```

> For types that don't have an available sentinel value, Pandas automatically type-casts when NA values are present. For example, if we set a value in an integer array to `np.nan`, it will automatically be upcast to a floating-point type to accommodate the NA:

对于哪些没有通用哨兵值的类型，Pandas在发现出现了NA值的情况下会自动对它们进行类型转换。例如，如果我们在一个整数数组中设置了一个`np.nan`值，整个数组会自动向上扩展为浮点类型：

In [11]:

```
x = pd.Series(range(2), dtype=int)
x
```

Out[11]:

```
0    0
1    1
dtype: int64
```

In [12]:

```
x[0] = None
x
```

Out[12]:

```
0    NaN
1    1.0
dtype: float64
```

> Notice that in addition to casting the integer array to floating point, Pandas automatically converts the `None` to a `NaN` value. (Be aware that there is a proposal to add a native integer NA to Pandas in the future; as of this writing, it has not been included).

上述例子中除了将整数类型转换为浮点数类型之外，Pandas还自动将`None`转换成了`NaN`值。（在本文写的时候，有一个提议在Pandas的整数类型中加入一个NA值，不过还没有被采纳）。

> While this type of magic may feel a bit hackish compared to the more unified approach to NA values in domain-specific languages like R, the Pandas sentinel/casting approach works quite well in practice and in my experience only rarely causes issues.

虽然这种解决方案对比起类似R语言那样使用统一的NA值来标示的方案来说，显得有点像魔术。但是Pandas的这种哨兵+类型转换的方式在实践中运行良好，而且在作者的经验中，很少导致问题。

> The following table lists the upcasting conventions in Pandas when NA values are introduced:

下表列出了Pandas在出现NA值的时候向上类型扩展的规则：

大类型	当NA值存在时转换规则	NA哨兵值
浮点数	保持不变	`np.nan`
`object`	保持不变	`None` 或 `np.nan`
整数	转换为 `float64`	`np.nan`
布尔	转换为 `object`	`None` 或 `np.nan`

> Keep in mind that in Pandas, string data is always stored with an `object` dtype.

在Pandas中，字符串数据总是使用`object`类型存储的。

Operating on Null Values

操作空值

> As we have seen, Pandas treats `None` and `NaN` as essentially interchangeable for indicating missing or null values. To facilitate this convention, there are several useful methods for detecting, removing, and replacing null values in Pandas data structures. They are:
>
> - `isnull()`: Generate a boolean mask indicating missing values
> - `notnull()`: Opposite of `isnull()`
> - `dropna()`: Return a filtered version of the data
> - `fillna()`: Return a copy of the data with missing values filled or imputed

我们已经看到，Pandas将`None`和`NaN`看成是可以互相转换的缺失值或空值。与此同时，Pandas还提供了一些很有用的方法用来在数据集中发现、移除和替换空值。这些方法包括：

- `isnull()`：生成一个布尔遮盖数组指示缺失值的位置
- `notnull()`：`isnull()`相反方法
- `dropna()`：返回一个过滤掉缺失值、空值的数据集
- `fillna()`：返回一个数据集的副本，里面的缺失值、空值使用另外的值来替代

> We will conclude this section with a brief exploration and demonstration of these routines.

我们在最后讨论这些方法作为本节的总结。

Detecting null values

检测空值

> Pandas data structures have two useful methods for detecting null data: `isnull()` and `notnull()`. Either one will return a Boolean mask over the data. For example:

Pandas数据集有两个方法用来检测空值：`isnull()`和`notnull()`。它们都会返回一个布尔遮盖数组。例如：

In [13]:

```
data = pd.Series([1, np.nan, 'hello', None])
```

In [14]:

```
data.isnull()
```

Out[14]:

```
0    False
1     True
2    False
3     True
dtype: bool
```

> As mentioned in Data Indexing and Selection, Boolean masks can be used directly as a `Series` or `DataFrame` index:

在数据索引和选择中我们已经介绍过，布尔遮盖数组可以直接在`Series`或`DataFrame`对象上作为索引使用：

In [15]:

```
data[data.notnull()]
```

Out[15]:

```
0        1
2    hello
dtype: object
```

> The `isnull()` and `notnull()` methods produce similar Boolean results for `DataFrame`s.

在`DataFrame`对象上，`isnull()`和`notnull()`方法也会产生相似的布尔数组。

Dropping null values

去除空值

> In addition to the masking used before, there are the convenience methods, `dropna()` (which removes NA values) and `fillna()` (which fills in NA values). For a `Series`, the result is straightforward:

除了上面的遮盖之外，还有两个很方便的方法`dropna()`（移除NA值）和`fillna()`（填充NA值）。对于`Series`对象来说，结果显而易见：

In [16]:

```
data.dropna()
```

Out[16]:

```
0        1
2    hello
dtype: object
```

> For a `DataFrame`, there are more options. Consider the following `DataFrame`:

对于`DataFrame`对象，提供了更多选项。考虑下面的`DataFrame`：

In [17]:

```
df = pd.DataFrame([[1,      np.nan, 2],
                   [2,      3,      5],
                   [np.nan, 4,      6]])
df
```

Out[17]:

	0	1	2
0	1.0	NaN	2
1	2.0	3.0	5
2	NaN	4.0	6

> We cannot drop single values from a `DataFrame`; we can only drop full rows or full columns. Depending on the application, you might want one or the other, so `dropna()` gives a number of options for a `DataFrame`.

我们不能在`DataFrame`中移除单个空值；我们只能移除整行或者整列。取决于需求，你可能想移除行或列之一，`dropna()`为`DataFrame`对象提供了一些参数选择。

> By default, `dropna()` will drop all rows in which *any* null value is present:

默认，`dropna()`会移除出现了空值的整行：

In [18]:

```
df.dropna()
```

Out[18]:

	0	1	2
1	2.0	3.0	5

> Alternatively, you can drop NA values along a different axis; `axis=1` drops all columns containing a null value:

你可以通过设置axis参数（如`axis=1`）来沿着不同的维度来移除空值，下面是移除含有空值的列的例子：

In [19]:

```
df.dropna(axis='columns')
```

Out[19]:

	2
0	2
1	5
2	6

> But this drops some good data as well; you might rather be interested in dropping rows or columns with *all* NA values, or a majority of NA values. This can be specified through the `how` or `thresh` parameters, which allow fine control of the number of nulls to allow through.

但是这会移除一些良好的数据；你可能更希望移除那些*全部*是NA值或者大部分是NA值的行或列。这可以通过设置`how`或`thresh`参数来实现，它们可以更加精细地控制移除的行或列包含的空值个数。

> The default is `how='any'`, such that any row or column (depending on the `axis` keyword) containing a null value will be dropped. You can also specify `how='all'`, which will only drop rows/columns that are *all* null values:

默认的情况是`how='any'`，因此任何行或列只要含有空值都会被移除。你可以将它设置为`how=all`，这样只有那些行或列*全部*由空值构成的情况下才会被移除：

In [20]:

```
df[3] = np.nan
df
```

Out[20]:

	0	1	2	3
0	1.0	NaN	2	NaN
1	2.0	3.0	5	NaN
2	NaN	4.0	6	NaN

In [21]:

```
df.dropna(axis='columns', how='all')
```

Out[21]:

	0	1	2
0	1.0	NaN	2
1	2.0	3.0	5
2	NaN	4.0	6

> For finer-grained control, the `thresh` parameter lets you specify a minimum number of non-null values for the row/column to be kept:

如果需要更加精细的控制，`thresh`参数可以让你指定结果中每行或列至少包含非空值的个数：

In [22]:

```
df.dropna(axis='rows', thresh=3) # 行中如果有3个或以上的非空值，将会被保留
```

Out[22]:

	0	1	2	3
1	2.0	3.0	5	NaN

> Here the first and last row have been dropped, because they contain only two non-null values.

上例中第一行和第三行被移除了，因为它们都只含有2个非空值。

Filling null values

填充空值

> Sometimes rather than dropping NA values, you'd rather replace them with a valid value. This value might be a single number like zero, or it might be some sort of imputation or interpolation from the good values. You could do this in-place using the `isnull()` method as a mask, but because it is such a common operation Pandas provides the `fillna()` method, which returns a copy of the array with the null values replaced.

有时我们想要的不是移除NA值，而是希望将它们替换为正确的值。替换后的值可能是一个标量如0，或者从其他正确数值归并或插补的值。你当然可以使用`isnull()`然后赋值的方式来实现，但是因为这个需求是如此广泛，Pandas提供了`fillna()`方法，用来返回一个替换空值后的数据集副本。

> Consider the following `Series`:

考虑下面的`Series`：

In [23]:

```
data = pd.Series([1, np.nan, 2, None, 3], index=list('abcde'))
data
```

Out[23]:

```
a    1.0
b    NaN
c    2.0
d    NaN
e    3.0
dtype: float64
```

> We can fill NA entries with a single value, such as zero:

我们可以将NA值替换成为一个标量，例如0：

In [24]:

```
data.fillna(0)
```

Out[24]:

```
a    1.0
b    0.0
c    2.0
d    0.0
e    3.0
dtype: float64
```

> We can specify a forward-fill to propagate the previous value forward:

我们也可以指定填充的方法，如向前填充，将前一个值传播到下一个空值：

In [25]:

```
# 向前填充
data.fillna(method='ffill')
```

Out[25]:

```
a    1.0
b    1.0
c    2.0
d    2.0
e    3.0
dtype: float64
```

> Or we can specify a back-fill to propagate the next values backward:

或者使用向后填充，使用后一个有效值传播到前一个空值：

In [26]:

```
# 向后填充
data.fillna(method='bfill')
```

Out[26]:

```
a    1.0
b    2.0
c    2.0
d    3.0
e    3.0
dtype: float64
```

> For `DataFrame`s, the options are similar, but we can also specify an `axis` along which the fills take place:

对于`DataFrame`对象，选项是类似的，但是我们可以指定`axis`参数让填充沿着某个特定维度进行：

In [27]:

```
df
```

Out[27]:

	0	1	2	3
0	1.0	NaN	2	NaN
1	2.0	3.0	5	NaN
2	NaN	4.0	6	NaN

In [28]:

```
# 按列进行向前填充
df.fillna(method='ffill', axis=1)
```

Out[28]:

	0	1	2	3
0	1.0	1.0	2.0	2.0
1	2.0	3.0	5.0	5.0
2	NaN	4.0	6.0	6.0

> Notice that if a previous value is not available during a forward fill, the NA value remains.

结果看到如果空值的前面没有值（此处的`df.loc[2, 0]`前面已经没有列，沿着列填充），那么NA值将会保留下来。

< 在Pandas中操作数据 | 目录 | 层次化的索引 >

Open in Colab