Handling Missing Data

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.

In this chapter, we will discuss some general considerations for missing data, look at how Pandas chooses to represent it, and explore some built-in Pandas tools for handling missing data in Python. Here and throughout the book, I will refer to missing data in general as *null*, *NaN*, or *NA* values.

Trade-offs in Missing Data Conventions

A number of approaches have been developed to track 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.

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

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 that is part of the IEEE floating-point specification.

Neither 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 nonoptimized) logic in CPU and GPU arithmetic, because common special values like `NaN` are not available for all data types.

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 to indicate an NA state.

Missing Data in Pandas

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.

Perhaps 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 has just 4 main data types, NumPy supports *far* more than this: for example, while R has a single integer

type, NumPy supports 14 basic integer types once you account for available bit widths, 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 would significantly reduce the range of values it can represent.

Because of these constraints and trade-offs, Pandas has two "modes" of storing and manipulating null values:

- The default mode is to use a sentinel-based missing data scheme, with sentinel values `NaN` or `None` depending on the type of the data.
- Alternatively, you can opt in to using the nullable data types (dtypes) Pandas provides (discussed later in this chapter), which results in the creation an accompanying mask array to track missing entries. These missing entries are then presented to the user as the special `pd.NA` value.

In either case, the data operations and manipulations provided by the Pandas API will handle and propagate those missing entries in a predictable manner. But to develop some intuition into *why* these choices are made, let's dive quickly into the trade-offs inherent in `None`, `NaN`, and `NA`. As usual, we'll start by importing NumPy and Pandas:

In [1]:
```
import numpy as np
import pandas as pd
```

None as a Sentinel Value

For some data types, Pandas uses `None` as a sentinel value. `None` is a Python object, which means that any array containing `None` must have `dtype=object`—that is, it must be a sequence of Python objects.

For example, observe what happens if you pass `None` to a NumPy array:

In [2]:
```
vals1 = np.array([1, None, 2, 3])
vals1
```

Out[2]:
```
array([1, None, 2, 3], 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. The downside of using `None` in this way is that 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:

In [3]:
```
%timeit np.arange(1E6, dtype=int).sum()
```

```
2.73 ms ± 288 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

In [4]:
```
%timeit np.arange(1E6, dtype=object).sum()
```

```
92.1 ms ± 3.42 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Further, because Python does not support arithmetic operations with `None`, aggregations like `sum` or `min` will generally lead to an error:

In [5]:
```
vals1.sum()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/var/folders/xc/sptt9bk14s34rgxt7453p03r0000gp/T/ipykernel_91333/1181914653.py in <module>
----> 1 vals1.sum()

~/.local/share/virtualenvs/python-data-science-handbook-2e-u_kwqDTB/lib/python3.9/site-packages/numpy/core/_methods.py in _sum(a, axis, dtype, out, keepdims, initial, where)
     46 def _sum(a, axis=None, dtype=None, out=None, keepdims=False,
     47          initial=_NoValue, where=True):
---> 48     return umr_sum(a, axis, dtype, out, keepdims, initial, where)
     49
     50 def _prod(a, axis=None, dtype=None, out=None, keepdims=False,

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

For this reason, Pandas does not use `None` as a sentinel in its numerical arrays.

NaN: Missing Numerical Data

The other missing data sentinel, `NaN` is different; it is a special floating-point value recognized by all systems that use the standard IEEE floating-point representation:

In [6]:
```
vals2 = np.array([1, np.nan, 3, 4])
vals2
```

Out[6]:
```
array([ 1., nan,  3.,  4.])
```

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. Keep in mind 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`:

In [7]:
```
1 + np.nan
```

Out[7]:
```
nan
```

In [8]:
```
0 * np.nan
```

Out[8]:
```
nan
```

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

In [9]:
```
vals2.sum(), vals2.min(), vals2.max()
```

Out[9]:
```
(nan, nan, nan)
```

That said, NumPy does provide `NaN` -aware versions of aggregations that will ignore these missing values:

In [10]:
```
np.nansum(vals2), np.nanmin(vals2), np.nanmax(vals2)
```

Out[10]:
```
(8.0, 1.0, 4.0)
```

The main downside of `NaN` is that it is specifically a floating-point value; there is no equivalent `NaN` value for integers, strings, or other types.

NaN and None in Pandas

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

In [11]:
```
pd.Series([1, np.nan, 2, None])
```

Out[11]:
```
0    1.0
1    NaN
2    2.0
3    NaN
dtype: float64
```

For types that don't have an available sentinel value, Pandas automatically typecasts 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:

In [12]:
```
x = pd.Series(range(2), dtype=int)
x
```

Out[12]:
```
0    0
1    1
dtype: int64
```

In [13]:
```
x[0] = None
x
```

Out[13]:
```
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.

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.

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

Typeclass	Conversion when storing NAs	NA sentinel value
`floating`	No change	`np.nan`
`object`	No change	`None` or `np.nan`
`integer`	Cast to `float64`	`np.nan`
`boolean`	Cast to `object`	`None` or `np.nan`

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

Pandas Nullable Dtypes

In early versions of Pandas, `NaN` and `None` as sentinel values were the only missing data representations available. The primary difficulty this introduced was with regard to the implicit type casting: for example, there was no way to represent a true integer array with missing data.

To address this difficulty, Pandas later added *nullable dtypes*, which are distinguished from regular dtypes by capitalization of their names (e.g., `pd.Int32` versus `np.int32`). For backward compatibility, these nullable dtypes are only used if specifically requested.

For example, here is a `Series` of integers with missing data, created from a list containing all three available markers of missing data:

In [14]:
```
pd.Series([1, np.nan, 2, None, pd.NA], dtype='Int32')
```

Out[14]:
```
0       1
1    <NA>
2       2
3    <NA>
4    <NA>
dtype: Int32
```

This representation can be used interchangeably with the others in all the operations explored

Operating on Null Values

As we have seen, Pandas treats `None`, `NaN`, and `NA` as essentially interchangeable for indicating missing or null values. To facilitate this convention, Pandas provides several methods for detecting, removing, and replacing null values in Pandas data structures. They are:

- `isnull`: Generates a Boolean mask indicating missing values
- `notnull`: Opposite of `isnull`
- `dropna`: Returns a filtered version of the data
- `fillna`: Returns a copy of the data with missing values filled or imputed

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

Detecting Null Values

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:

In [15]:
```
data = pd.Series([1, np.nan, 'hello', None])
```

In [16]:
```
data.isnull()
```

Out[16]:
```
0    False
1     True
2    False
3     True
dtype: bool
```

As mentioned in Data Indexing and Selection (03.02-Data-Indexing-and-Selection.ipynb), Boolean masks can be used directly as a `Series` or `DataFrame` index:

In [17]:
```
data[data.notnull()]
```

Out[17]:
```
0        1
2    hello
dtype: object
```

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

Dropping Null Values

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

In [18]:
```
data.dropna()
```

Out[18]:
```
0        1
2    hello
dtype: object
```

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

In [19]:
```
df = pd.DataFrame([[1,      np.nan, 2],
                   [2,      3,      5],
                   [np.nan, 4,      6]])
df
```

Out[19]:

	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 entire rows or columns. Depending on the application, you might want one or the other, so `dropna` includes a number of options for a `DataFrame`.

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

In [20]:
```
df.dropna()
```

Out[20]:

	0	1	2
1	2.0	3.0	5

Alternatively, you can drop NA values along a different axis. Using `axis=1` or `axis='columns'` drops all columns containing a null value:

In [21]:
```
df.dropna(axis='columns')
```

Out[21]:

	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.

The default is `how='any'`, such that any row or column containing a null value will be dropped.

In [22]:
```
df[3] = np.nan
df
```

Out[22]:

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

In [23]:
```
df.dropna(axis='columns', how='all')
```

Out[23]:

	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:

In [24]:
```
df.dropna(axis='rows', thresh=3)
```

Out[24]:

	0	1	2	3
1	2.0	3.0	5	NaN

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

Filling Null Values

Sometimes rather than dropping NA values, you'd like to 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.

Consider the following `Series`:

In [25]:
```
data = pd.Series([1, np.nan, 2, None, 3], index=list('abcde'), dtype='Int32')
data
```

Out[25]:
```
a       1
b    <NA>
c       2
d    <NA>
e       3
dtype: Int32
```

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

In [26]:
```
data.fillna(0)
```

Out[26]:
```
a    1
b    0
c    2
d    0
e    3
dtype: Int32
```

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

In [27]:
```
# forward fill
data.fillna(method='ffill')
```

Out[27]:
```
a    1
b    1
c    2
d    2
e    3
dtype: Int32
```

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

In [28]:
```
# back fill
data.fillna(method='bfill')
```

Out[28]:
```
a    1
b    2
c    2
d    3
e    3
dtype: Int32
```

In the case of a `DataFrame`, the options are similar, but we can also specify an `axis` along which the fills should take place:

In [29]:
```
df
```

Out[29]:

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

In [30]:
```
df.fillna(method='ffill', axis=1)
```

Out[30]:

	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.