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.

Trade-Offs in Missing Data Conventions

A number of schemes 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.

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.

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.

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.

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 a 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.

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.

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.

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.

None: Pythonic missing data

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

```
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:

```
In[3]: for dtype in ['object', 'int']:
           print("dtype =", dtype)
           %timeit np.arange(1E6, dtype=dtype).sum()
           print()
```

```
dtype = object
10 loops, best of 3: 78.2 ms per loop

dtype = int
100 loops, best of 3: 3.06 ms per loop
```

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:

```
In[4]: vals1.sum()
```

```
TypeError                                 Traceback (most recent call last)

<ipython-input-4-749fd8ae6030> in <module>()
----> 1 vals1.sum()


/Users/jakevdp/anaconda/lib/python3.5/site-packages/numpy/core/_methods.py ...
     30
     31 def _sum(a, axis=None, dtype=None, out=None, keepdims=False):
---> 32     return umr_sum(a, axis, dtype, out, keepdims)
     33
     34 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.

NaN: Missing numerical data

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:

```
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:

```
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:

```
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:

```
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.

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[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:

```
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.)

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.

Table 3-2 lists the upcasting conventions in Pandas when NA values are introduced.

Table 3-2. Pandas handling of NAs by type

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.

Operating on Null Values

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

We will conclude this section 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[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" on page 107, Boolean masks can be used directly as a `Series` or `DataFrame` index: