Handling Missing Data

It's important to remember that real-world data is seldom tidy and homogenous, unlike the data presented in many courses. It's important to note that a significant portion of research-worthy datasets will likely be incomplete. However, things might become much more difficult since various data sources could present the absence of data in different ways.

Here, we'll go over some high-level points to keep in mind when dealing with missing data, go over how Pandas chooses to represent it, and show off some of the built-in tools available in Pandas for dealing with missing data in Python. Null, NaN, or NA values will be used to represent missing information here and throughout the Study.

Trade-Offs in Missing Data Conventions

Different systems have been devised to flag the existence of blank cells in a DataFrame or table. Both utilizing a mask to generally indicate missing entries and using a sentinel value to signify a missing item are common practices.

In masking, the mask might be a whole independent Boolean array, or it could require repurposing a single bit in the data representation to signal the absence of a value at a certain location.

To indicate a missing integer value, the sentinel value in the sentinel technique may be -9999 or an unusual bit pattern; to indicate a missing floating-point value, the sentinel value might be NaN (Not a Number), a special value which is part of the IEEE floating-point standard.

In order to employ a mask array, more space must be allocated for a Boolean array, which increases the overall storage and processing requirements. In addition to potentially requiring additional (and generally non-optimized) logic in CPU and GPU arithmetic, a sentinel value limits the range of acceptable numbers that may be represented. Not all data types support frequently used special values like NaN.

Conventions vary across languages and systems because there is often no "best" way to do things. For instance, the SciDB system appends an additional byte to every cell that indicates a NA condition, whereas the R programming language utilizes sentinel values (reserved bit patterns inside each data type) to signify missing data..

Missing Data in Pandas

Because it is dependent on the NumPy package, which does not have a built-in concept of NA values for non-floating-point data types, Pandas is limited in how it handles missing values.

To signify nullness, Pandas might have followed R's example and required users to provide bit patterns for each data type, but this proved to be too cumbersome. NumPy provides significantly more than R's four fundamental data types, including, for example, fourteen basic integer types after you account for available precisions, signedness, and endianness of the encoding. An unmanageable amount of extra work would be required to special case different operations for different NumPy types if a reserved bit pattern was used in all of the possible

kinds, and a new fork of the NumPy package would even be needed. In addition, for smaller data types (such 8-bit integers), giving up a bit to use as a mask can drastically decrease the range of values it may represent.

Masked arrays, or arrays with a Boolean mask array attached to indicate whether data is "good" or "bad," are supported in NumPy. Pandas might have been created from this, but it's not a good idea due to the extra work involved in storing and processing data and in keeping the code up to date.

In light of these limitations, Pandas relies on sentinels to fill in gaps in data and makes use of two pre-existing null values in Python: the special floating-point NaN value and the Python None object. As we'll see, this option isn't without its drawbacks, but it ends up being a reasonable middle ground in most relevant scenarios.

`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 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):

```
import numpy as np
import pandas as pd
vals1 = np.array([1, None, 3, 4])
vals1
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:

```
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:

```
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
in _sum(a, axis, dtype, out, keepdims)
```

```
     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:

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

```
1 + np.nan
nan
0 *  np.nan
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:

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

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

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

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

```
x = pd.Series(range(2), dtype=int)
x
0    0
1    1
dtype: int64
x[0] = None
x
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.

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.

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:

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

```
data[data.notnull()]
0        1
2    hello
dtype: object
```

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

Dropping null values

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:

```
data.dropna()
0        1
2    hello
dtype: object
```

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

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

	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`.

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

```
df.dropna()
```

	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:

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

	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 (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:

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

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

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

	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:

```
df.dropna(axis='rows', thresh=3)
```

	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.

Filling null values

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.

Consider the following `Series`:

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

```
data.fillna(0)
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:

```
# forward-fill
data.fillna(method='ffill')
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:

```
# back-fill
data.fillna(method='bfill')
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:

```
df
```

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

```
df.fillna(method='ffill', axis=1)
```

	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.

Combining Datasets: Concat and Append

When many data sets are combined, new insights might be uncovered. These procedures might range from the simple concatenation of two datasets to the more complex joins and merges performed in a database to account for any overlaps between the datasets. These tasks are optimized for Series and DataFrames, and Pandas provides functions and methods to make data wrangling quick and easy.

In this section, we'll look at how to use the pd.concat method to connect several Series or DataFrames together, and in a subsequent section, we'll explore the more complex in-memory merges and joins available in Pandas. When many data sets are combined, new insights might be uncovered. These procedures might range from the simple concatenation of two datasets to the more complex joins and merges performed in a database to account for any overlaps between the datasets. These tasks are optimized for Series and DataFrames, and Pandas provides functions and methods to make data wrangling quick and easy.

In this section, we'll look at how to use the pd.concat method to connect several Series or DataFrames together, and in a subsequent section, we'll explore the more complex in-memory merges and joins available in Pandas.

We begin with the standard imports:

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

For convenience, we'll define this function which creates a `DataFrame` of a particular form that will be useful below:

```
def make_df(cols, ind):
    """Quickly make a DataFrame"""
    data = {c: [str(c) + str(i) for i in ind]
            for c in cols}
    return pd.DataFrame(data, ind)

# example DataFrame
make_df('ABC', range(3))
```

	A	B	C
0	A0	B0	C0
1	A1	B1	C1
2	A2	B2	C2

In addition, we'll create a quick class that allows us to display multiple `DataFrame`s side by side. The code makes use of the special `_repr_html_` method, which IPython uses to implement its rich object display:

```
class display(object):
    """Display HTML representation of multiple objects"""
```

```
    template = """<div style="float: left; padding: 10px;">
    <p style='font-family:"Courier New", Courier, monospace'>{0}</p>{1}
    </div>"""
    def __init__(self, *args):
        self.args = args

    def _repr_html_(self):
        return '\n'.join(self.template.format(a, eval(a)._repr_html_())
                         for a in self.args)

    def __repr__(self):
        return '\n\n'.join(a + '\n' + repr(eval(a))
                           for a in self.args)
```

The use of this will become clearer as we continue our discussion in the following section.

Recall: Concatenation of NumPy Arrays

Concatenation of `Series` and `DataFrame` objects is very similar to concatenation of Numpy arrays, which can be done via the `np.concatenate` function. Recall that with it, you can combine the contents of two or more arrays into a single array:

```
x = [1, 2, 3]
y = [4, 5, 6]
z = [7, 8, 9]
np.concatenate([x, y, z])
array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

The first argument is a list or tuple of arrays to concatenate. Additionally, it takes an `axis` keyword that allows you to specify the axis along which the result will be concatenated:

```
x = [[1, 2],
     [3, 4]]
np.concatenate([x, x], axis=1)
array([[1, 2, 1, 2],
       [3, 4, 3, 4]])
```

Simple Concatenation with `pd.concat`

Pandas has a function, `pd.concat()`, which has a similar syntax to `np.concatenate` but contains a number of options that we'll discuss momentarily:

```
# Signature in Pandas v0.18
pd.concat(objs, axis=0, join='outer', join_axes=None, ignore_index=False,
          keys=None, levels=None, names=None, verify_integrity=False,
          copy=True)
```

`pd.concat()` can be used for a simple concatenation of `Series` or `DataFrame` objects, just as `np.concatenate()` can be used for simple concatenations of arrays:

```
ser1 = pd.Series(['A', 'B', 'C'], index=[1, 2, 3])
ser2 = pd.Series(['D', 'E', 'F'], index=[4, 5, 6])
pd.concat([ser1, ser2])
```

```
1    A
2    B
3    C
4    D
5    E
6    F
dtype: object
```

It also works to concatenate higher-dimensional objects, such as `DataFrame`s:

```
df1 = make_df('AB', [1, 2])
df2 = make_df('AB', [3, 4])
display('df1', 'df2', 'pd.concat([df1, df2])')
```

`df1`

	A	B
1	A1	B1
2	A2	B2

`df2`

	A	B
3	A3	B3
4	A4	B4

`pd.concat([df1, df2])`

	A	B
1	A1	B1
2	A2	B2
3	A3	B3
4	A4	B4

By default, the concatenation takes place row-wise within the `DataFrame` (i.e., `axis=0`). Like `np.concatenate`, `pd.concat` allows specification of an axis along which concatenation will take place. Consider the following example:

```
df3 = make_df('AB', [0, 1])
df4 = make_df('CD', [0, 1])
display('df3', 'df4', "pd.concat([df3, df4], axis='col')")
```

`df3`

	A	B
0	A0	B0
1	A1	B1

```
df4
```

	C	D
0	C0	D0
1	C1	D1

```
pd.concat([df3, df4], axis='col')
```

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1

We could have equivalently specified `axis=1`; here we've used the more intuitive `axis='col'`.

Duplicate indices

One important difference between `np.concatenate` and `pd.concat` is that Pandas concatenation *preserves indices*, even if the result will have duplicate indices! Consider this simple example:

```
x = make_df('AB', [0, 1])
y = make_df('AB', [2, 3])
y.index = x.index  # make duplicate indices!
display('x', 'y', 'pd.concat([x, y])')
```

```
x
```

	A	B
0	A0	B0
1	A1	B1

```
y
```

	A	B
0	A2	B2
1	A3	B3

```
pd.concat([x, y])
```

	A	B
0	A0	B0
1	A1	B1
0	A2	B2
1	A3	B3

Notice the repeated indices in the result. While this is valid within `DataFrame`s, the outcome is often undesirable. `pd.concat()` gives us a few ways to handle it.

Catching the repeats as an error

If you'd like to simply verify that the indices in the result of `pd.concat()` do not overlap, you can specify the `verify_integrity` flag. With this set to True, the concatenation will raise an exception if there are duplicate indices. Here is an example, where for clarity we'll catch and print the error message:

```
try:
    pd.concat([x, y], verify_integrity=True)
except ValueError as e:
    print("ValueError:", e)
ValueError: Indexes have overlapping values: [0, 1]
```

Ignoring the index

Sometimes the index itself does not matter, and you would prefer it to simply be ignored. This option can be specified using the `ignore_index` flag. With this set to true, the concatenation will create a new integer index for the resulting `Series`:

```
display('x', 'y', 'pd.concat([x, y], ignore_index=True)')
```

x

	A	B
0	A0	B0
1	A1	B1

y

	A	B
0	A2	B2
1	A3	B3

```
pd.concat([x, y], ignore_index=True)
```

	A	B
0	A0	B0
1	A1	B1
2	A2	B2
3	A3	B3

Adding MultiIndex keys

Another option is to use the `keys` option to specify a label for the data sources; the result will be a hierarchically indexed series containing the data:

```
display('x', 'y', "pd.concat([x, y], keys=['x', 'y'])")
```

x

	A	B
0	A0	B0
1	A1	B1

y

	A	B
0	A2	B2
1	A3	B3

```
pd.concat([x, y], keys=['x', 'y'])
```

		A	B
x	**0**	A0	B0
	1	A1	B1
y	**0**	A2	B2
	1	A3	B3

.

Concatenation with joins

In the simple examples we just looked at, we were mainly concatenating `DataFrame`s with shared column names. In practice, data from different sources might have different sets of column names, and `pd.concat` offers several options in this case. Consider the concatenation of the following two `DataFrame`s, which have some (but not all!) columns in common:

```
df5 = make_df('ABC', [1, 2])
df6 = make_df('BCD', [3, 4])
```

```
display('df5', 'df6', 'pd.concat([df5, df6])')
```

df5

	A	B	C
1	A1	B1	C1
2	A2	B2	C2

df6

	B	C	D
3	B3	C3	D3
4	B4	C4	D4

```
pd.concat([df5, df6])
```

	A	B	C	D
1	A1	B1	C1	NaN
2	A2	B2	C2	NaN
3	NaN	B3	C3	D3
4	NaN	B4	C4	D4

By default, the entries for which no data is available are filled with NA values. To change this, we can specify one of several options for the `join` and `join_axes` parameters of the concatenate function. By default, the join is a union of the input columns (`join='outer'`), but we can change this to an intersection of the columns using `join='inner'`:

```
display('df5', 'df6',
        "pd.concat([df5, df6], join='inner')")
```

df5

	A	B	C
1	A1	B1	C1
2	A2	B2	C2

df6

	B	C	D
3	B3	C3	D3
4	B4	C4	D4

```
pd.concat([df5, df6], join='inner')
```

	B	C
1	B1	C1
2	B2	C2
3	B3	C3
4	B4	C4

Another option is to directly specify the index of the remaininig colums using the `join_axes` argument, which takes a list of index objects. Here we'll specify that the returned columns should be the same as those of the first input:

```
display('df5', 'df6',
        "pd.concat([df5, df6], join_axes=[df5.columns])")
```

`df5`

	A	B	C
1	A1	B1	C1
2	A2	B2	C2

`df6`

	B	C	D
3	B3	C3	D3
4	B4	C4	D4

```
pd.concat([df5, df6], join_axes=[df5.columns])
```

	A	B	C
1	A1	B1	C1
2	A2	B2	C2
3	NaN	B3	C3
4	NaN	B4	C4

The combination of options of the `pd.concat` function allows a wide range of possible behaviors when joining two datasets; keep these in mind as you use these tools for your own data.

The `append()` method

Because direct array concatenation is so common, `Series` and `DataFrame` objects have an `append` method that can accomplish the same thing in fewer keystrokes. For example, rather than calling `pd.concat([df1, df2])`, you can simply call `df1.append(df2)`:

```
display('df1', 'df2', 'df1.append(df2)')
```

```
df1
```

	A	B
1	A1	B1
2	A2	B2

```
df2
```

	A	B
3	A3	B3
4	A4	B4

```
df1.append(df2)
```

	A	B
1	A1	B1
2	A2	B2
3	A3	B3
4	A4	B4

Keep in mind that the append() function in Pandas does not alter the old object but rather generates a new object with the merged data, in contrast to the append() and extend() methods of Python lists. Since a new index and data buffer must be created, this approach is also not particularly efficient. Therefore, it is often preferable to construct a list of DataFrames and provide them all at once to the concat() method if you want to execute numerous append operations.