Handling Missing Data

In [ ]:

```
important as presence of missing values may lead to error (If None values are present), wrong result (if NAN values are

```

```
## Missing Data in Pandas

In Pandas both None and NAN are treated as the indicators of missing data
```

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):

In [4]:

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

In [5]:

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

Out[5]:

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

In [7]:

```
vals1.sum() ###Gives an error
###Bcz you cannot perform any operation with None type
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-7-b9ecc17abfdf> in <module>
----> 1 vals1.sum() ###Gives an error
      2 ###Bcz you can not perform any operation with None type

C:\ProgramData\Anaconda3\lib\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.

`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 [25]:

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

Out[25]:

```
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 [26]:
```
1 + np.nan
```

Out[26]:
```
nan
```

In [27]:
```
0 *  np.nan
```

Out[27]:
```
nan
```

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:

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 [ ]:
```
import pandas as pd
import numpy as np
pd.Series([1, np.nan, 2, None])
```

In [8]:
```
x = pd.Series(range(2), dtype=int)
x
```

Out[8]:
```
0    0
1    1
dtype: int32
```

In [9]:
```
x[0] = None
x
```

Out[9]:
```
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:

In [10]:
```
data = pd.Series([1, np.nan, 'hello', None])
```

In [11]:
```
data.isnull()
```

Out[11]:
```
0    False
1     True
2    False
3     True
dtype: bool
```

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

In [12]:
```
data[data.notnull()]
```

Out[12]:
```
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:

In [13]:
```
data.dropna()
```

Out[13]:
```
0        1
2    hello
dtype: object
```

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

In [14]:

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

Out[14]:

	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:

In [15]:

```
df.dropna()
```

Out[15]:

	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:

In [16]:
```python

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

Out[16]:

	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:

In [17]:
```python
df[3] = np.nan
df
```

Out[17]:

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

In [18]:
```python
df.dropna(axis='columns', how='all') ###Check the result the third column is dropped (removed)
```

Out[18]:

	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 [19]:
```
df.dropna(axis='rows', thresh=3)
```

Out[19]:

	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` :

In [20]:
```
data = pd.Series([1, np.nan, 2, None, 3], index=list('abcde'))
data
```

Out[20]:
```
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 or any number:

In [21]:
```
print(data.fillna(0))
data.fillna(8)
```

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

Out[21]:
```
a    1.0
b    8.0
c    2.0
d    8.0
e    3.0
dtype: float64
```

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

In [22]:
```
# forward-fill
data.fillna(method='ffill')
```

Out[22]:
```
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:

In [23]:
```
# back-fill
data.fillna(method='bfill')
```

Out[23]:
```
a    1.0
b    2.0
c    2.0
d    3.0
e    3.0
dtype: float64
```