9

Missing Data

Rarely will you be given a data set without any missing values. There are many representations of missing data. In databases, they are `NULL` values; certain programming languages use `NA`; and depending on where you get your data, missing values can be an empty string, `"`, or even numeric values such as `88` or `99`. Pandas displays missing values as `NaN`.

Learning Objectives

- Identify how missing values are represented in pandas
- Recognize potential ways data can go missing in data processing
- Use different functions to fill in missing values

9.1 What Is a NaN Value?

The `NaN` value in Pandas comes from `numpy`. Missing values may be used or displayed in a few ways in Pandas — `NaN`, `NAN`, or `nan` — they are all the same in terms of how you specify a missing (floating point) number, but they are not the same in terms of equality. Appendix I describes how these missing values are imported.

Click here to view code image

```
# Just import the numpy missing values
from numpy import NaN, NAN, nan
```

Missing values are different than other types of data in that they don't really equal anything, not even to themselves. The data is missing, so there is no concept of equality. `NaN` is not equivalent to `0` or an empty string, `''`. This is known as "three-valued logic."

```
print(NaN == True)
```

```
False
```

```
print(NaN == 0)
```

```
False
```

```
print(NaN == '')
```

```
False
```

```
print(NaN == NaN)
```

```
False
```

```
print(NaN == NAN)
```

```
False
```

```
print(NaN == nan)
```

```
False
```

```
print(nan == NAN)
```

```
False
```

Pandas has functions to test for missing values, `isnull()`.

```
import pandas as pd
```

```
print(pd.isnull(NaN))
```

```
True
```

```
print(pd.isnull(nan))
```

```
True
```

```
print(pd.isnull(NAN))
```

```
True
```

Pandas also has functions for testing non-missing values, `notnull()`.

```
print(pd.notnull(NaN))
```

```
False
```

```
print(pd.notnull(42))
```

```
True
```

```
print(pd.notnull('missing'))
```

```
True
```

9.2 Where Do Missing Values Come From?

We can get missing values when we load in a data set with missing values, or from the data munging process.

9.2.1 Load Data

The survey data we used in **Chapter 6** included a data set, `visited`, that contained missing data. When we loaded the data, Pandas automatically found the missing data cell and gave us a dataframe with the `NaN` value in the appropriate cell. In the `read_csv()` function, three parameters relate to reading missing values: `na_values`, `keep_default_na`, and `na_filter`.

The `na_values` parameter allows you to specify additional missing or `NaN` values. You can pass in either a Python `str` (i.e., string) or a list-like object to be automatically coded as missing values when the file is read. Of course, default missing values, such as `NA`, `NaN`, or `nan`, are already available, which is why this parameter is not always used. Some health data may code `99` as a missing value; to specify the use of this value, you would set `na_values=[99]`.

The `keep_default_na` parameter is a `bool` (i.e., `True` or `False` boolean) that allows you to specify whether any additional values need to be considered as missing. This parameter is `True` by default, meaning any additional missing values specified with the `na_values` parameter will be appended to the list of missing values. However, `keep_default_na` can also be set to `keep_default_na=False`, which will **only** use the missing values specified in `na_values`.

Lastly, `na_filter` is a `bool` that will specify whether any values will be read as missing. The default value of `na_filter=True` means that missing values will be coded as `NaN`. If we assign `na_filter=False`, then nothing will be recoded as missing. This parameter can be thought of as a means to turn off all the parameters set for `na_values` and `keep_default_na`, but it is more likely to be used when you want to achieve a performance boost by loading in data without missing values.

Click here to view code image

```
# set the location for data
visited_file = 'data/survey_visited.csv'
```

```
print(pd.read_csv(visited_file))
```

```
   ident   site       dated
0    619   DR-1  1927-02-08
1    622   DR-1  1927-02-10
2    734   DR-3  1939-01-07
3    735   DR-3  1930-01-12
4    751   DR-3  1930-02-26
5    752   DR-3         NaN
6    837  MSK-4  1932-01-14
7    844   DR-1  1932-03-22
```

```
print(pd.read_csv(visited_file, keep_default_na=False))
```

```
   ident   site       dated
0    619   DR-1  1927-02-08
1    622   DR-1  1927-02-10
2    734   DR-3  1939-01-07
3    735   DR-3  1930-01-12
4    751   DR-3  1930-02-26
5    752   DR-3
6    837  MSK-4  1932-01-14
7    844   DR-1  1932-03-22
```

```
print(
  pd.read_csv(visited_file, na_values=[""], keep_default_na=False)
)
```

```
   ident   site       dated
0    619   DR-1  1927-02-08
1    622   DR-1  1927-02-10
2    734   DR-3  1939-01-07
3    735   DR-3  1930-01-12
4    751   DR-3  1930-02-26
5    752   DR-3         NaN
6    837  MSK-4  1932-01-14
7    844   DR-1  1932-03-22
```

9.2.2 Merged Data

Chapter 6 showed you how to combine data sets. Some of the examples in that chapter included missing values in the output. If we recreate the merged table from Section 6.4.3, we will see missing values in the merged output.

Click here to view code image

```
visited = pd.read_csv('data/survey_visited.csv')
survey = pd.read_csv('data/survey_survey.csv')
```

```
print(visited)
```

```
   ident   site       dated
0    619   DR-1  1927-02-08
1    622   DR-1  1927-02-10
2    734   DR-3  1939-01-07
```

```
3    735   DR-3  1930-01-12
4    751   DR-3  1930-02-26
5    752   DR-3         NaN
6    837  MSK-4  1932-01-14
7    844   DR-1  1932-03-22
```

```
print(survey)
```

```
    taken person quant  reading
0     619   dyer   rad     9.82
1     619   dyer   sal     0.13
2     622   dyer   rad     7.80
3     622   dyer   sal     0.09
4     734     pb   rad     8.41
..    ...    ...   ...      ...
16    752    roe   sal    41.60
17    837   lake   rad     1.46
18    837   lake   sal     0.21
19    837    roe   sal    22.50
20    844    roe   rad    11.25

[21 rows x 4 columns]
```

```
vs = visited.merge(survey, left_on='ident', right_on='taken')
print(vs)
```

```
    ident   site       dated  taken person quant  reading
0     619   DR-1  1927-02-08    619   dyer   rad     9.82
```

```
1     619   DR-1  1927-02-08    619   dyer   sal     0.13
2     622   DR-1  1927-02-10    622   dyer   rad     7.80
3     622   DR-1  1927-02-10    622   dyer   sal     0.09
4     734   DR-3  1939-01-07    734     pb   rad     8.41
..    ...    ...           ...   ...    ...   ...     ...
16    752   DR-3           NaN   752    roe   sal    41.60
17    837  MSK-4  1932-01-14    837   lake   rad     1.46
18    837  MSK-4  1932-01-14    837   lake   sal     0.21
19    837  MSK-4  1932-01-14    837    roe   sal    22.50
20    844   DR-1  1932-03-22    844    roe   rad    11.25

[21 rows x 7 columns]
```

9.2.3 User Input Values

The user can also create missing values—for example, by creating a vector of values from a calculation or a manually curated vector. To build on the examples from **Section 2.1**, we will create our own data with missing values. `NaN` values are valid for both `Series` and `DataFrame` objects.

Click here to view code image

```
# missing value in a series
num_legs = pd.Series({'goat': 4, 'amoeba': nan})
print(num_legs)
```

```
goat      4.0
amoeba    NaN
```

```
dtype: float64
```

```
# missing value in a dataframe
scientists = pd.DataFrame(
  {
    "Name": ["Rosaline Franklin", "William Gosset"],
    "Occupation": ["Chemist", "Statistician"],
    "Born": ["1920-07-25", "1876-06-13"],
    "Died": ["1958-04-16", "1937-10-16"],
    "missing": [NaN, nan],
  }
)
print(scientists)
```

```
                Name    Occupation        Born        Died  missing
0  Rosaline Franklin       Chemist  1920-07-25  1958-04-16      NaN
1     William Gosset  Statistician  1876-06-13  1937-10-16      NaN
```

You will notice the `dtype` of the `missing` column will be a `float64`. This is because the `NaN` missing value from `numpy` is a floating point value.

```
print(scientists.dtypes)
```

```
Name          object
Occupation    object
Born          object
Died          object
```

```
missing       float64
dtype: object
```

You can also assign a column of missing values to a dataframe directly.

Click here to view code image

```
# create a new dataframe
scientists = pd.DataFrame(
  {
    "Name": ["Rosaline Franklin", "William Gosset"],
    "Occupation": ["Chemist", "Statistician"],
    "Born": ["1920-07-25", "1876-06-13"],
    "Died": ["1958-04-16", "1937-10-16"],
  }
)

# assign a column of missing values
scientists["missing"] = nan

print(scientists)
```

```
                Name    Occupation        Born        Died  missing
0  Rosaline Franklin       Chemist  1920-07-25  1958-04-16      NaN
1     William Gosset  Statistician  1876-06-13  1937-10-16      NaN
```

9.2.4 Reindexing

Another way to introduce missing values into your data is to reindex your dataframe. This is useful when you want to add new indices to your dataframe, but still want to retain its original values. A common usage is when the index represents some time interval, and you want to add more dates.

If we wanted to look at only the years from 2000 to 2010 from the Gapminder data plot in Section 1.5, we could perform the same grouped operations, subset the data, and then reindex it.

Click here to view code image

```
gapminder = pd.read_csv('data/gapminder.tsv', sep='\t')

life_exp = gapminder.groupby(['year'])['lifeExp'].mean()
print(life_exp)
```

```
year
1952    49.057620
1957    51.507401
1962    53.609249
1967    55.678290
1972    57.647386
          ...
1987    63.212613
1992    64.160338
1997    65.014676
```

```
2002    65.694923
2007    67.007423
Name: lifeExp, Length: 12, dtype: float64
```

We can reindex by subsetting the data and use the `.reindex()` method.

Click here to view code image

```
# subset
y2000 = life_exp[life_exp.index > 2000]
print(y2000)
```

```
year
2002    65.694923
2007    67.007423
Name: lifeExp, dtype: float64
```

```
# reindex
print(y2000.reindex(range(2000, 2010)))
```

```
year
2000          NaN
2001          NaN
2002    65.694923
2003          NaN
2004          NaN
2005          NaN
```

```
2006          NaN
2007    67.007423
2008          NaN
2009          NaN
Name: lifeExp, dtype: float64
```

9.3 Working With Missing Data

Now that we know how missing values can be created, let's see how they behave when we are working with data.

9.3.1 Find and Count Missing Data

One way to look at the number of missing values is to `count()` them.

Click here to view code image

```
ebola = pd.read_csv('data/country_timeseries.csv')
```

```
# count the number of non-missing values
print(ebola.count())
```

```
Date                   122
Day                    122
Cases_Guinea            93
Cases_Liberia           83
Cases_SierraLeone       87
```

```
                      ...
Deaths_Nigeria         38
Deaths_Senegal         22
Deaths_UnitedStates    18
Deaths_Spain           16
Deaths_Mali            12
Length: 18, dtype: int64
```

You can also subtract the number of non-missing rows from the total number of rows.

Click here to view code image

```
num_rows = ebola.shape[0]
num_missing = num_rows - ebola.count()
print(num_missing)
```

```
Date                    0
Day                     0
Cases_Guinea           29
Cases_Liberia          39
Cases_SierraLeone      35
                      ...
Deaths_Nigeria         84
Deaths_Senegal        100
Deaths_UnitedStates   104
Deaths_Spain          106
Deaths_Mali           110
Length: 18, dtype: int64
```

If you want to count the total number of missing values in your data, or count the number of missing values for a particular column, you can use the `count_nonzero()` function from `numpy` in conjunction with the `.isnull()` method.

Click here to view code image

```
import numpy as np

print(np.count_nonzero(ebola.isnull()))
```

```
1214
```

```
print(np.count_nonzero(ebola['Cases_Guinea'].isnull()))
```

```
29
```

Another way to get missing data counts is to use the `.value_counts()` method on a series. This will print a frequency table of values. If you use the `dropna` parameter, you can also get a missing value count.

Click here to view code image

```
# value counts from the Cases_Guinea column
cnts = ebola.Cases_Guinea.value_counts(dropna=False)
```

```
print(cnts)
```

```
NaN       29
86.0       3
495.0      2
112.0      2
390.0      2
          ..
1199.0     1
1298.0     1
1350.0     1
1472.0     1
49.0       1
Name: Cases_Guinea, Length: 89, dtype: int64
```

The results are sorted so you can subset the count vector to just look at the missing values.

Click here to view code image

```
# select the values in the Series where the index is a NaN value
print(cnts.loc[pd.isnull(cnts.index)])
```

```
NaN    29
Name: Cases_Guinea, dtype: int64
```

In Python, `True` values equate to the integer value `1`, and `False` values equate to the integer value `0`. We can use this behavior to get the number of missing values by summing up a boolean vector with the `.sum()` method.

Click here to view code image

```
# check if the value is missing, and sum up the results
print(ebola.Cases_Guinea.isnull().sum())
```

```
29
```

9.3.2 Clean Missing Data

There are many different ways we can deal with missing data. For example, we can replace the missing data with another value, fill in the missing data using existing data, or drop the data from our data set.

9.3.2.1 Recode or Replace

We can use the `.fillna()` method to recode the missing values to another value. For example, suppose we wanted the missing values to be recoded as a 0. When we use `.fillna()`, we can recode the values to a specific value.

Click here to view code image

```
# fill the missing values to 0 and only look at the first 5 columns
```

```
print(ebola.fillna(0).iloc[:, 0:5])
```

```
           Date  Day  Cases_Guinea  Cases_Liberia  Cases_SierraLeone
0      1/5/2015  289        2776.0            0.0            10030.0
1      1/4/2015  288        2775.0            0.0             9780.0
2      1/3/2015  287        2769.0         8166.0             9722.0
3      1/2/2015  286           0.0         8157.0                0.0
4    12/31/2014  284        2730.0         8115.0             9633.0
..          ...  ...           ...            ...                ...
117   3/27/2014    5         103.0            8.0                6.0
118   3/26/2014    4          86.0            0.0                0.0
119   3/25/2014    3          86.0            0.0                0.0
120   3/24/2014    2          86.0            0.0                0.0
121   3/22/2014    0          49.0            0.0                0.0

[122 rows x 5 columns]
```

9.3.2.2 Forward Fill

We can use built-in methods to fill forward or backward. When we fill data forward, the last known value (from top to bottom) is used for the next missing value. In this way, missing values are replaced with the last known and recorded value.

Click here to view code image

```
print(ebola.fillna(method='ffill').iloc[:, 0:5])
```

```
           Date  Day  Cases_Guinea  Cases_Liberia  Cases_SierraLeone
0      1/5/2015  289        2776.0            NaN            10030.0
1      1/4/2015  288        2775.0            NaN             9780.0
2      1/3/2015  287        2769.0         8166.0             9722.0
3      1/2/2015  286        2769.0         8157.0             9722.0
4    12/31/2014  284        2730.0         8115.0             9633.0
..          ...  ...           ...            ...                ...
117   3/27/2014    5         103.0            8.0                6.0
118   3/26/2014    4          86.0            8.0                6.0
119   3/25/2014    3          86.0            8.0                6.0
120   3/24/2014    2          86.0            8.0                6.0
121   3/22/2014    0          49.0            8.0                6.0

[122 rows x 5 columns]
```

If a column begins with a missing value, then that data will remain missing because there is no previous value to fill in.

9.3.2.3 Backward Fill

We can also have Pandas fill data backward. When we fill data backward, the newest value (from top to bottom) is used to replace the missing data. In this way, missing values are replaced with the newest value.

Click here to view code image

```
print(ebola.fillna(method='bfill').iloc[:, 0:5])
```

```
            Date  Day  Cases_Guinea  Cases_Liberia  Cases_SierraLeone
0       1/5/2015  289        2776.0         8166.0            10030.0
1       1/4/2015  288        2775.0         8166.0             9780.0
2       1/3/2015  287        2769.0         8166.0             9722.0
3       1/2/2015  286        2730.0         8157.0             9633.0
4     12/31/2014  284        2730.0         8115.0             9633.0
..           ...  ...           ...            ...                ...
117    3/27/2014    5         103.0            8.0                6.0
118    3/26/2014    4          86.0            NaN                NaN
119    3/25/2014    3          86.0            NaN                NaN
120    3/24/2014    2          86.0            NaN                NaN
121    3/22/2014    0          49.0            NaN                NaN

[122 rows x 5 columns]
```

If a column ends with a missing value, then it will remain missing because there is no new value to fill in.

9.3.2.4 Interpolate

Interpolation uses existing values to fill in missing values. There are many ways to fill in missing values, the interpolation in Pandas fills in missing values linearly. Specifically, it treats the missing values as if they should be equally spaced apart.

Click here to view code image

```
print(ebola.interpolate().iloc[:, 0:5])
```

```
          Date  Day  Cases_Guinea  Cases_Liberia  Cases_SierraLeone
0     1/5/2015  289        2776.0            NaN            10030.0
1     1/4/2015  288        2775.0            NaN             9780.0
2     1/3/2015  287        2769.0         8166.0             9722.0
3     1/2/2015  286        2749.5         8157.0             9677.5
4   12/31/2014  284        2730.0         8115.0             9633.0
..         ...  ...           ...            ...                ...
117  3/27/2014    5         103.0            8.0                6.0
118  3/26/2014    4          86.0            8.0                6.0
119  3/25/2014    3          86.0            8.0                6.0
120  3/24/2014    2          86.0            8.0                6.0
121  3/22/2014    0          49.0            8.0                6.0

[122 rows x 5 columns]
```

Notice how it behaves kind of in a forward fill fashion, but instead of passing on the last known value, it will fill in the differences between values.

The `.interpolate()` method has a `method` parameter that can change the interpolation method.[1] Possible values at the time of writing have been reproduced in Table 9.1.

---

1. `Series.interpolate()` documentation: https://pandas.pydata.org/docs/reference/api/pandas.Series.interpolate.html

Table 9.1 **Possible Values (at the Time of Writing) to Pass Into the** `method`

Parameter in the `.interpolate()` Method

	Technique	Description
1	linear	Ignore the index and treat the values as equally spaced. This is the only method supported on Multi-Indexes
2	time	Works on daily and higher resolution data to interpolate given length of interval
3	index, values	Use the actual numerical values of the index
4	pad	Fill in NaNs using existing values
5	nearest, zero, slinear, quadratic, cubic, spline, barycentric, polynomial	Passed to `scipy.interpolate.interp1d`; these methods use the numerical values of the index
6	krogh, piecewise_polynomial,	Wrappers around the SciPy interpolation methods of similar names

	Technique	Description
	spline, pchip, akima, cubicspline	
7	from_derivatives	Refers to `scipy.interpolate.BPoly`

9.3.2.5 Drop Missing Values

The last way to work with missing data is to drop observations or variables with missing data. Depending on how much data is missing, keeping only complete case data can leave you with a useless data set. Perhaps the missing data is not random, so that dropping missing values will leave you with a biased data set, or perhaps keeping only complete data will leave you with insufficient data to run your analysis.

We can use the `.dropna()` method to drop missing data, and specify parameters to this method that control how data are dropped. For instance, the `how` parameter lets you specify whether a row (or column) is dropped when `'any'` or `'all'` of the data is missing. The `thresh` parameter lets you specify how many non-`NaN` values you have before dropping the row or column.

```
print(ebola.shape)
```

```
(122, 18)
```

If we keep only complete cases in our Ebola data set, we are left with just one row of data.

Click here to view code image

```
ebola_dropna = ebola.dropna()
print(ebola_dropna.shape)
```

```
(1, 18)
```

```
print(ebola_dropna)
```

```
          Date  Day  Cases_Guinea  Cases_Liberia  Cases_SierraLeone  \
19  11/18/2014  241        2047.0         7082.0             6190.0

    Cases_Nigeria  Cases_Senegal  Cases_UnitedStates  Cases_Spain  \
19           20.0            1.0                 4.0          1.0

    Cases_Mali  Deaths_Guinea  Deaths_Liberia  Deaths_SierraLeone  \
19         6.0         1214.0          2963.0              1267.0

    Deaths_Nigeria  Deaths_Senegal  Deaths_UnitedStates  \
19             8.0             0.0                  1.0
```

```
    Deaths_Spain  Deaths_Mali
19           0.0          6.0
```

9.3.3 Calculations With Missing Data

Suppose we wanted to look at the case counts for multiple regions. We can add multiple regions together to get a new column holding the case counts.

```
ebola["Cases_multiple"] = (
  ebola["Cases_Guinea"]
  + ebola["Cases_Liberia"]
  + ebola["Cases_SierraLeone"]
)
```

Let's look at the first 10 lines of the calculation.

Click here to view code image

```
ebola_subset = ebola.loc[
    :,
    [
        "Cases_Guinea",
        "Cases_Liberia",
        "Cases_SierraLeone",
        "Cases_multiple",
    ],
]
print(ebola_subset.head(n=10))
```

```
   Cases_Guinea  Cases_Liberia  Cases_SierraLeone  Cases_multiple
0        2776.0            NaN            10030.0             NaN
1        2775.0            NaN             9780.0             NaN
2        2769.0         8166.0             9722.0         20657.0
3           NaN         8157.0                NaN             NaN
4        2730.0         8115.0             9633.0         20478.0
5        2706.0         8018.0             9446.0         20170.0
6        2695.0            NaN             9409.0             NaN
7        2630.0         7977.0             9203.0         19810.0
8        2597.0     NaN 9004.0                NaN
9        2571.0         7862.0             8939.0         19372.0
```

You can see that a value for `Cases_multiple` was calculated only when there was no missing value for `Cases_Guinea`, `Cases_Liberia`, and `Cases_SierraLeone`. Calculations with missing values will typically return a missing value, unless the function or method called has a means to ignore missing values in its calculations.

Examples of built-in methods that can ignore missing values include `.mean()` and `.sum()`. These functions will typically have a `skipna` parameter that will still calculate a value by skipping over the missing values.

Click here to view code image

```
# skipping missing values is True by default
print(ebola.Cases_Guinea.sum(skipna = True))
```

```
84729.0
```

```
print(ebola.Cases_Guinea.sum(skipna = False))
```

```
nan
```

9.4 Pandas Built-In NA Missing

Pandas 1.0 introduced a built-in `NA` value (`pd.NA`). At the time of writing this feature is still "experimental."[2] The main goal of this feature is to provide a missing value that works across different data types.

---

2. Pandas experimental `NA`: https://pandas.pydata.org/docs/user_guide/missing_data.html#experimental-na-scalar-to-denote-missing-values

Let's use our previous `scientists` data set from earlier and look at the `.dtypes`.

Click here to view code image

```
scientists = pd.DataFrame(
  {
    "Name": ["Rosaline Franklin", "William Gosset"],
```

```
        "Occupation": ["Chemist", "Statistician"],
        "Born": ["1920-07-25", "1876-06-13"],
        "Died": ["1958-04-16", "1937-10-16"],
        "Age": [37, 61]
    }
)

print(scientists)
```

```
                Name    Occupation        Born        Died  Age
0  Rosaline Franklin       Chemist  1920-07-25  1958-04-16   37
1     William Gosset  Statistician  1876-06-13  1937-10-16   61
```

```
print(scientists.dtypes)
```

```
Name          object
Occupation    object
Born          object
Died          object
Age            int64
dtype: object
```

```
scientists.loc[1, "Name"] = pd.NA
scientists.loc[1, "Age"] = pd.NA

print(scientists)
```

```
                 Name    Occupation        Born        Died   Age
0  Rosaline Franklin         Chemist  1920-07-25  1958-04-16    37
1                <NA>  Statistician  1876-06-13  1937-10-16  <NA>
```

```
print(scientists.dtypes)
```

```
Name          object
Occupation    object
Born          object
Died          object
Age           object
dtype: object
```

Compare the `.dtypes` from `pd.NA` and `np.NaN` from earlier in this chapter.

Click here to view code image

```
scientists = pd.DataFrame(
  {
    "Name": ["Rosaline Franklin", "William Gosset"],
    "Occupation": ["Chemist", "Statistician"],
    "Born": ["1920-07-25", "1876-06-13"],
    "Died": ["1958-04-16", "1937-10-16"],
    "Age": [37, 61]
  }
)
```

```
scientists.loc[1, "Name"] = np.NaN
scientists.loc[1, "Age"] = np.NaN

print(scientists.dtypes)
```

```
Name          object
Occupation    object
Born          object
Died          object
Age          float64
dtype: object
```

Since `pd.NA` is still experimental, best follow up with its behavior in the official documentation.

Conclusion

It is rare to have a data set without any missing values. It is important to know how to work with missing values because, even when you are working with data that is complete, missing values can still arise from your own data munging. In this chapter, we examined some of the basic methods used in the data analysis process that pertain to data validity. By looking at your data and tabulating missing values, you can start the process of assessing whether the data is of sufficiently high quality for making decisions and drawing inferences.