Handling NaNs

`NaN` (Not a Number) is a special value used to indicate missing data in many scientific programming languages.

Using `NaN` instead of a numerical dummy value like 9999 or 0 is helpful because Python functions either ignore NaNs by default, or can be set to ignore NaNs using an optional function argument.

In this section we will review:

- Why `NaN` is better than a numerical dummy value
- How to check for NaNs in a dataframe
- Setting the NaN-handling in Python functions

Set up Python Libraries

As usual you will need to run this code block to import the relevant Python libraries

In [11]:
```
# Set-up Python libraries - you need to run this but you don't need to ch
import numpy as np
import matplotlib.pyplot as plt
import scipy.stats as stats
import pandas as pd
import seaborn as sns
sns.set_theme(style='white')
import statsmodels.api as sm
import statsmodels.formula.api as smf
```

Import a dataset to work with

We again work with the NYC heart attack dataset

In [12]:
```
hospital=pd.read_csv('https://raw.githubusercontent.com/jillxoreilly/Stat
display(hospital)
```

	CHARGES	LOS	AGE	SEX	DRG	DIED
0	4752.00	10	79.0	F	122.0	0.0
1	3941.00	6	34.0	F	122.0	0.0
2	3657.00	5	76.0	F	122.0	0.0
3	1481.00	2	80.0	F	122.0	0.0
4	1681.00	1	55.0	M	122.0	0.0
...	...	...	...	...	...	...
12839	22603.57	14	79.0	F	121.0	0.0
12840	NaN	7	91.0	F	121.0	0.0
12841	14359.14	9	79.0	F	121.0	0.0
12842	12986.00	5	70.0	M	121.0	0.0
12843	NaN	1	81.0	M	123.0	1.0

12844 rows × 6 columns

NaN is not a number!

Humans may recognise dummy values like 9999 for what they are, but the computer will treat them as numbers.

Say we want to find the mean and standard deviation of the age of patients in out hospital dataset (remembering that issing data were coded as 9999):

In [13]:
```
print(hospital.AGE.mean())
print(hospital.AGE.std())
```

```
67.11672636660957
88.92599753124392
```

Think is the value for standard deviation realistic?

These values include the 9999s just as if there were really people 9999 years old in the sample.

If we replace the 9999s with `NaN` we get the correct mean and standard deviation for the 'real' values, excluding the missing data

In [22]:
```
hospital.AGE.replace(9999, np.nan, inplace=True)
print(hospital.AGE.mean())
print(hospital.AGE.std())
```

```
66.34327544583755
774.0
```

The mean has changed slightly, and the standard deviation is now much more reasonable.

Creating NaNs

If we want to set a value to `NaN`, we can't just type NaN or "NaN"

Istead, we 'create' the value `NaN` using the `numpy` function `np.nan`, for example:

```
hospital.loc[1, 'CHARGES']=np.nan # set the value of CHARGES in
row 2 to be `NaN`
```

Check for NaNs

- `df.isna()`
- `df.isna().sum()`

`NaN`s are ignored by many Python functions, however you may still want to know if there were any (and how many) in any given set of data.

To check for missing values, coded as `NaN`, we use the function `df.isna()`:

In [16]:
```
hospital.AGE.isna()
```

Out[16]:
```
0        False
1        False
2        False
3        False
4        False
         ...
12839    False
12840    False
12841    False
12842    False
12843    False
Name: AGE, Length: 12844, dtype: bool
```

`df.isna()` returned a column with `True` or `False` for each value of AGE - `True` for people where the age is coded as `NaN` and `False` otherwise.

This isn't very readable, but if we want to know how many `NaN`s were in the column, we can use a trick: Python treats `True` as `1` and `False` as `0`. So if we just take the sum of the column, we get the total nuber of `NaN`s:

In [17]:
```
hospital.AGE.isna().sum()
```

Out[17]:
```
2
```

Two people's age was coded as `NaN`.

NaN handling by Python functions

Many Python functions automatically ignore NaNs.

These include

- `df.mean()`
- `df.std()`
- `df.quantile()` .... and most other descriptive statistics
- `sns.histogram()`
- `sns.scatter()` ... and most other `Seaborn` and `Matplotlib` functions

However, some functions do *not* automatically ignore `NaN`s, and instead will give an error message, or return the value `NaN`, if the input data contains `NaN`s.

This includes a lot of functions from the library `scipy.stats`, which we will use later in the course. For example, say I want to use a t-test to ask if the male patients are older than the females

- don't worry if you don't yet know what a t-test is - this will make sense when you return to it for revision

In [34]:
```
stats.ttest_ind(hospital.query('SEX == "M"').AGE, hospital.query('SEX ==
```

Out[34]:
```
103.0
```

The function `stats.ttest_ind()` performs an independent samples t-test between the two samples we gave it (the ages of male and female patients) and should return a t-value (`statistic`) and a p value (`pvalue`)

Right now both of these are `NaN` because the `NaN`s in the input were not ignored.

We can tell the function `stats.ttest_ind()` to ignore `NaN`s, using the argumment `nan_policy='omit'`:

In [33]:
```
stats.ttest_ind(hospital.query('SEX == "M"').AGE, hospital.query('SEX ==
```

Out[33]:
```
Ttest_indResult(statistic=-35.41617555682539, pvalue=3.1864909732541125e-
262)
```

Now we have actual values instead of NaN: $t = -35.4$ and $p = 3.1x10^{-262}$ (a very small number)

If you run a Python function and the output is `NaN`, you very probably need to change how the function handles `NaN`s using an argument. Check the function's help page online to get the correct syntax.

In [ ]: