CM4125 Topic 5: Data Manipulation (Part 1)

Lecture objectives

Lecture objectives

1. Load a set of datasets containing different characteristics

Lecture objectives

1. Load a set of datasets containing different characteristics

2. Understand the best methods to manipulate it prior to analysis

Lecture objectives

1. Load a set of datasets containing different characteristics
2. Understand the best methods to manipulate it prior to analysis
3. Apply more elaborated pre-processing techniques to clean data and fill in missing values

Data by Columns

In [1]:

```
# Importing a dataset that I have in dropbox related to Time magazine MoTY
import pandas as pd
time = pd.read_csv('https://www.dropbox.com/scl/fi/9olvhtj9zwdj857ol7fdk/times
time
```

Out[1]:

	Year	Honor	Name	Country	Birth Year	Death Year	Title	Category
0	1927	Man of the Year	Charles Lindbergh	United States	1902.0	1974.0	US Air Mail Pilot	NaN
1	1928	Man of the Year	Walter Chrysler	United States	1875.0	1940.0	Founder of Chrysler	Economics
2	1929	Man of the Year	Owen D. Young	United States	1874.0	1962.0	Member of the German Reparations International...	Diplomacy
3	1930	Man of the Year	Mahatma Gandhi	India	1869.0	1948.0	NaN	Revolution
4	1931	Man of the Year	Pierre Laval	France	1883.0	1945.0	Prime Minister of France	Politics
...	...	...	...	...	...	...	...	...
92	2018	Person of the Year	The Guardians	NaN	NaN	NaN	NaN	Media
93	2019	Person of the	Greta Thunberg	Sweden	2003.0	NaN	Environmental activist	Environment

In [1]:

```
# Importing a dataset that I have in dropbox related to Time magazine MoTY
import pandas as pd
time = pd.read_csv('https://www.dropbox.com/scl/fi/9olvhtj9zwdj857ol7fdk/times
time
```

Out[1]:

	Year	Honor	Name	Country	Birth Year	Death Year	Title	Category
0	1927	Man of the Year	Charles Lindbergh	United States	1902.0	1974.0	US Air Mail Pilot	NaN
1	1928	Man of the Year	Walter Chrysler	United States	1875.0	1940.0	Founder of Chrysler	Economics
2	1929	Man of the Year	Owen D. Young	United States	1874.0	1962.0	Member of the German Reparations International...	Diplomacy
3	1930	Man of the Year	Mahatma Gandhi	India	1869.0	1948.0	NaN	Revolution
4	1931	Man of the Year	Pierre Laval	France	1883.0	1945.0	Prime Minister of France	Politics
...	...	...	...	...	...	...	...	...
92	2018	Person of the Year	The Guardians	NaN	NaN	NaN	NaN	Media
93	2019	Person of the	Greta Thunberg	Sweden	2003.0	NaN	Environmental activist	Environment

Last week, we added a new column to a dataframe (the "watched" column at the end of the Netflix dataset)

Last week, we added a new column to a dataframe (the "watched" column at the end of the Netflix dataset)

Today, we will learn other ways in which we can manipulate, add or relate data from columns

Mapping Columns

Mapping Columns

Sometimes we want to convert a column of categorical data from strings (text) to integers (numbers)

Mapping Columns

Sometimes we want to convert a column of categorical data from strings (text) to integers (numbers)

We can define a `dictionary` which maps the strings (as keys) to integers (as values) using the `map` method

Mapping Columns

Sometimes we want to convert a column of categorical data from strings (text) to integers (numbers)

We can define a `dictionary` which maps the strings (as keys) to integers (as values) using the `map` method

Another option is to define a `set` of the unique categories as follows.

In [2]:
```
categories = set(time['Category'])
categories
```

Out[2]:
```
{'Diplomacy',
 'Economics',
 'Environment',
 'Media',
 'Philanthropy',
 'Politics',
 'Religion',
 'Revolution',
 'Science',
 'Space',
 'Technology',
 'War',
 nan}
```

Then, we can then assign unique numbers to this `set` as a `dictionary` (defined in Python using the `{}`)

Then, we can then assign unique numbers to this `set` as a `dictionary` (defined in Python using the `{}`)

In [3]:
```
mapping = {category : index for index, category in enumerate(categories)}
mapping
```

Out[3]:
```
{'Diplomacy': 0,
 'Economics': 1,
 'Space': 2,
 'War': 3,
 'Politics': 4,
 'Philanthropy': 5,
 'Environment': 6,
 'Media': 7,
 'Religion': 8,
 nan: 9,
 'Technology': 10,
 'Science': 11,
 'Revolution': 12}
```

We now have a `map` which allows us to convert from a category string to an `int`. For example:

We now have a `map` which allows us to convert from a category string to an `int`. For example:

In [4]:
```
mapping['Diplomacy']
```

Out[4]:
```
0
```

We now have a `map` which allows us to convert from a category string to an `int`. For example:

In [4]:
```
mapping['Diplomacy']
```

Out[4]:
```
0
```

In [5]:
```
mapping['Media']
```

Out[5]:
```
7
```

We can use the `map` method on the chosen column to do the conversion

We can use the `map` method on the chosen column to do the conversion

The `map` method takes our `dictionary` as an argument and returns a new column with the mapping applied

In [6]:
```
time['Category'].map(mapping)
```

Out[6]:
```
0      9
1      1
2      0
3     12
4      4
      ..
92     7
93     6
94     4
95    10
96     4
Name: Category, Length: 97, dtype: int64
```

If we wanted to overwrite the `Category` column with our new mapped values, we could do:

```
time['Category'] = time['Category'].map(mapping)
```

If we wanted to overwrite the `Category` column with our new mapped values, we could do:

```
time['Category'] = time['Category'].map(mapping)
```

However, it may be best to keep both. So we can instead assign it a different column name

In [7]:

```
time['Category Ordinal'] = time['Category'].map(mapping)
time
```

Out[7]:

	Year	Honor	Name	Country	Birth Year	Death Year	Title	Category
0	1927	Man of the Year	Charles Lindbergh	United States	1902.0	1974.0	US Air Mail Pilot	NaN
1	1928	Man of the Year	Walter Chrysler	United States	1875.0	1940.0	Founder of Chrysler	Economics
2	1929	Man of the Year	Owen D. Young	United States	1874.0	1962.0	Member of the German Reparations International...	Diplomacy
3	1930	Man of the Year	Mahatma Gandhi	India	1869.0	1948.0	NaN	Revolution
4	1931	Man of the Year	Pierre Laval	France	1883.0	1945.0	Prime Minister of France	Politics
...	...	...	...	...	...	...	...	...
92	2018	Person of the Year	The Guardians	NaN	NaN	NaN	NaN	Media
93	2019	Person of the	Greta Thunberg	Sweden	2003.0	NaN	Environmental activist	Environment

Manipulating Numerical Columns

We can perform arithmetic between columns as if they were simply numbers

We can perform arithmetic between columns as if they were simply numbers

We will take the column `Death Year`, subtract the column `Birth Year`, and storing the result in a (new) column called `Lifespan`

In [8]:

```
time['Lifespan'] = time['Death Year'] - time['Birth Year']
time
```

Out[8]:

	Year	Honor	Name	Country	Birth Year	Death Year	Title	Category
0	1927	Man of the Year	Charles Lindbergh	United States	1902.0	1974.0	US Air Mail Pilot	NaN
1	1928	Man of the Year	Walter Chrysler	United States	1875.0	1940.0	Founder of Chrysler	Economics
2	1929	Man of the Year	Owen D. Young	United States	1874.0	1962.0	Member of the German Reparations International...	Diplomacy
3	1930	Man of the Year	Mahatma Gandhi	India	1869.0	1948.0	NaN	Revolution
4	1931	Man of the Year	Pierre Laval	France	1883.0	1945.0	Prime Minister of France	Politics
...	...	...	...	...	...	...	...	...
92	2018	Person of the Year	The Guardians	NaN	NaN	NaN	NaN	Media
93	2019	Person of the	Greta Thunberg	Sweden	2003.0	NaN	Environmental activist	Environment

Notice that the method acknowledged that some columns were `NaN` (since the person is still alive) and thus put a `NaN` as a result in the lifespan column.

Notice that the method acknowledged that some columns were `NaN` (since the person is still alive) and thus put a `NaN` as a result in the lifespan column.

Sometimes you want to do "more difficult" numerical conversions

Notice that the method acknowledged that some columns were `NaN` (since the person is still alive) and thus put a `NaN` as a result in the lifespan column.

Sometimes you want to do "more difficult" numerical conversions

In those cases, it is more convenient to define a **function** and then use the `apply` method

In this example, we will define a function to convert years to decades to be applied in the different columns that contain years

In this example, we will define a function to convert years to decades to be applied in the different columns that contain years

In [9]:
```
def to_decade(value):
    return 10 * (value // 10) # // is a floor division
```

First, we should test our function on some examples to ensure the function is correct.

First, we should test our function on some examples to ensure the function is correct.

In [10]:
```
to_decade(1971)
```

Out[10]:
```
1970
```

Now, we can use `apply` to transform the entire column `Birth Year` into the corresponding decade

Now, we can use `apply` to transform the entire column `Birth Year` into the corresponding decade

In [11]:
```
time['Birth Decade'] = time['Birth Year'].apply(to_decade)
time
```

Out[11]:

	Year	Honor	Name	Country	Birth Year	Death Year	Title	Category
0	1927	Man of the Year	Charles Lindbergh	United States	1902.0	1974.0	US Air Mail Pilot	NaN
1	1928	Man of the Year	Walter Chrysler	United States	1875.0	1940.0	Founder of Chrysler	Economics
2	1929	Man of the Year	Owen D. Young	United States	1874.0	1962.0	Member of the German Reparations International...	Diplomacy
3	1930	Man of the Year	Mahatma Gandhi	India	1869.0	1948.0	NaN	Revolution
4	1931	Man of the Year	Pierre Laval	France	1883.0	1945.0	Prime Minister of France	Politics
...	...	...	...	...	...	...	...	...
92	2018	Person of the Year	The Guardians	NaN	NaN	NaN	NaN	Media
93	2019	Person of the	Greta Thunberg	Sweden	2003.0	NaN	Environmental activist	Environment

Manipulating String Columns

`apply` can also be used on strings

`apply` can also be used on strings

For example, we can apply the `len` function to `Name` to get the number of characters in each name.

`apply` can also be used on strings

For example, we can apply the `len` function to `Name` to get the number of characters in each name.

In [12]:
```
time['Name Length'] = time['Name'].apply(len)
time
```

Out[12]:

	Year	Honor	Name	Country	Birth Year	Death Year	Title	Category
0	1927	Man of the Year	Charles Lindbergh	United States	1902.0	1974.0	US Air Mail Pilot	NaN
1	1928	Man of the Year	Walter Chrysler	United States	1875.0	1940.0	Founder of Chrysler	Economics
2	1929	Man of the Year	Owen D. Young	United States	1874.0	1962.0	Member of the German Reparations International...	Diplomacy
3	1930	Man of the Year	Mahatma Gandhi	India	1869.0	1948.0	NaN	Revolution
4	1931	Man of the Year	Pierre Laval	France	1883.0	1945.0	Prime Minister of France	Politics
...	...	...	...	...	...	...	...	...
92	2018	Person of the Year	The Guardians	NaN	NaN	NaN	NaN	Media
93	2019	Person of the	Greta Thunberg	Sweden	2003.0	NaN	Environmental activist	Environment

Another example, we can create a function to get the initials from a name.

Another example, we can create a function to get the initials from a name.

In [13]:
```
# First we define the function
def to_initials(name):
    import numpy as np # we need this for Python to know what NaN is!
    if name == np.nan:
        return np.nan
    else:
        initials = ""
        for word in name.split(' '):
            first_letter = word[0]
            initials += first_letter
        return initials
    # Then, we try it
to_initials("Barack Obama")
```

Out[13]:
```
'BO'
```

Applying this gives us a name column with the person's initials.

Applying this gives us a name column with the person's initials.

In [14]:
```
time['Initials'] = time['Name'].apply(to_initials)
time
```

Out[14]:

	Year	Honor	Name	Country	Birth Year	Death Year	Title	Category
0	1927	Man of the Year	Charles Lindbergh	United States	1902.0	1974.0	US Air Mail Pilot	NaN
1	1928	Man of the Year	Walter Chrysler	United States	1875.0	1940.0	Founder of Chrysler	Economics
2	1929	Man of the Year	Owen D. Young	United States	1874.0	1962.0	Member of the German Reparations International...	Diplomacy
3	1930	Man of the Year	Mahatma Gandhi	India	1869.0	1948.0	NaN	Revolution
4	1931	Man of the Year	Pierre Laval	France	1883.0	1945.0	Prime Minister of France	Politics
...	...	...	...	...	...	...	...	...
92	2018	Person of the Year	The Guardians	NaN	NaN	NaN	NaN	Media
93	2019	Person of the	Greta Thunberg	Sweden	2003.0	NaN	Environmental activist	Environment

Removing Columns (Unwanted Variables)

There are two main ways to remove them, either we specify which columns we want to **keep** or we specify which we want to **remove**

There are two main ways to remove them, either we specify which columns we want to **keep** or we specify which we want to **remove**

Here is the version where we specify, by name, which columns to **keep**.

There are two main ways to remove them, either we specify which columns we want to **keep** or we specify which we want to **remove**

Here is the version where we specify, by name, which columns to **keep**.

In [15]:
```
time[['Year', 'Honor', 'Name']]
```

Out[15]:

	Year	Honor	Name
0	1927	Man of the Year	Charles Lindbergh
1	1928	Man of the Year	Walter Chrysler
2	1929	Man of the Year	Owen D. Young
3	1930	Man of the Year	Mahatma Gandhi
4	1931	Man of the Year	Pierre Laval
...	...	...	...
92	2018	Person of the Year	The Guardians
93	2019	Person of the Year	Greta Thunberg
94	2020	Person of the Year	Joe Biden & Kamala Harris
95	2021	Person of the Year	Elon Musk
96	2022	Person of the Year	Volodymyr Zelensky

97 rows × 3 columns

Here is the version where we specify, by name, which columns to **remove**

Here is the version where we specify, by name, which columns to **remove**

In [16]:
```
time.drop(columns=['Year', 'Honor', 'Name'])
```

Out[16]:

	Country	Birth Year	Death Year	Title	Category	Context	Category Ordinal	L
0	United States	1902.0	1974.0	US Air Mail Pilot	NaN	First Solo Transatlantic Flight	9	
1	United States	1875.0	1940.0	Founder of Chrysler	Economics	Chrysler/ Dodge Merger	1	
2	United States	1874.0	1962.0	Member of the German Reparations International...	Diplomacy	Young Plan	0	
3	India	1869.0	1948.0	NaN	Revolution	Salt March	12	
4	France	1883.0	1945.0	Prime Minister of France	Politics	NaN	4	
...	...	...	...	...	...	...	...	
92	NaN	NaN	NaN	NaN	Media	Journalists Facing Persecusion	7	
93	Sweden	2003.0	NaN	Environmental ctivist	Environment	School Strike for Climate Campaign	6	

To apply the changes, assign the result back to the same `time` variable

To apply the changes, assign the result back to the same `time` variable

In [17]:
```
time = time.drop(columns=['Year', 'Honor', 'Name'])
time
```

Out[17]:

	Country	Birth Year	Death Year	Title	Category	Context	Category Ordinal	L
0	United States	1902.0	1974.0	US Air Mail Pilot	NaN	First Solo Transatlantic Flight	9	
1	United States	1875.0	1940.0	Founder of Chrysler	Economics	Chrysler/ Dodge Merger	1	
2	United States	1874.0	1962.0	Member of the German Reparations International...	Diplomacy	Young Plan	0	
3	India	1869.0	1948.0	NaN	Revolution	Salt March	12	
4	France	1883.0	1945.0	Prime Minister of France	Politics	NaN	4	
...	...	...	...	...	...	...	...	
92	NaN	NaN	NaN	NaN	Media	Journalists Facing Persecusion	7	
93	Sweden	2003.0	NaN	Environmental ctivist	Environment	School Strike for Climate Campaign	6	

Some functions/methods also have a parameter called `inplace` which, if set = True, apply the change to the original variable

Some functions/methods also have a parameter called `inplace` which, if set = True, apply the change to the original variable

In [18]:
```
time.drop(columns=['Lifespan'], inplace=True)
time
```

Out[18]:

	Country	Birth Year	Death Year	Title	Category	Context	Category Ordinal	D
0	United States	1902.0	1974.0	US Air Mail Pilot	NaN	First Solo Transatlantic Flight	9	
1	United States	1875.0	1940.0	Founder of Chrysler	Economics	Chrysler/ Dodge Merger	1	
2	United States	1874.0	1962.0	Member of the German Reparations International...	Diplomacy	Young Plan	0	
3	India	1869.0	1948.0	NaN	Revolution	Salt March	12	
4	France	1883.0	1945.0	Prime Minister of France	Politics	NaN	4	
...	...	...	...	...	...	...	...	
92	NaN	NaN	NaN	NaN	Media	Journalists Facing Persecusion	7	
93	Sweden	2003.0	NaN	Environmental ctivist	Environment	School Strike for Climate Campaign	6	

Renaming Columns

You can do this using a `dictionary`

You can do this using a `dictionary`

In [19]:
```
time = time.rename(columns={'Birth Year' : 'Born',
                            'Death Year' : 'Died'})
time
```

Out[19]:

	Country	Born	Died	Title	Category	Context	Category Ordinal	D
0	United States	1902.0	1974.0	US Air Mail Pilot	NaN	First Solo Transatlantic Flight	9	
1	United States	1875.0	1940.0	Founder of Chrysler	Economics	Chrysler/ Dodge Merger	1	
2	United States	1874.0	1962.0	Member of the German Reparations International...	Diplomacy	Young Plan	0	
3	India	1869.0	1948.0	NaN	Revolution	Salt March	12	
4	France	1883.0	1945.0	Prime Minister of France	Politics	NaN	4	
...	...	...	...	...	...	...	...	
92	NaN	NaN	NaN	NaN	Media	Journalists Facing Persecusion	7	
93	Sweden	2003.0	NaN	Environmental ctivist	Environment	School Strike for Climate Campaign	6	

Missing Data

Fundamentals of missing data

In the previous example, you saw that some entries had `NaN` to fill in for **missing data**

In the previous example, you saw that some entries had `NaN` to fill in for **missing data**

Missing data is one of the most recurrent problems in data analysis

In the previous example, you saw that some entries had `NaN` to fill in for **missing data**

Missing data is one of the most recurrent problems in data analysis

In fact, there are a lot of studies and methods on how to handle it!

We will work with a small dataset called missing.csv (it's the same as data.csv from last week, but now with missing values)

We will work with a small dataset called missing.csv (it's the same as data.csv from last week, but now with missing values)

In [20]:
```
df = pd.read_csv('https://www.dropbox.com/s/5h4k4rszebd6p0n/missing.csv?raw=1'
df
```

Out[20]:

	Name	Age	Height
0	Nick	21.0	1.85
1	Chris	29.0	Unknown
2	Tim	28.0	1.75
3	Ron	NaN	1.81
4	Monica	35.0	Unknown
5	Cassandra	21.0	1.66

The first thing to notice is that the second column is being shown with decimals, even though they were integers in the original file

The first thing to notice is that the second column is being shown with decimals, even though they were integers in the original file

Moreover, the blank entry is displayed as `NaN`

The first thing to notice is that the second column is being shown with decimals, even though they were integers in the original file

Moreover, the blank entry is displayed as `NaN`

`NaN` stands for *Not A Number* as is part of the floating point specification, typically used when a calculation has no valid numerical result

In fact, Python considers `NaN` as a floating point (or `float64`)

In fact, Python considers `NaN` as a floating point (or `float64`)

In [21]:
```
df
```

Out[21]:

	Name	Age	Height
0	Nick	21.0	1.85
1	Chris	29.0	Unknown
2	Tim	28.0	1.75
3	Ron	NaN	1.81
4	Monica	35.0	Unknown
5	Cassandra	21.0	1.66

In fact, Python considers `NaN` as a floating point (or `float64`)

In [21]:
```
df
```

Out[21]:

	Name	Age	Height
0	Nick	21.0	1.85
1	Chris	29.0	Unknown
2	Tim	28.0	1.75
3	Ron	NaN	1.81
4	Monica	35.0	Unknown
5	Cassandra	21.0	1.66

In [22]:
```
df.at[3, 'Age']
```

Out[22]:
```
np.float64(nan)
```

As a result, the entire column has been turned into `float64`, instead of `int64`

As a result, the entire column has been turned into `float64`, instead of `int64`

This is why it was being shown as decimals!

As a result, the entire column has been turned into `float64`, instead of `int64`

This is why it was being shown as decimals!

In [23]:
```
df['Age']
```

Out[23]:
```
0    21.0
1    29.0
2    28.0
3     NaN
4    35.0
5    21.0
Name: Age, dtype: float64
```

Incorrectly Imported Missing Data

The missing value in the `Age` column was generated because the cell was blank

The missing value in the `Age` column was generated because the cell was blank

By default, `Pandas` will treat certain cell values, such as a *blank,* `NULL`, `NaN`, and `n/a` as missing values

The missing value in the `Age` column was generated because the cell was blank

By default, `Pandas` will treat certain cell values, such as a *blank,* `NULL`, `NaN`, and `n/a` as missing values

The `Height` column contains some entries which we recognise as missing data (i.e. `Unknown`) but that Pandas did not

The missing value in the `Age` column was generated because the cell was blank

By default, `Pandas` will treat certain cell values, such as a *blank*, `NULL`, `NaN`, and `n/a` as missing values

The `Height` column contains some entries which we recognise as missing data (i.e. `Unknown`) but that Pandas did not

In [24]:
```
df.at[1, 'Height']
```

Out[24]:
```
'Unknown'
```

In fact, the entire column has been imported as strings, not numbers!

In fact, the entire column has been imported as strings, not numbers!

In [25]:
```
df
```

Out[25]:

	Name	Age	Height
0	Nick	21.0	1.85
1	Chris	29.0	Unknown
2	Tim	28.0	1.75
3	Ron	NaN	1.81
4	Monica	35.0	Unknown
5	Cassandra	21.0	1.66

In fact, the entire column has been imported as strings, not numbers!

In [25]:
```
df
```

Out[25]:

	Name	Age	Height
0	Nick	21.0	1.85
1	Chris	29.0	Unknown
2	Tim	28.0	1.75
3	Ron	NaN	1.81
4	Monica	35.0	Unknown
5	Cassandra	21.0	1.66

In [26]:
```
print(df.at[1, 'Height'])
print(type(df.at[1, 'Height']))
```

```
Unknown
<class 'str'>
```

This in turn made `Pandas` believe that the whole column has strings!

This in turn made `Pandas` believe that the whole column has strings!

In [27]:
```
print(df.at[0, 'Height'])
print(type(df.at[0, 'Height']))
```

```
1.85
<class 'str'>
```

What happens if we multiply this column by `100`?

What happens if we multiply this column by `100` ?

In [28]:
```
df['Height times 100'] = df['Height'] * 100
df
```

Out[28]:

	Name	Age	Height	Height time
0	Nick	21.0	1.85	1.851.851.851.851.851.851.851.851.851.
1	Chris	29.0	Unknown	UnknownUnknownUnknownUnknownUnknownUnknownU
2	Tim	28.0	1.75	1.751.751.751.751.751.751.751.751.751.
3	Ron	NaN	1.81	1.811.811.811.811.811.811.811.811.811.
4	Monica	35.0	Unknown	UnknownUnknownUnknownUnknownUnknownUnknownU
5	Cassandra	21.0	1.66	1.661.661.661.661.661.661.661.661.661.

We need to tell Pandas in advance to treat `Unknown` not as a string, but as a missing value

We need to tell Pandas in advance to treat `Unknown` not as a string, but as a missing value

Lets re-import the data set passing a list of values which represent missing values as the `na_values` parameter

We need to tell Pandas in advance to treat `Unknown` not as a string, but as a missing value

Lets re-import the data set passing a list of values which represent missing values as the `na_values` parameter

In [29]:
```
df = pd.read_csv('https://www.dropbox.com/s/5h4k4rszebd6p0n/missing.csv?raw=1'
                 na_values=['Unknown'])
df
```

Out[29]:

	Name	Age	Height
0	Nick	21.0	1.85
1	Chris	29.0	NaN
2	Tim	28.0	1.75
3	Ron	NaN	1.81
4	Monica	35.0	NaN
5	Cassandra	21.0	1.66

Now the column is `float64`

Now the column is `float64`

In [30]:
```
df['Height']
```

Out[30]:
```
0    1.85
1     NaN
2    1.75
3    1.81
4     NaN
5    1.66
Name: Height, dtype: float64
```

Multiplying `Height` now works as expected.

Multiplying `Height` now works as expected.

In [31]:
```
df['Height times 100'] = df['Height'] * 100
df
```

Out[31]:

	Name	Age	Height	Height times 100
0	Nick	21.0	1.85	185.0
1	Chris	29.0	NaN	NaN
2	Tim	28.0	1.75	175.0
3	Ron	NaN	1.81	181.0
4	Monica	35.0	NaN	NaN
5	Cassandra	21.0	1.66	166.0

In [32]:
```
# Another example
df['Test'] = df['Age'] + df['Height']
df
```

Out[32]:

	Name	Age	Height	Height times 100	Test
0	Nick	21.0	1.85	185.0	22.85
1	Chris	29.0	NaN	NaN	NaN
2	Tim	28.0	1.75	175.0	29.75
3	Ron	NaN	1.81	181.0	NaN
4	Monica	35.0	NaN	NaN	NaN
5	Cassandra	21.0	1.66	166.0	22.66

Methods to address missing data

As mentioned before, there are different ways to deal with missing data

As mentioned before, there are different ways to deal with missing data

The method you should use depends on your analysis, the application domain and what you want to achieve

As mentioned before, there are different ways to deal with missing data

The method you should use depends on your analysis, the application domain and what you want to achieve

Here are some common methods:

```
1. Leave the missing values as NaN
2. Lookup the values from another data source
3. Delete the rows / observations
4. Compute reasonable guesses about the values
```

In **option 1**, it is often reasonable to just leave the data as missing data

In **option 1**, it is often reasonable to just leave the data as missing data

For example, if we wanted to calculate average `Height`, it may not matter that we are missing `Age` for one row

In **option 1**, it is often reasonable to just leave the data as missing data

For example, if we wanted to calculate average `Height`, it may not matter that we are missing `Age` for one row

As for **option 2**, if we have some other ways of going back and getting correct data, we could just update the values with `df.at[3, 'Age'] = ...`

In **option 1**, it is often reasonable to just leave the data as missing data

For example, if we wanted to calculate average `Height`, it may not matter that we are missing `Age` for one row

As for **option 2**, if we have some other ways of going back and getting correct data, we could just update the values with `df.at[3, 'Age'] = ...`

Let's assume we are going for **option 3** and want to remove any rows where we do no have `Age`

If we know the row we want to delete (`3`), we can use the `drop` method, only this time we specify rows, not columns. As in:

```
df.drop(3)
```

If we know the row we want to delete (`3`), we can use the `drop` method, only this time we specify rows, not columns. As in:

```
df.drop(3)
```

However, often there are many missing values, and we don't want to check on the index of each

If we know the row we want to delete (`3`), we can use the `drop` method, only this time we specify rows, not columns. As in:

```
df.drop(3)
```

However, often there are many missing values, and we don't want to check on the index of each

A good way to remove rows with missing data is to specify which rows to **keep**

The `isnull` method will filter for rows which contains `NaN`.

The `isnull` method will filter for rows which contains `NaN`.

In [33]:
```
df[df['Age'].isnull()]
```

Out[33]:

	Name	Age	Height	Height times 100	Test
3	Ron	NaN	1.81	181.0	NaN

Of course, this is the *opposite* of specifying which to keep

Of course, this is the *opposite* of specifying which to keep

The **notnull** method will show us which rows do not contain `NaN` in the specified column.

Of course, this is the *opposite* of specifying which to keep

The **notnull** method will show us which rows do not contain `NaN` in the specified column.

In [34]:
```
df[df['Age'].notnull()]
```

Out[34]:

	Name	Age	Height	Height times 100	Test
0	Nick	21.0	1.85	185.0	22.85
1	Chris	29.0	NaN	NaN	NaN
2	Tim	28.0	1.75	175.0	29.75
4	Monica	35.0	NaN	NaN	NaN
5	Cassandra	21.0	1.66	166.0	22.66

Notice that the filter is only checking the column we specified `Age` , not the other column with `NaN` values

Notice that the filter is only checking the column we specified `Age` , not the other column with `NaN` values

We can apply the filter by assigning the result back to `df`

Notice that the filter is only checking the column we specified `Age` , not the other column with `NaN` values

We can apply the filter by assigning the result back to `df`

In [35]:
```
df = df[df['Age'].notnull()]
df
```

Out[35]:

	Name	Age	Height	Height times 100	Test
0	Nick	21.0	1.85	185.0	22.85
1	Chris	29.0	NaN	NaN	NaN
2	Tim	28.0	1.75	175.0	29.75
4	Monica	35.0	NaN	NaN	NaN
5	Cassandra	21.0	1.66	166.0	22.66

Since the `Age` column no longer contains any non-integers, it is safe to convert the column to `int64` type

Since the `Age` column no longer contains any non-integers, it is safe to convert the column to `int64` type

We can do this with the `astype` method

Since the `Age` column no longer contains any non-integers, it is safe to convert the column to `int64` type

We can do this with the `astype` method

In [36]:
```
df = df.astype({'Age': 'int64'})
df
```

Out[36]:

	Name	Age	Height	Height times 100	Test
0	Nick	21	1.85	185.0	22.85
1	Chris	29	NaN	NaN	NaN
2	Tim	28	1.75	175.0	29.75
4	Monica	35	NaN	NaN	NaN
5	Cassandra	21	1.66	166.0	22.66

Updating Indices

After these operations, the current data frame has the index 0, 1, 2, 4, 5

After these operations, the current data frame has the index 0, 1, 2, 4, 5

We could leave it like this, being careful to select the correct rows with `loc` . However, we can also reset the index with `reset_index`

Note that we need to specify `drop=True`, without this we will end up with an extra column with the old index

Note that we need to specify `drop=True` , without this we will end up with an extra column with the old index

In [37]:
```
df = df.reset_index(drop=True)
df
```

Out[37]:

	Name	Age	Height	Height times 100	Test
0	Nick	21	1.85	185.0	22.85
1	Chris	29	NaN	NaN	NaN
2	Tim	28	1.75	175.0	29.75
3	Monica	35	NaN	NaN	NaN
4	Cassandra	21	1.66	166.0	22.66

Alternatively, we can set the index to a particular column if we don't want default indexing

Alternatively, we can set the index to a particular column if we don't want default indexing

In [38]:
```
df = df.set_index('Name')
df
```

Out[38]:

Name	Age	Height	Height times 100	Test
Nick	21	1.85	185.0	22.85
Chris	29	NaN	NaN	NaN
Tim	28	1.75	175.0	29.75
Monica	35	NaN	NaN	NaN
Cassandra	21	1.66	166.0	22.66

Say we want to go with **option 4** and replace missing `Height` with a computed value

Say we want to go with **option 4** and replace missing `Height` with a computed value

In this example we simply calculate the mean height of other people in the data set (to the nearest cm)

Say we want to go with **option 4** and replace missing `Height` with a computed value

In this example we simply calculate the mean height of other people in the data set (to the nearest cm)

Note: The `mean` method just ignores the missing data and gives us the average of the non-missing data, which is what we want

In [39]:
```
mean_height = df['Height'].mean()
mean_height
```

Out[39]:
```
np.float64(1.7533333333333332)
```

In [39]:
```
mean_height = df['Height'].mean()
mean_height
```

Out[39]:
```
np.float64(1.7533333333333332)
```

In [40]:
```
mean_height = round(mean_height, 2)
mean_height
```

Out[40]:
```
np.float64(1.75)
```

In [39]:
```
mean_height = df['Height'].mean()
mean_height
```

Out[39]:
```
np.float64(1.7533333333333332)
```

In [40]:
```
mean_height = round(mean_height, 2)
mean_height
```

Out[40]:
```
np.float64(1.75)
```

Of course there are many more sophisticated ways to predict a replacement value!

For instance, could fit a regression with Age

For instance, could fit a regression with `Age`

Or if we had categories such as `Gender` we could group by categories and do predictions for each

For instance, could fit a regression with `Age`

Or if we had categories such as `Gender` we could group by categories and do predictions for each

In this example we will just use `1.75` for every missing value.

In [41]:
```
df = df.fillna({'Height' : mean_height, 'Height times 100':175})
df
```

Out[41]:

Name	Age	Height	Height times 100	Test
Nick	21	1.85	185.0	22.85
Chris	29	1.75	175.0	NaN
Tim	28	1.75	175.0	29.75
Monica	35	1.75	175.0	NaN
Cassandra	21	1.66	166.0	22.66

Lab