Part 2 — Workshop 2: Introduction to pandas

TECH2: Introduction to Programming, Data, and Information Technology

Richard Foltyn
NHH Norwegian School of Economics

October 3, 2025

Exercise 1: Data cleaning

Before doing actual data analysis, we usually first need to clean the data. This might involve steps such as dealing with missing values and encoding categorical variables as integers. In this exercise, you will perform such steps based on the Titanic passenger data.

1. Load the Titanic data set in `titanic.csv` located in the `data/` folder.
2. Report the number of observations with missing `Age`, for example using `isna()`.
3. Compute the average age in the data set. Use the following approaches and compare your results:
    1. Use pandas's `mean()` method.
    2. Convert the `Age` column to a NumPy array using `to_numpy()`. Experiment with NumPy's `np.mean()` and `np.nanmean()` to see if you obtain the same results.
4. Replace the all missing ages with the mean age you computed above, rounded to the nearest integer. Note that in "real" applications, replacing missing values with sample means is usually not a good idea.
5. Convert this updated `Age` column to integer type using `astype()`.
6. Generate a new column `Female` which takes on the value one if `Sex` is equal to `"female"` and zero otherwise. This is called an *indicator* or *dummy* variable, and is preferrable to storing such categorical data as strings. Delete the original column `Sex`.
7. Save your cleaned data set as `titanic-clean.csv` using `to_csv()` with `,` as the field separator. Tell `to_csv()` to *not* write the `DataFrame` index to the CSV file as it's not needed in this example.

Solution.

Part 1: Loading the data

[1]:
```python
import pandas as pd

# Relative path to data directory
DATA_PATH = '../../data'

# Path to Titanic CSV file
fn = f'{DATA_PATH}/titanic.csv'

# Read in the CSV file, use default separator (comma)
df = pd.read_csv(fn)
```

Part 2: Number of missing values

We can count the number of missing values directly by summing the return values of `isna()` which returns `True` whenever an observation is missing:

[2]:
```
# Number of missing age observations
df['Age'].isna().sum()
```

[2]:
```
np.int64(177)
```

Alternatively, the number of non-missing values can be displayed using the `info()` method. For larger DataFrames, pandas does not automatically report the number of nonmissing ("Non-Null") observations, so we might need to request this explicitly by passing the `show_counts=True` argument.

[3]:
```
# Display missing counts for each column
df.info(show_counts=True)
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 10 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   PassengerId  891 non-null    int64
 1   Survived     891 non-null    int64
 2   Pclass       891 non-null    int64
 3   Name         891 non-null    object
 4   Sex          891 non-null    object
 5   Age          714 non-null    float64
 6   Ticket       891 non-null    object
 7   Fare         891 non-null    float64
 8   Cabin        204 non-null    object
 9   Embarked     889 non-null    object
dtypes: float64(2), int64(3), object(5)
memory usage: 69.7+ KB
```

Part 3: Compute mean age

We compute the mean age using the three different methods. As you can see, `np.mean()` cannot deal with missing values and returns `NaN` (*"not a number"*).

[4]:
```
import numpy as np

# Compute mean age using the DataFrame.mean() method
mean_age = df['Age'].mean()

# Convert Age column to NumPy array
age_array = df['Age'].to_numpy()

# Compute mean using np.mean()
mean_age_np = np.mean(age_array)

# Compute mean using np.nanmean()
mean_age_np_nan = np.nanmean(age_array)

print(f'Mean age using pandas:       {mean_age:.3f}')
print(f'Mean age using np.mean():    {mean_age_np:.3f}')
print(f'Mean age using np.nanmean(): {mean_age_np_nan:.3f}')
```

```
Mean age using pandas:       29.699
Mean age using np.mean():    nan
Mean age using np.nanmean(): 29.699
```

Part 4: Replace missing values

There are several ways to replace missing values. First, we can “manually” identify these using boolean indexing and assign a new value to such observations.

[5]:
```
# Round average age
mean_age = np.round(mean_age)

# boolean arrays to select missing observations
is_missing = df['Age'].isna()

# Update missing observations with rounded mean age
df.loc[is_missing, 'Age'] = mean_age
```

There is also the convenience routine `fillna()` which automates this step. To illustrate, we need to reload the original data as we have just replaced all missing values.

[6]:
```
# Re-load data to get the original missing values
df = pd.read_csv(fn)

df['Age'] = df['Age'].fillna(value=mean_age)
```

Part 5: Convert age column to integer type

Since age is usually recorded as an integer, there is no reason to store it as a `float` once we have dealt with the missing values.

[7]:
```
df['Age'] = df['Age'].astype(int)
```

Part 6: Generate Female indicator

An indicator variable can be obtained as a result of a logical operation (`==`, `!=`, etc.). This value contains `True` or `False` values, which we can convert to 1 or 0 by changing the data type to integer.

[8]:
```
# Generate boolean array (True/False) whether passenger is female
is_female = (df['Sex'] == 'female')

# Add Female dummy variable, converted to integer
df['Female'] = is_female.astype(int)

# Delete original Sex column, no longer needed
del df['Sex']

# Alternatively, you can use
# df = df.drop(columns=['Sex'])
```

Part 7: Save cleaned file

We can use `info()` again to confirm that `Age` has no missing values and all columns are of the desired data type:

[9]:
```
df.info(show_counts=True)
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 10 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
```

```
0   PassengerId  891 non-null    int64
1   Survived     891 non-null    int64
2   Pclass       891 non-null    int64
3   Name         891 non-null    object
4   Age          891 non-null    int64
5   Ticket       891 non-null    object
6   Fare         891 non-null    float64
7   Cabin        204 non-null    object
8   Embarked     889 non-null    object
9   Female       891 non-null    int64
dtypes: float64(1), int64(5), object(4)
memory usage: 69.7+ KB
```

[10]:
```
# Save cleaned file
fn_clean = f'{DATA_PATH}/titanic-cleaned.csv'

df.to_csv(fn_clean, sep=',', index=False)
```

Exercise 2: Selecting subsets of data

In this exercise, you are asked to select subsets of macroeconomic data for the United States based on some criteria.

1. Load the annual data from FRED which are located in `FRED_annual.xlsx` in the `data/FRED` folder.
2. Print the list of columns and the number of non-missing observations.
3. Since we are dealing with time series data, set the column `Year` as the DataFrame index.
4. Print all observations for the 1960s decade using at least two different methods.
5. Using the data in the column `GDP`, compute the annual GDP growth in percent and store it in the column `GDP_growth`. Select the years in which
   1. GDP growth was above 5%.
   2. GDP growth was negative, but inflation as still above 5% (such episodes are called "stagflation" since usually negative GDP growth is associated with low inflation).

      Use at least two methods to select such years.

   Hint: You can compute changes relative to the previous observation using the `pct_change()` method.

Solution.

Part 1: Loading the data

[11]:
```
# Relative path to data directory
DATA_PATH = '../../data'

# Path to annual FRED data in Excel format
fn = f'{DATA_PATH}/FRED/FRED_annual.xlsx'

# Read in data
df = pd.read_excel(fn)
```

Part 2: Reporting columns and observation count

We print the columns and their corresponding number of observations using the `info()` method. This data set is small enough so that pandas automatically reports the number of observations, but we can still pass `show_counts=True` if we want.

[12]:
```
# List columns, force pandas to show number of observations
df.info(show_counts=True)
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 70 entries, 0 to 69
Data columns (total 6 columns):
 #   Column     Non-Null Count  Dtype  
---  ------     --------------  -----  
 0   Year       70 non-null     int64  
 1   GDP        70 non-null     float64
 2   CPI        70 non-null     float64
 3   UNRATE     70 non-null     float64
 4   FEDFUNDS   70 non-null     float64
 5   INFLATION  69 non-null     float64
dtypes: float64(5), int64(1)
memory usage: 3.4 KB
```

In this case, the result is the same if you call `info()` without additional arguments, but that might not be the case for larger DataFrames, depending on your local configuration settings:

[13]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 70 entries, 0 to 69
Data columns (total 6 columns):
 #   Column     Non-Null Count  Dtype  
---  ------     --------------  -----  
 0   Year       70 non-null     int64  
 1   GDP        70 non-null     float64
 2   CPI        70 non-null     float64
 3   UNRATE     70 non-null     float64
 4   FEDFUNDS   70 non-null     float64
 5   INFLATION  69 non-null     float64
dtypes: float64(5), int64(1)
memory usage: 3.4 KB
```

Part 3: Set **Year** as the index

[14]:
```
# Set Year as index, discard previous (default) index
df = df.set_index('Year')
```

Part 4: Select data from the 1960s

There are various ways to select all observations for the 1960s. Since we have set the `Year` as an index, we can use `.loc[]` to select by label:

[15]:
```
df.loc[1960:1969]
```

[15]:

Year	GDP	CPI	UNRATE	FEDFUNDS	INFLATION
1960	3500.3	29.6	5.5	3.2	1.369863
1961	3590.1	29.9	6.7	2.0	1.013514
1962	3810.1	30.3	5.6	2.7	1.337793
1963	3976.1	30.6	5.6	3.2	0.990099

```
1964  4205.3  31.0     5.2       3.5   1.307190
1965  4478.6  31.5     4.5       4.1   1.612903
1966  4773.9  32.5     3.8       5.1   3.174603
1967  4904.9  33.4     3.8       4.2   2.769231
1968  5145.9  34.8     3.6       5.7   4.191617
1969  5306.6  36.7     3.5       8.2   5.459770
```

Alternatively, we can use the `query()` method which allows us to select observations not only based on column values, but also based on values of the index (`Year` in this case):

[16]:
```
df.query('Year >= 1960 & Year <= 1969')
```

[16]:
```
         GDP   CPI  UNRATE  FEDFUNDS  INFLATION
Year
1960  3500.3  29.6     5.5       3.2   1.369863
1961  3590.1  29.9     6.7       2.0   1.013514
1962  3810.1  30.3     5.6       2.7   1.337793
1963  3976.1  30.6     5.6       3.2   0.990099
1964  4205.3  31.0     5.2       3.5   1.307190
1965  4478.6  31.5     4.5       4.1   1.612903
1966  4773.9  32.5     3.8       5.1   3.174603
1967  4904.9  33.4     3.8       4.2   2.769231
1968  5145.9  34.8     3.6       5.7   4.191617
1969  5306.6  36.7     3.5       8.2   5.459770
```

Part 5: GDP growth

We can use the `pct_change()` method to compute changes relative to the previous year. To convert these changes to percent, we need to multiply the values returned by this method by 100.

[17]:
```
# Compute GDP growth as percentage change in GDP vs. the previous year
df['GDP_growth'] = df['GDP'].pct_change() * 100
```

We can now use this column select episodes when GDP growth was above 5%, for example by using the `query()` method.

[18]:
```
# Select high growth episodes using query()
df.query('GDP_growth > 5')
```

[18]:
```
          GDP    CPI  UNRATE  FEDFUNDS  INFLATION  GDP_growth
Year
1955   3083.0   26.8     4.4       1.8  -0.371747    7.134170
1959   3412.4   29.2     5.4       3.3   1.038062    6.931562
1962   3810.1   30.3     5.6       2.7   1.337793    6.127963
1964   4205.3   31.0     5.2       3.5   1.307190    5.764443
1965   4478.6   31.5     4.5       4.1   1.612903    6.498942
1966   4773.9   32.5     3.8       5.1   3.174603    6.593578
1972   5780.0   41.8     5.6       4.4   3.209877    5.255490
1973   6106.4   44.4     4.9       8.7   6.220096    5.647059
1976   6387.4   56.9     7.7       5.0   5.762082    5.386989
1978   7052.7   65.2     6.1       7.9   7.590759    5.535105
1984   8195.3  103.9     7.5      10.2   4.317269    7.236042
2021  21494.8  271.0     5.4       0.1   4.714065    6.054984
```

Next, we want to select stagflationary periods where the US economy stagnated (GDP growth below 0%), but inflation was nevertheless high (above 5%). We can achieve this by using `query()` with a boolean *and* operator, `&`:

[19]:
```
df.query('GDP_growth < 0 & INFLATION > 5')
```

[19]:

Year	GDP	CPI	UNRATE	FEDFUNDS	INFLATION	GDP_growth
1974	6073.4	49.3	5.6	10.5	11.036036	-0.540417
1975	6060.9	53.8	8.5	5.8	9.127789	-0.205816
1980	7257.3	82.4	7.2	13.4	13.498623	-0.257009
1982	7307.3	96.5	9.7	12.3	6.160616	-1.803400

Alternatively, we can combine two individual boolean Series, one for each condition, and use this to select the relevant years.

[20]:
```
condition = (df['GDP_growth'] < 0) & (df['INFLATION'] > 5)
df[condition]
```

[20]:

Year	GDP	CPI	UNRATE	FEDFUNDS	INFLATION	GDP_growth
1974	6073.4	49.3	5.6	10.5	11.036036	-0.540417
1975	6060.9	53.8	8.5	5.8	9.127789	-0.205816
1980	7257.3	82.4	7.2	13.4	13.498623	-0.257009
1982	7307.3	96.5	9.7	12.3	6.160616	-1.803400

Exercise 3: Labor market statistics for the US

In this exercise, you are asked to compute some descriptive statistics for the unemployment rate and the labor force participation (the fraction of the working-age population in the labor force, i.e., individuals who are either employed or unemployed) for the United States.

1. Load the monthly time series from FRED which are located in `FRED_monthly.csv` in the `data/FRED` folder.

   Hint: You can use `pd.read_csv(..., parse_dates=['DATE'])` to automatically parse strings stored in the `DATE` column as dates.

2. Print the list of columns and the number of non-missing observations.

3. Since we are dealing with time series data, set the column `DATE` as the DataFrame index. Using the date index, select all observations from the first three months of the year 2020.

4. For the columns `UNRATE` (unemployment rate) and `LFPART` (labor force participation), compute and report the mean, minimum and maximum values for the whole sample. Round your results to one decimal digit.

   Hint: You can use the DataFrame methods `mean()`, `min()`, and `max()` to compute the desired statistics.

   Hint: You can use the DataFrame method `round()` to truncate the number of decimal digits.

5. You are interested in how the average unemployment rate evolved over the last few decades.

   - Add a new column `Decade` to the DataFrame which contains the starting year for each decade (e.g., this value should be 1950 for the years 1950-1959, and so on).

     Hint: The decade can be computed from the column `Year` using truncated integer division:

     ```
     df['Year'] // 10 * 10
     ```

   - Write a loop to compute and report the average unemployment rate (column `UNRATE`) for each decade.

     Include only the decades from 1950 to 2010 for which you have all observations.

Solution.

Part 1: Loading the data

When reading CSV files, we can use the `parse_dates` argument to tell pandas which columns should be interpreted as dates and pandas will automatically parse them as date objects. While this is not strictly needed for this exercise, it is good practice to automatically parse dates, in particular if we want to use the date column as the index.

[21]:
```
import pandas as pd

# Relative path to data directory
DATA_PATH = '../../data'

# Path to monthly FRED CSV file
fn = f'{DATA_PATH}/FRED/FRED_monthly.csv'

# Read the CSV file, setting the 'DATE' column as the index and parsing it as a date
df = pd.read_csv(fn, parse_dates=['DATE'])
```

Part 2: Reporting columns and observation count

We print the columns and their corresponding number of observations using the `info()` method. For larger DataFrames, pandas does not automatically report the number of nonmissing ("Non-Null") observations, so we might need to request this explicitly by passing the `show_counts=True` argument.

As you can see, not all variables have the same number of observations, as for example the real interest rate (`REALRATE`) and the Federal Funds Rate (`FEDFUNDS`) are only available for the later years.

[22]:
```
# List columns and number of non-missing observations
df.info(show_counts=True)
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 924 entries, 0 to 923
Data columns (total 8 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   DATE      924 non-null    datetime64[ns]
 1   Year      924 non-null    int64
 2   Month     924 non-null    int64
 3   CPI       924 non-null    float64
 4   UNRATE    924 non-null    float64
 5   FEDFUNDS  846 non-null    float64
 6   REALRATE  516 non-null    float64
 7   LFPART    924 non-null    float64
dtypes: datetime64[ns](1), float64(5), int64(2)
memory usage: 57.9 KB
```

Part 3: Settings `DATE` as the index

[23]:
```
# Set the column DATE as the index, discarding the previous (default) index
df = df.set_index('DATE')
```

Using the date as the index is particularly convenient for time series data. For example, we can select the observations for a specific date or period (such as the first three months of 2020) by directly specifying that date in `.loc[]`.

[24]:
```
df.loc['2020-01':'2020-03']
```

[24]:
```
            Year  Month    CPI  UNRATE  FEDFUNDS  REALRATE  LFPART
DATE
```

```
2020-01-01  2020     1  259.1     3.6       1.6      -0.6     63.3
2020-02-01  2020     2  259.2     3.5       1.6      -0.5     63.3
2020-03-01  2020     3  258.1     4.4       0.6       3.4     62.6
```

Part 4: Labor market statistics for whole sample

There are various ways to solve this part. One approach is to select the columns `UNRATE` and `LFPART` and compute the desired statistics (mean, min, max), and round the result to one decimal digit:

[25]:
```
print("Average:")
print(df[['UNRATE', 'LFPART']].mean().round(1))
```

```
Average:
UNRATE     5.7
LFPART    62.8
dtype: float64
```

[26]:
```
print("Minimum:")
print(df[['UNRATE', 'LFPART']].min().round(1))
```

```
Minimum:
UNRATE     2.5
LFPART    58.1
dtype: float64
```

[27]:
```
print("Maximum:")
print(df[['UNRATE', 'LFPART']].max().round(1))
```

```
Maximum:
UNRATE    14.8
LFPART    67.3
dtype: float64
```

Alternatively, we can perform this task in a single line leveraging to `describe()` method. Recall that `describe()` computes various summary statistics for each column:

[28]:
```
df[['UNRATE', 'LFPART']].describe()
```

[28]:

	UNRATE	LFPART
count	924.000000	924.000000
mean	5.683442	62.837013
std	1.708977	2.902365
min	2.500000	58.100000
25%	4.400000	59.800000
50%	5.500000	62.900000
75%	6.700000	65.900000
max	14.800000	67.300000

The output contains statistics we are not interested in, but we can use `loc[]` to select only the desired rows and round them to one decimal digit:

[29]:
```
df[['UNRATE', 'LFPART']].describe().loc[['mean', 'min', 'max']].round(1)
```

[29]:

	UNRATE	LFPART
mean	5.7	62.8
min	2.5	58.1
max	14.8	67.3

Part 5: Labor market statistics by decade

First, we add a new column `Decade` which contains the starting year of each decade corresponding to the observation date:

[30]:
```
df['Decade'] = df['Year'] // 10 * 10
```

The following table shows that not all decades include the full set of observations:

[31]:
```
df['Decade'].value_counts().sort_index()
```

[31]:
```
Decade
1940     24
1950    120
1960    120
1970    120
1980    120
1990    120
2000    120
2010    120
2020     60
Name: count, dtype: int64
```

We therefore restrict our attention to the decades from 1950 to 2010 and compute the average unemployment rate for each decade:

[32]:
```
decades = np.arange(1950, 2011, 10)

print("Unemployment rate by decade:")
for decade in decades:
    mean = df.loc[df['Decade'] == decade, "UNRATE"].mean()
    print(f"Decade starting in {decade}: {mean:5.1f}")
```

```
Unemployment rate by decade:
Decade starting in 1950:   4.5
Decade starting in 1960:   4.8
Decade starting in 1970:   6.2
Decade starting in 1980:   7.3
Decade starting in 1990:   5.8
Decade starting in 2000:   5.5
Decade starting in 2010:   6.2
```

Note that this task can be achieved much more elegantly using grouping operations which we will study in the next lectures.

Exercise 4: Working with string data (advanced)

Most of the data we deal with contain strings, i.e., text data (names, addresses, etc.). Often, such data is not in the format needed for analysis, and we have to perform additional string manipulation to extract the exact data we need. This can be achieved using the pandas string methods.

To illustrate, we use the Titanic data set for this exercise.

1. Load the Titanic data and restrict the sample to men. (This simplifies the task. Women in this data set have much more complicated names as they contain both their husband's and their maiden name)
2. Print the first five observations of the `Name` column. As you can see, the data is stored in the format *"Last name, Title First name"* where title is something like Mr., Rev., etc.

3. Split the `Name` column by `,` to extract the last name and the remainder as separate columns. You can achieve this using the `partition()` string method.
4. Split the remainder (containing the title and first name) using the space character `" "` as separator to obtain individual columns for the title and the first name.
5. Store the three data series in the original `DataFrame` (using the column names `FirstName`, `LastName` and `Title`) and delete the `Name` column which is no longer needed.
6. Finally, extract the ship deck from the values in `Cabin`. The ship deck is the first character in the string stored in `Cabin` (A, B, C, …). You extract the first character using the `get()` string method. Store the result in the column `Deck`.

Hint: Pandas's string methods can be accessed using the `.str` attribute. For example, to partition values in the column `Name`, you need to use

```
df['Name'].str.partition()
```

Solution.

Part 1: Import data and restrict to male sub-sample

[33]:
```
# Path to data directory
DATA_PATH = '../../data'

# Alternatively, load data directly from GitHub
# DATA_PATH = 'https://raw.githubusercontent.com/richardfoltyn/TECH2-H25/main/data'
```

[34]:
```
import pandas as pd

# Path to Titanic CSV file
fn = f'{DATA_PATH}/titanic.csv'

df = pd.read_csv(fn)
```

We restrict the sample either with boolean indexing of with the `query()` method.

[35]:
```
# Restrict sample to men
df = df.loc[df['Sex'] == 'male'].copy()

# Alternatively, we can do this with a query()
df = df.query('Sex == "male"')
```

Part 2: Inspect the `Name` column

[36]:
```
# Print first 10 Name observations
df['Name'].head(10)
```

[36]:
```
0               Braund, Mr. Owen Harris
4              Allen, Mr. William Henry
5                      Moran, Mr. James
6               McCarthy, Mr. Timothy J
7        Palsson, Master Gosta Leonard
12      Saundercock, Mr. William Henry
13         Andersson, Mr. Anders Johan
16                 Rice, Master Eugene
17        Williams, Mr. Charles Eugene
20                Fynney, Mr. Joseph J
Name: Name, dtype: object
```

Part 3: Split into last name and remainder

Note that `partition()` returns *three* columns, the second on containing the separator you specified. This second column can be ignored.

[37]:
```
# Split names by comma, create DataFrame with a column for each token
names = df['Name'].str.partition(',')

# Print first 5 rows or resulting DataFrame
names.head(5)
```

[37]:
```
          0  1                       2
0    Braund  ,         Mr. Owen Harris
4     Allen  ,       Mr. William Henry
5     Moran  ,               Mr. James
6  McCarthy  ,           Mr. Timothy J
7   Palsson  ,   Master Gosta Leonard
```

[38]:
```
# Extract last name stored in 1st column, strip any remaining white space
last_name = names[0].str.strip()

# Print first 5 observations
last_name.head(5)
```

[38]:
```
0      Braund
4       Allen
5       Moran
6    McCarthy
7     Palsson
Name: 0, dtype: object
```

Part 4: Split title and first name

[39]:
```
# Title and first name (potentially multiple) are separated by space
title_first = names[2].str.strip().str.partition(' ')

title_first.head(5)
```

[39]:
```
        0  1               2
0     Mr.        Owen Harris
4     Mr.      William Henry
5     Mr.              James
6     Mr.          Timothy J
7  Master      Gosta Leonard
```

[40]:
```
# Extract title from 1st column, strip any remaining white space
title = title_first[0].str.strip()
title.head(5)
```

[40]:
```
0       Mr.
4       Mr.
5       Mr.
6       Mr.
7    Master
Name: 0, dtype: object
```

[41]:
```
# Extract first name(s) from 3rd column, strip any remaining white space
first_name = title_first[2].str.strip()

# Print first 5 observations
```

```
first_name.head(5)
```

[41]:
```
0       Owen Harris
4     William Henry
5             James
6         Timothy J
7     Gosta Leonard
Name: 2, dtype: object
```

Part 5: Store name components in original DataFrame

[42]:
```
# Merge all name components back into original DataFrame
df['FirstName'] = first_name
df['LastName'] = last_name
df['Title'] = title

# Delete Name column
del df['Name']
```

[43]:
```
df.head(5)
```

[43]:
```
   PassengerId  Survived  Pclass   Sex   Age     Ticket     Fare Cabin  \
0            1         0       3  male  22.0  A/5 21171   7.2500   NaN
4            5         0       3  male  35.0     373450   8.0500   NaN
5            6         0       3  male   NaN     330877   8.4583   NaN
6            7         0       1  male  54.0      17463  51.8625   E46
7            8         0       3  male   2.0     349909  21.0750   NaN

  Embarked      FirstName LastName   Title
0        S    Owen Harris   Braund     Mr.
4        S  William Henry    Allen     Mr.
5        Q          James    Moran     Mr.
6        S      Timothy J McCarthy     Mr.
7        S  Gosta Leonard  Palsson  Master
```

Part 6: Extract deck

We can use the `get()` string method to extract the first element of the cabin string (if present). Note that observations with a missing value for `Cabin` will also be assigned a missing value for `Deck`.

[44]:
```
df['Deck'] = df['Cabin'].str.strip().str.get(0)
```

[45]:
```
# Print histogram of the number of cabins by deck
df['Deck'].value_counts().sort_index()
```

[45]:
```
Deck
A    14
B    20
C    32
D    15
E    17
F     8
T     1
Name: count, dtype: int64
```