Part 2 — Lecture 2

TECH2: Introduction to Programming, Data, and Information Technology

Richard Foltyn
Norwegian School of Economics (NHH)

October 9, 2024

See GitHub repository for notebooks and data:

https://github.com/richardfoltyn/TECH2-H24

Contents

1 Introduction to pandas

1.1 Motivation

So far, we have encountered built-in Python containers (`list`, `tuple`, `dict`) and NumPy arrays as the only way to store data. However, while NumPy arrays are great for storing *homogenous* data without any particular structure, they are somewhat limited when we want to use them for data analysis.

For example, we usually want to process data sets with

1. several variables;
2. multiple observations, which need not be identical across variables (some values may be missing);
3. non-homogenous data types: for examples, names need to be stored as strings, birthdays as dates and income as a floating-point number.

While NumPy can in principle handle such situations, it puts all the burden on the user. Most users would prefer to not have to deal with such low-level details.

Pandas was created to offer more versatile data structures that are straightforward to use for storing, manipulating and analyzing heterogeneous data:

1. Data is clearly organized in *variables* and *observations*, similar to econometrics programs such as Stata and R.
2. Each variable is permitted to have a different data type.

3. We can use *labels* to select observations instead of having to use a linear numerical index as with NumPy.

   We could, for example, index a data set using National Insurance Numbers or time stamps for time series data.

4. Pandas offers many convenient data aggregation and reduction routines that can be applied to subsets of data.

   For example, we can easily group observations by city and compute average incomes.

5. Pandas also offers many convenient data import / export functions that go beyond what's in NumPy.

Should we be using pandas at all times, then? No!

- For low-level tasks where performance is essential, use NumPy.
- For homogenous data without any particular data structure, use NumPy.
- On the other hand, if data is heterogeneous, needs to be imported from an external data source and cleaned or transformed before performing computations, use pandas.

There are numerous tutorials on pandas on the internet. Useful additional material includes:

- The official user guide.
- The official pandas cheat sheet which nicely illustrates the most frequently used operations.
- The official API reference with details on every pandas object and function.
- There are numerous tutorials (including videos) available on the internet. See here for a list.

1.2 Creating pandas data structures

Pandas has two main data structures:

1. `Series` represents observations of a single variable.
2. `DataFrame` is a container for several variables. You can think of each individual column of a `DataFrame` as a `Series`, and each row represents one observation.

The easiest way to create a `Series` or `DataFrame` is to create them from pre-existing data.

To access pandas data structures and routines, we need to import them first. The near-universal convention is to make pandas available using the name `pd`:

```
import pandas as pd
```

Example: Create Series from 1-dimensional NumPy array

[1]:
```
import numpy as np
import pandas as pd             # universal convention: import using pd

data = np.arange(5, 10)

# Create pandas Series from 1d array
pd.Series(data)
```

[1]:
```
0    5
1    6
2    7
3    8
4    9
dtype: int64
```

Example: Create DataFrame from NumPy array

We can create a `DataFrame` from a NumPy array:

[2]:
```
# Create matrix of data
data = np.arange(15).reshape((-1, 3))

# Define variable (or column) names
varnames = ['A', 'B', 'C']

# Create pandas DataFrame from matrix
pd.DataFrame(data, columns=varnames)
```

[2]:

	A	B	C
0	0	1	2
1	3	4	5
2	6	7	8
3	9	10	11
4	12	13	14

This code creates a `DataFrame` of three variables called `A`, `B` and `C` with 5 observations each.

Example: Create from dictionary

Alternatively, we can create a `DataFrame` from non-homogenous data as follows:

[3]:
```
# Names (strings)
names = ['Alice', 'Bob']

# Birth dates (datetime objects)
bdates = pd.to_datetime(['1985-01-01', '1997-05-12'])

# Incomes (floats)
incomes = np.array([600000, np.nan])         # code missing income as NaN

# create DataFrame from dictionary
pd.DataFrame({'Name': names, 'Birthdate': bdates, 'Income': incomes})
```

[3]:

	Name	Birthdate	Income
0	Alice	1985-01-01	600000.0
1	Bob	1997-05-12	NaN

If data types differ across columns, as in the above example, it is often convenient to create the `DataFrame` by passing a dictionary as an argument. Each key represents a column name and each corresponding value contains the data for that variable.

1.3 Importing data

1.3.1 Loading text data with NumPy

We often use files that store data as text files containing character-separated values (CSV) since virtually any application supports this data format. The most important functions to read text data are:

- `np.loadtxt()`: load data from a text file.
- `np.genfromtxt()`: load data from a text file and handle missing data.

There are a few other input/output functions in NumPy, for example to write arrays as raw binary data. We won't cover them here, but you can find them in the official documentation.

Example: Load character-separated text data

Imagine we have the following tabular data from FRED, where the first two rows look as follows:

Year	GDP	CPI	UNRATE	FEDFUNDS
1954	2877.7	26.9	5.6	1.0
1955	3083.0	26.8	4.4	1.8

These data are stored as character-separated values (CSV). To load this CSV file as a NumPy array, we use `loadtxt()`. It is advantageous to globally set the path to the `data/` directory that can point either to the local directory or to the `data/` directory on GitHub.

[4]:
```
# Uncomment this to use files in the local data/ directory
DATA_PATH = '../data'

# Load data directly from GitHub
# DATA_PATH = 'https://raw.githubusercontent.com/richardfoltyn/TECH2-H24/main/data'
```

[5]:
```
import numpy as np

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

# load CSV
data = np.loadtxt(file, skiprows=1, delimiter=',')

data[:2]        # Display first two rows
```

[5]:
```
array([[1.9540e+03, 2.8777e+03, 2.6900e+01, 5.6000e+00, 1.0000e+00],
       [1.9550e+03, 3.0830e+03, 2.6800e+01, 4.4000e+00, 1.8000e+00]])
```

The default settings will in many cases be appropriate to load whatever CSV file we might have. However, we'll occasionally want to specify the following arguments to override the defaults:

- `delimiter`: Character used to separate individual fields (default: space).
- `skiprows=n`: Skip the first `n` rows. For example, if the CSV file contains a header with variable names, `skiprows=1` needs to be specified as NumPy by default cannot process these names.
- `encoding`: Set the character encoding of the input data. This is usually not needed, but can be required to import data with non-latin characters that are not encoded using Unicode.

While `loadtxt()` is simple to use, it quickly reaches its limits with more complex data sets. For example, when we try to load the demo data set in `missing.csv` using `loadtxt()`, we get the following error:

[6]:
```
file = f'{DATA_PATH}/missing.csv'

# Attempt to load CSV
data = np.loadtxt(file, skiprows=1, delimiter=';')
```

```
ValueError: could not convert string '' to float64 at row 1, column 3.
```

This code fails because `loadtxt()` does not support files with missing values. One can use the more flexible function `np.genfromtxt()` which allows us to parse files with missing values:

[7]:
```
file = f'{DATA_PATH}/missing.csv'

# Load CSV file using genfromtxt() instead of loadtxt()
data = np.genfromtxt(file, skip_header=True, delimiter=';')

# Display first 2 rows
data[:2]
```

[7]:
```
array([[0.6824, 0.0538, 0.2204],
       [0.1844, 0.1759,    nan]])
```

However, it is usually not worthwhile to figure out how to load complex data with NumPy as this is much easier with pandas.

1.3.2 Loading data with Pandas

Pandas's input/output routines are more powerful than those implemented in NumPy:

- They support reading and writing numerous file formats.
- They support heterogeneous data without having to specify the data type in advance.
- They gracefully handle missing values.

For these reasons, it is often preferable to directly use pandas to process data instead of NumPy.

The most important routines are:

- `read_csv()`, `to_csv()`: Read or write CSV text files
- `read_fwf()`: Read data with fixed field widths, i.e., text data that does not use delimiters to separate fields.
- `read_excel()`, `to_excel()`: Read or write Excel spreadsheets
- `read_stata()`, `to_stata()`: Read or write Stata's `.dta` files.

For a complete list of I/O routines, see the official documentation.

To illustrate, we repeat the above examples using pandas's `read_csv()`:

[8]:
```
import pandas as pd

# relative path to CSV file
file = f'{DATA_PATH}/missing.csv'

df = pd.read_csv(file, sep=';')
df.head(2)          # Display the first 2 rows of data
```

[8]:

	Variable1	Variable2	Variable3
0	0.6824	0.0538	0.2204
1	0.1844	0.1759	NaN

Your turn. Use the pandas functions listed above to import data from the following files located in the `data/` folder:

1. `titanic.csv`
2. `FRED.xlsx`

To load Excel files, you need to have the package `openpyxl` installed.

1.4 Viewing data

With large data sets, you hardly ever want to print the entire `DataFrame`. Pandas by default limits the amount of data shown. You can use the `head()` and `tail()` methods to explicitly display a specific number of rows from the top or the end of a `DataFrame`.

To illustrate, we use a data set of passengers on board of the Titanic's maiden voyage stored in `titanic.csv` which contains the following columns:

1. `PassengerId`
2. `Survived`: indicator whether the person survived

3. `Pclass`: accommodation class (first, second, third)
4. `Name`: Name of passenger (last name, first name)
5. `Sex`: `male` or `female`
6. `Age`
7. `Ticket`: Ticket number
8. `Fare`: Fare in pounds
9. `Cabin`: Deck + cabin number
10. `Embarked`: Port at which passenger embarked: `C` - Cherbourg, `Q` - Queenstown, `S` - Southampton

Before we read in any data, it is convenient to define a variable pointing to the directory where the data resides. We can either use a relative local path `../data`, or alternatively, we can use the full URL to the data file in the GitHub repository.

[9]:
```
# Uncomment this to use files in the local data/ directory
DATA_PATH = '../data'

# Uncomment this to load data directly from GitHub
# DATA_PATH = 'https://raw.githubusercontent.com/richardfoltyn/TECH2-H24/main/data'
```

We can now read in the data stored in the file `titanic.csv` like this:

[10]:
```
import pandas as pd

# URL to CSV file in GitHub repository
file = f'{DATA_PATH}/titanic.csv'

# Load sample data set of Titanic passengers. Individual fields are separated
# using a comma, which is the default.
df = pd.read_csv(file, sep=',')
```

We can now display the first and last three rows:

[11]:
```
df.head(3)      # show first three rows
```

[11]:
```
   PassengerId  Survived  Pclass  \
0            1         0       3
1            2         1       1
2            3         1       3

                                                Name     Sex   Age  \
0                            Braund, Mr. Owen Harris    male  22.0
1  Cumings, Mrs. John Bradley (Florence Briggs Th...  female  38.0
2                             Heikkinen, Miss Laina  female  26.0

             Ticket     Fare Cabin Embarked
0         A/5 21171   7.2500   NaN        S
1          PC 17599  71.2833   C85        C
2  STON/O2. 3101282   7.9250   NaN        S
```

[12]:
```
df.tail(3)      # show last three rows
```

[12]:
```
     PassengerId  Survived  Pclass                                      Name  \
888          889         0       3  Johnston, Miss Catherine Helen "Carrie"
889          890         1       1                     Behr, Mr. Karl Howell
890          891         0       3                       Dooley, Mr. Patrick

        Sex   Age      Ticket   Fare Cabin Embarked
888  female   NaN  W./C. 6607  23.45   NaN        S
889    male  26.0      111369  30.00  C148        C
890    male  32.0      370376   7.75   NaN        Q
```

To quickly compute some descriptive statistics for the *numerical* variables in the `DataFrame`, we use `describe()`:

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

[13]:

	PassengerId	Survived	Pclass	Age	Fare
count	891.000000	891.000000	891.000000	714.000000	891.000000
mean	446.000000	0.383838	2.308642	29.699118	32.204208
std	257.353842	0.486592	0.836071	14.526497	49.693429
min	1.000000	0.000000	1.000000	0.420000	0.000000
25%	223.500000	0.000000	2.000000	20.125000	7.910400
50%	446.000000	0.000000	3.000000	28.000000	14.454200
75%	668.500000	1.000000	3.000000	38.000000	31.000000
max	891.000000	1.000000	3.000000	80.000000	512.329200

Note that this automatically ignores the columns `Name`, `Sex`, `Ticket` and `Cabin` as they contain strings, and computing the mean, standard deviation, etc. of a string variable does not make sense.

For categorical data, we can use `value_counts()` to tabulate the number of unique values of a variable. For example, the following code tabulates passengers by sex:

[14]:
```
df['Sex'].value_counts()
```

[14]:
```
Sex
male      577
female    314
Name: count, dtype: int64
```

Lastly, to see low-level information about the data type used in each column and the number of non-missing observations, we call `info()`:

[15]:
```
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
```

Pandas automatically discards missing information in computations. For example, the age column has several missing values, so the number of reported `Non-Null` values is lower than for the other columns.

1.5 Indexing

Pandas supports two types of indexing:

1. Indexing by position. This is basically identical to the indexing of other Python and NumPy containers.
2. Indexing by label, i.e., by the values assigned to the row or column index. These labels need not be integers in increasing order, as is the case for NumPy. We will see how to assign labels below.

Pandas indexing is performed either by using brackets `[]`, or by using `.loc[]` for label indexing, or `.iloc[]` for positional indexing.

Indexing via `[]` can be somewhat confusing:

- specifying `df['name']` returns the column `name` as a `Series` object.
- On the other hand, specifying a range such as `df[5:10]` returns the *rows* associated with the *positions* 5,...,9.

Example: Selecting columns

[16]:
```
import pandas as pd

# Load sample data of Titanic passengers
df = pd.read_csv(f'{DATA_PATH}/titanic.csv')
df['Name']              # select a single column
```

[16]:
```
0                              Braund, Mr. Owen Harris
1    Cumings, Mrs. John Bradley (Florence Briggs Th...
2                               Heikkinen, Miss Laina
3         Futrelle, Mrs. Jacques Heath (Lily May Peel)
4                             Allen, Mr. William Henry
5                                     Moran, Mr. James
6                              McCarthy, Mr. Timothy J
7                       Palsson, Master Gosta Leonard
8    Johnson, Mrs. Oscar W (Elisabeth Vilhelmina Berg)
9                  Nasser, Mrs. Nicholas (Adele Achem)
                            ...
881                               Markun, Mr. Johann
882                     Dahlberg, Miss Gerda Ulrika
883                   Banfield, Mr. Frederick James
884                          Sutehall, Mr. Henry Jr
885                Rice, Mrs. William (Margaret Norton)
886                           Montvila, Rev. Juozas
887                    Graham, Miss Margaret Edith
888            Johnston, Miss Catherine Helen "Carrie"
889                               Behr, Mr. Karl Howell
890                                 Dooley, Mr. Patrick
Name: Name, Length: 891, dtype: object
```

[17]:
```
df[['Name', 'Sex']]     # select multiple columns using a list
```

[17]:
```
                                                Name     Sex
0                            Braund, Mr. Owen Harris    male
1  Cumings, Mrs. John Bradley (Florence Briggs Th...  female
2                             Heikkinen, Miss Laina  female
3       Futrelle, Mrs. Jacques Heath (Lily May Peel)  female
4                           Allen, Mr. William Henry    male
5                                  Moran, Mr. James    male
6                            McCarthy, Mr. Timothy J    male
7                     Palsson, Master Gosta Leonard    male
8  Johnson, Mrs. Oscar W (Elisabeth Vilhelmina Berg)  female
9                Nasser, Mrs. Nicholas (Adele Achem)  female
..                                               ...     ...
881                             Markun, Mr. Johann    male
882                   Dahlberg, Miss Gerda Ulrika  female
883                 Banfield, Mr. Frederick James    male
884                        Sutehall, Mr. Henry Jr    male
885              Rice, Mrs. William (Margaret Norton)  female
886                         Montvila, Rev. Juozas    male
887                  Graham, Miss Margaret Edith  female
888          Johnston, Miss Catherine Helen "Carrie"  female
```

```
889                              Behr, Mr. Karl Howell    male
890                                Dooley, Mr. Patrick    male

[891 rows x 2 columns]
```

Note: In order to select multiple columns you *must* specify these as a `list`, not a `tuple`.

Example: Selecting rows by position

To return the rows at positions 1, 2 and 3 we use

[18]:
```
df[1:4]
```

[18]:
```
   PassengerId  Survived  Pclass  \
1            2         1       1
2            3         1       3
3            4         1       1

                                                Name     Sex   Age  \
1  Cumings, Mrs. John Bradley (Florence Briggs Th...  female  38.0
2                             Heikkinen, Miss Laina  female  26.0
3       Futrelle, Mrs. Jacques Heath (Lily May Peel)  female  35.0

             Ticket     Fare Cabin Embarked
1          PC 17599  71.2833   C85        C
2  STON/O2. 3101282   7.9250   NaN        S
3            113803  53.1000  C123        S
```

Pandas follows the Python convention that indices are 0-based, and the endpoint of a slice is not included.

1.5.1 Creating and manipulating indices

Pandas uses *labels* to index and align data. These can be integer values starting at 0 with increments of 1 for each additional element, which is the default, but they need not be. The three main methods to create/manipulate indices are:

- Create a new `Series` or `DataFrame` object with a custom index using the `index=` argument.
- `set_index(keys=['column1', ...])` uses the values of `column1` and optionally additional columns as indices, discarding the current index.
- `reset_index()` resets the index to its default value, a sequence of increasing integers starting at 0.

Creating custom indices

First, consider the following code with creates a `Series` with three elements `[10, 20, 30]` using the default index `[0,1,2]`:

[19]:
```
import pandas as pd

# Create Series with default integer index
pd.Series([10, 20, 30])
```

[19]:
```
0    10
1    20
2    30
dtype: int64
```

We can use the `index=` argument to specify a custom index, for example one containing the lower-case characters `a`, `b`, `c` as follows:

[20]:
```
# Create Series with custom index [a, b, c]
pd.Series([10, 20, 30], index=['a', 'b', 'c'])
```

[20]:
```
a    10
b    20
c    30
dtype: int64
```

Manipulating indices

To modify the index of an *existing* `Series` or `DataFrame` object, we use the methods `set_index()` and `reset_index()`. Note that these return a new object and leave the original `Series` or `DataFrame` unchanged. If we want to change the existing object, we need to pass the argument `inplace=True`.

For example, we can replace the row index and use the Roman lower-case characters `a`, `b`, `c`, ... as labels instead of integers:

[21]:
```
import string

# Step 1: Create list of lower-case ASCII characters
letters = list(string.ascii_lowercase)

# print list
letters
```

[21]:
```
['a',
 'b',
 'c',
 'd',
 'e',
 'f',
 'g',
 'h',
 'i',
 'j',
 'k',
 'l',
 'm',
 'n',
 'o',
 'p',
 'q',
 'r',
 's',
 't',
 'u',
 'v',
 'w',
 'x',
 'y',
 'z']
```

We now read in the Titanic data set, add `letters` as a new column and set it as the index.

[22]:
```
import pandas as pd

# Read in Titanic passenger data
df = pd.read_csv(f'{DATA_PATH}/titanic.csv')

# For demo purposes, restrict sample to first N=26 observations
N = len(letters)
df = df[:N]
```

```
df['letters'] = letters                              # create new column 'letters'
df.set_index(keys=['letters'], inplace=True)         # set letters as index!

# Display updated DataFrame
df.head(1)
```

[22]:
```
         PassengerId  Survived  Pclass                     Name   Sex   Age  \
letters
a                  1         0       3  Braund, Mr. Owen Harris  male  22.0

            Ticket  Fare Cabin Embarked
letters
a        A/5 21171  7.25   NaN        S
```

We can now use these new labels to select records in the `DataFrame`:

[23]:
```
# print first 3 rows using labels
df['a':'c']             # This is the same as df[:3]
```

[23]:
```
         PassengerId  Survived  Pclass  \
letters
a                  1         0       3
b                  2         1       1
c                  3         1       3

                                                  Name     Sex   Age  \
letters
a                                Braund, Mr. Owen Harris    male  22.0
b        Cumings, Mrs. John Bradley (Florence Briggs Th...  female  38.0
c                                   Heikkinen, Miss Laina  female  26.0

                    Ticket     Fare Cabin Embarked
letters
a                A/5 21171   7.2500   NaN        S
b                 PC 17599  71.2833   C85        C
c        STON/O2. 3101282   7.9250   NaN        S
```

Note that when specifying a range in terms of labels, the last element *is* included! Hence the row with index `c` in the above example is shown.

We can reset the index to its default integer values using the `reset_index()` method:

[24]:
```
# Reset index labels to default value (integers 0, 1, 2, ...) and print
# first three rows
df.reset_index(drop=True).head(3)
```

[24]:
```
   PassengerId  Survived  Pclass  \
0            1         0       3
1            2         1       1
2            3         1       3

                                                Name     Sex   Age  \
0                            Braund, Mr. Owen Harris    male  22.0
1  Cumings, Mrs. John Bradley (Florence Briggs Th...  female  38.0
2                              Heikkinen, Miss Laina  female  26.0

             Ticket     Fare Cabin Embarked
0         A/5 21171   7.2500   NaN        S
1          PC 17599  71.2833   C85        C
2  STON/O2. 3101282   7.9250   NaN        S
```

The `drop=True` argument tells pandas to throw away the old index values instead of storing them as a column of the resulting `DataFrame`.

Your turn. Read in the following data files from the `data/` folder and manipulate the dataframe index:

1. Read in the file `FRED.csv` and set the column `Year` as the index.
2. Read in the file `FRED-monthly.csv` and set the columns `Year` and `Month` as the index

Experiment what happens if you use the `inplace=True` and `append=True` options of `set_index()`. Restore the original (default) index after you are done.

1.5.2 Selecting elements

To more clearly distinguish between selection by label and by position, pandas provides the `.loc[]` and `.iloc[]` methods of indexing. To make your intention obvious, you should therefore adhere to the following rules:

1. Use `df['name']` only to select *columns* and nothing else.
2. Use `.loc[]` to select by label.
3. Use `.iloc[]` to select by position.

Selection by label

To illustrate, using `.loc[]` unambiguously indexes by label:

[25]:
```
df.loc['d':'f', ['Name', 'Sex']]
```

[25]:
```
                                                Name     Sex
letters
d        Futrelle, Mrs. Jacques Heath (Lily May Peel)  female
e                            Allen, Mr. William Henry    male
f                                    Moran, Mr. James    male
```

With `.loc[]` we can even perform slicing on column names, which is not possible with the simpler `df[]` syntax:

[26]:
```
df.loc['d':'f', 'Name':'Age']
```

[26]:
```
                                                Name     Sex   Age
letters
d        Futrelle, Mrs. Jacques Heath (Lily May Peel)  female  35.0
e                            Allen, Mr. William Henry    male  35.0
f                                    Moran, Mr. James    male   NaN
```

This includes all the columns between `Name` and `Age`, where the latter is included since we are slicing by label.

Trying to pass in positional arguments will return an error for the given `DataFrame` since the index labels are `a`, `b`, `c`,... and not 0, 1, 2...

[27]:
```
df.loc[0:4]
```

```
TypeError: cannot do slice indexing on Index with these indexers [0] of type int
```

However, we can reset the index to its default value. Then the index labels are integers and coincide with their position, so that `.loc[]` works:

[28]:
```
df = df.reset_index(drop=True)      # reset index labels to integers,
                                    # drop original index
df.loc[0:4]
```

[28]:
```
   PassengerId  Survived  Pclass  \
0            1         0       3
1            2         1       1
2            3         1       3
3            4         1       1
4            5         0       3

                                                Name     Sex   Age  \
0                            Braund, Mr. Owen Harris    male  22.0
1  Cumings, Mrs. John Bradley (Florence Briggs Th...  female  38.0
2                             Heikkinen, Miss Laina  female  26.0
3       Futrelle, Mrs. Jacques Heath (Lily May Peel)  female  35.0
4                           Allen, Mr. William Henry    male  35.0

             Ticket     Fare Cabin Embarked
0         A/5 21171   7.2500   NaN        S
1          PC 17599  71.2833   C85        C
2  STON/O2. 3101282   7.9250   NaN        S
3            113803  53.1000  C123        S
4            373450   8.0500   NaN        S
```

Again, the end point with label 4 is included because we are selecting by label.

Indexing via `.loc[]` supports a few more types of arguments, see the official documentation for details.

Selection by position

Conversely, if we want to select items exclusively by their position and ignore their labels, we use `.iloc[]`:

[29]:
```
df.iloc[0:4, 0:2]          # select first 4 rows, first 2 columns
```

[29]:
```
   PassengerId  Survived
0            1         0
1            2         1
2            3         1
3            4         1
```

Again, `.iloc[]` supports a multitude of other arguments, see the official documentation for details.

Boolean indexing

Similar to NumPy, pandas allows us to select a subset of rows in a `Series` or `DataFrame` if they satisfy some condition. The simplest use case is to create a column of boolean values (`True` or `False`) as a result of some logical operation:

This even works without explicitly using the `.loc[]` attribute:

[30]:
```
import pandas as pd

# Read in Titanic passenger data
df = pd.read_csv(f'{DATA_PATH}/titanic.csv')

# Check whether passenger embarked in Southampton
df['Embarked'] == "S"
```

[30]:
```
0       True
1      False
2       True
3       True
4       True
5      False
6       True
7       True
8       True
```

```
9      False
       ...
881     True
882     True
883     True
884     True
885    False
886     True
887     True
888     True
889    False
890    False
Name: Embarked, Length: 891, dtype: bool
```

Such boolean arrays can be used to select a subset of entries:

[31]:
```
df.loc[df['Embarked'] == 'S', 'Name':'Age']
```

[31]:
```
                                                 Name     Sex   Age
0                             Braund, Mr. Owen Harris    male  22.0
2                              Heikkinen, Miss Laina  female  26.0
3         Futrelle, Mrs. Jacques Heath (Lily May Peel)  female  35.0
4                           Allen, Mr. William Henry    male  35.0
6                            McCarthy, Mr. Timothy J    male  54.0
7                     Palsson, Master Gosta Leonard    male   2.0
8    Johnson, Mrs. Oscar W (Elisabeth Vilhelmina Berg)  female  27.0
10                   Sandstrom, Miss Marguerite Rut  female   4.0
11                          Bonnell, Miss Elizabeth  female  58.0
12                   Saundercock, Mr. William Henry    male  20.0
..                                              ...     ...   ...
877                            Petroff, Mr. Nedelio    male  19.0
878                              Laleff, Mr. Kristo    male   NaN
880       Shelley, Mrs. William (Imanita Parrish Hall)  female  25.0
881                            Markun, Mr. Johann    male  33.0
882                      Dahlberg, Miss Gerda Ulrika  female  22.0
883                    Banfield, Mr. Frederick James    male  28.0
884                           Sutehall, Mr. Henry Jr    male  25.0
886                            Montvila, Rev. Juozas    male  27.0
887                      Graham, Miss Margaret Edith  female  19.0
888           Johnston, Miss Catherine Helen "Carrie"  female   NaN

[644 rows x 3 columns]
```

Boolean indexing also works directly with `[]` without having to specify `.loc[]`, but then it is not possible to also select a subset of columns at the same time:

[32]:
```
df[df['Embarked'] == 'S']
```

[32]:
```
     PassengerId  Survived  Pclass  \
0              1         0       3
2              3         1       3
3              4         1       1
4              5         0       3
6              7         0       1
7              8         0       3
8              9         1       3
10            11         1       3
11            12         1       1
12            13         0       3
..           ...       ...     ...
877          878         0       3
878          879         0       3
880          881         1       2
```

```
881          882         0       3
882          883         0       3
883          884         0       2
884          885         0       3
886          887         0       2
887          888         1       1
888          889         0       3

                                                Name     Sex   Age  \
0                              Braund, Mr. Owen Harris    male  22.0
2                                Heikkinen, Miss Laina  female  26.0
3         Futrelle, Mrs. Jacques Heath (Lily May Peel)  female  35.0
4                             Allen, Mr. William Henry    male  35.0
6                              McCarthy, Mr. Timothy J    male  54.0
7                           Palsson, Master Gosta Leonard    male   2.0
8    Johnson, Mrs. Oscar W (Elisabeth Vilhelmina Berg)  female  27.0
10                     Sandstrom, Miss Marguerite Rut  female   4.0
11                            Bonnell, Miss Elizabeth  female  58.0
12                     Saundercock, Mr. William Henry    male  20.0
..                                                ...     ...   ...
877                              Petroff, Mr. Nedelio    male  19.0
878                                Laleff, Mr. Kristo    male   NaN
880       Shelley, Mrs. William (Imanita Parrish Hall)  female  25.0
881                                Markun, Mr. Johann    male  33.0
882                          Dahlberg, Miss Gerda Ulrika  female  22.0
883                        Banfield, Mr. Frederick James    male  28.0
884                               Sutehall, Mr. Henry Jr    male  25.0
886                                Montvila, Rev. Juozas    male  27.0
887                          Graham, Miss Margaret Edith  female  19.0
888              Johnston, Miss Catherine Helen "Carrie"  female   NaN

                Ticket     Fare Cabin Embarked
0            A/5 21171   7.2500   NaN        S
2    STON/O2. 3101282   7.9250   NaN        S
3               113803  53.1000  C123        S
4               373450   8.0500   NaN        S
6                17463  51.8625   E46        S
7               349909  21.0750   NaN        S
8               347742  11.1333   NaN        S
10             PP 9549  16.7000    G6        S
11              113783  26.5500  C103        S
12           A/5. 2151   8.0500   NaN        S
..                 ...      ...   ...      ...
877             349212   7.8958   NaN        S
878             349217   7.8958   NaN        S
880             230433  26.0000   NaN        S
881             349257   7.8958   NaN        S
882               7552  10.5167   NaN        S
883  C.A./SOTON 34068  10.5000   NaN        S
884   SOTON/OQ 392076   7.0500   NaN        S
886             211536  13.0000   NaN        S
887             112053  30.0000   B42        S
888         W./C. 6607  23.4500   NaN        S

[644 rows x 10 columns]
```

Multiple conditions can be combined using the & (logical and) or | (logical or) operators:

[33]:
```
# Select men who embarked in Southampton
df.loc[(df['Embarked'] == 'S') & (df['Sex'] == 'male'), ['Name', 'Embarked', 'Sex']]
```

[33]:
```
                              Name Embarked   Sex
0            Braund, Mr. Owen Harris        S  male
4           Allen, Mr. William Henry        S  male
6            McCarthy, Mr. Timothy J        S  male
7     Palsson, Master Gosta Leonard        S  male
12   Saundercock, Mr. William Henry        S  male
13      Andersson, Mr. Anders Johan        S  male
17     Williams, Mr. Charles Eugene        S  male
20              Fynney, Mr. Joseph J        S  male
21            Beesley, Mr. Lawrence        S  male
23     Sloper, Mr. William Thompson        S  male
..                              ...      ...   ...
870               Balkic, Mr. Cerin        S  male
872        Carlsson, Mr. Frans Olof        S  male
873     Vander Cruyssen, Mr. Victor        S  male
876   Gustafsson, Mr. Alfred Ossian        S  male
877            Petroff, Mr. Nedelio        S  male
878              Laleff, Mr. Kristo        S  male
881              Markun, Mr. Johann        S  male
883   Banfield, Mr. Frederick James        S  male
884          Sutehall, Mr. Henry Jr        S  male
886           Montvila, Rev. Juozas        S  male

[441 rows x 3 columns]
```

If we want to include rows where an observation takes on one of multiple values, the `isin()` method can be used:

[34]:
```
# Select passengers who embarked in Southampton or Queenstown
df.loc[df['Embarked'].isin(('S', 'Q')), ['Name', 'Embarked']]
```

[34]:
```
                                                 Name Embarked
0                             Braund, Mr. Owen Harris        S
2                               Heikkinen, Miss Laina        S
3         Futrelle, Mrs. Jacques Heath (Lily May Peel)        S
4                            Allen, Mr. William Henry        S
5                                    Moran, Mr. James        Q
6                             McCarthy, Mr. Timothy J        S
7                       Palsson, Master Gosta Leonard        S
8    Johnson, Mrs. Oscar W (Elisabeth Vilhelmina Berg)        S
10                     Sandstrom, Miss Marguerite Rut        S
11                            Bonnell, Miss Elizabeth        S
..                                                ...      ...
880       Shelley, Mrs. William (Imanita Parrish Hall)        S
881                                Markun, Mr. Johann        S
882                         Dahlberg, Miss Gerda Ulrika        S
883                       Banfield, Mr. Frederick James        S
884                              Sutehall, Mr. Henry Jr        S
885               Rice, Mrs. William (Margaret Norton)        Q
886                              Montvila, Rev. Juozas        S
887                        Graham, Miss Margaret Edith        S
888            Johnston, Miss Catherine Helen "Carrie"        S
890                                Dooley, Mr. Patrick        Q

[721 rows x 2 columns]
```

Finally, `DataFrame` implements a `query()` method which allows us to combine multiple conditions in a single string in an intuitive fashion. Column names can be used directly within this string to put restrictions on their values.

[35]:
```
# Select passengers who embarked in Southampton and were above age 70
df.query('Embarked == "S" & Age > 70')
```

[35]:
```
     PassengerId  Survived  Pclass                                 Name  \
630          631         1       1  Barkworth, Mr. Algernon Henry Wilson
851          852         0       3                    Svensson, Mr. Johan

      Sex   Age  Ticket    Fare Cabin Embarked
630  male  80.0   27042  30.000   A23        S
851  male  74.0  347060   7.775   NaN        S
```

Your turn. Load the Titanic passenger data set `data/titanic.csv` and select the follow subsets of data:

1. Select all passengers with passenger IDs from 10 to 20
2. Select the 10th to 20th (inclusive) row of the dataframe
3. Using `query()`, select the sub-sample of female passengers aged 30 to 40. Display only the columns Name, Age, and Sex (in that order)
4. Repeat the last exercise without using `query()`
5. Select all men who embarked in Queenstown or Cherbourg