1 Pandas

**Pandas** (derived from the term “panel data”) is Python’s primary data analysis library. Built on NumPy, it provides a vast range of data-wrangling capabilities that are fast, flexible, and intuitive. Unlike NumPy, pandas allows for the ingestion of heterogeneous data types *via* its two main data structures: pandas *series* and pandas *data frames*.

To begin, execute the following command to import pandas. (Let’s also import NumPy for good measure.)

```python
import pandas as pd
import numpy as np
```

1.1 pandas Series

A pandas *series* is a *one-dimensional* array-like object that allows us to index data in various ways. It acts much like an *ndarray* in NumPy, but supports many more data types such as *integers*, *strings*, *floats*, *Python objects*, etc. The basic syntax to create a pandas series is

```python
s = pd.Series(data, index=index)
```

where

- *data* can be e.g. a Python dictionary, list, or ndarray.
- *index* is a list of axis labels the *same length* as *data*.

Note that Series is like a NumPy array, but we can prescribe *custom indices* instead of the usual numeric 0 to *N − 1*.

**Creating pandas Series**

```python
# Example: create series using ndarray
s1 = pd.Series(np.arange(0,5), index = ['I', 'II', 'III', 'IV', 'V'])
print(s1)
```
One important difference from NumPy is that the entries in `data` do not need to be of the same type.

```
# Example: heterogeneous data types
s2 = pd.Series(data = [0.1, 12, 'Bristol', 1000], index = ['a', 'b', 'c', 'd'])
print(s2)
```

```
a  0.1
b  12
c  Bristol
d  1000
dtype: object
```

We can also create a Series from Python dictionaries. Note that when a Series is instantiated from a dictionary, we do not specify the index.

```
d1 = {'q': 8, 'r': 16, 's': 24}  # create dictionary
s3 = pd.Series(d1)
print(s3)
```

```
q  8
r 16
s 24
dtype: int64
```

Retrieving the names of Series indices

We can retrieve the Series indices as follows:

```
s1.index
```

```
Index(['I', 'II', 'III', 'IV', 'V'], dtype='object')
```

Extract elements from Series by index name

To call/extract elements, we use the `.loc[index name]` command. Note the use of *square brackets*. If a label is used that is not in the Series, an exception is raised.

```
s2.loc['a']
```
To access multiple entries, we use

```python
s2.loc[['d', 'c']]
```

d 1000
c Bristol
dtype: object

Extract elements from Series by integer location (.iloc)

Alternatively, we can use the integer-based .iloc command that extracts elements based on their numeric index.

```python
s2.iloc[[2, 3, 0]]
```

c Bristol
d 1000
a 0.1
dtype: object

### 1.2 pandas DataFrame

A pandas DataFrame is a two-dimensional data structure that supports heterogeneous data with labelled axes for rows and columns. The columns can have different types. DataFrames’s are the more commonly used pandas data structures. It can be useful to think of a DataFrame as being analogous to something like a spreadsheet in Excel.

**Creating DataFrames**

One way to create a pandas DataFrame is through a dictionary of Python Series.

```python
# Create a DataFrame from dictionary of Python series

d = {'X': pd.Series(np.arange(0,5), index = ['cheese', 'wine', 'bread', 'olives', 'gin']),
     'Y': pd.Series(data = ['Glasgow', 'London', 'Bristol'], index = ['wine',
                      'cheese', 'cider'])}

dF = pd.DataFrame(d)
dF
```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>bread</td>
<td>2.0</td>
</tr>
<tr>
<td>cheese</td>
<td>0.0</td>
</tr>
<tr>
<td>cider</td>
<td>NaN</td>
</tr>
<tr>
<td>gin</td>
<td>4.0</td>
</tr>
<tr>
<td>olives</td>
<td>3.0</td>
</tr>
</tbody>
</table>
Let’s pause to think a little about the output here. In particular, note the occurrence of the values NaN in both columns. We note that the indices are the union of the indices of the various Series that make up our data frame. In other words, the indices are merged.

There are numerous other ways to construct DataFrames in pandas. In the Worksheet, you will learn how to create a DataFrame from a list of Python dictionaries.

**Retrieving DataFrame index and column names**

To obtain the DataFrame index and column names, we execute:

```
[35]: dF.index
Index(['bread', 'cheese', 'cider', 'gin', 'olives', 'wine'], dtype='object')

[36]: dF.columns
Index(['X', 'Y'], dtype='object')

[37]: dF['X']
```

```
bread  2.0  
cheese  0.0  
cider   NaN  
gin     4.0  
olives  3.0  
wine    1.0  
Name: X, dtype: float64
```

**Indexing & selection**

Indexing DataFrames follows essentially the same syntax as Series. To access:

- a column, we use `dF[column name]` OR `dF.column name`
- a row, we use either (i) its index label `dF.loc[index label]` or (ii) its integer location `dF.iloc[integer location]`
- multiple rows, we use slice indexing e.g. `dF[0:3]`. **Note:** if you try to use a single integer, `dF[0]` say, an exception will be thrown as pandas thinks you’re trying to access a column called 0.

```
# By column
print(dF['X'])
print()
print(dF.X)
print()
```
# By row, index

```
print(dF.loc['bread'])
print()
```

# By row, integer location

```
print(dF.iloc[1])
print()
```

# Multiple rows by integer location

```
print(dF[0:3])
print()
```

```
bread  2.0  
cheese 0.0  
cider  NaN  
gin    4.0  
olives 3.0  
wine   1.0  
Name: X, dtype: float64
```

```
bread  2.0  
cheese 0.0  
cider  NaN  
gin    4.0  
olives 3.0  
wine   1.0  
Name: X, dtype: float64
```

```
X    2
Y    NaN  
Name: bread, dtype: object
```

```
X     0
Y   London  
Name: cheese, dtype: object
```

```
      X   Y
bread 2.0  NaN
cheese 0.0  London
cider NaN  Bristol  
```

**Boolean indexing**

Like in NumPy we can apply *Boolean filtering/indexing* to extract specific elements in a DataFrame.
Here we apply a Boolean filter `dF['X'] > 2` which gives the values True or False for each value in the column `X` depending on whether the condition is satisfied or not. We then provide this indexing to the DataFrame `dF` to extract the rows where the condition is satisfied, giving a new DataFrame `dF_new`.

### 1.3 Data ingestion

Pandas really comes into its own when dealing with large data sets with potentially millions of entries of different data types and formats.

We will concentrate here on the NBA Players Database (called `NBA_Stats.csv`), a publicly available database of NBA statistics on the website Kaggle, which provides basic statistics on NBA basketball players up to the year 2020. To import the `.csv` file, we use the pandas function `.read_csv()`.

```python
NBA = pd.read_csv('./NBA_Stats.csv', sep = ',')
print(type(NBA))
```

<class 'pandas.core.frame.DataFrame'>

We can get some information about our DataFrame `NBA` using the `.info()` command. This shows us that the DataFrame has 22 columns of information and 11700 rows. Note the data types of each column. Further, notice that the indices in this DataFrame are just the integers 0 to 11700.

```python
NBA.info()
```

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 11700 entries, 0 to 11699
Data columns (total 22 columns):
<table>
<thead>
<tr>
<th>#</th>
<th>Column</th>
<th>Non-Null Count</th>
<th>Dtype</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Unnamed: 0</td>
<td>11700 non-null</td>
<td>int64</td>
</tr>
<tr>
<td>1</td>
<td>player_name</td>
<td>11700 non-null</td>
<td>object</td>
</tr>
<tr>
<td>2</td>
<td>team_abbreviation</td>
<td>11700 non-null</td>
<td>object</td>
</tr>
<tr>
<td>3</td>
<td>age</td>
<td>11700 non-null</td>
<td>float64</td>
</tr>
<tr>
<td>4</td>
<td>player_height</td>
<td>11700 non-null</td>
<td>float64</td>
</tr>
<tr>
<td>5</td>
<td>player_weight</td>
<td>11700 non-null</td>
<td>float64</td>
</tr>
<tr>
<td>6</td>
<td>college</td>
<td>11700 non-null</td>
<td>object</td>
</tr>
<tr>
<td>7</td>
<td>country</td>
<td>11700 non-null</td>
<td>object</td>
</tr>
<tr>
<td>8</td>
<td>draft_year</td>
<td>11700 non-null</td>
<td>object</td>
</tr>
<tr>
<td>9</td>
<td>draft_round</td>
<td>11700 non-null</td>
<td>object</td>
</tr>
<tr>
<td>10</td>
<td>draft_number</td>
<td>11700 non-null</td>
<td>object</td>
</tr>
<tr>
<td>11</td>
<td>gp</td>
<td>11700 non-null</td>
<td>int64</td>
</tr>
<tr>
<td>12</td>
<td>pts</td>
<td>11700 non-null</td>
<td>float64</td>
</tr>
<tr>
<td>13</td>
<td>reb</td>
<td>11700 non-null</td>
<td>float64</td>
</tr>
<tr>
<td>14</td>
<td>ast</td>
<td>11700 non-null</td>
<td>float64</td>
</tr>
<tr>
<td>15</td>
<td>net_rating</td>
<td>11700 non-null</td>
<td>float64</td>
</tr>
<tr>
<td>16</td>
<td>oreb_pct</td>
<td>11700 non-null</td>
<td>float64</td>
</tr>
<tr>
<td>17</td>
<td>dreb_pct</td>
<td>11700 non-null</td>
<td>float64</td>
</tr>
<tr>
<td>18</td>
<td>usg_pct</td>
<td>11700 non-null</td>
<td>float64</td>
</tr>
<tr>
<td>19</td>
<td>ts_pct</td>
<td>11700 non-null</td>
<td>float64</td>
</tr>
<tr>
<td>20</td>
<td>ast_pct</td>
<td>11700 non-null</td>
<td>float64</td>
</tr>
<tr>
<td>21</td>
<td>season</td>
<td>11700 non-null</td>
<td>object</td>
</tr>
</tbody>
</table>

dtypes: float64(12), int64(2), object(8)
memory usage: 2.0+ MB

We can view the first few rows using the `.head()` function (which prints the first 5 rows by default) or the last few rows using `.tail()`.

```
# Print the first 10 rows
NBA.head()
```

```
   Unnamed: 0  player_name team_abbreviation   age  player_height   player_weight  college country draft_year draft_round
0          0       Travis Knight     LAL  22.0        213.36    106.59412 Connecticut  USA  1996         1
1          1         Matt Fish     MIA  27.0        210.82    106.59412 North Carolina-Wilmington USA  1992         2
2          2        Matt Bullard    HOU  30.0        208.28    106.59412 Iowa USA Undrafted Undrafted
3          3      Marty Conlon     BOS  29.0        210.82    111.13004 Providence USA Undrafted Undrafted
4          4  Martin Muursepp     DAL  22.0        205.74    106.59412 None USA  1996         1
```
<table>
<thead>
<tr>
<th></th>
<th>pts</th>
<th>reb</th>
<th>ast</th>
<th>net_rating</th>
<th>oreb_pct</th>
<th>dreb_pct</th>
<th>usg_pct</th>
<th>ts_pct</th>
<th>ast_pct</th>
<th>season</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4.8</td>
<td>4.5</td>
<td>0.5</td>
<td>6.2</td>
<td>0.127</td>
<td>0.182</td>
<td>0.142</td>
<td>0.536</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0.3</td>
<td>0.8</td>
<td>0.0</td>
<td>-15.1</td>
<td>0.143</td>
<td>0.267</td>
<td>0.265</td>
<td>0.333</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4.5</td>
<td>1.6</td>
<td>0.9</td>
<td>0.9</td>
<td>0.016</td>
<td>0.115</td>
<td>0.151</td>
<td>0.535</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>7.8</td>
<td>4.4</td>
<td>1.4</td>
<td>-9.0</td>
<td>0.083</td>
<td>0.152</td>
<td>0.167</td>
<td>0.542</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>3.7</td>
<td>1.6</td>
<td>0.5</td>
<td>-14.5</td>
<td>0.109</td>
<td>0.118</td>
<td>0.233</td>
<td>0.482</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>ast_pct</th>
<th>season</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.052</td>
<td>1996-97</td>
</tr>
<tr>
<td>1</td>
<td>0.000</td>
<td>1996-97</td>
</tr>
<tr>
<td>2</td>
<td>0.099</td>
<td>1996-97</td>
</tr>
<tr>
<td>3</td>
<td>0.101</td>
<td>1996-97</td>
</tr>
<tr>
<td>4</td>
<td>0.114</td>
<td>1996-97</td>
</tr>
</tbody>
</table>

[5 rows x 22 columns]
[5 rows x 22 columns]