Data wrangling in Python - Data wrangling with Pandas - 1

One should look for what is and not what he thinks should be. (Albert Einstein)

Data Wrangling with Pandas: Topic introduction

In this part of the course, we will cover the following concepts:

- Pandas use cases and basic operations
- Dataframe definition and manipulation

Warm up

- In the next few modules, we will use another popular Python library, **Pandas**, to continue performing some basic operations
- Python can open a lot of analytical doors, even when you're just starting out
- Take a few minutes to skim through ***this list of 12 cool data science projects***
- Which application most interests you? Why?

12 Cool Data Science Projects Ideas for Beginners and Experts

"How many data science projects have you completed so far?"

Photo by bongkarn thanyakij from Pexels

The domain of Data Science brings with itself a variety of scientific tools, processes, algorithms, and knowledge extraction systems from structured and unstructured data alike, for identifying meaningful patterns in it.

Module completion checklist

Objective	Complete
Summarize use cases of Pandas	
Demonstrate use of basic operations on series	

Dataset manipulation with Pandas

- Pandas is a powerful library for **cleaning and analyzing datasets in Python**
- A dataset is a collection of data, usually in tabular format, where:
  - Every column represents a particular **variable**
  - Every row represents a given **record**
- We learned about NumPy, to help us work with arrays of numbers, one very specific kind of dataset
- Pandas will help us with cleaning and analyzing datasets of all kinds
- For complete documentation, ***click here***

A little more about Pandas

- Pandas is an effective tool to read, write and manipulate data
- Pandas contains tools to perform high-performance merging and joining datasets
- Pandas is highly optimized for performance, with critical code paths written in C

Import pandas and pathlib

- Let's import the `pandas` library
- **Note:** it is not required that you also import `numpy` in order to use `pandas`
- However, you will often see both of them imported since many projects make use of both

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

- We will now introduce a package that allows you to reference your **working directory**
- This will be the directory where your data is stored, allowing you to import data directly from there

```
from pathlib import Path
```

Directory settings

- In order to maximize the efficiency of your workflow, you should encode your directory structure into `variables`
- We will use the `pathlib` library
- Let the `main_dir` be the variable corresponding to your `course` folder
- Let `data_dir` be the variable corresponding to your `data` folder

```
# Set 'main_dir' to location of the project folder
home_dir = Path(".").resolve()
main_dir = home_dir.parent.parent
print(main_dir)
```

```
data_dir = str(main_dir) + "/data"
print(data_dir)
```

Module completion checklist

Objective	Complete
Summarize use cases of Pandas	✔
Demonstrate use of basic operations on series	

Series

- The first Pandas object we'll learn about is a `Series`, which has many more additional properties and methods than a NumPy `array`
- ***Click here to learn more***
- We can create a `Series` from a normal Python `list`

```
num_series = pd.Series([45, 89, 67, 33])
print(num_series)
```

```
0    45
1    89
2    67
3    33
dtype: int64
```

- The in the series are actually stored in an `ndarray`
- To extract just the values as an `ndarray`, use the `.values` property of `Series`

```
print(num_series.values)
```

```
[45 89 67 33]
```

Date series: ranges by month

- Pandas supports series of dates, making it a great choice for time series analysis
- Date series can be created in a couple ways

```
# Create a date range in monthly intervals
print(pd.date_range(start = '20170101', end = '20170331', freq = 'M'))
```

```
DatetimeIndex(['2017-01-31', '2017-02-28', '2017-03-31'], dtype='datetime64[ns]', freq='M')
```

```
# Create a date range with no end date, but specifying start date, frequency, and number of periods
print(pd.date_range(start = '20170101', freq = 'M', periods = 4))
```

```
DatetimeIndex(['2017-01-31', '2017-02-28', '2017-03-31', '2017-04-30'],
dtype='datetime64[ns]', freq='M')
```

Date series: ranges by hour

- This function can also create hourly series

```
print(pd.date_range(start = '20170101', end = '20170102', freq = 'H'))
```

```
DatetimeIndex(['2017-01-01 00:00:00', '2017-01-01 01:00:00',
               '2017-01-01 02:00:00', '2017-01-01 03:00:00',
               '2017-01-01 04:00:00', '2017-01-01 05:00:00',
               '2017-01-01 06:00:00', '2017-01-01 07:00:00',
               '2017-01-01 08:00:00', '2017-01-01 09:00:00',
               '2017-01-01 10:00:00', '2017-01-01 11:00:00',
               '2017-01-01 12:00:00', '2017-01-01 13:00:00',
               '2017-01-01 14:00:00', '2017-01-01 15:00:00',
               '2017-01-01 16:00:00', '2017-01-01 17:00:00',
               '2017-01-01 18:00:00', '2017-01-01 19:00:00',
               '2017-01-01 20:00:00', '2017-01-01 21:00:00',
               '2017-01-01 22:00:00', '2017-01-01 23:00:00',
               '2017-01-02 00:00:00'],
              dtype='datetime64[ns]', freq='H')
```

- You can create `Series` by year, by minute, by second, without needing a date
- Many formats are available

Series methods

- `Series` are more powerful than base Python lists due to the additional attributes and methods they possess

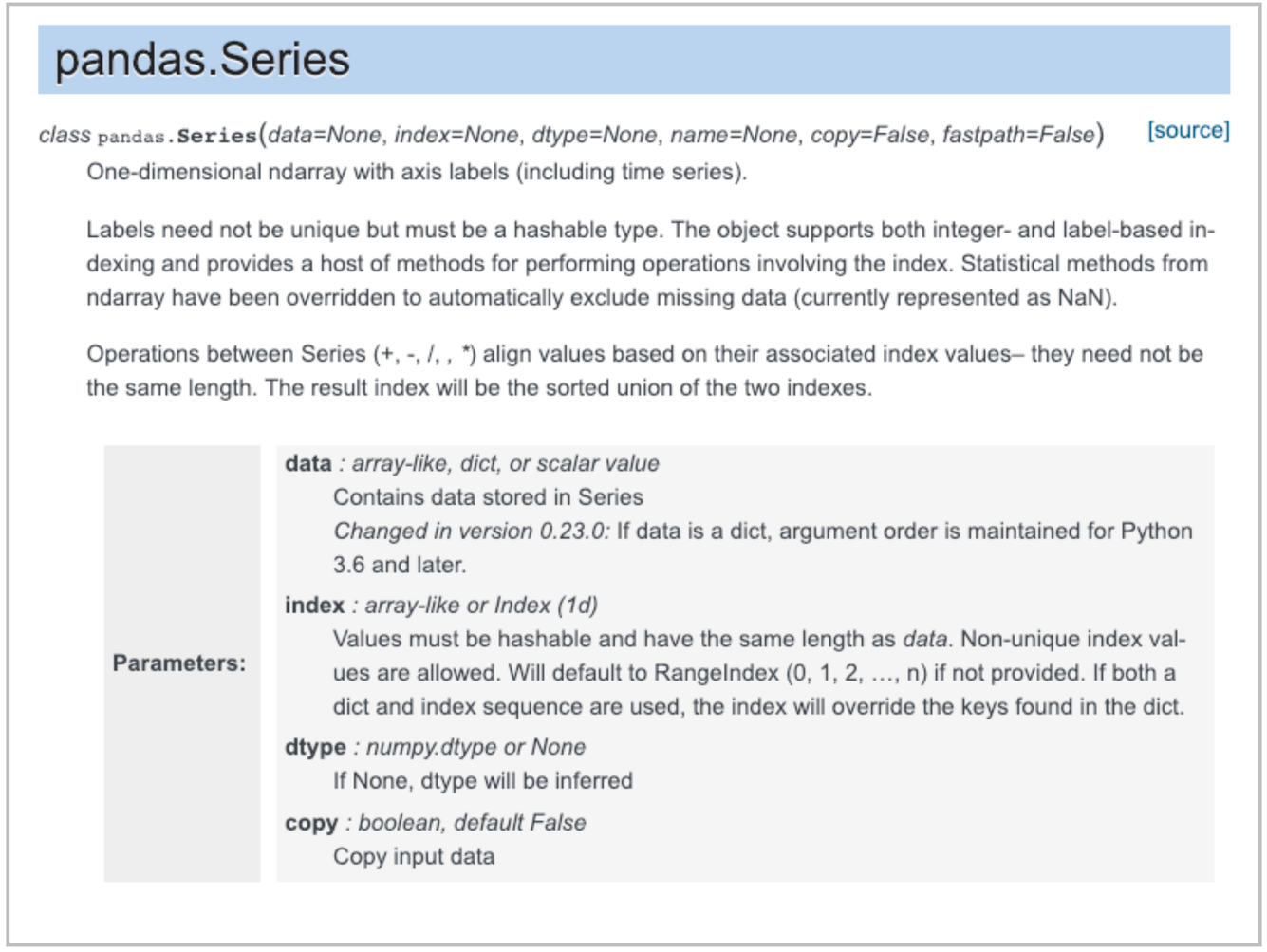
```
norm_series = pd.Series(np.arange(5, 20, 5))
print(norm_series)
```

```
0     5
1    10
2    15
dtype: int64
```

- Here `0`, `1` and `2` in the first column specify the index
- `5`, `10` and `15` are the values at the corresponding index in the `Series`

Series methods (cont'd)

- Here is more information about the `pandas.Series`



pandas.Series

class `pandas.`**`Series`**(*data=None*, *index=None*, *dtype=None*, *name=None*, *copy=False*, *fastpath=False*) [source]

One-dimensional ndarray with axis labels (including time series).

Labels need not be unique but must be a hashable type. The object supports both integer- and label-based indexing and provides a host of methods for performing operations involving the index. Statistical methods from ndarray have been overridden to automatically exclude missing data (currently represented as NaN).

Operations between Series (+, -, /, , *) align values based on their associated index values– they need not be the same length. The result index will be the sorted union of the two indexes.

Parameters:

data *: array-like, dict, or scalar value*
Contains data stored in Series
Changed in version 0.23.0: If data is a dict, argument order is maintained for Python 3.6 and later.

index *: array-like or Index (1d)*
Values must be hashable and have the same length as *data*. Non-unique index values are allowed. Will default to RangeIndex (0, 1, 2, …, n) if not provided. If both a dict and index sequence are used, the index will override the keys found in the dict.

dtype *: numpy.dtype or None*
If None, dtype will be inferred

copy *: boolean, default False*
Copy input data

Series - functions

- Now let's apply some mathematical functions to this `Series`

```
print(norm_series.shape)     #<- number of rows and columns
```

```
(3,)
```

```
print(norm_series.mean())    #<- series mean
```

```
10.0
```

```
print(norm_series.median()) #<- series median
```

```
10.0
```

```
print(norm_series.std())     #<- series std deviation
```

```
5.0
```

Series - functions

- Here are some ways to count items in a `Series`

```
# Show only unique values.
print(norm_series.unique())
```

```
[ 5 10 15]
```

```
# Show number of unique values.
print(norm_series.nunique())
```

```
3
```

```
# Show counts of unique values.
print(norm_series.value_counts())
```

```
5     1
10    1
15    1
dtype: int64
```

```
# Position of the min value.
print(norm_series.idxmin())
```

```
0
```

```
# Position of the max value.
print(norm_series.idxmax())
```

```
2
```

Series - rank

- We can rank items in ascending order:

```
# Ranks from smallest to largest.
print(norm_series.rank())
```

```
0    1.0
1    2.0
2    3.0
dtype: float64
```

- And in descending order:

```
# Ranks from largest to smallest.
print(norm_series.rank(ascending = False))
```

```
0    3.0
1    2.0
2    1.0
dtype: float64
```

Series - sort and cumulative sum

- We can sort `Series`:

```
# Sorts values.
print(norm_series.sort_values())
```

```
0     5
1    10
2    15
dtype: int64
```

- And find the cumulative sum:

```
# Returns a series that is the cumulative sum of
`norm_series`.
print(norm_series.cumsum())
```

```
0     5
1    15
2    30
dtype: int64
```

Knowledge check

Link: ***https://forms.gle/JQaTy2Dthr1L27Yd9***

Module completion checklist

Objective	Complete
Summarize use cases of Pandas	✔
Demonstrate use of basic operations on series	✔

Congratulations on completing this module!

You are now ready to try Tasks 1-4 in the Exercise for this topic