DATA SOCIETY®

Day 3 - Data Wrangling with Python

"One should look for what is and not what he thinks should be."
-Albert Einstein.

Module completion checklist

Objective	Complete
Demonstrate use of basic operations on series	
Apply basic operations on dataframes	
Load data into Python using pandas	

Recap

- In the last module you:
  - Explored numpy and pandas packages
  - Created, filtered and reshaped NumPy arrays
  - Updated Directory settings for efficient workflow
- Now we will head on to two important objects of Pandas, called **series** and **dataframes**
- We then head on to loading a dataset from our data folder

Import Pandas and os

- Let's import the `pandas` and `os` library
- Note: it is not required that you also import `numpy` in order to use `pandas`

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

Series

- The first `pandas` object we'll learn about is a `Series`
- Think of `Series` as a NumPy array but with ***many additional properties and methods***
- We can create `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
```

- In fact, the values are 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

```
# Go in intervals of month.
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')
```

```
# Not specifying end, but instead the start, freq, and how many 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!

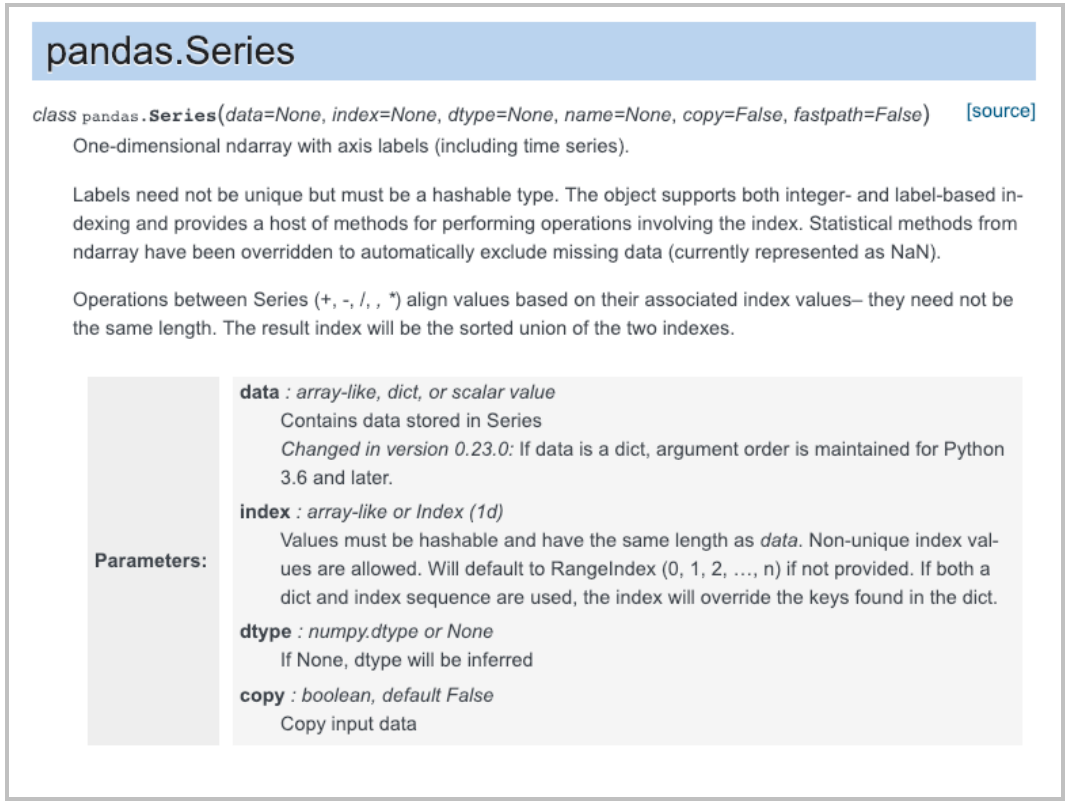
Series methods

- `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, and
- `5`, `10` and `15` are the values at the corresponding index in the 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())
```

```
15    1
10    1
5     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 a series, 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 1

Exercise 1

Module completion checklist

Objective	Complete
Demonstrate use of basic operations on series	✔
Apply basic operations on dataframes	
Load data into Python using pandas	

Dataframes

- Now that we have reviewed `Series`, let's look at what a `dataframe` is
- **A `dataframe` is the single most important object in `pandas`**
  - It is a collection of `series` of equal lengths
  - Just like `series`, dataframes come with many useful methods
- Review complete documentation of the `DataFrame` function ***here***
- For this simple example, we'll build a dataframe using one series similar to what we just built, `Timestamp`
- The second series will be a set of numbers representing the average number of days people were out of office `OOO`

Series to dataframe

- We create a `dataframe` object with the `pd.dataframe` function, and we specify the `Series` we want to include (in this case, it's `times` and `days` out of office)
- We are going to create two `series`:
  - Our first `series` will consist of times
  - We will use the `date_range` method
  - A second `series` will be made of the average number of days people were out of office, constructed from a list of numbers

```
# Series 1 - times:
times = pd.date_range(start = '20170101', end = '20170630', freq = 'M')

# Series 2 - days out of the office:
days = pd.Series([2, 2, 6, 6, 2, 3])
```

Generate dataframe from series

- Create a `dataframe` using dictionary-like syntax:
  - Dictionary `keys` become column names of the dataframe, and
  - Dictionary `values` become column values
- Inspect the `dataframe` by looking at the first few rows, using `.head()`

```
# Create a dataframe from the two series we just created, as a dictionary.
average_ooo = pd.DataFrame({'Timestamp': times, 'OOO': days})

# View the first few rows of the dataframe, using the pandas function `.head()`.
print(average_ooo.head())
```

```
   Timestamp  OOO
0 2017-01-31    2
1 2017-02-28    2
2 2017-03-31    6
3 2017-04-30    6
4 2017-05-31    2
```

Look-up dataframe information

- As with arrays and lists, we can look up the `type` of the created object as well as its `shape`

```
# Look up the type of object.
print(type(average_ooo))
```

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

```
# Look up its shape.
print(average_ooo.shape)
```

```
(6, 2)
```

`DataFrame` is a rectangular object - it will have rows and columns just like a matrix:

1. The first number in parentheses gives us the number of rows, and
2. The second number is the number of columns

Dataframe description metrics

- There are many metrics you can pull from a `DataFrame` object
- We will now review some key metrics that will help us understand our data
  - `.columns` returns column names
  - `.info()` gives us some extra info about each column like its data type, and how many null values it has
  - `.describe()` computes summary statistics on any numeric column

Dataframe description metrics

- Now, let's preview these metrics on the `average_ooo` dataset

```
print(average_ooo.columns)
```

```
Index(['Timestamp', 'OOO'], dtype='object')
```

```
print(average_ooo.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6 entries, 0 to 5
Data columns (total 2 columns):
 #   Column     Non-Null Count  Dtype         
---  ------     --------------  -----         
 0   Timestamp  6 non-null      datetime64[ns]
 1   OOO        6 non-null      int64         
dtypes: datetime64[ns](1), int64(1)
memory usage: 224.0 bytes
None
```

```
print(average_ooo.describe())
```

```
              OOO
count    6.000000
mean     3.500000
std      1.974842
min      2.000000
25%      2.000000
50%      2.500000
75%      5.250000
max      6.000000
```

Extracting a single column

- To extract a column, just put its name in quotation marks into square brackets like this: `data_frame['column_name']`

```
print(average_ooo['Timestamp'])
```

```
0   2017-01-31
1   2017-02-28
2   2017-03-31
3   2017-04-30
4   2017-05-31
5   2017-06-30
Name: Timestamp, dtype: datetime64[ns]
```

- The resulting object is a `Series` type
- If you would like to get a `DataFrame` object with a single column, then pass the `list` with a single column name into the square brackets like this: `data_frame[['column_name']]`

Extracting multiple columns

- To extract multiple columns, just pass a list of columns

```
print(average_ooo[['Timestamp', 'OOO']])
```

```
   Timestamp  OOO
0 2017-01-31    2
1 2017-02-28    2
2 2017-03-31    6
3 2017-04-30    6
4 2017-05-31    2
5 2017-06-30    3
```

Extracting a single row

- To extract a particular row from a dataframe, we can use a syntax similar to what we used for `ndarray`, but with one small change: **we must use the `iloc` method!**

```
june_ooo = average_ooo.iloc[5, :]
print(june_ooo)
```

```
Timestamp    2017-06-30 00:00:00
OOO                            3
Name: 5, dtype: object
```

```
june_ooo = average_ooo.iloc[5] #<- equivalent without the colon
print(june_ooo)
```

```
Timestamp    2017-06-30 00:00:00
OOO                            3
Name: 5, dtype: object
```

- `iloc` can be used when the index label of a dataframe is numeric (*integer* in *i* loc) or if you aren't sure of the index label

Working with dataframe indices

- Dataframes in `pandas` have a property called the `index`
- The `index` serves many purposes and is an important concept to understand within `pandas`

- Main purposes:
  - **identifying** data using known indicators, which is important for analysis, visualization, and interactive console display
  - **enabling** automatic and explicit data alignment
  - **allowing** intuitive getting and setting of subsets of the dataset

Index for our dataset

- The `average_ooo` dataframe has an unlabeled column with the numbers `0` to `5`, this is the `index` of our dataframe
- By default, the `index` is simply the row number (starting with `0`), but it can sometimes make sense to use something more descriptive for the index
- We are going to use `set_index` to set our index in `average_ooo`

```
# Let's use the `Timestamp` column as our new index.
average_ooo = average_ooo.set_index('Timestamp')
print(average_ooo)
```

```
             OOO
Timestamp
2017-01-31     2
2017-02-28     2
2017-03-31     6
2017-04-30     6
2017-05-31     2
2017-06-30     3
```

Looking up by the new index

- Now the rows of our dataframe are indexed by the time stamp and the `Timestamp` column has been removed
- This makes it really easy to look up values corresponding to a particular time stamp
- To do this, we now use the `.loc()` method

```
print(average_ooo.index)
```

```
DatetimeIndex(['2017-01-31', '2017-02-28', '2017-03-31', '2017-04-30',
               '2017-05-31', '2017-06-30'],
              dtype='datetime64[ns]', name='Timestamp', freq=None)
```

```
# Look up a specific row by index.
print(average_ooo.loc['2017-02-28'])
```

```
OOO    2
Name: 2017-02-28 00:00:00, dtype: int64
```

Loc vs. iloc

- Notice we used `loc` not `iloc` like in the first example
- The "`i`" in `iloc` stands for integer
- We use `loc` to get rows or columns with particular **labels** from the index
- We use `iloc` to get rows or columns at particular **positions** from the index

- As it turns out, the row we wanted was in position 1, so we could also say:

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

```
OOO    2
Name: 2017-02-28 00:00:00, dtype: int64
```

Reset the index

- To change the index back to the default, use `.reset_index()`, it will
  - Change the Index back to `0..5`
  - Move the `Timestamp` values back into the dataframe as a column

```
average_ooo = average_ooo.reset_index()
print(average_ooo.index)
```

```
RangeIndex(start=0, stop=6, step=1)
```

```
# You can see that now `Timestamp` is once again a column vs. what it looked like when it was an index.
print(average_ooo)
```

```
   Timestamp  OOO
0 2017-01-31    2
1 2017-02-28    2
2 2017-03-31    6
3 2017-04-30    6
4 2017-05-31    2
5 2017-06-30    3
```

Knowledge check 2

Module completion checklist

Objective	Complete
Demonstrate use of basic operations on series	✔
Apply basic operations on dataframes	✔
Load data into Python using pandas	

Loading data into Python using pandas

- Now that we know some of the key functions of pandas, we can work with actual datasets
- We will be using two datasets
- **One dataset in slides, to learn the concepts**
  - Costa Rica household poverty data by the Inter-American Development Bank

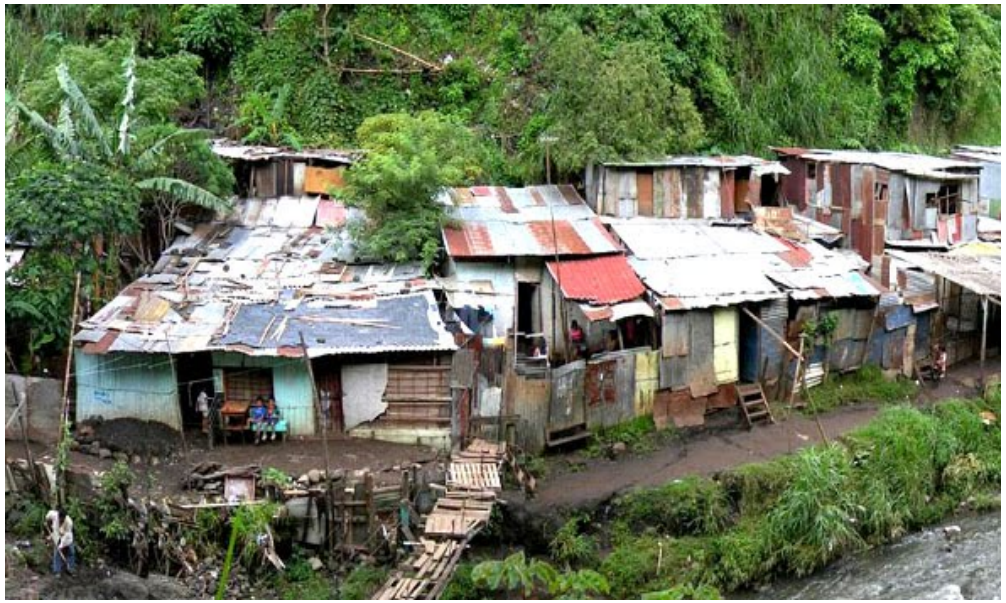
- **One dataset for your exercises**
  - Worldwide tuberculosis estimates by the World Health Organization (WHO)

Costa Rican poverty: case study

- We will be diving into a case study from the **Inter-American Development Bank (IDB)**
- The **IDB** conducted a competition amongst data scientists on ***Kaggle.com***
- Agencies would like to view families' observable household attributes like the material of their walls and ceiling, or the assets found in the home, to classify them and predict their level of need

Costa Rican poverty

- The given dataset contains variables about:
  - **Households**: features of the house the family lives in region, etc.
  - **Individuals in the household** : gender, age, education, etc.
- The **Target** has four categories:
  - extreme poverty
  - moderate poverty
  - vulnerable households
  - non-vulnerable households



Reading data from a file

- Your data will most likely be stored either in a database or in a file, **you will need to import it into your environment**
- A common data format used for storing and sharing data is a `csv` file format (i.e. comma separated value)
- `pandas` has a `read_csv` function to import such files
- In your course materials, you should have a `csv` file called `household_poverty.csv` - we will use this dataset to experiment with various dataframe functions

Reading data from a file

- In addition to csv data, Pandas can read a variety of formats, including Excel, JSON, HTML, Stata, SAS, and even from a SQL connection - the full list of readable and writable file formats is available ***here***.
- **Remember to set your data directory before you begin**
- You *MUST* be pointed to the directory where your data is located

Directory settings

- In order to maximize the efficiency of your workflow, you should encode your directory structure into `variables`
- Let the `main_dir` be the variable corresponding to your `skill-soft` folder

```
# Set `main_dir` to the location of your `skill-soft` folder (for Linux).
main_dir = "/home/[username]/Desktop/skill-soft"
```

```
# Set `main_dir` to the location of your `skill-soft` folder (for Mac).
main_dir = '/Users/[username]/Desktop/skill-soft'
```

```
# Set `main_dir` to the location of your `skill-soft` folder (for Windows).
main_dir = "C:\\Users\\[username]\\Desktop\\skill-soft"
```

```
# Make `data_dir` from the `main_dir` and
# remainder of the path to data directory.
data_dir = main_dir + "/data"

# Create a plot directory to save our plots
plot_dir = main_dir + "/plots"
```

Setting working directory

- Set working directory to `data_dir`

```
# Set working directory.
os.chdir(data_dir)
```

```
# Check working directory.
print(os.getcwd())
```

```
/home/[user-name]/Desktop/skill-soft/data
```

Read data from csv file

- We are now going to use the function `read_csv` to read in our `household_poverty` dataset

```
household_poverty = pd.read_csv('household_poverty.csv')
print(household_poverty.head())
```

```
   male      hh_ID  rooms  ...  water_inside  years_of_schooling  Target
0     1  21eb7fcc1      3  ...             1                  10       4
1     1  0e5d7a658      4  ...             1                  12       4
2     0  2c7317ea8      8  ...             1                  11       4
3     1  2b58d945f      5  ...             1                   9       4
4     0  2b58d945f      5  ...             1                  11       4

[5 rows x 14 columns]
```

- We now know how to read data into `pandas`!

Exercise 2

Module completion checklist

Objective	Complete
Demonstrate use of basic operations on series	✔
Apply basic operations on dataframes	✔
Load data into Python using pandas	✔

Summary and next steps

In this module, we:

- Performed basic operations on Series
- Learned to use Dataframes
- Loaded data sets into Python using Pandas

In the next module, we will:

- Review loaded dataset using Pandas functions
- Summarize and reshape data using Pandas

This completes our module

Congratulations!