Introduction to pandas Part 01

References

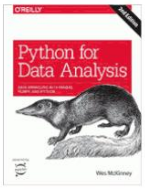
Links:

1. https://pandas.pydata.org/ pandas homepage
2. https://pandas.pydata.org/pandas-docs/stable/ Download documentation v1.3.5 (12 December 2021). Can download pdf (14.7MB – ***3609 pages***) or Zipped HTML (35.4MB). In the documentation, you can **search** for pandas functions and other stuff.
3. https://pandas.pydata.org/pandas-docs/stable/getting_started Getting started with pandas
4. https://pandas.pydata.org/pandas-docs/stable/getting_started/intro_tutorials/index.html Getting started Tutorials
5. https://pandas.pydata.org/pandas-docs/stable/user_guide/10min.html ***10 minutes to pandas.*** This is a short introduction to pandas, geared mainly for new users
6. https://pandas.pydata.org/pandas-docs/stable/user_guide/index.html pandas user Guide
7. https://pandas.pydata.org/pandas-docs/stable/reference/index.html pandas API Reference
8. https://pandas.pydata.org/pandas-docs/stable/getting_started/tutorials.html This is a guide to many pandas **tutorials**, geared mainly for new users. There are various links on that page
9. https://pandas.pydata.org/pandas-docs/stable/user_guide/cookbook.html pandas **cookbook**. This is a repository for short examples and links for useful pandas recipes
10. https://realpython.com/pandas-python-explore-dataset/ Using Pandas and Python to Explore Your Dataset – from Real Python
11. https://www.tutorialspoint.com/python_pandas/ Tutorialspoint Python pandas Tutorial
12. https://www.w3schools.com/python/pandas_tutorial.asp Pandas Tutorial by w3schools

Books:

Python for Data Analysis - Data Wrangling...Pandas, NumPy, and IPython,2nd (Wes McKinney Oreilly 2017) – **Chp05 Getting Started with pandas**

Author is the pandas developer. Book is advertised on the pandas website: https://pandas.pydata.org/



Source code can be downloaded from: https://github.com/wesm/pydata-book

Note: Downloaded HTML and use **index.html**. Do search and examples will be given at the bottom of page. Demo in class.

Note: Word document has been provided to students so that they can insert **code** and **results** after running examples provided!

Introduction

pandas is an open-source Python library providing data structures and data manipulation tools designed to make data cleaning and analysis fast and easy in Python. pandas is often used together with other python libraries such as NumPy, scikit-learn etc.

```
pip show pandas
Requires: pytz, python-dateutil, numpy
Required-by: statsmodels, seaborn
```

Note: Results shown above using show depends on the packages installed in your computer

pandas library is built on top of numPy. It uses most of the functionalities of numPy. It is suggested that you have a basic knowledge of numpy before studying pandas. Basics were covered in `chapter 9: Introduction to Numpy`.

While pandas adopts many coding idioms from numPy, the biggest difference is that pandas is designed for working with tabular or **heterogeneous** data. numPy, by contrast, is best suited for working with **homogeneous** numerical array data.

Demo: EACountries.csv

Installation

- `pip install pandas`
- pandas is part of anaconda distribution. Can use conda in anaconda to update pandas
- Latest version of pandas as per 10 January 2022 is v1.3.5 which was released on 12 December 2021.

Note:

- `pandas provides too many ways to accomplish the same task (so it's hard to figure out the best practice) => There is more than one way to do everything in pandas!!`

Demo: In the notebook

Pandas data structures

https://pandas.pydata.org/pandas-docs/stable/user_guide/dsintro.html Intro to data structures

- Pandas deals with two data structures: `Series` and `DataFrame`.
- `DataFrame` is a container of `Series` and is widely used.

pandas Series

- Series is a one-dimensional array-like structure with **homogeneous** data.
- It can hold data of any type (`integer, string, float,` etc.).
- A series is similar to a NumPy array, but it differs by having an **`index`**, which allows for much richer `lookup` of items instead of just a zero-based array index value.
- An example of Structure of Series is shown below.

Index	Value
0	Tanzania
1	Kenya
2	Uganda
3	Rwanda
4	Burundi
5	South Sudan

Creating pandas Series

https://pandas.pydata.org/pandas-docs/stable/reference/series.html

A pandas Series can be created using the following syntax:

```
pandas.Series(data, index, dtype) # note: S in Series uppercase
```

Parameters are:

- **data**: data takes various forms like ndarray, list, dictionary, constants
- **index**: Index values have same length as data. Default to `np.arange(n)` if no index is passed, where n is the length of data.
- **dtype**: dtype is for data type. If not specified, data type will be **inferred** from data provided

Various ways of creating a series:

- **Create an Empty Series**
- **Create Series from a list without specifying index**
- **Create a Series from ndarray**
- **Create a Series from dictionary.**
    - dictionary `keys` are used to construct index.
- **Create a Series and specify index**

- **Create a Series from Scalar**

Series Basic Functionality

https://pandas.pydata.org/pandas-docs/stable/reference/series.html

SN	Attribute or method	Description
1.	index	Return the index of a series
2.	dtype	Returns the data type of the series
3.	empty	Returns True if series is empty else return False
4.	ndim	Returns the number of dimensions of the underlying data, by definition a series has ndim=1
5.	size	Returns the number of elements in the underlying data
6.	values	Returns the Series values as ndarray
7.	head(n)	Returns the **first n rows**. The default number of elements to display is **five**, **(**using `head()`**)** but you may pass a **custom number**.
8.	tail(n)	Returns the **last n rows**. The default number of elements to display is **five, (**using `tail()`**)** but you may pass a **custom number**.
9.	shape	Return number of rows (for a series) as a tuple.

Demo: Basic series functionalities In the notebook

Changing the index of a series

A Series's index can be altered in-place by assignment.

Demo: In the notebook

Changing data type of a series

Use **astype** function.

Demo: In the notebook

Accessing Data from Series with Position

Data in the series can be accessed similar to that in numpy.

The counting starts from zero for the array, which means the first element is stored at zeroth position and so on.

Retrieve items from series using numeric index

Demo: In the notebook

Note: `s[[3]]` – retrieves the fourth element including the index!

```
s[[3]]=
 RW    Rwanda
```

Retrieve Data Using slices

Demo: In the notebook

Note: indices values are also returned

Retrieve Data Using Label (Index)

A Series is like a fixed-size dict in that you can get and set values by **index** label.

Demo: In the notebook

Note:
To get also the index, use double square brackets (i.e. **s[['TZ']]**)

```
s['TZ']=  Tanzania
s[['TZ']]=  TZ    Tanzania
```

Note: For multiples elements, the indices should be enclosed in a list.

If an index specified does not exist in the series, then a KeyError results.

Slicing using index labels

Demo: In the notebook

Note that the boundaries are included!

Creating a view of a series from existing series

This is similar to numpy. A view is created, ***and any change in the view will affect the base series***.

Demo: In the notebook

To create a new series which is a copy use `.copy()` function as it was in numpy.

Demo: In the notebook

Rows selection using .loc

- `.loc[]` is used to access a group of rows and columns by **label**(s) or a **boolean array**.

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.loc.html

- `.loc[]` is primarily `label` based

Warning: Note that contrary to usual python slices, both the **start** and the **stop** (i.e. boundaries) are **included.**

Demo: In the notebook

Note:

```
s.loc['TZ']=
 Tanzania

s.loc[['TZ']]=
 TZ    Tanzania
```

Rows selection using .iloc

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.iloc.html

`.iloc` is primarily **integer position** based (from 0 to length-1 of the axis), but may also be used with a boolean array.

`.iloc` will raise IndexError if a requested indexer is out-of-bounds, except **slice indexers** which allow out-of-bounds indexing (this conforms with python/numpy slice semantics).

Using iloc when Index is string

Demo: In the notebook

Using iloc when Index is numeric

Demo: In the notebook

Note: Using slices with iloc **DOES NOT** include upper boundaries

Inserting a new value in a series

Specify a new index and a new value.

Demo: In the notebook

Removing items from a series using drop function

```
Series.drop(labels=None, axis=0, index=None, columns=None, level=None, inplace=False, errors='raise')
```

Return Series with specified `index labels` removed.

labels : single label or list-like. Index labels to drop.

index, columns : None. **Redundant** for application on Series

inplace : bool, default False. If True, do operation inplace and return None.

Using drop when index is a label

Demo: In the notebook

Using drop with numeric indices

Demo: In the notebook

Using del to delete a single item

Demo: In the notebook

Note that pythonic way is to use **drop** function

value_counts function

Return a Series containing counts of unique values.

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.value_counts.html

Syntax:

```
Series.value_counts(self, normalize=False, sort=True, ascending=False, bins=None, dropna=True)
```

The resulting object will be in descending order so that the first element is the most frequently-occurring element. **Excludes NaN (NA) values by default using dropna=True**.

Normalize : boolean, default False. Used if percentages are desired.

Demo: In the notebook

count function

syntax: `Series.count(level=None)`

Return number of non-NA/null observations in the Series

Return: Number of non-null values in the Series.

Demo: In the notebook

Unique function

Return unique values of Series object.

Demo: In the notebook

isin function

`syntax:  series.isin(values)`

Compute boolean array indicating whether each Series value is contained in the passed sequence of values. Return a boolean Series showing whether each element in the Series matches an element in the passed sequence of values exactly. **Note**: Values must be enclosed in a list even a single value!

Demo: In the notebook

DataFrame

A DataFrame is a **two-dimensional** array structure with **`heterogeneous`** (consisting of many `different` kinds of thing) data. Data is aligned in a tabular

fashion in rows and columns. Pandas dataframe structure can be thought as a **spreadsheet** data representation. Example of DataFrame is shown in table below where unit of population (of a country) is million.

Index	Name	Capital	Population
TZ	Tanzania	Dodoma	60
KE	Kenya	Nairobi	51.6
UG	Uganda	Kampala	45
RW	Rwanda	Kigali	12.65
BI	Burundi	Gitega	11.38
SS	South Sudan	Juba	12.58

Creating pandas DataFrame

A pandas DataFrame can be created using the following constructor:

```
pandas.DataFrame(data, index, columns, dtype)
```

Parameters are:

- **data**: data takes various forms like ndarray, series, lists, dict, constants and also another DataFrame.
- **index**: For the row labels, the index to be used for the resulting frame is optional Defaults to `np.arange(n)` if no index is passed.
- **columns:** For column labels, the optional default syntax is `np.arange(n)`. This is only true if no index is passed.
- **dtype**: Data type of each column.

Demo: Following are demonstrated in the notebook

- **Create an Empty DataFrame**
- **Create a DataFrame from Lists without index and columns names**

Note: Since index and columns names were not specified, then default index `(0,1,2,..)` and columns `names (0,1,2..)` are used.

- **Create a DataFrame from Lists and specifying index and columns names**
- **Create a DataFrame from Dict**

Note: The names of the columns in a DataFrame are accessible via the .columns Property. Example: `df.columns`

DataFrame Basic Functionality

The following tables lists down some attributes or methods that help in DataFrame Basic Functionality.

https://pandas.pydata.org/pandas-docs/stable/reference/frame.html

SN	Attribute or method	Description
1.	`T`	Transposes rows and columns.
2.	`dtypes`	Returns the dtypes of this object.
3.	`columns`	The column labels of the DataFrame.
4.	`index`	The index (row labels) of the DataFrame.
5.	`empty`	Indicator whether DataFrame is empty.
6.	`ndim`	Return an int representing the number of axes / array dimensions.
7.	`shape`	Return a tuple representing the dimensionality of the DataFrame
8.	`size`	Return an int representing the number of elements in this object.
9.	`values`	Return a Numpy representation of the DataFrame.
10.	`head(n)`	Returns the first n rows. The default number of elements to display is **five**, but you may pass a custom number.
11.	`tail(n)`	Returns the last n rows. The default number of elements to display is **five**, but you may pass a custom number.

Demo: In the notebook

Getting info of dataframe using info()

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.info.html

This method prints information about a DataFrame including the index dtype and column dtypes, non-null values and memory usage.

Syntax:

```
DataFrame.info(self, verbose=None, buf=None,
max_cols=None, memory_usage=None, null_counts=None)
```

Demo: In the notebook

Reading and Writing Data in csv Format

References:

- https://www.tutorialspoint.com/python_pandas/python_pandas_io_tool.htm
- https://pandas.pydata.org/pandas-docs/stable/user_guide/io.html IO tools
- https://en.wikipedia.org/wiki/Comma-separated_values Comma-separated values

- Python for Data Analysis - Data Wrangling...Pandas, NumPy, and IPython,2nd(McKinney Oreiily 2017) – **Chapter 6 - Data Loading, Storage, and File Formats**

CSV formatted data is likely to be one of the **most** common forms of data you may use in pandas. It is an easy format to use and is **commonly used** as an export format for spreadsheet applications such as Excel.

A CSV is a file consisting of multiple lines of text-based data, with values separated by commas. It can be thought of as a table of data similar to a single sheet in a spreadsheet program. Each row of data is in its own line in the file, and each column for each row is stored in text format, with a comma separating the data in each column.

Writing Data to csv Format

Using DataFrame's `to_csv` method, data can be written to a comma-separated value (csv) file.

https://pandas.pydata.org/pandas-docs/stable/user_guide/io.html#io-store-in-csv

Parameters to pass to the function can be read from above link as shown below.

```
DataFrame.to_csv(path_or_buf=None, sep=',', na_rep='', float_format=None,
columns=None, header=True, index=True, index_label=None, mode='w',
encoding=None, compression='infer', quoting=None, quotechar='"',
line_terminator=None, chunksize=None, date_format=None, doublequote=True,
escapechar=None, decimal='.', errors='strict', storage_options=None)
```

Demo: In the notebook

Observation:

If the above saved file (`EAcountries.csv`) is opened in excel, it will look as follows:

	A	B	C	D
1		name	capital	pop
2	TZ	Tanzania	Dodoma	60
3	KE	Kenya	Nairobi	51.6
4	UG	Uganda	Kampala	45
5	RW	Rwanda	Kigali	12.65
6	BI	Burundi	Gitega	11.38
7	SS	South Sud	Juba	12.58

If the file is opened in text file, it will look as follows:

```
,name,capital,pop
TZ,Tanzania,Dodoma,60.0
KE,Kenya,Nairobi,51.6
```

```
UG,Uganda,Kampala,45.0
RW,Rwanda,Kigali,12.65
BI,Burundi,Gitega,11.38
SS,South Sudan,Juba,12.58
```

Note: In the heading in text editor, there is a comma before `name`.

Reading Data from csv Format

Use **`read_csv`** function with default options to read data which was saved to `EAcountries.csv`. Parameters to pass to the function can be read from `the`

https://pandas.pydata.org/pandas-docs/stable/user_guide/io.html#io-read-csv-table

```
pandas.read_csv(filepath_or_buffer, sep=<object object>, delimiter=None,
header='infer', names=None, index_col=None, usecols=None, squeeze=False,
prefix=None, mangle_dupe_cols=True, dtype=None, engine=None,
converters=None, true_values=None, false_values=None,
skipinitialspace=False, skiprows=None, skipfooter=0, nrows=None,
na_values=None, keep_default_na=True, na_filter=True, verbose=False,
skip_blank_lines=True, parse_dates=False, infer_datetime_format=False,
keep_date_col=False, date_parser=None, dayfirst=False, cache_dates=True,
iterator=False, chunksize=None, compression='infer', thousands=None,
decimal='.', lineterminator=None, quotechar='"', quoting=0,
doublequote=True, escapechar=None, comment=None, encoding=None,
dialect=None, error_bad_lines=True, warn_bad_lines=True,
delim_whitespace=False, low_memory=True, memory_map=False,
float_precision=None, storage_options=None)
```

The parameter: `index_col=0` should be included to show that the first column represents an index. If it is not included, then unnamed column is created.

Demo: In the notebook

Index objects

Checking if a certain item exists in the index or in the columns names.

Demo: In the notebook

Creating a series from a dataframe

Demo: In the notebook

Selecting a column

A column in a DataFrame can be retrieved as a Series either by dict-like notation example `df['columnName']` or by attribute example `df.columnName`.

More than one column can also be selected by using a `list` of columns names.

Demo: In the notebook

Important: `df['columnName']` works for any column name, but `df.columnName` **ONLY** works when the column name is a valid Python identifier (**no space** in the column name i.e single word).

Bracket notation will always work, whereas **dot notation** has limitations:

- Dot notation doesn't work if there are **spaces** in the Series name
- Dot notation doesn't work if the Series has the same name as a **DataFrame method or attribute** (like 'head' or 'shape')
- Dot notation can't be used to define the name of a **new Series**

Note: The column returned from indexing a DataFrame is a **view** on the underlying data, **NOT** a copy. Thus, any in-place modifications to the Series will be reflected in the DataFrame. The column can be explicitly copied with the Series's **copy** method.

Rows and columns selection using loc

Rows can be selected by passing row label to a **loc** function

Access a group of rows and columns by label(s) or a boolean array.
.loc[] is primarily label based, but may also be used with a boolean array.

Demo: In the notebook

Warning:

Note that contrary to usual python slices, both the **`start`** and the **`stop`** are **`included`**

Row and columns Selection using iloc

Rows and columns can be selected by passing integer location to an **iloc** function.

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.iloc.html

Purely **integer-location** based indexing for selection by position.
.iloc[] is primarily **integer position** based (from 0 to length-1 of the axis), but may also be used with a boolean array.

.iloc will raise IndexError if a requested indexer is out-of-bounds, except *slice* indexers which allow out-of-bounds indexing (this conforms with python/numpy *slice* semantics).

Demo: In the notebook

Boolean indexing – Very powerful in filtering data which meets a certain condition

This is the use of boolean vectors to filter the data. Use one of the six comparison operators: ==, >, >=, <, <=, !=

The comparison operators can be combined with: | for or, **&** for and, and ~ for not. **These must be grouped by using parentheses**, since by default Python will evaluate an expression such as `df.A > 2 & df.B < 3` as `df.A > (2 & df.B) < 3`, while the desired evaluation order is `(df.A > 2) & (df.B < 3)`.

Using a boolean vector to index a Series works exactly as in a NumPy ndarray.

Rows selection are based upon values in specific columns.

Demo: In the notebook

Renaming index labels or Columns labels

The `rename()` method allows you to relabel an axis based on some mapping (a dict or Series).

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.rename.html

```
DataFrame.rename(mapper=None, index=None, columns=None,
axis=None, copy=True, inplace=False, level=None)
```

To change column names using **rename** function in pandas, one needs to specify a mapper, a dictionary with old name as `keys` and new name as `values`. **inplace=True** can be used to change column names in place.

To change row indexes or row names, just use index argument and specify index. Use old name as `keys` and new name as `values`.

Demo: In the notebook

The **rename()** method provides an **inplace** named parameter, which by default is False and copies the underlying data. Pass **inplace=True** to rename the data in place.

Adding a new Column

Column addition can be added by using `df['newColName']=.....`

Demo: In the notebook

Note: New columns **cannot** be created with the dot notation syntax. The following statement:

```
df.currency=currencyNames
```

Will give the error: Pandas doesn't allow columns to be created via attribute na me.

Note: A new column can also be a function of existing column (*calculated field*). Example:

```
df['newpop]=df['pop ']*1.15   # newpop is 15% increment
                               # of pop.
```

Deleting row(s) and column(s) using drop

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.drop.html

`syntax:`

```
DataFrame.drop(labels=None, axis=0, index=None, columns=None,
level=None, inplace=False, errors='raise')
```

By default: axis=0 (index axis). Use inplace=True to effect complete deletion.

Drop specified labels from rows

This is similar to what was done in dropping a row in a pandas series

Demo: In the notebook

```
df.drop('RW',inplace=True)
```

The above statement will affect df because of using `inplace=True.`

Warning! Be careful with the inplace, as it destroys any data that is dropped

Drop column(s)

You need to specify the column names with the `axis=1` or `axis='columns'`.

Demo: In the notebook

Drop column using del keyword

del keyword can be used to delete a single column. Syntax is:

```
del dataFramename[columnName]
```

Example: `del df['name']  #  del df.name does not work`

Demo: In the notebook

Sorting index using sort_index function

Sort object by labels (along an axis)

Syntax:

```
DataFrame.sort_index(axis=0, level=None, ascending=True, inplace=False,
kind='quicksort', na_position='last', sort_remaining=True, by=None)
```

- **axis**: index, columns to direct sorting
- **ascending**: boolean, default True. Sort ascending vs. descending
- **inplace**: bool, default False. if True, perform operation in-place

Demo: In the notebook

Note: This function has `inplace` parameter

Sorting by Value

sort_values() is the method for sorting by values. It accepts a '**by**' argument which will use the column name of the DataFrame with which the values are to be sorted.

Syntax:

```
DataFrame.sort_values(by, axis=0, ascending=True, inplace=False,
kind='quicksort', na_position='last')
```

Sort by the values along either axis

by : str or list of str. Name or list of names to sort by.

- **axis** : {0 or 'index', 1 or 'columns'}, default 0. Axis to be sorted
- **ascending** : bool or list of bool, default True. Sort ascending vs. descending. Specify list for multiple sort orders. If this is a list of bools, must match the length of the by.
- **inplace** : bool, default False. if True, perform operation in-place
- **na_position** : {'first', 'last'}, default 'last'. first puts NaNs at the beginning, last puts NaNs at the end

Demo: In the notebook

Note:

- **sort_values()** provides a provision to choose the algorithm from mergesort, heapsort and quicksort. Quicksort is the default although *Mergesort is the only stable algorithm*. Use kind parameter to specify the algorithm.
- Any missing values (NaN) are sorted to the end of the Series by default. Can change that using `na_position` parameter.

Summarizing data statistics using describe function

The **describe()** function computes a summary of statistics pertaining to the DataFrame columns.

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.describe.html

Syntax:

```
DataFrame.describe(self, percentiles=None, include=None,
exclude=None)
```

Generate descriptive statistics that summarize the central tendency, dispersion and shape of a dataset's distribution, **excluding NaN values**.

For **numeric data**, the result's index will include count, mean, std, min, max as well as lower, 50 and upper percentiles. By default the lower percentile is 25 and the upper percentile is 75. The 50 percentile is the same as the median.

For **object data** (e.g. strings or timestamps), the result's index will include count, unique, top, and freq. The top is the most common value. The freq is the most common value's frequency. Timestamps also include the first and last items.

'include' is the argument which is used to pass necessary information regarding what columns need to be considered for summarizing. Takes the list of values; **by default, 'number'.**

- **object** − Summarizes String columns. Example:
  `print(df.describe(include='object')) # strings columns only`
- **number** − Summarizes Numeric columns. Example:
  `print(df.describe()) # Numeric columns only`

- **all** – Summarizes all columns together (Should not pass it as a list value). Example: `print(df.describe(include='all')) # numeric and string columns`

Demo: In the notebook

Aggregate functions

The following table list down the important functions under Descriptive Statistics in Python Pandas functions:

SN	Function	Description
1.	**count()**	Number of non-null observations
2.	**sum()**	Sum of values
3.	**mean()**	Mean of Values
4.	**median()**	Median of Values
5.	**mode()**	Mode of values
6.	**std()**	Standard Deviation of the Values
7.	**min()**	Minimum Value
8.	**max()**	Maximum Value
9.	**abs()**	Absolute Value
10.	**prod()**	Product of Values
11.	**cumsum()**	Cumulative Sum
12.	**cumprod()**	Cumulative Product

Note: Since DataFrame is a `Heterogeneous` data structure, generic operations **don't work** with all functions.

- Functions like **sum()**, **cumsum()** work with **both numeric and character** (or) string data elements without any error. Though in practice, character aggregations are never used generally, these functions do not throw any exception.
- Functions like **abs()**, **cumprod()** throw exception when the DataFrame contains character or string data because such operations cannot be performed.
- Functions **min()** and **max()** work with **both numeric and character**

Demo: In the notebook

*=== END OF **PART 01** ===*