tutorial015-Pandas-Introduction

November 3, 2022

1 Pandas I - Introduction to fundamental objects

`pandas` is the Python library that structures and simplifies data manipulation and analysis. The name, pandas is derived by **pan**el and **da**ta. The library indeed focusses on representing data as tables (`DataFrames`), indices (`Index`) and data series (`Series`).

`pandas` uses `NumPy` arrays to represent data. It provides specific functions to manipulated data as `code objects`. This means that to use pandas you need to import also NumPy. `pandas` requires and builds on top of `NumPy`.

There are three main data object types in pandas:

- `Series` - A mutable, one-dimensional array of indexed data.
- `DataFrames` - A two-dimensional, size-mutable, potentially heterogeneous tabulated data structure.
- `Index` - An one-dimensional, immutable array or ordered set (technically a multi-set, as Index objects may contain repeated values).

Below we will learn a little bit more about these three data objects.

A short history of Pandas.

Wes McKinney started developing what then became pandas while working at the capital management firm Applied Quantitative Research (AQR). Pandas was developed initially as a closed-source project and was made open source in 2009. Pandas is sponsored by NumFOCUS, Inc. that promotes support and sponsorships of python based open source code.

[2]:
```
import numpy as np
import pandas as pd
```

1.0.1 The pandas `Series` object

`Series` are dictionary-like objects. Pandas Series is a one-dimensional array that comes with labels assigned. They similar to NumPy one-dimensional arrays but they are labeled or indexed. So they are built on top of NumPy arrays but come with additional functionality as well as constrains. They can be thought of as a specification of NumPy arrays. For example, let's evaluate the code below.

First we define a pandas Series objects and assign a set of string values to the elements of the object:

[2]:
```
data = pd.Series(['a', 'b', 'c', 'd'])
```

After defining the object let's explore it.

[4]:
```
print (data)
```

```
0    a
1    b
2    c
3    d
dtype: object
```

The data type is defined as `object`, the values we assigned are stored in the second column. The first column is the `index` automatically assigned by pandas to each value. Now, exch value comes with an index. This is a foundamental data organization aspect of pandas. Values always have indeces, because pandas deals with panel data, i.e., tables. So even a one-dimensional array as a Series is assigned indices just like a table is.

We will discuss pandas index objects later.

A Pandas Series can be created directly by assigning values into an array (using `[]`), and that array to a series (`pd.Series()`). Yet, the final product of that series definition create something different than an Array. It creates a set of pairs of values where a label (`index`) is associated to a corresponding value.

Series are capable of holding data of any type (integer, string, float, python objects, etc.). For example:

[5]:
```
data = pd.Series(['string',10, np.nan,0.01])
print(data)
```

```
0    string
1        10
2       NaN
3      0.01
dtype: object
```

Once the Series object is created, the Index is created and it becomes part of the methods of the Series object. This means that the Index to the entries can be retrieved and used to address the corresponding entry. For example, we can call the `Index` as a method and it will return the range, with the min value, the max value and the steps in btween them:

[6]:
```
data.index
```

[6]:
```
RangeIndex(start=0, stop=4, step=1)
```

The index can be used to address the corresponding value in the Series object:

[7]:
```
data[0:3]
```

[7]:
```
0    string
1        10
2       NaN
dtype: object
```

Because the Index in pandas is explictly defined and it becomes a method (this is in contrast with NumPy arrays where indices are impliclty defined and hence not addressable or callable), the pandas Series object becomes much more useful and strctured.

For example, we can create a Series object by explictly assigning values and indices:

[8]:
```
data1 = pd.Series(['string',10, np.nan,0.01],index=[0,3,2,4])

print(data1)
```

```
0    string
3        10
2       NaN
4      0.01
dtype: object
```

Even though `data` and `data1` might look the same, they are not. The indices are different:

[9]:
```
print(data.index)
print(data1.index)
```

```
RangeIndex(start=0, stop=4, step=1)
Int64Index([0, 3, 2, 4], dtype='int64')
```

So, that, if we address the second entry in either object we will get different values:

[10]:
```
print(data[3])
print(data1[3])
```

```
0.01
10
```

One way to think about pandas Series objects is that they are dictionaries where a label is paired to a value, say `label 1` is assigned to `value 1`, or `label 2` to `value 3` etc. Indeed, a pandas Series object can be constructed by hand building a dictionary a set of pairing of labels and values.

For example, let's build a Python dictionary of cognitive health. The dictionary pairs a label to a value:

[11]:
```
cognitive_health = {'happyness':10,
                    'language': 2,
                    'energy': 5,
                    'memory': 3}
```

Note that python dictionaries have attributes (you can type `cognitive_health.` and press `tab` twice to get a list of attributes), yet the python dictionary does not have `index` as an attribute.

The following line will return an error.

```
[12]: cognitive_health.index
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In [12], line 1
----> 1 cognitive_health.index

AttributeError: 'dict' object has no attribute 'index'
```

Python dictionaries can be used to set pandas Series directly:

```
[13]: cognitive_health_Series = pd.Series(cognitive_health)
```

The dictionary has been made into a panda Series and it is now ordered and labelled. So, the following operation will not return an error, but the labels of the Series (the indices):

```
[14]: cognitive_health_Series.index
```

```
[14]: Index(['happyness', 'language', 'energy', 'memory'], dtype='object')
```

Another important property that the pandas Series have and python dictionaries do not is the ability to allow slicing. Whereas, a dictionary would return an error if called as follows:

```
[15]: cognitive_health['happyness':'energy']
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In [15], line 1
----> 1 cognitive_health['happyness':'energy']

TypeError: unhashable type: 'slice'
```

A pandas Series do allow slicing:

```
[16]: cognitive_health_Series['happyness':'energy']
```

```
[16]: happyness    10
      language      2
      energy        5
      dtype: int64
```

To summarize what we have learned about pandas Series:

- They are ordered and labelled one-dimensional arrays.

- They can be populated by assigning
- values only (indeces are automatically assigned): `series = pd.Series(['a','b','c'])`
- values and indices explicitly: `series = pd.Series(['a','b','c'],index = [1,3,2])`
- python dictionaries directly: `dic = {1:'a',2:'b',3:'c'}`, `series = pd.Series(dic)`
- They always come with the property `index`

Complete the following exercise.

- Use the cell below to create a `Pandas Series` containing the months of the year as labels (index) and their duration in months as values (use a dictionary to create the `Series`):

[19]:
```
months_dict = {'january': 31, 'february': 28, 'march' : 31, 'april' : 30, 'may'␣
 ↪: 31, 'june' : 30,
               'july': 31, 'august': 31, 'september': 30, 'october': 31,␣
 ↪'november': 30, 'december': 31}
months_series = pd.Series(months_dict)
print(months_series)
```

```
january      31
february     28
march        31
april        30
may          31
june         30
july         31
august       31
september    30
october      31
november     30
december     31
dtype: int64
```

- Use the cell below to create a Pandas Series containing the months of the year as labels (index) and their duration in months as values (define index and values explicitly, or in other words do not use a dictionary to define the `Series`):

[22]:
```
months_series_2 = pd.Series(['january', 'february', 'march', 'april', 'may',␣
 ↪'june',
                             'july', 'august', 'september', 'october',␣
 ↪'november', 'december'],
                            index = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31,␣
 ↪30, 31])
print(months_series_2)
```

```
31      january
28     february
31        march
30        april
31          may
```

```
30          june
31          july
31        august
30     september
31       october
30      november
31      december
dtype: object
```

1.0.2 Pandas `index` object

Let's briefly discuss the `index` object in pandas.

An `Index` is the pandas object that hosts information regarding the ordering and labels of the arrays inside other objects such as `Series` and `DataFrames`.

Index objects also have many of the attributes familiar from NumPy arrays (e.g., shape, dimensions, size, etc):

[23]:
```python
ind = pd.Index([1, 2, 3, 4, 5])

print(ind.size, ind.shape, ind.ndim, ind.dtype)

print(ind)
```

```
5 (5,) 1 int64
Int64Index([1, 2, 3, 4, 5], dtype='int64')
```

Pandas `Index` objects are immutable.

An index is similar to a NumPy array, but it is immutable. This means that once an Index is defined the values inside the index cannot be changed.

Where arrays can be modified after definition, a pandas `index` cannot. Let's try this. Above we defined `ind` as a pandas `index` and set in the third position the value `3`.

Let's try to change that value, evaluate the following operation in which we attempt to set the value `10` in the third position of `ind`:

[24]:
```python
ind[2] = 10
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In [24], line 1
----> 1 ind[2] = 10

File /opt/homebrew/lib/python3.10/site-packages/pandas/core/indexes/base.py:
↪5340, in Index.__setitem__(self, key, value)
   5338 @final
   5339 def __setitem__(self, key, value):
-> 5340     raise TypeError("Index does not support mutable operations")
```

```
TypeError: Index does not support mutable operations
```

Complete the following exercise.

- Use the cell below to create a `Pandas index` called `PandasIndexOne` containing the days of the week and a corresponding `Numpy array` called `NumpyArrayOne` containing the same values:

```
[32]: PandasIndexOne = pd.Index(['sunday', 'monday', 'tuesday', 'wednesday',␣
       ↪'thursday', 'friday', 'saturday'])
      NumpyArrayOne = np.array(['sunday', 'monday', 'tuesday', 'wednesday',␣
       ↪'thursday', 'friday', 'saturday'])
```

- Show that the `PandasIndexOne` is immutable and `NumpyArrayOne` is not.

```
[34]: NumpyArrayOne[2] = 'monday'
      PandasIndexOne[2] = 'monday'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In [34], line 2
      1 NumpyArrayOne[2] = 'monday'
----> 2 PandasIndexOne[2] = 'monday'

File /opt/homebrew/lib/python3.10/site-packages/pandas/core/indexes/base.py:
 ↪5340, in Index.__setitem__(self, key, value)
   5338 @final
   5339 def __setitem__(self, key, value):
-> 5340     raise TypeError("Index does not support mutable operations")

TypeError: Index does not support mutable operations
```

The error should return the following line at the end:

`TypeError: Index does not support mutable operations`

A pandas `Index` does not allow changes. This is helpful. One way to think about the index is that it is a specialized NumPy array. Specialized means that it has a more narrow scope than the more general goal of a NumPy array. The scope is that to define the (ahem) index of a data frame. Because of this scope (store an index) changes to the values of the array are not allowed, in other words the `Index` is immutable, or cannot be changed after definition by assiging a different value to any of its elements.

If we think about it, this makes sense. Changing a value inside an index of a data frame would change the definition of the data frame and really invalidating the purpuse of the index and of the data frame.

Pandas `Index` objects are designed to facilitate operations on the array and serve the task of keeping track of positions of data entries in the objects.

They support and facilitiate operations such as joining datasets. Because of this the pandas index object follows many operations of the built in python datatype `set`.

Because of this the Index object allows many of operations also served by Python set data structure, such as unions, intersections, differences, and other combinations can be computed in a familiar way.

For example, two pandas index object can be united. This means that the unique indices are combined:

[35]:
```
index1 = pd.Index([1, 2, 3, 4, 5])
index2 = pd.Index([1, 3, 4, 6, 20])

index1.union(index2)
```

[35]:
```
Int64Index([1, 2, 3, 4, 5, 6, 20], dtype='int64')
```

Other logical operations can be performed using pandas index objects, for example intersection (find the common elements):

[36]:
```
index1.intersection(index2)
```

[36]:
```
Int64Index([1, 3, 4], dtype='int64')
```

[27]:
```
myseries = pd.Series([1,3,4,6,2,8])
myseries[index1.intersection(index2)]
```

[27]:
```
1    3
3    6
4    2
dtype: int64
```

In sum, pandas index objects allow performing slicing, indexing operations and facilitate keeping track of, ahem, the indices.

Complete the following exercise.

- Use the cell below to create two `Pandas Series`. The first Series should be called `PandasSeriesOne` contain titles of five songs from your favourite artist as indices and their respective duration (it is fine to only write the first four words of the title). The second Series should be called `PandasSeriesTwo` and contain the songs of your colleague sitting to your right (or left if no one sites to your right) favourite artist as indices and their correspdoning duration as value.

[11]:
```
PandasSeriesOne = pd.Series(['Concorde', 'Maud Gone', 'I Can Change',
                             'Helplessness Blues', 'Never There'], index = [6.
 ↪04, 5.58, 5.52, 5.03, 2.44])
PandasSeriesTwo = pd.Series(['Landslide', 'Stranded Lullaby','In The Flowers',
                             'People-Vultures', 'Telescope'], index = [3.19, 3.
 ↪40, 5.22, 4.46, 3.48])
```

- Use the cell below to find the duration of the songs you and your colleague have in common.

```
[12]: index1 = pd.Index([6.04, 5.58, 5.52, 5.03, 2.44])
      index2 = pd.Index([3.19, 3.40, 5.22, 4.46, 3.48])
      index1.intersection(index2)
```

```
[12]: Float64Index([], dtype='float64')
```

1.0.3 Pandas `DataFrame` objects

Whereas pandas `Index` and `Series` objects are the backbone of the pandas library, the `DataFrame` object is the workhorse of the library. DataFrames have effectively made pandas the library for data science.

The DataFrame is an exstension of the Series object. It is a 2-dimensional, mutable array that it can be conceptualized as either a more general NumPy array, or a specialized Python dictionary.

A DataFrame can be defined most simply by setting some values to it. The indexing and labelling is automatically assigned by pandas. For example, we can create a one-dimensional list and assign the values of the list to a DataFrame:

```
[57]: list = ['a','b','c','d']
      data_frame1 = pd.DataFrame(list)
      print(data_frame1)
```

```
   0
0  a
1  b
2  c
3  d
```

Even though the result might seem very similar to that obtained about with the Series object, in reality, the DataFrame object has assigned not one but two labels, one label for the rows dimension (`[0:3]`) and one for the column dimension (`0`).

Indeed, we can notice from the `print()` output that two dimensions were automatically labelled (indexed) with numbers. Whereas the Series is created as a one-dimensional object, the DataFrame is created as a two-dimensional object.

Another way to think about a DataFrame is that it is a sequence of *aligned* Series objects. What do we mean by that?

Each column in the DataFrame can be thought of as a Series. Each series is labelled by the column index. Importantley, the series are aligned, this means that the set of Series (the columns of the DataFrame) are indexed by a common `Index` object.

Let's take a look at all this.

Let's construct a new Series object similar to `cognitive_health_Series`, the object we created above (make sure that object is still in memeory, if needed re-run that section of the cells). Let's assume that that original series represented the data collected on a subject.

[37]:
```
cognitive_health_Series
```

[37]:
```
happyness    10
language      2
energy        5
memory        3
dtype: int64
```

The new Series will represent data on a second subject. To build the series we will use steps similar to the ones used above. We will first make a python dictionary and then make a pandas Series object out of it.

[38]:
```
cognitive_health_subject2_dict = {'happyness': 15,
                                  'language': 4,
                                  'energy': 9,
                                  'memory': 6}
cognitive_health_subject2_series = pd.Series(cognitive_health_subject2_dict)
cognitive_health_subject2_series
```

[38]:
```
happyness    15
language      4
energy        9
memory        6
dtype: int64
```

Now that we have two pandas Series, we can construct a pandas DataFrame by combining the two Series objects.

To do so, we will first create a single dictionary containing the two series, labelled as `subject 1` and `subject 2` and then use that dictionary to make a DataFrame.

The dictionary for the DataFrame is formed by assigning each subject to a label:

[39]:
```
dict = {'subject 1' : cognitive_health_Series,
        'subject 2' : cognitive_health_subject2_series}
print(dict)
```

```
{'subject 1': happyness    10
language      2
energy        5
memory        3
dtype: int64, 'subject 2': happyness    15
language      4
energy        9
memory        6
dtype: int64}
```

The dictionary can now be used to build a DataFrame:

```
[40]: sample = pd.DataFrame(dict)
      sample
```

```
[40]:            subject 1  subject 2
      happyness         10         15
      language           2          4
      energy             5          9
      memory             3          6
```

```
[41]: cognitive_health_subject2_dict = {'happyness': 15,
                                         'language': 4,
                                         'energy': 9,
                                         'memory': 6,
                                         'attention': 22}
      cognitive_health_subject2_series = pd.Series(cognitive_health_subject2_dict)
      cognitive_health_subject2_series

      dict = {'subject 1' : cognitive_health_Series,
              'subject 2' : cognitive_health_subject2_series}
      print(dict)

      sample = pd.DataFrame(dict)
      sample
```

```
{'subject 1': happyness    10
language      2
energy        5
memory        3
dtype: int64, 'subject 2': happyness    15
language      4
energy        9
memory        6
attention    22
dtype: int64}
```

```
[41]:            subject 1  subject 2
      attention        NaN         22
      energy           5.0          9
      happyness       10.0         15
      language         2.0          4
      memory           3.0          6
```

Excellent. Pandas did its Kung Fu. The dictionary comprising two pandas Series, was organized into a pandas DataFrame.

Next try adding two more subjects to the same DataFrame (`sample`). Let's practice this with a subject that has all values higher than subject 2 by a value of 3 (just add 3) and another subject that has all values lower than subject 2 by a value of 2 (just subtract 3).

[42]:
```
cognitive_health_subject4_dict = {'happyness': 15-2,
                                  'language': 4-2,
                                  'energy': 9-2,
                                  'memory': 6-2}
cognitive_health_subject4_series = pd.Series(cognitive_health_subject4_dict)
cognitive_health_subject4_series
```

[42]:
```
happyness    13
language      2
energy        7
memory        4
dtype: int64
```

[43]:
```
cognitive_health_subject3_dict = {'happyness': 15+3,
                                  'language': 4+3,
                                  'energy': 9+3,
                                  'memory': 6+3}
cognitive_health_subject3_series = pd.Series(cognitive_health_subject3_dict)
cognitive_health_subject3_series
```

[43]:
```
happyness    18
language      7
energy       12
memory        9
dtype: int64
```

[44]:
```
dict = {'subject 1' : cognitive_health_Series,
        'subject 2' : cognitive_health_subject2_series,
        'subject 3' : cognitive_health_subject3_series,
        'subject 4' : cognitive_health_subject4_series,
       }
print(dict)
```

```
{'subject 1': happyness    10
language      2
energy        5
memory        3
dtype: int64, 'subject 2': happyness    15
language      4
energy        9
memory        6
attention    22
dtype: int64, 'subject 3': happyness    18
language      7
energy       12
memory        9
dtype: int64, 'subject 4': happyness    13
language      2
energy        7
memory        4
dtype: int64}
```

[45]:
```python
sample = pd.DataFrame(dict)
sample
```

[45]:
```
           subject 1  subject 2  subject 3  subject 4
attention        NaN         22        NaN        NaN
energy           5.0          9       12.0        7.0
happyness       10.0         15       18.0       13.0
language         2.0          4        7.0        2.0
memory           3.0          6        9.0        4.0
```

[46]:
```python
sample.isna()
```

[46]:
```
           subject 1  subject 2  subject 3  subject 4
attention       True      False       True       True
energy         False      False      False      False
happyness      False      False      False      False
language       False      False      False      False
memory         False      False      False      False
```

The newly created DataFrame has various attributes that we can explore. We can address and extract the columns for example:

[47]:
```python
cols = sample.columns
print(cols)
```

```
Index(['subject 1', 'subject 2', 'subject 3', 'subject 4'], dtype='object')
```

We can address the rows, which in technical terms are called the labels or the index:

[48]:
```python
rows = sample.index
print(rows)
```

```
Index(['attention', 'energy', 'happyness', 'language', 'memory'],
dtype='object')
```

Complete the following exercise.

- Use the cell below to create a Pandas `DataFrame`. Each column of the DataFrame should be the title of song you pick (it is fine to clip the title tothe first 4 words in the title). Each row of the data frame should be the name of one of your colleagues, use 3-4 names. Interview 3-4 of your colleagues and add the value in each cell representing the rating of the corresponding song that each colleague provides using a scale between 0-5, where 5 is an awesome song and 0 is a boring song.

```
[9]: # roommate data
     ryan_rating = {'concorde': 2,
                    'maud gone': 3,
                    'i can change': 5,
                    'helplessness blues': 5,
                    'never there': 4}
     ryan_series = pd.Series(ryan_rating)

     marie_rating = {'concorde': 3,
                     'maud gone': 2,
                     'i can change': 4,
                     'helplessness blues': 5,
                     'never there': 2}
     marie_series = pd.Series(marie_rating)

     dom_rating = {'concorde': 4,
                   'maud gone': 3, 'i can change': 4,
                   'helplessness blues': 5,
                   'never there': 5}
     dom_series = pd.Series(dom_rating)

     # dictionary

     ratings_dict = {'Ryan': ryan_series,
            'Marie': marie_series,
            'Dom': dom_series}
     #print(ratings_dict)

     # data frame

     ratings_table = pd.DataFrame(dict)
     ratings_table
```

```
[9]:                     Ryan  Marie  Dom
     concorde               2      3    4
     maud gone              3      2    3
     i can change           5      4    4
     helplessness blues     5      5    5
     never there            4      2    5
```

- Use the cell below to create a Pandas `DataFrame`. Each column of the DataFrame should be the name of an band/singer of your pick (it is fine to clip the title tothe first 4 words in the title). Each row of the data frame should be the name of one of your colleagues, use 3-4 names. Interview 3-4 of your colleagues and add the value in each cell representing the rating of the corresponding song that each colleague provides using a scale between 0-5, where 5 is an awesome song and 0 is a boring song.

[10]:
```
ratings_table.T
```

[10]:
```
        concorde  maud gone  i can change  helplessness blues  never there
Ryan           2          3             5                   5            4
Marie          3          2             4                   5            2
Dom            4          3             4                   5            5
```

1.0.4 Summary

We have learned about the library `pandas` and the three fundamental objects of the library: - Index - Series - DataFrames

These last one are the most used, versatile data representation objects and are most commonly used for data science projects.

Pandas DataFrames are 2-dimensional objects composed by collections of one-dimensional objects called Series. The one-dimensional objects in a DataFrame are `aligned` meaning that all entries of a series are matched, they are related, each row is indexed by the same index no matter what series.

Complete the following exercise.

To practice with pandas Series and DataFrames, build a new dataset that has 10 series (subject 1-10) representing measurements from the following measures of brain health (use a tidy format with columns as variables and rows as subjects). Guess appropriate values for neuronal activity, blood oxygenation, pulsation and cortical thickness.

`['neuronal activity', 'blood oxygenation', 'blood pulsation rate', 'cortical thickness']`

[14]:
```
# subjects

subject_01 = {'neuronal activity': 40,
              'blood oxygenation': 95,
              'blood pulsation rate': 110,
              'cortical thickness': 2.5}
subject01_series = pd.Series(subject_01)

subject_02 = {'neuronal activity': 35,
              'blood oxygenation': 90,
              'blood pulsation rate': 130,
              'cortical thickness': 2.0}
subject02_series = pd.Series(subject_02)

subject_03 = {'neuronal activity': 33,
              'blood oxygenation': 86,
              'blood pulsation rate': 94,
              'cortical thickness': 2.7}
subject03_series = pd.Series(subject_03)
```

```
subject_04 = {'neuronal activity': 32,
              'blood oxygenation': 85,
              'blood pulsation rate': 140,
              'cortical thickness': 1.9}
subject04_series = pd.Series(subject_04)

subject_05 = {'neuronal activity': 47,
              'blood oxygenation': 98,
              'blood pulsation rate': 115,
              'cortical thickness': 2.6}
subject05_series = pd.Series(subject_05)

subject_06 = {'neuronal activity': 31,
              'blood oxygenation': 89,
              'blood pulsation rate': 102,
              'cortical thickness': 2.3}
subject06_series = pd.Series(subject_06)

subject_07 = {'neuronal activity': 39,
              'blood oxygenation': 85,
              'blood pulsation rate': 88,
              'cortical thickness': 1.7}
subject07_series = pd.Series(subject_07)

subject_08 = {'neuronal activity': 40,
              'blood oxygenation': 91,
              'blood pulsation rate': 105,
              'cortical thickness': 2.6}
subject08_series = pd.Series(subject_08)

subject_09 = {'neuronal activity':44,
              'blood oxygenation': 92,
              'blood pulsation rate': 123,
              'cortical thickness': 2.8}
subject09_series = pd.Series(subject_09)

subject_10 = {'neuronal activity': 30,
              'blood oxygenation': 85,
              'blood pulsation rate': 150,
              'cortical thickness': 1.5}
subject10_series = pd.Series(subject_10)

#dictionary

dict = {'Subject 1': subject01_series, 'Subject 2': subject02_series, 'Subject␣
 ↪3': subject03_series,
        'Subject 4': subject04_series, 'Subject 5': subject05_series, 'Subject␣
↪6': subject06_series,
        'Subject 7': subject07_series, 'Subject 8': subject08_series, 'Subject␣
↪9': subject09_series,
        'Subject 10': subject10_series}

#df
dataset = pd.DataFrame(dict)
dataset.T
```

```
[14]:             neuronal activity  blood oxygenation  blood pulsation rate  \
      Subject 1                40.0               95.0                 110.0
      Subject 2                35.0               90.0                 130.0
      Subject 3                33.0               86.0                  94.0
      Subject 4                32.0               85.0                 140.0
      Subject 5                47.0               98.0                 115.0
      Subject 6                31.0               89.0                 102.0
      Subject 7                39.0               85.0                  88.0
      Subject 8                40.0               91.0                 105.0
      Subject 9                44.0               92.0                 123.0
      Subject 10               30.0               85.0                 150.0

                  cortical thickness
      Subject 1                  2.5
      Subject 2                  2.0
      Subject 3                  2.7
      Subject 4                  1.9
      Subject 5                  2.6
      Subject 6                  2.3
      Subject 7                  1.7
      Subject 8                  2.6
      Subject 9                  2.8
      Subject 10                 1.5
```