01_basic_pandas

August 13, 2021

1 Pandas Fundamentals

Use the Pandas library to do statistics on tabular data.

Pandas is a an open source library providing high-performance, easy-to-use data structures and data analysis tools. Pandas is particularly suited to the analysis of tabular data, i.e. data that can can go into a table.

1.0.1 In other words, if you can imagine the data in an Excel spreadsheet, then Pandas is the tool for the job.

Pandas DataFrames are 2-dimensional tables whose columns have names and potentially have different data types.

Pandas Dataframes are pretty much like excel spreadsheets! **and excel spreadsheets are pretty much like matricies** - make Nick draw this on the board and talk throughs some examples

1.0.2 installing `pandas` with (Ana)conda (if needed…)

to get `pandas` (you only need to do this the first time we go throug this): 1. go to the little `+` on the left to open a `launcher` window. 1. click the 'terminal' tile to open a terminal 1. type `conda env list` and hit enter. Make sure that there is a `*` next to the line that says swbc 1. if that's right type `conda install pandas` 1. when it asks `proceed ([y]/n)?` type `y` and hit enter

1.1 Credit:

this comes from Abernathys open book, which we will be looking at a lot! https://earth-env-data-science.github.io/lectures/core_python/python_fundamentals.html

[1]:
```
# import numpy, matplotlib.pyplot, pandas

import pandas as pd
import numpy as np
from matplotlib import pyplot as plt
%matplotlib inline
```

1.2 Pandas Data Structures: Series

We've seen several data structures so far: `lists dictionaries`, `arrays`. The Pandas labrary provides several data structures which are **super** useful.

The `Series` data structure represents a one-dimensional array of data. The main difference between a `Series` and numpy `array` is that a Series has an *index*. The index contains the labels that we use to access the data.

There are many ways to create a Series. We will just show a few.

[2]:
```
names = ['Ryan', 'Chiara', 'Johnny']
values = [36, 37, 2.7]
ages = pd.Series(values, index=names)
ages
```

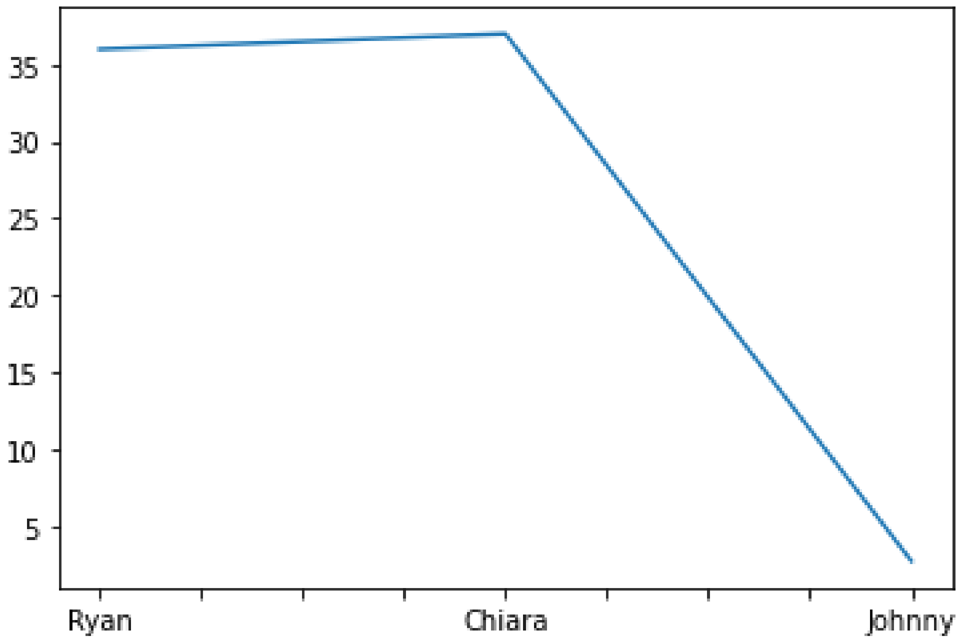
[2]:
```
Ryan      36.0
Chiara    37.0
Johnny     2.7
dtype: float64
```

If you think back to the `dictionary` you will see some similarities: namely labels (`keys` for `dict`, `index` for pandas `series`)

Series have built in plotting methods that will let you very quickly make some default plots

[3]:
```
# make a default plot
ages.plot()
```

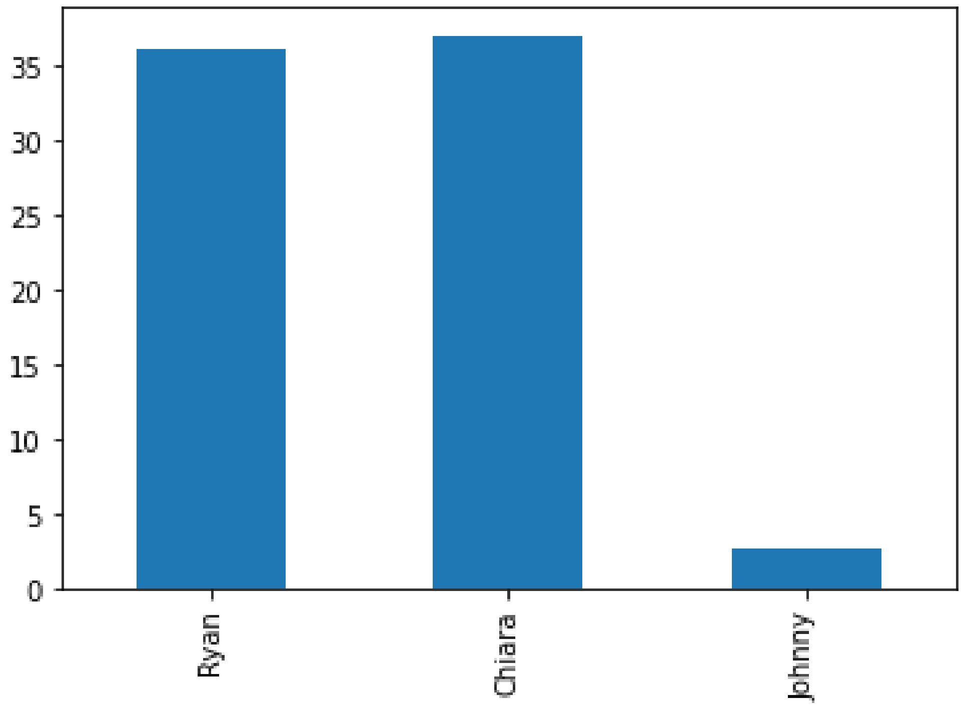
[3]:
```
<AxesSubplot:>
```



These default plots are configurable in many ways:

[4]:
```
# change to a bar plot
ages.plot(kind='bar')
```

[4]: <AxesSubplot:>



Arithmetic operations and most numpy function can be applied to Series. An important point is that the Series keep their index during such operations.

[5]:
```
np.log(ages) / ages**2
```

[5]:
```
Ryan      0.002765
Chiara    0.002638
Johnny    0.136249
dtype: float64
```

We can access the underlying `index` object if we need to by asking for the `.index` for a pandas series code object:

[6]:
```
ages.index
```

[6]:
```
Index(['Ryan', 'Chiara', 'Johnny'], dtype='object')
```

1.2.1 Indexing

We talked about indexing (or grabbing data from a position) before: for example we looked at the first patient data with something like `patient_0 = data[0,:]`. There we used a number to get the data position.

With pandas we can use the index to grab data. In this case the index is a bunch of names, so we can ask for the data from `Johnny` for example using the `.loc` attribute:

[7]:
```
ages.loc['Johnny']
```

[7]:
```
2.7
```

Or we can still use the number position `.iloc`

[8]:
```
ages.iloc[2]
```

[8]:
```
2.7
```

You can already maybe see part of why pandas is so great. What is easier to understand?

`ages.loc['Johnny']`

or

`ages.iloc[2]`

To me, being able to ask for the index using a label like `Johnny` makes the code much easier to understand, and makes my analysis more clear in my head.

We can pass a list or array to loc to get multiple rows back:

[9]:
```
ages.loc[['Ryan', 'Johnny']]
```

[9]:
```
Ryan      36.0
Johnny     2.7
dtype: float64
```

And we can even use slice notation

[10]:
```
ages.loc['Ryan':'Johnny']
```

[10]:
```
Ryan      36.0
Chiara    37.0
Johnny     2.7
dtype: float64
```

[11]:
```
ages.iloc[:2]
```

[11]:
```
Ryan      36.0
Chiara    37.0
dtype: float64
```

If we need to, we can always get the raw data back out as well

[12]:
```
ages.values # a numpy array
```

[12]:
```
array([36. , 37. ,  2.7])
```

[13]:
```
ages.index # a pandas Index object
```

[13]:
```
Index(['Ryan', 'Chiara', 'Johnny'], dtype='object')
```

1.3 Pandas Data Structures: DataFrame

There is a lot more to Series, but they are limited to a single "column". A more useful Pandas data structure is the `DataFrame`. A `DataFrame` is basically a **bunch of series that share the same index**. It's a lot like a table in a spreadsheet.

Below we create a DataFrame.

[14]:
```
# first we create a dictionary
data = {'age': [36, 37, 1.7],
        'height': [180, 155, 90],
        'weight': [78, np.nan, 11.3]}
df = pd.DataFrame(data, index=['Ryan', 'Chiara', 'Johnny'])
df
```

[14]:

	age	height	weight
Ryan	36.0	180	78.0
Chiara	37.0	155	NaN
Johnny	1.7	90	11.3

Pandas handles missing data very elegantly, keeping track of it through all calculations.

We can get some basic information about our `dataframe` data structure by using its `.info()` function:

[15]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 3 entries, Ryan to Johnny
Data columns (total 3 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   age     3 non-null      float64
 1   height  3 non-null      int64
 2   weight  2 non-null      float64
dtypes: float64(2), int64(1)
memory usage: 96.0+ bytes
```

A wide range of statistical functions are available on both Series and DataFrames.

[16]:
```
df.min()
```

[16]:
```
age        1.7
height    90.0
weight    11.3
dtype: float64
```

[17]:
```
df.mean()
```

[17]:
```
age        24.900000
height    141.666667
```

```
weight     44.650000
dtype: float64
```

[18]:
```
df.std()
```

[18]:
```
age       20.098010
height    46.457866
weight    47.164022
dtype: float64
```

[19]:
```
df.describe()
```

[19]:
```
            age      height     weight
count   3.00000    3.000000   2.000000
mean   24.90000  141.666667  44.650000
std    20.09801   46.457866  47.164022
min     1.70000   90.000000  11.300000
25%    18.85000  122.500000  27.975000
50%    36.00000  155.000000  44.650000
75%    36.50000  167.500000  61.325000
max    37.00000  180.000000  78.000000
```

We can get a single column as a Series using python's getitem syntax on the DataFrame object.

[20]:
```
df['height']
```

[20]:
```
Ryan      180
Chiara    155
Johnny     90
Name: height, dtype: int64
```

…or using attribute syntax.

[21]:
```
df.height
```

[21]:
```
Ryan      180
Chiara    155
Johnny     90
Name: height, dtype: int64
```

Indexing works very similar to series

[22]:
```
df.loc['Johnny']
```

[22]:
```
age        1.7
height    90.0
weight    11.3
Name: Johnny, dtype: float64
```

[23]:
```
df.iloc[2]
```

[23]:
```
age        1.7
height    90.0
weight    11.3
Name: Johnny, dtype: float64
```

But we can also specify the column we want to access

[24]:
```
df.loc['Johnny', 'age']
```

[24]:
```
1.7
```

[25]:
```
df.iloc[:2, 0]
```

[25]:
```
Ryan      36.0
Chiara    37.0
Name: age, dtype: float64
```

If we make a calculation using columns from the DataFrame, it will keep the same index:

[26]:
```
df.weight / df.height
```

[26]:
```
Ryan      0.433333
Chiara         NaN
Johnny    0.125556
dtype: float64
```

Which we can easily add as another column to the DataFrame:

[27]:
```
df['density'] = df.weight / df.height
df
```

[27]:

	age	height	weight	density
Ryan	36.0	180	78.0	0.433333
Chiara	37.0	155	NaN	NaN
Johnny	1.7	90	11.3	0.125556

1.4 Merging Data

Pandas supports a wide range of methods for merging different datasets. These are described extensively in the documentation. Here we just give a few examples.

[28]:
```
education = pd.Series(['BS', 'PhD', None, 'masters'],
                      index=['Ryan', 'Chiara', 'Johnny', 'Xiaomeng'],
                      name='education')
education
```

[28]:
```
Ryan              BS
Chiara           PhD
```

```
Johnny         None
Xiaomeng    masters
Name: education, dtype: object
```

We can add the data from the series `education` to our dataframe `df` using the function `.join()`, which will match overlapping indexes and add the new series as a column:

[29]:
```
# returns a new DataFrame
df.join(education)
```

[29]:
```
         age  height  weight   density education
Ryan    36.0     180    78.0  0.433333        BS
Chiara  37.0     155     NaN       NaN       PhD
Johnny   1.7      90    11.3  0.125556      None
```

[30]:
```
# returns a new DataFrame
df.join(education, how='right')
```

[30]:
```
           age  height  weight   density education
Ryan      36.0   180.0    78.0  0.433333        BS
Chiara    37.0   155.0     NaN       NaN       PhD
Johnny     1.7    90.0    11.3  0.125556      None
Xiaomeng   NaN     NaN     NaN       NaN   masters
```

We can also index using a boolean series. This is very useful

[31]:
```
adults = df[df.age > 18]
adults
```

[31]:
```
         age  height  weight   density
Ryan    36.0     180    78.0  0.433333
Chiara  37.0     155     NaN       NaN
```

[32]:
```
df['is_adult'] = df.age > 18
df
```

[32]:
```
         age  height  weight   density  is_adult
Ryan    36.0     180    78.0  0.433333      True
Chiara  37.0     155     NaN       NaN      True
Johnny   1.7      90    11.3  0.125556     False
```

1.4.1 Modifying Values

We often want to modify values in a dataframe based on some rule. To modify values, we need to use `.loc` or `.iloc`

[33]:
```
df.loc['Johnny', 'height'] = 95
df.loc['Ryan', 'weight'] += 1
df
```

[33]:

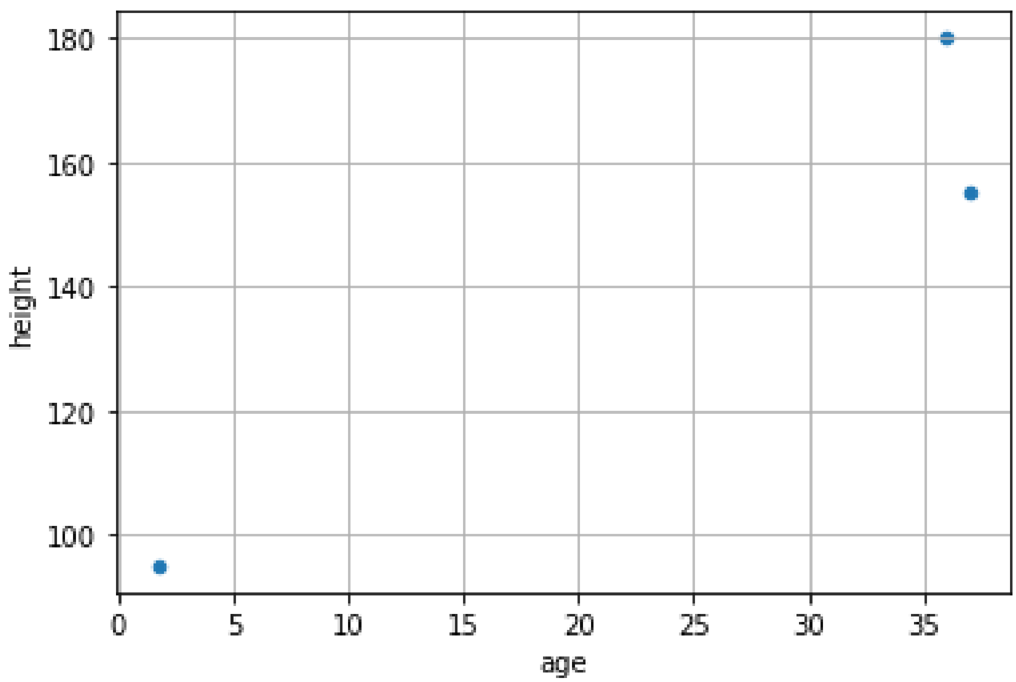
	age	height	weight	density	is_adult
Ryan	36.0	180	79.0	0.433333	True
Chiara	37.0	155	NaN	NaN	True
Johnny	1.7	95	11.3	0.125556	False

1.5 Plotting

DataFrames have all kinds of useful plotting built in.

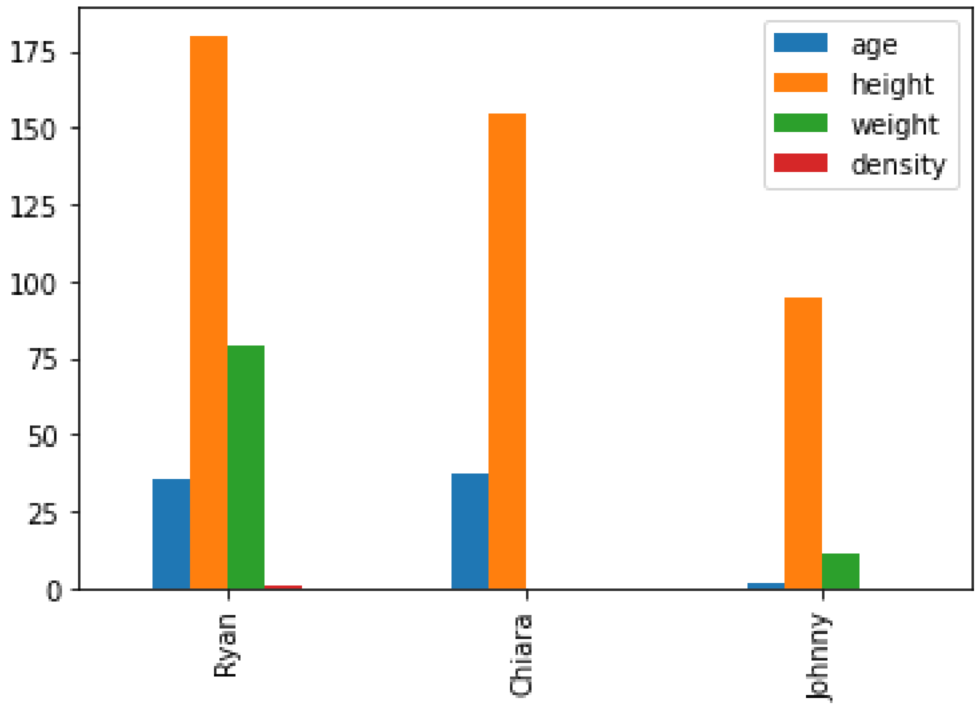
[34]:
```
df.plot(kind='scatter', x='age', y='height', grid=True)
```

[34]: <AxesSubplot:xlabel='age', ylabel='height'>



[35]:
```
df.plot(kind='bar')
```

[35]: <AxesSubplot:>



Later we will dig deeper into resampling, rolling means, and grouping operations (groupby).