First steps with pandas

Fabrice Rossi

Pandas is an important Python library that provides way to handle data sets in a data science workflow.

Examples in this series of exercises are based on the data sets available on the course web page. To read a data set into a data frame, use the following code

```
import pandas as pd
df = pd.read_csv('dataset.csv')
```

with an appropriate path to the file.

One of the main features of pandas is the `DataFrame` type. `df` in the code above is an object of this type.

Exercise 1 (*Getting along with DataFrame*)

Question 1 Using the interactive shell, load the `bank-short.csv` file into a `DataFrame` `bank`.

Question 2 Using the `shape` attribute of the object, get the size of the data set (number of observations and number of variables).

Question 3 Using the `columns` attribute of the object, get the names of the variables in the data set.

Question 4 Access to a column using the fact that a `DataFrame` is indexable with column's name as index.

Question 5 Use the `describe()` method of `DataFrame` to get a summary of the data frame. What are the limitations of this method, if any?

> **Lecture notes**
>
> In addition to `DataFrame` pandas defines two very important types, `Index` and `Series`. If `df` is a `DataFrame`, then `df.columns` and `df.index` are two objects of type `Index`. If `foo` is a variable name of `df`, then `df['foo']` is an object of type `Series`. In informal terms, a data frame is a collection of `Series` which all share the same `Index` (`df.index`). The collection itself is indexed by the `df.columns` `Index`.
>
> A `Series` is a sequence of values of the same type which is associated to an index. A `Series` is indexable and its content can be accessed via numerical indices, as any indexable object. In addition, it can be index by the indices from its index. The index itself is simply a sequence of objects than enable named accessed to the `Series`.
>
> Let us consider this simple example:
>
> ```
> # -*- coding: utf-8 -*-
> import pandas as pd
> ```

```
x = pd.Series(range(4), index=list('abcd'))
print(x)
print(x[1])
print(x['c'])
```

It prints

```
a    0
b    1
c    2
d    3
dtype: int64
1
2
```

As shown in this example, the characters specified in the index can be used to access to the values of the series exactly as when using integer indices.

When the index contains integers, accessing via `[]` might become ambiguous, therefore `[]` defaults to index values rather than to position in this situation, as shown in this program

```
# -*- coding: utf-8 -*-
import pandas as pd
x = pd.Series(range(4),index=[1, 3, 5, 7])
print(x[1])
print(x[3])
```

which prints

```
0
1
```

A non ambiguous alternative to `[]` is the pair `iloc[]` and `loc[]`. The first one `iloc[]` always uses position based indexing, while the second `loc[]` always uses index based one. The following program illustrates this:

```
# -*- coding: utf-8 -*-
import pandas as pd
x = pd.Series(range(4),index=[1, 3, 5, 7])
print(x)
print(x.loc[1])
print(x.iloc[1])
```

It prints

```
1    0
3    1
5    2
```

```
7    3
dtype: int64
0
1
```

Exercise 2 (*DataFrame row index*)

Question 1 Compare the index of the data frame loaded in the previous exercise (`bank.index`) which the index of a column of the data frame (e.g. `bank['age'].index`). Use in particular the `is` operator which enables deciding if two objects are actually the same object.

Question 2 How can the index of the data frame be interpreted from a data point of view?

Question 3 What non numerical index would make sense in a data set about persons?

Exercise 3 (*Series and Index*)

Question 1 Using a list comprehension, create a `Series`, `squares`, containing the squares of integers from 1 to 10, indexed by those integers.

Question 2 Set the name of the index with `squares.index.name='something'`. What is the effect of this operation on printing the `Series`?

Question 3 Create a new `Series`, `cubes`, containing the cubes of integers from 2 to 11, indexed by those integers.

Question 4 What is the result of `squares + cubes`?

Question 5 Extend `squares` and `cubes` so that they have the same index. This can be done using the fact that if `t` is a `Series` and `x` is a value that is not in the index of `t`, then `t[x]=y` will add `x` to the index and `y` to the `Series`.

Question 6 Test binary operations between `squares` and `cubes`, numerical as well as logical ones.

Question 7 Test statistical and aggregation methods for `Series`, such as `min`, `max`, `sum`, `mean`, `median`, etc.

Exercise 4 (*Filtering*)

A `Series` or a `DataFrame` can be filtered by means of a Boolean valued `Series`. In this exercise, load the `departements.csv` file into a `DataFrame` `dep`.

Question 1 Using a well chosen filtering technique, extract a sub data set from `dep` that contains only départements from Ile-de-France and print their names.

If `t` is a `Series`, `t.str` provides a string oriented interface. For instance

```
t.str.startswith('something')
```

returns a `Series` of boolean values stating whether the corresponding value is the original `Series` starts with `'something'` or not, as in this example

```
# -*- coding: utf-8 -*-
import pandas as pd
t = pd.Series(['toto','toti','tito','titi'])
print(t.str.startswith('to'))
```

which prints

```
0     True
1     True
2    False
3    False
dtype: bool
```

Question 2 Using the fact Corsican départements have codes that ends with A and B, print their names by filtering the `dep DataFrame` (using `endswith`).

The global pandas function `pd.to_numeric` can be used to convert strings to numerical values. This may be coupled with `str.isnumeric` to avoid errors. Another solution is to set the `errors` parameter of `pd.to_numeric` to `'coerce'` which produces `nan` (missing values) when the conversion fails.

Question 3 Compute a `Series codenum` which contains numerical versions of the codes that are numeric values.

New variables can be created using the `[]` operator (or `loc[]`). If `df` is a `DataFrame`, `df['foo']=...` creates or update variable `foo`.

Question 4 Add a variable `Num` to the `dep DataFrame` that contains those numerical values. What happens with Corsican départements?

Question 5 Using the fact DOM and TOM have codes larger than 900, select them and print their names.

Question 6 Create a new variable boolean `Métropole` which contains `True` is the département belongs to the Métropole and `False` if it is a DOM or a TOM.

Exercise 5 (*Creating and transforming variables*)

In this exercise, load the `bank-short.csv` file into a `DataFrame bank`.

Question 1 Convert the variables with values `'yes'` or `'no'` into Boolean variables.

Question 2 Add a new Boolean variable `hasloan` which `True` if the person as at least one loan (whether it is a housing loan or a personal loan).

The global pandas function `pd.cut` can be used to discretize continuous data into bins. A simple way to use it is as follows

```
pd.cut(t, [a, b, c, d])
```

where `[a, b, c, d]` are cutting points for the bins (the number of cuts is arbitrary). Here, pandas will create 3 bins, with intervals $]a;b]$, $]b;c]$, $]c;d]$ and transform numerical values into categorical values associated to the intervals. For instance, the following program

```
# -*- coding: utf-8 -*-
import pandas as pd
import random as rd
df = pd.DataFrame({'x':pd.Series([rd.random() for _ in range(10)])})
df['y'] = pd.cut(df['x'], [0, 0.1, 0.3, 0.5, 0.8, 1])
print(df)
```

might print

```
          x           y
0  0.272958  (0.1, 0.3]
1  0.441756  (0.3, 0.5]
2  0.900891    (0.8, 1]
3  0.959820    (0.8, 1]
4  0.405838  (0.3, 0.5]
5  0.223856  (0.1, 0.3]
6  0.933289    (0.8, 1]
7  0.806379    (0.8, 1]
8  0.057178    (0, 0.1]
9  0.218254  (0.1, 0.3]
```

Question 3 Use the `pd.cut` function to add to the `bank DataFrame` a new variable `agecat` with age categories corresponding to 5 year intervals (e.g. of the form $]20, 25]$).

The `groupby` method of `DataFrame` enables conditional analysis (a.k.a. multidimensional analysis). The standard use is as follows

```
df.groupby(conditioning)[variables].summary()
```

where `conditioning` is one or several conditioning variable(s), `variables` is the subset of non conditional variables to summarize and `summary` is the summary function (e.g. `mean`).

Exercise 6 (*Grouping and summarizing*)

In this exercise, load the `bank-short.csv` file into a `DataFrame bank`.

Question 1 Compute the conditional median of the age of participants given their marital status and their education level.

Question 2 What is the type of the result in the previous question? What is the type of the associated index?

Question 3 Compute the "distribution" of the answers to the marketing campaign (`y` variable) given the marital status and the education level, using the `size` summary method (which computes the size of each group).

Question 4 Call the `unstack` method on the result of the previous question. What is the type of the resulting object?

In addition to column oriented operations, pandas provide row oriented ones, as long as the `DataFrame` is homogeneous in type. For instance if a `DataFrame df` is numerical, `df.sum()` implements a column-wise sum while `df.sum(axis=1)` implements a row-wise sum.

Question 5 Using a row-wise operation, compute the total number of persons in each marital status and their education level group from the result of question 4 (the one obtained after calling `unstack`).

Question 6 Using a column-wise operation, compute the percentage of each answer type relatively to each marital status and their education level group. In other words, compute the conditional distribution of `y` variable given the marital status and the education level.