Combining datasets in pandas

Disclaimer: These notes and examples are an adaptation of the references listed at the end. They are compiled to fit the scope of this specific course.

Introduction

In most practical problems, combining two or more datasets is inevitable. As Dataframes contain tabular datasets, similar to databases, it is natural for `pandas` to offer capabilities for combining datasets. We have seen in the past that `pandas` often provide more than one way of combining Dataframes, which is one of the confusing points about combining datasets using `pandas`. The problem gets exacerbated knowing that most students have `SQL` or `R` programming experiences before joining this class. These languages use different function names to perform the same dataset-combining operations, making learning dataset-combining in `python` a bit more challenging.

This handout introduces `pandas`'s most common dataset joining capabilities with straightforward examples.

Overview

There are four broad ways of combining datasets in `pandas`:

- Using `concat`
- Using `merge` as a `pandas`'s function
- Using `merge` as a dataframe method

- Using `join` as a dataframe method

Bringing two pieces of a `Dataframe` together

Assume that you have two Dataframes with students names as follows:

Example

```
import pandas as pd
df_students_1 = pd.DataFrame({'student_ID':[101, 102, 103], 'first_name':['Donald',
'Joe','Linda']})
df_students_2 = pd.DataFrame({'student_ID':[104, 105, 106], 'first_name':['Milo',
'Daisy','Jay']})
df_students_all = pd.concat([df_students_1, df_students_2])
df_students_all
```

yields a Dataframe with rows from both `df_students_1` and `df_students_2` as

	student_ID	first_name
0	101	Donald
1	102	Joe
2	103	Linda
0	104	Milo
1	105	Daisy
2	106	Jay

`concat` is used for union of two or more dataframes. `panda`'s `concat` accepts a **list** of dataframes and combines them. `pd.concat` is often used when two dataframes have precisely the same columns, and we would like to bring all the rows together in one dataframe. A few important points:

- Notice that `concat` function preserves the indices from the original dataframes. This may create duplicate row indices and confuse the user later on. To resolve this issue, you can use `reset_index(drop = True)` with `concat` function. Alternatively, you can use `ignore_index = True` inside the `concat` function.

- `concat` function works by matching multiple dataframe's column names. If column names are not the same, the resulting dataframe will not be the ideal union of the original dataframes. Try the following snippet for yourself:

```
import pandas as pd
df_students_1 = pd.DataFrame({'STUDENT_ID':[101, 102, 103], 'first_name':['Donald',
'Joe','Linda']})
df_students_2 = pd.DataFrame({'student_ID':[104, 105, 106], 'first_name':['Milo',
'Daisy','Jay']})
df_students_all = pd.concat([df_students_1, df_students_2])
df_students_all
```

- If you use the `axis = 1` parameter with `concat` function, you will combine dataframes along their columns. Try the following example for yourself:

```
import pandas as pd
df_students_1 = pd.DataFrame({'STUDENT_ID':[101, 102, 103], 'first_name':['Donald',
'Joe','Linda']})
df_students_2 = pd.DataFrame({'student_ID':[104, 105, 106], 'first_name':['Milo',
'Daisy','Jay']})
df_students_all = pd.concat([df_students_1, df_students_2], axis = 1)
df_students_all
```

Note 1: To avoid having duplicate values joining two datasets, it is very common to use `.drop_duplicates()` method after using `concat` function.

Note 2: Once duplicate values are dropped, it usually is a good practice to reset the resulting dataframe index using `reset_index(drop=True)` method.

> **Note 3:** The resulting common syntax for the `concat` function, after appling **Note 1** and **Note 2**, would be
>
> ```
> import pandas as pd
> pd.concat([df_1,df_2]).drop_duplicates().reset_index(drop=True)
> ```

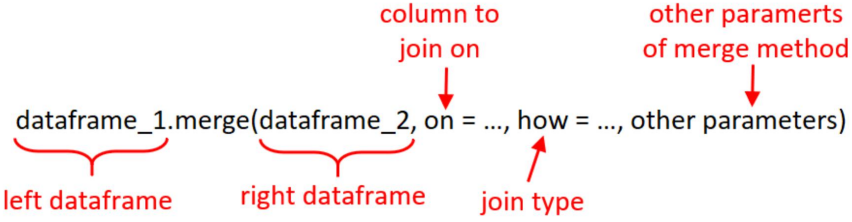
We will conclude this section with the following graphics illustrating `concat` function:

Dataframe Joins

So far, we have learned how to bring two pieces of one Dataframe together using `pd.concat` function. This section will teach how to combine multiple dataframes with different columns. Similar to databases, there are four common kinds of joins:

- `inner` join
- `outer` join
- `left` join
- `right` join

`pandas` offer `merge` function (in the form of `pd.merge`) and two dataframe methods of `join` and `merge` (in the forms of `df.join` and `df.merge`) for performing these joins. This handout will cover dataframe's `merge` method. This method offers ample flexibility in all practical situations. The general syntax of joins using `merge` method is as folows:



The critical decision you need to make in all of the joins is to decide on which column in both dataframes you like to join dataframes on.

The rest of this section explains different kinds of joins.

`Inner` Join

Consider the following dataframes:

> **Example**

```
import pandas as pd
df_students_names = pd.DataFrame({'student_ID':[101, 102, 103, 104], 'first_name':
['Donald', 'Joe','Linda', 'Milo']})
df_students_cars = pd.DataFrame({'student_ID':[101, 101, 102, 101, 200], 'car':
['Benz', 'BMW','Toyota', 'Bentley', 'Ford']})
```

First, we need to decide which column to join these two dataframes on. It is evident in this example that `students_ID` is the right column as it will allow us to match entries from the first dataframe with the entries from the second dataframe. `merge` uses the `on` parameter for communicating this column. Next, we need to determine the join type. `merge` 's default join type is `inner` ; however, it is an important practice to communicate the join type using the `how` parameter. The final snippet for the (inner) join of the above dataframes is

```
df_students_names.merge(df_students_cars, on = 'student_ID', how = 'inner')
```

yielding

	student_ID	first_name	car
0	101	Donald	Benz
1	101	Donald	BMW
2	101	Donald	Bentley
3	102	Joe	Toyota

Rows with values in the `on` column that appear on both dataframes are kept, and the rest are discarded. In this example, `student_ID` with values 101 and 102, which exist on both dataframes, are kept and merged; the other rows are discarded.

Note: `inner` is the default join for the `merge` method, so if the `how` parameter is omitted, the `inner` join will be provided.

Question: Why does running the following snippet produce an error?

```
import pandas as pd
df_students_names = pd.DataFrame({'student_ID':[101, 102, 103, 104], 'first_name':
['Donald', 'Joe','Linda', 'Milo']})
df_students_cars = pd.DataFrame({'Student_ID':[101, 101, 102, 101, 200], 'car':
['Benz', 'BMW','Toyota', 'Bentley', 'Ford']})
df_students_names.merge(df_students_cars, on = 'student_ID', how = 'inner')
```

Answer: Close examination of the snippet reveals that the `on` column is not the same in the above dataframes (`student_ID` vs `Student_ID`). Slight name differences is a common occurrence in practical datasets, and the following are two common remedies for solving this problem:

- **Solution 1 (Renaming column names):** An obvious solution would be to rename column names for one of the dataframes. Running the following snippet before the `merge` operation would address the issue.

```
df_students_cars.rename(columns = {'Student_ID': 'student_ID'}, inplace = True)
```

- **Solution 2: (use `left_on` and `right_on` parameters** Another approach would be to use `left_on` and `right_on` parameters. `left_on` would be the joining column name in the left dataframe and `right_on` would be the joining column name in the right dataframe. Here is an example:

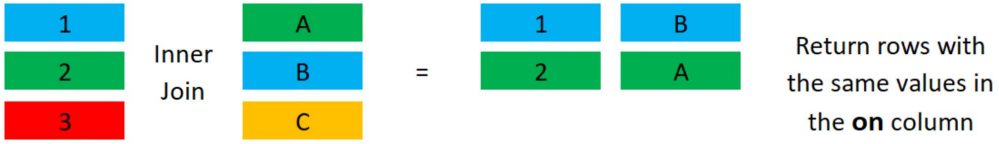
```
import pandas as pd
df_students_names = pd.DataFrame({'student_ID':[101, 102, 103, 104], 'first_name':
['Donnald', 'Joe','Linda', 'Milo']})
df_students_cars = pd.DataFrame({'Student_ID':[101, 101, 102, 101, 200], 'car':
['Benz', 'BMW','Toyota', 'Bentley', 'Ford']})
df_students_all = df_students_names.merge(df_students_cars, left_on = 'student_ID',
right_on = 'Student_ID', how = 'inner')
df_students_all
```

yields

	student_ID	first_name	Student_ID	car
0	101	Donald	101	Benz
1	101	Donald	101	BMW
2	101	Donald	101	Bentley
3	102	Joe	102	Toyota

`student_ID` and `Student_ID` columns are identical, and you can drop one of them using `df_student_all.drop(columns = ['Student_ID'], inplace = True)` .

The following graphics illustrate the `inner` join:



`outer` Join

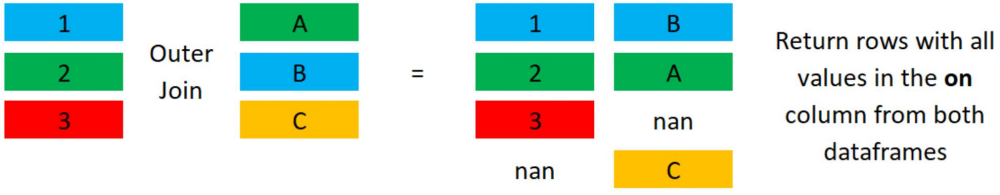
An `outer` join on a column includes all the values in that column from both dataframes. This join adds values from the other columns of both dataframes. If either dataframe has a value in the column that we join on that was absent from the other dataframe, the new columns are filled with `nan`. Let's see an example:

> **Example**
>
> ```
> import pandas as pd
> df_students_names = pd.DataFrame({'student_ID':[101, 102, 103, 104], 'first_name':
> ['Donald', 'Joe','Linda', 'Milo']})
> df_students_cars = pd.DataFrame({'student_ID':[101, 101, 102, 101, 200], 'car':
> ['Benz', 'BMW','Toyota', 'Bentley', 'Ford']})
> df_students_names.merge(df_students_cars, on = 'student_ID', how = 'outer')
> ```
>
> yields

	student_ID	first_name	car
0	101	Donald	Benz
1	101	Donald	BMW
2	101	Donald	Bentley
3	102	Joe	Toyota
4	103	Linda	nan
5	104	Milo	nan
6	200	nan	Ford

> The resulting dataframe has all `student_ID` from both dataframes; whenever a `student_ID` value is present in both dataframes, the `first_name` and `car` columns are completed by the values from either of the dataframes. When a `student_ID` value is not present in one of the dataframes, such as 103 missing from `df_students_names`, there is a `nan` value present in the corresponding column(s).
>
> **Note:** You can also use `left_on` and `right_on` with `outer` join.

The following graphics illustrate the `outer` join:



`left` Join

A `left` join keeps all joining column values in the left dataframe. Joining column values that are present only in the right dataframe are dropped. Here is an example:

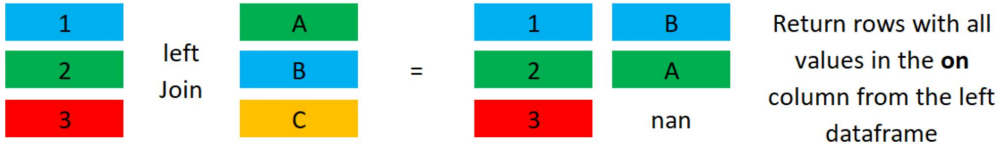
> **Example**
>
> ```
> import pandas as pd
> df_students_names = pd.DataFrame({'student_ID':[101, 102, 103, 104], 'first_name':
> ['Donald', 'Joe','Linda', 'Milo']})
> df_students_cars = pd.DataFrame({'student_ID':[101, 101, 102, 101, 200], 'car':
> ['Benz', 'BMW','Toyota', 'Bentley', 'Ford']})
> df_students_names.merge(df_students_cars, on = 'student_ID', how = 'left')
> ```
>
> yields
>
> | | student_ID | first_name | car |
> |---|---|---|---|
> | 0 | 101 | Donald | Benz |
> | 1 | 101 | Donald | BMW |

	student_ID	first_name	car
2	101	Donald	Bentley
3	102	Joe	Toyota
4	103	Linda	nan
5	104	Milo	nan

The resulting dataframe has all `student_ID` from the left dataframes with their matching values from the right dataframe; `student_ID` of 200 is present only in the right dataframe; hence, it is dropped. Notice that `student_ID` of 101 is repeated multiple times in the resulting dataframe because the row of 101 from the left dataframe is matched with all three 101 rows from the right dataframe.

Note: You can also use `left_on` and `right_on` with `left` join.

The following graphics illustrate the `left` join:



`right` Join

A `right` join keeps all joining column values in the right dataframe. Joining column values that are present only in the left dataframe are dropped. Here is an example:

Example

```
import pandas as pd
df_students_names = pd.DataFrame({'student_ID':[101, 102, 103, 104], 'first_name':
['Donald', 'Joe','Linda', 'Milo']})
df_students_cars = pd.DataFrame({'student_ID':[101, 101, 102, 101, 200], 'car':
['Benz', 'BMW','Toyota', 'Bentley', 'Ford']})
df_students_names.merge(df_students_cars, on = 'student_ID', how = 'right')
```

> yields
>
> | | student_ID | first_name | car |
> |---|---|---|---|
> | 0 | 101 | Donald | Benz |
> | 1 | 101 | Donald | BMW |
> | 2 | 102 | Joe | Toyota |
> | 3 | 101 | Donald | Bentley |
> | 4 | 200 | nan | Ford |
>
> The resulting dataframe has all `student_ID` from the right dataframes with their matching values from the left dataframe; `student_ID` of 103 and 104 are present only in the left dataframe; hence, they are dropped. Notice that `student_ID` of 200 is present in the right dataframe and is absent in the left dataframe. As a result, `first_name` is missing for the row of 200 in the resulting dataframe.
>
> **Note:** You can also use `left_on` and `right_on` with the `right` join.

As you can see, `left` and `right` joins are similar. By changing the position of the left and right dataframes, the left/right join will change to the right/left join.

The following graphics illustrate the `right` join:

`cross` Join

`cross` join is a less known/used kind of join. It combines every row of the left dataframe with every row of the right dataframe; hence, it doesn't need a joining column. This kind of join gets real big fast. Here is an example:

> **Example**

```
import pandas as pd
df_students_names = pd.DataFrame({'student_ID':[101, 102, 103, 104], 'first_name':
['Donald', 'Joe','Linda', 'Milo']})
df_students_cars = pd.DataFrame({'student_ID':[101, 101, 102, 101, 200], 'car':
['Benz', 'BMW','Toyota', 'Bentley', 'Ford']})
df_students_names.merge(df_students_cars, how = 'cross')
```

yields a dataframe with $4 \times 5 = 20$ rows.

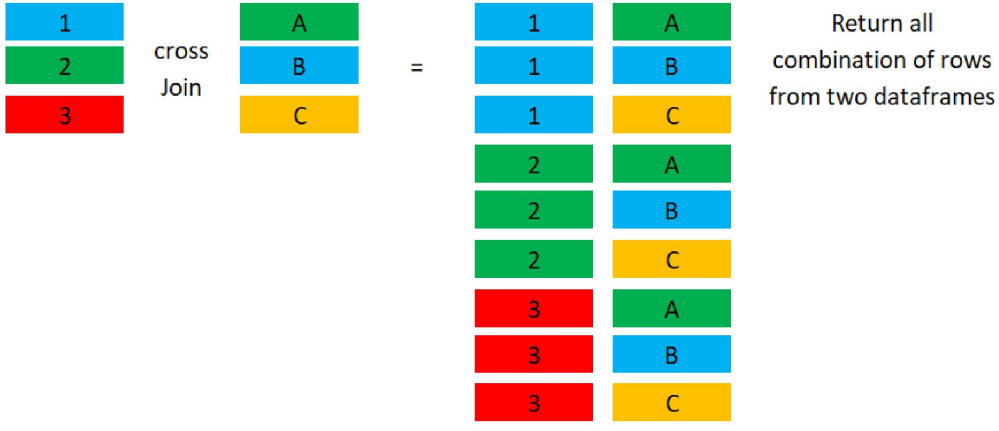
	student_ID_x	first_name	student_ID_y	car
0	101	Donald	101	Benz
1	101	Donald	101	BMW
2	101	Donald	102	Toyota
3	101	Donald	101	Bentley
4	101	Donald	200	Ford
5	102	Joe	101	Benz
6	102	Joe	101	BMW
7	102	Joe	102	Toyota
8	102	Joe	101	Bentley
9	102	Joe	200	Ford
10	103	Linda	101	Benz
11	103	Linda	101	BMW
12	103	Linda	102	Toyota
13	103	Linda	101	Bentley
14	103	Linda	200	Ford
15	104	Milo	101	Benz
16	104	Milo	101	BMW
17	104	Milo	102	Toyota
18	104	Milo	101	Bentley

	student_ID_x	first_name	student_ID_y	car
19	104	Milo	200	Ford

The resulting dataframe has the Cartesian Product of rows in both dataframes.

Note: `cross` join does not accept `on` , `left_on` , or `right_on` parameters.

The following graphics illustrate the `cross` join:



`merge` method has several other useful parameters. The next section will cover `indicator` and `validate` parameters.

`indicator` parameter

`indicator` parameter tells us where the data in the row come from. By including `indicator = True` inside the `merge` method, the resulting dataframe will have an additional column indicating whether the row comes from `left_only` dataframe, `right_only` dataframe, or `both` dataframes. Here is an example:

Example

```
import pandas as pd
df_students_names = pd.DataFrame({'student_ID':[101, 102, 103, 104], 'first_name':
['Donald', 'Joe','Linda', 'Milo']})
df_students_cars = pd.DataFrame({'student_ID':[101, 101, 102, 101, 200], 'car':
['Benz', 'BMW','Toyota', 'Bentley', 'Ford']})
df_students_names.merge(df_students_cars, on = 'student_ID', how = 'inner', indicator
= True)
```

yields

	student_ID	first_name	car	_merge
0	101	Donald	Benz	both
1	101	Donald	BMW	both
2	101	Donald	Bentley	both
3	102	Joe	Toyota	both

the `_merge` column indicates that each row comes from `both` dataframes, as it should be due to the `inner` join type.

Here is another example

Example

```
import pandas as pd
df_students_names = pd.DataFrame({'student_ID':[101, 102, 103, 104], 'first_name':
['Donald', 'Joe','Linda', 'Milo']})
df_students_cars = pd.DataFrame({'student_ID':[101, 101, 102, 101, 200], 'car':
['Benz', 'BMW','Toyota', 'Bentley', 'Ford']})
df_students_names.merge(df_students_cars, on = 'student_ID', how = 'outer', indicator
= True)
```

yields

	student_ID	first_name	car	_merge
0	101	Donald	Benz	both
1	101	Donald	BMW	both
2	101	Donald	Bentley	both

	student_ID	first_name	car	_merge
3	102	Joe	Toyota	both
4	103	Linda	nan	left_only
5	104	Milo	nan	left_only
6	200	nan	Ford	right_only

the `_merge` column indicates which row comes from the `both` dataframes, which one comes from the `left` dataframe, and which one comes from the `right` dataframe.

`validate` parameter

To understand the importance of `validate` parameter, let's revisit our earlier example.

```
df_students_names = pd.DataFrame({'student_ID':[101, 102, 103, 104], 'first_name':
['Donald', 'Joe','Linda', 'Milo']})
df_students_cars = pd.DataFrame({'student_ID':[101, 101, 102, 101, 200], 'car':
['Benz', 'BMW','Toyota', 'Bentley', 'Ford']})
```

In this example, `student_ID` column has unique values in the `df_students_names` dataframe; whereas, in the `df_students_cars` dataframe, the column of `student_ID` has duplicate values. It turns out this is an important observation in deciding the correct join type. In this section, we will learn the basics of `validate` parameter; the next section will guide us through practicing using this parameter.

`validate` parameter is one of the more recent features of `merge` method. This parameter ensures that your dataframes are merged the way you have assumed. `validate` parameter can take `1:1` (one on one), `1:m` (one on many), or `m:1` (many on one) values. Here is a description of `1:1`, `1:m`, and `m:1` values:

- `1:1`: by setting `validate = '1:1'`, the `merge` method ensures that values in the joining column are unique in both left and right dataframes; otherwise, it will raise an error.
- `1:m`: by setting `validate = '1:m'`, the `merge` method ensures that the values in the joining column of the left dataframe are unique (`1`); otherwise, it will raise an error.
- `m:1`: by setting `validate = 'm:1'`, the `merge` method ensures that the values in the joining column of the right dataframe are unique (`1`); otherwise, it will raise an error.

Note: `validate` parameter can also take `m:m` (many on many) value; but `m:m` value is less practical, more complicated, and will not be covered in this handout.

Let's practice `validate` parameter with a few examples:

> **Example 1:** In this example, you are looking for the average price of cars in inventory. Here is the data:
>
> ```
> import pandas as pd
> df_car_inventory = pd.DataFrame({'car_model':['BMW_X3', 'BMW_X7', 'Benz_GLE'],
> 'inventory':[20, 15, 4]})
> df_car_price = pd.DataFrame({'car_model':['BMW_X3', 'BMW_X7', 'Benz_GLE',
> 'Bentley_Bentayga'], 'price':[50, 78, 82, 185]})
> ```
>
> It looks like we do not have a positive inventory of every type of car. We first join these two datasets on `car_model` column. The right kind of join is `left` (we can use `inner` join as well). Additionally, we can use `validate = '1:1'` as, in this example, each car model has a unique `car_model` value in either of the dataframes. The `merge` snippet will be:
>
> ```
> df_car_info = df_car_inventory.merge(df_car_price, on = 'car_model', how = 'left',
> validate = '1:1')
> df_car_info
> ```
>
> yields
>
> | | car_model | inventory | price |
> |---|---|---|---|
> | 0 | BMW_X3 | 20 | 50 |
> | 1 | BMW_X7 | 15 | 78 |
> | 2 | Benz_GLE | 4 | 82 |
>
> Finally, the snippet of `df_car_info.price.mean()` will provide the average price of cars with positive inventory levels.
>
> **Note:** Enforcing `validate = '1:1'` does not raise an error; hence, the merge went as planned.
>
> **Example 2:** In this example, you are interested in the average tier of students in a business school. Here is the data:
>
> ```
> import pandas as pd
> df_students_list = pd.DataFrame({'student_name':['Jenny', 'Bella', 'Ray', 'Naser',
> 'Milo', 'Ezra', 'Noah'], 'college':['Borol', 'UPenn', 'Cornell', 'Cornell', 'Drool',
> 'UPenn', 'UCSD']})
> df_college_tier = pd.DataFrame({'college':['Penn_state', 'Indiana', 'Stanford', 'JHU',
> ```

```
    'Cornell', 'Drool', 'Borol', 'UCSB', 'UCSD', 'UCLA', 'NYU', 'UPenn'], 'tier':[1, 1, 1,
    1, 1, 4, 3, 1, 1, 1, 1, 1]})
```

It is expected to have multiple students from the same school. `how = left` and `validate = 'm:1'` are the right settings for this merge. The following snippet will answer the question:

```
df_student_college_tier = df_students_list.merge(df_college_tier, on = 'college', how
= 'left', validate = 'm:1')
```

yields

	student_name	college	tier
0	Jenny	Borol	3
1	Bella	UPenn	1
2	Ray	Cornell	1
3	Naser	Cornell	1
4	Milo	Drool	4
5	Ezra	UPenn	1
6	Noah	UCSD	1

Finally, the snippet of `df_student_college_tier.tier.mean()` will provide the answer.

Note: As a practice, try `validate = '1:m'` in the above snippet and see what happens!

Problems in the next section will help you practice selecting the correct type of joins.

Final comments on joining `dataframes`

As you can imagine, there are so much more details associated with joining two dataframes. This handout has covered the basics. Please note the following:

Note 1: One can join dataframes on multiple columns by listing the joining columns in a list (`on = [col_1, col_2]`). Please note that this operation can take a long time for larger dataframes.

Note 2: Creation of missing values due to joining dataframes is inevitable. The next step after joining dataframes usually involves cleaning the dataset and handling the generated `NaN` values. There is no general formula or guideline for handling these values; sometimes, replacing these `NaN` values with `0` is the correct course of action; sometimes dropping them is the proper action, and yet other times, leaving them as is, is the right thing to do. We will use the following practice problems to learn joining datasets and handling the generated `NaN` values.

Practice problems

1- Using `merge`, create a dataframe containing all possible combinations of these clothing pieces and their colors.

- `cloths = {'jeans', 'shirt', 'sneakers', 'jackets'}`
- `colors = {'blue', 'red', 'yellow'}`

The final output should look like

	cloths	color
0	shirt	yellow
1	shirt	blue
2	shirt	red
3	jeans	yellow
4	jeans	blue
5	jeans	red

2- This question has multiple parts. First, download the following three files on your local hard drive:

- `dataset 1: shorturl.at/gyKY1`
- `dataset 2: shorturl.at/nI137`
- `dataset 3: shorturl.at/eEGOW`

Now answer the following questions:

2-1- What fraction of students who spent time for interview preparation are male?

2-2- What is the maximum number of interviews for students who have spent some time preparing for the interview?

2-3- What is the average age of students who spent some time on job interview preparation?

2-4- Is there any correlation between the number of interviews and the amount of time spent preparing for interviews?

2-5- What is the average age of students who spent at least 80 hours on interview preparation?

2-6- What fraction of students who have received at least 15 interviews live in DC?

2-7- What fraction of female students' schools are located in the state of Maryland?

References

This document is an adaption from the following references:

1- *Python online documentation*; available at https://docs.python.org/3/

2- *A Byte of Python*; available at https://Python.swaroopch.com/

3- *Introduction to programming in Python*; available at https://introcs.cs.princeton.edu/Python/home

4- *Introduction to Computation and Programming Using Python: With Application to Understanding Data*, John Guttag, The MIT Press, 2nd edition, 2016

5- *Introduction to Python for Computer Science and Data Science*, Paul J. Deitel, Harvey Deitel, Pearson, 1st edition, 2019

6- *Data Mining for Business Analytics*, Galit Shmueli, Peter C. Bruce, Peter Gedeck, Nitin R. Patel, Wiley, 2020

7- *Python for Data Analysis, Data Wrangling with Pandas, NumPy, and IPython*; Wes McKinney, O'Reilly Media; 2nd edition, 2017

8- *Python Data Science Handbook: Essential Tools for Working with Data*, Jake VanderPlas, O'Reilly Media; 1st edition, 2016

9- *Data Visualization in Python with Pandas and Matplotlib*, David Landup, Independently published, 2021

10- *Introduction to Programming Using Java*; available at http://math.hws.edu/javanotes/

11- *Python for Programmers with introductory AI case studies*, Paul Deitel, Harvey Deitel, Pearson, 1st edition, 2019

12- *Effective Pandas: Patterns for Data Manipulation (Treading on Python)*, Matt Harrison, Independently published, 2021

13- https://betterprogramming.pub/

14- https://www.learnpython.org/

15- https://realpython.com/