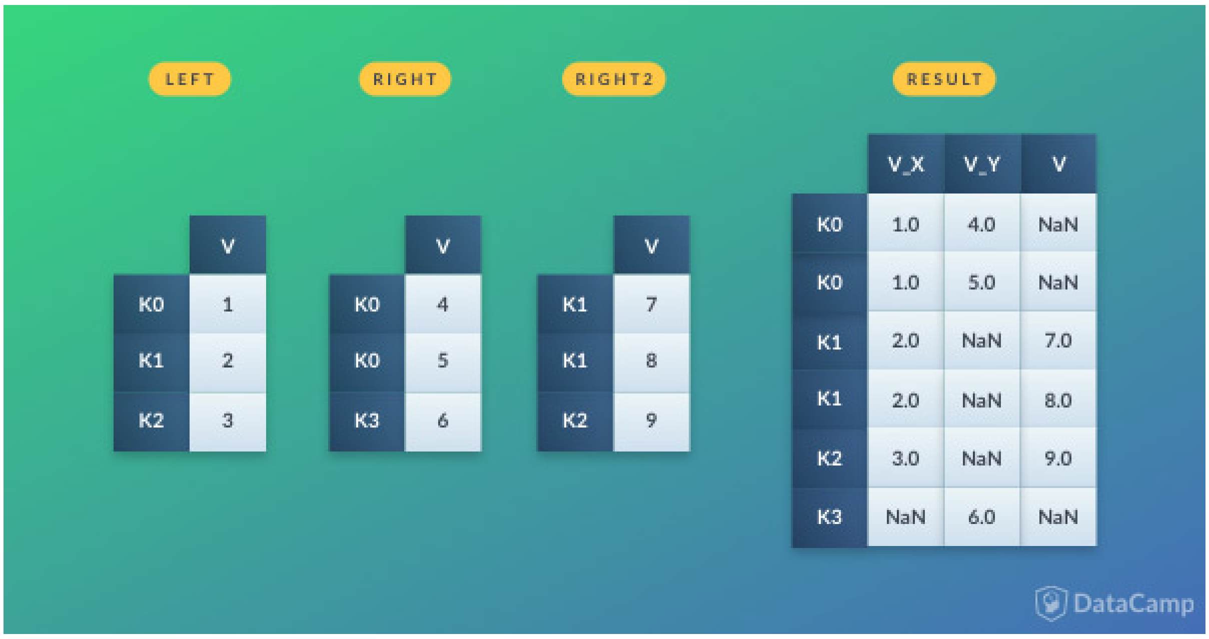
Karlijn Willems
January 14th, 2019

MUST READ PYTHON +3

Pandas Tutorial: DataFrames in Python

Explore data analysis with Python. Pandas DataFrames make manipulating your data easy, from selecting or replacing columns and indices to reshaping your data.



LEFT

	V
K0	1
K1	2
K2	3

RIGHT

	V
K0	4
K0	5
K3	6

RIGHT2

	V
K1	7
K1	8
K2	9

RESULT

	V_X	V_Y	V
K0	1.0	4.0	NaN
K0	1.0	5.0	NaN
K1	2.0	NaN	7.0
K1	2.0	NaN	8.0
K2	3.0	NaN	9.0
K3	NaN	6.0	NaN

Pandas is a popular Python package for data science, and with good reason: it offers powerful, expressive and flexible data structures that make data manipulation and analysis easy, among many other things. The DataFrame is one of these structures.

This tutorial covers Pandas DataFrames, from basic manipulations to advanced operations, by tackling 11 of the most popular questions so that you understand -and avoid- the doubts of the Pythonistas who have gone before you.

(For more practice, try the first chapter of this Pandas DataFrames course for free!)

What Are Pandas Data Frames?

Before you start, let's have a brief recap of what DataFrames are.

Those who are familiar with R know the data frame as a way to store data in rectangular grids that can easily be overviewed. Each row of these grids corresponds to measurements or values of an instance, while each column is a vector containing data for a specific variable. This means that a data frame's rows do not need to contain, but can contain, the same type of values: they can be numeric, character, logical, etc.

Now, DataFrames in Python are very similar: they come with the Pandas library, and they are defined as two-dimensional labeled data structures with columns of potentially different types.

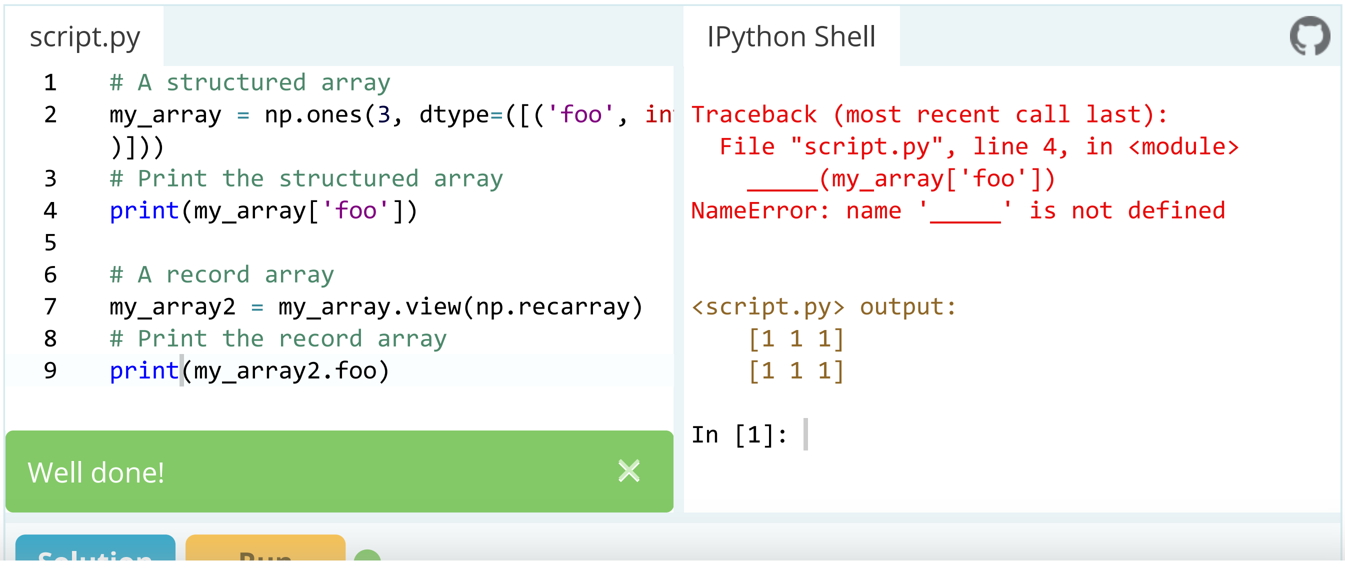
- a Pandas `DataFrame`
- a Pandas `Series`: a one-dimensional labeled array capable of holding any data type with axis labels or index. An example of a Series object is one column from a DataFrame.
- a NumPy `ndarray`, which can be a record or structured
- a two-dimensional `ndarray`
- dictionaries of one-dimensional `ndarray`'s, lists, dictionaries or Series.

Note the difference between `np.ndarray` and `np.array()` . The former is an actual data type, while the latter is a function to make arrays from other data structures.

Structured arrays allow users to manipulate the data by named fields: in the example below, a structured array of three tuples is created. The first element of each tuple will be called `foo` and will be of type `int`, while the second element will be named `bar` and will be a float.

Record arrays, on the other hand, expand the properties of structured arrays. They allow users to access fields of structured arrays by attribute rather than by index. You see below that the `foo` values are accessed in the `r2` record array.

An example:



```python
# A structured array
my_array = np.ones(3, dtype=([('foo', in
)]))
# Print the structured array
print(my_array['foo'])

# A record array
my_array2 = my_array.view(np.recarray)
# Print the record array
print(my_array2.foo)
```

```
Traceback (most recent call last):
  File "script.py", line 4, in <module>
    _____(my_array['foo'])
NameError: name '_____' is not defined


<script.py> output:
    [1 1 1]
    [1 1 1]

In [1]:
```

Want to leave a comment?

indicate the difference in columns. You will see later that these two components of the DataFrame will come in handy when you're manipulating your data.

If you're still in doubt about Pandas DataFrames and how they differ from other data structures such as a NumPy array or a Series, you can watch the small presentation below:

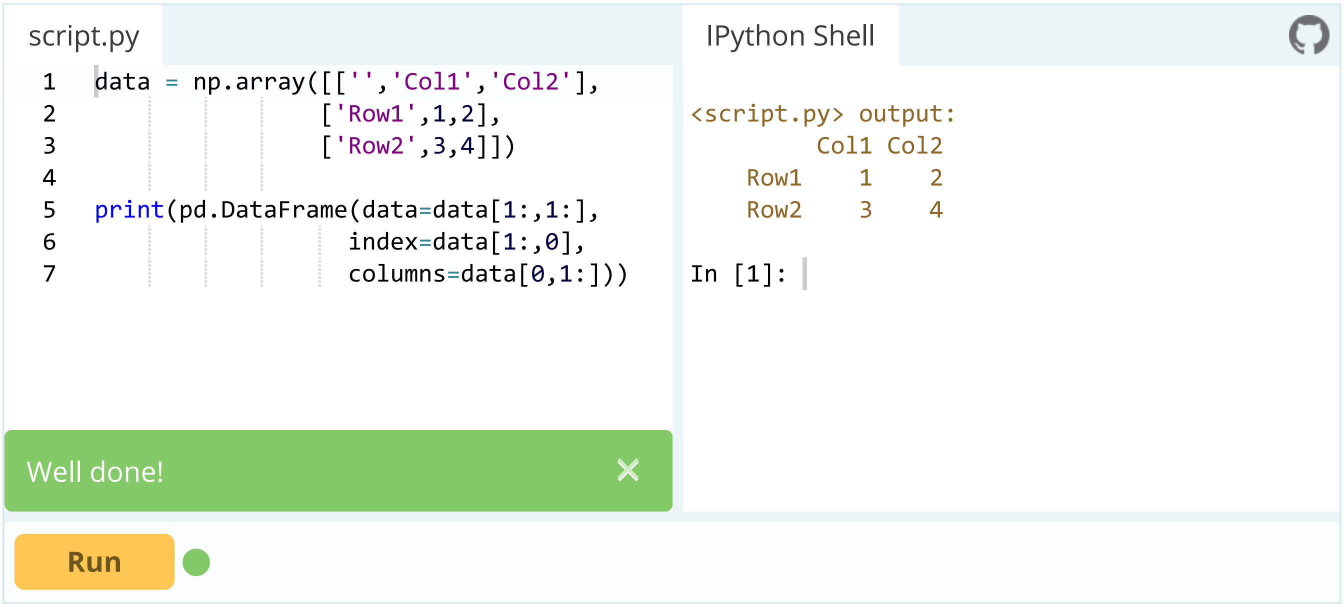
Note that in this post, most of the times, the libraries that you need have already been loaded in. The Pandas library is usually imported under the alias `pd`, while the NumPy library is loaded as `np`. **Remember** that when you code in your own data science environment, you shouldn't forget this import step, which you write just like this:

that users have about working with them!

1. How To Create a Pandas DataFrame

Obviously, making your DataFrames is your first step in almost anything that you want to do when it comes to data munging in Python. Sometimes, you will want to start from scratch, but you can also convert other data structures, such as lists or NumPy arrays, to Pandas DataFrames. In this section, you'll only cover the latter. However, if you want to read more on making empty DataFrames that you can fill up with data later, go to question 7.

Among the many things that can serve as input to make a 'DataFrame', a NumPy `ndarray` is one of them. To make a data frame from a NumPy array, you can just pass it to the `DataFrame()` function in the `data` argument.



```
data = np.array([['','Col1','Col2'],
                ['Row1',1,2],
                ['Row2',3,4]])

print(pd.DataFrame(data=data[1:,1:],
                  index=data[1:,0],
                  columns=data[0,1:]))
```

```
<script.py> output:
         Col1 Col2
    Row1    1    2
    Row2    3    4

In [1]:
```

Pay attention to how the code chunks above select elements from the NumPy array to construct the DataFrame: you first select the values that are contained in the lists that start with `Row1` and `Row2`, then you select the index or row numbers `Row1` and `Row2` and then the column names `Col1` and `Col2`.

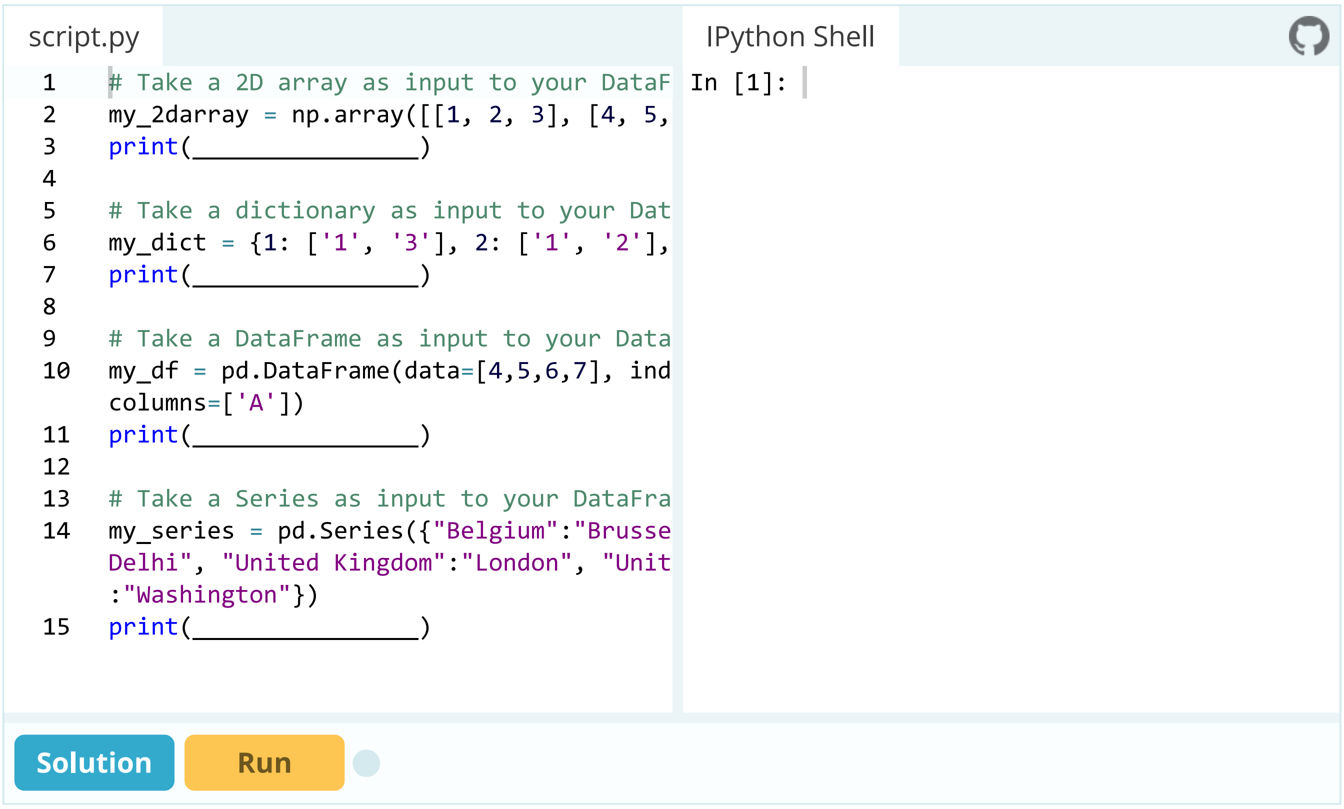
Want to leave a comment?

selecting `1`, `2`, `3` and `4`.

This approach to making DataFrames will be the same for all the structures that `DataFrame()` can take on as input.

Try it out in the code chunk below:

Remember that the Pandas library has already been imported for you as `pd`.



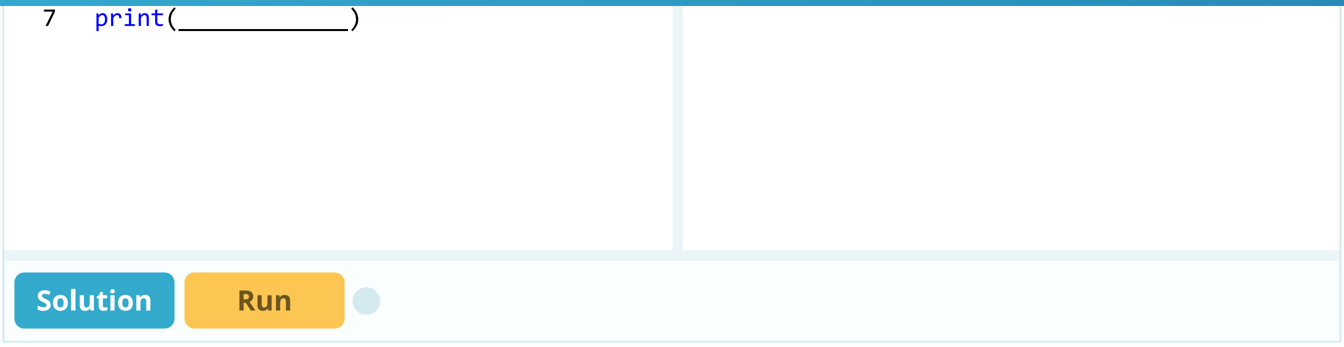
```python
# Take a 2D array as input to your DataF
my_2darray = np.array([[1, 2, 3], [4, 5,
print(________________)

# Take a dictionary as input to your Dat
my_dict = {1: ['1', '3'], 2: ['1', '2'],
print(________________)

# Take a DataFrame as input to your Data
my_df = pd.DataFrame(data=[4,5,6,7], ind
columns=['A'])
print(________________)

# Take a Series as input to your DataFra
my_series = pd.Series({"Belgium":"Brusse
Delhi", "United Kingdom":"London", "Unit
:"Washington"})
print(________________)
```

```
In [1]:
```

Note that the index of your Series (and DataFrame) contains the keys of the original dictionary, but that they are sorted: Belgium will be the index at 0, while the United States will be the index at 3.

After you have created your DataFrame, you might want to know a little bit more about it. You can use the `shape` property or the `len()` function in combination with the `.index` property:

Want to leave a comment?



```
print(___________)
```

These two options give you slightly different information on your DataFrame: the `shape` property will provide you with the dimensions of your DataFrame. That means that you will get to know the width and the height of your DataFrame. On the other hand, the `len()` function, in combination with the `index` property, will only give you information on the height of your DataFrame.

This all is totally not extraordinary, though, as you explicitly give in the `index` property.

You could also use `df[0].count()` to get to know more about the height of your DataFrame, but this will exclude the `NaN` values (if there are any). That is why calling `.count()` on your DataFrame is not always the better option.

If you want more information on your DataFrame columns, you can always execute `list(my_dataframe.columns.values)`. Try this out for yourself in the DataCamp Light block above!

Fundamental DataFrame Operations

Now that you have put your data in a more convenient Pandas DataFrame structure, it's time to get to the real work!

This first section will guide you through the first steps of working with DataFrames in Python. It will cover the basic operations that you can do on your newly created DataFrame: adding, selecting, deleting, renaming, ... You name it!

Want to leave a comment?

Even though you might still remember how to do it from the previous section: selecting an index, column or value from your DataFrame isn't that hard, quite the contrary. It's similar to what you see in other languages (or packages!) that are used for data analysis. If you aren't convinced, consider the following:

In R, you use the [,] notation to access the data frame's values.

Now, let's say you have a DataFrame like this one:

```
   A  B  C
0  1  2  3
1  4  5  6
2  7  8  9
```

And you want to access the value that is at index 0, in column 'A'.

Various options exist to get your value `1` back:

script.py

```python
# Using `iloc[]`
print(df.iloc[0][0])

# Using `loc[]`
print(df.loc[0]['A'])

# Using `at[]`
print(df.at[0,'A'])

# Using `iat[]`
print(df.iat[0,0])
```

IPython Shell

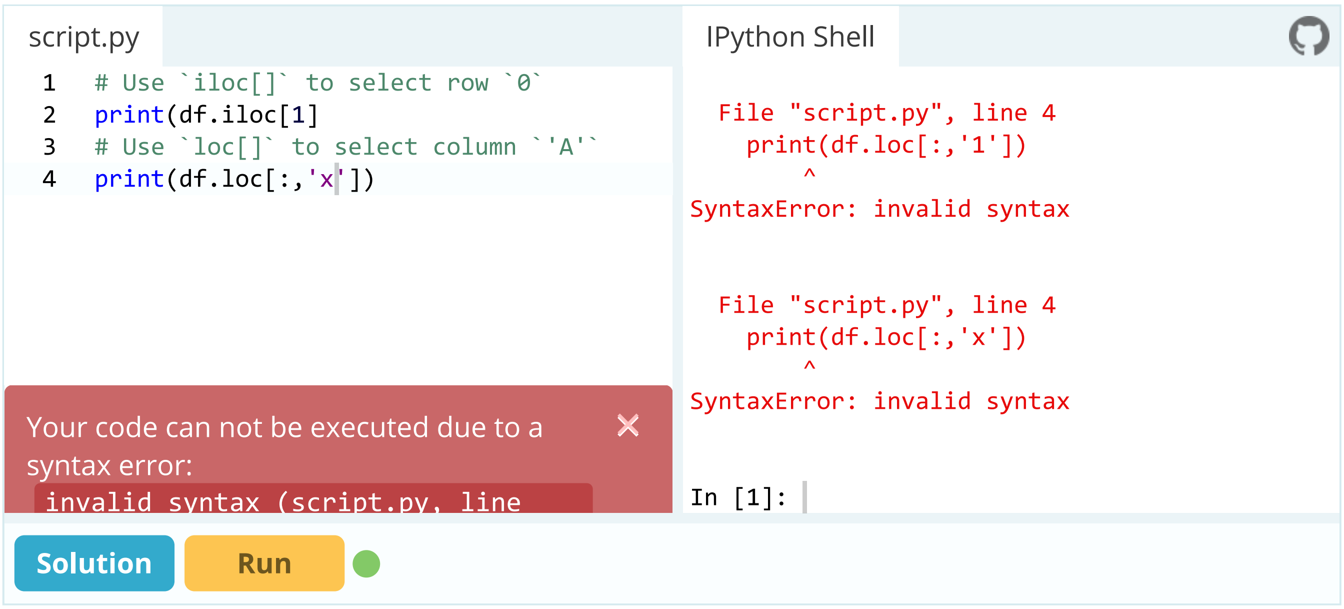
```
<script.py> output:
    1
    1
    1
    1

In [1]:
```

Great work!

Run

The most important ones to remember are, without a doubt, `.loc[]` and `.iloc[]`. The

Want to leave a comment?



```
# Use `iloc[]` to select row `0`
print(df.iloc[1]
# Use `loc[]` to select column `'A'`
print(df.loc[:,'x'])
```

```
  File "script.py", line 4
    print(df.loc[:,'1'])
        ^
SyntaxError: invalid syntax


  File "script.py", line 4
    print(df.loc[:,'x'])
        ^
SyntaxError: invalid syntax


In [1]:
```

Your code can not be executed due to a syntax error:
`invalid syntax (script.py, line`

For now, it's enough to know that you can either access the values by calling them by their label or by their position in the index or column. If you don't see this, look again at the slight differences in the commands: one time, you see `[0][0]`, the other time, you see `[0,'A']` to retrieve your value `1`.

3. How To Add an Index, Row or Column to a Pandas DataFrame

Now that you have learned how to select a value from a DataFrame, it's time to get to the real work and add an index, row or column to it!

Adding an Index to a DataFrame

When you create a DataFrame, you have the option to add input to the 'index' argument to make sure that you have the index that you desire. When you don't specify this, your DataFrame will have, by default, a numerically valued index that starts with 0 and continues until the last row of your DataFrame.

However, even when your index is specified for you automatically, you still have the power to re-use one of your columns and make it your index. You can easily do this by

Want to leave a comment?

```
# Set 'C' as the index of your DataFrame
df.set_index('C')
```

```
<script.py> output:
   A  B  C
0  1  2  3
1  4  5  6

<script.py> output:
   A  B  C
0  1  2  3
1  4  5  6

In [1]:
```

Well done!

Solution Run

Adding Rows to a DataFrame

Before you can get to the solution, it's first a good idea to grasp the concept of `loc` and how it differs from other indexing attributes such as `.iloc[]` and `.ix[]`:

- `.loc[]` works on labels of your index. This means that if you give in `loc[2]`, you look for the values of your DataFrame that have an index labeled `2`.
- `.iloc[]` works on the positions in your index. This means that if you give in `iloc[2]`, you look for the values of your DataFrame that are at index '2`.
- `.ix[]` is a more complex case: when the index is integer-based, you pass a label to `.ix[]`. `ix[2]` then means that you're looking in your DataFrame for values that have an index labeled `2`. This is just like `.loc[]`! However, if your index is not solely integer-based, `ix` will work with positions, just like `.iloc[]`.

This all might seem very complicated. Let's illustrate all of this with a small example:

script.py

```
df = pd.DataFrame(data=np.array([[1, 2,
, 8, 9]]), index= [2, 'A', 4], columns=[

# Pass `2` to `loc`
print(df.loc[2])

# Pass `2` to `iloc`
```

IPython Shell

```
<script.py> output:
   48    1
   49    2
   50    3
   Name: 2, dtype: int64
   48    7
```

Want to leave a comment?

Solution Run

Note that in this case, you used an example of a DataFrame that is not solely integer-based as to make it easier for you to understand the differences. You clearly see that passing `2` to `.loc[]` or `.iloc[]`/`.ix[]` does not give back the same result!

- You know that `.loc[]` will go and look at the values that are at label `2`. The result that you get back will be

```
48    1
49    2
50    3
```

- You also know that `.iloc[]` will go and look at the positions in the index. When you pass `2`, you will get back:

```
48    7
49    8
50    9
```

- Since the index doesn't only contain integers, `.ix[]` will have the same behavior as `iloc` and look at the positions in the index. You will get back the same result as `.iloc[]`.

Now that the difference between `.iloc[]`, `.loc[]` and `.ix[]` is clear, you are ready to give adding rows to your DataFrame a go!

Tip: as a consequence of what you have just read, you understand now also that the general recommendation is that you use `.loc` to insert rows in your DataFrame. That is because if you would use `df.ix[]`, you might try to reference a numerically valued index with the index value and accidentally overwrite an existing row of your DataFrame. You better avoid this!

Want to leave a comment?

DataCamp

```
, 8, 9]]), index= [2.5, 12.6, 4.8], colu

# There's no index labeled `2`, so you w
index at position `2`
df.ix[2] = [60, 50, 40]
print(df)

# This will make an index labeled `2` an
values
df.loc[2] = [11, 12, 13]
print(df)
```

Solution Run

You can see why all of this can be confusing, right?

Adding a Column to Your DataFrame

In some cases, you want to make your index part of your DataFrame. You can easily do this by taking a column from your DataFrame or by referring to a column that you haven't made yet and assigning it to the `.index` property, just like this:

script.py

```
df = pd.________(data=np.array([[1, 2, 3
8, 9]]), columns=['A', 'B', 'C'])

# Use `.index`
df['D'] = df.index

# Print `df`
print(__)
```

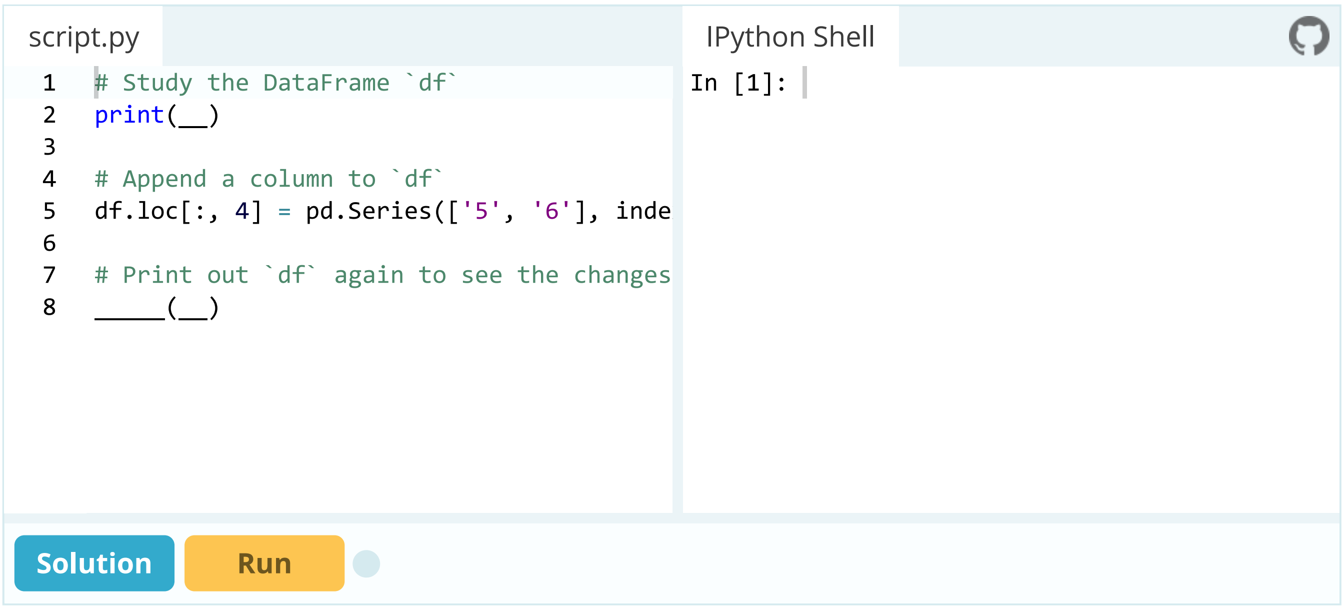
IPython Shell

```
In [1]:
```

Solution Run

In other words, you tell your DataFrame that it should take column `A` as its index.

Want to leave a comment?



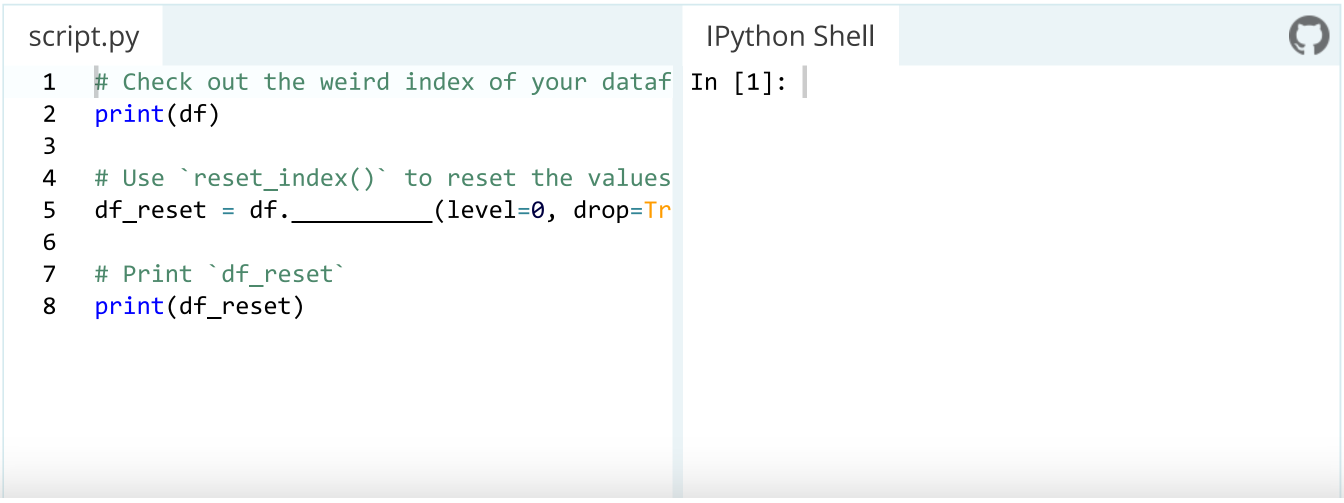
```
# Study the DataFrame `df`
print(__)

# Append a column to `df`
df.loc[:, 4] = pd.Series(['5', '6'], inde

# Print out `df` again to see the changes
_____(__)
```

```
In [1]:
```

Remember a Series object is much like a column of a DataFrame. That explains why you can easily add a Series to an existing DataFrame. Note also that the observation that was made earlier about `.loc[]` still stays valid, even when you're adding columns to your DataFrame!

Resetting the Index of Your DataFrame

When your index doesn't look entirely the way you want it to, you can opt to reset it. You can easily do this with `.reset_index()`. However, you should still watch out, as you can pass several arguments that can make or break the success of your reset:



```
# Check out the weird index of your dataf
print(df)

# Use `reset_index()` to reset the values
df_reset = df.__________(level=0, drop=Tr

# Print `df_reset`
print(df_reset)
```

```
In [1]:
```

Want to leave a comment?

Now try replacing the `drop` argument by `inplace` in the code chunk above and see what happens!

Note how you use the `drop` argument to indicate that you want to get rid of the index that was there. If you would have used `inplace`, the original index with floats is added as an extra column to your DataFrame.

4. How to Delete Indices, Rows or Columns From a Pandas Data Frame

Now that you have seen how to select and add indices, rows, and columns to your DataFrame, it's time to consider another use case: removing these three from your data structure.

Deleting an Index from Your DataFrame

If you want to remove the index from your DataFrame, you should reconsider because DataFrames and Series always have an index.

However, what you *can* do is, for example:

- resetting the index of your DataFrame (go back to the previous section to see how it is done) or
- remove the index name, if there is any, by executing `del df.index.name`,
- remove duplicate index values by resetting the index, dropping the duplicates of the index column that has been added to your DataFrame and reinstating that duplicateless column again as the index:

script.py IPython Shell

```
df = pd.DataFrame(data=np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [40, 50, 60], [23, 3
                  index= [2.5, 12.6, 4.8, 4.8, 2.5],
                  columns=[48, 49, 50])

df.___________.drop_duplicates(subset='index', keep='last').set_index('index')
```

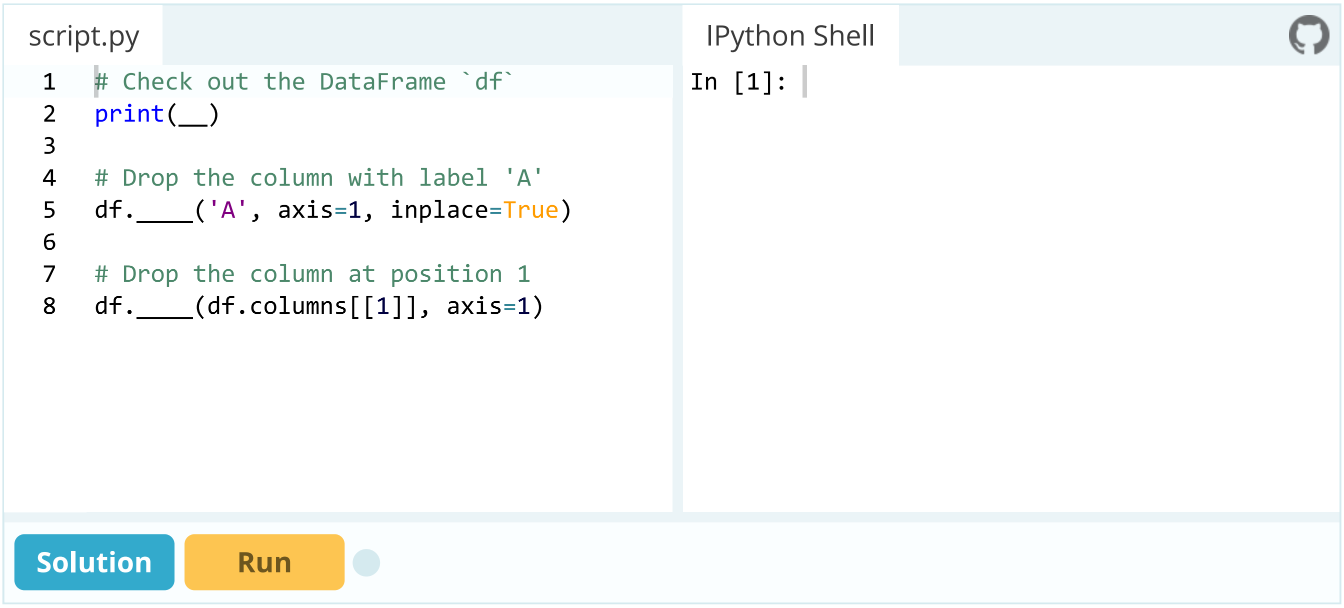
Want to leave a comment?

- and lastly, remove an index, and with it a row. This is elaborated further on in this tutorial.

Now that you know how to remove an index from your DataFrame, you can go on to removing columns and rows!

Deleting a Column from Your DataFrame

To get rid of (a selection of) columns from your DataFrame, you can use the `drop()` method:


script.py

```
# Check out the DataFrame `df`
print(__)

# Drop the column with label 'A'
df.____('A', axis=1, inplace=True)

# Drop the column at position 1
df.____(df.columns[[1]], axis=1)
```

IPython Shell

```
In [1]:
```

You might think now: well, this is not so straightforward; There are some extra arguments that are passed to the `drop()` method!

- The `axis` argument is either 0 when it indicates rows and 1 when it is used to drop columns.

Want to leave a comment?

Removing a Row from Your DataFrame

You can remove duplicate rows from your DataFrame by executing `df.drop_duplicates()`. You can also remove rows from your DataFrame, taking into account only the duplicate values that exist in one column.

Check out this example:

script.py

```
# Check out your DataFrame `df`
print(__)

# Drop the duplicates in `df`
df.________([48], keep='last')
```

IPython Shell

```
In [1]:
```

Solution Run

If there is no uniqueness criterion to the deletion that you want to perform, you can use the `drop()` method, where you use the `index` property to specify the index of which rows you want to remove from your DataFrame:

script.py

```
# Check out the DataFrame `df`
print(__)

# Drop the index at position 1
df.____(df.index[_])
```

IPython Shell

```
In [1]:
```

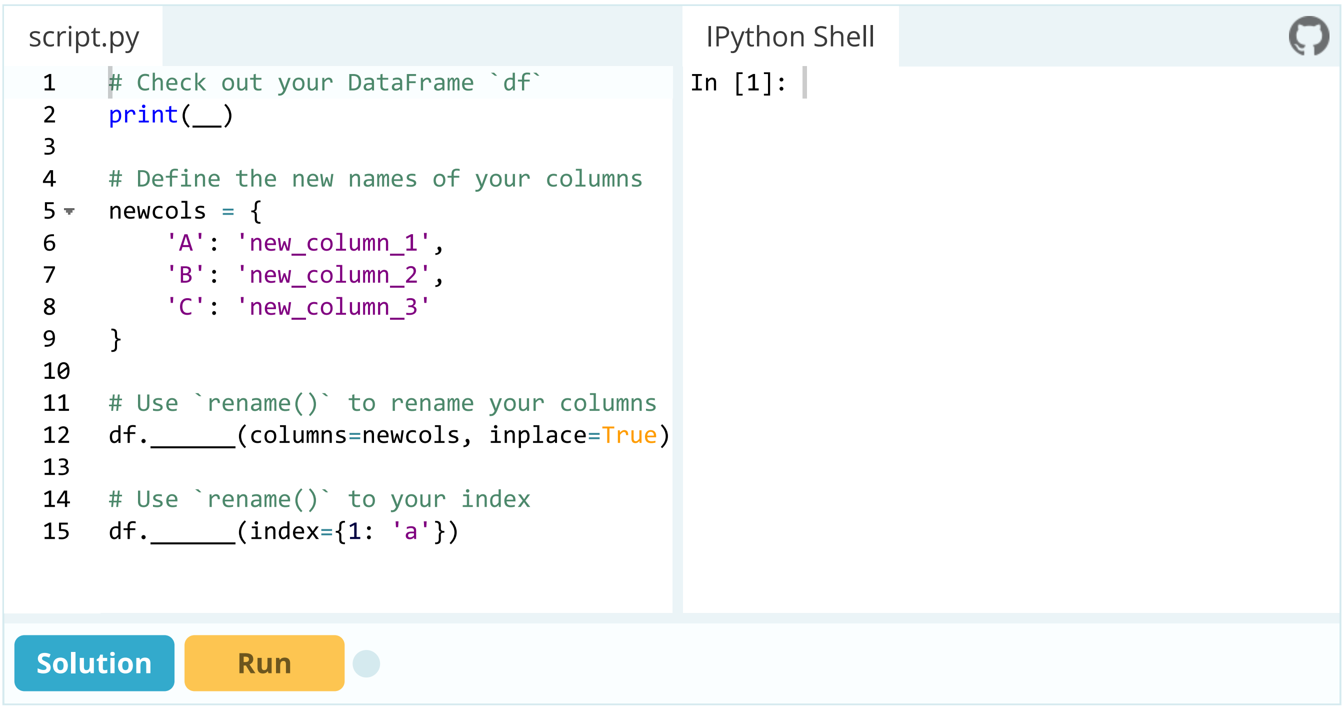
Want to leave a comment?

After this command, you might want to reset the index again.

Tip: try resetting the index of the resulting DataFrame for yourself! Don't forget to use the `drop` argument if you deem it necessary.

5. How to Rename the Index or Columns of a Pandas DataFrame

To give the columns or your index values of your dataframe a different value, it's best to use the `.rename()` method.



```python
# Check out your DataFrame `df`
print(__)

# Define the new names of your columns
newcols = {
    'A': 'new_column_1',
    'B': 'new_column_2',
    'C': 'new_column_3'
}

# Use `rename()` to rename your columns
df.______(columns=newcols, inplace=True)

# Use `rename()` to your index
df.______(index={1: 'a'})
```

```
In [1]:
```

Tip: try changing the `inplace` argument in the first task (renaming your columns) to `False` and see what the script now renders as a result. You see that now the DataFrame hasn't been reassigned when renaming the columns. As a result, the second task takes the original DataFrame as input and not the one that you just got back from the first `rename()` operation.

Beyond The Pandas DataFrame Basics

Want to leave a comment?

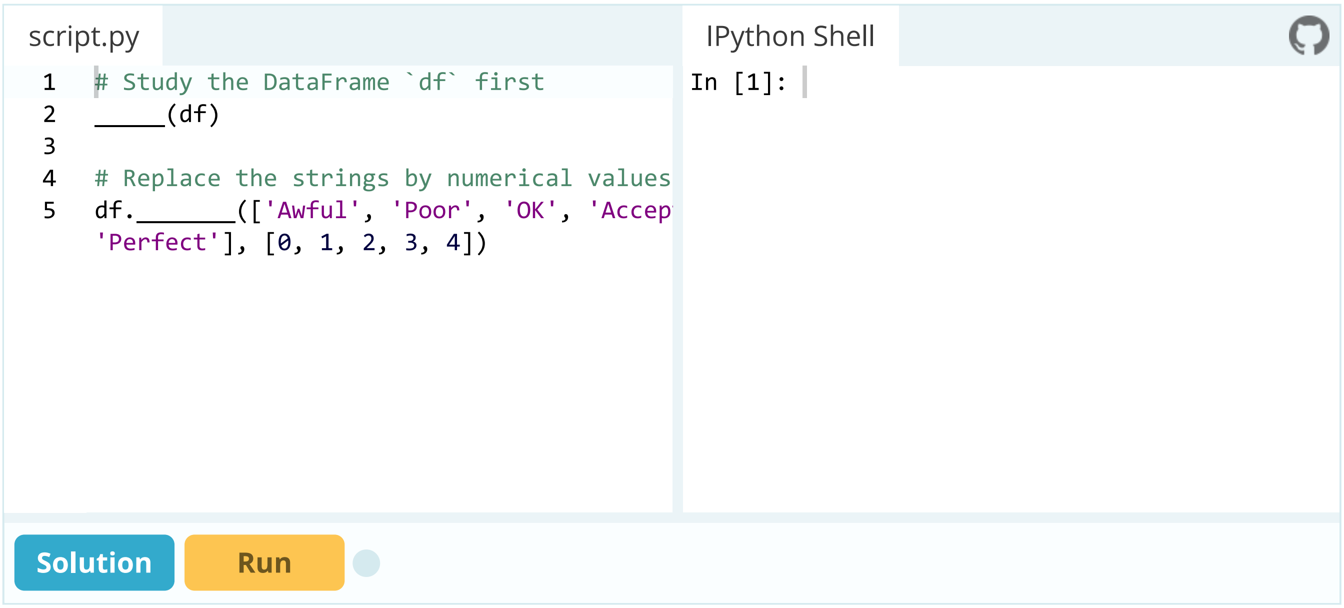
6. How To Format The Data in Your Pandas DataFrame

Most of the times, you will also want to be able to do some operations on the actual values that are contained within your DataFrame. In the following sections, you'll cover several ways in which you can format your DataFrame's values

Replacing All Occurrences of a String in a DataFrame

To replace certain strings in your DataFrame, you can easily use `replace()`: pass the values that you would like to change, followed by the values you want to replace them by.

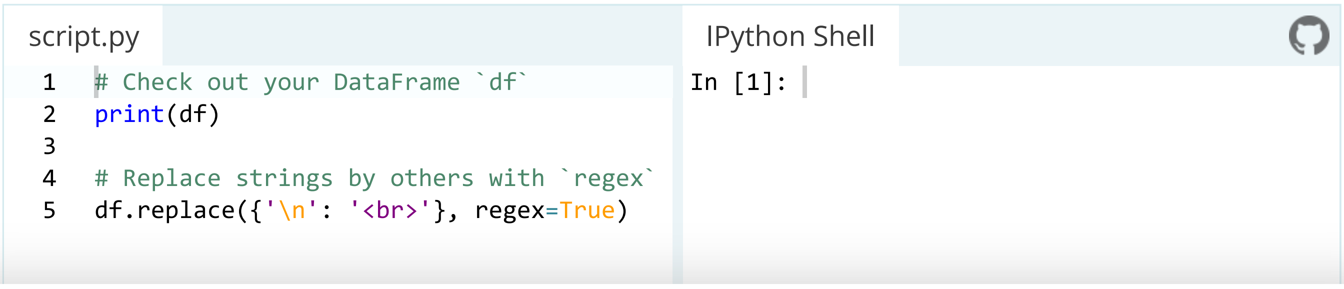
Just like this:



```
# Study the DataFrame `df` first
_____(df)

# Replace the strings by numerical values
df._______(['Awful', 'Poor', 'OK', 'Accep
'Perfect'], [0, 1, 2, 3, 4])
```

```
In [1]:
```

Note that there is also a `regex` argument that can help you out tremendously when you're faced with strange string combinations:



```
# Check out your DataFrame `df`
print(df)

# Replace strings by others with `regex`
df.replace({'\n': '<br>'}, regex=True)
```

```
In [1]:
```

Want to leave a comment?

Solution Run

In short, `replace()` is mostly what you need to deal with when you want to replace values or strings in your DataFrame by others!

Removing Parts From Strings in the Cells of Your DataFrame

Removing unwanted parts of strings is cumbersome work. Luckily, there is an easy solution to this problem!

script.py | IPython Shell

```python
# Check out your DataFrame
_____(df)

# Delete unwanted parts from the strings column
df['result'] = df['result'].map(lambda x: ).rstrip('aAbBcC'))

# Check out the result again
df
```

```
In [1]:
```

Solution Run

You use `map()` on the column `result` to apply the lambda function over each element or element-wise of the column. The function in itself takes the string value and strips the `+` or `-` that's located on the left, and also strips away any of the six `aAbBcC` on the right.

Splitting Text in a Column into Multiple Rows in a DataFrame

This is somewhat a more difficult formatting task. However, the next code chunk will walk you through the steps:

script.py | IPython Shell

```python
# Inspect your DataFrame `df`
```

```
In [1]:
```

Want to leave a comment?

```
# Get rid of the stack:
# Drop the level to line up with the Dat
ticket_series.index = ticket_series.inde

# Make your `ticket_series` a dataframe
ticketdf = pd.________(ticket_series)

# Delete the `Ticket` column from your D
del df['Ticket']

# Join the `ticketdf` DataFrame to `df`
df.____(ticketdef)

# Check out the new `df`
df
```

Solution Run

In short, what you do is:

- First, you inspect the DataFrame at hand. You see that the values in the last row and in the last column are a bit too long. It appears there are two tickets because a guest has taken a plus-one to the concert.
- You take the `Ticket` column from the DataFrame `df` and strings on a space. This will make sure that the two tickets will end up in two separate rows in the end. Next, you take these four values (the four ticket numbers) and put them into a Series object:

```
         0         1
0  23:44:55       NaN
1  66:77:88       NaN
2  43:68:05  56:34:12
```

  That still doesn't seem quite right. You have `NaN` values in there! You have to stack the Series to make sure you don't have any `NaN` values in the resulting Series.
- Next, you see that your Series is stacked.

```
0  0     23:44:55
```

Want to leave a comment?

That is not ideal either. That is why you drop the level to line up with the DataFrame:

```
0    23:44:55
1    66:77:88
2    43:68:05
2    56:34:12
dtype: object
```

That is what you're looking for.

- Transform your Series to a DataFrame to make sure you can join it back to your initial DataFrame. However, to avoid having any duplicates in your DataFrame, you can delete the original `Ticket` column.

Applying A Function to Your Pandas DataFrame's Columns or Rows

You might want to adjust the data in your DataFrame by applying a function to it. Let's begin answering this question by making your own lambda function:

```
doubler = lambda x: x*2
```

Tip: if you want to know more about functions in Python, consider taking this Python functions tutorial.

script.py

```
# Study the `df` DataFrame
_____(__)

# Apply the `doubler` function to the `A`
df['A'].apply(doubler)
```

IPython Shell

```
In [1]:
```

Want to leave a comment?

Note that you can also select the row of your DataFrame and apply the `doubler` lambda function to it. Remember that you can easily select a row from your DataFrame by using `.loc[]` or `.iloc[]`.

Then, you would execute something like this, depending on whether you want to select your index based on its position or based on its label:

```
df.loc[0].apply(doubler)
```

Note that the `apply()` function only applies the `doubler` function along the axis of your DataFrame. That means that you target either the index or the columns. Or, in other words, either a row or a column.

However, if you want to apply it to each element or element-wise, you can make use of the `map()` function. You can just replace the `apply()` function in the code chunk above with `map()`. Don't forget to still pass the `doubler` function to it to make sure you multiply the values by 2.

Let's say you want to apply this doubling function not only to the `A` column of your DataFrame but to the whole of it. In this case, you can use `applymap()` to apply the `doubler` function to every single element in the entire DataFrame:

script.py

```
doubled_df = df.applymap(doubler)
print(doubled_df)
```

IPython Shell

```
In [1]:
```

Run

Want to leave a comment?

example:

script.py

```python
def doubler(x):
    if x % 2 == 0:
        return x
    else:
        return x * 2

# Use `applymap()` to apply `doubler()`
doubled_df = df.applymap(doubler)

# Check the DataFrame
print(doubled_df)
```

IPython Shell

```
In [1]:
```

Run

If you want more information on the flow of control in Python, you can always read up on it here.

7. How To Create an Empty DataFrame

The function that you will use is the Pandas `Dataframe()` function: it requires you to pass the data that you want to put in, the indices and the columns.

Remember that the data that is contained within the data frame doesn't have to be homogenous. It can be of different data types!

There are several ways in which you can use this function to make an empty DataFrame. Firstly, you can use `numpy.nan` to initialize your data frame with `NaN`s. Note that `numpy.nan` has type `float`.

script.py

```python
df = pd.DataFrame(np.nan, index=[0,1,2,3]
print(df)
```

IPython Shell

```
In [1]:
```

Want to leave a comment?

DataCamp

Run

Right now, the data type of the data frame is inferred by default: because `numpy.nan` has type float, the data frame will also contain values of type float. You can, however, also force the DataFrame to be of a particular type by adding the attribute `dtype` and filling in the desired type. Just like in this example:

script.py

```
df = pd.DataFrame(index=range(0,4),column
='float')
print(df)
```

IPython Shell

```
In [1]:
```

Run

Note that if you don't specify the axis labels or index, they will be constructed from the input data based on common sense rules.

8. Does Pandas Recognize Dates When Importing Data?

Pandas can recognize it, but you need to help it a tiny bit: add the argument `parse_dates` when you'reading in data from, let's say, a comma-separated value (CSV) file:

```
import pandas as pd
pd.read_csv('yourFile', parse_dates=True)
```

Want to leave a comment?

There are, however, always weird date-time formats.

No worries! In such cases, you can construct your own parser to deal with this. You could, for example, make a lambda function that takes your DateTime and controls it with a format string.

```
import pandas as pd
dateparser = lambda x: pd.datetime.strptime(x, '%Y-%m-%d %H:%M:%S')

# Which makes your read command:
pd.read_csv(infile, parse_dates=['columnName'], date_parser=dateparse)

# Or combine two columns into a single DateTime column
pd.read_csv(infile, parse_dates={'datetime': ['date', 'time']}, date_parser=dateparse)
```

9. When, Why And How You Should Reshape Your Pandas DataFrame

Reshaping your DataFrame is transforming it so that the resulting structure makes it more suitable for your data analysis. In other words, reshaping is not so much concerned with formatting the values that are contained within the DataFrame, but more about transforming the shape of it.

This answers the when and why. But how would you reshape your DataFrame?

There are three ways of reshaping that frequently raise questions with users: pivoting, stacking and unstacking and melting.

Pivotting Your DataFrame

You can use the `pivot()` function to create a new derived table out of your original one. When you use the function, you can pass three arguments:

3. `index`: whatever you pass to this argument will become an index in your resulting table.

script.py | IPython Shell

```
# Import pandas
import ______ as pd

products = pd.DataFrame({'category': ['C
'Cleaning', 'Entertainment', 'Entertainm
'Tech'],
                        'store': ['Walma
'Walmart', 'Fnac', 'Dia','Walmart'],
                        'price':[11.42,
.95, 55.75, 111.55],
                        'testscore': [4,

# Use `pivot()` to pivot the DataFrame
pivot_products = products._____(index='c
='store', values='price')

# Check out the result
print(pivot_products)
```

```
In [1]:
```

Solution Run

When you don't specifically fill in what values you expect to be present in your resulting table, you will pivot by multiple columns:

script.py | IPython Shell

```
# Import the Pandas library
import ______ as pd

# Construct the DataFrame
products = pd.DataFrame({'category': ['C
'Cleaning', 'Entertainment', 'Entertainm
'Tech'],
                        'store': ['Walma
'Walmart', 'Fnac', 'Dia','Walmart'],
                        'price':[11.42,
.95, 55.75, 111.55],
                        'testscore': [4,

# Use `pivot()` to pivot your DataFrame
pivot_products = products.     (index='c
```

```
In [1]:
```

Want to leave a comment?

Note that your data can not have rows with duplicate values for the columns that you specify. If this is not the case, you will get an error message. If you can't ensure the uniqueness of your data, you will want to use the `pivot_table` method instead:

script.py

```
# Import the Pandas library
import ______ as pd

# Your DataFrame
products = pd.DataFrame({'category': ['C
'Cleaning', 'Entertainment', 'Entertainm
'Tech'],
                        'store': ['Walma
'Walmart', 'Fnac', 'Dia','Walmart'],
                        'price':[11.42,
.95, 19.99, 111.55],
                        'testscore': [4,

# Pivot your `products` DataFrame with `
pivot_products = products.___________(in
columns='store', values='price', aggfunc

# Check out the results
print(pivot_products)
```

IPython Shell

```
In [1]:
```

Solution Run

Note the additional argument `aggfunc` that gets passed to the `pivot_table` method. This argument indicates that you use an aggregation function used to combine multiple values. In this example, you can clearly see that the `mean` function is used.

Using `stack()` and `unstack()` to Reshape Your Pandas DataFrame

You have already seen an example of stacking in the answer to question 5! In essence, you might still remember that, when you stack a DataFrame, you make it taller. You move the innermost column index to become the innermost row index. You return a DataFrame with an index with a new inner-most level of row labels.

Go back to the full walk-through of the answer to question 5 if you're unsure of the

Want to leave a comment?

DataCamp

For a good explanation of pivoting, stacking and unstacking, go to this page.

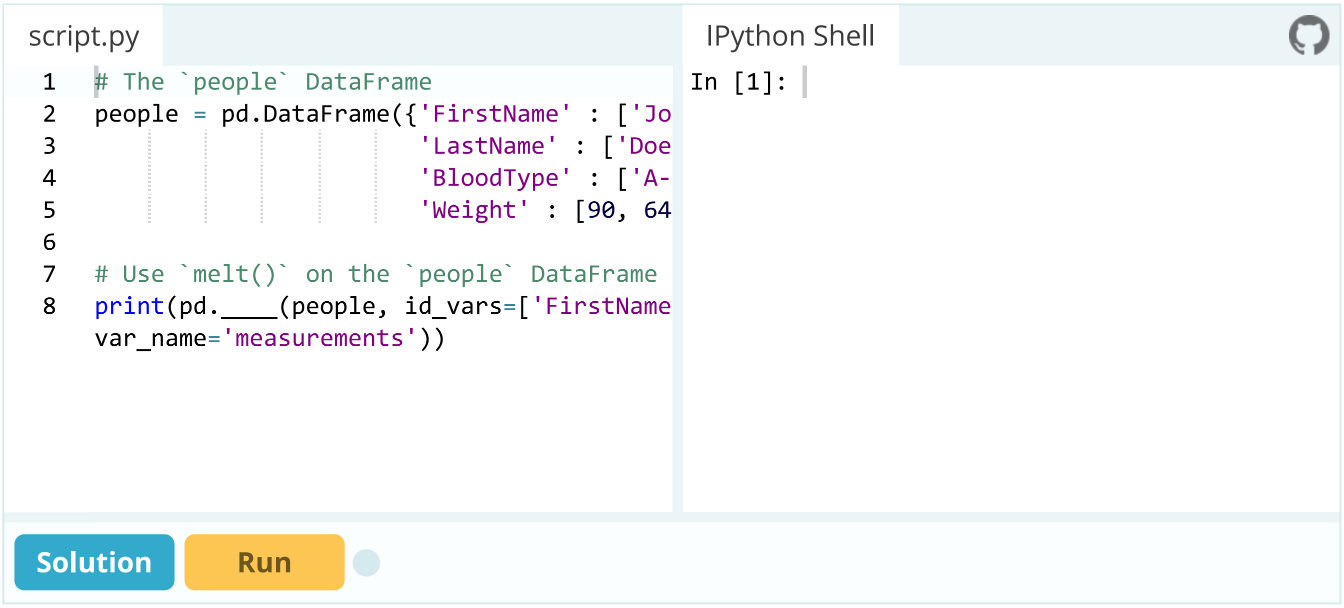
Reshape Your DataFrame With `melt()`

Melting is considered useful in cases where you have data that has one or more columns that are identifier variables, while all other columns are considered measured variables.

These measured variables are all "unpivoted" to the row axis. That is, while the measured variables that were spread out over the width of the DataFrame, the melt will make sure that they will be placed in the height of it. Or, yet in other words, your DataFrame will now become longer instead of wider.

As a result, you have two non-identifier columns, namely, 'variable' and 'value'.

Let's illustrate this with an example:



```
# The `people` DataFrame
people = pd.DataFrame({'FirstName' : ['Jo
                       'LastName' : ['Doe
                       'BloodType' : ['A-
                       'Weight' : [90, 64

# Use `melt()` on the `people` DataFrame
print(pd.____(people, id_vars=['FirstName
var_name='measurements'))
```

```
In [1]:
```

If you're looking for more ways to reshape your data, check out the documentation.

10. How To Iterate Over a Pandas DataFrame

Want to leave a comment?

```

for index, row in df.iterrows() :
    print(row['A'], row['B'])
```

Solution Run

`iterrows()` allows you to efficiently loop over your DataFrame rows as (index, Series) pairs. In other words, it gives you (index, row) tuples as a result.

11. How To Write a Pandas DataFrame to a File

When you have done your data munging and manipulation with Pandas, you might want to export the DataFrame to another format. This section will cover two ways of outputting your DataFrame: to a CSV or to an Excel file.

Output a DataFrame to CSV

To write a DataFrame as a CSV file, you can use `to_csv()`:

```
import pandas as pd
df.to_csv('myDataFrame.csv')
```

That piece of code seems quite simple, but this is just where the difficulties begin for most people because you will have specific requirements for the output of your data. Maybe you don't want a comma as a delimiter, or you want to specify a specific encoding, ...

Don't worry! You can pass some additional arguments to `to_csv()` to make sure that your data is outputted the way you want it to be!

Want to leave a comment?

- To use a specific character encoding, you can use the `encoding` argument:

```
import pandas as pd
df.to_csv('myDataFrame.csv', sep='\t', encoding='utf-8')
```

- Furthermore, you can specify how you want your `NaN` or missing values to be represented, whether or not you want to output the header, whether or not you want to write out the row names, whether you want compression, ... Read up on the options here.

Writing a DataFrame to Excel

Similarly to what you did to output your DataFrame to CSV, you can use `to_excel()` to write your table to Excel. However, it is a bit more complicated:

```
import pandas as pd
writer = pd.ExcelWriter('myDataFrame.xlsx')
df.to_excel(writer, 'DataFrame')
writer.save()
```

Note, however, that, just like with `to_csv()`, you have a lot of extra arguments such as `startcol`, `startrow`, and so on, to make sure output your data correctly. Go to this page to read up on them.

If, however, you want more information on IO tools in Pandas, you check out this page.

Python For Data Science Is More Than DataFrames

That's it! You've successfully completed the Pandas DataFrame tutorial!

The answers to the 11 frequently asked Pandas questions represent essential functions

Want to leave a comment?

DataCamp

Introduction to Python & Machine Learning is a must-complete.

188

33

RELATED POSTS

PYTHON +1

5 Tips To Write Idiomatic Pandas Code

Yassine Alouini
May 29th, 2017

MUST READ DATA MANIPULATION +2

Groupby, split-apply-combine and pandas

Hugo Bowne-Anderson
September 26th, 2017

PYTHON +2

Hierarchical indices, groupby and pandas

Hugo Bowne-Anderson

Want to leave a comment?

Sri Ramanujam
15/02/2018 03:55 AM
Please do not do this pop up. Thank you. Why should I leave a comment, when I am reading. Please stop

▲ 5 REPLY

Opeyemi Emoruwa
18/02/2018 05:14 PM
df.drop_duplicates([48], keep='last') shouldn't it be 4.8 instead of 48, theres no index with the value 48 here

▲ 4 REPLY

Karlijn Willems
18/02/2018 06:40 PM
Hi Opeyemi, what happens in the example shown in the tutorial is that you return a DataFrame with the duplicate rows removed, but only considering a certain column, namely the column [48].

In other words, in ``df.drop_duplicates([48], keep='last')``, the first argument specifies that you're only going to look at whether or not there are duplicate rows in column with name 48. In this case, you have two rows that have the same values, those with index 4.8 and 2.5. With a regular ```drop_duplicates()```, you would remove all duplicate rows but in this case, you specify that you want to keep only the last row of the two - so you will only remove one. As a result, only the row with index 4.8 will be removed.

I hope this helps!

▲ 4 REPLY

manolo711
27/02/2018 01:24 AM
Hi,

I think the part of reset_index is wrong... in order for the index to properly reset you need the following line:

df = df.reset_index(level=0, drop=True)

Once you print(df) then you can see the new df with the index reset

Want to leave a comment?

```
df.applymap(doubler)
```

```
print(df)
```

Shouldn't it be??:

```
df = df.applymap(doubler)
```

```
print(df)
```

So that the doubler functions does get implemented in the DataFrame. Can you explain why you are not showing this in the code?

▲ 1 REPLY

Karlijn Willems
27/02/2018 08:47 AM
Hi Manolo - You're indeed right in the first case. I have adjusted the exercise accordingly. In the second case, those exercises were meant as open-ended, which means that you should fiddle around the DataCamp Light chunks yourself to get to where you need to be and indeed assign the line of code to your DataFrame to overwrite it or, alternatively, create another variable to save your newly adjusted `df` DataFrame. Nevertheless, I can see where the confusion stems from and I have also adjusted those exercises as well.

▲ 2

Vitalii Kochubei
12/03/2018 09:47 AM
Would be perfect to have this kind of a cheatSheet!
Like this article!

▲ 2 REPLY

Karlijn Willems
12/03/2018 10:50 AM
Hi Vitalii - We actually do have 2 cheat sheets available for you :)) You can find them here.
Have fun!

Want to leave a comment?

DataCamp

TY. using them intensely

▲ 2

rahulsulakhia4
22/03/2018 12:53 PM

hello ,

very good material

i want to know how to work with pyplot in python from basic

▲ 2 REPLY

Karlijn Willems
22/03/2018 12:57 PM

Hi rahulsulakhia4 - You might be interested in this tutorial!

▲ 1 REPLY

rahulsulakhia4
22/03/2018 12:55 PM

transfer data between csv file and dataframe objects

transfer data between sql and dataframe objects

THANKS

▲ 1 REPLY

Marc Bertucci
24/03/2018 08:44 PM

Can you break down this section further:

data = np.array([['','Col1','Col2'],

['Row1',1,2],

Want to leave a comment?

```
index=data[1:,0],

columns=data[0,1:]))
```

I can't seem to wrap my head around the slicing of data, index and columns.

▲ 1 REPLY

T L
26/03/2018 05:16 AM
I love you tutorial. This is very useful!

▲ 1 REPLY

Sridivya Posa
01/04/2018 01:33 PM
Can you please explain the difference between df.rename(columns={0:1}, copy=False) and df.rename(columns={0:1}, copy=True). I am getting the same result for both. Pls explain the significance of "copy" parameter.

Thanks

Divya

▲ 2 REPLY

menghsichen
27/04/2018 04:42 PM
Thanks for sharing!

▲ 1 REPLY

Cuong Nhat Ha
11/05/2018 10:02 AM
Thank you very much for the tutorial!

But some example is still poor, e.g. **Adding an Index to a DataFrame.**

Want to leave a comment?

DataCamp

28/05/2018 01:08 PM

This is a fantastic tutorial. very well done. thank you.

▲ 1 REPLY

Hamish Sweidan
04/06/2018 08:22 PM

Hi,

the **Adding an Index to a DataFrame** example is not quite right, the index does not get set to 'C'. Can confirm by doing print(df)

You need to either specify inplace=True, or assign the result to a variable: df.set_index('ID', inplace=True)

Cheers,

Hamish

▲ 1 REPLY

Alexandre George Lustosa
05/06/2018 08:57 PM

Hi, nice Tutorial...

How can I remove "$" from the column below (the column is a float from a Dataframe)?

$108.101,09

$99.082,04

If i try this command: fake["AmountPaid"] = fake["AmountPaid"].apply(lambda x: x.replace(",",""))

I receive this error: "float object has no attribute replace (also, strip, etc)

Tks!

```
'float' object has no attribute 'replace'
```

Want to leave a comment?

DataCamp

```
'float' object has no attribute 'replace'
```

▲ 1 REPLY

Lukas Thaler
18/09/2018 06:51 AM

Simply convert your float to a string before doing the replace call and then back to float after you're done, as in fake["AmountPaid"] = fake["AmountPaid"].apply(lambda x: float(str(x).replace("$","")))

▲ 1 REPLY

Nikhil Jain
28/06/2018 10:06 PM

Awesome Tutorial

▲ 1 REPLY

Tracy Crider
11/09/2018 10:23 PM

Karlijn, The section on "Adding Rows to a Data Frame" could use some work. The comparisons aren't clear and the English used is confusing. Try comparing it to a SQL index or a ISAM data base, or merely a sorted list.

▲ 1 REPLY

Sarmad Abdulrahman
05/10/2018 11:56 AM

Thanks Karlijn Willems

for this great tutorial

▲ 1 REPLY

Ranjeet Tate
07/10/2018 07:27 PM

In "adding an index", df.set_index('C') only changes the view, not the DF itself.

Want to leave a comment?

DataCamp

"Now try replacing the `drop` argument by `inplace` in the code chunk above and see what happens!

Note how you use the `drop` argument to indicate that you want to get rid of the index that was there. If you would have used `inplace`, the original index with floats is added as an extra column to your DataFrame."

Doesn't the "inplace" refer to the DF, which is why the second print(df_reset) yields None?

▲ 1 REPLY

S. Imran
04/11/2018 09:54 AM
Alle benötigten Konzepte sind gut erklärt. Danke für das Tutorial.

▲ 1 REPLY

mohammad mustafa
06/11/2018 06:37 AM
Bogus plotter for python

▲ 1 REPLY

Goran Biljetina
13/12/2018 12:00 AM
I have a question specifically about dataFrame.to_csv() ... about 'path_or_buf' parameter accepting a file handle, in this case a RotatingFileHandle? if anyone has successfully done it I'd be super thankful

▲ 1 REPLY

Viraj B
18/01/2019 07:02 AM
I really **loved the feature** wherein one can create a code and run it side by side. Really helpful article though. Thank you. :)

▲ 1 REPLY

Want to leave a comment?

```
df = df.set_index("C")
```

and works well. The output was:

```
  A B
C
3 1 2
6 4 5
9 7 8
```

▲ 1 REPLY

Sandy Bagster
03/04/2019 03:11 PM
Great tutorial, for a killed wed developers who already provide web developing services, and for beginners who need programming basis.

▲ 1 REPLY

Anand Prakash
04/04/2019 06:44 AM
The rename is not working.It giving the same df as earlier.

▲ 1 REPLY