Stop Hurting Your Pandas!

Tags: Pandas, Programming, Python

This post will address the issues that can arise when Pandas slicing is used improperly. If you see the warning that reads "A value is trying to be set on a copy of a slice from a DataFrame", this post is for you.

By Pawel Rzeszucinski, Codewise

Source: Wikimedia Commons

Pandas is the king of data wrangling for virtually all data related tasks performed in Python.It's been around for 12 years now, although we've only just seen the release of the version 1.0 back in January of 2020. Manipulation, slicing and updating data with Pandas is very intuitive which is probably why the package has been a success from day one. Nevertheless, despite the simple and coherent syntax, there are situations where extra care should be taken to do exactly what is intended. This post will address the issues that can

arise when Pandas slicing is used improperly. If you see the warning that reads "*A value is trying to be set on a copy of a slice from a DataFrame*", this post is for you.
Pandas provides clear rules how to properly slice DataFrames and a good overview can be found here. However, we don't always follow best practice as doing so requires both acquiring necessary knowledge and maintaining certain levels of self-rigor. Apart from the options outlined in guidelines, Pandas allows us to access elements of Dataframes in many different ways. This may create a temptation to also try and perform data assignments in ways that may turn out to be inappropriate, resulting in some unexpected effects.
Let's start off by defining out a simple test dataframe:

```
df = pd.DataFrame({'x':[1,5,4,3,4,5],
                                  'y':[.1,.5,.4,.3,.4,.5],
                                  'w':[11,15,14,13,14,15]})


   x    y   w
0  1  0.1  11
1  5  0.5  15
2  4  0.4  14
3  3  0.3  13
4  4  0.4  14
5  5  0.5  15
```

Say we wanted to find all the DataFrame elements that correspond to 'x' column being larger than 3, and based on this change all the corresponding 'y' values to 50.

How to perform this correctly, according to Pandas best practice? By using the `.loc` method in this case:

```
df.loc[df['x']>3,'y']=50
```

We locate the row elements that meet our initial criterion (first argument), and the column which we want to update (second argument), all evaluated in a single call to the DataFrame.

The result is as expected.

```
   x     y   w
0  1   0.1  11
1  5  50.0  15
2  4  50.0  14
3  3   0.3  13
4  4  50.0  14
```

```
5   5   50.0   15
```

As mentioned a moment ago, Pandas provides a number of different ways of accessing (but not necessarily modifying!) the data.

Not sticking to the script (guidelines) may lead us into trouble. For instance it may be more natural for some people to write the same operation as follows:

```
df[df['x']>3]['y']=50
```

That's pretty clear, isn't it? Take the subset of the `df` that corresponds to `'x'>3`, and subsequently change the values in column 'y' to be equal to 50. Let's do that:

```
    x     y    w
0   1   0.1   11
1   5   0.5   15
2   4   0.4   14
3   3   0.3   13
4   4   0.4   14
5   5   0.5   15
```

I probably did a typo or something, let me run this once again.

```
df[df['x']>3]['y']=50


    x     y    w
0   1   0.1   11
1   5   0.5   15
2   4   0.4   14
3   3   0.3   13
4   4   0.4   14
5   5   0.5   15
```

No change whatsoever! Why?

We've encountered a so called 'chained indexing' effect, where essentially two indexers are used one after another e.g. `df[][]`
Let's decompose our command:

- `df[df['x']>3]` results in Pandas creating a separate copy of the original DataFrame
- `df[df['x']>3]['y'] = 50` assigns the new values to the column 'y' but on this temporarily created copy, not our original DataFrame.

A way of observing this explicitly is to use the base 'id' function which returns the address of the given object in the memory of the machine.

```
id(df)
2838845867680

id(df[df['x']>3])
2838845989832

id(id) == id(df[df['x']>3])
False
```

Interestingly, when we invert the order of the slice in our command i.e. call the columns first and then the criterion we want to satisfy, we get the expected result:

```
df['y'][df['x']>3]=50


   x     y   w
0  1   0.1  11
1  5  50.0  15
2  4  50.0  14
3  3   0.3  13
4  4  50.0  14
5  5  50.0  15
```

That's down to a fact, that when we select only one column from the DataFrame, Pandas creates a view, not a copy.

What's a view? It's essentially a proxy for the same object i.e. no new objects are created in the process.

```
z = df['y'] #z being a view of df['y']
id(df['y']) == id(z)
True
```

Even though we've achieved our goal, some side effects might have been triggered:

Let's look at this sequence of commands.

```
df # original dataframe
   x    y   w
0  1  0.1  11
1  5  0.5  15
2  4  0.4  14
```

```
3  3  0.3  13
4  4  0.4  14
5  5  0.5  15

z = df['y'] # view of column 'y'
z[z>=0.5] = 30

z
0      0.1
1     30.0
2      0.4
3      0.3
4      0.4
5     30.0

df
   x     y   w
0  1   0.1  11
1  5  30.0  15
2  4   0.4  14
3  3   0.3  13
4  4   0.4  14
5  5  30.0  15
```

Whoa! We though we created a separate object, called 'z' that is independent from `df` and the values of `df` are safe when we manipulate 'z'. Nope.
We've only created a view. The good thing is that Pandas will display the good ol' warning.

Pandas is doing this because it doesn't know if we want to change just the 'y' series (via proxy 'z'), or the value of the original `df`.
OK, so what if we wanted to extract 'z' as an independent object? The Pandas method `.copy()` serves exactly this purpose.
When we update our command to the one shown below, we will create a completely new object with its own address in the memory, and any updated on 'z' will leave `df` unaffected.

```
z = df['y'].copy()
id(df['y']) == id(z)
False
```

There are really two takeaways that will keep us guarded from any unwanted effects when working with slices and data manipulations:

1. avoid chained indexing. Always go for the `.loc`/`.iloc` (or `.at`/`.iat`) option:

2. `copy()` your variables to create independent objects and safeguard original sources from being unwillingly manipulated.

Bio: **Dr Pawel Rzeszucinski** is the author of over 30 publications and patents in the field of broadly defined data analytics. He obtained a master's degree in computer science from Cranfield University, after which he moved to the University of Manchester, where he obtained a PhD for the project for QinetiQ related to analytical solutions for the diagnosis of helicopter gearboxes. After returning to Poland, he worked as Senior Scientist at ABB Corporate Research Center and Senior Risk Modeler at Strategic Analytics for HSBC. He is currently working as a Chief Data Scientist at Codewise, an AdTech company. Dr Pawel Rzeszucinski is a member of the Forbes Technology Council.