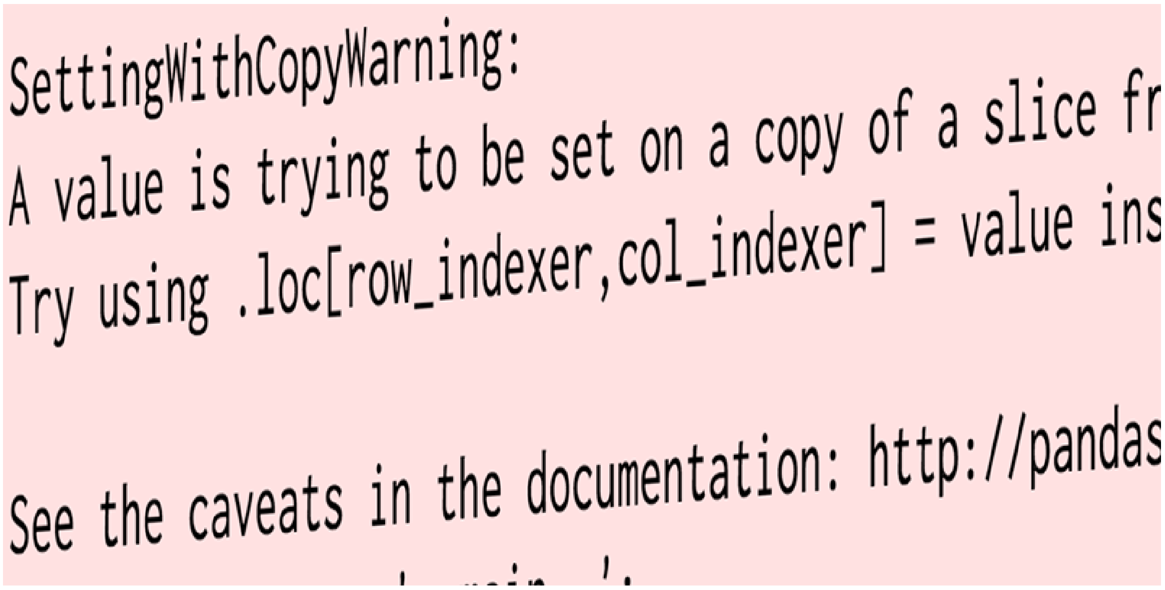
4 JULY 2017 / PANDAS

Understanding SettingwithCopyWarning in pandas


```
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice fr
Try using .loc[row_indexer,col_indexer] = value ins

See the caveats in the documentation: http://pandas
```

`SettingWithCopyWarning` is one of the most common hurdles people run into when learning pandas. A quick web search will reveal scores of Stack Overflow questions, GitHub issues and forum posts from programmers trying to wrap their heads around what this warning means in their particular situation. It's no surprise that many struggle with this; there are so many ways to index pandas data structures, each with its own particular nuance, and even pandas itself does not guarantee one single outcome for two lines of code that may look identical.

This guide explains why the warning is generated and shows you how to solve it. It also includes under-the-hood details to give you a better understanding of what's happening and provides some history on the topic, giving you perspective on why it all works this way.

In order to explore `SettingWithCopyWarning`, we're going to use a data set of the prices of Xboxes sold in 3-day auctions on eBay from the book Modelling Online Auctions. Let's take a look:

```
import pandas as pd

data = pd.read_csv('xbox-3-day-auctions.csv')
data.head()
```

	auctionid	bid	bidtime	bidder	bidderrate	ope
0	8213034705	95.0	2.927373	jake7870	0	95.
1	8213034705	115.0	2.943484	davidbresler2	1	95.
2	8213034705	100.0	2.951285	gladimacowgirl	58	95.
3	8213034705	117.5	2.998947	daysrus	10	95.
4	8213060420	2.0	0.065266	donnie4814	5	1.0

As you can see, each row of our data set concerns a single bid on a specific eBay Xbox auction. Here is a brief description of each column:

- `auctionid` — A unique identifier of each auction.

- `bid` — The value of the bid.
- `bidtime` — The age of the auction, in days, at the time of the bid.
- `bidder` — eBay username of the bidder.
- `bidderrate` - The bidder's eBay user rating.
- `openbid` — The opening bid set by the seller for the auction.
- `price` — The winning bid at the close of the auction.

What is SettingWithCopyWarning?

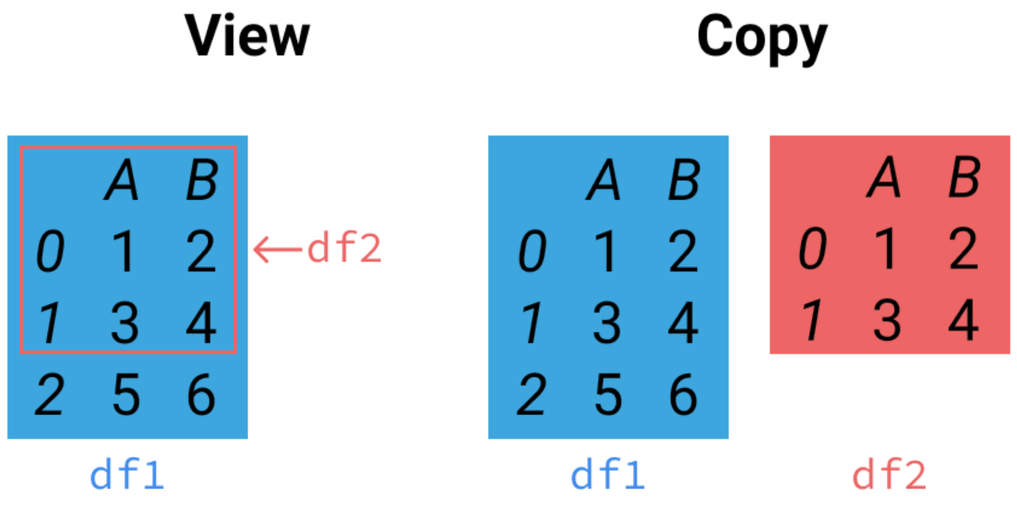
The first thing to understand is that `SettingWithCopyWarning` is a warning, and not an error.

While an error indicates that something is broken, such as invalid syntax or an attempt to reference an undefined variable, the job of a warning is to alert the programmer to potential bugs or issues with their code that are still permitted operations within the language. In this case, the warning very likely indicates a serious but inconspicuous mistake.

`SettingWithCopyWarning` informs you that your operation might not have worked as expected and that you should check the result to make sure you haven't made a mistake.

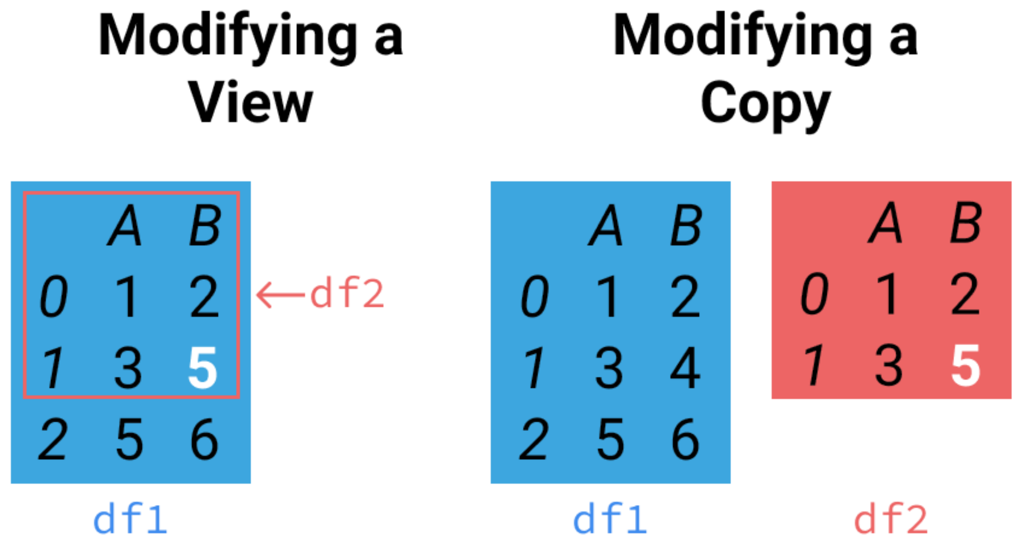
It can be tempting to ignore the warning if your code still works as expected. This is bad practice and `SettingWithCopyWarning` should **never** be ignored. Take some time to understand why you are getting the warning before taking action.

To understand what `SettingWithCopyWarning` is about, it's helpful to understand that some actions in pandas can return a view of your data, and others will return a copy.



As you can see above, the view `df2` on the left is just a subset of the original `df1` , whereas the copy on the right creates a new, unique object `df2` .

This potentially causes problem when we try to make changes:



Depending on what we're doing we might want to be modifying the original `df1` (left), or we might want to be modifying only `df2` (right). The warning is letting us know that our code may have done one, when we want it to have done the other.

We'll look at this in depth later, but for now let's get to grips with the two main causes of the warning and how to fix them.

Chained assignment

Pandas generates the warning when it detects something called chained assignment. Let's define a few terms we'll be using to explain things:

- Assignment — Operations that set the value of something, for example `data = pd.read_csv('xbox-3-day-auctions.csv')`. Often referred to as a **set**.
- Access — Operations that return the value of something, such as the below examples of indexing and chaining. Often referred to as a **get**.
- Indexing — Any assignment or access method that references a subset of the data; for example `data[1:5]`.
- Chaining — The use of more than one indexing operation back-to-back; for example `data[1:5][1:3]`.

Chained assignment is the combination of chaining and assignment. Let's take a quick look at an example with the data set we loaded earlier. We will go over this in more detail later on. For the sake of this example, let's say that we have been told that the user `'parakeet2004'` 's bidder rating is incorrect and we must update it. Let's start by looking at the current values.

```
data[data.bidder == 'parakeet2004']
```

	auctionid	bid	bidtime	bidder	bidderrate	open
6	8213060420	3.00	0.186539	parakeet2004	5	1.0
7	8213060420	10.00	0.186690	parakeet2004	5	1.0
8	8213060420	24.99	0.187049	parakeet2004	5	1.0

We have three rows to update the `bidderrate` field on; let's go ahead and do that.

```
data[data.bidder == 'parakeet2004']['bidderrate'] = 100
```

```
/Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/
A value is trying to be set on a copy of a slice from a DataFrame
```

```
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/pa
  if __name__ == '__main__':
```

Oh no! We've mysteriously stumbled upon the `SettingWithCopyWarning`!

If we take a look, we can see that in this case the values were not changed:

```
data[data.bidder == 'parakeet2004']
```

	auctionid	bid	bidtime	bidder	bidderrate	open
6	8213060420	3.00	0.186539	parakeet2004	5	1.0
7	8213060420	10.00	0.186690	parakeet2004	5	1.0
8	8213060420	24.99	0.187049	parakeet2004	5	1.0

The warning was generated because we have chained two indexing operations together. This is made easier to spot because we've used square brackets twice, but the same would be true if we used other access methods such as `.bidderrate`, `.loc[]`, `.iloc[]`, `.ix[]` and so on. Our chained operations are:

- `data[data.bidder == 'parakeet2004']`
- `['bidderrate'] = 100`

These two chained operations execute independently, one after another. The first is an access method (get operation), that will return a `DataFrame` containing all rows where `bidder` equals `'parakeet2004'`. The second is an assignment operation (set operation), that is called on this new `DataFrame`. We are not operating on the original `DataFrame` at all.

The solution is simple: combine the chained operations into a single operation using `loc` so that pandas can ensure the original `DataFrame` is set. Pandas will always ensure that unchained set operations, like the below, work.

```
# Setting the new value
data.loc[data.bidder == 'parakeet2004', 'bidderrate'] = 100

# Taking a look at the result
data[data.bidder == 'parakeet2004']['bidderrate']
```

```
6    100
7    100
8    100
Name: bidderrate, dtype: int64
```

This is what the warning suggests we do, and it works perfectly in this case.

Hidden chaining

Moving on to the second most common way people encounter `SettingWithCopyWarning`. Let's investigate winning bids. We will create a new dataframe to work with them, taking care to use `loc` going forward now that we have learned our lesson about chained assignment.

```
winners = data.loc[data.bid == data.price]
winners.head()
```

	auctionid	bid	bidtime	bidder	bidderrate
3	8213034705	117.5	2.998947	daysrus	10
25	8213060420	120.0	2.999722	djnoeproductions	17
44	8213067838	132.5	2.996632	*champaignbubbles*	202
45	8213067838	132.5	2.997789	*champaignbubbles*	202
66	8213073509	114.5	2.999236	rr6kids	4

We might write several subsequent lines of code working with our winners variable.

```
mean_win_time = winners.bidtime.mean()
... # 20 lines of code
mode_open_bid = winners.openbid.mode()
```

By chance, we come across another mistake in our `DataFrame`. This time the `bidder` value is missing from the row labelled `304`.

```
winners.loc[304, 'bidder']
```

```
nan
```

For the sake of our example, let's say that we know the true username of this bidder and update our data.

```
winners.loc[304, 'bidder'] = 'therealname'
```

```
/Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/pa
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/pan
  self.obj[item] = s
```

Another `SettingWithCopyWarning` ! But we used `loc` , how has this happened again? To investigate, let's take a look at the result of our code:

```
print(winners.loc[304, 'bidder'])
```

```
therealname
```

It worked this time, so why did we get the warning?

Chained indexing can occur across two lines as well as within one. Because `winners` was created as the output of a get operation (`data.loc[data.bid == data.price]`), it might be a copy of the original `DataFrame` or it might not be, but until we checked there was no way to know! When we indexed `winners` , we were actually using chained indexing.

This means that we may have also modified `data` as well when we were trying to modify `winners` .

In a real codebase, these lines could occur very far apart so tracking down the source of the problem might be more difficult, but the situation is the same.

To prevent the warning in this case, the solution is to explicitly tell pandas to make a copy when we create the new dataframe:

```
winners = data.loc[data.bid == data.price].copy()
winners.loc[304, 'bidder'] = 'therealname'
print(winners.loc[304, 'bidder'])
print(data.loc[304, 'bidder'])
```

```
therealname
nan
```

And that's it! It's that simple.

The trick is to learn to identify chained indexing and avoid it at all costs. If you want to change the original, use a single assignment operation. If you want a copy, make sure you force pandas to do just that. This will save time and make your code water-tight.

Also note that even though the `SettingWithCopyWarning` will only occur when you are setting, it's best to avoid chained indexing for gets too. Chained operations are slower and will cause problems if you decide to add assignment operations later on.

Tips and tricks for dealing with SettingWithCopyWarning

Before we do a much more in-depth analysis down below, let's pull out the microscope and take a look at some of the finer points and nitty-gritty details of the `SettingWithCopyWarning` .

Turning off the warning

First off, this article would never be complete without discussing how to explicitly control the `SettingWithCopy` settings. The pandas `mode.chained_assignment` option can take one of the values:

- `'raise'` — to raise an exception instead of a warning.
- `'warn'` — to generate a warning (default).
- `None` — to switch off the warning entirely.

For example, let's switch off the warning:

```
pd.set_option('mode.chained_assignment', None)
data[data.bidder == 'parakeet2004']['bidderrate'] = 100
```

Because this gives us no warning whatsoever, it's not recommended unless you have a full grasp of what you are doing. If you feel even the tiniest inkling of doubt, this isn't advised. Some developers take `SettingWithCopy` very seriously and choose to elevate it to an exception instead, like so:

```
pd.set_option('mode.chained_assignment', 'raise')
data[data.bidder == 'parakeet2004']['bidderrate'] = 100
```

```
-------------------------------------------------------------------
SettingWithCopyError                      Traceback (most recent c
<ipython-input-13-80e3669cab86> in <module>()
      1 pd.set_option('mode.chained_assignment', 'raise')
----> 2 data[data.bidder == 'parakeet2004']['bidderrate'] = 100
```

```
/Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/pa
   2427         else:
   2428             # set column
-> 2429             self._set_item(key, value)
   2430 
   2431     def _setitem_slice(self, key, value):

/Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/pa
   2500         # value exception to occur first
   2501         if len(self):
-> 2502             self._check_setitem_copy()
   2503 
   2504     def insert(self, loc, column, value, allow_duplicates=

/Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/pa
   1758 
   1759             if value == 'raise':
-> 1760                 raise SettingWithCopyError(t)
   1761             elif value == 'warn':
   1762                 warnings.warn(t, SettingWithCopyWarning, s

SettingWithCopyError:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/pan
```

This can be especially useful if you're working on a project with inexperienced pandas developers on your team, or a project that requires a high level of rigor or certainty in its integrity.

A more precise way to use this setting is by using a context manager.

```
# resets the option we set in the previous code segment
pd.reset_option('mode.chained_assignment')

with pd.option_context('mode.chained_assignment', None):
    data[data.bidder == 'parakeet2004']['bidderrate'] = 100
```

As you can see, this approach enables fine-grained warning suppression, rather than indiscriminately affecting the entire environment.

The is_copy property

Another trick that can be used to avoid the warning, is to modify one of the tools pandas uses to interpret a `SettingWithCopy` scenario. Each `DataFrame` has an `is_copy` property that is `None` by default but uses a `weakref` to reference the source `DataFrame` if it's a copy. By setting `is_copy` to `None`, you can avoid generating a warning.

```
winners = data.loc[data.bid == data.price]
winners.is_copy = None
winners.loc[304, 'bidder'] = 'therealname'
```

However, note this will **not** miraculously solve the problem, but it does make bug detection potentially very difficult.

Single vs multi-dtyped objects

A further point that is worth stressing is the distinction between single-dtyped and multi-dtyped objects. A `DataFrame` is single-dtyped if all its columns are of the same dtype; for example:

```
import numpy as np

single_dtype_df = pd.DataFrame(np.random.rand(5,2), columns=list('
print(single_dtype_df.dtypes)
single_dtype_df
```

```
A    float64
B    float64
dtype: object
```

	A	B
0	0.383197	0.895652
1	0.077943	0.905245
2	0.452151	0.677482
3	0.533288	0.768252
4	0.389799	0.674594

Whereas a `DataFrame` is multi-dtyped if its columns do not all have the same dtype, such as:

```
multiple_dtype_df = pd.DataFrame({'A': np.random.rand(5),'B': list
print(multiple_dtype_df.dtypes)
multiple_dtype_df
```

```
A    float64
B     object
dtype: object
```

	A	B
0	0.615487	a
1	0.946149	b
2	0.701231	c
3	0.756522	d
4	0.481719	e

For reasons explained in the *History* section below, an indexer-get operation on a multi-dtyped object will always return a copy. However, mainly for efficiency, an indexer get operation on a single-dtyped object almost always returns a view; the caveat here being that this depends on the memory layout of the object and is not guaranteed.

False positives

False positives, or situations where chained assignment is inadvertently reported, used to be more common in earlier versions of pandas but have since been mostly ironed out. For completeness, it's useful to include some examples here of fixed false positives. If you experience any of the situations below with earlier versions of pandas, then the warning can safely be ignored or suppressed (or avoided altogether by upgrading!)

Adding a new column to a `DataFrame` using a current column's values used to generate a warning, but this has been fixed.

```
data['bidtime_hours'] = data.bidtime.map(lambda x: x * 24)
data.head(2)
```

	auctionid	bid	bidtime	bidder	bidderrate	openb
0	8213034705	95.0	2.927373	jake7870	0	95.0
1	8213034705	115.0	2.943484	davidbresler2	1	95.0

Until recently, a false positive also occurred when setting using the `apply` method on a slice of a `DataFrame`, although this too has been fixed.

```
data.loc[:, 'bidtime_hours'] = data.bidtime.apply(lambda x: x * 24
data.head(2)
```

	auctionid	bid	bidtime	bidder	bidderrate	openb
0	8213034705	95.0	2.927373	jake7870	0	95.0
1	8213034705	115.0	2.943484	davidbresler2	1	95.0

And finally, until version 0.17.0, there was a bug in the `DataFrame.sample` method that caused spurious `SettingWithCopy` warnings. The `sample` method now returns a copy every time.

```
sample = data.sample(2)
sample.loc[:, 'price'] = 120
sample.head()
```

	auctionid	bid	bidtime	bidder	bidderrate	oper
481	8215408023	91.01	2.990741	sailer4eva	1	0.9
503	8215571039	100.00	1.965463	lambonius1	0	50.(

Chained assignment in Depth

Let's reuse our earlier example where we were trying to update the `bidderrate` column for each row in `data` with a `bidder` value of `'parakeet2004'`.

```
data[data.bidder == 'parakeet2004']['bidderrate'] = 100
```

```
/Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/ip
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/pan
  if __name__ == '__main__':
```

What pandas is really telling us with this `SettingWithCopyWarning` is that the behavior of our code is ambiguous, but to understand why this is and the wording of the warning, it will be helpful to go over a few concepts.

We talked briefly about views and copies earlier. There are two possible ways to access a subset of a `DataFrame`: either one could create a reference to the original data in memory (a view) or copy the subset into a new, smaller `DataFrame` (a copy). A view is a way of looking at a particular portion the **original** data, whereas a copy is a **clone** of that data to a new location in memory. As our diagram earlier showed, modifying a view will modify the original variable but modifying a copy will not.

For reasons that we will get into later, the output of 'get' operations in pandas is not guaranteed. Either a view or a copy could be returned when you index a pandas data structure, which means get operations on a `DataFrame` return a new `DataFrame` that can contain either:

- A copy of data from the original object.
- A reference to the original object's data without making a copy.

Because we don't know what will happen and each possibility has very different behavior, ignoring the warning is playing with fire.

To illustrate views, copies and this ambiguity more clearly, let's create a simple `DataFrame` and index it:

```
df1 = pd.DataFrame(np.arange(6).reshape((3,2)), columns=list('AB')
df1
```

	A	B
0	0	1
1	2	3
2	4	5

And let's assign a subset of `df1` to `df2`:

```
df2 = df1.loc[:1]
df2
```

	A	B
0	0	1
1	2	3

Given what we have learned, we know that `df2` could be a view on `df1` or a copy of a subset of `df1`.

Before we can get to grips with our problem, we also need to take another look at chained indexing. Expanding on our example with `'parakeet2004'`, we have chained together two indexing operations:

```
data[data.bidder == 'parakeet2004']
__intermediate__['bidderrate'] = 100
```

Where `__intermediate__` represents the output of the first call and is completely hidden from us. Remember that we would get the same problematic outcome if we had used attribute access:

```
data[data.bidder == 'parakeet2004'].bidderrate = 100
```

The same applies to any other form of chained call **because we are generating this intermediate object**.

Under the hood, chained indexing means making more than one call to `__getitem__` or `__setitem__` to accomplish a single operation. These are special Python methods that are invoked by the use of square brackets on an instance of a class that implements them, an example of what is called syntactic sugar. Let's look at what the Python interpreter will execute in our example.

```
# Our code
data[data.bidder == 'parakeet2004']['bidderrate'] = 100

# Code executed
data.__getitem__(data.__getitem__('bidder') == 'parakeet2004').__s
```

As you may have realized already, `SettingWithCopyWarning` is generated as a result of this chained `__setitem__` call. You can try this for yourself - the lines above function identically. For clarity, note that the second `__getitem__` call (for the `bidder` column) is nested and not at all part of the chaining problem here.

In general, as discussed, pandas does not guarantee whether a get operation will return a view or a copy of the data. If a view is returned in our example, the second expression in our chained assignment will be a call to `__setitem__` on the original object. But, if a copy is returned, it's the copy that will be modified instead - the original object does not get modified.

This is what the warning means by "a value is trying to be set on a copy of a slice from a DataFrame". As there are no references to this copy, it will ultimately be garbage collected. The `SettingWithCopyWarning` is letting us know that pandas cannot determine whether a view or a copy was returned by the first `__getitem__` call, and so it's unclear whether the assignment changed the original object or not. Another way to think about why pandas gives us this warning is because the answer to the question "are we modifying the original?" is unknown.

We do want to modify the original, and the solution that the warning suggests is to convert these two separate, chained operations into a single assignment operation using `loc` . This will remove chained indexing from our code and we will no longer receive the warning. Our fixed code and its expanded version will look like this:

```
# Our code
data.loc[data.bidder == 'parakeet2004', 'bidderrate'] = 100

# Code executed
data.loc.__setitem__((data.__getitem__('bidder') == 'parakeet2004'
```

Our DataFrame's `loc` property is guaranteed to be the original `DataFrame` itself but with expanded indexing capabilities.

False negatives

Using `loc` doesn't end our problems because get operations with `loc` can still return either a view or a copy. Let's quickly examine a somewhat convoluted example.

```
data.loc[data.bidder == 'parakeet2004', ('bidderrate', 'bid')]
```

	bidderrate	bid
6	100	3.00
7	100	10.00
8	100	24.99

We've pulled two columns out this time rather than just the one. Let's try to set all the `bid` values.

```
data.loc[data.bidder == 'parakeet2004', ('bidderrate', 'bid')]['bi
data.loc[data.bidder == 'parakeet2004', ('bidderrate', 'bid')]
```

	bidderrate	bid
6	100	3.00
7	100	10.00
8	100	24.99

No effect and no warning! We have set a value on a copy of a slice but it was not detected by pandas - this is a false negative. Just because we have used `loc` doesn't mean we can start using chained assignment again. There is an old, unresolved issue on GitHub for this particular bug.

The correct way to do this is as follows:

```
data.loc[data.bidder == 'parakeet2004', 'bid'] = 5.0
data.loc[data.bidder == 'parakeet2004', ('bidderrate', 'bid')]
```

	bidderrate	bid
6	100	5.0
7	100	5.0
8	100	5.0

You might wonder how someone could possibly end up with such a problem in practice, but it's easier than you might expect when assigning the results of `DataFrame` queries to variables as we do in the next section.

Hidden chaining

Let's look again at our hidden chaining example from earlier, where we were trying to set the `bidder` value from the row labelled `304` in our `winners` variable.

```
winners = data.loc[data.bid == data.price]
winners.loc[304, 'bidder'] = 'therealname'
```

```
/Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/pa
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/pan
  self.obj[item] = s
```

We get another `SettingWithCopyWarning` even though we used `loc`. This problem can be incredibly confusing as the warning message appears to be suggesting that we do what we have already done.

But think about the `winners` variable. What really is it? Given that we instantiated it via `data.loc[data.bid == data.price]`, we cannot know whether it's a view or a copy of our original `data` `DataFrame` (because get operations return either a view or a copy). Combining the instantiation with the line that generated the warning makes clear our mistake.

```
data.loc[data.bid == data.price].loc[304, 'bidder'] = 'therealname
```

```
/Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/pa
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/pan
  self.obj[item] = s
```

We used chained assignment again, but this time it was broken across two lines. Another way to think about this is to ask the question "does this modify one or two things?" In our case, the answer is unknown: if `winners` is a copy then only `winners` is affected but if it's a view both `winners` and `data` will show updated values. This situation can occur between lines that are very far apart within a script or codebase, making the source of the problem potentially very difficult to track down.

The intention of the warning here to prevent us from thinking our code will modify the original `DataFrame` when it won't, or that we're modifying a copy rather than the original. Delving into old issues on pandas' GitHub repo, you can read the devs explaining this themselves.

How we resolve this problem depends very much on our own intentions. If we are happy to work with a copy of our original data, the solution is simply to force pandas to make a copy.

```
winners = data.loc[data.bid == data.price].copy()
winners.loc[304, 'bidder'] = 'therealname'

print(data.loc[304, 'bidder']) # Original
print(winners.loc[304, 'bidder']) # Copy
```

```
nan
therealname
```

If, on the other hand, you require that the original `DataFrame` is updated then you should work with the original `DataFrame` instead of instantiating other variables with unknown behavior. Our prior code would become:

```
# Finding the winners
winner_mask = data.bid == data.price

# Taking a peek
data.loc[winner_mask].head()

# Doing analysis
mean_win_time = data.loc[winner_mask, 'bidtime'].mean()
... # 20 lines of code
mode_open_bid = data.loc[winner_mask, 'openbid'].mode()

# Updating the username
data.loc[304, 'bidder'] = 'therealname'
```

In more complex circumstances, such as modifying a subset of a subset of a `DataFrame` , instead of using chained indexing one can modify the slices one is making via `loc` on the original `DataFrame` .

For example, you could change our new `winner_mask` variable above or create a new variable that selected a subset of winners, like so:

```
high_winner_mask = winner_mask & (data.price > 150)
data.loc[high_winner_mask].head()
```

	auctionid	bid	bidtime	bidder	bidderrate
225	8213387444	152.0	2.919757	uconnbabydoll1975	15
328	8213935134	207.5	2.983542	toby2492	0
416	8214430396	199.0	2.990463	volpendesta	4
531	8215582227	152.5	2.999664	ultimatum_man	2

This technique is more robust to future codebase maintenance and scaling.

History

You might be wondering why the whole `SettingWithCopy` problem can't simply be avoided entirely by explicitly specifying indexing methods that return either a view or a copy rather than creating the confusing situation we find ourselves in. To understand this, we must look into pandas' past.

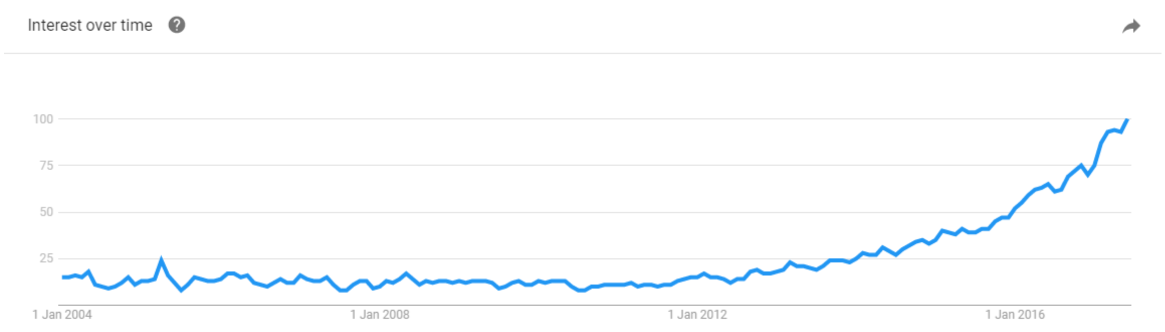
The logic pandas uses to determine whether it returns a view or a copy stems from its use of the NumPy library, which underlies pandas' operation. Views actually entered the pandas lexicon via NumPy. Indeed, views are useful in NumPy because they are returned predictably. Because NumPy arrays are single-typed, pandas attempts to minimize space and processing requirements by using the most appropriate dtype. As a result, slices of a `DataFrame` that contain a single dtype can be returned as a view on a single NumPy array, which is a highly efficient way to handle the operation. However, multi-dtype slices can't be stored in the same way in NumPy so efficiently. Pandas juggles versatile indexing functionality with the ability to use its NumPy core most effectively.

Ultimately, indexing in pandas was designed to be useful and versatile in a way that doesn't exactly marry the functionality of the underlying NumPy arrays at its core. The interaction between these elements of design and function over time has led to a complex set of rules that determine whether or not a view or a copy can be returned. Experienced pandas developers are generally happy with pandas' behaviors because they are comfortable l navigating its indexing behaviors.

Unfortunately for newcomers to the library, chained indexing is almost unavoidable despite not being the intended approach simply because get operations return indexable pandas objects. Furthermore, in the words of Jeff Reback, one of the core developers of pandas for several years, "It's simply not possible from a language perspective to detect chain indexing directly; it has to be inferred".

Consequently, the warning was introduced in version 0.13.0 near the end of 2013 as a solution to the silent failure of chained assignment encountered by many developers.

Prior to version 0.12, the `ix` indexer was the most popular (in the pandas nomenclature, "indexers" such as `ix`, `loc` and `iloc` are simply constructs that allow objects to be indexed with square brackets just like arrays, but with special behavior). But it was around this time, in mid-2013, that the pandas project was beginning to gain momentum and catering to novice users was of rising importance. Since this release the `loc` and `iloc` indexers have consequently been preferred for their more explicit nature and easier to interpret usages.



Google Trends: pandas

The `SettingWithCopyWarning` has continued to evolve after its introduction, was hotly discussed in many GitHub issues for several years, and is even still being updated, but it's here to stay and understanding it remains crucial to becoming a pandas expert.

Wrapping up

The complexity underlying the `SettingWithCopyWarning` is one of the few rough edges in the pandas library. Its roots are very deeply embedded in the library and should not be ignored. In Jeff Reback's own words there "are no cases that I am aware [of] that you should actually ignore this warning. ... If you do certain types of indexing it will never work, others it will work. You are really playing with fire."

Fortunately, addressing the warning only requires you to identify chained assignment and fix it. If there's just one thing to take away from all this, it's that.

Benjamin Pryke
Python and web developer with a background in computer science and machine learning. Co-founder of FinTech firm Machina Capital. Part-time gymnast and digital bohemian.

Read More

— Dataquest Data Science Blog —
Pandas

Data Science Portfolio Project: Where to Advertise an E-learning Product

Data Science Portfolio Project: Is Fandango Still Inflating Ratings?

Jupyter Notebook for Beginners: A Tutorial

See all 27 posts →

LEARN PYTHON
Jul 12, 2017
Should I learn Python 2 or 3?

A walkthrough of the key history of Python 2 and Python 3 to help you make the best decision when learning the popular programming language.

JOSH DEVLIN

LEARN PYTHON
Jun 28, 2017
Web Scraping with Python and BeautifulSoup

This intermediate tutorial teaches you how to scrape data from multiple pages using Python and BeautifulSoup.

ALEX OLTEANU