python pandas dataframe, is it pass-by-value or pass-by-reference

Asked 4 years ago Active 2 years, 10 months ago Viewed 56k times

80

31

If I pass a dataframe to a function and modify it inside the function, is it pass-by-value or pass-by-reference?

I run the following code

```
a = pd.DataFrame({'a':[1,2], 'b':[3,4]})
def letgo(df):
    df = df.drop('b',axis=1)
letgo(a)
```

the value of `a` does not change after the function call. Does it mean it is pass-by-value?

I also tried the following

```
xx = np.array([[1,2], [3,4]])
def letgo2(x):
    x[1,1] = 100
def letgo3(x):
    x = np.array([[3,3],[3,3]])
```

It turns out `letgo2()` does change `xx` and `letgo3()` does not. Why is it like this?

python pandas pass-by-reference pass-by-value

edited Aug 11 '16 at 12:15

asked Aug 11 '16 at 11:59

nos
14.4k 21 75 99

3 For explanations of Python's pass by assignment model, read Facts and Myths about Pythons names and values, FAQ: How do I write a function with output parameters (call by reference)?, SO: How do I pass a variable by reference?. – unutbu Aug 11 '16 at 12:11

6 Answers

Active Oldest Votes

87

The short answer is, Python always does pass-by-value, but every Python variable is actually a pointer to some object, so sometimes it looks like pass-by-reference.

In Python every object is either mutable or non-mutable. e.g., lists, dicts, modules and Pandas data frames are mutable, and ints, strings and tuples are non-mutable. Mutable objects can be changed internally (e.g., add an element to a list), but non-mutable objects cannot.

you are doing is changing the local variable to point to a different object. This doesn't alter (mutate) the original object that the variable pointed to, nor does it make the external variable point to the new object. At this point, the external variable still points to the original object, but the internal variable points to a new object.

If you want to alter the original object (only possible with mutable data types), you have to do something that alters the object *without* assigning a completely new value to the local variable. This is why `letgo()` and `letgo3()` leave the external item unaltered, but `letgo2()` alters it.

As @ursan pointed out, if `letgo()` used something like this instead, then it would alter (mutate) the original object that `df` points to, which would change the value seen via the global `a` variable:

```
def letgo(df):
    df.drop('b', axis=1, inplace=True)

a = pd.DataFrame({'a':[1,2], 'b':[3,4]})
letgo(a)  # will alter a
```

In some cases, you can completely hollow out the original variable and refill it with new data, without actually doing a direct assignment, e.g. this will alter the original object that `v` points to, which will change the data seen when you use `v` later:

```
def letgo3(x):
    x[:] = np.array([[3,3],[3,3]])

v = np.empty((2, 2))
letgo3(v)   # will alter v
```

Notice that I'm not assigning something directly to `x`; I'm assigning something to the entire internal range of `x`.

If you absolutely must create a completely new object and make it visible externally (which is sometimes the case with pandas), you have two options. The 'clean' option would be just to return the new object, e.g.,

```
def letgo(df):
    df = df.drop('b',axis=1)
    return df

a = pd.DataFrame({'a':[1,2], 'b':[3,4]})
a = letgo(a)
```

Another option would be to reach outside your function and directly alter a global variable. This changes `a` to point to a new object, and any function that refers to `a` afterward will see that new object:

```
def letgo():
    global a
    a = a.drop('b',axis=1)
```

Directly altering global variables is usually a bad idea, because anyone who reads your code will have a hard time figuring out how `a` got changed. (I generally use global variables for shared parameters used by many functions in a script, but I don't let them alter those global variables.)

edited Aug 13 '16 at 1:13

answered Aug 12 '16 at 19:30

Matthias Fripp
11.4k 3 22 33

8

The question isn't PBV vs. PBR. These names only cause confusion in a language like Python; they were invented for languages that work like C or like Fortran (as the quintessential PBV and PBR languages). It is true, but not enlightening, that Python always passes by value. The question here is whether the value itself is mutated or whether you get a new value. Pandas usually errs on the side of the latter.

http://nedbatchelder.com/text/names.html explains very well what Python's system of names is.

answered Aug 11 '16 at 12:21

Mike Graham
59.8k 12 82 119

1 The semantics of passing and assigning in Python are exactly the same as in Java, and the same things you say can be equally applied to Java. Yet on StackOverflow and elsewhere on the Internet people apparently find it "enlightening" to impress upon you that Java is always pass by value whenever this issue comes up. – newacct Aug 12 '16 at 2:35

7

To add to @Mike Graham's answer, who pointed to a very good read:

In your case, what is important to remember is the difference between *names* and *values*. `a`, `df`, `xx`, `x`, are all *names*, but they refer to the same or different *values* at different points of your examples:

- In the first example, `letgo` **rebinds** `df` to another value, because `df.drop` returns a new `DataFrame` unless you set the argument `inplace = True` (see doc). That means that the name `df` (local to the `letgo` function), which was referring to the value of `a`, is now referring to a new value, here the `df.drop` return value. The value `a` is referring to still exists and hasn't changed.
- In the second example, `letgo2` **mutates** `x`, without rebinding it, which is why `xx` is modified by `letgo2`. Unlike the previous example, here the local name `x` always refers to the value the name `xx` is referring to, and changes that value *in place*, which is why the value `xx` is referring to has changed.
- In the third example, `letgo3` **rebinds** `x` to a new `np.array`. That causes the name `x`, local to `letgo3` and previously referring to the value of `xx`, to now refer to another value, the new `np.array`. The value `xx` is referring to hasn't changed.

answered Aug 12 '16 at 18:48

ursan

3

Python is neither pass by value nor pass by reference. It is pass by assignment.

Supporting reference, the Python FAQ: https://docs.python.org/3/faq/programming.html#how-do-i-write-a-function-with-output-parameters-call-by-reference

IOW:

1. If you pass an immutable value, changes to it do not change its value in the caller - because you are rebinding the name to a new object.
2. If you pass a mutable value, changes made in the called function, also change the value in the caller, so long as you do not rebind that name to a new object. If you reassign the variable, creating a new object, that change and subsequent changes to the name are not seen in the caller.

So if you pass a list, and change its 0th value, that change is seen in both the called and the caller. But if you reassign the list with a new list, this change is lost. But if you slice the list and replace *that* with a new list, that change is seen in both the called and the caller.

EG:

```
def change_it(list_):
    # This change would be seen in the caller if we left it alone
    list_[0] = 28

    # This change is also seen in the caller, and replaces the above
    # change
    list_[:] = [1, 2]

    # This change is not seen in the caller.
    # If this were pass by reference, this change too would be seen in
    # caller.
    list_ = [3, 4]

thing = [10, 20]
change_it(thing)
# here, thing is [1, 2]
```

If you're a C fan, you can think of this as passing a pointer by value - not a pointer to a pointer to a value, just a pointer to a value.

HTH.

answered Oct 5 '17 at 20:36

dstromberg
6,007 14 20

0

Here is the doc for drop:

> Return new object with labels in requested axis removed.

But as for all objects in python, the data frame is passed to the function by reference.

answered Aug 11 '16 at 12:02

Israel Unterman
10.9k 2 21 31

but I assigned it to `df` inside the function, doesn't it mean the referenced value has been changed to the new object? – nos Aug 11 '16 at 12:07

Assigning to a local name will never change what object a name is bound to in another scope. – Mike Graham Aug 11 '16 at 12:22

0

you need to make 'a' global at the start of the function otherwise it is a local variable and does not change the 'a' in the main code.

answered Aug 11 '16 at 12:06

zosan
1 2