

Principles of Programming in Econometrics

Introduction, structure, and advanced programming techniques

Charles S. Bos

Vrije Universiteit Amsterdam
Tinbergen Institute

`c.s.bos@vu.nl`

August 2020 – Version Python

Separate lecture slides

Compilation: August 3, 2020

PPEctr

└ Variables: [View](#) or [copy](#)

View or copy

What does assignment do in Python? Check out this code:

view_copy.py

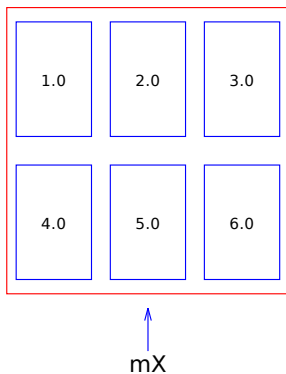
```
mX= np.arange(6)+1.0    # Get vector of numbers 1.0, 2.0, ..., 6.0
print ('Shape :', mX.shape)
mX.shape= (2, 3)        # Assign TO shape characteristic
print ('Shape :', mX.shape)
print ('What is mX now?\n', mX)

mY= mX                  # New view of mX
mY[0, 0]= 0             # Change element of Y
print ('What is mX now, after changing element of Y?\n', mX)

mY= np.copy(mX)         # New copy of mX
mY[0, 0]= -1
print ('What is mX now, after re-copying y, putting a -1 in first location?\n', mX)
print ('What is mY now?\n', mY)
```

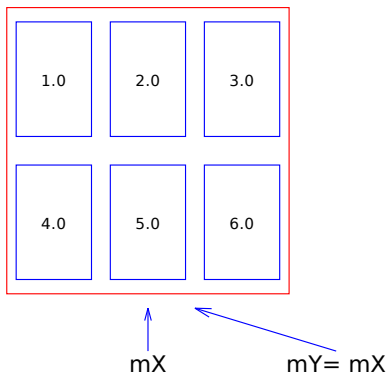
What happens here?

View or copy II



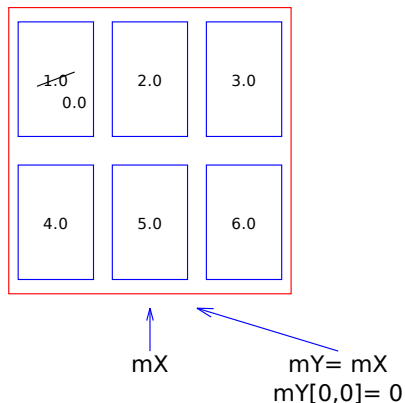
Step 1: Creating `mX`

View or copy II



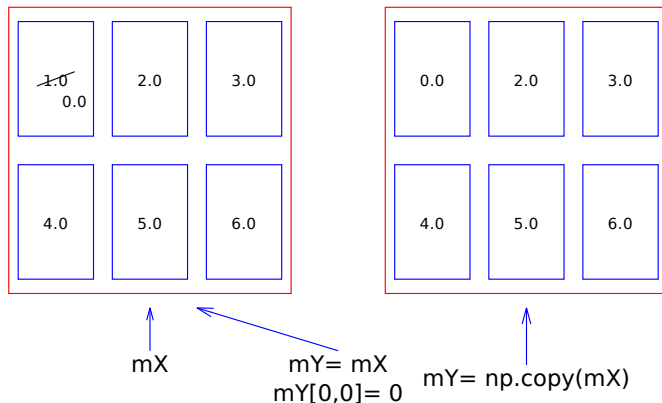
Step 2: Creating `mY = mX`, new *view* of *same matrix*

View or copy II



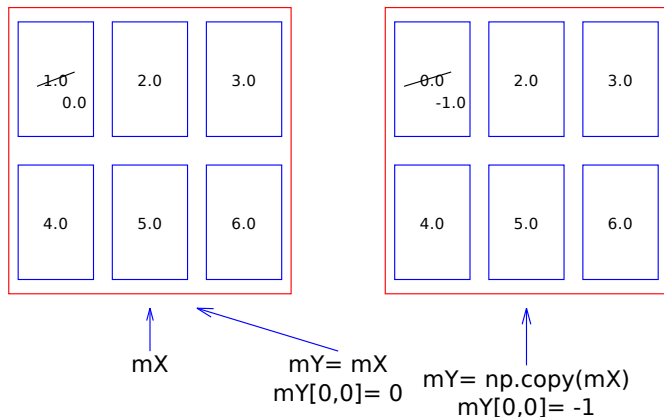
Step 3: Alter `mY[0,0] = 0` changes `mX` as well...

View or copy II



Step 4: Now explicitly copy over `mY = np.copy(mX)`

View or copy II



Step 5: Change $mY[0,0] = -1$ leaves mX unaltered

View or copy III

How can I know whether I get a view or a copy?

```
print ('Is mX the same as mY? ', id(mX) == id(mY))  
print ('id(mX)=%i, id(mY)=%i' % (id(mX), id(mY)))
```

Check the id...

View or copy III

How can I know whether I get a view or a copy?

```
print ('Is mX the same as mY? ', id(mX) == id(mY))  
print ('id(mX)=%i, id(mY)=%i' % (id(mX), id(mY)))
```

Check the id...

What is the advantage of the 'view' of an object, not copying?

- ▶ Save memory, not having multiple copies of same (large) object
- ▶ Pass a (view to) a mutable object (ndarray/matrix/vector/dataframe) to a function, change *part* of it

View or copy III

Change part of a matrix, output value through argument:

view_copy2.py

```
def FillRes(mRes):
    """
    Purpose:
        Perform (fake) calculating, filling mRes column by column

    Inputs:
        mRes      iR x iC matrix, to be overwritten

    Outputs:
        mRes      iR x iC matrix, filled by column

    Return value:
        dR        double, sum of all results
    """
    (iR, iC) = mRes.shape
    dR = 0.0
    for c in range(iC):
        vC = np.random.randn(iR)      # Do computations. Here: Get R random outcomes
        mRes[:,c] = vC
        dR += vC.sum()

    return dR
```

Passing effectively a 'basket' to the function, allowing change of

Basket: Mutable vs immutable

Python hands over a new 'view' of a list to a function. This implies:

- ▶ The function can access *the same* list/matrix/array/dataframe
- ▶ As long as it is careful not to replace the list, it can alter elements
- ▶ Replaced elements will be handed back to the main program, as such

Examples:

- ▶ `lX[1]= 'hello':` Replace second list item by a new string
- ▶ `mX[0,4]= 3.14:` Replace element in row 1, column 5, by 3.14
- ▶ `mX[:, :]= mX * mX:` Replace all elements of existing matrix `mX` by their squares, keeping same 'basket'

Q: What is difference of last example, `mX[:, :]= mX * mX`, with `mX= mX * mX`?

Python and other languages

Concepts are similar

- ▶ Python (and e.g. Ox/Gauss/Matlab) have automatic typing. Use it, but carefully...
- ▶ C/C++/Fortran need to have types and sizes specified at the start. More difficult, but still same concept of variables.
- ▶ Precise manner for specifying a matrix differs from language to language. Python needs some getting used to, but is (very...) flexible in the end
- ▶ Remember: An element has a value and a name
- ▶ A program moves the elements around, hopefully in a smart manner

**Keep track of your variables,
know what is their *type*, *size*, and *scope***

Python and other languages II

Concepts similar, implementation different:

- ▶ Python (and e.g. R, Julia) have object-like variables: Each variable has *characteristics*
- ▶ Python uses views of the data, often without copying, dangerous
- ▶ Powerful but sometimes confusing (see before)

Warning: Too much to discuss here, but dangerous implications... See e.g. <https://medium.com/@larmalade/python-everything-is-an-object-and-some-objects-are-mutable-4f55eb2b468b>

All languages

Programming is exact science

- ▶ Keep track of your variables
- ▶ Know what is their scope
- ▶ Program in small bits
- ▶ Program *extremely* structured
- ▶ Document your program wisely
- ▶ Think about algorithms, data storage, outcomes etc.

Further topics: Scope

Any variable is available only within the block in which it is declared.

In practice:

1. Arguments to a function, e.g. `mX` in `fnPrint(mX)`, are available within this function
2. A local variable `mY` is only known *below* its first use, within the present function
3. A global variable, indicated with `global g_mZ` at the start of a function, and retains its value between functions.

Further topics: Scope

Any variable is available only within the block in which it is declared.

In practice:

1. Arguments to a function, e.g. `mX` in `fnPrint(mX)`, are available within this function
2. A local variable `mY` is only known *below* its first use, within the present function
3. A global variable, indicated with `global g_mZ` at the start of a function, and retains its value between functions.

(but forget about globals... or use them the absolute minimum?)

Further topics: Scope II

Listing 1: scope_global.py

```
def localfunc():
    global g_sX
    print ("In localfunc: g_sX= ", g_sX)

    g_sX= "and goodbye"    # Change the full global variable

#####
### main
def main():
    global g_sX

    g_sX= "Hello"
    localfunc()
    print ("In main, after localfunc: g_sX= ", g_sX)
```

Rules for globals:

- ▶ Only use them when absolutely necessary (dangerous!)
- ▶ Annotate them, g_
- ▶ Fill them at *last possible moment*
- ▶ Do not change them afterwards (unless absolutely necessary)