

# 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: July 27, 2020

## Overview

# Principles of Programming in Econometrics

D0: Syntax, example 2<sup>8</sup>

D1: Structure, scope

D2: Numerics, packages

D3: Optimisation, speed

# Day 0: Syntax

## 9.30 Introduction

Example:  $2^8$

Elements

Main concepts

Closing thoughts

Revisit E0

## 13.30 Practical (at VU, main building)

- ▶ Checking variables, types, conversion and functions
- ▶ Implementing Backsubstitution

## Recap

But let us recap the first lessons, and extend the knowledge...

## All work in functions

All work is done in functions (or at least, that's what we'll do!)

### Listing 1: recap1.py

```
def main():  
    dX= 5.5  
    dX2= dX ** 2  
  
    print ("The square of ", dX, " is ", dX2)  
  
#####  
## start main  
if __name__ == "__main__":  
    main()
```

Note:

- ▶ This function `main()` takes no arguments
- ▶ ...but Python only executes the first line outside a function
- ▶ ...which is an `if` statement, calling `main()`
- ▶ ...only if we call this routine as a separate program (allows us to import files later)

## Quiz-time: Main

### Listing 2: recap\_quiz.py

```
def main():  
    print ("Hello world")  
  
#####  
### start main  
print ("This is an orphan statement")  
if __name__ == "__main__":  
    main()
```

**Q1** What is the output of this program?

**Q2** Would anything change if the line starting with `if` is skipped?

**Q3** And why does one use the conditional statement?

## Quiz-time: Main

### Listing 3: recap\_quiz.py

```
def main():  
    print ("Hello world")  
  
#####  
### start main  
print ("This is an orphan statement")  
if __name__ == "__main__":  
    main()
```

**Q1** What is the output of this program?

**Q2** Would anything change if the line starting with `if` is skipped?

**Q3** And why does one use the conditional statement?

Answer: Deep Python philosophy. But follow the custom...

## Squaring and printing

Use other functions to do your work for you

### Listing 4: recap2.py

```
import math

def printsquare(dIn):
    dOut= math.pow(dIn, 2)
    print ("The square of ", dIn, " is ", dOut)

def main():
    dX= 5.5
    printsquare(dX)

    printsquare(6.3)
```

Here, printsquare does not give a return value, only screen output.

printsquare takes in one argument, with a value locally called dIn. Can either be a true variable (dX), a constant (6.3), or even the outcome of a calculation (dX-5).

Note the usage of import math for the math.pow() function.



## Return

Use `return` a to give one value back to the calling function (as e.g. the `math.pow()` function also gives a value back).

### Listing 5: recap\_return.py

```
def createones(iR, iC):  
    mX= np.ones((iR, iC))    # Use numpy, handing over Tuple (iR, iC)  
    return mX  
  
def main():  
    iR= 2                      # Magic numbers  
    iC= 5  
    mX= createones(iR, iC)    # Estimation, catch output of createones  
    print ("Matrix mX=\n", mX)    # Output
```

Alternative: See below, altering pre-defined mutable (= matrix) argument

## Return: A tuple

Alternatively, return a *tuple* if multiple values should be handed back to the calling routine:

### Listing 6: recap\_return\_tuple.py

```
def createones_size(iR, iC):  
    mX= np.ones((iR, iC))    # Use numpy, handing over Tuple (iR, iC)  
    iSize= iR*iC  
    return (mX, iR*iC)  
  
def main():  
    iR= 2                # Magic numbers  
    iC= 5  
    (mX, iSize)= createones_size(iR, iC)                # Estimation  
    print ("Matrix mX=\n", mX, "\nof size ", iSize) # Output
```

Alternative: See below, altering pre-defined mutable (= matrix) argument

**Q: Why is this example rather stupid/non-robust?**

## Return: A tuple

Alternatively, return a *tuple* if multiple values should be handed back to the calling routine:

### Listing 7: recap\_return\_tuple.py

```
def createones_size(iR, iC):  
    mX= np.ones((iR, iC))    # Use numpy, handing over Tuple (iR, iC)  
    iSize= iR*iC  
    return (mX, iR*iC)  
  
def main():  
    iR= 2           # Magic numbers  
    iC= 5  
    (mX, iSize)= createones_size(iR, iC)           # Estimation  
    print ("Matrix mX=\n", mX, "\nof size ", iSize) # Output
```

Alternative: See below, altering pre-defined mutable (= matrix) argument

**Q: Why is this example rather stupid/non-robust?**

**A: Rather use `mX.size`, no space for errors**

## Indexing

A matrix is a NumPy array of multiple doubles, a string consists of multiple characters, a list of multiple elements. Get to those elements by using indices (starting at 0):

### Listing 8: recap3.py

```
def index(mA, sB, lC):
    print ("Element [0,1] of\n", mA, "\nis %g" % mA[0,1])
    print ("Elements [0:5] of '%s' are '%s'" % (sB, sB[0:5]))
    print ("Element [4] of '%s' is letter '%s'" % (sB, sB[4]))
    print ("Element [1] of\n", lC, "\nis '%s'" % lC[1])

#####
### main
def main():
    mX= np.random.randn(2, 3)      # Some random numbers
    sY= 'Hello world'              # A string
    lZ= [mX, sY, 6.3]              # A list of items
    index(mX, sY, lZ)
```

### Warnings:

- ▶ Indexing starts at [0] (as in C, Java, Julia, Ox etc, fine)
- ▶ Selecting a range indicates [start:end+1]... **Extremely dangerous, if you use other languages... And ugly, according to Prof E.W. Dijkstra**

## Indexing matrices

Python indexes 'logically'..., but sometimes counterintuitively.

- ▶ A matrix is effectively an array of an array
- ▶ A one-dimensional array can (often) be used as both row/column vector, `vX1d= np.array([1,2,3])`.
- ▶ Though sometimes an explicitly *two*-dimensional array is more useful, `vX2d= np.array([1, 2, 3]).reshape(-1, 1)` (depends on the situation, be careful)
- ▶ But then check the difference between `vX1d[0]`, `vX2d[0]`, `vX2d[0,0]`, `vX2d[0:1]` and `vX2d[0:1,0]`

See `recap4.py`...

## Indexing matrices II

### Listing 9: recap4.py

```
import numpy as np

#####
### main
def main():
    vX= np.array([1, 2, 3]).reshape(-1, 1)    # A column vector

    print ("vX=\n", vX)
    print ("Note how vX is a lists-of-lists, cast to a two-dimensional array\n")

    print ("vX[0]= ", vX[0], "(a one-dimensional array)")
    print ("vX[0,0]= ", vX[0,0], "(a scalar)")
    print ("vX[0:1]= ", vX[0:1], "(a 1 x 1 matrix)")
#####
### start main
if __name__ == "__main__":
    main()
```

## Stepwise Indexing

An index may also take a step:

Listing 10: recap4b.py

```
import numpy as np

#####
### main
def main():
    vX= np.random.randn(10)

    print ("Full vX:\n", vX)
    print ("Every second element:\n", vX[::2])
    print ("Every second element, starting at second:\n", vX[1::2])
```

Convenient for selecting subsets!

## Boolean Indexing

One can also index using (a vector of) booleans, to select only the rows/columns/elements where the boolean is True:

### Listing 11: recap4c.py

```
import numpy as np

#####
### main
def main():
    vX= np.random.randn(10)
    vI= vX >= 0

    print ('vX:', vX)
    print ('vI:', vI)

    vXp= vX[vI]
    print ("Non-negative elements:\n", vXp)
    print ("(Careful with resulting type/size!)")
```

Convenient for selecting subsets!



## Matrices

A matrix:

- ▶ ... is the work-horse of most econometric work (data, linear algebra, likelihoods and derivatives etc)
- ▶ ... is not natively included in Python
- ▶ ... hence we'll take the numpy array instead
- ▶ (Note: We'll choose not to use the numpy matrix)
- ▶ Matrices tend to be two-dimensional
- ▶ ... hence we'll often force our matrices/vectors into such shape:

```
vX= [1, 2, 3]           # A one-dimensional list
vX= np.array(vX)        # ... transformed into a one-dimensional array
vX= vX.reshape(3, 1)    # ... and made into a two-dimensional matrix
vX= vX.reshape(-1, 1)   # ... same thing (or more robust), Python checks r
```

- ▶ Important: Check your matrices, make sure you distinguish matrix/one-dimensional array/scalar!

## Matrices II

Matrices can be used, after starting with e.g. `mX= np.random.randn(3, 4)`,

- ▶ as *arguments* of functions: `dSum= np.sum(mX)`
- ▶ or applying a function on a matrix directly, `dSum= mX.sum()`;  
`vSum= mX.sum(axis=0)`; `vX= mX.reshape(1, -1)`
- ▶ looking at its *characteristics*, `(iR, iC)= mX.shape`
- ▶ changing its characteristics even: `mX.shape= (1, iR*iC)`

(see `recap4d.py`)

**Q:** What is difference between `dSum` and `vSum`?

## Matrices II

Matrices can be used, after starting with e.g. `mX= np.random.randn(3, 4)`,

- ▶ as *arguments* of functions: `dSum= np.sum(mX)`
- ▶ or applying a function on a matrix directly, `dSum= mX.sum()`;  
`vSum= mX.sum(axis=0)`; `vX= mX.reshape(1, -1)`
- ▶ looking at its *characteristics*, `(iR, iC)= mX.shape`
- ▶ changing its characteristics even: `mX.shape= (1, iR*iC)`

(see `recap4d.py`)

**Q:** What is difference between `dSum` and `vSum`?

**Hint:** Always, *always* keep track of what your matrix is, and check yourself...

## Indexing and non-matrices

There is more than matrices...

- Strings, lists, ...

### Listing 12: recap5.py

```
def showelement(sElem, aElem):
    print (sElem, "=", aElem, " with type ", type(aElem),
          " with shape ", np.shape(aElem), ", size ", np.size(aElem),
          " and len ", len(aElem))

def main():
    lX= [[1, 2, 'hello'],
         ['there', "A", 4.5]]
    print ("Show the full list:")
    showelement("lX", lX)           # a two-dimensional list
    print ("Reference first list:")
    showelement("lX[0]", lX[0])     # a one-dimensional list
    print ("Reference the third element [2] of the first list lX[0]:")
    showelement("lX[0][2]", lX[0][2]) # a string

    print ("It would be incorrect to reference lX[0,2]")
    # showelement("lX[0,2]", lX[0,2]) # an error...
```

Q1: How do I get 'here' by referencing a part of lX?

Q2: What is difference in np.shape(), np.size(), len()?

## Scope

Each variable has a *scope*, a part of the program where it is known. The scope is either

- ▶ **local**: The variable is known within the present function only
- ▶ **global**: ...

### Listing 13: recap6.py

```
def localfunc(aX):  
    sX= "local var"  
    print ("In localfunc: Local arg aX: ", aX)  
    print ("In localfunc: Local var sX: ", sX)  
    # Next line gives an error  
    # print ("Double dY: ", dY)  
  
def main():  
    dY= 5.5  
    localfunc("a variable from main")  
    print ("In main: Double dY= ", dY)  
    # Next line gives an error  
    # print ("In main: sX= ", sX)
```

**Q:** What variable is known where exactly?

## Scope II

Each function (including main)

- ▶ can create/use at will new **local** variables
- ▶ can receive through arguments variables from other functions

Additionally, each function can

- ▶ share a **global** variable
- ▶ where the **global** variable shall be prefixed by **g\_**, as in **g\_mX**
- ▶ ... where the variable is declared `global` within a function, before its use, see `recap7.py`

## Scope III

### Listing 14: recap7.py

```
#####  
### localfunc(iX)  
def localfunc(iX):  
    global g_lX  
    print ("In localfunc: argument iX: ", iX)  
    print ("In localfunc: g_lX: ", g_lX)  
  
    g_lX[1]= iX          # Change a single element in global  
    print ("In localfunc: g_lX after changing an element: ", g_lX)  
  
    g_lX= list(range(iX, 2*iX)) # Change the full variable  
    print ("In localfunc: g_lX, after changing all: ", g_lX)  
  
#####  
### main  
def main():  
    global g_lX  
  
    iY= 5  
    g_lX= [1, 2, 3]  
    localfunc(iY)  
    print ("In main: Global var= ", g_lX)
```

## Scope IV

Each function (including main)

- ▶ can create/use at will new **local** variables
- ▶ can receive through arguments variables from other functions
- ▶ can use **global** variables (but please **forget** them...)

Additionally, each function can

- ▶ change *part* of the *mutable* variable (list/array/matrix) ...

Then *the variable does not change*, only part of the *contents*

[Example: See `recap8.py` below]



## Function arguments

In Python, functions can alter **contents** of variables, but **not** the full variable itself:

Listing 15: recap8.py

```
def func_nochange(mX):  
    mX= np.random.randn(3, 4)  
    print ("In func_nochange, changing mX locally to mX=\n", mX)  
  
def func_change(mX):  
    iR, iC= mX.shape  
    mX[:, :]= np.random.randn(iR, iC)  
    print ("In func_change, changing mX locally to mX=\n", mX)  
  
def main():  
    mX= np.array([[1.0,2,3],[4,5,6]])  
    func_nochange(mX)  
    print ("In main, after func_nochange: mX=\n", mX)  
    func_change(mX)  
    print ("In main, after func_change: mX=\n", mX)
```

## Function arguments II

Limitations: Changing function arguments

- ▶ works with *mutable* variables (i.e. lists, arrays, NumPy matrices, Pandas dataframes),
- ▶ does not work with *immutable* variables (i.e. strings, tuples, doubles, integers)
- ▶ allows for changes in value, (generally (...)) not in size of argument
- ▶ which implies that arguments have to be pre-assigned at the correct size

Example:

Listing 16: e0\_elim.py

```
def ElimElement(mC, i, j):  
    ...  
    mC[i,j:] = mC[i,j:] - dF*mC[j,j:]  
    return True
```

## Function arguments III

### Notes (**IMPORTANT**):

- ▶ If you are going to change an input argument to a function *MENTION IT IN THE DOCSTRING*, listing the variable under the Outputs
- ▶ General rule of thumb: A function argument can be changed when you assign to *a part of* the argument, as in `mC[1,2]=5`. The moment you do a full `mC= np.random.rand(3,4)` the full variable is overwritten, and the result is *not* available to the outside routine.
- ▶ Exception to size changing argument: In Pandas, you are allowed to extend an existing dataframe with additional columns.