

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 29, 2020

Overview

Principles of Programming in Econometrics

D0: Syntax, example 2⁸

D1: Structure, scope

D2: Numerics, packages

D3: Optimisation, speed

Day 1: Structure

9.30 Introduction

- ▶ Programming in theory
- ▶ Science, data, hypothesis, model, **estimation**

Structure & Blocks (Droste)

Further concepts of

- ▶ Data/Variables/Types
- ▶ Functions
- ▶ Scope, globals

13.30 Practical

- ▶ Regression: Simulate data
- ▶ Regression: Estimate model

What is programming about?

Managing DATA, in the form of VARIABLES, usually through a set of predefined FUNCTIONS or ACTIONS

Of central importance: Understand *variables*, *functions* at all times...

So let's exaggerate

Variable

- ▶ A *variable* is an item which can have a certain *value*.
- ▶ Each variable has *one* value at each point in time.
- ▶ The value is of a specific *type*.
- ▶ A program works by managing *variables*, changing the *values* until reaching a final *outcome*

[Example: Paper integer 5]

Integer

iX = 5



- ▶ An integer is a number without fractional part, in between -2^{31} and $2^{31} - 1$ (C/Ox/Matlab) or limitless (Python 3.X)
- ▶ Distinguish between the *name* and *value* of a variable.
- ▶ A variable can usually *change value*, but never *change its name*

Double

dX= 5.5



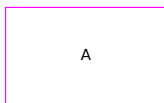
5.5

- ▶ A double (aka float) is a number with possibly a fractional part.
- ▶ Note that 5.0 is a double, while 5 is an integer.
- ▶ A computer is not 'exact', careful when comparing integers and doubles
- ▶ If you add a double to an integer, the result is double (in Python 3/Ox at least, language dependent)

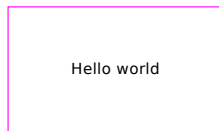
[Example: dAdd= 1/3; iD= 0; dD= iD + dAdd; type(dD)]

String

sX= 'A'



sY= 'Hello world'



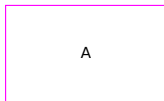
- ▶ A character is a string of length one.
- ▶ A string is a collection of characters.
- ▶ The ' are not part of the string, they are the *string delimiters*.
- ▶ One or multiple characters of a string are a string as well, sY[0:4], sY[1], sY[1:2] are strings.

[Example: sY= 'Hello world']

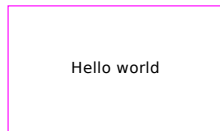
Q: Trick question: What is difference between sY[1] and sY[1:2]?

String

sX= 'A'



sY= 'Hello world'



- ▶ A character is a string of length one.
- ▶ A string is a collection of characters.
- ▶ The ' are not part of the string, they are the *string delimiters*.
- ▶ One or multiple characters of a string are a string as well, sY[0:4], sY[1], sY[1:2] are strings.

[Example: sY= 'Hello world']

Q: Trick question: What is difference between sY[1] and sY[1:2]?

A: Check sY[1] == sY[1:2]

‘Simple’ types

- ▶ Boolean
- ▶ Integer
- ▶ Double/float
- ▶ String

Check type using

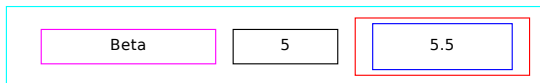
```
bX= True  
type(bX)
```

'Difficult' types

- ▶ List
- ▶ Tuple
- ▶ Matrix
- ▶ Function
- ▶ Lambda function
- ▶ DataFrame
- ▶ ...

List

`lX= ['Beta', 5, [5.5]]`

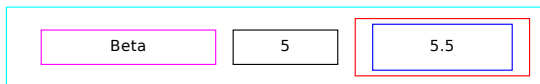


- ▶ A *list* is a collection of *other objects*.
- ▶ A list itself has one *dimension*, but can contain lists.
- ▶ An element of a list can be of any type (integer, double, function, matrix, list etc)
- ▶ A list of a list of a list has *three* dimensions etc.
- ▶ One may replace elements of a list (*a list is mutable*)

[Example: `lX= ['Beta', 5, [5.5]]`; `lX[0]= 'Alpha'`]

Tuple

tX= ('Beta', 5, [5.5])



- ▶ A *tuple* is a collection of *other objects*.
- ▶ A tuple itself has one *dimension*, but can contain lists.
- ▶ An element of a tuple can be of any type (integer, double, function, matrix, list, tuple etc)
- ▶ A tuple of a tuple of a tuple has *three* dimensions etc.
- ▶ One may **NOT** replace elements of a tuple (*a tuple is immutable*)

[Example:

```
tX= ('Beta', 5, [5.5]); # Error: tX[0]= 'Alpha' ]
```

Matrix

```
mX= np.array([[1.0, 2, 3], [4, 5, 6]])
```

1.0	2.0	3.0
4.0	5.0	6.0

- ▶ A *matrix* (to an Econometrician at least) is a collection of *doubles*; in Python a matrix may also contain other types.
- ▶ A matrix has (generally) two *dimensions*.
- ▶ A matrix of size $k \times 1$ or $1 \times k$ we tend to call a *vector*, vX
- ▶ Watch out: NumPy allows single-dimensional k vectors, different from $k \times 1$ matrices.
- ▶ Later on we'll see how matrix operations can simplify/speed up calculations.

Matrix II

```
mX= np.array([[1.0, 2, 3], [4, 5, 6]])
```

1.0	2.0	3.0
4.0	5.0	6.0

In Python:

- ▶ we'll use a list-of-lists as input into a NumPy array
- ▶ ensure we have doubles by making at least one of the entries a double (here: 1.0), `type(mX[1,2])`, or use
`mX= np.array([[1,2,3], [4, 5, 6]]).astype(float)`
- ▶ if needed force it into a 2-dimensional shape,
`mX.shape= (6, 1)`

[Example: `mX= np.array([[1.0, 2, 3], [4, 5, 6]])`]

Function

`print ("Hello world")`



`print()`

- ▶ A *function* performs a certain task, usually on a (number of) variables
- ▶ Hopefully the name of the function helps you to understand its task
- ▶ You can assign a function to a variable,
`fnMyPrintFunction= print`

[Example: `fnMyPrintFunction('Hello world')`]

Function II

Listing 1: pow6.py

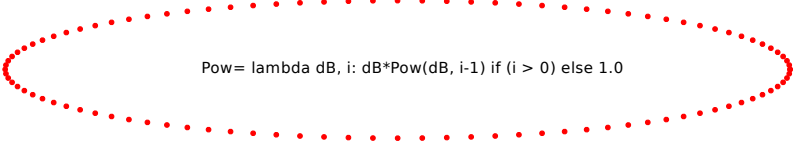
```
def Pow(dBase, iPow):  
    dRes= 1  
    i= 0  
    while (i < iPow):  
        # print ( 'i= ', i )  
        dRes= dRes * dBase  
        i+= 1  
    return dRes
```

- ▶ You can define your own routines/functions
- ▶ You decide the output
- ▶ You tend to return the output
- ▶ (later: You may alter mutable arguments)

[Example: dPow= Pow(2.0, 8)]

Lambda Function

Pow(2.0, 8)



Pow= lambda dB, i: dB*Pow(dB, i-1) if (i > 0) else 1.0

- ▶ A *lambda function* is a single line locally declared function
- ▶ It can access the present value of variables in the *scope*
- ▶ Hence it can *hide* passing of variables
- ▶ More details in the last lecture, when useful for optimising
- ▶ Syntax:
name= **lambda** arguments: expression(arguments)

Listing 2: pow_lambda.py

```
Pow= lambda dB,i: dB*Pow(dB,i-1) if (i > 0) else 1.0  
dPow= Pow(2.0, 8)
```

List comprehension

Alternative to a *Lambda* function can be a *list comprehension*, in certain cases. A *list comprehension*

- ▶ applies a function successively on all items in a list
- ▶ and returns the list of results

Structure:

```
List = [ func(i) for i in somelist]
```

Examples:

```
[i for i in range (10)]  
[i for i in range (10) if i%2 == 0]  
[i**2 for i in range(10)]  
[np.sqrt(mS2[i,i]) for i in range(iK)]
```

Q: Can you predict the outcome of each of these statements?

DataFrame

- ▶ A **Pandas** *dataframe* is an object made for input/output of data
- ▶ It can be used to read/store/show your data
- ▶ And has plenty more options
- ▶ Very useful for data handling!

```
[ Example: import pandas as pd; lc= list('ABC');  
df= pd.DataFrame(np.random.randn(4,3), columns=lc); df ]
```

DataFrame II

Listing 3: stackols.py

```
sData= 'data/stackloss.csv'
sY= 'Air Flow'
asX= ['Water Temperature', 'Acid Concentration', 'Stack Loss']

# Initialisation
df= pd.read_csv(sData)          # Read csv into dataframe
vY= df[sY].values               # Extract y-variable
mX= df[asX].values              # Extract x-variables
iN= vY.size                     # Check number of observations
mX= np.hstack([np.ones((iN, 1)), mX]) # Append a vector of 1s
asX= ['constant']+asX

# Estimation
vBeta= np.linalg.lstsq(mX, vY)[0] # Run OLS  $y = X \beta + e$ 

# Output
print ('Ols estimates')
print (pd.DataFrame(vBeta, index=asX, columns=['beta']))
```