

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

Elements to consider

- ▶ Comments: # (until end of line)
- ▶ Docstring: """ Docstring """
- ▶ import statements: At front of each code file
- ▶ Spacing: Important for routines/loops/conditional statements
- ▶ Variables, types and naming (subset):

boolean	bX=True
scalar integer	iN= 20
scalar double/float	dC= 4.5
string	sName='Beta1'
list	lX= [1, 2, 3], lY= ['Hello', 2, True]
tuple	tX= (1, 2, 3)
vector	vX= np.array([1, 2, 3, 4])
matrix	mX= np.array([[1, 2.5], [3, 4]])
function	fnFunc = print

Elements: Comments

Use: # (until end of line)

- ▶ To explain reasoning behind code
- ▶ ...but sparingly: Code should be self-explanatory(?)
- ▶ ...while maintaining readability: Will you, or someone else, understand after three years/months?
- ▶ ...Hence use for quick additions to code
- ▶ **and** ...for temporarily turning off parts of the code (e.g., checks?)

Important, very...

Elements: Docstrings

Use:

- ▶ To explain the functions/modules you write
- ▶ Either single-line
(`"""Return the iPow'th power of dBase."""`),
- ▶ or multi-line, after function definition:

```
def Pow_Recursion(dBase, iPow):  
    """  
    Purpose:  
        Calculate dBase^iPow through recursion  
  
    Inputs:  
        dBase      double, base  
        iPow       integer, power  
  
    Return value:  
        dRes       double, dBase^iPow  
    """
```

- ▶ ... and at start of module, explaining
name/purpose/version/date/author

Important, indeed...

Elements: Docstrings II

IPython 6.1.0 -- An enhanced Interactive Python. Type '?' for help.

```
In [1]: run pow6
```

The result of $2^8 =$

- Using Pow(): 256
- Using Pow_Recursion(): 256
- Using **: 256
- Using math.pow: 256.0

```
In [2]: ?Pow_Recursion
```

Signature: Pow_Recursion(dBase, iPow)

Docstring:

Purpose:

Calculate $dBase^{iPow}$ through recursion

Inputs:

dBase	double, base
iPow	integer, power

Return value:

dRes	double, $dBase^{iPow}$
------	------------------------

File: ~/vu/ppectr18/lists_py/power/pow6.py

Type: function

Elements: Imagine variables

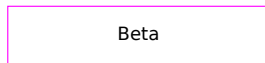
iX= 5



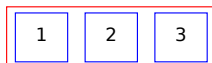
dX= 5.5



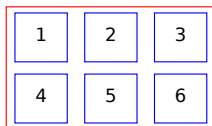
sX= 'Beta'



lX= [1, 2, 3]



mY= [[1, 2, 3], [4, 5, 6]]



Every element has its representation in memory — no magic

Try out variables

Listing 1: variables.py

```
bX= True
type(bX)

iN= 20
type(iN)

dC= 4.5
type(dC)

sX='Beta1'
type(sX)

lX= [1, 2, 3]
type(lX)

mY= [[1, 2, 3], [4, 5, 6]]
type(mY)

mZ= np.array(mY)
type(mZ)

fnX= print
type(fnX)

rX= range(4)
type(rX)
print ("Range rX= ", rX)
print ("List of contents of range rX= ", list(rX))
```

Hungarian notation prefixes

prefix	type	example
i	integer	iX
b	boolean	bX
d	double	dX
m	matrix	mX
v	vector	vX
s	string	sX
fn	Function	fnX
l	list	lX
g-	variable with global scope	g_mX

Use them *everywhere, always*.

Possible exception: Counters i, j, k etc.

Hungarian 2

Python does not force Hungarian notation. Why would you?

- ▶ Forces you to think: What should each object be?
- ▶ Improves readability of code
- ▶ Helps (tremendously) in debugging

Drawbacks:

- ▶ Python recognizes many different types; in 'EOR/QRM/PhD', not all are useful to track
- ▶ Hungarian notation best used for 'intention': vector vX for 1-dimensional list or array or a $n \times 1$ or $1 \times n$ matrix, matrix mX for 2-dimensional list/array

Hungarian 3

Correct but *very* ugly is

Listing 2: nohun.py

```
def main():  
    iX= 'Hello'  
    sX= 5
```

Instead, *always* use

Listing 3: hun.py

```
def main():  
    sX= 'Hello'  
    iX= 5
```