Lecture 2: Fundamentals of Programming
Programming and Numerical Methods for Economics—ECNM10115

Albert Rodriguez-Sala & Jacob Adenbaum
School of Economics, The University of Edinburgh

Winter 2024

This class

Basics of Python Syntax

Operators

Data Types

Loops

Conditional Statements

Functions

Basics of Python Syntax

Basic syntax in Python

- Each line in a Python script is a **statement**. Unless broken with \ or in lists.
- Naming convention in Python. **Identifier** is the name given to our programming elements (variables, functions, classes, modules, packages, etc.).
  - An **identifier should start** with either an alphabet letter or an underscore (_).
  - After that, more than one alphabet letter (a-z or A-Z), digits (0-9), or underscores may be used to form an identifier. No other characters are allowed.
  - Identifiers in Python are **case sensitive** variables *list*_1 and *List*_1 are different.
- **Display Output**: the print() function serves as an output statement in Python.

Indexing and slicing in Python

Objects in Python can be non-sliceable or sliceable. **Sliceable objects**—as strings, lists, tuples,arrays, dataframes, etc—are objects containing *indexed elements*—as cells in an array, elements in a list, columns/rows in a dataframe.

Caution: indexing in Python starts with zero!!!

- First element [0], second element[1], nth element [n-1].
- In a "2-D" object: [0,:] first set of elements, like the first row in an array or a dataframe. [0,0] first element—first row, first column. [:,0] first column.
- To recover last elements we use the minus: last element[-1], previous last element [-2], [:,-1] entire last column.
- **Slicing**: to recover closed intervals $[i, i+n]$ on x we use the syntax $x[i-1 : i+n]$. From the 2nd to the 4th element: [1:4]. From the 4th element to the last one [3:]. Last 2 rows, first three columns [-2:, 0:3]

Python works with identation

- Python uses **whitespace** and **indentation** to construct the **code structure**. Thus, identation is the equivalent to the curly brackets $\{\}$ in Stata, R, and other programs.
- Identation helps preserve the order and readiness of the code. Identation also makes it faster to write codes and avoid missing $\}$ or $\{$.
- **Sypder has automated identation**. You should use the tab whenever you want to indent (each tab one indent).

Blocks of code

Identation rules to indicate blocks of code.

- Use the colon : to start a block and press Enter.
- All the lines in a block must use the **same indentation**, either 4 space or a tab.
- A block can have **inner blocks** with next-level indentation.

Example

```
# Import NumPy library to work with matrices/arrays.
import numpy as np
def log_u(c):  # first block (the function)
    if c<0.00000001: #1 indent
        u = -np.infty  # 2 indents (witihin an inner block)
    else:
        u = np.log(c)
    return u
```

Modules

Python has a relatively small core language supported by many modules or libraries. We import them using the **import** routine.

- NumPy: fundamental package for **scientific computing**. *import numpy as np*
- PanDas: fundamental library for **data manipulation and data analysis**. *import pandas as pd*
- SciPy: fundamental library for **numerical analysis**: optimization, integration, interpolation, etc. *import scipy.optimize as opt* or *from scipy.optimize import minimize*.[1]
- Matplotlib: main library for **plots** and visual representation. *import matplotlib.pyplot as plt*
- Other libraries we will use: seaborn—**statistical data visualization**—statsmodels—fundamental statistics and **econometrics** package—etc.

We can create our own modules! Any script we do is a module itself. In Python pretty much everything you want to do, there is a great module designed for that. Explore them!

[1] We can also import an specific function as in any of the libraries: *from numpy import log*

Some basic syntax

#	Write a comment
'blu', "blu"	We use " and "' to create strings.
"' text "'	Multiple lines string. Also can be used to write long comments.
.	To call a routine from a module or class module.object. object.method()
=	assign
\	Multiline. Continues code in the next line.
[]	Used for indexing, lookup, and slicing in referenced objects. Also to create lists.
()	To preserve order of operations. Used in functions or method calls. to create tuples.
{}	to create dictionaries and sets. In string formatting to indicate replacement fields.

Operators

Operators

Some of the main operators in Python are [2]

1. Arithmetic operators

+/-	Addition/substraction.
*	Multiplication (element-wise).
@	Matrix multiplication (in NumPy arrays).
/	Division.
//	Floor division.
**	Power.

[2] Not all objects in Python can use all the operators. Depending on the object type, some operators will work or not.

Operators

2. Comparison operators

==	equal conditional statement.
< / <=	Smaller than / equal or smaller than. (same for >).
!=	not equal.

3. Identical operators

is	True if both are true (they are equal).
is not	True if they are not equal

4. Belonging operators

in	True if element is present in the object.
not in	True if element is not present in the object.

Operators

5. Logical operators

and	True if both are true.
or	True if at least one is true.
not	returns true if an expression is false.

6. Bitwise operators[3]

&	bitwise **and** logic operator.
|	bitwise **or** logic operator.
$\sim$	invert operator (the opposite).[4]

[3] Bitwise is level of operation that involves working with individual bits which are the smallest units of data in a computing system.

[4] Ex: $\sim 6 = -6$. Select data from a dataset not in "data_bad", data[$\sim$data_bad].

Data Types

Data types in Python

The OOP paradigm: data and **functions** are **bundled together** into **objects**

- In the OOP functions are usually called methods.
- In Python the data and methods of an object are referred to as attributes. Depending on the class of object (like float, array, string, dataframe, etc) the object will have different methods.
- In the OOP functions are usually called methods.

Let's see the main data types in Python.

Variables and data types

A **variable** in Python is a *reference* to a place in memory where data resides. There are many types of variables. The most simple types are called **atomic variables**, because they cannot be changed, only overwritten—store data directly. another types of variables are containers for data—**containers**.

Atomic variables:

- Integers.
- Floats (decimal numbers).
- Booleans (True/False).
- Strings.

Containers:

- Lists.
- Arrays.
- Dictionaries.
- Tuples.
- Pandas Data Frames.
- etc.

All variables are objects: bundles of data and functions.

Integers (Int)

Python includes three numeric types to represent numbers: integers, float, and complex number.

Integers are zero, positive or negative whole numbers without a fractional part and having *unlimited precision*.

- Example:

```
y = 2
x= 2.0
type(y)  # >> int
type(x)  # >> float
```

- To convert a float to an integer:

```
z = int(x)
```

Floats

Floats—floating point numbers—are positive and negative real numbers with a fractional part. Floats have limited precision, computer cannot store infinite digits (like with π). By default, Python works with **double precison**—8 byte, 64 bit—**float64**. Double precision numbers are accurate up to **sixteen decimal places** (roughly).

- Example:

```
a = 2.4
b= 9.5
```

- **float**: to convert other atomic types to a float.

```
c = float(y)   #>>2.0
d= float('4')   #>>4.0
e = float(True) #>>1
```

- **NaNs**: Stands for Not a Number. Unfortunately, while programming we will deal a lot with nans. In many cases they indicate we are doing something wrong. EX: np.log(-1)= NaN.

Functions and operators for Int and Floats

Some useful built-in functions and operators for integers and floats.

- **abs**: returns the absolute value of a number without considering its sign.
- **round**: Returns the rounded number.
- **augmentation operators** (e.g. +=, -=, *=, /=)

```
x += 1 #equal to x = x+1
x *= 2 # equal to x = x*2
x /= 2 # equal to x = x/2
```

Boolean values (Bools)

Boolean values represent the *truth value* of an expression. Boolean values can either be True or False.

- Example:

```
x= True
y = 100 < 10
type(y) # >> bool
print(y) # False
```

- Booleans are considered a numeric type in Python. Thus, we can apply arithmetic operations to booleans and compare them to numbers.

```
print(x+y)    #>>1
print(x*y)    #>>0
bools = [True, True, False, True]
sum(bools) #>>3
```

- For numbers, **bool** is equivalent to x != 0.

```
bool(3) #>>True
```

Common Boolean operators

We will use operators as logical comparison, ans identity operators with Boolean values. The likes of: as or, and, not, == , != , is, etc.

- Example:

```
z = 100>10
x and (z or y)    #>> True
(x or z) and y    #>> False
```

Strings

A **string** is an *immutable* sequence of Unicode characters wrapped inside single, double, or triple quotes.

- Example:

```
str1='This is a string in Python'
```

- **str**: to convert objects (as integers or floats) into strings

```
str2 = str(4.29)
```

- Since a string is a sequence, it is an *ordered collection* of items. Thus, strings are **sliceable**:

```
str1[0]   #>>T
str1[6]    #>>s
str1[10:16]   #>>string
```

- A string is an immutable object:

```
Str1[6] = '3' # >> Error
```

Some string operators

+	Appends the second string to the first.
*	Concatenates multiple copies of the same string.
in	Returns true if a character exists in the given string.

Common methods for strings

string.capitalize()	Returns the copy of the string with its first character capitalized
string.lower()	Returns a lowered case string.
string.upper()	Returns a string in the upper case.
string.count(substring)	Search the specified substring in the given string and returns an Integer indicating occurrences of the substring.
string.find(substring)	Returns the index of the first occurrence of a substring in the given string.
string.index(substring)	Returns the index of the first occurrence of a substring in the given string.

Containers

A **container** is an object, which consists of several objects, for instance atomic types. Therefore, containers are also called *collection types*.

Types of containers:

- Lists
- Arrays
- Dictionaries
- Tuples
- Pandas Data Frames
- etc.

.

Lists

A **list** is a *mutable sequence* type. A list object contains one or more items of *different data types* in the square brackets [] separated by a comma.

- Main virtue of lists: Its versatility. List of dataframes. List of lists of lists, lists of arrays, strings, etc.

```
list_1 = [1,2,3]
list_obj = [1,2,3,'a','b', y]
list_obj2 = [[1,2,3,], ['a','b']]
list_2 = ['hello']
list_2 = list('hello')
```

- A list is **sliceable** (indexed):

```
list_obj[0]
list_obj[1:3]
list_obj2[0][0]
```

Lists are mutable

A list is **mutable**, we can change its elements.

- Replace element *list_obj[0] = np.ones(5)*
- **Add** element: *list_1*.**append***(4)*
- **Remove** element: *list_1*.**remove***(2)*

Common methods in lists

list.append()	Adds a new item at the end of the list.
list.clear()	Removes all the items from the list and make it empty.
list.copy()	Returns a copy of a list.
list.count()	Returns the number of times an element occurs in the list.
list.extend()	Adds all the items of the specified iterable to the end of the list.
list.index()	Returns the index position of the first occurrence of the specified item.
list.insert()	Inserts an item at a given position.
list.remove()	Removes the first occurrence of the specified item from the list.
list.reverse()	Reverses the index positions of the elements in the list.
list.sort()	Sorts the list items in ascending, descending, or in a custom order.

Arrays (NumPy library)

NumPy **arrays** are somewhat like native Python lists, except that *data must be homogeneous* and *numerical*. These types must be one of the data types (types) provided by NumPy.

- The most important of these dtypes are:
  - float64: 64 bit floating-point number
  - int64: 64 bit integer
  - bool: 8 bit True or False
- Example:

```
import numpy as np
flat_a = np.array([1,2,3,4,5,6]) #1-D or flat array

flat_a[0]    # indexing works as in lists
flat_a[3:]
```

N-D Arrays

We can also create 2-D (or in general N-D) arrays.

- Example: a 2x3 Matrix.

```
a = np.array([[1,2,3],[4,5,6]])   # a matrix or 2-D array

# Indexing in 2-D arrays
a[0,1]
a[0,:]  #first row
a[:,1] # second column

#to get the dimensions
a.shape()   >>(2,3)
```

Creating arrays

np.zeros(N)	an array of N zeros.
Np.zeros((3,4))	2-d array (or matrix) of zeros with 3 rows and 4 columns.
Np.ones(N)	Array of N ones.
Np.linspace(a,b,N)	Create an even-spaced grid from a to b with N points.

```
#Example:
z= np.linspace(2,4,5) >>array([2,2.5,3.0,3.5,5])
```

Some main methods for NumPy arrays

array.sort()	Sort by ascending, descending or costumer order the order of the element
array.sum()	Sum all the elements in a given axis.
array.mean()	Return the mean along the given axis.
array.max()	Return the maximum element in the array.
array.argmax()/argmin()	Return indices of the maximum(minimum) values along the given axis.
array.T()	transpose
array.all()	Returns True if all elements evaluate to True.
array.any()	Returns True if any of the elements of a evaluate to True.
array.copy()	Return a copy of the array.
array.fill()	Fill the array with a scalar value.
array.flatten()	Return a copy of the array collapsed into one dimension.
array.reshape()	Returns an array containing the same data with a new shape.

For a more complete list of array methods see NumPy array documentation.

Example array methods

```
a = np.array([[1,2,3],[4,5,6]])   # a matrix or 2-D array

# Summing along axes
a.sum()      # sum all elements in the matrix
a.sum(axis=0) # sum per each column
a.sum(axis=1)  # sum per each row

# Mean
a.mean()       # matrix mean
a.mean(axis=0) # column-wise mean
a.mean(axis=1) # row-wise mean

# Applying comparisons
(a>3)        ## Element-wise array([[False, False, False, True, True, True]])
(a>3).any()   ## >>True
(a>3).all()   ## >> False

a.argmax()  # 5 element... but that is not very useful since matrices are indexed by rows, and columns
```

Tuples

A **tuple** is an *immutable* list. So, once a tuple is created, any operation that seeks to change its contents is not allowed.

- We create tuples with soft parenthesis, or just separating each element by a comma.

```
y= ('a','b')
y = 'a','b'
z=4, 5
```

- As with lists, tuples are sliceable: *z[0] >>4*. But tuples are immutable: *z[0]=10 >>error*.
- We use tuples to pass variables around that should not change by accident.
- Functions will output tuples if they have multiple output variables. Tuples can also be used as arguments to a function.

Dictionaries

Dictionaries are lists but ordered by names instead of numbers. Useful to create datasets, tables, etc.

- To create a dictionary we use curly brackets $\{\}$.
- **Keys:** all immutable objects are valid keys (eg. str or int). **Values:** fully unrestricted.
- **Example**

```
individual = {'name':'Francis', 'age':28, 'weight':100}
#name, age... are  the keys, Francis, 28... are the values.
individual['age']
>>28
type(individual)
```

Loops

Loops

The previous examples were scripts in which the commands are executed line-by-line only once. However, we often need more complex routines, for example:

- repeating many times the same operation.
- choosing different alternatives in different circumstances.
- iterating/repeating the same operation till some criterion is met (i.e. function is as its maximum point).

With **loop control statements**, you can **REPEATEDLY execute a block of code**.

Two types of loops: **for** and **while**.

For loops

for statement *performs a set of expressions* that depend on an **index** that is *updated with each iteration*—a specific number of times.

- Syntax: for **i** in **container**:
  statement(s) to repeat
  Where **i** is called the **iterator** or **counter** as it changes its value in every iteration of the loop.

```
#Example 1
for i in range(0,10):
    print(i) # #>> w 0 .. 1 .. 2 ...  9

#Example 2
for i in range(0,10):
    a = i**2
    print(a)   #>> 0 .. 1 .. 4 .... 81
```

more examples for loops

```
# Example 2: to store results, we can append them to an empty list:
squares = []
for i in range(0,10):
    a =i**2
    squares.append(a)
print(squares)   #>> [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

#Example 3
list_verbs = ['run','sit','jump','play','dance']
for verb in list_verbs:
    print('I '+verb)
>>I run
>>I sit
>>I jump
...
```

More for loops

- **Enumerated loops:** Looping through the content of a list.
  Syntax: For i,x in enumerate(mylist):
  statement(s) to repeat
  i is the counter (index), **x** is the ith element in the list.

```
#enumerate loop example
for i,verb in enumerate(list_verbs):
    print('Verb '+str(i+1)+': '+verb)  # remember that indexing starts with 0, so I ad
>> Verb 1: run .. Verb 2: sit ... Verb 5: dance
    ....
```

- **Zip loops:** We can **loop over 2 lists at the same time**. If you have two or more lists that you want to loop through simultaneously **in tandem**.

```
list_verbs_spanish = ['correr','sentarse','saltar','jugar','bailar']
#zip loop example
for v_eng, v_spa in zip(list_verbs, list_verbs_spanish):
    print(v_eng+': '+v_spa)
>> run: correr .. sit: sentarse .. jump: saltar .. play: jugar .. dance: bailar
```

Nested loops

Many times we are going to loop over a loop. This is a **nested loop, a loop inside a loop**. We refer to the loop inside as the **inner loop**, while the loop outside as the **outer loop**. Examples

```
x = [1, 2]
y = [4, 5]
for i in x:   # outer loop
  for j in y:  # Inner loop
    print(i, j)

>> 1 4   # first i, first j
   1 5    # first i, second j...
   2 4
   2 5
# note the order of the nested loop: For each i, apply j loop.
```

Note that we can do loops inside a loop inside a loop inside a loop, etc.

2nd Example nested loop

```
# Example 2: compute the multiplication table of 6 and 7.
for i in range(6, 8):
    print('Multiplication table of', i) # print inside outer loop.
    for j in range(1, 11):
        print(i, "*", j, "=", i*j) #print inside inner loop
```

```
>> Multiplication table of 6
6 * 1 = 6
6 * 2 = 12
....
6 * 10 = 60
Multiplication table of 7
7 * 1 = 7
7 * 2 = 14
....
7 * 10 = 70
```

While loop

A **while loop** executes a collection of commands (the block) until a condition is satisfied.

- Syntax: while condition:
  statement(s) to execute
- The statements will be repeatedly executed as long as the relation remains true.
- It is sometimes convenient to have an exit condition: if reached "100" iterations stop the loop. We can do so with the routine **break**.

```
#Example   1
count = 1
while count < 5:
    print(count)
    count = count + 1
>> 1 .. 2 .. 3 .. 4
```

Example while loops

Example 2: searching for a solution: suppose we have a simple equation $3 = \frac{x}{2}$

```
LHS = 3
eps = 0.001 # a small number
x=20 # after running it, then change x to 100
RHS =x/2
count=1
while np.abs(LHS-RHS)>eps:
    x -=1
    RHS = x/2
    count+=1
    if np.abs(LHS-RHS)<eps:
        print('solution is x=', x)
    elif count==20:
        print('no solution found with', count, 'iterations')
        break
    else:
        continue
>> solution is x= 6
```

Conditional Statements

Conditional statements

An **if statement**, evaluates an expression and executes a list of statements (block of code) if the expression is true.

If not, the condition in the if statement is wrong, we can branch off our program into another direction and assign different commands, an alternative code block—usually with the command **else**.

This allows us to branch our program into two (or more) separate possible directions.

We can have *multiple if conditions* with the **elif** command—stands for else if. if condition not satisfied, then checks elif condition. If elif condition not sastisified, check another elif or just else.

Example conditional statements

```
#Example 1: winning lotteries
import random
x = random.random()
print('Random Number is x=', x)
if (x > 0.3 and x < 0.5):
    print("You win a price of $1, congratulations!")
else:
    print("Sorry, you win nothing!")
```

Condtional statements and loops

We will typically combine conditional statements and loops.

```
# Example 2: Check numbers are even or odd
n = 7
while n > 0:
    # check even and odd
    if n % 2 == 0:    # % remainder of division
        print(n, 'is an even number')
    else:
        print(n, 'is an odd number')
    # decrease number by 1 in each iteration
    n = n - 1
>> 7 is an odd number
6 is an even number
5 is an odd number
....
```

Functions

Functions

Functions are extremely useful in programming. Besides others, we use functions for:

- to have mathematical functions that we can pass to algorithms—like minimization routines.
- when we want to repeat a procedure or operation. Code reusing.
- keep the code ordered and clean. Writing code in functions highly improves readability and minimizes errors. Better modularity.

Functions

A **Function** is a *block of organized, reusable code* that is used to *perform a single, related action*. Functions

- are **objects**.
- can have multiple (or no) arguments (inputs) and outputs.
- can have positional and keyword arguments.
- can use local or global variables (scope).

Functions

- Functions are very flexible on Python:
  - Any number of functions can be defined in a given file.
  - Functions can be defined inside other functions.
  - any object can be passed to a function as an argument, including functions.
  - A function can return any kind of object, including functions.
- Syntax: def myfunction(inputs):
  computations
  return output

Examples

```
# Example 1
def say_hi(name):
    return 'Hi '+name+'! We welcome you at the econ-programming course in the UoE

say_hi('Brad Pitt')
>>  'Hi Brad Pitt! We welcome you at the econ-programming course in the UoE.'

#Example 2
def f(x):
    return x**2

f(3)
>> 9
```

One-line functions

lambda is used to create one-line functions:

```
#Example 3
f= lambda x: x**2
f(3)
>>9

#Example 4 (from PSO): Pass function multiple arguments to single argument.
obj_func = lambda x1: -utility_function(x1,alpha,y,p1,p2)
```

Multiple inputs, multiple outputs functions

```
#Example 5: multiple inputs
def f(x,y):
    return x**2 + y**3

f(2,2)

# Example 6: multiple outputs: the output is a tuple.
def f(x,y):
   z = x**2
  q = y**3
   return z,q
res = f(2,2)
res[0]
res[1]
```

Keyword arguments

In general we have two types of arguments: **positional** arguments and **keyword** arguments. Keyword arguments are inputs with default values.

```
# Example 7: keyword arguments
def f(x,y,a=2,b=3):
    return x**a + y*b
f(2,2)
f(2,2,b=6)
f(2,2,a=6,b=3)
```

Local vs global variables

In terms of the scope, variables in functions can be either

- **Global** variables: the variable is declared outside the function. Global variables are visible and available to all statements in a setup script that follow its declaration.
- **Local** variables: the variable is declared inside the function. Local variables are visible and available only within the function where they are declared.
- **Recommendation:** don't use global variables inside your functions.

Local vs global variables

```
# example 8: global vs local variables
a = 2 # a global variable
def f(x):
   return x**a # a is GLOBAL. this is messy.
def g(x,a=4):
    # better: a as a default argument
    return x**a
def h(x):
    a = 4 # a is LOCAL variable
    return x**a
```