Lecture 0

Introduction to Python

Randall Romero Aguilar, PhD
II Semestre 2018
Last updated: July 12, 2018

Universidad de Costa Rica
SP6534 - Economía Computacional

Table of contents

Getting Python and CompEcon

About Python i

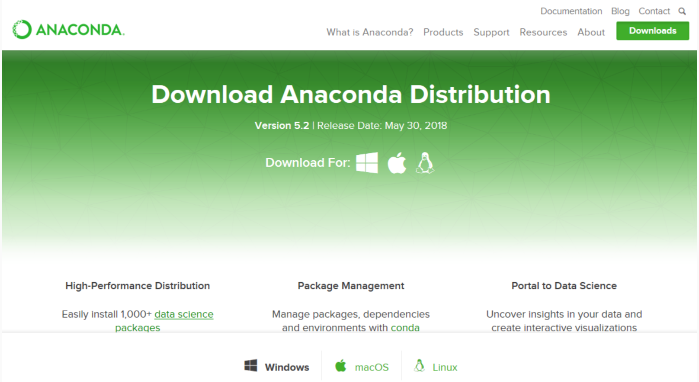
- Python is free – is open source distributable software
- Python is easy to learn – has a simple language syntax
- Python is easy to read – is uncluttered by punctuation
- Python is easy to maintain – is modular for simplicity
- Python is "batteries included" – provides a large standard library for easy integration into your own programs
- Python is interactive – has a terminal for debugging and testing snippets of code
- Python is portable – runs on a wide variety of hardware platforms and has the same interface on all platforms
- Python is interpreted – there is no compilation required

About Python ii

- Python is high-level – has automatic memory management
- Python is extensible – allows the addition of low-level modules to the interpreter for customization
- Python is versatile – supports both procedure-orientated programming and objectorientated programming (OOP)
- Python is flexible – can create console programs, windowed GUI (Graphical User Interface) applications, and CGI (Common Gateway Interface) scripts to process web data

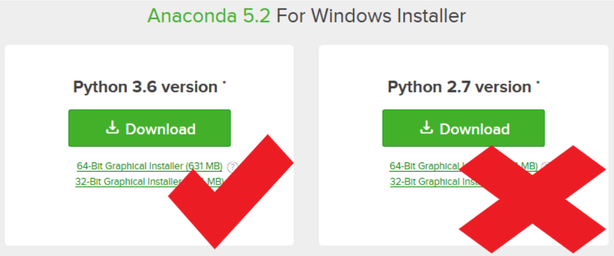
Downloading Python

The easiest way to get Python is to download it from Anaconda at https://www.anaconda.com/download/



Python 2 vs Python 3

- There are two major versions of Python: 2 vs 3
- Make sure to get the 3.6 version.



Installing CompEcon

- In this class we will use the CompEcon package.
- To install it, open the conda terminal and run this

```
(C:\ProgramData\Anaconda3) C:\Users>pip install compecon
```

- To verify that this was done correctly, run

```
(C:\ProgramData\Anaconda3) C:\Users>python -c "import compecon"
```

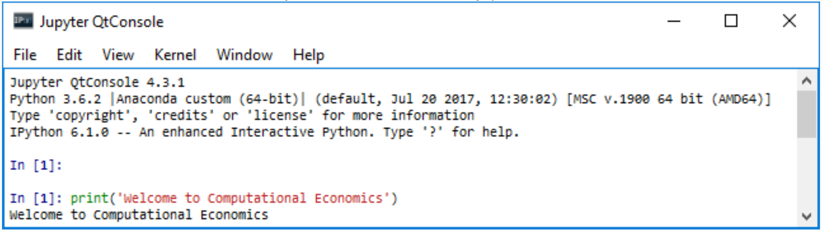
- After a few seconds, you should just get the prompt back.

Running Python

There are several ways to run Python code, among them

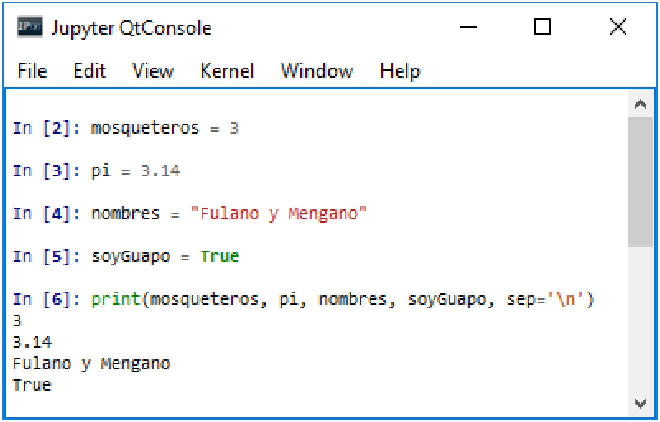
- in a terminal (command window) type python
- in Jupyter QtConsole
- in Jupyter Notebook
- in Spyder

For the first few examples, we use Jupyter QtConsole.



Python basics

- A variable is a container in which a data value can be stored within the computer’s memory.
- The stored value can then be referenced using the variable’s name.




```
In [2]: mosqueteros = 3

In [3]: pi = 3.14

In [4]: nombres = "Fulano y Mengano"

In [5]: soyGuapo = True

In [6]: print(mosqueteros, pi, nombres, soyGuapo, sep='\n')
3
3.14
Fulano y Mengano
True
```

Data types

There are four basic data types

int	integers
float	floating points (decimal numbers)
bool	boolean (True or False)
str	strings (text)

Jupyter QtConsole

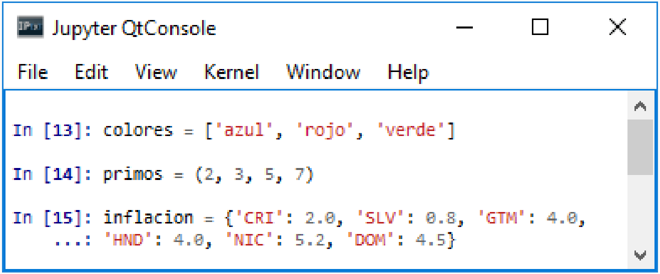
File Edit View Kernel Window Help

```
In [9]: type(mosqueteros)
Out[9]: int

In [10]: type(nombres)
Out[10]: str

In [11]: type(pi)
Out[11]: float

In [12]: type(soyGuapo)
Out[12]: bool
```

Collection types

The most commonly used collection typer are

- **list** an ordered, mutable list of values
- **tuple** an ordered, unmutable list of values
- **set** a unordered, mutable list of unique values
- **dict** an unordered dictionary



Arithmetic operators

Operator	Operation	Example	Result
+	Addition	2 + 3	5
-	Subtraction	5 - 1.0	4.0
*	Multiplication	4 * 4	16
/	Division	9 / 3	3.0
%	Modulus	10 % 3	1
//	Floor division	10 // 3	3
**	Exponent	5 ** 2	25

Defining multiple variables

In Python it is possible to define several variables in a single statement

```
n, a, b = 12, -2.0, 2.0
x = y = z = 1
```

Assigning values

Operator	Example	Equivalent
`=`	`a = b`	`a = b`
`+=`	`a += b`	`a = ( a + b )`
`-=`	`a -= b`	`a = ( a - b )`
`*=`	`a *= b`	`a = ( a * b )`
`/=`	`a /= b`	`a = ( a / b )`
%=	a %= b	a = (a % b)
`//=`	`a //= b`	`a = ( a // b )`
`**=`	`a **= b`	`a = ( a ** b )`

Comparing values

Operator	Comparative test	Example	Result
`==`	Equality	`5 == 5.0`	`True`
`!=`	Inequality	`4 != 4.0`	`False`
`>`	Greater than	`5 > 4`	`True`
`<`	Less than	`5 < 4`	`False`
`>=`	Greater than or equal to	`4 >= 4`	`True`
`<=`	Less than or equal to	`5 <= 5`	`True`

Assessing logic

Operator	Operation	Example	Result
`and`	Logical AND	`1 > 2 and 1 < 4`	`False`
`or`	Logical OR	`1 > 2 or 1 < 4`	`True`
`not`	Logical NOT	`not (5 > 4)`	`False`

Examining conditions

```
ifTrueThis if testExpression else ifFalseThis
```

For example, to pick the smallest number from a pair

```
smallest =  a if a < b else b
```

Casting data types

Function	Description
`int(x)`	Converts x to an integer whole number
`float(x)`	Converts x to a floating-point number
`str(x)`	Converts x to a string representation

Some examples

```
'8' + '4'                        # '84'
int('8') + int('4')              # 12
float('8') + float('4')          # 12.0
str(8) + str(4)                  # '84'
```

Making lists

To make a list, enumerate its elements within a pair of "[ ]"

```
seasons = ['Spring', 'Summer','Autumn','Winter']
```

seasons =	'Spring'	'Summer'	'Autumn'	'Winter'
	0	1	2	3

To access data:

```
seasons[2]      # 'Autumn'
seasons[-3]     # 'Summer'
```

To modify data:

```
seasons[2] = 'Fall'
```

Slicing

Lists can have heterogeneous data elements

```
mylist = [4, 3.0, 'abc', 5, 8, -3, 0 , 2]
```

To access a slice of data:

```
mylist[2:4]   # ['abc', 5]
mylist[:3]    # [4, 3.0, 'abc']
mylist[5:]    # [-3, 0 , 2]
mylist[-2:]   # [0 , 2]
mylist[3:4]   # [5]
mylist[::2]   # [4, 'abc', 8, 0]
mylist[1::2]  # [3.0, 5, -3, 2]
```

Manipulating lists

List Method	Description
list.append(x)	Adds item x to the end of the list
list.extend(L)	Adds all items in list L to the end of the list
list.insert(i,x)	Inserts item x at index position i
list.remove(x)	Removes first item x from the list
list.pop(i)	Removes item at index position i and returns it
list.index(x)	Returns the index position in the list of first item x
list.count(x)	Returns the number of times x appears in the list
list.sort()	Sort all list items, in place
list.reverse()	Reverse all list items, in place

Tuples

A tuple is similar to a list, but once defined its content cannot be changed. It is defined by enumerating its elements within a pair of “( )”

```
seasons = ['Spring', 'Summer','Autumn','Winter']
```

seasons =	'Spring'	'Summer'	'Autumn'	'Winter'
	0	1	2	3

To access data:

```
seasons[2]      # 'Autumn'
seasons[-3]     # 'Summer'
```

Slicing

Tuples support slicing too

```
M = ('Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', \
     'Jul','Aug','Sep','Oct','Nov','Dec')
```

To split the months into quarters:

```
Q1, Q2, Q3, Q4 = M[:3], M[3:6], M[6:9], M[9:]
```

Then, for example, typing Q2 returns

```
('Apr', 'May', 'Jun')
```

Sets

Set Method	Description
set.add(x)	Adds item x to the set
set.update([x,y,z])	Adds multiple items to the set
set.copy()	Returns a copy of the set
set.pop()	Removes one random item from the set
set.discard(x)	Removes item x from set if it's a member
set1.union(set2)	Returns items that appear in either set
set1.intersection(set2)	Returns items that appear in both sets
set1.difference(set2)	Returns items in set1 but not in set2
set1.isdisjoint(set2)	True if sets have no items in common

Sets: some examples

To make a set, enumerate its elements within “{ }”

```
M2 = {2, 4, 6, 8, 10, 12, 14}
M3 = {3, 6, 9, 12, 15}

M2.difference(M3)   # {2, 4, 8, 10, 14}
M3.difference(M2)   # {3, 9, 15}
M2.intersection(M3) # {6, 12}
M2.union(M3) # {2, 3, 4, 6, 8, 9, 10, 12, 14, 15}

M3.update([18, 21]) #  {3, 6, 9, 12, 15, 18, 21}
M2.add(16)  # {2, 4, 6, 8, 10, 12, 14, 16}
M2.isdisjoint(M3)  # False
```

Dictionaries

- In Python programming a “dictionary” is a data container that can store multiple items of data as a list of key:value pairs.
- Unlike regular list container values, which are referenced by their index number, values stored in dictionaries are referenced by their associated key.
- The key must be unique within that dictionary and is typically a string name although numbers may be used.

Dictionaries (example)

```python
king = {'name': 'John Snow',
        'age': 24,
        'home': 'Winterfell'}

friend = dict(name='Samwell Tarly', age=22)

king['age']  # 24
king['home'] = 'Castle Black'
king['lover'] = 'Ygritte'
king['knows'] = None
del king['lover'] # killed by Olly!
king['lover'] = 'Daenerys Targaryen'
```

Execution control

Execution control

- Scripts are usually executed by running every statement in the order they appear
- Some times, we need to execute some statements in other ways.
- For this, we use execution control statements:

  `if`, `elif`, `else` execute some statements once, only if certain condition is true

  `while` execute some statements several times, only while certain condition is true

  `for` execute some statements several times, iterating over a list

  `continue` jump to the next iteration of a **`while`** or **`for`** loop

  `break` stop execution of a **`while`** or **`for`** loop

Conditional branching with if

- The Python `if` keyword performs evaluates a given expression for a Boolean value of `True` or `False`.
- This allows a program to proceed in different directions according to the result of the test.
- The tested expression must be followed by a : colon, then statements to be executed when the test succeeds should follow below on separate lines and each line must be indented from the if test line.
- The size of the indentation is not important but it must be the same for each line.
- So the syntax looks like this:

```
if test-expression :
    statements-to-execute-when-test-expression-is-True
    statements-to-execute-when-test-expression-is-True
```

if: example

To determine if a number m is even or odd:

```
if m % 2 == 0:
    print('m is even')
else:
    print('m is odd')
```

The test does not necessarily have to be a boolean. The number 0,the value None, and an empty string '', list [], tuple () or set {}, are all interpreted as False.

```
if m % 5:
    print('m is not divisible by 5')
else:
    print('m is divisible by 5')
```

Looping while true i

- A loop is a piece of code in a program that automatically repeats.
- One complete execution of all statements within a loop is called an “ iteration” or a “pass”.
- The length of the loop is controlled by a conditional test made within the loop.
- While the tested expression is found to be True the loop will continue – until the tested expression is found to be False, at which point the loop ends.

Looping while true ii

- In Python programming, the **while** keyword creates a loop. It is followed by the test expression then a : colon character.
- Statements to be executed when the test succeeds should follow below on separate lines and each line must be indented the same space from the while test line.
- This statement block must include a statement that will at some point change the result of the test expression evaluation – otherwise an infinite loop is created.
- So the syntax looks like this:

```
while test-expression :
    statements-to-execute-when-test-expression-is-True
    statements-to-execute-when-test-expression-is-True
```

while: example

To get the Fibonacci series up to 100

```python
a, b = 0, 1
while b < 100:
    print(b, end=', ')
    a, b = b, a + b
```

```
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89,
```

A different approach

```python
fib = [1, 1]
while fib[-1]<100:
    fib.append(fib[-2] + fib[-1])
```

```
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144]
```

The range function

Sometimes we need to iterate over the integers. We can generate then using the `range` function.

```
range(6)        # 0, 1, 2, 3, 4, 5
range(2,8)      # 2, 3, 4, 5, 6, 7
range(2,9,3)    # 2, 5, 8
range(4, 0,-1) # 4, 3, 2, 1
```

Looping over items of an iterable

- In Python programming the **`for`** keyword loops over all items in any iterable specified to the **`in`** keyword.
- The syntax looks like this:

```
for item in iterable :
    statements-to-execute-on-each-iteration
    statements-to-execute-on-each-iteration
```

- Examples of iterables:
    - ranges
    - lists, tuples, sets, dictionaries
    - strings
    - text files
    - numpy arrays

`for ... in ...`: examples

- Iterating over strings

```python
for letter in 'abcd':
    print(letter.upper(), end=' ')
```

```
A B C D
```

- Iterating over strings

```python
for k in range(6):
    print(k**2, end=' ')
```

```
0 1 4 9 16 25
```

`for ... in ...`: more examples

- To keep track of the iteration number use `enumerate`

```
for i, letter in enumerate('abcd'):
    print(f'{i} = {letter}', end=' | ')
```

```
0 = a | 1 = b | 2 = c | 3 = d |
```

- To iterate over two iterables in parallel, use `zip`

```
quantities = [3, 2, 4]
fruits = ('apple','banana','coconut')
for n, fruit in zip(quantities,fruits):
    print(f'{n} {fruit}s')
```

```
3 apples
2 bananas
4 coconuts
```

List comprehensions

- Oftentimes we need to make a list of elements satisfying certain condition, with code like this

```
lst = list()
for item in iterable:
    if conditional:
        lst.append(expression)
```

- This is done more succinctly by

```
lst =  [expression for item in iterable if conditional]
```

List comprehensions: examples

- To generate the squares of even numbers less than 12

```
[k**2 for k in range(12) if k%2==0]
```

```
[0, 4, 16, 36, 64, 100]
```

- To count the number of letters in a list of words

```
food = ['apple','banana','carrot','grape']
[len(item) for item in food]
```

```
[5, 6, 6, 5]
```

Importing modules

Importing modules

- Python function definitions can be stored in one or more separate files for easier maintenance and to allow them to be used in several programs without copying the definitions into each one.
- Each file storing function definitions is called a “module” and the module name is the file name without the “.py” extension.
- Functions stored in the module are made available to a program using the Python **`import`** keyword followed by the module name.
- Although not essential, it is customary to put any import statements at the beginning of the program.
- Imported functions can be called using their name dot-suffixed after the module name. For example, a “sqrt” function from an imported module named “numpy” can be called with `numpy.sqrt()`

Some very useful modules

In our course, the following packages (= collection of modules) will be very useful

numpy Base N-dimensional array package. Math operations, especially linear algebra

matplotlib Comprehensive 2D plotting

pandas Data structures and analysis

scipy Fundamental library for scientific computing

compecon To solve computational economics models

Importing modules: examples

- To import `numpy`

```python
import numpy
numpy.sqrt(9)
```

```
3.0
```

- Same example, but giving an "alias" to the module

```python
import numpy as np
np.sqrt(9)
```

```
3.0
```

- Same example, but importing only the `sqrt` function

```python
from numpy import sqrt
sqrt(9)
```

```
3.0
```

Why working with modules

- One advantage of organizing code in modules and packages is to avoid messing the namespace.
- Modules allow having functions with the same name in different namespaces, forcing us to be explicit about which one we use.

Why working with modules: examples

- Both `math` and `numpy` have a function `cos` to compute cosine, but their implementation is quite different.
- With numpy:

```python
import numpy as np
print(np.cos(0))
print(np.cos([0,1, np.pi]))
```

```
1.0
[ 1.          0.54030231 -1.        ]
```

- With math

```python
import math
print(math.cos(0))
print(math.cos([0,1, np.pi]))
```

```
1.0
TypeError                 Traceback (most recent call last)
<ipython-input-53-78f2f7c53c4e> in <module>()
      1 print(math.cos(0))
----> 2 print(math.cos([0,1, math.pi]))

TypeError: must be real number, not list
```

Working with decimals

- Computer programs that attempt floating-point arithmetic can produce unexpected and inaccurate results because the floating-point numbers cannot accurately represent all decimal numbers.

```python
item, rate = 0.70, 1.05
tax = item * rate
total = item + tax
txt, val = ['item','tax','total'], [item,tax,total]

for tt, vv in zip(txt, val):
    print(f'{tt:5s} = {vv:.2f}')
```

```
item  = 0.70
tax   = 0.73
total = 1.44     ]
```

- With more decimals

```python
for tt, vv in zip(txt, val):
    print(f'{tt:5s} = {vv:.20f}')
```

```
item  = 0.69999999999999995559
tax   = 0.73499999999999998668
total = 1.43500000000000005329
```

Working with decimals: solving the rounding error

- Errors in floating-point arithmetic can be avoided using Python's "decimal" module. This provides a `Decimal()` object with which floating-point numbers can be more accurately represented.

```
from decimal import Decimal
item, rate = Decimal('0.70'), Decimal('1.05')
tax = item * rate
total = item + tax
txt, val = ['item','tax','total'], [item,tax,total]

for tt, vv in zip(txt, val):
    print(f'{tt:5s} = {vv:.20f}')
```

```
item  = 0.70
tax   = 0.74
total = 1.44     ]
```

- With more decimals

```
for tt, vv in zip(txt, val):
    print(f'{tt:5s} = {vv:.20f}')
```

```
item  = 0.70000000000000000000
tax   = 0.73500000000000000000
total = 1.43500000000000000000
```

Defining functions

Defining functions

- A custom function is created using the **`def`** (definition) keyword followed by a name of your choice and ( ) parentheses.
- The programmer can choose any name for a function except the Python keywords and the name of an existing built-in function.
- This line must end with a : colon character, then the statements to be executed whenever the function gets called must appear on lines below and indented.
- So the syntax looks like this:

```
def function-name ( ) :
    statements-to-be-executed
    statements-to-be-executed
```

Example: a function without arguments

```python
def hello( ):
    print('Hello')
    print('Welcome to Computational Economics!')

hello()
```

```
Hello
Welcome to Computational Economics!
```

Example: a function with arguments

```python
def c2f(c):
    f = 1.8 * c + 32
    print(f'{c:.1f}° Celsius equals {f:.1f} Fahrenheit')

c2f(15)
```

```
15.0° Celsius equals 59.0 Fahrenheit
```

Example: a function returning a value

```python
def c2f(c):
    f = 1.8 * c + 32
    return f

x = c2f(15)
print(x)
```

```
59.0
```

Example: a function with default arguments

```python
def c2f(c, show=False):
    f = 1.8 * c + 32
    if show:
        print(f'{c:.1f}° Celsius = {f:.1f} Fahrenheit')
    return f

c2f(15)
```

```
59.0
```

```python
c2f(15, show=True)      # same as c2f(15, True)
```

```
15.0° Celsius equals 59.0 Fahrenheit
59.0
```

Example: understanding scope

```python
pi = 3.1415
def area(r):
    A = pi * r**2
    return A
print(area(10))
print(A)
```

```
314.15000000000003
---------------------------------------------------------------------------
NameError     Traceback (most recent call last)
<ipython-input-91-4280c1b5ea18> in <module>()
      6
      7 print(area(10))
----> 8 print(A)

NameError: name 'A' is not defined
```

References

McGrath, Mike (2016). *Python in Easy Steps*. In Easy Steps Limited.