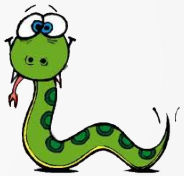
esipap

European School of Instrumentation
in Particle & Astroparticle Physics



Python,
a short introduction

Jérôme ODIER

Philosophy

- Beautiful is better than ugly
- Explicit is better than implicit
- Simple is better than complex
- Complex is better than complicated
- Readability counts

Python

Appeared in 1991; 24 years ago

Designed by Guido van Rossum

Stable release 2.7.X
3.3.X

URL http://www.python.org/

OS cross-platform

What is Python ?

- Python is:
  - structured (if, for, etc...) (see slide part 1)
  - object-oriented (see slide part 3)
  - module-oriented (see slide part 4)

    ↳ Like Java or C#... not C++

Like Java or C#...

  - Modern (garbage collector, introspection)
  - Cross-platform
  - Interpreted (has a bytecode virtual machine)
  - Not optimized for performance but easy wrapping with C, C++, Java, etc... (see pyCuda → speed x 100)

Slower than C/C++, but...

Why Python ?

- Python is perfect for:
  - writing scripts
  - command line tools
  - symbolic computation
  - data analysis
  - ...
  - physicists

```
pyROOT
NumPy
SciPy
Matplotlib
SymPy
pyCuda
...
```

see later

!!! Important !!!

- There are backwards-incompatibilities between major releases !
- Current stable versions:
  - 2.7.X
    - discussed in this tutorial
  - 3.3.X
    - less used then 2.7.X
    - minor incompatibilities
    - see http://docs.python.org/2/library/2to3.html

Plan

(1)The Python language + exercises

(2)Data structures + exercises

(3)Objects + exercises

(4)Modules + exercises

(5)The Python standard library and more

The Python language

The Python console

- Running the Python console:

```
$ python

Python 2.7.2 (default, Oct 11 2012, 20:14:37)
[GCC 4.2.1 Compatible Apple Clang 4.0 (tags/Apple/clang-418.0.60)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>1+1
2
>>CTRL+D

$
```

- Executing a script:

```
$ python hello.py

Hello World !

$
```

hello.py

```
print('Hello World !')
```

Comments

- Writing a comment:

```
# this is a comment
```

  - No /* */ comment in python...

Operators

- Arithmetic op.: **+ - * / % ****
  - Example: x ***** (y **+** 1) **/** z
- Assignment op.: **= += -= *= /=**
  - Example: a **=** 2
    a **+=** 1
- Comparison op.: **== < <= > >=**
  - Example: 1 **==** 0 returns **False**
    1 **>=** 0 returns **True**
- Logical op.: **and or not**
  - Example: (x == 0) **or** (y **<** 0)
    not (x == 0)

```
Quite similar to C/C++
```

```
Multiple assignment:

i, j = 1, 2
```

String operators

- String concatenation operator: **+**
  - Example:'Hello' + ' ' + 'world'
- String formatting operator: **%**

```
'integer: %d, real %f, string %s' % (0, 3.14, 'hello')
```

  - Returns: integer: 0, real 3.14, string: hello

~ equivalent to:

```
'integer: ' + str(0) + ', real: ' + str(3.14) + ', string: ' + 'hello'
```

```
'%08.4' % 3.14
```

  - Returns: 00000003.1400

  - cf. C/C++ “printf” function

Special characters in a string

- Line return:
  - \n
- Back slash:
  - \\
- Simple quote in a string with simple quotes:
  - \'
- Double quote in a string with double quotes:
  - \”
- Percent in a formatted string:
  - %%

Cf. C/C++

Variables and types

- The first assignment to a variable creates it
  - Integer variables:
    - i = 0
  - Real variables:
    - f = 3.14
  - Strings:
    - s1 = 'hello'
    - s2 = “world”
    - s3 = '''Once upon a time
      the world was round
      and you could go on it
      around and around.
      '''
  - Boolean variables:
    - b = True
    - b = False

4 ways to declare strings

    - s4 = """Once upon a time
      the world was round
      and you could go on it
      around and around.
      """

None

- **None** is frequently used to represent the absence of a value.
- A function that returns nothing returns **None**
- Example:

```
x = None
```

- The operator **is** is used to check if a variable is **None**:

```
x is None          → returns True
not x is None      → returns False
```

- **None** ~ **NULL** in C/C++

x == None

White spaces & blocks

- Whitespace is meaningful in Python: especially **indentation** and **placement of newlines**:
  - use a newline to terminate a line of code.
    - Use **** when to go to next line prematurely (long lines).
  - no braces **{ }** to mark blocks of code in Python…
    Use consistent indentation instead (tabs **x**or spaces).
    - The first line with less indentation is outside of the block.
    - The first line with more indentation starts a nested block.
- A colon **:** appears before a new block (E.g. for function and class definitions, etc...).
- Blocks have to contain at least one instruction
  - The keyword **pass** can be used to write an empty blocks

Functions

- Declaring a function:

```
def my_function(my_1st_parameter, my_2nd_parameter, ...):
      my_1st_instruction
      my_2nd_instruction
      ...
```

Instruction block

```
def sum(a, b):
      return a + b

print(sum(1, 2))     # will display 3
```

```
def sum(a = 0, b = 32): ...

print(sum())         # will display 32
print(sum(1))        # will display 33
print(sum(b = 2))    # will display 2
```

if statement 1/2

- Syntax:

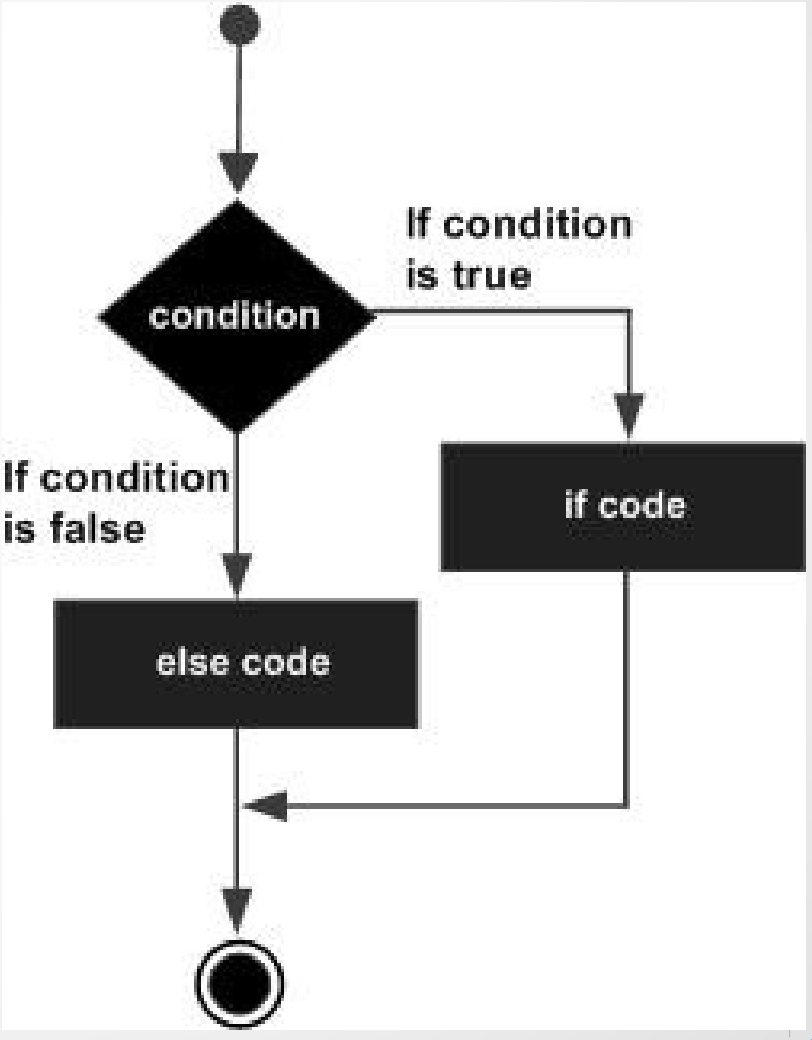
```
if condition:
   block

...

if condition:
   block1
else:
   block2

...

if  condition1:
   block1
elif condition2:
   block2
elif condition3:
   block3
else:
   block4
```



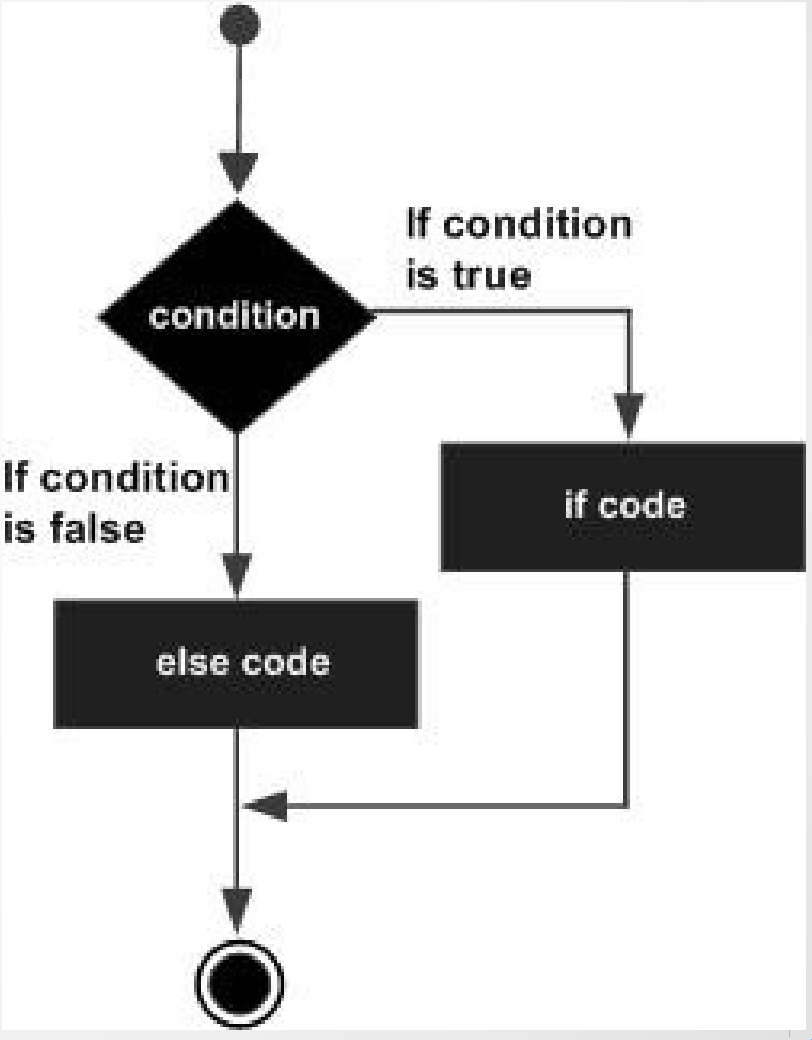
if statement 2/2

- Example

```
if 1 == 1:
  print('Yes ! 1 == 1')


if (x == y) or (x == 0):
  print('foo')
else:
  print('bar')


Etc...
```



while statement

- Syntax:

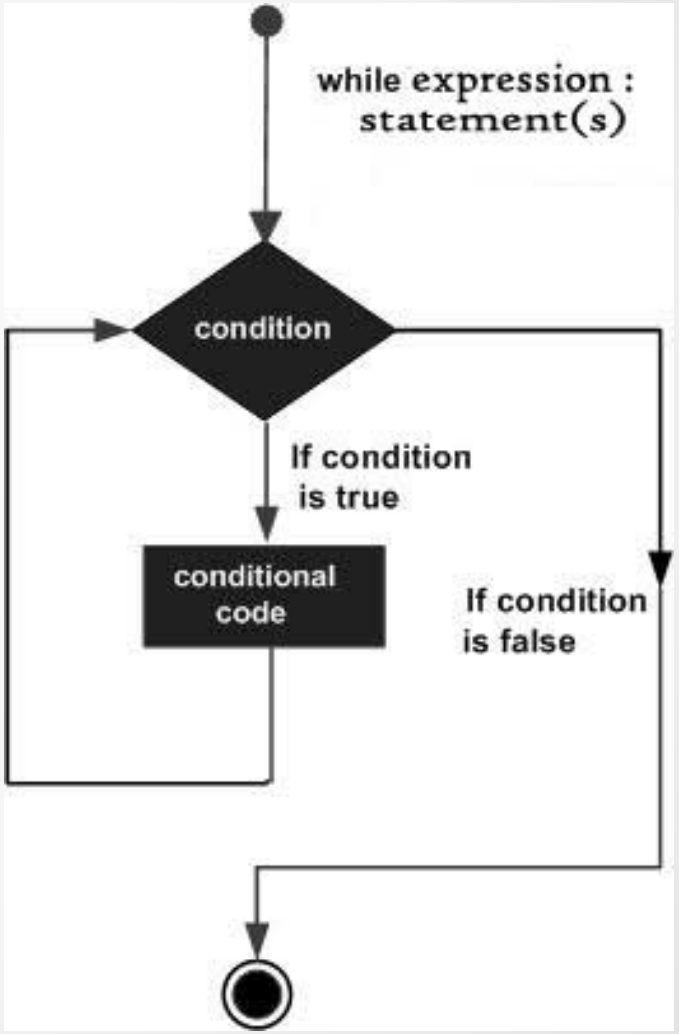
```
while condition:
    block
```

- Example:

```
i = 10

while i > 0:
   i -= 1
   print(i)
```

will display: 9, 8, 7, 6, 5, 4, 3, 2, 1, 0



for statement

- Syntax:

List, dictionnary, tuple, iterable... see slide 28

```
for item in sequence:
    block
```

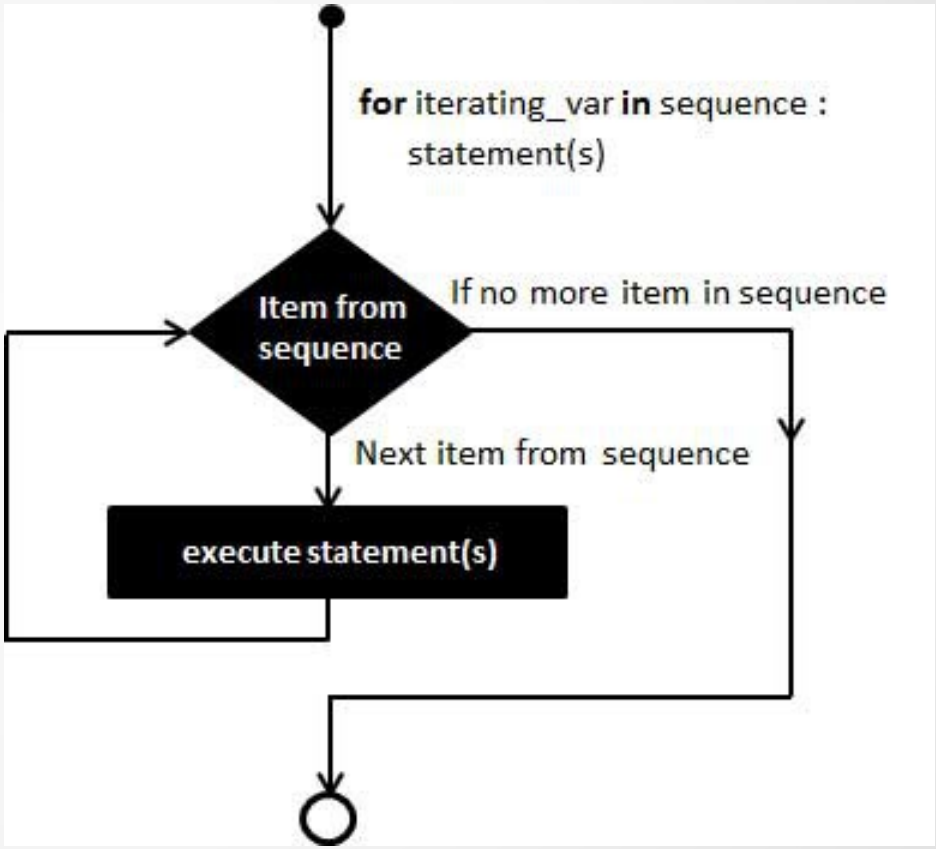
- Example:

```
for item in [0, 1, 2, 3]:
      print(item)

# or equivalently

for item in xrange(4):
      print(item)
```

both will display : 0, 1, 2, 3



break and **continue**

- In a while/for loop:
  - **break** to terminate the loop
  - **continue** to skip the current iteration

```
i = 0                       # i is set to zero

while True:                 # infinite loop
      print(i)              # display i
      i += 1                # increment i
      if i == 4:            # if i = 4
          break             #   terminate loop
```

Gives: 0, 1, 2, 3

```
for item in [0, 1, 2, 3]: # iterate on 0-3
      if item == 1:       # skip i = 1
          continue        #
      print(i)            # display i
```

Gives: 0, 2, 3

Exceptions

- Methodology:
  - When an error occurs, the program raises an exception and its execution is suspended:
    - If the exception is not caught, the program terminates.
    - If the exception is caught, the program resumes its execution in the exception handler.

- Raising an exception:

String message describing the exception

```
raise MyException(message)
```

Exceptions

- Common exceptions:

```
Exception     # All user-defined exceptions should be derived from it.
SyntaxError   # Syntax error.
IOError       # I/O exception (file not found, disk full, …).
OSError       # System-related error (see module os).
KeyError      # Dictionary error (see part 2).
```

- User-defined exceptions:

```
class MyException(Exception):
      pass
```

Exceptions

- Using exceptions

```
try:                                    # Exception-protected session
  …
  if error:
     raise MyException(message)         # Raise MyException
  …

except MyException as e:
   print(e)                             # Print message
```

Jumps here

```
try:                                    # Exception-protected session
  …
  if error1:
     raise MyException1(message1)       # Raise MyException1
  …
  if error2:
     raise MyException2(message1)       # Raise MyException2
  …

except MyException1 as e:
   print(e)                             # Print message1

except MyException2 as e:
   print(e)                             # Print message2
```

Anatomy of a script files

- example.py

```
#!/usr/bin/env python                          # interpret with python

import sys                                     # load module `sys`

def my_entry_point(argv):                      # program entry point
      print(argv)                              # print cmdline argument list

      return 0                                 # no error, EE if error

if __name__ == '__main__':
      sys.exit(my_entry_point(sys.argv))
```

```
$ chmod a+x example.py
```

Useful functions

- Printing a string, an integer, etc... to screen:
  - print(obj1,[obj2,...])
- Reading a string from keyboard:
  - raw_input([prompt])
- Terminate the program:
  - import sys
  - sys.exit(0) # if success
  - sys.exit(n) # if error with n != 0
- See the complete list of builtin functions:
  - http://docs.python.org/2/library/functions.html

1st exercice (~20min)

- Write a cmdline program that append a line (read from keyboard) to a file.
- In the shell:

```
$ ./append.py file.txt
```

- Reading a file content:

```
f = open('file.txt', 'r')
content = f.read()
f.close()
```

- Writing a file content:

```
f = open('file.txt', 'w')
f.write(content)
f.close()
```

These functions can raise IOError

- len(argv) is the number of cmdline arguments. argv[1] is the first argument.

Data structures

References:
http://docs.python.org/2/tutorial/datastructures.html

Lists

- Lists are Python's workhorse datatype.
- Creating a list:

```
L1 = []                     # an empty list
L2 = list()                 # an empty list
L3 = [1, 2, 3, 4]           # or quickly L = range(1, 5)
L4 = L3                     # is just a reference to L3
L5 = list(L3)               # L5 is a clone of L3
L6 = 3 * [1, 2]             # returns [1, 2, 1, 2, 1, 2]
```

- Accessing items:

```
L = [1, 2, 3, 4]
L[0]                        # returns 1
L[3]                        # returns 4
L[-1]                       # returns 4
L[-2]                       # returns 3
L[1: ]                      # returns [2, 3, 4]
L[1: 3]                     # returns [2, 3]
L[1: -1]                    # returns [2, 3]
L[: 3]                      # returns [1, 2, 3]
L[:]                        # returns [1, 2, 3, 4]
L[4]                        # raise IndexError
```

slicing

Lists

- Append an item to the end of the list:

```
L.append(item)
```

- Insert an item at the given position:

```
L.insert(position, item)
```

- Extend the list by appending all the items of the given list:

```
L.extend(L2)
```

- Remove the first item from the list whose value is x:

```
L.remove(item)
```

- Remove the last item, and get it:

```
L.pop()
```

Lists

- Return the index of the first item whose value is x:

```
L.index(x)        # on error raise ValueError
```

- Return the number of times that x appears in the list:

```
L.count(x)
```

- Number of items in the list:

```
len(L)
```

- Check is an item is in the list:

```
if item in L:
```

- Iterate over the list content:

```
for item in L:
```

range and **xrange**

- range(stop)
- range(start, stop[, step=1])
  - Returns [start, start + step, start + 2 * step, …, stop – 1]

```
range(7)          # returns [0, 1, 2, 3, 4, 5, 6]
range(0, 7)       # returns [0, 1, 2, 3, 4, 5, 6]
range(1, 7)       # returns [1, 2, 3, 4, 5, 6]
range(0, 7, 2)    # returns [0, 2, 4, 6]
```

- xrange(stop)

  xrange(start, stop[, step=1])
  - This function is very similar to **range**(), but it returns an **xrange** object instead of a list. This is an opaque sequence type which yields the same values as the corresponding list, without storing them all simultaneously. → see python iterators

Dictionaries (associative arrays, hash map, map)

- Unlike lists, which are indexed by numbers, dictionaries are indexed by keys, which can be any immutable type; strings and numbers can always be keys...
- Creating a dictionary:

```
D1 = {}                     # an empty dict
D2 = dict()                 # an empty dict
D3 = {'foo': 1, 'bar': 2} #
D4 = L3                     # is juste a reference on D3
D5 = dict(L3)               # D5 is a clone of D3
```

- Accessing items:

```
D = {'foo': 1, 'bar': 2}
D['foo']                    # returns 1
D['bar']                    # returns 2
D['foobar']                 # raise NameError
```

Dictionaries

- Add an item to the end of the dict:

```
D['key'] = value
```

- Remove the item of then given key:

```
del D['key']
```

- Extend the dict by appending all the items in the given dict:

```
D.update(D2)
```

- Returns a list of all the keys in the dictionary:

```
D.keys()
```

- Check if a key exists:

```
D.has_key('key')     or      'key' in D.keys()
```

Dictionaries

- Iterate the list of all the keys used in the dictionary:

```
for key in D:          or          for key in D.keys():
```

- Example:

```
D = {'foo': 1, 'bar': 2}

for key in D:
      print('%s: %s' % (key, D[key]))
```

- Gives:

```
bar: 2
foo: 1
```

!!! keys are in arbitrary order !!!
in a dictionnary

Sets and tuples

- Sets: A set is an unordered collection with no duplicate elements...
  - Methods: add, remove, pop, update, intersection_update, difference_update, symmetric_difference_update, issubset, issuperset, union, intersection, difference, symmetric_difference,…
  - Create a set: set(L), set([1, 2, 4]), {1, 2, 3}
- Tuples: A tuple comma-separated set of values (of different types)...
  - Create a tuple: T = 1, 2, 3
  - A tuple can be returned by a function:

```
def foo():
       return 1, 'Hello !'

a, b = foo()
my_tuple = foo()
print(tuple)                    # returns 'Hello !'
```

Strings

- Strings can be ~seen as a list of bytes:

```
s = 'Example'
s[0]                    # returns 'E'
s[-1]                   # returns 'e'
s[1: ]                  # returns 'xample'
etc...
```

- Common methods:
  - s.lower/upper()
    - Returns a copy of s, converted to lower/upper case.
  - s.split(sep)
    - Returns a list of the words of the string s.

'foo,bar'.split(',') gives ['foo', 'bar']

  - s.find|index(sub[, start[, end]])
    - Returns the lowest index in s where the substring sub is found such that sub is wholly contained in s[start: end].
    - index() is like find() but raise ValueError if the substring is not found.