A short introduction to Python

Florent Lamiraux
CNRS/LAAS
October 8, 2014
Why Python?

- Readability and expressiveness,
  - high level features (lists, sets, mappings,...),
- user friendly
  - automatic allocation and garbage collecting,
  - huge standard library (regexp, numpy,...),
  - dynamic typing,
- glue for many applications
  - Python/C API,
  - Bindings for many tools (CORBA, OpenCV,...)
- Object oriented
- Efficiency
  - Compilation on the fly
Documentation and tutorials

- Python 2: https://docs.python.org/2
- Python 3: https://docs.python.org/3
Syntax

- One instruction per line, except in case of opening symbol "", (, [, : line ends up at closing symbol.
- Blocs are defined by 4 space indentations,
- semicolon allows to put several instructions on the same line,
- backslash allows to extend an instruction on the following line.
- # for comments,
- instruction pass does nothing
String - literal

- delimited by simple or double quotes
- examples
  
  ```
  print "string between double quotes."
  print 'string between double quotes.'
  print u’unicode string.’
  ```
- from python 3.x on, print becomes a function.
- " " "allows to define a multiline string."
Numeric types - literal

- integer
  - OS size or long integer without bound,
  - automatic conversion.
- floating point numbers
  - OS size for double
- complex numbers
  - 1.+2.5j
- Boolean: True or False
- None : non-typed value meaning no value.
Numeric types - operators

- +, -, *, / the same as in C
- // division between integers
- % modulo
- divmod(x, y) returns a pair (x//y, x%y)
- x**y the same as pow(x, y)
Conversions

- \( \text{int}(x) \) convert to integer,
- \( \text{long}(x) \) convert to long integer,
- \( \text{float}(x) \) convert to float,
- \( \text{complex}(x, y) \) create complex number \( x+yj \),
- \( \text{str}(x) \) convert to a string.
Variables

- No declaration. Variables are defined at affectation with =
  ```python
  x = 25
  text = 'My text'
  ```
- A variable can change type during execution:
  ```python
  x = 'My text'
  ```
- Multiple affectation
  ```python
  x = y = 0
  ```
- Parallel affectation
  ```python
  x, y = 10, 20
  ```
- Reading a non-defined variable raises an exception
  ```python
  >>> print z
  NameError: name 'z' is not defined
  ```
Tuples and lists

- Tuples and lists are iterable containers
  - access by index starting from 0,
  - elements can be of different types.
- lists are defined by [],
  ```python
  >>> L = [10, 'toto', 20]
  >>> L[1]
  'toto'
  ```
- tuple are defined by () and are not modifiable
  ```python
  >>> T = (10, 'toto', 20)
  >>> T[1] = 2
  TypeError: 'tuple' object does not support item assignment
  ```
Tuples and lists

- **Function `len()`** returns the size
  ```python
  >>> L = [10, 'toto', 20]
  >>> len(L)
  3
  ```

- **Function `min()` and `max()`** return the min and max values
  ```python
  >>> print (min(L), max(L))
  (10, 'toto')
  ```

- **`sorted(sequence)`** return a sorted iterator of the sequence
  ```python
  >>> for i in sorted(L):
  >>>     print i
  10
  20
  'toto'
  ```
Tuples and lists

- *in*, *not in* test whether an element belongs to a sequence
  ```python
  >>> if 10 in L: ...
  ```

- `==`, `<`, `>`, `<=`, `>=`, `!=` lexicographic comparison:
  ```python
  >>> (5,3,1) < (5,2,10)
  False
  ```

- `count(value)` method returns the number of occurrences of `value` in the sequence

- `index(value)` method returns the index of the first occurrence of `value` in the sequence, raises an exception if `value` is not in the sequence
Modifying a list

- `L.append(x)` add element `x` at end of list `L`.
- `L1.extend(L2)` add list `L2` at the end of list `L1`.
- `L.insert(i, x)` insert `x` at position `i`.
- `del L[i]` remove value at position `i`.
- `L.pop(i)` remove and return value at position `i`.
- `L.sort()` sort a list,
- `list(sequence)` convert a sequence into a list
- `tuple(sequence)` convert a sequence into a tuple.
Slices

- \( L[i:j:k] \) extract the sub-sequence starting at \( i \) ending at \( j-1 \), by steps of \( k \).
- if \( k \) not specified, \( k=1 \),
- if \( j \) not specified, up to end of list,
- if \( i \) not specified, \( i=0 \),
- if \( i \) or \( j \) < 0, from end of list
- if \( j<i \), empty sequence,
- if \( i \) or \( j \) out of range, replaced by beginning or end.
Slices

```python
>>> L=[0,10,20,30,40,50,60,70,80,90]
>>> L[2:8]
[20, 30, 40, 50, 60, 70]
>>> L[2:8:2]
[20, 40, 60]
>>> L[8:]
[80, 90]
>>> L[::3]
[0, 30, 60, 90]
>>> L[-3:-1:]
[70, 80]
>>> L[-3:]
[70, 80, 90]
>>> L[::-1]
[90, 80, 70, 60, 50, 40, 30, 20, 10, 0]
```
Slices

• Slices and parallel affectation
  

• Affectation with a sequence or parallel affectation of different size

\[
>>> L[:3]=('a','b') \quad \text{#affectation from a tuple}
\]

\[
>>> L[-2:]= 'y', 'z' \quad \text{#parallel affectation}
\]

\[
>>> L
\]

\[
[a', b', 30, 40, 50, 60, 70, y', z']
\]

0, 10, 20 replaced by 'a', 'b'; 80, 90 by 'y', 'z'.

Lists and copy

• By default, lists are not copied except when slicing

```python
>>> L=[0,10,20,30,40,50,60,70,80,90]
>>> L2=L
>>> L3=L[: :]
>>> L2[3]='copy'
>>> L3[4]='copy'
>>> L
[0, 10, 20, 'copy', 40, 50, 60, 70, 80, 90]
```

• L2 is a reference to L while L3 is a copy of L.
Dictionary

- Mapping (key, value)
  - key can be any immutable object,
  - value can be any object.
  - `items()` method returns a list of tuples (key, value),
  - `keys()` method returns the list of keys,
  - `values()` method returns the list of values,
  - `copy()` method returns a copy of the dictionary.
Dictionary

- Mapping (key, value)
  - key can be any immutable object
  - value can be any object

```python
>>> D=
>>> D['name']='Lamiraux'
>>> D[(7,'avenue du Colonel Roche')]='LAAS'
>>> D[(14,'avenue Edouard Belin')]='CNRS-DR14'
>>> D
{(7, 'avenue du Colonel Roche'): 'LAAS', 'name': 'Lamiraux', (14, 'avenue Edouard Belin'): 'CNRS-DR14'}
>>> D[(14,'avenue Edouard Belin')]
'CNRS-DR14'
>>> for k,v in D.items():
    print k,v
(7, 'avenue du Colonel Roche') LAAS
name Lamiraux
(14, 'avenue Edouard Belin') CNRS-DR14
```
Set

- Represent sets in the mathematical meaning
  - created by `set()`,
  - `add()` method adds an element,
  - `remove()` method removes an element,
  - operators `in,<,>,<=>,=-,|,&,^`
  - methods `isdisjoint`, `issubset`, `issuperset`, `union`, `intersection`, `difference`, `symmetric_difference`, `copy`
Set

```python
>>> S1=set([1,2,3,4,5,6]) # or S1={1,2,3,4,5,6} from python2.7 on
>>> S2=set([5,6,7,8])
>>> S1.union(S2)
set([1, 2, 3, 4, 5, 6, 7, 8])
>>> S1.intersection(S2)
set([5, 6])
>>> 5 in S1
True
>>> S2-S1
set([8, 7])
>>> S1^S2
set([1, 2, 3, 4, 7, 8])
```
Frozenset

- Immutable set,
- Can be used as dictionary key (unlike sets)
- created by \texttt{frozenset()}. 
String

- Object belonging to class `str` or `unicode`
- Can be handled as tuple of characters,
- `lower()`, `upper()`, `capitalize()` change the case
- `replace(old,new[,count])` replace occurences of `old` by `new`,
- `find(sub[,start[,end]])` find first occurrence of `sub`.
- `strip([chars])` erase spaces or `chars` at beginning and end of string, also `rstrip`, `lstrip`
- `split(sep)` extract elements of a string separated by `sep`

```python
>>> '10,20,30'.split(',')
[''10', ''20', ''30'']
```
String formatting

- `string%parameters`
- Parameters can be
  - a value
  - a sequence,
  - a dictionary

```python
>>> s = "Mr %s is %i year old."

>>> s%('Dupond', 30)
'Mr Dupond is 30 years old.'

>>> d={'name':'Durand', 'age':45}

>>> s='Mr %(name)s is %(age)d year old'

>>> s%d
'Mr Durand is 45 year old'
```
String formatting

- %s display result of `str()`
- %r display result of `repr()`
- %d,i display decimal integer
- %f,g,e display floating point number
- %x,X hexadecimal
- %o octal
**Instruction if**

`if boolean_expression:
    indented_conditional_instruction`

or

`if boolean_expr : conditional_instr`

- Example

  ```python
  if i>8:
      print('i is greater than 8.')
  if i > 22: print('i is greater than 22.')
  ```
Instruction \textbf{if}

- Conditional expressions can be built with boolean operator
  \hspace{1cm} \textbf{and, or, not}

- with \textbf{and} and \textbf{or} expressions are evaluated only if necessary.
Instructions if, elif, else

if X>0:
    print 'x positive'
elif X==0:
    print 'x equal 0'
else:
    print 'x negative'
Conditional expression

• value if condition else other value
• example

'positive' if X >= 0 else 'negative'
Instruction **while**

- Iterate while a condition is true
  ```
  i=0
  while i<5:
    print i
    i+=1
  else: print 'end'
  ```

- **break** get out of the loop,

- **continue** go to next iteration.
Instruction for

- Iterate over a sequence
  
  ```python
  L=[0,10,20,30]
  for e in L:
      print e
  ```

- Iterate over a sequence of integers
  
  ```python
  for i in range(0,5):
      print i
  ```
Instruction for

- Don't do that
  
  ```python
  L=[0,10,20,30]
  for i in range (len (L)):
    print L [i]
  ```

- but instead
  
  ```python
  L=[0,10,20,30]
  for i in L:
    print L
  ```
List comprehension

• Syntax:
  
  expression for target in sequence if condition

• "if condition" is not mandatory

• Example

  >>> L=[x**2 for x in xrange(10) if x%2==0]
  >>> L
  [0,4,16,36,64]
Function

- Functions are objects that can be manipulated as such
- Definition
  ```python
def function_name(arg1, arg2):
    ...
    return ...
  ```
- Call
  ```python
  function_name(x, y) or function_name(arg1=x, arg2=y)
  ```
- Example
  ```python
  >>> def sum(a, b):
  ...     c = a + b
  ...     return c
  >>> A = 2; B = 3; sum(A, B)
  5
  ```
Lambda function

• A practical way of defining function

  
  function_name = lambda x1,x2,...: expression

  equivalent to

  def function_name(x1,x2,...):
      return expression

• Example

  >>> square = lambda x: x**2

  >>> square(3)

  9
Function

• Note that functions are objects, as such
  – they can be put into variables, sequences,...
  – They can be returned by a function.

• Example

  >>> def add_constant(constant):
  >>>     return lambda x:x+constant
  >>> add_three = add_constant(3)
  >>> add_three(5)
  8
function `zip`

- `zip(list1, list2)` return a list of tuples of two elements composed of elements of `list1` and `list2`.

```python
>>> L1=(10, 20, 30)
>>> L2=[100, 200, 300, 400]
>>> zip(L1, L2)
[(10, 100), (20, 200), (30, 300)]
```

```python
>>> for i, j in zip(L1, L2):
...     print i, j
10 100
20 200
30 300
```
Operations on sequences

map(fct, seq) apply fct to each element of seq.

filter(fct, seq) build the sequence of elements for which fct returns True.

reduce(fct, seq, init) apply fct of two arguments cumulatively to the elements of seq so as to reduce it to a single value.

```python
>>> L=(10,20,30)
>>> map(lambda x:x**2,L)
[100, 400, 900]
>>> filter(lambda x:True if x<25 else False, L)
(10, 20)
>>> reduce(lambda x,y:x+y**2, L, 0)
1400  (0 + 10*10 + 20*20 + 30*30)
```
Modules

• A module is
  – a python file
  – or a shared object,
  – or a directory containing a file named __init__.py

• `import mod` command import all objects defined in file `mod.py`.

• Objects defined in `mod.py` are accessible through namespace `mod`.

• Example

```python
#file mod.py
a=0

>>> import mod
>>> mod.a
0

>>> import mod as m
>>> m.a
0

>>> from mod import a
>>> a
0
```
Class

• A class may have
  – a constructor,
  – instance methods and members,
  – class methods and members,

• Definition

  ```python
  class class_name :
    class declaration
  ```

• Construction of an instance

  >>> a = class_name()
Instance method

- function that applies to the object that calls it
- the first parameter is the object and is usually denoted by self

```python
class class_name:
    def method(self):
        self.x = 3

>>> a = class_name()
>>> a.method()
>>> a.x
3
```
Constructor

- Constructor is a method denoted by `__init__` and called at instance creation
- It can have parameters and usually defines instance members.

```python
class class_name:
    def __init__(self, a):
        self.A = a

...
Class member

- Shared by all instances of the class
  ```python
class class_name:
    A=0
  ```
- access
  ```python
class_name.A
read-only access through instances. Modification through instance yields the creation of an instance member
>>> obj1=class_name()
>>> obj1.A
0
>>> obj1.A=1
>>> obj2=class_name()
>>> obj2.A
0
```
Class method

• Method that apply to a class and not to an instance

```python
class class_name:
    @staticmethod
    def method():
        ...
```

• access

```python
class_name.method or
a = class_name()
a.method()
```
Class inheritance

• Allow to create a class that inherits another class methods and members,
• methods can be redefined
• constructor may call parent constructor

```python
class child (parent):
    def __init__(self):
        parent.__init__(self)
```

• It is recommended to make base class derive from `object`. Class is thus said “new style” class.
Special methods

• `__lt__`, `__le__`, `__eq__`, `__ne__`, `__gt__`, `__ge__` (self, other) overload operators `<`, `<=`, `==`, `!=`, `>`, `>=`

• `__str__` (self) define the conversion of the object as a string (print).
Exceptions

- Errors in python are dealt with using exceptions
- Some are defined by the language, but exception classes can be defined
- Example

  ```python
  >>> a
  Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
  NameError: name 'a' is not defined
  ```
Exceptions

• Exceptions derive from `BaseException` class and more frequently from `Exception` sub-class.
• It is possible to catch an exception in order to handle an error
• Exceptions not caught terminate execution.
Catching exceptions

- `try/except` catches exceptions

Example

```python
try:
    f = open('myFile', 'r')
except IOError as exc
    print exc
```

- In this example, only IOError are caught.
Defining new exceptions

Class MyException(Exception):
    def __init__(self, msg):
        self.message = msg
    def __str__(self):
        return self.message

- Raising an exception

    try:
        raise MyException('this is my error')
    except MyException as exc:
        print exc
try, except, else, finally

- **else** clause is executed if no exception is caught,
- **finally** clause is executed whatever happens:

```python
def divide(x,y):
    try:
        result = x/y
    except ZeroDivisionError:
        print('division by 0')
    else:
        print('result is %s'%result)
    finally:
        print('finally')
```
Inline help and docstring

- In interactive mode, help can be invoked on any symbol

```python
class A (object):

    ""
    This is the documentation of class A
    ""

def __init__ (self):

    ""
    This is the documentation of A constructor
    ""

    pass
```
Inline help and docstring

- In interactive mode, help can be invoked on any symbol

```python
>>> help(A)
Help on class A in module __main__:

class A(__builtin__.object)

    This is the documentation of class A

    Methods defined here:

    __init__(self)

    This is the documentation of A constructor

    ...
Inline help and docstring

- In interactive mode, help can be invoked on any symbol

```python
>>> help (A)
```

```txt
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Data descriptors defined here:</td>
</tr>
<tr>
<td><strong>dict</strong></td>
</tr>
<tr>
<td>dictionary for instance variables (if defined)</td>
</tr>
<tr>
<td><strong>weakref</strong></td>
</tr>
<tr>
<td>list of weak references to the object (if defined)</td>
</tr>
</tbody>
</table>
```

(END)
Notice that most objects have an internal dictionary named `__dict__`

```python
>>> A.__dict__.keys()
['__dict__', '__module__', '__weakref__', '__doc__', '__init__']
```

```python
>>> A.__dict__['__module__']
'__main__'
```

```python
>>> A.__module__
'__main__'
```
Coding style: PEP 8

- 4-space indentation, no tabs,
- wrap lines so that they don’t exceed 79 characters,
- use blank lines to separate functions and classes, and larger blocks of code inside functions,
- use docstrings.
- use spaces around operators and after commas, but not directly inside bracketing constructs: \( a = f(1, 2) + g(3, 4) \),
- Name your classes and functions consistently
  - CamelCase for classes,
  - lower_case_with_underscores for functions and methods,
  - use self as the name for the first method argument
- Only use Plain ASCII
Exercise 1

- Define two lists: 11 with names and 12 with ages,
- Define a function taking two arguments: a name and an age and print
  
  Mr name is age year(s) old.

- Use this function to print the above sentence for each name in 11 with ages in 12.
Exercise 2

Compute prime numbers up to 1000:

1. using functions
   - define a function that returns the set of multiples not greater than 1000 of an integer,
   - From the set of integers, successively remove multiples of 2, 3, 5, 7...

2. using list comprehension
Exercise 3

- Write a class `Vector` with the following features:
  - The constructors take as input a tuple,
  - Operators +, * return sum and inner product of 2 vectors,
  - Exception is raised when size mismatch.

- Hint
  - To define operators +, *, define methods `__add__` and `__mul__`
Manipulating files

- **read** (n) reads n characters
- **write** (s) writes string s
- **readline** () reads one line
- **close** () closes the file
- Instruction **for** reads a file line by line:
  ```python
  f = open('myFile', 'r')
  for line in f:
      print (line)
  ```
Instruction with

- `with` automatically calls methods `__enter__()` and `__exit__()` even if an exception is raised.
- Class file implements these methods. Thus, we can write

  ```python
  with open('myFile', 'r') as f:
    for line in f:
      print(line)
  ```

- No need to call `close()` anymore.
Selected parts of the library
System-specific parameters and functions

- Module `sys`
  - `sys.path`: A list of strings that specifies the search path for modules (command `import`). Initialized from the environment variable PYTHONPATH, plus an installation-dependent default.
Miscellaneous operating system interfaces

• Module `os`
  - `os.environ`: a dictionary representing the string environment. For example, `environ['HOME']` is the pathname of your home directory (on some platforms), and is equivalent to `getenv("HOME")` in C.
  - `os.getenv(varname[,value])`: return the value of the environment variable `varname` if it exists, or `value` if it doesn’t. `value` defaults to `None`.
Regular expression operations

- Module `re` provides regular expression matching operations similar to those found in Perl

```python
>>> import re
>>> m = re.search ('(\d*)x(\w*)', '12xy')
>>> m.groups()
('12', 'y')
```

- Note that manipulation on strings is already powerful.