The Python interpreter

Remi Lehe

Lawrence Berkeley National Laboratory (LBNL)

US Particle Accelerator School (USPAS) Summer Session
Self-Consistent Simulations of Beam and Plasma Systems
S. M. Lund, J.-L. Vay, R. Lehe & D. Winklehner
Colorado State U, Ft. Collins, CO, 13-17 June, 2016

Python interpreter: Outline

1. Overview of the Python language
2. Python, numpy and matplotlib
3. Reusing code: functions, modules, classes
4. Faster computation: Forthon

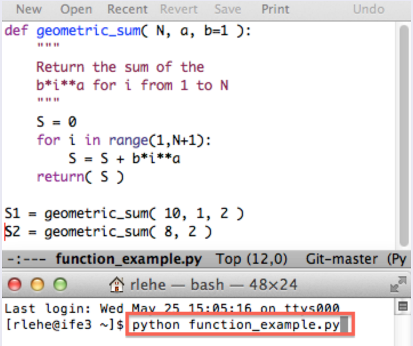
Overview of the Python programming language

- **Interpreted language** (i.e. not compiled)
  → Interactive, but not optimal for computational speed
- **Readable and non-verbose**
  No need to declare variables
  Indentation is enforced
- **Free and open-source**
  + Large community of open-souce packages
- **Well adapted for scientific and data analysis applications**
  Many excellent packages, esp. numerical computation (`numpy`), scientific applications (`scipy`), plotting (`matplotlib`), data analysis (`pandas, scikit-learn`)

Interfaces to the Python language

Scripting

- Code written in a **file**, with a text editor (`gedit`, `vi`, `emacs`)
- Execution via command line (`python` + filename)



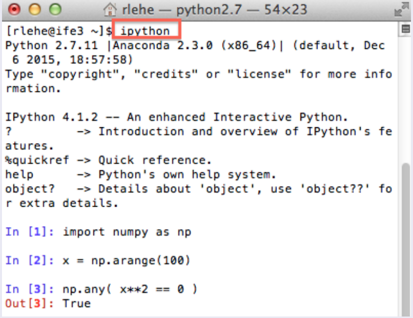
```
def geometric_sum( N, a, b=1 ):
    """
    Return the sum of the
    b*i**a for i from 1 to N
    """
    S = 0
    for i in range(1,N+1):
        S = S + b*i**a
    return( S )

S1 = geometric_sum( 10, 1, 2 )
S2 = geometric_sum( 8, 2 )
```

```
Last login: Wed May 25 15:05:16 on ttys000
[rlehe@ife3 ~]$ python function_example.py
```

Convenient for **long-term** code

Interactive shell

- Obtained by typing `python` or (better) `ipython`
- Commands are typed in and executed one by one



```
[rlehe@ife3 ~]$ ipython
Python 2.7.11 |Anaconda 2.3.0 (x86_64)| (default, Dec
 6 2015, 18:57:58)
Type "copyright", "credits" or "license" for more info
rmation.

IPython 4.1.2 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's fe
atures.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' fo
r extra details.

In [1]: import numpy as np

In [2]: x = np.arange(100)

In [3]: np.any( x**2 == 0 )
Out[3]: True
```

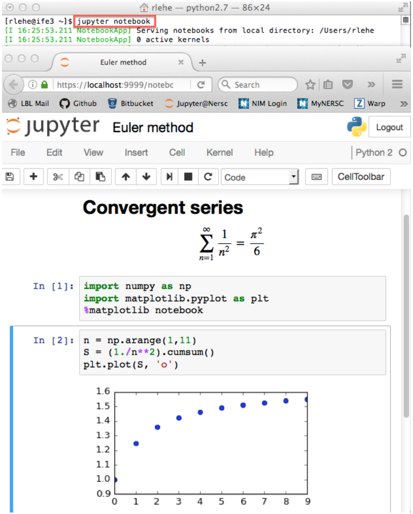
Convenient for **exploratory work, debugging, rapid feedback**, etc...

Interfaces to the Python language

IPython (a.k.a Jupyter) notebook

- Notebook interface, similar to `Mathematica`.
- Intermediate between **scripting** and **interactive shell**, through **reusable cells**
- Obtained by typing `jupyter notebook`, opens in your web browser

Convenient for **exploratory work, scientific analysis and reports**



```
[rlehe@ife3 ~]$ jupyter notebook
[I 16:25:53.211 NotebookApp] Serving notebooks from local directory: /Users/rlehe
[I 16:25:53.211 NotebookApp] 0 active kernels
```

Convergent series

$$\sum_{n=1}^{\infty} \frac{1}{n^2} = \frac{\pi^2}{6}$$

In [1]:

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib notebook
```

In [2]:

```
n = np.arange(1,11)
S = (1./n**2).cumsum()
plt.plot(S, 'o')
```

Overview of the Python language

This lecture

Reminder of the main points of the **Scipy lecture notes** through an example problem.

Example problem: Euler's method

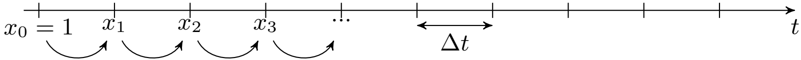
Use Euler's method to **numerically integrate**, between $t = 0$ and 10:

$$\frac{d\,x(t)}{dt} = x(t)\cos(t) \qquad \text{with} \qquad x(0) = 1$$

Compare it with the exact solution: $x(t) = e^{\sin(t)}$

Reminder: In this case, Euler's method gives:

$$\begin{aligned} t_i &= i\Delta t \\ x_i &= x_{i-1} + \Delta t \times x_{i-1}\cos(t_{i-1}) \end{aligned}$$

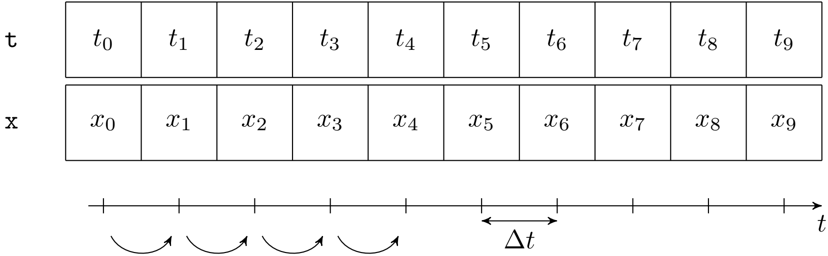


Python interpreter: Outline

1. Overview of the Python language
2. Python, numpy and matplotlib
3. Reusing code: functions, modules, classes
4. Faster computation: Forthon

Example problem: Structure of the code

Storage in memory:



`t`	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9
`x`	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9

For loop:

Repeatedly apply: $x_i = x_{i-1} + \Delta t \times x_{i-1} \cos(t_{i-1})$

Numpy arrays

Numpy arrays

Provide efficient **memory storage** and **computation**,
for large number of elements of the same type.

- Standard import: `import numpy as np`
- Creation of numpy arrays:
  `np.arange`, `np.zeros`, `np.random.rand`, `np.empty`, etc...
  (In **ipython**, use e.g. `np.arange?` to read the documentation)
- Individual elements are accessed with square brackets:
  `x[i]` (1D array), `y[i,j,k]` (3D array)
  For an array with `N` elements, the indices start at `0` (included) and end at `N-1` (included)
- Subsets of the array are accessed using **slicing syntax**:
  `x[` *start index* `:` *end index* `:` *step* `]` ; in particular:
  - `x[` *start index* `:` *end index* `]` : slicing with step 1 by default
  - `x[` `:` *end index* `]` : slicing with start index 0 by default
  - `x[` *start index* `:` -1 `]` : slicing up to the last-but-one element

For loops

For loop

Repeatedly perform a given operation
(e.g. apply the same operation to every element of a numpy array)

Syntax:

`for i in range(` *start index* `,` *end index* `,` *step* `):`
    Perform some operation that depends on `i`

- Indentation and the use of column (`:`) are key.
- The `range` function can be used with 1, 2 or 3 arguments:
  - `range(N)`: loop from index `0` to index `N-1` (included)
  - `range(i,N)`: loop from index `i` to index `N-1` (included)
  - `range(i,N,k)`: loop from index `i` to index `N-1` with step `k`
- In the above, `range` can also be replaced by a list or any iterable.

Numpy and for loops: task

Task 1

In a text editor, write a python script (named `euler.py`) which:

- Sets the number of integration steps to `N = 200`, and the timestep to `dt = 10./N`
- Initializes the array `t` (with `N` elements) using `np.arange` so that

$$t_i = i\Delta t$$

- Initializes the array `x` (with `N` elements) using `np.empty` and setting the initial point `x[0]` to 1.
- Loops through the array `x` and applies Euler's method: (Here, the loop should start at $i = 1$, not $i = 0$)

$$x_i = x_{i-1} + \Delta t \times x_{i-1} \cos(t_{i-1})$$

Run the script (`python euler.py`), to check that there is no error.

Comparison with the exact solution

`t`	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9
`x`	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9
`x_exact`	$e^{\sin(t_0)}$	$e^{\sin(t_1)}$	$e^{\sin(t_2)}$	$e^{\sin(t_3)}$	$e^{\sin(t_4)}$	$e^{\sin(t_5)}$	$e^{\sin(t_6)}$	$e^{\sin(t_7)}$	$e^{\sin(t_8)}$	$e^{\sin(t_9)}$

We wish to compare the two results by:

- Calculating the RMS error:

$$\epsilon_{RMS} = \sqrt{\frac{1}{N}\sum_{i=0}^{N-1}(x_i - x_{exact,i})^2}$$

- Plotting x and x_{exact} versus t.

Numpy arrays: element-wise operations

Element-wise operation

Operation that is repeated for each element of an array
and does not depend on previous/next elements.

e.g. $$x_{exact,i} = e^{\sin(t_i)} \qquad \forall i \in [0, N-1]$$

- Could be done with a `for` loop:

```
for i in range(N):
    x_exact[i] = np.exp( np.sin( t[i] ) )
```

- But is **computationally faster** with numpy vector syntax:

```
x_exact = np.exp( np.sin( t ) )
```

Numpy vector syntax also works for the element-wise operations:
`+`, `-`, `*`, `/`, `**` (power), `np.sqrt` (square-root), `np.log`, etc...

Numpy arrays: reduction operations

Reduction operation

Operation that extracts a single scalar from a full array

e.g.

$$S = \sum_{i=0}^{N-1} y_i$$

- Again, could be done with a `for` loop:

```
S = 0
for i in range(N):
    S = S + y[i]
```

- But is **computationally faster** with numpy reduction methods

```
S = np.sum( y )
```

Other reduction operations:

`np.product`, `np.max`, `np.mean`, etc... (for real or integer arrays)

`np.any`, `np.all`, etc... (for boolean arrays)

Plotting package: matplotlib

Other Python plotting packages: `pygist`, `bokeh`, `seaborn`, `bqplot`, ...

Pros of matplotlib

- Publication-quality figures
- Extremely versatile and customizable
- **Standard plotting package in the Python community**

Cons of matplotlib

- Slow
- Sometimes verbose
- Limited interactivity

- Standard import: `import matplotlib.pyplot as plt`
- Basic plotting commands:
  `plt.plot( t, x )` (plots 1darray `x` as a function of 1darray `t`)
- Show the image to the screen:
  `plt.show()` (unneeded when using `ipython --matplotlib`)
- Save the figure to a file:
  `plt.savefig(` *file name* `)`

Numpy and matplotlib: task

Task 2

In a text editor, add the following features to `euler.py`:

- Create the array `x_exact` so that $x_{exact,i} = e^{\sin(t_i)}$
- Calculate the RMS error, without using any for loop:

  $$\epsilon_{RMS} = \sqrt{\frac{1}{N}\sum_{i=0}^{N-1}(x_i - x_{exact,i})^2}$$

  Use the `print` statement, to show the value of the RMS error
- Plot x and x_{exact} as a function of t on the same figure, and show it to the screen. (Use `plot(t, x_exact, '--')` to show the exact solution with dashed lines.)

Run the script (`python euler.py`), to check that it works.

Python interpreter: Outline

1. Overview of the Python language
2. Python, numpy and matplotlib
3. Reusing code: functions, modules, classes
4. Faster computation: Forthon

Reusing code for the example problem

Example problem

Compare the results of Euler's method for different values of `N` (and thus of `dt`) **on the same plot**.

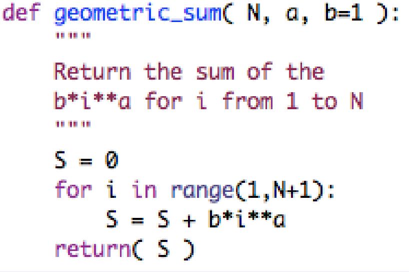
→ Not possible with the code from task 2
(unless we copy and paste a lot of code)

We need to make the code more **abstract** and **reusable**:

- Define **functions** that depend on `N` and initialize the arrays, perform Euler integration, and plot the results.
- Place these functions inside a **module** so that they can be imported and used elsewhere.

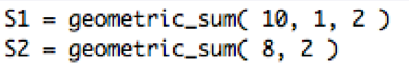
Functions

Example for function definition



```
def geometric_sum( N, a, b=1 ):
    """
    Return the sum of the
    b*i**a for i from 1 to N
    """
    S = 0
    for i in range(1,N+1):
        S = S + b*i**a
    return( S )
```

Example for function call



```
S1 = geometric_sum( 10, 1, 2 )
S2 = geometric_sum( 8, 2 )
```

- **Key syntax:** `def`, `()` and `:`, the body is indented
- The **"docstring"** is optional. Users can see it in **ipython** with `geometric_sum?` or `help(geometric_sum)`
- Here, `b` has a **default value**, which is used when only 2 arguments are given
- Functions can also return **several objects** (e.g. `return(x, a, b)`) or **nothing** (no `return` statement)
- Similarly, functions can be defined with **no arguments**

Modules

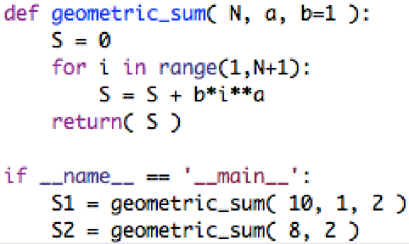
Module

Defines variables to be **imported** by **other Python sessions**.

- Any Python script can be treated as a module. `numpy` is a set of modules.
- The section `if __name__ == '__main__':` is executed if the script is **run** (e.g. `python geometric.py`) but not when it is **imported** (`import geometric as gm`)

Example module

In file `geometric.py`:



```
def geometric_sum( N, a, b=1 ):
    S = 0
    for i in range(1,N+1):
        S = S + b*i**a
    return( S )

if __name__ == '__main__':
    S1 = geometric_sum( 10, 1, 2 )
    S2 = geometric_sum( 8, 2 )
```

Example import and use

In e.g. `ipython`:

```
import geometric as gm
S = gm.geometric_sum(8, 2)
```

Importing modules

Different import styles:

- `import geometric`
  → `S = geometric.geometric_sum(8,2)`
- `import geometric as gm`
  → `S = gm.geometric_sum(8,2)`
- `from geometric import geometric_sum`
  or `from geometric import *` (imports all variables)
  → `S = geometric_sum(8,2)`

The source file of the module needs to be:

- in the same directory
- or in the default Python path
  (case of installed packages like `numpy`, `matplotlib` or even `warp`)

Functions and modules: task

Task 3

Reorganize the script `euler.py` so as to make it a reusable module:

- Start with the import statements (`numpy` and `matplotlib`)
- Write a function with signature `initialize_arrays(N, T=10.)` which sets `dt = T/N`, initializes `t` and `x`, and returns `t`, `x`, `dt`
- Write a function `euler_integration( t, x, dt, N )`, which fills the array `x` (this function does not return anything)
- Write a function `evaluate_result(t, x, N)`, which computes the exact result, prints the RMS error, and plots the arrays
- Finally, create a section `if __name__ == '__main__':`, in which you set `N = 200`, and call the 3 functions successively

Type `python euler.py` to check that the final section runs.

Functions and modules: task

Task 4

Use the module that you wrote, inside `ipython`

- In the shell, type `ipython --matplotlib`
- Then, inside `ipython`, type `from euler import *`
- Then set `N1 = 100`, `N2 = 200` and create the corresponding variables `t1,x1,dt1` and `t2,x2,dt2` with `initialize_array`.
- Then call `euler_integration` and `evaluate_result` on each set of arrays and values. Compare the results.

(NB: Do not hesitate to use tab completion in `ipython`)

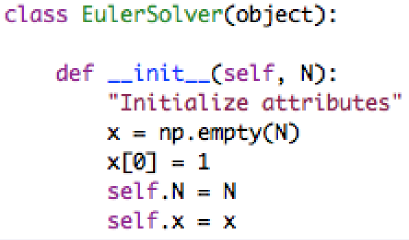
Although the code works, note that it is tedious to:

- create 4 different variables with a suffix `1` or `2`
- pass these variables as arguments to the different functions

This is solved by **object-oriented programming** and **classes**.

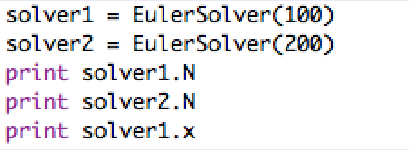
Classes: initialization and attributes

Example of class definition



```
class EulerSolver(object):

    def __init__(self, N):
        "Initialize attributes"
        x = np.empty(N)
        x[0] = 1
        self.N = N
        self.x = x
```

Example of use

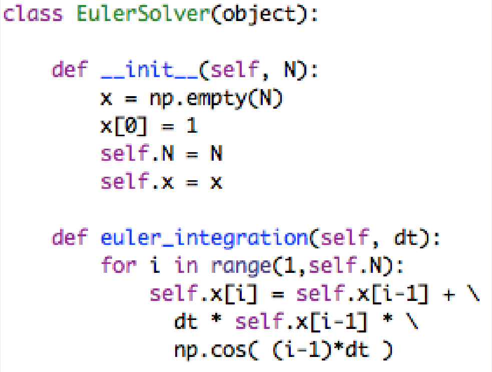


```
solver1 = EulerSolver(100)
solver2 = EulerSolver(200)
print solver1.N
print solver2.N
print solver1.x
```

- Classes are "**containers**": Variables are **encapsulated** together as **attributes** of an **instance** of the class.
- **Creation of an instance** (e.g. `EulerSolver(100)`) executes the code in `__init__`.
- **Accessing attributes** replace `self` by the name of the instance.
- **Predefined syntax:** Use the keywords `class`, `(object):` and `__init__` Note that `__init__` takes `self` as first argument when defined, but this is skipped when creating an instance.

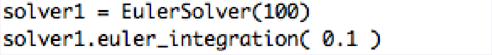
Classes: methods

Example of class definition



```
class EulerSolver(object):

    def __init__(self, N):
        x = np.empty(N)
        x[0] = 1
        self.N = N
        self.x = x

    def euler_integration(self, dt):
        for i in range(1,self.N):
            self.x[i] = self.x[i-1] + \
              dt * self.x[i-1] * \
              np.cos( (i-1)*dt )
```

Example of use



```
solver1 = EulerSolver(100)
solver1.euler_integration( 0.1 )
```

- **Methods** are functions which can access the **attributes** of a class.
  → The attributes do not need to be passed as arguments.
- **Syntax for definition**
  Pass `self` as first arguments, then use `self.` to access attributes
- **Syntax for calling**
  Prefix with name of the instance, then skip `self` in arguments

Classes: task

Task 5

Rewrite `euler.py` so as to define a class `EulerSolver`

- Replace the function `initialize_arrays` by a corresponding method `__init__(self, N, T=10.)` This method should define `N, x, t, dt` as attributes.
- Replace the functions `euler_integration` and `evaluate_result` by methods with the same name respectively. These methods should take no argument (besides `self`), but should use the attributes through the `self.` syntax.
- In the section `if __name__ == '__main__':`, type the code:

```
solver = EulerSolver( 200 )
solver.euler_integration()
solver.evaluate_result()
```

Execute the file (`python euler.py`) to check that it works.
Then in `ipython`, compare again N=100 and N=200.

Python interpreter: Outline

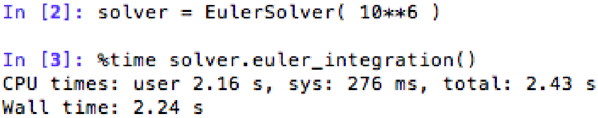
1. Overview of the Python language
2. Python, numpy and matplotlib
3. Reusing code: functions, modules, classes
4. Faster computation: Forthon

Faster computation

Problem

Large `for` loops are slow in Python.

Example:



```
In [2]: solver = EulerSolver( 10**6 )

In [3]: %time solver.euler_integration()
CPU times: user 2.16 s, sys: 276 ms, total: 2.43 s
Wall time: 2.24 s
```

Solution

- If the operation is of type **element-wise** or **reduction**: Use **numpy** syntax
- Otherwise, rewrite the `for` loop in a **compiled** language (e.g. Fortran, C) and link it to the rest of the Python code

→ **High-level control** with Python (modularity, interactivity)
→ **Low-level number-crunching** with e.g. Fortran or C (efficiency)

Faster computation: Forthon

Forthon

- Generates links between Fortran and Python
- Open-source, created by D. P. Grote (LLNL)
  `https://github.com/dpgrote/Forthon`
- Heavily used in Warp for low-level number crunching

On the user side:

- Write Fortran **subroutines** and **modules** in a `.F` file
- Write a `.v` file to tell which variables to link to Python
- Compile with Forthon → produces a Python module
- Import the module in Python and use the linked variables

NB: Other similar solutions exist: `f2py` (links Fortran code), `Cython` (generates and links C code), `Numba` (compiles Python code), etc...

Faster computation: task

Task 6

Download and decompress the code from
http://github.com/RemiLehe/uspas_exercise/raw/master/Forthon_task.tgz
The files `acc_euler.F` and `acc_euler.v` are the files needed by Forthon, while `euler.py` is the code from task 5.

- The Fortran file `acc_euler.F` contains an error in the line that starts with `x(i) =` . Spot it and correct it.
- Compile the code with Forthon by typing `make` in the shell. A new file `acc_eulerpy.so` should be created.
- At the beginning of the file `euler.py`, add `from acc_eulerpy import forthon_integration` then create a new method `acc_euler_integration(self)`, which calls `forthon_integration` (see `acc_euler.F` for its signature).

In `ipython`, create an instance with `N=10**6`, and compare the runtime of `euler_integration` and `acc_euler_integration`

References

Scipy lecture notes:
`http://www.scipy-lectures.org/` (G. Varoquaux et al., 2015)

Python tutorial:
`https://docs.python.org/3/tutorial/` (Python Software foundation, 2016)

Forthon:
`https://github.com/dpgrote/Forthon` (D. Grote et al., 2016)