Simulation and modeling of natural processes

Week 2: Introduction to programming with Python 3

Orestis Malaspinas

Introduction to high performance computing for modeling

Why do we need programming in this course?

- We want to represent nature through complex mathematical models.
- These models are too complex to be solved analytically.
- Either they do not possess analytical solutions or it would take too long to compute it.
- Numerical methods exist to approximate the mathematical models used to represent nature.
- Computers are very efficient in doing large amount of computations.
- We need an efficient way to make the computer use the numerical methods.

Computers are fast but... we always want more

For a computer model to be satisfying it must...

- Represent with good accuracy the process it is supposed to model.
- Give a solution in a reasonable amount of time.

For a program to be faster one can...

- Wait for the hardware to become faster (Moore's law).
- Optimize the algorithm (bubble sort, quick sort).
- Optimize the way algorithm implemented.

$\boxed{\textit{End of module}}$

Introduction to high performance computing

$\boxed{\textit{Coming next}}$

Concepts of code optimization

Concepts of code optimization

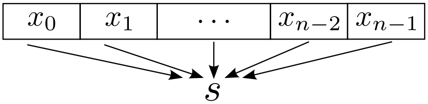
Computers are fast but... we always want more

Example: sum of the numbers stored in an array.

Mathematical representation

$$A = \{x_i\}_{i=0}^{n-1},\ x_i \in \mathbb{N}, \quad s = \sum_{i=0}^{n-1} x_i.$$

Graphical representation

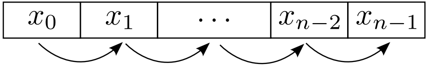


How do we implement this on a computer?

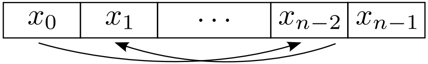
Computers are fast but... we always want more

Performed in 3 different ways with Python 3 (there are more)

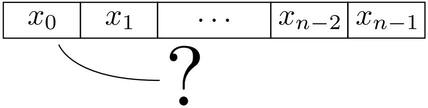
- The array is read linearly.



- The array is read randomly.



- The sum is performed with `sum`.



Benchmark: Sum of $n = 10^6$ integers

Efficiency depends on the algorithm.

	linear	random	sum
time [s]	0.0594	0.270	0.00654
t/t_{sum}	9.08	41.3	1

Efficiency depends on the programming language

	Python 3 (sum)	NumPy	C++ (linear)
time [s]	0.00654	0.000634	0.000478
t/t_{C++}	13.7	1.3264	1

By coding naively C++ is 100 times faster than Python but with NumPy the performance is equivalent.

$\boxed{\textit{End of module}}$

Concepts of code optimization

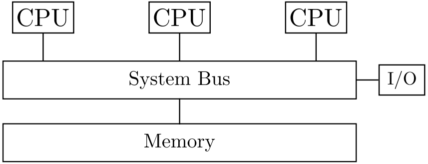
$\boxed{\textit{Coming next}}$

Concepts of parallelism

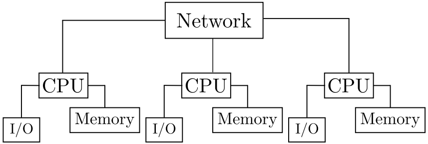
Concepts of parallelism

A computer is fast... many computers together are faster

Shared memory (OpenMP)



Distributed memory (MPI)



Performance

- One modern processor $\sim$ 100 Gflops $= 10^{11}$ flops.
- The fastest supercomputer (as of 4.1.2015) $\sim$ 52 Pflops $= 52 \cdot 10^{18}$ flops.

A computer is fast... many computers together are faster

Not all problems are parallelizable

Parallelizable problem

Can be decomposed into pieces that can be executed simultaneously.

Non-parallelizable problem

Fibonacci series computation

$$F(i) = F(i-1) + F(i-2),$$
$$F(1) = 1,\ F(2) = 1.$$

$\boxed{\textit{End of module}}$

Concepts of parallelism

$\boxed{\textit{Coming next}}$

Palabos, a parallel lattice Boltzmann solver

Palabos, a parallel lattice Boltzmann solver

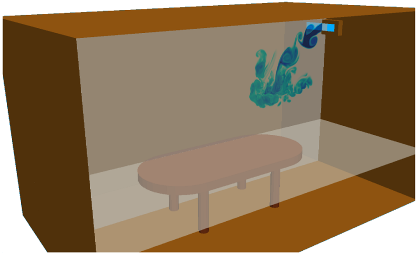
The Palabos library

Palabos, `http://www.palabos.org`

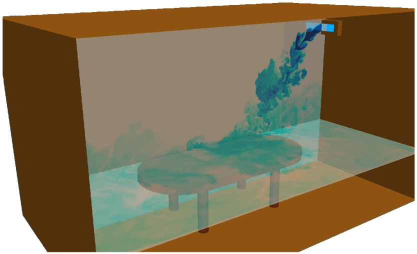
- C++ parallel open-source library.
- Parallelism completely transparent.
- Fluid flow solver based on LBM.
- Multi-physics modules.

Example: Air-conditioning

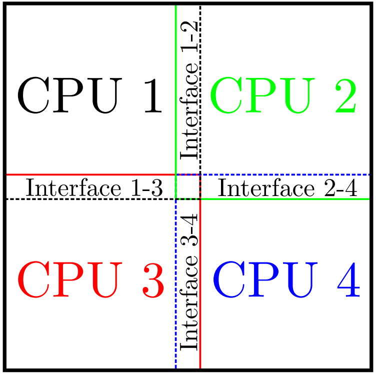
After 8 seconds



After 2 minutes



Parallelism in Palabos: Multi-Block approach



- Spatial domain cut into many blocks.
- Blocks are dispatched on different CPUs.
- Communication is done through the interfaces.
- Efficient if communication $\ll$ computation and the load on each processor is well balanced.

$\boxed{\textit{End of module}}$

Palabos, a parallel lattice Boltzmann solver

$\boxed{\textit{Coming next}}$

An introduction to Python 3

An introduction to Python 3

Introduction to Python 3

- Multi-platform programming language used in this course (along with NumPy).
- With NumPy good trade-off between efficiency and simplicity.
- Large collection of advanced open-source libraries.
- Only Python 3 course (not compatible with Python 2).
- Introduction intended for beginners.
- Prerequisite installation of Python 3 (see `https://www.python.org/downloads/`) and NumPy (see `http://www.scipy.org/scipylib/download.html/`).

Literature on Python 3

- Python 3 documentation : `https://docs.python.org/3/`
- Dive into Python 3: `http://www.diveintopython3.net/`
- Rapid GUI Programming with Python and Qt: The Definitive Guide to PyQt Programming, Mark Summerfield, ISBN-10: 0132354187 – ISBN-13: 978-0132354189
- NumPy documentation : `http://docs.scipy.org/doc/numpy/index.html`

Interpreted vs Compiled languages

Interpreted

- Program is directly run by the interpreter.
- The interpreter translates a code into its subroutines that are precompiled.
- Examples: Python, JavaScript, Ruby, ...

Compiled

- Program transformed into machine code and saved into an executable file.
- The executable file can then be run.
- Examples: C/C++, Fortran, ...

Interpreted vs Compiled languages

Advantages of each approach (only tendencies)

- Compiled code tends to be faster.
- Interpreted code tends to be easier and faster to develop.
- Interpreted code tends to be more portable.
- Interpreted code can be dynamically typed.

$\boxed{\textit{End of module}}$

An introduction to Python 3

$\boxed{\textit{Coming next}}$

Running a Python program

Running a Python program

The interactive shell

Python 3 interactive shell

```
modeling@mooc:~$ python3
Python 3.4.2 (default, Oct  8 2014, 10:45:20)
[GCC 4.9.1] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Hello World!

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

Calculator

```
>>> 4+8*12
100
>>> 56+3*90
326
>>>
```

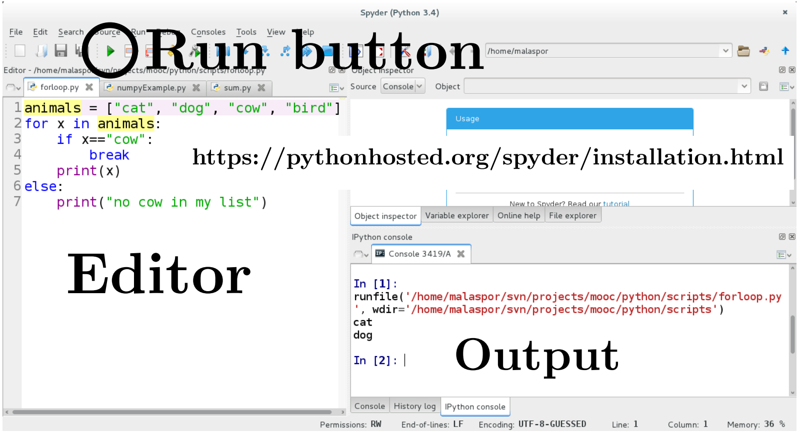
Errors

```
>>> print(Hello World!)
File "<stdin>", line 1
  print(Hello World!)
                  ^
SyntaxError: invalid syntax
>>>
```

Quit

```
>>> quit()
modeling@mooc:~$
```

Spyder 3



Spyder (Python 3.4)
File Edit Search Source Run Debug Consoles Tools View Help
Run button
/home/malaspor
Editor - /home/malaspor/svn/projects/mooc/python/scripts/forloop.py
forloop.py
numpyExample.py
sum.py
animals = ["cat", "dog", "cow", "bird"]
for x in animals:
if x=="cow":
break
print(x)
else:
print("no cow in my list")
Editor
Object inspector
Source Console
Object
Usage
https://pythonhosted.org/spyder/installation.html
New to Spyder? Read our tutorial
Object inspector Variable explorer Online help File explorer
IPython console
Console 3419/A
In [1]:
runfile('/home/malaspor/svn/projects/mooc/python/scripts/forloop.py', wdir='/home/malaspor/svn/projects/mooc/python/scripts')
cat
dog
In [2]:
Output
Console History log IPython console
Permissions: RW End-of-lines: LF Encoding: UTF-8-GUESSED Line: 1 Column: 1 Memory: 36 %

Running a Python script

Needed when programs become too long to use the interpreter.

- Create a file "`myscript.py`" and edit it with your favorite text editor or IDE.
- Add the following line in it

```
print("Hello World!")
```

With text editor

In terminal or command prompt:

```
modeling@mooc:~$ python3 myscript.py
Hello World!
```

With Spyder

Hit the run button!

```
runfile('~/myscript.py', wdir='~/')
Hello World!
```

$\boxed{\textit{End of module}}$

Running a Python program

$\boxed{\textit{Coming next}}$

Variables and data types

Variables and data types

Variables

Container to store values that can accessed and changed.

In Python (not the case in C/C++, Java for example):

- No need to declare the data type of a variable.
- Variables can change data type.

Example

```
1 a = 10
2 print(a)
3                     10
4 a = a+1
5 print(a)
6                     11
7 a = "Hello World!"
8 print(a)
9                     Hello World!
```

Discussion

1: The value 10 is stored in `a` (`a` is the identifier).

4: `a` is taking a new value which is `a+1`.

7: The string “Hello World!” is stored in `a`.

Variables: identifiers

Rules for variable identifiers

1. The first character can be ANY Unicode letter or the underscore.
2. The other characters can be ANY Unicode letter or digit or the underscore.
3. All the Python keywords are forbidden.
4. Identifiers are case sensitive.

Examples

```
>>> àél3_aerkcàéal = 10
>>> 2àél3_aerkcàéal = 10
  File "<stdin>", line 1
    2él3_aerkcàéal = 10
                  ^
SyntaxError: invalid syntax
```

```
>>> True = 10
  File "<stdin>", line 1
SyntaxError: can't assign to keyword
>>> a,A = 10,20
>>> print(a,A)
10 20
```

Data types

Built-in types

Numbers

1. Integers (arbitrary size)

```
>>> type(2378)
<class 'int'>
```

2. Floating-point numbers

```
>>> type(1.23456e-6)
<class 'float'>
```

3. Complex numbers

```
>>> type(3+6j)
<class 'complex'>
```

Strings

Immutable chain of Unicode characters.

```
>>> 'Single line strings defined with single quotes'
'Single line strings defined with single quotes'
>>> "Or with double quotes"
'Or with double quotes'
>>> c = '''Triple quotes for multiline
... and can contain ' or " '''
>>> c[3] = "a"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Data types (continued)

Built-in types

Other types

1. Boolean type

```
>>> print(type(True),type(False))
<class 'bool'> <class 'bool'>
```

2. None Type

```
>>> type(None)
<class 'NoneType'>
```

Examples

```
>>> type(True)
<class 'bool'>
>>> bool("")
False
>>> bool("hi")
True
>>> bool(0)
False
>>> bool(10)
True
>>> print(True+True,True+False)
2 1
```

Data types (continued)

Built-in types

Sequence Types

1. `list` : mutable sequence.

```
>>> type([1,’is an int’,2.0,’is a float’])
<class ’list’>
```

2. `tuple` : immutable sequence.

```
>>> type((1,’is an int’,2.0,’is a float’))
<class ’tuple’>
```

3. `range` : immutable number sequence.

```
>>> type(range(10))
<class ’range’>
```

Examples

```
>>> a = [1,7,29,1,39]; a[2]
29
>>> a[3] = "a"; print(a)
[1, 7, 29, ’a’, 39]
>>> b=(’njd’,[1,1],1.0)
>>> b[0] = 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: ’tuple’ object does not
support item assignment
>>> b[1][0] = 4
>>> print(b)
(’njd’, [4, 1], 1.0)
>>> print(list(range(4)))
[0, 1, 2, 3]
```

$\boxed{\textit{End of module}}$

Variables and data types

$\boxed{\textit{Coming next}}$

Operators

Operators

Operators: on numeric types

Operators (numeric type)

- Addition `+`, Subtraction `-`, Multiplication `*`.
- Real division `/`, Integer division `//`, Modulo `%`.
- Exponentiation `**` .
- Unary `+`, `-`.

Examples

```
>>> 12+15-37
-10
>>> 12*15*37
6660
>>> 10/3
3.3333333333333335
>>> 10//3
3
>>> 10%3
1
>>> 2.0**10
1024.0
>>> +2, -2
(2, -2)
```

Operators: on sequence types

Operators (sequence types)

- Concatenation + .
- Copies of a sequence $\star$.
- Access to element [] .
- Slice : .

Examples

```
>>> a = [1,2,3,4,"b"]
>>> a[0]
1
>>> a[0] = 9; print(a)
[9, 2, 3, 4, 'b']
>>> a+a
[9, 2, 3, 4, 'b', 9, 2, 3, 4, 'b']
>>> 2*a
[9, 2, 3, 4, 'b', 9, 2, 3, 4, 'b']
>>> a[1:4]
[2, 3, 4]
>>> a[0:4:2]
[9, 3]
>>> a[2:]
[3, 4, 'b']
>>> a[:2]
[9, 2]
```

Operators: Boolean operations and comparisons

Operators

- `<`, `<=`, `>`, `>=`, `==`, `!=`.
- `or`, `and`, `not` .
- `in` or `not in` a sequence.

Remark

- `or` and `and` always return one of their operands.

Examples

```
>>> 2<1
False
>>> a,b = 1,2
>>> a == b
False
>>> 1 != 2
True
>>> 1 != 2 and 1 == 1
True
>>> 1 != 2 and not 1 == 1
False
```

More examples

```
>>> 2 or 3
2
>>> 2 and 3
3
>>> "a" == "a"
True
>>> "a" > "b"
False
>>> "a" and "b"
'b'
>>> a = [1,2,3,4,"b"]
>>> "b" in a
True
```

$\boxed{\textit{End of module}}$

Operators

$\boxed{\textit{Coming next}}$

Control structures: conditional statements

Control structures: conditional statements

Control structures: conditional statements

Conditional branching based on boolean information.

The if statement

```
if condition_1:
    statements_1
[elif condition_2:
    statements_2
...
elif condition_n:
    statements_n]
<else:
    statements>
```

Remarks

- There can be zero or more `elif`.
- There can be at most one `else`.
- The statements can be of more than one line.
- Indentation is important!

```
if condition:          if condition:
    statement_1            statement_1
statement_2                statement_2
```

Conditional statements (example)

Velocity on the motorway

```
x = int(input("Enter your velocity : "))

if x >= 120:
    y = x-120
    x = 120
    print("Going too fast. Velocity reduced by ", y)
elif x <= 80:
    y = 80-x
    x = 80
    print("Going too slow. Velocity increased by ", y)
else:
    print("Everything is fine.")
```

Output 1

```
Enter your velocity : 60
Going too slow. Velocity increased by 20
```

Output 2

```
Enter your velocity : 151
Going too fast. Velocity reduced by 31
```

Output 3

```
Enter your velocity : 89
Everything is fine.
```

Control structures: conditional statements

The ternary (or inline) if

```
a = x if condition else y
```

Remark

- `x` is returned if `condition` is `True`, else `y` is returned.
- No `elif` allowed.

Example

```
x = int(input("Input an integer: "))
a = "even" if x%2 == 0 else "odd"
print(x, " is an ", a, " number.")
```

Output

```
Input an int: 10
10 is an even number.

Input an int: 15
15 is an odd number.
```

$\boxed{\textit{End of module}}$

Control structures: conditional statements

$\boxed{\textit{Coming next}}$

Control structures: loops

Control structures: loops

Control structures: for loop

Loop over any kind of ordered sequence.

The for statement

```
for variable in sequence:
    statements_1
[else:
    statements_2]
```

Remarks

- Indentation is important!
- The `sequence` can be of any type.
- If `sequence` is numbers use `range`.
- The `else` statement is optional.
- `break` statement exits the loop.

Example

```
animals = ["cat", "dog", "cow", "bird"]
for x in animals:
    if x=="cow":
        break
    print(x)
else:
    print("no cow in my list")

cat
dog
```

Control structures: while loop

Perform an action as long as a condition is satisfied.

The while statement

```
while condition:
    statements_1
[else:
    statements_2]
```

Remarks

- Indentation is important!
- The `else` statement is optional.
- The loop can be exited with the `break`.

Example

```
x = int(input("Enter a number : "))
ary = list(range(x))
tot = 0
i = 0
while i < len(ary):
    tot += ary[i]
    i += 1
else:
    print("Total ended normally.")

print("The sum is: ",tot)
```

Output

```
Enter a number : 5
Total ended normally.
The sum is:  10
```

$\boxed{\textit{End of module}}$

Control structures: loops

$\boxed{\textit{Coming next}}$

Functions

Functions

Functions

Definition and use

- Provided some input parameters compute one or more results ($f : x \to \sqrt{x}$).
- More generally a set of statements that can be reused in a program.
- A basic building block of a program.

Syntax

```
def function_name(parameter_list):
    [""" comments (doc) """]
    statements
    [return]
```

Remarks

- The `return` and `comments` statements are optional.
- The `return` statement ends the function and returns the result.
- Functions always return `None` by default.
- The number of parameters can be zero or more.
- Indentation is important.

Functions (example)

Code

```
def isPrime(number):
    """Check if number is prime"""
    if number <= 1:
        return False

    for i in range(2,number):
        if number % i == 0:
            return False
    return True

help(isPrime)
for j in range(0,13):
    if (isPrime(j)):
        print(j, " is a prime number")
    else:
        print(j, " is not a prime number")
```

Result

```
isPrime(number)
    Check if number is prime

0  is not a prime number
1  is not a prime number
2  is a prime number
3  is a prime number
4  is not a prime number
5  is a prime number
6  is not a prime number
7  is a prime number
8  is not a prime number
9  is not a prime number
10  is not a prime number
11  is a prime number
12  is not a prime number
```

Functions: Scope of variables

Variables have the scope of the block they are declared in and can be used after their point of declaration.

Global variable

```
def foo():
    print(a)

a = "Hello!"
foo()
Hello!
```

Local variable

```
def bar():
    a="Hi!"
    print(a)

a = "Hello!"
bar()
Hi!
```

Error

```
def baz():
    print(a)
    a="Hi!"
    print(a)

a = "Hello!"
baz()
Traceback (most recent call last):
  File "scope.py", line 27, in <module>
    foo()
  File "scope.py", line 22, in foo
    print(a)
UnboundLocalError: local variable 'a'
referenced before assignment
```

Functions: Mutability/Immutability

Number

```
def foo(a):
    a=4
    print(a)

a=2
foo(a)
print(a)
4
2
```

List

```
def bar(a):
    a[1]=-2
    print(a)

a=[2,4,5]
bar(a)
print(a)
[2, -2, 5]
[2, -2, 5]
```

String, Tuple

```
def baz(a):
    a[1]="n"
    print(a)

a="abcdef"
baz(a)
print(a)
Traceback (most recent call last):
  File "immutable_mutable.py",
  line 37, in <module>
    foo(a)
  File "immutable_mutable.py",
  line 33, in foo
    a[1]="n"
TypeError: 'str' object does not support
item assignment
```

Modules

Modular programming

- Complex programs can be separated into many simple parts.
- Parts can be assembled to construct different programs.

Creation and use

- Saving the functions into `.py` files.
- Using the `import...[as]...` keyword.

Example

- Save `isPrime` function into a file named `checkPrime.py`.

```
>>> import checkPrime
>>> checkPrime.isPrime(8)
False
```

- Using alias

```
>>> import checkPrime as cp
>>> cp.isPrime(8)
False
```

- All math functions are in `math` module

$\boxed{\textit{End of module}}$

Functions

$\boxed{\textit{Coming next}}$

NumPy

NumPy

NumPy

- Open-source library (module) extension to Python.
- Fast precompiled numerical routines.
- Support for large multi-dimensional arrays (Matlab-like).
- Linear algebra, Fourier transform, and random number features.
- Integration of C/C++ or Fortran code.
- With SciPy even more advanced mathematical functions.
- Installation guide on `http://www.numpy.org/`.
- If using Spyder NumPy is already available.

NumPy (continued)

Example

```
import time
import numpy

def list_ver(length):
    X = list(range(length))
    Y = list(range(length))
    t1 = time.time()
    Z = [0]*length
    for i in range(length):
        Z[i] = X[i] + Y[i]
    return time.time() - t1

def numpy_ver(length):
    X = numpy.arange(length)
    Y = numpy.arange(length)
    t1 = time.time()
    Z = X + Y
    return time.time() - t1
```

Execution

```
l = 10000000
print("t List  = ",list_ver(l))
print("t NumPy = ",numpy_ver(l))
t List   =  0.999732255935669
t Numpy =  0.029507875442504883
```

Discussion

- Creation of two arrays (`X`, `Y`) of size `10000000`.
- `X` and `Y` are summed and the result is stored in `Z`.
- These sums are performed with Lists or NumPy arrays.
- The performance is measured for the two cases.
- With NumPy no `for` loop and more efficient.

NumPy: ndarray manipulations

Matrix-Scalar manipulations

Example NumPy

```
import numpy as np

X = np.ones((2,2))
print(1+2.5*X)
[[ 3.5  3.5]
 [ 3.5  3.5]]
```

Example List

```
Y = [[0]*2, [0]*2]
X = [[1]*2, [1]*2]
for i in range(2):
    for j in range(2):
        Y[i][j]=1+2.5*X[i][j]

print(Y)
[[ 3.5  3.5], [ 3.5  3.5]]
```

NumPy applies the multiplication to the whole array.

NumPy: ndarray manipulations

Matrix-Matrix manipulations

Example NumPy

```
import numpy as np

X = np.arange(4).reshape(2,2)
Y = np.arange(4).reshape(2,2)
Z = X+Y
print(Z)
print(Z*Z)
[[0 2]
 [4 6]]
[[ 0  4]
 [16 36]]
```

Example List

```
X = [list(range(2)), list(range(3,5))]
Y = [list(range(2)), list(range(3,5))]
Z = [[0]*2, [0]*2]
for i in range(2):
    for j in range(2):
        Z[i][j]=X[i][j]+Y[i][j]
print(Z)
ZZ = [[0]*2, [0]*2]
for i in range(2):
    for j in range(2):
        ZZ[i][j]=Z[i][j]*Z[i][j]
print(ZZ)
[[0, 2], [4, 6]]
[[0, 4], [16, 36]]
```

The addition/multiplication element-wise to the whole array.

NumPy: linalg

Basic linear algebra

```
>>> import numpy as np
>>> A = np.arange(4).reshape(2,2)
>>> b = np.arange(1,3)
>>> print(A,b,np.dot(A,b))
[[0 1]
 [2 3]] [1 2] [2 8]
>>> invA = np.linalg.inv(A)
>>> print(np.dot(invA,A))
[[ 1.  0.]
 [ 0.  1.]]
>>> print(np.dot(invA,b))
[-0.5  1. ]
>>> print(np.linalg.solve(A,b))
[-0.5  1. ]
```

And many more

- Determinant, Condition number.
- Cholesky decomposition, QR decomposition.
- Eigenvalue decomposition.

Among others.

$\boxed{\textit{End of module}}$

NumPy

$\boxed{\textit{End of Week 2}}$

Thank you for your attention!