Modeling and Simulation

Dr. Belkacem KHALDI
b.khaldi@esi-sba.dz
ESI-SBA, Algeria

Level: 2nd SC Class
Date:February 3, 2025

Lecture 2:

Introduction to Programming with Python 3 for Scientific Computing

Why do we need programming in this course?

- We want to represent systems through modeling techniques such as 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 systems.
- 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.
- Optimize the algorithm (e.g.: quick sort).
- Optimize the way algorithm is implemented.

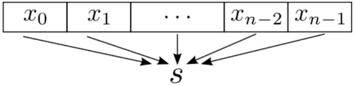
Concepts of code optimization

Example: Sum of the numbers stored in an array.

Mathematical representation

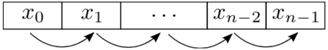
$$A = \{x_i\}_{i=0}^{n-1},\ x_i \in \mathbb{N}, \quad s = \sum_{i=0}^{n-1} x_i.$$

Graphical representation



This can be Performed in 3 different ways:

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	1.0

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	1.0

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

Why Python?

- **Python** is a modern, general-purpose, object-oriented, high-level programming language.
- **clean and simple language:** Easy-to-read and intuitive code, easy-to-learn minimalistic syntax, maintainability scales well with size of project.
- **expressive language:** Fewer lines of code, fewer bugs, easier to maintain
- **Extensive ecosystem:** many scientific libraries and environments could enhance its capability (e.g.: Numpy, Scipy, matplotlib)
- **Great performance** due to close integration with time-tested and highly optimized codes written in C.

Literature on Python 3

- Python 3 documentation : https://docs.python.org/3/
- Dive into Python 3: http://www.diveintopython3.net/
- numpy: http://numpy.scipy.org - Numerical Python.
- scipy: http://www.scipy.org - Scientific Python.
- matplotlib: http://www.matplotlib.org - graphics library

Part I

Generalities on Python 3

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]
```

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 * .
- 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
```

```
>>> 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
```

Control structures: conditional statements

The if statement:

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

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

Examples

```
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.")
```

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

Control structures: loops

The for statement:

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

Examples

```
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: loops

The while statement:

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

Examples

```
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)
```

Functions

Syntax:

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

Remarks

- Functions Provided some input parameters compute one or more results (e.g.: $f : x \longrightarrow \sqrt{x}$).
- The *return* and *comments* statements are optional.
- The *return* statement ends the function and returns.
- Functions always return *None* by default.

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
```

Part II

Introduction to Numpy

NumPy
Base N-dimensional array package

NumPy

- Extension package to Python for multi-dimensional **arrays**.
- Build around high-performance multi-dimensional array data structures.
- Closer to hardware (efficiency).
- Designed for scientific computation (convenience).
- Also known as array oriented computing.

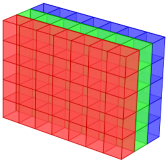
Efficiency vs python array

```
In [1]: L = range(1000)
In [2]: %timeit [i**2 for i in L]
1000 loops, best of 3: 403 us per loop
In [3]: a = np.arange(1000)
In [4]: %timeit a**2
100000 loops, best of 3: 12.7 us per loop
```

An array could represent:

- Values of an experiment/simulation at discrete time steps
- Signal recorded by a measurement device, e.g. sound wave
- Pixels of an image, grey-level or colour
- 3-D data measured at different X-Y-Z positions, e.g. MRI scan
- ...

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}$$



colour image as $N \times M \times 3$-array

Numpy: The basics

Import conventions:

```python
import numpy as np
```

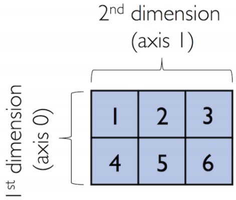
Example of arrays:

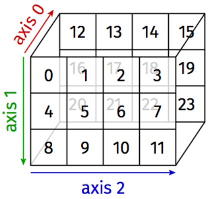
- **1-D:**

```python
    a = np.array([0,1,2,3])
```

- **2-D, 3-D, ...:**

```python
    # 2 x 3 array
    b = np.array([[1,2,3],
                  [4,5,6]])
```






```
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]],

       [[12, 13, 14, 15],
        [16, 17, 18, 19],
        [20, 21, 22, 23]]])
```

Numpy: Array Construction Routines

Homogeneous data

- **Zeros:**

```
a=np.zeros((3, 3))
```

- **Ones:**

```
b=np.ones((3, 3))
```

- **diag:**

```
c=np.diag(np.array([1, 2, 3]))
```

- **eye:**

```
d=np.eye(3)
```

a

0	0	0
0	0	0
0	0	0

b

1	1	1
1	1	1
1	1	1

c

1	0	0
0	2	0
0	0	3

d

1	0	0
0	1	0
0	0	1

Numpy: Array Construction Routines

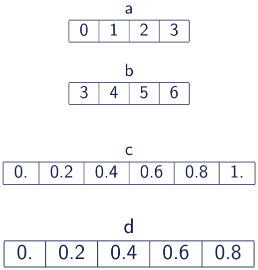
Numerical ranges

- **Evenly spaced:**

```python
    a=np.arange(4) #0 .. n-1  (!)
    b=np.arange(3,7) #start, end-1
```

- **By number of points:**

```python
    #start, end, num, endpoint
    c=np.linspace(0, 1, 6)
    d=np.linspace(0, 1, 6, endpoint=False)
```



Numpy: Array Construction Routines

Basic data types

```
a=np.array([1, 2, 3])
a.dtype
```

- **dtype('int64')**

```
b=np.array([1., 2., 3.])
b.dtype
```

- **dtype('float64')**

```
c=np.array([1, 2, 3], dtype=float)
c.dtype
```

- **dtype('float64')**

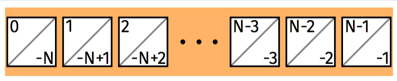
There are also other types:

- Complex.
- Bool.
- Strings.
- int32.
- uint32.
- uint64.

Indexing and slicing

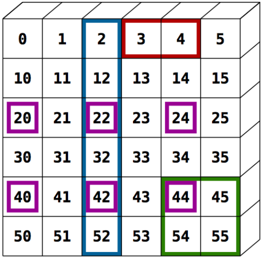
Basics

- Indexing starts at 0.
- Negative indices count from the end of the list to the beginning

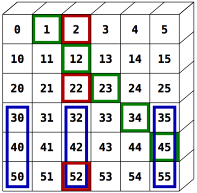


- Basic slices syntax: [start:stop:step]
- if values are omitted:
  - start: starting from 1st element.
  - stop: until (and including) the last element.
  - step: all elements between start and stop-1.

```
>>> a[0, 3:5]
array([3, 4])

>>> a[4:, 4:]
array([[44, 55],
       [54, 55]])

>>> a[:, 2]
a([2, 12, 22, 32, 42, 52])

>>> a[2::2, ::2]
array([[20, 22, 24],
       [40, 42, 44]])
```




```
>>> a[(0,1,2,3,4), (1,2,3,4,5)]
array([1, 12, 23, 34, 45])

>>> a[3:, [0,2,5]]
array([[30, 32, 35],
       [40, 42, 45],
       [50, 52, 55]])

>>> mask = np.array([1,0,1,0,0,1], dtype=bool)
>>> a[mask, 2]
array([2, 22, 52])
```



Numerical operations on arrays

Elementwise operations

- **With scalars:**

```
a=np.array([1, 2, 3, 4])
a=a + 1
a=2**a
```

a

1	2	3	4

a

2	3	4	5

a

4	9	16	25

- **Arithmetic operates elementwise:**

```
b=np.ones(4) + 1
b=a - b
b=a * b
```

b

2	2	2	2

b

2	7	14	23

b

8	63	224	575

- **Warning: * NOT matrix multiplication!**

```
c = np.ones((3, 3))
c=c*c #NOT matrix multiplication!
```

Numerical operations on arrays

Dot Product

- **ndarray:**

```
x=np.array([1,2,3])
y=np.array([-7,8,9])
dot=np.dot(x,y)#dot=36
```

- **matrix Class:**

```
x=np.matrix([[2, 3], [3, 5]])
y=np.matrix([[2, 2], [5, -1]])
z=x * y
```

x

1	2	3

y

-7	8	9

x

2	3
3	5

y

2	2
5	-1

z

17	1
28	1

Numerical operations on arrays

Universal functions (ufuncs)

Trigonometric functions
sin, cos, tan, arcsin, arccos, arctan, hypot, arctan2, degrees, radians, unwrap, deg2rad, rad2deg

Hyperbolic functions
sinh, cosh, tanh, arcsinh, arccosh, arctanh

Rounding
around, round_, rint, fix, floor, ceil, trunc

Sums, products, differences
prod, sum, nansum, cumprod, cumsum, diff, ediff1d, gradient, cross, trapz

Exponents and logarithms
exp, expm1, exp2, log, log10, log2, log1p, logaddexp, logaddexp2

Other special functions
i0, sinc

Floating point routines
signbit, copysign, frexp, ldexp

Arithmetic operations
add, reciprocal, negative, multiply, divide, power, subtract, true_divide, floor_divide, fmod, mod, modf, remainder

Handling complex numbers
angle, real, imag, conj

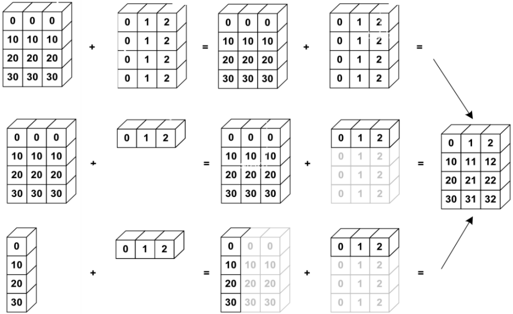
Miscellaneous
convolve, clip, sqrt, square, absolute, fabs, sign, maximum, minimum, fmax, fmin, nan_to_num, real_if_close, interp

Numerical operations on arrays

Broadcasting

- Basic operations on numpy arrays (addition, etc.) are elementwise.
- This works on arrays of the same size.
- It's also possible to do operations on arrays of different sizes if NumPy can transform these arrays so that they all have the same size: this conversion is called broadcasting.
- e.g:

```
np.array([1, 2, 3]) + 1:
```



Numerical operations on arrays

Array shape manipulation

- **Flattening:**

```
a=np.array([[1,2,3],[4,5,6]])
a.ravel()
```

- **Reshaping:** The inverse operation to flattening

```
b=a.ravel()
c=b.reshape((2, 3))
```

a

1	2	3
4	5	6

a.ravel()

1	2	3	4	5	6

b

1	2	3	4	5	6

c

1	2	3
4	5	6

Summary

Tips

What do you need to know to get started?:

- Know how to create arrays : array, arange, ones, zeros.
- Know the shape of the array with array.shape, then use slicing to obtain different views of the array: array[::2], etc. Adjust the shape of the array using reshape or flatten it with ravel.
- Obtain a subset of the elements of an array and/or modify their values.
- Know miscellaneous operations on arrays, such as finding the mean or max (array.max(), array. mean()). No need to retain everything, but have the reflex to search in the documentation (online docs, help(), lookfor())!!
- For advanced use: master the indexing with arrays of integers, as well as broadcasting. Know more NumPy functions to handle various array operations.

Part III

Introduction to SciPy

SciPy

- Extension package contains various toolboxes dedicated to common issues in scientific computing.
- Built on numpy. All functionality from numpy seems to be available in scipy as well.
- Composed of task-specific sub-modules such as interpolation, integration, optimization, image processing, statistics, special functions, etc..

SciPy sub-modules:

`scipy.cluster`	Vector quantization / Kmeans
`scipy.constants`	Physical and mathematical constants
`scipy.fftpack`	Fourier transform
`scipy.integrate`	Integration routines
`scipy.interpolate`	Interpolation
`scipy.io`	Data input and output
`scipy.linalg`	Linear algebra routines
`scipy.ndimage`	n-dimensional image package
`scipy.odr`	Orthogonal distance regression
`scipy.optimize`	Optimization
`scipy.signal`	Signal processing
`scipy.sparse`	Sparse matrices
`scipy.spatial`	Spatial data structures and algorithms
`scipy.special`	Any special mathematical functions
`scipy.stats`	Statistics

Numerical integration: scipy.integrate

Function integrals

The most generic integration routine is **scipy.integrate.quad()**.

For example, to compute $\int_0^{\pi/2} sin(t)dt$:

```
from scipy.integrate import quad
res, err = quad(np.sin, 0, np.pi/2)
# res is the result, is should be close to 1
np.allclose(res, 1)
# err is an estimate of the err
np.allclose(err, 1 - res)
```

Other integration schemes are available: **scipy.integrate.fixed_quad()**, **scipy.integrate.quadrature()**, **scipy.integrate.romberg()**...

Numerical integration: scipy.integrate

Integrating differential equations

scipy.integrate also features routines for integrating Ordinary Differential Equations (ODE). In particular, **scipy.integrate.odeint()** solves ODE of the form: $dy/dt = f(y1, y2, .., t0, ...)$
For example, to solve the ODE $\frac{dy}{dt} = -2 * y$ between $t = 0 \dots 4$, with the initial condition $y(t = 0) = 1$:

```
from scipy.integrate import odeint
def calc_derivative(ypos, time):
    return -2 * ypos

time = np.linspace(0, 4, 40)
y = odeint(calc_derivative, y0=1, t=time)
```

End of Lecture 2:

Introduction to Programming with Python 3 for Scientific Computing