Lecture_01 Python Introduction

October 3, 2019

1 Numerical Methods in finance

1.1 Lecture 1: Introduction to Python

by: Tomasz Romaczukiewicz web: th.if.uj.edu.pl/~trom/ rm B-2-03

1.1.1 Outline

- Intro
- Python among other programming languages
- Why is the allegedly fast Python so slow?
- Why is it fast?
- IDEs
- First codes

Intro

Python among other programming languages Programing languages: - 700 (wiki) - 8945 (Historical Encyclopaedia of Programming Languages) - compiled (C/C++, Java, Fortran, Assembler, Pascal ...) usually faster program is compiled into machine code and executed afterwards. - interpreted (Bash, Perl, Python, Julia, Octave, Matlab ...) usually much slower code (strings) are converted (many operations) and executed at the running time Some of the languages can be "compiled" to speed up (Julia, Python), but usually it requires additional tools and may be complicated

Why is the allegedly fast Python so slow?

- Python productivity: usually little code is needed to implement some algorythm so it is fast to write
- Easy to read, the code is usually clean with low WTF/s
- It is advertised as fast

- but codes rewritten from standard books run 50-200 times slower than those in C
- Loops (and recursive functions) are interpreted as many times as the loops are executed
- Calculations are encoded by strings
- String operations are much slower and more complicated than calculations on numbers

OK, since it's slow, why is it fast?

- It is usually faster than most interpreted languages such as Matlab, Octave but is slower than some others like Julia (advertised as fast as C as productive as Python)
- With additional work some (but not all) problems can be vectorized (for example with numpy) and rewtitten without loops
- Vectorized expresion is interpreted once and all the calculations are done by compiled methods (usually written in highly optimized C) and runs very fast
- Iterations when some value is calculated from previos value, such as $F_n = F_{n-1} + F_{n-2}$ cannot be vectorized and still run 100x slower than C
- Some (partial) solutions:
  - Cython - almost Python code translated to C, compiled and run
  - numba - *functions* compiled at run time

It is usually listed as the second best tool for a given problem: - www: JavaScript is on top but Python is close behind, the list is changing fast - interpretted language for numerical calculations: Matlab/Octave beats Python in productivity Python beats Matlab in speed but Julia is faster - Plotting (my personal list) 1. Gnuplot 2. Python

Example of vectorization Task: Calculate a sume $s = \sum_{n=1}^{N} n$ for some large N

```
In [1]: %%time
        N = int(1e7)
        s = 0
        for n in range(1,N+1):
            s += n
```

```
CPU times: user 8.56 s, sys: 19.3 ms, total: 8.57 s
Wall time: 8.78 s
```

```
In [4]: import numpy as np
```

```
In [3]: %%time
        N = int(1e7)
        s = np.sum(np.arange(1,N+1))
```

```
CPU times: user 102 ms, sys: 140 ms, total: 241 ms
Wall time: 282 ms
```

Python can be

- run in iterative shell (just type python3 or ipython3 in a terminal) and executed on a fly values are displayed right after execution
- edited in some editor (even as simple as vi, vim, gedit, kate etc) and then run as scripts from commandline or via some embedded tools
- editted dedicated IDEs (Spyder, Pycharm, Pydev etc) run from within IDE with additional features such as list output with plots variable lists with current values
- edited, run and presented in notebooks such as this one (Jupyter)

1.1.2 First programs

- Beginers should go through these codes as homework: run, change and play with the codes until everything is clear
- write some code by your own
- You can/should study some online tutorials
  - Official tutorial
  - W3School
  - other tutorials and books
  - Let me know which worked best for you
- print out some cheat codes
- basics
- numpy

1.1.3 Python as a calculator

open iterative console: type python3 or ipython3 in a terminal)

```
In [4]: 1+2

Out[4]: 3

In [5]: 0.1+0.2

Out[5]: 0.30000000000000004

In [6]: 0.1+0.2-0.3

Out[6]: 5.551115123125783e-17

In [7]: type(1+2)

Out[7]: int

In [8]: type(0.1+0.2)

Out[8]: float

In [9]: 1/2

Out[9]: 0.5

In [10]: type(1/2)

Out[10]: float

In [11]: 1//2

Out[11]: 0
```

```
In [12]: type(1//2)

Out[12]: int

In [13]: 2**5

Out[13]: 32

In [14]: 6%4

Out[14]: 2

In [15]: help(print)

Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```

Iterative console is fun (automatic output) but for serious job we need a proper editor (IDE) * My choice: Spyder - many features: - variable explorer, - integrated iconsole with possibility to render plots and LaTeX - nice help * Pycharm - Inteliji fork for Python - free but for cool features you have to pay * Pydev * but you can use any editor and use its tools to run scripts or run in external terminal python3 *scriptname*.py

```
In [16]: # Ordinary comment
         """
             Multiline documentation
         """

         print("Hello world") # print out a string

         a = 5         # assign a variable
         b = 6
         print(a, b)   # print out numbers

         a, b = 3, 2   # multiple assignment, a tupple
         print("a = ", a, " b = " , b)

         a, b = b, a   # simple swap
```

```
        print("a = ", a, " b/a = ", b/a, "b//a = ", b//a)  # unlike in C,Java etc b/a is not
                                                           # but b//a is an integer
        print("Formatted output: {:10d}, {:.4f}".format(a, b/a))
        print("Conversion from floats to ints:", int(1000*a/b))
        print("modulo division:", 15%6)
```

```
Hello world
5 6
a =  3  b =  2
a =  2  b/a =  1.5 b//a =  1
Formatted output:          2, 1.5000
Conversion from floats to ints: 666
modulo division: 3
```

```
In [17]: """ Logic operators """
        to_be = True
        print("1.",  to_be or not to_be)
        to_be = False
        print("2.", to_be or not to_be)
        print("3.", to_be and not to_be)
        to_be = 3
        print("4.", not to_be)
        print("5.", 7 | 2, 7&2, ~7)
        print("6.", 1<<3, 3<<3, 25>>2)
        print("7.", (7%3!=0), (7%3!=0)and(7%2==0), (666<=777))
```

```
1. True
2. True
3. False
4. False
5. 7 2 -8
6. 8 24 6
7. True False True
```

```
In [18]: """ From 3.6 version """

        # import math as mth  # including the full math module with prefix mth
        # print(f"sin(1) = {mth.sin(1)}")
        # print(f"cos(1) = {mth.cos(1)}")

        # import numpy as np  # including the full numpy module with prefix np, in order not
        # print(f"sin(1) = {np.sin(1)}")
        # print(f"cos(1) = {np.cos(1)}")

        # from math import exp, cosh # including only two functions but without prefix
        # print(f"exp(1) = {exp(1)}")
        # print(f"cosh(1) = {cosh(1)}")
```

```
Out[18]: ' From 3.6 version '
```

```
In [1]: import math as mth  # including the full math module with prefix mth
        print("sin(1) = ", mth.sin(1))
        print("cos(1) = ", mth.cos(1))

        import numpy as np  # including the full numpy module with prefix np, in order not to
        print("sin(1) = ", np.sin(1))
        print("cos(1) = ", np.cos(1))

        from math import exp, cosh # including only two functions but without prefix
        print("exp(1) = ", exp(1))
        print("cosh(1) =", cosh(1))
```

```
sin(1) =  0.8414709848078965
cos(1) =  0.5403023058681398
sin(1) =  0.841470984808
cos(1) =  0.540302305868
exp(1) =  2.718281828459045
cosh(1) = 1.5430806348152437
```

Exercise: Calculate the kinetic energy using relativistic and newtonian definitions

$$E_{rel} = \frac{mc^2}{\sqrt{1-\frac{v^2}{c^2}}} - mc^2$$

$$E_{Newt} = \frac{1}{2}mv^2$$

for $v = 1\frac{\mathrm{m}}{\mathrm{s}}$, $c = 3 \cdot 10^9 \frac{\mathrm{m}}{\mathrm{s}}$, $m = 1\mathrm{kg}$

```
In [20]: from math import sqrt

         v, c, m = 1, 3e9, 1
         Erel = m*c**2/sqrt(1-v**2/c**2)-m*c**2
         ENewt = 0.5*m*v**2

         print("E_rel = ", Erel, "\nE_Newt = ", ENewt)
```

```
E_rel =  0.0
E_Newt =  0.5
```

What's wrong?

```
In [21]: v**2/c**2
```

```
Out[21]: 1.111111111111111e-19
```

```
In [22]: 1-v**2/c**2
```

```
Out[22]: 1.0
```

Standard floats (64 bits) can handle only up to 16 digits So the accurate equation reduces to

$$E_{rel} = \frac{mc^2}{\sqrt{1-\frac{v^2}{c^2}}} - mc^2 \to \frac{mc^2}{1} - mc^2 = 0$$

which is painfully wrong. Due to the cancellation errors sometimes it is better to use approximation rather than the full formula

$$E_{rel} = mc^2\left(1-\frac{v^2}{c^2}\right)^{1/2} - mc^2 \approx mc^2\left(1-\frac{v^2}{2c^2}-1\right) = \frac{1}{2}mv^2$$

But this approximation is wrong for large values of v. Is there any way to have one formula which works in both cases?

With a bit of magic (short multiplication formulas)

$$E_{rel} = mc^2\left(\frac{c}{\sqrt{c^2-v^2}}-1\right) = mc^2\frac{\frac{c^2}{c^2-v^2}-1}{\frac{c}{\sqrt{c^2-v^2}}+1} = \frac{mv^2}{\sqrt{1-\frac{v^2}{c^2}}+1-\frac{v^2}{c^2}}$$

```
In [23]: g = 1-v**2/c**2
         E3 = m*v**2/(sqrt(g)+g)
         print(E3)
```

```
0.5
```

The above formula is mathematically equivalent with the definition but cancellation does not produce errors for small values of v.

```
In [22]: """ Simple calculations: but what can go wrong? """
         from numpy import cosh
         print("1. ",  0.3+0.2-0.5)  # this is 0
         print("2. ",  0.1+0.2-0.3)  # this is not 0, beware of round of errors

         x = 20
         print("3. ", (1-cosh(x))/(1+cosh(x)))

         x = 800
         print("4. ", (1-cosh(x))/(1+cosh(x))) # this produces overflow error
         print("5. ", (1/cosh(x)-1)/(1/cosh(x)+1)) # this produces overflow error
```

```
1.  0.0
2.  5.551115123125783e-17
3.  -0.999999991755
4.  nan
5.  -1.0
```

```
/usr/local/lib/python3.5/dist-packages/ipykernel_launcher.py:10: RuntimeWarning: overflow encou
  # Remove the CWD from sys.path while we load stuff.
/usr/local/lib/python3.5/dist-packages/ipykernel_launcher.py:10: RuntimeWarning: invalid value
  # Remove the CWD from sys.path while we load stuff.
/usr/local/lib/python3.5/dist-packages/ipykernel_launcher.py:11: RuntimeWarning: overflow encou
  # This is added back by InteractiveShellApp.init_path()
```

```
In [27]: """ List examples """
         v = [3, 4, 5, 8, -1]       # example array
         print ("1:", v)

         print ("2:", v[0], v[4], v[-2])   # elements:     [(3), (4), 5, (8), -1]
                                           # indexes:        0    1   2   3    4
                                           # negative ind:  -5   -4  -3  -2   -1
         v.append(7)                # adding a new element
         print("3:", v)

         v.insert(2, 6)             # inserting at certain position
         print("4:", v)

         print("5:", v[2:4])        # slices  [3, 4, (6, 5), 8, -1, 7]

         v.sort()                   # hmm, what could that be?
         print("6:", v)
```

```
1: [3, 4, 5, 8, -1]
2: 3 -1 8
3: [3, 4, 5, 8, -1, 7]
4: [3, 4, 6, 5, 8, -1, 7]
5: [6, 5]
6: [-1, 3, 4, 5, 6, 7, 8]
```

```
In [43]: w = list(range(1, 15, 3))  # each 3rd integer number from [1,15)
         print (" 7:", w)

         print(" 8:", w[2:5]+v[0:4:2])  # join list slices [1, 4, (7, 10, 13)] [(-1), 3, (4),
                                        # indices:          0  1  2   3   4       0   1   2
         print(" 9:", 2*w)    # the list is repeated twice

         w.reverse()
         print("10:", w)
         print("11:", w.index(7))
         w.append("The last in line")
         print(w)
         print("12:", w.pop())
         print("13:", w)
```

```
 7: [1, 4, 7, 10, 13]
 8: [7, 10, 13, -1, 4]
 9: [1, 4, 7, 10, 13, 1, 4, 7, 10, 13]
10: [13, 10, 7, 4, 1]
11: 2
[13, 10, 7, 4, 1, 'The last in line']
12: The last in line
13: [13, 10, 7, 4, 1]
```

```
In [26]: x = (1, 3, 6.62354, "Hello", "world")  # example of a tuple
         print(x)
         print(x[3], x[4])
         #x[1] = 2  # an error occurs, tuple as not mutable

         y = {2,3,4,2,3, "hello", "hello"}      # example of a set
         print(y)
         #  print(y[2])  # error:  'set' object does not support indexing

         z = {'a': 5, 'b': 3.1415, 'c': "some text"} # example of a dictionary
         print(z)
         print(z['c'])
```

```
(1, 3, 6.62354, 'Hello', 'world')
Hello world
{2, 'hello', 3, 4}
{'a': 5, 'b': 3.1415, 'c': 'some text'}
some text
```

```
In [27]: """ Flow control """
         from math import sqrt

         a = 1; b = 2; c = -1;
          = b**2-4*a*c
         print ("  = ", )
         if  <0:
             print (" is negative, no real solutions")
         elif ==0:   # note: '==' means comparison, '=' is assignment
             print (" is equal to 0, one real solution x =", -b/(2*a))
         else:
             print (" is positive, two real solutions")
             x1 = (-b+sqrt())/(2*a)
             x2 = (-b-sqrt())/(2*a)
             print ("x1 = ", x1, "x2 =",  x2)
```

```
  =  8
 is positive, two real solutions
x1 =  0.41421356237309515 x2 = -2.414213562373095
```

Excercise:

- Run the code to see all the possibilities.
- **Check $a = c = 10^{-4},\ b = 2 \cdot 10^{4}$. Solve it analytically as well.**
- Extend the program to include the case for $a = 0$ (linear equation) and consider all posible cases ($b = 0, c \neq 0$ and $b = 0, c = 0$)

```
In [28]: """ Loops """

         for n in range(2, 15, 3):
             print (n)
```

```
2
5
8
11
14
```

```
In [29]: F = [1, 1]
         for n in range(2, 10):
             F.append(F[n-2]+F[n-1])

         print ("Fibbonaci series: ", F)
```

```
Fibbonaci series:  [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

```
In [30]: a = 120
         divisors =[]
         for k in range(2,a+1):
             if a%k==0:
                 divisors.append(k)
         print ("Divisors of ", a, ": ", divisors)
```

```
Divisors of  120 :  [2, 3, 4, 5, 6, 8, 10, 12, 15, 20, 24, 30, 40, 60, 120]
```

```
In [46]: """ control and some more """
         for n in range(10):
             if (n%2==0): continue  # skip the rest and go to the beginning of the loop
             if (n==7): break       # finish the loop when n is equal to 7
             print(n)
```

```
1
3
5
```

```
In [32]: lst=["This", 'is', "a", 3.1415, "complicated list"]

         for idx, element in enumerate(lst):
             print("index=", idx, "value=", element)
```

```
index= 0 value= This
index= 1 value= is
index= 2 value= a
index= 3 value= 3.1415
index= 4 value= complicated list
```

```
In [33]: squares = [k**2 for k in range(10)]
         print (squares)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```
In [56]: """Iterations"""
         x, k = 0.3, 4
         for n in range(20):
             x = k*x*(1-x)
             print(x)
```

```
0.84
0.5376000000000001
0.9943449599999999
0.02249224209039382
0.08794536454456375
0.32084390959875014
0.8716123810885569
0.4476169528867727
0.9890240655005337
0.043421853445318986
0.1661455843547689
0.5541649166167251
0.9882646472316129
0.04639053705515447
0.17695382050755523
0.5825646636613406
0.9727323052579588
0.10609667026198408
0.3793606672852156
0.9417846056085262
```

Consider a logistic map $x \ni [0,1] \to f(x) = kx(1-x) \in [0,1]$ for $k \in [0,4]$. The iteration

$$x_{n+1} = kx_n(1-x_n)$$

has two fixed points (solution to $x = f(x)$): $x = 0$ and $x = 1 - \frac{1}{k}$.

Q: What happens when $k = 0.5$, $k = 2.5$, $k = 3.1$, $k = 3.5$, $k = 4$? And why? Around a fixed point $x_* = f(x_*)$:

$$x_n = x_* + h_n$$

$$h_{n+1} = x_{n+1} - x_* = f(x_* + h_n) - x_*$$

Linear stability

$$h_{n+1} \approx f(x_*) + h_n f'(x_*) - x_* + \mathcal{O}(h_n^2)$$

Neglecting nonlinear terms we have a geometric series:

$$h_{n+1} = h_n f'(x_*)$$

Fixed point x_* of an iteration $x_{n+1} = f(x_n)$ is linearly stable when $|f'(x_*)| < 1$ and is unstable when $|f'(x_*)| > 1$. Moreover the smaller $|f'(x_*)|$ the faster the convergence.

$$f'(0) = k \qquad f'\left(1 - \frac{1}{k}\right) = 2 - k$$

- $k < 1$ $x = 0$ is the only stable point,
- for $1 < k < 2$ $x = 1 - \frac{1}{k}$ is the only stable piont,
- for $k > 3$ there are no stable points
- but there are stable 2-cycles, 4-cycles etc $x_{n+2} = x_n$
- for certain values of k there is pure chaos

It is interesting from the point of view of a dynamical systems but very bad as a method for solving equations. But we can influence the stability

$$x = f(x) \quad \Big| \; + \omega x$$

$$(1 + \omega)x = f(x) + \omega x$$

$$x = \frac{f(x) + \omega x}{1 + \omega} \equiv \tilde{f}(x)$$

Parameter ω can control the stability ad convergence rate

$$\tilde{f}'(0) = \frac{k + \omega}{1 + \omega} \qquad \tilde{f}'\left(1 - \frac{1}{k}\right) = \frac{2 - k + \omega}{1 + \omega}$$

```
In [35]: """Increase stabililty of the iteration method """
         x, k,   = 0.5, 4, 1.5
         for n in range(20):
             x = ( k*x*(1-x) + *x) / (1+)
             print(x)
```

```
0.7
0.756
0.7487423999999999
0.750248989507584
0.7499501029052433
0.7500099754353992
0.7499980047537053
0.7500003990428894
0.7499999201911673
0.7500000159617564
0.7499999968076484
0.7500000006384704
0.7499999998723059
0.7500000000255389
0.7499999999948923
0.7500000000010216
0.7499999999997957
0.7500000000000409
0.7499999999999918
0.7500000000000016
```

Another simple iteration:

$$I_n = \frac{1}{e}\int_0^1 e^x x^n\,dx$$

generates the following conditions (decreasing but positive sequence)

$$0 < I_n < I_{n-1}$$

and

$$I_0 = 1 - \frac{1}{e} \qquad I_n = 1 - nI_{n-1}$$

```
In [53]: import numpy as np
         I = 1-1/np.e
         for n in range(1,25):
             if n>15: print("{:2d}\t{:15.8f}".format(n, I))
             I = 1-n*I
```

```
16              0.05903379
17              0.05545930
18              0.05719187
19             -0.02945367
20              1.55961974
21            -30.19239489
22            635.04029260
23         -13969.88643714
24         321308.38805421
```

```
In [37]: """ Functions """
         def Function():
             print("This is a simple function. Just some code")

         Function(); Function()   # function calls
```

```
This is a simple function. Just some code
This is a simple function. Just some code
```

```
In [38]: def Sum(a, b):   # arguments
             return a*b;  # returned value

         result = Sum(3, 222)
         print(result, Sum(3,4))
```

```
666 12
```

```
In [39]: def g(a, b):
             return (a+b, a*b, a/b, a**b) # return a tuple of four values

         s, m, d, p = g(3, 4)
         print(s, m, d, p)
         tup = g(3, 4)
         print(tup)
```

```
7 12 0.75 81
(7, 12, 0.75, 81)
```

Implementation of the recursive definition

$$n! = \begin{cases} 1 & \text{for } n = 0 \\ n\,(n-1)! & \text{otherwise} \end{cases}$$

```
In [44]: """ Recursive functions """
         def Factorial(n):
             if n<1: return 1
             else:   return n*Factorial(n-1)

         Factorial(4)
```

```
Out[44]: 24
```

```
In [45]: """ Recursive functions """
         def Factorial(n):
             print ("starting the calculation of ", n, "!")
             if n<1: result = 1
             else:   result = n*Factorial(n-1)
```

```
            print ("finished the calculation of ", n, "! = ", result)
            return result

        Factorial(4);
```

```
starting the calculation of  4 !
starting the calculation of  3 !
starting the calculation of  2 !
starting the calculation of  1 !
starting the calculation of  0 !
finished the calculation of  0 ! =  1
finished the calculation of  1 ! =  1
finished the calculation of  2 ! =  2
finished the calculation of  3 ! =  6
finished the calculation of  4 ! =  24
```

Fibonacci series: $F_1 = F_2 = 1$ and $F_n = F_{n-1} + F_{n-2}$ for $n > 2$

```
In [6]: def Fib(n):
            if n>2: return Fib(n-1)+Fib(n-2)
            else: return 1

        Fib(10)
```

```
Out[6]: 55
```

```
In [62]: """Very bad Fibbonacci"""
         counter = 0
         def Fib(n):
             global counter  # global counter
             counter+=1
             if n<3: return 1
             else: return Fib(n-1)+Fib(n-2)

         print("Fib(10) = ", Fib(10))
         print("counter = ", counter)
```

```
Fib(10) =  55
counter =  109
```

```
In [70]: """ Functions as arguments """
         import math as mt

         def make_table(f):
             print("==========",f.__name__,"============")
             for n in range(10):
                 #print(f"{n*0.1:.2f}\t\t{f(n*0.1): .8f}")
```

```
            print("{:.2f}\t\t{: .8f}". format(n*0.1, f(n*0.1)))

    make_table(mt.sin)              # another function as an argument
```

```
========== sin ============
0.00                0.00000000
0.10                0.09983342
0.20                0.19866933
0.30                0.29552021
0.40                0.38941834
0.50                0.47942554
0.60                0.56464247
0.70                0.64421769
0.80                0.71735609
0.90                0.78332691
```

```
In [71]: make_table(lambda x: x**2-1)     # anonymous lambda function
```

```
========== <lambda> ============
0.00               -1.00000000
0.10               -0.99000000
0.20               -0.96000000
0.30               -0.91000000
0.40               -0.84000000
0.50               -0.75000000
0.60               -0.64000000
0.70               -0.51000000
0.80               -0.36000000
0.90               -0.19000000
```

```
In [74]: """Functions with default and key arguments """

    def f1(a, b=3, c=4):
        print("f1:", a, b, c)

    f1(3, 4)
    f1(2, c=7)
```

```
f1: 3 4 4
f1: 2 3 7
```

```
In [75]: def f2(a, *b):   # additional arguments are used as a tuple
            print("f2:", a, b)

    f2("Hello")
    f2("Hello", "world", 777)
```

```
f2: Hello ()
f2: Hello ('world', 777)
```

```
In [73]: def f3(**k):      # all arguments are a dictionary
             print("f3:", k)
             for n in k:
                 print("f3: ", n," = ", k[n])

         f3(a=1, b=2, word="Hello")
```

```
f3: {'a': 1, 'b': 2, 'word': 'Hello'}
f3:  a  =  1
f3:  b  =  2
f3:  word  =  Hello
```

Summary of Python

1. Python is a very intuitive and powerfull language.
2. Basit types are strings, integers and floating-point numbers.
3. They can be encapsulated in lists [], tuples () and sets {}.
4. Lists are mutable, tuples and sets are not.
5. Division / of two numbers always give a floating point number.
6. Integer division: //, power: **.

7. if ..else statement controls the flow of the programs (decision making).
8. for loop can loop thourgh the lists, butis't slow.
9. Functions can be defined to use fragments of codes efficiently.
10. Anonymous lambda functions can be defined, when small function can be used as an argument.

Summary of the numerical part

1. All floating point calculations generate errors.
2. Even simple equations can give huge relative errors (cancelation problem).
3. Sometimes formulea can be rewritten in more numerical friendly way.
4. Sometimes approximations are better than exact formulae.

5. Badly written program can lead to extreme complexity (recursive Fibbonacci)
6. Iterative methods can have lots of problems:

- stability issues,
- two-,four- or more cycles,

7. Sometimes modification can change convergence and stability.
8. Some error can accumulate very quickly.

Calculations in Python/numpy

```
In [76]: import numpy as np  # np is used as a namespace
         x_list = list(range(5))
         print("list:    ", x_list)
         x = np.arange(5)
         print("np.array:", x)

         print("2*x_list:", 2*x_list)  # lists are joined
         x = np.arange(5)
         print("2*x:     ", 2*x)       # arrays are calculated elementwise

         print("cubes:   ", x**3)
```

```
list:     [0, 1, 2, 3, 4]
np.array: [0 1 2 3 4]
2*x_list: [0, 1, 2, 3, 4, 0, 1, 2, 3, 4]
2*x:      [0 2 4 6 8]
cubes:    [ 0  1  8 27 64]
```

```
In [81]: x = np.linspace(0, np.pi, 8)  # horizontal vector
         print ("shape of x:", np.shape(x))
         print ("\ntable of sins: ", np.sin(x))

         y = np.array( [x, np.sin(x), np.cos(x), np.sin(x)**2+np.cos(x)**2 ] )
         # 4 rows of horizontal vectors
         print ("\nshape of y", np.shape(y))
         print (y.transpose())  # transpose for nice output
```

```
shape of x: (8,)

table of sins:  [  0.00000000e+00   4.33883739e-01   7.81831482e-01   9.74927912e-01
   9.74927912e-01   7.81831482e-01   4.33883739e-01   1.22464680e-16]

shape of y (4, 8)
[[  0.00000000e+00   0.00000000e+00   1.00000000e+00   1.00000000e+00]
 [  4.48798951e-01   4.33883739e-01   9.00968868e-01   1.00000000e+00]
 [  8.97597901e-01   7.81831482e-01   6.23489802e-01   1.00000000e+00]
 [  1.34639685e+00   9.74927912e-01   2.22520934e-01   1.00000000e+00]
 [  1.79519580e+00   9.74927912e-01  -2.22520934e-01   1.00000000e+00]
 [  2.24399475e+00   7.81831482e-01  -6.23489802e-01   1.00000000e+00]
 [  2.69279370e+00   4.33883739e-01  -9.00968868e-01   1.00000000e+00]
 [  3.14159265e+00   1.22464680e-16  -1.00000000e+00   1.00000000e+00]]
```

```
In [80]: sums=np.zeros(4)
         print ("\nsums after initiation:", sums)
```

```
        for k in range(4):
            sums[k] = np.sum(y[:,k])
        print ("Calculated sums:", sums)
```

```
sums after initiation: [ 0.  0.  0.  0.]
Calculated sums: [ 2.          2.78365156  3.30291919  3.5438457 ]
```

Recall the example: $S = \sum_{n=1}^{N} n$. Numpy can be much much faster than standard Python loops

```
In [1]: %%time
        N=int(1e7)
        s = 0
        for n in range(1,N+1):
            s += n
```

```
CPU times: user 8.57 s, sys: 20.1 ms, total: 8.59 s
Wall time: 8.76 s
```

```
In [7]: %%time
        N=int(1e7)
        s = np.sum(np.arange(1,N+1))
```

```
CPU times: user 60.7 ms, sys: 339 ms, total: 400 ms
Wall time: 937 ms
```

```
In [8]: """Broadcasting examples"""
        a = np.array([1, 2, 3])
        3*a   # 3*a_i
```

```
Out[8]: array([3, 6, 9])
```

```
In [9]: a*a  # a_i^2
```

```
Out[9]: array([1, 4, 9])
```

```
In [18]: b = np.ones([2,3])
         2*b
```

```
Out[18]: array([[ 2.,  2.,  2.],
                [ 2.,  2.,  2.]])
```

```
In [19]: a*b
```

```
Out[19]: array([[ 1.,  2.,  3.],
                [ 1.,  2.,  3.]])
```

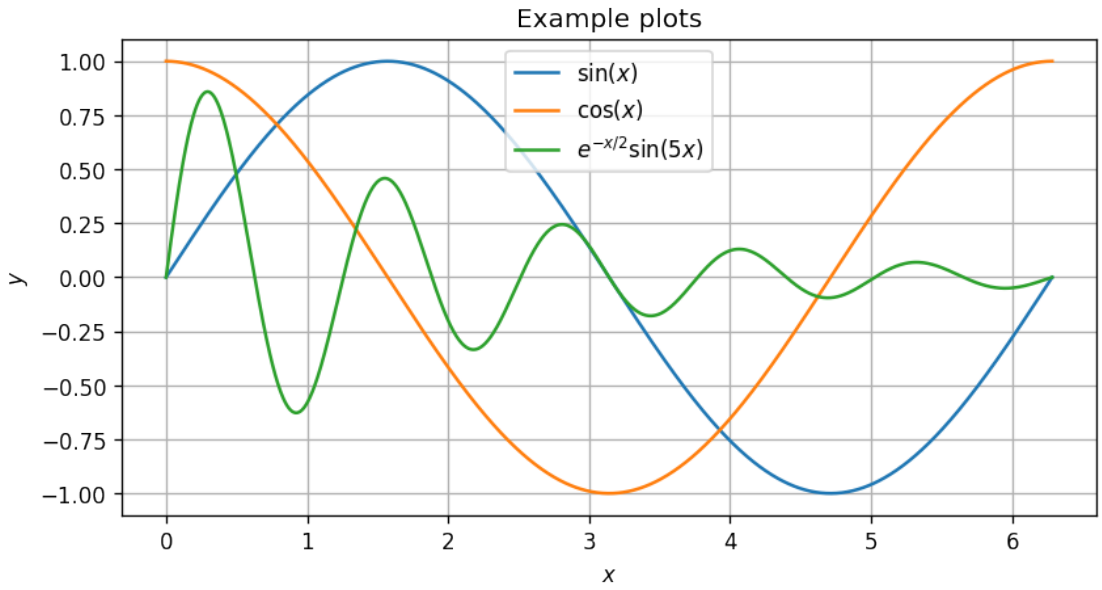
```
In [20]: b*a

Out[20]: array([[ 1.,  2.,  3.],
                [ 1.,  2.,  3.]])
```

```
In [106]: import matplotlib.pyplot as plt  # ploting library
          def make_plot():
              x = np.linspace(0, 2*np.pi, 1000)
              y = np.array( [ np.sin(x), np.cos(x), np.exp(-0.5*x)*np.sin(5*x) ] ).transpose()

              plt.figure(figsize=(8,4), dpi=120)
              plt.plot(x, y)
              plt.legend(['$\sin(x)$', '$\cos(x)$', '$e^{-x/2}\sin(5x)$'])
              plt.title('Example plots')
              plt.xlabel('$x$')
              plt.ylabel("$y$")
              plt.grid(True)
              plt.show()
```

```
In [107]: make_plot()
```




```
In [85]: """Sympy example"""
         import sympy as sp
         x, k = sp.var('x k')
         f = k*x*(1-x)
         print("Expression:", f)
         eq1 = sp.Eq(f-x)
         sols = sp.solve(eq1,x)
         print("Fixed points:", sols)
```

```
Expression: k*x*(1 - x)
Fixed points: [0, (k - 1)/k]
```

```
In [86]: df = sp.diff(f,x)
         print("derivative:", df)
         for (n,s) in enumerate(sols):
             print("f'(x_",n,") = ", sp.simplify(df.subs(x, s)))

derivative: -k*x + k*(1 - x)
f'(x_ 0 ) =  k
f'(x_ 1 ) =  2 - k
```

```
In [101]: g=f.subs(x,f)   # g(x) = f(f(x))
          print(g, "\n")
          two_cycle = sp.solve(sp.Eq(g-x),x)
          print(*two_cycle, "\n", sep="\n")
          print( sp.latex(two_cycle[2]))
          print(sp.latex(two_cycle[3]))

k**2*x*(1 - x)*(-k*x*(1 - x) + 1)

0
(k - 1)/k
(k - sqrt(k**2 - 2*k - 3) + 1)/(2*k)
(k + sqrt(k**2 - 2*k - 3) + 1)/(2*k)
```

```
\frac{k - \sqrt{k^{2} - 2 k - 3} + 1}{2 k}
\frac{k + \sqrt{k^{2} - 2 k - 3} + 1}{2 k}
```

```
In [105]:
```

```
          File "<ipython-input-105-307fab640f6d>", line 5
    SyntaxError: can't use starred expression here
```