Accuracy and Speed

1. Variables and ranges

2. Numerical error

3. Program speed

Accuracy and ranges

- Finite range for floating-point values (-10^{308} to 10^{308})
- Note that numbers specified in scientific notation *x*e*y* or *x*E*y* are always floats
- If the variable is overflowed, python will not give an error message, but will set the variable to the special value “inf”, which means infinity.
- There is also a smallest number that be represented by a floating-point variable. In Python this number is 10^{-308} roughly. If the calculation underflows, the computer will just set the number to be zero.

How about integer?

- In Python, it can represent integers to ***arbitrary precision.***

- However, it will take longer time with more digits.

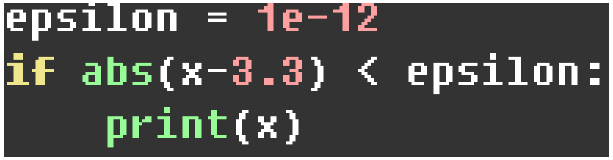
Try *print(2**1000000)*

Numerical error

- Integer variable are more accurate than floating-point variables. Floating-point calculations on computers are not infinitely accurate. The difference between the true value of a number and its value on the computer is called *rounding error*.

- The standard level of precision is 16 significant digits. The difference between1 and 0.9999999999999999 or 1.0000000000000001 could be crucially important.

- If you want to test the equality of floats, you should do something like



```
epsilon = 1e-12
if abs(x-3.3) < epsilon:
    print(x)
```

- The value of epsilon has to be chosen appropriately for the situation- there is nothing special or universal about the value of 10^{-12} used above and a different value may be appropriate in another calculation.

How to measure error

- It’s usually a good assumption to consider the error to be a random number with *standard deviation* $\sigma = Cx$, where $C \approx 10^{-16}$ in Python and is referred as *error constant*.

- In many ways the rounding error on a number behaves similarly to measurement error in a laboratory experiment, and the rules for combining errors are the same.

- E.g. we are calculating the sum of N number $x_1, x_2, \ldots, x_N$with errors having standard deviation $\sigma_i = Cx_i$, then the variance on the final result is the sum of the variances on the individual numbers $\sigma^2 = \sum_{i=1}^{N} \sigma_i^2 = \sum_{i=1}^{N} C^2 x_i^2 = C^2 N \overline{x^2}$, where is the mean-square value of x. Thus the standard deviation on the final result is $\sigma = C\sqrt{N}\sqrt{\overline{x^2}}$. The more numbers we combine, the larger the error on the result.

Fractional error

- We can also ask about the fractional error on , i.e., the total error divided by the value of the sum. The size of the fractional error is given by

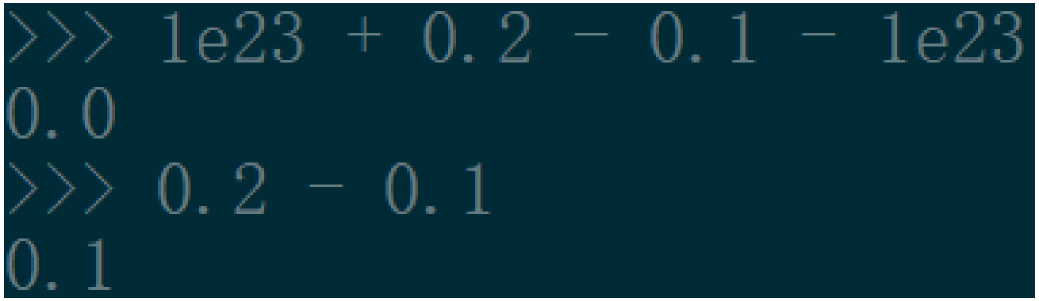
$$\frac{\sigma}{\sum_i x_i} = \frac{C\sqrt{N\overline{x^2}}}{N\bar{x}} = \frac{C}{\sqrt{N}}\frac{\sqrt{\overline{x^2}}}{\bar{x}}$$

- At first glance this appears to be pretty good. Actually, there are a couple of them. One is when the sizes of the numbers you are adding vary widely. If some are much smaller than others then the smaller ones may get lost.

Examples and Quizzes

```
a=2**100
b=2
c=1
d=3.0

e=a+b-c-a
f=a-a+b-c
g=a+b-d-a
h=a-a+b-d

### what are the values of e,f,g and h?
```

Error from arithmetic



```
>>> 1e23 + 0.2 - 0.1 - 1e23
0.0
>>> 0.2 - 0.1
0.1
```

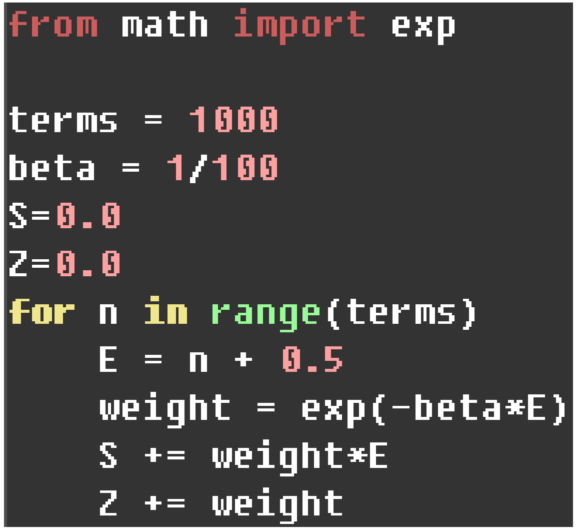
- If the difference between two numbers is very small, comparable with the error on the numbers, i.e., with the accuracy of the computer, then the fractional error can become large and you may have a problem.

- This issue of large errors in calculations that involve the subtraction of numbers that are nearly equal, arises with some frequency in scientific calculations.

- **KEEP IN MIND**

Program speed

- Computers are not infinitely accurate, and neither are they infinitely fast.
- Not all operations are equal, and it makes a difference whether we are talking about additions or multiplications of single numbers.



```
from math import exp

terms = 1000
beta = 1/100
S=0.0
Z=0.0
for n in range(terms)
    E = n + 0.5
    weight = exp(-beta*E)
    S += weight*E
    Z += weight
```

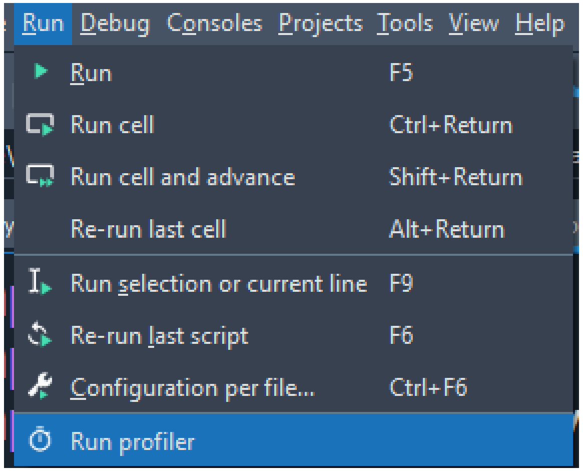
- First, a billion operations is indeed doable. If a calculation is important to us, we can wait twenty minutes for an answer.
- Second, there is a balance to be struck between time spent and accuracy.
- Third, it's worth taking a moment, before you spend a whole lot time writing and running a program, to do a quick estimate. **VERY IMPORTANT!**

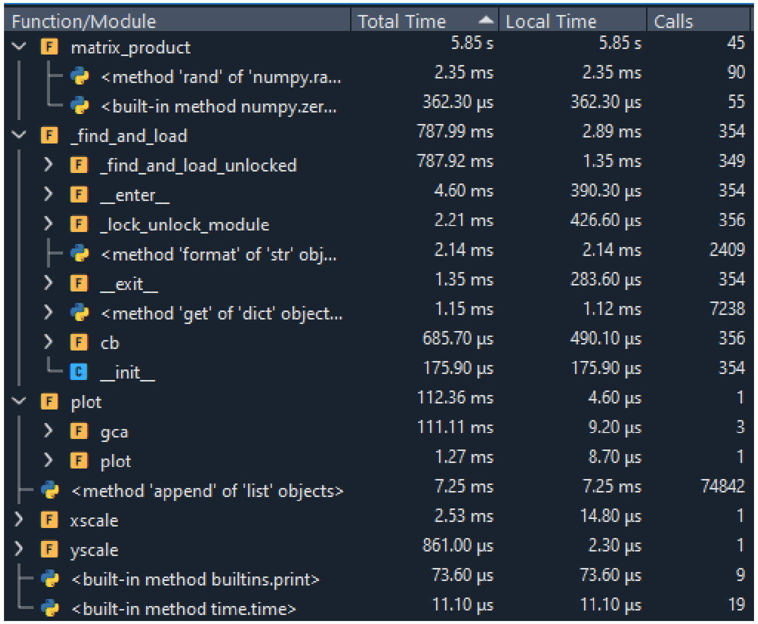
Monitor the running time

- The simplest way

```
import time
start_time=time.time()
######  the codes you want to monitor   #####
print("--- %s seconds ---"%(time.time()-start_time))
```

- Many different profiling tools

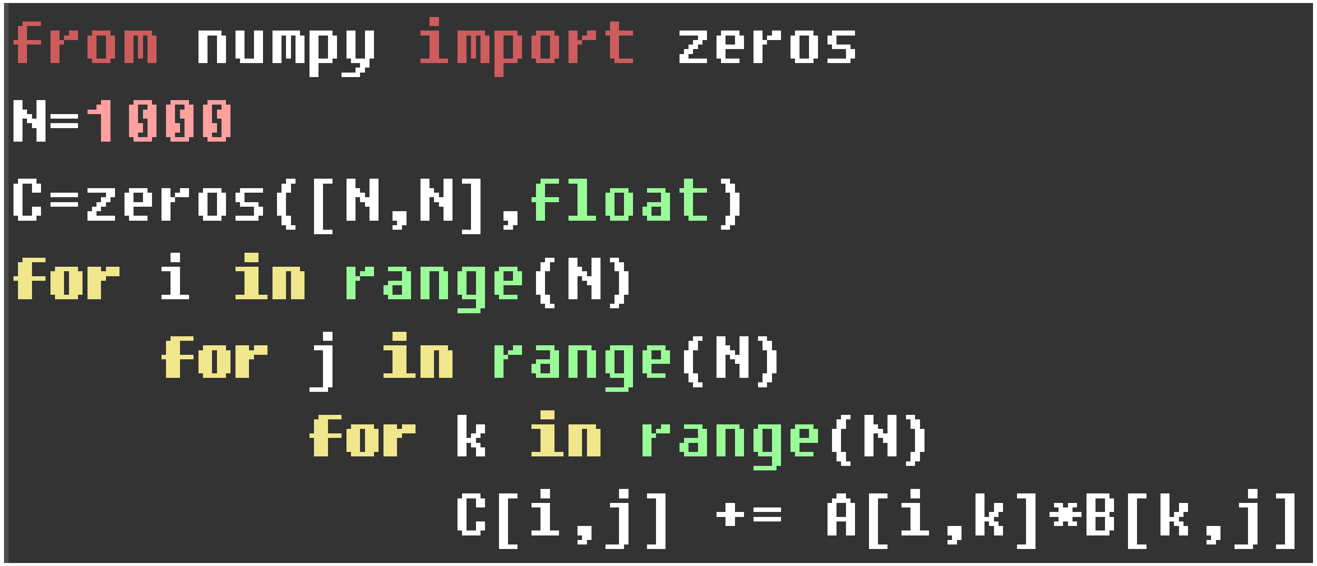

Run
Debug
Consoles
Projects
Tools
View
Help
Run F5
Run cell Ctrl+Return
Run cell and advance Shift+Return
Re-run last cell Alt+Return
Run selection or current line F9
Re-run last script F6
Configuration per file... Ctrl+F6
Run profiler



Function/Module	Total Time	Local Time	Calls
matrix_product	5.85 s	5.85 s	45
<method 'rand' of 'numpy.ra...	2.35 ms	2.35 ms	90
<built-in method numpy.zer...	362.30 µs	362.30 µs	55
_find_and_load	787.99 ms	2.89 ms	354
_find_and_load_unlocked	787.92 ms	1.35 ms	349
__enter__	4.60 ms	390.30 µs	354
_lock_unlock_module	2.21 ms	426.60 µs	356
<method 'format' of 'str' obj...	2.14 ms	2.14 ms	2409
__exit__	1.35 ms	283.60 µs	354
<method 'get' of 'dict' object...	1.15 ms	1.12 ms	7238
cb	685.70 µs	490.10 µs	356
__init__	175.90 µs	175.90 µs	354
plot	112.36 ms	4.60 µs	1
gca	111.11 ms	9.20 µs	3
plot	1.27 ms	8.70 µs	1
<method 'append' of 'list' objects>	7.25 ms	7.25 ms	74842
xscale	2.53 ms	14.80 µs	1
yscale	861.00 µs	2.30 µs	1
<built-in method builtins.print>	73.60 µs	73.60 µs	9
<built-in method time.time>	11.10 µs	11.10 µs	19

- Use the one you are familiar to.

Complexity and scaling



```
from numpy import zeros
N=1000
C=zeros([N,N],float)
for i in range(N)
    for j in range(N)
        for k in range(N)
            C[i,j] += A[i,k]*B[k,j]
```

- It will take $2N^3$ operations overall. The complexity is $O(N^3)$.
- Thus the largest matrices we can multiply are about 1000×1000 in size.

Examples and Quizzes

$$x^3 + y^3 + z^3 = N$$

- Given an integer number N, please find the **integer solution** of the equation x, y and z.
- For example, $7^3 + (-5)^3 + (-6)^3 = 2$

$$x^3 + y^3 + z^3 = 6$$

$$x^3 + y^3 + z^3 = 7$$

$$x^3 + y^3 + z^3 = 8$$

$$x^3 + y^3 + z^3 = 9$$

$$x^3 + y^3 + z^3 = 10$$

$$x^3 + y^3 + z^3 = 11$$

$$x^3 + y^3 + z^3 = 12$$

- What is the complexity? Try $x^3 + y^3 + z^3 = 42$

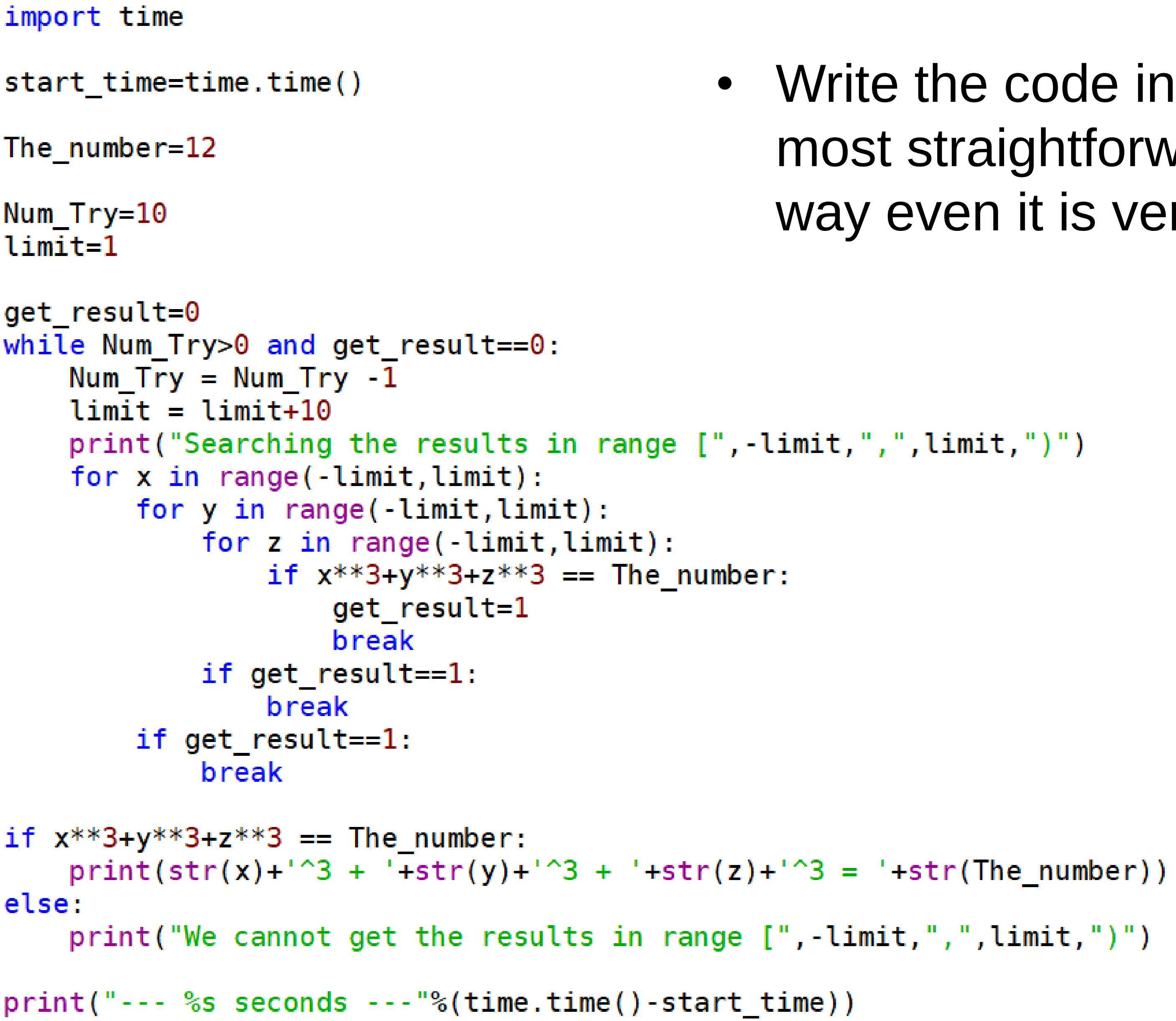
Nerd’s culture about 42

The Commercial Council of Magrathea
MAGRATHEAN PUBLIC ARCHIVE

- We just get the solution for 42 in 2019

$$(-80538738812075974)^3 + 80435758145817515^3 + 12602123297335631^3 = 42$$

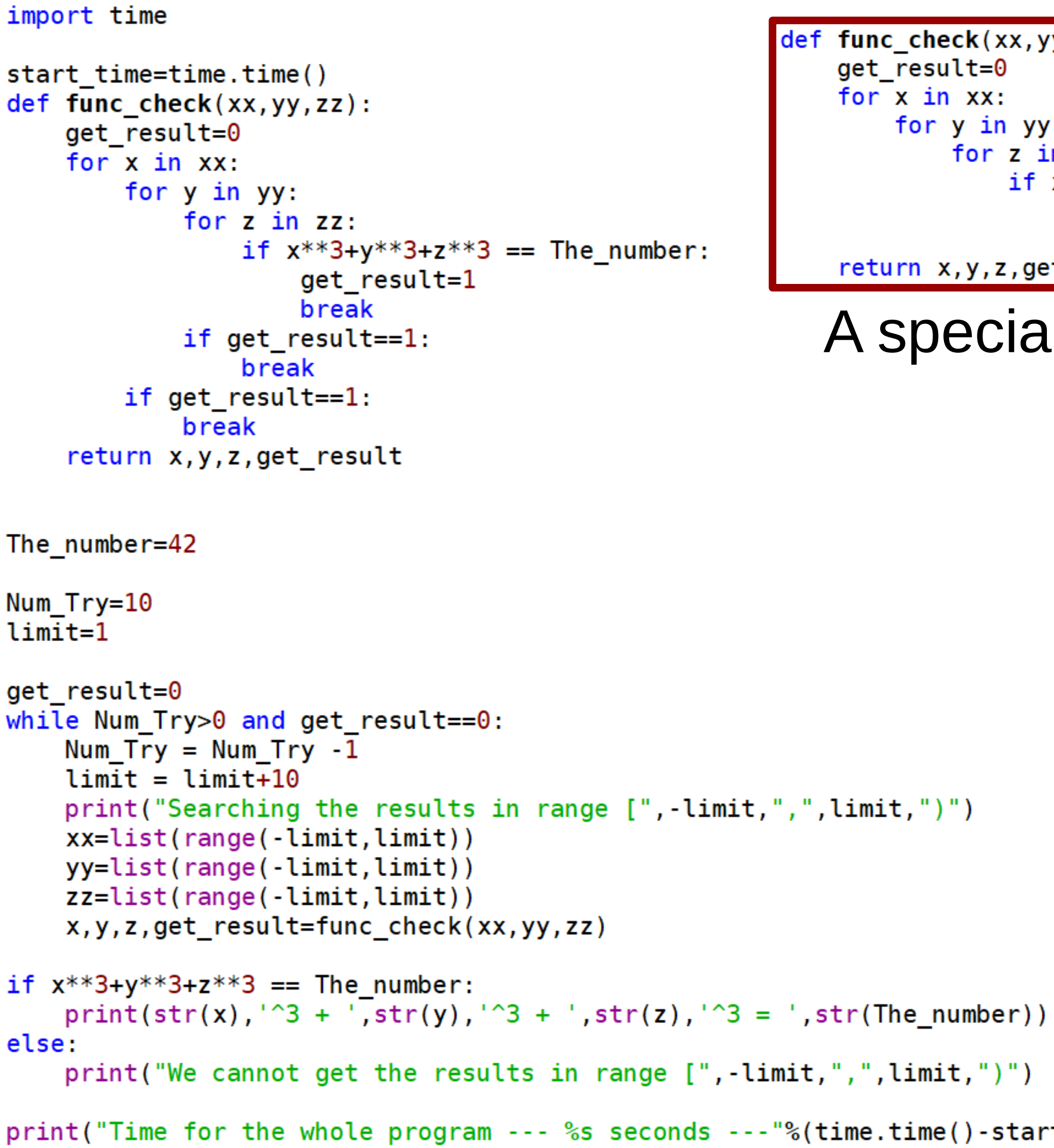
V0: the most straightforward version



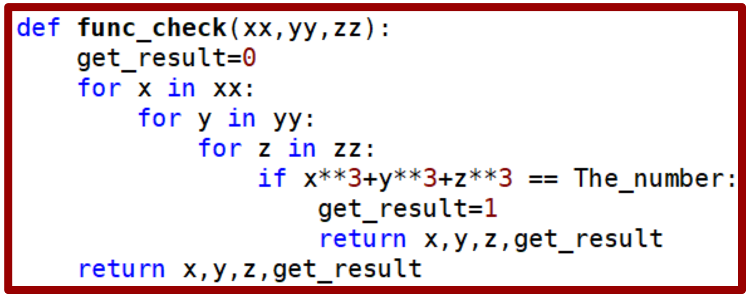
```python
import time

start_time=time.time()

The_number=12

Num_Try=10
limit=1

get_result=0
while Num_Try>0 and get_result==0:
    Num_Try = Num_Try -1
    limit = limit+10
    print("Searching the results in range [",-limit,",",limit,")")
    for x in range(-limit,limit):
        for y in range(-limit,limit):
            for z in range(-limit,limit):
                if x**3+y**3+z**3 == The_number:
                    get_result=1
                    break
            if get_result==1:
                break
        if get_result==1:
            break

if x**3+y**3+z**3 == The_number:
    print(str(x)+'^3 + '+str(y)+'^3 + '+str(z)+'^3 = '+str(The_number))
else:
    print("We cannot get the results in range [",-limit,",",limit,")")

print("--- %s seconds ---"%(time.time()-start_time))
```

- Write the code in the most straightforward way even it is very slow

V1: structuralize the codes



```
import time

start_time=time.time()
def func_check(xx,yy,zz):
    get_result=0
    for x in xx:
        for y in yy:
            for z in zz:
                if x**3+y**3+z**3 == The_number:
                    get_result=1
                    break
            if get_result==1:
                break
        if get_result==1:
            break
    return x,y,z,get_result


The_number=42

Num_Try=10
limit=1

get_result=0
while Num_Try>0 and get_result==0:
    Num_Try = Num_Try -1
    limit = limit+10
    print("Searching the results in range [",-limit,",",limit,")")
    xx=list(range(-limit,limit))
    yy=list(range(-limit,limit))
    zz=list(range(-limit,limit))
    x,y,z,get_result=func_check(xx,yy,zz)

if x**3+y**3+z**3 == The_number:
    print(str(x),'^3 + ',str(y),'^3 + ',str(z),'^3 = ',str(The_number))
else:
    print("We cannot get the results in range [",-limit,",",limit,")")

print("Time for the whole program --- %s seconds ---"%(time.time()-start_time))
```



```
def func_check(xx,yy,zz):
    get_result=0
    for x in xx:
        for y in yy:
            for z in zz:
                if x**3+y**3+z**3 == The_number:
                    get_result=1
                    return x,y,z,get_result
    return x,y,z,get_result
```

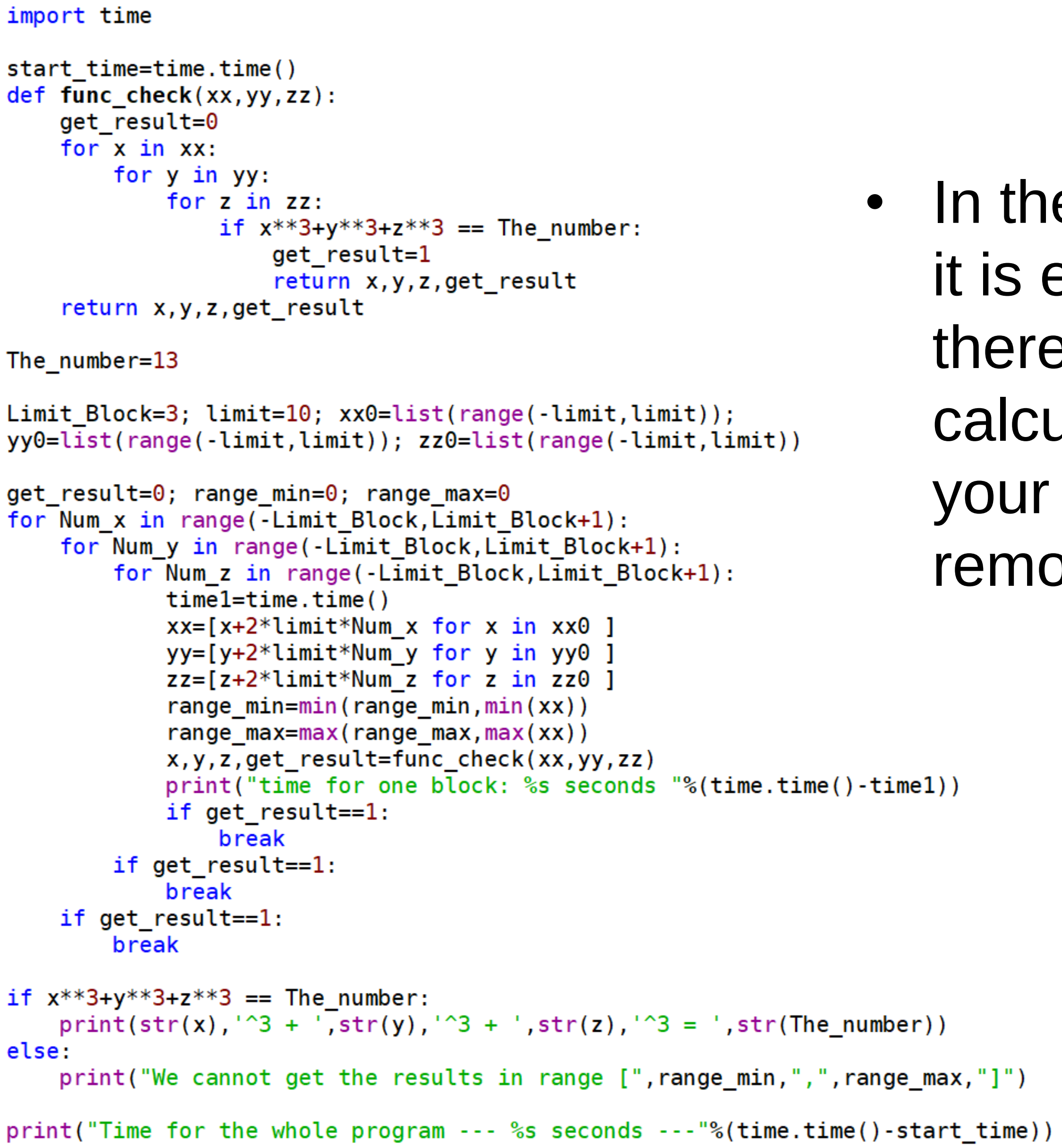
A special use of return

- Organize the code using functions or other structures

✓ Easy to read

✓ Easy to find the redundancy or inefficient parts

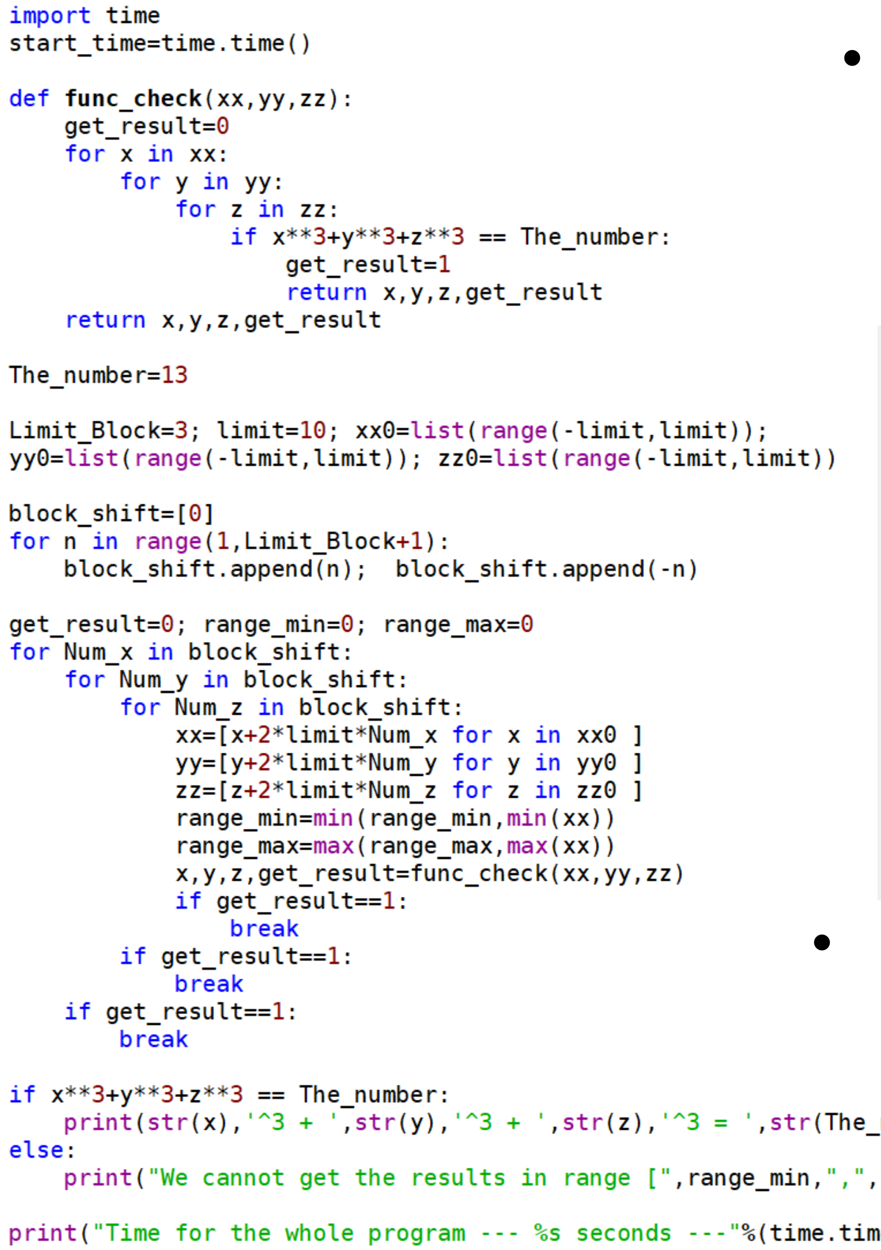
V2: remove the repeated calculations



```
import time

start_time=time.time()
def func_check(xx,yy,zz):
    get_result=0
    for x in xx:
        for y in yy:
            for z in zz:
                if x**3+y**3+z**3 == The_number:
                    get_result=1
                    return x,y,z,get_result
    return x,y,z,get_result

The_number=13

Limit_Block=3; limit=10; xx0=list(range(-limit,limit));
yy0=list(range(-limit,limit)); zz0=list(range(-limit,limit))

get_result=0; range_min=0; range_max=0
for Num_x in range(-Limit_Block,Limit_Block+1):
    for Num_y in range(-Limit_Block,Limit_Block+1):
        for Num_z in range(-Limit_Block,Limit_Block+1):
            time1=time.time()
            xx=[x+2*limit*Num_x for x in xx0 ]
            yy=[y+2*limit*Num_y for y in yy0 ]
            zz=[z+2*limit*Num_z for z in zz0 ]
            range_min=min(range_min,min(xx))
            range_max=max(range_max,max(xx))
            x,y,z,get_result=func_check(xx,yy,zz)
            print("time for one block: %s seconds "%(time.time()-time1))
            if get_result==1:
                break
        if get_result==1:
            break
    if get_result==1:
        break

if x**3+y**3+z**3 == The_number:
    print(str(x),'^3 + ',str(y),'^3 + ',str(z),'^3 = ',str(The_number))
else:
    print("We cannot get the results in range [",range_min,",",range_max,"]")

print("Time for the whole program --- %s seconds ---"%(time.time()-start_time))
```

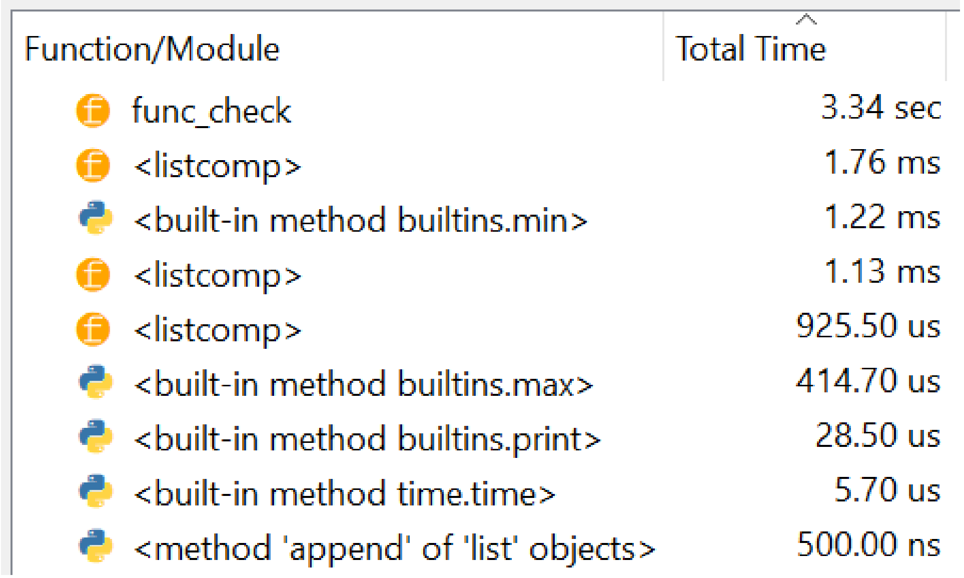
- In the structuralized codes, it is easy to find whether there are some repeated calculations. If so, change your codes or algorithms to remove it.

V3: choose a suitable calculation ordering



```
import time
start_time=time.time()

def func_check(xx,yy,zz):
    get_result=0
    for x in xx:
        for y in yy:
            for z in zz:
                if x**3+y**3+z**3 == The_number:
                    get_result=1
                    return x,y,z,get_result
    return x,y,z,get_result

The_number=13

Limit_Block=3; limit=10; xx0=list(range(-limit,limit));
yy0=list(range(-limit,limit)); zz0=list(range(-limit,limit))

block_shift=[0]
for n in range(1,Limit_Block+1):
    block_shift.append(n);  block_shift.append(-n)

get_result=0; range_min=0; range_max=0
for Num_x in block_shift:
    for Num_y in block_shift:
        for Num_z in block_shift:
            xx=[x+2*limit*Num_x for x in xx0 ]
            yy=[y+2*limit*Num_y for y in yy0 ]
            zz=[z+2*limit*Num_z for z in zz0 ]
            range_min=min(range_min,min(xx))
            range_max=max(range_max,max(xx))
            x,y,z,get_result=func_check(xx,yy,zz)
            if get_result==1:
                break
        if get_result==1:
            break
    if get_result==1:
        break

if x**3+y**3+z**3 == The_number:
    print(str(x),'^3 + ',str(y),'^3 + ',str(z),'^3 = ',str(The_number))
else:
    print("We cannot get the results in range [",range_min,",",range_max,"]")

print("Time for the whole program --- %s seconds ---"%(time.time()-start_time))
```

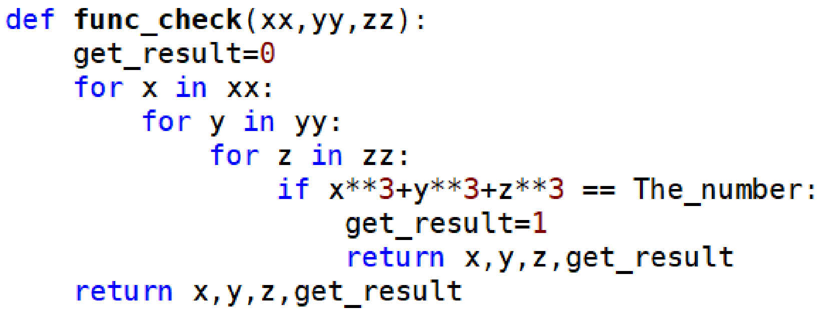
- In some cases, we guess the answer in some regions. Then we first check them.



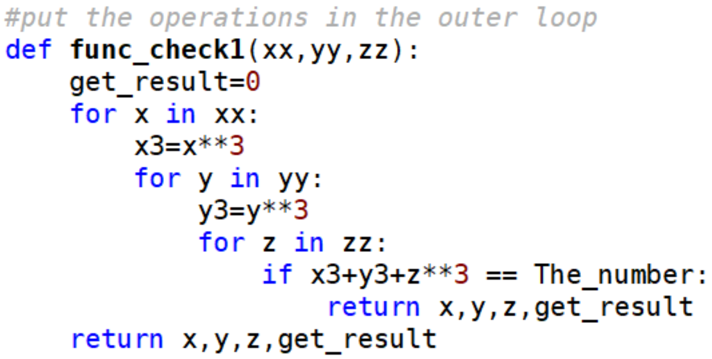
Function/Module	Total Time
func_check	3.34 sec
<listcomp>	1.76 ms
<built-in method builtins.min>	1.22 ms
<listcomp>	1.13 ms
<listcomp>	925.50 us
<built-in method builtins.max>	414.70 us
<built-in method builtins.print>	28.50 us
<built-in method time.time>	5.70 us
<method 'append' of 'list' objects>	500.00 ns

- Based on the profiling, most of time is spent in func_check().

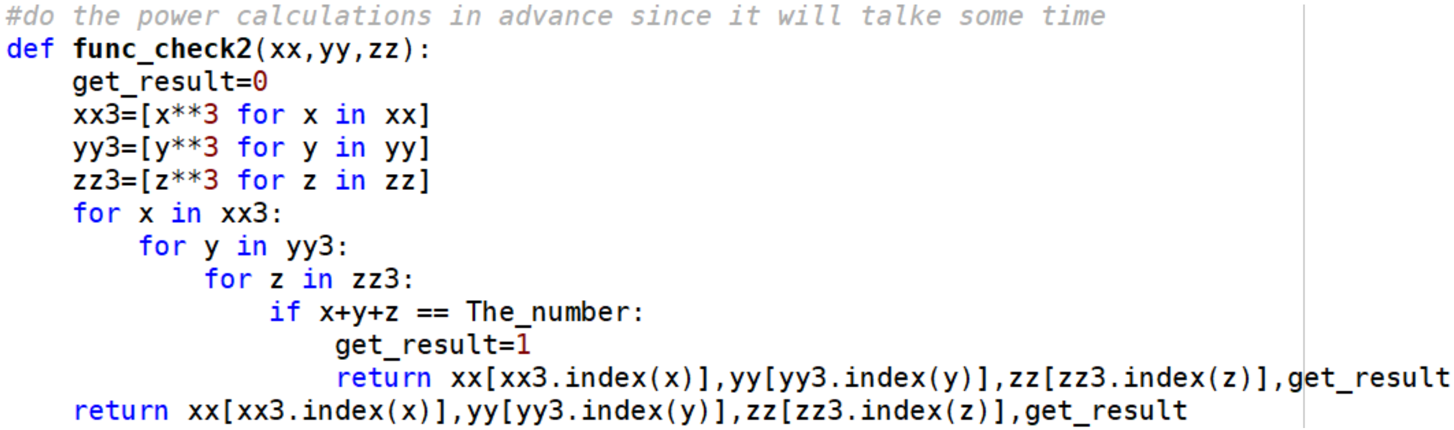
Small change but big improvements



```python
def func_check(xx,yy,zz):
    get_result=0
    for x in xx:
        for y in yy:
            for z in zz:
                if x**3+y**3+z**3 == The_number:
                    get_result=1
                    return x,y,z,get_result
    return x,y,z,get_result
```



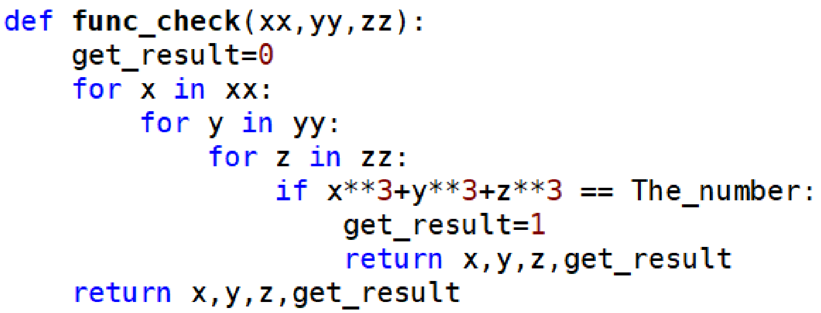
```python
#put the operations in the outer loop
def func_check1(xx,yy,zz):
    get_result=0
    for x in xx:
        x3=x**3
        for y in yy:
            y3=y**3
            for z in zz:
                if x3+y3+z**3 == The_number:
                    return x,y,z,get_result
    return x,y,z,get_result
```



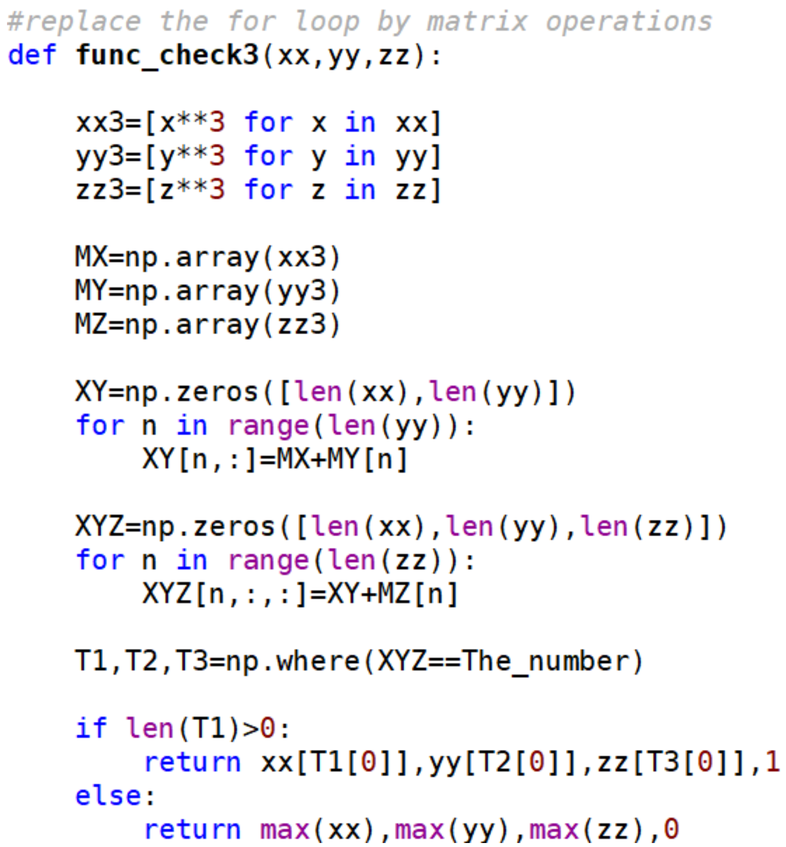
```python
#do the power calculations in advance since it will talke some time
def func_check2(xx,yy,zz):
    get_result=0
    xx3=[x**3 for x in xx]
    yy3=[y**3 for y in yy]
    zz3=[z**3 for z in zz]
    for x in xx3:
        for y in yy3:
            for z in zz3:
                if x+y+z == The_number:
                    get_result=1
                    return xx[xx3.index(x)],yy[yy3.index(y)],zz[zz3.index(z)],get_result
    return xx[xx3.index(x)],yy[yy3.index(y)],zz[zz3.index(z)],get_result
```

- When there are multiple loops, it can significantly speed up the codes by moving the commands from the inner loops to outer loops

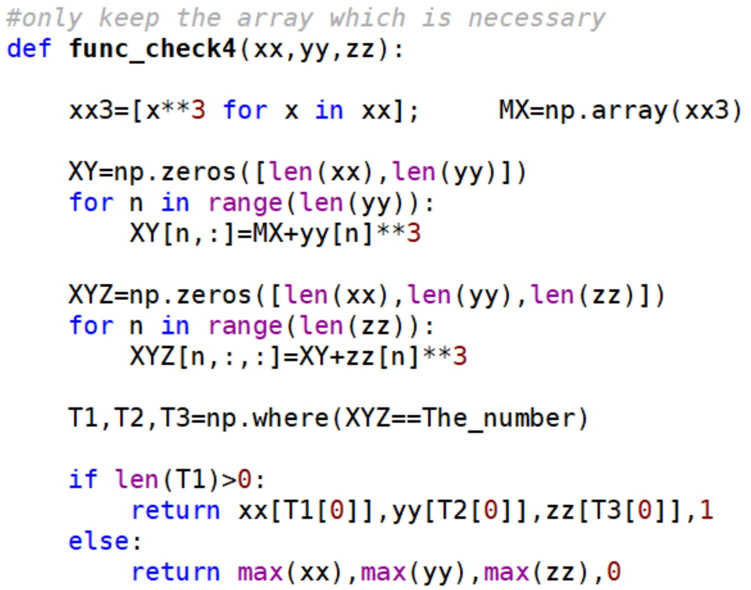
for loop → array operation



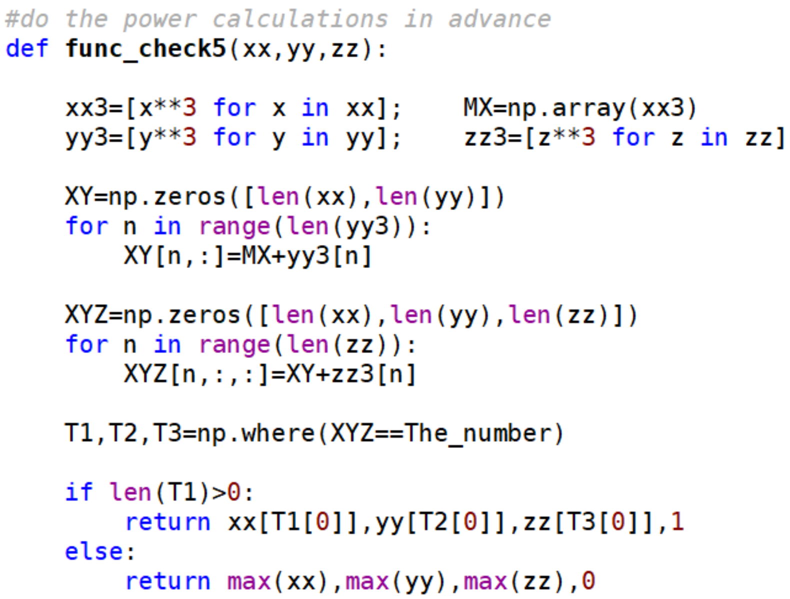
```
def func_check(xx,yy,zz):
    get_result=0
    for x in xx:
        for y in yy:
            for z in zz:
                if x**3+y**3+z**3 == The_number:
                    get_result=1
                    return x,y,z,get_result
    return x,y,z,get_result
```



```
#replace the for loop by matrix operations
def func_check3(xx,yy,zz):

    xx3=[x**3 for x in xx]
    yy3=[y**3 for y in yy]
    zz3=[z**3 for z in zz]

    MX=np.array(xx3)
    MY=np.array(yy3)
    MZ=np.array(zz3)

    XY=np.zeros([len(xx),len(yy)])
    for n in range(len(yy)):
        XY[n,:]=MX+MY[n]

    XYZ=np.zeros([len(xx),len(yy),len(zz)])
    for n in range(len(zz)):
        XYZ[n,:,:]=XY+MZ[n]

    T1,T2,T3=np.where(XYZ==The_number)

    if len(T1)>0:
        return xx[T1[0]],yy[T2[0]],zz[T3[0]],1
    else:
        return max(xx),max(yy),max(zz),0
```

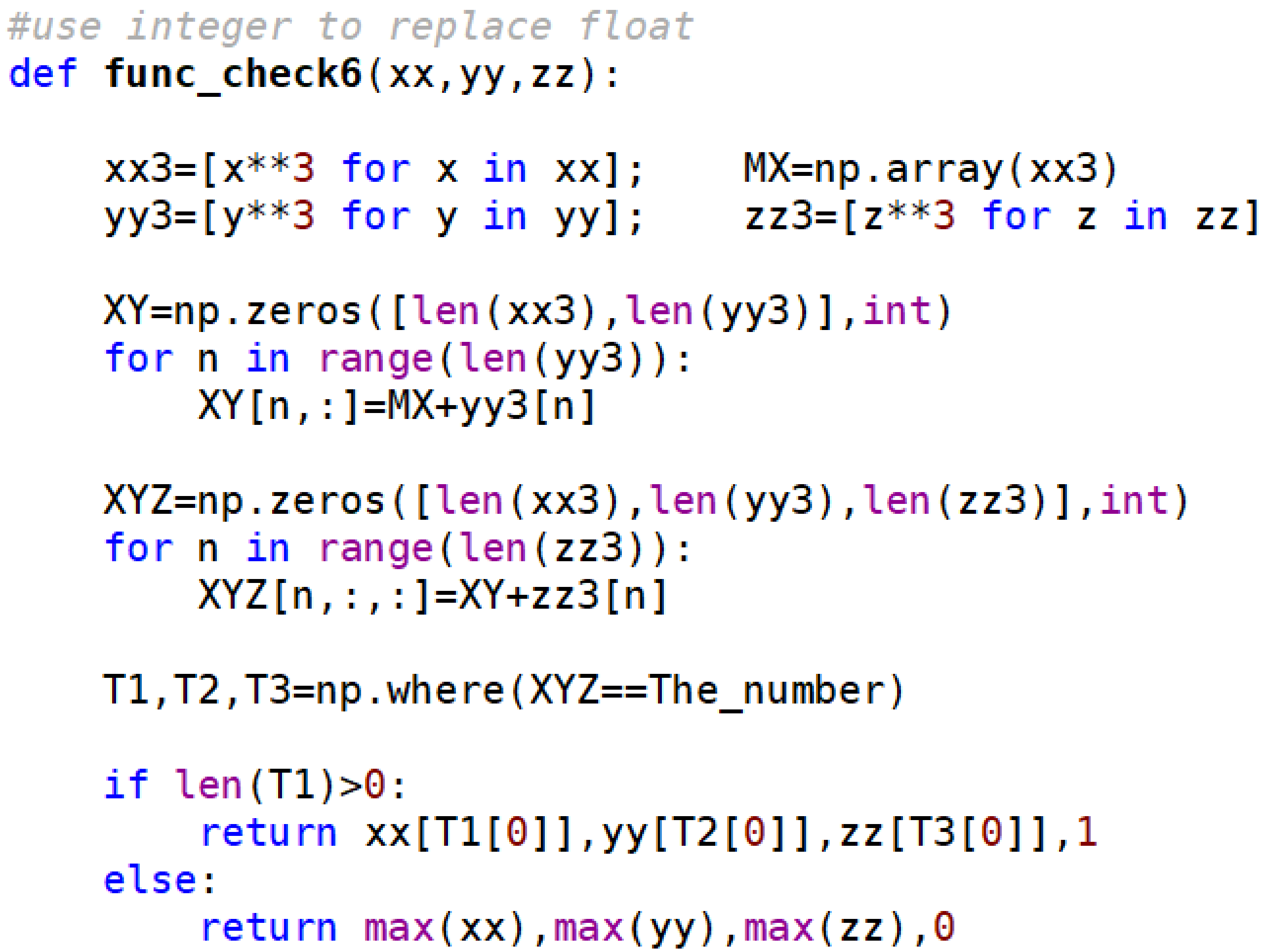


```
#only keep the array which is necessary
def func_check4(xx,yy,zz):

    xx3=[x**3 for x in xx];     MX=np.array(xx3)

    XY=np.zeros([len(xx),len(yy)])
    for n in range(len(yy)):
        XY[n,:]=MX+yy[n]**3

    XYZ=np.zeros([len(xx),len(yy),len(zz)])
    for n in range(len(zz)):
        XYZ[n,:,:]=XY+zz[n]**3

    T1,T2,T3=np.where(XYZ==The_number)

    if len(T1)>0:
        return xx[T1[0]],yy[T2[0]],zz[T3[0]],1
    else:
        return max(xx),max(yy),max(zz),0
```



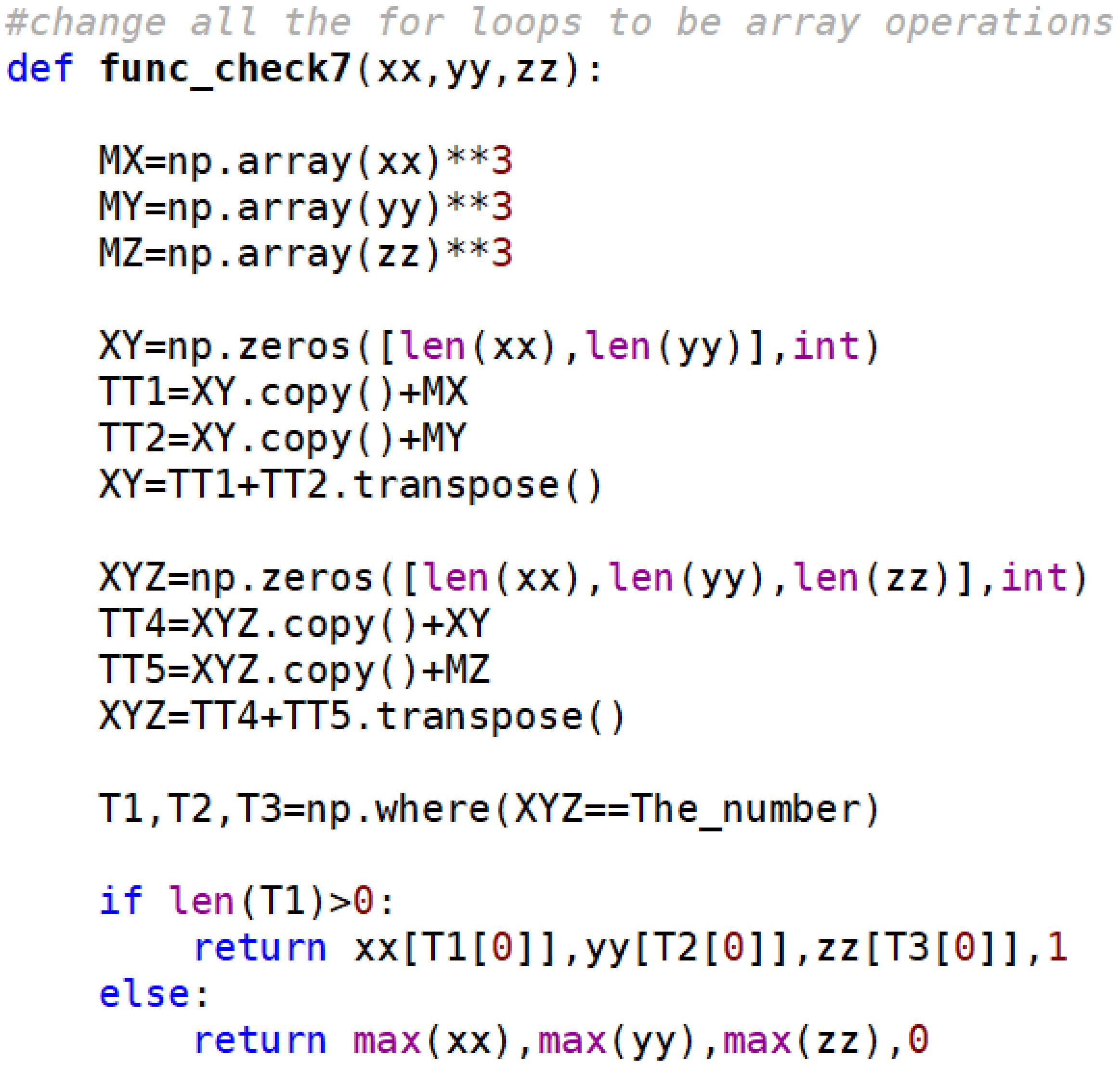
```
#do the power calculations in advance
def func_check5(xx,yy,zz):

    xx3=[x**3 for x in xx];    MX=np.array(xx3)
    yy3=[y**3 for y in yy];    zz3=[z**3 for z in zz]

    XY=np.zeros([len(xx),len(yy)])
    for n in range(len(yy3)):
        XY[n,:]=MX+yy3[n]

    XYZ=np.zeros([len(xx),len(yy),len(zz)])
    for n in range(len(zz)):
        XYZ[n,:,:]=XY+zz3[n]

    T1,T2,T3=np.where(XYZ==The_number)

    if len(T1)>0:
        return xx[T1[0]],yy[T2[0]],zz[T3[0]],1
    else:
        return max(xx),max(yy),max(zz),0
```

Choose suitable type of variable



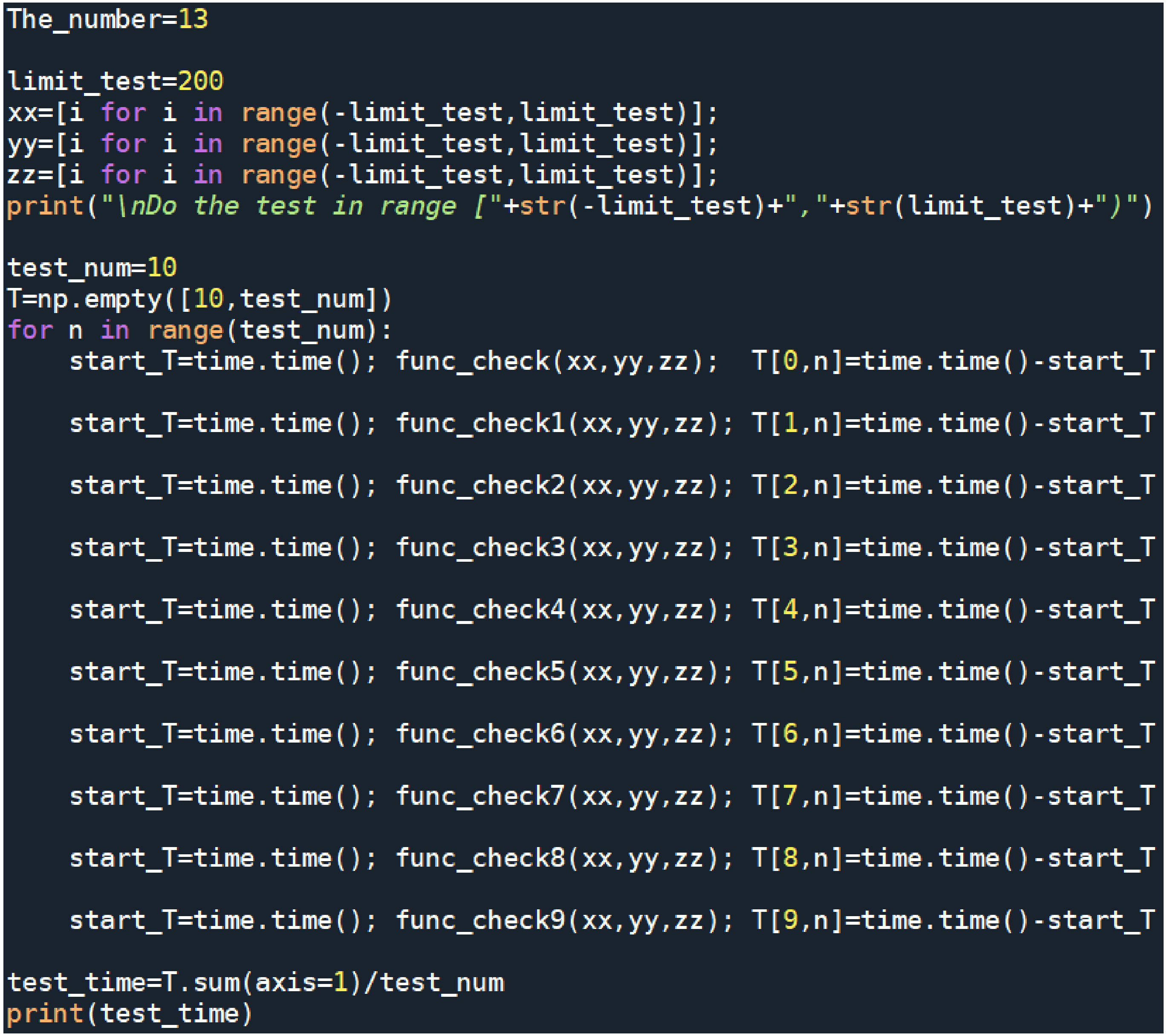
```
#use integer to replace float
def func_check6(xx,yy,zz):

    xx3=[x**3 for x in xx];    MX=np.array(xx3)
    yy3=[y**3 for y in yy];    zz3=[z**3 for z in zz]

    XY=np.zeros([len(xx3),len(yy3)],int)
    for n in range(len(yy3)):
        XY[n,:]=MX+yy3[n]

    XYZ=np.zeros([len(xx3),len(yy3),len(zz3)],int)
    for n in range(len(zz3)):
        XYZ[n,:,:]=XY+zz3[n]

    T1,T2,T3=np.where(XYZ==The_number)

    if len(T1)>0:
        return xx[T1[0]],yy[T2[0]],zz[T3[0]],1
    else:
        return max(xx),max(yy),max(zz),0
```

Change all for loops to be array operations



```python
#change all the for loops to be array operations
def func_check7(xx,yy,zz):

    MX=np.array(xx)**3
    MY=np.array(yy)**3
    MZ=np.array(zz)**3

    XY=np.zeros([len(xx),len(yy)],int)
    TT1=XY.copy()+MX
    TT2=XY.copy()+MY
    XY=TT1+TT2.transpose()

    XYZ=np.zeros([len(xx),len(yy),len(zz)],int)
    TT4=XYZ.copy()+XY
    TT5=XYZ.copy()+MZ
    XYZ=TT4+TT5.transpose()

    T1,T2,T3=np.where(XYZ==The_number)

    if len(T1)>0:
        return xx[T1[0]],yy[T2[0]],zz[T3[0]],1
    else:
        return max(xx),max(yy),max(zz),0
```

Test the speed

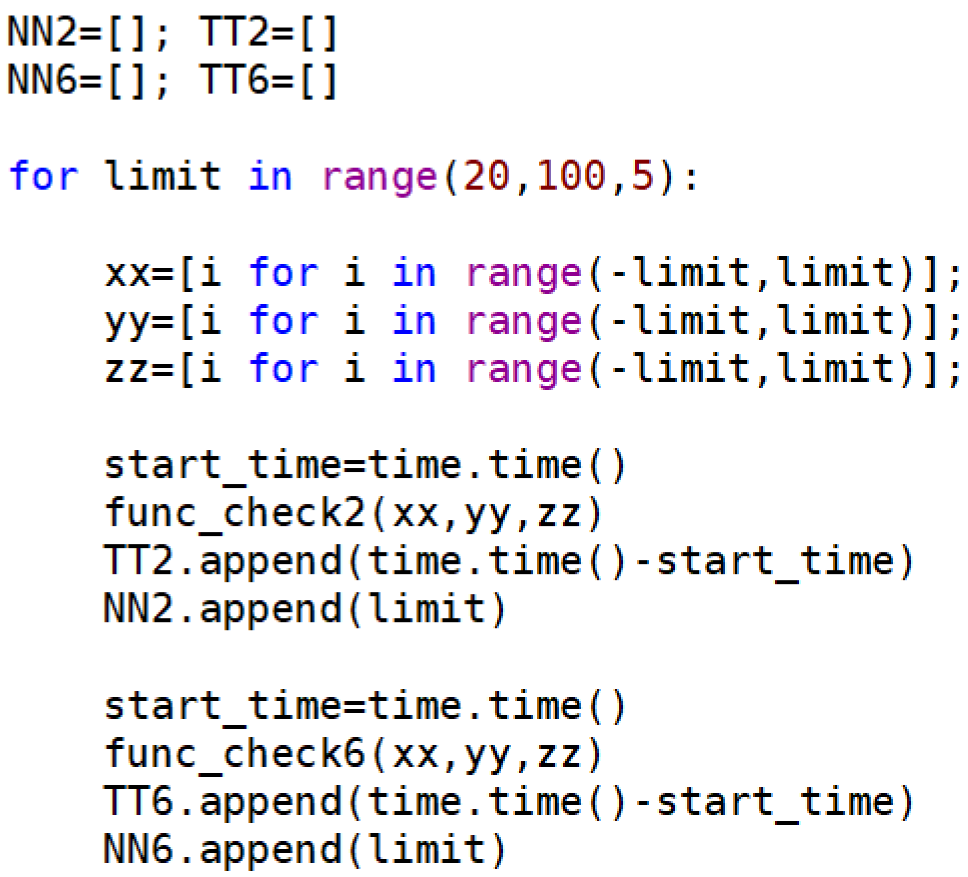


```python
The_number=13

limit_test=200
xx=[i for i in range(-limit_test,limit_test)];
yy=[i for i in range(-limit_test,limit_test)];
zz=[i for i in range(-limit_test,limit_test)];
print("\nDo the test in range ["+str(-limit_test)+","+str(limit_test)+")")

test_num=10
T=np.empty([10,test_num])
for n in range(test_num):
    start_T=time.time(); func_check(xx,yy,zz);  T[0,n]=time.time()-start_T

    start_T=time.time(); func_check1(xx,yy,zz); T[1,n]=time.time()-start_T

    start_T=time.time(); func_check2(xx,yy,zz); T[2,n]=time.time()-start_T

    start_T=time.time(); func_check3(xx,yy,zz); T[3,n]=time.time()-start_T

    start_T=time.time(); func_check4(xx,yy,zz); T[4,n]=time.time()-start_T

    start_T=time.time(); func_check5(xx,yy,zz); T[5,n]=time.time()-start_T

    start_T=time.time(); func_check6(xx,yy,zz); T[6,n]=time.time()-start_T

    start_T=time.time(); func_check7(xx,yy,zz); T[7,n]=time.time()-start_T

    start_T=time.time(); func_check8(xx,yy,zz); T[8,n]=time.time()-start_T

    start_T=time.time(); func_check9(xx,yy,zz); T[9,n]=time.time()-start_T

test_time=T.sum(axis=1)/test_num
print(test_time)
```

The time used for each function is

```
[49.21206236
19.86828437
5.54533165
0.58016756
0.58497243
0.56971071
  0.33580039
1.49211164
1.33724504
1.21054277]
```

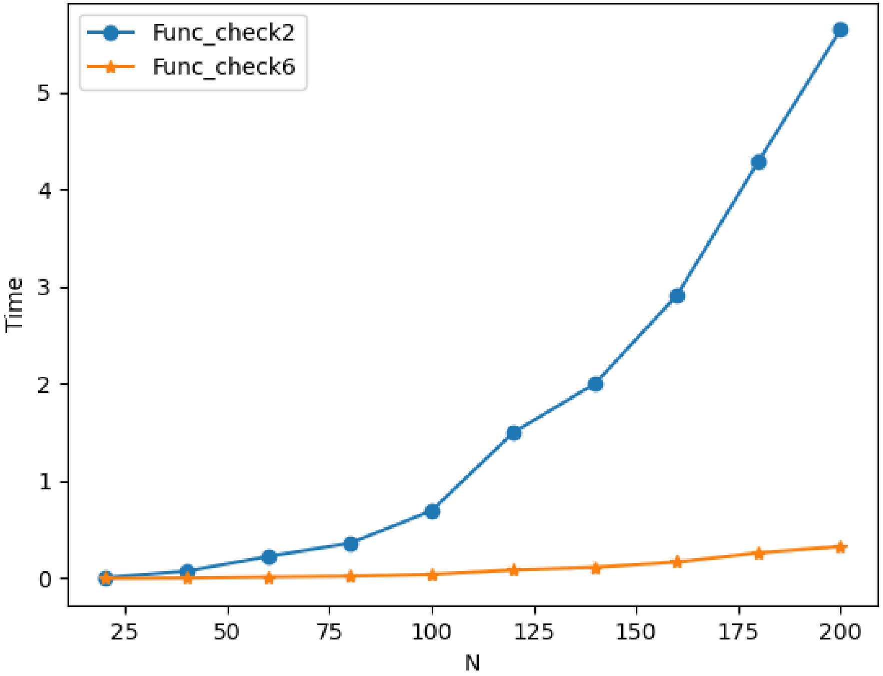
Check the complexity

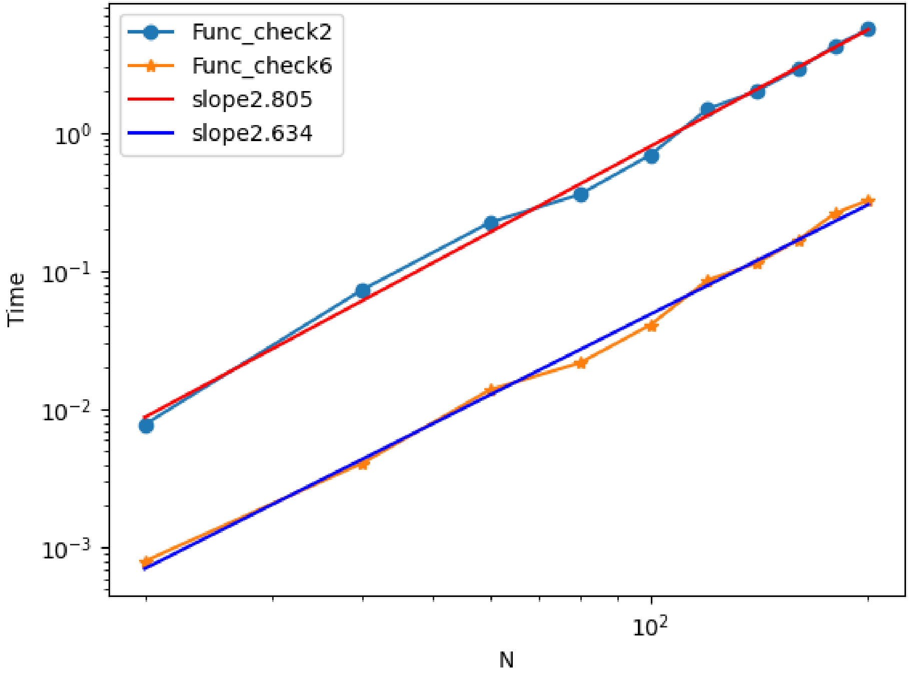


```
NN2=[]; TT2=[]
NN6=[]; TT6=[]

for limit in range(20,100,5):

    xx=[i for i in range(-limit,limit)];
    yy=[i for i in range(-limit,limit)];
    zz=[i for i in range(-limit,limit)];

    start_time=time.time()
    func_check2(xx,yy,zz)
    TT2.append(time.time()-start_time)
    NN2.append(limit)

    start_time=time.time()
    func_check6(xx,yy,zz)
    TT6.append(time.time()-start_time)
    NN6.append(limit)
```

N	Time_func2	Time_func6	Ratio
20	0.017164	0.002590	6.627731
40	0.089149	0.010472	8.512722
60	0.160927	0.021317	7.549214
80	0.324337	0.040314	8.045207
100	0.723974	0.085675	8.450219
120	1.078973	0.129651	8.322139
140	1.753251	0.189896	9.232677
160	2.464803	0.267897	9.200545
180	4.641647	0.446792	10.38884

Right way to show the data





- The complexity usually has the form $T = f(N) = \gamma_0 N^0 + \gamma_1 N^1 + \gamma_2 N^2 + \gamma_3 N^3 + \cdots + \gamma_m N^m \approx \gamma_m N^m$

- It is convenient to use log-log plot $\log(T) \approx \log(\gamma_m) + m \log(N)$ since one can easily get m by using a linear regression.

NumPy

- NumPy is the fundamental package for **scientific computing** in Python.

- At the core of the NumPy package, is the **ndarray object**. This encapsulates n-dimensional arrays of homogeneous data types, with many operations being performed in compiled code for performance.

- NumPy arrays have **a fixed size** at creation. Changing the size of an array will create a new array and delete the original.

- The elements in an array are of **the same data type**.

- NumPy arrays facilitate advanced mathematical and other types of operations on large numbers of data. (**much faster**)

Why so fast? Vectorization and Broadcasting

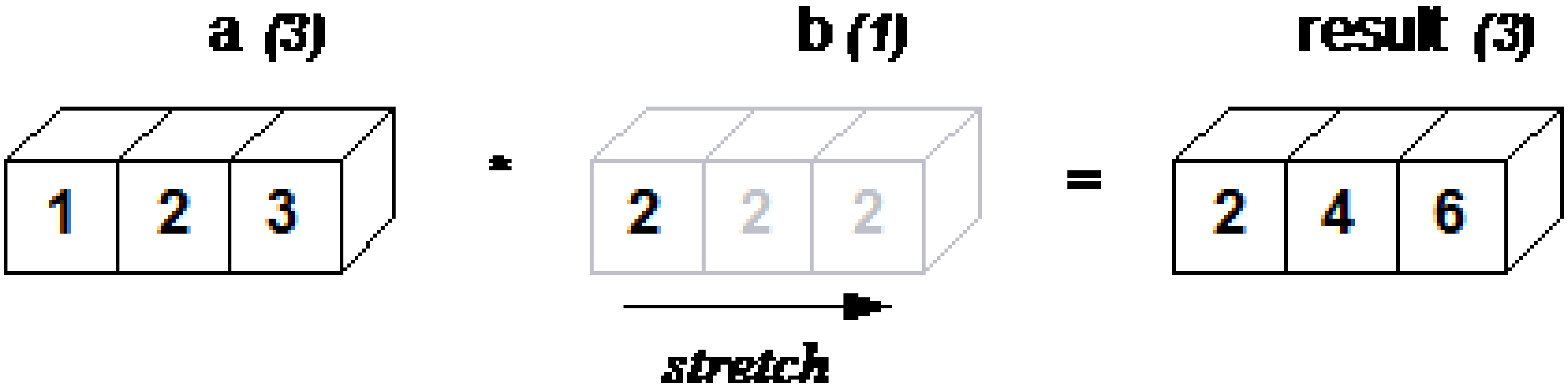
- Vectorization describes the absence of any explicit looping, indexing, etc., in the code - these things are taking place, of course, just “behind the scenes” in optimized, pre-compiled C code. Vectorized code has many advantages.

- Broadcasting means the **implicit element-by-element behavior of operations**; in NumPy all operations including arithmetic operations, logical, bit-wise, functional, etc., behave in the broadcasting fashion. Moreover, the objects could be multidimensional arrays of the same shape, or a scalar and an array, or even two arrays of with different shapes, provided that the smaller array is “expandable” to the shape of the larger in such a way that the resulting broadcast is unambiguous.

- These features save the overhead in interpreting the Python code and manipulating Python objects, while they do not change the complexity.)

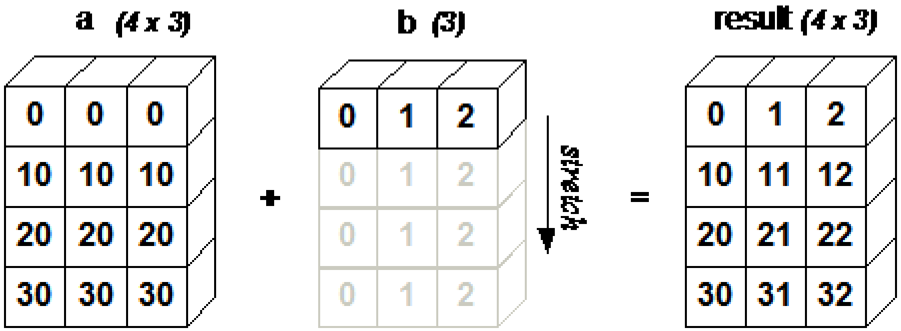
Broadcasting

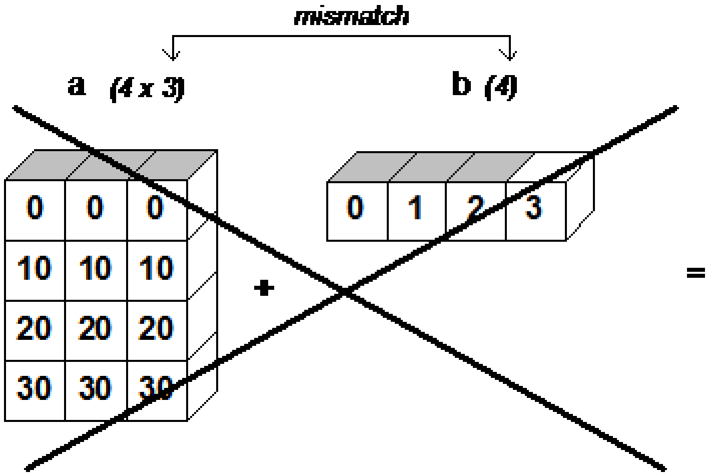
- Broadcasting describes how NumPy treats arrays with **different shapes** during operations. Subject to certain constraints, **the smaller array** is “broadcast” across **the larger array** so that they have **compatible shapes**.
- Broadcasting provides a means of **vectorizing array operations** so that **looping occurs in C** instead of Python. It does this without making needless copies of data and usually leads to efficient algorithm implementations.
- In some cases, broadcasting is a bad idea because it leads to **inefficient use of memory** that slows computation.



Rules of broadcasting

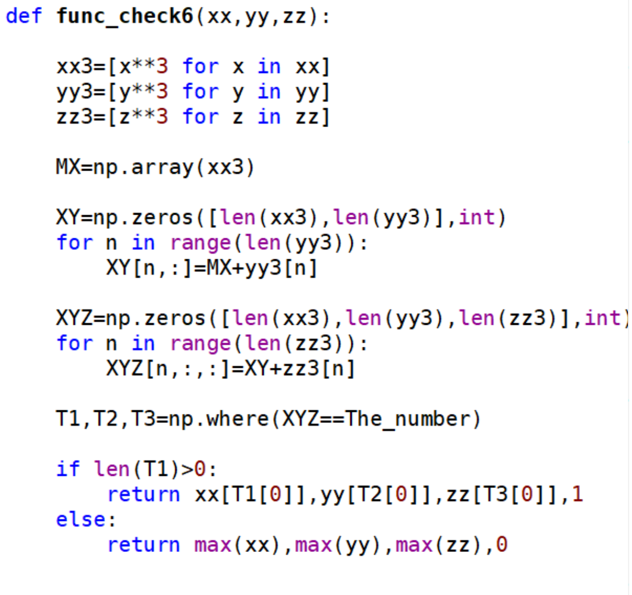
- The basic rule of broadcasting is “**the size of the trailing axes for both arrays in an operation must either be the same size or one of them must be one.**”
  - ❑ If this condition is not met, a **ValueError('frames are not aligned')** exception is thrown indicating that two arrays have incompatible shapes.
  - ❑ The size of the result array is the maximum size along each dimension from the input arrays.
  - ❑ Two arrays don’t need to have the same number of dimensions.



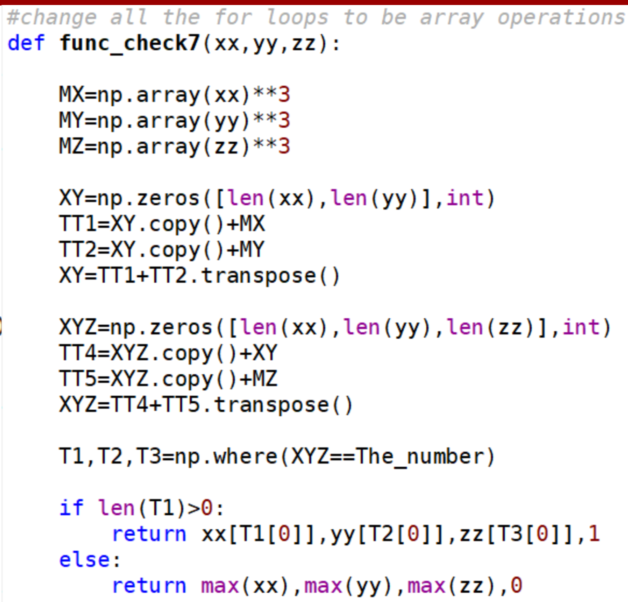


- **DO THE TEST BEFORE USE IT.**

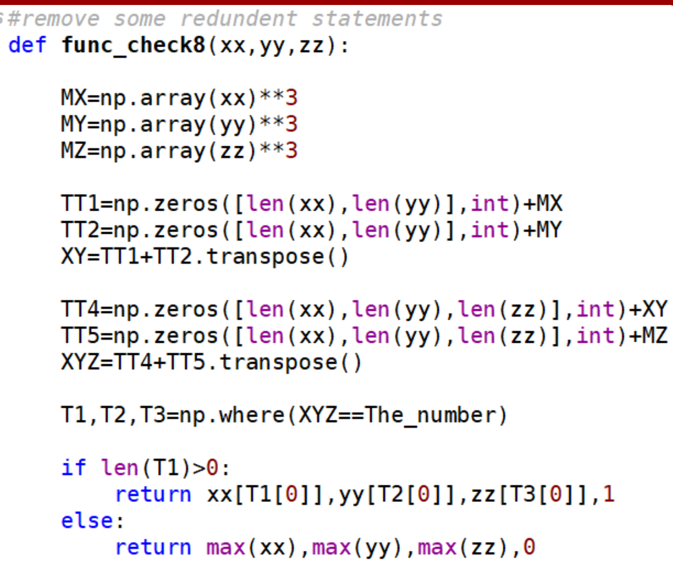
Not always good



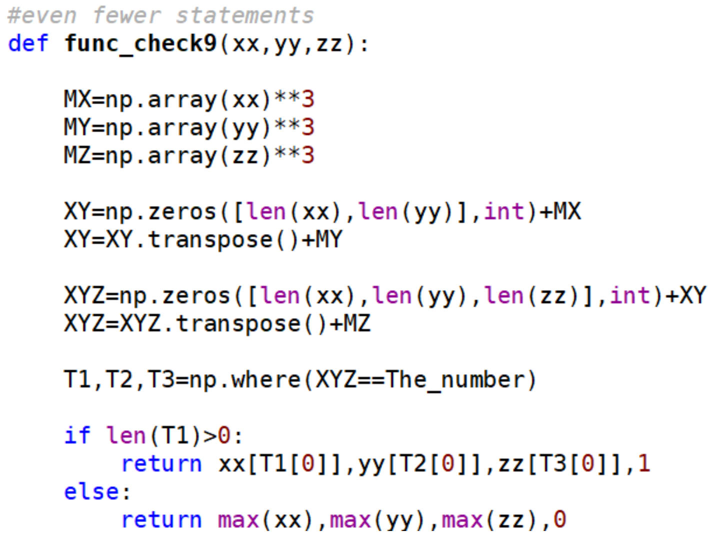
```
def func_check6(xx,yy,zz):

    xx3=[x**3 for x in xx]
    yy3=[y**3 for y in yy]
    zz3=[z**3 for z in zz]

    MX=np.array(xx3)

    XY=np.zeros([len(xx3),len(yy3)],int)
    for n in range(len(yy3)):
        XY[n,:]=MX+yy3[n]

    XYZ=np.zeros([len(xx3),len(yy3),len(zz3)],int)
    for n in range(len(zz3)):
        XYZ[n,:,:]=XY+zz3[n]

    T1,T2,T3=np.where(XYZ==The_number)

    if len(T1)>0:
        return xx[T1[0]],yy[T2[0]],zz[T3[0]],1
    else:
        return max(xx),max(yy),max(zz),0
```



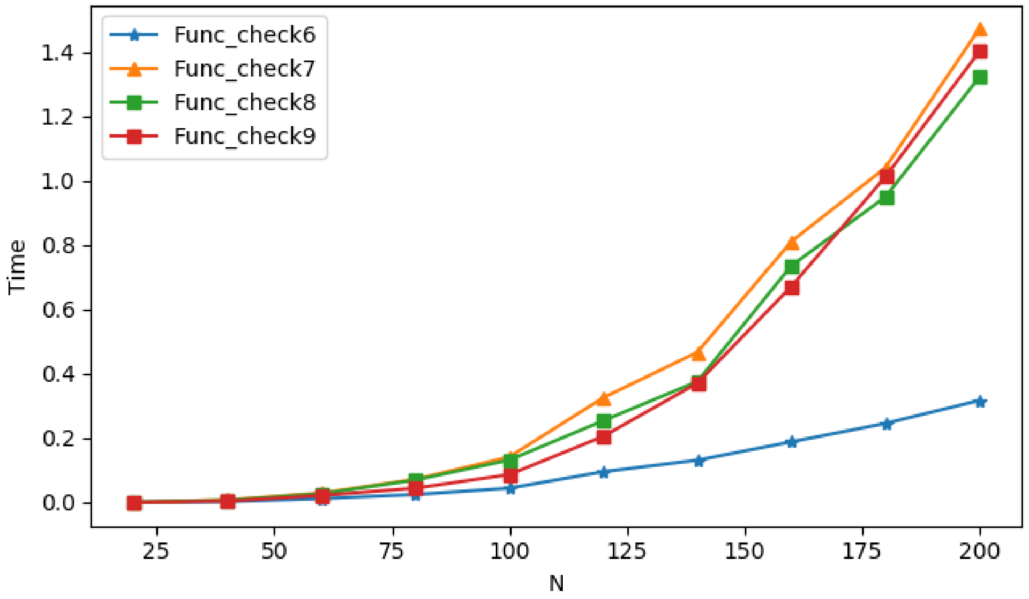
```
#change all the for loops to be array operations
def func_check7(xx,yy,zz):

    MX=np.array(xx)**3
    MY=np.array(yy)**3
    MZ=np.array(zz)**3

    XY=np.zeros([len(xx),len(yy)],int)
    TT1=XY.copy()+MX
    TT2=XY.copy()+MY
    XY=TT1+TT2.transpose()

    XYZ=np.zeros([len(xx),len(yy),len(zz)],int)
    TT4=XYZ.copy()+XY
    TT5=XYZ.copy()+MZ
    XYZ=TT4+TT5.transpose()

    T1,T2,T3=np.where(XYZ==The_number)

    if len(T1)>0:
        return xx[T1[0]],yy[T2[0]],zz[T3[0]],1
    else:
        return max(xx),max(yy),max(zz),0
```



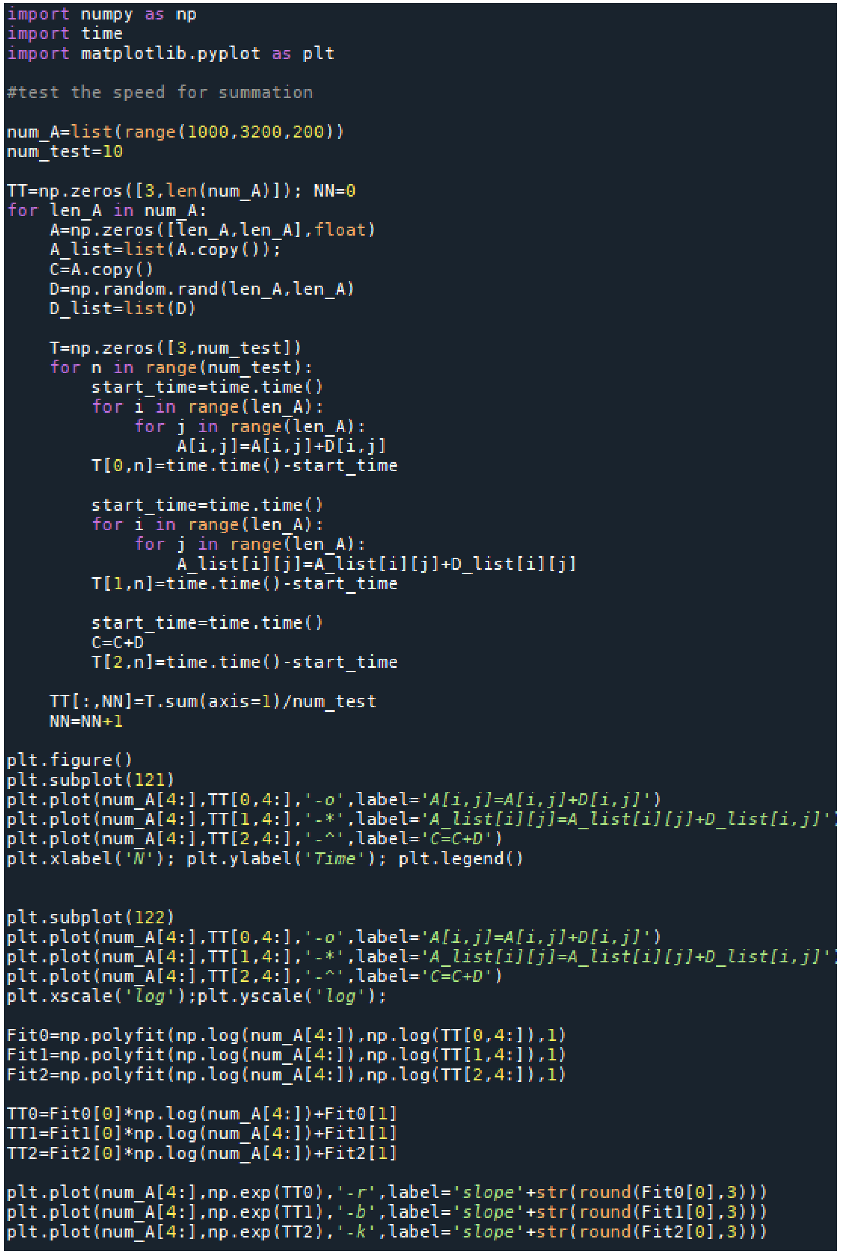
```
#remove some redundent statements
def func_check8(xx,yy,zz):

    MX=np.array(xx)**3
    MY=np.array(yy)**3
    MZ=np.array(zz)**3

    TT1=np.zeros([len(xx),len(yy)],int)+MX
    TT2=np.zeros([len(xx),len(yy)],int)+MY
    XY=TT1+TT2.transpose()

    TT4=np.zeros([len(xx),len(yy),len(zz)],int)+XY
    TT5=np.zeros([len(xx),len(yy),len(zz)],int)+MZ
    XYZ=TT4+TT5.transpose()

    T1,T2,T3=np.where(XYZ==The_number)

    if len(T1)>0:
        return xx[T1[0]],yy[T2[0]],zz[T3[0]],1
    else:
        return max(xx),max(yy),max(zz),0
```



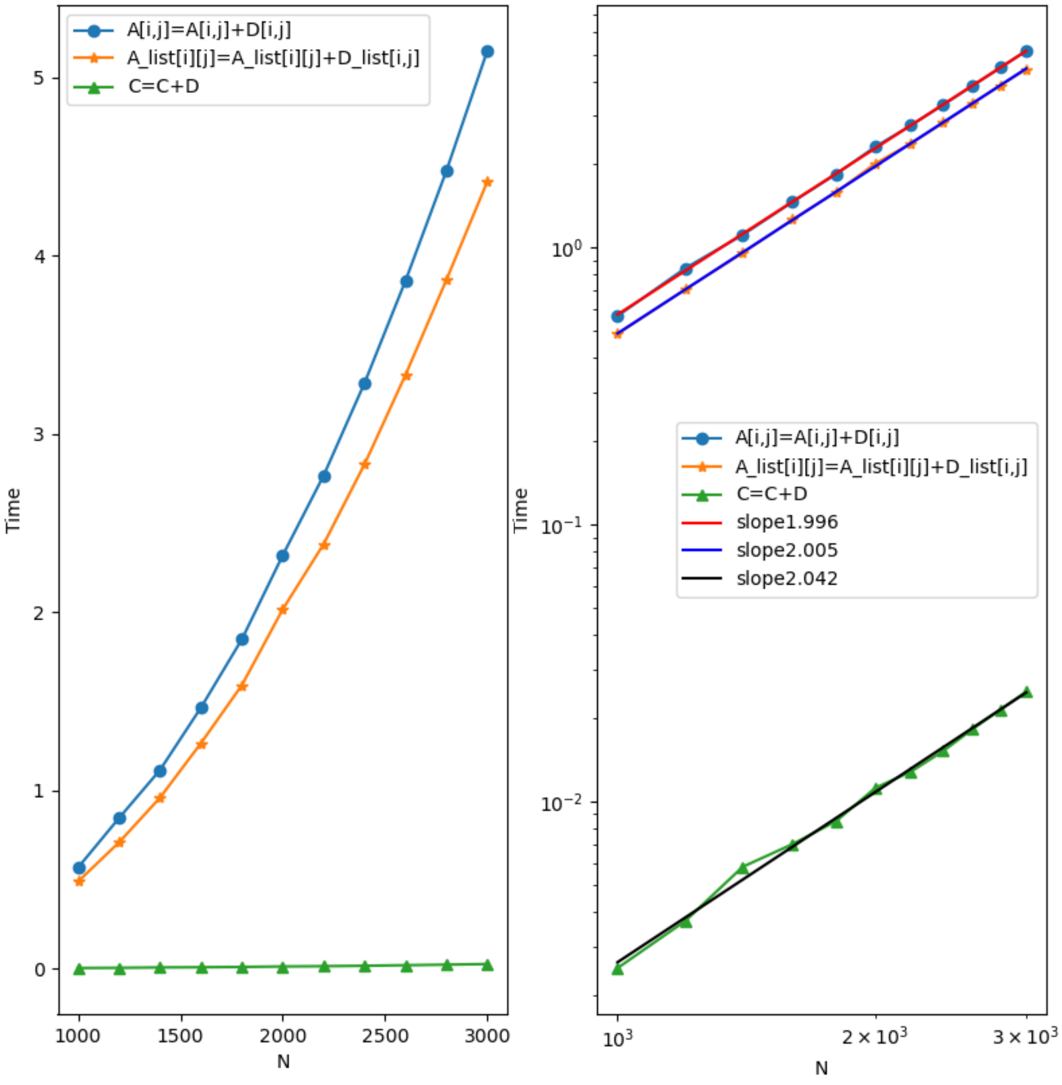
```
#even fewer statements
def func_check9(xx,yy,zz):

    MX=np.array(xx)**3
    MY=np.array(yy)**3
    MZ=np.array(zz)**3

    XY=np.zeros([len(xx),len(yy)],int)+MX
    XY=XY.transpose()+MY

    XYZ=np.zeros([len(xx),len(yy),len(zz)],int)+XY
    XYZ=XYZ.transpose()+MZ

    T1,T2,T3=np.where(XYZ==The_number)

    if len(T1)>0:
        return xx[T1[0]],yy[T2[0]],zz[T3[0]],1
    else:
        return max(xx),max(yy),max(zz),0
```



Speed comparison



```
import numpy as np
import time
import matplotlib.pyplot as plt

#test the speed for summation

num_A=list(range(1000,3200,200))
num_test=10

TT=np.zeros([3,len(num_A)]); NN=0
for len_A in num_A:
    A=np.zeros([len_A,len_A],float)
    A_list=list(A.copy());
    C=A.copy()
    D=np.random.rand(len_A,len_A)
    D_list=list(D)

    T=np.zeros([3,num_test])
    for n in range(num_test):
        start_time=time.time()
        for i in range(len_A):
            for j in range(len_A):
                A[i,j]=A[i,j]+D[i,j]
        T[0,n]=time.time()-start_time

        start_time=time.time()
        for i in range(len_A):
            for j in range(len_A):
                A_list[i][j]=A_list[i][j]+D_list[i][j]
        T[1,n]=time.time()-start_time

        start_time=time.time()
        C=C+D
        T[2,n]=time.time()-start_time

    TT[:,NN]=T.sum(axis=1)/num_test
    NN=NN+1

plt.figure()
plt.subplot(121)
plt.plot(num_A[4:],TT[0,4:],'-o',label='A[i,j]=A[i,j]+D[i,j]')
plt.plot(num_A[4:],TT[1,4:],'-*',label='A_list[i][j]=A_list[i][j]+D_list[i,j]'
plt.plot(num_A[4:],TT[2,4:],'-^',label='C=C+D')
plt.xlabel('N'); plt.ylabel('Time'); plt.legend()


plt.subplot(122)
plt.plot(num_A[4:],TT[0,4:],'-o',label='A[i,j]=A[i,j]+D[i,j]')
plt.plot(num_A[4:],TT[1,4:],'-*',label='A_list[i][j]=A_list[i][j]+D_list[i,j]'
plt.plot(num_A[4:],TT[2,4:],'-^',label='C=C+D')
plt.xscale('log');plt.yscale('log');

Fit0=np.polyfit(np.log(num_A[4:]),np.log(TT[0,4:]),1)
Fit1=np.polyfit(np.log(num_A[4:]),np.log(TT[1,4:]),1)
Fit2=np.polyfit(np.log(num_A[4:]),np.log(TT[2,4:]),1)

TT0=Fit0[0]*np.log(num_A[4:])+Fit0[1]
TT1=Fit1[0]*np.log(num_A[4:])+Fit1[1]
TT2=Fit2[0]*np.log(num_A[4:])+Fit2[1]

plt.plot(num_A[4:],np.exp(TT0),'-r',label='slope'+str(round(Fit0[0],3)))
plt.plot(num_A[4:],np.exp(TT1),'-b',label='slope'+str(round(Fit1[0],3)))
plt.plot(num_A[4:],np.exp(TT2),'-k',label='slope'+str(round(Fit2[0],3)))
```



Create arrays: use intrinsic numpy array creation

- **A=numpy.zeros([N_1,N_2,…,N_m],variable_type)**

N_i the size of ith dimension. All the elements are 0.

- **A=numpy.ones([N_1,N_2,…,N_m],variable_type)**

All the elements are 1.

- **A=numpy.empty([N_1,N_2,…,N_m],variable_type)**

Question: what is the value of the elements?

- **A= numpy.diag(E_{11},E_{22},...,E_{nn})**

A diagonal matrix with diagonal matrix elements as E_{11},E_{22},...,E_{nn}

- **A= numpy.arange(Min,Max,interval)**

create arrays with regularly incrementing values

- **A= numpy.linspace(Min,Max,number_of_elments)**

create arrays with regularly incrementing values, both Min and Max included

- **Many other methods. DO THE TEST BEFORE USE IT.**

Create arrays: Use of special library functions

- **A=numpy.random.rand(N_1,N_2,…,N_m)**
  A matrix with elements randomly sampled between 0 and 1.
  (**Question**: Whether 0 or 1 included??)

- **A= numpy.random.randint(Min,Max,[N_1,N_2,…,N_m])**
  A matrix with integer elements randomly sampled between Min and Max.
  (**Question**: Whether Min or Max included??)

- **A= numpy.random.randn([N_1,N_2,…,N_m])**
  A matrix with integer elements randomly sampled from the standard normal distribution.
  (**Question**: how to generate samples for a general normal distribution?)

Create arrays: conversion from list or tuple

- We can convert a list/tuple A to be an array by **numpy.array(A).** Be careful about the variable type when you do the conversion
- We can also convert an array to be a list using the function **list()**
- Check whether an element a in a list or an array A

Value=a in A

- Another way to check whether an element a in an array A

Value=(A==a).any() Question: whether is this faster??

Question: do we use list or array if we want to check whether a number in a large group of numbers??

```
import time
B=np.zeros(100000000);
B[100]=2; B[-100]=3; A=list(B);
tst=3; #how about tst=2 or 1??
st=time.time(); tst in A; print(time.time()-st)
st=time.time(); tst in B; print(time.time()-st)
st=time.time(); (B==tst).any();
print(time.time()-st)
```

Revisit the quiz

- We prepare a 3D list in the following code. Please find out how many 0,1 and -1 in the 3D list.

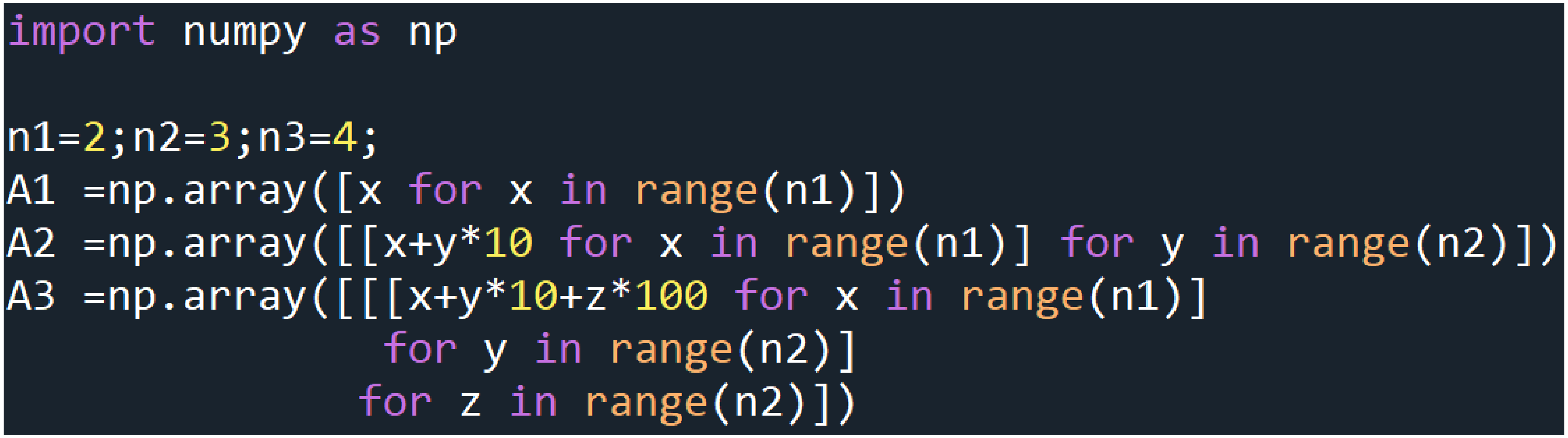
```
from random import randint, seed
n=20;m=30;p=40;
seed(10)
A =[[[randint(-1,1) for x in range(n)] for y in range(m)] for z in range(p)]

### write your codes bellow to get the answer
import numpy as np
B=np.array(A)
Num1=sum(sum(sum(B==1)))
Num2=sum(sum(sum(B==0)))
Num3=sum(sum(sum(B==-1)))

### do a check
print(Num1+Num2+Num3-n*m*p)
```

indexing

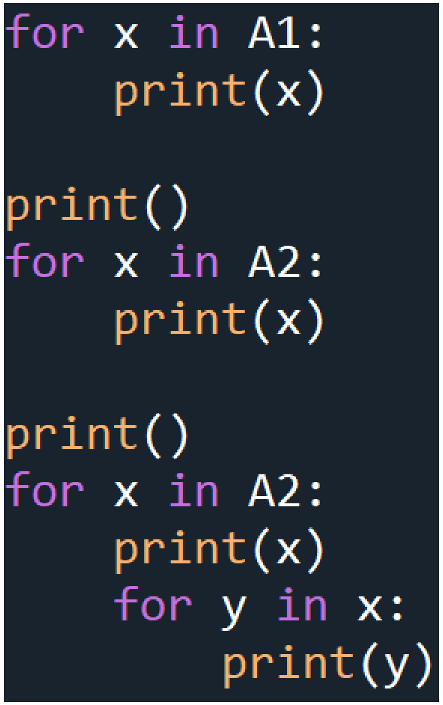
- $A[n_1,n_2,\ldots,n_M]$: To get one element in the position $(n_1,n_2,\ldots,n_M)$
- **slicing**: Get multiple elements at once.
  - ➢ A [start:end:step]. Default value, start=0, end=-1, step=1
  - ➢ The slicing can be negative, e.g., reverse the array A[::-1]
  - ➢ One can use slicing in any dimension
- Using **tuple** to get multiple elements at once, e.g., we have M tuple T_1 and $T_2,\ldots, T_M$, and they have the **same size** K then you can use $A[T_1,T_2,\ldots,T_M]$ to get the all $A[T_1[0],T_2[0],\ldots,T_M[0]]$, $A[T_1[1],T_2[1],\ldots,T_M[1]],\ldots, A[T_1[K-1],T_2[K-1],\ldots,T_M[K-1]]$. (**Note1**: You can also use list or array to do these, while it is better to use tuple; **Note2**: you can use high-dimension tuple)
- To facilitate easy matching of array shapes, **numpy.newaxis** object can be used to add new dimensions with a size of 1.
  x = np.arange(5); x[:,np.newaxis] + x[np.newaxis,:]

Array Iterating

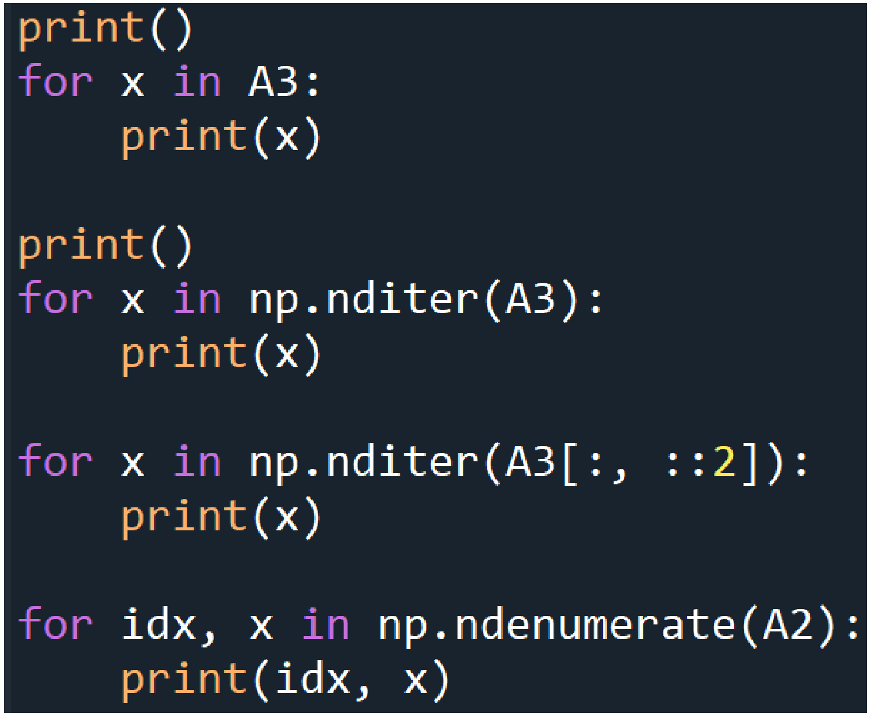
- Iterating means going through elements one by one. As we deal with multi-dimensional arrays in numpy, we can do this using basic **for** loop of python.
- If we iterate on a 1-D array it will go through each element one by one. In a 2-D array it will go through all the rows. To return the actual values, the scalars, we have to iterate the arrays in each dimension. If we iterate on a n-D array it will go through n-1th dimension one by one.
- The function **nditer()** is a helping function that can be used from very basic to very advanced iterations. It solves some basic issues which we face in iteration. numpy.nditer — NumPy v1.25 Manual
- Enumeration means mentioning sequence number of somethings one by one. Sometimes we require corresponding index of the element while iterating, the **ndenumerate()** method can be used for those cases.



```python
import numpy as np

n1=2;n2=3;n3=4;
A1 =np.array([x for x in range(n1)])
A2 =np.array([[x+y*10 for x in range(n1)] for y in range(n2)])
A3 =np.array([[[x+y*10+z*100 for x in range(n1)]
               for y in range(n2)]
              for z in range(n2)])
```



```python
for x in A1:
    print(x)

print()
for x in A2:
    print(x)

print()
for x in A2:
    print(x)
    for y in x:
        print(y)
```



```python
print()
for x in A3:
    print(x)

print()
for x in np.nditer(A3):
    print(x)

for x in np.nditer(A3[:, ::2]):
    print(x)

for idx, x in np.ndenumerate(A2):
    print(idx, x)
```

Shape of high-dimension array

- The size of each dimensions of the array A is **A.shape**; and the number of dimensions is **A.ndim**
- We can change the shape of an array A using **A.reshape(N_1,N_2,…,N_m)** (**Question**: what is the ordering?)
- We can change an array A to be one-dimension use **A.flatten()**
- Axes are defined for high-dimension arrays Many operations can take place along one of these axes, e.g.

  x = np.arange(12).reshape((3,4))

  x.sum();   x.sum(axis=0);   x.sum(axis=1)
- We can use **numpy.transpose()** to change the dimension order of an array A. **B=numpy.transpose(A,axes=[2,1,0])**
- Row major order (default). The rightmost index "varies the fastest". Use keyword **order='C' or 'F'** to set it. For example,

  a=numpy.arange(27).reshape(3,3,3,order='C')

  a=numpy.arange(27).reshape(3,3,3,order='F')

Shape and Reshape()

- The shape of an array is the number of elements in each dimension. ***a.shape***

- We can **reshape** the array by adding or removing dimensions or changing number of elements in each dimension.
- We can reshape an array into any shape as long as the elements required for reshaping are equal in both shapes.
- You are allowed to have one "unknown" dimension. Pass -1 as the value, and NumPy will calculate this number for you.

e.g.

```
A=np.array([[1, 2, 3, 4], [5, 6, 7, 8]]); B=A.reshape(2,2,-1)
C=A.reshape(-1)
```

Q: Is B or C an independent new array??

Joining Array

- Joining means putting contents of two or more arrays in a single array. In NumPy we join arrays ***by axes***.
- ***concatenate()***, join arrays along with the axis. If axis is not explicitly passed, it is taken as 0.
- **stack()**, join array along a new axis; **hstack(),** stack along rows; **vstack()**, stack along columns; **dstack()**, stack along heights/depths.

```
arr1 = np.array([[1, 2], [3, 4]])
arr2 = np.array([[5, 6], [7, 8]])
arr_j0 = np.concatenate((arr1, arr2), axis=0)
arr_j1 = np.concatenate((arr1, arr2), axis=1)

print(arr_j0)
print(arr_j1)
```

```
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
arr_s0 = np.stack((arr1, arr2), axis=0)
arr_s1 = np.stack((arr1, arr2), axis=0)
arr_hs = np.hstack((arr1, arr2))
arr_vs = np.vstack((arr1, arr2))
arr_ds = np.dstack((arr1, arr2))
print(arr_s0)
print(arr_s1)
print(arr_hs)
print(arr_vs)
print(arr_ds)
```

Splitting Array

- We use ***array_split()*** for splitting one array into multiple.
- If the array has less elements than required, it will adjust from the end accordingly.
- For high dimension array, you can specify which ***axis*** you want to do the split around.
- Similarly, we can have ***hsplit()***, ***vsplit()*** and ***dsplit()***.

```
arr = np.array([1, 2, 3, 4, 5, 6])
newarr = np.array_split(arr, 4)

arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12], [13, 14,
15], [16, 17, 18]])
newarr2d_a0 = np.array_split(arr2d, 3)
newarr2d_a1 = np.array_split(arr2d, 3, axis=1)

print(newarr2d_a0); print(newarr2d_a1)
```

Searching Arrays

- To search an array, use the ***where()*** method.
- ***searchsorted()*** can performs a binary search in the array, and returns the index where the specified value would be inserted to maintain the search order. By default the left most index is returned, but we can give ***side='right'*** to return the right most index instead.
- To search for more than one value, use an array with the specified values.

```
arr = np.array([1, 2, 3, 4, 2, 6])
x = np.where(arr%2 == 0)
print(x)

#It starts the search from the left and returns the first index
# where the number 2.5 is no longer larger than the next value.
x_s = np.searchsorted(arr, 2.5)
x_ms = np.searchsorted(arr, [2.5,4.5,6.5])
```

Sorting Arrays

- The NumPy ndarray object has a function called ***sort()***, that will sort a specified array. This method returns a copy of the array, leaving the original array unchanged. This method can also use axis to specify which axis.
- ***argsort()***, returns an array of indices of the same shape as a that index data along the given axis in sorted order.
- Sort a numpy 2d array by a certain row with columns maintained, on need to use NumPy indexing.

```
arr = np.array([[3,2,4], [5,0,1],[7,9,3]])
print(np.sort(arr,axis=0)) ;print(np.sort(arr,axis=1))

ind=np.argsort(arr,axis=0);
print(np.take_along_axis(arr,ind,axis=0))

#sort based on the first row
print(arr[:, arr[0].argsort()])
```

Filter Array

- You filter an array using a boolean index list. If the value at an index is True that element is contained in the filtered array, otherwise the element is excluded.
- filter must be of the same shape as the initial dimensions of the array being indexed.
- There are many different ways to create the filter array. For example, one can use ***for*** loop to do it.
- Be careful of the dimension of the results.

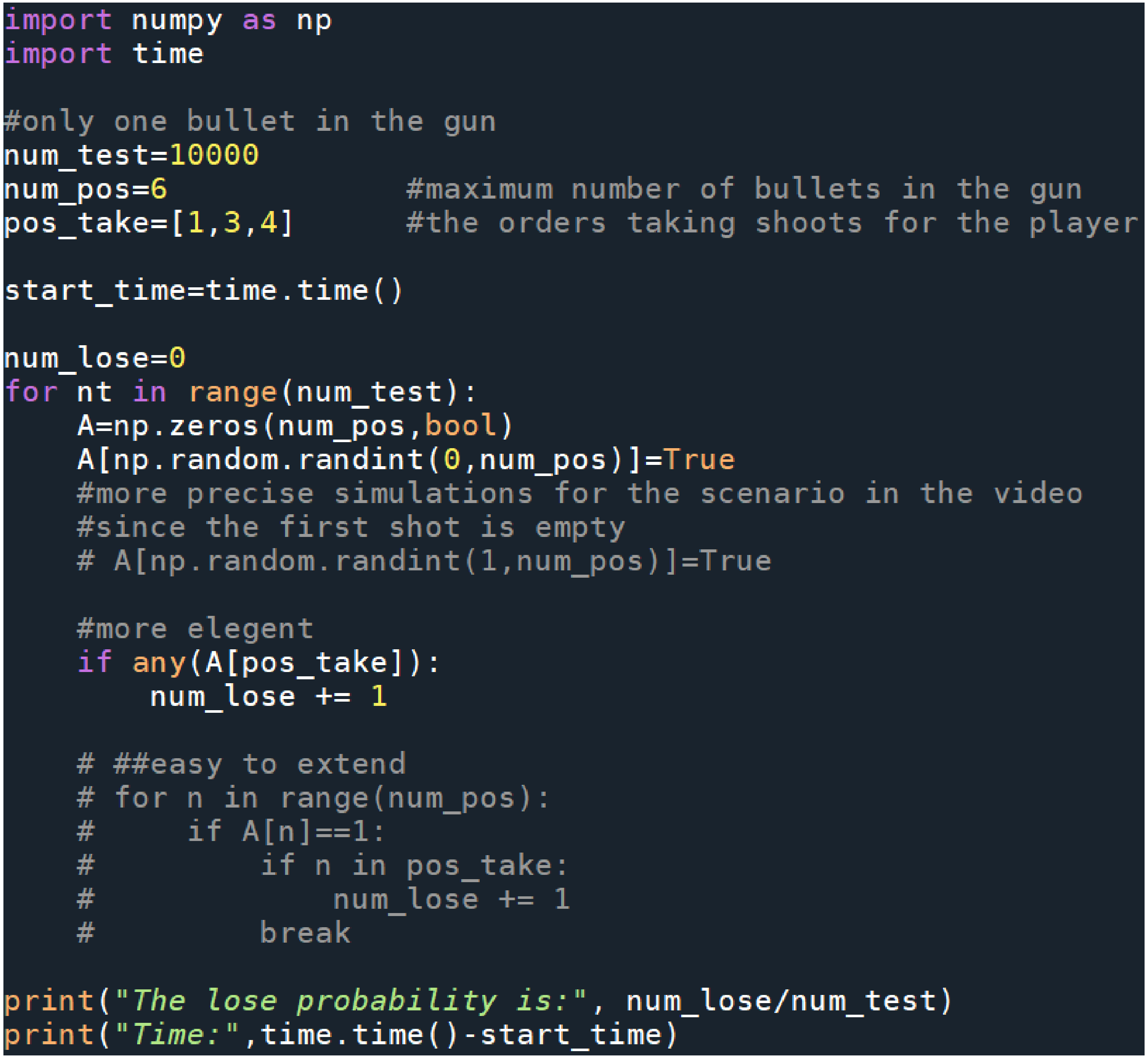
```
arr = np.array([41,42,43,44])
x = [True,False,True,False]
newarr = arr[x]; print(newarr)

filter_arr = arr > 42; print(arr[filter_arr])

filter_arr = arr%2==0
print(arr[filter_arr])
```

```
filter_arr = []

for element in arr:
if element % 2 == 0:
    filter_arr.append(True)
  else:
    filter_arr.append(False)
```

Simulation: a simple method for modeling

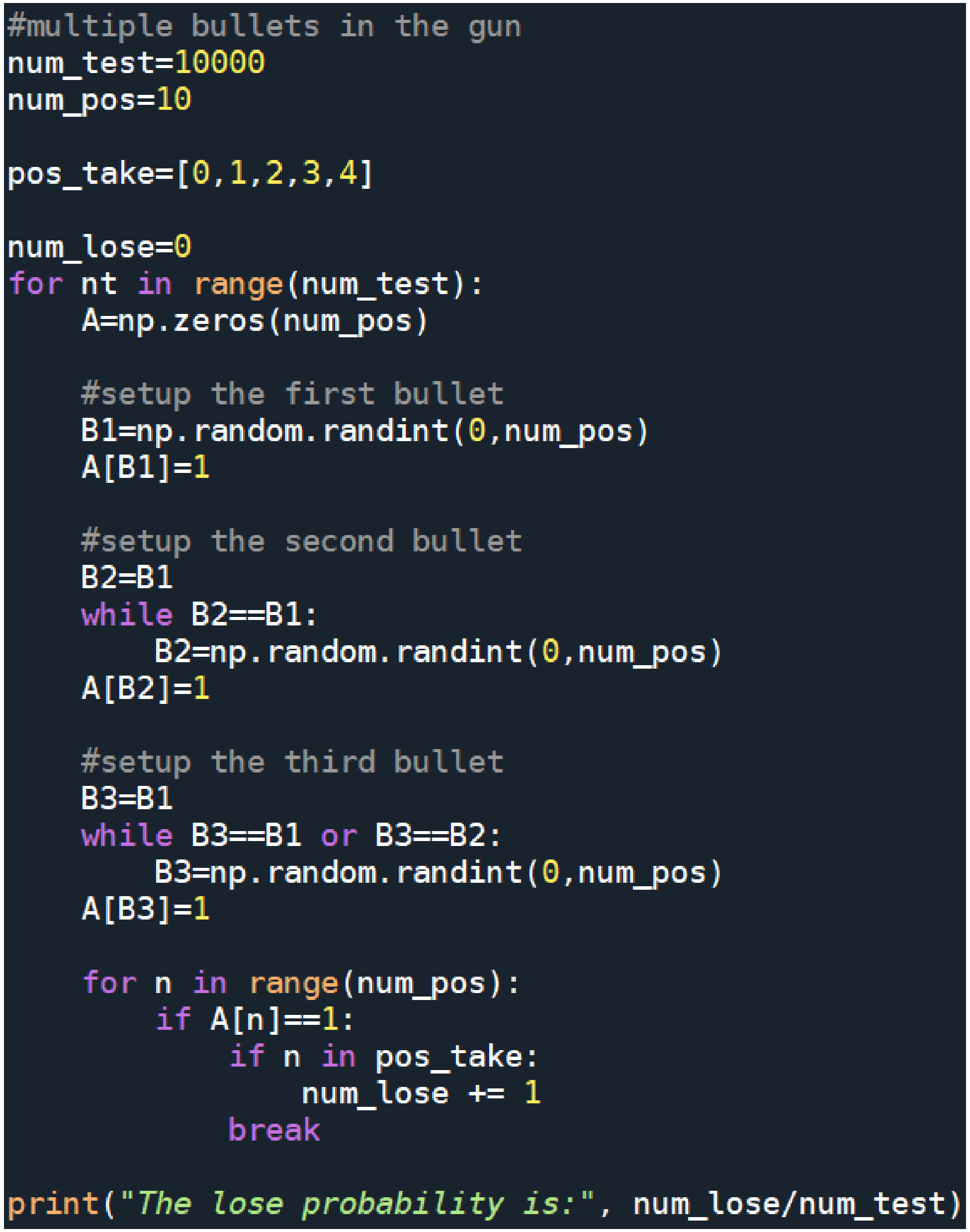
Does this behavior show
more courage??

Simulation: a simple method for modeling



```
import numpy as np
import time

#only one bullet in the gun
num_test=10000
num_pos=6             #maximum number of bullets in the gun
pos_take=[1,3,4]      #the orders taking shoots for the player

start_time=time.time()

num_lose=0
for nt in range(num_test):
    A=np.zeros(num_pos,bool)
    A[np.random.randint(0,num_pos)]=True
    #more precise simulations for the scenario in the video
    #since the first shot is empty
    # A[np.random.randint(1,num_pos)]=True

    #more elegent
    if any(A[pos_take]):
        num_lose += 1

    # ##easy to extend
    # for n in range(num_pos):
    #     if A[n]==1:
    #         if n in pos_take:
    #             num_lose += 1
    #         break

print("The lose probability is:", num_lose/num_test)
print("Time:",time.time()-start_time)
```

How about more bullets in the gun?



```
#multiple bullets in the gun
num_test=10000
num_pos=10

pos_take=[0,1,2,3,4]

num_lose=0
for nt in range(num_test):
    A=np.zeros(num_pos)

    #setup the first bullet
    B1=np.random.randint(0,num_pos)
    A[B1]=1

    #setup the second bullet
    B2=B1
    while B2==B1:
        B2=np.random.randint(0,num_pos)
    A[B2]=1

    #setup the third bullet
    B3=B1
    while B3==B1 or B3==B2:
        B3=np.random.randint(0,num_pos)
    A[B3]=1

    for n in range(num_pos):
        if A[n]==1:
            if n in pos_take:
                num_lose += 1
            break

print("The lose probability is:", num_lose/num_test)
```