07 Exercises

September 13, 2020

Make NumPy available:

```
[0]: import numpy as np
     import numba
```

0.1 Exercise 07.1 (indexing and timing)

Create two very long NumPy arrays `x` and `y` and sum the arrays using:

1. The NumPy addition syntax, `z = x + y`; and
2. A `for` loop that computes the sum entry-by-entry

Compare the time required for the two approaches for vectors of different lengths (use a very long vector for the timing). The values of the array entries are not important for this test. Use `%time` to report the time.

Hint: To loop over an array using indices, try a construction like:

```
[13]: x = np.ones(10)
      y = np.ones(len(x))
      for i in range(len(x)):
          print(x[i]*y[i])
```

```
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
```

(1) Add two vectors using built-in addition operator:

```
[0]: x = np.random.rand(100000000)
     y = np.random.rand(100000000)
```

```
[15]: %time z0 = x+y
      z0
```

```
CPU times: user 204 ms, sys: 933 µs, total: 205 ms
Wall time: 205 ms
```

```
[15]: array([1.27532589, 1.19715057, 0.50306628, …, 0.987681  , 1.86086337,
             0.86666424])
```

(2) Add two vectors using own implementation:

```
[16]: @numba.jit
      def compute_sum(x, y):
          z = np.zeros(len(x))
          for i in range(len(x)):
              z[i] = x[i]+y[i]
          return z
      %time norm =compute_sum(x, y)
      print(norm)
```

```
CPU times: user 514 ms, sys: 12.1 ms, total: 526 ms
Wall time: 558 ms
[1.27532589 1.19715057 0.50306628 … 0.987681   1.86086337 0.86666424]
```

0.1.1 Optional extension: just-in-time (JIT) compilation

You will see a large difference in the time required between your NumPy and 'plain' Python implementations. This is due to Python being an *interpreted* language as opposed to a *compiled* language. A way to speed up plain Python implementions is to convert the interpreted Python code into compiled code. A tool for doing this is Numba.

Below is an example using Numba and JIT to accelerate a computation:

```
[17]: !pip -q install numba
      import numba
      import math

      def compute_sine_native(x):
          z = np.zeros(len(x))
          for i in range(len(z)):
              z[i] = math.sin(x[i])
          return z

      @numba.jit
      def compute_sine_jit(x):
          z = np.zeros(len(x))
          for i in range(len(z)):
              z[i] = math.sin(x[i])
          return z
```

```
x = np.ones(10000000)
%time z = compute_sine_native(x)
compute_sine_jit(x)
%time z = compute_sine_jit(x)
```

```
CPU times: user 2.94 s, sys: 0 ns, total: 2.94 s
Wall time: 2.94 s
CPU times: user 159 ms, sys: 928 µs, total: 160 ms
Wall time: 160 ms
```

Task: Test if Numba can be used to accelerate your implementation that uses indexing to sum two arrays, and by how much.

0.2 Exercise 07.2 (member functions and slicing)

Anonymised scores (out of 60) for an examination are stored in a NumPy array. Write:

1. A function that takes a NumPy array of the raw scores and returns the scores as percentages, sorted from lowest to highest (try using `scores.sort()`, where `scores` is a NumPy array holding the scores).
2. A function that returns the maximum, minimum and mean of the raw scores as a dictionary with the keys '`min`', '`max`' and '`mean`'. Use the NumPy array functions `min()`, `max()` and `mean()` to do the computation, e.g. `max = scores.max()`.

   Design your function for the min, max and mean to optionally exclude the highest and lowest scores from the computation of the min, max and mean.

   Hint: sort the array of scores and use array slicing to exclude the first and the last entries.

Use the scores

```
scores = np.array([58.0, 35.0, 24.0, 42, 7.8])
```

to test your functions.

[18]:
```
x = range(5)
x
```

[18]:
```
range(0, 5)
```

[0]:
```
def to_percentage_and_sort(scores):
    nscores = []
    for i in range(len(scores)):
        nscores.append(scores[i]/60*100)
    nscores.sort()
    return nscores

def statistics(scores, exclude=False):
    x = {'max':0.0,
          'min':0.0,
          'mean':0.0}
    if exclude==False:
        x['max']=scores.max()
        x['min']=scores.min()
        x['mean']=scores.mean()
        return x
    else:
        scores.sort()
        x['max']=scores[1:-1].max()
        x['min']=scores[1:-1].min()
        x['mean']=scores[1:-1].mean()
        return x
```

[20]:
```
scores = np.array([58.0, 35.0, 24.0, 42, 7.8])
print(to_percentage_and_sort(scores))
assert np.isclose(to_percentage_and_sort(scores), [ 13.0, 40.0, 58.33333333, ␣
 ↪70.0, 96.66666667]).all()

s0 = statistics(scores)
print(s0)
assert round(s0["min"] - 7.8, 10) == 0.0
assert round(s0["mean"] - 33.36, 10) == 0.0
assert round(s0["max"] - 58.0, 10) == 0.0

s1 = statistics(scores, True)
print(s1)
assert round(s1["min"] - 24.0, 10) == 0.0
assert round(s1["mean"] - 33.666666666666667, 10) == 0.0
assert round(s1["max"] - 42.0, 10) == 0.0
```

```
[13.0, 40.0, 58.333333333333336, 70.0, 96.66666666666667]
{'max': 58.0, 'min': 7.8, 'mean': 33.36}
{'max': 42.0, 'min': 24.0, 'mean': 33.666666666666664}
```

0.3 Exercise 07.3 (slicing)

For the two-dimensional array

[21]:
```
A = np.array([[4.0, 7.0, -2.43, 67.1],
             [-4.0, 64.0, 54.7, -3.33],
             [2.43, 23.2, 3.64, 4.11],
             [1.2, 2.5, -113.2, 323.22]])
print(A)
```

```
[[   4.      7.     -2.43   67.1 ]
 [  -4.     64.     54.7    -3.33]
 [   2.43   23.2     3.64    4.11]
 [   1.2     2.5  -113.2   323.22]]
```

use array slicing for the below operations, printing the results to the screen to check. Try to use array slicing such that your code would still work if the dimensions of A were enlarged.

1. Extract the third column as a 1D array

```
[24]: print(A[2,:])
```

```
[ 2.43 23.2   3.64  4.11]
```

2. Extract the first two rows as a 2D sub-array

```
[28]: print(A[:2,:])
```

```
[[ 4.    7.   -2.43 67.1 ]
 [-4.   64.   54.7  -3.33]]
```

3. Extract the bottom-right 2×2 block as a 2D sub-array

```
[29]: print(A[:2,:2])
```

```
[[ 4.  7.]
 [-4. 64.]]
```

4. Sum the last column

```
[33]: print(A[3,:])
      print('Sum : {}'.format(np.sum(A[3,:])))
```

```
[   1.2     2.5  -113.2   323.22]
Sum : 213.72000000000003
```

Compute transpose Compute the transpose of A (search online to find the function/syntax to do this).

```
[35]: print(A.transpose())
```

```
[[   4.     -4.      2.43    1.2 ]
 [   7.     64.     23.2     2.5 ]
 [  -2.43   54.7     3.64 -113.2 ]
 [  67.1    -3.33    4.11  323.22]]
```

0.4 Exercise 07.4 (optional extension)

In a previous exercise you implemented the bisection algorithm to find approximate roots of a mathematical function. Use the SciPy bisection function `optimize.bisect` (http://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.optimize.bisect.html) to find roots of the mathematical function that was used in the previous exercise. Compare the results computed by SciPy and your program from the earlier exercise, and compare the computational time (using `%time`).

[0]:
```python
def my_f(x):
    "Evaluate polynomial function"
    return x**3 - 6*x**2 + 4*x + 12
```

[41]:
```python
def compute_root(f, x0, x1, tol, max_it):
    "Compute roots of a function using bisection"
    # Implement bisection algorithm here, and return when tolerance is␣
 ↪satisfied or
    # Initial end points
    error = tol + 1.0
    # Iterate until tolerance is met
    it = 0
    while error > tol:
        it += 1
        # Compute midpoint
        x_mid = (x0 + x1)/2
        f = my_f(x0)
        f_mid = my_f(x_mid)
        # Condition:
        if f*f_mid < 0:
            x1=x_mid
        else:
            x0=x_mid
        if abs(f_mid)< tol:
            break
        # number of iterations exceeds max_it
    if it > max_it:
        print("Oops, iteration count is very large. Breaking out of while loop.
 ↪")
    # Return the approximate root, value of f(x) and the number of iterations  ␣
 ↪
    return x_mid, f_mid, it

x, f_x, num_it = compute_root(my_f, x0=3, x1=6, tol=1.0e-6, max_it=1000)
print(x, f_x, num_it)
%time num_it
```

```
4.534070134162903 -7.047073751209609e-07 23
CPU times: user 3 µs, sys: 0 ns, total: 3 µs
Wall time: 5.25 µs
```

[41]: 23

[47]:
```python
import scipy
from scipy import optimize

value=scipy.optimize.bisect(my_f, a=3, b=6, rtol=1.0e-6, maxiter=1000)
```

```
%time value
```

```
CPU times: user 3 µs, sys: 0 ns, total: 3 µs
Wall time: 5.48 µs
```

```
[47]: 4.534071922302246
```