Simple Speed-Up Tricks

Note: these examples use python3 and are sometimes not terribly Pythonic. Some examples are deliberately inefficient to illustrate how to improve common issues

These examples compare the speed of different ways of solving a simple problem. To run the examples, either start python and import the example, then run the "main" function, or run python{3}{name.py} and observe. Examine the code, run the profiler etc to see what's changed and why it helps.

- 01_permutations_max.py

  - Finds whether a sequence of numbers contains one bigger than a target

  Info:

  We generate num_items values, of which a random set of size num_high are value 10, and the rest value 1. We then use several ways to show that the list contains values bigger than 5. num_items and num_high can be given on the command line (both must be given and they default to 10000 and 10 respectively) or supplied to main, which runs all 6 methods on the SAME data

  Try:

  Which method is fastest for (10000,10)? Does this change for different parameters? Does it change with and without the JIT (use_jit = True/False)?

  The last method uses a simple loop with early-break once an element is found. Why does the timing change so much if you change num_high?

  Based on the previous part, can we guess whether numpy's built-in "in" method has an early break?

- 02_permutations_early_break.py

  - Breaking early can really help

  Info:

  This example runs multiple iterations of the simple loop elemenf finders, with and without the JIT and with and without an early termination. As above you can specifiy num_els and num_high (both or neither) and if you specify these you can also specify the number of iterations

  Try:

  Run a few times with the default values. Note how the early break gives a much wider range of runtimes, particularly the minimum

  Why might the early term not _always_ take less time (particularly the max)? About how much faster is it in the average. Why? You might need rather more than 100 iterations to get reliable stats

  Change num_high to be large. Why do things get so much faster?

  Are there circumstance where early break might be bad for performance?

- 03_bisect_example.py

  - For ordered data, bisection is king - and there's a built-in for it!

  Info:

  The code generates some ordered random data and searches for a given random element. You can supply the number of items and the number of random elements to search for (one after another) on the command line (both or neither) or give them as parameters to main

  Try:

  Run the code with default values. How much faster should bisecting be than linear search on average? At best? And how much faster is it?

  Bisection needs the data to be sorted. How long does the sort take? How many look-ups are needed before it is worth it? How does this vary with the number of elements in the list? (For the exact answer we need to know which sort .sort() is using)