Eratosthenes' sieve

Eric Martin, CSE, UNSW

COMP9021 Principles of Programming

```
[1]: from math import sqrt
     from itertools import chain
     from timeit import timeit
```

Let n be a natural number. If a natural number m at most equal to n is not prime then m is of the form $p_1 \times \cdots \times p_k$ for some $k \geq 2$ and prime numbers p_1, ..., p_k with $p_1 \leq \cdots \leq p_k$; hence $n \geq m \geq p_1^2$, hence $p_1 \leq \sqrt{n}$. This implies that all natural numbers at most equal to n that are not prime have a proper factor at most equal to $\lfloor \sqrt{n} \rfloor$. So to identify all prime numbers up to and possibly including n, it suffices to cross out, from the collection of all numbers between 2 and n, all proper multiples at most equal to n of 2, 3, ... up to and including $\lfloor \sqrt{n} \rfloor$. Moreover, given a number p at most equal to $\lfloor \sqrt{n} \rfloor$, if all proper multiples at most equal to n of all numbers greater than 1 and smaller than p have been crossed out, then either p has been crossed out (together with all its multiples at most equal to n, case in which p is not prime), or only its proper multiples at least equal to p^2 and at most equal to n remain to be crossed out (case in which p is prime).

There is a risk that the computation of $\lfloor \sqrt{n} \rfloor$ yields a smaller number. The risk seems particularly high in case n is the perfect square of a prime: if the computation of $\lfloor \sqrt{n} \rfloor$ yielded a smaller number, then n would not be crossed out and be incorrectly part of the collection of integers eventually declared to be prime.

To appreciate the imprecision of floating point computation, let us witness computations of $(\sqrt{n})^2$ that are too small, correct (as a floating point number), or too large:

```
[2]: too_small = []
     just_right = []
     too_large = []

     n = 1
     while len(too_small) < 10 or len(just_right) < 10 or len(too_large) < 10:
         sqrt_n = sqrt(n)
         if sqrt_n ** 2 < n and len(too_small) < 10:
             too_small.append((n, sqrt_n, sqrt_n ** 2))
         elif sqrt_n ** 2 == n and len(just_right) < 10:
             just_right.append((n, sqrt_n, sqrt_n ** 2))
         elif sqrt_n ** 2 > n and len(too_large) < 10:
             too_large.append((n, sqrt_n, sqrt_n ** 2))
         n += 1

     print('Too small!')
```

```python
for triple in too_small:
    print(triple)
print('\nJust right!')
for triple in just_right:
    print(triple)
print('\nToo large!')
for triple in too_large:
    print(triple)
```

```
Too small!
(3, 1.7320508075688772, 2.9999999999999996)
(6, 2.449489742783178, 5.999999999999999)
(12, 3.4641016151377544, 11.999999999999998)
(13, 3.605551275463989, 12.999999999999998)
(18, 4.242640687119285, 17.999999999999996)
(23, 4.795831523312719, 22.999999999999996)
(24, 4.898979485566356, 23.999999999999996)
(26, 5.0990195135927845, 25.999999999999996)
(29, 5.385164807134504, 28.999999999999996)
(31, 5.5677643628300215, 30.999999999999996)

Just right!
(1, 1.0, 1.0)
(4, 2.0, 4.0)
(9, 3.0, 9.0)
(11, 3.3166247903554, 11.0)
(14, 3.7416573867739413, 14.0)
(16, 4.0, 16.0)
(17, 4.123105625617661, 17.0)
(21, 4.58257569495584, 21.0)
(22, 4.69041575982343, 22.0)
(25, 5.0, 25.0)

Too large!
(2, 1.4142135623730951, 2.0000000000000004)
(5, 2.23606797749979, 5.000000000000001)
(7, 2.6457513110645907, 7.000000000000001)
(8, 2.8284271247461903, 8.000000000000002)
(10, 3.1622776601683795, 10.000000000000002)
(15, 3.872983346207417, 15.000000000000002)
(19, 4.358898943540674, 19.000000000000004)
(20, 4.47213595499958, 20.000000000000004)
(28, 5.291502622129181, 28.000000000000004)
(32, 5.656854249492381, 32.00000000000001)
```

The square roots of the perfect squares that have been considered in the previous code fragment have all been computed correctly (as floating point numbers). Also observe that they have been squared correctly (as floating point numbers), but for large enough perfect squares, that does not

hold any more:

```
[3]: too_small = None
     too_large = None

     i = 1
     while too_small is None or too_large is None:
         i_square = i ** 2
         if sqrt(i_square) ** 2 < i_square:
             too_small = i, i_square, sqrt(i_square), sqrt(i_square) ** 2
         elif sqrt(i_square) ** 2 > i_square:
             too_large = i, i_square, sqrt(i_square), sqrt(i_square) ** 2
         i += 1

     print('Too small!')
     print(too_small)
     print('\nToo large!')
     print(too_large)
```

```
Too small!
(94906299, 9007205589877401, 94906299.0, 9007205589877400.0)

Too large!
(94906301, 9007205969502601, 94906301.0, 9007205969502602.0)
```

The previous code fragment leaves open the possibility that the computation of the square root of a perfect square is always correct (as a floating point number), and in particular, is never smaller than $\lfloor \sqrt{n} \rfloor$. It is also possible that when n is not a perfect square, then the computation of $\sqrt{n}$, though often incorrect, and in particular often smaller than $\sqrt{n}$, is still never smaller than $\lfloor \sqrt{n} \rfloor$. So whether n is a perfect square or not, changing the type of the computation of $\sqrt{n}$ from floating point to integer might result in a correct computation of $\lfloor \sqrt{n} \rfloor$. Still, to be on the safe side, it is preferable to use `round()` rather than `int()`.

Compare:

```
[4]: int(3.01), round(3.01)
     int(2.99), round(2.99)
```

```
[4]: (3, 3)
```

```
[4]: (2, 3)
```

A natural question in relation to `round()` is: for a given integer k, what is $k + 0.5$ rounded to? It turns out to be the one of k and $k + 1$ which is closest to 0:

```
[5]: round(2.5), round(-2.5)
```

```
[5]: (2, -2)
```

`round()` also lets us specify a precision:

3

```
[6]: round(1.9876543, 0)
     round(1.9876543, 1)
     round(1.9876543, 2)
     round(1.9876543, 3)
     round(1.9876543, 10)
```

[6]: 2.0

[6]: 2.0

[6]: 1.99

[6]: 1.988

[6]: 1.9876543

A list `sieve` of length $n + 1$ can be used to record whether i is prime for $2 \leq i \leq n$, storing `True` or `False` at index i depending on whether i is believed to be prime or not. The first two elements of `sieve`, of index 0 and 1, are unused. To start with, we assume that all numbers are prime.

For illustration purposes, let us fix n to some value, make it the value of a variable `n`, and define `sieve` accordingly:

```
[7]: n = 37
     sieve = [True] * (n + 1)
```

To nicely display `sieve`'s contents and indexes at various stages of the procedure, we know that we can make use of formatted strings and in particular, output decimal numbers within a particular field width, if necessary padding with spaces (the default) or with 0's; the decimal number and the field width can be the values of variables that both occur within a pair of curly braces within a formatted string:

```
[8]: x = 100; w = 5

     f'|{x:{w}}|'
     f'|{x:0{w}}|'
```

[8]: '| 100|'

[8]: '|00100|'

For now we fix the field width to 3 but below, to appropriately deal with a sieve of arbitrary size, we will compute the field width and make it a function of the largest prime to display.

```
[9]: def print_sieve_contents_and_indexes():
         for e in sieve:
             print('  T', end='') if e else print('  F', end='')
         print()
         for i in range(len(sieve)):
```

```
        print(f'{i:3}', end='')

print_sieve_contents_and_indexes()
```

```
  T  T  T  T  T  T  T  T  T  T  T  T  T  T  T  T  T  T  T  T  T  T  T  T  T  T
  T  T  T  T  T  T  T  T  T  T  T  T
  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
 26 27 28 29 30 31 32 33 34 35 36 37
```

To cross out all multiples at most equal to n of a prime number p, starting with p^2 if the multiples at most equal to n of all smaller primes have been crossed out already, we need to generate a sequence of the form p^2, $p^2 + p$, $p^2 + 2p$... This is easily done with `range()`:

[10]:
```
# One argument, the ending point, which is excluded.
# The starting point is 0, the default,
# The step is 1, the default
list(range(4))
# Two arguments, the starting point, which is included,
# and the ending point, which is excluded.
# The step is 1, the default
list(range(4, 10))
# Three arguments, the starting point, which is included,
# the ending point, which is excluded, and the step.
list(range(3, 11, 2))
list(range(3, 11, 3))
list(range(11, 3, -2))
list(range(11, 3, -3))
```

[10]: [0, 1, 2, 3]

[10]: [4, 5, 6, 7, 8, 9]

[10]: [3, 5, 7, 9]

[10]: [3, 6, 9]

[10]: [11, 9, 7, 5]

[10]: [11, 8, 5]

To observe how, with `n` set to 37, proper multiples of 2, 3 and 5 are crossed out while 4 and 6 are found out to be crossed out (together with their multiples) already, we successively call the following function with `p` set to 2, 3, 4, 5 and 6 (note that $6 = \lfloor\sqrt{37}\rfloor$) as argument:

[11]:
```
def cross_out_proper_multiples(p):
    # We assume that this function will be called in the order
    #   eliminate_proper_multiples(2)
    #   eliminate_proper_multiples(3)
```

```python
        #   eliminate_proper_multiples(4)
        #   ...
    if not sieve[p]:
        print(f'{p} has been crossed out '
              'as a multiple of a smaller number.'
              )
    else:
        print(f'{p} is not a multiple of a smaller number, '
              'hence it is prime.'
              )
        print('Now crossing out all proper multiples '
              f'of {p} at most equal to {n}.'
              )
        for i in range(p * p, n + 1, p):
            print(f' Crossing out {i}')
            sieve[i] = False
        print_sieve_contents_and_indexes()
```

[12]:
```python
cross_out_proper_multiples(2)
```

```
2 is not a multiple of a smaller number, hence it is prime.
Now crossing out all proper multiples of 2 at most equal to 37.
 Crossing out 4
 Crossing out 6
 Crossing out 8
 Crossing out 10
 Crossing out 12
 Crossing out 14
 Crossing out 16
 Crossing out 18
 Crossing out 20
 Crossing out 22
 Crossing out 24
 Crossing out 26
 Crossing out 28
 Crossing out 30
 Crossing out 32
 Crossing out 34
 Crossing out 36
  T  T  T  T  F  T  F  T  F  T  F  T  F  T  F  T  F  T  F  T  F  T  F  T  F  T
 F  T  F  T  F  T  F  T  F  T  F  T
  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
 26 27 28 29 30 31 32 33 34 35 36 37
```

[13]:
```python
cross_out_proper_multiples(3)
```

```
3 is not a multiple of a smaller number, hence it is prime.
```

```
Now crossing out all proper multiples of 3 at most equal to 37.
 Crossing out 9
 Crossing out 12
 Crossing out 15
 Crossing out 18
 Crossing out 21
 Crossing out 24
 Crossing out 27
 Crossing out 30
 Crossing out 33
 Crossing out 36
  T  T  T  T  F  T  F  T  F  F  F  F  T  F  T  F  F  F  T  F  T  F  F  F  T  F  T
F  F  F  T  F  T  F  F  F  T  F  T
  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
26 27 28 29 30 31 32 33 34 35 36 37
```

[14]: `cross_out_proper_multiples(4)`

```
4 has been crossed out as a multiple of a smaller number.
```

[15]: `cross_out_proper_multiples(5)`

```
5 is not a multiple of a smaller number, hence it is prime.
Now crossing out all proper multiples of 5 at most equal to 37.
 Crossing out 25
 Crossing out 30
 Crossing out 35
  T  T  T  T  F  T  F  T  F  F  F  F  T  F  T  F  F  F  T  F  T  F  F  F  T  F  F
F  F  F  T  F  T  F  F  F  F  F  T
  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
26 27 28 29 30 31 32 33 34 35 36 37
```

[16]: `cross_out_proper_multiples(6)`

```
6 has been crossed out as a multiple of a smaller number.
```

[17]:
```python
print(f'The prime numbers at most equal to {n} are:')
for p in range(2, n + 1):
    if sieve[p]:
        print(f'{p:4}', end='')
```

```
The prime numbers at most equal to 37 are:
   2   3   5   7  11  13  17  19  23  29  31  37
```

Putting it all together:

[18]:
```python
def sieve_of_primes_up_to(n):
    primes_sieve = [True] * (n + 1)
    for p in range(2, round(sqrt(n)) + 1):
```

```
            if primes_sieve[p]:
                for i in range(p * p, n + 1, p):
                    primes_sieve[i] = False
        return primes_sieve
```

To display all prime numbers at most equal to n, we define two functions. One function, `sequence_and_max_size_from()`, is designed to, from the list returned by `sieve_of_primes_up_to()`, determine and return the corresponding sequence of primes σ together with the number of digits l in the last (and largest) prime in the sequence; σ and l will be assigned to both arguments, `sequence` and `max_size`, respectively, of the other function, `nicely_display()`. We will utilise this function again when we implement other sieve methods. It is general enough to nicely display any sequence of data all of which are output with at most `max_size` many characters. More precisely, `nicely_display()` has the following features. It outputs at most 80 characters per line. Two spaces precede the display of the data that are output with `max_size` many characters. Three spaces precede the display of the data that are output with `max_size` minus 1 many characters, if any. Four spaces precede the display of the data that are output with `max_size` minus 2 many digits, if any... That way, all data will be nicely aligned column by column, with the guarantee that at least two spaces will separate two consecutive data on the same line. If l is the value of `max_size`, then exactly $\lfloor \frac{80}{l+2} \rfloor$ data will be displayed per line, with the possible exception of the last line:

```
[19]: def nicely_display(sequence, max_size):
          field_width = max_size + 2
          nb_of_fields = 80 // field_width
          count = 0
          for e in sequence:
              print(f'{e:{field_width}}', end='')
              count += 1
              if count % nb_of_fields == 0:
                  print()

      nicely_display(range(200), 3)
```

```
   0    1    2    3    4    5    6    7    8    9   10   11   12   13   14   15
  16   17   18   19   20   21   22   23   24   25   26   27   28   29   30   31
  32   33   34   35   36   37   38   39   40   41   42   43   44   45   46   47
  48   49   50   51   52   53   54   55   56   57   58   59   60   61   62   63
  64   65   66   67   68   69   70   71   72   73   74   75   76   77   78   79
  80   81   82   83   84   85   86   87   88   89   90   91   92   93   94   95
  96   97   98   99  100  101  102  103  104  105  106  107  108  109  110  111
 112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127
 128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143
 144  145  146  147  148  149  150  151  152  153  154  155  156  157  158  159
 160  161  162  163  164  165  166  167  168  169  170  171  172  173  174  175
 176  177  178  179  180  181  182  183  184  185  186  187  188  189  190  191
 192  193  194  195  196  197  198  199
```

To determine the value of `max_size` when using `nicely_display()` to display all prime numbers

up to a largest prime number p, we need to determine the number of digits in p, which is easily done by letting `str()` produce a string from an integer, and calling `len()` on the former:

```
[20]: str(991)
      len(str('991'))
```

```
[20]: '991'
```

```
[20]: 3
```

In `nicely_display()`, a `for` statement processes its first argument, `sequence`. So `sequence` has to be an iterable, and possibly an iterator. The `next()` method can be applied to an iterator. From an iterable that is not an iterator, one can get an iterator thanks to the `iter()` function. The `iter()` function can be applied to any iterable, so also to an iterator, in which case it just returns its argument:

```
[21]: # An iterable (an object of the range class) that is not an iterator
      x = range(2)
      x is iter(x)
      y = iter(x)
      next(y)
      next(y)

      # An iterable (a list) that is not an iterator
      x = [10, 11]
      x is iter(x)
      y = iter(x)
      next(y)
      next(y)

      # An iterable (a generator expression) that is an iterator
      x = (u for u in (100, 200))
      x is iter(x)
      next(x)
      next(x)
```

```
[21]: False
```

```
[21]: 0
```

```
[21]: 1
```

```
[21]: False
```

```
[21]: 10
```

```
[21]: 11
```

9

[21]: True

[21]: 100

[21]: 200

When a `for` statement processes an iterator, it calls `next()` again and again, until a `StopIteration` is generated, causing it to gracefully stop execution. When a `for` statement processes an iterable that is not an iterator, it first gets an iterator from the iterable thanks to `iter()`, iterator which is then processed as described. So the argument `sequence` of `nicely_display()` can be either an iterable that is not an iterator, like a list, or a tuple; or it can be an iterator, like a generator expression. The second option can lead to more effective code than the first one. Indeed, when a `for` statement processes a list or tuple, then that list or tuple had to be created in the first place, which the `for` statement then processes by implicit calls to `next()` on an iterator produced from that list or tuple by `iter()`. On the other hand, when a `for` statement processes a generator expression, then only a mechanism to produce a sequence had to be created in the first place, and that mechanism is activated (`next()` is implicitly called again and again) to generate all elements in the sequence and process them "on the fly":

```python
sieve = [True, True, True, True, False, True, False, True, False]
# A list created from sieve thanks to a list comprehension.
# sieve has been scanned from beginning to end to create primes.
primes = [i for i in range(2, len(sieve)) if sieve[i]]
primes
# An iterator is created from primes, to generate all members of primes
# and print them out.
# So eventually, two sequences will have been processed.
for e in primes:
    print(e, end=' ')

sieve = [True, True, True, True, False, True, False, True, False]
# A generator expression defined from sieve.
# sieve has not been scanned from beginning to end to create primes;
# primes is a mechanism to generate some numbers from sieve.
primes = (i for i in range(2, len(sieve)) if sieve[i])
primes
# The mechanism is activated as the for loop is executed.
# As an effect, sieve is scanned from beginning to end,
# numbers are generated and printed out on the fly.
# So eventually, only one sequence will have been processed.
for e in primes:
    print(e, end=' ')
```

[22]: [2, 3, 5, 7]

    2 3 5 7

[22]: <generator object <genexpr> at 0x10dab9970>

```
2 3 5 7
```

Based on these considerations, we define `sequence_and_max_size_from()` as follows:

```python
[23]: def sequence_and_max_size_from(sieve):
          largest_prime = len(sieve) - 1
          while not sieve[largest_prime]:
              largest_prime -= 1
          return (p for p in range(2, len(sieve)) if sieve[p]),\
                  len(str(largest_prime))
```

We now have everything we need to nicely display all prime numbers at most equal to n:

```python
[24]: nicely_display(*sequence_and_max_size_from(sieve_of_primes_up_to(1_000)))
```

```
  2   3   5   7  11  13  17  19  23  29  31  37  41  43  47  53
 59  61  67  71  73  79  83  89  97 101 103 107 109 113 127 131
137 139 149 151 157 163 167 173 179 181 191 193 197 199 211 223
227 229 233 239 241 251 257 263 269 271 277 281 283 293 307 311
313 317 331 337 347 349 353 359 367 373 379 383 389 397 401 409
419 421 431 433 439 443 449 457 461 463 467 479 487 491 499 503
509 521 523 541 547 557 563 569 571 577 587 593 599 601 607 613
617 619 631 641 643 647 653 659 661 673 677 683 691 701 709 719
727 733 739 743 751 757 761 769 773 787 797 809 811 821 823 827
829 839 853 857 859 863 877 881 883 887 907 911 919 929 937 941
947 953 967 971 977 983 991 997
```

To save half of the sieve's space and not have to cross out the proper multiples of 2, one can change `sieve` and make it a list of length $\lfloor \frac{n+1}{2} \rfloor$, with indexes 0, 1, 2, 3, 4, 5... meant to refer to the numbers 2, 3, 5, 7, 9... The price we pay for this is that we lose the simple equivalence between "number p is prime" and "`sieve` eventually stores `True` at index p": the equivalence becomes: "number p is prime" iff:

- $p = 2$ or p is odd, and
- `sieve` eventually stores `True` at index $\lfloor \frac{p-1}{2} \rfloor$).

Let p be a number between 3 and $\lfloor \sqrt{n} \rfloor$. The index that refers to p in this modified `sieve` is $k = \frac{p-1}{2}$, hence the index that refers to p^2 is $\frac{p^2-1}{2} = \frac{(p-1)(p+1)}{2} = \frac{p-1}{2} 2(\frac{p-1}{2} + 1) = 2k(k+1)$. Also, only the proper odd multiples at most equal to n of p have to be crossed out; so after having crossed out such a multiple a, the next multiple of p that needs to crossed out (in case it is still at most equal to n), is referred to at index $\frac{a+2p-1}{2} = \frac{a-1}{2} + p = \frac{a-1}{2} + 2k + 1$, so $2k + 1$ needs to be added to the index that refers to a to refer to that next multiple of p.

Putting it all together:

```python
[25]: def optimised_sieve_of_primes_up_to(n):
          n_index = (n - 1) // 2
          sieve = [True] * (n_index + 1)
          for k in range(1, (round(sqrt(n)) + 1) // 2):
              if sieve[k]:
```

11

```python
            for i in range(2 * k * (k + 1), n_index + 1, 2 * k + 1):
                sieve[i] = False
    return sieve
```

To display all prime numbers at most equal to n from the list returned by `optimised_sieve_of_primes_up_to()`, we need to adapt the function `sequence_and_max_size_from()`. Essentially, one has to generate all numbers of the form $2i + 1$ for all $1 \le i \le \lfloor \frac{n-1}{2} \rfloor$ such that the list `sieve` returned by `optimised_sieve_of_primes_up_to()` has a value of `True` at index i; such is the relationship between the odd prime numbers at most equal to n and the strictly positive indexes in `sieve`. But these odd prime numbers have to be preceded with 2. We still want to return an iterator. The simplest solution is to create an iterator from an iterator meant to generate 2 only, and the generator expression (`2 * p + 1 for p in range(1, len(sieve)) if sieve[p]`)). The `chain()` function from the `itertools` module lets us combine a sequence of iterables (some of which can be iterators) into one iterator:

```python
[26]: # Providing as argument to list() an iterator created from two iterators
      list(chain(iter(range(2)), (i for i in [10, 20, 30])))
      # Providing as argument to list() an iterator created from one iterator
      # and one iterable that is not an iterator
      list(chain(range(2), (i for i in [10, 20, 30])))
      # Providing as argument to list() an iterator created from two iterables
      # that are not iterators
      list(chain(range(2), [10, 20, 30]))
```

```
[26]: [0, 1, 10, 20, 30]
```

```
[26]: [0, 1, 10, 20, 30]
```

```
[26]: [0, 1, 10, 20, 30]
```

Based on these considerations, we nicely display all prime numbers identified by `optimised_sieve_of_primes_up_to()` as follows:

```python
[27]: def optimised_sequence_and_max_size_from(sieve):
          largest_prime = len(sieve) - 1
          while not sieve[largest_prime]:
              largest_prime -= 1
          return chain((2,), (2 * p + 1 for p in range(1, len(sieve)) if sieve[p])),\
              len(str(largest_prime))

      nicely_display(*optimised_sequence_and_max_size_from(
                                    optimised_sieve_of_primes_up_to(1_000)
                                    )
                    )
```

2	3	5	7	11	13	17	19	23	29	31	37	41	43	47	53
59	61	67	71	73	79	83	89	97	101	103	107	109	113	127	131
137	139	149	151	157	163	167	173	179	181	191	193	197	199	211	223

12

```
227   229   233   239   241   251   257   263   269   271   277   281   283   293   307   311
313   317   331   337   347   349   353   359   367   373   379   383   389   397   401   409
419   421   431   433   439   443   449   457   461   463   467   479   487   491   499   503
509   521   523   541   547   557   563   569   571   577   587   593   599   601   607   613
617   619   631   641   643   647   653   659   661   673   677   683   691   701   709   719
727   733   739   743   751   757   761   769   773   787   797   809   811   821   823   827
829   839   853   857   859   863   877   881   883   887   907   911   919   929   937   941
947   953   967   971   977   983   991   997
```

Let us get an idea of how large we can afford n to be and how more efficient `optimised_sieve_of_primes_up_to()` is compared to `sieve_of_primes_up_to()`. We ask the `timeit()` method from the `timeit` module to executing once (`number=1`) the code `sieve_of_primes_up_to(10_000_000)`, the assignment of the value returned by `globals()` to `globals` being needed to let `timetit()` know about the names `sieve_of_primes_up_to` and `optimised_sieve_of_primes_up_to`:

```
[28]: type(globals())
      'sieve_of_primes_up_to' in globals()
      'optimised_sieve_of_primes_up_to' in globals()
```

[28]: dict

[28]: True

[28]: True

```
[29]: timeit('sieve_of_primes_up_to(10_000_000)', globals=globals(), number=1)
      timeit('optimised_sieve_of_primes_up_to(10_000_000)', globals=globals(),
          number=1
          )
```

[29]: 1.4917003529999988

[29]: 0.6268495150000035