9 Profiling

Lab Objective: *Efficiency is essential to algorithmic programming. Profiling is the process of measuring the complexity and efficiency of a program, allowing the programmer to see what parts of the code need to be optimized. In this lab we present common techniques for speeding up Python code, including the built-in profiler and the Numba module.*

Magic Commands in IPython

IPython has tools for quickly timing and profiling code. These "magic commands" start with one or two `%` characters—one for testing a single line of code, and two for testing a block of code.

- `%time`: Execute some code and print out its execution time.
- `%timeit`: Execute some code several times and print out the average execution time.
- `%prun`: Run a statement through the Python code profiler,[1] printing the number of function calls and the time each takes. We will demonstrate this tool a little later.

```
# Time the construction of a list using list comprehension.
In [1]: %time x = [i**2 for i in range(int(1e5))]
CPU times: user 36.3 ms, sys: 3.28 ms, total: 39.6 ms
Wall time: 40.9 ms

# Time the same list construction, but with a regular for loop.
In [2]: %%time                          # Use a double %% to time a block of code.
   ...: x = []
   ...: for i in range(int(1e5)):
   ...:     x.append(i**2)
   ...:
CPU times: user 50 ms, sys: 2.79 ms, total: 52.8 ms
Wall time: 55.2 ms                      # The list comprehension is faster!
```

[1] `%prun` is a shortcut for `cProfile.run()`; see `https://docs.python.org/3/library/profile.html` for details.

Choosing Faster Algorithms

The best way to speed up a program is to use an efficient algorithm. A bad algorithm, even when implemented well, is never an adequate substitute for a good algorithm.

Problem 1. This problem comes from `https://projecteuler.net` (problems 18 and 67).

By starting at the top of the triangle below and moving to adjacent numbers on the row below, the maximum total from top to bottom is 23.

$$\begin{array}{c} \mathbf{3} \\ \mathbf{7}\ 4 \\ 2\ \mathbf{4}\ 6 \\ 8\ 5\ \mathbf{9}\ 3 \end{array}$$

That is, $3 + 7 + 4 + 9 = 23$.

The following function finds the maximum path sum of the triangle in `triangle.txt` by recursively computing the sum of every possible path—the "brute force" approach.

```python
def max_path(filename="triangle.txt"):
    """Find the maximum vertical path in a triangle of values."""
    with open(filename, 'r') as infile:
        data = [[int(n) for n in line.split()]
                        for line in infile.readlines()]
    def path_sum(r, c, total):
        """Recursively compute the max sum of the path starting in row r
        and column c, given the current total.
        """
        total += data[r][c]
        if r == len(data) - 1:      # Base case.
            return total
        else:                       # Recursive case.
            return max(path_sum(r+1, c,   total), # Next row, same column.
                       path_sum(r+1, c+1, total)) # Next row, next column.

    return path_sum(0, 0, 0)        # Start the recursion from the top.
```

The data in `triangle.txt` contains 15 rows and hence 16384 paths, so it is possible to solve this problem by trying every route. However, for a triangle with 100 rows, there are 2^{99} paths to check, which would take billions of years to compute even for a program that could check one trillion routes per second. No amount of improvement to `max_path()` can make it run in an acceptable amount of time on such a triangle—we need a different algorithm.

Write a function that accepts a filename containing a triangle of integers. Compute the largest path sum with the following strategy: starting from the next to last row of the triangle, replace each entry with the sum of the current entry and the greater of the two "child entries." Continue this replacement up through the entire triangle. The top entry in the triangle will be the maximum path sum. In other words, work from the bottom instead of from the top.

$$\begin{array}{c}3\\7\ 4\\2\ 4\ 6\\8\ 5\ 9\ 3\end{array} \longrightarrow \begin{array}{c}3\\7\ 4\\10\ 13\ 15\\8\ 5\ 9\ 3\end{array} \longrightarrow \begin{array}{c}3\\20\ 19\\10\ 13\ 15\\8\ 5\ 9\ 3\end{array} \longrightarrow \begin{array}{c}\mathbf{23}\\20\ 19\\10\ 13\ 15\\8\ 5\ 9\ 3\end{array}$$

Use your function to find the maximum path sum of the 100-row triangle stored in `triangle_large.txt`. Make sure that your new function still gets the correct answer for the smaller `triangle.txt`. Finally, use `%time` or `%timeit` to time both functions on `triangle.txt`. Your new function should be about 100 times faster than the original.

The Profiler

The profiling command `%prun` lists the functions that are called during the execution of a piece of code, along with the following information.

Heading	Description
`primitive calls`	The number of calls that were not caused by recursion.
`ncalls`	The number of calls to the function. If recursion occurs, the output is `<total number of calls>/<number of primitive calls>`.
`tottime`	The amount of time spent in the function, not including calls to other functions.
`percall`	The amount of time spent in each call of the function.
`cumtime`	The amount of time spent in the function, including calls to other functions.

```
# Profile the original function from Problem 1.
In [3]: %prun max_path("triangle.txt")
```

```
         81947 function calls (49181 primitive calls) in 0.036 seconds
   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
  32767/1    0.025    0.000    0.034    0.034 profiling.py:18(path_sum)
    16383    0.005    0.000    0.005    0.000 {built-in method builtins.max}
    32767    0.003    0.000    0.003    0.000 {built-in method builtins.len}
        1    0.002    0.002    0.002    0.002 {method 'readlines' of '_io._IOBase' objects}
        1    0.000    0.000    0.000    0.000 {built-in method io.open}
        1    0.000    0.000    0.036    0.036 profiling.py:12(max_path)
        1    0.000    0.000    0.000    0.000 profiling.py:15(<listcomp>)
        1    0.000    0.000    0.036    0.036 {built-in method builtins.exec}
        2    0.000    0.000    0.000    0.000 codecs.py:318(decode)
        1    0.000    0.000    0.036    0.036 <string>:1(<module>)
       15    0.000    0.000    0.000    0.000 {method 'split' of 'str' objects}
        1    0.000    0.000    0.000    0.000 _bootlocale.py:23(getpreferredencoding)
        2    0.000    0.000    0.000    0.000 {built-in method _codecs.utf_8_decode}
        1    0.000    0.000    0.000    0.000 {built-in method _locale.nl_langinfo}
        1    0.000    0.000    0.000    0.000 codecs.py:259(__init__)
        1    0.000    0.000    0.000    0.000 codecs.py:308(__init__)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

Optimizing Python Code

A poor implementation of a good algorithm is better than a good implementation of a bad algorithm, but clumsy implementation can still cripple a program's efficiency. The following are a few important practices for speeding up a Python program. Remember, however, that such improvements are futile if the algorithm is poorly suited for the problem.

Avoid Repetition

A clean program does no more work than is necessary. The `ncalls` column of the profiler output is especially useful for identifying parts of a program that might be repetitive. For example, the profile of `max_path()` indicates that `len()` was called 32,767 times—exactly as many times as `path_sum()`. This is an easy fix: save `len(data)` as a variable somewhere outside of `path_sum()`.

```
In [4]: def max_path_clean(filename="triangle.txt"):
   ...:     with open(filename, 'r') as infile:
   ...:         data = [[int(n) for n in line.split()]
   ...:                         for line in infile.readlines()]
   ...:     N = len(data)       # Calculate len(data) outside of path_sum().
   ...:     def path_sum(r, c, total):
   ...:         total += data[r][c]
   ...:         if r == N - 1:  # Use N instead of len(data).
   ...:             return total
   ...:         else:
   ...:             return max(path_sum(r+1, c,   total),
   ...:                        path_sum(r+1, c+1, total))
   ...:     return path_sum(0, 0, 0)
   ...:
In [5]: %prun max_path_clean("triangle.txt")
```

```
        49181 function calls (16415 primitive calls) in 0.026 seconds
  Ordered by: internal time

  ncalls  tottime  percall  cumtime  percall filename:lineno(function)
 32767/1    0.020    0.000    0.025    0.025 <ipython-input-5-9e8c48bb1aba>:6(path_sum)
   16383    0.005    0.000    0.005    0.000 {built-in method builtins.max}
       1    0.002    0.002    0.002    0.002 {method 'readlines' of '_io._IOBase' objects}
       1    0.000    0.000    0.000    0.000 {built-in method io.open}
       1    0.000    0.000    0.026    0.026 <ipython-input-5-9e8c48bb1aba>:1(max_path_clean)
       1    0.000    0.000    0.000    0.000 <ipython-input-5-9e8c48bb1aba>:3(<listcomp>)
       1    0.000    0.000    0.027    0.027 {built-in method builtins.exec}
      15    0.000    0.000    0.000    0.000 {method 'split' of 'str' objects}
       1    0.000    0.000    0.027    0.027 <string>:1(<module>)
       2    0.000    0.000    0.000    0.000 codecs.py:318(decode)
       1    0.000    0.000    0.000    0.000 _bootlocale.py:23(getpreferredencoding)
       2    0.000    0.000    0.000    0.000 {built-in method _codecs.utf_8_decode}
       1    0.000    0.000    0.000    0.000 {built-in method _locale.nl_langinfo}
       1    0.000    0.000    0.000    0.000 codecs.py:308(__init__)
       1    0.000    0.000    0.000    0.000 codecs.py:259(__init__)
       1    0.000    0.000    0.000    0.000 {built-in method builtins.len}
       1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

Note that the total number of primitive function calls decreased from 49,181 to 16,415. Using `%timeit` also shows that the run time decreased by about 15%. Moving code outside of a loop or an often-used function usually results in a similar speedup.

Another important way of reducing repetition is carefully controlling loop conditions to avoid unnecessary iterations. Consider the problem of identifying Pythagorean triples, sets of three distinct integers $a < b < c$ such that $a^2 + b^2 = c^2$. The following function identifies all such triples where each term is less than a parameter N by checking all possible triples.

```
>>> def pythagorean_triples_slow(N):
...     """Compute all pythagorean triples with entries less than N."""
...     triples = []
...     for a in range(1, N):                   # Try values of a from 1 to N-1.
...         for b in range(1, N):               # Try values of b from 1 to N-1.
...             for c in range(1, N):           # Try values of c from 1 to N-1.
...                 if a**2 + b**2 == c**2 and a < b < c:
...                     triples.append((a,b,c))
...     return triples
...
```

Since $a < b < c$ by definition, any computations where $b \leq a$ or $c \leq b$ are unnecessary. Additionally, once a and b are chosen, c can be no greater than $\sqrt{a^2 + b^2}$. The following function changes the loop conditions to avoid these cases and takes care to only compute $a^2 + b^2$ once for each unique pairing (a, b).

```
>>> from math import sqrt
>>> def pythagorean_triples_fast(N):
...     """Compute all pythagorean triples with entries less than N."""
...     triples = []
...     for a in range(1, N):                   # Try values of a from 1 to N-1.
...         for b in range(a+1, N):             # Try values of b from a+1 to N-1.
...             _sum = a**2 + b**2
...             for c in range(b+1, min(int(sqrt(_sum))+1, N)):
...                 if _sum == c**2:
...                     triples.append((a,b,c))
...     return triples
...
```

These improvements have a drastic impact on run time, even though the main approach—checking by brute force—is the same.

```
In [6]: %time triples = pythagorean_triples_slow(500)
CPU times: user 1min 51s, sys: 389 ms, total: 1min 51s
Wall time: 1min 52s          # 112 seconds.

In [7]: %time triples = pythagorean_triples_fast(500)
CPU times: user 1.56 s, sys: 5.38 ms, total: 1.57 s
Wall time: 1.57 s            # 98.6% faster!
```

Problem 2. The following function computes the first N prime numbers.

```
def primes(N):
    """Compute the first N primes."""
    primes_list = []
    current = 2
    while len(primes_list) < N:
        isprime = True
        for i in range(2, current):     # Check for nontrivial divisors.
            if current % i == 0:
                isprime = False
        if isprime:
            primes_list.append(current)
        current += 1
    return primes_list
```

This function takes about 6 minutes to find the first 10,000 primes on a fast computer.

Without significantly modifying the approach, rewrite `primes()` so that it can compute the first 10,000 primes in under 0.1 seconds. Use the following facts to reduce unnecessary iterations.

- A number is not prime if it has one or more divisors other than 1 and itself. (Hint: recall the `break` statement.)
- If $p \nmid n$, then $ap \nmid n$ for any integer a. Also, if $p \mid n$ and $0 < p < n$, then $p \leq \sqrt{n}$.
- Except for 2, primes are always odd.

Your new function should be helpful for solving problem 7 on `https://projecteuler.net`.

Avoid Loops

NumPy routines and built-in functions are often useful for eliminating loops altogether. Consider the simple problem of summing the rows of a matrix, implemented in three ways.

```
>>> def row_sum_awful(A):
...     """Sum the rows of A by iterating through rows and columns."""
...     m,n = A.shape
...     row_totals = np.empty(m)        # Allocate space for the output.
...     for i in range(m):              # For each row...
...         total = 0
...         for j in range(n):          # ...iterate through the columns.
...             total += A[i,j]
...         row_totals[i] = total       # Record the total.
...     return row_totals
...
>>> def row_sum_bad(A):
...     """Sum the rows of A by iterating through rows."""
...     return np.array([sum(A[i,:]) for i in range(A.shape[0])])
...
>>> def row_sum_fast(A):
...     """Sum the rows of A with NumPy."""
...     return np.sum(A, axis=1)    # Or A.sum(axis=1).
...
```

None of the functions are fundamentally different, but their run times differ dramatically.

```
In [8]: import numpy as np
In [9]: A = np.random.random((10000, 10000))

In [10]: %time rows = row_sum_awful(A)
CPU times: user 22.7 s, sys: 137 ms, total: 22.8 s
Wall time: 23.2 s          # SLOW!

In [11]: %time rows = row_sum_bad(A)
CPU times: user 8.85 s, sys: 15.6 ms, total: 8.87 s
Wall time: 8.89 s          # Slow!

In [12]: %time rows = row_sum_fast(A)
CPU times: user 61.2 ms, sys: 1.3 ms, total: 62.5 ms
Wall time: 64 ms           # Fast!
```

In this experiment, `row_sum_fast()` runs several hundred times faster than `row_sum_awful()`. This is primarily because looping is expensive in Python, but NumPy handles loops in C, which is much quicker. Other NumPy functions like `np.sum()` with an `axis` argument can often be used to eliminate loops in a similar way.

Problem 3. Let A be an $m \times n$ matrix with columns $\mathbf{a}_0, \ldots, \mathbf{a}_{n-1}$, and let $\mathbf{x}$ be a vector of length m. The *nearest neighbor problem*[a] is to determine which of the columns of A is "closest" to $\mathbf{x}$ with respect to some norm. That is, we compute

$$\underset{j}{\operatorname{argmin}} \, \|\mathbf{a}_j - \mathbf{x}\|.$$

The following function solves this problem naïvely for the usual Euclidean norm.

```
def nearest_column(A, x):
    """Find the index of the column of A that is closest to x."""
    distances = []
    for j in range(A.shape[1]):
        distances.append(np.linalg.norm(A[:,j] - x))
    return np.argmin(distances)
```

Write a new version of this function without any loops or list comprehensions, using array broadcasting and the `axis` keyword in `np.linalg.norm()` to eliminate the existing loop. Try to implement the entire function in a single line.
(Hint: See the NumPy Visual Guide in the Appendix for a refresher on array broadcasting.)

Profile the old and new versions with `%prun` and compare the output. Finally, use `%time` or `%timeit` to verify that your new version runs faster than the original.

[a]The nearest neighbor problem is a common problem in many fields of artificial intelligence. The problem can be solved more efficiently with a k-d tree, a specialized data structure for storing high-dimensional data.

Use Data Structures Correctly

Every data structure has strengths and weaknesses, and choosing the wrong data structure can be costly. Here we consider three ways to avoid problems and use sets, dictionaries, and lists correctly.

- **Membership testing**. The question "is `<value>` a member of `<container>`" is common in numerical algorithms. Sets and dictionaries are implemented in a way that makes this a trivial problem, but lists are not. In other words, the `in` operator is near instantaneous with sets and dictionaries, but not with lists.

```
In [13]: a_list = list(range(int(1e7)))

In [14]: a_set = set(a_list)

In [15]: %timeit 12.5 in a_list
413 ms +- 48.2 ms per loop (mean+-std.dev. of 7 runs, 1 loop each)

In [16]: %timeit 12.5 in a_set
170 ns +- 3.8 ns per loop (mean+-std.dev. of 7 runs, 10000000 loops each)
```

  Looking up dictionary values is also almost immediate. Use dictionaries for storing calculations to be reused, such as mappings between letters and numbers or common function outputs.

- **Construction with comprehension**. Lists, sets, and dictionaries can all be constructed with comprehension syntax. This is slightly faster than building the collection in a loop, and the code is highly readable.

```
# Map the integers to their squares.
In [17]: %%time
    ...: a_dict = {}
    ...: for i in range(1000000):
    ...:     a_dict[i] = i**2
    ...:
CPU times: user 432 ms, sys: 54.4 ms, total: 486 ms
Wall time: 491 ms

In [18]: %time a_dict = {i:i**2 for i in range(1000000)}
CPU times: user 377 ms, sys: 58.9 ms, total: 436 ms
Wall time: 440 ms
```

- **Intelligent iteration**. Unlike looking up dictionary values, indexing into lists takes time. Instead of looping over the indices of a list, loop over the entries themselves. When indices and entries are both needed, use `enumerate()` to get the index and the item simultaneously.

```
In [19]: a_list = list(range(1000000))

In [20]: %%time          # Loop over the indices of the list.
    ...: for i in range(len(a_list)):
    ...:     item = a_list[i]
    ...:
CPU times: user 103 ms, sys: 1.78 ms, total: 105 ms
Wall time: 107 ms

In [21]: %%time          # Loop over the items in the list.
    ...: for item in a_list:
    ...:     _ = item
    ...:
CPU times: user 61.2 ms, sys: 1.31 ms, total: 62.5 ms
Wall time: 62.5 ms      # Almost twice as fast as indexing!
```

Problem 4. This is problem 22 from `https://projecteuler.net`.

Using the rule $A \mapsto 1, B \mapsto 2, \ldots, Z \mapsto 26$, the *alphabetical value* of a name is the sum of the digits that correspond to the letters in the name. For example, the alphabetic value of "COLIN" is $3 + 15 + 12 + 9 + 14 = 53$.

The following function reads the file `names.txt`, containing over five-thousand first names, and sorts them in alphabetical order. The *name score* of each name in the resulting list is the alphabetic value of the name multiplied by the name's position in the list, starting at 1. "COLIN" is the 938th name alphabetically, so its name score is $938 \times 53 = 49714$. The function returns the total of all the name scores in the file.

```
def name_scores(filename="names.txt"):
    """Find the total of the name scores in the given file."""
    with open(filename, 'r') as infile:
        names = sorted(infile.read().replace('"', '').split(','))
    total = 0
    for i in range(len(names)):
        name_value = 0
        for j in range(len(names[i])):
            alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
            for k in range(len(alphabet)):
                if names[i][j] == alphabet[k]:
                    letter_value = k + 1
            name_value += letter_value
        total += (names.index(names[i]) + 1) * name_value
    return total
```

Rewrite this function—removing repetition, eliminating loops, and using data structures correctly—so that it runs in less than 10 milliseconds on average.

Use Generators

A *generator* is an iterator that yields multiple values, one at a time, as opposed to returning a single value. For example, `range()` is a generator. Using generators appropriately can reduce both the run time and the spatial complexity of a routine. Consider the following function, which constructs a list containing the entries of the sequence $\{x_n\}_{n=1}^N$ where $x_n = x_{n-1} + n$ with $x_1 = 1$.

```
>>> def sequence_function(N):
...     """Return the first N entries of the sequence x_n = x_{n-1} + n."""
...     sequence = []
...     x = 0
...     for n in range(1, N+1):
...         x += n
...         sequence.append(x)
...     return sequence
...
>>> sequence_function(10)
[1, 3, 6, 10, 15, 21, 28, 36, 45, 55]
```

A potential problem with this function is that all of the values in the list are computed before anything is returned. This can be a big issue if the parameter N is large. A generator, on the other hand, *yields* one value at a time, indicated by the keyword `yield` (instead of `return`). When the generator is asked for the next entry, the code resumes right where it left off.

```
>>> def sequence_generator(N):
...     """Yield the first N entries of the sequence x_n = x_{n-1} + n."""
...     x = 0
...     for n in range(1, N+1):
...         x += n
...         yield x         # "return" a single value.
...
# Get the entries of the generator one at a time with next().
>>> generated = sequence_generator(10)
>>> next(generated)
1
>>> next(generated)
3
>>> next(generated)
6

# Put each of the generated items in a list, as in sequence_function().
>>> list(sequence_generator(10))    # Or [i for i in sequence_generator(10)].
[1, 3, 6, 10, 15, 21, 28, 36, 45, 55]

# Use the generator in a for loop, like range().
```

```
>>> for entry in sequence_generator(10):
...     print(entry, end=' ')
...
1 3 6 10 15 21 28 36 45 55
```

Many generators, like `range()` and `sequence_generator()`, only yield a finite number of values. However, generators can also continue yielding indefinitely. For example, the following generator yields the terms of $\{x_n\}_{n=1}^{\infty}$ forever. In this case, using `enumerate()` with the generator is helpful for tracking the index n as well as the entry x_n.

```
>>> def sequence_generator_forever():
...     """Yield the sequence x_n = x_{n-1} + n forever."""
...     x = 0
...     n = 1
...     while True:
...         x += n
...         n += 1
...         yield x         # "return" a single value.
...

# Sum the entries of the sequence until the sum exceeds 1000.
>>> total = 0
>>> for i, x in enumerate(sequence_generator_forever()):
...     total += x
...     if total > 1000:
...         print(i)        # Print the index where the total exceeds.
...         break           # Break out of the for loop to stop iterating.
...
17

# Check that 18 terms are required (since i starts at 0 but n starts at 1).
>>> print(sum(sequence_generator(17)), sum(sequence_generator(18)))
969 1140
```

Problem 5. This is problem 25 from `https://projecteuler.net`.

The *Fibonacci sequence* is defined by the recurrence relation $F_n = F_{n-1} + F_{n-2}$, where $F_1 = F_2 = 1$. The 12th term, $F_{12} = 144$, is the first term to contain three digits.

Write a generator that yields the terms of the Fibonacci sequence indefinitely. Next, write a function that accepts an integer N. Use your generator to find the first term in the Fibonacci sequence that contains N digits. Return the index of this term.
(Hint: a generator can have more than one `yield` statement.)

Problem 6. The function in Problem 2 could be turned into a prime number generator that yields primes indefinitely, but it is not the only strategy for yielding primes. The *Sieve of Eratosthenes*[a] is a faster technique for finding all of the primes below a certain number.

1. Given a cap N, start with all of the integers from 2 to N.
2. Remove all integers that are divisible by the first entry in the list.
3. Yield the first entry in the list and remove it from the list.
4. Return to step 2 until the list is empty.

Write a generator that accepts an integer N and that yields all primes (in order, one at a time) that are less than N using the Sieve of Eratosthenes. Your generator should be able to find all primes less than 100,000 in under 5 seconds.

Your generator and your fast function from Problem 2 may be helpful in solving problems 10, 35, 37, 41, 49, and 50 (for starters) of `https://projecteuler.net`.

[a]See `https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes`.

Numba

Python code is simpler and more readable than many languages, but Python is also generally much slower than compiled languages like C. The `numba` module bridges the gap by using *just-in-time* (JIT) compilation to optimize code, meaning that the code is actually compiled right before execution.

```
>>> from numba import jit

>>> @jit                # Decorate a function with @jit to use Numba.
... def row_sum_numba(A):
...     """Sum the rows of A by iterating through rows and columns,
...     optimized by Numba.
...     """
...     m,n = A.shape
...     row_totals = np.empty(m)
...     for i in range(m):
...         total = 0
...         for j in range(n):
...             total += A[i,j]
...         row_totals[i] = total
...     return row_totals
```

Python is a *dynamically typed* language, meaning variables are not defined explicitly with a datatype (`x = 6` as opposed to `int x = 6`). This particular aspect of Python makes it flexible, easy to use, and slow. Numba speeds up Python code primarily by assigning datatypes to all the variables. Rather than requiring explicit definitions for datatypes, Numba attempts to infer the correct datatypes based on the datatypes of the input. In `row_sum_numba()`, if `A` is an array of integers, Numba will infer that `total` should also be an integer. On the other hand, if `A` is an array of floats, Numba will infer that `total` should be a *double* (a similar datatype to float in C).

Once all datatypes have been inferred and assigned, the original Python code is translated to machine code. Numba caches this compiled version of code for later use. The first function call takes the time to compile and then execute the code, but subsequent calls use the already-compiled code.

```
In [22]: A = np.random.random((10000, 10000))

# The first function call takes a little extra time to compile first.
In [23]: %time rows = row_sum_numba(A)
CPU times: user 408 ms, sys: 11.5 ms, total: 420 ms
Wall time: 425 ms

# Subsequent calls are consistently faster that the first call.
In [24]: %timeit row_sum_numba(A)
138 ms +- 1.96 ms per loop (mean +- std. dev. of 7 runs, 10 loops each)
```

Note that the only difference between `row_sum_numba()` and `row_sum_awful()` from a few pages ago is the `@jit` decorator, and yet the Numba version is about 99% faster than the original!

The inference engine within Numba does a good job, but it's not always perfect. Adding the keyword argument `nopython=True` to the `@jit` decorator raises an error if Numba is unable to convert each variable to explicit datatypes. The `inspect_types()` method can also be used to check if Numba is using the desired types.

```
# Run the function once first so that it compiles.
>>> rows = row_sum_numba(np.random.random((10,10)))
>>> row_sum_numba.inspect_types()
# The output is very long and detailed.
```

Alternatively, datatypes can be specified explicitly in the `@jit` decorator as a dictionary via the `locals` keyword argument. Each of the desired datatypes must also be imported from Numba.

```
>>> from numba import int64, double

>>> @jit(nopython=True, locals=dict(A=double[:,:], m=int64, n=int64,
...                                 row_totals=double[:], total=double))
... def row_sum_numba(A):           # 'A' is a 2-D array of doubles.
...     m,n = A.shape               # 'm' and 'n' are both integers.
...     row_totals = np.empty(m)    # 'row_totals' is a 1-D array of doubles.
...     for i in range(m):
...         total = 0               # 'total' is a double.
...         for j in range(n):
...             total += A[i,j]
...         row_totals[i] = total
...     return row_totals
...
```

While it sometimes results in a speed boost, there is a caveat to specifying the datatypes: `row_sum_numba()` no longer accepts arrays that contain anything other than floats. When datatypes are not specified, Numba compiles a new version of the function each time the function is called with a different kind of input. Each compiled version is saved, so the function can still be used flexibly.

Problem 7. The following function calculates the nth power of an $m \times m$ matrix A.

```python
def matrix_power(A, n):
    """Compute A^n, the n-th power of the matrix A."""
    product = A.copy()
    temporary_array = np.empty_like(A[0])
    m = A.shape[0]
    for power in range(1, n):
        for i in range(m):
            for j in range(m):
                total = 0
                for k in range(m):
                    total += product[i,k] * A[k,j]
                temporary_array[j] = total
            product[i] = temporary_array
    return product
```

1. Write a Numba-enhanced version of `matrix_power()` called `matrix_power_numba()`.
2. Write a function that accepts an integer n. Run `matrix_power_numba()` once with a small random input so it compiles. Then, for $m = 2^2, 2^3, \ldots, 2^7$,
   (a) Generate a random $m \times m$ matrix A with `np.random.random()`.
   (b) Time (separately) `matrix_power()`, `matrix_power_numba()`, and NumPy's `np.linalg.matrix_power()` on A with the specified value of n.
   (If you are unfamiliar with timing code inside of a function, see the Additional Material section on timing code.)

   Plot the times against the size m on a log-log plot with a base 2 scale (use `plt.loglog()`).

With $n = 10$, the plot should show that the Numba and NumPy versions far outperform the pure Python implementation, with NumPy eventually becoming faster than Numba.

Achtung!

Optimizing code is an important skill, but it is also important to know when to refrain from optimization. The best approach to coding is to write unit tests, implement a solution that works, test and time that solution, **then** (and only then) optimize the solution with profiling techniques. As always, the most important part of the process is choosing the correct algorithm to solve the problem. Don't waste time optimizing a poor algorithm.

Additional Material

Other Timing Techniques

Though %time and %timeit are convenient and work well, some problems require more control for measuring execution time. The usual way of timing a code snippet by hand is via the time module (which %time uses). The function time.time() returns the number of seconds since the Epoch[2]; to time code, measure the number of seconds before the code runs, the number of seconds after the code runs, and take the difference.

```
>>> import time

>>> start = time.time()             # Record the current time.
>>> for i in range(int(1e8)):       # Execute some code.
...     pass
... end = time.time()               # Record the time again.
... print(end - start)              # Take the difference.
...
4.20402193069458 # (seconds)
```

The timeit module (which %timeit uses) has tools for running code snippets several times. The code is passed in as a string, as well as any setup code to be run before starting the clock.

```
>>> import timeit

>>> timeit.timeit("for i in range(N): pass", setup="N = int(1e6)", number=200)
4.884839255013503       # Total time in seconds to run the code 200 times.
>>> _ / 200
0.024424196275067516    # Average time in seconds.
```

The primary advantages of these techniques are the ability automate timing code and being able save the results. For more documentation, see https://docs.python.org/3.6/library/time.html and https://docs.python.org/3.6/library/timeit.html.

Customizing the Profiler

The output from %prun is generally long, but it can be customized with the following options.

Option	Description
-l <limit>	Include a limited number of lines in the output.
-s <key>	Sort the output by call count, cumulative time, function name, etc.
-T <filename>	Save profile results to a file (results are still printed).

For example, %prun -l 3 -s ncalls -T path_profile.txt max_path() generates a profile of max_path() that lists the 3 functions with the most calls, then write the results to path_profile.txt. See http://ipython.readthedocs.io/en/stable/interactive/magics.html#magic-prun for more details.

[2]See https://en.wikipedia.org/wiki/Epoch_(reference_date)#Computing.