Performance Python

7 Strategies for Optimizing Your Numerical Code

Jake VanderPlas @jakevdp
PyCon 2018
Python is Fast.

Dynamic, interpreted, & flexible: fast Development

# Hello World in Python
print("hello world")

/* Hello World in Java */
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World");
    }
}

Python is Slow.

CPython has constant overhead per operation

```python
In [3]: # A silly function implemented in Python
def func_python(N):
    d = 0.0
    for i in range(N):
        d += (i % 3 - 1) * i
    return d
```

```python
In [4]: # Use IPython timeit magic to time the execution
%timeit func_python(10000)
```

1000 loops, best of 3: 1.69 ms per loop
Python is Slow.

CPython has constant overhead per operation

```python
In [5]: %load_ext fortranmagic

In [6]: %fortran
subroutine func_fort(n, d)
   integer, intent(in) :: n
double precision, intent(out) :: d
integer :: i
d = 0
do i = 0, n - 1
   d = d + (mod(i, 3) - 1) * i
end do
end subroutine func_fort

In [10]: %timeit func_fort(10000)

100000 loops, best of 3: 17.9 µs per loop
```
Python is Slow.

CPython has constant overhead per operation

```
In [4]: # Use IPython timeit magic to time the execution
   : %timeit func_python(10000)
   :
1000 loops, best of 3: 1.69 ms per loop

In [10]: %timeit func_fort(10000)
   :
1000000 loops, best of 3: 17.9 µs per loop
```

Fortran is 100x faster for this simple task!
The best of both worlds?
Seven Strategies
For Optimizing Your
Numerical Python Code
Example:
K-means Clustering
Example: K-means Clustering

Algorithm:
1. Choose some Cluster Centers
2. Repeat:
   a. Assign points to nearest center
   b. Update center to mean of points
   c. Check if Converged
Example: K-means Clustering

Algorithm:
1. Choose some Cluster Centers
2. Repeat:
   a. Assign points to nearest center
   b. Update center to mean of points
   c. Check if Converged
Example: K-means Clustering

Algorithm:
1. Choose some Cluster Centers
2. Repeat:
   a. Assign points to nearest center
   b. Update center to mean of points
   c. Check if Converged
**Example:**
K-means Clustering

**Algorithm:**
1. Choose some Cluster Centers
2. Repeat:
   a. Assign points to nearest center
   b. Update center to mean of points
   c. Check if Converged
Example: K-means Clustering

Algorithm:
1. Choose some Cluster Centers
2. Repeat:
   a. Assign points to nearest center
   b. Update center to mean of points
   c. Check if Converged
Example: K-means Clustering

Algorithm:
1. Choose some Cluster Centers
2. Repeat:
   a. Assign points to nearest center
   b. Update center to mean of points
   c. Check if Converged
Example: K-means Clustering

Algorithm:
1. Choose some Cluster Centers
2. Repeat:
   a. Assign points to nearest center
   b. Update center to mean of points
   c. Check if Converged
Example: K-means Clustering

Algorithm:
1. Choose some Cluster Centers
2. Repeat:
   a. Assign points to nearest center
   b. Update center to mean of points
   c. Check if Converged
Example: K-means Clustering

Algorithm:
1. Choose some Cluster Centers
2. Repeat:
   a. Assign points to nearest center
   b. Update center to mean of points
   c. Check if Converged
**Example: K-means Clustering**

**Algorithm:**
1. Choose some Cluster Centers
2. Repeat:
   a. Assign points to nearest center
   b. Update center to mean of points
   c. Check if Converged
Example:
K-means Clustering

Algorithm:
1. Choose some Cluster Centers
2. Repeat:
   a. Assign points to nearest center
   b. Update center to mean of points
   c. Check if Converged
Example: K-means Clustering

Algorithm:
1. Choose some Cluster Centers
2. Repeat:
   a. Assign points to nearest center
   b. Update center to mean of points
   c. Check if Converged
Example: K-means Clustering

Algorithm:
1. Choose some Cluster Centers
2. Repeat:
   a. Assign points to nearest center
   b. Update center to mean of points
   c. Check if Converged
Example: K-means Clustering

Algorithm:
1. Choose some Cluster Centers
2. Repeat:
   a. Assign points to nearest center
   b. Update center to mean of points
   c. Check if Converged
Example:
K-means Clustering

Algorithm:
1. Choose some Cluster Centers
2. Repeat:
   a. Assign points to nearest center
   b. Update center to mean of points
   c. Check if Converged
Example: K-means Clustering

Algorithm:
1. Choose some Cluster Centers
2. Repeat:
   a. Assign points to nearest center
   b. Update center to mean of points
   c. Check if Converged
Example: K-means Clustering

Algorithm:
1. Choose some Cluster Centers
2. Repeat:
   a. Assign points to nearest center
   b. Update center to mean of points
   c. Check if Converged
Example: K-means Clustering

Algorithm:
1. Choose some Cluster Centers
2. Repeat:
   a. Assign points to nearest center
   b. Update center to mean of points
   c. Check if Converged
Example: K-means Clustering

Algorithm:
1. Choose some Cluster Centers
2. Repeat:
   a. Assign points to nearest center
   b. Update center to mean of points
   c. Check if converged
Example: K-means Clustering

Algorithm:
1. Choose some Cluster Centers
2. Repeat:
   a. Assign points to nearest center
   b. Update center to mean of points
   c. Check if Converged
Example: K-means Clustering

Algorithm:
1. Choose some Cluster Centers
2. Repeat:
   a. Assign points to nearest center
   b. Update center to mean of points
   c. Check if Converged ✓
Implementing K Means in Python
def dist(x, y):
    return sum((xi - yi) ** 2
                for xi, yi in zip(x, y))

def find_labels(points, centers):
    labels = []
    for point in points:
        distances = [dist(point, center)
                     for center in centers]
        labels.append(distances.index(min(distances)))
    return labels
```python
def dist(x, y):
    return sum((xi - yi) ** 2 for xi, yi in zip(x, y))

def find_labels(points, centers):
    labels = []
    for point in points:
        distances = [dist(point, center) for center in centers]
        labels.append(distances.index(min(distances)))
    return labels
```
Python Implementation

```python
def dist(x, y):
    return sum((xi - yi) ** 2
                 for xi, yi in zip(x, y))

def find_labels(points, centers):
    labels = []
    for point in points:
        distances = [dist(point, center)
                     for center in centers]
        labels.append(distances.index(min(distances)))
    return labels
```
Python Implementation

```python
def dist(x, y):
    return sum((xi - yi) ** 2
        for xi, yi in zip(x, y))

def find_labels(points, centers):
    labels = []
    for point in points:
        distances = [dist(point, center)
            for center in centers]
        labels.append(distances.index(min(distances)))
    return labels
```
def compute_centers(points, labels):
    n_centers = len(set(labels))
    n_dims = len(points[0])

    centers = [[0 for i in range(n_dims)] for j in range(n_centers)]
    counts = [0 for j in range(n_centers)]

    for label, point in zip(labels, points):
        counts[label] += 1
        centers[label] = [a + b for a, b in zip(centers[label], point)]

    return [[x / count for x in center] for center, count in zip(centers, counts)]
```python
def compute_centers(points, labels):
    n_centers = len(set(labels))
    n_dims = len(points[0])

    centers = [[0 for i in range(n_dims)]
                for j in range(n_centers)]
    counts = [0 for j in range(n_centers)]

    for label, point in zip(labels, points):
        counts[label] += 1
        centers[label] = [a + b for a, b in zip(centers[label], point)]

    return [[[x / count for x in center]
              for center, count in zip(centers, counts)]
          for x in range(4)]
```
def compute_centers(points, labels):
    n_centers = len(set(labels))
    n_dims = len(points[0])

    centers = [[0 for i in range(n_dims)]
                for j in range(n_centers)]
    counts = [0 for j in range(n_centers)]

    for label, point in zip(labels, points):
        counts[label] += 1
        centers[label] = [a + b for a, b in
                          zip(centers[label], point)]

    return=[[x / count for x in center]
            for center, count in zip(centers, counts)]
```python
def compute_centers(points, labels):
    n_centers = len(set(labels))
    n_dims = len(points[0])

    centers = [[0 for i in range(n_dims)]
               for j in range(n_centers)]
    counts = [0 for j in range(n_centers)]

    for label, point in zip(labels, points):
        counts[label] += 1
        centers[label] = [a + b for a, b in
                          zip(centers[label], point)]

    return [[x / count for x in center]
             for center, count in zip(centers, counts)]
```
```python
def compute_centers(points, labels):
    n_centers = len(set(labels))
    n_dims = len(points[0])

    centers = [[0 for i in range(n_dims)]
                for j in range(n_centers)]
    counts = [0 for j in range(n_centers)]

    for label, point in zip(labels, points):
        counts[label] += 1
        centers[label] = [a + b for a, b in zip(centers[label], point)]

    return [[x / count for x in center]
            for center, count in zip(centers, counts)]
```
def kmeans(points, n_clusters):
    centers = points[-n_clusters:].tolist()
    while True:
        old_centers = centers
        labels = find_labels(points, centers)
        centers = compute_centers(points, labels)
        if centers == old_centers:
            break
    return labels

timeit kmeans(points, 10)

7.44 s ± 122 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```python
def kmeans(points, n_clusters):
    centers = points[-n_clusters:].tolist()

    while True:
        old_centers = centers
        labels = find_labels(points, centers)
        centers = compute_centers(points, labels)
        if centers == old_centers:
            break
    return labels

%timeit kmeans(points, 10)
```

7.44 s ± 122 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
Python Implementation

```python
def kmeans(points, n_clusters):
    centers = points[-n_clusters:].tolist()
    while True:
        old_centers = centers
        labels = find_labels(points, centers)
        centers = compute_centers(points, labels)
        if centers == old_centers:
            break
    return labels

%timeit kmeans(points, 10)
```

7.44 s ± 122 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
def kmeans(points, n_clusters):
    centers = points[-n_clusters:].tolist()
    while True:
        old_centers = centers
        labels = find_labels(points, centers)
        centers = compute_centers(points, labels)
        if centers == old_centers:
            break
    return labels

%timeit kmeans(points, 10)

7.44 s ± 122 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
Python Implementation

```python
def kmeans(points, n_clusters):
    centers = points[-n_clusters:].tolist()
    while True:
        old_centers = centers
        labels = find_labels(points, centers)
        centers = compute_centers(points, labels)
        if centers == old_centers:
            break
    return labels

timeit kmeans(points, 10)
```

7.44 s ± 122 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
Python Implementation

```python
def kmeans(points, n_clusters):
    centers = points[-nClusters:].tolist()
    while True:
        old_centers = centers
        labels = find_labels(points, centers)
        centers = compute_centers(points, labels)
        if centers == old_centers:
            break
    return labels
```

```
%timeit kmeans(points, 10)
```

```
7.44 s ± 122 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```
What to do when Python is too slow?
Seven Strategies:

1. Line Profiling
Seven Strategies:

1. Line Profiling

“Premature optimization is the root of all evil”
- Donald Knuth
%load_ext line_profiler
%lprun -f kmeans kmeans(points, 10)

Timer unit: 1e-06 s

Total time: 11.8153 s
File: <ipython-input-1-0104920df0d9>
Function: kmeans at line 27

<table>
<thead>
<tr>
<th>Line #</th>
<th>Hits</th>
<th>Time</th>
<th>Per Hit</th>
<th>% Time</th>
<th>Line Contents</th>
</tr>
</thead>
</table>
| 27     |      |       |         |        | def kmeans(points, n_clusters):
| 28     | 1    | 16    | 16.0    | 0.0    | centers = points[-n_clusters:].tolist()           |
| 29     | 1    | 2     | 2.0     | 0.0    | while True:
| 30     | 54   | 55    | 1.0     | 0.0    | old_centers = centers                             |
| 31     | 54   | 11012265 | 203930.8 | 93.2   | labels = find_labels(points, labels)               |
| 32     | 54   | 802873 | 14868.0 | 6.8    | centers = compute_centers(points, labels)        |
| 33     | 54   | 116   | 2.1     | 0.0    | if centers == old_centers:
| 34     | 1    | 0     | 0.0     | 0.0    | break                                              |
| 35     | 1    | 1     | 1.0     | 0.0    | return labels                                      |
How can we optimize repeated operations on arrays?
Seven Strategies:

1. Line Profiling
2. Numpy Vectorization
def dist(x, y):
    return sum((xi - yi) ** 2
                for xi, yi in zip(x, y))

def find_labels(points, centers):
    labels = []
    for point in points:
        distances = [dist(point, center)
                     for center in centers]
        labels.append(distances.index(min(distances)))
    return labels
def dist(x, y):
    return sum((xi - yi) ** 2
               for xi, yi in zip(x, y))

def find_labels(points, centers):
    labels = []
    for point in points:
        distances = [dist(point, center)
                     for center in centers]
        labels.append(distances.index(min(distances)))
    return labels

import numpy as np

def find_labels(points, centers):
    diff = (points[:, None, :] - centers) ** 2
    distances = diff.sum(-1)
    return distances.argmin(1)
def dist(x, y):
    return sum((xi - yi) ** 2
                   for xi, yi in zip(x, y))

def find_labels(points, centers):
    labels = []
    for point in points:
        distances = [dist(point, center)
                     for center in centers]
        labels.append(distances.index(min(distances)))
    return labels

import numpy as np

def find_labels(points, centers):
    diff = (points[:, None, :] - centers) ** 2
    distances = diff.sum(-1)
    return distances.argmin(1)
def dist(x, y):
    return sum((xi - yi) ** 2 for xi, yi in zip(x, y))

def find_labels(points, centers):
    labels = []
    for point in points:
        distances = [dist(point, center) for center in centers]
        labels.append(distances.index(min(distances)))
    return labels

import numpy as np

def find_labels(points, centers):
    diff = (points[:, None, :] - centers) ** 2
    distances = diff.sum(-1)
    return distances.argmin(1)
def dist(x, y):
    return sum((xi - yi) ** 2
                for xi, yi in zip(x, y))

def find_labels(points, centers):
    labels = []
    for point in points:
        distances = [dist(point, center)
                     for center in centers]
        labels.append(distances.index(min(distances)))
    return labels

import numpy as np

def find_labels(points, centers):
    diff = (points[:, None, :] - centers) ** 2
    distances = diff.sum(-1)
    return distances.argmin(1)
%lprun -f kmeans kmeans(points, 10)

Timer unit: 1e-06 s

Total time: 0.960594 s
File: <ipython-input-5-72ba0feed022>
Function: kmeans at line 23

Line # Hits Time Per Hit % Time Line Contents
==============================================================
23                                           def kmeans(points, n_clusters):
24         1           11     11.0      0.0      centers = points[-n_clusters:]
25         1            0      0.0      0.0      while True:
26        54           50      0.9      0.0          old_centers = centers
27        54        87758   1625.1      9.1          labels = find_labels(points, centers)
28        54       872625  16159.7     90.8          centers = compute_centers(points, labels)
29        54          149      2.8      0.0          if centers == old_centers:
30         1            1      1.0      0.0              break
31         1            0      0.0      0.0      return labels
%lprun -f kmeans kmeans(points, 10)

Timer unit: 1e-06 s

Total time: 0.960594 s
File: <ipython-input-5-72ba0feed022>
Function: kmeans at line 23

Line #    Hits    Time    Per Hit    % Time    Line Contents
==============================================
   23                                           def kmeans(points, n_clusters):
   24       1     11     11.0       0.0     centers = points[-n_clusters:]
   25       1      0     0.0       0.0          while True:
   26      54     50     0.9       0.0     old_centers = centers
   27      54   87758   1625.1     9.1     labels = find_labels(points, centers)
   28      54   872625  16159.7    90.8     centers = compute_centers(points, labels)
   29      54    149     2.8       0.0     if centers == old_centers:
   30       1     1     1.0       0.0              break
   31       1      0     0.0       0.0     return labels
def compute_centers(points, labels):
    n_centers = len(set(labels))
    n_dims = len(points[0])

    centers = [[0 for i in range(n_dims)]
                for j in range(n_centers)]
    counts = [0 for j in range(n_centers)]

    for label, point in zip(labels, points):
        counts[label] += 1
        centers[label] = [a + b for a, b in zip(centers[label], point)]

    return [[x / count for x in center]
            for center, count in zip(centers, counts)]
def compute_centers(points, labels):
    n_centers = len(set(labels))
    n_dims = len(points[0])
    
centers = [[0 for i in range(n_dims)]
               for j in range(n_centers)]
    counts = [0 for j in range(n_centers)]
    
    for label, point in zip(labels, points):
        counts[label] += 1
        centers[label] = [a + b for a, b in
                          zip(centers[label], point)]
    
    return [[x / count for x in center]
            for center, count in zip(centers, counts)]

numPy

def compute_centers(points, labels):
    n_centers = len(set(labels))
    return np.array([[points[labels == i].mean(0)
                       for i in range(n_centers)]])
def compute_centers(points, labels):
    n_centers = len(set(labels))
    n_dims = len(points[0])

    centers = [[0 for i in range(n_dims)]
                for j in range(n_centers)]
    counts = [0 for j in range(n_centers)]

    for label, point in zip(labels, points):
        counts[label] += 1
        centers[label] = [a + b for a, b in zip(centers[label], point)]

    return [[x / count for x in center]
             for center, count in zip(centers, counts)]

def compute_centers(points, labels):
    n_centers = len(set(labels))
    return np.array([[points[labels == i].mean(0)
                      for i in range(n_centers)]]).T
def compute_centers(points, labels):
    n_centers = len(set(labels))
    n_dims = len(points[0])

    centers = [[0 for i in range(n_dims)]
               for j in range(n_centers)]
    counts = [0 for j in range(n_centers)]

    for label, point in zip(labels, points):
        counts[label] += 1
        centers[label] = [a + b for a, b in zip(centers[label], point)]

    return [[x / count for x in center]
            for center, count in zip(centers, counts)]

def compute_centers(points, labels):
    n_centers = len(set(labels))
    return np.array([[points[labels == i].mean(0)
                      for i in range(n_centers)]]).T
%timeit kmeans(points, 10)

131 ms ± 3.68 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

Down from 7.44 seconds to 0.13 seconds!

Key: repeated operations pushed into a compiled layer:

Python overhead **per array** rather than **per array element**.
Advantages:
- Python overhead *per array* rather than *per array element*
- Compact domain specific language for array operations
- NumPy is widely available

Disadvantages:
- Batch operations can lead to excessive memory usage
- Different way of thinking about writing code

Recommendation: Use NumPy everywhere!
Deeper dive into NumPy Vectorization

“Losing your Loops” / PyCon 2015
Seven Strategies:

1. Line Profiling
2. Numpy Vectorization
3. Specialized Data Structures
import numpy as np

def find_labels(points, centers):
    diff = (points[:, None, :] - centers) ** 2
    distances = diff.sum(-1)
    return distances.argmin(1)
KD-Tree:

Data structure designed for nearest neighbor searches

**Scipy**

```python
from scipy.spatial import cKDTree

def find_labels(points, centers):
    distances, labels = cKDTree(centers).query(points, 1)
    return labels
```

**Numpy Code**

```python
import numpy as np

def find_labels(points, centers):
    diff = (points[:, None, :] - centers) ** 2
    distances = diff.sum(-1)
    return distances.argmin(1)
```
def compute_centers(points, labels):
    n_centers = len(set(labels))
    return np.array([[points[labels == i]].mean(0)
                     for i in range(n_centers)])
Pandas Dataframe:
Efficient structure for group-wise operations

```python
import pandas as pd

def compute_centers(points, labels):
    df = pd.DataFrame(points)
    return df.groupbyby(labels).mean().values
```

```python
def compute_centers(points, labels):
    n_centers = len(set(labels))
    return np.array([points[labels == i].mean(0)
                     for i in range(n_centers)])
```
```python
%timeit kmeans(points, 10)
```

102 ms ± 2.52 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

Compared to:
- 7.44 Seconds in Python
- 131 ms with NumPy
Other Useful Data Structures

`scipy.spatial`
  for spatial queries like distances, nearest neighbors, etc.

`pandas`
  for SQL-like grouping & aggregation

`xarray`
  for grouping across multiple dimensions

`scipy.sparse`
  sparse matrices for 2-dimensional structured data

`sparse package`
  for N-dimensional structured data

`scipy.sparse.csgraph`
  for graph-like problems (e.g. finding shortest paths)
Advantages:
- Often fastest possible way to solve a particular problem

Disadvantages:
- Requires broad & deep understanding of both algorithms and their available implementations

Recommendation: Use whenever possible!
Seven Strategies:

1. Line Profiling
2. Numpy Vectorization
3. Specialized Data Structures
4. Cython
```python
def dist(x, y):
    dist = 0
    for i in range(len(x)):
        dist += (x[i] - y[i]) ** 2
    return dist

def find_labels(points, centers):
    labels = []
    for point in points:
        distances = [dist(point, center) for center in centers]
        labels.append(distances.index(min(distances)))
    return labels

centers = points[:10]
%timeit find_labels(points, centers)

122 ms ± 5.82 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```
%%cython
cimport numpy as np

cdef double dist(double[::] x, double[::] y):
    cdef double dist = 0
    for i in range(len(x)):
        dist += (x[i] - y[i]) ** 2
    return dist

def find_labels(points, centers):
    labels = []
    for point in points:
        distances = [dist(point, center) for center in centers]
        labels.append(distances.index(min(distances)))
    return labels

centers = points[:10]
%timeit find_labels(points, centers)

97.7 ms ± 12.2 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
%%cython
cimport numpy as np

cdef double dist(double[:,:] x, double[:,:] y):
    cdef double dist = 0
    for i in range(len(x)):
        dist += (x[i] - y[i]) ** 2
    return dist

def find_labels(points, centers):
    labels = []
    for point in points:
        distances = [dist(point, center) for center in centers]
        labels.append(distances.index(min(distances)))
    return labels

centers = points[:,:10]
%timeit find_labels(points, centers)

97.7 ms ± 12.2 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
def find_labels(points, centers):
    labels = []
    for point in points:
        distances = [dist(point, center) for center in centers]
        labels.append(distances.index(min(distances)))
    return labels

centers = points[:10]
%timeit find_labels(points, centers)

97.7 ms ± 12.2 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
def find_labels(double[:, :] points, double[:, :] centers):
    cdef int n_points = points.shape[0]
    cdef int n_centers = centers.shape[0]
    cdef double[:,] labels = np.zeros(n_points)
    cdef double distance, nearest_distance
    cdef int nearest_index

    for i in range(n_points):
        nearest_distance = np.inf
        for j in range(n_centers):
            distance = dist(points[i], centers[j])
            if distance < nearest_distance:
                nearest_distance = distance
                nearest_index = j
        labels[i] = nearest_index
    return np.asarray(labels)

centers = points[:10]
%timeit find_labels(points, centers)

1.72 ms ± 27.3 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```python
def find_labels(double[:, :, :] points, double[:, :, :] centers):
cdef int n_points = points.shape[0]
cdef int n_centers = centers.shape[0]
cdef double[:, :] labels = np.zeros(n_points)
cdef double distance, nearest_distance
cdef int nearest_index

for i in range(n_points):
    nearest_distance = np.inf
    for j in range(n_centers):
        distance = dist(points[i], centers[j])
        if distance < nearest_distance:
            nearest_distance = distance
            nearest_index = j
    labels[i] = nearest_index
return np.asarray(labels)

centers = points[:10]
%timeit find_labels(points, centers)
1.72 ms ± 27.3 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```
```python
def find_labels(double[:, :, :] points, double[:, :, :] centers):
    cdef int n_points = points.shape[0]
    cdef int n_centers = centers.shape[0]
    cdef double[:, :] labels = np.zeros(n_points)
    cdef double distance, nearest_distance
    cdef int nearest_index

    for i in range(n_points):
        nearest_distance = np.inf
        for j in range(n_centers):
            distance = dist(points[i], centers[j])
            if distance < nearest_distance:
                nearest_distance = distance
                nearest_index = j
        labels[i] = nearest_index

    return np.asarray(labels)
```

```text
centers = points[:, :10]
%timeit find_labels(points, centers)

1.72 ms ± 27.3 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```
Advantages:
- Python-like code at C-like speeds!

Disadvantages:
- Explicit type annotation can be cumbersome
- Often requires restructuring code
- Code build becomes more complicated

Recommendation: use for operations that can't easily be expressed in NumPy
Seven Strategies:

1. Line Profiling
2. Numpy Vectorization
3. Specialized Data Structures
4. Cython
5. Numba
```python
def dist(x, y):
    dist = 0
    for i in range(len(x)):
        dist += (x[i] - y[i])**2
    return dist

def find_labels(points, centers):
    labels = []
    min_dist = np.inf
    min_label = 0
    for i in range(len(points)):
        for j in range(len(centers)):
            distance = dist(points[i], centers[j])
            if distance < min_dist:
                min_dist, min_label = distance, j
        labels.append(min_label)
    return labels

centers = points[:10]
%timeit find_labels(points, centers)

97.7 ms ± 12.2 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```
import numba

@numba.jit(nopython=True)
def dist(x, y):
    dist = 0
    for i in range(len(x)):
        dist += (x[i] - y[i]) ** 2
    return dist

@numba.jit(nopython=True)
def find_labels(points, centers):
    labels = []
    min_dist = np.inf
    min_label = 0
    for i in range(len(points)):
        for j in range(len(centers)):
            distance = dist(points[i], centers[j])
            if distance < min_dist:
                min_dist, min_label = distance, j
        labels.append(min_label)
    return labels

centers = points[:10]
%timeit find_labels(points, centers)

1.47 ms ± 14.2 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
Advantages:
- Python code JIT-compiled to fortran speeds!

Disadvantages:
- Heavy dependency chain (LLVM)
- Some Python constructs not supported
- Still a bit finicky

Recommendation: use for analysis scripts where dependencies are not a concern.

See my blog post *Optimizing Python in the Real World: NumPy, Numba, and the NUFFT*
Seven Strategies:

1. Line Profiling
2. Numpy Vectorization
3. Specialized Data Structures
4. Cython
5. Numba
6. Dask
Parallel Computation:

Typical data manipulation with NumPy:

```python
import numpy as np

a = np.random.randn(1000)
b = a * 4
b_min = b.min()
print(b_min)

-13.2982888603
```

http://dask.pydata.org/
Parallel Computation:

Same operation with dask

```python
import dask.array as da

a2 = da.from_array(a, chunks=200)

b2 = a2 * 4

b2_min = b2.min()
print(b2_min)

dask.array<amin-aggregate, shape=(),
    dtype=float64, chunksize=()>
```

http://dask.pydata.org/
```python
import dask

a2 = dask.array.from_array(a, chunks=200)

b2 = a2 * 4

b2_min = b2.min()

print(b2_min)

dask.array<amin-aggregate, shape=(), dtype=float64, chunksize=()>

Same operation with dask
"Task Graph"
```
Parallel Computation:

Same operation with dask

```python
import dask.array as da

a2 = da.from_array(a, chunks=200)
b2 = a2 * 4
b2_min = b2.min()
print(b2_min)

dask.array<amin-aggregate, shape=(),
          dtype=float64, chunksize=()>

b2_min.compute()

-13.298288860312757
```

http://dask.pydata.org/
def find_labels(points, centers):
    diff = (points[::, None, :] - centers) ** 2
    distances = diff.sum(-1)
    return distances.argmin(1)

labels = find_labels(points, centers)
from dask import array as da

def find_labels(points, centers):
    diff = (points[:, None, :] - centers) ** 2
    distances = diff.sum(-1)
    return distances.argmin(1)

points = da.from_array(points, chunks=1000)
centers = da.from_array(centers, chunks=5)

labels = find_labels(points, centers)
from dask import array as da

def find_labels(points, centers):
    diff = points[:, None, :] - centers
    distances = (diff ** 2).sum(-1)
    return distances.argmin(1)

points_dask = da.from_array(points, chunks=1000)
centers_dask = da.from_array(centers, chunks=5)
labels = find_labels(points_dask, centers_dask)
```python
def find_labels(points, centers):
    diff = (points[:, None, :) - centers)
    distances = (diff ** 2).sum(-1)
    return distances.argmin(1)

def compute_centers(points, labels):
    points_df = dd.from_dask_array(points)
    labels_df = dd.from_dask_array(labels)
    return points_df.groupby(labels_df).mean()

def kmeans(points, n_clusters):
    centers = points[-n_clusters:]
    points = da.from_array(points, 1000)
    while True:
        old_centers = centers
        labels = find_labels(points, da.from_array(centers, 5))
        centers = compute_centers(points, labels)
        centers = centers.compute().values
        if np.all(centers == old_centers):
            break
    return labels.compute()

%timeit kmeans(points, 10)
3.28 s ± 192 ms per loop (mean ± std. dev. of 7 runs)
```
def find_labels(points, centers):
    diff = (points[:, None, :] - centers)
    distances = (diff ** 2).sum(-1)
    return distances.argmin(1)

def compute_centers(points, labels):
    points_df = dd.from_dask_array(points)
    labels_df = dd.from_dask_array(labels)
    return points_df.groupbyby(labels_df).mean()

def kmeans(points, n_clusters):
    centers = points[-n_clusters:]
    points = da.from_array(points, 1000)
    while True:
        old_centers = centers
        labels = find_labels(points, da.from_array(centers, 5))
        centers = compute_centers(points, labels)
        centers = centers.compute().values
        if np.all(centers == old_centers):
            break
    return labels.compute()

%timeit kmeans(points, 10)

3.28 s ± 192 ms per loop (mean ± std. dev. of 7 runs)
def find_labels(points, centers):
    diff = (points[:, None, :] - centers)
    distances = (diff ** 2).sum(-1)
    return distances.argmin(1)

def compute_centers(points, labels):
    points_df = dd.from_dask_array(points)
    labels_df = dd.from_dask_array(labels)
    return points_df.groupby(labels_df).mean()

def kmeans(points, n_clusters):
    centers = points[-n_clusters:]
    points = da.from_array(points, 1000)
    while True:
        old_centers = centers
        labels = find_labels(points, da.from_array(centers, 5))
        centers = compute_centers(points, labels)
        centers = centers.compute().values
        if np.all(centers == old_centers):
            break
    return labels.compute()

%timeit kmeans(points, 10)

3.28 s ± 192 ms per loop (mean ± std. dev. of 7 runs)
Advantages:
- Easy distributed coding, often with no change to NumPy or Pandas code!
- Even works locally on out-of-core data

Disadvantages:
- High overhead, so not suitable for smaller problems

Recommendation: use when data size or computation time warrants

See my blog post *Out of Core Dataframes in Python: Dask and OpenStreetMap*
http://jakevdp.github.io/blog/2015/08/14/out-of-core-dataframes-in-python/
Seven Strategies:

1. Line Profiling
2. Numpy Vectorization
3. Specialized Data Structures
4. Cython
5. Numba
6. Dask
Seven Strategies:

1. Line Profiling
2. Numpy/Vectorsization
3. Specialized Data Structures
4. Cython
5. Numba
6. Dask
Seven Strategies:

1. Line Profiling
2. Numpy Vectorization
3. Specialized Data Structures
4. Cython
5. Numba
6. Dask
7. Find an Existing Implementation!
from sklearn.cluster import KMeans

%timeit KMeans(4).fit_predict(points)

28.5 ms ± 701 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)

from dask_ml.cluster import KMeans

%timeit KMeans(4).fit(points).predict(points)

8.7 s ± 202 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

Recommendation: resist the urge to reinvent the wheel.
You can implement it yourself . . . you can make your numerical code fast!

But the community is Python’s greatest strength.
Thank You!

- Email: jakevdp@uw.edu
- Twitter: @jakevdp
- Github: jakevdp
- Web: http://vanderplas.com/
- Blog: http://jakevdp.github.io/

Slides available at https://speakerdeck.com/jakevdp/
Notebook with code from this talk: http://goo.gl/d8ZWwp