Losing Your Loops

Fast Numerical Computing with NumPy

Jake VanderPlas

PyCon 2015

Python is Fast

. . . for Writing, Testing, and Developing Code

Python is Fast

. . . for Writing, Testing, and Developing Code

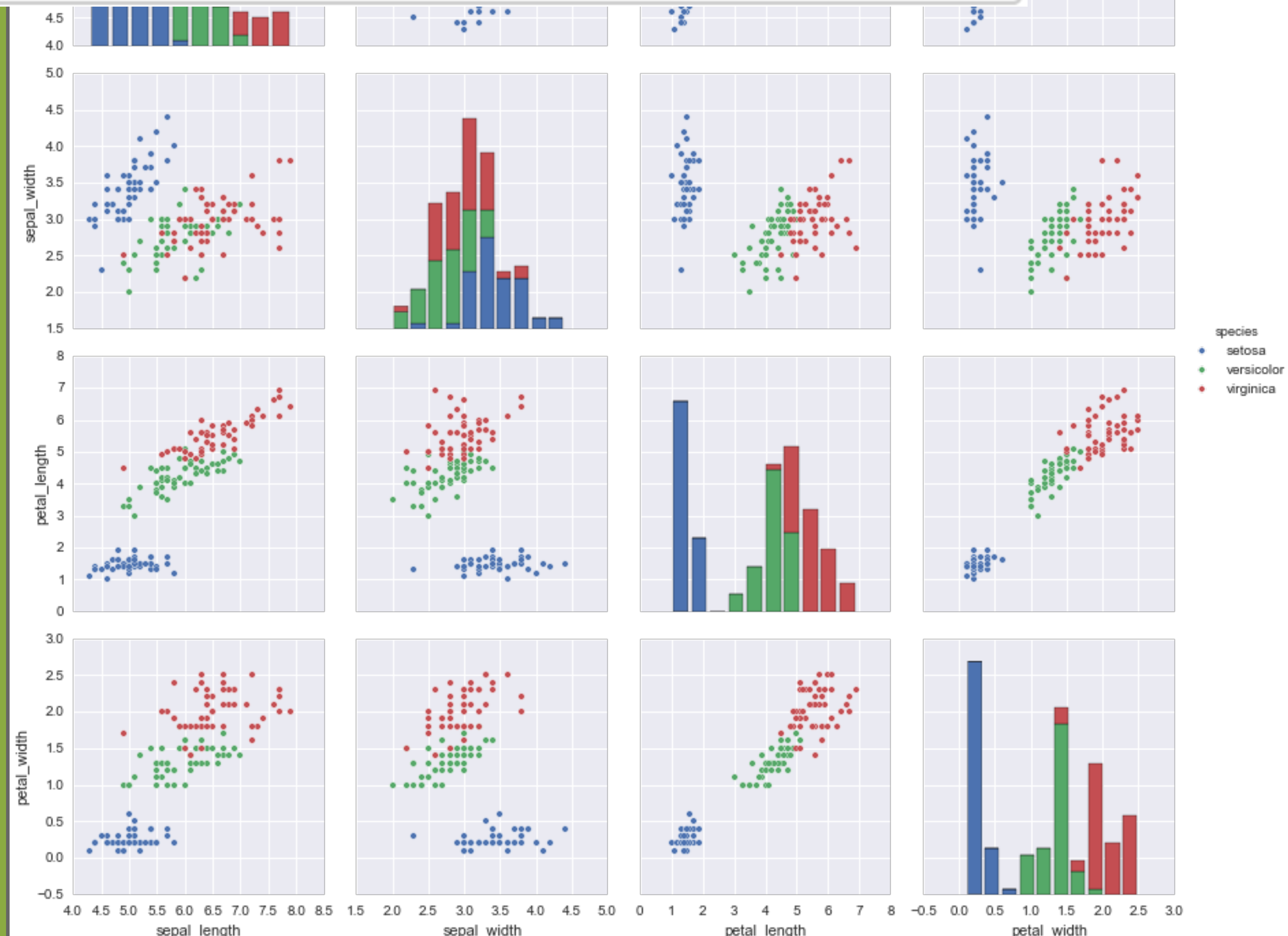
```
# Hello World in Python
print("hello world")
```

Python is Fast

. . . for Writing, Testing, and Developing Code

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

```python
import seaborn as sns
data = sns.load_dataset("iris")
sns.pairplot(data, hue="species");
```



Python is Fast

. . . because it is interpreted,
dynamically typed,
and high-level

Python is Slow

. . . for Repeated Execution of Low-level Tasks

A simple function implemented in 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
```

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

        1000 loops, best of 3: 1.69 ms per loop
```

(`%timeit` is a useful *magic command* available in IPython)

The same function implemented in Fortran . . .

```
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
```

```
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)

         100000 loops, best of 3: 17.9 µs per loop
```

Python is ~100x slower than Fortran for this simple task!

Why is Python Slow?

Python is a **high-level**, **interpreted** and **dynamically-typed** language.

Each Python operation comes with a small type-checking overhead.

With many repeated small operations (e.g. in a loop), this overhead becomes significant!

The paradox . . .

what makes Python fast

(for development)

is

what makes Python slow

(for code execution)

* Though JIT compilers like PyPy, Numba, etc. may change this soon . . .

`import numpy`

NumPy is designed to help us get the best of both worlds . . .

- Fast *development time* of Python
- Fast *execution time* of C/Fortran

. . . by pushing repeated operations into a statically-typed compiled layer.

Four Strategies

For Speeding-up Code with NumPy

1. Use NumPy's **ufuncs**
2. Use NumPy's **aggregations**
3. Use NumPy's **broadcasting**
4. Use NumPy's **slicing, masking,** and **fancy indexing**

Overall goal: push repeated operations into compiled code and *Get Rid of Slow Loops!*

Strategy #1:

Use NumPy's ufuncs

Strategy #1:

ufuncs are NumPy's Use NumPy's ufuncs

Universal Functions . . .

They operate element-wise on arrays.

Element-wise operations . . .
. . . with Python lists:

Strategy #1:

Use NumPy's ufuncs

```
a = [1, 3, 2, 4, 3, 1, 4, 2]
b = [val + 5 for val in a]
print(b)
```

```
[6, 8, 7, 9, 8, 6, 9, 7]
```

Element-wise operations . . .

. . . with Python lists:

```python
a = [1, 3, 2, 4, 3, 1, 4, 2]
b = [val + 5 for val in a]
print(b)
```

```
[6, 8, 7, 9, 8, 6, 9, 7]
```

. . . with NumPy arrays:

```python
import numpy as np
a = np.array(a)
```

```python
b = a + 5  # element-wise
print(b)
```

```
[6 8 7 9 8 6 9 7]
```

Ufuncs are fast . . .

```
a = list(range(100000))
%timeit [val + 5 for val in a]
```

```
100 loops, best of 3: 7.19 ms per loop
```

Ufuncs are fast . . .

```
a = list(range(100000))
%timeit [val + 5 for val in a]
```

```
100 loops, best of 3: 7.19 ms per loop
```

```
a = np.array(a)
%timeit a + 5
```

```
10000 loops, best of 3: 82.4 µs per loop
```

Ufuncs are fast . . .

```
a = list(range(100000))
%timeit [val + 5 for val in a]
```

100 loops, best of 3: 7.19 ms per loop

```
a = np.array(a)
%timeit a + 5
```

10000 loops, best of 3: 82.4 µs per loop

. . . 100x speedup with NumPy!

Strategy #1:

Use NumPy's ufuncs

There are many ufuncs available:

- Arithmetic Operators: `+ - * / // % **`
- Bitwise Operators: `& | ~ ^ >> <<`
- Comparison Oper's: `< > <= >= == !=`
- Trig Family: `np.sin, np.cos, np.tan ...`
- Exponential Family: `np.exp, np.log, np.log10 ...`
- Special Functions: `scipy.special.*`

. . . and many, many more.

Strategy #2:

Use NumPy's aggregations

Strategy #2:

Use NumPy's aggregations

Aggregations are functions which summarize the values in an array (e.g. min, max, sum, mean, etc.)

NumPy aggregations are much faster than Python built-ins . . .

```
from random import random
c = [random() for i in range(100000)]
```

```
%timeit min(c)
```

```
100 loops, best of 3: 2.18 ms per loop
```

NumPy aggregations are much faster than Python built-ins . . .

```python
from random import random
c = [random() for i in range(100000)]
```

```python
%timeit min(c)
```

```
100 loops, best of 3: 2.18 ms per loop
```

```python
c = np.array(c)
```

```python
%timeit c.min()
```

```
10000 loops, best of 3: 30.8 µs per loop
```

NumPy aggregations are much faster than Python built-ins . . .

Strategy #2:

```
from random import random
c = [random() for i in range(100000)]
```

```
%timeit min(c)
```

```
100 loops, best of 3: 2.18 ms per loop
```

```
c = np.array(c)
```

```
%timeit c.min()
```

```
10000 loops, best of 3: 30.8 µs per loop
```

~70x speedup with NumPy!

NumPy aggregations also work on multi-dimensional arrays . . .

```
M = np.random.randint(0, 10, (3, 5))
M
```

```
array([[2, 9, 2, 1, 3],
       [2, 1, 5, 2, 4],
       [8, 9, 1, 3, 5]])
```

```
M.sum()
```

```
57
```

NumPy aggregations also work on multi-dimensional arrays . . .

```
M = np.random.randint(0, 10, (3, 5))
M
```

```
array([[2, 9, 2, 1, 3],
       [2, 1, 5, 2, 4],
       [8, 9, 1, 3, 5]])
```

```
M.sum(axis=0)
```

```
array([12, 19,  8,  6, 12])
```

```
M.sum(axis=1)
```

```
array([17, 14, 26])
```

Strategy #2:

Use NumPy's aggregations

Lots of aggregations available . . .

```
np.min() np.max() np.sum() np.prod()
np.mean() np.std() np.var() np.any()
np.all() np.median() np.percentile()
np.argmin() np.argmax()...

np.nanmin() np.nanmax() np.nansum()...
```

. . . and all have the same call signature. Use them often!

Strategy #3:

Use NumPy's broadcasting

Strategy #3:

Use NumPy's broadcasting

Broadcasting is a set of rules by which *ufuncs* operate on arrays of different sizes and/or dimensions.

Visualizing Broadcasting...

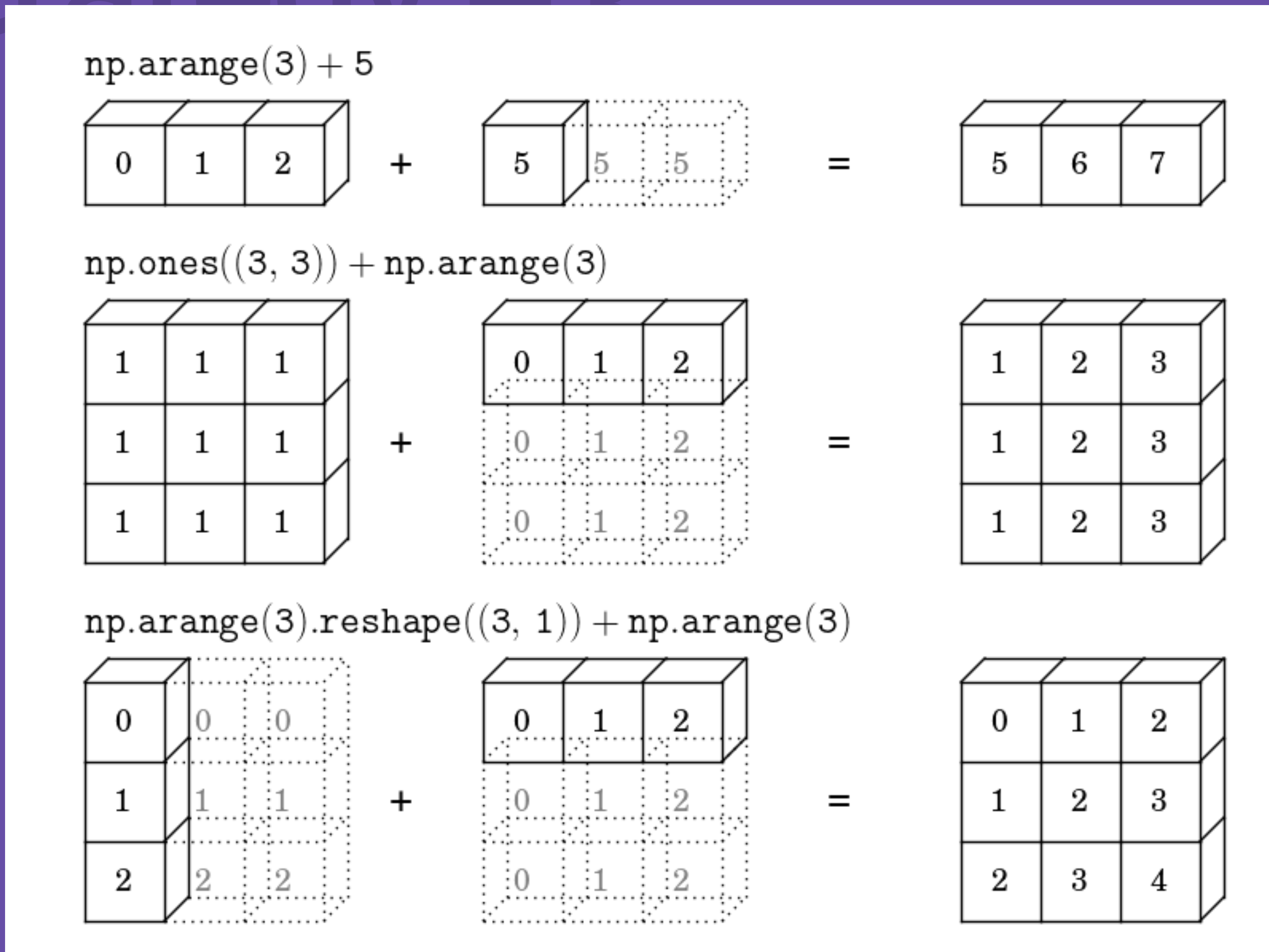


Image source: `http://astroML.org/`

Broadcasting rules . . .

Strategy #3:

Use NumPy's broadcasting

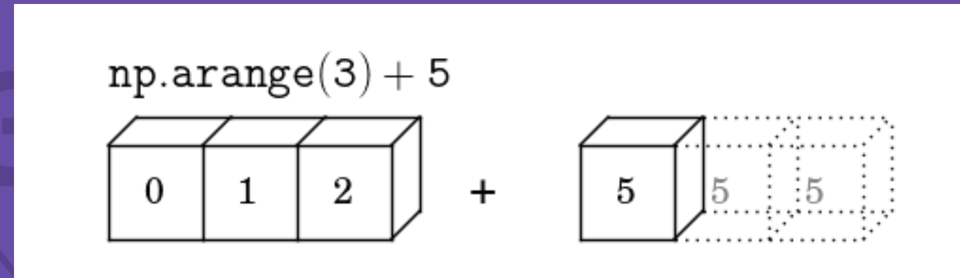
1. If array shapes differ, left-pad the smaller shape with 1s

2. If any dimension does not match, broadcast the dimension with size=1

3. If neither non-matching dimension is 1, raise an error.

Strategy #3
Use NumPy's broadcasting

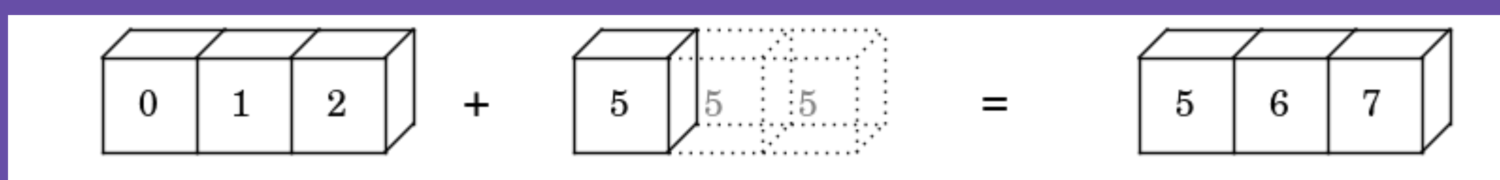
1. If array shapes differ, left-pad the smaller shape with 1s



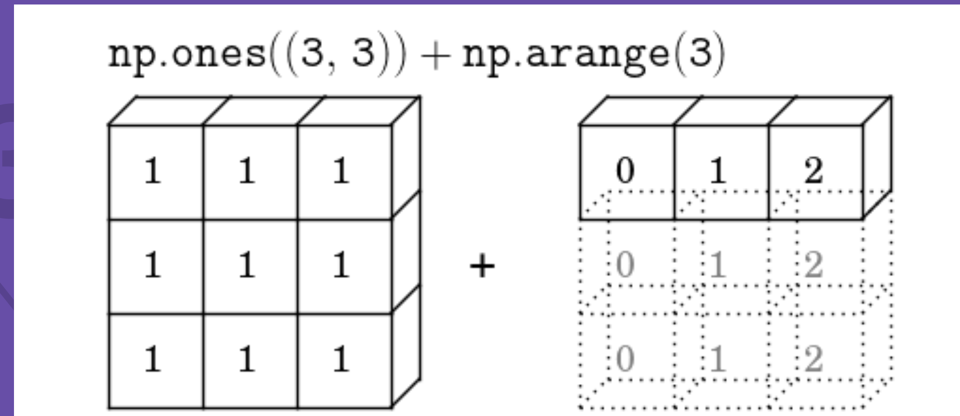
2. If any dimension does not match, broadcast the dimension with size=1

```
   shape=[3]    shape=[]

1. shape=[3]    shape=[1]

2. shape=[3]    shape=[3]

final shape = [3]
```

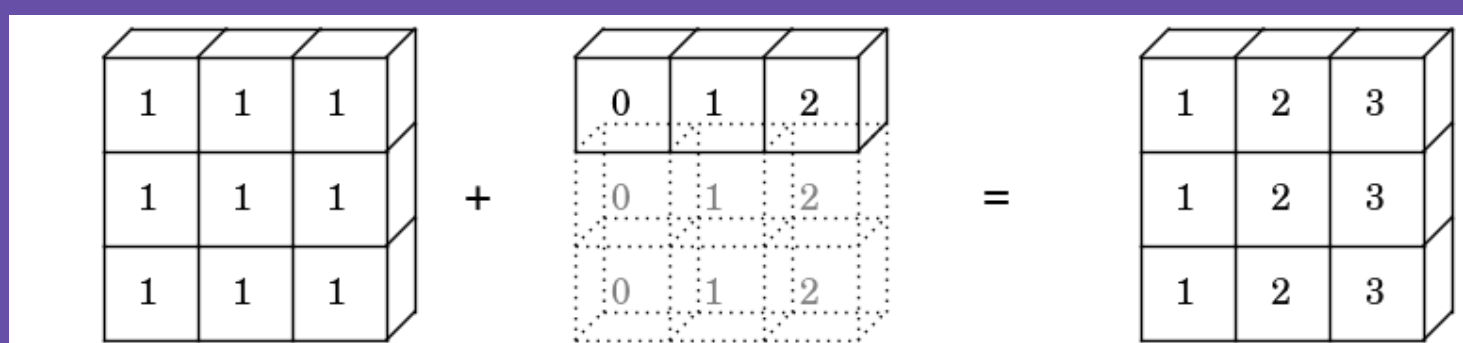
3. If neither non-matching dimension is 1, raise an error.



1. If array shapes differ, left-pad the smaller shape with 1s
2. If any dimension does not match, broadcast the dimension with size=1
3. If neither non-matching dimension is 1, raise an error.

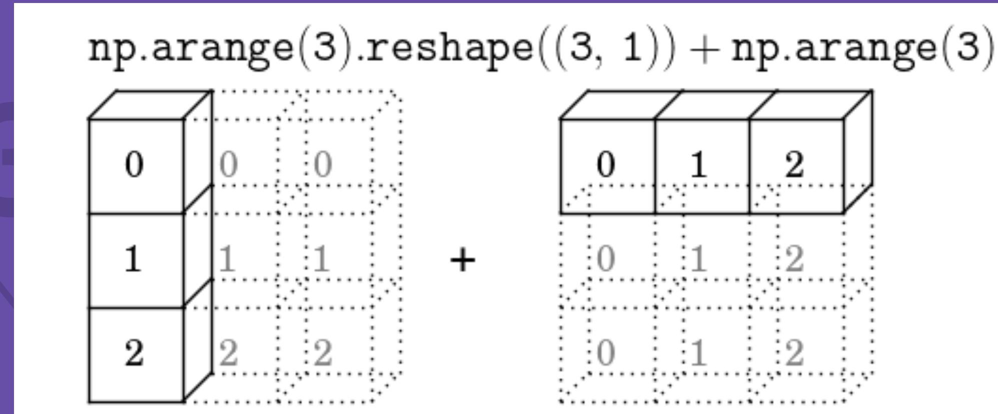



```
   shape=[3,3] shape=[3]

1. shape=[3,3] shape=[1,3]

2. shape=[3,3] shape=[3,3]

final shape = [3,3]
```

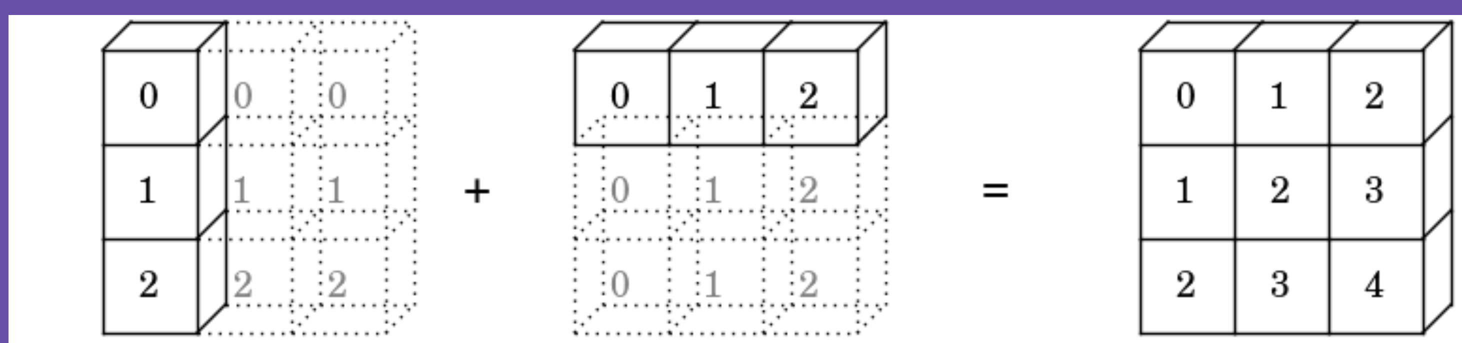


Strategy #3

1. If array shapes differ, left-pad the smaller shape with 1s
2. If any dimension does not match, broadcast the dimension with size=1
3. If neither non-matching dimension is 1, raise an error.




```
   shape=[3,1] shape=[3]

1. shape=[3,1] shape=[1,3]

2. shape=[3,3] shape=[3,3]

final shape = [3,3]
```



Strategy #4:

Use NumPy's slicing, masking, and fancy indexing

With Python lists, indexing accepts integers or slices . . .

```
L = [2, 3, 5, 7, 11]
```

```
L[0]  # integer index
```

```
2
```

```
L[1:3]  # slice for multiple elements
```

```
[3, 5]
```

NumPy arrays are similar . . .

```
L = np.array(L)
L
```

```
array([ 2,  3,  5,  7, 11])
```

```
L[0]
```

```
2
```

```
L[1:3]
```

```
array([3, 5])
```

Strategy #4:

Use NumPy's slicing, masking, and fancy indexing

. . . but NumPy offers other fast and convenient indexing options as well.

“Masking”: indexing with boolean masks

```
L
```

```
array([ 2,  3,  5,  7, 11])
```

A mask is a boolean array:

```
mask = np.array([False, True, True,
                 False, True])
L[mask]
```

```
array([ 3,  5, 11])
```

"Masking": indexing with boolean masks

```
L
```

```
array([ 2,  3,  5,  7, 11])
```

Masks are often constructed using comparison operators and boolean logic, e.g.

```
mask = (L < 4) | (L > 8) # "|" = "bitwise OR"
L[mask]
```

```
array([ 2,  3, 11])
```

“Fancy Indexing”: passing a list/array of indices . . .

```
L
```

```
array([ 2,  3,  5,  7, 11])
```

```
ind = [0, 4, 2]
L[ind]
```

```
array([ 2, 11,  5])
```

Multiple dimensions:
use commas to separate indices!

```
M = np.arange(6).reshape(2, 3)
M
```

```
array([[0, 1, 2],
       [3, 4, 5]])
```

```
# multiple indices separated by comma
M[0, 1]
```

```
1
```

Multiple dimensions:
use commas to separate indices!

```
M = np.arange(6).reshape(2, 3)
M
```

```
array([[0, 1, 2],
       [3, 4, 5]])
```

```
# mixing slices and indices
M[:, 1]
```

```
array([1, 4])
```

Masking in multiple dimensions . . .

```
M = np.arange(6).reshape(2, 3)
M
```

```
array([[0, 1, 2],
       [3, 4, 5]])
```

```
# masking the full array
M[abs(M - 3) < 2]
```

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

Mixing fancy indexing and slicing . . .

```
M = np.arange(6).reshape(2, 3)
M
```

```
array([[0, 1, 2],
       [3, 4, 5]])
```

```
# mixing fancy indexing and slicing
M[[1, 0], :2]
```

```
array([[3, 4],
       [0, 1]])
```

Mixing masking and slicing . . .

```python
M = np.arange(6).reshape(2, 3)
M
```

```
array([[0, 1, 2],
       [3, 4, 5]])
```

```python
# mixing masking and slicing
M[M.sum(axis=1) > 4, 1:]
```

```
array([[4, 5]])
```

Strategy #4:

Use NumPy's slicing, masking, and fancy indexing

All of these operations can be composed and combined in nearly limitless ways!

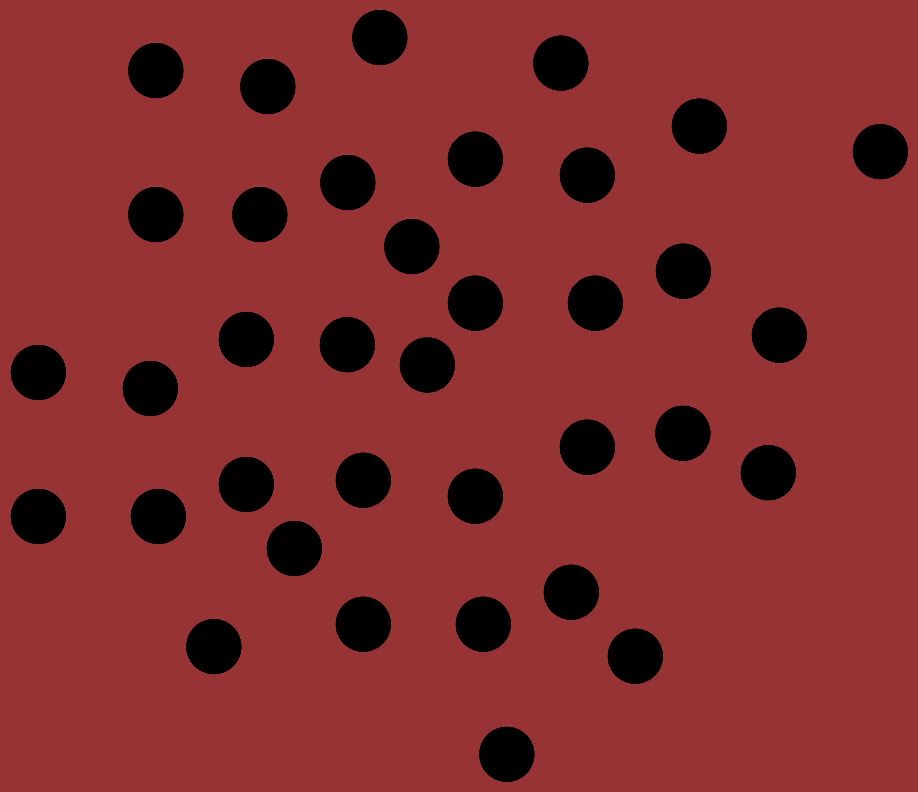
Example:

Computing Nearest Neighbors

Let's combine all these ideas to compute nearest neighbors of points without a single loop!

Example:

Computing Nearest Neighbors

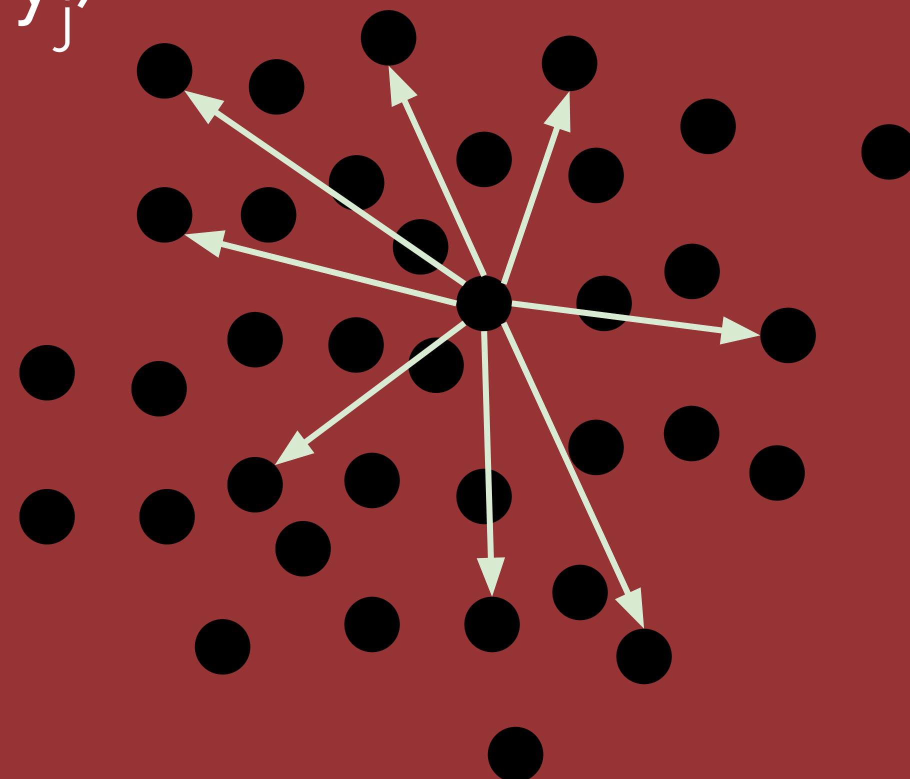


Example:

Computing Nearest Neighbors

Naive approach requires three nested loops . . .

$$D_{ij}^2 = (x_i - x_j)^2 + (y_i - y_j)^2$$



. . . but we can do better.

```python
# 1000 points in 3 dimensions
X = np.random.random((1000, 3))
X.shape
```

```
(1000, 3)
```

```python
# 1000 points in 3 dimensions
X = np.random.random((1000, 3))
X.shape
```

```
(1000, 3)
```

```python
# Broadcasting to find pairwise differences
diff = X.reshape(1000, 1, 3) - X
diff.shape
```

```
(1000, 1000, 3)
```

```
# Aggregate to find pairwise distances
D = (diff ** 2).sum(2)
D.shape
```

```
(1000, 1000)
```

```
# Aggregate to find pairwise distances
D = (diff ** 2).sum(2)
D.shape
```

```
(1000, 1000)
```

```
# set diagonal to infinity to skip self-neighbors
i = np.arange(1000)
D[i, i] = np.inf
```

```python
# print the indices of the nearest neighbor
i = np.argmin(D, 1)
print(i[:10])
```

```
[779 958 801 155  25  24  75 243 911 235]
```

```python
# print the indices of the nearest neighbor
i = np.argmin(D, 1)
print(i[:10])
```

```
[779 958 801 155  25  24  75 243 911 235]
```

```python
# double-check with scikit-learn
from sklearn.neighbors import NearestNeighbors
d, i = NearestNeighbors().fit(X).kneighbors(X, 2)
print(i[:10, 1])
```

```
[779 958 801 155  25  24  75 243 911 235]
```

Summary . . .

- Writing Python is fast; loops can be slow
- NumPy pushes loops into its compiled layer:
  - fast *development* time of Python
  - fast *execution* time of compiled code

Strategies:

1. **ufuncs** for element-wise operations
2. **aggregations** for array summarization
3. **broadcasting** for combining arrays
4. **slicing, masking,** and **fancy indexing** for selecting and operating on subsets of arrays

~ Thank You! ~

Email: jakevdp@uw.edu

Twitter: @jakevdp

Github: jakevdp

Web: http://vanderplas.com/

Blog: http://jakevdp.github.io/