NumPy Techniques

- NumPy universal functions
- Aggregation
- Broadcasting
- Manipulating arrays using Boolean logic
- Additional techniques

NumPy universal functions

- Overview
- Using a loop
- Using a universal function
- Arithmetic
- Reduction and accumulation
- Additional math functions

Overview

Universal functions are a key reason why NumPy arrays are so efficient to manipulate

A universal function is a method on a NumPy array that executes on all elements very efficiently

- Much faster and cleaner than using an explicit loop

Consider the examples on the next 2 slides…

- The 1st example uses a loop - slow and cumbersome
- The 2nd example uses a universal function - fast and elegant

You will always use universal functions rather than loops to process NumPy arrays

Using a loop

This code uses a loop to process data in a NumPy array

- Loops through elements and applies ** to each element
- Gathers results into another NumPy array, one-by-one

```
import numpy as np
from timeit import default_timer as timer

def compute_cubes_loop(data):
    result = np.empty(len(data))
    for i in range(len(data)):
        result[i] = data[i] ** 3
    return result

np.random.seed(0)
data = np.random.randint(1, 10, size=10_000_000)

start = timer()
cubes = compute_cubes_loop(data)
end = timer()
print('Execution time using a loop', end - start)
```

Using a universal function

This code has the same effect, using a universal function

- Applies the ** operator to the NumPy array itself
- NumPy applies the operator to each element implicitly

```
import numpy as np
from timeit import default_timer as timer

def compute_cubes_ufunc(data):
    result = data ** 3
    return result

np.random.seed(0)
data = np.random.randint(1, 10, size=10000000)

start = timer()
cubes = compute_cubes_ufunc(data)
end = timer()
print('Execution time using a universal function', end - start)
```

Arithmetic

NumPy implements all the usual arithmetic operators as universal functions

- You can use an operator directly, or call the equivalent function

```
import numpy as np

a = np.arange(10)

print('Using operators')
print('a + 2  = ', a + 2)
print('a - 2  = ', a - 2)
print('a * 2  = ', a * 2)
print('a / 2  = ', a / 2)
print('a // 2 = ', a // 2)
print('a % 2  = ', a % 2)
print('a ** 2 = ', a ** 2)

print('\nUsing ufuncs explicitly')
print('np.add(a, 2)          = ', np.add(a, 2))
print('np.subtract(a, 2)     = ', np.subtract(a, 2))
print('np.multiply(a, 2)     = ', np.multiply(a, 2))
print('np.divide(a, 2)       = ', np.divide(a, 2))
print('np.floor_divide(a, 2) = ', np.floor_divide(a, 2
print('np.mod(a, 2)          = ', np.mod(a, 2))
print('np.power(a, 2)        = ', np.power(a, 2))
```

Reduction and accumulation

For binary ufuncs such as add, multiply, etc.

- You can call reduce() to reduce array to a single result
- You can call accumulate() to accumulate intermediate results

```
1   import numpy as np
2
3   a = np.arange(2, 5)
4
5   print('reduce() examples')
6   print('Sum    ', np.add.reduce(a))
7   print('Product', np.multiply.reduce(a))
8   print('Power  ', np.power.reduce(a))
9
10  print('accumulate() examples')
11  print('Sum    ', np.add.accumulate(a))
12  print('Product', np.multiply.accumulate(a))
13  print('Power  ', np.power.accumulate(a))
```

Additional math functions

NumPy has universal functions for trig, log, etc.

```
import numpy as np
import scipy.special as sp  # Additional specialized f

a = np.linspace(0, np.pi, 3)
print('sin(a)  = ', np.sin(a))
print('cos(a)  = ', np.cos(a))
print('tan(a)  = ', np.tan(a))
print('sinh(a) = ', np.sinh(a))
print('cosh(a) = ', np.cosh(a))
print('tanh(a) = ', np.tanh(a))

b = np.linspace(0.1, 0.9, 3)
print('arcsin(b)  = ', np.arcsin(b))
print('arccos(b)  = ', np.arccos(b))
print('arctan(b)  = ', np.arctan(b))
print('arcsinh(b) = ', np.arcsinh(b))
print('arccosh(b) = ', np.arccosh(b))
print('arctanh(b) = ', np.arctanh(b))

c = np.array([1, 10, 100])
print('log(c)   = ', np.log(c))       # 2.718281 to th
print('log2(c)  = ', np.log2(c))      # 2 to the power
print('log10(c) = ', np.log10(c))     # 10 to the powe
print('exp(c)   = ', np.exp(c))       # 2.718281 to th
print('exp2(c)  = ', np.exp2(c))      # 2 to the power
print('exp10(c) = ', sp.exp10(c))     # 10 to the powe
```

Aggregation

- Overview
- NumPy vs. Python aggregation functions
- NumPy aggregation functions available
- Working with multidimensional arrays

Overview

When dealing with large amounts of data, you'll probably want to compute statistics such as:

- Sum, product
- Minimum, maximum
- Mean, median, mode
- Variance, standard deviation
- Percentiles

NumPy has aggregation functions for performing these computations very efficiently

NumPy vs. Python aggregation functions

NumPy aggregation functions look very similar to functions available in the standard Python library

- But the NumPy functions are much quicker, so use them!

Consider the following example:

```
import numpy as np
from timeit import default_timer as timer

data = np.random.rand(100_000_000)

start1 = timer()
result1 = sum(data)      # Python sum() function
end1 = timer()
print('Execution time using Python sum():  ', end1 - start1)

start2 = timer()
result2 = np.sum(data)   # NumPy sum() function
end2 = timer()
print('Execution time using NumPy sum(): ', end2 - start2)
```

NumPy aggregation functions available

This example shows the NumPy aggregation functions

- There are also NaN-friendly functions, e.g. np.nansum()

```
import numpy as np
from scipy import stats

data = np.random.rand(100_000_000)

print('Sum            ', np.sum(data))
print('Product        ', np.prod(data))
print('Minimum        ', np.min(data))
print('Maximum        ', np.max(data))
print('Mean           ', np.mean(data))
print('Median         ', np.median(data))
print('Mode           ', stats.mode(data))
print('Variance       ', np.var(data))
print('Std dev        ', np.std(data))
print('50th percentile', np.percentile(data, 50))
```

Working with multidimensional arrays

The aggregation functions work over the entire array

- If the array is multidimensional, all elements are processed

You can also get aggregation results for rows or columns

- Specify the axis parameter, to collapse data on that axis

```
import numpy as np

data = np.arange(9).reshape([3,3])

# Print the entire array.
print('Array data:\n', data)

# Calculate the sum over the entire array.
print('Sum of whole array:', np.sum(data))

# Collapse axis 0 (i.e. collapse the rows), to get sum on each column.
print('Sum for each column:', np.sum(data, axis=0))

# Collapse axis 1 (i.e. collapse the columns), to get sum on each row.
print('Sum for each row:', np.sum(data, axis=1))
```

Broadcasting

- Universal functions and same-shape arrays
- Universal functions and different-shape arrays
- Broadcasting rules
- Understanding the broadcasting rules

Universal functions and same-shape arrays

We discussed universal functions earlier in the chapter

- We showed how to add/subtract/etc. scalars to an array

```
import numpy as np

a = np.array([0, 1, 2])

result = a + 100
print(result)
```

Universal functions also work with arrays for both args In this example, the arrays are the same shape (3,)

```
import numpy as np

a1 = np.array([0, 1, 2])
a2 = np.array([4, 5, 6])

result = a1 + a2
print(result)
```

Universal functions and different-shape arrays

Universal functions also work with different-shape arrays

- This is called broadcasting - see next slide for details

Here's a simple example of broadcasting

- a is one row, m is two rows
- The values in a are "broadcast" across both rows in m

```
1   import numpy as np
2
3   a = np.array([10, 11, 12])
4   print(a.shape)       # (3,)
5
6   m = np.array([[20, 21, 22], [30, 31, 32]])
7   print(m.shape)       # (2,3)
8
9   result = a + m
10  print(result.shape)  # (2, 3)
11  print(result)        # [[30 32 34] [40 42 44]]
```

Broadcasting rules

Broadcasting allows NumPy to stretch arrays of different shapes, to enable binary operations to take place

There are three rules about how broadcasting works, which are applied in the following order:

1. If arrays have a different number of dimensions, the shape of the array with fewer dimensions is filled with 1 on leading edge
2. If shape of arrays is different in any dimension, the array with shape 1 in that dimension is stretched to match the other shape
3. If shape in any dimension is different (and not 1), an error occurs

Understanding the broadcasting rules

Let's re-examine the example from a couple of slides ago

```
a = np.array([10, 11, 12])
m = np.array([[20, 21, 22], [30, 31, 32]])
result = a + m     # [[30 32 34] [40 42 44]]
```

Let's apply broadcasting rule 1 first…

- a and m have different number of dimensions: a is 1D, m is 2D
- a has fewer dimensions, so a shape is filled with 1 on leading edge
- Thus the shape of a becomes (1, 3) i.e. 1 row of 3 columns

Now let's apply broadcasting rule 2…

- a and m have different shapes: a shape is (1,3), m shape is (2,3)
- a shape (1,3) is stretched to match m shape (2,3)

Complex Broadcasting

In this example, both arrays need to be broadcast

```
import numpy as np

a = np.array([[10],[11],[12]])   # Shape (3,1)
b = np.array([20, 21, 22])       # Shape (3,)
result = a + b                   # Shape (3,3)
print(result)                    # [[30 31 32] [31 32 33] [32 33 34]]
```

Let’s apply broadcasting rule 1 first…

- a and b have different number of dimensions: a is 2D, b is 1D
- b has fewer dimensions, so b shape is filled with 1 on leading edge
- Thus the shape of b becomes (1, 3) i.e. 1 row of 3 columns

Now let’s apply broadcasting rule 2…

- a and b have different shapes: a shape is (3,1), b shape is (1,3)
- a cols stretched to match b cols, so a becomes (3,3)

Manipulating arrays using Boolean logic

- Boolean operations
- Boolean aggregation
- Boolean masking

Boolean operations (1/2)

We've seen how to use math operators with NumPy arrays

- `+ - * / // etc.`

You can also use Boolean operators

- `> ⩾ < ⩽ ⩵ ≠ `
- Returns a NumPy array containing True/False in each position

```
import numpy as np

def process_marks(m):
    print('Exam marks\n',      m)
    print('Passes?\n',         m ⩾ 50)
    print('Full marks?\n',     m ⩵ 100)
    print('Not full marks?\n', m ≠ 100)

my_exam_marks = np.array([71, 95, 49, 100, 65])
process_marks(my_exam_marks)

our_exam_marks = np.array([[71, 95, 49, 100, 65], [99, 23, 78, 88, 92]])
process_marks(our_exam_marks)
```

Boolean operations (2/2)

You can combine Boolean operations together

- & and
- | inclusive or
- ^ exclusive or
- ~ not

```
import numpy as np

def process_marks(m)
    print('\nExam marks\n',                m)
    print('B?\n',                          (m ⩾ 60) & (m < 70))
    print('A or U?\n',                     (m ⩾ 70) | (m < 30))
    print('A or even, but not both?\n', (m ⩾ 70) ^ (m % 2 == 0))
    print('Not (A or U)?\n',               ~((m ⩾ 70) | (m < 30)))

my_exam_marks = np.array([71, 95, 49, 100, 65])
process_marks(my_exam_marks)

our_exam_marks = np.array([[71, 95, 49, 100, 65], [99, 22, 78, 88, 92]])
process_marks(our_exam_marks)
```

Note the need for parentheses for operator precedence

Boolean aggregation

You can perform various aggregation operations on the Boolean result arrays

- all() - are all results True?
- any() - are any results True?
- count_nonzero() - count of non-zero (i.e. True) results

```
1   import numpy as np
2
3   def process_marks(m)
4       print('\nExam marks\n',  m)
5       print('All passes?    ', np.all(m ⩾ 50))
6       print('Any passes?    ', np.any(m ⩾ 50))
7       print('Count of passes', np.count_nonzero(m ⩾ 50))
8       print('Count of B     ', np.count_nonzero((m ⩾ 60) & (m < 70)))
9       print('Count of A or U', np.count_nonzero((m ⩾ 70) | (m < 30)))
10
11  my_exam_marks = np.array([71, 95, 49, 100, 65])
12  process_marks(my_exam_marks)
13
14  our_exam_marks = np.array([[71, 95, 49, 100, 65], [99, 22, 78, 88, 92]])
15  process_marks(our_exam_marks)
```

Additional techniques

- Fancy indexing
- Partitioning
- Sorting

Fancy indexing (1/3)

Often you want to get several elements at specific indices

- You can pass an array of indices into []
- Returns a result array with the elements from those indices

```
1   import numpy as np
2
3   a = np.arange(10, 20)
4
5   # Get some elements into a Python list.
6   result1 = [a[1], a[4], a[7]]
7   print('type(result1)', type(result1))
8   print('result1      ', result1)
9
10  # Get some elements into a NumPy array, using fancy ir
11  idx = [1, 4, 7]
12  result2 = a[idx]
13  print('\ntype(result2)', type(result2))
14  print('result2.shape',   result2.shape)
15  print('result2      ',   result2)
16
17  # Get some elements into a 2D NumPy array, using fancy
18  idx = np.array([[1, 4, 7], [2, 5, 8]])
19  result3 = a[idx]
20  print('\ntype(result3)', type(result3))
21  print('result3.shape',   result3.shape)
22  print('result3      ',   result3)
```

Fancy indexing (2/3)

You can use fancy indexing with multidimensional arrays

- Specify a fancy index indicating desired rows
- Specify another fancy index indicating desired columns

```
import numpy as np

a = np.arange(49).reshape(7,7)

# Use fancy indexing to specify rows and columns desired.
ridx = [0, 2, 4]
cidx = [1, 3, 5]
result = a[ridx, cidx]
print('result.shape', result.shape)
print('result      ', result)
```

Fancy indexing (3/3)

You can combine fancy indexing with other techniques

- E.g. regular indexing, slicing, masking

```
import numpy as np

a = np.arange(49).reshape(7,7)

# Combine fancy indexing with regular indexing.
cidx = [1, 3, 5]
result1 = a[2, cidx]
print('result1.shape', result1.shape)
print('result1\n',     result1)

# Combine fancy indexing with slicing.
cidx = [1, 3, 5]
result2 = a[2:5, cidx]
print('\nresult2.shape', result2.shape)
print('result2\n',       result2)

# Combine fancy indexing with masking.
rmask = [True, True, False, False, False, False, True]
cidx = [1, 3, 5]
result3 = a[rmask, cidx]
print('\nresult3.shape', result3.shape)
print('result3\n',       result3)
```

Partitioning

You can partition an array via partition()

- You specify an index position, returns an array where all elements up to that position are smaller than all values after that position

Notes:

- Elements are unsorted within each partition
- For multidimensional arrays, the default axis of partitioning is 0
- There's also a partition() instance method, partitions in-place

```
1   import numpy as np
2
3   a = np.random.randint(0, 101, 12)
4   print('Unpartitioned 1D array', a)
5   print('Partitioned at index 2', np.partition(a, 2))
6   print('Partitioned at index 4', np.partition(a, 4))
7
8   b = np.random.randint(0, 101, 49).reshape((7,7))
9   print('\nUnpartitioned 2D array\n', b)
10  print('\nPartitioned at col index 2\n', b.partition(2,
11  print('\nPartitioned at col index 4\n', np.partition(b
12  print('\nPartitioned at row index 2\n', np.partition(b
13  print('\nPartitioned at row index 4\n', np.partition(b
```

Sorting

You can sort an array via sort() Returns a sorted array

```
import numpy as np

a = np.random.randint(0, 101, 12)
print('Unsorted 1D array', a)
print('Sorted           ', np.sort(a))

b = np.random.randint(0, 101, 49).reshape((7,7))
print('\nUnsorted 2D array\n', b)
print('\nSorted across cols\n', np.sort(b, axis=1))
print('\nSorted down rows  \n', np.sort(b, axis=0))
```

Notes:

- For multidimensional arrays, the default axis of partitioning is 0
- There’s also a sort() instance method, sorts in-place
- There’s also a sort() instance method, sorts in-place

Any questions?