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

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

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

Ex1a

```
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=10000000)

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

Execution time using a loop

Ex1b

```
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)
```

Execution time using a universal function

NumPy implements all the usual arithmetic operators as universal functions
- You can use an operator directly, or call the equivalent function

Ex2

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

Ex3

NumPy has universal functions for trig, log, etc.

Ex4

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

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 aggregation functions look very similar to functions available in the standard Python library
- But the NumPy functions are much quicker, so use them!

Ex5

There are lots of aggregation function - there are also NaN-friendly functions, eg. np.nansum().

Ex6

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

Ex7

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

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

Ex8a

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

Ex8b

Universal functions also work with different-shape arrays
- This is called broadcasting

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

Ex9

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

```
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)

```
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)
- b rows stretched to match a rows, so b becomes (3,3)

- Boolean operations
- Boolean aggregation
- Boolean masking

We've seen how to use math operators with NumPy arrays
- + - * / // etc.

Ex 11a

You can also use Boolean operators
- > >= < <= == !=
- Returns a NumPy array containing True/False in each position

You can combine Boolean operations together
- & and
- | inclusive or
- ^ exclusive or
- ~ not

Note the need for parentheses, for operator precedence

Ex 11b

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

Ex 12

You can use a Boolean result matrix as a mask
- array[booleanTest]
- Yields all array elements that have a True Boolean result

Ex 13

- Fancy indexing
- Partitioning
- Sorting

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

Ex14a

You can use fancy indexing with multidimensional arrays
- Specify a fancy index indicating desired rows
- Specify another fancy index indicating desired columns

Ex14b

You can combine fancy indexing with other techniques
- E.g. regular indexing, slicing, masking

Ex14c

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

Note:
- 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

Ex15

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

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

Ex16