Universl Function (UFUNCS):

The key to making it fast is to use vectorized operations, generally implemented through NumPy's universal functions (ufuncs).

Python's default implementation (known as CPython) does some operations very slowly. This is in part due to the dynamic, interpreted nature of the language: the fact that types are flexible, so that sequences of operations cannot be compiled down to efficient machine code as in languages like C

```
#Imagine we have an array of values and we'd like to compute the reciprocal
of each.
#A straightforward approach might look like this:

import numpy as np
np.random.seed(0)

def compute_reciprocals(values):
    output = np.empty(len(values))
    for i in range(len(values)):
        output[i] = 1.0/values[i]
    return output

values = np.random.randint(1, 10, size=5)
print(values)
compute_reciprocals(values)
[6 1 4 4 8]
array([0.16666667, 1.        , 0.25      , 0.25      , 0.125     ])
#lets measure the execution time

big_array = np.random.randint(1, 100, size=1000000)
%timeit compute_reciprocals(big_array)
1 loop, best of 3: 4.13 s per loop
#For many types of operations, NumPy provides a convenient interface into
just this kind of statically typed,
#compiled routine. This is known as a vectorized operation.
#This can be accomplished by simply performing an operation on the array,
#which will then be applied to each element.
#This vectorized approach is designed to push the loop into the compiled
layer that underlies NumPy,
#leading to much faster execution.

print(compute_reciprocals(values))
print(1.0 / values)
[ 0.16666667  1.          0.25        0.25        0.125     ]
[ 0.16666667  1.          0.25        0.25        0.125     ]
%timeit (1.0 / big_array)
100 loops, best of 3: 9.91 ms per loop
#Vectorized operations in NumPy are implemented via ufuncs, whose main
purpose is to quickly
#execute repeated operations on values in NumPy arrays.

#Ufuncs are extremely flexible – before we saw an operation between
#a scalar and an array, but we can also operate between two arrays:

np.arange(5) / np.arange(1, 6)
array([ 0.        ,  0.5       ,  0.66666667,  0.75      ,  0.8       ])
```

```
#And ufunc operations are not limited to one-dimensional arrays–they can
also act on multi-dimensional arrays as well:

x = np.arange(9).reshape((3, 3))
2 ** x
array([[  1,   2,   4],
       [  8,  16,  32],
       [ 64, 128, 256]], dtype=int32)
```

Computations using vectorization through ufuncs are nearly always more efficient than their counterpart implemented using Python loops, especially as the arrays grow in size. Any time you see such a loop in a Python script, you should consider whether it can be replaced with a vectorized expression.

Exploring NumPy's UFuncs

Ufuncs exist in two flavors: unary ufuncs, which operate on a single input, and binary ufuncs, which operate on two inputs.

```
#Array arithmetic

# The standard addition, subtraction, multiplication, and division can all
be used:
import numpy as np
x = np.arange(4)
print("x     =", x)
print("x + 5 =", x + 5)
print("x - 5 =", x - 5)
print("x * 2 =", x * 2)
print("x / 2 =", x / 2)
print("x // 2 =", x // 2)
x     = [0 1 2 3]
x + 5 = [5 6 7 8]
x - 5 = [-5 -4 -3 -2]
x * 2 = [0 2 4 6]
x / 2 = [ 0.   0.5  1.   1.5]
x // 2 = [0 0 1 1]
print(x)
print(sum(x,2))
print(np.add(x,2))
sum(np.add(x,2))
[0 1 2 3]
8
[2 3 4 5]
14
#There is also a unary ufunc for negation, and a ** operator for
exponentiation, and a % operator for modulus:

print("-x     = ", -x)
print("x ** 2 = ", x ** 2)
print("x % 2  = ", x % 2)
-x     =  [ 0 -1 -2 -3]
x ** 2 =  [0 1 4 9]
x % 2  =  [0 1 0 1]
#Each of these arithmetic operations are simply convenient wrappers around
specific functions built
#into NumPy; for example, the + operator is a wrapper for the add function:
```

```
np.add(x, 2)
array([2, 3, 4, 5])
```

The following table lists the arithmetic operators implemented in NumPy:

Operator Equivalent ufunc Description

+ np.add Addition (e.g., 1 + 1 = 2)

- np.subtract Subtraction (e.g., 3 - 2 = 1)

- np.negative Unary negation (e.g., -2)

np.multiply Multiplication (e.g., 2 3 = 6)

/ np.divide Division (e.g., 3 / 2 = 1.5)

// np.floor_divide Floor division (e.g., 3 // 2 = 1)

np.power Exponentiation (e.g., 2 3 = 8)

% np.mod Modulus/remainder (e.g., 9 % 4 = 1)

```
#Absolute value

x = np.array([-2, -1, 0, 1, 2])
print(abs(x))

#The corresponding NumPy ufunc is np.absolute, which is also available
under the alias np.abs:

print(np.absolute(x))
print(np.abs(x))
[2 1 0 1 2]
[2 1 0 1 2]
[2 1 0 1 2]
```

```
#Trigonometric functions

theta = np.linspace(0, np.pi, 3)

print("theta      = ", theta)
print("sin(theta) = ", np.sin(theta))
print("cos(theta) = ", np.cos(theta))
print("tan(theta) = ", np.tan(theta))
theta      =  [ 0.          1.57079633  3.14159265]
sin(theta) =  [  0.00000000e+00   1.00000000e+00   1.22464680e-16]
cos(theta) =  [  1.00000000e+00   6.12323400e-17  -1.00000000e+00]
tan(theta) =  [  0.00000000e+00   1.63312394e+16  -1.22464680e-16]
#Exponents and logarithms

x = [1, 2, 3]
print("x     =", x)
print("e^x   =", np.exp(x))
print("2^x   =", np.exp2(x))
print("3^x   =", np.power(3, x))
x     = [1, 2, 3]
e^x   = [  2.71828183   7.3890561   20.08553692]
2^x   = [ 2.  4.  8.]
3^x   = [ 3  9 27]
#The inverse of the exponentials, the logarithms, are also available.
#The basic np.log gives the natural logarithm;
#if you prefer to compute the base-2 logarithm or the base-10 logarithm,
these are available as well:

x = [1, 2, 4, 10]
print("x        =", x)
print("ln(x)    =", np.log(x))
print("log2(x)  =", np.log2(x))
print("log10(x) =", np.log10(x))
x        = [1, 2, 4, 10]
ln(x)    = [ 0.          0.69314718  1.38629436  2.30258509]
log2(x)  = [ 0.          1.          2.          3.32192809]
log10(x) = [ 0.          0.30103     0.60205999  1.        ]
#Advanced Ufunc Features

#Specifying output

#For large calculations, it is sometimes useful to be able to specify the
array where
#the result of the calculation will be stored. Rather than creating a
temporary array,
#this can be used to write computation results directly to the memory
location where
#you'd like them to be. For all ufuncs, this can be done using the "out"
argument of the function:

x = np.arange(5)
y = np.empty(5)
np.multiply(x, 10, out=y)
#y = np.multiply(x, 10)

print(y)
[  0.  10.  20.  30.  40.]
#However, because it executes the operation in compiled code, NumPy's
version of the
#operation is computed much more quickly:
```

```
big_array = np.random.rand(1000000)
%timeit sum(big_array)
%timeit np.sum(big_array)
1 loop, best of 3: 170 ms per loop
100 loops, best of 3: 2.43 ms per loop
#Minimum and Maximum

#Python has built-in min and max functions, used to find the minimum value
and maximum value of any given array:

min(big_array), max(big_array)
(7.0712031718933588e-07, 0.99999972076563337)
#NumPy's corresponding functions have similar syntax, and again operate
much more quickly:

np.min(big_array), np.max(big_array)
(7.0712031718933588e-07, 0.99999972076563337)
#For min, max, sum, and several other NumPy aggregates, we use methods of
the array object itself:
print(big_array.min(), big_array.max(), big_array.sum())
7.07120317189e-07 0.999999720766 500213.239662
#Multi dimensional aggregates

#M = np.random.random((3, 4))
M = np.array([[1,2,3],[2,3,4]])
print(M)
[[1 2 3]
 [2 3 4]]
array([6, 9])
#By default, each NumPy aggregation function will return the aggregate over
the entire array:

print("Aggregate over entire Array :",np.sum(M))

#Aggregation functions take an additional argument specifying the axis
along which the aggregate is computed


print("Aggregate over First Axis  :",np.sum(M,axis = 0))
print("Aggregate over Second Axis :",np.sum(M,axis = 1))
Aggregate over entire Array : 7.54975676611
Aggregate over First Axis  : [ 1.71071124  2.73806914  1.72117537
1.37980101]
Aggregate over Second Axis : [ 2.90456177  1.97233402  2.67286098]
```

Computation on Arrays: Broadcasting

Another means of vectorizing operations is to use NumPy's broadcasting functionality. Broadcasting is simply a set of rules for applying binary ufuncs (e.g., addition, subtraction, multiplication, etc.) on arrays of different sizes.

```
#arrays of the same size, binary operations are performed on an element-by-
element basis:
a = np.array([0, 1, 2])
b = np.array([5, 5, 5])
a + b
array([5, 6, 7])
```

```
#Broadcasting allows these types of binary operations to be performed on
arrays of
#different sizes–for example, we can just as easily add a scalar
#(think of it as a zero-dimensional array) to an array:
a + 5
array([5, 6, 7])
#We can similarly extend this to arrays of higher dimension. Observe the
result when we
#add a one-dimensional array to a two-dimensional array:
import numpy as np
M = np.ones((3, 3))
a = np.array([0, 1,1])
#a1 = np.array([[0, 1,1],[0, 1,1]])
print("Two Dimensional Array :\n",M)
print("One Dimensional Array :\n",a)
print("Broadcasting :\n", M + a)
#print("Broadcasting :\n", M + a1)
Two Dimensional Array :
 [[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
One Dimensional Array :
 [0 1 1]
Broadcasting :
 [[1. 2. 2.]
 [1. 2. 2.]
 [1. 2. 2.]]
a = np.arange(3)
b = np.arange(3).reshape(3,1)

print(a.ndim)
print(b.ndim)
print(a.shape)
print(b.shape)
print(a)
print(b)
print(a + b)
1
4
(3,)
(3, 1, 4, 5)
[0 1 2]
[[[[ 0  1  2  3  4]
   [ 5  6  7  8  9]
   [10 11 12 13 14]
   [15 16 17 18 19]]]


 [[[20 21 22 23 24]
   [25 26 27 28 29]
   [30 31 32 33 34]
   [35 36 37 38 39]]]


 [[[40 41 42 43 44]
   [45 46 47 48 49]
   [50 51 52 53 54]
   [55 56 57 58 59]]]]
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-7-07988c1b3540> in <module>()
```

```
      8 print(a)
      9 print(b)
---> 10 print(a + b)

ValueError: operands could not be broadcast together with shapes (3,)
(3,1,4,5)
```

Rules of Broadcasting

Broadcasting in NumPy follows a strict set of rules to determine the interaction between the two arrays:

1. If the two arrays differ in their number of dimensions, the shape of the one with fewer dimensions is padded with ones on its leading (left) side.

2. If the shape of the two arrays does not match in any dimension, the array with shape equal to 1 in that dimension is stretched to match the other shape.

3. If in any dimension the sizes disagree and neither is equal to 1, an error is raised.

```
M = np.ones((2, 3))
a = np.arange(3)

print("M Shape :",M.shape)
print("a Shape :",a.shape)

# by rule 1 that the array a has fewer dimensions, so we pad it on the left
with ones:

#M.shape -> (2, 3)
#a.shape -> (1, 3)

#By rule 2, we now see that the first dimension disagrees, so we stretch
this dimension to match:

#M.shape -> (2, 3)
#a.shape -> (2, 3)
```

```
#The shapes match, and we see that the final shape will be (2, 3):

print("M:\n",M)
print("a:\n",a)
print("a + M:\n", a+M)
M Shape : (2, 3)
a Shape : (3,)
M:
 [[ 1.  1.  1.]
 [ 1.  1.  1.]]
a:
 [0 1 2]
a + M:
 [[ 1.  2.  3.]
 [ 1.  2.  3.]]
#Let's take a look at an example where both arrays need to be broadcast:

a = np.arange(3).reshape((3, 1))
b = np.arange(3)

#Again, we'll start by writing out the shape of the arrays:

#a.shape = (3, 1)
#b.shape = (3,)

#Rule 1 says we must pad the shape of b with ones:

#a.shape -> (3, 1)
#b.shape -> (1, 3)

#And rule 2 tells us that we upgrade each of these ones to match the
corresponding size of the other array:

#a.shape -> (3, 3)
#b.shape -> (3, 3)
#Because the result matches, these shapes are compatible. We can see this
here:
print("a:\n",a)
print("b:\n",b)
print("a+b:\n",a+b)
a:
 [[0]
 [1]
 [2]]
b:
 [0 1 2]
a+b:
 [[0 1 2]
 [1 2 3]
 [2 3 4]]
a = np.arange(12).reshape((2, 6))
b = np.arange(6).reshape((2, 3))
print(a)
print(b)
a* b
[[ 0  1  2  3  4  5]
 [ 6  7  8  9 10 11]]
[[0 1 2]
 [3 4 5]]
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
```

```
<ipython-input-4-7aec35ddaada> in <module>()
      3 print(a)
      4 print(b)
----> 5 a* b

ValueError: operands could not be broadcast together with shapes (2,6)
(2,3)
#Now let's take a look at an example in which the two arrays are not
compatible:

M = np.ones((3, 2))
a = np.arange(3)

#This is just a slightly different situation than in the first example:
#the matrix M is transposed. How does this affect the calculation? The
shape of the arrays are

#M.shape = (3, 2)
#a.shape = (3,)
#Again, rule 1 tells us that we must pad the shape of a with ones:

#M.shape -> (3, 2)
#a.shape -> (1, 3)

#By rule 2, the first dimension of a is stretched to match that of M:

#M.shape -> (3, 2)
#a.shape -> (3, 3)

#Now we hit rule 3–the final shapes do not match, so these two arrays are
incompatible,
#as we can observe by attempting this operation:

M+ a
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-47-14d102482917> in <module>()
     22 #as we can observe by attempting this operation:
     23
---> 24 M+ a

ValueError: operands could not be broadcast together with shapes (3,2) (3,)
#Working with Boolean Array
import numpy as np
rng = np.random.RandomState(0)
x = rng.randint(10, size=(3, 4))

print("Original Array :\n",x)

print("How many values less than 6 :\n",np.count_nonzero(x < 6))

#Another Way
print("How many values less than 6 :\n",np.sum(x < 6))

print("How many values less than 6 in each row:\n",
      np.sum(x < 6,axis = 0))
Original Array :
 [[5 0 3 3]
 [7 9 3 5]
 [2 4 7 6]]
How many values less than 6 :
```

```
 8
How many values less than 6 :
 8
How many values less than 6 in each row:
 [2 2 2 2]
x < 6
array([[ True,  True,  True,  True],
       [False, False,  True,  True],
       [ True,  True, False, False]])
#If we're interested in quickly checking whether any or all the values are
true,
#we can use "np.any" or "np.all":

print("Are there any value greater than 8? :\n",np.any(x > 8))
print("Are all values less than 10? :\n",np.all(x < 10))
print("Are all values greater than 10? :\n",np.all(x > 10))
Are there any value greater than 8? :
 True
Are all values less than 10? :
 True
Are all values greater than 10? :
 False
#np.all and np.any can be used along particular axes as well. For example:
print(x)
print("Are all values in each row less than 8?",np.all(x < 8,
                                                       axis=0))
[[5 0 3 3]
 [7 9 3 5]
 [2 4 7 6]]
Are all values in each row less than 8? [ True False  True  True]
```

Advance Array Manipulation

While much of the heavy lifting for data analysis applications is handled by higher level functions in pandas, you may at some point need to write a data algorithm that is not found in one of the existing libraries.

1. Reshaping Arrays

Given what we know about NumPy arrays, it should come as little surprise that you can convert an array from one shape to another without copying any data. To do this, pass a tuple indicating the new shape to the reshape array instance method. For example, suppose we had a one-dimensional array of values that we wished to rearrange into a matrix:

```
arr = np.arange(8)
print("Array is: ",arr)
print("Dimension of Array is: ", arr.ndim)

#Reshape the array
print("Reshaped Array: \n",arr.reshape((4, 2)))


##A multidimensional array can also be reshaped:

print("Reshaped multidimensional Array: \n",
      arr.reshape((4, 2)).reshape(2,4))
Array is:  [0 1 2 3 4 5 6 7]
```

```
Dimension of Array is:  1
Reshaped Array:
 [[0 1]
 [2 3]
 [4 5]
 [6 7]]
Reshaped multidimensional Array:
 [[0 1 2 3]
 [4 5 6 7]]
 One of the passed shape dimensions can be -1, in which case the value used
for that dimension will be inferred from the data:
import numpy as np
arr = np.arange(15)

print("One Dimensional array: \n",arr)

print("5 rows: \n",arr.reshape((5, -1)))

print("3 cols: \n",arr.reshape((-1, 3)))
One Dimensional array:
 [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14]
5 rows:
 [[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]
 [12 13 14]]
3 cols:
 [[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]
 [12 13 14]]
```

The opposite operation of reshape from one-dimensional to a higher dimension is typically known as flattening or raveling:

```
arr = np.arange(15).reshape((5, 3))

print("Two Dimensional Array :\n",arr)

print("Ravelling Array :\n",arr.ravel())

#ravel does not produce a copy of the underlying data if it does not have
#to. The flatten
#method behaves like ravel except it always returns a copy of the data:

print("Flatten Array :\n",arr.flatten())
Two Dimensional Array :
 [[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]
 [12 13 14]]
Ravelling Array :
 [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14]
Flatten Array :
 [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14]
```

2.Concatinating:

numpy.concatenate takes a sequence (tuple, list, etc.) of arrays and joins them together in order along the input axis.

```
arr1 = np.array([[1, 2, 3], [4, 5, 6]])
arr2 = np.array([[7, 8, 9], [10, 11, 12]])

print("Array1 :\n",arr1)
print("Array2 :\n",arr2)
print("Concatinated Array ROw wise: \n",
      np.concatenate([arr1, arr2], axis=0))
print("Concatinated Array Column wise: \n",
      np.concatenate([arr1, arr2],axis=1))
```

```
Array1 :
 [[1 2 3]
 [4 5 6]]
Array2 :
 [[ 7  8  9]
 [10 11 12]]
Concatinated Array ROw wise:
 [[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
Concatinated Array Column wise:
 [[ 1  2  3  7  8  9]
 [ 4  5  6 10 11 12]]
#There are some convenience functions, like vstack and hstack, for common
kinds of concatenation.
#The above operations could have been expressed as:
```

```
print("Vertical Stack :\n",np.vstack((arr1, arr2)))
print("Horizontal Stack :\n",np.hstack((arr1, arr2)))
```

```
Vertical Stack :
 [[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
Horizontal Stack :
 [[ 1  2  3  7  8  9]
 [ 4  5  6 10 11 12]]
```

Fancy Indexing

Fancy indexing is like the simple indexing we've already seen, but we pass arrays of indices in place of single scalars. This allows us to very quickly access and modify complicated subsets of an array's values.

```
rand = np.random.RandomState(42)

x = rand.randint(100, size=10)
print(x)
```

```
[51 92 14 71 60 20 82 86 74 74]
```

```
[x[3], x[7], x[2]]
```

```
[71, 86, 14]
```

```
#Alternatively, we can pass a single list or array of indices to obtain the
same result:

ind = [3, 7, 4]
x[ind]
array([71, 86, 60])
#When using fancy indexing, the shape of the result reflects the shape of
the index arrays
#rather than the shape of the array being indexed:

ind = np.array([[3, 7],
                [4, 5]])
x[ind]
array([[71, 86],
       [60, 20]])
#Fancy indexing also works in multiple dimensions. Consider the following
array:

X = np.arange(12).reshape((3, 4))
X
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
row = np.array([0, 1, 2])
col = np.array([2, 1, 3])
X[row, col]
array([ 2,  5, 11])
```

NumPy's Structured Arrays

We have seen homogeneous array so far.. It is possible to create structured array using Numpy, which provides efficient storage for hetrogeneous data.

Imagine that we have several categories of data on a number of people (say, name, age, and weight), and we'd like to store these values for use in a Python program. It would be possible to store these in three separate arrays:

```
import numpy as np

name = ['Alice', 'Bob', 'Cathy', 'Doug']
age = [25, 45, 37, 19]
weight = [55.0, 85.5, 68.0, 61.5]
```

There's nothing here that tells us that the three arrays are related; it would be more natural if we could use a single structure to store all of this data. NumPy can handle this through structured arrays, which are arrays with compound data types.

```
#Create a simple array
x = np.zeros(4, dtype=int)

#Create a compound Array (structured arrays)

data = np.zeros(4, dtype={'names':('name', 'age', 'weight'),
                          'formats':('U10', 'i4', 'f8')})
print(data.dtype)
[('name', '<U10'), ('age', '<i4'), ('weight', '<f8')]
```

Here 'U10' translates to "Unicode string of maximum length 10," 'i4' translates to "4-byte (i.e., 32 bit) integer," and 'f8' translates to "8-byte (i.e., 64 bit) float.

```
#Now that we've created an empty container array, we can fill the array
with our lists of values:

data['name'] = name
data['age'] = age
data['weight'] = weight
print(data)
print(type(data))
[('Alice', 25,  55. ) ('Bob', 45,  85.5) ('Cathy', 37,  68. )
 ('Doug', 19,  61.5)]
<class 'numpy.ndarray'>
# Get all names
data['name']
array(['Alice', 'Bob', 'Cathy', 'Doug'],
      dtype='<U10')
# Get first row of data
data[0]
('Alice', 25,  55.)
# Get the name from the last row
data[-1]['name']
'Doug'
# Get names where age is under 30
data[data['age'] < 30]['name']
array(['Alice', 'Doug'],
      dtype='<U10')
```

Pandas provides a Dataframe object, which is a structure built on NumPy arrays that offers a variety of useful data manipulation functionality