Computation on NumPy Arrays: Universal Functions

Why NumPy is so **important** in the Python data science world?

Namely, because it provides an **easy and flexible interface** to **optimize computation** with arrays of **data.**

Computation on NumPy arrays can be **very fast,** or it can be **very slow.**

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

This chapter **motivates** the need for NumPy's ufuncs, which can be used to make **repeated calculations** on **array elements much more efficient.**

It then **introduces** many of the most common and **useful arithmetic ufuncs** available in the NumPy package.

The Slowness of Loops

Python's **default implementation** (known as CPython) does **some operations very slowly.**

This is partly **due to** the **dynamic, interpreted nature** of the language:

Types are flexible, so sequences of operations **cannot be compiled** down to **efficient machine code** as in languages **like C and Fortran.**

Recently there have been **various attempts to address this weakness:**

- well-known examples are the **PyPy** project, a just-in-time compiled implementation of Python;
- the **Cython** project, which converts Python code to compilable C code;
- and the **Numba** project, which converts snippets of Python code to fast LLVM bytecode.

Each of these **approaches** has its **strengths** and **weaknesses.**

But it is safe to say that none of the three approaches has yet surpassed the reach and **popularity** of the standard **CPython engine.**

The relative **sluggishness of Python** generally **manifests** itself in situations where many small **operations are being repeated.**

In particular, looping over arrays to operate on each element.

For example, imagine we have an array of values and we'd like to compute the reciprocal of each.

A **straightforward approach** might look like this:

In [15]:
```
import numpy as np
rng = np.random.default_rng(seed=1701)

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

values = rng.integers(1, 10, size=5)
compute_reciprocals(values)
```

Out[15]:
```
array([0.11111111, 0.25      , 1.        , 0.33333333, 0.125
])
```

This **implementation** probably feels **fairly natural** to someone from, say, a **C or Java** background.

But if we measure the **execution time** of this code for a **large input,** we see that this operation is **very slow** — perhaps surprisingly so!

We'll **benchmark** this with IPython's `%timeit` magic (discussed in **Profiling and Timing Code**):

In [17]:
```
big_array = rng.integers(1, 100, size=1000000)
%timeit compute_reciprocals(big_array)
```

```
2.13 s ± 50.6 ms per loop (mean ± std. dev. of 7 runs, 1 loop ea
ch)
```

It takes **several seconds** to compute these million operations and to store the result!

This is almost **absurdly slow.**

The bottleneck here is **not the operations themselves;**

but the type **checking and function dispatches** that CPython must do at **each cycle** of the loop.

> ***Note:*** Dynamic dispatch is the process of selecting which implementation of a polymorphic operation (method or function) to call at run time.

Each time the reciprocal is computed, Python first **examines the object's type** and does a dynamic **lookup of the correct function** to use for that type.

If we were working in **compiled code instead,** this **type** specification would be **known before the code executed** and the result could be computed **much more efficiently.**

Introducing Ufuncs

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.**

For **simple operations** like the element-wise division here, **vectorization is as simple as** using **Python arithmetic operators directly on the array object.**

This **vectorized approach** is designed to push the loop into the **compiled layer** that underlies NumPy, leading to much faster execution.

This **approach** relies upon **SIMD** (Single Instruction/Multiple Data) architecture.

> ***Important:*** Refer to the **Vectorized Computation** folder in the course materials.

Compare the results of the following **two operations:**

In [24]:
```
print(compute_reciprocals(values))
print(1.0 / values)
```

```
[0.11111111 0.25       1.         0.33333333 0.125     ]
[0.11111111 0.25       1.         0.33333333 0.125     ]
```

Looking at the **execution time** for our **big array,** we see that it completes **orders of magnitude faster** than the Python loop:

In [28]:
```
%timeit (1.0 / big_array)
```

```
3.75 ms ± 232 µs per loop (mean ± std. dev. of 7 runs, 100 loops
each)
```

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:**

- We saw an **operation** between **a scalar and an array.**
- We can also operate between **two arrays.**

In [ ]:
```
np.arange(5)
```

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

In [ ]:
```
np.arange(1, 6)
```

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

In [ ]:
```
np.arange(5) / np.arange(1, 6)
```

Out[ ]:
```
array([0.        , 0.5       , 0.66666667, 0.75      , 0.8
])
```

And **ufunc** operations are **not limited** to **one-dimensional** arrays. They can act on **multidimensional** arrays as well:

In [ ]:
```
x = np.arange(9).reshape((3, 3))
x
```

Out[ ]:
```
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

In [ ]:
```
2 ** x
```

Out[ ]:
```
array([[  1,   2,   4],
       [  8,  16,  32],
       [ 64, 128, 256]])
```

Computations using vectorization through ufuncs are **nearly always more efficient** than their counterparts implemented using **Python loops.**

Especially as the **arrays grow in size.**

Any time you see such a **loop in a NumPy 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.**
- **Binary ufuncs**, which operate on **two inputs.**

We'll see **examples of both** these types of functions here.

Array Arithmetic

NumPy's **ufuncs feel very natural** to use because they make use of Python's **native arithmetic operators.**

The **standard addition, subtraction, multiplication, and division** can all be used:

In [ ]:
```
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)  # floor division
```

```
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]
```

There is also a **unary ufunc** for **negation**, a `**` operator for **exponentiation**, and a `%` operator for **modulus:**

In [ ]:
```python
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]
```

In addition, these can be **strung together however you wish,** and the standard **order of operations is respected:**

In [ ]:
```python
-(0.5*x + 1) ** 2
```

Out[ ]:
```
array([-1.  , -2.25, -4.  , -6.25])
```

All of these **arithmetic operations** are simply **convenient wrappers** around **specific ufuncs built into NumPy.**

For example, the `+` operator is a wrapper for the `add` ufunc:

In [ ]:
```
np.add(x, 2)
```

Out[ ]:
```
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`)

Operator	Equivalent ufunc	Description
`%`	`np.mod`	Modulus/remainder (e.g., `9 % 4 = 1`)

Additionally, there are **Boolean/bitwise operators;** we will **explore** these in **Comparisons, Masks, and Boolean Logic.**

Absolute Value

Just as NumPy understands Python's built-in arithmetic operators, it also understands Python's **built-in absolute value function:**

In [ ]:
```
x = np.array([-2, -1, 0, 1, 2])
abs(x)
```

Out[ ]:
```
array([2, 1, 0, 1, 2])
```

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

In [ ]:
```
np.absolute(x)
```

Out[ ]:
```
array([2, 1, 0, 1, 2])
```

In [ ]:
```
np.abs(x)
```

Out[ ]:
```
array([2, 1, 0, 1, 2])
```

This ufunc can also handle **complex data,** in which case it returns the **magnitude**:

In [ ]:
```
x = np.array([3 - 4j, 4 - 3j, 2 + 0j, 0 + 1j])
np.abs(x)
```

Out[ ]:
```
array([5., 5., 2., 1.])
```

Trigonometric Functions

NumPy provides a **large number of useful ufuncs.**

Some of the most useful for the data scientist are the **trigonometric functions.**

We'll start by defining an **array of angles:**

In [ ]:
```
theta = np.linspace(0, np.pi, 3)
```

Now we can **compute some trigonometric functions** on these values:

In [ ]:
```
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) =  [0.0000000e+00 1.0000000e+00 1.2246468e-16]
cos(theta) =  [ 1.000000e+00  6.123234e-17 -1.000000e+00]
tan(theta) =  [ 0.00000000e+00  1.63312394e+16 -1.22464680e-16]
```

Inverse trigonometric functions are also available:

In [35]:
```python
x = [-1, 0, 1]
print("x         = ", x)
print("arcsin(x) = ", np.arcsin(x))
print("arccos(x) = ", np.arccos(x))
print("arctan(x) = ", np.arctan(x))
```

```
x         =  [-1, 0, 1]
arcsin(x) =  [-1.57079633  0.          1.57079633]
arccos(x) =  [3.14159265 1.57079633 0.        ]
arctan(x) =  [-0.78539816  0.          0.78539816]
```

Exponents and Logarithms

Other common operations available in NumPy ufuncs are the **exponentials:**

In [ ]:
```
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:

In [ ]:
```
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.        ]
```

There are also some **specialized versions** that are useful for **maintaining precision with very small input:**

In [ ]:
```
x = [0, 0.001, 0.01, 0.1]
print("exp(x) - 1 =", np.expm1(x))
print("log(1 + x) =", np.log1p(x))
```

```
exp(x) - 1 = [0.         0.0010005  0.01005017 0.10517092]
log(1 + x) = [0.         0.0009995  0.00995033 0.09531018]
```

When `x` **is very small,** these functions give **more precise** values than if the raw `np.log` or `np.exp` were to be used.

Specialized Ufuncs

NumPy has many **more ufuncs available,** including:

- Hyperbolic trigonometry
- Bitwise arithmetic
- Comparison operations
- Conversions from radians to degrees
- Rounding and remainders
- And much more

A look through the **NumPy documentation** reveals a lot of **interesting functionality.**

Another **excellent source for more specialized ufuncs** is the **submodule** `scipy.special`.

If you want to compute some **obscure mathematical function** on your data, chances are it is **implemented in** `scipy.special`.

There are **far too many functions** to list them all, but the **following snippet** shows a couple that might come up in a **statistics context:**

In [ ]:

```
from scipy import special
```

In [ ]:

```
# Gamma functions (generalized factorials) and related functior
x = [1, 5, 10]
print("gamma(x)     =", special.gamma(x))
print("ln|gamma(x)| =", special.gammaln(x))
print("beta(x, 2)   =", special.beta(x, 2))
```

```
gamma(x)     = [1.0000e+00 2.4000e+01 3.6288e+05]
ln|gamma(x)| = [ 0.          3.17805383 12.80182748]
beta(x, 2)   = [0.5        0.03333333 0.00909091]
```

In [ ]:
```
# Error function (integral of Gaussian),
# its complement, and its inverse
x = np.array([0, 0.3, 0.7, 1.0])
print("erf(x)  =", special.erf(x))
print("erfc(x) =", special.erfc(x))
print("erfinv(x) =", special.erfinv(x))
```

```
erf(x)  = [0.         0.32862676 0.67780119 0.84270079]
erfc(x) = [1.         0.67137324 0.32219881 0.15729921]
erfinv(x) = [0.          0.27246271 0.73286908        inf]
```

There are **many, many more ufuncs available** in both NumPy and `scipy.special`.

Because the documentation of these packages is available online, a **web search along the lines of "gamma function python"** will generally find the relevant information.

Advanced Ufunc Features

Many NumPy users make use of ufuncs **without ever learning their full set of features.**

A few specialized features of ufuncs:

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.**

For all ufuncs, this can be done using the `out` **argument** of the function:

In [ ]:
```
x = np.arange(5)
x
```

Out[ ]: array([0, 1, 2, 3, 4])

In [ ]:
```
y = np.empty(5)
y
```

Out[ ]: array([0.0e+000, 4.9e-324, 9.9e-324, 1.5e-323, 2.0e-323])

In [ ]:
```
np.multiply(x, 10, out=y)
print(y)
```

```
[ 0. 10. 20. 30. 40.]
```

This can even be **used with array views.**

For example, we can **write the results** of a computation **to every other element of a specified array:**

In [ ]:
```
y = np.zeros(10)
np.power(2, x, out=y[::2])
print(y)
```

```
[ 1.  0.  2.  0.  4.  0.  8.  0. 16.  0.]
```

If we had instead written `y[::2] = 2 ** x`

This would have resulted in the **creation of a temporary array** to hold the results of `2 ** x`, followed by a second operation copying those values into the `y` array.

This doesn't make much of a difference for such a small computation, but for **very large arrays** the **memory savings** from **careful use of the** `out` argument can be significant.

Aggregations

For **binary ufuncs, aggregations** can be computed **directly** from the object.

For example, if we'd like to **reduce** an array with a particular operation, we can use the `reduce` method of any ufunc.

A reduce repeatedly applies a given operation to the elements of an array until only a single result remains.

For example, calling `reduce` on the `add` ufunc returns the sum of all elements in the array:

```
In [ ]: x = np.arange(1, 6)
        np.add.reduce(x)
```

```
Out[ ]: 15
```

Similarly, calling `reduce` on the `multiply` ufunc results in the product of all array elements:

```
In [ ]: np.multiply.reduce(x)
```

```
Out[ ]: 120
```

> ***Question***: How much vectorized operations `add.reduce(x)` embrace compared to `x+x`?

If we'd like to **store all the intermediate results** of the computation, we can instead use **`accumulate`**:

In [ ]:
```
np.add.accumulate(x)
```

Out[ ]:
```
array([ 1,  3,  6, 10, 15])
```

In [ ]:
```
np.multiply.accumulate(x)
```

Out[ ]:
```
array([  1,   2,   6,  24, 120])
```

Note that for these particular cases, there are **dedicated NumPy functions** to compute the results (`np.sum` , `np.prod` , `np.cumsum` , `np.cumprod`), which we'll explore in **Aggregations: Min, Max, and Everything In Between.**

Outer Products

Finally, **any ufunc** can compute the **output of all pairs of two different inputs** using the `outer` **method.**

This allows you, **in one line,** to do **things like create a multiplication table:**

In [ ]:
```
x = np.arange(1, 6)
x
```

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

In [ ]:
```
np.multiply.outer(x, x)
```

```
Out[ ]: array([[ 1,  2,  3,  4,  5],
               [ 2,  4,  6,  8, 10],
               [ 3,  6,  9, 12, 15],
               [ 4,  8, 12, 16, 20],
               [ 5, 10, 15, 20, 25]])
```

The `ufunc.at` and `ufunc.reduceat` methods are **useful** as well, and we will explore them in **Fancy Indexing.**

We will also encounter the ability of **ufuncs** to operate between arrays of **different shapes and sizes,** a set of operations known as **broadcasting.**

This subject is important enough that we will devote a whole chapter to it: **Computation on Arrays: Broadcasting**.