Computation on NumPy Arrays: Universal Functions

NumPy数组运算：通用函数

> Up until now, we have been discussing some of the basic nuts and bolts of NumPy; in the next few sections, we will dive into the reasons that NumPy is so important in the Python data science world. Namely, it provides an easy and flexible interface to optimized computation with arrays of data.

直到目前为止，我们已经讨论了一些NumPy的基本构件；在下面几个小节中，我们会深入讨论NumPy能在Python数据科学中占据重要地位的原因。简而言之，NumPy提供了简单和灵活的接口来对数组数据计算进行优化。

> 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 section 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.

对NumPy的数组进行计算相较其他普通的实现方式而言是非常快的。快的原因关键在于使用了*向量化*的操作，因为它们都是通过NumPy的*通用函数*（ufuncs）实现的。希望通过本节的介绍，能让读者习惯使用ufuncs，它们能使在数组元素上的重复计算更加快速和高效。本节还会介绍多NumPy中最常用的ufuncs数学计算方法。

The Slowness of Loops

循环，慢的实现

> 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 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 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.

Python的默认实现（被称为CPython）对于一些操作执行效率很低。这部分归咎于语言本身的动态和解释执行特性：因为类型是动态的，因此不到执行时，无法预知变量的类型，因此不能像C或者Fortran那样预先将代码编译成机器代码来执行。近年来，也出现了很多尝试来弥补这个缺陷：其中比较流行和著名的包括PyPy，Python的JIT编译实现；Cython，可以将Python代码转换为可编译的C代码；和Numba，可以将Python代码片段转换为LLVM字节码。

> The relative sluggishness of Python generally manifests itself in situations where many small operations are being repeated – for instance 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:

Python另一个表现相对低效的方面是当重复进行很多细微操作时，比方说对一个数组中的每个元素进行循环操作。例如，我们有一个数组，现在我们需要计算每个元素的倒数。一个很直接的实现方式就像下面的代码：

In [1]:
```python
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)
compute_reciprocals(values)
```

Out[1]:
```
array([0.16666667, 1.        , 0.25      , 0.25      , 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):

上面的代码实现对于很多具有C或者Java语言背景的读者来说是非常自然的。但是如果我们在一个很大的数据集上测量上面代码的执行时间，我们会发现这个操作很慢，甚至慢的让你吃惊。下面使用`%timeit` 魔术指令（参见性能测算和计时）对一个大数据集进行测时：

In [2]:
```python
big_array = np.random.randint(1, 100, size=1000000)
%timeit compute_reciprocals(big_array)
```

```
4.07 s ± 68.8 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

> It takes several seconds to compute these million operations and to store the result! When even cell phones have processing speeds measured in Giga-FLOPS (i.e., billions of numerical operations per second), this seems almost absurdly slow. It turns out that 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. 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 executes and the result could be computed much more efficiently.

这个操作对于百万级的数据集耗时需要几秒。当现在手机的每秒浮点数运算次数都已经已经达到10亿级别，这实在是不可思议的慢了。通过分析发现瓶颈并不是代码本身，而是每次循环时CPython必须执行的类型检查和函数匹配。每次计算倒数时，Python首先需要检查对象的类型，然后再找一个最合适的函数用于这种类型进行计算。如果我们使用编译型的语言实现上面的代码，每次计算的时候，类型和应该执行的函数都已经确定，因此执行的时间肯定短很多。

Introducing UFuncs

UFuncs介绍

> 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.

对于许多操作，NumPy都为这种静态类型提供了编译好的函数。被称为*向量化*的操作。向量化操作可以简单应用在数组上，实际上会应用在每一个元素上。实现原理就是将循环的部分放进NumPy编译后的那个层次，从而提高性能。

> Compare the results of the following two:

比较一下下述两种方式得到的结果：

In [3]:
```python
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     ]
```

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

下面使用ufunc来测算执行时间，我们可以看到执行时间相差了好几个数量级：

In [4]:
```python
%timeit (1.0 / big_array)
```

```
1.53 ms ± 24.3 µs per loop (mean ± std. dev. of 7 runs, 1000 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 – before we saw an operation between a scalar and an array, but we can also operate between two arrays:

NumPy中的向量化操作通过*ufuncs*实现的，其主要目的就是在NumPy数组中快速执行重复的元素操作。Ufuncs是极端灵活的，我们上面看到是标量和数组间的操作，但是我们也可以将它们用在两个数组之间：

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

Out[5]:
```
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:

而且ufuncs也不仅限于一维数组，多维数组同样适用：

In [6]:
```python
x = np.arange(9).reshape((3, 3))
2 ** x
```

Out[6]:
```
array([[  1,   2,   4],
       [  8,  16,  32],
       [ 64, 128, 256]])
```

> 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.

通过ufuncs向量化计算基本上都会比使用Python循环实现的相同方法要更加高效，特别是数组的长度增长的情况下。任何情况下，如果你看到Python的数组循环操作，都可以替换成为向量化形式。

Exploring NumPy's UFuncs

NumPy的UFuncs

> Ufuncs exist in two flavors: *unary ufuncs*, which operate on a single input, and *binary ufuncs*, which operate on two inputs. We'll see examples of both these types of functions here.

Ufuncs有两种类型：一元*ufuncs*（仅对一个输入值进行操作）和二元*ufuncs*（对两个输入值进行操作）。下面我们会看到它们的使用例子。

Array arithmetic

数组运算

> 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:

NumPy的ufuncs用起来非常的自然和人性化，因为它们采用了Python本身的算术运算符号 - 标准的加法、剪发、乘法和除法实现：

In [7]:
```python
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)  # 整除
```

```
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, and a `**` operator for exponentiation, and a `%` operator for modulus:

下面是一元的取反，`**` 求幂和 % 取模：

In [8]:
```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:

当然，你可以将这些运算按照你的需要组合起来，运算顺序与标准运算一致：

In [9]:
```python
-(0.5*x + 1) ** 2
```

Out[9]:
```
array([-1.  , -2.25, -4.  , -6.25])
```

> 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:

上面看到的这些算术运算操作，都是NumPy中相应函数的简化写法；例如 `+` 号实际上是 `add` 函数的封装：

In [10]:
```python
np.add(x, 2)
```

Out[10]:
```
array([2, 3, 4, 5])
```

> he following table lists the arithmetic operators implemented in NumPy:

下表列出NumPy实现的运算符号及对应的ufunc函数:

运算符	对应的ufunc函数	说明
+	np.add	加法 (例如 1 + 1 = 2)
-	np.subtract	减法 (例如 3 - 2 = 1)
-	np.negative	一元取负 (例如 -2)
*	np.multiply	乘法 (例如 2 * 3 = 6)
/	np.divide	除法 (例如 3 / 2 = 1.5)
//	np.floor_divide	整除 (例如 3 // 2 = 1)
**	np.power	求幂 (例如 2 ** 3 = 8)
%	np.mod	模除 (例如 9 % 4 = 1)

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

除此之外还有布尔和二进制位操作；我们会在比较，遮盖和布尔逻辑中介绍它们。

Absolute value

绝对值

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

就像NumPy能够理解Python内建的算术操作一样，它同样能理解Python内建的绝对值函数：

In [11]:
```python
x = np.array([-2, -1, 0, 1, 2])
abs(x)
```

Out[11]:
```
array([2, 1, 0, 1, 2])
```

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

对应的NumPy的ufunc是`np.absolute`，还有一个简短的别名 `np.abs`：

In [12]:
```python
np.absolute(x)
```

Out[12]:
```
array([2, 1, 0, 1, 2])
```

In [13]:
```python
np.abs(x)
```

Out[13]:
```
array([2, 1, 0, 1, 2])
```

> This ufunc can also handle complex data, in which the absolute value returns the magnitude:

这个ufunc可以处理复数，返回的是矢量的长度：

In [14]:
```python
x = np.array([3 - 4j, 4 - 3j, 2 + 0j, 0 + 1j])
np.abs(x)
```

Out[14]:
```
array([5., 5., 2., 1.])
```

Trigonometric functions

三角函数

> NumPy provides a large number of useful ufuncs, and some of the most useful for the data scientist are the trigonometric functions. We'll start by defining an array of angles:

NumPy提供了大量的有用的ufuncs，对于数据科学家来说非常有用的还包括三角函数。我们先定义一个角度的数组：

In [15]:
```python
theta = np.linspace(0, np.pi, 3)
```

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

然后来计算这个数组的一些三角函数值：

In [16]:
```python
print("theta      = ", theta)
print("sin(theta) = ", np.sin(theta)) # 正弦
print("cos(theta) = ", np.cos(theta)) # 余弦
print("tan(theta) = ", np.tan(theta)) # 正切
```

```
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]
```

> The values are computed to within machine precision, which is why values that should be zero do not always hit exactly zero. Inverse trigonometric functions are also available:

计算得到的值受到计算机浮点数精度的限制，因为上面看到的结果中应该为0的地方并不精确的等于0。还提供了逆三角函数：

In [17]:
```python
x = [-1, 0, 1]
print("x         = ", x)
print("arcsin(x) = ", np.arcsin(x)) # 反正弦
print("arccos(x) = ", np.arccos(x)) # 反余弦
print("arctan(x) = ", np.arctan(x)) # 反正切
```

```
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

指数和对数

> Another common type of operation available in a NumPy ufunc are the exponentials:

NumPy中另一种常用操作是指数：

In [18]:
```python
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:

指数的逆操作，对数函数。`np.log` 求的是自然对数；如果你需要计算2的对数或者10的对数，也有相应的函数：

In [19]:
```python
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:

还有当输入值很小时，可以保持精度的指数和对数函数：

In [20]:
```python
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.

当 x 很小时，这些函数会比 `np.log` 或 `np.exp` 计算得到更加精确的结果。

Specialized ufuncs

特殊的ufuncs

> NumPy has many more ufuncs available, including hyperbolic trig functions, bitwise arithmetic, comparison operators, conversions from radians to degrees, rounding and remainders, and much more. A look through the NumPy documentation reveals a lot of interesting functionality.

NumPy包含更多的ufuncs，包括双曲函数，二进制位运算，比较操作，角度弧度转换，舍入以及求余数等等。参考NumPy的在线文档你可以看到很多有趣的函数说明。

> Another excellent source for more specialized and obscure 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:

在 `scipy.special` 模块中还有更多的特殊及难懂的ufuncs。如果你需要计算使用到晦涩数学函数操作你的数据，基本上你都可以在这个模块中找到。下面列出了部分与数据统计相关的ufuncs，还有很多因为篇幅关系并未列出。

In [21]:
```python
from scipy import special
```

In [22]:
```python
# 伽玛函数（通用阶乘函数）及相关函数
x = [1, 5, 10]
print("gamma(x)     =", special.gamma(x)) # 伽玛函数
print("ln|gamma(x)| =", special.gammaln(x)) # 伽玛函数的自然对数
print("beta(x, 2)   =", special.beta(x, 2)) # 贝塔函数（第一类欧拉积分）
```

```
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 [23]:
```python
# 误差函数（高斯函数积分）
# 互补误差函数，逆误差函数
x = np.array([0, 0.3, 0.7, 1.0])
print("erf(x)  =", special.erf(x)) # 误差函数
print("erfc(x) =", special.erfc(x)) # 互补误差函数
print("erfinv(x) =", special.erfinv(x)) # 逆误差函数
```

```
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.

还有很多很多ufuncs，你可以在NumPy和 `scipy.special` 中找到。因为这些函数的文档都有在线版本，你可以用"gamma函数 python"就可以找到相关的信息。

Advanced Ufunc Features

高级Ufunc特性

> Many NumPy users make use of ufuncs without ever learning their full set of features. We'll outline a few specialized features of ufuncs here.

许多NumPy用户在使用ufuncs的时候都没有了解它们完整特性。我们在这里会简单介绍一些特别的特性。

Specifying output

指定输出

> 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:

对于大数据量的计算，有时指定存储输出数据的数组是很有用的。指定输出结果的内存位置能够避免创建临时的数组。所有的ufuncs都能通过指定 `out` 参数来指定输出的数组。

In [24]:
```python
x = np.arange(5)
y = np.empty(5)
np.multiply(x, 10, out=y) # 指定结果存储在y数组中
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:

输出结果甚至可以指定为数组的视图。例如，你可以将结果隔一个元素写入到一个数组中：

In [25]:
```python
y = np.zeros(10)
np.power(2, x, out=y[::2]) # 指定结果存储在y数组中，每隔一个元素存一个
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.

如果你使用的不是 `out` 参数，而是写成 `y[::2] = 2 ** x`，这回导致首先创建一个临时数组用来存储 `2 ** x`，然后再将这些值复制到y数组中。对于上面这么小的数组来说，其实没有什么区别，但是如果对象是一个非常大的数组，使用 `out` 参数会节省很多内存空间。

Aggregates

聚合

> For binary ufuncs, there are some interesting aggregates that 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.

对于二元ufuncs来说，还有一些很有趣的聚合函数可以直接从数组中计算出结果。例如，如果你想 `reduce` 一个数组，你可以对于任何ufuncs应用 `reduce` 方法。reduce会重复在数组的每一个元素进行ufunc的操作，直到最后得到一个标量。

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

例如，在 `add` ufunc上调用 `reduce` 会返回所有元素的总和：

In [26]:
```python
x = np.arange(1, 6)
np.add.reduce(x)
```

Out[26]:
```
15
```

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

相应的，在 `multiply` ufunc上调用 `reduce` 会返回所有元素的乘积：

In [27]:
```python
np.multiply.reduce(x)
```

Out[27]:
```
120
```

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

如果你需要得到每一步计算得到的中间结果，你可以调用 `accumulate`：

In [28]:
```python
np.add.accumulate(x)
```

Out[28]:
```
array([ 1,  3,  6, 10, 15])
```

In [29]:
```python
np.multiply.accumulate(x)
```

Out[29]:
```
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.

注意对于上面这种特殊情况，NumPy也提供了相应的函数直接计算结果（`np.sum`，`np.prod`，`np.cumsum`，`np.cumprod`），我们会在聚合：Min, Max, 以及其他中详细讨论。

Outer products

外积

> 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:

最后，任何ufunc都可以计算输入的每一对元素的结果，使用 `outer` 方法。你可以一行代码就完成类似创建乘法表的功能：

In [30]:
```python
x = np.arange(1, 6)
np.multiply.outer(x, x)
```

Out[30]:
```
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, which we'll explore in Fancy Indexing, are very helpful as well.

`ufunc.at` 和 `ufunc.reduceat` 方法也非常有用，我们会在高级索引中详细讨论。

> Another extremely useful feature of ufuncs is the ability to operate between arrays of different sizes and shapes, a set of operations known as *broadcasting*. This subject is important enough that we will devote a whole section to it (see Computation on Arrays: Broadcasting).

Ufuncs还有一个极端有用的特性，能让ufuncs在不同长度和形状的数组之间进行计算，这是一组被称为广播的方法。这是一个非常重要的内容，因此我们会专门在在数组上计算：广播小节中进行介绍。

Ufuncs: Learning More

Ufuncs：更多资源

> More information on universal functions (including the full list of available functions) can be found on the NumPy and SciPy documentation websites.

更多有关ufuncs的信息（包括完整的函数列表）可以在NumPy 和 SciPy的在线文档获得。

> Recall that you can also access information directly from within IPython by importing the packages and using IPython's tab-completion and help (`?`) functionality, as described in Help and Documentation in IPython.

不要忘记了我们可以使用IPython的帮助工具 ? 来获取任何相关的帮助信息，正如我们在IPython的帮助和文档中介绍过的那样。