Numpy 배열 연산 : 유니버설 함수

Numpy 는 데이터 배열을 사용하여 최적화된 연산을 위한 쉽고 유연한 인터페이스를 제공한다. Numpy 배열의 연산은 아주 빠르거나 아주 느릴 수 있다.

이 연산을 빠르게 만드는 핵심은 벡터화 연산을 사용하는 것인데, 그것은 일반적으로 Numpy 의 유니버설 함수를 통해 구현된다. 이번절에서는 배열 요소에 대한 반복적인 계산을 더 효율적으로 수행하게 해주는 Numpy 의 ununcs 의 필요성에 대해 생각해 보겠다. 그리고 나서 Numpy 패키지에서 사용할 수 있는 가장 보편적이면서 유용한 여러 가지 산술 유니버설 함수를 소개한다.

루프는 느리다

파이썬의 기본구현에서 몇 가지 연산은 매우 느리게 수행된다. 이는 부분적으로 파이썬이 동적인 인터프린터 언어이기 때문이다. 타입이 유연하다는 사실은 결국 일련의 연산들이 C 와 포트란 같은 언어에서처럼 효율적인 머신코드로 컴파일될 수 없다는 뜻이다. 최근에 이러한 취약점을 해결하기 위한 다양한 시도가 있었다. 유명한 사례로 JIT(Just-In-Time)컴파일하는 파이썬을 구현하는 파이파이 프로젝트(PyPy project, http://www.pypy.org). 파이썬 코드 조작을 빠른 LLVM 바이트 코드로 반환하는 넘바(Numba)프로젝트가 있다. 각 프로젝트가 장단점은 있지만 세 가지 접근법 중 어느 것도 표준 CPython 엔진의 범위와 대중성을 넘어서지는 못했다고 하는 게 맞을 것같다.

파이썬은 수많은 작은 연산이 반복되는 상황에서 확연히 느리다. 배열을 반복해서 각 요소를 조작하는 것을 예로 들 수 있다. 일례로 값으로 이루어진 배열이 있고 각각의 역수를 계산하려고 한다고 가정하자. 직관적인 방식은 다음과 같다.

```
In [1]: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     ])
```

이 구현은 아마 C 자바 경험을 가진 사람에게는 꽤 자연스러워 보일 것이다. 그러나 큰 입력값을 넣고 이 코드의 실행 시간을 측정해 보면 이 연산이 놀라울 정도로 느리다는 것을 알게 될 것이다.

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

```
2.51 s ± 245 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

이 경우에는 백만 번 연산하고 그 결과를 저장하는데 수 초가 걸린다. 휴대전화의 처리 속도도 Giga-FLOPS (즉, 초당 수십억의 수치 연산)로 측정되는 상황에서 이것은 거의 터무니 없을 정도로 느려 보인다. 여기서 병목은 연산 자체에 있는 것이 아니라 Cpython 이 루프의 사이클마다 수행해야 하는 타입 확인과 함수 디스패치에서 발생한다. 액수가 계산될 때마다 파이썬은 먼저 객체의 타입을 확인하고 해당타입에 맞게 사용할 적절한 함수를 동적으로 검색한다. 만약 컴파일된 코드로 작업했다면 코드를 실행하기 전에 타입을 알았을 것이고 결과값은 좀 더 효율적으로 계산됐을 것이다.

UFuncs 소개

Numpy 는 여러 종류의 연산에 대해 이러한 종류의 정적 타입 체계를 가진 컴파일된 루틴에 편리한 인터페이스를 제공한다. 이를 벡터화 연산이라고 한다. 벡터화 연산은 간단히 배열로 연산을 수행해 각 요소에 적용함으로써 수행할 수 있다. 이 벡터화 방식은 루프를 Numpy 의 가치를 이루는 컴파일된 계층으로 밀어 넣음으로써 훨씬 빠르게 실행되도록 설계됐다.

다음 두 결과를 비교해 보자:

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

이 대규모 배열에 대한 실행 시간을 보면 이 코드가 파이썬 루프보다 수백 배 빠른 속도로 작업을 완료한다는 것을 알 수 있다.

```
In [4]: %timeit (1.0 / big_array)
2.51 ms ± 157 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Numpy 에서 벡터화 연산은 Numpy 배열의 값에 반복된 연산을 빠르게 수행하는 것을 주목적으로 하는 ufuncs 를 통해 구현된다. 유니버설 함수는 매우 유연해서 이전에 스칼라와 배열 사이의 연산을 봤지만 두 배열 간의 연산도 가능하다.

:

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

Ufunc 연산은 1 차원 배열에 국한되지 않고 다차원 배열에서도 동작한다.

```
In [6]:x = np.arange(9).reshape((3, 3))
       2 ** x
Out[6]:array([[  1,   2,   4],
       [  8,  16,  32],
       [ 64, 128, 256]])
```

ufunc 를 통한 벡터화를 이용한 연산은 파이썬 루프를 통해 구현된 연산보다 대부분 더 효율적이며, 특히 배열의 크기가 커질수록 그 차이가 확연하다. 파이썬 스크립트에서 그러한 루프를 보면 항상 벡터화 표현식으로 교체할 수 있을지 고민해야 한다.

UFuncs 유니버설 함수(UFuncs)

Ufuncs 에는 단일 입력값에 동작하는 단항 ufuncs 와 두 개의 입력값에 동작하는 이항 ufuncs 로 두 종류가 있다.

배열 산술 연산

NumPy ufuncs 는 파이썬의 기본 산술 연산자를 사용하기 때문에 자연스럽게 사용할 수 있다. 표준 덧셈, 뺄셈, 곱셈, 나눗셈 모두 사용할 수 있다.

```
In [7]: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)  # 나머지 연산
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]
```

또한, 음수를 만드는 단항 ufuncs 와 지수 연산자 `**`, 나머지 연산자 `%` 가 있다.

```
In [8]: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]
```

이 연산들은 원하는 만큼 함께 사용할 수 있으며 표준 연산 순서를 따른다.

```
In [9]:-(0.5*x + 1) ** 2
Out[9]:array([-1.  , -2.25, -4.  , -6.25])
```

이 산술 연산은 모두 사용상 편의를 위해 NumPy 에 내장된 특정 함수를 감싼 것이다. 예를 들면 , +연산자는 add 함수의 래퍼(wrapper)함수다.

```
In [10]:np.add(x, 2)
Out[10]:array([2, 3, 4, 5])
```

Operator	Equivalent ufunc	Description
+	np.add	덧셈(e.g., 1 + 1 = 2)

Operator	Equivalent ufunc	Description
`-`	`np.subtract`	뺄셈 (e.g., `3 - 2 = 1`)
`-`	`np.negative`	단항 음수 (e.g., `-2`)
`*`	`np.multiply`	곱셈 (e.g., `2 * 3 = 6`)
`/`	`np.divide`	나눗셈 (e.g., `3 / 2 = 1.5`)
`//`	`np.floor_divide`	몫 나눗셈 (e.g., `3 // 2 = 1`)
`**`	`np.power`	지수 연산 (e.g., `2 ** 3 = 8`)
`%`	`np.mod`	나머지 연산 (e.g., `9 % 4 = 1`)

절대값 함수

NumPy 는 파이썬 내장된 산술 연산자를 이해하는 것과 마찬가지로 파이썬에 내장된 절댓값함수도 이해한다.

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

이 절대값 함수에 대응하는 NumPy ufunc 는 `np.absolute` 로, `np.abs` 라는 별칭으로도 사용할 수 있다.

```
In [12]:np.absolute(x)
Out[12]:array([2, 1, 0, 1, 2])
In [13]:np.abs(x)
Out[13]:array([2, 1, 0, 1, 2])
```

이 ufunc 는 복소수 데이터도 처리할 수 있으며, 이 경우 절댓값은 크기를 반환한다.

```
In [14]:x = np.array([3 - 4j, 4 - 3j, 2 + 0j, 0 + 1j])
        np.abs(x)
Out[14]:array([ 5.,  5.,  2.,  1.])
```

삼각함수

NumPy 는 수많은 유용한 유니버설 함수를 제공하는데, 데이터 과학자에게 가장 유용한 함수 중 일부가 삼각함수다. 먼저 각도 배열을 정의하자.

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

이제 이 값들로 몇가지 삼각함수를 계산할 수 있다.:

```
In [16]: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]
```

이 값들은 기계 정밀도 내에서 계산되며, 그래서 0 이어야 하는 값이 언제나 0 이 되지 않는다.

```
In [17]: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]
```

지수와 로그

NumPy 유니버설 함수에서 사용할 수 있는 또 다른 보편적인 유형의 연산은 지수 연산이다:

```
In [18]: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]
```

지수의 역인 로그도 사용할 수 있다. 기본 np.log 는 자연로그를 제공한다. 2 를 밑으로 하는 로그를 계산하거나 10 을 밑으로 하는 로그를 계산하는 것 역시 가능하다.

```
In [19]: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.        ]
```

매우 작은 입력값의 정확도를 유지하고자 할 때 유용한 특화된 버전도 있다.

```
In [20]: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]
```

이 함수들은 x 가 매우 작을 때 np.log 나 np.exp 를 사용했을 때보다 더 정확한 값을 내 놓는다..

특화된 유니버설 함수

NumPy 에는 쌍곡선 삼각함수, 비트 연산, 비교 연산자, 라디안을 각도로 변환, 반올림과 나머지 등 훨씬 더 많은 ufunc 가 있다. NumPy 문서에서 수많은 흥미로운 기능을 확인 할 수 있다.

좀 더 전문적이고 보기 드문 ufunc 에 대한 또 다른 훌륭한 소스로 서브 모듈인 scipy.special 이 있다. 잘 알려지지 않은 수학적 함수를 사용하여 데이터를 계산하고자 한다면 scipy.special 로 구현할 가능성이 크다. 포함된 함수가 너무 많아서 모두 나열하기 어렵지만 다음 코드에서 통계학에서나 등장할 법한 함수 몇 가지를 확인할 수 있다.

```
(pip install scipy)
In [21]:from scipy import special
In [22]:# 감마 함수(일반화된 계승)와 관련 함수
        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)      = [  1.00000000e+00   2.40000000e+01   3.62880000e+05]
ln|gamma(x)| = [  0.           3.17805383  12.80182748]
beta(x, 2)    = [ 0.5         0.03333333  0.00909091]
In [23]:# 오차 함수(가우스 격분),  그 보수(complement), 와 역수(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]
```

NumPy 와 scipy.special 에는 훨씬 더 많은 ufruncs 가 있다. 이 패키지 문서는 온라인에서 볼 수 있으며 웹에서 '감사 함수 파이썬'으로 검색하면 관련 정보를 찾을 수 있다.

고급 Ufunc 기능

수 많은 NumPy 사용자가 ufuncs 의 기능을 완전히 배우지 않고 사용한다. 여기에 몇가지 전문화된 기능을 정리한다.

출력 지정

대규모 연산인 경우, 연산 결과를 저장할 배열을 지정하는 것이 유용할 때가 있다. 임시 배열을 생성하지 않고 지정한 배열을 이용해 원하는 메모리 위치에 직접 연산 결과를 쓸 수 있다. 모든 ufuncs 에서 함수의 out 인수를 사용해 출력을 지정할 수 있다.

```
In [24]:x = np.arange(5)
        y = np.empty(5)
        np.multiply(x, 10, out=y)
        print(y)
```

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

이것은 배열 뷰와 함께 사용할 수도 있다. 예를 들어, 연산 결과를 지정된 배열의 요소에 하나씩 건너뛰면서 기록할 수 있다.

```
In [25]:y = np.zeros(10)
        np.power(2, x, out=y[::2])
        print(y)
```

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

대신 y[::2] = 2 ** x 로 작성했다면 2 ** x 의 결과를 담고 있는 임시 배열을 생성한 다음, 그 값을 y 배열에 복사했을 것이다. 이것이 소규모 연산에서는 별 차이가 없지만, 대단히 큰 규모의 배열에서는 out 인수를 신중하게 사용함으로써 절약되는 메모리가 상당히 크다.

집계

이항 ufuncs 의 경우 객체로 부터 직접 연산할 수 있는 흥미로운 집계 함수가 몇가지 있다. 가령 배열을 특정 연산으로 축소하고자 한다면 ufunc 의 reduce 메서드를 사용하면 된다. Reduce 메서드는 결과가 하나만 남을 때까지 해당 연산을 배열 요소에 반복해서 작성한다.

예를 들어: add ufunc 의 redue 를 호출하면 배열의 모든 요소의 합을 반환한다.

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

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

마찬가지로 multiply ufunc 에 educe 를 호출하면 모든 배열 요소의 곱을 반환한다.

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

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

계산의 중간 결과를 모두 저장하고 싶다면 대신 accumulate 를 사용하면 된다.

```
In [28]:np.add.accumulate(x)
Out[28]:array([ 1,  3,  6, 10, 15])
In [29]:np.multiply.accumulate(x)
Out[29]:array([  1,   2,   6,  24, 120])
```

이 연산의 경우, 그 결과를 계산하는 전용 NumPy 함수(np.sum, np.prod, np.cumsum, np.cum prod)도 존재한다.

외적(Outer products)

마지막으로 모든 ufunc 는 outer 메서드를 이용해 서로 다른 두 입력값의 모든 쌍이 대한 출력값을 계산할 수 있다. 이렇게 하면 코드 한 줄로 곱 셈 테이블을 만드는 것과 같은 일을 할 수 있다.

```
In [30]: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]])
```