Data Types and computation in NumPy by Mrittika Megaraj

Understanding Data Types in Python

Effective data-driven research and computing involves knowledge of the storage and manipulation of data. This section compares and contrasts how arrays of data are handled by the Python programming language and by NumPy. Understanding this distinction is crucial to comprehending most of the next section's content.

Dynamic typing is a component of Python's user-friendliness that attracts many programmers. A statically-typed language, such as C or Java, requires that each variable be declared explicitly, while a dynamically-typed language, such as Python, omits this need. For instance, you might express a certain operation in C as follows:

```
/* C code */
int result = 0;
for(int i=0; i<100; i++){
    result += i;
}
```

While in Python the equivalent operation could be written this way:

```
# Python code
result = 0
for i in range(100):
    result += i
```

In C, the data types of each variable are explicitly defined, but in Python, the data types are inferred dynamically. This allows us, for instance, to assign any kind of data to any variable:

```
# Python code
x = 4
x = "four"
```

Here, the integer content of x has been converted to a string. Depending on the compiler settings, doing the same in C would result in a compilation error or other unintended outcomes.

```
/* C code */
int x = 4;
x = "four";  // FAILS
```

One thing that makes Python and other dynamically-typed languages useful and easy to use is that they give you this kind of freedom. To learn how to use Python to analyze data quickly and effectively, it is important to understand how this works. But this type flexibility also shows that Python variables are more than just their value; they have extra information about the type of the value. We'll talk more about this in the sections that come after this one.

A Python Integer Is More Than Just an Integer

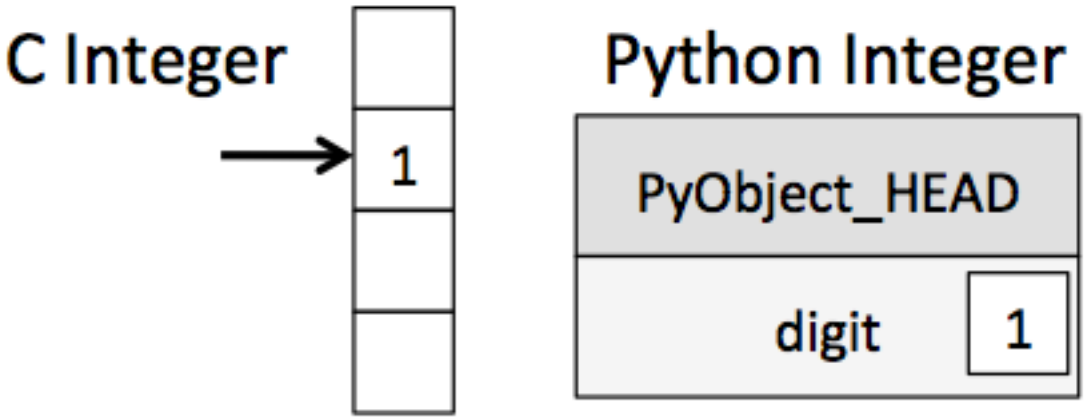
C is used to write the standard Programming language. This means that every Python object is just a cleverly disguised C structure that holds not only its value but also other information. For instance, when we use x = 10000 to define an integer in Python, x is not just a "raw" integer. It's actually a pointer to a C structure that has several values in it. When we look at the source code for Python 3.4, we see that the integer (long) type definition looks like this (after the C macros are expanded):

```
struct _longobject {
    long ob_refcnt;
    PyTypeObject *ob_type;
    size_t ob_size;
    long ob_digit[1];
};
```

A single integer in Python 3.4 actually contains four pieces:

- `ob_refcnt`, a reference count that helps Python silently handle memory allocation and deallocation
- `ob_type`, which encodes the type of the variable
- `ob_size`, which specifies the size of the following data members
- `ob_digit`, which contains the actual integer value that we expect the Python variable to represent.

This means that there is some overhead in storing an integer in Python as compared to an integer in a compiled language like C, as illustrated in the following figure:



Here `PyObject_HEAD` is the part of the structure containing the reference count, type code, and other pieces mentioned before.

Notice the difference here: a C integer is essentially a label for a position in memory whose bytes encode an integer value. A Python integer is a pointer to a position in memory containing all the Python object information, including the bytes that contain the integer value. This extra information in the Python integer structure is what allows Python to be coded so freely and

dynamically. All this additional information in Python types comes at a cost, however, which becomes especially apparent in structures that combine many of these objects.

A Python List Is More Than Just a List

Let's consider now what happens when we use a Python data structure that holds many Python objects. The standard mutable multi-element container in Python is the list. We can create a list of integers as follows:

```
L = list(range(10))
L
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
type(L[0])
int
```

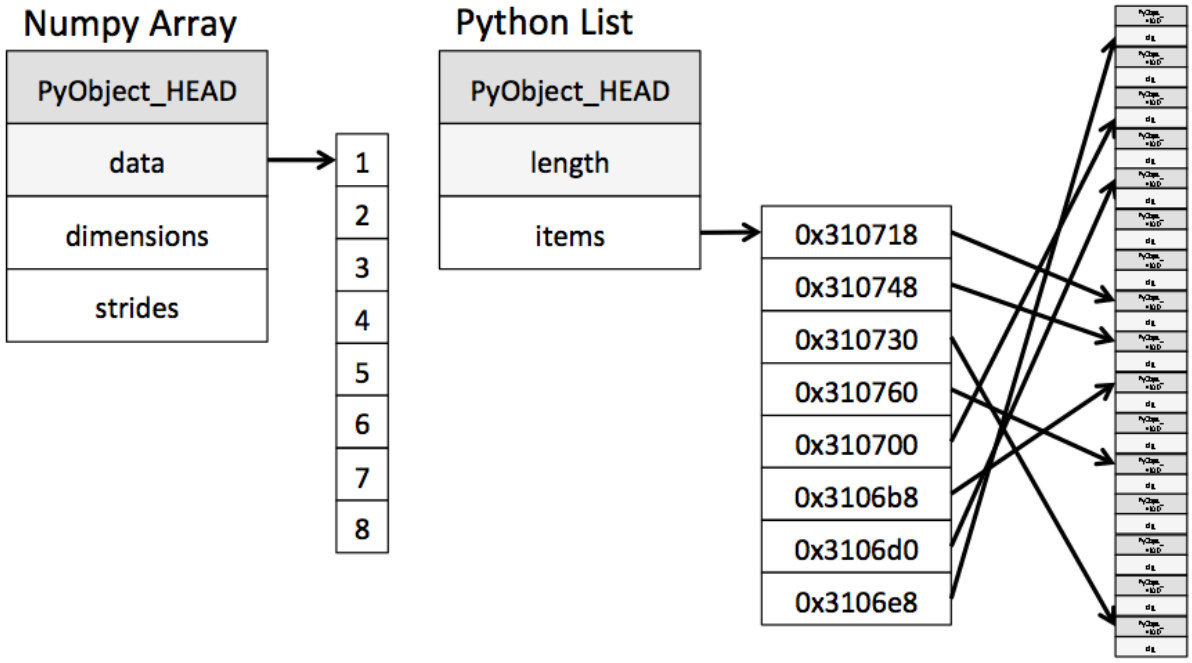
Or, similarly, a list of strings:

```
L2 = [str(c) for c in L]
L2
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
type(L2[0])
str
```

Because of Python's dynamic typing, we can even create heterogeneous lists:

```
L3 = [True, "2", 3.0, 4]
[type(item) for item in L3]
[bool, str, float, int]
```

But this flexibility comes at a cost: to allow these flexible types, each item in the list must contain its own type info, reference count, and other information–that is, each item is a complete Python object. In the special case that all variables are of the same type, much of this information is redundant: it can be much more efficient to store data in a fixed-type array. The difference between a dynamic-type list and a fixed-type (NumPy-style) array is illustrated in the following figure:



Technically speaking, an array is only a reference to a single chunk of data. On the other hand, each member of a Python list is a reference to a whole Python object, such as the Python integer we saw before. Once again, the list's versatility is a benefit; as each list element is a complete structure comprising both data and type information, the list may be populated with information of any desired kind. Fixed-type NumPy-style arrays are far more efficient for storing and processing data, but they lack this flexibility.

Fixed-Type Arrays in Python

Python offers several different options for storing data in efficient, fixed-type data buffers. The built-in `array` module (available since Python 3.3) can be used to create dense arrays of a uniform type:

```
import array
L = list(range(10))
A = array.array('i', L)
A
array('i', [0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Here `'i'` is a type code indicating the contents are integers.

Much more useful, however, is the `ndarray` object of the NumPy package. While Python's `array` object provides efficient storage of array-based data, NumPy adds to this efficient *operations* on that data. We will explore these operations in later sections; here we'll demonstrate several ways of creating a NumPy array.

We'll start with the standard NumPy import, under the alias `np`:

```
import numpy as np
```

Creating Arrays from Python Lists

First, we can use `np.array` to create arrays from Python lists:

```
# integer array:
np.array([1, 4, 2, 5, 3])
array([1, 4, 2, 5, 3])
```

Remember that unlike Python lists, NumPy is constrained to arrays that all contain the same type. If types do not match, NumPy will upcast if possible (here, integers are up-cast to floating point):

```
np.array([3.14, 4, 2, 3])
array([ 3.14,  4.  ,  2.  ,  3.  ])
```

If we want to explicitly set the data type of the resulting array, we can use the `dtype` keyword:

```
np.array([1, 2, 3, 4], dtype='float32')
array([ 1.,  2.,  3.,  4.], dtype=float32)
```

Finally, unlike Python lists, NumPy arrays can explicitly be multi-dimensional; here's one way of initializing a multidimensional array using a list of lists:

```
# nested lists result in multi-dimensional arrays
np.array([range(i, i + 3) for i in [2, 4, 6]])
array([[2, 3, 4],
       [4, 5, 6],
       [6, 7, 8]])
```

The inner lists are treated as rows of the resulting two-dimensional array.

Creating Arrays from Scratch

Especially for larger arrays, it is more efficient to create arrays from scratch using routines built into NumPy. Here are several examples:

```
# Create a length-10 integer array filled with zeros
np.zeros(10, dtype=int)
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
# Create a 3x5 floating-point array filled with ones
np.ones((3, 5), dtype=float)
array([[ 1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.]])
# Create a 3x5 array filled with 3.14
np.full((3, 5), 3.14)
array([[ 3.14,  3.14,  3.14,  3.14,  3.14],
       [ 3.14,  3.14,  3.14,  3.14,  3.14],
       [ 3.14,  3.14,  3.14,  3.14,  3.14]])
# Create an array filled with a linear sequence
# Starting at 0, ending at 20, stepping by 2
# (this is similar to the built-in range() function)
np.arange(0, 20, 2)
```

```
array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])
# Create an array of five values evenly spaced between 0 and 1
np.linspace(0, 1, 5)
array([ 0.  ,  0.25,  0.5 ,  0.75,  1.  ])
# Create a 3x3 array of uniformly distributed
# random values between 0 and 1
np.random.random((3, 3))
array([[ 0.99844933,  0.52183819,  0.22421193],
       [ 0.08007488,  0.45429293,  0.20941444],
       [ 0.14360941,  0.96910973,  0.946117  ]])
# Create a 3x3 array of normally distributed random values
# with mean 0 and standard deviation 1
np.random.normal(0, 1, (3, 3))
array([[ 1.51772646,  0.39614948, -0.10634696],
       [ 0.25671348,  0.00732722,  0.37783601],
       [ 0.68446945,  0.15926039, -0.70744073]])
# Create a 3x3 array of random integers in the interval [0, 10)
np.random.randint(0, 10, (3, 3))
array([[2, 3, 4],
       [5, 7, 8],
       [0, 5, 0]])
# Create a 3x3 identity matrix
np.eye(3)
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
# Create an uninitialized array of three integers
# The values will be whatever happens to already exist at that memory
location
np.empty(3)
array([ 1.,  1.,  1.])
```

NumPy Standard Data Types

NumPy arrays contain values of a single type, so it is important to have detailed knowledge of those types and their limitations. Because NumPy is built in C, the types will be familiar to users of C, Fortran, and other related languages.

The standard NumPy data types are listed in the following table. Note that when constructing an array, they can be specified using a string:

```
np.zeros(10, dtype='int16')
```

Or using the associated NumPy object:

```
np.zeros(10, dtype=np.int16)
```

Data type	Description
`bool_`	Boolean (True or False) stored as a byte
`int_`	Default integer type (same as C `long`; normally either `int64` or `int32`)
`intc`	Identical to C `int` (normally `int32` or `int64`)
`intp`	Integer used for indexing (same as C `ssize_t`; normally either `int32` or `int64`)
`int8`	Byte (-128 to 127)
`int16`	Integer (-32768 to 32767)
`int32`	Integer (-2147483648 to 2147483647)
`int64`	Integer (-9223372036854775808 to 9223372036854775807)
`uint8`	Unsigned integer (0 to 255)
`uint16`	Unsigned integer (0 to 65535)
`uint32`	Unsigned integer (0 to 4294967295)
`uint64`	Unsigned integer (0 to 18446744073709551615)
`float_`	Shorthand for `float64`.
`float16`	Half precision float: sign bit, 5 bits exponent, 10 bits mantissa
`float32`	Single precision float: sign bit, 8 bits exponent, 23 bits mantissa
`float64`	Double precision float: sign bit, 11 bits exponent, 52 bits mantissa
`complex_`	Shorthand for `complex128`.
`complex64`	Complex number, represented by two 32-bit floats
`complex128`	Complex number, represented by two 64-bit floats

Computation on NumPy Arrays: Universal Functions

In the last few parts, we have been going over some of the fundamentals of NumPy; in the next few sections, we will dive into the reasons why NumPy is such an essential part of the Python data science community. To be more specific, it offers a straightforward and adaptable user interface for performing optimal computations with arrays of data.

The speed of computation on NumPy arrays may range anywhere from extremely fast to very sluggish. Utilizing vectorized operations, which are often carried out with the assistance of NumPy's universal functions (ufuncs), is the essential component in making it fast. This section explains why NumPy needs ufuncs, which are functions that may be used to do iterative computations on array members in a way that is much more time and space efficient. The next step is that it presents a significant number of the NumPy package's arithmetic ufuncs that are the most popular and helpful..

The Slowness of Loops

Python's default implementation (known as CPython) does some operations very slowly. This is because the language is dynamic and interpreted, which prevents sequences of operations from being efficiently converted down to machine code in the same way that they can be in languages like C and Fortran. Several initiatives have been launched recently to address this shortcoming; some of the most well-known are the Cython project, which translates Python into C, the Numba project, which translates Python code snippets into fast LLVM bytecode, and the PyPy project, a just-in-time compiled implementation of Python. There are advantages and disadvantages to each of these three methods, but it's fair to say that none of them have had as much of an impact as the original CPython engine.

For example, looping through arrays to perform an action on each element is a good example of a scenario where Python's relative slowness becomes apparent. Here's an example: we want to get the reciprocal of every value in a given array. The following is an example of a simple strategy:

```
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)
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:

```
big_array = np.random.randint(1, 100, size=1000000)
%timeit compute_reciprocals(big_array)
1 loop, best of 3: 2.91 s per loop
```

In order to calculate and save the outcome of this million operations, several seconds are required. This appears almost laughably slow in a world when even mobile phones can handle data at rates measured in Giga-FLOPS (billions of numerical operations per second). It turns out that the bottleneck here is not the operations themselves, but the type-checking and function dispatching that CPython must conduct at each cycle of the loop. Python initially checks the object's type and then dynamically looks for the appropriate function to employ to calculate the reciprocal. This type definition would be known before the code runs in compiled code, allowing for more efficient computation of the result.

Introducing UFuncs

NumPy offers an easy interface into exactly this sort of statically typed and built routine, which may be used for a wide variety of different tasks. This kind of operation is referred to as a vectorized operation. It is possible to achieve this goal by just carrying out an operation on the array, which will thereafter be carried out on each element. This vectorized technique aims to push the loop into the compiled layer that NumPy is built on, which ultimately results in a significant increase in the program's execution speed. Compare the results of the following two:

```
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:

```
%timeit (1.0 / big_array)
100 loops, best of 3: 4.6 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]])
```

As the size of the arrays increase, the efficiency gains from employing vectorization through ufuncs exceed those from using Python loops. Consider if a vectorized expression may be substituted for the loop whenever you see one in a Python script.

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

Array arithmetic

Since NumPy ufuncs employ Python's built-in numeric operators, they are immediately familiar to users. Addition, subtraction, multiplication, and division may be used normally:

```
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, 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]
```

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

```
-(0.5*x + 1) ** 2
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:

```
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`)

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:

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

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

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

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

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

Trigonometric functions

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:

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

Now we can compute some trigonometric functions on these values:

```
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]
```

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:

```
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

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

```
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.        ]
```

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

```
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

In addition to the ufuncs already mentioned, NumPy provides access to hyperbolic trig functions, bitwise arithmetic, comparison operators, radian to degree conversions, rounding, and remainders. NumPy has a ton of cool features, as you can see by reading through the docs.

The scipy.special submodule is another great place to look for unique and uncommon ufuncs. scipy.special is the place to look if you need to do some arcane mathematical computation on your data. The following code sample demonstrates two functions that may be useful in a statistical setting, although there are simply too many to mention them all.

```
from scipy import special
# Gamma functions (generalized factorials) and related functions
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]
# 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

Unfortunately, many of those who use NumPy also utilize ufuncs without understanding what they can do. We'll describe some of ufuncs' more advanced capabilities.

Specifying output

There are occasions when being able to choose the array in which the calculation's output is stored would be helpful, especially for lengthy computations. This may be used to avoid constructing a temporary array and instead save the results of a calculation in the desired position in memory. This may be accomplished using the out parameter of any ufunc:

```
x = np.arange(5)
y = np.empty(5)
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:

```
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.

Aggregates

Many interesting aggregates may be built on binary ufuncs at runtime. Any ufunc may be used to execute a specified action on an array, such shrinking its size, thanks to its reduction function. To reduce an array, one applies an operation to its elements repeatedly until there is only one value left..

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

```
x = np.arange(1, 6)
np.add.reduce(x)
15
```

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

```
np.multiply.reduce(x)
120
```

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

```
np.add.accumulate(x)
array([ 1,  3,  6, 10, 15])
```

```
np.multiply.accumulate(x)
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:

```
x = np.arange(1, 6)
np.multiply.outer(x, x)
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 `ufu`

`nc.reduceat` methods, which we'll explore in Fancy Indexing, are very helpful as well.

Broadcasting is a collection of operations that allows you to perform actions across arrays of various sizes and shapes, and is another powerful aspect of ufuncs. We feel this topic warrants its own section because of its significance. The broadcasting set of operations in ufuncs is another powerful tool for working with arrays of varying sizes and shapes. Given the significance of the issue at hand, we have decided to dedicate an entire chapter to it.