

# Understanding Data Types in Python

Effective data-driven science and computation requires understanding how data is stored and manipulated. This chapter outlines and contrasts how arrays of data are handled in the Python language itself, and how NumPy improves on this. Understanding this difference is fundamental to understanding much of the material throughout the rest of the book.

Users of Python are often drawn in by its ease of use, one piece of which is dynamic typing. While a statically typed language like C or Java requires each variable to be explicitly declared, a dynamically typed language like Python skips this specification. For example, in C you might specify a particular operation 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
```

Notice one main difference: in C, the data types of each variable are explicitly declared, while in Python the types are dynamically inferred. This means, for example, that we can assign any kind of data to any variable:

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

Here we've switched the contents of `x` from an integer to a string. The same thing in C would lead (depending on compiler settings) to a compilation error or other unintended consequences:

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

This sort of flexibility is one element that makes Python and other dynamically typed languages convenient and easy to use. Understanding *how* this works is an important piece of learning to analyze data efficiently and effectively with Python. But what this type flexibility also points to is the fact that Python variables are more than just their values; they also contain extra information about the *type* of the value. We'll explore this more in the sections that follow.

## A Python Integer Is More Than Just an Integer

The standard Python implementation is written in C. This means that every Python object is simply a cleverly disguised C structure, which contains not only its value, but other information as well. For example, when we define an integer in Python, such as `x = 10000`, `x` is not just a "raw" integer. It's actually a pointer to a compound C structure, which contains several values. Looking through the Python 3.10 source code, we find that the integer (long) type definition effectively looks like this (once 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 3.10 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 involved in storing an integer in Python as compared to 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 multielement container in Python is the list. We can create a list of integers as follows:

```
In [1]: L = list(range(10))  
L
```

```
Out[1]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
In [2]: type(L[0])
```

```
Out[2]: int
```

Or, similarly, a list of strings:

```
In [3]: L2 = [str(c) for c in L]  
L2
```

```
Out[3]: ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
```

```
In [4]: type(L2[0])
```

```
Out[4]: str
```

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

```
In [5]: L3 = [True, "2", 3.0, 4]  
[type(item) for item in L3]
```

```
Out[5]: [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, 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, so it can be much more efficient to store the 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:

 Array Memory Layout

At the implementation level, the array essentially contains a single pointer to one contiguous block of data. The Python list, on the other hand, contains a pointer to a block of pointers, each of which in turn points to a full Python object like the Python integer we saw earlier. Again, the advantage of the list is flexibility: because each list element is a full structure containing both data and type information, the list can be filled with data of any desired type. Fixed-type NumPy-style arrays lack this flexibility, but are much more efficient for storing and manipulating data.

# 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:

```
In [6]: import array
        L = list(range(10))
        A = array.array('i', L)
        A
```

```
Out[6]: 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 chapters; next, I'll show you a few different ways of creating a NumPy array.

## Creating Arrays from Python Lists

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

```
In [ ]: import numpy as np
```

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

```
In [8]: # Integer array
        np.array([1, 4, 2, 5, 3])
```

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

Remember that unlike Python lists, NumPy arrays can only contain data of the same type. If the types do not match, NumPy will upcast them according to its type promotion rules; here, integers are upcast to floating point:

```
In [9]: np.array([3.14, 4, 2, 3])
```

```
Out[9]: 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:

```
In [10]: np.array([1, 2, 3, 4], dtype=np.float32)
```

```
Out[10]: array([1., 2., 3., 4.], dtype=float32)
```

Finally, unlike Python lists, which are always one-dimensional sequences, NumPy arrays can be multidimensional. Here's one way of initializing a multidimensional array using a list of lists:

```
In [11]: # Nested lists result in multidimensional arrays  
np.array([range(i, i + 3) for i in [2, 4, 6]])
```

```
Out[11]: 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:

```
In [12]: # Create a length-10 integer array filled with 0s  
np.zeros(10, dtype=int)
```

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

```
In [13]: # Create a 3x5 floating-point array filled with 1s  
np.ones((3, 5), dtype=float)
```

```
Out[13]: array([[1., 1., 1., 1., 1.],  
               [1., 1., 1., 1., 1.],  
               [1., 1., 1., 1., 1.]])
```

```
In [14]: # Create a 3x5 array filled with 3.14  
np.full((3, 5), 3.14)
```

```
Out[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]])
```

```
In [15]: # 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)
```

```
Out[15]: array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])
```

```
In [16]: # Create an array of five values evenly spaced between 0 and 1
np.linspace(0, 1, 5)
```

```
Out[16]: array([0. , 0.25, 0.5 , 0.75, 1.  ])
```

```
In [17]: # Create a 3x3 array of uniformly distributed
# pseudorandom values between 0 and 1
np.random.random((3, 3))
```

```
Out[17]: array([[0.09610171, 0.88193001, 0.70548015],
                [0.35885395, 0.91670468, 0.8721031 ],
                [0.73237865, 0.09708562, 0.52506779]])
```

```
In [18]: # Create a 3x3 array of normally distributed pseudorandom
# values with mean 0 and standard deviation 1
np.random.normal(0, 1, (3, 3))
```

```
Out[18]: array([[ -0.46652655, -0.59158776, -1.05392451],
                [-1.72634268,  0.03194069, -0.51048869],
                [ 1.41240208,  1.77734462, -0.43820037]])
```

```
In [19]: # Create a 3x3 array of pseudorandom integers in the interval [0, 10)
np.random.randint(0, 10, (3, 3))
```

```
Out[19]: array([[4, 3, 8],
                [6, 5, 0],
                [1, 1, 4]])
```

```
In [20]: # Create a 3x3 identity matrix
np.eye(3)
```

```
Out[20]: array([[1., 0., 0.],
                [0., 1., 0.],
                [0., 0., 1.]])
```

```
In [21]: # Create an uninitialized array of three integers; the values will be
# whatever happens to already exist at that memory location
np.empty(3)
```

```
Out[21]: 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

More advanced type specification is possible, such as specifying big- or little-endian numbers; for more information, refer to the [NumPy documentation \(http://numpy.org/\)](http://numpy.org/). NumPy also supports compound data types, which will be covered in [Structured Data: NumPy's Structured Arrays \(02.09-Structured-Data-NumPy.ipynb\)](#).