< Numpy介绍 | 目录 | Numpy数组基础 >

Open in Colab

Understanding Data Types in Python

理解Python中的数据类型

> Effective data-driven science and computation requires understanding how data is stored and manipulated. This section 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.

想要有效的掌握数据驱动科学和计算需要理解数据是如何存储和处理的。本节将描述和对比数组在Python语言中和在NumPy中是怎么处理的，NumPy是如何优化了这部分的内容。理解这个区别是理解本书后续内容的基础。

> 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:

Python的用户通常都是被它的易用性吸引来的，其中很重要一环就是动态类型。静态类型的语言，例如C或者Java，每个变量都需要明确声明，而动态类型语言如Python就略过了这个部分。例如，在C中，你可能会写如下的代码片段：

```c
int result = 0;
for(int i=0; i<100; i++){
    result += i;
}
```

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

但是在Python当中，等效的代码如下：

```python
result = 0
for i in range(100):
    result += i
```

> Notice the 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:

注意其中主要的区别：在C当中，每个变量都需要显式声明，Python的类型是动态推断的。这意味着，我们可以给任何的变量赋值为任何类型的数据，例如：

```python
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 unintented consequences:

上面的例子中我们将`x`变量的内容从一个整数变成了一个字符串。如果你想在C语言中这样做，取决于不同的编译器，可能会导致一个编译错误或者其他无法预料的结果。

```c
int x = 4;
x = "four";  // 编译错误
```

> This sort of flexibility is one piece 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 value; they also contain extra informatiinon about the type of the value. We'll explore this more in the sections that follow.

这种灵活性提供了Python和其他动态类型语言在使用上的简易性。但是，理解这里面的*工作原理*对于在Python中高效准确的学习和分析数据是非常重要的。Python的这种类型灵活性，实际上是付出了额外的存储代价的，变量不仅仅存储了数据本身，还需要存储其相应的类型。我们会在本节接下来的部分继续讨论。

A Python Integer Is More Than Just an Integer

Python的整数不仅仅是一个整数

> 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 3.4 source code, we find that the integer (long) type definition effectively looks like this (once the C macros are expanded):

标准的Python实现是使用C语言编写的。这意味着每个Python当中的对象都是一个伪装良好的C结构体，结构体内不仅仅包括它的值，还有其他的信息。例如，当我们在Python中定义了一个整数，比方说`x=10000`，`x`不仅仅是一个原始的整数，它在底层实际上是一个指向复杂C结构体的指针，里面含有若干个字段。当你查阅Python 3.4的源代码的时候，你会发现整数（实际上是长整形）的定义如下（我们将C语言中的宏定义展开后）：

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

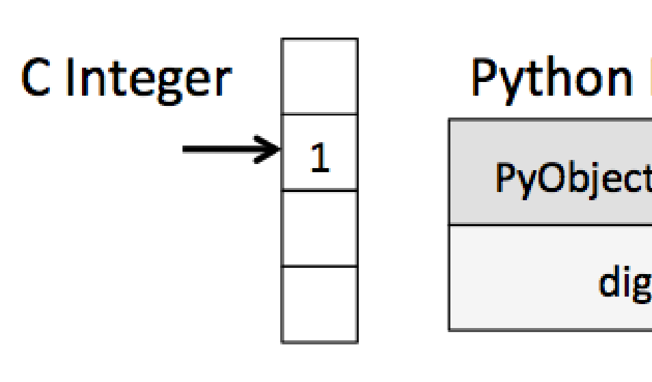
> A single integer in Python 3.4 actually contains four pieces:
>
> - `ob_refcnt`, a reference count that helps Python silently handle memory allocation and deallocation
> - `ob_type`, which encodes the type of the variable
> - `ob_size`, which specifies the size of the following data members
> - `ob_digit`, which contains the actual integer value that we expect the Python variable to represent.

一个Python的整数实际上包含四个部分：

- `ob_refcnt`：引用计数器，Python用这个字段来进行内存分配和垃圾收集
- `ob_type`：变量类型的编码内容
- `ob_size`：表示下面的数据字段的长度
- `ob_digit`：真正的整数值存储在这个字段

> 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:

这意味着在Python中存储一个整数要比在像C这样的编译语言中存储一个整数要有损耗，就像下图展示的那样：



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

这里的`PyObject_HEAD`代表了前面的引用计数器、类型代码和数据长度的三个字段内容。

> 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.

再次注意一下这里的区别：C的整数就是简单一个内存位置，这个位置上的固定长度的字节可以表示一个整数；Python中的一个整数是一个指向内存位置的指针，该内存位置包括Python需要表示一个整数的所有信息，其中最后固定长度的字节才真正存储这个整数。这些额外的信息提供了Python的灵活性和易用性。这些Python类型需要的额外信息是有额外损失的，特别是当有一个集合需要存储许多这种类型的数据时。

A Python List Is More Than Just a List

Python的列表不仅仅是一个列表

> 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:

现在我们继续考虑当我们使用Python的数据结构来存储许多这样的Python对象时的情况。Python中标准的可变多元素的容器集合就是列表。我们按如下的方式创建一个整数的列表：

```
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:

又或者，类似的，字符串的列表：

```
In [4]: L2 = [str(c) for c in L] # 列表解析
        L2
Out[4]: ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
```

```
In [5]: type(L2[0])
Out[5]: str
```

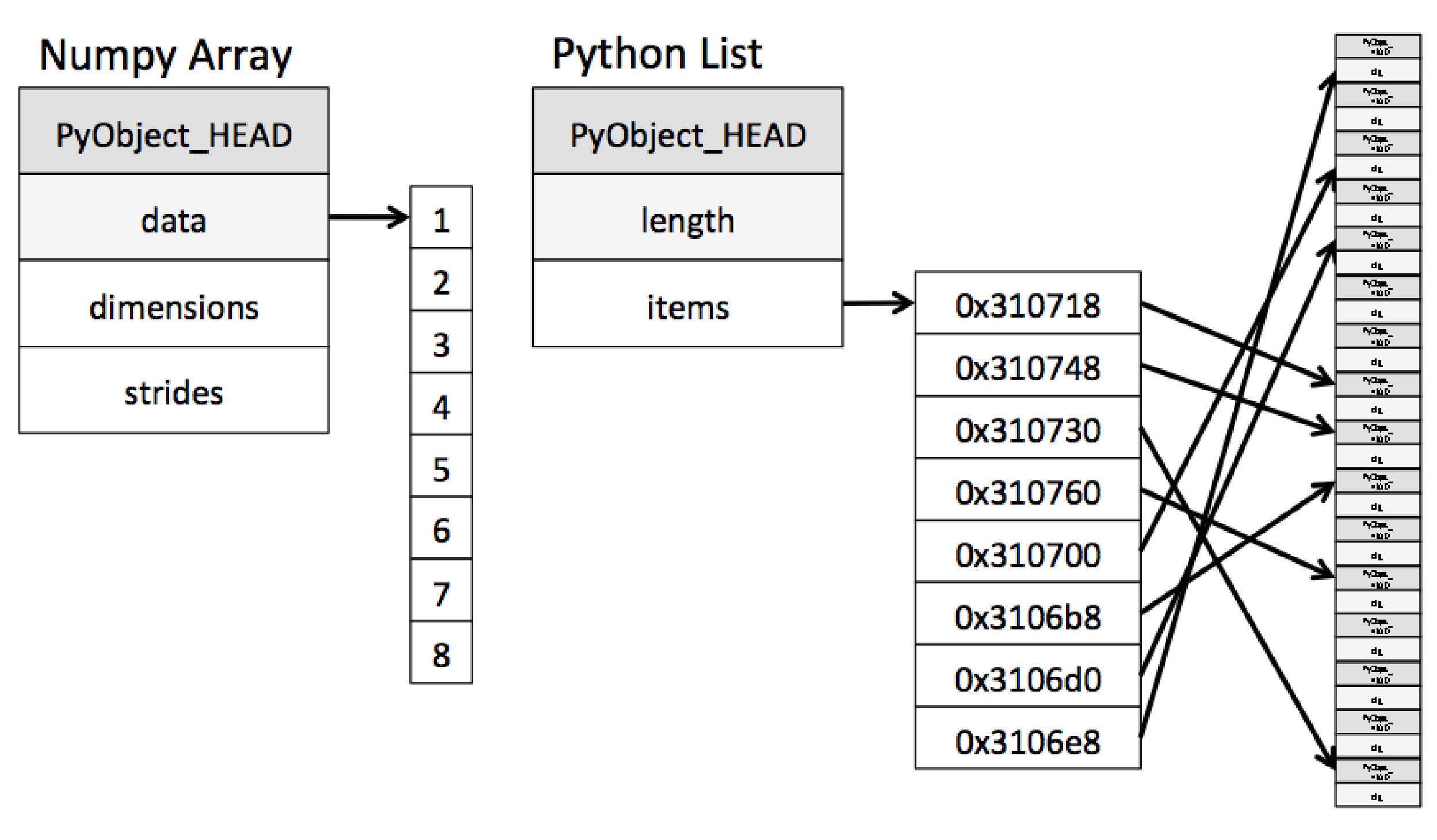
> Because of Python's dynamic typing, we can even create heterogeneous lists:

因为Python是动态类型，我们甚至可以创建不同类型元素的列表：

```
In [6]: L3 = [True, "2", 3.0, 4]
        [type(item) for item in L3]
Out[6]: [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:

这种灵活性是要付出代价的：要让列表能够容纳不同的类型，每个列表中的元素都必须带有自己的类型信息、引用计数器和其他的信息，一句话，里面的每个元素都是一个完整的Python的对象。如果在所有的元素都是同一种类型的情况下，这里面绝大部分的信息都是冗余的：如果我们能将数据存储在一个固定类型的数组中，显然会更加高效。下图展示了动态类型的列表和固定类型的数组（NumPy实现的）的区别：



> 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.

从底层实现上看，数组仅仅包含一个指针指向一块连续的内存空间。而Python列表，含有一个指针指向一块连续的指针内存空间，里面的每个指针再指向内存中每个独立的Python对象，如我们前面看到的整数。列表的优势在于灵活：因为每个元素都是完整的Python的类型对象结构，包含了数据和类型信息，因此列表可以存储任何类型的数据。NumPy使用的固定类型的数组缺少这种灵活性，但是对于存储和操作数据会高效许多。

Fixed-Type Arrays in Python

Python的固定类型数组

> 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:

Python提供了许多不同的选择能让你高效的存储数据，使用固定类型数据。內建的`array`模块（从Python 3.3开始提供）可以用来创建同一类型的数组：

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

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

这里的`i`是表示数据类型是整数的类型代码。

> 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.

更常用的是`ndarray`对象，由NumPy包提供。虽然Python的`array`提供了数组的高效存储，NumPy更加提供了数组的高效*运算*。我们会在后续小节中陆续介绍这些操作；这里我们首先介绍创建NumPy数组的集中方式。

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

当然最开始要做的是将NumPy包载入，惯例上提供别名`np`：

```
In [8]: import numpy as np
```

Creating Arrays from Python Lists

使用Python列表创建数组

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

首先，我们可以使用`np.array`来将一个Python列表变成一个数组：

```
In [9]: # 整数数组:
        np.array([1, 4, 2, 5, 3])
Out[9]: 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):

记住和Python列表不同，NumPy数组只能含有同一种类型的数据。如果类型不一样，NumPy会尝试向上扩展类型（下面例子中会将整数向上扩展为浮点数）：

```
In [10]: np.array([3.14, 4, 2, 3])
Out[10]: 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:

如果你需要明确指定数据的类型，你可以使用`dtype`关键字参数：

```
In [11]: np.array([1, 2, 3, 4], dtype='float32')
Out[11]: 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:

最后，不同于Python的列表，NumPy的数组可以明确表示为多维；下面例子是一个使用列表的列表来创建二维数组的方法：

```
In [13]: # 更准确的说，应该是生成器的列表，列表解析中有三个range生成器
         # 分别是range(2, 5), range(4, 7) 和 range(6, 9)
         np.array([range(i, i + 3) for i in [2, 4, 6]])
Out[13]: array([[2, 3, 4],
                [4, 5, 6],
                [6, 7, 8]])
```

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

内部的列表作为二维数组的行。

Creating Arrays from Scratch

从头开始创建数组

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

使用NumPy的方法从头创建数组会更加高效，特别对于大型数组来说。下面有几个例子：

```
In [14]: # zeros将数组元素都填充为0，10是数组长度
         np.zeros(10, dtype=int)
Out[14]: array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

```
In [15]: # ones将数组元素都填充为1，(3, 5)是数组的维度说明，表明数组是二维的3行5列
         np.ones((3, 5), dtype=float)
Out[15]: array([[1., 1., 1., 1., 1.],
                [1., 1., 1., 1., 1.],
                [1., 1., 1., 1., 1.]])
```

```
In [16]: # full将数组元素都填充为参数值3.14，(3, 5)是数组的维度说明，表明数组是二维的3行5列
         np.full((3, 5), 3.14)
Out[16]: 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 [17]: # arange类似range，创建一段序列值
         # 起始值是0（包含），结束值是20（不包含），步长为2
         np.arange(0, 20, 2)
Out[17]: array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])
```

```
In [18]: # linspace创建一段序列值，其中元素按照区域进行线性（平均）划分
         # 起始值是0（包含），结束值是1（包含），共5个元素
         np.linspace(0, 1, 5)
Out[18]: array([0.  , 0.25, 0.5 , 0.75, 1.  ])
```

```
In [19]: # random.random随机分布创建数组
         # 随机值范围为[0, 1)，(3, 3)是维度说明，二维数组3行3列
         np.random.random((3, 3))
Out[19]: array([[0.28957547, 0.80872794, 0.36451325],
                [0.30178461, 0.13998063, 0.21693246],
                [0.81413802, 0.26299406, 0.53082583]])
```

```
In [18]: # random.normal正态分布创建数组
         # 均值0，标准差1，(3, 3)是维度说明，二维数组3行3列
         np.random.normal(0, 1, (3, 3))
Out[18]: array([[ 1.51772646,  0.39614948, -0.10634696],
                [ 0.25671348,  0.00732722,  0.37783601],
                [ 0.68446945,  0.15926039, -0.70744073]])
```

```
In [19]: # random.randint随机整数创建数组，随机数范围[0, 10)
         np.random.randint(0, 10, (3, 3))
Out[19]: array([[2, 3, 4],
                [5, 7, 8],
                [0, 5, 0]])
```

```
In [20]: # 3x3的单位矩阵数组
         np.eye(3)
Out[20]: array([[ 1.,  0.,  0.],
                [ 0.,  1.,  0.],
                [ 0.,  0.,  1.]])
```

```
In [20]: # empty创建一个未初始化的数组，数组元素的值保持为原有的内存空间值
         np.empty(3)
Out[20]: array([5.74020278e+180, 4.00193173e-322, 4.66594353e-310])
```

NumPy Standard Data Types

NumPy标准数据类型

> 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.

NumPy数组仅包含一种类型数据，因此它的类型系统和Python也有所区别，因为对于每一种NumPy类型，都需要更详细的类型信息和限制。因为NumPy是使用C构建的，它的类型系统对于C、Fortran的用户来说不会陌生。

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

标准NumPy数据类型见下表。正如上面介绍的，当我们创建数组的时候，我们可以将`dtype`参数指定为下面类型的字符串名称来指定数组的数据类型。

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

> Or using the associated NumPy object:

也可以将`dtype`指定为对应的NumPy对象：

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

Data type	Description
`bool_`	布尔(True 或 False) 一个字节
`int_`	默认整数类型 (类似C的`long`; 通常可以是`int64`或`int32`)
`intc`	类似C的`int` (通常可以是`int32`或`int64`)
`intp`	用于索引值的整数(类似C的`ssize_t`; 通常可以是`int32`或`int64`)
`int8`	整数，1字节 (-128 ～ 127)
`int16`	整数，2字节 (-32768 ～ 32767)
`int32`	整数，4字节 (-2147483648 ～ 2147483647)
`int64`	整数，8字节 (-9223372036854775808 ～ 9223372036854775807)
`uint8`	字节 (0 ～ 255)
`uint16`	无符号整数 (0 ～ 65535)
`uint32`	无符号整数 (0 ～ 4294967295)
`uint64`	无符号整数 (0 ～ 18446744073709551615)
`float_`	`float64`的简写
`float16`	半精度浮点数: 1比特符号位, 5比特指数位, 10比特尾数位
`float32`	单精度浮点数: 1比特符号位, 8比特指数位, 23比特尾数位
`float64`	双精度浮点数: 1比特符号位, 11比特指数位, 52比特尾数位
`complex_`	`complex128`的简写
`complex64`	复数, 由2个单精度浮点数组成
`complex128`	复数, 由2个双精度浮点数组成

> More advanced type specification is possible, such as specifying big or little endian numbers; for more information, refer to the NumPy documentation. NumPy also supports compound data types, which will be covered in Structured Data: NumPy's Structured Arrays.

还有更多的高级的类型声明，比如指定大尾或小尾表示；需要获得更多内容，请查阅NumPy在线文档。NumPy也支持复合数据类型，这部分我们将在结构化数据：NumPy里的结构化数组中进行介绍

< Numpy介绍 | 目录 | Numpy数组基础 >

Open in Colab