Theory: Data types in NumPy

21 minutes 0 / 5 problems solved Skip this topic Start practicing

286 users solved this topic. Latest completion was about 9 hours ago.

In this topic, we are going to get acquainted with **NumPy data types**. You have already learned some basic data types provided in Python: `string` for text data, `float` for floating-point numbers, etc. NumPy and Python data types share some common features, but at the same time, NumPy data has some unique peculiarities. We will discuss the basic NumPy data types and methods of working with them.

§1. Types of data in NumPy

Imagine that you strive to know everything about all data types in NumPy and you wonder how to do it. In this case `np.sctypeDict` can help you. It prints a dictionary in which all the keys are different data types.

```
print(np.sctypeDict)

# {'?': <class 'numpy.bool_'>, 0: <class 'numpy.bool_'>, 'byte': <class 'numpy.int8'>, 'b': <class 'numpy.int8'>, 1: <class 'numpy.int8'>, 'ubyte': <class 'numpy.uint8'>, ...}
```

Of course, such a dictionary is not the most convenient way to explore the data types. The summary of the main types with their brief descriptions is shown below. The most widely-used types are the ones that represent integers and floating numbers, but there are other types, too. It is worth mentioning that NumPy data types are the same as in C programming language because NumPy itself is a wrapper on C, which makes NumPy operations so fast.

1. Integer data

For integer numbers, there are a lot of specific data types in NumPy.

Data type	Description
`int_` or `int32` or `int64`	Integer type set by default (depends on a user's operation system)
`int8`	Integers ranging from -128 to 127
`int16`	Integers ranging from -32 768 to 32 767
`int32`	Integers ranging from -2 147 483 648 to 2 147 483 647
`int64`	Integers ranging from -9 223 372 036 854 775 808 to 9 223 372 036 854 775 807
`uint8`	Non-negative integers ranging from 0 to 255
`uint16`	Non-negative integers ranging from 0 to 65 535
`uint32`	Non-negative integers ranging from 0 to 4 294 967 295
`uint64`	Non-negative integers ranging from 0 to 18 446 744 073 709 551 615

Mind different ranges of numbers. As we have mentioned, NumPy specifies the size of the data: the thing is, one byte (or 8 bits) is required to decode integers from `-128` to `127`. If we take only positive ones, the range is from `0` to `255`, respectively. For larger integers, we need two bytes (16 bits), four bytes (32 bits), or eight bytes (64 bits).

The range thresholds are determined by different powers of two. In case of `int8`, it is $2^{(8-1)}$ = 128. One bit is reserved for the negative / positive sign, so it's 2^7 rather than 2^8. Also, one place is taken by zero, so the upper boundary is 128 - 1 = 127. Similarly, for the upper limit for `int64`, 9 223 372 036 854 775 807 = $2^{(64-1)}$ - 1.

Current topic:

Data types in NumPy

Topic depends on:

- ✓ List 13 Stage 1
- ✓ Dictionary 6 Stage 1
- ✗ Intro to NumPy

Table of contents:

It is important to note that `-3`, for instance, can have either `int8`, `int16`, `int32`, or `int64` data type. On the other hand, `3` can represent any of `int` and `uint` formats, it all depends on what you are working with.

2. Float data

For float values, there are the following data types:

Data type	Description
`float16`	Half-Precision Floating-Point Format (2 Bytes)
`float32`	Single-Precision Floating-Point Format (4 bytes)
`float_` or `float64`	Double-Precision Floating-Point Format (8 bytes)
`float128`	Quadruple-Precision Floating-Point Format (16 bytes)

It is necessary to clarify the types of precisions. Half-precision floating-point format provides results that are not that accurate. If you want to obtain more decimal spaces, you need to use the single-precision, double-precision, or quadruple-precision formats. It is shown in the example below.

```
n = 22/7
print(np.float16(n))  # 3.143
print(np.float32(n))  # 3.142857
print(np.float64(n))  # 3.142857142857143
```

For other purposes, you can use boolean, complex, or string data. `bool_` is for boolean data in NumPy, it works the same way as the boolean type in Python. For complex numbers, there are several data types: `complex64`, `complex_` (`complex128`), and `complex256`. String data is represented by `str_` (standard Python string) and `unicode_` (Unicode string), and bytes as `bytes_` (a sequence of encoded bytes which is ready to be stored in the memory of a program). You can read more about these data types in the official NumPy documentation.

§2. Specifying data types

In NumPy, the `np.dtype` class stores different objects of the data type. Below there are some sample queries:

```
print(np.float64)  # <class 'numpy.float64'>
print(np.str_)     # <class 'numpy.str_'>
```

To find out the data type of an existing array, use `array.dtype`:

```
array_1 = np.array([[5, 6], [15, 0]])
array_2 = np.array([[2.5, 5.7], [1.2, 6.9]])
array_3 = np.array([[2, 34], [78.9, 4.5]])
print(array_1.dtype)  # int64
print(array_2.dtype)  # float64
print(array_3.dtype)  # float64
```

Be aware of the `array_3` variable. It contains an array with different types of data — integers and floats. When combining data, NumPy will convert all the data into one type, so that all the integers become floating numbers.

When creating an array, you can specify the type of data in the array with the help of the `dtype` argument:

```
array_4 = np.array([[1, 2], [3, 4]], dtype=np.float32)
print(array_4)
# [[1. 2.]
#  [3. 4.]]

array_5 = np.arange(7, dtype=np.float64)
print(array_5)  # [0. 1. 2. 3. 4. 5. 6.]

array_6 = np.arange(7, dtype=np.int64)

print(array_6)  # [0 1 2 3 4 5 6]
```

§3. Changing data types

To transform one data type into another, use the `np.<type>` function:

```
print(np.bool(1))  # True
print(np.float64(35))  # 35.0
```

If you want to do it with an array rather than with single values, use the `array.astype()` function. It copies the array and converts it into the specified type.

```
array_7 = np.array([[-129, 45], [34, 129]], dtype=np.int64)
print(array_7)
# [[-129   45]
#  [  34  129]]

print(array_7.astype(np.float64))
# [[-129.   45.]
#  [  34.  129.]]


print(array_7.astype(np.str_))

# [['-129' '45']

#  ['34' '129']]



print(array_7.astype(np.bool_))

# [[ True  True]

#  [ True  True]]
```

In the example above, you can see that we can easily transform our `array_7` into float, string, or boolean data. However, if we try to change the type to `int8`, a problem will arise:

```
print(array_7.astype(np.int8))
# [[ 127   45]
#  [  34 -127]]
```

As you can remember, `int8` can store integers from `-128` to `127` that take up to one byte. When we try to convert into `int8` numbers that are not in its range, NumPy will not display a memory error, but instead, it just starts counting from the very beginning. `-129` was changed to `127`, the first *positive* integer. Similarly, `129` was changed to the second negative integer.

What happens in the example above, is called an **integer overflow**. It occurs when an arithmetic operation attempts to create a number outside of the range that can be represented within a given data type. Be careful with this, it can lead to errors!

§4. Conclusion

NumPy provides a great number of data types compared to basic Python data types. In this topic we've learned:

- main types of data in NumPy;
- how to specify data types when you create arrays;
- how to print the data type of an existing array;
- the way of changing data types in arrays.

The next step is practicing your skills!