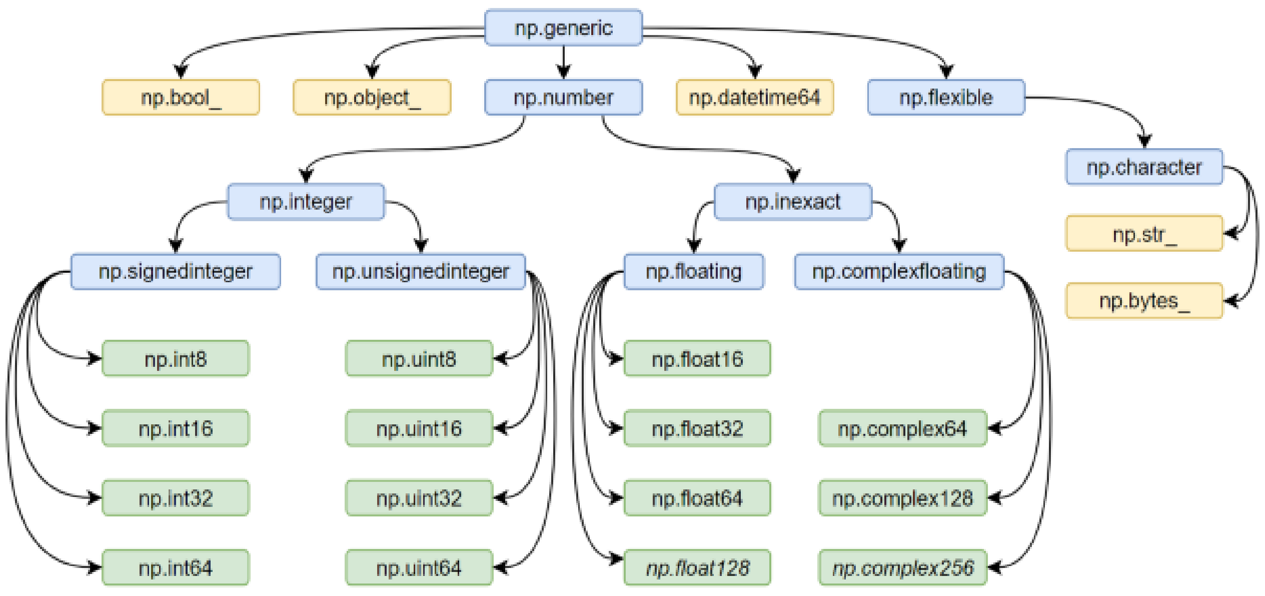
Lev Maximov Follow
Dec 29, 2021 · 17 min read ★ · Listen

# A Comprehensive Guide to NumPy Data Types

What else is out there besides int32 and float64?



All images credit: Author

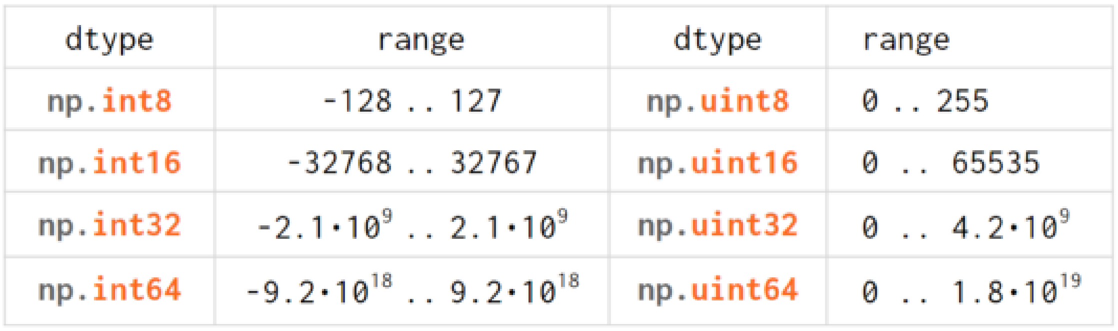
NumPy, one of the most popular Python libraries for both data science and scientific computing, is pretty omnivorous when it comes to data types.

It has its own set of 'native' types which it is capable of processing at full speed, but it can also work with pretty much *anything* known to Python.

The article is written as a supplement to my NumPy Illustrated guide and is divided

## 1. Integers

The integer types table in NumPy is trivial for anyone with minimal experience in C/C++:



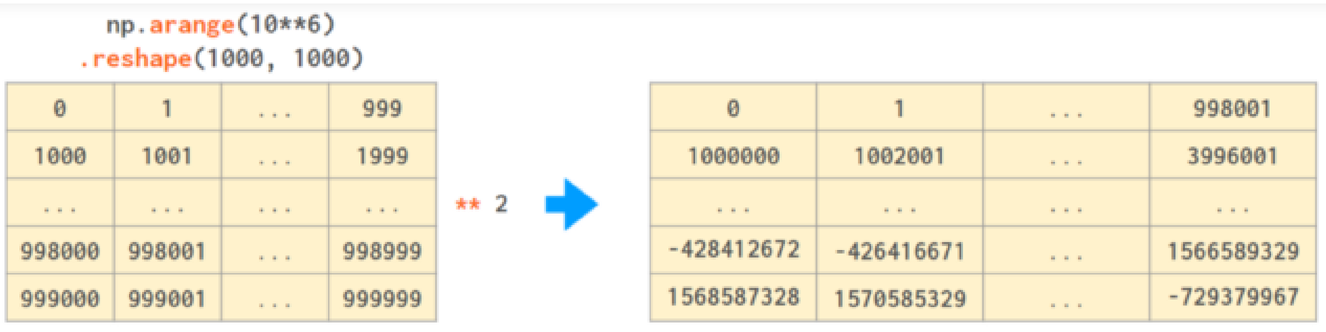
| dtype | range | dtype | range |
|---|---|---|---|
| np.int8 | -128 .. 127 | np.uint8 | 0 .. 255 |
| np.int16 | -32768 .. 32767 | np.uint16 | 0 .. 65535 |
| np.int32 | $-2.1\cdot10^{9}$ .. $2.1\cdot10^{9}$ | np.uint32 | 0 .. $4.2\cdot10^{9}$ |
| np.int64 | $-9.2\cdot10^{18}$ .. $9.2\cdot10^{18}$ | np.uint64 | 0 .. $1.8\cdot10^{19}$ |

Just like in C/C++, 'u' stands for 'unsigned' and the digits represent the number of bits used to store the variable in memory (eg `np.int64` is an 8-bytes-wide signed integer).

When you feed a Python `int` into NumPy, it gets converted into a native NumPy type called `np.int32` (or `np.int64` depending on the OS, Python version, and the magnitude of the initializers):

```
>>> np.array([1, 2, 3]).dtype
dtype('int32')                # int32 on Windows, int64 on Linux and MacOS
```

If you're unhappy with the flavor of the integer type that NumPy has chosen for you, you can specify one explicitly via the 'dtype' (=data type) argument which accepts either a dtype object `np.array([1,2,3], np.uint8)` or a string `np.array([1,2,3],`

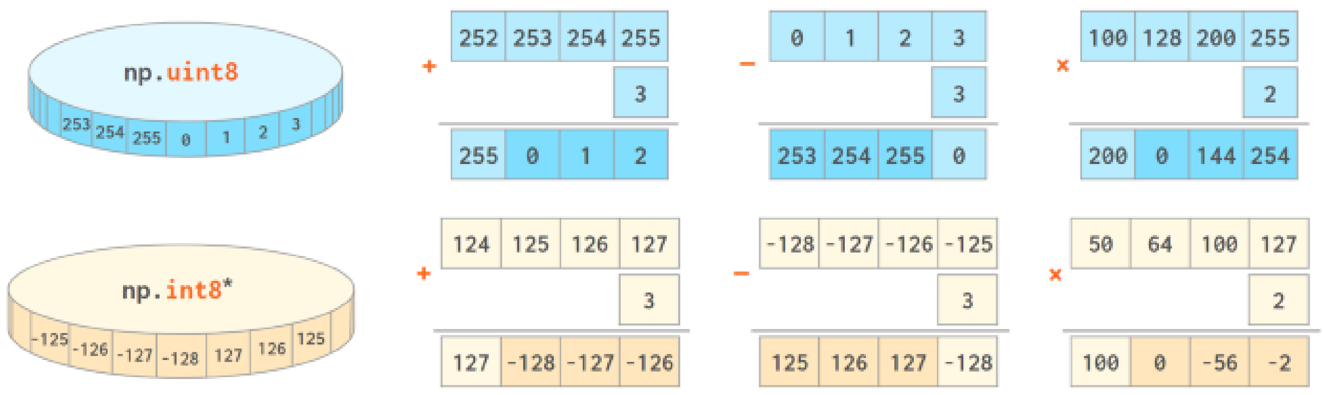


| | | | |
|---|---|---|---|
| 0 | 1 | ... | 999 |
| 1000 | 1001 | ... | 1999 |
| ... | ... | ... | ... |
| 998000 | 998001 | ... | 998999 |
| 999000 | 999001 | ... | 999999 |

| | | | |
|---|---|---|---|
| 0 | 1 | ... | 998001 |
| 1000000 | 1002001 | ... | 3996001 |
| ... | ... | ... | ... |
| -428412672 | -426416671 | ... | 1566589329 |
| 1568587328 | 1570585329 | ... | -729379967 |

What's going on here?

NumPy works best when the width of the array elements is fixed. It is faster and takes less memory, but unlike an ordinary Python `int` (that works in arbitrary precision arithmetic), the values of an array will wrap when they cross the maximum (or minimum) value for the corresponding data type:



* Strictly speaking, the C standard defines this wraparound only for **unsigned** integers; the overflow behavior for **signed** integers is undefined and can't be relied upon (in both C and NumPy). Signed integers are silently wrapped around now, but there's no guarantee they always will.

```
>>> np.array([255], np.uint8) + 1    # 2**8-1 is INT_MAX for uint8
array([0], dtype=uint8)

>>> np.array([2**31-1])              # 2**31-1 is INT_MAX for int32
array([2147483647])

>>> np.array([2**31-1]) + 1          # or np.array([2**31-1], np.int32)+1 on linux
array([-2147483648])

>>> np.array([2**63-1]) + 1          # always np.int64 since v > 2**32-1
array([-9223372036854775808])
```

int_wrapping.py

With scalars, it is a different story: first NumPy tries its best to promote the value to a wider type, then, if there is none, fires the overflow warning (to avoid flooding the output with warnings — only once):

```
>>> np.array([255], np.uint8)[0] + 1   # ok, promoted to int32(win)/int64(linux)
256
>>> np.array([2**31-1])[0] + 1         # warning!
RuntimeWarning: overflow encountered in long_scalars
-2147483648
>>> np.array([2**63-1])[0] + 1         # ok, warned already
-9223372036854775808
```

overflow_warning.py

The reasoning behind such a discrimination is like this:

> *Unlike true floating point errors (where the hardware FPU sets a flag whenever it does an atomic operation that overflows), we need to implement the integer overflow detection ourselves. We do it on the scalars, but not arrays because it would be too slow to implement for every atomic operation on arrays. (Robert Kern, one of the NumPy core developers)*

You can turn it into an error:

```
>>> with np.errstate(over='raise'):
>>>     print(np.array([2**31-1])[0]+1)
FloatingPointError: overflow encountered in long_scalars
```

overflow_error.py

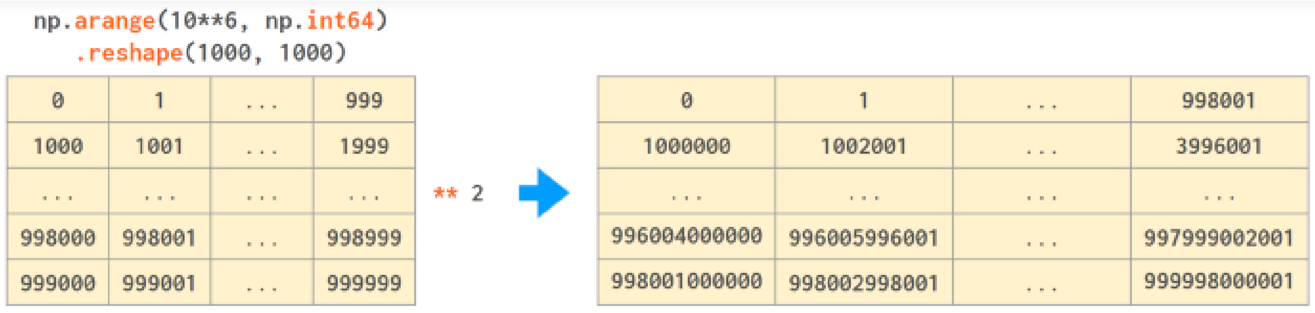
or suppress it temporarily:

```
>>> with np.errstate(over='ignore'):
>>>     print(np.array([2**31-1])[0]+1)
-2147483648
```

overflow_ignored.py

Or completely: `np.warnings.filterwarnings('ignore', 'overflow')`

But you can't expect it to be detected when dealing with any arrays.



```
np.arange(10**6, np.int64)
  .reshape(1000, 1000)
```

|  |  |  |  |
|---|---|---|---|
| 0 | 1 | ... | 999 |
| 1000 | 1001 | ... | 1999 |
| ... | ... | ... | ... |
| 998000 | 998001 | ... | 998999 |
| 999000 | 999001 | ... | 999999 |

|  |  |  |  |
|---|---|---|---|
| 0 | 1 | ... | 998001 |
| 1000000 | 1002001 | ... | 3996001 |
| ... | ... | ... | ... |
| 996004000000 | 996005996001 | ... | 997999002001 |
| 998001000000 | 998002998001 | ... | 999998000001 |

On Linux/MacOS it is np.int64 by default

NumPy also has a bunch of C-style aliases (eg. `np.byte` is `np.int8`, `np.short` is `np.int16`, `np.intc` is an int with whichever width int type has in C, etc), but they are getting gradually phased out (eg deprecation of np.long in NumPy v1.20.0) as 'explicit is better than implicit' (but see a present-day usage of `np.longdouble` below).

And yet some more exotic aliases:

- `np.int_` is `np.int32` on 64bit Windows but `np.int64` on 64bit Linux/MacOS, used to designate the 'default' int. Specifying `np.int_` (or just int) as a dtype means '*do what you would do if I didn't specify any dtype at all*': `np.array([1,2])`, `np.array([1,2], np.int_)` and `np.array([1,2], int)` are all the same thing.

- `np.intp` is `np.int32` on 32bit Python but `np.int64` on 64bit Python, ≈ `ssize_t` in C, used in Cython as a type for pointers.

Occasionally it happens that some values in the array display anomalous behavior or missing, and you want to process the array without deleting them (eg there's some valid data in other columns).

You can't put `None` there because it doesn't fit in the consecutive `np.int64` values and also because `1+None` is an unsupported operation.

Pandas has a separate data type for that, but NumPy's way of dealing with the missed values is through the so-called masked array: you mark the invalid values with a boolean mask and then all the operations are carried out as if the values are not there.

```
>>> np.array([4,0,6]).mean()          # the value 0 means 'missing' here
```

masked_array.py

Finally, if for some reason you need arbitrary-precision integers (Python `int` s) in ndarrays, numpy is capable of doing it, too:

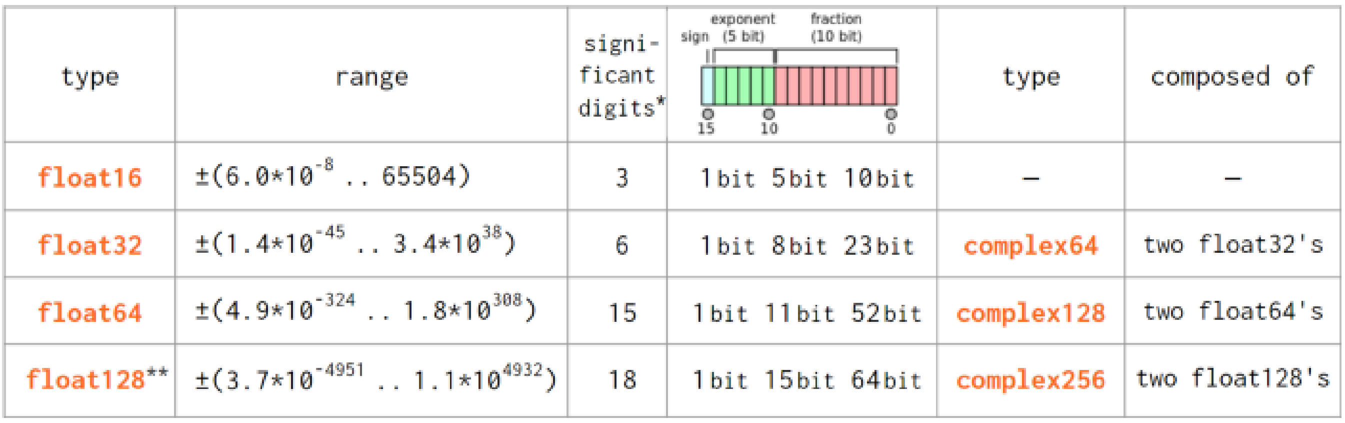
```
>>> a = np.array([10], dtype=object)
>>> len(str(a**1000))                 # '[1000...0]'
1003
```

int_type.py

— but without the usual speedup as it will store references instead of the numbers themselves, keep boxing/unboxing Python objects when processing, etc.

## 2. Floats

As pure Python `float` did not diverge from the IEEE 754-standardized C `double` type (note the difference in naming), the floating point numbers transition from Python to NumPy is pretty much hassle-free: Python `float` is directly compatible with `np.float64` and Python `complex` — with `np.complex128` .


sign
exponent (5 bit)
fraction (10 bit)
15
10
0

| type | range | signi-ficant digits* | | type | composed of |
|---|---|---|---|---|---|
| float16 | $\pm(6.0*10^{-8}$ .. 65504) | 3 | 1bit 5bit 10bit | – | – |
| float32 | $\pm(1.4*10^{-45}$ .. $3.4*10^{38})$ | 6 | 1bit 8bit 23bit | complex64 | two float32's |
| float64 | $\pm(4.9*10^{-324}$ .. $1.8*10^{308})$ | 15 | 1bit 11bit 52bit | complex128 | two float64's |
| float128** | $\pm(3.7*10^{-4951}$ .. $1.1*10^{4932})$ | 18 | 1bit 15bit 64bit | complex256 | two float128's |

** As reported by `np.finfo(np.float<nn>).precision`. Two alternative definitions give 15 and 17 digits for `np.float64`, 6 and 9 for `np.float32`, etc.*

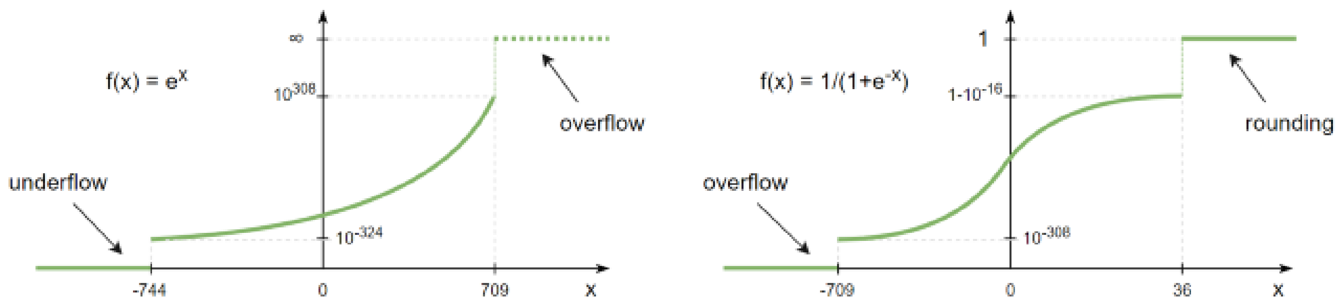
*** As of today, `np.float128` is Unix-only (not available on Windows).*

Suppose you're calculating a sigmoid activation function of the array and one of its elements happens to be

```
>>> x = np.array([-1234.5])
>>> 1/(1+np.exp(-x))
RuntimeWarning: overflow encountered in exp
array([0.])
>>> np.exp(np.array([1234.5]))
RuntimeWarning: overflow encountered in exp
array([inf])
```

float_overflow.py

What this warning is trying to tell you is that NumPy is aware that mathematically speaking `1/(1+exp(-x))` can never be zero, but in this particular case due an overflow it is.



Such warnings can be 'promoted' to exceptions or silenced via the `errstate` or `filterwarnings` as described in the 'integers' section above — and maybe for this particular case that would be enough — but if you *really* want to get the exact value you can select a wider `dtype` :

```
>>> x = np.array([-1234.5], dtype=np.float128)
>>> 1/(1+np.exp(-x))
array([7.30234068e-537], dtype=float128)
```

float128.py: np.float128 is currently available on Linux/MacOS, not on Windows

One thing that distinguishes floats from integers is that they are *inexact*. You can't compare two floats with `a == b` , unless you're sure they are represented *exactly*. You can expect floats to exactly represent integers — but only below a certain level (limited

```
>>> a = np.array([2**24], np.float32); a     # 2^(mantissa_bits+1)
array([16777216.], dtype=float32)
>>> a+1
array([16777216.], dtype=float32)
>>> 9279945539648888.0+1     # for float64 it is 2.**53
9279945539648888.0
>>> len('9279945539648888') # Don't trust the 16th decimal digit!
16
```

max_consecutive_int.py

Also exactly representable are fractions like 0.5, 0.125, 0.875 where the denominator is a power of 2 (0.5=1/2, 0.125=1/8, 0.875 =7/8, etc). Any other denominator will result in a rounding error so that 0.1+0.2!=0.3.

The standard approach of dealing with this problem (as well as with source #2 of inexactness: rounding of the results of the calculations) is to compare them with a relative tolerance (to compare two non-zero arguments) and absolute tolerance (if one of the arguments is zero). For scalars, it is handled by `math.isclose(a, b, *, rel_tol=1e-09, abs_tol=0.0)` , for NumPy arrays there's a vector version `np.isclose(a, b, rtol=1e-05, atol=1e-08)` . Note that the tolerances have different names and defaults.

For the financial data `decimal.Decimal` type is handy as it involves no additional tolerances at all:

```
>>> from decimal import Decimal as D
>>> a = np.array([D('0.1'), D('0.2')]); a
array([Decimal('0.1'), Decimal('0.2')], dtype=object)
>>> a.sum()                    # == Decimal('0.3'), exactly
Decimal('0.3')
```

decimal.py

But it is not a silver bullet: it also has rounding errors (see source #2 above). The only problem it solves is the exact representation of decimal numbers that humans are used to.

Plus it doesn't support anything more complicated than arithmetic operations (though

```
>>> from fractions import Fraction
>>> a = np.array([1, 2]) + Fraction(); a
array([Fraction(1, 1), Fraction(2, 1)], dtype=object)
>>> a/=10; a
array([Fraction(1, 10), Fraction(1, 5)], dtype=object)
>>> a.sum()
Fraction(3, 10)
```

fractions.py

It can exactly represent any rational numbers *and* is not subject to rounding errors during the calculations (source #2) but π and *e* are out of luck! If you need them, too, then SymPy is your friend.

Both `Decimal` and `Fraction` are not native types for NumPy but it is capable of working with them with all the niceties like multi-dimensions and fancy indexing, albeit at the cost of slower processing speed than that of native `int` s or `float` s.

Complex numbers are treated the same way as floats. There are extra convenience functions with intuitive names like np.real(z), np.imag(z), np.abs(z), np.angle(z) that work on both scalars and arrays as a whole. The only difference from the pure Python `complex` , `np.complex_` does not work with integers:

```
>>> np.array([1+2j])                # .dtype == np.complex128
array([1.+2.j])
```

int_complex.py

Just like with the integers, in float (and complex) arrays it is also sometimes useful to treat certain values as 'missing'. Floats are better suited for storing anomalous data: they have a `math.nan` (or `np.nan` or `float('nan')` ) value which can be stored inline with the 'valid' numeric values.

But `nan` is contagious in the sense that all the arithmetic with `nan` results in `nan`. Most common statistical functions have a nan-resistant version (np.nansum, np.nanstd, etc), but other operations on that column or array would require prefiltering. Masked arrays automate this step: the mask can only be built once, then it is 'glued' to the original array so that all subsequent operations only see the unmasked values and operate on

```
>>> a = np.array([4., np.nan, 6.])
>>> a.mean()
nan
>>> a.nanmean()
5.0
>>> a[~np.isnan(a)].mean()
5.0
>>> ma.array(a, mask=[0,1,0]).mean() # nan is not required here, could be anything
5.0
```

masked_floats.py

Also the names `float96` / `float128` are somewhat misleading. Under the hood it is not `__float128` but whichever `longdouble` means in the local C++ flavor. On x86_64 Linux it is `float80` (padded with zeros for memory alignment) which is certainly wider than `float64` , but it comes at the cost of the processing speed. Also you risk losing precision if you inadvertently convert to Python `float` type. For better portability it is recommended to use an alias `np.longdouble` instead of `np.float96 / np.float128` because that's what will be used internally anyway.

More insights on floats can be found in the following sources:

- short and nicely illustrated 'Half precision floating point visualized[1]' (eg what's the difference between normal and subnormal numbers)
- more lengthy but very to-the-point, a dedicated website 'Floating point guide[2]' (eg why 0.1+0.2!=0.3)
- long-read, a deep and thorough 'What every computer scientist should know about floating-point arithmetic[3]' (eg what's the difference between catastrophic vs. benign cancellation)

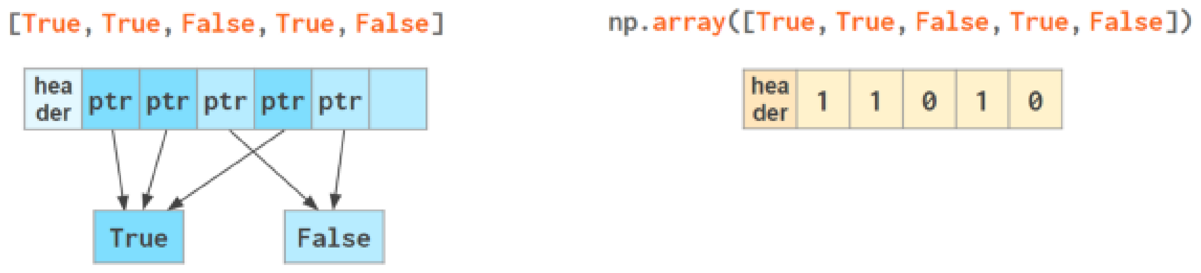
## 3. Bools

The boolean values are stored as single bytes for better performance. `np.bool_` is a separate type from Python's `bool` because it doesn't need reference counting and a link

```
>>> sys.getsizeof(True)
28
```

size_of_bool.py

`np.bool` is 28 times more memory efficient than Python's `bool` ) — though in real-world scenarios the rate is lower: when you pack NumPy bools into an array, they will take 1 byte each, but if you pack Python bools into a list it will reference the same two values every time, costing effectively 8 bytes per element on x86_64:


[True, True, False, True, False]
header
ptr ptr ptr ptr ptr
True
False
np.array([True, True, False, True, False])
header
1 1 0 1 0

The underlines in `bool_`, `int_`, etc are there to avoid clashes with Python's types. It's a bad idea to use reserved keywords for other things, but in this case it has an additional advantage of allowing (a generally discouraged, but useful in rare cases) `from numpy import *` without shadowing Python `bool` s, `int` s, etc. As of today, `np.bool` still works but displays a deprecation warning.

## 4. Strings

Initializing a NumPy array with a list of Python strings packs them into a fixed-width native NumPy dtype called `np.str_`. Reserving a space necessary to fit the longest string for every element might look wasteful (especially in the fixed USC-4 encoding as opposed to 'dynamic' choice of the UTF width in Python `str` )

```
>>> np.array(['abcde', 'x', 'y', 'x'])          # 4 bytes per any character
array(['abcde', 'x', 'y', 'x'], dtype='<U5')   # => 5*4 bytes per element
```

dtype_str_.py

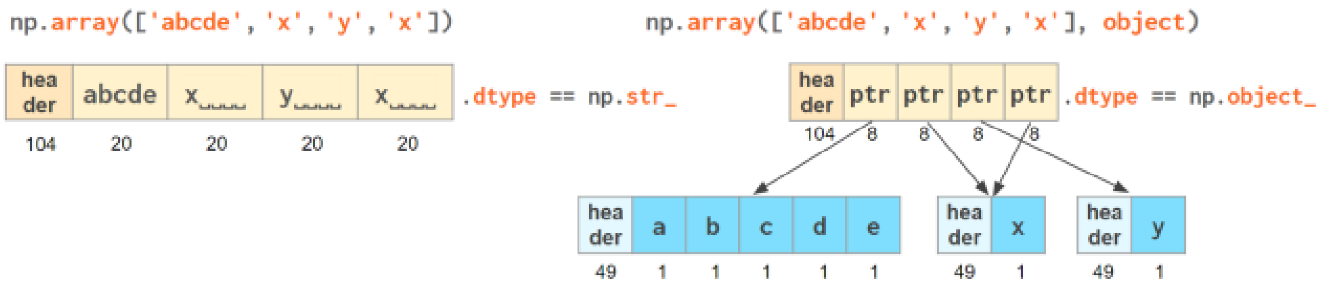
unreadable as this one — luckily have they adopted human-readable names at least for the most used dtypes.

Another option is to keep references to Python `str` s in a NumPy array of objects:

```
>>> np.array(['abcde', 'x', 'y', 'x'], object)      # 1 byte per ascii character
array(['abcde', 'x', 'y', 'x'], dtype=object)      # => 49+len(el) per element
```

dtype_str.py

The first array memory footprint amounts to 164 bytes, the second one takes 128 bytes for the array itself + 154 bytes for the three Python `str` s:



Depending on the relative lengths of the strings and the number of the repeated string either one approach can be a significant win or the other.

If you're dealing with a raw sequence of bytes NumPy has a fixed-length version of a Python `bytes` type called `np.bytes_` :

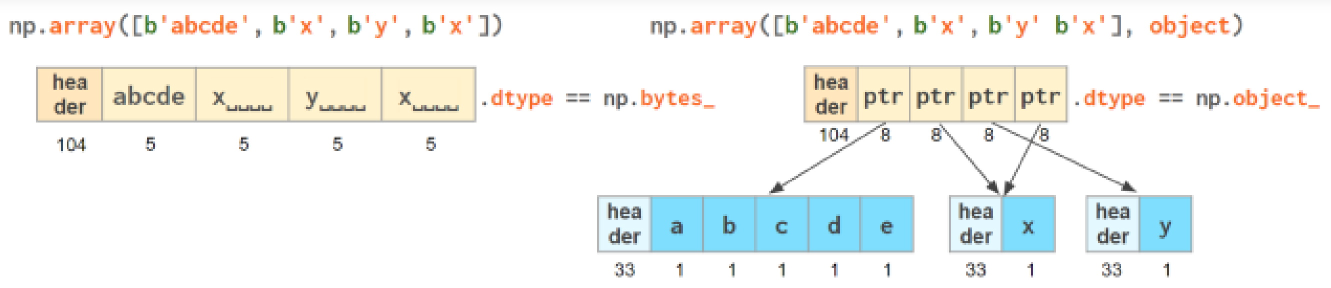
```
>>> np.array([b'abcde', b'x', b'y', b'x'])          # 1 byte per ascii character
array([b'abcde', b'x', b'y', b'x'], dtype='|S5')    # => 5 bytes per element
```

dtype_bytes_.py

Here `|S5` means 'endianness-unappliable sequence of bytes 5 elements long'.

Once again, an alternative is to store the Python `bytes` in the NumPy array of objects:

```
>>> np.array([b'abcde', b'x', b'y', b'x'], object)    # 1 byte per ascii character
array([b'abcde', b'x', b'y', b'x'], dtype=object)     # => 33+len(el) per element
```



We see that `str_` is smaller again, yet for more diverse lengths `str` can take the win.

As for the native `np.str_` and `np.bytes_` types, NumPy has a handful of common string operations. They mirror Python's `str` methods, live in the `np.char` module and operate over the whole array:

```
>>> np.char.upper(np.array([['a','b'],['c','d']]))
array([['A', 'B'],
       ['C', 'D']], dtype='<U1')
```

char_upper.py

With object-mode strings the loops must happen on the Python level:

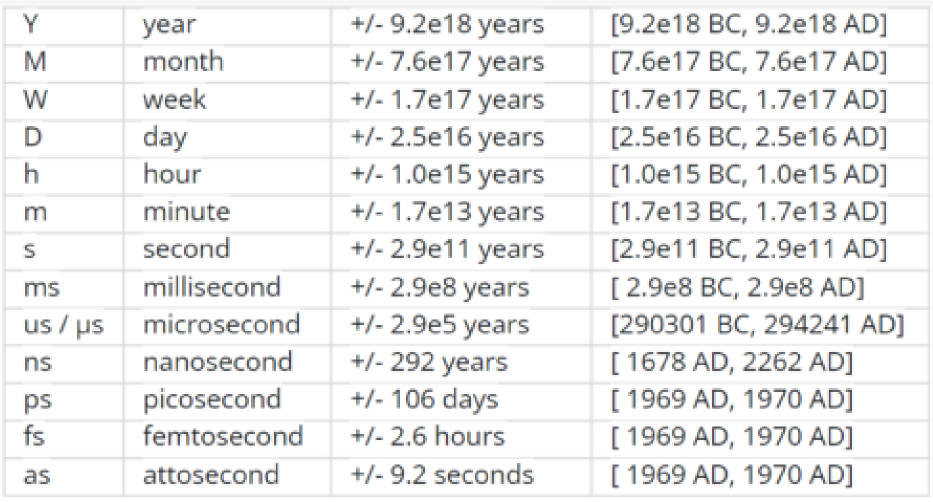
```
>>> a = np.array([['a','b'],['c','d']], object)
>>> np.vectorize(lambda x: x.upper(), otypes=[object])(a)
array([['A', 'B'],
       ['C', 'D']], dtype=object)
```

vectorize_upper.py

According to my benchmarks, basic operations work somewhat faster with `str` than with `np.str_`.

## 5. Datetimes

NumPy introduces an interesting native data type for datetimes, similar to a POSIX timestamp (aka Unix time, the number of seconds since 1 Jan 1970) but capable of counting time with a configurable granularity — from years to attoseconds —



| | | | |
|---|---|---|---|
| Y | year | +/- 9.2e18 years | [9.2e18 BC, 9.2e18 AD] |
| M | month | +/- 7.6e17 years | [7.6e17 BC, 7.6e17 AD] |
| W | week | +/- 1.7e17 years | [1.7e17 BC, 1.7e17 AD] |
| D | day | +/- 2.5e16 years | [2.5e16 BC, 2.5e16 AD] |
| h | hour | +/- 1.0e15 years | [1.0e15 BC, 1.0e15 AD] |
| m | minute | +/- 1.7e13 years | [1.7e13 BC, 1.7e13 AD] |
| s | second | +/- 2.9e11 years | [2.9e11 BC, 2.9e11 AD] |
| ms | millisecond | +/- 2.9e8 years | [ 2.9e8 BC, 2.9e8 AD] |
| us / μs | microsecond | +/- 2.9e5 years | [290301 BC, 294241 AD] |
| ns | nanosecond | +/- 292 years | [ 1678 AD, 2262 AD] |
| ps | picosecond | +/- 106 days | [ 1969 AD, 1970 AD] |
| fs | femtosecond | +/- 2.6 hours | [ 1969 AD, 1970 AD] |
| as | attosecond | +/- 9.2 seconds | [ 1969 AD, 1970 AD] |

Table from the official docs

- Years granularity means 'just count the years' — no real improvement against storing years as an integer.
- Days granularity is an equivalent of Python's `datetime.date` .
- Microseconds — of Python's `datetime.datetime` .

And everything below is unique to `np.datetime64` .

When creating an instance of `np.datetime64` , NumPy chooses the most coarse granularity that can still hold such data:

```
>>> np.datetime64('today')          # days granularity (in local time UTC+7)
numpy.datetime64('2021-12-25')

>>> np.datetime64('now')            # seconds granularity (in UTC)
numpy.datetime64('2021-12-24 18:14:00')

>>> np.datetime64(dt.utcnow())      # microsecond granularity
numpy.datetime64('2021-12-24 18:14:23.404438')

>>> np.datetime64('2021-12-24 18:14:23.404438123')   # nanosecond granularity
numpy.datetime64('2021-12-24 18:14:23.404438123')
```

constructing_dt64.py

Note that the string initializer is not so lenient as in `pd.to_datetime` : it must be in this

When creating an array you decide if you are ok with the granularity that NumPy has chosen for you or you insist on, say, nanoseconds or what not, and it'll give you $2^{63}$ equidistant moments measured in the corresponding units of time to either side of 1 Jan 1970.

```
>>> np.datetime64(dt.utcnow(), 'datetime64[ns]')        # us is too coarse for me!
numpy.datetime64('2021-12-24 18:14:23.404438000', dtype='datetime64[ns]')
```

datetime64.py

It is possible to have a multiple of a base unit. For example, if you only need a precision of 0.1 sec, you don't necessarily need to store milliseconds:

```
>>> a = np.array([dt.utcnow()], dtype='datetime64[100ms]'); a
array(['2022-12-24T18:15:08.300'], dtype='datetime64[100ms]')

>>> a + 1
array(['2022-12-24T18:15:08.400'], dtype='datetime64[100ms]')
```

d64_100ms.py

To get a machine-readable representation of the granularity without parsing the dtype string:

```
>>> a[0].dtype
dtype('<M8[100ms]')
>>> np.datetime_data(a[0])
('ms', 100)
```

datetime_data.py

Just like in pure Python when you subtract one `np.datetime64` from another you get a `np.timedelta64` object (also represented as a single int64 with a configurable granularity). For example, to get the number of seconds until the New Year,

```
>>> z = np.datetime64('2022-01-01') - np.datetime64(dt.now()); z
numpy.timedelta64(295345588878,'us')

>>> z.item()                  # getting an ordinary datetime
datetime.timedelta(3, 36353, 424753)
```

Or if you don't care about the fractional part, simply

```
>>> np.datetime64('2022-01-01') - np.datetime64(dt.now(), 's')
numpy.timedelta64(295259,'s')
```

new_year1.py

Once constructed there's not much you can do about the datetime or timedelta objects. For the sake of speed, the amount of available operations is kept to the bare minimum: only conversions and basic arithmetic. For example, there are no 'years' or 'days' helper methods.

To get a particular field from a datetime64/timedelta64 scalar you can convert it to a conventional datetime:

```
>>> np.datetime64('2021-12-24 18:14:23').item()
datetime.datetime(2021, 12, 24, 18, 14, 23, 000000)
>>> np.datetime64('2021-12-24 18:14:23').item().month
12
```

item.py

For the arrays like this one

```
>>> a = np.arange(np.datetime64('2021-01-20'),
                  np.datetime64('2021-12-20'),
                  np.timedelta64(90, 'D')); a
array(['2021-01-20', '2021-04-20', '2021-07-19', '2021-10-17'],
      dtype='datetime64[D]')
```

arange.py

you can either make conversions between `np.datetime64` subtypes (faster)

```
>>> (a.astype('M8[M]') - a.astype('M8[Y]')).view(np.int64)
array([0, 3, 6, 9], dtype=int64)
```

astype.py

or use Pandas (2-4 times slower):

```
>>> s = pd.DatetimeIndex(a); s              # or pd.to_datetime(a)
DatetimeIndex(['2021-01-20', '2021-04-20', '2021-07-19', '2021-10-17'],
              dtype='datetime64[ns]', freq=None)
>>> s.month
Int64Index([1, 4, 7, 10], dtype='int64')
```

to_datetime.py

Here's a useful function that decomposes a `datetime64` array to an array of 7 integer columns (years, months, days, hours, minutes, seconds, microseconds):

```
def dt2cal(dt):
    # allocate output
    out = np.empty(dt.shape + (7,), dtype="u4")
    # decompose calendar floors
    Y, M, D, h, m, s = [dt.astype(f"M8[{x}]") for x in "YMDhms"]
    out[..., 0] = Y + 1970 # Gregorian Year
    out[..., 1] = (M - Y) + 1 # month
    out[..., 2] = (D - M) + 1 # dat
    out[..., 3] = (dt - D).astype("m8[h]") # hour
    out[..., 4] = (dt - h).astype("m8[m]") # minute
    out[..., 5] = (dt - m).astype("m8[s]") # second
    out[..., 6] = (dt - s).astype("m8[us]") # microsecond
    return out

>>> dt2cal(a)
array([[2021,   12,   15,    9,    0,    0,    0],
       [2021,   12,   18,    9,    0,    0,    0],
       [2021,   12,   21,    9,    0,    0,    0],
       [2021,   12,   24,    9,    0,    0,    0]], dtype=uint32)
```

dt2cal.py

A couple of gotchas with datetimes:

1. Even though leap years are supported,

```
>>> np.array(['2020-03-01', '2022-03-01', '2024-03-01'], np.datetime64) - \
    np.array(['2020-02-01', '2022-02-01', '2024-02-01'], np.datetime64)
array([29, 28, 29], dtype='timedelta64[D]')
```

leap_year.py

leap seconds (an essential part of both UTC and ordinary wall time) are not:

To be fair, neither `datetime.datetime` nor even `pytz` counts them, either (although in general it is possible to extract info about them with pytz). `time` module supports them only formally (accepts 60th second, but gives incorrect intervals).

It looks as if only astropy processes them correctly so far,

```
>>> from astropy.time import Time
>>> (Time('2017-01-01') - Time('2016-12-31 23:59')).sec
61.00000000001593
```

astropy.py

others adhere to the proleptic Gregorian calendar with its exactly 86400 SI seconds a day that has already gained about half a minute difference with the wall time since 1970 due to irregularities of the Earth rotation.

The practical implications of using this calendar are:
– mistake when calculating intervals that include one or more leap seconds
– exception when trying to construct a datetime64 from a timestamp taken during a leap second

2. As both `np.datetime64` and `np.timedelta64` have the same width, care must be taken with large timedeltas:

```
>>> np.datetime64('2262-01-01', 'ns') - np.datetime64('1678-01-01', 'ns')
numpy.timedelta64(-17537673709551616,'ns')
```

negative_timedelta.py

Finally, note that all the times in `np.datetime64` are 'naive': they are not 'aware' of daylight saving (so it is recommended to store all datetimes in UTC) and are not capable of being converted from one timezone to another (use pytz for timezone conversions):

```
>>> a = np.arange(np.datetime64('2022-01-01 12:00'),
                  np.datetime64('2022-01-03 12:00'),
                  np.timedelta64(1, 'D'))

>>> np.datetime_as_string(a)
array(['2022-01-01T12:00', '2022-01-02T12:00'], dtype='<U35')

>>> np.datetime_as_string(a, timezone='local')
array(['2022-01-01T19:00+0700', '2022-01-02T19:00+0700'], dtype='<U39')

>>> np.datetime_as_string(a, timezone=pytz.timezone('US/Eastern'))
array(['2022-01-01T07:00-0500', '2022-01-02T07:00-0500'], dtype='<U39')
```

datetime_as_string.py

## 6. Combinations thereof

A 'structured array' in NumPy is an array with a custom `dtype` made from the types described above as the basic building blocks (akin to `struct` in C). A typical example is an RGB pixel color: a 3 bytes long type (usually 4 for alignment), in which the colors can be accessed by name:

```
>>> rgb = np.dtype([('x', np.uint8), ('y', np.uint8), ('z', np.uint8)])
>>> a = np.zeros(5, z); a
array([(0, 0, 0), (0, 0, 0), (0, 0, 0), (0, 0, 0), (0, 0, 0)],
      dtype=[('x', 'u1'), ('y', 'u1'), ('z', 'u1')])
>>> a[0]
(0, 0, 0)
>>> a[0]['x']
0
>>> a[0]['x'] = 10
>>> a
array([(10, 0, 0), ( 0, 0, 0), ( 0, 0, 0), ( 0, 0, 0), ( 0, 0, 0)],
      dtype=[('x', 'u1'), ('y', 'u1'), ('z', 'u1')])
>>> a['z'] = 5
>>> a
array([(10, 0, 5), ( 0, 0, 5), ( 0, 0, 5), ( 0, 0, 5), ( 0, 0, 5)],
      dtype=[('x', 'u1'), ('y', 'u1'), ('z', 'u1')])
```

structured_array.py

```
>>> b = a.view(np.recarray)
>>> b
rec.array([(10, 0, 5), ( 0, 0, 5), ( 0, 0, 5), ( 0, 0, 5), ( 0, 0, 5)],
          dtype=[('x', 'u1'), ('y', 'u1'), ('z', 'u1')])
>>> b[0].x
10
>>> b.y=7; b
rec.array([(10, 7, 5), ( 0, 7, 5), ( 0, 7, 5), ( 0, 7, 5), ( 0, 7, 5)],
          dtype=[('x', 'u1'), ('y', 'u1'), ('z', 'u1')])
```

recarray.py

Here it works like `reinterpret_cast` in C++, but sure enough, recarray can be created on its own, without being a view of something else.

Types for the structured arrays do not necessarily need to be homogeneous and can even include subarrays.

With structured arrays and recarrays can get the 'look and feel' of a basic Pandas DataFrame:

- you can address columns by names,
- do some arithmetic and statistic calculations with them,
- you can handle missing values efficiently,
- some operations are faster in NumPy than in Pandas

But they lack:

- grouping (except what is offered by itertools.groupby)
- the mighty pandas Index and MultiIndex (so no pivot tables) and
- other niceties like convenient sorting, etc.

The gotcha here is that even though this syntax is convenient for addressing particular columns as a whole, neither structured arrays nor recarrays are something you'd want to use in the innermost loop of a compute-intensive code:

```
a = np.random.rand(100000, 4)

b = a.view(dtype=[('x', np.float64), ('y', np.float64)])

c = np.recarray(buf=a, shape=len(a), dtype=
                [('x', np.float64), ('y', np.float64)])

s1 = 0
for r in a:
    s1 += (r[0]**2 + r[1]**2)**-1.5          # reference

s2 = 0
for r in b:
    s2 += (r['x']**2 + r['y']**2)**-1.5      # 5x slower

s3 = 0
for r in c:
    s3 += (r.x**2 + r.y**2)**-1.5            # 7x slower

S1 = np.sum((a[:, 0]**2 + a[:, 1]**2)**-1.5) # 20x faster
S2 = np.sum((b['x']**2 + b['y']**2)**-1.5)   # same as S1
S3 = np.sum((c.x**2 + c.y**2)**-1.5)         # same as S1
```

recarray_benchmark.py

*Note: profiling python code can sometimes be counter-intuitive: changing x**2 to x*x can make the code run 1.5 faster or slower depending on the nature of x.*

## 7. Type Checks

One way to check NumPy array type is to run `isinstance` against its element:

```
>>> a = np.array([1, 2, 3])
>>> v = a[0]
>>> isinstance(v, np.int32)    # might be np.int64 on a different OS
True
```

isinstance.py

All the NumPy types are interconnected in an inheritance tree displayed at the top of the article (blue=abstract classes, green=numeric types, yellow=others) so instead of

```
>>> isinstance(v, np.integer)        # true for all integers
True
>>> isinstance(v, np.number)         # true for integers and floats
True
>>> isinstance(v, np.floating)       # true for floats except complex
False
>>> isinstance(v, np.complexfloating) # true for complex floats only
False
```

using_abstract_dtypes.py

The downside of this method is that it only works against a *value* of the array, not against the array itself. Which is not useful when the array is empty, for example. Checking the type of the *array* is more tricky.

For basic types the `==` operator does the job for a single type-check:

```
>>> a.dtype == np.int32
True
>>> a.dtype == np.int64
False
```

and `in` operator for checking against a group of types:

```
>>> x.dtype in (np.half, np.single, np.double, np.longdouble)
False
```

in.py

But for more sophisticated types like `np.str_` or `np.datetime64` they don't.

The recommended way[4] of checking the `dtype` against the abstract types is

```
>>> np.issubdtype(a.dtype, np.integer)
True
>>> np.issubdtype(a.dtype, np.floating)
False
```

issubdtype.py

It works with all native NumPy types, but the necessity of this method looks somewhat

If you have Pandas installed, its type checking tools work with NumPy dtypes, too:

```
>>> pd.api.types.is_integer_dtype(a.dtype)
True
>>> pd.api.types.is_float_dtype(a.dtype)
False
```

pd_api_types.py

Yet another method is to use (undocumented, but used in SciPy/NumPy codebases, eg here) `np.typecodes` dictionary. The tree it represents is way less branchy:

```
>>> np.typecodes
{'Character': 'c',
'Integer': 'bhilqp',
'UnsignedInteger': 'BHILQP',
'Float': 'efdg',
'Complex': 'FDG',
'AllInteger': 'bBhHiIlLqQpP',
'AllFloat': 'efdgFDG',
'Datetime': 'Mm',
'All': '?bhilqpBHILQPefdgFDGSUVOMm'}
```

typecodes.py

Its primary application is to generate arrays with specific dtypes for testing purposes, but it can also be used to distinguish between different groups of dtypes:

```
>>> a.dtype.char in np.typecodes['AllInteger']
True
>>> a.dtype.char in np.typecodes['Datetime']
False
```

char.py

*Note that using `a.dtype.kind` instead of `a.dtype.char` is a mistake: `np.zeros(1, dtype=np.uint8).dtype.kind == 'u'` is missing from `np.typecodes` while `<…>.char == 'B'` is listed there.*

One downside of this method is that bools, strings, bytes, objects, and voids ('?', 'U', 'S', 'O', and 'V', respectively) don't have dedicated keys in the dict.

I would like to thank members of the NumPy team for their help in chasing the typos and for the productive discussion of some advanced concepts.

**References**

1. Ricky Reusser, Half-Precision Floating-Point, Visualized

2. Floating point guide https://floating-point-gui.de/

3. David Goldberg, What Every Computer Scientist Should Know About Floating-Point Arithmetic, Appendix D

4. NumPy issue #17325, Add a canonical way to determine if dtype is integer, floating point or complex.

**License**