# Size of `numpy` arrays

## Introduction

Different Python instances occupy different amounts of memory.

Regarding `numpy` arrays, the memory footprint mainly depends on the data type of the data (`dtype`). The knowledge of typical memory use of some dtypes can help estimate the memory use of any scientific problem.

## `nbytes` attribute

The memory footprint of the data of `np.ndarray` objects is determined using the `nbytes` attribute.

## Example: array of integers

Let's define an array with dtype `uint8`:

- 'u': *unsigned*
- 'int': *integers*
- 8: stored on 8 bits (8 bits=1 byte)

This data type can store integers from 0 to 255 (included) since 8 bits can store $2^8 = 256$ values.

In [1]:
```python
import numpy as np


arr = np.arange(1000, dtype='uint8')
arr.nbytes
```

Out[1]:
```
1000
```

`arr` has 1000 values each occupying 1 byte, thus the total is 1000 bytes.

Processing math: 100%

## Example: array of floating values

The default behaviour of `numpy` is storing data using a `np.float64` dtype. This data type takes 8 bytes (hence 64 bits) per value.

In [2]:

```
arr = np.random.rand(1000)
print(arr.dtype)
print(arr.nbytes)
```

```
float64
8000
```

**Using this data type for an array of 125 million values would use 1 GB of memory**.

Recall that a recent computer has typically 8 GB of memory yet **you must always keep some memory left to store intermediate results during computation**.

125 millions is a large number that can be obtained using multidimensionnal data: a 4 dimensions array with shape (100, 100, 100, 100) has 100 millions values.

## Specifying *dtype* to reduce memory use?

What about changing the dtype to gain some memory? In most cases, given our skills in computer sciences, this is often a bad idea!

Let's inspect some reasons for this.

### Smaller range of possible values

Let's see what are the numbers that can be represented using a specific dtype. For float values, this is done using `numpy.finfo`:

In [3]:
```
finfo = np.finfo(np.float64)
finfo
```

Out[3]:
```
finfo(resolution=1e-15, min=-1.7976931348623157e+308, max=1.797693134862315
7e+308, dtype=float64)
```

`float64` dtype can handle values from `min` to `max`. Let's see what happens in other cases:

In [4]:
```
np.array([1.79e308], dtype=np.float64)  # value is smaller than max possible value
```

Out[4]:
```
array([1.79e+308])
```

In [5]:
```
np.array([1.80e308], dtype=np.float64)  # value is larger than max possible value
                                          # seen as infinite
```

Out[5]:
```
array([inf])
```

Processing math: 100%

**Advanced**: Python floats are float64, and passing values in a float format have them evaluated by Python before `numpy`. Thus, a way to build arrays with more complete dtypes than float64 is to pass the values as strings:

In [6]:
```
np.array(['1.80e308'], dtype=np.float128)  # a float128 can handle this value...
```

Out[6]:
```
array([1.8e+308], dtype=float128)
```

In [7]:
```
np.array([1.80e308], dtype=np.float128)  # but beware of prior evaluation to inf by P
```

Out[7]:
```
array([inf], dtype=float128)
```

Lower numerical resolution

Storing a float value in a low memory footprint dtype comes with **a resolution loss**. Thus, going from `np.float64` (default) to `np.float32` divides the memory footprint by 2 but strongly deteriorates the resolution.

In [8]:

```
print(np.finfo(np.float64).resolution)
print(np.finfo(np.float32).resolution)
```

```
1e-15
1e-06
```

In [9]:

```python
np.array([1, 1e-15]).sum(dtype=np.float64)   # result of the operation
                                             # can be understood
                                             # with the resolution of float64
```

Out[9]:

```
1.0000000000000001
```

In [10]:

```python
np.array([1, 1e-15]).sum(dtype=np.float32)   # result of the operation
                                             # comes with a loss of
                                             # resolution if float32 is used
```

Out[10]:

```
1.0
```

### Poorer CPU performances

Modern CPU are not optimized to work with unconventionnal floating points resolution. Thus, computation time can increase with memory efficient dtypes.

### Unexpected casting

`numpy` casting rules may be complex and may lead in unexpected results and having a fine control over all dtypes in a large problem is very time consuming during the development phase.

## Conclusion

For all the reasons presented above, the preferred way is **not to change the dtype**. In some cases, however, the `astype` method can be used.

Processing math: 100%

# References of Python instances (advanced)

Processing math: 100%

# Theory

A variable is only a name that is associated with a content in memory.

Sometimes this content is shared by several variables. All of these are references. Thus the two following operations are very different:

- copying all the memory content (deep copy)
- adding a reference to some memory content (reference)

Many functions and methods rely on references up to a point. For instance, `np.ravel(my_array)` will build a new array with some attributes different from the one of `my_array` (e.g. shape...) but keep the same memory content, i.e. a reference to the data of `my_array`.

That makes it pretty difficult to assign some memory use to a specific function call.

## Howto

The `getrefcount` function of package `sys` **returns the number of Python instances that share the same memory content**.

Let's create an object stored in `var`.

The memory content of `var` is 2: one for `var` and one created during the call of `getrefcount`.

In [11]:
```
from sys import getrefcount
var = [(5, 4)]
getrefcount(var)
```

Out[11]:
```
2
```

Let's add a reference to `var`. Its ref count is incremented since `var2` now redirects to the memory content of `var`.

In [12]:
```
var2 = var
getrefcount(var)
```

Out[12]:
```
3
```

What if a deep copy is performed? The `var` refcount does not change.

In [13]:

```
var3 = var.copy()
print(getrefcount(var))
print(getrefcount(var3))
```

```
3
2
```

Processing math: 100%

## Note

When creating `var3`, a new list is created (different memory address than the one of `var`). Yet, this list shares the content of `var` (the tuple)!

In [14]:
```
print(getrefcount(var[0]))
print(getrefcount(var3[0]))
```

```
3
3
```

## Garbage collection

Whenever no reference exists for a memory content, Python makes this space available for other use; This is called garbage collection.

In [15]:
```
print(getrefcount(var))
del var2
print(getrefcount(var))
del var
# memory content of `var` no longer exists
```

```
3
2
```

In the every day life, **scoping rules** of Python are such that a variable defined inside a function is detached from its memory content when exiting the function, thus there are not many cases where `del` should be called.