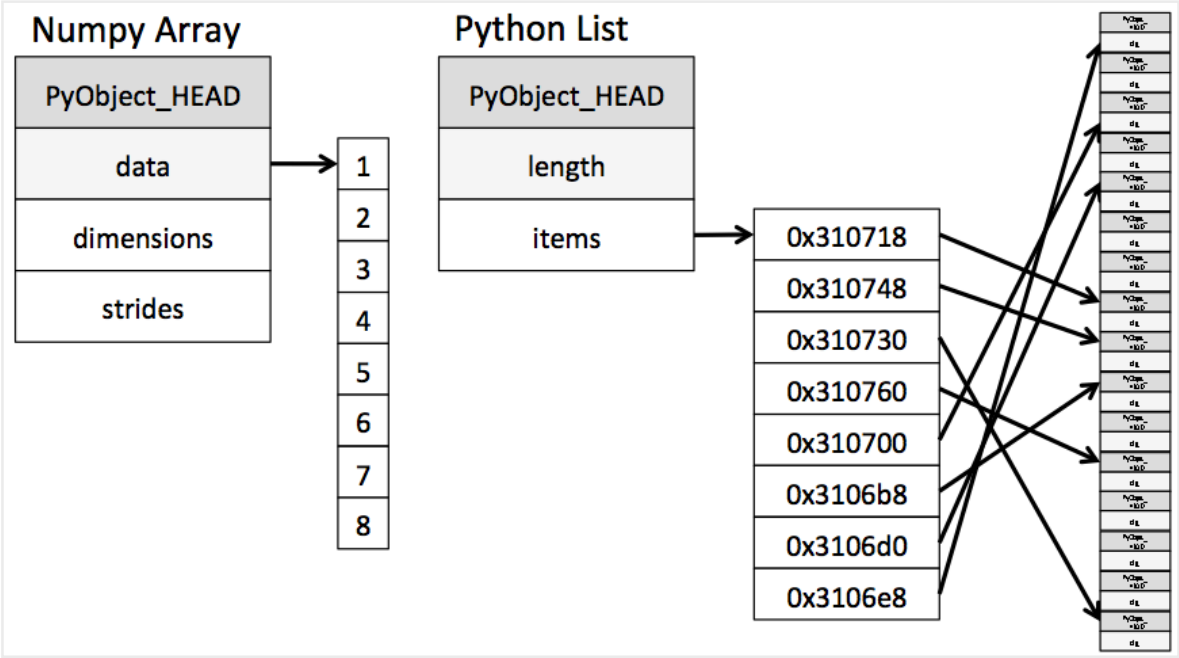
Making Python Faster
— Python Performance Guide

Important Tips

- use array memory efficiently!!!
  - Do not build new arrays repeatedly
- minimise number of loops
  - single loop is always faster than nested loops
- do not put operations that can be done beforehand in loops
  - for example, getting the shapes of input
- use NumPy instead of Pandas, NumPy is faster

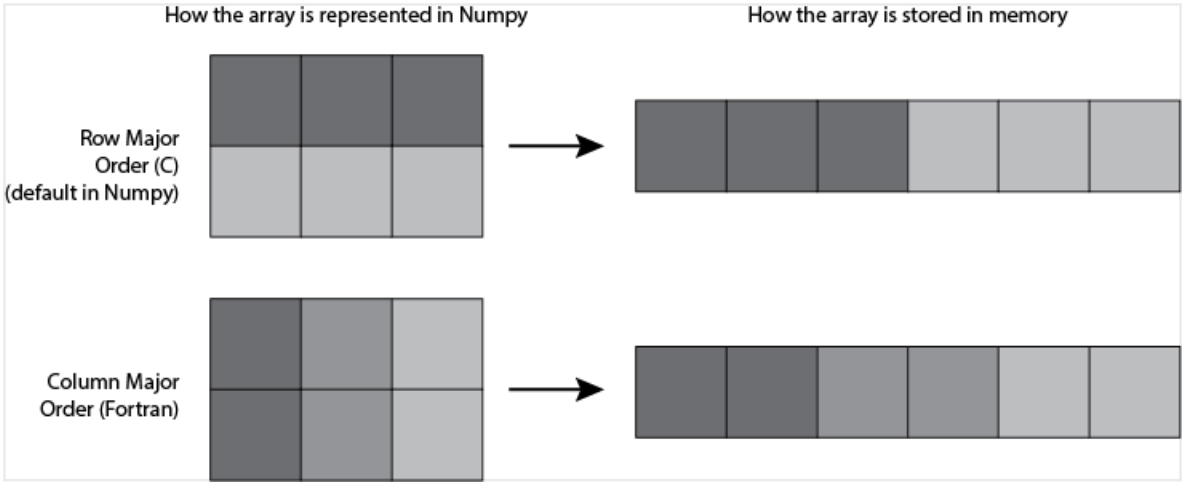
Array storage

NumPy array is faster than python List, as python stores lists with discontinuous address



Array order

There are two ways NumPy stores arrays, C-type and Fortran type




```
# C-type
np.array([values], order='C')
# Fortran (by column)
np.array([values], order='F')
```

- Row-merging operations (axis=0) are faster with C-type, column-merging operations (axis=1) are faster with Fortran type.
  - Note: np.concatenate is faster than np.stack
- Row-slicing [index, :] is faster with C-type, column-slicing [:, index] is faster with F-type
- Choose between C/F according to your demands!

Shallow and deep copy

Shallow copy: only copy address, changes to shallow copy will affect the original variable!!
Deep copy: create a new object, changes to deep copy will NOT affect the original variable

Use more shallow copies to improve speed, but be careful with modifying original variable.

examples

```
for numbers a, b:
b = a      # deep copy
a = a + 1     # deep copy
a += 1     # shallow copy
a = np.add(a, 1, out=a)     # shallow copy (this one is slight faster than a += 1)
## "out" means store output of np.add to location a.
## many math operations have out argument.
```

- add
- subtract
- multiply
- matmul
- divide
- exp
- etc.

```
A = np.array([1, 2, 3, 4])
b = A[[1, 2]]     # deep copy
b = np.take(A, [1, 2])   # deep copy, but faster
b = A[1:2]     # shallow copy
b = A[1:2].copy()    # deep copy
b = A[::2]    # shallow copy
b = A[::2].copy()    # deep copy

## mask slicing
b = A[True, True, False, False]     # deep copy
b = np.compress([True, True, False, False], A)    # deep copy, but faster
b = A[not np.isnan(A)]           # deep copy
b = np.compress(not np.isnan(A), A)    # deep copy, but faster

## the following codes flattens a matrix to 1D array
A.flatten()   # returns a deep copy
A.ravel()     # usually returns a shallow copy
```

Build structured array

Build array with different data types while enjoying the speed of NumPy array

```
np.array(values, dtype = [('name1', np.int32), ('name2', np.float64)])
# the array will have two columns, one with type int32, another with type float64
```

lru_cache — Cache slow algorithms

previously calculated values are not cached in python, but if you add @lru_cache(), the results of this function will be stored.

```
@lru_cache()
@TimeTool.getRunTime
```

```
def complexAlgorithm(a):
    ...

complexAlgorithm(1)    # first run, very slow
complexAlgorithm(1)    # second run, 0 ns, simply get the value from cache
```

Loop Optimisation

1. if you have to use nested loop, put larger loop inside.

```
for i in range(4):
    for j in range(999999):
```

2. try to replace every operation with the faster version

for example, “hello”.join(“world”) is faster than “hello”+”world”

3. Move operations outside loops as far as possible

```
# slow version
for i in range(999):
    for j in range(999):
        a = np.add(a, exp(i) + j * a, out = a)

# faster version
for i in range(999):
    c = exp(i)
    for j in range(999):
        a = np.add(a, c + j * a, out = a)
```

4. use local variables, global variable selecting is slow
5. if lists, arrays are involved, try to add/delete things at the tail. Inserting is slow.
6. Use functions from itertools to replace range(n)

```
# slow multiple-layer loop
for a in list1:
    for b in list2:
        for c in list3:
            ...

# faster version with product()
from itertools import product
for a, b, c in product(list1, list2, list3)
    ...
```

```
# use enumerate to grab index
from itertools import enumerate
for i, object in enumerate(list):
        print(i, object)

# use islice to jump over useless entries
from itertools import islice
for object in islice(list, start, end, step):
        ...

# use takewhile instead of break
from itertools import takewhile
for object in takewhile(predicate, list)
        ...
# predicate should be a function that take object as input and return true/
false values. For example, whether a user is qualified.


## you can build/use your own filter
def is_prime(n):
        ... (some function that tells if n is prime)

for num in is_prime(num_list):
        ...
```

Use packages that can improve efficiency

Packages include Cython, PyPy, Pyrex

Cython is C-extension for python
PyPy is python made from python

Avoid finding inverse of matrix!

this is especially time-consuming
If there is another scheme that solves linear systems instead of finding inverse, you should use that scheme.

References:

Basic functions in NumPy https://numpy.org/doc/stable/reference/

ufuncs.html#available-ufuncs
Why Python is slow https://jakevdp.github.io/blog/2014/05/09/why-python-is-slow/
Itertools Documentation https://docs.python.org/3/library/itertools.html