

High Performance Computing with Python

Reference counting, garbage collection and the global interpreter lock

Rafael Sarmiento

ETHZürich / CSCS

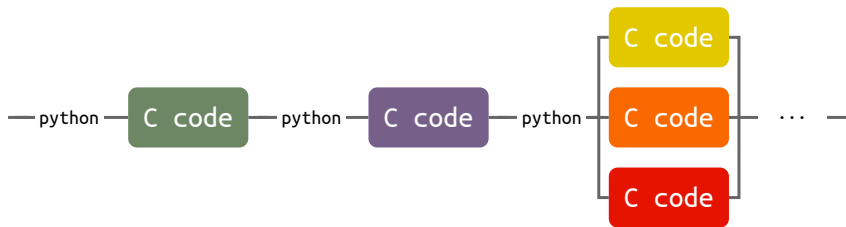
CSCS/USI Summer University 2023

Python

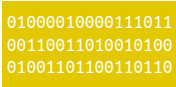
- Most of its operations can be overloaded
- It's fairly easy to glue it to other languages like C and Fortran

Python

- Most of its operations can be overloaded
- It's fairly easy to glue it to other languages like C and Fortran



Reference counting and garbage collection

a →  (ref = 1) a = np.random.random(m)

Reference counting and garbage collection

a → 01000010000111011
b → 00110011010010100
01001101100110110

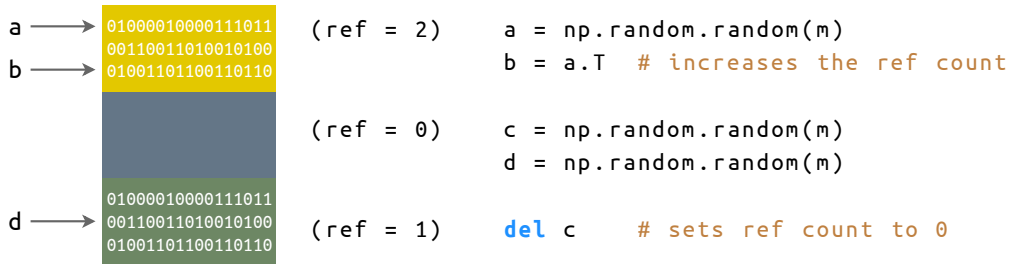
(ref = 2)

```
a = np.random.random(m)
b = a.T # increases the ref count
```

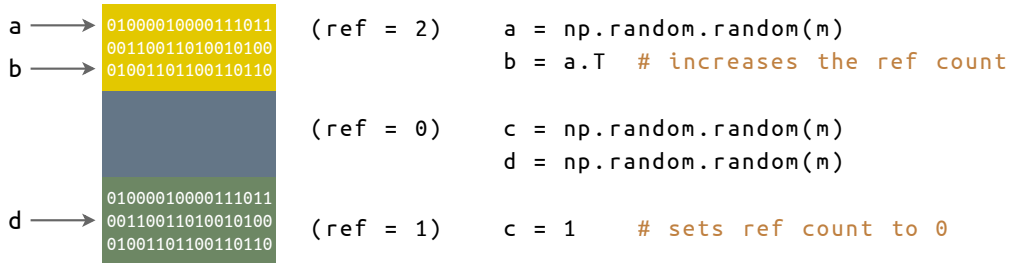
Reference counting and garbage collection

a →	01000010000111011 00110011010010100 01001101100110110	(ref = 2)	a = np.random.random(m)
b →			b = a.T # increases the ref count
c →	01000010000111011 00110011010010100 01001101100110110	(ref = 1)	c = np.random.random(m)
d →	01000010000111011 00110011010010100 01001101100110110	(ref = 1)	d = np.random.random(m)

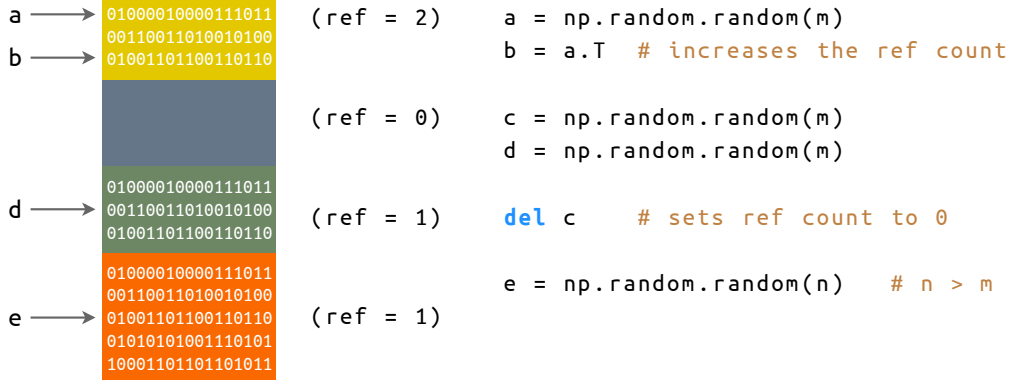
Reference counting and garbage collection



Reference counting and garbage collection



Reference counting and garbage collection



Reference counting and garbage collection

a →	01000010000111011 00110011010010100	(ref = 2)	a = np.random.random(m)
b →	01001101100110110		b = a.T # increases the ref count
f →	01000010000111011 00110011010010100 01001101100110110	(ref = 1)	c = np.random.random(m)
d →	01000010000111011 00110011010010100 01001101100110110	(ref = 1)	d = np.random.random(m)
e →	01000010000111011 00110011010010100 01001101100110110 01010101001110101 10001101101101011	(ref = 1)	del c # sets ref count to 0
			e = np.random.random(n) # n > m
			f = np.random.random(m)

Global interpreter lock (GIL) in CPython

A **Lock** is a mechanism for enforcing limits for accessing resources in an environment where there are many threads of execution

Global interpreter lock (GIL) in CPython

A **Lock** is a mechanism for enforcing limits for accessing resources in an environment where there are many threads of execution

Locks have two methods:

- `acquire()`
- `release()`

Global interpreter lock (GIL) in CPython

- CPU bound

```
...  
acquire_lock()  
    // do something  
release_lock() // let other threads do something  
...
```

Global interpreter lock (GIL) in CPython

- CPU bound

```
...  
acquire_lock()  
    // do something  
release_lock() // let other threads do something  
...
```

- IO bound (waiting from OS calls)

```
...  
release_lock() // let other threads do something  
    // do the io task  
acquire_lock()  
    // go back to the interpreter  
...
```

Global interpreter lock (GIL) in CPython

```
... //some_numpy_function.c

// release the GIL
NPY_LOOP_BEGIN_THREADS

// do something

// acquire the GIL
NPY_LOOP_END_THREADS
...
```

Thank you for your attention!