High Performance Computing with Python

Reference counting, garbage collection and the global interpreter lock

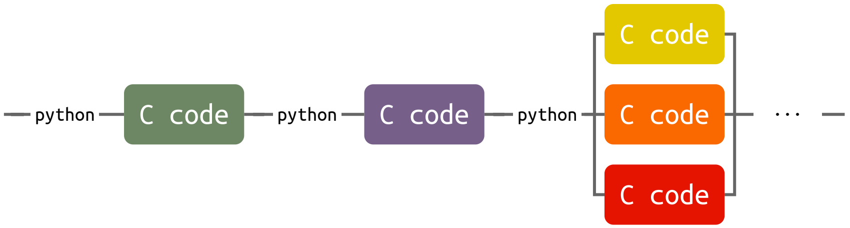
Rafael Sarmiento
ETHZürich / CSCS
CSCS/USI Summer University 2022

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

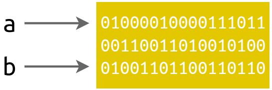


Reference counting and garbage collection

a → 010000100001110110 0110011010010100 010011011001101 10 (ref = 1)

```
a = np.random.random(m)
```

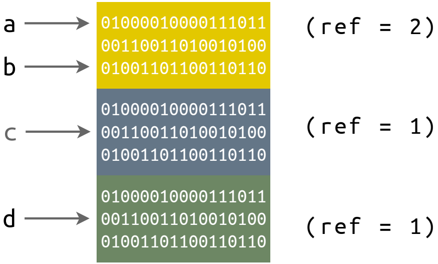
Reference counting and garbage collection



(ref = 2)

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

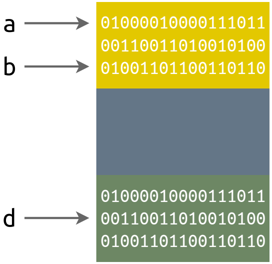
Reference counting and garbage collection




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

c = np.random.random(m)
d = np.random.random(m)
```

Reference counting and garbage collection

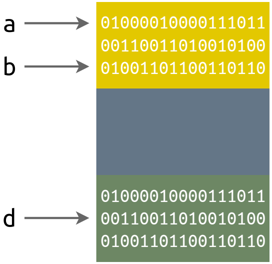



```
(ref = 2)      a = np.random.random(m)
               b = a.T   # increases the ref count

(ref = 0)      c = np.random.random(m)
               d = np.random.random(m)

(ref = 1)      del c     # sets ref count to 0
```

Reference counting and garbage collection



(ref = 2)

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

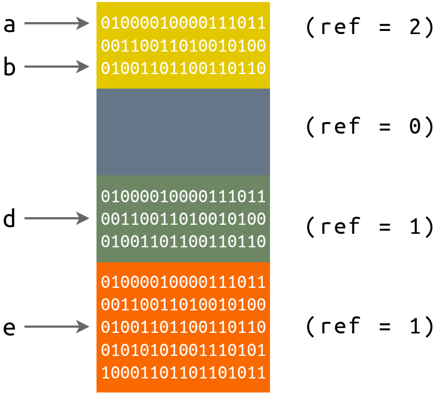
(ref = 0)

```
c = np.random.random(m)
d = np.random.random(m)
```

(ref = 1)

```
c = 1     # sets ref count to 0
```

Reference counting and garbage collection



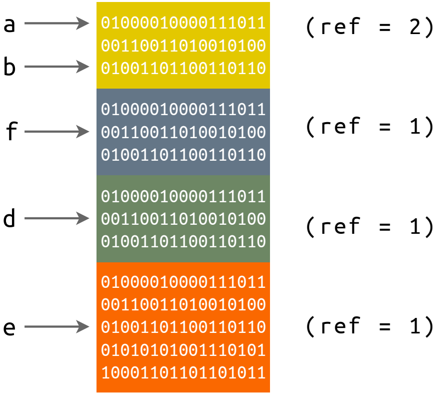

```python
a = np.random.random(m)
b = a.T   # increases the ref count

c = np.random.random(m)
d = np.random.random(m)

del c     # sets ref count to 0

e = np.random.random(n)   # n > m
```

Reference counting and garbage collection




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

c = np.random.random(m)
d = np.random.random(m)

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
...
```

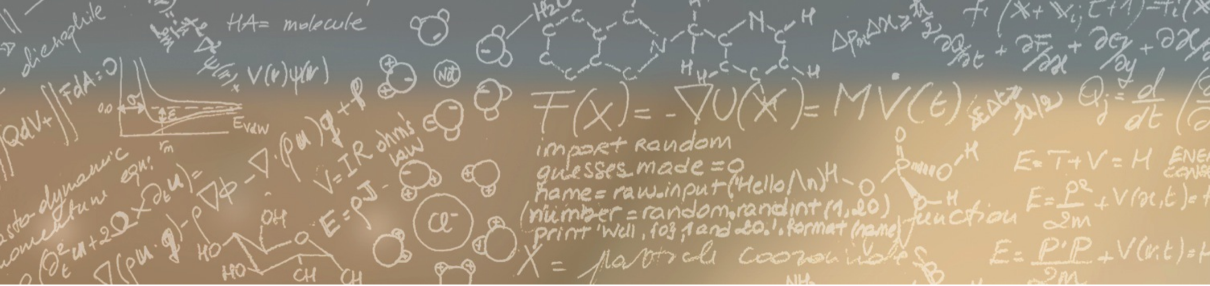
Global interpreter lock (GIL) in CPython

```
... //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!