High Performance Computing with Python

Reference counting, garbage collection and the global interpreter lock

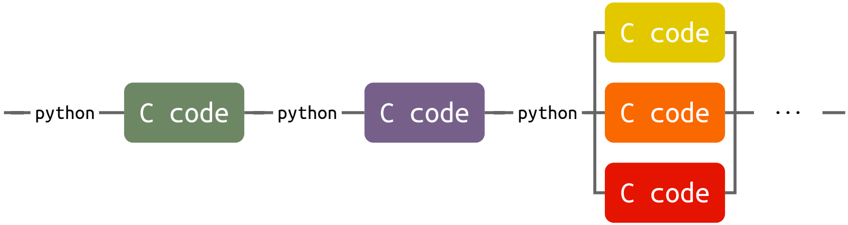
R. Sarmiento, V. Karakasis, T. Manitaras and T. Robinson
ETHZürich / CSCS
Lugano - 06-08.07.2020

Python

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

Python

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

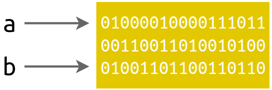


Reference counting and garbage collection

a → 010000100001110110 011001101001010001 001101100110110 (ref = 1)

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

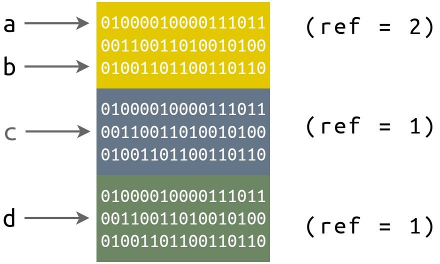
Reference counting and garbage collection



(ref = 2)

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

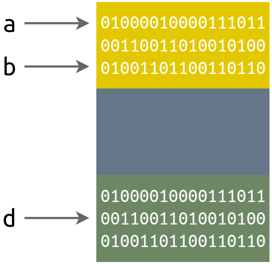
Reference counting and garbage collection




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

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

Reference counting and garbage collection

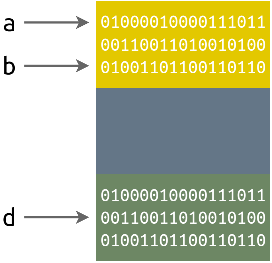



```
(ref = 2)      a = np.random.random(m)
               b = a.T   # increases the ref count

(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
```

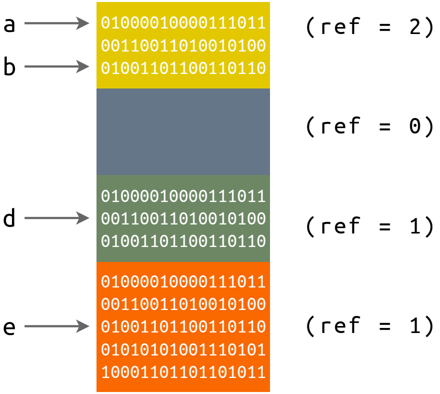
(ref = 0)

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

(ref = 1)

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

Reference counting and garbage collection



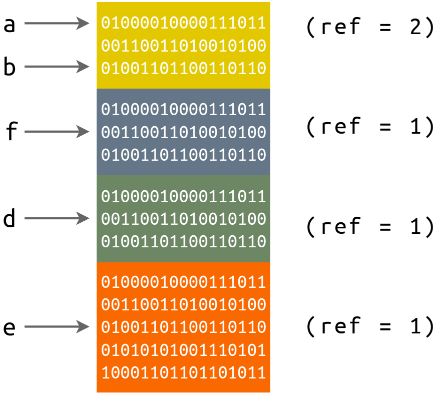

```python
a = np.random.random(m)
b = a.T   # increases the ref count

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 on access to a resource in an environment where there are many threads of execution

Global interpreter lock (GIL) in CPython

A **Lock** is a mechanism for enforcing limits on access to a resource 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
...
```

Python distributions on Piz Daint

- `cray-python`
- `module load cray-python/<version>`
- Uses `cray-libsci` as backend for NumPy

```
>>> import numpy as np
>>> np.show_config()
openblas_info:
    libraries = ['sci_gnu_mp', 'sci_gnu_mp']
    library_dirs = ['/opt/cray/pe/libsci/default/GNU/7.1/x86_skylake/lib']
    language = c
    define_macros = [('HAVE_CBLAS', None)]
blas_opt_info:
    libraries = ['sci_gnu_mp', 'sci_gnu_mp']
    library_dirs = ['/opt/cray/pe/libsci/default/GNU/7.1/x86_skylake/lib']
    language = c
    define_macros = [('HAVE_CBLAS', None)]
...
```

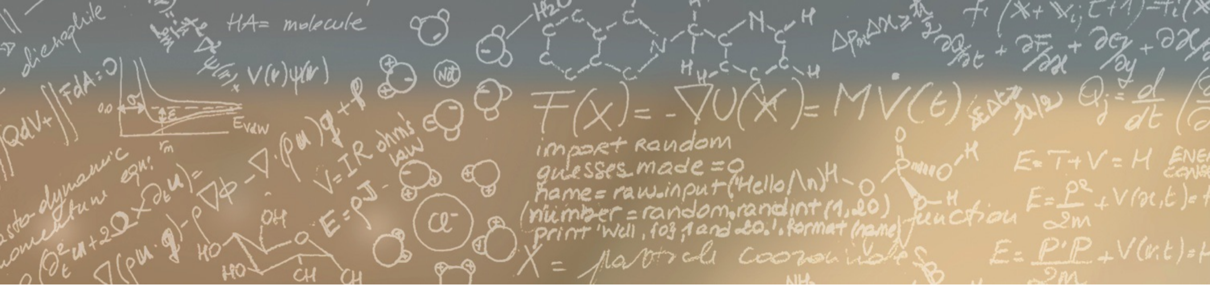
- Uses the libraries installed in the system by Cray
- `module load PyExtensions/<cray-python-version>`
- `module load TensorFlow`
- `pip install --user <package>`

- Anaconda/Miniconda
- Needs to be installed by the user
- Uses Intel's MKL as backend for NumPy

```
>>> import numpy as np
>>> np.show_config()
blas_mkl_info:
    libraries = ['mkl_rt', 'pthread']
    library_dirs = ['/home/user/software/anaconda3.6/lib']
    define_macros = [('SCIPY_MKL_H', None), ('HAVE_CBLAS', None)]
    include_dirs = ['/home/user/software/anaconda3.6/include']
blas_opt_info:
    libraries = ['mkl_rt', 'pthread']
    library_dirs = ['/home/user/software/anaconda3.6/lib']
    define_macros = [('SCIPY_MKL_H', None), ('HAVE_CBLAS', None)]
    include_dirs = ['/home/user/software/anaconda3.6/include']
...
```

- Anaconda brings it's own libraries. An Anaconda installation shouldn't be mixed with Cray's modules.
- `conda install -c <channel> package`
- `pip install <package>`

- Python Package Index (PyPI)
- `pip install --user <package>`
- In general PyPi offers binaries built without a specific target architecture to ensure their portability.
- Before installing with `pip` or `conda`, it might be a good idea to check the recommended installation in package’s homepage or consider building it from sources.



Thank you for your attention!