LUDWIG-
MAXIMILIANS-
UNIVERSITÄT
MÜNCHEN

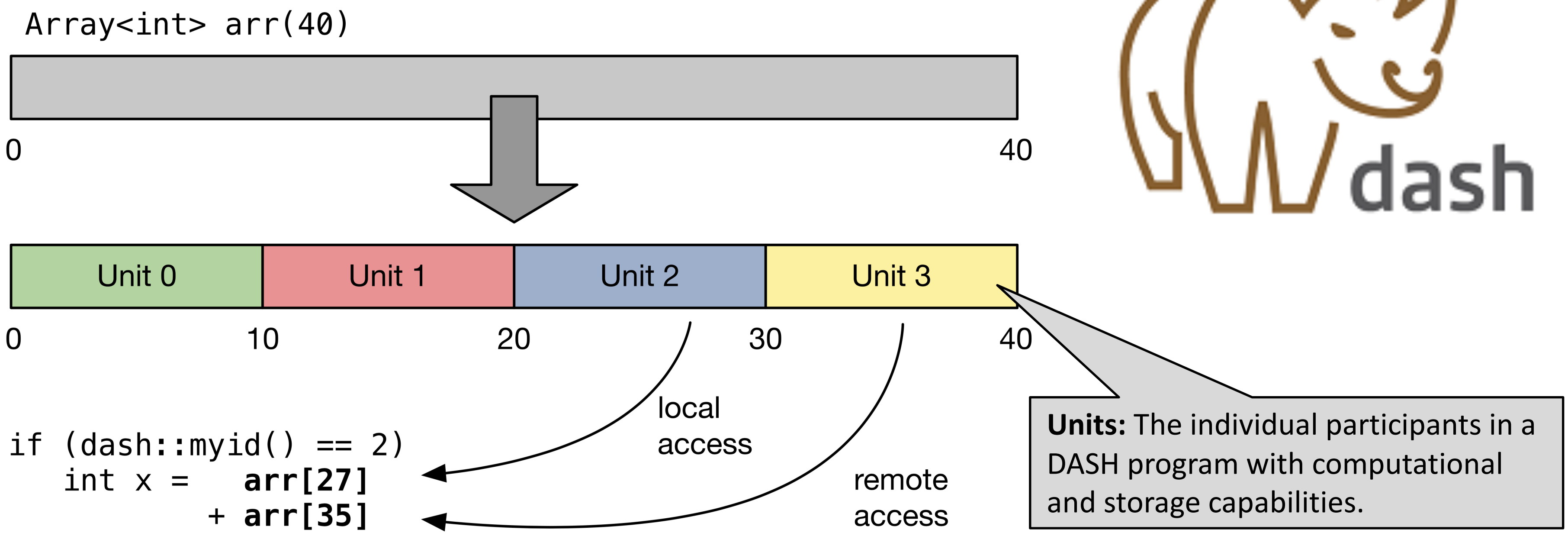
Josef Schäffer

Python-Bindings for the DASH C++ Template Library

- Abschlussvortrag zur Bachelorarbeit
- Aufgabensteller**: Prof. Dr. Dieter Kranzlmüller**
- Betreuer: **Tobias Fuchs**
- Datum des Vortrags: **07.06.2017**

DASH

- C++ Template Library with complete Data-oriented PGAS programming system
- Provides **distributed data structures** and **parallel algorithms** with C++ STL conform interfaces
- Build upon One-sided Communication Substrate, e.g. **MPI**
- DASH Programs follow the **SPMD** Execution Model



```
Array<int> arr(40)
```

```
if (dash::myid() == 2)
   int x =    arr[27]
            + arr[35]
```

Units: The individual participants in a DASH program with computational and storage capabilities.

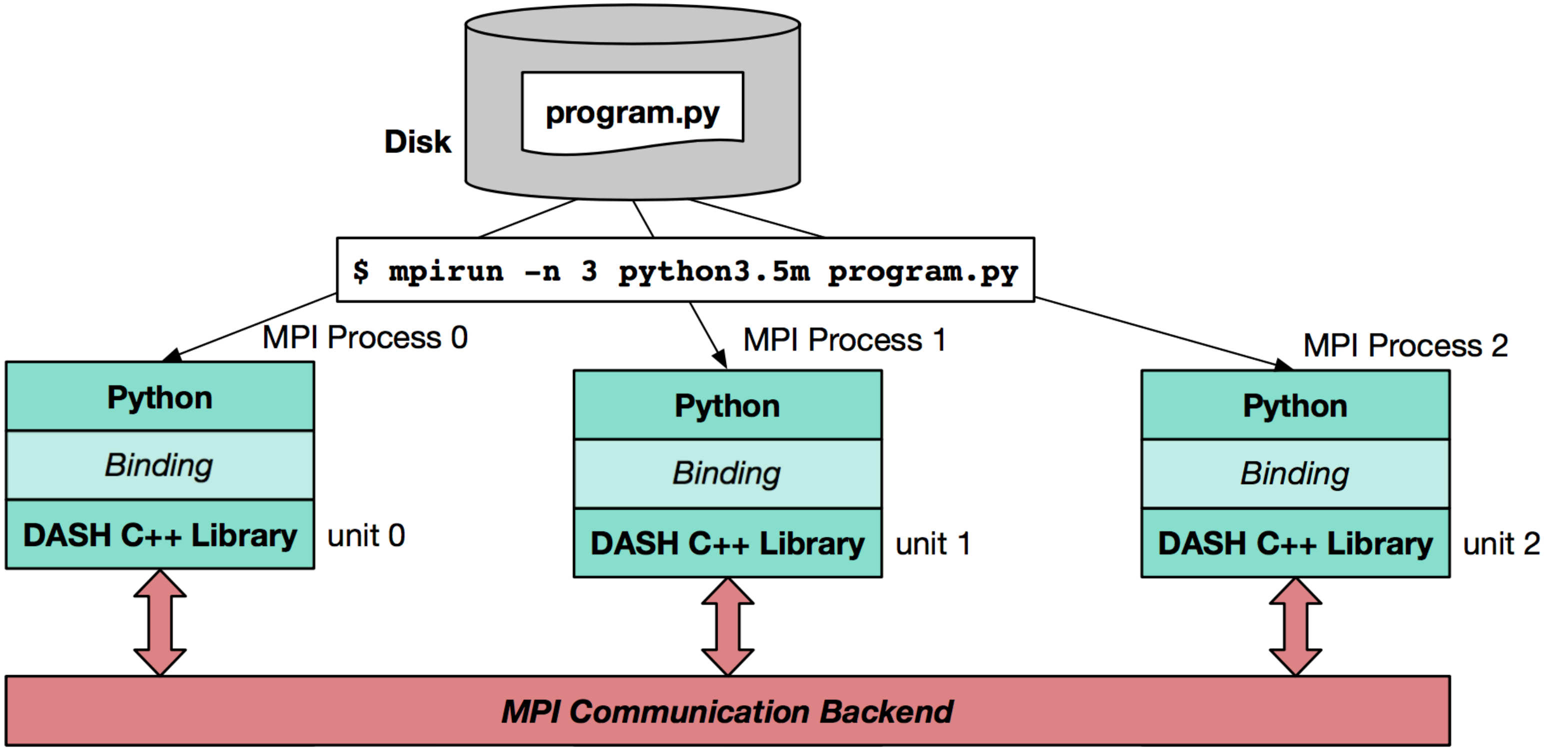
PyDASH: Python-binding for the DASH C++ library

- As part of the bachelor thesis, an implementation of a Python-binding for DASH was developed
- Uses the Python standard as model for API, but follow DASH semantics
- Requires only Python & MPI!

```
# #include<libdash.h>
import pydash

# dash::init(&argc, char *argv[])
pydash.initialize(0, "")

# int myid = dash::myid();
myid = pydash.myid().id()
# int nunits = dash::size();
nunits = pydash.nunits()

print("Hello World! My unit id: {} team size: {}".format(myid, nunits))

# dash::finalize()
pydash.finalize()
```

Demo at the end!

PyDASH Execution



```
$ mpirun -n 3 python3.5m program.py
```

```
Hello World! My unit id: 0 team size: 3
Hello World! My unit id: 2 team size: 3
Hello World! My unit id: 1 team size: 3
```

Implementation Details

Demo at the end!

	DASH	PyDASH

Data structure	`dash::array<int> =`
	`DASH::array`
		`a = pydash.ArrayInt(size)`

Iteration Space	`for(auto it = a.begin();`
	`it != a.end();`
	`++it)`
		`for val in array:`
		`    val.set(x)`

Algorithm	`auto it_min =`
	`dash::min_element(`
	`a.begin(),`
	`a.end())`
		`min_val = array.min(-1)`

Evaluation of Binding Approaches

		CRITERIA (with respect to DASH)			
		Expressiveness	Extensibility	Efficiency	Maturity
APPROACHES	Python C-API	👍	⏳	🐇	👍
	SWIG	👍	👍	🐌	👍
	Boost.Python	⛔	👍	🐌	⛔
	CFFI	⛔	👍	👍	👍
	cppyy	👍	👍	👍	?
	PyCXX	👍	👍	⛔	⛔
	pybind11	👍	👍	👍	👍

- **Conceptual Differences:**

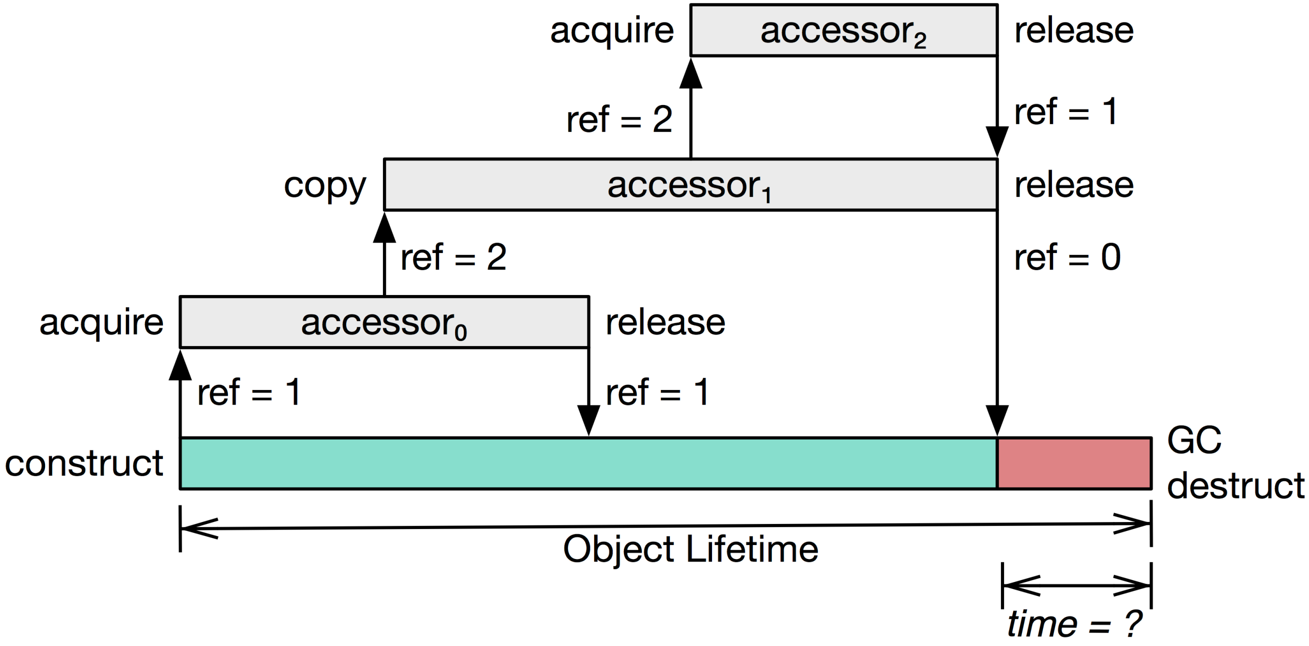
	Python	C++
Resource Reclamation	Reference Counting (RC)	Resource Acquisition Is Initialization (RAII)
Iteration Space	Ranges	Iterators

- **Challenge:** Mapping between conceptual differences of C++ and Python

Conceptual Difference: Lifetime Management of Object

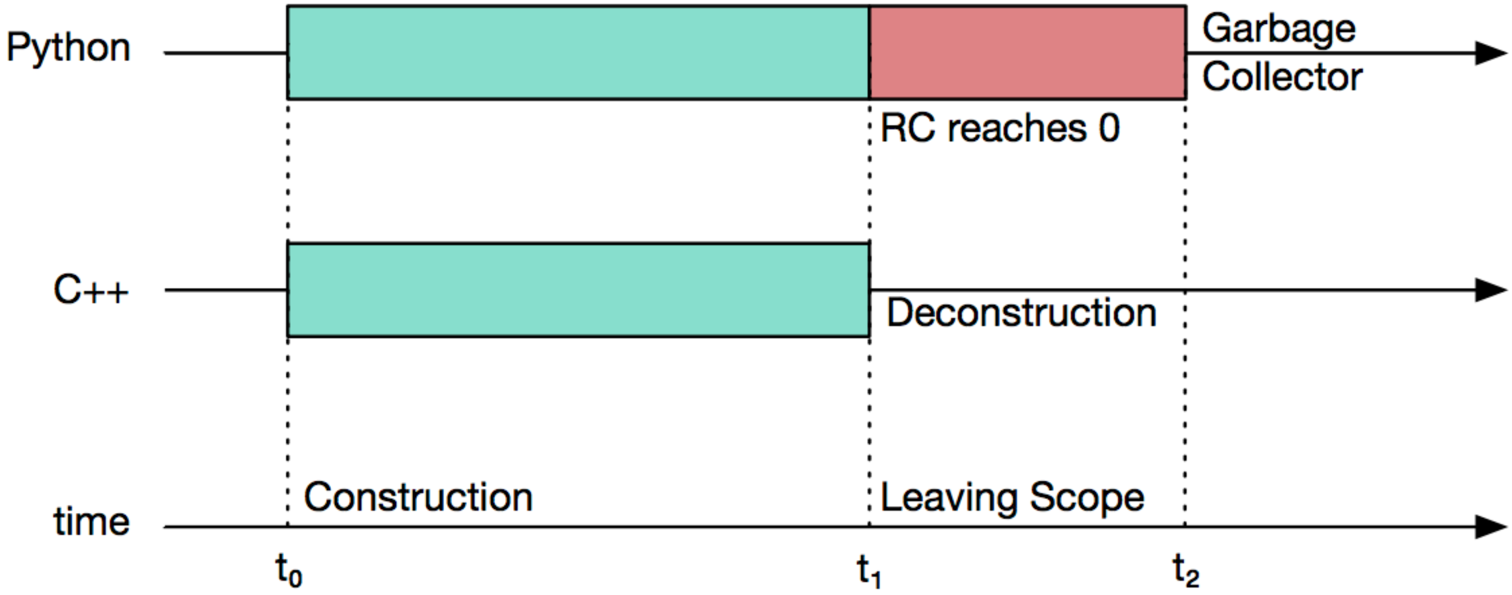
Reference Counting

```
def accessor_2(x):
  return x

def accessor_1(x):
  b = accessor_2(x)
  return b

def accessor_0(x):
  async accessor_1(x)

def main():
  obj = new_object()
  accessor_0(obj)
```



Conceptual Difference: Lifetime Management of Object

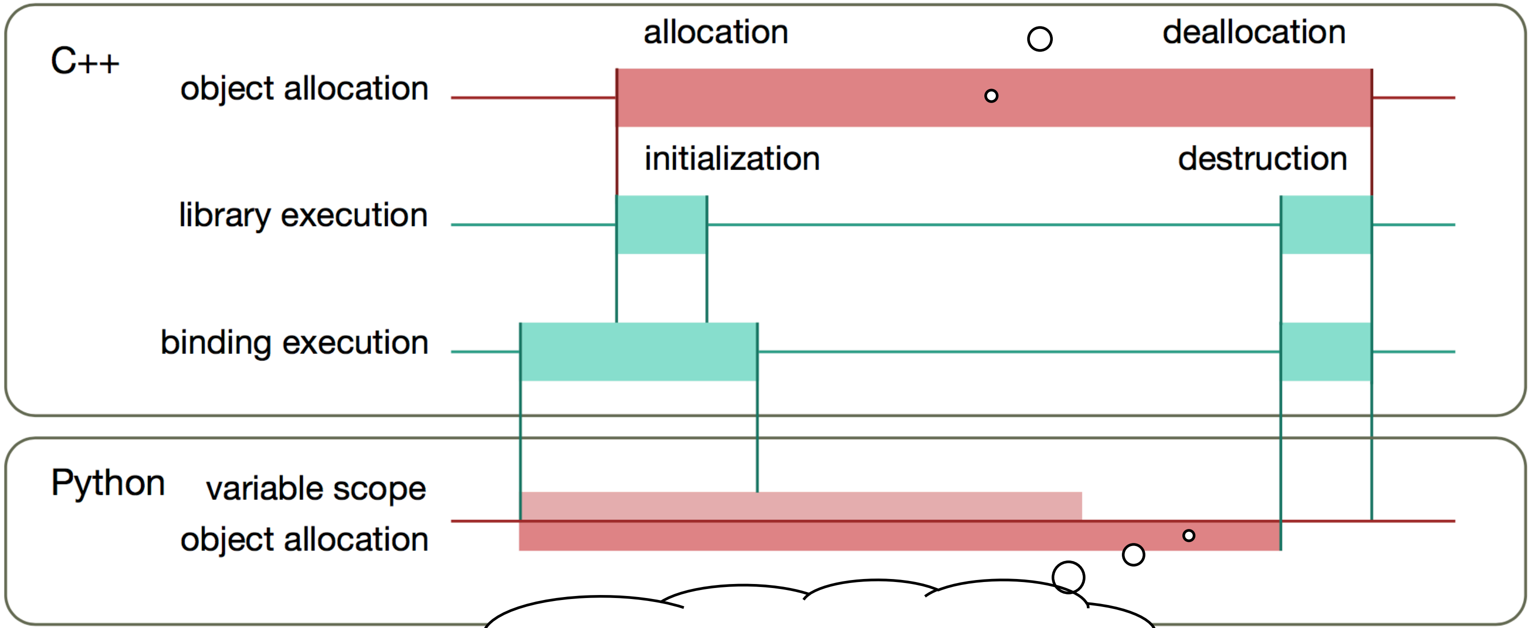
```
# Python                            // C++
if (cond) {                         if (cond) {
    a = MyType()                        MyType a;
    a.execute()                         a.execute();
} # end of scope                      } // end of scope
```



Conceptual Difference: Lifetime Management of Object

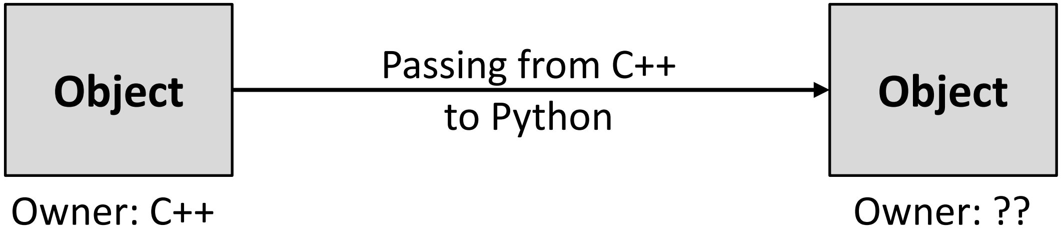
- Resource Reclamation follows different concepts (RC vs. RAII)
- **Combination of C++ and Python raises questions:**
  - Ownership of Object
  - Deconstruction of Object

Who owns the object?



How is the deconstruction handled?

Conceptual Difference: Lifetime Management of Object



```
/* Code: */
pydash::value_type obj = pydash::value_type();

pydash::value_type return_object(){ return obj; }

/* Binding: */
m.def("return_object",
      &return_object,
      py::return_value_policy::copy);
```

Solution

pybind11 allow to specify suitable ownership semantics (move, pass-by-value, shared ownership) via return value policies

Conceptual Difference: Iterator ./. Range

- **C++ Iterators / DASH Global Iterators**
  - The Iterator Concept is part of the C++ STL, DASH iterators comply with STL
  - Iterators as a generalized form of pointers
  - DASH iterators in global address space have additional implications compared to STL, e.g. communication when dereferenced
- **Python Ranges**
  - Iteration over container *values*

	Iterator	Range
Iteration order	User-defined	One direction
Predictability of Expression Evaluation	No	Yes

Syntactic Difference	`for (auto iter  = a.begin();`
	`     iter != a.end(); ++iter)`
		`for (int & value_ref : a)`

Algorithms	`auto it =`
	`  std::algorithm(a.begin(), a.end())`
		`val = a.`*`algorithm`*`()`

Solution: pybind11 offers binding for Iterator to Range

```
/* Binding: */

py_array.def("__iter__",
       [](dash_array_t & arr) {
          return py::make_iterator<

py::return_value_policy::reference_internal,
                  iterator_t, iterator_t, dash::GlobRef<T>
                  >(arr.begin(), arr.end());
       },
       /* ... */
);
```

Demo

Conclusion

- Identified conceptual differences between Python and C++ could be resolved with pybind11 and proved as robust in evaluation
- We offer an easy-to-use toolset for writing HPC applications with Python to combine the intuitive syntax of Python with the expressiveness and instruction-level performance of the DASH C++ library
- **We believe that PyDASH can boost the DASH user base!**

- **Download and test PyDASH!**

  github.com/dash-project/pydash

Backup

BACKUP SLIDES

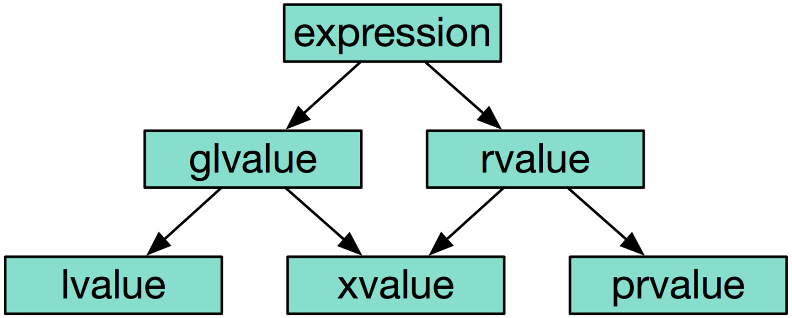
(1) Evaluation: Lifetime of PyDASH Object

```
def test_return_object(x):
  if x:
    print("Entering Scope")
    a = pydash.return__object()
    print ("Leaving Scope")
  print ("Left Scope")
```

```
>>> import pydash
[ LOG |               logged_val | @:0x..a28 --- default construct X

>>> test_return_object(True)
Entering Scope
[ LOG | logged_val(const self &) | @:0x..a60 --- create copy of @0x..a28: X
[ LOG | logged_val(const self &) | @:0x..a60 --- copied value
[ LOG |      logged_val(self &&) | @:0x..fe0 --- move  * <- @0x..a60: X
[ LOG |            ~logged_val() | @:0x..a60 --- destroy X
Leaving Scope
Left Scope
[ LOG |            ~logged_val() | @:0x..fe0 --- destroy X

>>> exit()
[ LOG |            ~logged_val() | @:0x..a28 --- destroy X
```

Lifetime of a PyDASH Objects (2)

- **Ownership**:
  - Owner of Object
  - Responsible for destruction of object
- **Value Category:** Property of Expression



- **Move Semantics:**
  - Address, not Values handed over and Ownership shifted
  - Rely on *rvalue references* (C++11)
- **Return Value Policy**: solution offered by pybind11

Classification of Garbage Collectors

		Determinism	
		non-deterministic	deterministic
Layer	application	Mark-and-Sweep Application Specific GCs	Reference Counting RAII Hazard Pointers C++
	runtime	Mark-and-Sweep	Reference Counting Pool Reclamation

(2) Evaluation: Ma
Iterators to Ranges

Backup Slide:
Live Demonstration

```
import pydash

pydash.initialize(0, "")
myid   = pydash.myid().id()
nunits = pydash.nunits()

# Collectively instantiate
array = pydash.ArrayInt(3 * nunits)

# Initialize array:
array[myid * 3 + 0] = 100 * (1 + myid) + 0
array[myid * 3 + 1] = 100 * (1 + myid) + 1
array[myid * 3 + 2] = 100 * (1 + myid) + 2

#Wait for all units:
pydash.barrier()

#Unit 0 prints out values of array
if myid == 0:
  for val in array:
    print(val.get())
```

```
#Unit 0 sets all values of array to 99
if myid == 0:
  for val in array:
    val.set(99)

#Unit 0 prints out values of array
if myid == 0:
  for val in array:
    print(val.get())

pydash.finalize()
```

```
$ mpirun -n 4 python test.py

100
101
...
402
```