@quansightai

https://www.quansight.com

Updates on Numba, XND and uarray

SciPy LATAM
Bogotá Colombia, Universidad de Los Andes
October 9, 2019

travis@quansight.com
@teoliphant

ANACONDA.

an open source JIT compiler that translates a subset of Python and NumPy code into fast machine code.

http://numba.pydata.org

Accelerate Python Functions

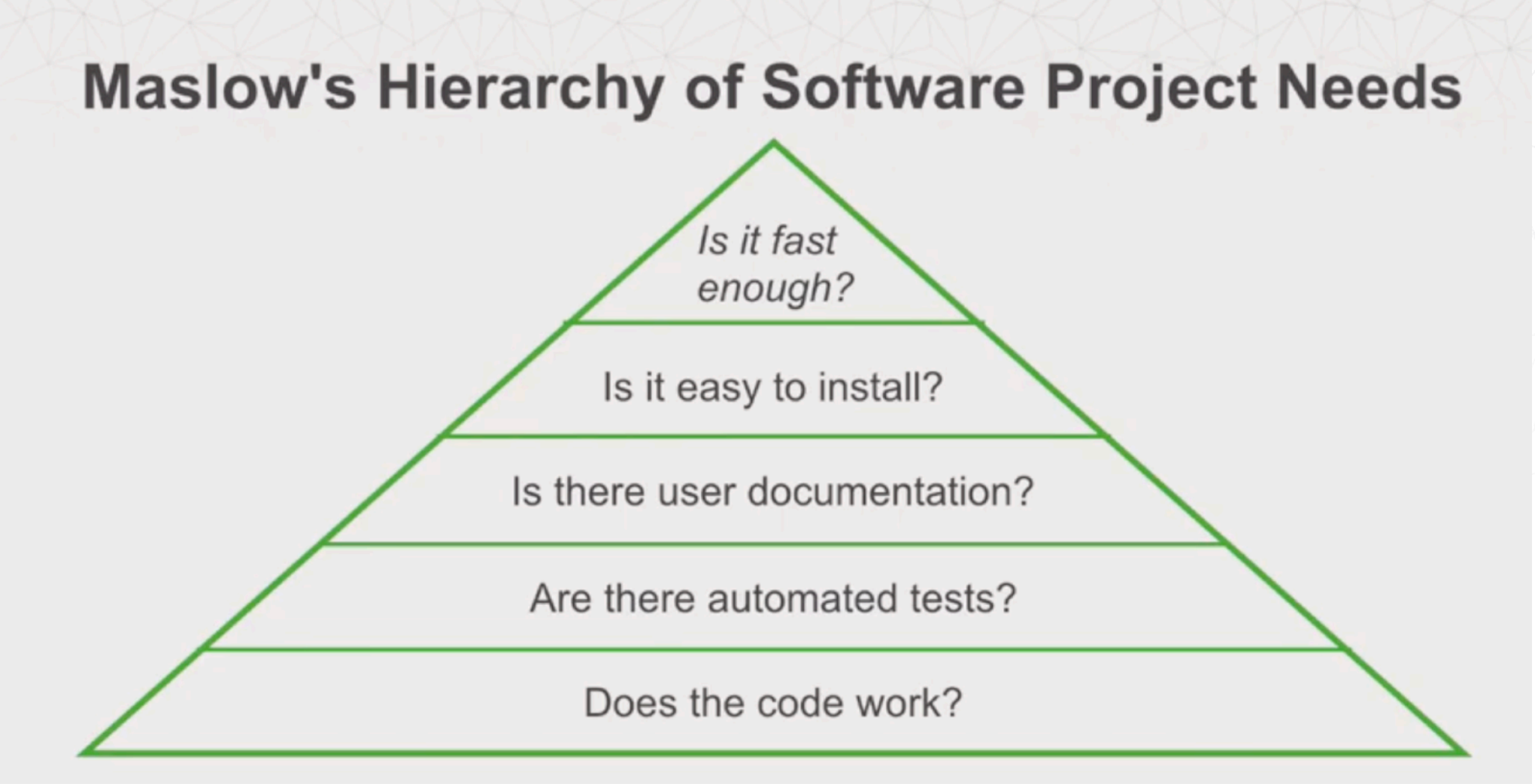
Numba translates Python functions to optimized machine code at runtime using the industry-standard LLVM compiler library. Numba-compiled numerical algorithms in Python can approach the speeds of C or FORTRAN.

You don't need to replace the Python interpreter, run a separate compilation step, or even have a C/C++ compiler installed. Just apply one of the Numba decorators to your Python function, and Numba does the rest.

```
from numba import jit
import random

@jit(nopython=True)
def monte_carlo_pi(nsamples):
    acc = 0
    for i in range(nsamples):
        x = random.random()
        y = random.random()
        if (x ** 2 + y ** 2) < 1.0:
            acc += 1
    return 4.0 * acc / nsamples
```

Before you try to optimize speed!



Stan Seibert, “How to Accelerate an Existing Codebase with Numba” SciPy 2019

LLVMlite - foundations

0.30.0 just released

```
"""
This file demonstrates a trivial function "fpadd" returning the sum of
two floating-point numbers.
"""

from llvmlite import ir

# Create some useful types
double = ir.DoubleType()
fnty = ir.FunctionType(double, (double, double))

# Create an empty module...
module = ir.Module(name=__file__)
# and declare a function named "fpadd" inside it
func = ir.Function(module, fnty, name="fpadd")

# Now implement the function
block = func.append_basic_block(name="entry")
builder = ir.IRBuilder(block)
a, b = func.args
result = builder.fadd(a, b, name="res")
builder.ret(result)

# Print the module IR
print(module)
```

```
from __future__ import print_function

from ctypes import CFUNCTYPE, c_double

import llvmlite.binding as llvm


# All these initializations are required for code generation!
llvm.initialize()
llvm.initialize_native_target()
llvm.initialize_native_asmprinter()  # yes, even this one

llvm_ir = """
   ; ModuleID = "examples/ir_fpadd.py"
   target triple = "unknown-unknown-unknown"
   target datalayout = ""

   define double @"fpadd"(double %".1", double %".2")
   {
   entry:
     %"res" = fadd double %".1", %".2"
     ret double %"res"
   }
   """

def create_execution_engine():
    """
    Create an ExecutionEngine suitable for JIT code generation on
    the host CPU.  The engine is reusable for an arbitrary number of
    modules.
    """
    # Create a target machine representing the host
    target = llvm.Target.from_default_triple()
    target_machine = target.create_target_machine()
    # And an execution engine with an empty backing module
    backing_mod = llvm.parse_assembly("")
    engine = llvm.create_mcjit_compiler(backing_mod, target_machine)
    return engine
```

Two additional efforts in 2006

Buffer Protocol (PEP 3118)

Way for all Python objects to share memory using NumPy-like data-structures (strided memory layout with a shape). "memoryview"

Type system not solved at the time (punted to the struct module syntax extended with character codes)
("I 2s f") == dtype('u4, 2S, f')

__array_interface__

Protocol approach. Any object can define this attribute to explain how it could be interpreted as an array — still tied to NumPy (strided layout)

What if we revisit these earlier efforts

Buffer Protocol (PEP 3118)

Cross-language buffer-protocol plus numpy-like math libraries

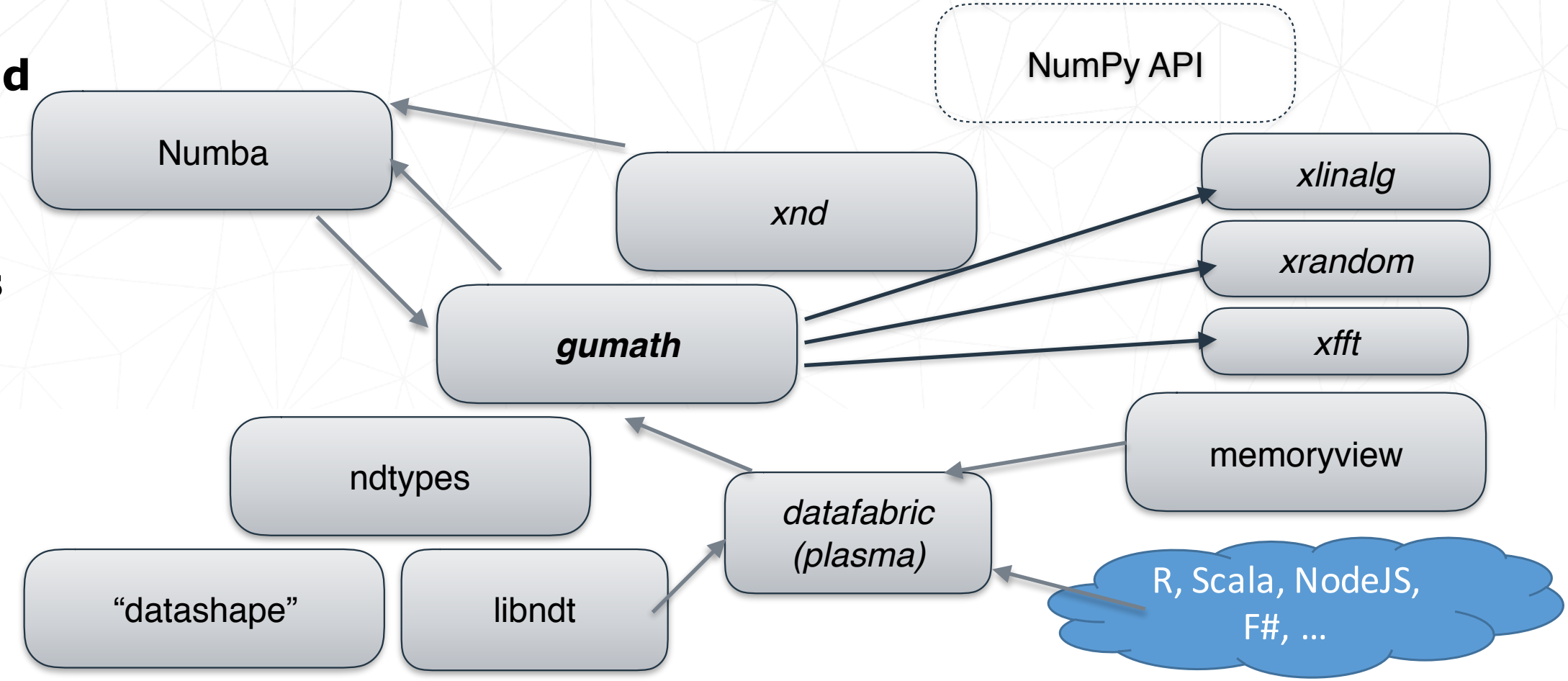
__array_interface__

uarray

New project to formalize and generalize array protocol for Python while that downstream projects can depend on (rather than a single array)

NumPy Refactored

Bring NumPy and Array Capability to other languages and run-times



NumPy's Key Parts

dtype

Description of what is "in the array" — data-description language but missing key primitives (pointer, missing-data types, categoricals, new float types, etc.)
Strictly extensible —- but not easily.

Innovation was ability to map to any memory pointer that you could describe via dtype "language" and then "slice and dice"

umath

Math and functions for arrays. Started as "scalar" kernels (ufuncs) that are applied over the array.

DEShaw added "generalized ufuncs" which allowed the kernel applied over the array to involve "inner-dimensions" (i.e. dot, cholesky, svd, argmax, can be a kernel)

ndarray

Pointer to data described by "dtype" with shape and strides information and powerful "indexing" capabilities.

Mapping pointers to the start of a data-structure you can describe with dtype and then applying (generalized) ufuncs is the essence of array-oriented computing

Language Bindings
(Python, Ruby, …)

Need: C++, Scala, Node,
F#, C#

C-libraries with
defined API/ABI

ndtypes

gumath

xnd

libndtypes

libgumath

libxnd

Generalization of dtype. Description of “any” container

Generalization of numpy array container and Universal functions (arbitray kernels applied over the data)

Examples

```
[5]: xnd([[1., 1.5], [-1.5, 1.]]) # xnd
```

```
[5]: xnd([[1.0, 1.5], [-1.5, 1.0]], type='2 * 2 * float64')
```

```
[7]: xnd(["this", "is", "a", "test", "notebook"]) # xnd
```

```
[7]: xnd(['this', 'is', 'a', 'test', 'notebook'], type='5 * string')
```

```
[9]: xnd([[1,5,2], [1], [7,9,10,20,13]]) # xnd
```

```
[9]: xnd([[1, 5, 2], [1], [7, 9, 10, 20, 13]], type='var * var * int64')
```

Examples

[11]:
```
levels = ['red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'violet']
rainbow = xnd(['red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'violet', 'red', 'green'], levels=levels)
rainbow
```

[11]:
```
xnd(['red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'violet', 'red', 'green'],
    type='9 * categorical('red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'violet')')
```

[12]:
```
rainbow.type
```

[12]:
```
ndt("9 * categorical('red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'violet')")
```

Example

[14]:
```
data = [{'title': 'Introduction to Digital Signal Processing',
         'speaker': 'Allen Downey',
          'room': 10},
        {'title': 'Making Art with Python',
         'speaker':'Emily Xie',
         'room': 16},
        {'title': 'Foundations of Numerical Computing in Python',
         'speaker': 'Scott Sanderson',
         'room': 20},
        {'title':'Exploratory Data Visualization with Vega, Vega-Lite, and Altair',
         'speaker':'Jake VanderPlas',
         'room': 21}]



x = xnd(data)
x
```

[14]:
```
xnd([{'title': 'Introduction to Digital Signal Processing', 'speaker': 'Allen Downey', 'room': 10},
     {'title': 'Making Art with Python', 'speaker': 'Emily Xie', 'room': 16},
     {'title': 'Foundations of Numerical Computing in Python', 'speaker': 'Scott Sanderson', 'room': 20},
     {'title': 'Exploratory Data Visualization with Vega, Vega-Lite, and Altair',
      'speaker': 'Jake VanderPlas',
      'room': 21}],
    type='4 * {title : string, speaker : string, room : int64}')
```

[15]:
```
x[0]
```

[15]:
```
xnd({'title': 'Introduction to Digital Signal Processing', 'speaker': 'Allen Downey', 'room': 10},
    type='{title : string, speaker : string, room : int64}')
```

[16]:
```
x[1, 0]
```

[16]:
```
'Making Art with Python'
```

Math

NumPy became popular because of SciPy and scikits — xnd will need math to be useful.

Because it satisfies buffer protocol for all compatible types, NumPy ecosystem functions already work

xndtools — makes gumath functions from “interface definition file” from f2py author (Pearu Peterson who now works at Quansight Labs)

Numba interface (you can make gumath functions using numba)

+


```
from numba import jit
from xnd import xnd

# register types
import numba_xnd


@numba_xnd.gumath.register_kernel(
    [
        "... * N * M * int64, ... * M * K * int64 -> ... * N * K * int64",
        "... * N * M * float64, ... * M * K * float64 -> ... * N * K * float64",
    ]
)
def simple_matrix_multiply(a, b, c):
    n, m = a.type.shape
    m_, p = b.type.shape
    for i in range(n):
        for j in range(p):
            c[i, j] = 0
            for k in range(m):
                c[i, j] = c[i, j].value + a[i, k].value * b[k, j].value

a = xnd([[1, 2, 3], [4, 5, 6]])
b = xnd([[7, 8], [9, 10], [11, 12]])
c = xnd([[58, 64], [139, 154]])
assert simple_matrix_multiply(a, b) == c
```

Is a set of developer libraries

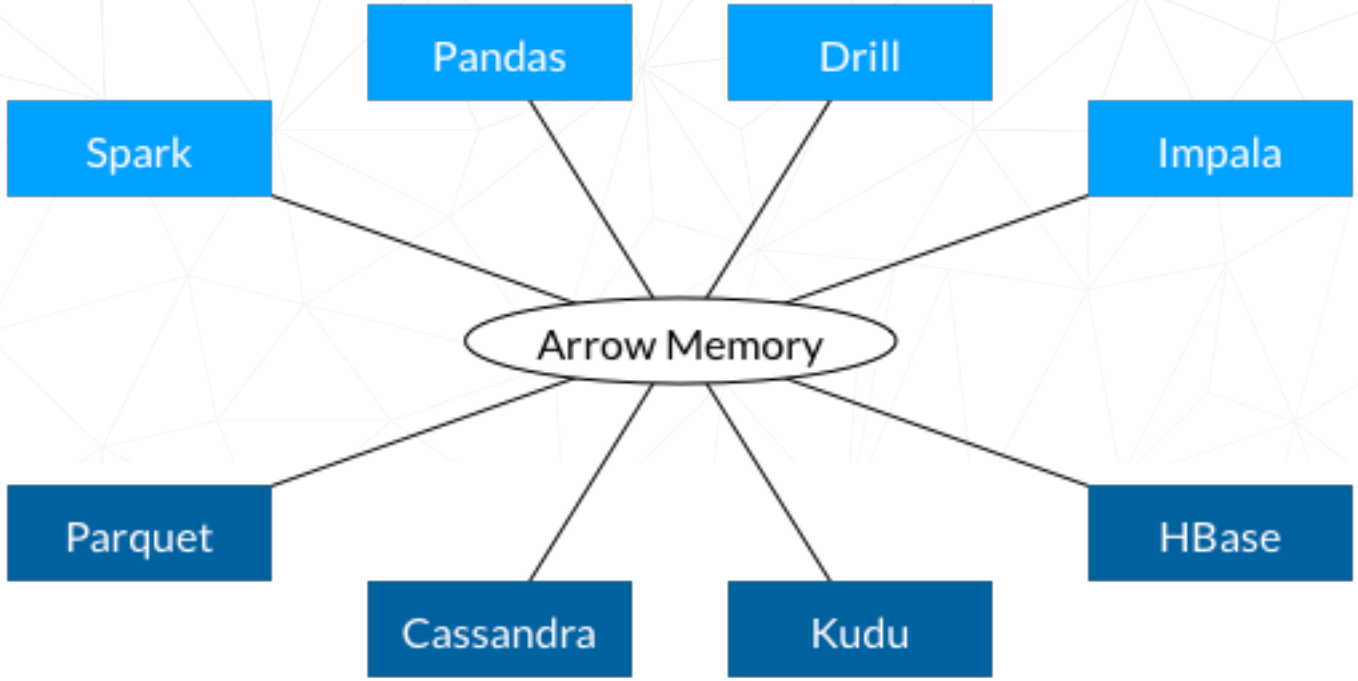
XND is not meant to replace NumPy directly for end-users. It could be used by dask.array, by xarray, by pandas, by PyTorch, by Tensorflow, etc. in places where they depend on NumPy.

Focus on developer-friendly APIS:

- raise error (don't guess or try to make it too easy)
- have documented error handling
- light-weight layers over low-level code
- focus on integration

There are still missing features: advanced indexing being a big one

Apache Arrow

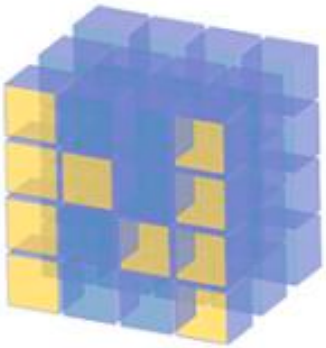


Apache Arrow is a cross-language development platform for in-memory data. It specifies a standardized language-independent columnar memory format for flat and hierarchical data, organized for efficient analytic operations on modern hardware.

Is a generalization of Arrow — you could describe an Arrow container with XND
Like Pandas columns are NumPy arrays.

pandas

$$y_i t = \beta' x_{it} + \mu_i + \epsilon_{it}$$



NumPy

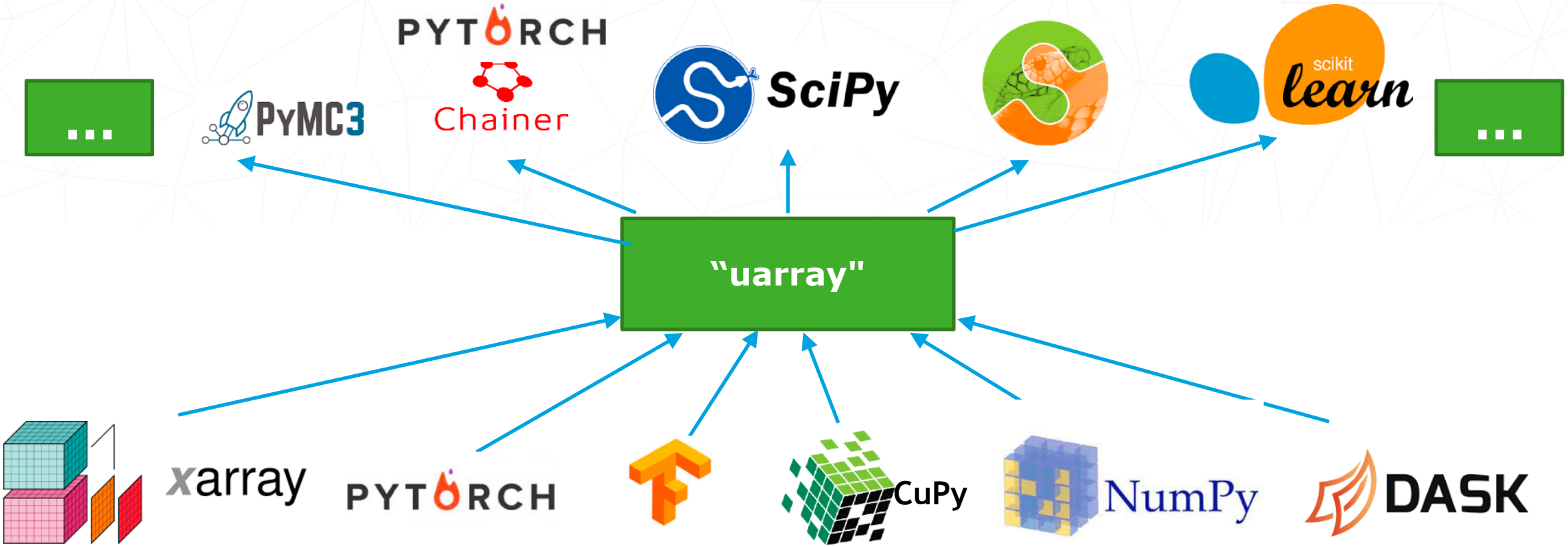
ARROW

Unified Array Interface

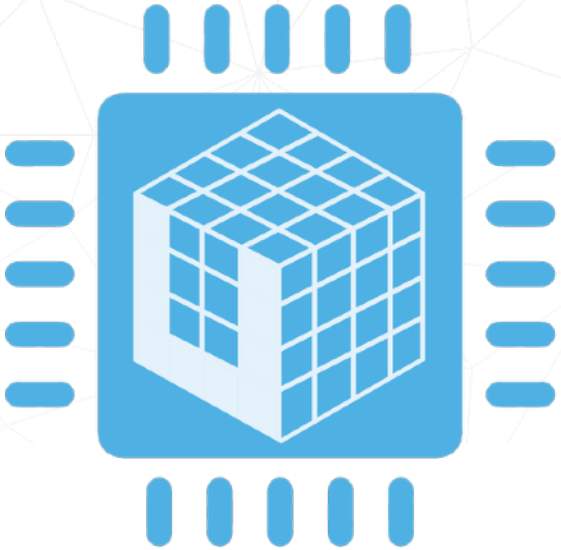
Just started Project!

Need to fix the "string / bytes" problem of the array world!

Logical array vs. strided-pointer of numpy



Uarray



Multi-method system for PyData and SciPy ecosystem

`__ua_domain__`

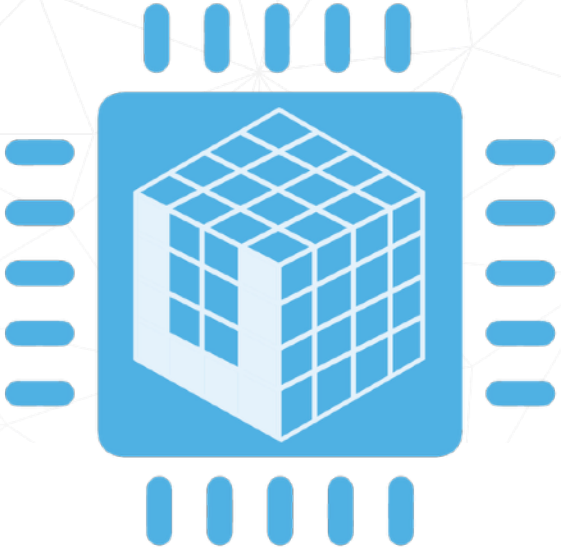
`__ua_domain__` is a string containing the domain of the backend. This is, by convention, the name of the module (or one of its dependencies or parents) that contain the multimethods. For example, `scipy` and `numpy.fft` could both use the `numpy` domain.

`__ua_function__`

This is the most important protocol, one that defines the implementation of a multimethod. It has the signature `(method, args, kwargs)`. Note that it is called in this form, so if your backend is an object instead of a module, you should add `self`. `method` is the multimethod being called, and it is guaranteed that it is in the same domain as the backend. `args` and `kwargs` are the arguments to the function, possibly after conversion (explained below)

Returning `NotImplemented` signals that the backend does not support this operation.

Uarray



Multi-method system for PyData and SciPy ecosystem

`__ua_convert__`

All dispatchable arguments are passed through `__ua_convert__` before being passed into `__ua_function__`. This protocol has the signature `(dispatchables, coerce)`, where `dispatchables` is an iterable of `Dispatchable` and `coerce` is whether or not to coerce forcefully. `dispatch_type` is the mark of the object to be converted, and `coerce` specifies whether or not to "force" the conversion. By convention, operations larger than `O(log n)` (where `n` is the size of the object in memory) should only be done if `coerce` is `True`. In addition, there are arguments wrapped as non-coercible via the `coercible` attribute, if these *must* be coerced, then one should return `NotImplemented`.

A convenience wrapper for converting a single object, `wrap_single_convertor` is provided.

Returning `NotImplemented` signals that the backend does not support conversion of the given object.