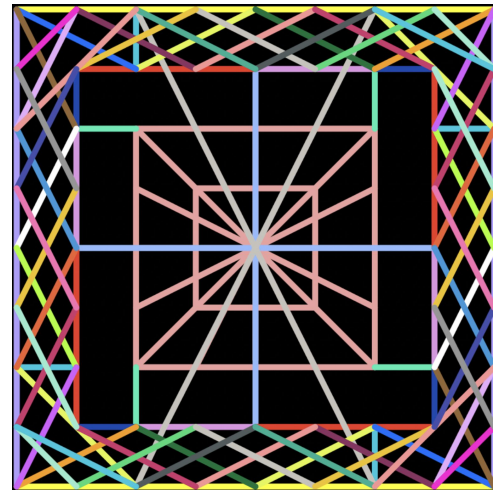


Understanding Dispatching Approaches in the Scientific Python Ecosystem

- By Aditi Juneja and Sebastian Berg

Aditi Juneja (@Schefflera-Arboricola)

- **Grants/programs:** Google Summer of Code 2024, NumFOCUS's Small Development Grant (R3, 2024), CZI*
- **Open source projects/contributions:** nx-parallel, NetworkX (Core developer), scikit-image (dispatching), other small small contributions in various scientific Python projects!
- **Communities and Volunteering roles:** SciPy India (core-organiser), GSoC 2025 mentor (NetworkX), SciPy 2025 (proceedings paper reviewer), Scientific Python (maintainer - dispatch team), IndiaFOSS 2025 ("FOSS in Science" devroom manager), PyDelhi (reviewer), PyData Global 2024 (reviewer), PyCon US 2025 (proposal mentor)



Find work blogs, talk slides, CV, etc. at <https://github.com/Schefflera-Arboricola/blogs>

Dispatching??

Arriving Sunday

Dispatched



Apsara Dustless Chalks | 4x Longer Than Regular Chalks | Hypoallergenic Chalk for Safe Using | Non-dust Chalk for Clean Writing | Available in Vibrant Colors | Ideal for Schools Box of 100 Chalks.

Sold by: Cocoblu Retail

₹343.00

Track package

Request cancellation

Return or replace items

Share gift receipt

Leave seller feedback

Write a product review

Dispatching– in general

Arriving Sunday

Dispatched



Apsara Dustless Chalks | 4x Longer Than Regular Chalks | Hypoallergenic Chalk for Safe Using | Non-dust Chalk for Clean Writing | Available in Vibrant Colors | Ideal for Schools Box of 100 Chalks.

Sold by: Cocoblu Retail

₹343.00

Track package

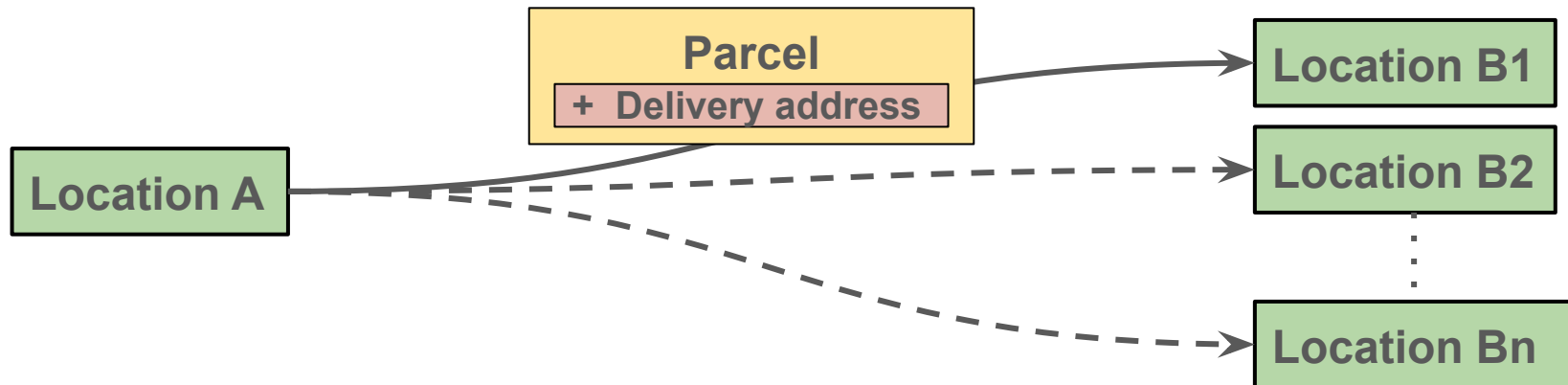
Request cancellation

Return or replace items

Share gift receipt

Leave seller feedback

Write a product review



Dispatching – in programming

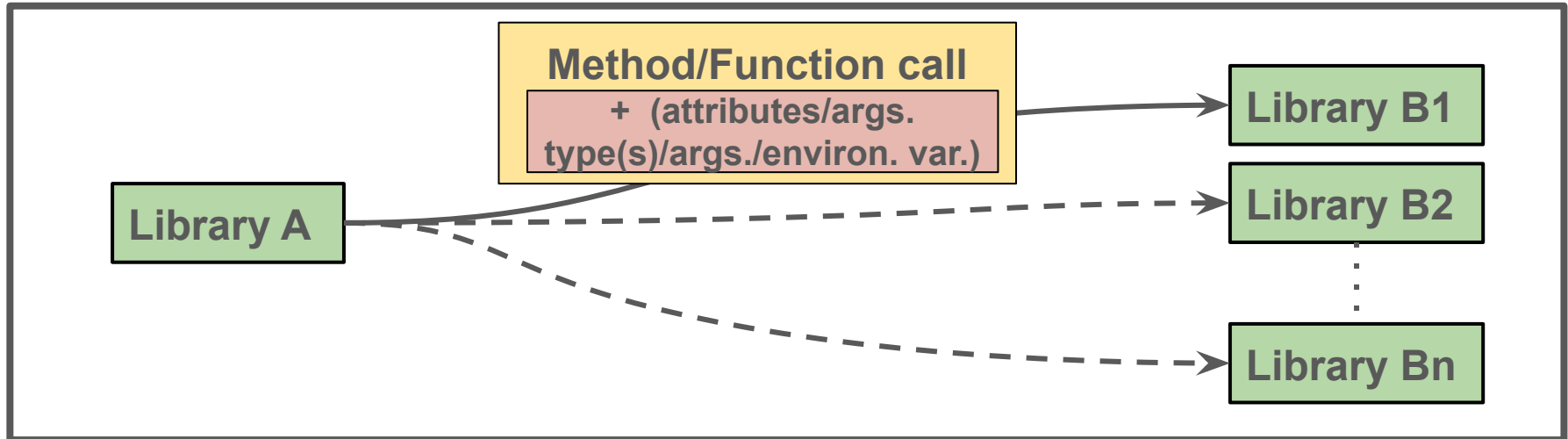
“In **computer science**, **dynamic dispatch** is the process of selecting which implementation of a **polymorphic** operation (**method** or function) to call at **run time**.”

Ref. https://en.wikipedia.org/wiki/Dynamic_dispatch

Dispatching – in programming

“In **computer science**, **dynamic dispatch** is the process of selecting which implementation of a **polymorphic** operation (**method** or function) to call at **run time**.”

Ref. https://en.wikipedia.org/wiki/Dynamic_dispatch





**But.. Why would
anyone want to
dispatch?**



**But.. Why would
anyone want to
dispatch?**

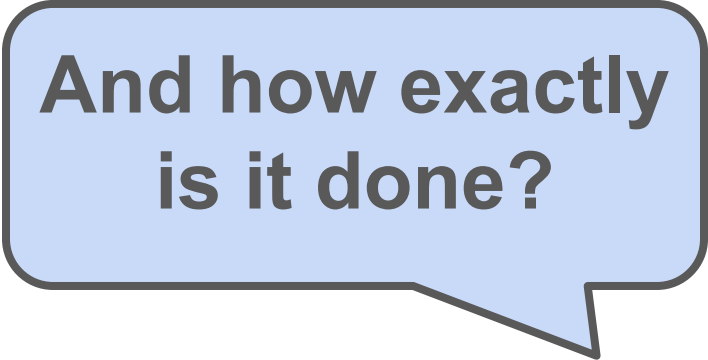
**Improve
Performance**

**Type
Compatibility**

**Enable different
workflows**



**But.. Why would
anyone want to
dispatch?**



**And how exactly
is it done?**

Example

What does dispatching mean here?

library_A

```
def add(x, y, z):  
    return x + y + z
```

Importing and Calling (hard-coded)

Importing and Calling (hard-coded)

library_A

```
def add(x, y, z, libB=None):  
    if libB == "library_B1":  
        import library_B1 as libB1  
        z = float(z)  
        return libB1.mod1.add(x, y, z)  
    elif libB == "library_B2" :  
        ...  
        ...  
    else:  
        return x + y + z
```

Importing and Calling (hard-coded)

`library_A`

```
def add(x, y, z, libB=None):  
    if libB == "library_B1":  
        import library_B1 as libB1  
        z = float(z)  
        return libB1.mod1.add(x, y, z)  
    elif libB == "library_B2" :  
        ...  
        ...  
    else:  
        return x + y + z
```

Can we make
this more
general?

Importing and Calling (hard-coded)

Inside `library\_A`

```
def add(x, y, z, libB=None):  
    if libB == "library_B1":  
        import library_B1 as libB1  
        z = float(z)  
        return libB1.mod1.add(x, y, z)  
    elif libB == "library_B2" :  
        ...  
        ...  
    else:  
        return x + y + z
```

Can we make
this more
general?

Steps involved

Steps involved

library_A

```
def add(x, y, z, libB=None):  
    if libB == "library_B1":  
        import library_B1 as libB1  
        z = float(z)  
        return libB1.mod1.add(x, y, z)  
    elif libB == "library_B2" :  
        ...  
        ...  
    else:  
        return x + y + z
```

Get `libB`

Import `libB`

Convert args for `libB`

Find `add` in `libB` and call it with converted args

Steps involved

library_A

```
def add(x, y, z, libB=None):  
    if libB == "library_B1":  
        import library_B1 as libB1  
        z = float(z)  
        return libB1.mod1.add(x, y, z)  
    elif libB == "library_B2" :  
        ...  
        ...  
    else:  
        return x + y + z
```

... maybe we can
delegate some of
these steps to `libB` ...

Get `libB`

Import `libB`

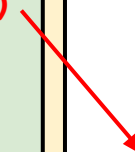
Convert args for `libB`

Find `add` in `libB` and
call it with converted
args

Step 4: Find `add` in `libB` and call it

Inside `library\_A`

```
def add(x, y, z, libB=None):  
    if libB == "library_B1":  
        import library_B1 as libB1  
        z = float(z)  
        return libB1.mod1.add(x, y, z)  
    elif libB == "library_B2" :  
        ...  
        ...  
    else:  
        return x + y + z
```



Find `add` in `libB` and call it with converted args

Step 4: Find `add` in `libB` and call it

Inside `library\_A`

```
def add(x, y, z, libB=None):  
    if libB == "library_B1":  
        import library_B1 as libB1  
        z = float(z)  
        all_funcs = libB1.get_all_funcs()  
        libb1_add = all_funcs.add  
        return libB1_add(x, y, z)  
    elif libB == "library_B2" :  
        ...  
        ...  
    else:  
        return x + y + z
```

Get a namespace* of all the functions in `libB1`

Extract `add` from this namespace

Call the extracted `add`

**Assumption: all functions in the namespace have unique names*

Step 3: convert args for `add` in `libB`

Inside `library\_A`

```
def add(x, y, z, libB=None):  
    if libB == "library_B1":  
        import library_B1 as libB1  
        z = float(z)  
        all_funcs = libB1.get_all_funcs()  
        libb1_add = all_funcs.add  
        return libB1_add(x, y, z)  
    elif libB == "library_B2" :  
        ...  
        ...  
    else:  
        return x + y + z
```

Convert args for `libB`



Step 3: convert args for `add` in `libB`

Inside `library\_A`

```
def add(x, y, z, libB=None):  
    if libB == "library_B1":  
        import library_B1 as libB1  
        x, y, z = libB1.convert_args(add, x, y, z)  
        all_funcs = libB1.get_all_funcs()  
        libb1_add = all_funcs.add  
        return libB1_add(x, y, z)  
    elif libB == "library_B2" :  
        ...  
        ...  
    else:  
        return x + y + z
```

Convert function
in `libB`



Step 2: import `libB`

Inside `library\_A`

```
def add(x, y, z, libB=None):  
    if libB == "library_B1":  
        import library_B1 as libB1  
        x, y, z = libB1.convert_args(add, x, y, z)  
        all_funcs = libB1.get_all_funcs()  
        libb1_add = all_funcs.add  
        return libB1_add(x, y, z)  
    elif libB == "library_B2" :  
        ...  
        ...  
    else:  
        return x + y + z
```

Import `libB`

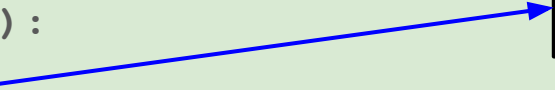


Step 2: import `libB`

Inside `library\_A`

```
def add(x, y, z, libB=None):  
    lib = __import__(libB)  
    if libB == "library_B1":  
        x, y, z = lib.convert_args(add, x, y, z)  
        all_funcs = lib.get_all_funcs()  
        lib_add = all_funcs.add  
        return lib_add(x, y, z)  
    elif libB == "library_B2" :  
        ...  
        ...  
    else:  
        return x + y + z
```

Generalising importing



Step 2: import `libB`

Inside `library\_A`

```
def add(x, y, z, libB=None):  
    lib = __import__(libB)  
    if libB == "library_B1":  
        x, y, z = lib.convert_args(add, x, y, z)  
        all_funcs = lib.get_all_funcs()  
        lib_add = all_funcs.add  
        return lib_add(x, y, z)  
    elif libB == "library_B2" :  
        ...  
        ...  
    else:  
        return x + y + z
```

Generalising importing

Implication:
code inside
if-else also
became
generalised

Step 2: import `libB`

Inside `library\_A`

```
def add(x, y, z, libB=None):  
    if libB != None:  
        lib = __import__(libB)  
        x, y, z = lib.convert_args(add, x, y, z)  
        all_funcs = lib.get_all_funcs()  
        lib_add = all_funcs.add  
        return lib_add(x, y, z)  
    else:  
        return x + y + z
```

Generalising importing

Getting rid of
`if-else`
conditions

Dispatching

Inside `library\_A`

```
def add(x, y, z, libB=None):  
    if libB == "library_B1":  
        import library_B1 as libB1  
        z = float(z)  
        return libB1.mod1.add(x, y, z)  
    elif libB == "library_B2" :  
        ...  
        ...  
    else:  
        return x + y + z
```

Get `libB`

`library\_A`

Import `libB`

`library\_A`

Convert args
for `libB`

`library\_B1`

Find `add` in `libB` and call
it with converted args

`library\_B1`

Dispatching

Inside `library\_A` (import and call)

```
def add(x, y, z, libB=None):
    if libB == "library_B1":
        import library_B1 as libB1
        z = float(z)
        return libB1.mod1.add(x, y, z)
    elif libB == "library_B2" :
        ...
        ...
    else:
        return x + y + z
```

Inside `library\_A` (dispatching)

```
def add(x, y, z, libB=None):
    if libB != None:
        lib = __import__(libB)
        x, y, z = lib.convert_args(add,
        x, y, z)
        all_funcs = lib.get_all_funcs()
        lib_add = all_funcs.add
        return lib_add(x, y, z)
    else:
        return x + y + z
```

Dispatching

Inside `library\_A` (import and call)

```
def add(x, y, z, libB=None):
    if libB == "library_B1":
        import library_B1 as libB1
        z = float(z)
        return libB1.mod1.add(x, y, z)
    elif libB == "library_B2" :
        ...
        ...
    else:
        return x + y + z
```

Can we do something more here?

Inside `library\_A` (dispatching)

```
def add(x, y, z, libB=None):
    if libB != None:
        lib = __import__(libB)
        x, y, z = lib.convert_args(add,
                                   x, y, z)
        all_funcs = lib.get_all_funcs()
        lib_add = all_funcs.add
        return lib_add(x, y, z)
    else:
        return x + y + z
```

Dispatching

Yes! use decorator.
Generalising for all functions

Inside `library\_A`

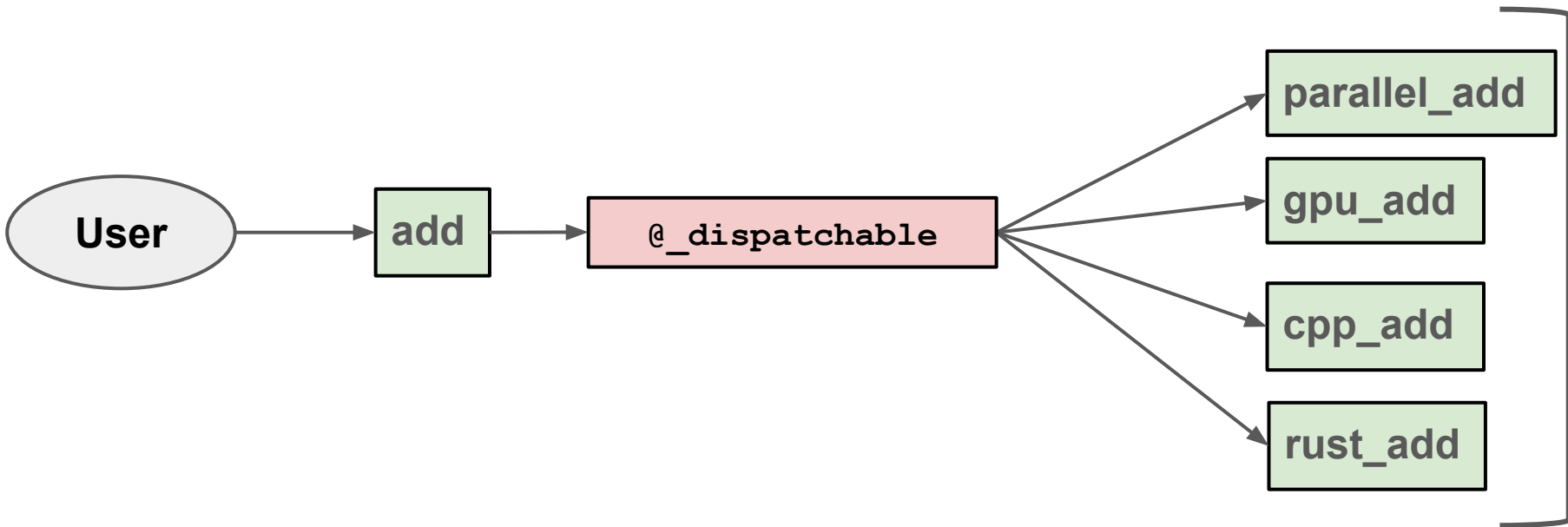
```
@_dispatchable
def add(x, y, z):
    return x + y + z
```

**there are other ways
of dispatching as well.*

```
def _dispatchable():
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        # check for libB kwarg in the function signature
        libB = kwargs.get("libB")
        try:
            lib = __import__(libB)
            args = lib.convert_args(func, *args, **kwargs)
            all_funcs = lib.get_all_funcs()
            lib_func = all_funcs.func
            return lib_func(args)
        except ImportError:
            return func(*args, **kwargs)
    return wrapper
```

Dispatching – Summary

Backends/alt.
implementations





**How is dispatching
done in real projects?**

Some projects we'll discuss:

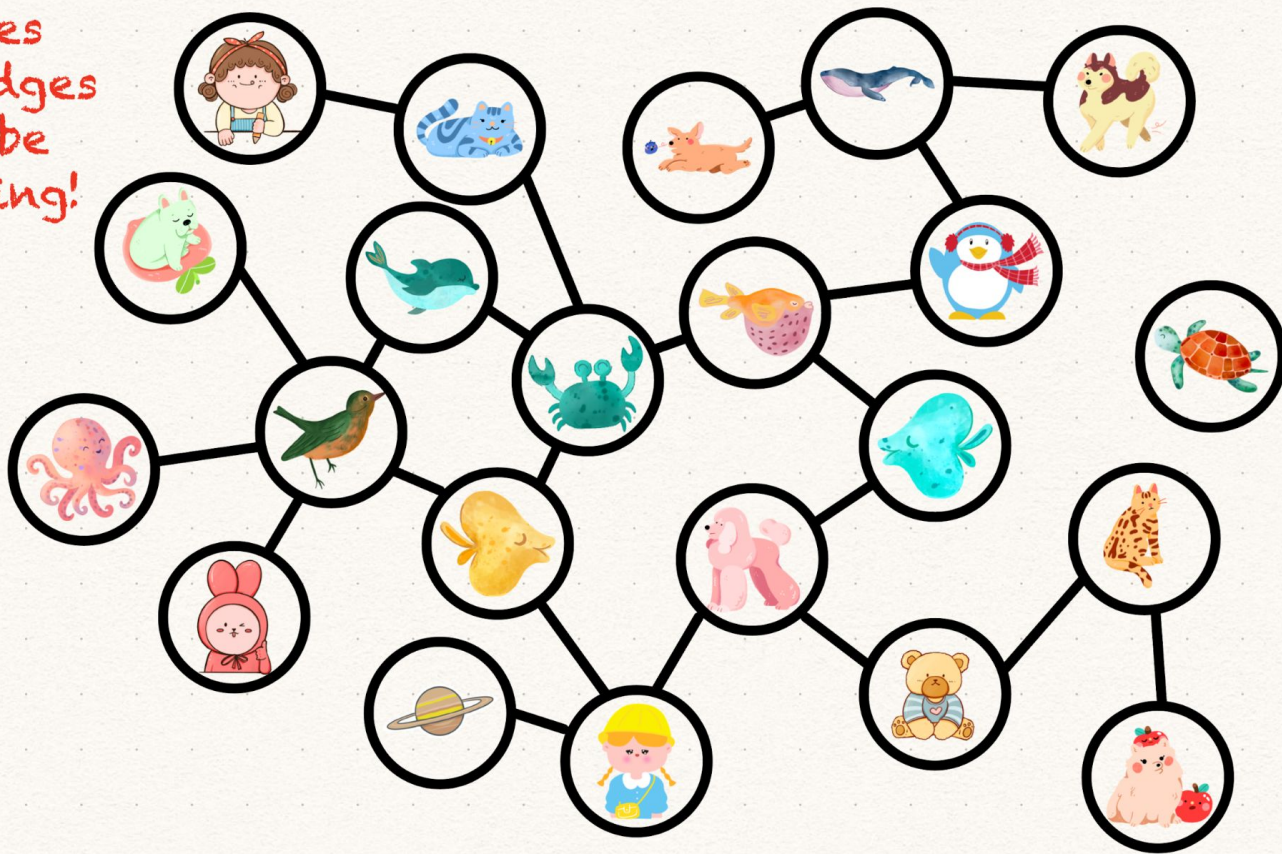
- Graphs: **NetworkX**
- Arrays:
 - **NumPy** : `__array_function__`
 - Array API standards → Pradyot's talk!
- **Scikit-image** and **spatch**

Dispatching in NetworkX for Graphs

What is NetworkX?

NetworkX
is a graph
analysis
library

Nodes
and edges
can be
anything!



Steps involved in dispatching

Inside `library\_A`

```
def add(x, y, z, libB=None):  
    if libB == "library_B1":  
        import library_B1 as libB1  
        z = float(z)  
        return libB1.mod1.add(x, y, z)  
    elif libB == "library_B2" :  
        ...  
        ...  
    else:  
        return x + y + z
```

Get `libB`

Import `libB`

*

Convert args for `libB`

Find `add` in `libB` and call it with converted args

Getting the backend name in NetworkX

Demo:

<https://colab.research.google.com/drive/16bUxGvBoBAg1dBRc6yfjxixlo7DocM8s?usp=sharing>

4 ways of to get backend name in NetworkX

```
nx.betweenness centrality(CuG)
```

CuG.__networkx_backend__
contains the backend name
as a string

```
nx.betweenness centrality(G,  
backend = "parallel")
```

```
with nx.config(backend_priority =  
["cugraph", "graphblas", "parallel"]):  
    nx.betweenness centrality(G)
```

```
$ NETWORKX_BACKEND_PRIORITY="graphblas"  
$ python nx_code.py
```

Inside `networkx`

```
@_dispatchable  
def betweenness centrality(  
    G, k, ... ,seed  
):  
    ...
```

nx-parallel

nx-cugraph

Python-
graphblas

4 ways of to get backend name in NetworkX

```
nx.betweenness centrality(CuG)
```

**Type-based
dispatching**

CuG.__networkx_backend__
contains the backend name
as a string

Inside `networkx`

```
@_dispatchable  
def betweenness centrality(  
    G, k, ... ,seed  
    ...  
)
```

nx-parallel

nx-cugraph

Python-
graphblas

```
nx.betweenness centrality(G,  
    backend = "parallel")
```

**Name-based
dispatching**

```
with nx.config(backend_priority =  
    ["cugraph", "graphblas", "parallel"]):  
    nx.betweenness centrality(G)
```

```
$ NETWORKX_BACKEND_PRIORITY="graphblas"  
$ python nx_code.py
```

Getting the backend implementation

Python entry-points to get all the installed backends...

... and their metadata – supported functions and convert functions.

What are entry-points?

Entry-points are used **to extend a functionality** of a library.

Main library



Entry point

Extension (or plug-in) libraries



Dispatching with entry-points is exploiting entry-points to the max—

i.e. change the whole machinery inside the vacuum cleaner (main library's implementation)— and just keep the outside red UI-plastic-buttons frame (user-API).

In NetworkX

```
>>> from importlib.metadata import entry_points
>>> entry_points(group="networkx.backends")
(
  EntryPoint(
    name="parallel",
    value="nx_parallel.interface: BackendInterface",
    group="networkx.backends",
  ),
  ...
  ...
  EntryPoint(
    name="cugraph",
    value="nx_cugraph.interface:BackendInterface",
    group="networkx.backends",
  ),
)
```

Inside nx-parallel backend

In backend's `pyproject.toml`

```
[project.entry-points."networkx.backends"]
```

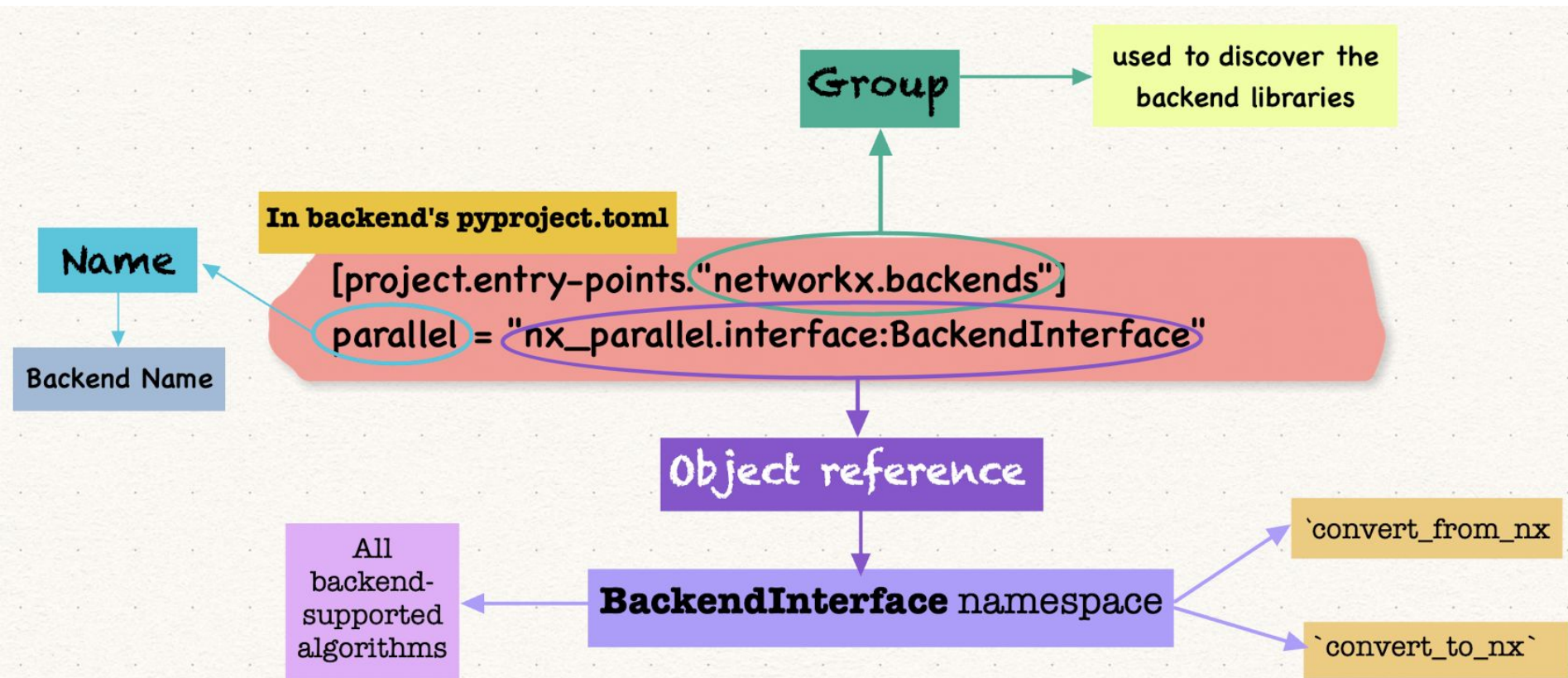
```
parallel = "nx_parallel.interface:BackendInterface"
```

Group

Name

Object reference

Inside nx-parallel backend



Other features of NetworkX dispatching (if time permits)

- `can_run` and `should_run` : quick checks by backend to optimise dispatching workflow
- Testing backend on NetworkX's test suite - `NETWORKX_TEST_BACKEND="parallel"`
- Showing Backend docs in NetworkX docs ([see here](#))
- Caching of converted graphs
- `.backends` attribute to get the set of all the installed backends that implement a particular function. For example:

```
>>> nx.betweenness centrality.backends  
{ 'parallel' }
```
- Specialised backend priority for algorithms, generators,.. etc.
- Logging
- Fallback

Read more at <https://networkx.org/documentation/latest/reference/backends.html>

Dispatching in NumPy for Arrays

Steps involved in dispatching

Inside `library\_A`

```
def add(x, y, z, libB=None):  
    if libB == "library_B1":  
        import library_B1 as libB1  
        z = float(z)  
        return libB1.mod1.add(x, y, z)  
    elif libB == "library_B2" :  
        ...  
        ...  
    else:  
        return x + y + z
```

Get `libB`

Import `libB` *

Convert args for `libB`

Find `add` in `libB` and call it with converted args *

Steps involved in dispatching

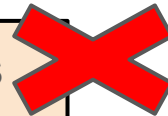
Inside `library\_A`

```
def add(x, y, z, libB=None):  
    if libB == "library_B1":  
        import library_B1 as libB1  
        z = float(z)  
        return libB1.mod1.add(x, y, z)  
    elif libB == "library_B2" :  
        ...  
        ...  
    else:  
        return x + y + z
```

Get `libB`

Import `libB` *

Convert args
for `libB`



Array conversions are
more complex—
type-based dispatching
is preferred with arrays

Find `add` in `libB` and call it
with converted args *

Getting the backend implementation

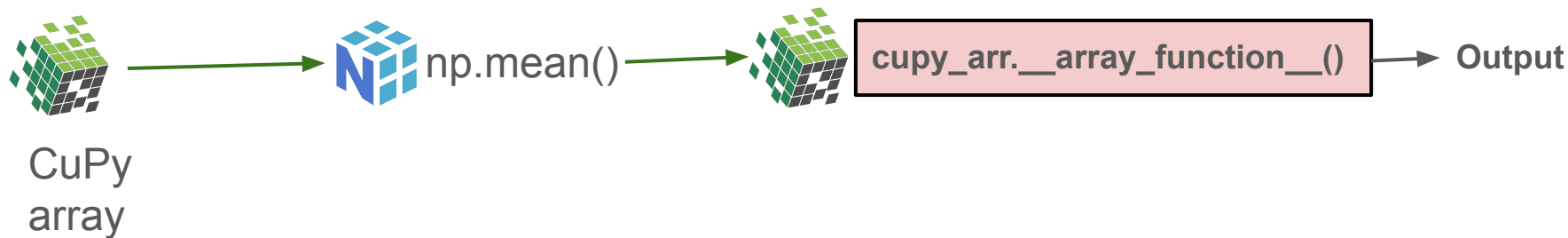
```
input_array.__array_function__()
```

directly calls the apt. array library's implementation, corresponding to the function that's being called, if it exists.

Getting the backend implementation

```
input_array.__array_function__()
```

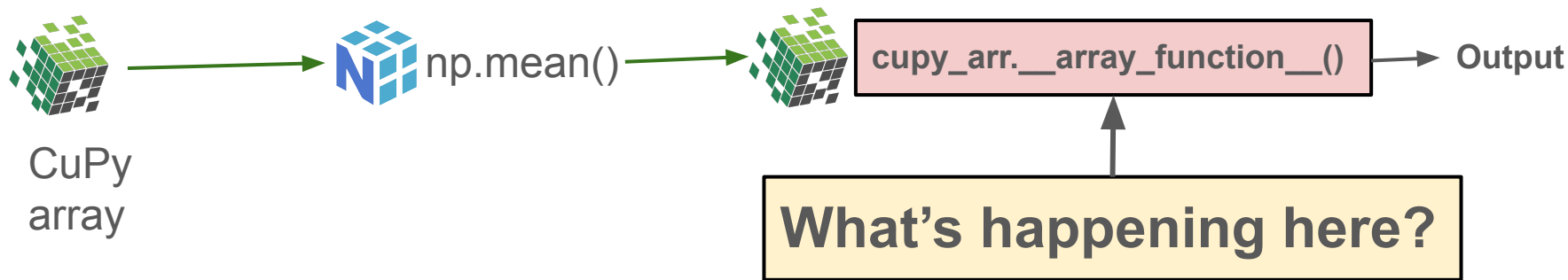
directly calls the apt. array library's implementation, corresponding to the function that's being called, if it exists.



Getting the backend implementation

```
input_array.__array_function__()
```

directly calls the apt. array library's implementation, corresponding to the function that's being called, if it exists.



Inside the CuPy (simplified)

```
class ndarray:
    def __array_function__(self, func, types, *args, **kw):
        if not supported_function(func): # np.mean not implemented
            return NotImplemented

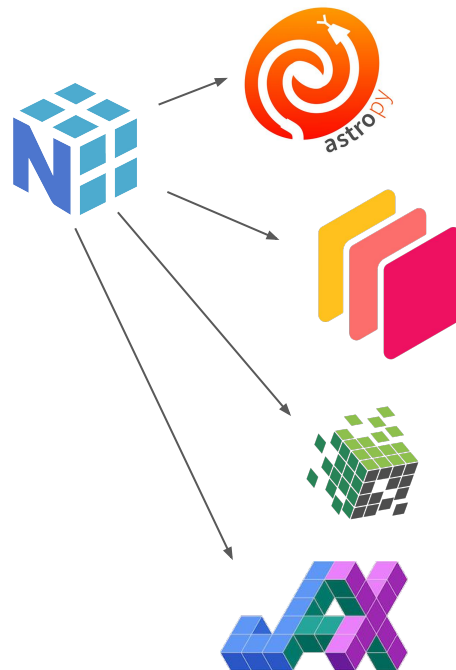
        for t in types: # checking array types
            if not issubclass(t, (ndarray, numpy.ndarray)):
                return NotImplemented

        return cupy.mean(*args, **kw)
```

No fallbacks!

Dispatching in NumPy

- Started around ~2013:
 - `__array_ufunc__` (NEP 13)
 - subset (“universal functions”)
 - `__array_function__` (NEP 18 + 35)
 - almost all other functions.
- Use-cases/users:
 - SciPy sparse
 - `astropy.units` (“NumPy array of meters”)
 - Dask array (distributed)
 - CuPy (GPU), JAX, cupynumeric...



Dispatching in scikit-image

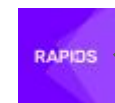
Dispatching in scikit-image

- **Challenge**: Hard to adopt Array API standards (Cython-heavy array consuming library)– therefore using entry-points for dispatching.
- **Goal**: want it to look like type-based dispatching internally entry-point based dispatching.
- **Current state**: No array conversions in dispatching code! – backend developers and users need to take care of the types. ([Read more](#))

Possible Solution : <https://github.com/scientific-python/spatch>

NetworkX

(type-based and name-based)



`nx.betweenness_centrality()`

`backend=
NETWORKX_BACKEND_PRIORITY
'nx.config()'`

**Backend
Name**

`G.__networkx_backend__`

Entry-points

BackendInterface Namespace

Backend
Implementation

Convert
functions

Output

NumPy

(type-based)



`np.mean()`



`dask_arr.__array_function__()`

Output

`cupy_arr.__array_function__()`

Not Implemented

Dispatching in Scientific Python ecosystem

- SPEC 2 : <https://scientific-python.org/specs/spec-0002/>
- spatch : <https://github.com/scientific-python/spatch/>
- Array API standards: <https://data-apis.org/array-api/latest/>
- NumPy's type-based dispatching
 - <https://numpy.org/neps/nep-0037-array-module.html>
 - <https://numpy.org/neps/nep-0047-array-api-standard.html>
- NetworkX
 - <https://networkx.org/documentation/latest/reference/backends.html>
 - <https://networkx.org/documentation/latest/reference/configs.html>
 - Dispatch meetings: <https://scientific-python.org/calendars/networkx.ics>
 - <https://github.com/networkx/networkx/issues?q=is%3Aissue%20state%3Aopen%20label%3ADispatching>
- Scikit-image
 - [scikit-image-PR#7520](https://github.com/scikit-image/scikit-image/pull/7727)
 - <https://github.com/scikit-image/scikit-image/pull/7727>
 - <https://github.com/rapidsai/cucim/issues/829>
- Scikit-learn
 - <https://github.com/scikit-learn/scikit-learn/pull/30250>
 - <https://youtu.be/f42C1daBNrg?si=A9mZ2mZd2HzEhu8S>
- SciPy's Array API adoption
 - https://docs.scipy.org/doc/scipy/dev/api-dev/array_api.html
 - https://youtu.be/16rB-fosAWw?si=ys_-ZTnUKvO_aZKu
- DataFrame API standards
 - <https://github.com/narwhals-dev/narwhals>
 - <https://data-apis.org/dataframe-api/draft/>
- Scientific Python discord(#dispatching thread): <https://discord.com/invite/vur45CbwMz>