Accelerating scientific Python code with dispatching: Graphs and Arrays

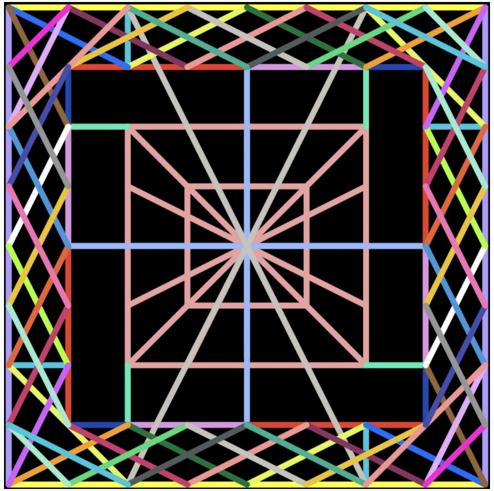
- By Aditi Juneja and Sebastian Berg

Aditi Juneja (@Schefflera-Arboricola)

- **Grants/programs**: Google Summer of Code 2024, NumFOCUS's Small Development Grant (R3, 2024), CZI (NetworkX Internship)

- **Open-source projects/contributions**: nx-parallel, NetworkX (Core developer), scikit-image (dispatching), other small small contributions in various scientific Python projects!

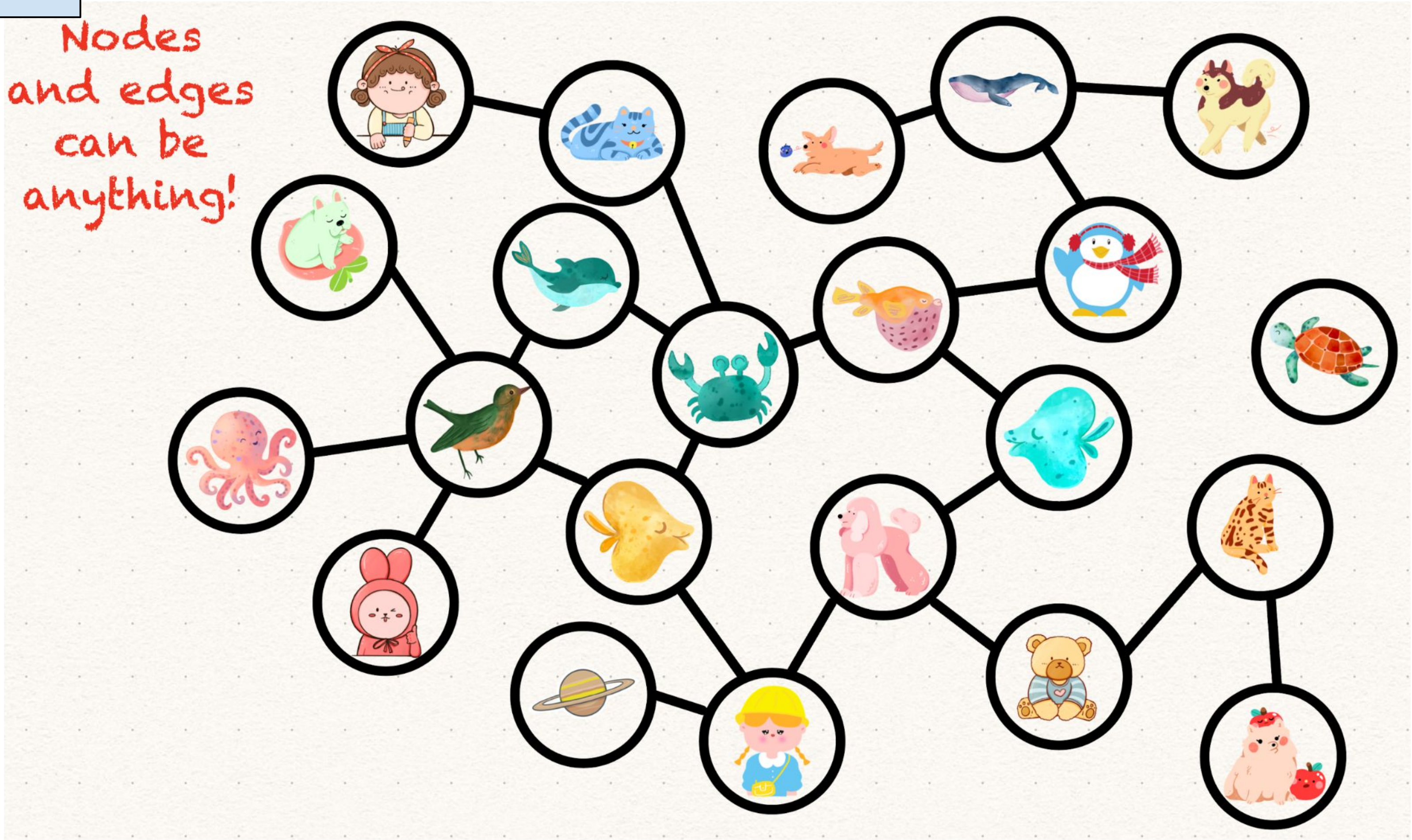
- **Communities and Volunteering roles**: SciPy India (core-organiser), GSoC 2025 mentor (NetworkX), SciPy 2025 (proceedings paper reviewer), Scientific Python (maintainer - dispatch team), IndiaFOSS 2025 ("FOSS in Science" devroom manager), PyDelhi (reviewer), PyData Global 2024 (reviewer), PyCon US 2025 (proposal mentor)



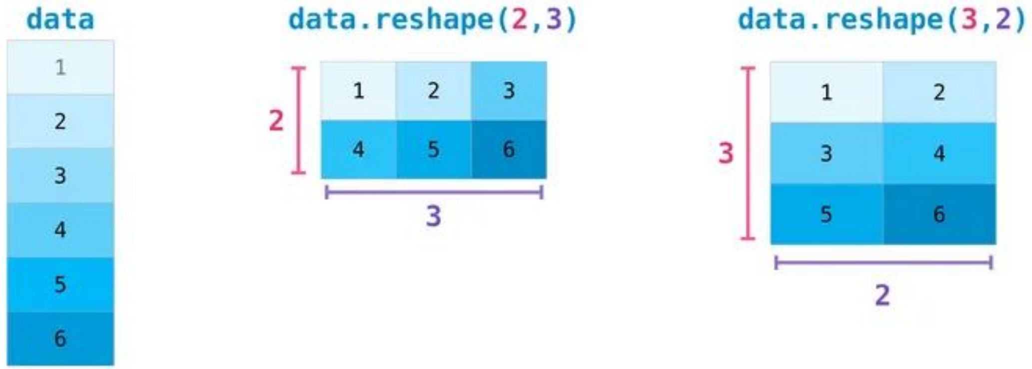
Find more about my work at https://github.com/Schefflera-Arboricola/blogs

Accelerating scientific Python code with dispatching: Graphs and Arrays

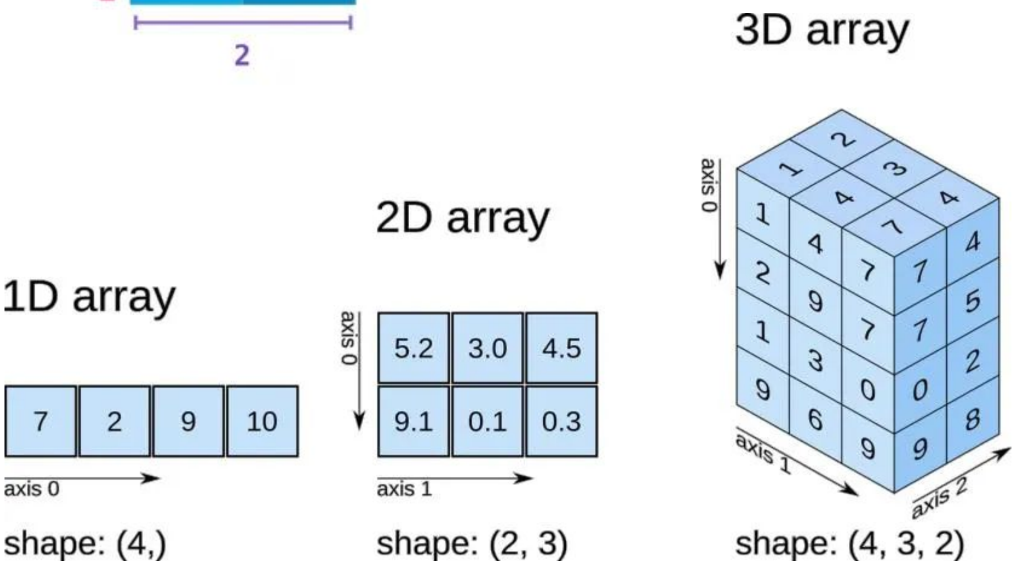
Graphs



Arrays



data
1
2
3
4
5
6
data.reshape(2,3)
1 2 3
4 5 6
2
3
data.reshape(3,2)
1 2
3 4
5 6
3
2

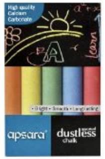


1D array
7 2 9 10
axis 0
shape: (4,)
2D array
5.2 3.0 4.5
9.1 0.1 0.3
axis 0
axis 1
shape: (2, 3)
3D array
axis 0
axis 1
axis 2
shape: (4, 3, 2)

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

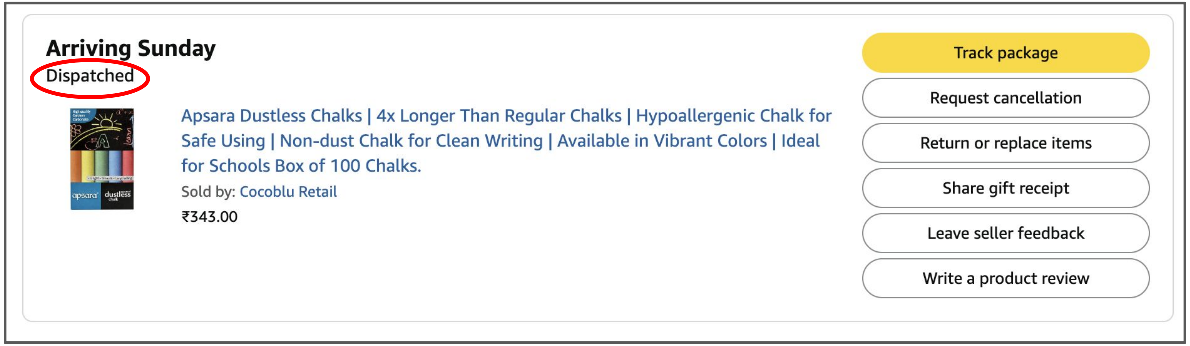
Return or replace items

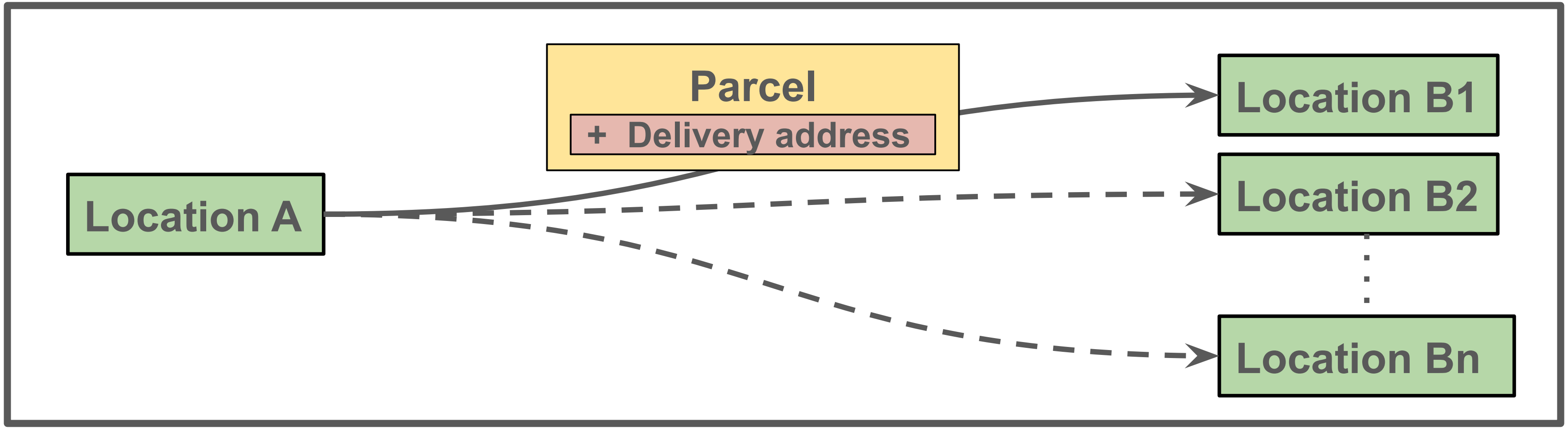
Share gift receipt

Leave seller feedback

Write a product review

Dispatching– in general





Dispatching – in programming

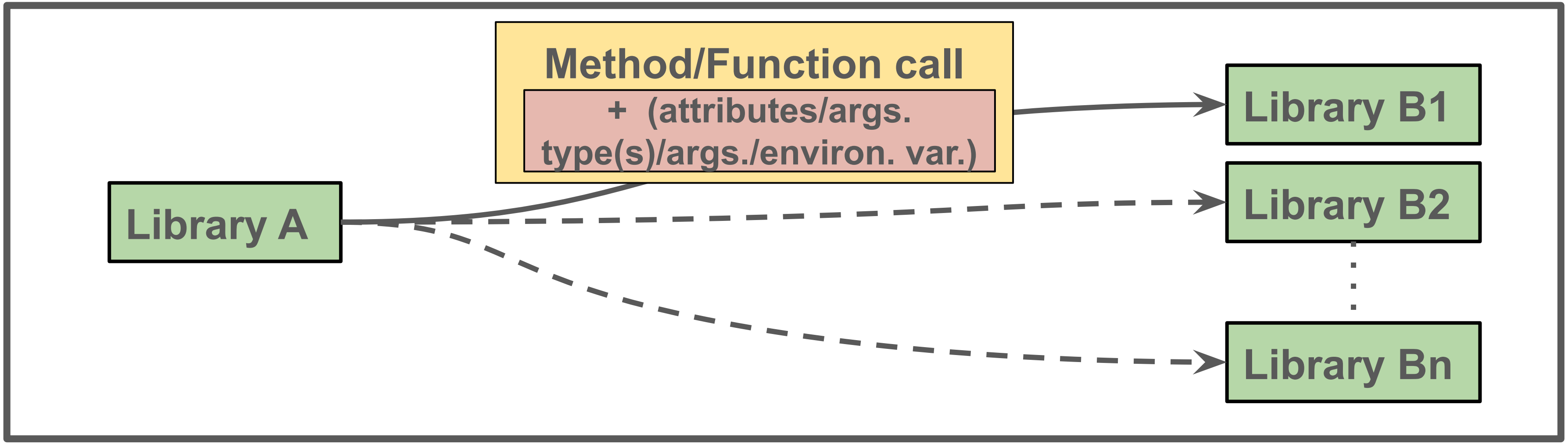
“In computer science, **dynamic dispatch** is the process of selecting which implementation of a polymorphic operation (method or function) to call at **run time**.”

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.**”

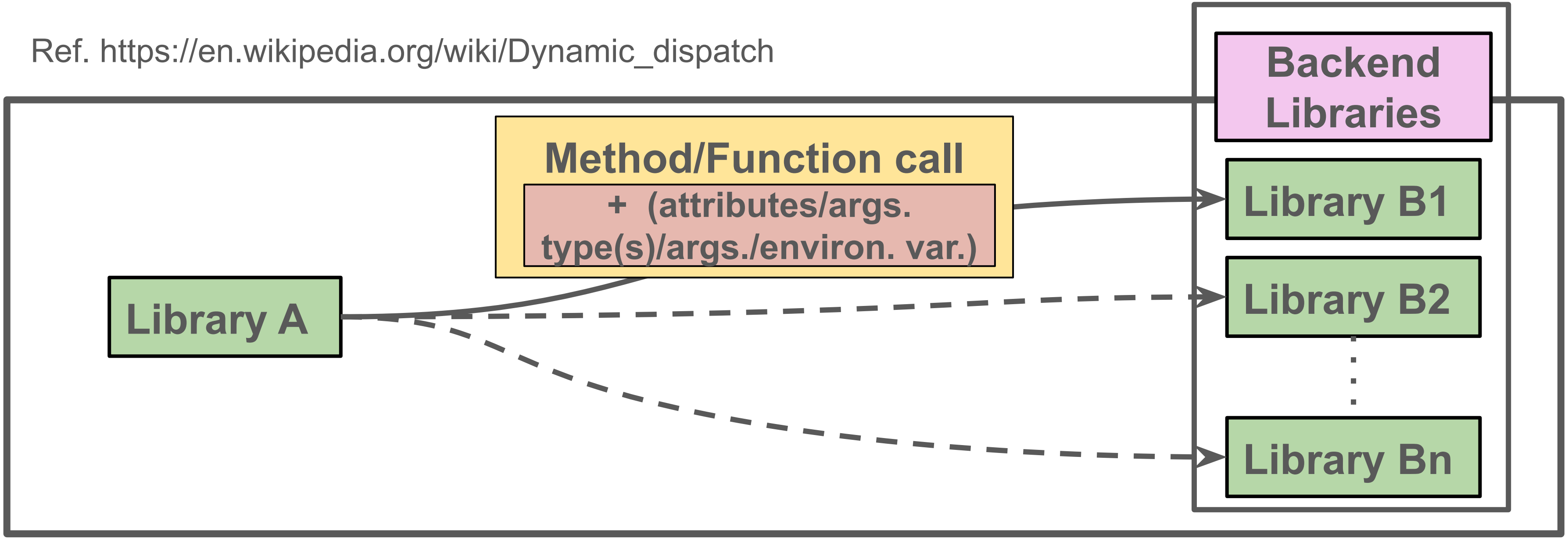
Ref. https://en.wikipedia.org/wiki/Dynamic_dispatch



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
we want to
dispatch?

But.. Why would we want to dispatch?

Improve Performance

Type Compatibility

Enable different workflows

But.. Why would we want to dispatch?

And how do we do it?

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)

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

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

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

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

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.__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.__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.__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.__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.__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.__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.__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

Step 2: import `libB`

Can we do something more here?

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.__all_funcs__()
        lib_add = all_funcs.add
        return lib_add(x, y, z)
    else:
        return x + y + z
```

Dispatching

Yes! Decorators.
Generalising for all functions

Inside `library_A`

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

```
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.__all_funcs__()
            lib_func = all_funcs.func
            return lib_func(args)
         except ImportError:
             return func(*args, **kwargs)
     return wrapper
```

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

How is dispatching done in real projects?

Some projects we’ll discuss:

- Graphs: **NetworkX**

- Arrays:
    - **NumPy** dispatching : __array_function__
    - **Array API standards**
    - **Scikit-image**

Dispatching in NetworkX for Graphs

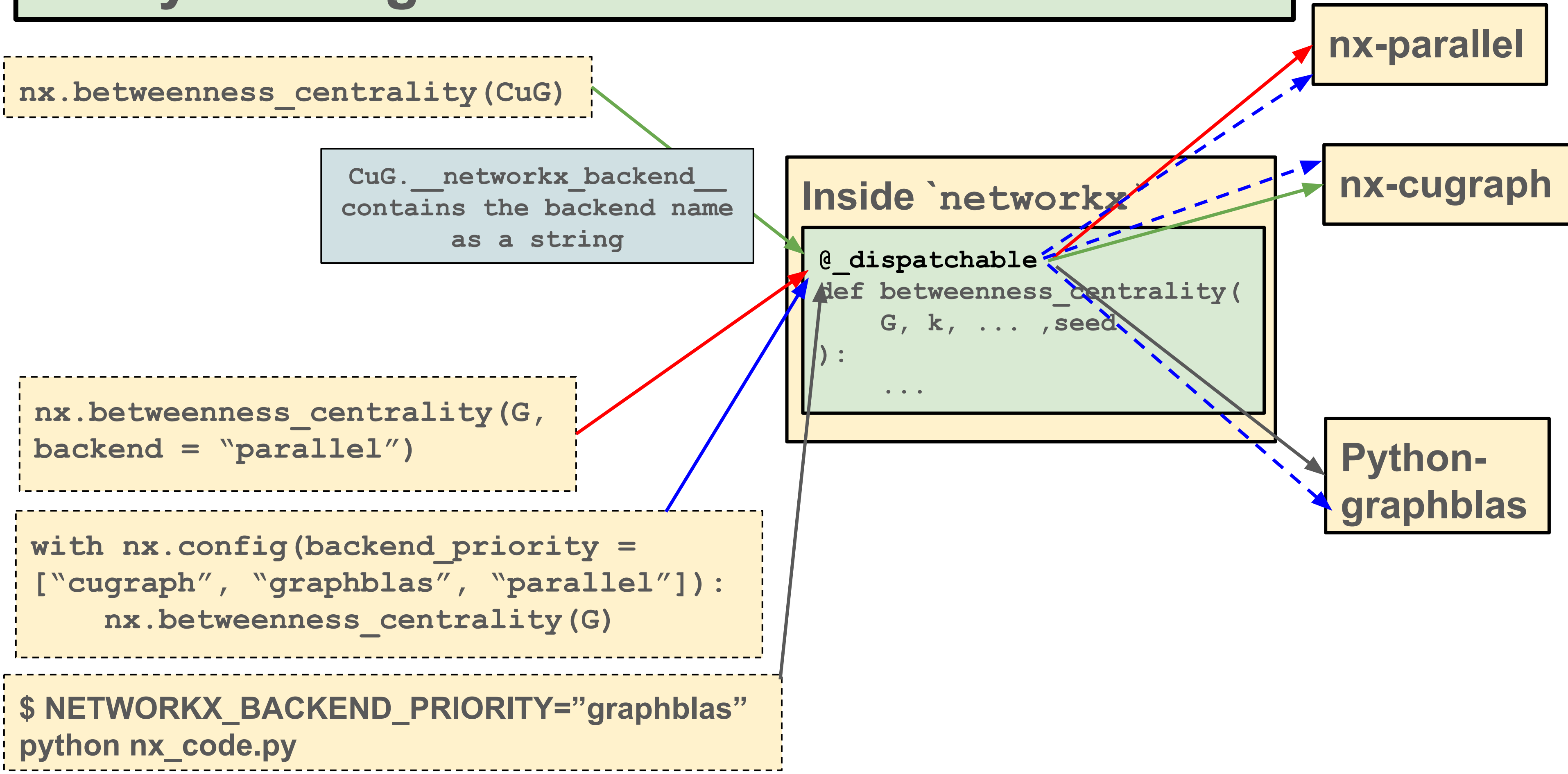
What is NetworkX?

NetworkX is a network/graph analysis library

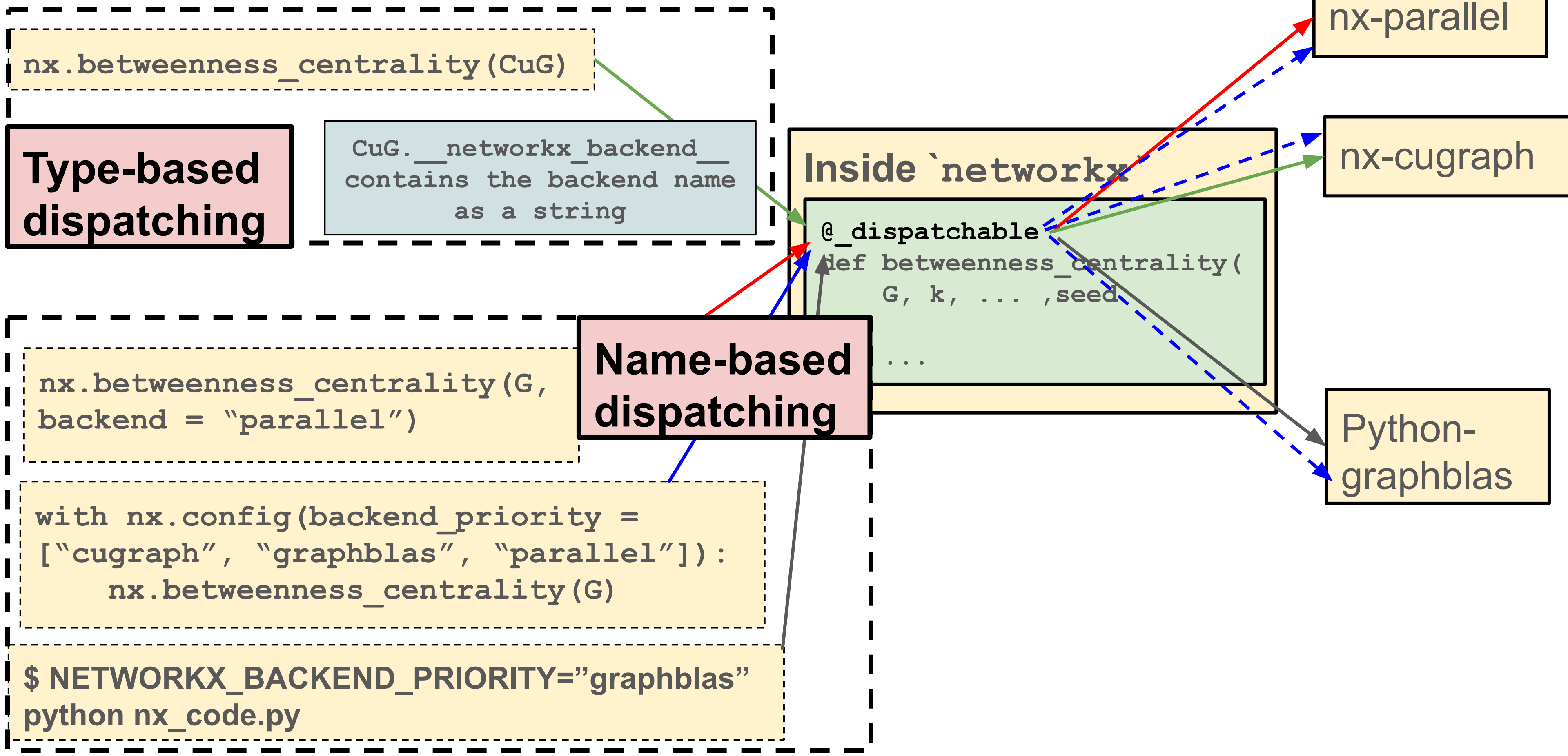
Demo

https://colab.research.google.com/drive/1n1oqMr__BXdg7RSaM8eZpwM07utaEru9?usp=sharing

4 ways of to get backend name in NetworkX



4 ways of to get backend name in NetworkX



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

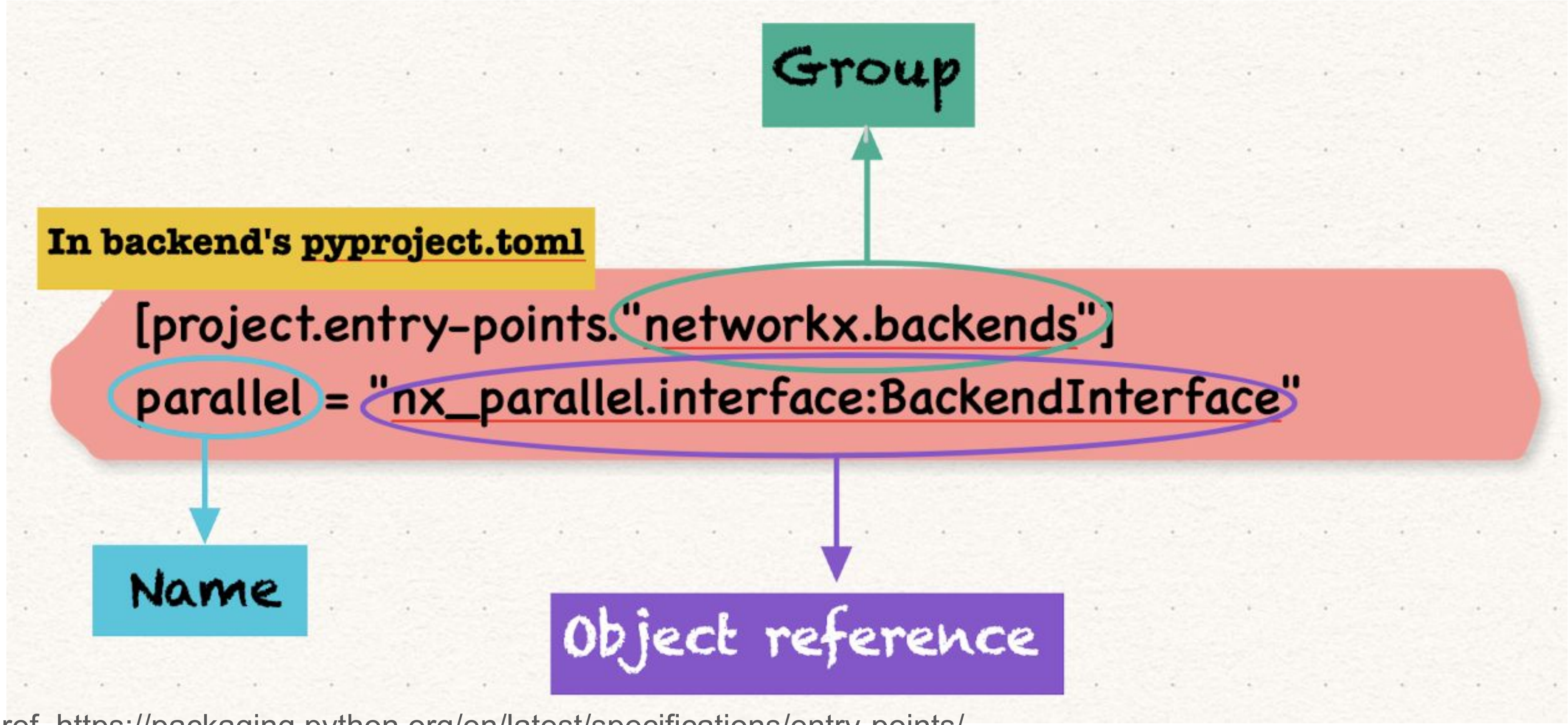
Entry point

Extension (or plug-in) libraries

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

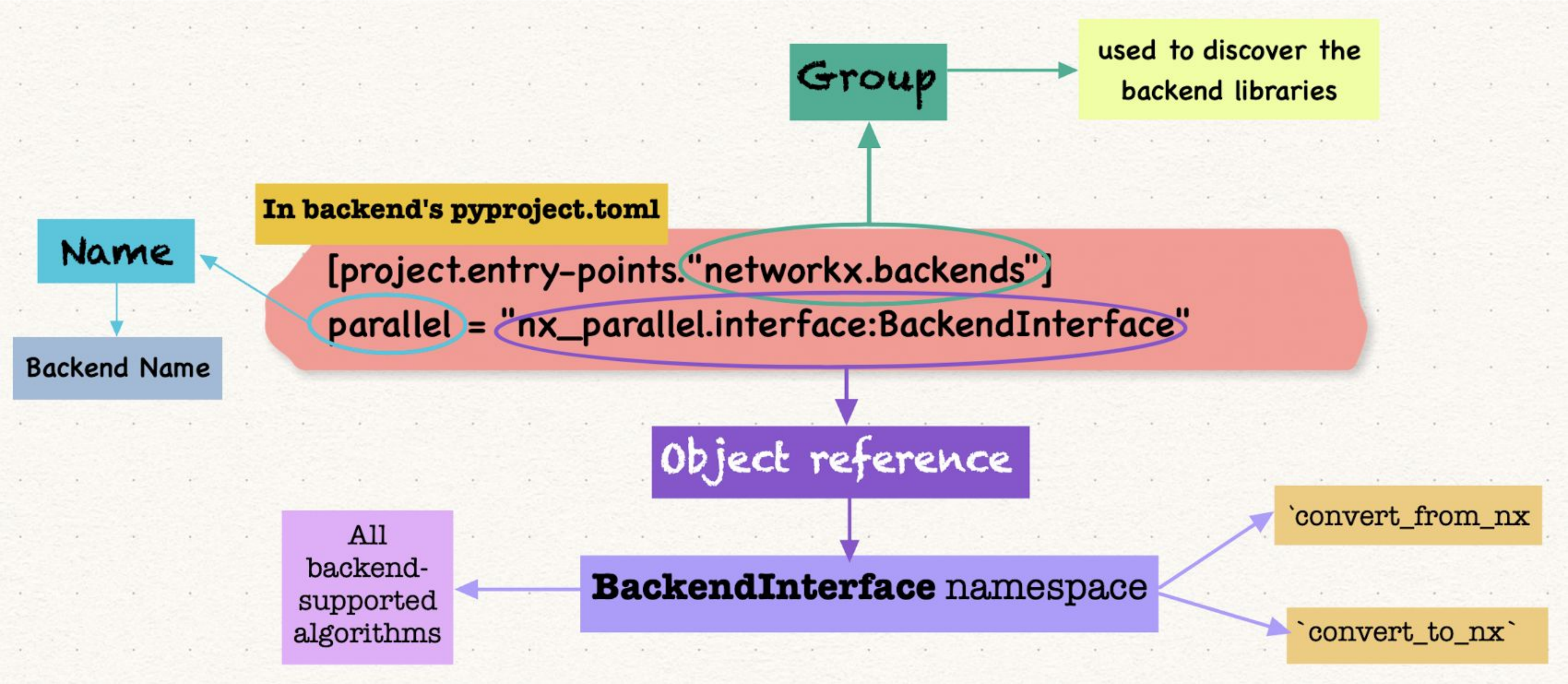
i.e. change the whole machinery inside the vacuum cleaner (main library's implementation)– and just keep the outside red UI-plastic-buttons frame (user-API).

Inside nx-parallel backend



ref. https://packaging.python.org/en/latest/specifications/entry-points/

Inside nx-parallel backend



ref. https://packaging.python.org/en/latest/specifications/entry-points/

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",
    ),
)
```

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:

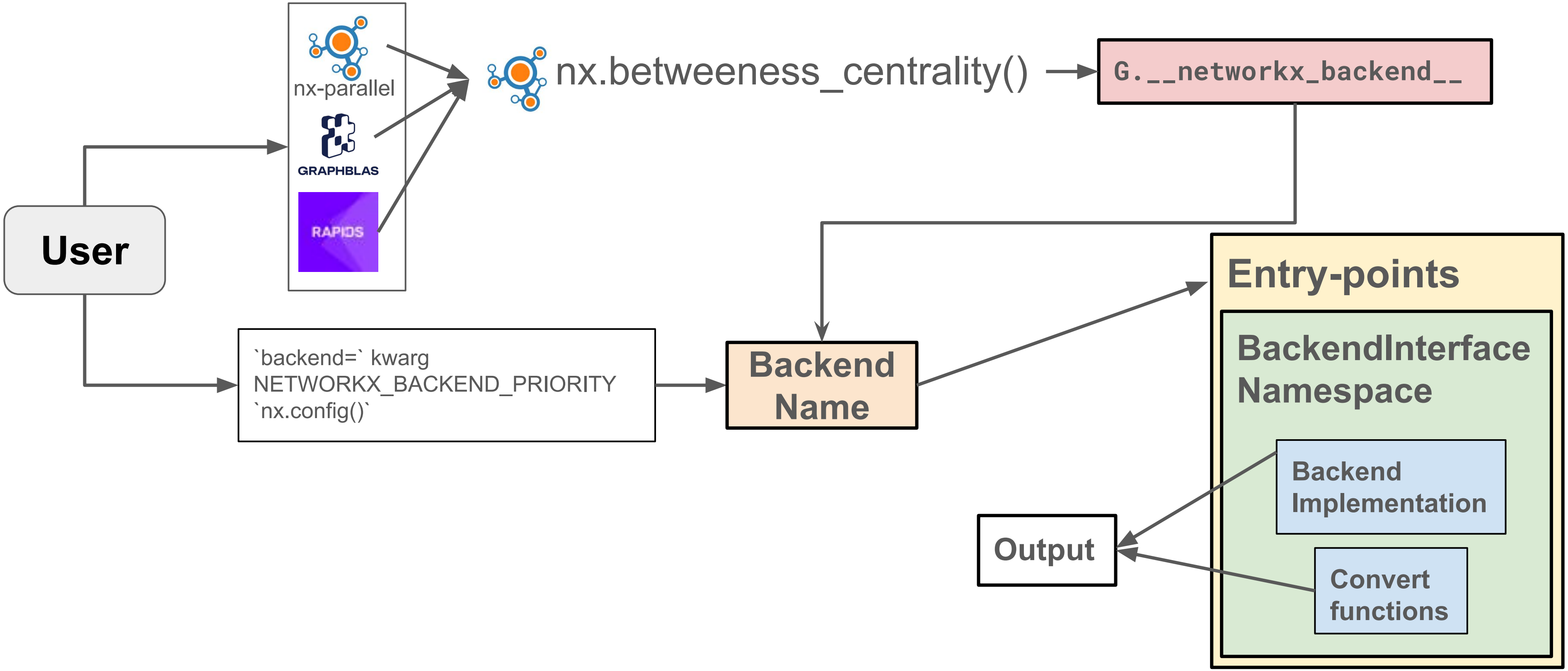
```
>>> nx.betweenness_centrality.backends
{'parallel'}
```

- Specialised backend priority for algorithms, generators,.. etc.
- Logging
- Fallback

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

NetworkX

(type-based and name-based)



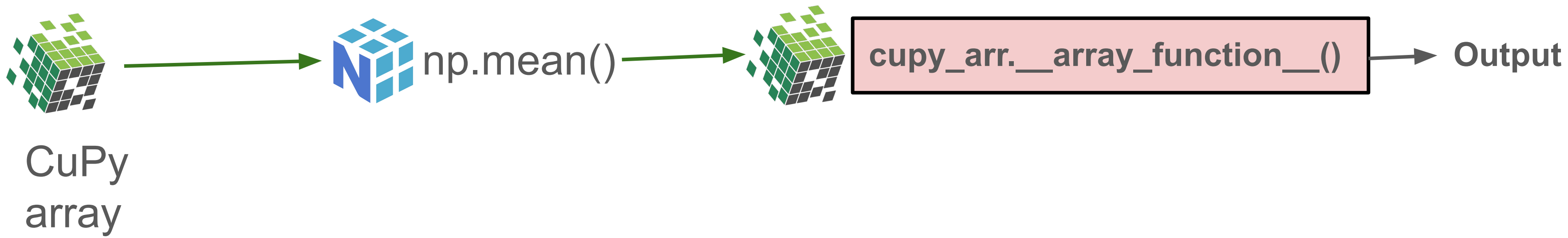
Dispatching in NumPy and for Arrays

Disclaimer: I’m not an expert in arrays or array dispatching. It is just a brief introduction!

NumPy Dispatching

`input_array.__array_function__()`

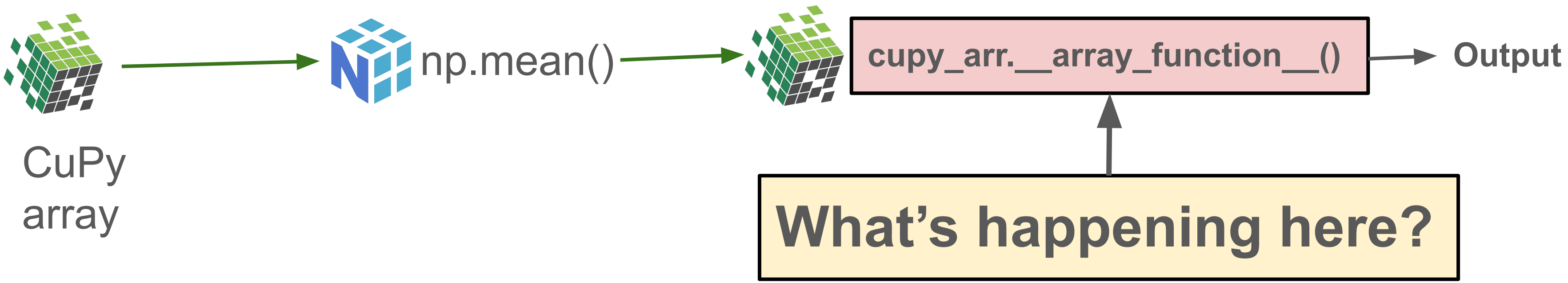
directly calls the apt. array library's implementation, corresponding to the function that's being called, if it exists.



NumPy Dispatching

`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):
          return NotImplemented

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

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

Some drawbacks…

No fallbacks!

Not a nice API: NumPy returning CuPy arrays

Other array libraries might have additional functionality on top of NumPy.

Array API Standards

Ecosystem of Array Libraries

Borrowed from Aaron Meurer’s SciPy 2023 talk: https://youtu.be/16rB-fosAWw?si=dzQ_VUGW7h25AG4j

Ecosystem of Array Libraries

Goal: Array agnostic functionality

Borrowed from Lucas Colley's EuroSciPy 2025 talk: https://lucascolley.github.io/talks/euroscipy-25-array-api/

Ecosystem of Array Libraries

Array (producer) libraries

Array consuming libraries

Borrowed from Aaron Meurer's SciPy 2023 talk: https://youtu.be/16rB-fosAWw?si=dzQ_VUGW7h25AG4j

Array API Standards

- Standard API guidelines for array producer libraries, for e.g., function names, argument names, etc.
- `__array_namespace__` for array consuming libraries to easily use all the supported functions
- But, it is just a standardisation guidelines – adopting it doesn't guarantee interoperability with all kinds of array libraries

Ref. https://data-apis.org/array-api/2022.12/

How to adopt it?

Array consuming library

```
import numpy as np

def covariance(x, y):
    mean_x, mean_y = np.mean(x), np.mean(y)
    cov = np.mean((x - mean_x) * (y - mean_y))
    return cov
```

How to adopt it?

Array consuming library

```
def covariance(x, y):
    xp = x.__array_namespace__()

    mean_x, mean_y = xp.mean(x), xp.mean(y)
    cov = xp.mean((x - mean_x) * (y - mean_y))
    return cov
```

How to adopt it?

Array consuming library

```
def covariance(x, y):
    xp = array_api_compat.array_namespace(x, y)

    mean_x, mean_y = xp.mean(x), xp.mean(y)
    cov = xp.mean((x - mean_x) * (y - mean_y))
    return cov
```

**array-api-compat*

Ref. https://github.com/data-apis/array-api-compat

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

Some more resources:

- 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
  - https://scikit-image.org/docs/dev/development/dispatching.html
  - https://github.com/rapidsai/cucim/issues/829
- Scikit-learn
  - https://scikit-learn.org/stable/modules/array_api.html
  - 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
- https://github.com/scikit-hep/ragged?tab=readme-ov-file
- https://github.com/Saransh-cpp/SwissPythonSummit25-GLASS-array-api