Understanding Dispatching Approaches in the Scientific Python Ecosystem

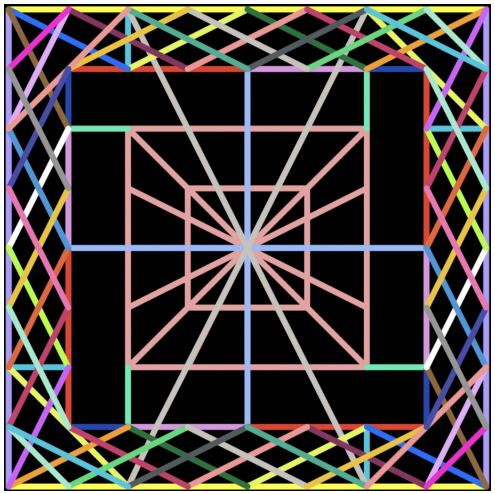
- By Sebastian Berg and Aditi Juneja

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 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

Contents

- Introduction to dispatching - 5 mins
- Dispatching in NumPy and for arrays - 6-7 mins
- Dispatching in networkx and for graphs - 10-12 mins
  - Scikit-image dispatching
- Overview - 4-5 mins
  - Spatch
  - Summary

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

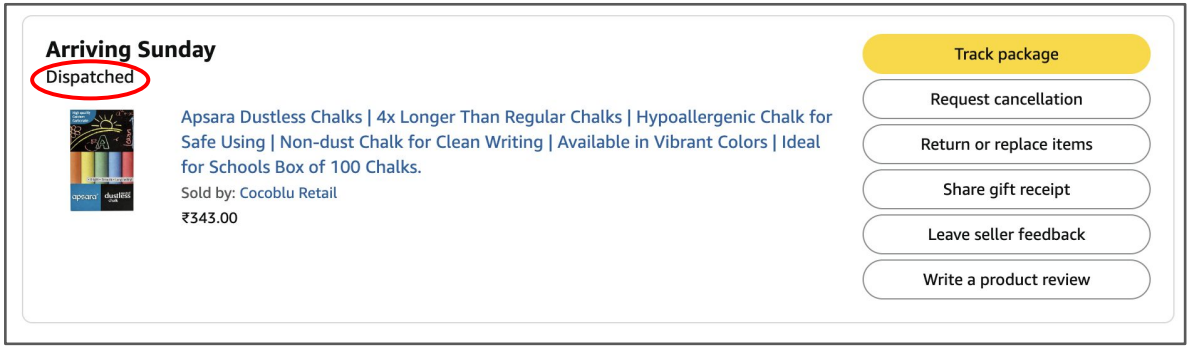
Return or replace items

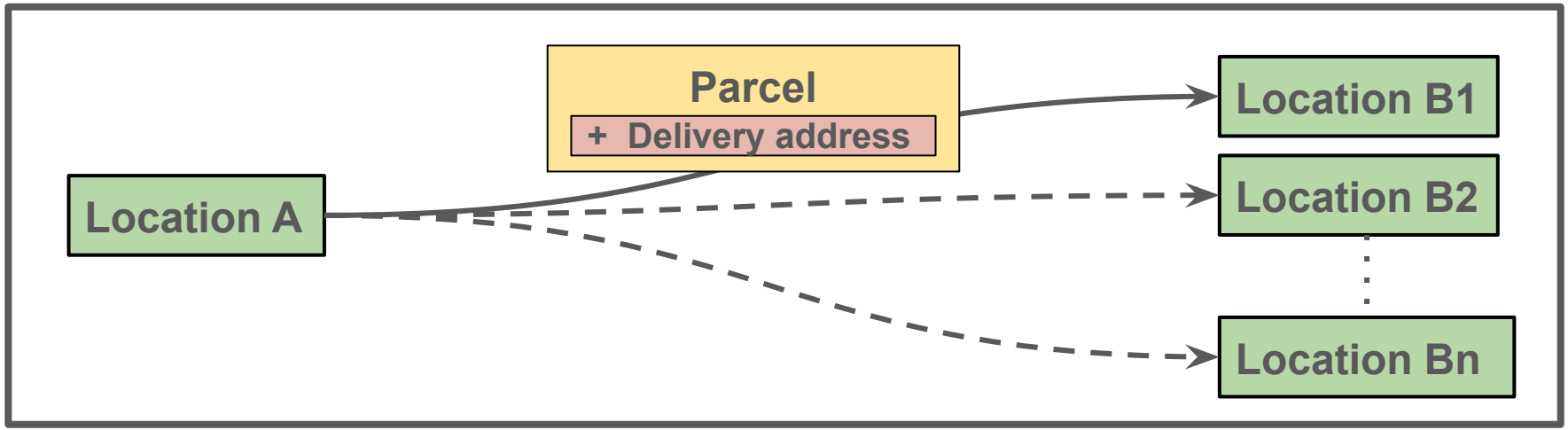
Share gift receipt

Leave seller feedback

Write a product review

Dispatching– in general





Dispatching – in programming

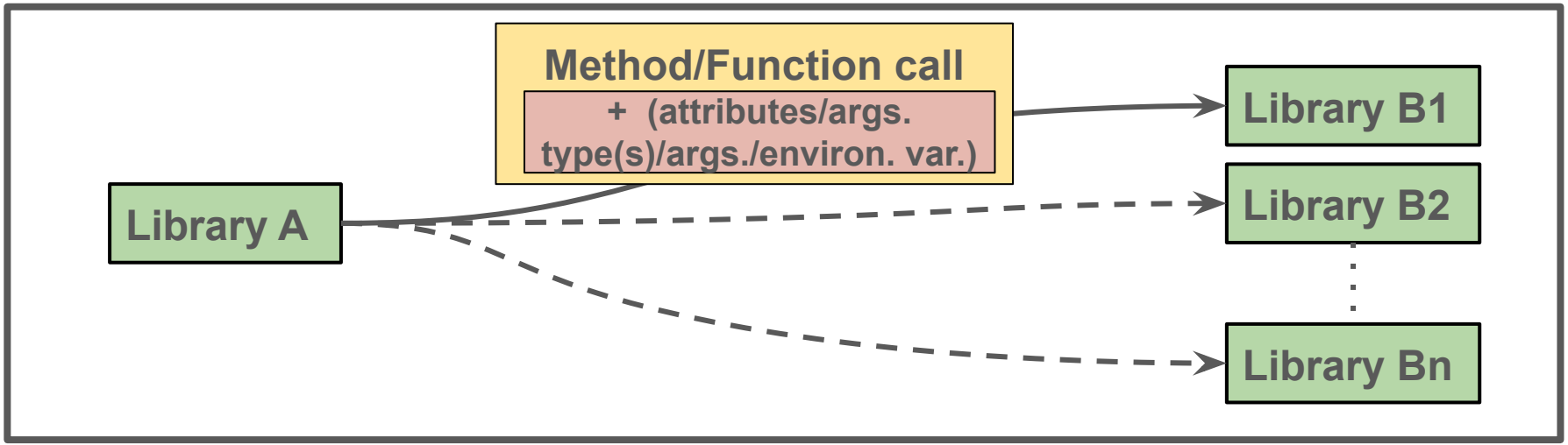
“In computer science, **dynamic dispatch** is the process of selecting which implementation of a polymorphic operation (method or function) to call at **run time.**”

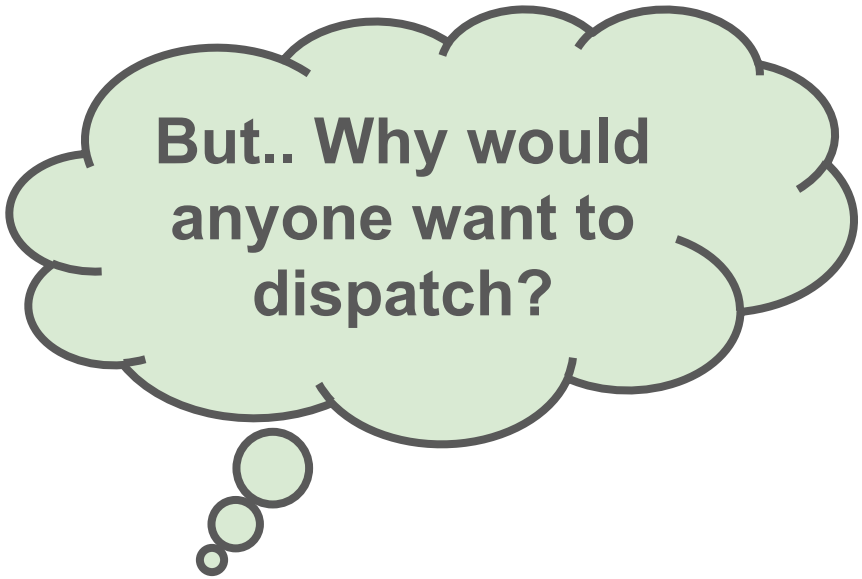
Ref. https://en.wikipedia.org/wiki/Dynamic_dispatch

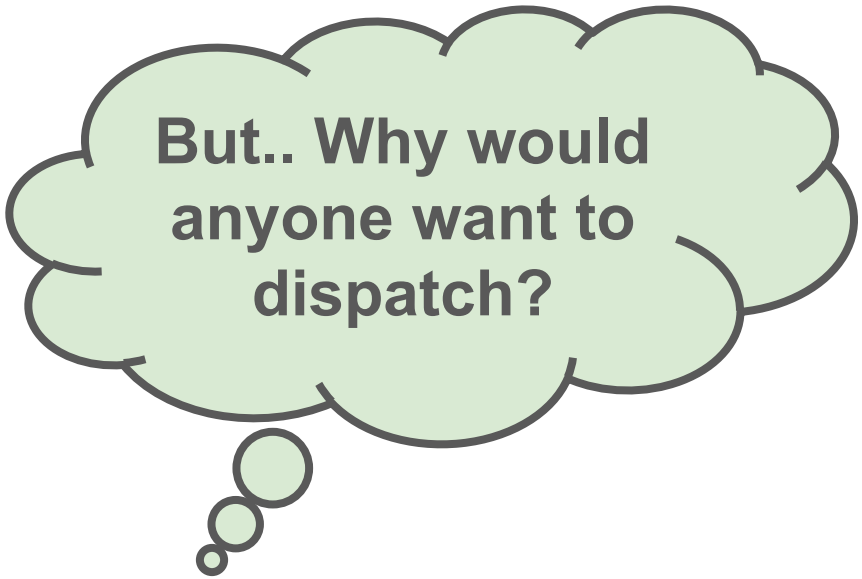
Dispatching – in programming

“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




But.. Why would anyone want to dispatch?

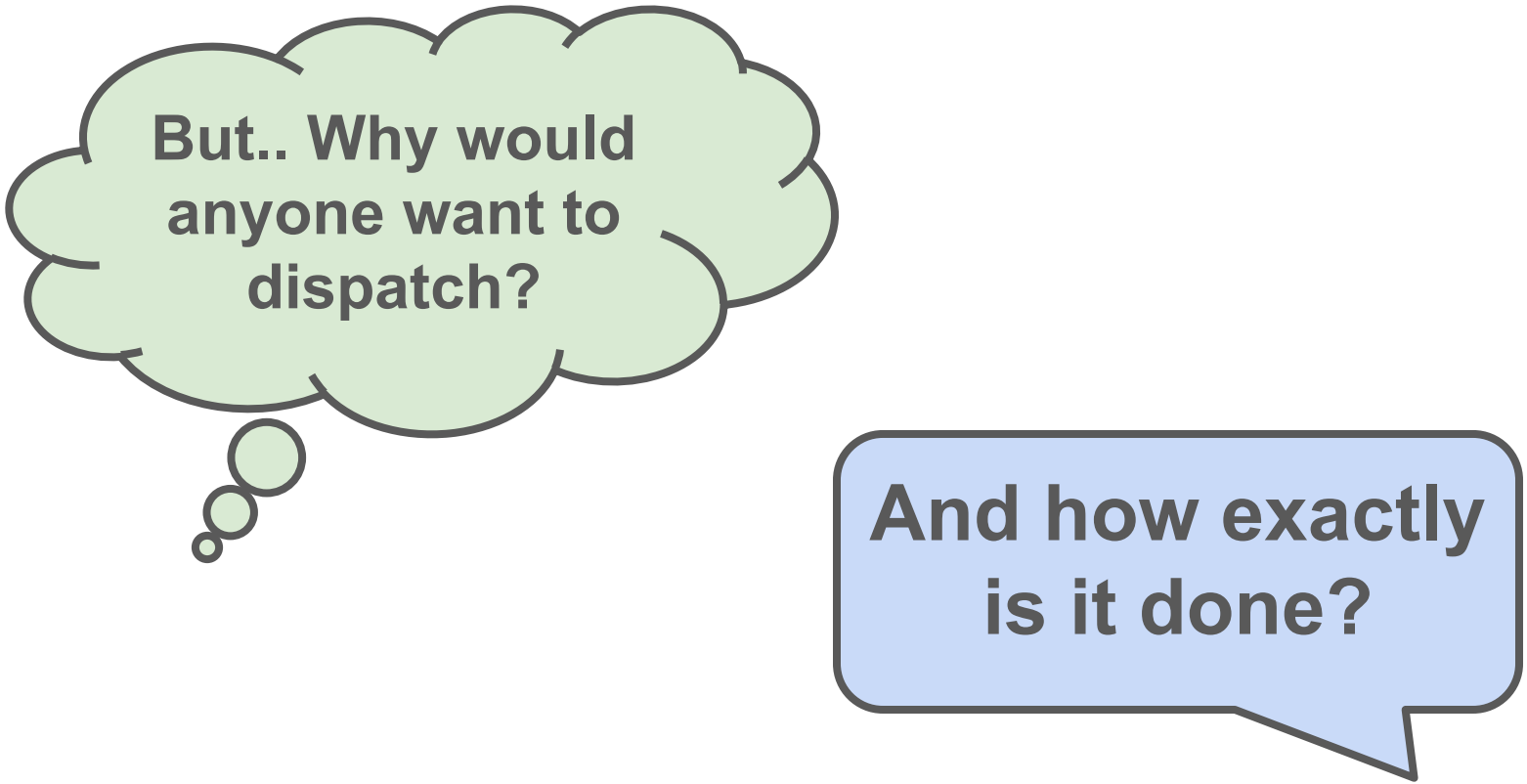


Improve Performance

Type Compatibility

Simplify User API

Enable different workflows


But.. Why would anyone want to dispatch?
And how exactly is it done?

Example

What does dispatching mean here?

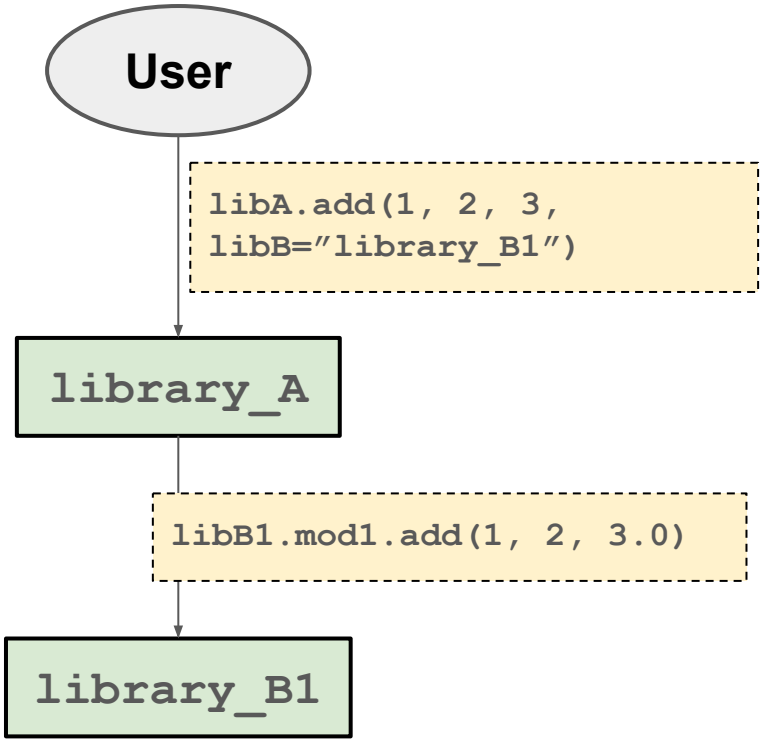
Inside `library_A`

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



Steps involved

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

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

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?

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

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

Can we do something more here?

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

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
```

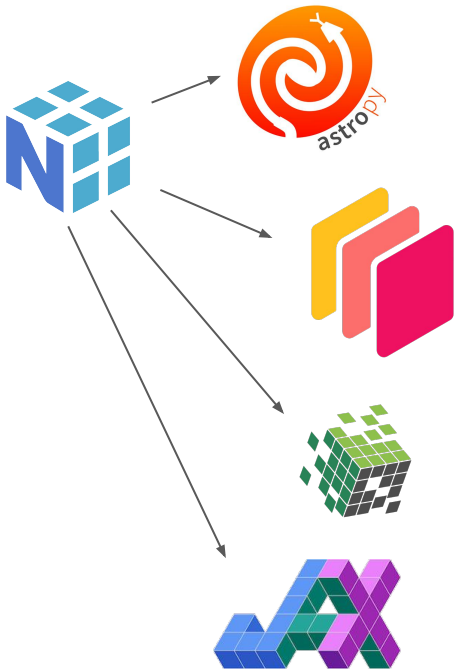
How is dispatching done in real projects?

Dispatching in NumPy and for Arrays

Dispatching in NumPy

- Started around ~2013:
  - `__array_ufunc__` (NEP 13)
    - subset (“universal functions”)
  - `__array_function__` (NEP 18 + 35)
    - almost all other functions.

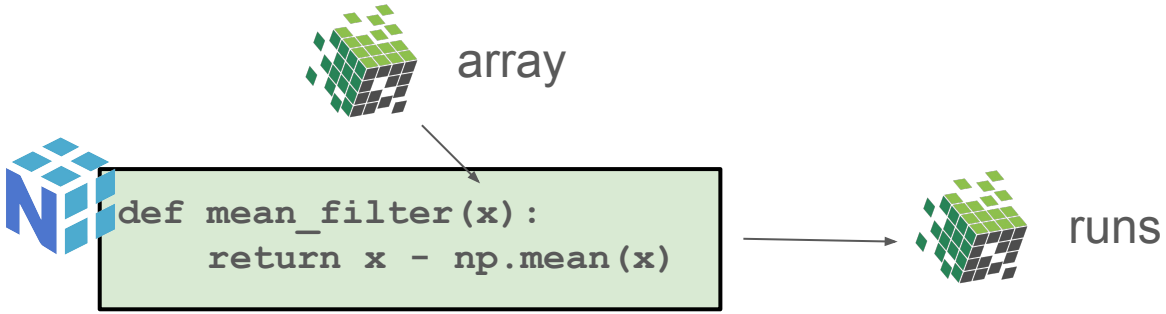
- Use-cases/users:
  - SciPy sparse
  - `astropy.units` (“NumPy array of meters”)
  - Dask array (distributed)
  - CuPy (GPU), JAX, cupynumeric…



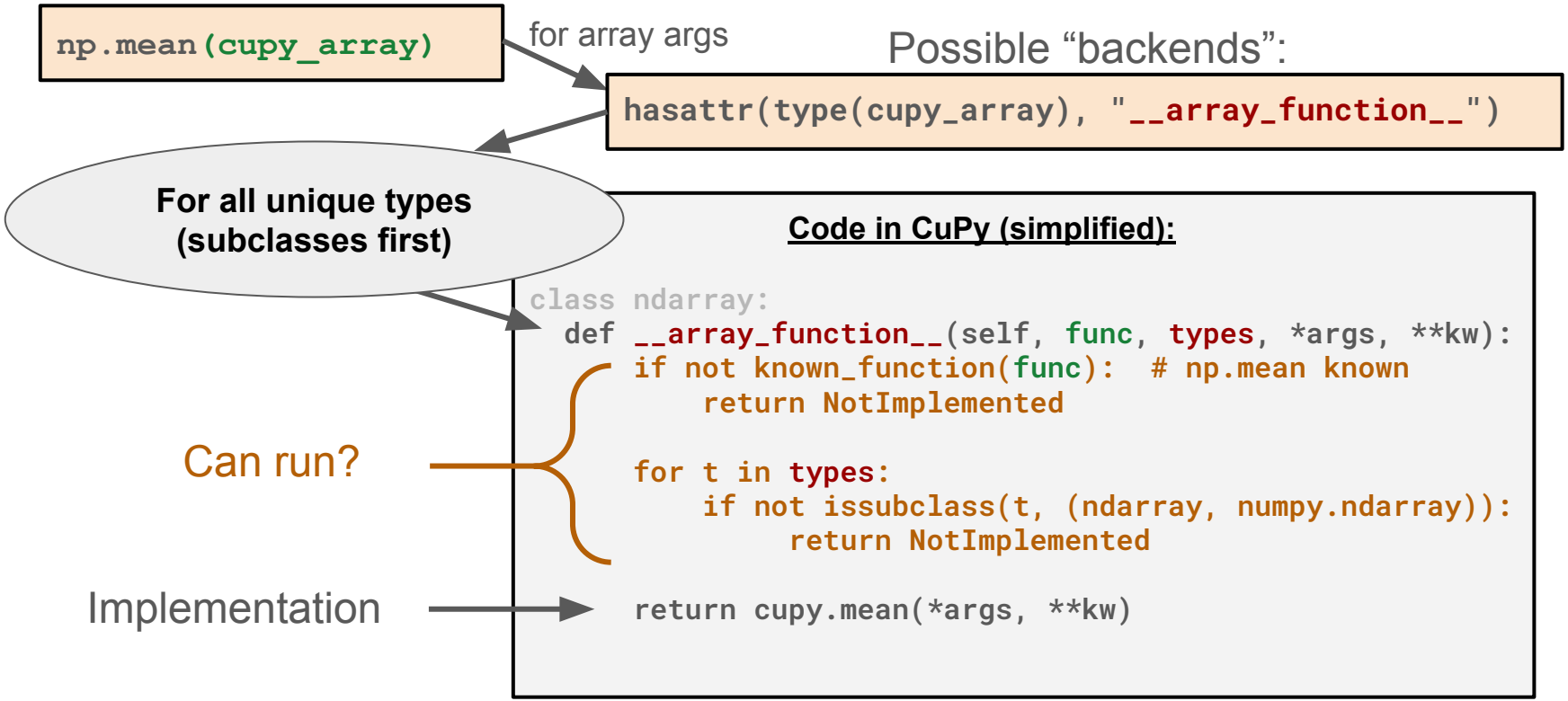
Dispatching in NumPy – Usage

- Astropy quantities:
  - User use: `np.mean` not `astropy.unit.mean`
- Cupy, etc.
  - `import cupy as np` possible
  - Dispatching allows function re-use!

Example:



Dispatching in NumPy – Implementation



Dispatching in NumPy – Summary

- Allows interoperating with NumPy
- `__special_method__()` much like Python binary operators
- Adopted by many NumPy like objects
- Good approach if you own type

But:

- E.g. `torch` did not adopt (backwards compatibility concerns).
- Libraries (SciPy, etc.) do not use it (to allow cupy, JAX, arrays…)

Beyond NumPy – Array API

- NEP 37 (not implemented) evolved into **Array API**
  - `xp = array_api_compat.array_namespace(*arrays)`
    - `xp` is `numpy`, `cupy`, `torch`, `jax.numpy`…
  - `xp` *explicit* state: dispatch only once
  - **Being adopted** into **SciPy, sklearn**, …

- For dataframes: **narwhals**

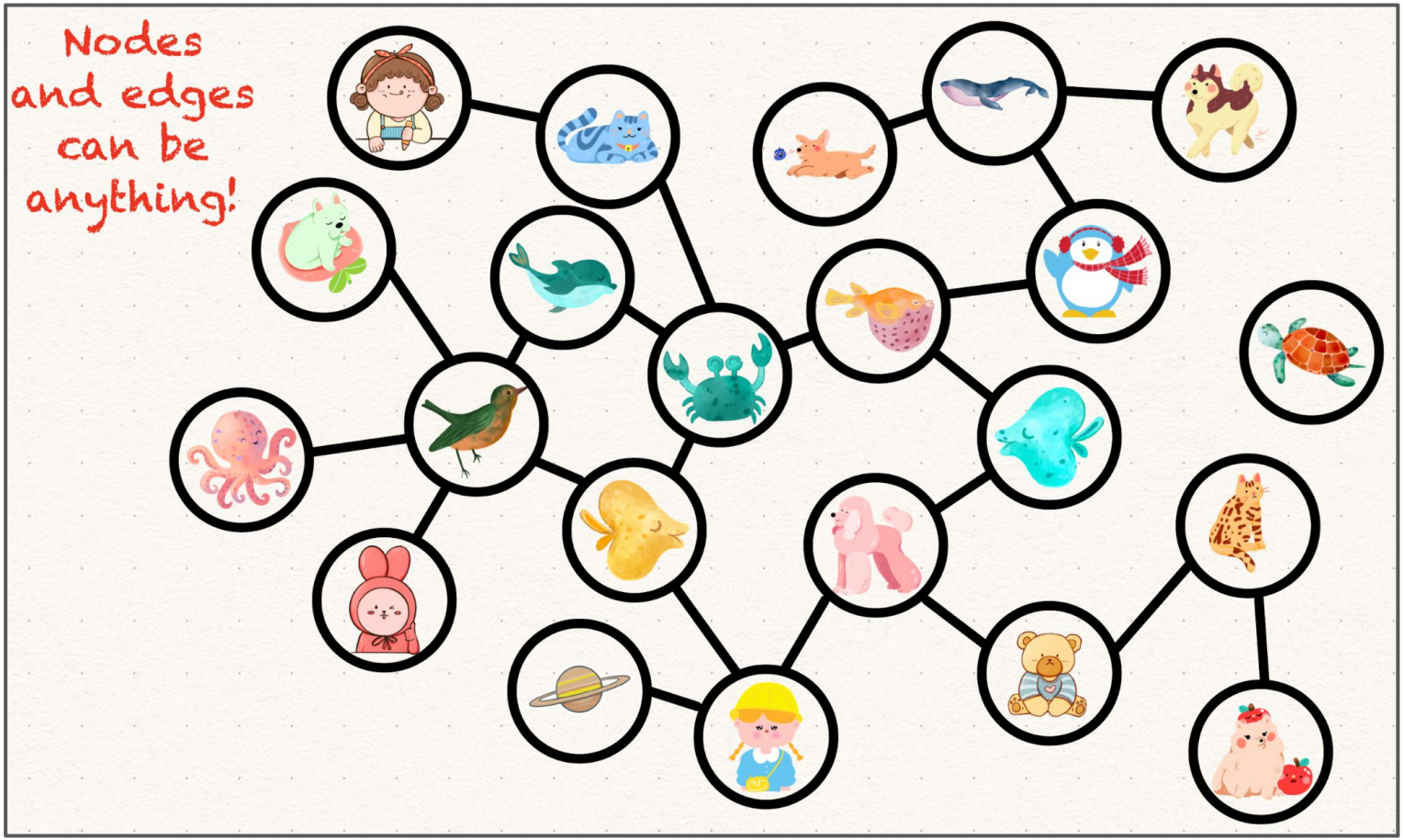
Clear ***library* focus**. End-users: `import cupy as np` is more practical.
(but every end-user also writes their small functions/libraries)

Dispatching in NetworkX for Graphs

What is networkx?

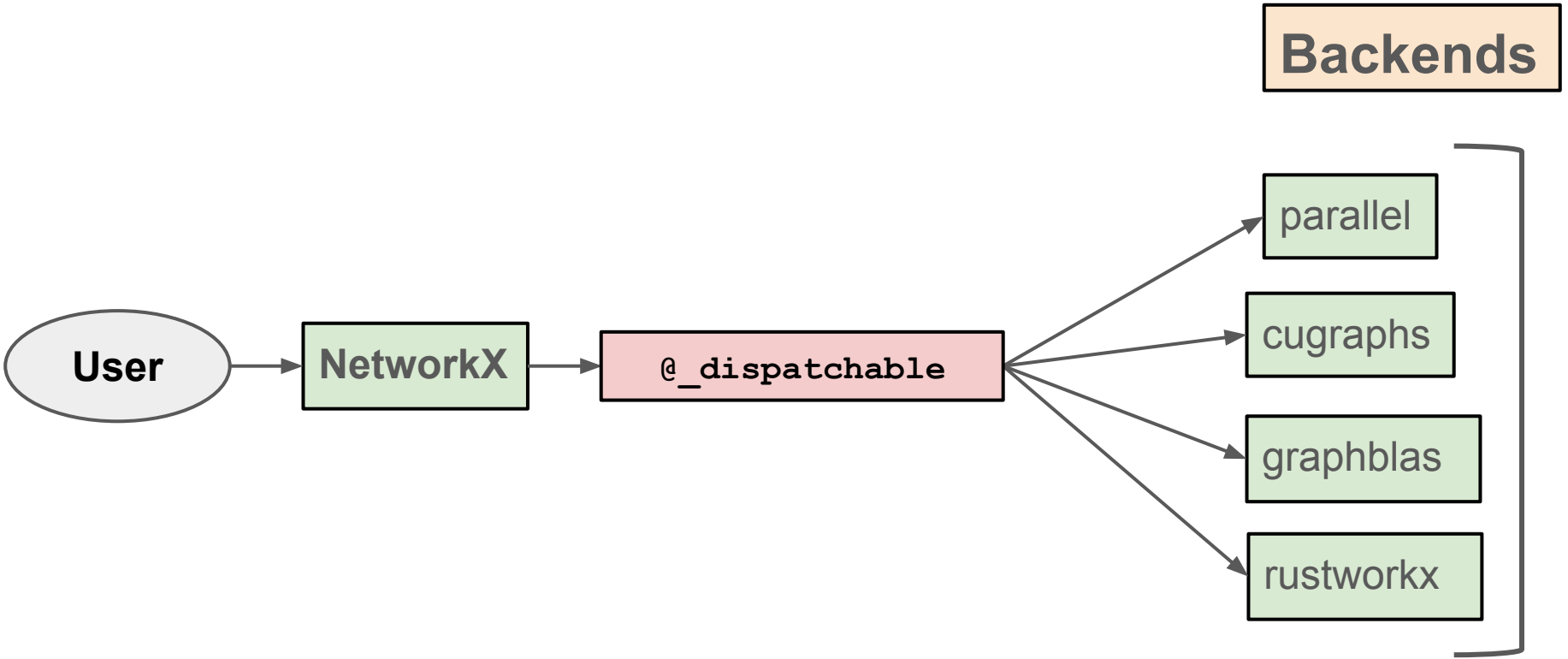
NetworkX is a graph analysis library

What are graphs/networks?



Need for dispatching in NetworkX

- **Speed**:
  - NetworkX is pure python, therefore slow;
  - but simple to understand, use, maintain, and contribute to
- **Standardisation** - Simplifying user API


Backends
User
NetworkX
@_dispatchable
parallel
cugraphs
graphblas
rustworkx

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

Step 1: Get the backend package name from the user

Step 1: Get the backend package name from the user

Speed-up/usage **demo** with cugraphs, graphblas and/or parallel backends

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
```

Inside `networkx`

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

nx-parallel

nx-cugraph

Python-graphblas

```
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
```

One-to-one mapping of backends and types

Step 2: Getting the function from the backend package

Step 2: Getting the function from the backend package

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.

What are entry-points?

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

Main library

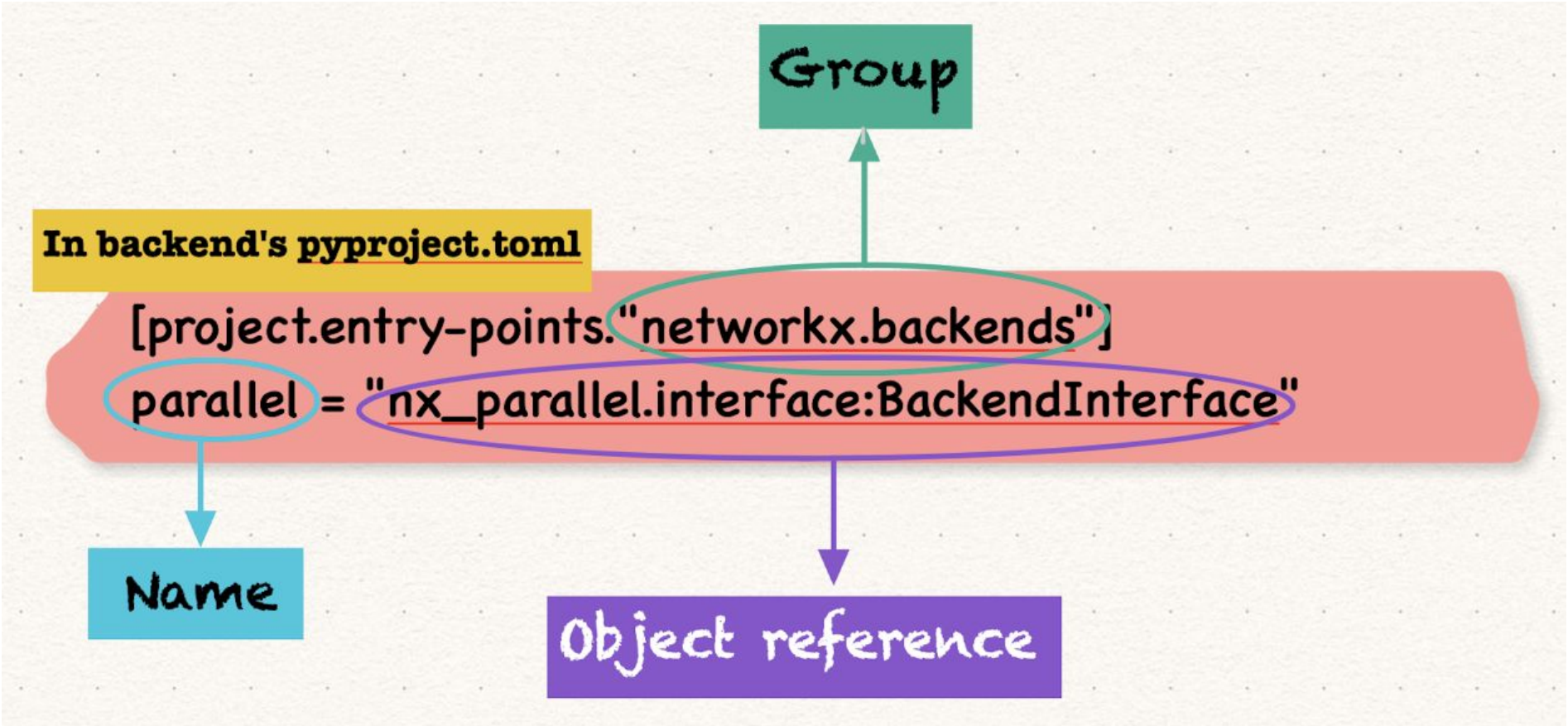
Entry point

Extension (or plug-in) libraries

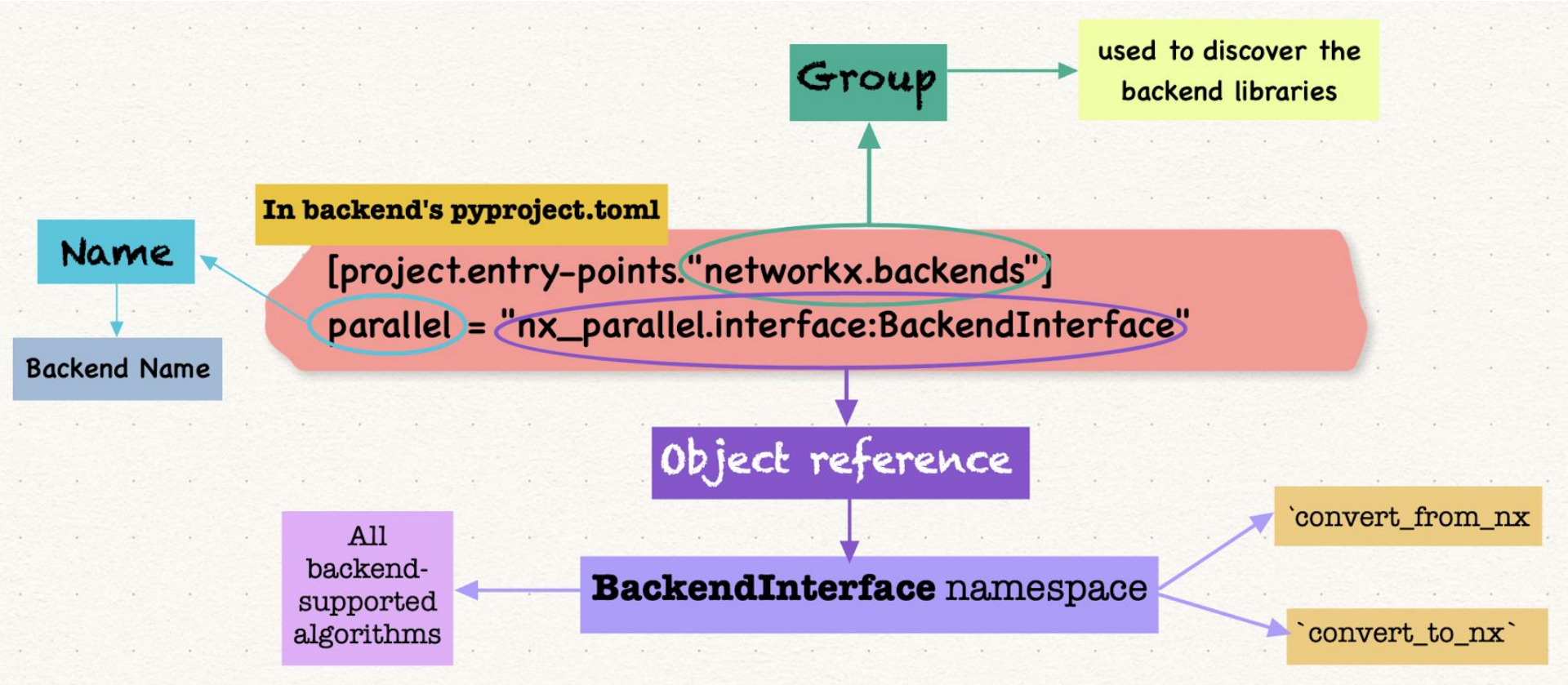
Dispatching with entry-points is exploiting entry-points to the max–

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).

Defining an entry-point in the backend library



Defining an entry-point in the backend library



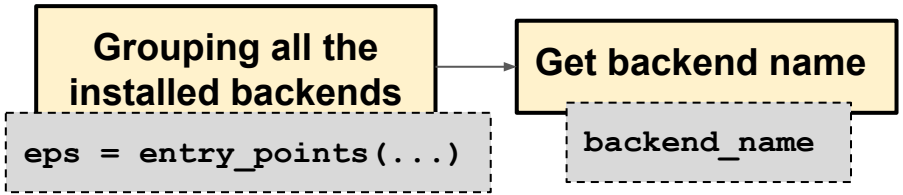
ref. https://packaging.python.org/en/latest/specifications/entry-points/

How does entry-point based dispatching work?

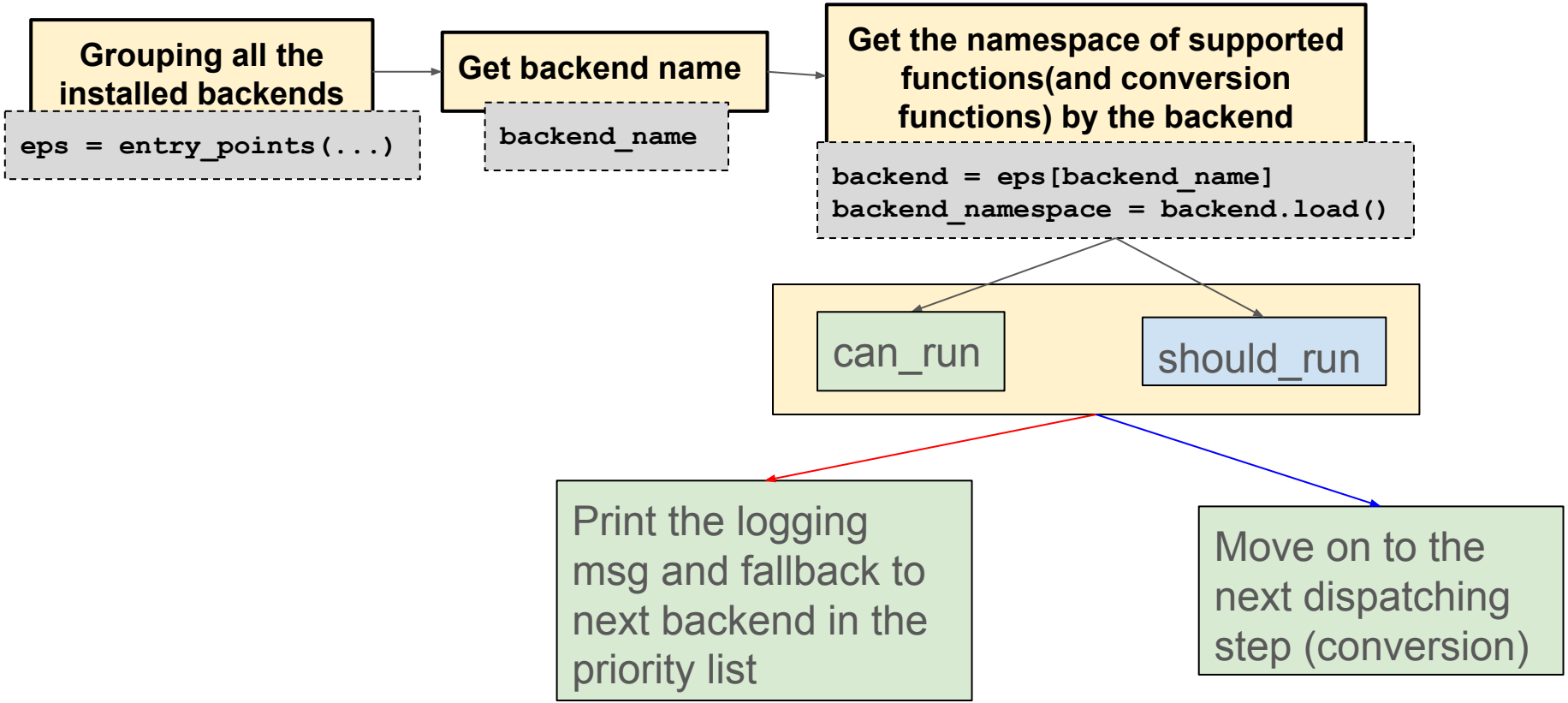
Grouping all the installed backends

```
eps = entry_points(...)
```

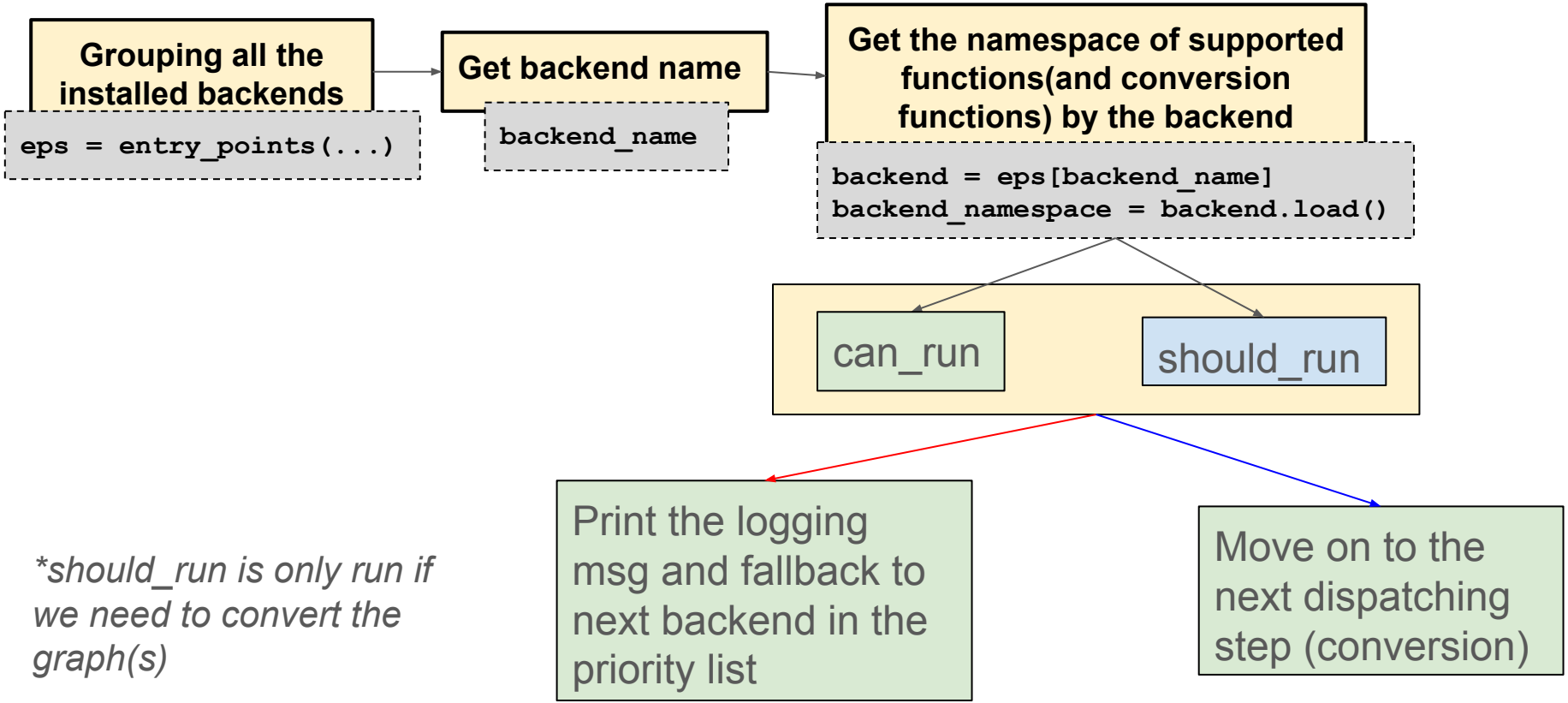
How does entry-point based dispatching work?



How does entry-point based dispatching work?

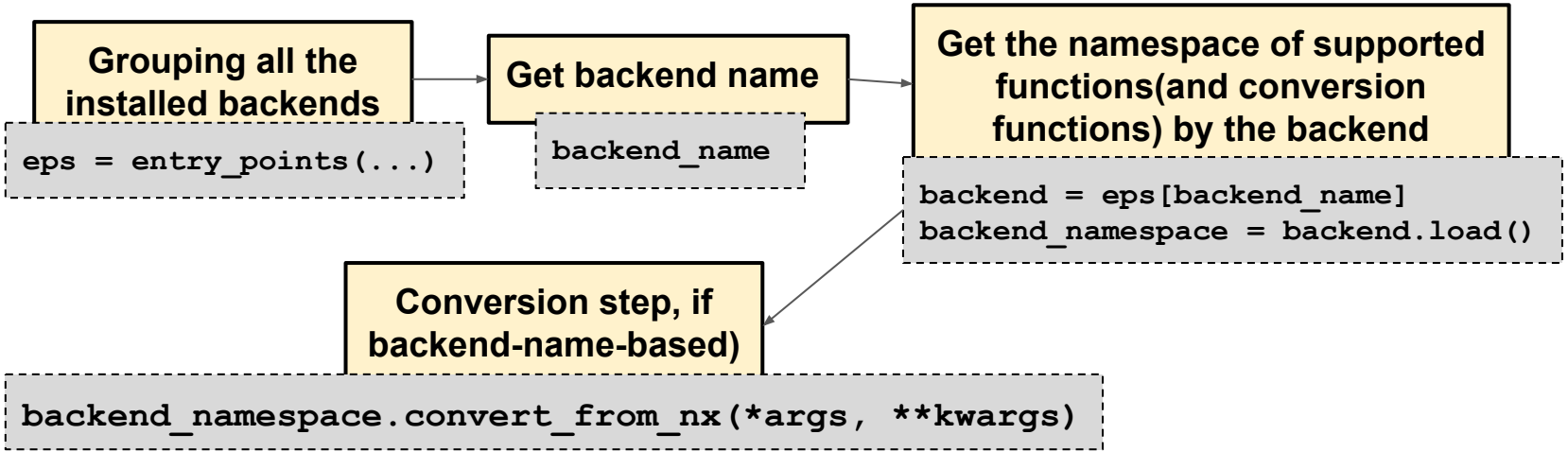


How does entry-point based dispatching work?

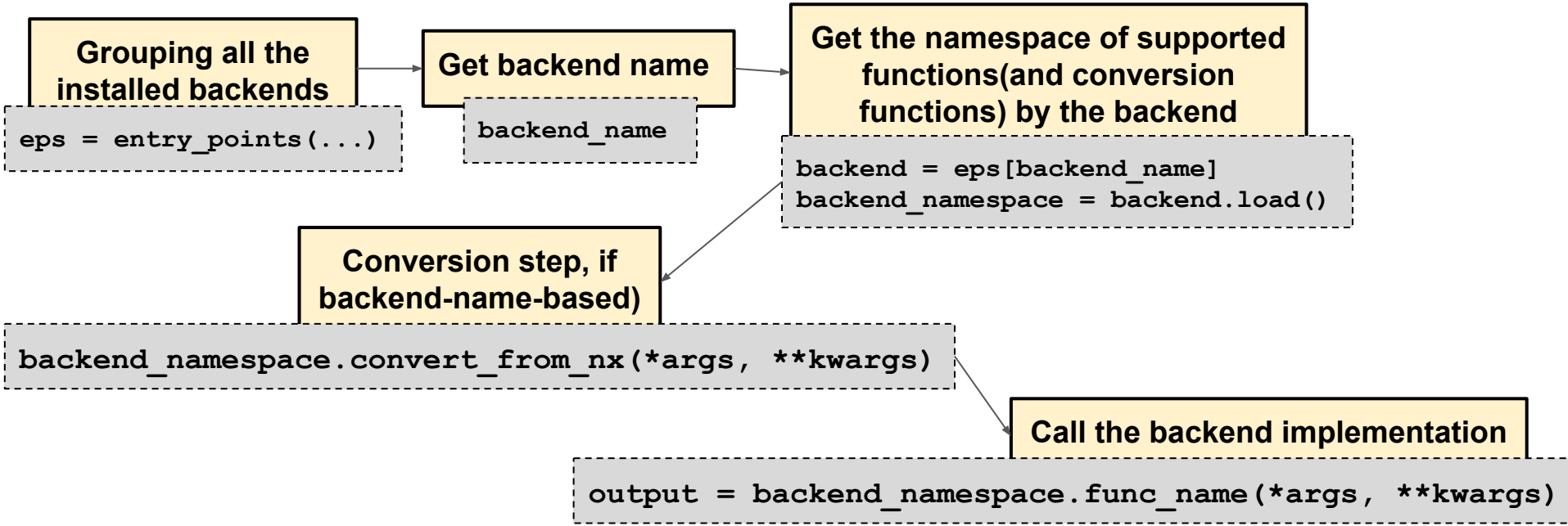


**should_run is only run if we need to convert the graph(s)*

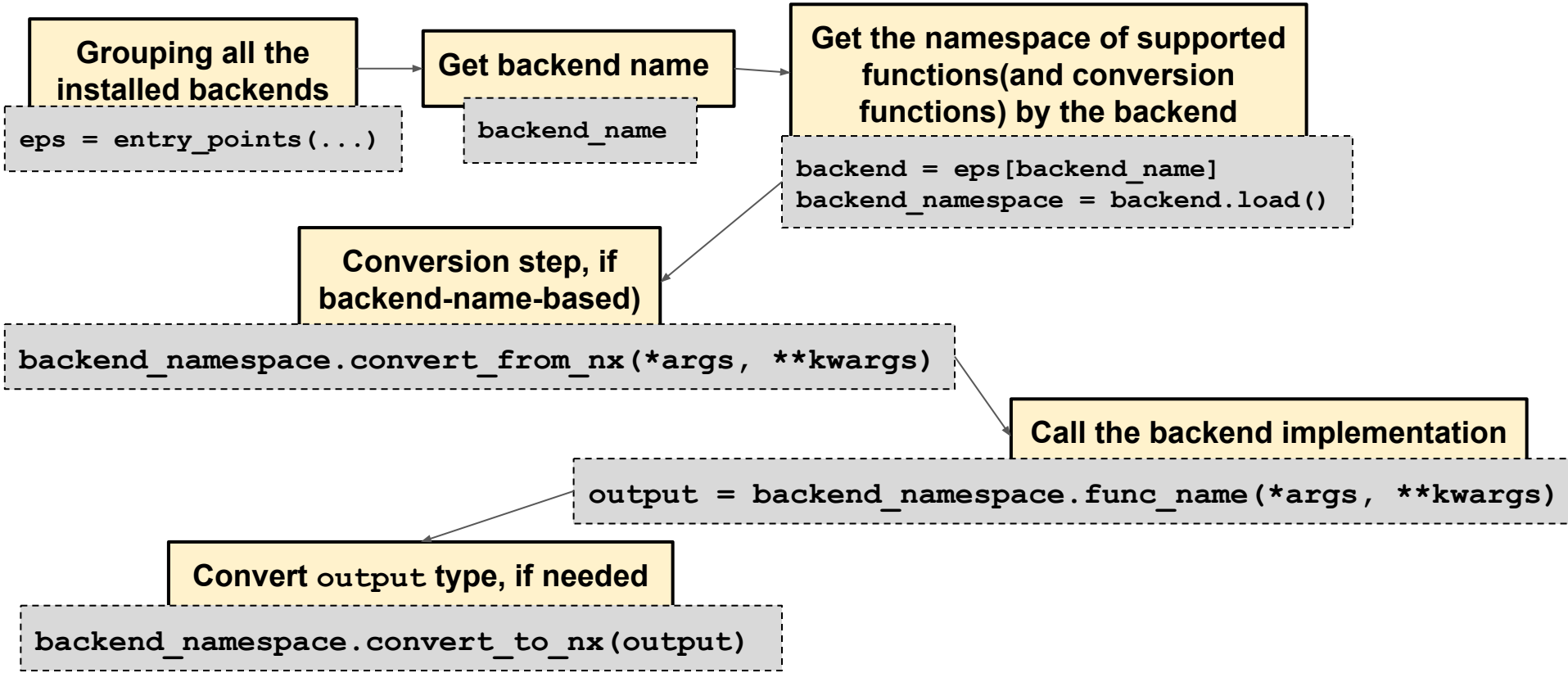
How does entry-point based dispatching work?



How does entry-point based dispatching work?



How does entry-point based dispatching work?



Other features of NetworkX dispatching (if time permits)

- Backend testing
- Docs : second entry point
- By looking at the `.backends` attribute, you can get the set of all currently 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 scikit-image

- Image object (`numpy.ndarray`) → not owned by scikit-image
  - cannot add backend name to the array object (like __networkx_backend__ in NetworkX)

Dispatching in scikit-image

- Image object (`numpy.ndarray`) → not owned by scikit-image
  - cannot add backend name to the array object (like __networkx_backend__ in NetworkX)

- **Goal**: want it to look like type based dispatching (with a component of backend selection) – internally entry-point based.

Dispatching in scikit-image

- Image object (`numpy.ndarray`) → not owned by scikit-image
  - cannot add backend name to the array object (like __networkx_backend__ in NetworkX)

- **Goal**: want it to look like type based dispatching (with a component of backend selection) – internally entry-point based.

- Why not use Array API standards? – Hard for Cython-heavy array consuming libraries to adopt these standards– therefore using entry-points for dispatching.

Dispatching in scikit-image

- Image object (`numpy.ndarray`) → not owned by scikit-image
  - cannot add backend name to the array object (like __networkx_backend__ in NetworkX)

- **Goal**: want it to look like type based dispatching (with a component of backend selection) – internally entry-point based.

- Why not use Array API standards? – Hard for Cython-heavy array consuming libraries to adopt these standards– therefore using entry-points for dispatching.

- **Current state**: No array conversions in dispatching code! – backend developers and users need to take care of the types.

Dispatching in scikit-image

- Image object (`numpy.ndarray`) → not owned by scikit-image
  - cannot add backend name to the array object (like __networkx_backend__ in NetworkX)

- **Goal**: want it to look like type based dispatching (with a component of backend selection) – internally entry-point based.

- Why not use Array API standards? – Hard for Cython-heavy array consuming libraries to adopt these standards– therefore using entry-points for dispatching.

- **Current state**: No array conversions in dispatching code! – backend developers and users need to take care of the types.

Possible Solution : spatch

Spatch – https://github.com/scientific-python/spatch

Spatch – https://github.com/scientific-python/spatch

Create library or blueprint for the type of dispatching scikit-image needs!
(based on NetworkX/skimage lessons.)

Goal:

- Help implement good dispatching and backends
- Develope a recognizable API for users

Principles:

- Entry-points (docs + introspection)
- Fast *type dispatching* (caching)
- *Named backend selection* support

Spatch – Key points

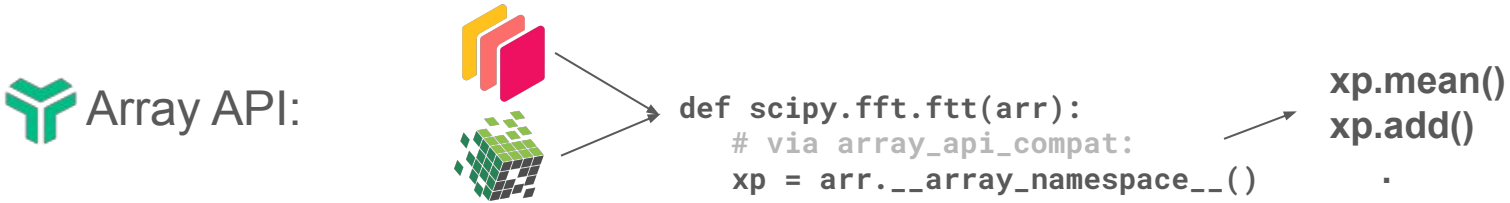
- Library:
  - Mostly add decorator.
- Backends – entrypoint:
  - Which types: `"~cupy:ndarray"` (cupy arrays and subclasses)
  - Auto-generated mapping:

```
functions = {
  "skimage.filters:gaussian": {
   "impl": "cucim.skimage.filters:gaussian"}
```

- User API:
  - (implicit) type dispatching
  - `with backend_opts(prioritize="opencv")`
  - …

Summary

Summary – Type based



For SciPy: export SCIPY_ARRAY_API=1

Summary – Type + Selection

nx.betweeness_centrality(G)

Backend name from graph object:

- `G.__networkx_backend__`

Backend name directly via user:

- **NETWORKX_BACKEND_PRIORITY**
- `nx.config()`
- `backend=`

entry-points

nx-parallel

nx-cugraph

with spatch

Via object:

- `"cupy:ndarray"`, …

User:

- environment / `with config`
- (future)

entry-points

Overview

	NumPy	Array API	NetworkX	spatch
Type dispatching	__array_function__()	__array_namespace__()	__networkx_backend__	"module:name"
Selection (speed)	no	no	backend=”name” env / with config()	with backend_opts() / env
entry-point	no	no	yes	yes
conversion	by backend	no	If needed (by convert functions)	by backend
Focus	end-user	Array consuming library	End-user (speed)	Help devs dispatch for end-users

Overview

	NumPy	Array API	NetworkX	spatch
Type dispatching	__array_function__()	__array_namespace__()	__networkx_backend__	"module:name"
Selection (speed)	no	no	backend=”name” env / with config()	with backend_opts() / env
entry-point	no	no	yes	yes
conversion	by backend	no	If needed (by convert functions)	by backend
Focus	end-user	Array consuming library	End-user (speed)	Help devs dispatch for end-users

Thank you!

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/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

FIN.