



Join at
slido.com
#4010 307

Which of these do you think represents an API?

\$ curl https://api.nasa.gov/planetary/apod?api_key=DEMO_KEY

☐ 0%

>>> arr = np.array([[1, 2, 3], [4, 5, 6]])

☐ 0%

None of them

☐ 0%

Both of them

☐ 0%

I have no idea what "API" even stands for

☐ 0%



Multiple choice ▾

93 votes



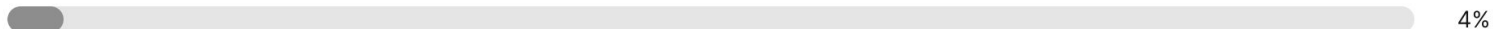
Poll settings

Which of these do you think represents an API?

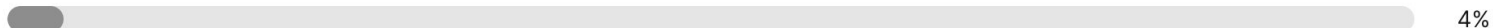
\$ curl https://api.nasa.gov/planetary/apod?api_key=DEMO_KEY



>>> arr = np.array([[1, 2, 3], [4, 5, 6]])



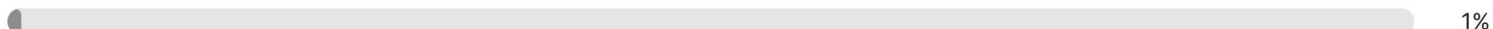
None of them



Both of them



I have no idea what "API" even stands for



+ Add option



Join at
slido.com
#4010 307

Which of these do you think represents an API?

\$ curl https://api.nasa.gov/planetary/apod?api_key=DEMO_KEY

☐ 0%

Web API

>>> arr = np.array([[1, 2, 3], [4, 5, 6]])

☐ 0%

Python Library API

None of them

☐ 0%

Both of them

☐ 0%

I have no idea what "API" even stands for

☐ 0%

In the context of this talk....

API == Python library API == Library == Package

Preferred term → Library

Understanding API dispatching



Slides and code :

https://github.com/Schefflera-Arboricola/pyconf_hyderabad_2025

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

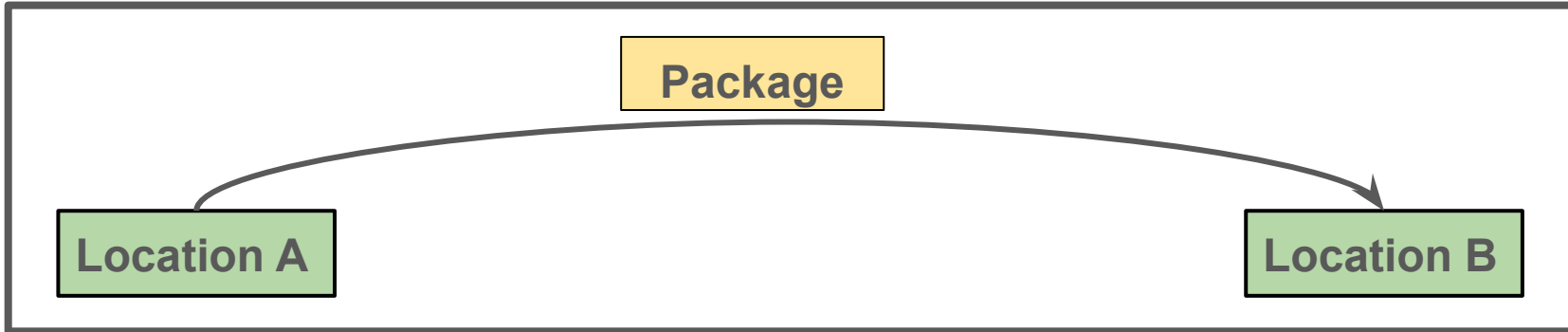
Return or replace items

Share gift receipt

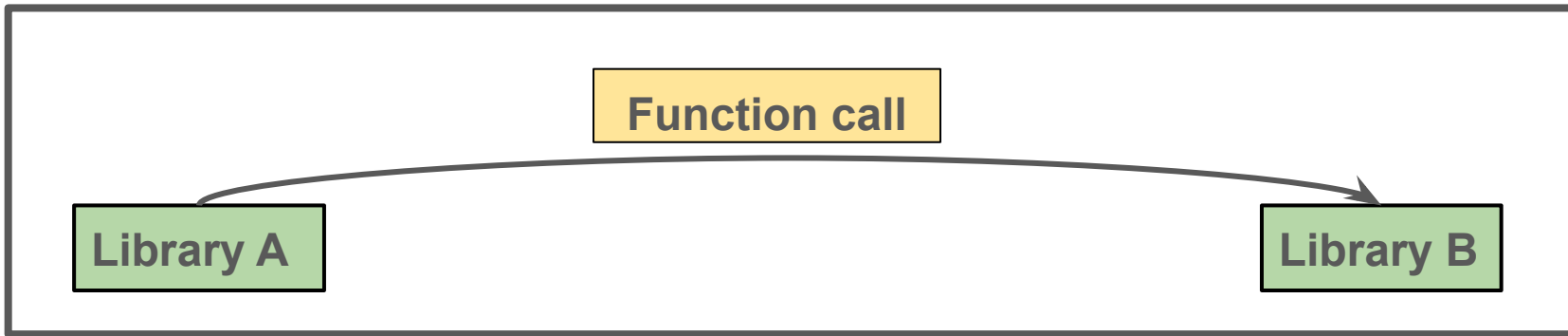
Leave seller feedback

Write a product review

Package dispatching



API dispatching



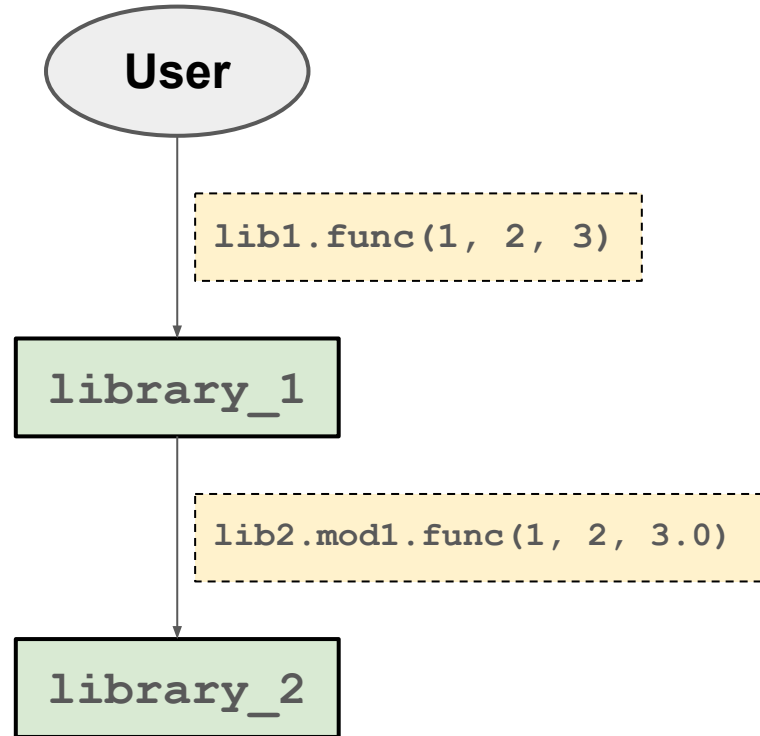
**But... what does it *really* mean to
dispatch a call from one library
to another?**

Dispatching is not Calling!

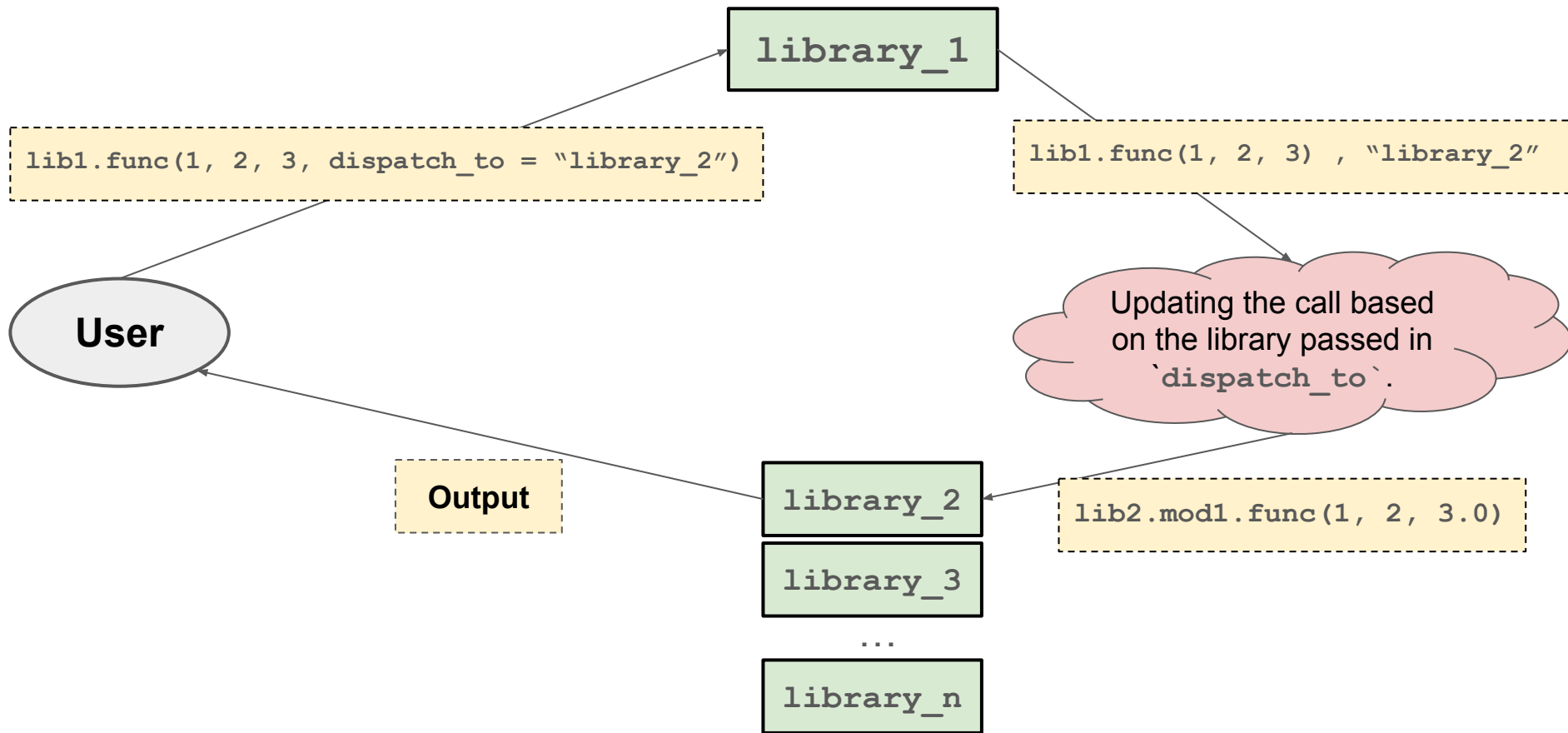
Calling

Inside `library_1`

```
def func(x, y, z, dispatch_to):  
    if dispatch_to == "library_2":  
        import library_2 as lib2  
        z = float(z)  
        return lib2.mod1.func(x, y, z)  
    elif dispatch_to == "...":  
        ...  
        ...  
    else:  
        return x + y + z
```



Dispatching



Dispatching

Inside ``library_1``

```
@dispatch  
def func(x, y, z):  
    return x + y + z
```

...


...

Updating the call based
on the library passed in
``dispatch_to``.

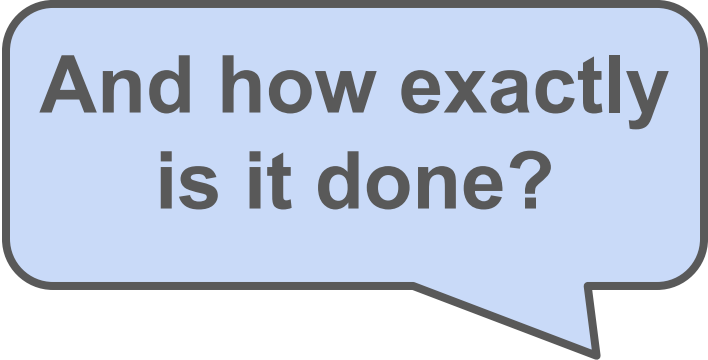
Calling is a part of this whole
dispatching process.

Dispatching is a way to call a function in a given library, without hard-coding it all...

Wait... let's take a step back



**Why would I
want to dispatch
a call?**



**And how exactly
is it done?**

**Let's understand API
dispatching with the
NetworkX library**

What is NetworkX?

**NetworkX is a graph (aka network)
analysis Python library**

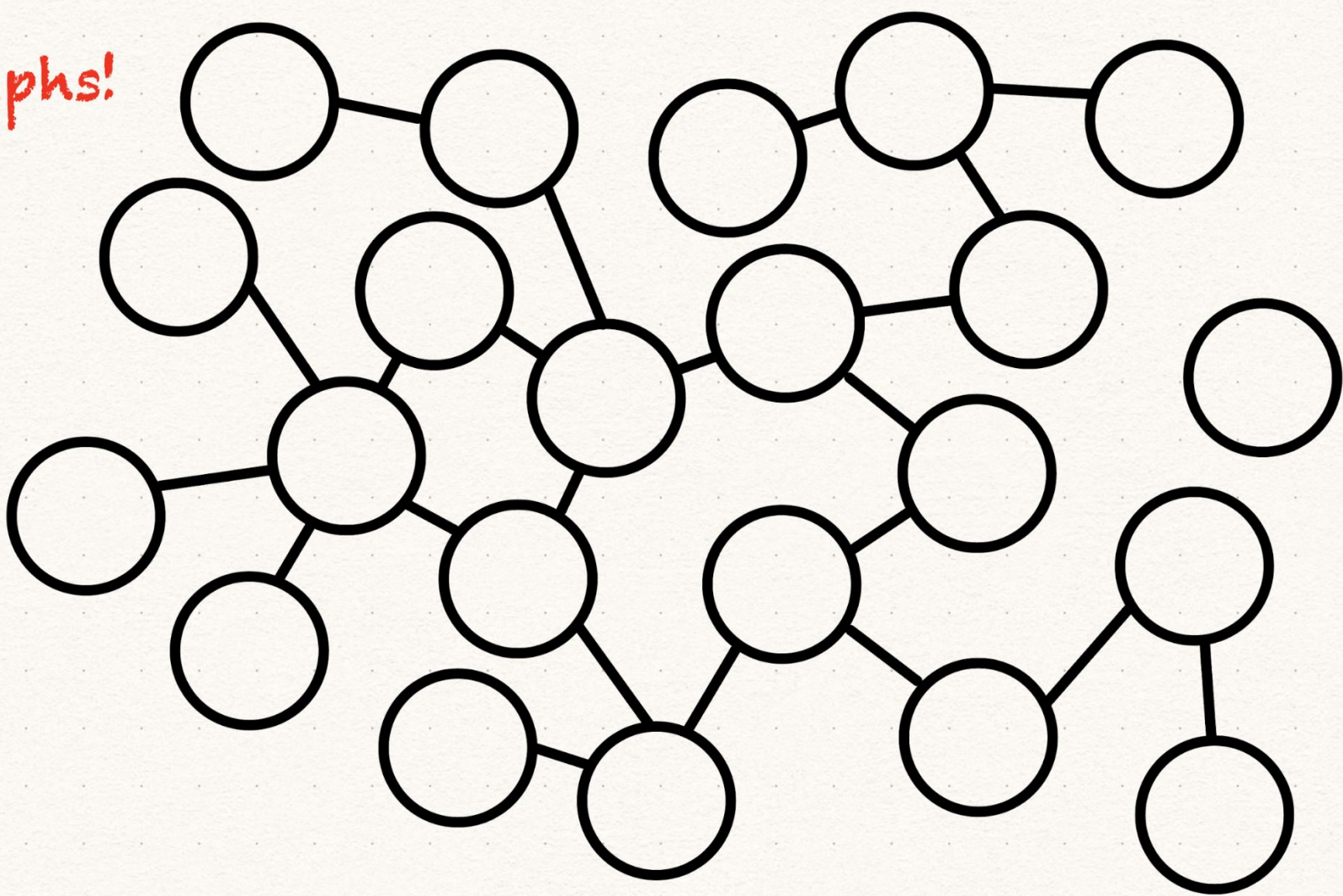
What is NetworkX?

NetworkX is a graph (aka network)
analysis Python library



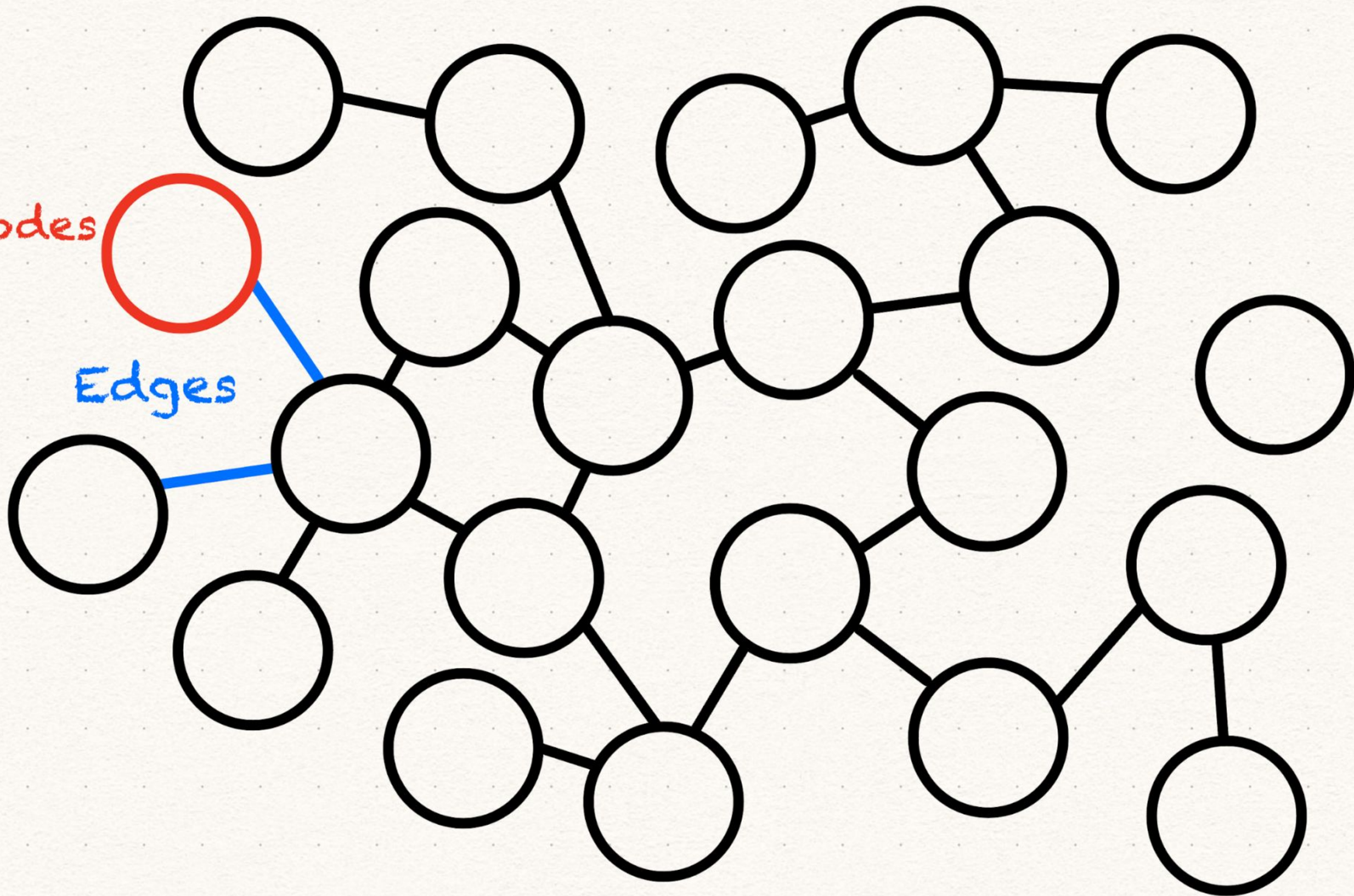
NOT these kind
of graphs!

Graphs!

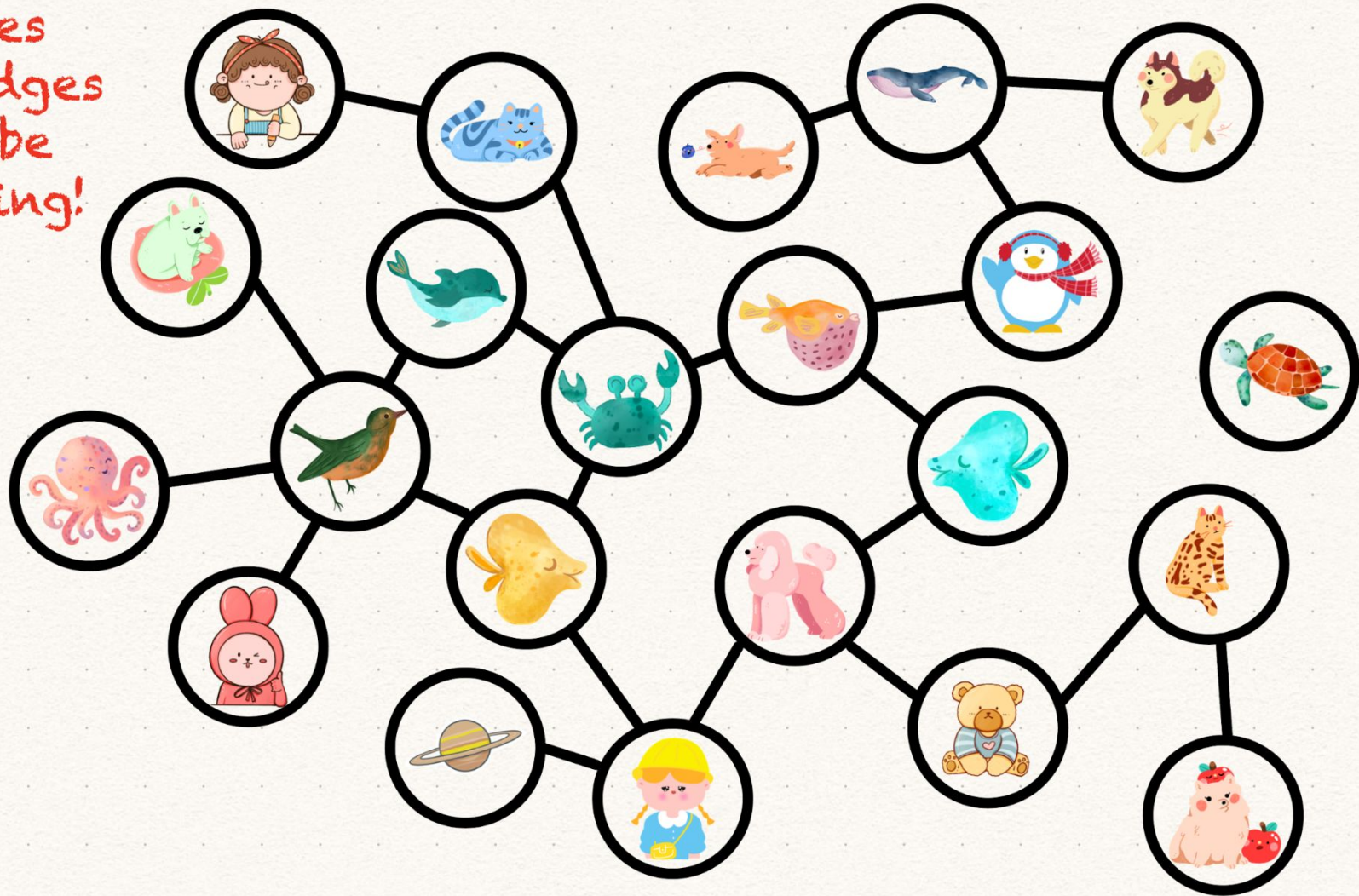


Nodes

Edges



Nodes
and edges
can be
anything!



<NetworkX DEMO>

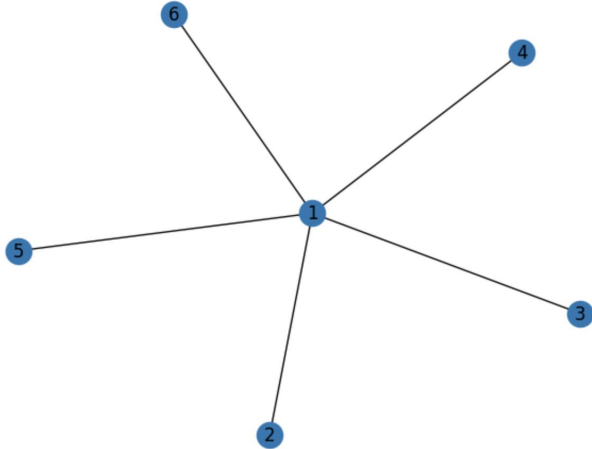
```
import networkx as nx
```

```
G = nx.Graph()
```

```
G.add_nodes_from([1, 2, 3, 4, 5, 6])
```

```
G.add_edges_from([(1, 2), (1, 3), (1, 4), (1, 5), (1, 6)])
```

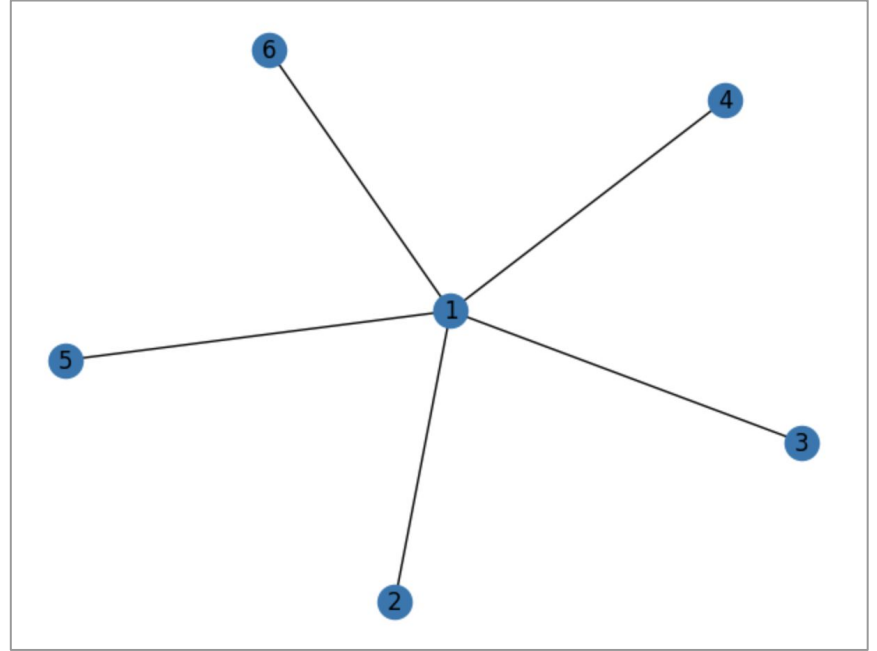
```
nx.draw(G, with_labels=True)
```



```
nx.betweenness centrality(G)
```

Tells us how important a node is.

```
{  
1: 1.0,  
2: 0.0,  
3: 0.0,  
4: 0.0,  
5: 0.0,  
6: 0.0  
}
```



Problem

```
big_G = nx.fast_gnp_random_graph(1000000, 0.5)  
nx.betweenness centrality(big_G)
```

Takes forever.... Probably a few years

Problem

```
big_G = nx.fast_gnp_random_graph(1000000, 0.5)  
nx.betweenness centrality(big_G)
```

Takes forever.... Probably a few years

Reason?

- Written in pure Python.
- But that is also what makes NetworkX simple.
- NetworkX was not created with performance in mind, but rather simplicity.

Workarounds...

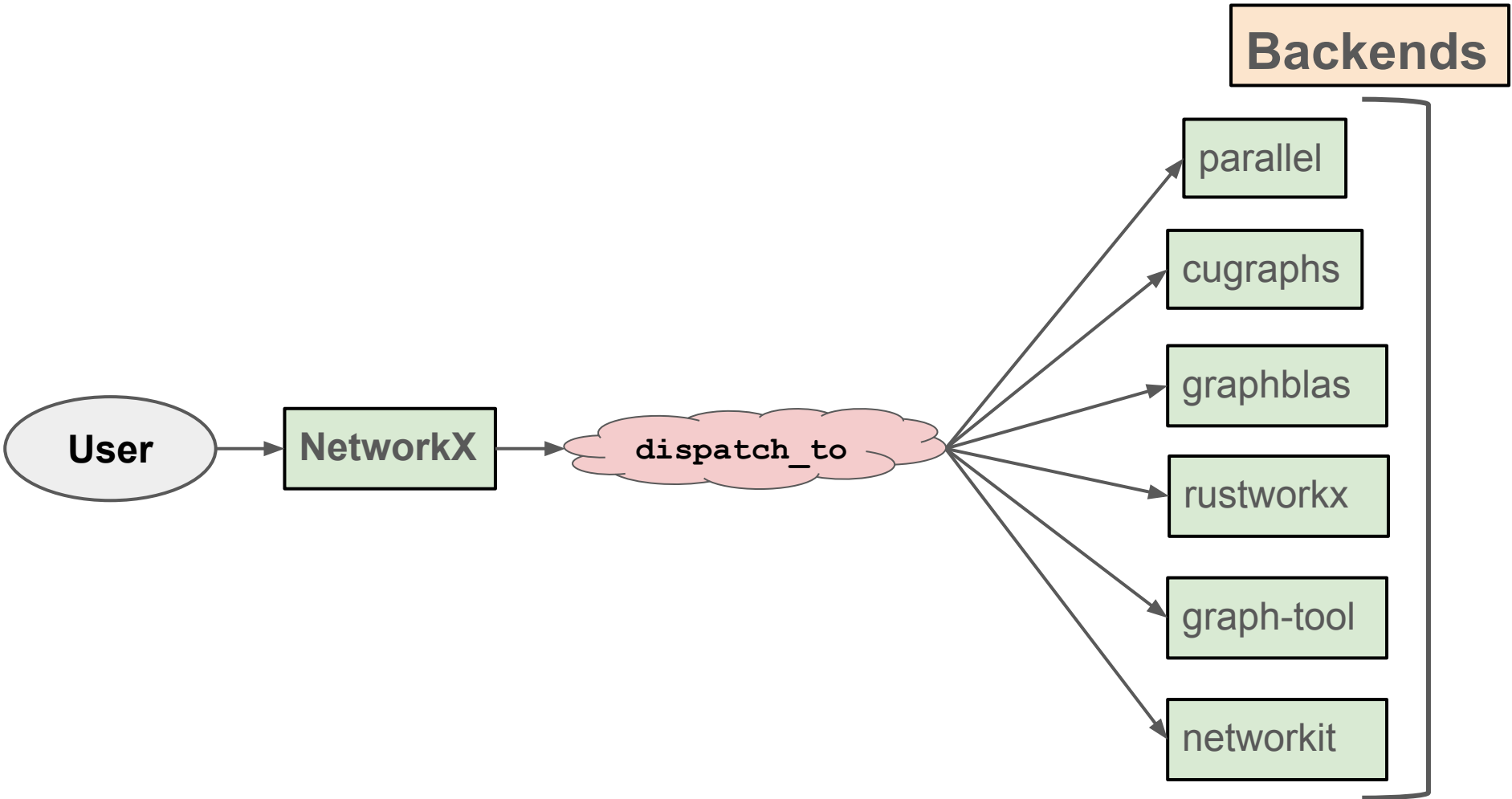
- **Switch to a faster graph library**
 - If cannot find one, then make one
 - Eg : graphblas, cugraphs, graph-tool, rustworkx, etc...
- Some issues - not as comprehensive and/or well-maintained as networkx or complex user-interface (**switching can be hard!**)

Workarounds...

- Switch to a faster library
 - If cannot find one, then make one
 - Eg : graphblas, cugraphs, graph-tool, rustworkx, etc...
- Some issues - not as comprehensive and/or well-maintained as networkx or complex user-interface (**switching can be hard!**)
- But, NetworkX's performance issue still persists...
 - cannot be re-written to be fast... NetworkX is too big for that(500-600 algos)
 - Also some consistency needed
- **Is there a way to integrate these different faster libraries into networkx?**

Workarounds...

- Switch to a faster library
 - If cannot find one, then make one
 - Eg : graphblas, cugraphs, graph-tool, rustworkx, etc...
- Some issues - not as comprehensive and/or well-maintained as networkx or complex user-interface (**switching can be hard!**)
- But, NetworkX's performance issue still persists...
 - cannot re-written to be fast... NetworkX is too big for that(500-600 algos)
 - Also some consistency needed
- **Is there a way to integrate these different faster libraries into networkx?**
 - **Dispatching!**



**Improving
Performance**

**Why would I
want to dispatch
a call?**

**And how exactly
is it done?**

<DISPATCHING DEMO>

4 ways of dispatching in NetworkX

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

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

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

graphblas

2 Types of Dispatching

Type based dispatching:

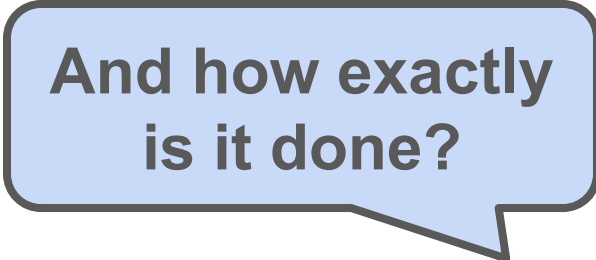
- `nx.betweenness centrality (CuG)`

Backend-name-based dispatching:

- `nx.betweenness centrality (G, backend = "parallel")`
- `with nx.config(backend_priority = ["cugraph", "graphblas"]):`
- `$ NETWORKX_BACKEND_PRIORITY="graphblas" && python nx_code.py`

Type-based dispatching and backend-name-based dispatching differences:

- For type-based dispatching,
 - we require each backend to have a unique type.
 - conversion of args and kwargs is not needed.



**And how exactly
is it done?**

**And how exactly
is it done?**

Python entry-points

used to extend the functionality of a project

Defined in the metadata files of a project

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

used by the networkx library to discover the backend libraries

Group

In backend's pyproject.toml

Name

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

parallel = "nx_parallel.interface:BackendInterface"

Backend Name

Object reference

All the backend algorithms

BackendInterface namespace

``convert_from_nx``

``convert_to_nx``

How does entry-point based dispatching work?

**Grouping all the
installed backends**

```
entry_points(group="networkx.backends")
```

Get all the installed packages having this entry-point defined in their metadata.

How does entry-point based dispatching work?

**Grouping all the
installed backends**

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

Get backend name:

1. ``__networkx_backend__`` attribute of the backend graph class (Type-based)
2. from the user (backend-name-based)

How does entry-point based dispatching work?

Grouping all the
installed backends

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

Get backend name

```
backend_name
```

Get the `EntryPoint` object
for the given backend

```
backend = eps[backend_name]
```


How does entry-point based dispatching work?

Grouping all the
installed backends

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

Get backend name

```
backend_name
```

Get the `EntryPoint` object
for the given backend

```
backend = eps[backend_name]
```

Get the object
reference of `backend`

```
backend_namespace = backend.load()
```

How does entry-point based dispatching work?

Grouping all the
installed backends

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

Get backend name

```
backend_name
```

Get the `EntryPoint` object
for the given backend

```
backend = eps[backend_name]
```

Get the object
reference of `backend`

```
backend_namespace = backend.load()
```

Conversion step, if
backend-name-based)

```
backend_namespace.convert_from_nx(*args, **kwargs)
```

How does entry-point based dispatching work?

Grouping all the installed backends

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

Get backend name

```
backend_name
```

Get the `EntryPoint` object for the given backend

```
backend = eps[backend_name]
```

Get the object reference of `backend`

```
backend_namespace = backend.load()
```

Conversion step, if backend-name-based)

```
backend_namespace.convert_from_nx(*args, **kwargs)
```

Load the backend implementation from the given function call it

```
output = backend_namespace.func_name(*args, **kwargs)
```

How does entry-point based dispatching work?

Grouping all the installed backends

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

Get backend name

```
backend_name
```

Get the `EntryPoint` object for the given backend

```
backend = eps[backend_name]
```

Get the object reference of `backend`

```
backend_namespace = backend.load()
```

Conversion step, if backend-name-based)

```
backend_namespace.convert_from_nx(*args, **kwargs)
```

Load the backend implementation from the given function call it

```
output = backend_namespace.func_name(*args, **kwargs)
```

Convert output type, if needed

```
backend_namespace.convert_to_nx(output)
```

How does entry-point based dispatching work?

Grouping all the installed backends

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

Get backend name

```
backend_name
```

Get the `EntryPoint` object for the given backend

```
backend = eps[backend_name]
```

Get the object reference of `backend`

```
backend_namespace = backend.load()
```

Conversion step, if backend-name-based)

```
backend_namespace.convert_from_nx(*args, **kwargs)
```

Load the backend implementation from the given function call it

```
output = backend_namespace.func_name(*args, **kwargs)
```

Convert output type, if needed

```
backend_namespace.convert_to_nx(output)
```

**Dispatching in NetworkX involves
some more smaller steps/parts but
these 7 processes/steps are the
main ones!**

Some more dispatching-related stuff:

logging

Automatic testing

```
NETWORKX_TEST_BACKEND=parallel  
NETWORKX_FALLBACK_TO_NX=True  
pytest --pyargs networkx
```

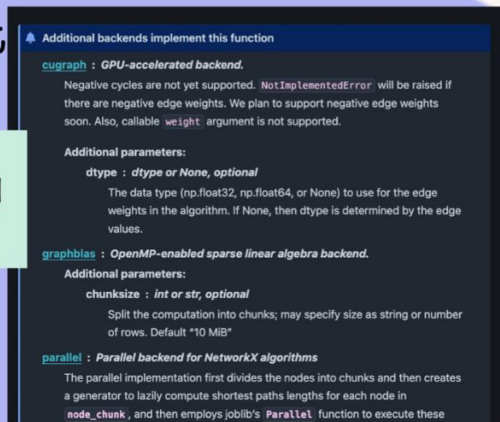
Additional backend args

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

2nd entry point

```
[project.entry-  
points."networkx.backend_info"]  
parallel = "nx_parallel:get_info"
```

source - https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.shortest_paths.weighted.all_pairs_bellman_ford_path_length.html



Configurations

```
nx.config.backends.parallel.n_jobs = 8  
nx.config.backends.parallel.verbose = 10
```

can_run
should_run

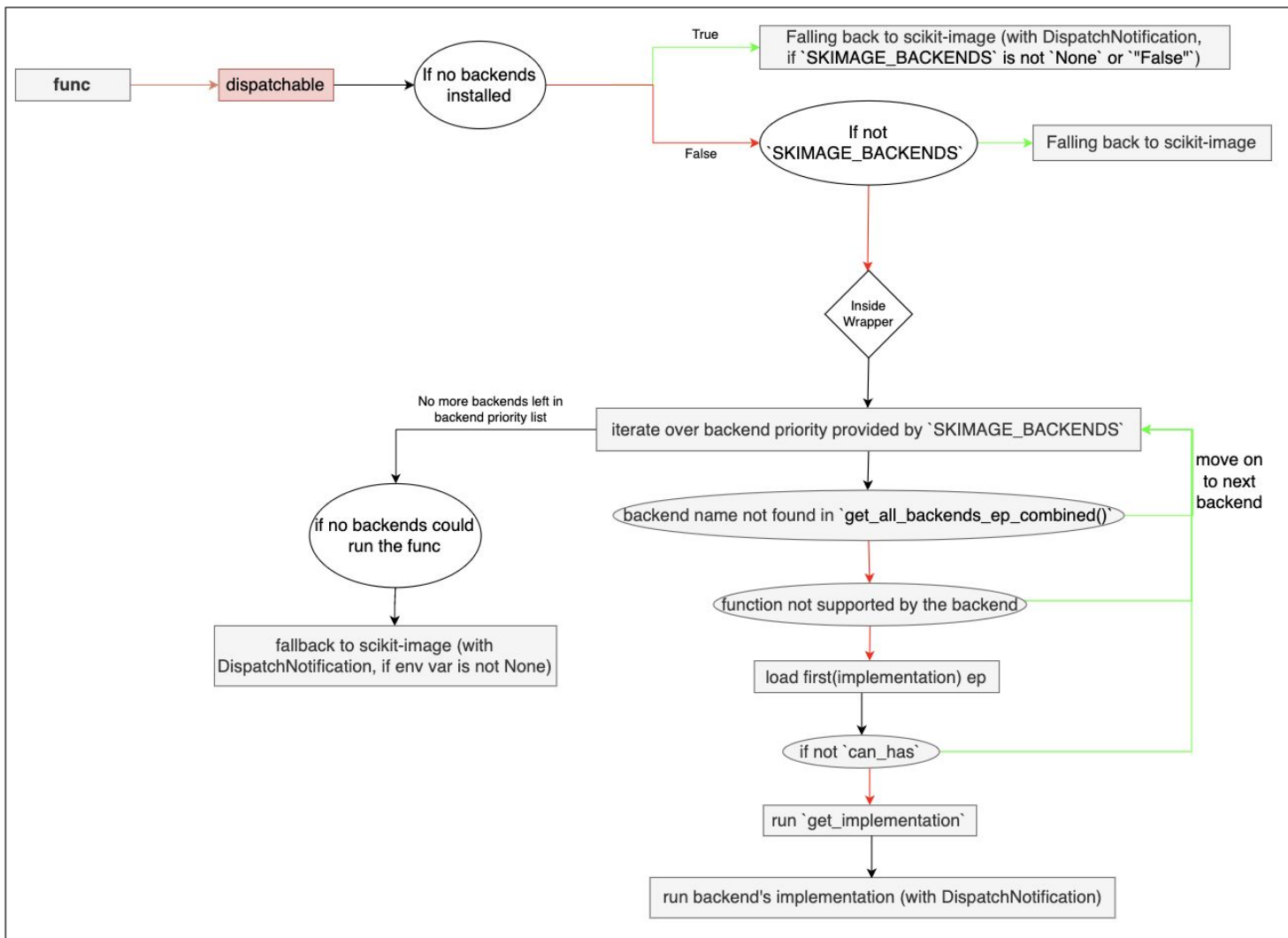
fallback
option

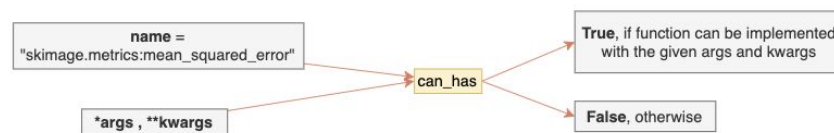
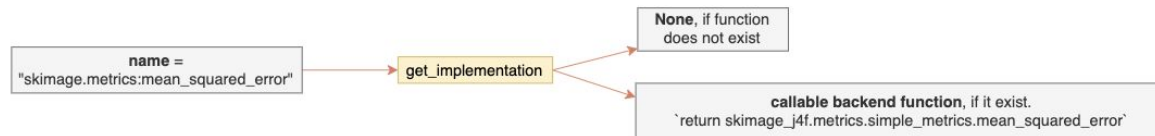
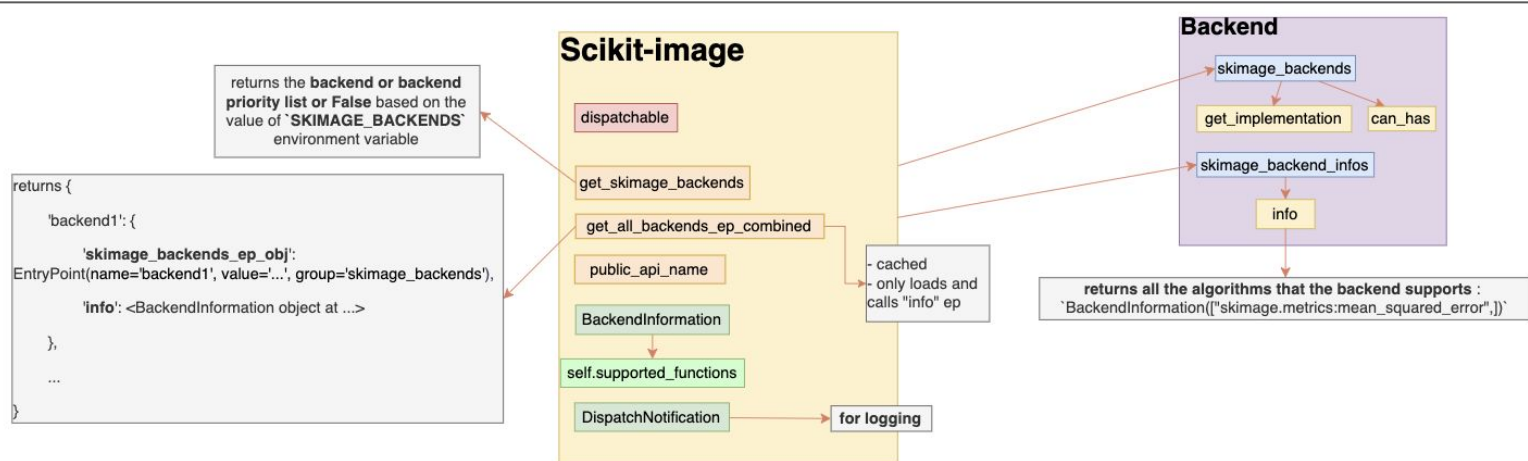
Caching
conversion

Dispatching in scikit-image

How is it different?

- Image object (`numpy.ndarray`) instead of a `nx.Graph` object
- No type-based dispatching, only backend-name-based dispatching (but maybe we'll have it in future?) because we want to allow multiple backends to support same array types
- No array conversions right now!
- Some other trivial differences but it's entry-point based dispatching only!





Mini-ecosystem of NetworkX backends:

Well-maintained:

- nx-parallel : <https://github.com/networkx/nx-parallel>
- nx-cugraph : <https://github.com/rapidsai/nx-cugraph>
- nx-arangodb : <https://github.com/arangodb/nx-arangodb>
- graphblas-algorithms : <https://github.com/python-graphblas/graphblas-algorithms>

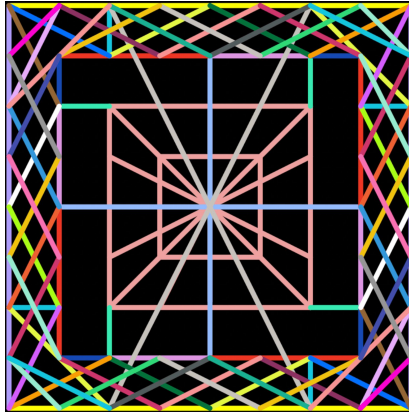
Experimental:

- nx-pandas : <https://github.com/networkx/nx-pandas>
- rustworkx-backend : <https://github.com/thomasjpfan/rustworkx-backend>
- Visualisation backend??
- ... more to come

Dispatching in Scientific Python ecosystem

- SPEC 2 : <https://scientific-python.org/specs/spec-0002/>
- spatch : <https://github.com/scientific-python/spatch/issues/1>
- Scientific Python discord: <https://discord.com/invite/vur45CbwMz>
- 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/7520)
 - <https://github.com/betatim/scikit-image/pull/1>
 - <https://github.com/rapidsai/cucim/issues/829>
- NumPy's type-based dispatching
 - <https://numpy.org/neps/nep-0037-array-module.html>
 - <https://numpy.org/neps/nep-0047-array-api-standard.html>
 - <https://data-apis.org/array-api/latest/>
- Scikit-learn
 - <https://github.com/scikit-learn/scikit-learn/pull/30250>

***Thank you to Scientific Python community and
PyConf Hyderabad team for all the support, and
thank you everyone for listening :)***



**@Schefflera-Arboricola
Aditi Juneja**