

Plug Life Into Your Codebase

Making an established Python
codebase pluggable

Bianca Henderson

April 22, 2023



Hello! 

Agenda

What we'll cover in this talk

- 1 | What are plugins?
- 2 | Introducing `pluggy`
- 3 | A real-world example
- 4 | Other use cases
- 5 | Reference materials

Who am I?

- Software engineer at Anaconda
- Taught myself Python via books + creating text-based adventure games
- My favorite thing to work on is CLIs
- Also interested in video games, board games, science/math, 3D printing, AI research, art, writing, music, etc.



Who is this talk for?

- Developers who are curious about plugin infrastructure
- Anyone interested in learning more about conda or pluggy
- Codebase maintainers who are interested in expanding the functionality of their Python project!



What are plugins?

*With a sufficient number of users of an API,
it does not matter what you promise in the contract:
all observable behaviors of your system will be
dependent on by somebody.*

– Hyrum's Law

Plugins are guardrails to alleviate the effects of Hyrum's Law

Analogy Time! 

What are plugins?



What are plugins?



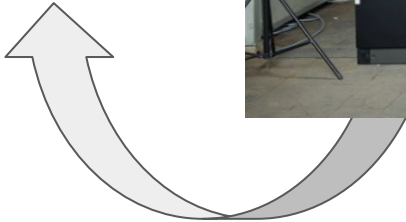
What are plugins?



What are plugins?



What are plugins?



What are plugins?



What are plugins? (recap)

- Customization or extra feature
- Not part of the default codebase
- Discoverable
- Can be distributed separately
- Can be maintained by others

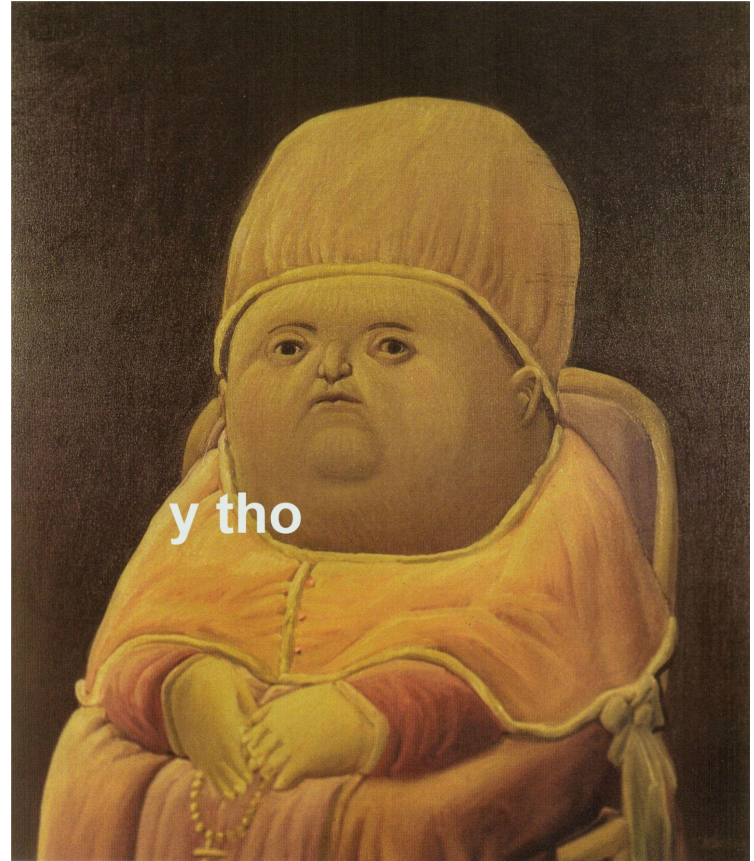
Plugin examples

- Browser extensions
- Language support in IDEs
- Media players
- ... and more



Why plugins?

- Encourages innovation
- Enforces separation of concern
- Enables community contributions
- Can address security issues
- Improves UX for end users



Plugin fundamentals

- Discovery
- Registration
- Application hooks
- API exposure

Introducing: pluggy!

- Open source and written in Python
`github.com/pytest-dev/pluggy`
- Host, hooks, plugin
- At the core of `pytest`, `tox`, `devpi`

Documentation!

`https://pluggy.readthedocs.io/en/stable/index.html#`



How to use pluggy

- Have host
- Write plugin
- Define hook specifications
- Mark hook implementation



Why pluggy?

- Enables more loosely coupled systems
- Responsibility is on host program designer, not plugin designer
- Clear framework
- Straightforward yet customizable



pluggy Example

```
# example.py

import pluggy

hookspec = pluggy.HookspecMarker("example_for_pycon")
hookimpl = pluggy.HookimplMarker("example_for_pycon")

class ExampleSpec:
    """A hook specification namespace."""

    @hookspec
    def myhook(self, arg1, arg2):
        """My special little hook that you can customize."""

~file shortened~
```

pluggy Example

```
# example.py
```

```
import pluggy
```

```
hookspec = pluggy.HookspecMarker("example_for_pycon")
```

```
hookimpl = pluggy.HookimplMarker("example_for_pycon")
```

```
class ExampleSpec:
```

```
    """A hook specification namespace."""
```

```
    @hookspec
```

```
    def myhook(self, arg1, arg2):
```

```
        """My special little hook that you can customize."""
```

```
~file shortened~
```

pluggy Example

```
# example.py
```

```
import pluggy
```

```
hookspec = pluggy.HookspecMarker("example_for_pycon")
```

```
hookimpl = pluggy.HookimplMarker("example_for_pycon")
```

```
class ExampleSpec:
```

```
    """A hook specification namespace."""
```

```
    @hookspec
```

```
    def myhook(self, arg1, arg2):
```

```
        """My special little hook that you can customize."""
```

```
~file shortened~
```

**Every `pluggy` application
needs a hook specification to
register any plugin hooks**

pluggy Example

```
# example.py

import pluggy

hookspec = pluggy.HookspecMarker("example_for_pycon")
hookimpl = pluggy.HookimplMarker("example_for_pycon")

class ExampleSpec:
    """A hook specification namespace."""

    @hookspec
    def myhook(self, arg1, arg2):
        """My special little hook that you can customize."""

~file shortened~
```

The `hookimpl` object marks the implementations of the plugin hooks we have defined

pluggy Example

```
# example.py

import pluggy

hookspec = pluggy.HookspecMarker("example_for_pycon")
hookimpl = pluggy.HookimplMarker("example_for_pycon")

class ExampleSpec:
    """A hook specification namespace."""

    @hookspec
    def myhook(self, arg1, arg2):
        """My special little hook that you can customize."""

~file shortened~
```

pluggy Example

```
# example.py
```

```
import pluggy
```

```
hookspec = pluggy.HookspecMarker("example_for_pycon")
```

```
hookimpl = pluggy.HookimplMarker("example_for_pycon")
```

```
class ExampleSpec:
```

```
    """A hook specification namespace."""
```

```
@hookspec
```

```
def myhook(self, arg1, arg2):
```

```
    """My special little hook that you can customize."""
```

```
~file shortened~
```

pluggy Example

```
# example.py (continued)

class MyPlugin:
    """A hook implementation namespace."""

    @hookimpl
    def myhook(self, arg1, arg2):
        print("inside Plugin_1.myhook()")
        return arg1 + arg2

pm = pluggy.PluginManager("example_for_pycon")
pm.add_hookspecs(ExampleSpec)

pm.register(MyPlugin())

results = pm.hook.myhook(arg1=1, arg2=2)
print(results)
```

pluggy Example

```
# example.py (continued)

class MyPlugin:
    """A hook implementation namespace."""

    @hookimpl
    def myhook(self, arg1, arg2):
        print("inside MyPlugin.myhook()")
        return arg1 + arg2

pm = pluggy.PluginManager("example_for_pycon")
pm.add_hookspecs(ExampleSpec)

pm.register(MyPlugin())

results = pm.hook.myhook(arg1=1, arg2=2)
print(results)
```

pluggy Example

```
# example.py (continued)
```

```
class MyPlugin:  
    """A hook implementation namespace."""
```

```
    @hookimpl
```

```
    def myhook(self, arg1, arg2):
```

```
        print("inside MyPlugin.myhook()")
```

```
        return arg1 + arg2
```

This function with the hook implementation decorator must have the same name as the function decorated by the corresponding hook specification!

```
pm = pluggy.PluginManager("example_for_pycon")
```

```
pm.add_hookspecs(ExampleSpec)
```

```
pm.register(MyPlugin())
```

```
results = pm.hook.myhook(arg1=1, arg2=2)
```

```
print(results)
```

pluggy Example

```
# example.py

import pluggy

hookspec = pluggy.HookspecMarker("example_for_pycon")
hookimpl = pluggy.HookimplMarker("example_for_pycon")

class ExampleSpec:
    """A hook specification namespace."""

    @hookspec
    def myhook(self, arg1, arg2):
        """My special little hook that you can customize."""

~file shortened~
```

pluggy Example

```
# example.py (continued)
```

```
class MyPlugin:
    """A hook implementation namespace."""

    @hookimpl
    def myhook(self, arg1, arg2):
        print("inside MyPlugin.myhook()")
        return arg1 + arg2
```

```
pm = pluggy.PluginManager("example_for_pycon")  
pm.add_hookspecs(ExampleSpec)
```

```
pm.register(MyPlugin())
```

```
results = pm.hook.myhook(arg1=1, arg2=2)  
print(results)
```

Here we create a plugin manager
and add the specification

pluggy Example

```
# example.py

import pluggy

hookspec = pluggy.HookspecMarker("example_for_pycon")
hookimpl = pluggy.HookimplMarker("example_for_pycon")

class ExampleSpec:
    """A hook specification namespace."""

    @hookspec
    def myhook(self, arg1, arg2):
        """My special little hook that you can customize."""

~file shortened~
```


pluggy Example

```
# example.py (continued)
```

```
class MyPlugin:
```

```
    """A hook implementation namespace."""
```

```
    @hookimpl
```

```
    def myhook(self, arg1, arg2):
```

```
        print("inside MyPlugin.myhook()")
```

```
        return arg1 + arg2
```

```
pm = pluggy.PluginManager("example_for_pycon")
```

```
pm.add_hookspecs(ExampleSpec)
```

```
pm.register(MyPlugin())
```

```
results = pm.hook.myhook(arg1=1, arg2=2)
```

```
print(results)
```

We register the plugin here

pluggy Example

```
# example.py (continued)

class MyPlugin:
    """A hook implementation namespace."""

    @hookimpl
    def myhook(self, arg1, arg2):
        print("inside MyPlugin.myhook()")
        return arg1 + arg2

pm = pluggy.PluginManager("example_for_pycon")
pm.add_hookspecs(ExampleSpec)

pm.register(MyPlugin())

results = pm.hook.myhook(arg1=1, arg2=2)
print(results)
```

We call the “myhook”
hook here

Running the `pluggy` Example

```
$ python3 example.py
```

Running the `pluggy` Example

```
$ python3 example.py
```

```
inside MyPlugin.myhook()
```

```
[3]
```

pluggy Example

```
# example.py (continued)
```

```
class MyPlugin:
```

```
    """A hook implementation namespace."""
```

```
    @hookimpl
```

```
    def myhook(self, arg1, arg2):
```

```
        print("inside MyPlugin.myhook()")  
        return arg1 + arg2
```

```
pm = pluggy.PluginManager("example_for_pycon")
```

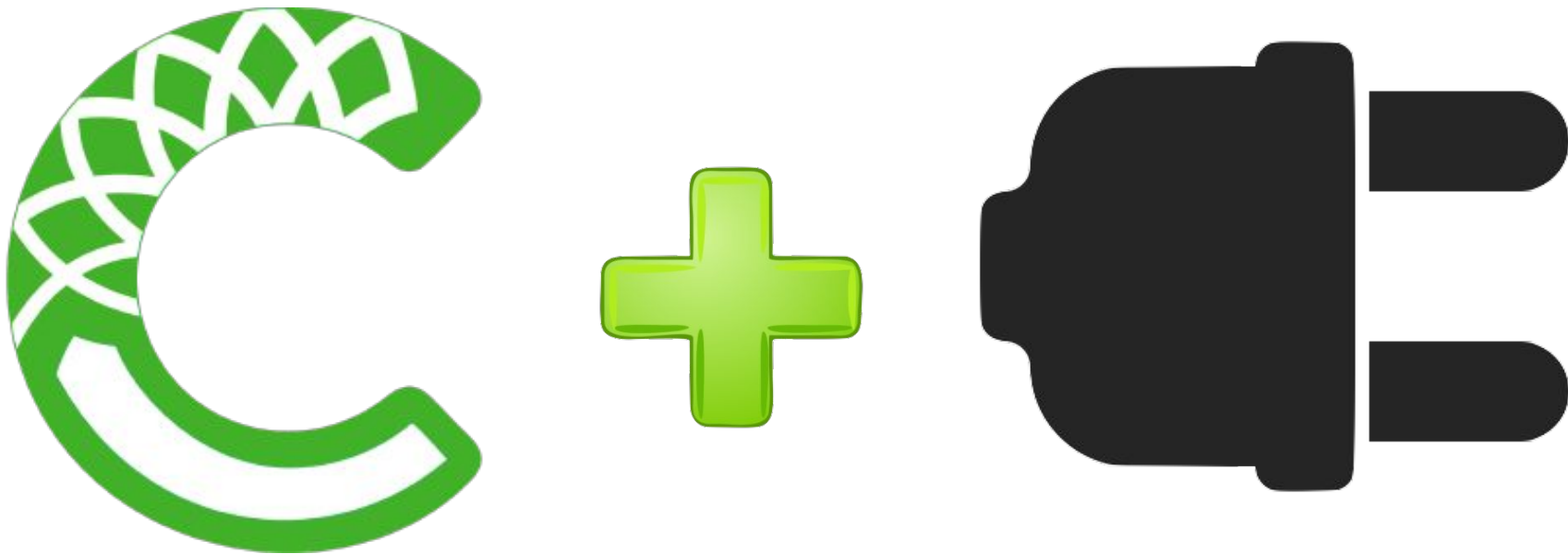
```
pm.add_hookspecs(ExampleSpec)
```

```
pm.register(MyPlugin())
```

```
results = pm.hook.myhook(arg1=1, arg2=2)  
print(results)
```

What the plugin is doing

A real-world example: Conda

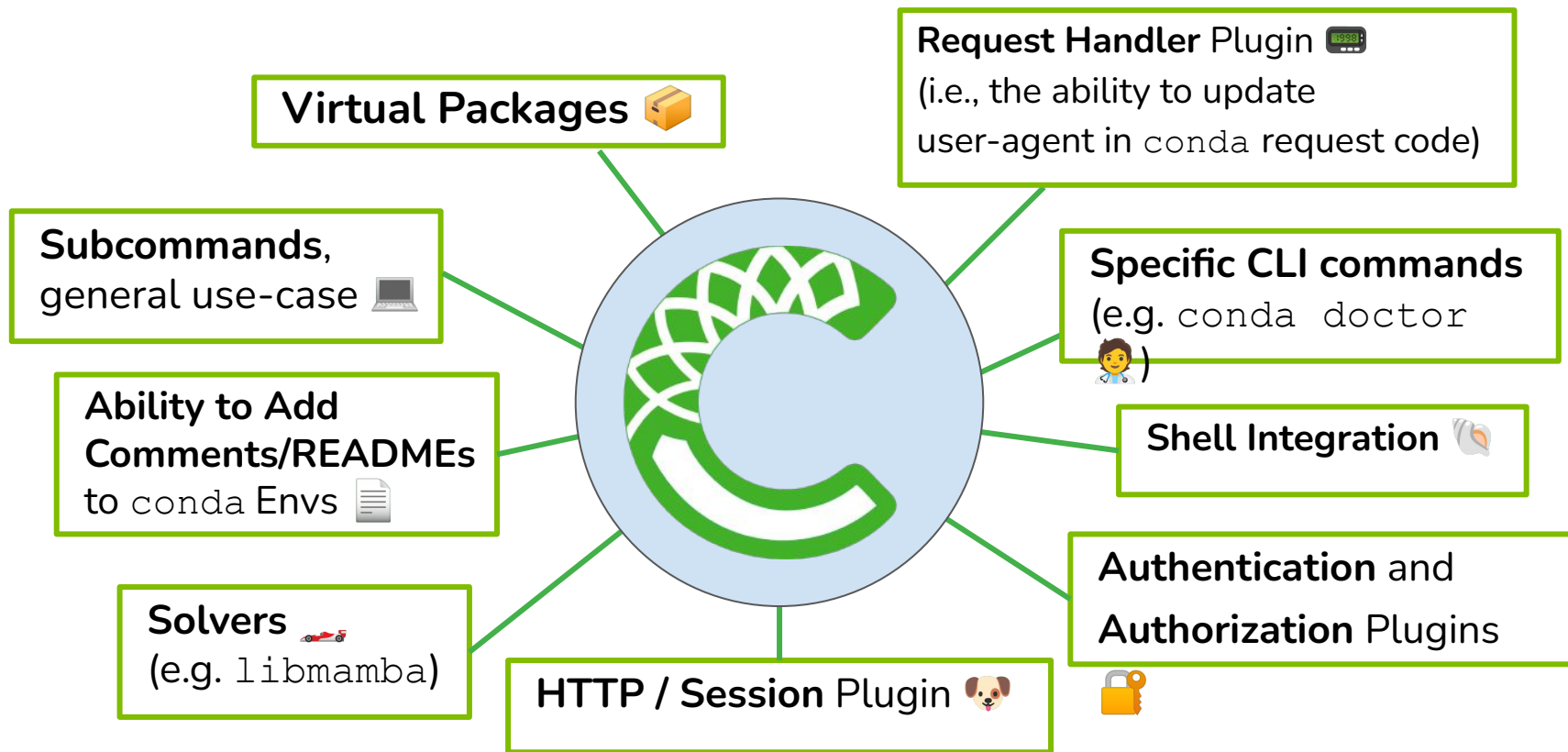


What is conda?

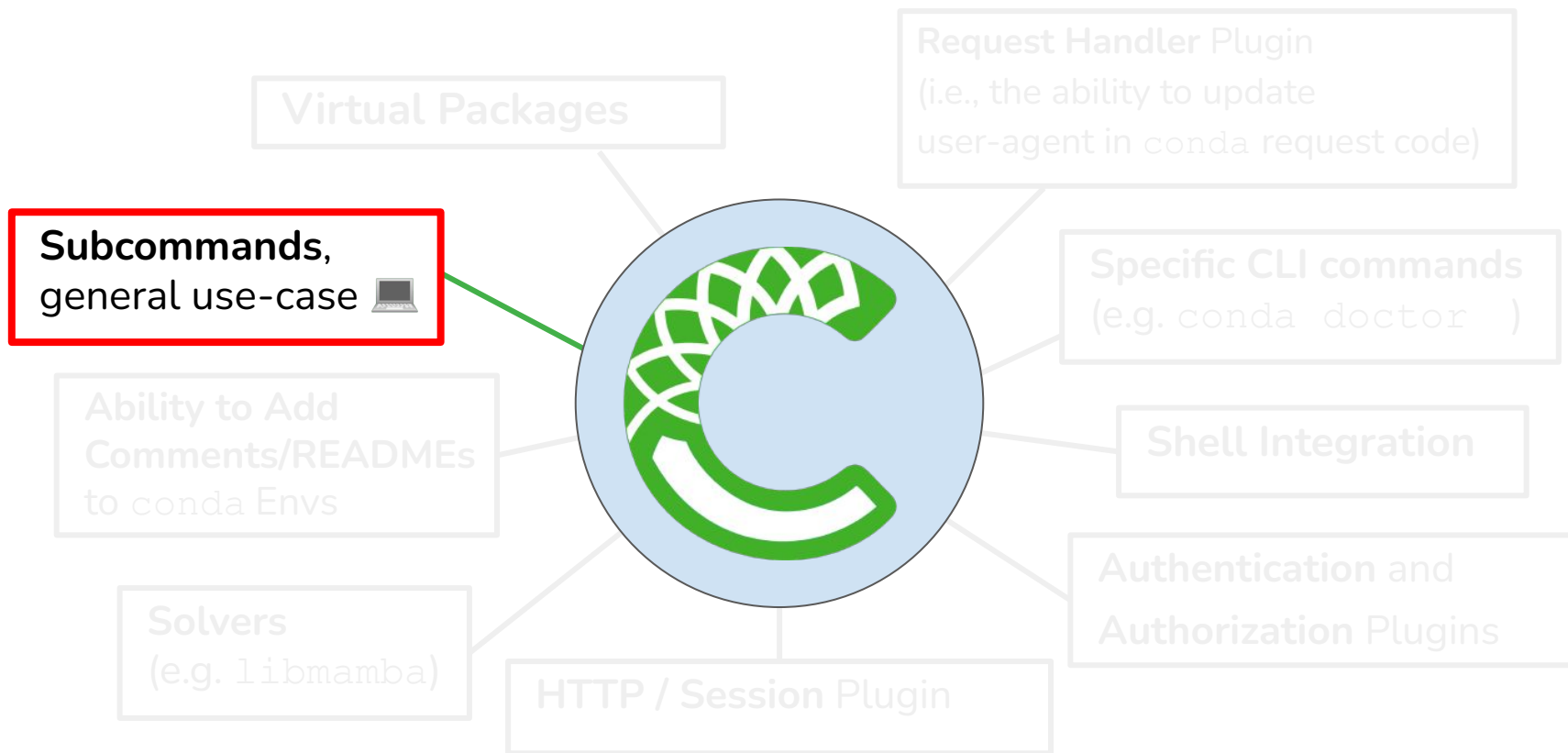
- Package and environment manager
- Written in Python
- macOS, Windows, Linux
- Open source
- Language-agnostic
- More than 35 million active users
- Over 10 years old



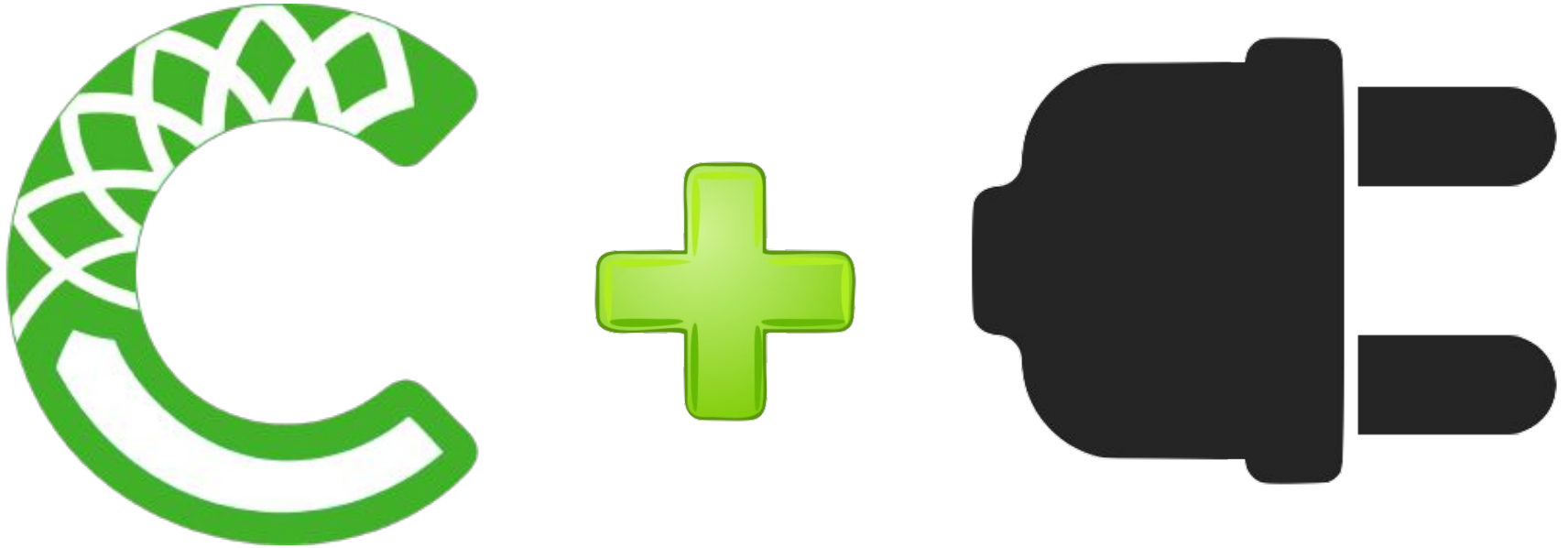
Conda plugins use cases (not a complete list!)



Conda plugins use cases (not a complete list!)



Let's get into it some details!



How to use pluggy

- Have host —> Conda
- Write plugin —> Subcommands
- Define hook specifications
- Mark hook implementation



Conda's plugin project structure

```
$ tree conda/plugins/
```

```
|— __init__.py
|— hookspec.py
|— manager.py
|— solvers.py
|— types.py
|— verbose_solver.py
└─ virtual_packages
    |— __init__.py
    |— archspec.py
    |— cuda.py
    |— linux.py
    |— osx.py
    |— specs
    └─ windows.py
```

Conda's plugin project structure

```
$ tree conda/plugins/
```

```
|— __init__.py  
|— hookspec.py  
|— manager.py  
|— solvers.py  
|— types.py  
|— verbose_solver.py
```

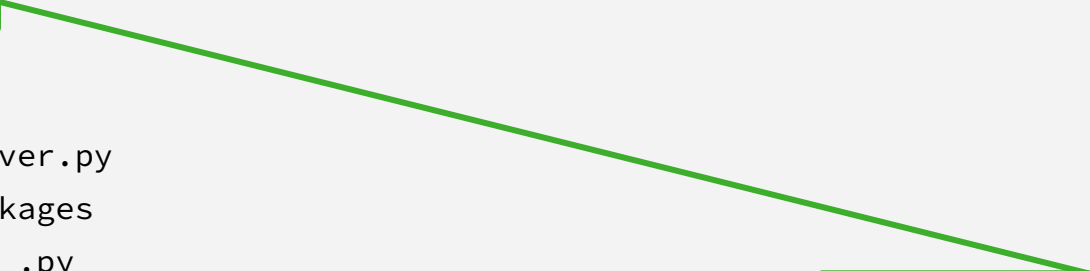
```
└─ virtual_packages  
   |— __init__.py  
   |— archs.py  
   |— cuda.py  
   |— linux.py  
   |— osx.py  
   |— specs  
   └─ windows.py
```

Specific plugin types can be shipped by default with the codebase and exist in their own directories

Conda's plugin project structure

```
$ tree conda/plugins/
```

```
|— __init__.py
|— hookspec.py
|— manager.py
|— solvers.py
|— types.py
|— verbose_solver.py
|— virtual_packages
|   |— __init__.py
|   |— archspec.py
|   |— cuda.py
|   |— linux.py
|   |— osx.py
|   |— specs
|   |— windows.py
```



Customization of plugin management is possible with custom classes that subclass from
`pluggy.PluginManager`

Conda's plugin project structure

```
$ tree conda/plugins/
```

```
|— __init__.py
|— hookspec.py
|— manager.py
|— solvers.py
|— types.py
|— verbose_solver.py
└─ virtual_packages
    |— __init__.py
    |— archspec.py
    |— cuda.py
    |— linux.py
    |— osx.py
    |— specs
    └─ windows.py
```

conda/plugins/**hookspec.py**

```
import pluggy

from .types import CondaSolver, CondaSubcommand, CondaVirtualPackage

spec_name = "conda"
hookspec = pluggy.HookspecMarker(spec_name)
hookimpl = pluggy.HookimplMarker(spec_name)

class CondaSpecs:
    """
    The conda plugin hookspecs, to be used by developers.
    """
    @hookspec
    def conda_subcommands(self) -> Iterable[CondaSubcommand]:
        """
        Register external subcommands in conda.
        :return: An iterable of subcommand entries.
        """
```


conda/plugins/**hookspec.py**

```
import pluggy
```

```
from .types import CondaSolver, CondaSubcommand, CondaVirtualPackage
```

```
spec_name = "conda"
```

```
hookspec = pluggy.HookspecMarker(spec_name)
```

```
hookimpl = pluggy.HookimplMarker(spec_name)
```

```
class CondaSpecs:
```

```
    """
```

```
    The conda plugin hookspecs, to be used by developers.
```

```
    """
```

```
    @hookspec
```

```
    def conda_subcommands(self) -> Iterable[CondaSubcommand]:
```

```
        """
```

```
        Register external subcommands in conda.
```

```
        :return: An iterable of subcommand entries.
```

```
        """
```

conda/plugins/**hookspec.py**

```
import pluggy

from .types import CondaSolver, CondaSubcommand, CondaVirtualPackage

spec_name = "conda"
hookspec = pluggy.HookspecMarker(spec_name)
hookimpl = pluggy.HookimplMarker(spec_name)

class CondaSpecs:
    """
    The conda plugin hookspecs, to be used by developers.
    """
    @hookspec
    def conda_subcommands(self) -> Iterable[CondaSubcommand]:
        """
        Register external subcommands in conda.
        :return: An iterable of subcommand entries.
        """
```

Every `pluggy` application
needs a hook specification to
register any plugin hooks

conda/plugins/**hookspec.py**

```
import pluggy

from .types import CondaSolver, CondaSubcommand, CondaVirtualPackage

spec_name = "conda"
hookspec = pluggy.HookspecMarker(spec_name)
hookimpl = pluggy.HookimplMarker(spec_name)

class CondaSpecs:
    """
    The conda plugin hookspecs, to be used by developers.
    """
    @hookspec
    def conda_subcommands(self) -> Iterable[CondaSubcommand]:
        """
        Register external subcommands in conda.
        :return: An iterable of subcommand entries.
        """
```

The `hookimpl` object marks the implementations of the plugin hooks we have defined

conda/plugins/**hookspec.py**

```
import pluggy

from .types import CondaSolver, CondaSubcommand, CondaVirtualPackage

spec_name = "conda"
hookspec = pluggy.HookspecMarker(spec_name)
hookimpl = pluggy.HookimplMarker(spec_name)
```

class CondaSpecs:

```
    """
    The conda plugin hookspecs, to be used by developers.
    """
    @hookspec
    def conda_subcommands(self) -> Iterable[CondaSubcommand]:
        """
        Register external subcommands in conda.
        :return: An iterable of subcommand entries.
        """
```

conda/plugins/**hookspec.py**

```
import pluggy

from .types import CondaSolver, CondaSubcommand, CondaVirtualPackage

spec_name = "conda"
hookspec = pluggy.HookspecMarker(spec_name)
hookimpl = pluggy.HookimplMarker(spec_name)

class CondaSpecs:
    """
    The conda plugin hookspecs, to be used by developers.
    """
    @hookspec
    def conda_subcommands(self) -> Iterable[CondaSubcommand]:
        """
        Register external subcommands in conda.
        :return: An iterable of subcommand entries.
        """
```

conda/plugins/**hookspec.py**

```
import pluggy

from .types import CondaSolver, CondaSubcommand, CondaVirtualPackage

spec_name = "conda"
hookspec = pluggy.HookspecMarker(spec_name)
hookimpl = pluggy.HookimplMarker(spec_name)

class CondaSpecs:
    """
    The conda plugin hookspecs, to be used by developers.
    """
    @hookspec
    def conda_subcommands(self) -> Iterable[CondaSubcommand]:
        """
        Register external subcommands in conda.
        :return: An iterable of subcommand entries.
        """
```

Conda's plugin project structure

```
$ tree conda/plugins/
```

```
|— __init__.py
|— hookspec.py
|— manager.py
|— solvers.py
|— types.py
|— verbose_solver.py
└─ virtual_packages
    |— __init__.py
    |— archspec.py
    |— cuda.py
    |— linux.py
    |— osx.py
    |— specs
    └─ windows.py
```

conda/plugins/**types.py**

```
from __future__ import annotations
from typing import Callable, NamedTuple
from ..core.solve import Solver
```

```
class CondaSubcommand(NamedTuple):
    """
    A conda subcommand.

    :param name: Subcommand name (e.g., ``conda my-subcommand-name``).
    :param summary: Subcommand summary, will be shown in ``conda --help``.
    :param action: Callable that will be run when the subcommand is invoked.
    """

    name: str
    summary: str
    action: Callable[
        [list[str]], # arguments
        int | None, # return code
    ]
```


conda/plugins/**types.py**

```
from __future__ import annotations
from typing import Callable, NamedTuple
from ..core.solve import Solver
```

```
class CondaSubcommand(NamedTuple):
    """
    A conda subcommand.
    :param name: Subcommand name (e.g., ``conda my-subcommand-name``).
    :param summary: Subcommand summary, will be shown in ``conda --help``.
    :param action: Callable that will be run when the subcommand is invoked.
    """

    name: str
    summary: str
    action: Callable[
        [list[str]], # arguments
        int | None, # return code
    ]
```

conda/plugins/**types.py**

```
from __future__ import annotations
from typing import Callable, NamedTuple
from ..core.solve import Solver
```

```
class CondaSubcommand(NamedTuple):
    """
    A conda subcommand.
    :param name: Subcommand name (e.g., ``conda my-subcommand-name``).
    :param summary: Subcommand summary, will be shown in ``conda --help``.
    :param action: Callable that will be run when the subcommand is invoked.
    """

    name: str
    summary: str
    action: Callable[
            [list[str]], # arguments
            int | None, # return code
    ]
```

conda/plugins/**types.py**

```
from __future__ import annotations
from typing import Callable, NamedTuple
from ..core.solve import Solver
```

```
class CondaSubcommand(NamedTuple):
    """
    A conda subcommand.
    :param name: Subcommand name (e.g., ``conda my-subcommand-name``).
    :param summary: Subcommand summary, will be shown in ``conda --help``.
    :param action: Callable that will be run when the subcommand is invoked.
    """
    name: str
    summary: str
    action: Callable[
        [list[str]], # arguments
        int | None, # return code
    ]
```

ascii_graph/**ascii_graph.py**

```
import argparse
from sympy import symbols
from sympy.plotting import textplot

import conda.plugins

def ascii_graph(argv: list):
    parser = argparse.ArgumentParser("conda ascii-graph")
    parser.add_argument("x", type=float, help="First coordinate to graph")
    parser.add_argument("y", type=float, help="Second coordinate to graph")
    parser.add_argument("z", type=float, help="Third coordinate to graph")
    args = parser.parse_args(argv)

    s = symbols('s')
    textplot(s**args.x,args.y,args.z)
```

[continued on next slide...]

ascii_graph/**ascii_graph.py**

```
import argparse
from sympy import symbols
from sympy.plotting import textplot
```

```
import conda.plugins
```

```
def ascii_graph(argv: list):
    parser = argparse.ArgumentParser("conda ascii-graph")
    parser.add_argument("x", type=float, help="First coordinate to graph")
    parser.add_argument("y", type=float, help="Second coordinate to graph")
    parser.add_argument("z", type=float, help="Third coordinate to graph")
    args = parser.parse_args(argv)

    s = symbols('s')
    textplot(s**args.x,args.y,args.z)
```

[continued on next slide...]

ascii_graph/**ascii_graph.py**

```
import argparse
from sympy import symbols
from sympy.plotting import textplot

import conda.plugins
```

The part that gets executed
as a plugin

```
def ascii_graph(argv: list):
    parser = argparse.ArgumentParser("conda ascii-graph")
    parser.add_argument("x", type=float, help="First coordinate to graph")
    parser.add_argument("y", type=float, help="Second coordinate to graph")
    parser.add_argument("z", type=float, help="Third coordinate to graph")
    args = parser.parse_args(argv)

    s = symbols('s')
    textplot(s**args.x,args.y,args.z)
```

[continued on next slide...]

ascii_graph/**ascii_graph.py**

[continued from previous slide...]

```
def ascii_graph(argv: list):
    parser = argparse.ArgumentParser("conda ascii-graph")
    parser.add_argument("x", type=float, help="First coordinate to graph")
    parser.add_argument("y", type=float, help="Second coordinate to graph")
    parser.add_argument("z", type=float, help="Third coordinate to graph")
    args = parser.parse_args(argv)
    s = symbols('s')
    textplot(s**args.x,args.y,args.z)
```

@conda.plugins.hookimpl

```
def conda_subcommands():
    yield conda.plugins.CondaSubcommand(
        name="ascii-graph",
        summary="A subcommand that takes three coordinates and prints out an ascii graph",
        action=ascii_graph,
    )
```

The `@hookimpl` decorator for marking the implementation of this subcommand plugin

ascii_graph/**ascii_graph.py**

[continued from previous slide...]

```
def ascii_graph(argv: list):  
    parser = argparse.ArgumentParser("conda ascii-graph")  
    parser.add_argument("x", type=float, help="First coordinate to graph")  
    parser.add_argument("y", type=float, help="Second coordinate to graph")  
    parser.add_argument("z", type=float, help="Third coordinate to graph")  
    args = parser.parse_args(argv)  
    s = symbols('s')  
    textplot(s**args.x,args.y,args.z)
```

```
@conda.plugins.hookimpl  
def conda_subcommands():
```

```
    yield conda.plugins.CondaSubcommand(  
        name="ascii-graph",  
        summary="A subcommand that takes three coordinates and prints out an ascii graph",  
        action=ascii_graph,  
    )
```

This function with the hook implementation decorator must have the same name as the function decorated by the corresponding hook specification!

conda/plugins/**hookspec.py**

```
import pluggy

from .types import CondaSolver, CondaSubcommand, CondaVirtualPackage

spec_name = "conda"
hookspec = pluggy.HookspecMarker(spec_name)
hookimpl = pluggy.HookimplMarker(spec_name)

class CondaSpecs:
    """
    The conda plugin hookspecs, to be used by developers.
    """
    @hookspec
    def conda_subcommands(self) -> Iterable[CondaSubcommand]:
        """
        Register external subcommands in conda.
        :return: An iterable of subcommand entries.
        """
```

ascii_graph/**ascii_graph.py**

[continued from previous slide...]

```
def ascii_graph(argv: list):
    parser = argparse.ArgumentParser("conda ascii-graph")
    parser.add_argument("x", type=float, help="First coordinate to graph")
    parser.add_argument("y", type=float, help="Second coordinate to graph")
    parser.add_argument("z", type=float, help="Third coordinate to graph")
    args = parser.parse_args(argv)
    s = symbols('s')
    textplot(s**args.x,args.y,args.z)

@conda.plugins.hookimpl
def conda_subcommands():
    yield conda.plugins.CondaSubcommand(
        name="ascii-graph",
        summary="A subcommand that takes three coordinates and prints out an ascii graph",
        action=ascii_graph,
    )
```

The ASCII Graph plugin project structure

```
$ tree ascii_graph/
```

```
└─ ascii_graph
   ├── ascii_graph.py
   └─ pyproject.toml
```

The ASCII Graph plugin project structure

```
$ tree ascii_graph/
```

```
├── ascii_graph
│   ├── ascii_graph.py
│   └── pyproject.toml
```

ascii_graph/**pyproject.toml**

```
[build-system]
requires = ["setuptools>=61.0", "setuptools-scm"]
build-backend = "setuptools.build_meta"
```

```
[project]
name = "ascii-graph"
version = "1.0"
description = "My ascii graph subcommand plugin"
requires-python = ">=3.7"
dependencies = ["conda", "sympy"]
```

```
[tools.setuptools]
py_modules=["ascii_graph"]
```

```
[project.entry-points.conda]
ascii-graph = "ascii_graph"
```

ascii_graph/**pyproject.toml**

```
[build-system]
requires = ["setuptools>=61.0", "setuptools-scm"]
build-backend = "setuptools.build_meta"
```

```
[project]
name = "ascii-graph"
version = "1.0"
description = "My ascii graph subcommand plugin"
```

```
requires-python = ">=3.7"
dependencies = ["conda", "sympy"]
```

```
[tools.setuptools]
py_modules=["ascii_graph"]
```

```
[project.entry-points.conda]
ascii-graph = "ascii_graph"
```

The plugin's requirements
and dependencies

ascii_graph/**pyproject.toml**

```
[build-system]
requires = ["setuptools>=61.0", "setuptools-scm"]
build-backend = "setuptools.build_meta"
```

```
[project]
name = "ascii-graph"
version = "1.0"
description = "My ascii graph subcommand plugin"
requires-python = ">=3.7"
dependencies = ["conda", "sympy"]
```

```
[tools.setuptools]
py_modules=["ascii_graph"]
```

```
[project.entry-points.conda]
ascii-graph = "ascii_graph"
```

Plugin Installation (in development mode)

```
$ pip install -e .
```

[lots o' output]

Successfully built ascii-graph

Installing collected packages: mpmath, sympy, ascii-graph

Successfully installed ascii-graph-1.0 mpmath-1.2.1 sympy-1.11.1

Plugin Installation (in development mode)

```
$ pip install -e .
```

[lots o' output]

Successfully built ascii-graph

Installing collected packages: mpmath, sympy, ascii-graph

Successfully installed **ascii-graph-1.0 mpmath-1.2.1 sympy-1.11.1**

Yay, everything the plugin needs is installed!

Invoking the subcommand plugin!

```
$ conda --help
```

```
usage: conda [-h] [-V] command ...
```

```
conda is a tool for managing and deploying applications, environments and packages.
```

```
Options:
```

```
~snip~
```

```
conda commands available from other packages:
```

```
  ascii-graph - A subcommand that takes three coordinates and prints out an ascii graph
```

```
conda commands available from other packages (legacy):
```

```
  content-trust
```

```
  env
```

```
  suggest
```

Invoking the subcommand plugin!

```
$ conda --help
usage: conda [-h] [-V] command ...
```

conda is a tool for managing and deploying applications, environments and packages.

Options:

~snip~

**Our new custom conda
subcommand is available for use!**

conda commands available from other packages:

ascii-graph - A subcommand that takes three coordinates and prints out an ascii graph

conda commands available from other packages (legacy):

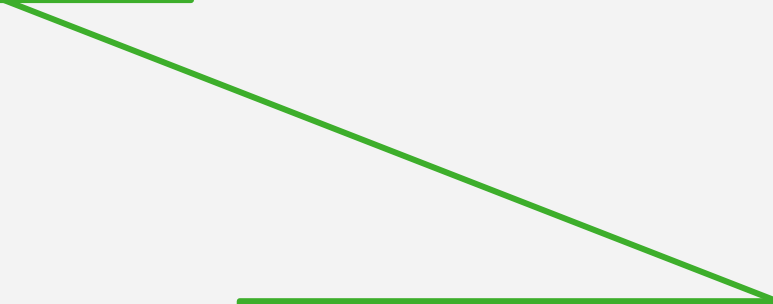
content-trust

env

suggest

Invoking the subcommand plugin!

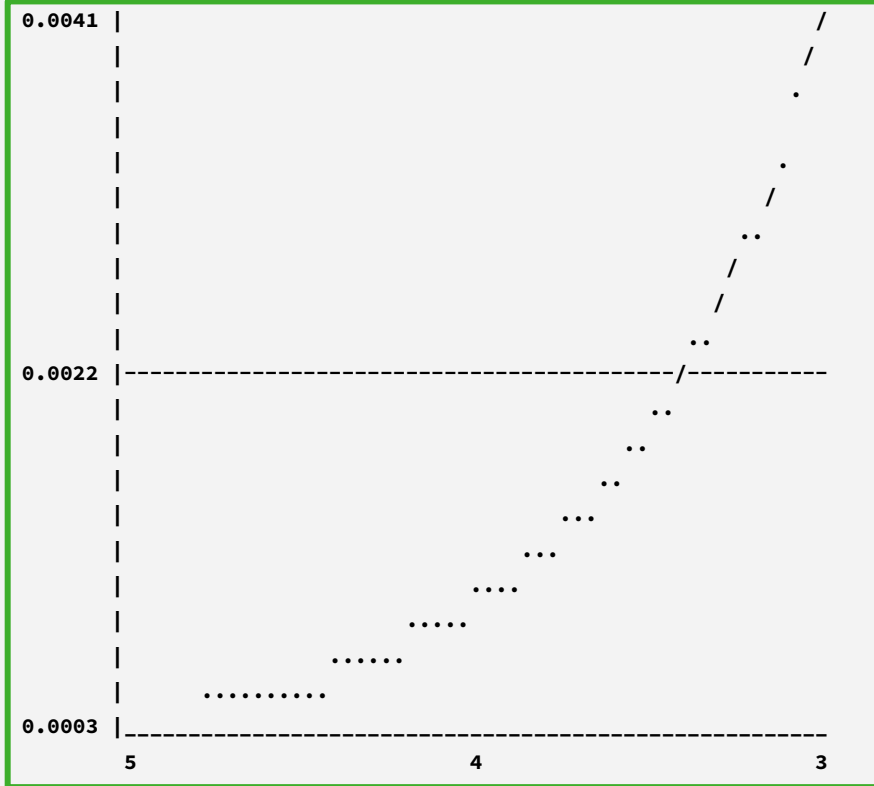
```
$ conda ascii-graph -5 5 3
```



We type: `conda ascii-graph -5 5 3`
and hit “enter” to get...

Invoking the subcommand plugin!

```
$ conda ascii-graph -5 5 3
```




Different plugins, same plugin API



What else?

Solvers 

Virtual
Packages 




Subcommands 




Request Handler 



Authentication 



Shell Integration 

Plugin use cases

- Vendor or client-specific extensions
- Code editor integrations
- Experimental features
- Refactoring
- ... and much more!

**Plugin
All the Things!**



Non-Python languages welcome!

- Write the plugin in Python
- Write a program in C/C++, Rust, Perl, Java, etc.
- Use a Foreign Function Interface (e.g., CFFI, Maturin)



Foreign
Function
Interface

Subcommand Tutorial



 bhenderson@anaconda.com / beeankha@gmail.com

 <https://github.com/beeankha>

 @bhenderson:matrix.org

 @bizonks

☰ README.md



Conda Plugin Tutorials: Subcommands: Python

In this tutorial, we will create a new conda subcommand written in Python that takes an input of three coordinates and prints out an ASCII graph.

To follow along with this guide, make sure you have the latest conda, conda-build, and pip installed:

```
(base) $ conda update conda conda-build pip
```

Project directory structure

Set up your working directory and files as shown below (or create a new repository using this [template](#)):

```
python/  
├─ recipe/  
│   └─ meta.yaml  
├─ ascii_graph.py  
└─ pyproject.toml (or setup.py)
```

The custom subcommand module

The following module implements a function, `ascii_graph` (where a set of three numbers gets converted into an ascii graph), and registers it with the plugin manager hook called `conda_subcommands` using the `@conda.plugins.hookimpl` decorator:

Presentation Materials



Plug Life Into Your Codebase

Making an established Python codebase pluggable

Bianca Henderson

April 22, 2023



pluggy Example

```
# example.py

import pluggy

hookspec = pluggy.HookspecMarker("example_for_pycon")
hookimpl = pluggy.HookimplMarker("example_for_pycon")

class ExampleSpec:
    """A hook specification namespace."""

    @hookspec
    def myhook(self, arg1, arg2):
        """My special little hook that you can customize."""
```

 bhenderson@anaconda.com / beeankha@gmail.com

 <https://github.com/beeankha>

 @bhenderson:matrix.org

 @bizonks

Conclusion

Thank you!

