Writing Plugin Friendly Applications in Python

By Travis Hathaway

Who am I?

Python Programer 🐍

GIS Enthusiast 🌎

Social Scientist 🏠

Guitarist/Musician 🎸

Where am I?

Website:
travishathaway.com

GitHub:
github.com/travishathaway

LinkedIn:
linkedin.com/in/thath

SoundCloud:
soundcloud.com/travis-hathaway

Where do I work?

We are hiring! 🎉 🙌

Check out our website for more information:

https://www.anaconda.com/careers

Remember to tell them that Travis sent you! 😉

What we'll cover:

What is even a plugin?

How do they work?

Should my application be plugin friendly?

How do we do it in Python?

Also, we look at a real-world example

Conda is a package manager written in Python. The application is currently undergoing renovations to make it more plugin friendly. We'll take a look at how it's going!

What is even a plugin?

"plug-in, also called add-on or extension, is computer software that adds new functions to a host program without altering the host program itself." (Sterne, 2014)





Plugins in the wild

Here are just a few examples of how you use plugins everyday.

Should my application be plugin friendly?

Pros and cons

Pros ✅

- Can make your application highly adaptable
- Enables you to grow a community of developers writing plugins
- For internal projects, could be a great way to help organize teams

Cons ❌

- Security risks by opening application up to third parties
- Increases complexity of code and project
- With each new version of your application comes a risk of breaking existing plugins

How do they work?

The plugin mechanism

Host programs provide the following:

- A way for the **plugin** to **register** itself
- A protocol to use for **exchanging data**

The **host** program operates independently of the **plugin**

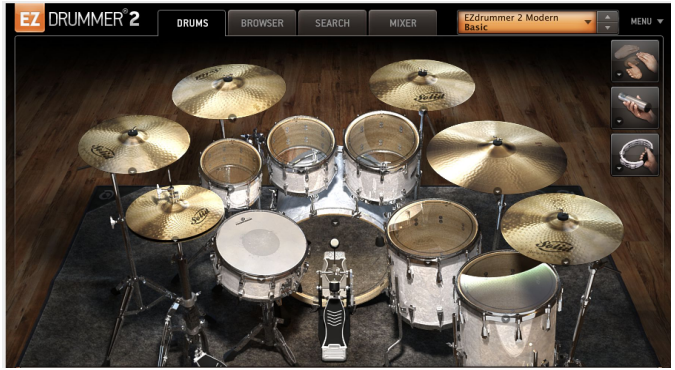
Plugins do not typically operate independently of the host program

(Wikipedia, 2023)

Closer look: VST

"**Virtual Studio Instrument (VST)** is an audio plug-in software interface that integrates software synthesizers and effects units into digital audio workstations."
(Wikipedia, 2023)

Virtual instruments like drums



Effects like reverb

How do VSTs work?

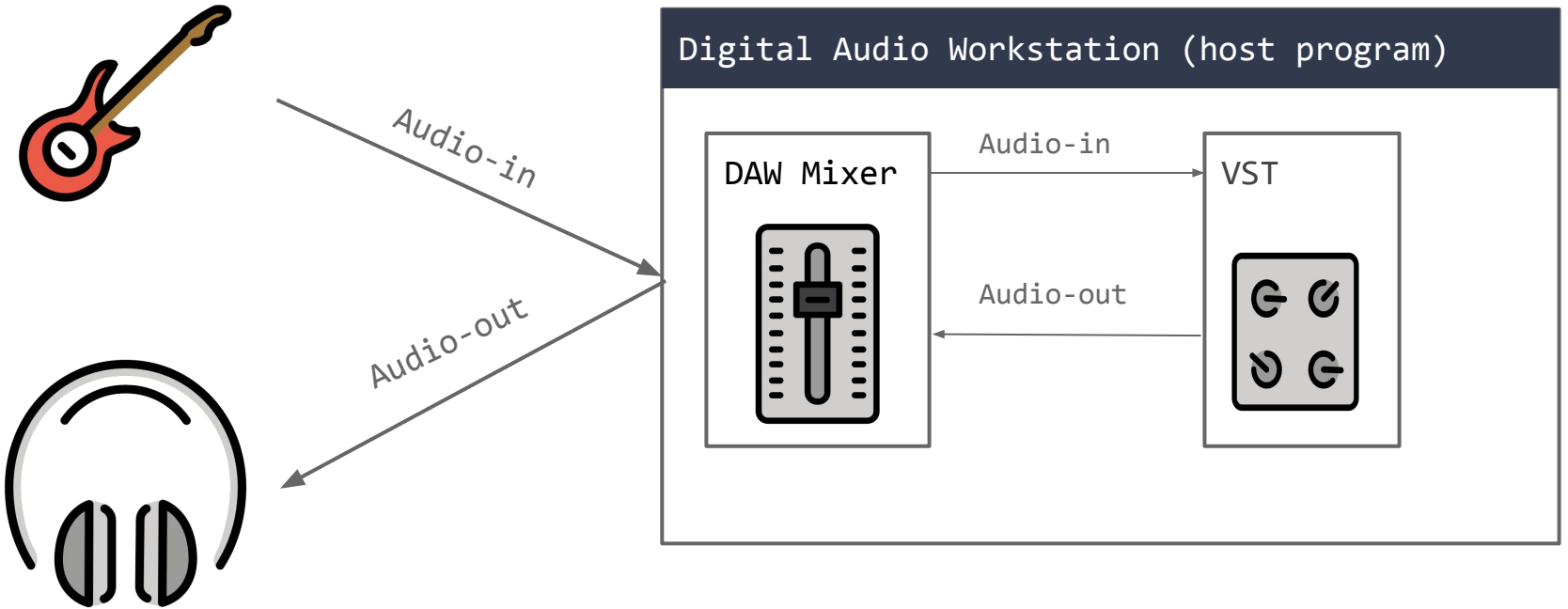
Registering

Applications which use VSTs typically have their users place all VSTs they want to use in a folder where this can be discovered.

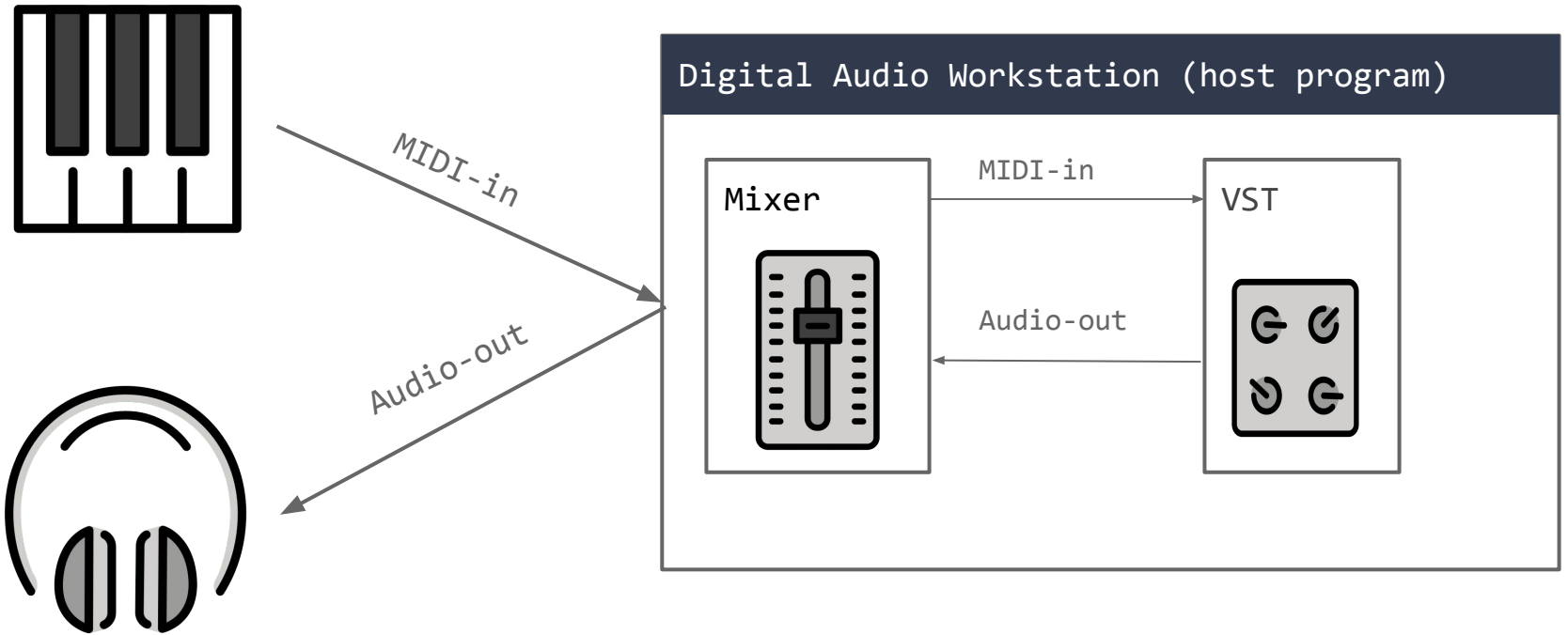
Data Exchange Protocol

To exchange data, VSTs primarily rely on two protocols: raw audio signals and MIDI

Data exchange protocol: raw audio



Data exchange protocol: MIDI



What’s going on underneath the surface?

Plugin Hooks

Host applications must expose an API to use for their **plugins**

One way they do this is by exposing so-called “**hooks**”. These hooks allow the plugin to be invoked for certain events or to extend functionality or appearance.

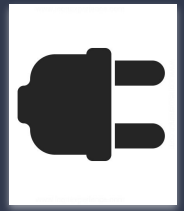
What hooks should a DAW* have?

`audio_hook`	Called when the DAW wants to **send** and **receive** an **audio** signal to the VST
`midi_hook`	Called when the DAW wants to **send** and **receive** an **MIDI** signal to the VST

*DAW = Digital Audio Workstation

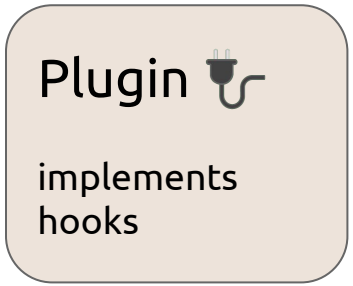
How do we do this in Python?

Introducing: Pluggy



Pluggy is a library that provides a system the that **host** program can use to expose **hooks** to the **plugin.**

Pluggy lies at the core of “pytest” a very popular library for writing tests in Python.



Let’s create a plugin

1. Create the hook
2. Implement a default
3. Use the hook in our application

Project structure for “fancy_print”

```
$ tree .
.
├── README.md
├── fancy_print
│   ├── __init__.py
│   ├── hooks.py
│   └── main.py
└── pyproject.toml
```

Creating hooks

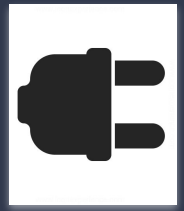
```python
# hooks.py
import pluggy

PROJECT_NAME = "fancy_print"

hookspec = pluggy.HookspecMarker(PROJECT_NAME)

class FancyPrintHookSpec:

    @hookspec
    def fancy_print_pad_char(self) -> str:
        """Used to override fancy print padding character"""
```

Creating hooks

```python
# hooks.py
import pluggy

PROJECT_NAME = "fancy_print"

hookspec = pluggy.HookspecMarker(PROJECT_NAME)

class FancyPrintHookSpec:

    @hookspec
    def fancy_print_pad_char(self) -> str:
        """Used to override fancy print padding character"""
```

Every pluggy application needs a hookspec object to register your hooks

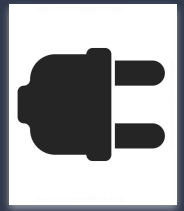
Creating hooks



```python
# hooks.py
import pluggy

PROJECT_NAME = "fancy_print"

hookspec = pluggy.HookspecMarker(PROJECT_NAME)

class FancyPrintHookSpec:

    @hookspec
    def fancy_print_pad_char(self) -> str:
        """Used to override fancy print padding character"""
```

To keep our hooks organized, we put them in a single class and register them individually with the hookspec decorator.

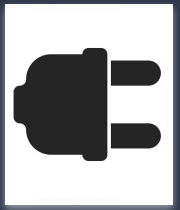
Creating hooks



```python
# hooks.py
import pluggy

PROJECT_NAME = "fancy_print"

hookspec = pluggy.HookspecMarker(PROJECT_NAME)

class FancyPrintHookSpec:

    @hookspec
    def fancy_print_pad_char(self) -> str:
        """Used to override fancy print padding character"""
```

Using type hints, we establish the specifics of our data exchange protocol. Our expects only strings to be returned from this hook.

Implementing a default



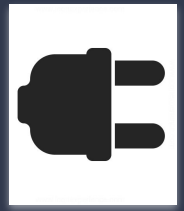
```python
# hooks.py
hookimpl = pluggy.HookimplMarker(PROJECT_NAME)

@hookimpl
def fancy_print_pad_char() -> str:
    """
    Default implementation for this application
    """
    return "🍇"
```

Implementing a default

```python
# hooks.py
hookimpl = pluggy.HookimplMarker(PROJECT_NAME)

@hookimpl
def fancy_print_pad_char() -> str:
    """
    Default implementation for this application
    """
    return "🍇"
```

We use the hookimpl object to mark implementations of the hooks we have defined.

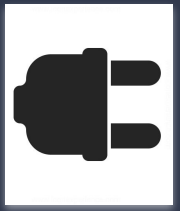
Implementing a default



```python
# hooks.py
hookimpl = pluggy.HookimplMarker(PROJECT_NAME)

@hookimpl
def fancy_print_pad_char() -> str:
    """
    Default implementation for this application
    """
    return "🍇"
```

It's then used as a decorator mark the implementations. These functions must have the same name as the hookspec!

Using it in our application



```python
# main.py
import pluggy

from fancy_print import hooks, PROJECT_NAME

def get_plugin_manager() -> pluggy.PluginManager:
    """
    Initializes and returns a `pluggy.PluginManager` object to use
    """
    plugin_manager = pluggy.PluginManager(PROJECT_NAME)
    plugin_manager.add_hookspecs(hooks.FancyPrintHookSpec)
    plugin_manager.register(hooks)
    plugin_manager.load_setuptools_entrypoints("fancy_print")

    return plugin_manager
```

Using it in our application

```python
# main.py
import pluggy

from fancy_print import hooks, PROJECT_NAME

def get_plugin_manager() -> pluggy.PluginManager:
    """
    Initializes and returns a `pluggy.PluginManager` object to use
    """
    plugin_manager = pluggy.PluginManager(PROJECT_NAME)
    plugin_manager.add_hookspecs(hooks.FancyPrintHookSpec)
    plugin_manager.register(hooks)
    plugin_manager.load_setuptools_entrypoints("fancy_print")

    return plugin_manager
```

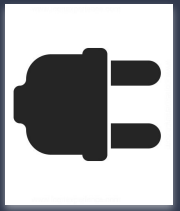
In our main application file we create a function to retrieve a PluginManager object

Using it in our application

```python
# main.py
import pluggy

from fancy_print import hooks, PROJECT_NAME

def get_plugin_manager() -> pluggy.PluginManager:
    """
    Initializes and returns a `pluggy.PluginManager` object to use
    """
    plugin_manager = pluggy.PluginManager(PROJECT_NAME)
    plugin_manager.add_hookspecs(hooks.FancyPrintHookSpec)
    plugin_manager.register(hooks)
    plugin_manager.load_setuptools_entrypoints("fancy_print")

    return plugin_manager
```

We create the PluginManager object and then register the hook specs and our default implementation

Using it in our application



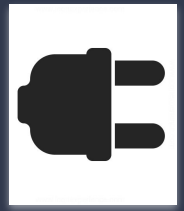
```python
# main.py
import pluggy

from fancy_print import hooks, PROJECT_NAME

def get_plugin_manager() -> pluggy.PluginManager:
    """
    Initializes and returns a `pluggy.PluginManager` object to use
    """
    plugin_manager = pluggy.PluginManager(PROJECT_NAME)
    plugin_manager.add_hookspecs(hooks.FancyPrintHookSpec)
    plugin_manager.register(hooks)
    plugin_manager.load_setuptools_entrypoints("fancy_print")

    return plugin_manager
```

This step allows us to register all plugins found as setuptools entrypoints. This is where all the “external” plugins are loaded.

Using it in our application

```python
# main.py

def main() -> None:
    """
    Main entry point for the program
    """
    plugin_manager = get_plugin_manager()

    pad_char = plugin_manager.hook.fancy_print_pad_char()
    pad_chars = ''.join(pad_char * 5)

    print(f"{pad_chars}\n\nso fancy!!!\n\n{pad_chars}")
```

Using it in our application



```python
# main.py

def main() -> None:
    """
    Main entry point for the program
    """
    plugin_manager = get_plugin_manager()

    pad_char = plugin_manager.hook.fancy_print_pad_char()
    pad_chars = ''.join(pad_char * 5)

    print(f"{pad_chars}\n\nso fancy!!!\n\n{pad_chars}")
```

This is how we invoke our plugin hooks in our main application

Quick recap

- Pluggy applications need hook definitions
- These hooks are marked with the **`HookspecMarker`** object
- Pluggy applications also need to define a **`HookimplMarker`** object
- This marker is used to register plugin hooks

A more complex example

Problem

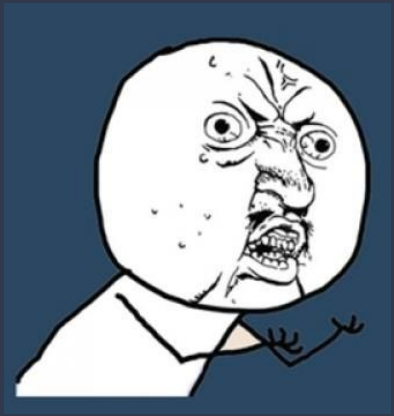
I want to create a command line application that can search for images on the internet.

I also want to support many image searching backends (e.g. Google Images, Unsplash, Imgur and more)

Solution

Write the application to be plugin friendly by allowing us to swap out the image search backend.

How make thing?!?!

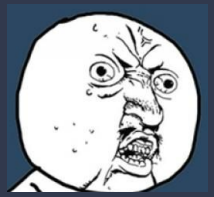


We need a way to **register** plugins that provide image search backends

We need a way to **configure** the image search backends at runtime

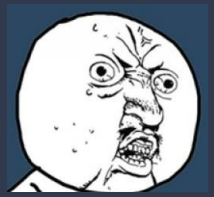
We need a clear **protocol** to follow when creating these image search backends (i.e. which methods to provide)

Image search backend: protocol



```python
from typing import Protocol

class ImageSearchBackendProtocol(Protocol):

    def search(self, query: str) -> list[dict[str, str]]:
        """
        Search method that must be implemented for protocol
        """
```

Image search backend: protocol

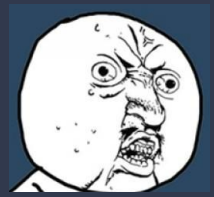


```python
from typing import Protocol
```

```python
class ImageSearchBackendProtocol(Protocol):

    def search(self, url: str) -> list[dict[str, str]]:
        """
        Search method that must be implemented for protocol
        """
```

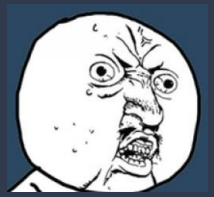
We create a protocol class that plugins will have to implement.

Image search backend: NamedTuple



```python
from typing import NamedTuple

class ImageSearchBackend(NamedTuple):
    """
    Represents all data needed to register a plugin for a
    new image search backend.
    """

    # Name of the plugin; this will be referenced in our configuration
    name: str

    # Class that implements our protocol class
    backend: type[ImageSearchBackendProtocol]
```

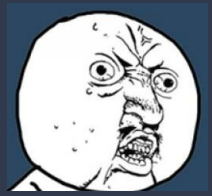
Image search backend: NamedTuple



```python
from typing import NamedTuple

class ImageSearchBackend(NamedTuple):
    """
    Represents all data needed to register a plugin for a
    new image search backend.
    """

    # Name of the plugin; this will be referenced in our configuration
    name: str

    # Class that implements our protocol class
    backend: type[ImageSearchBackendProtocol]
```

Next, we create a simple immutable data type by subclassing from NamedTuple. Our plugin hooks will return this object

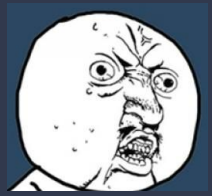
Image search backend: NamedTuple



```python
from typing import NamedTuple

class ImageSearchBackend(NamedTuple):
    """
    Represents all data needed to register a plugin for a
    new image search backend.
    """

    # Name of the plugin; this will be referenced in our configuration
    name: str

    # Class that implements our protocol class
    backend: type[ImageSearchBackendProtocol]
```

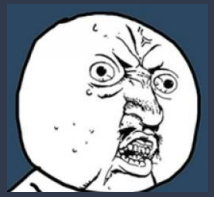
We use the protocol type we previously defined here to hint at the type of object that should be set here.

Image search backend: plugin code



```python
from fancy_print.hooks import hookimpl, ImageSearchBackend

class CustomBackend:
    def search(self, query: str) -> list[dict[str, str]]:
        # ... implementation goes here 😎

@hookimpl
def image_search_backend() -> ImageSearchBackend:
    """Hook responsible to register our image search backend"""
    return ImageSearchBackend(
        name="custom_backend",
        backend=CustomBackend
    )
```

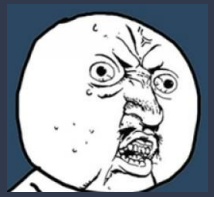
Image search backend: plugin code



```python
from fancy_print.hooks import hookimpl, ImageSearchBackend

class CustomBackend:
    def search(self, query: str) -> list[dict[str, str]]:
        # ... implementation goes here 😎

@hookimpl
def image_search_backend() -> ImageSearchBackend:
    """Hook responsible to register our image search backend"""
    return ImageSearchBackend(
        name="custom_backend",
        backend=CustomBackend
    )
```

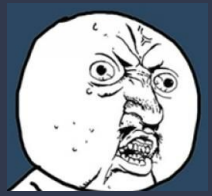
Here's the special type we defined below. If you have type hinting turned on, it will alert you if you have passed in an object that doesn't match the protocol.

Image search backend: plugin code



```
def main(query: str, backend_name: str):
    """
    Instantiates the plugin class and runs the image search
    """
    plugin_manager = get_plugin_manager()
    search_class = None

    for search_plugin in plugin_manager.hook.image_search_backend():
        if search_plugin.name == backend_name:
            search_class = search_plugin.backend

    if search_class is not None:
        search_obj = search_class()
        results = search_obj.search(query)
    # if not found, raise an exception or something...
```

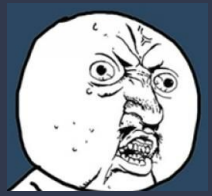
Image search backend: plugin code



```
def main(query: str, backend_name: str):
    """
    Instantiates the plugin class and runs the image search
    """
    plugin_manager = get_plugin_manager()
    search_class = None

    for search_plugin in plugin_manager.hook.image_search_backend():
        if search_plugin.name == backend_name:
            search_class = search_plugin.backend

    if search_class is not None:
        search_obj = search_class()
        results = search_obj.search(query)
    # if not found, raise an exception or something...
```

In our main function, we just have to find the correct hook. We use the name attribute from our special type.

Image search backend: plugin code



```
def main(query: str, backend_name: str):
    """
    Instantiates the plugin class and runs the image search
    """
    plugin_manager = get_plugin_manager()
    search_class = None

    for search_plugin in plugin_manager.hook.image_search_backend():
        if search_plugin.name == backend_name:
            search_class = search_plugin.backend

    if search_class is not None:
        search_obj = search_class()
        results = search_obj.search(query)
    # if not found, raise an exception or something...
```

Because we know the plugin provided search class conforms to our protocol, we can be using it fearlessly!

Want even more code to look at?

I recently created an example project to explore some of these ideas while also trying make it a starter project.

It's called "**latz**" and you can find it on my **GitHub** account:

https://github.com/travishathaway/latz

There's lots of documentation!

What about this conda stuff you mentioned earlier?

Ahh, yes, that…

Well, the interesting thing is that you've already seen how we setup our plugin hooks up!

The same pattern I just showed you all for my image search backend example application is also roughly how this is laid out in conda.

But wait… What is even…?

What is even conda?

Conda is a package manager supporting all major operating systems written in Python

It was originally developed to ease the pain of installing many of the Python libraries associated with scientific analysis

The project is now over 10 years old

Currently over 30 million active users

Why do we want to make it plugin friendly?

Many of the reasons mentioned before, but primarily to empower the community of users to begin contributing useful bits of functionality to the core of conda.

We also want to use plugins as a tool to encourage innovation and allow our conda users to better customize the tools to their specific needs.

But... how is it done?

Currently identifying what qualifies as a good candidate for plugin hooks

Potential future plugin hooks:

- CEP: Conda Generic Plugin Hooks
- CEP: Conda Fetch Plugin Hook

Current plugin hooks:

- Solver
- Subcommand
- Virtual Package

Concluding thoughts

Plugins can be a great way to organize your application, especially if you want to enable collaboration across organizations or large groups of people

Carefully plan the parts you expose to be plugin hooks and ensure you have use-cases to back them up!

Document your code! Make sure you have a clean house before inviting others in 🧹 🏠