Project name not set

Greg Caporaso and Evan Bolyen

Oct 31, 2024

CONTENTS

Your guide for writing, testing, and distributing QIIME 2 plugins, interfaces, and documentation.

Setting up your development environment

If you just want to find instructions for creating your QIIME 2 development environment, see *Set up your development environment*.

Development status of this content

Developing with QIIME 2 remains in very active development, and as a result some URLs may change. It should be getting more complete by the day.

The canonical URL for this project is now https://develop.qiime2.org.

The "old developer documentation", which was previously hosted at `https://dev.qiime2.org`, is now deprecated. All content that is still relevant has been ported from that documentation to *Developing with QIIME 2*. If you want to access that archival content, you can find it in the project's GitHub repository.

The *Tutorial: A step-by-step guide to building your first QIIME 2 plugin* chapter is where the focus is for the near future, though all of the *Plugin Development* chapters have useful and up-to-date content in them. You'll also find content in *Explanations* and various other chapters throughout, but those are currently less thorough and generally need some updates. Please let us know if you find anything that is inaccurate or outdated.

Developing **w**ith **QIIME 2** (DWQ2) is split into multiple *Parts* covering topics in QIIME 2 development, including *Plugin Development*, *Interface Development*, and *Docs Development*. You do not need to read all of these parts to develop with QIIME 2. If you are interested in creating plugins, then the only part you need to concern yourself with is *Plugin Development*. Similarly, if you want to build an interface, you only need *Interface Development*. Other parts, such as *Framework Development* and *Distribution Development*, are currently targeted primarily for the development team in the Caporaso Lab. *Docs Development* is slated for a full re-write as we *adapt our approach to user documentation*.

The content in each part of this book is organized under the Diátaxis framework [1]. This means that you can expect *Chapters* containing *Tutorials*, *How-To-Guides*, *Explanations*, and *References* in each part. Each serves a different goal for the reader:

Chapter	Purpose
Tutorials	Provide a guided exploration of a topic for **learning**.
How To Guides	Provide step-by-step instructions on how to **accomplish specific goals**.
Explanations	Provide a discussion intended to aid in **understanding** a specific topic.
References	Provide specific **information** (e.g., an API reference).

Chapters are generally broken up into *Sections*. For example, the *Tutorial: A step-by-step guide to building your first QIIME 2 plugin* chapter works through building a new QIIME 2 plugin from scratch. It does this in a series of sections that focus on different plugin components.

Acknowledgements

The authors would like to thank those who have contributed to the writing of *Developing with QIIME 2*.

The template plugin used in *Tutorial: A step-by-step guide to building your first QIIME 2 plugin* was derived from @misialq's plugin template.

Getting Help

For the most up-to-date information on how to get help with QIIME 2, as a user or developer, see here.

Contributing

To get information on contributing to *Developing with QIIME 2*, see *Developer documentation*.

Funding

This work was funded in part by NIH National Cancer Institute Informatics Technology for Cancer Research grant 1U24CA248454-01, and by grant DAF2019-207342 from the Chan Zuckerberg Initiative (CZI) DAF, an advised fund of Silicon Valley Community Foundation (CZI grant DOI: 10.37921/862772dbrrej; funder DOI 10.13039/100014989).

This book is built with MyST Markdown and Jupyter Book, which are supported in part with funding from the Alfred P. Sloan Foundation.

Initial support for the development of QIIME 2 was provided through a grant from the National Science Foundation.

License

Part I

Plugins

CHAPTER

ONE

PLUGIN DEVELOPMENT

1.1 Tutorial: A step-by-step guide to building your first QIIME 2 plugin

Note

My approach to writing *Developing with QIIME 2* is to publish early, and to publish often. This is well captured in this quote by Daniele Procida from Diátaxis:

> It's natural to want to complete large tranches of work before you publish them, so that you have something substantial to show each time. Avoid this temptation - every step in the right direction is worth publishing immediately. (source)

One caveat that I'll add, since *Developing with QIIME 2* is about developing software while Diátaxis is about developing documentation, is that functional code should only be "published" (e.g., applied in real world applications or shared with others who may do that) **after** it has been sufficiently tested. I'll come back to software testing shortly.

This tutorial is a work-in-progress.

This tutorial will walk step-by-step through building a first QIIME 2 plugin, and is intended to be read from beginning to end.

The plugin you'll create will provide support for some of the most fundamental algorithms in bioinformatics, and will ultimately contain a mix of QIIME 2 *methods*, *visualizers*, and *pipelines*. You'll learn to define new *artifact classes*, which let you expand QIIME 2 in new directions, and how artifact classes relate to *formats*, *transformers*, *semantic types*. You'll add parallel computing support to a *method* that is initially implemented without parallel support. You'll develop executable usage examples, so your users can learn how to apply your plugin, and you can receive automated notifications if changes you (or others) make to your code are backward incompatible, necessitating changes to the documentation. And, as with all QIIME 2 plugins, the plugin you build will record data provenance when it's used, support *provenance replay*, and will be immediately accessible through multiple interfaces including *q2cli* and the *Python 3 API*. You'll also be able to run a few additional commands to generate *Galaxy* workflows for all of your plugin actions. Finally, you'll receive guidance on how to move forward to create your own QIIME 2 plugin as a next step.

Let's *get started*!

Note

If you're already comfortable with plugin development and are looking for instructions to achieve a specific task, you may find more targeted instructions in the *Plugin Development How-To Guides*.

1.1.1 Tutorial table of contents

1.1.2 Create your plugin from a template

The easiest way to create a new QIIME 2 plugin is using our Cookiecutter template, which can be found at https://github.com/caporaso-lab/cookiecutter-qiime2-plugin. Here we'll work through building your QIIME 2 plugin from this template.

Install the tools needed for templating your plugin

To start building your new plugin, first install cookiecutter using their installation instructions. (If you opt to install cookiecutter with `pipx`, which the cookiecutter developers recommend, you can find the `pipx` installation instructions here.)

Optionally initialize a git repository during plugin templating

If `git` is installed in your environment, at the end of the templating process, a new git repository will be initialized and a first commit will be made. This facilitates managing your plugin in version control, and is especially good practice if you are templating a plugin that you ultimately plan to distribute to others.

You can check if you have git installed by running `git --version`. If you get a response with a version number (something like `git version 2.44.0`), git is installed and a new local repository will be initialized. If you get a response suggesting that `git` is not installed, you can just continue and not have cookiecutter create a git repository for you, or you can install git and then continue.

The git repository that is created will be a local git repository, meaning that it only exists on your computer and won't be shared through a site like GitHub. If you'd like to learn how to share your plugins, see *Distribute plugins on GitHub*.

Run `cookiecutter` to create your plugin

Next, run `cookiecutter` to create your plugin from the template using the following command. If you used `pipx` to install `cookiecutter`, follow the instructions in the *pipx* tab - otherwise follow the instructions in the *Other* tab.

pipx

```
pipx run cookiecutter gh:caporaso-lab/cookiecutter-qiime2-plugin
```

Other

```
cookiecutter gh:caporaso-lab/cookiecutter-qiime2-plugin
```

During the plugin templating process, you'll be prompted for information on your new plugin. For the questions about the *Target distribution* and whether you're *targeting the stable or latest development QIIME 2 release*, use the default values unless you have a specific reason not to; these are the last two questions, as of this writing in May 2024. For all of the other questions, feel free to customize your plugin by providing whatever values you'd like.

The plugin I'm going to create will be called `q2-dwq2` (for *Developing with QIIME 2*). After you've answered all of the questions, your plugin should have been successfully created and be ready to be installed and used.

Note

If you'd like to learn more about the files that were created in this process, you can refer to *The structure of QIIME 2 plugin packages*. You don't need to know what all of these files are to continue the tutorial though, so you can also come back to that later.

Install and test your new plugin

After the plugin has been created, change into the top-level directory for the plugin. For me, that's `q2-dwq2/`. In that directory, you'll find a file called `README.md`, which has a section in it containing *Installation instructions*. Follow all of the installation instructions, and then follow the instructions in that file for testing and using your new plugin.

After completing all of those steps, you now have a QIIME 2 *deployment* on your computer that includes your new plugin. When you requested help text on your plugin (e.g., `qiime dwq2 --help`), you should have seen some of the information you provided when creating the plugin.

The template plugin includes a simple (and silly) action called `duplicate-table`, along with associated unit tests. This provides an example action and example unit tests. You'll ultimately want to delete this action, but for now let's use it to make sure everything is working as expected.

Call your plugin's `duplicate-table` action with the `--help` parameter (e.g., `qiime dwq2 duplicate-table --help`). You should see text that looks like the following:

```
Usage: qiime dwq2 duplicate-table [OPTIONS]

  Create a copy of a feature table with a new uuid. This is for demonstration
  purposes only. 

Inputs:
  --i-table ARTIFACT FeatureTable[Frequency]
                           The feature table to be duplicated.       [required]
Outputs:
  --o-new-table ARTIFACT FeatureTable[Frequency]
                           The duplicated feature table.             [required]
Miscellaneous:
  --output-dir PATH        Output unspecified results to a directory
  --verbose / --quiet      Display verbose output to stdout and/or stderr
                           during execution of this action. Or silence output
                           if execution is successful (silence is golden).
  --example-data PATH      Write example data and exit.
  --citations              Show citations and exit.
  --help                   Show this message and exit.
```

If you'd like to try the action out, you can call your `duplicate-table` action on any QIIME 2 `FeatureTable[Frequency]` artifact (e.g., you can download one from the QIIME 2 user documentation). Load your duplicated table with QIIME 2 View, and poke through its Provenance to see how data provenance is recorded for your plugin.

Congratulations - you've created a working QIIME 2 plugin from a template! If you'd like to learn QIIME 2 plugin development, in the next step of the tutorial we'll *Add a first (real) Method*. If you're already comfortable with QIIME 2 plugin development, you're all set to make this plugin your own. In either case, if you'd like to host your plugin in a GitHub repository, you can refer to *Distribute plugins on GitHub*.

Tip

You can see my code after following these steps by looking at the specific commit in my plugin repository on GitHub: https://github.com/caporaso-lab/q2-dwq2/commit/3465ea40b18ae15825411a5930cfd24016f5d872. My code may look a little different than yours as I may have been using an older version of the template plugin than you used - everything in the tutorial will still work the same though.

1.1.3 Add a first (real) Method

At the most basic level, a QIIME 2 *Action* is simply an annotation of a Python function that describes in detail the inputs and outputs to that function. Here we'll create a powerful action that illustrates this idea.

The type of action that we'll create is a *method*, meaning that it will take zero or more QIIME 2 *artifacts* as input, and it will generate one or more QIIME 2 artifacts as output.

tl;dr

The complete code that I developed to add this action to my plugin can be found here: https://github.com/caporaso-lab/q2-dwq2/commit/e54d7438d409453093cbcb4f2c06c100784afbe8.

(*What does "tl;dr" mean?*)

Pairwise sequence alignment

One of the most fundamental algorithms in bioinformatics is *pairwise sequence alignment*. *Pairwise sequence alignment* forms the basis of BLAST, many genome assemblers, phylogenetic inference from molecular sequence data, assigning taxonomy to environmental DNA sequences, and so much more. The first real action that we'll add to our plugin is a method that performs pairwise sequence alignment using the Needleman-Wunsch global pairwise alignment algorithm [2].

You don't need to understand how the NW algorithm works interally to implement our method, because we're going to work with a Python function that implements the algorithm for us. But, you'll need to understand what its input and outputs are to be able to annotate them, and you'll need to understand at a basic level what it does so that you can test that your code is working as expected.

If you do want to learn more about pairwise sequence alignment, the specific algorithm and implementation that we're going to work with here is presented in detail in the *Pairwise Sequence Alignment* chapter of *An Introduction to Applied Bioinformatics* [3]. Briefly:

> The goal of pairwise sequence alignment is, given two DNA, RNA, or protein sequences, to generate a hypothesis about which sequence positions derived from a common ancestral sequence position. [3]

Our method will take two DNA sequences as input. It will attempt to align like positions with each other, inserting gap (i.e., −) characters where it seems likely that insertion/deletion events have occurred over the course of evolution. The output will be a pairwise sequence alignment (or more briefly, an *alignment*), which is a special case of a multiple sequence alignment that contains exactly two sequences. Our method will also take a few *parameters* as input. These parameters include things like the score that should be assigned when a pair of positions contains matching characters (we'll call this `match_score`), or the score penalty that is incurred when a new gap is opened in the alignment (`gap_open_penalty`).

The scikit-bio library implements Needleman-Wunsch global pairwise alignment algorithm as `skbio.alignment.global_pairwise_align_nucleotide`. We're going to make this accessible through our plugin by writing a simple wrapper of this function.

Write a wrapper function

> **Note**
>
> In subsequent sections of the book, we'll apply test-driven development where we write unit tests for our functions before writing the functions themselves. That may seem counter-intuitive if you've never done it before, but it's a powerful way to develop software as it focuses you on defining the specifications for your code before writing the code.

Technically we don't need to write a wrapper function of `skbio.alignment.global_pairwise_align_nucleotide`, but rather we could register that function directly. But for our first pass at this action there are a couple of things we're going to need to do to get data from QIIME 2 into `skbio.alignment.global_pairwise_align_nucleotide`. Let's add this wrapper to our `q2-dwq2/q2_dwq2/_methods.py`. (Remember that my package name is `q2-dwq2`, and my module name is `q2_dwq2`. If yours are different, you'll need to adjust that relative file path accordingly.) The `_` at the beginning of `_methods.py` is convention that conveys that this is intended to be a private submodule: in other words, consummers of this code outside of the this Python package shouldn't access anything in this file directly.

Since the `duplicate_table` method that was defined in this plugin was only intended to get us started, remove it now, replacing all of the content of that file with the following code:

```
# ----------------------------------------------------------------------------
# Copyright (c) 2024, Greg Caporaso.
#
# Distributed under the terms of the Modified BSD License.
#
# The full license is in the file LICENSE, distributed with this software.
# ----------------------------------------------------------------------------

from skbio.alignment import global_pairwise_align_nucleotide, TabularMSA

from q2_types.feature_data import DNAIterator


def nw_align(seq1: DNAIterator,
             seq2: DNAIterator,
             gap_open_penalty: float = 5,
             gap_extend_penalty: float = 2,
             match_score: float = 1,
             mismatch_score: float = -2) -> TabularMSA:
    seq1 = next(iter(seq1))
    seq2 = next(iter(seq2))

    msa, _, _ = global_pairwise_align_nucleotide(
        seq1=seq1, seq2=seq2, gap_open_penalty=gap_open_penalty,
        gap_extend_penalty=gap_extend_penalty, match_score=match_score,
        mismatch_score=mismatch_score
    )

    return msa
```

Here I defined a new function, `nw_align` (for *Needleman-Wunsch Alignment*). At its core, what this function is doing is passing some inputs through to `skbio.alignment.global_pairwise_align_nucleotide`. The aspects of this that might make this look different from what you typically see in a Python function definition are the type hints, which are not used by Python directly, but are intended to be used by other tools (like QIIME 2).

The type hints define the *data type* associated with each input and output from our function, and are one of the ways that we annotate a function so that QIIME 2 knows how to interact with it. Some of these are built-in types (in this case, we are providing four `float` values). The others, `DNAIterator` and `TabularMSA`, are defined in the q2-types QIIME 2 plugin and in scikit-bio, respectively. A `DNAIterator` is a Python object that enables iteration over a collection of zero or more `skbio.DNA` objects (which represent DNA sequences), and a `TabularMSA` object represents a **m**ultiple **s**equence **a**lignment. A little bit later we'll come back to how you decide what type hints to provide here.

The first couple of lines in this function are a little bit odd, and stem from the fact that (as of this writing) there isn't an existing QIIME 2 *artifact class* for individual DNA sequences, but rather only for collections of DNA sequences. We are therefore going to work-around this right now, and in a subsequent lesson we'll define our own QIIME 2 artifact class to represent a single DNA sequence. We need two sequences as input to pairwise sequence alignment (by definition), and we'll take those as inputs through the `seq1` and `seq2` parameters. These will come in as `DNAIterator` objects, and our work-around is that we read the first sequence from each of these two input sequence collections by getting the `next()` item from each collection, one time each, and reassigning them to the variables `seq1` and `seq2`.

Next, we call `skbio.alignment.global_pairwise_align_nucleotide`, passing in all of our inputs. `skbio.alignment.global_pairwise_align_nucleotide` returns three outputs. For now, we're only going to concern ourselves with the first: the multiple sequence alignment, which we'll store in a varriable called `msa`. By convention in Python, unused return values are assigned to a variable named _. Finally, we'll return the `msa` variable.

So for the most part, this works like a normal Python function. The unusual aspects are the type hints, and our workaround for getting the first sequence out of each of our input `DNAIterators`.

> **Note**
>
> Note that in defining this wrapper function, we haven't yet touched the concept of QIIME 2 *Artifacts*. The underlying functions that we register as methods or visualizers don't know anything about Artifacts. You can also use this function just as you would any other Python function - as mentioned above, Python itself ignores the type hints, so you could just consider these detailed documentation of the input and output types of your function.

Register the wrapper function as a plugin action

Now that we have a function, `nw_align`, that we want to register as an action in our plugin, let's do it.

Define a citation for this action

If the action that you're performing has a relevant citation, adding that during action registration will allow your users to discover what they should be citing when they use your action. This is a good way, for example, to ensure that your users know that they should be citing your work (and not just citing QIIME 2).

To associate a citation with our new action, the first thing we'll do is add the bibtex-formatted citation to `q2-dwq2/q2_dwq2/citations.bib`. Bibtex is a standard format recognizes by nearly all (or all) citation managers, including Paperpile and EndNote. You can generally export a bibtex citation from those tools, or alternatively get one from Google Scholar. The relevant citation for Needleman-Wunsch alignment is titled *A general method applicable to the search for similarities in the amino acid sequence of two proteins*, and was published in 1970. Find this citation using your favorite tool. If you use Google Scholar, search for the title of the article, and then identify it in the search results (it should be the first one). As of this writing, you would next click "Cite" under the search result, and then click "Bibtex". That should bring you to a page that contains the following bibtex-formatted citation:

```
@article{needleman1970general,
  title={A general method applicable to the search for similarities in the amino acid
↪sequence of two proteins},
  author={Needleman, Saul B and Wunsch, Christian D},
  journal={Journal of molecular biology},
  volume={48},
  number={3},
  pages={443--453},
  year={1970},
  publisher={Elsevier}
}
```

I copied this and then pasted it at the bottom of the `q2-dwq2/q2_dwq2/citations.bib` file. I also changed the citation key on the first line of this bibtext record (`needleman1970general`) to `Needleman1970`. This is how we'll reference this citation when we associate it with our action, and my version of the key is just a little easier for me to remember.

Register the action in `plugin_setup.py`

Now we have what we need to register our function as a plugin method. By convention, this is done in `q2-dwq2/q2_dwq2/plugin_setup.py`, and that's what we'll do here. Start by removing the registration of the `duplicate_table` action, now that we removed the underlying function from the plugin.

To register a function as a method, you'll use `plugin.methods.register_function`, where `plugin` is the `qiime2.plugin.Plugin` action that is instantiated in this file. Add the following code to your `q2-dwq2/q2_dwq2/plugin_setup.py` file, and then we'll work through it line by line. Note that this won't work yet - we still need to add some `import` statements to the top of the file, but we'll add those as we work through the code where the imported functionality is used.

```
plugin.methods.register_function(
    function=nw_align,
    inputs={'seq1': FeatureData[Sequence],
            'seq2': FeatureData[Sequence]},
    parameters={
        'gap_open_penalty': Float % Range(0, None, inclusive_start=False),
        'gap_extend_penalty': Float % Range(0, None, inclusive_start=False),
        'match_score': Float % Range(0, None, inclusive_start=False),
        'mismatch_score': Float % Range(None, 0, inclusive_end=True)},
    outputs={'aligned_sequences': FeatureData[AlignedSequence]},
    input_descriptions={'seq1': 'The first sequence to align.',
                        'seq2': 'The second sequence to align.'},
    parameter_descriptions={
        'gap_open_penalty': ('The penalty incurred for opening a new gap. By '
                             'convention this is a positive number.'),
        'gap_extend_penalty': ('The penalty incurred for extending an existing '
                               'gap. By convention this is a positive number.'),
        'match_score': ('The score for matching characters at an alignment '
                        'position. By convention, this is a positive number.'),
        'mismatch_score': ('The score for mismatching characters at an '
                           'alignment position. By convention, this is a '
                           'negative number.')},
    output_descriptions={
        'aligned_sequences': 'The pairwise aligned sequences.'
    },
    name='Pairwise global sequence alignment.',
```

(continues on next page)

(continued from previous page)

```
    description=("Align two DNA sequences using Needleman-Wunsch (NW). "
                 "This is a Python implementation of NW, so it is very slow! "
                 "This action is for demonstration purposes only. "),
    citations=[citations['Needleman1970']]
)
```

First, we call `plugin.methods.register_function`. This function call takes a number of parameters, and you can find full detail by following the `[source]` link from the *`Plugin` API documentation*. Here's what each is:

- `function`: This is the Python function to be registered as a plugin action. We defined ours above as `nw_align`. If you add the import statement `from q2_dwq2._methods import nw_align` to the top of this file, you can provide `nw_align`.
- `inputs`: This is a Python `dict` mapping the variable names of the *inputs* to the plugin action to the artifact classes of the inputs. As mentioned above, QIIME 2 doesn't define an artifact class for a single DNA sequence, so we're going to use the type that is commonly used for defining collections of DNA sequences, and we'll just end up working with the first sequence in each input. The type we use here is `FeatureData[Sequence]`. *A little bit later* we'll come back to how you identify the artifact classes that should be assigned to your input, and how to define your own artifact class if there isn't already a relevant one. *Inputs* to QIIME 2 actions are data in the form of *Artifacts*, and these are different than *Parameters*. It is at this stage, when registering a function as an action, that this distinction is made. The artifact classes we're using here need to be imported from q2-types. To do this, add the line `from q2_types.feature_data import FeatureData, Sequence` to the top of the file.
- `parameters`: This is a Python `dict` mapping the names of parameters to their *Primitive Type*. This information is used to validate input provided by users of your plugin, but more importantly to allow QIIME 2 interfaces to determine how this information should be collected from a user. For example, in a graphical interface the value of a `Boolean` parameter could be collected from a user using a checkbox, while a `Float` parameter could be collected using a text field that only accepts numbers. Our four parameters are all `Floats`, and each has a `Range` that values must fall in. Import these primitive types for use here by adding the line `from qiime2.plugin import Float, Range` to the top of the file. The `gap_open_penalty` parameter, for example, is defined here as taking a floating point value greater than 0.
- `outputs`: This is a Python `dict` mapping the variable names of the *outputs* from the plugin action to their artifact classes. The type we use here is `FeatureData[AlignedSequence]`, representing a collection of aligned DNA sequences. A more appropriate type might represent a pair of aligned sequences specifically, rather than one or more aligned sequences which is what this artifact class implies, but again we'll come back to that a later. We'll need to import `AlignedSequence` here as well, which you can do by adding the line `from q2_types.feature_data import AlignedSequence` to the top of the file (or adding `AlignedSequence` to the imports you already added from `q2_types.feature_data`).
- `input_descriptions`, `parameter_descriptions`, and `output_descriptions`: These are Python `dicts` that provide descriptions of each input, parameter, and output, respectively, for use in help text through different interfaces.
- `name`: A brief name for this action. This shows up, for example, when listing the actions that are available in a plugin.
- `description`: A longer description of the action. This is generally presented to a user when they request more detail on an action (for example, by passing a `--help` parameter through a command line interface).
- `citations`: A list of the citations that should be associated with this action. Earlier in the `plugin_setup.py` file we instantiated a `citations` lookup, and we can now use that to associate the citation we added to `citations.bib` with this action.

After adding this code and the corresponding `import` statements to your `plugin_setup.py`, you should be ready to try this action out.

My `import` statements now look like the following:

```
from qiime2.plugin import Citations, Plugin, Float, Range
from q2_types.feature_data import FeatureData, Sequence, AlignedSequence
from q2_dwq2 import __version__
from q2_dwq2._methods import nw_align
```

Calling the action with q2cli and the Python 3 API

Activate your development environment and run `qiime dev refresh-cache`. If your code doesn't have any syntax errors, and you addressed all of the additions described in this document, you should then be able to run `qiime dwq2 --help` (replacing `dwq2` with your plugin's name, if it's different), and see your new `nw-align` action show up in the list of actions associated with your plugin. It should look something like this:

> **Note**
>
> When I present command line calls and their output, I'll use `$` to indicate the command prompt.

```
$ qiime dwq2 --help
Usage: qiime dwq2 [OPTIONS] COMMAND [ARGS]...

  Description: A prototype of a demonstration plugin for use by readers of
  *Developing with QIIME 2* (DWQ2).

  Plugin website: https://develop.qiime2.org/

  Getting user support: Please post to the QIIME 2 forum for help with this
  plugin: https://forum.qiime2.org

Options:
  --version            Show the version and exit.
  --example-data PATH  Write example data and exit.
  --citations          Show citations and exit.
  --help               Show this message and exit.

Commands:
  nw-align  Pairwise global sequence alignment.
```

If you call `qiime dwq2 nw-align --help`, you'll see the more detailed help text for the `nw-align` action. It should look something like this:

```
$ qiime dwq2 nw-align --help
Usage: qiime dwq2 nw-align [OPTIONS]

  Align two DNA sequences using Needleman-Wunsch (NW). This is a Python
  implementation of NW, so it is very slow! This action is for demonstration
  purposes only. 

Inputs:
```

(continues on next page)

(continued from previous page)

```
  --i-seq1 ARTIFACT FeatureData[Sequence]
                           The first sequence to align.              [required]
  --i-seq2 ARTIFACT FeatureData[Sequence]
                           The second sequence to align.             [required]
Parameters:
  --p-gap-open-penalty NUMBER Range(0, None, inclusive_start=False)
                           The penalty incurred for opening a new gap. By
                           convention this is a positive number.   [default: 5]
  --p-gap-extend-penalty NUMBER Range(0, None, inclusive_start=False)
                           The penalty incurred for extending an existing gap.
                           By convention this is a positive number.
                                                                   [default: 2]
  --p-match-score NUMBER Range(0, None, inclusive_start=False)
                           The score for matching characters at an alignment
                           position. By convention, this is a positive number.
                                                                   [default: 1]
  --p-mismatch-score NUMBER Range(None, 0, inclusive_end=True)
                           The score for mismatching characters at an
                           alignment position. By convention, this is a
                           negative number.                      [default: -2]
Outputs:
  --o-aligned-sequences ARTIFACT FeatureData[AlignedSequence]
                           The pairwise aligned sequences.           [required]
Miscellaneous:
  --output-dir PATH        Output unspecified results to a directory
  --verbose / --quiet      Display verbose output to stdout and/or stderr
                           during execution of this action. Or silence output
                           if execution is successful (silence is golden).
  --example-data PATH      Write example data and exit.
  --citations              Show citations and exit.
  --help                   Show this message and exit.
```

Similarly, if you start an iPython session by calling `ipython` in your activated development environment, you can access the `nw_align` method through its Python 3 API as follows.

```
In [1]: import qiime2.plugins.dwq2
In [2]: qiime2.plugins.dwq2.actions.nw_align?
```

This call should produce the following help text:

```
Call signature:
qiime2.plugins.dwq2.actions.nw_align(
    seq1: FeatureData[Sequence],
    seq2: FeatureData[Sequence],
    gap_open_penalty: Float % Range(0, None, inclusive_start=False) = 5,
    gap_extend_penalty: Float % Range(0, None, inclusive_start=False) = 2,
    match_score: Float % Range(0, None, inclusive_start=False) = 1,
    mismatch_score: Float % Range(None, 0, inclusive_end=True) = -2,
) -> (FeatureData[AlignedSequence],)
Type:           Method
String form:    <method qiime2.plugins.dwq2.methods.nw_align>
File:           ~/miniconda3/envs/dwq2/lib/python3.8/site-packages/qiime2/sdk/action.
↪py
Docstring:      QIIME 2 Method
Call docstring:
Pairwise global sequence alignment.
```

(continues on next page)

(continued from previous page)

```
Align two DNA sequences using Needleman-Wunsch (NW). This is a Python
implementation of NW, so it is very slow! This action is for demonstration
purposes only. 

Parameters
----------
seq1 : FeatureData[Sequence]
    The first sequence to align.
seq2 : FeatureData[Sequence]
    The second sequence to align.
gap_open_penalty : Float % Range(0, None, inclusive_start=False), optional
    The penalty incurred for opening a new gap. By convention this is a
    positive number.
gap_extend_penalty : Float % Range(0, None, inclusive_start=False), optional
    The penalty incurred for extending an existing gap. By convention this
    is a positive number.
match_score : Float % Range(0, None, inclusive_start=False), optional
    The score for matching characters at an alignment position. By
    convention, this is a positive number.
mismatch_score : Float % Range(None, 0, inclusive_end=True), optional
    The score for mismatching characters at an alignment position. By
    convention, this is a negative number.

Returns
-------
aligned_sequences : FeatureData[AlignedSequence]
    The pairwise aligned sequences.
```

There are now two interfaces to your method, and you didn't have to write either of them — cool!

Take a minute to review both the command line and Python help text, and relate it to the parameters we set when we registered the action.

Write unit tests

Your code is *not ready for use* until you write unit tests, to ensure that it's doing what you expect. We'll write unit tests for `nw_align` in `q2_dwq2/tests/test_methods.py`. QIIME 2 provides a class, `TestPluginBase`, that facilitates unit testing plugins.

> **Warning**
>
> *Developing with QIIME 2* assumes that you have some background in software engineering. If writing unit tests or software testing in general are new to you, you should learn about these topics before developing software that you intend to use for "real" analysis. Small errors in code can have huge implications, including angry users, paper retractions, and even clinical errors that could impact someone's medical treatment.
>
> I highly recommend reading *The Pragmatic Programmer: Your Journey to Mastery* (20th Anniversary Edition) [4]. Topic 41 discusses software testing, but the whole book is worth reading if you're serious about developing high-quality software.

What to test and what not to test

When testing a QIIME 2 plugin, your goal is to confirm that the functionality that you developed works as expected. You can't test that *every* possible input produces its expected output, so instead you need to think about what tests will convince you that it's working across the range of inputs that would be expected. It's also a good idea to test that invalid input results in a failure, and ideally also provides an informative error message.

If you're simply wrapping a function, like we are here, you don't need to test the underlying function in detail as that should have been tested already in the library that provides that function. (If that's not the case, you should reconsider whether this function is the one that you want to use!)

You also don't need to test things such as whether your method works through q2cli, the Python 3 API, and Galaxy. That is functionality that you get for free when developing QIIME 2 plugins: the developers of the QIIME 2 framework and the other related tools have already tested this, and this should work as long as you're not adopting any *plugin development antipatterns*.

A first test of our plugin action

The following is a first test of our `nw_align` method. Remove the tests of `duplicate_table` that in this file, since we removed that action from the plugin.

> **Note**
>
> There are a couple of extra `imports` in here right now. We'll use those shortly.

```
from skbio.alignment import TabularMSA
from skbio.sequence import DNA

from qiime2.plugin.testing import TestPluginBase
from qiime2.plugin.util import transform
from q2_types.feature_data import DNAFASTAFormat, DNAIterator

from q2_dwq2._methods import nw_align


class NWAlignTests(TestPluginBase):
    package = 'q2_dwq2.tests'

    def test_simple1(self):
        # test alignment of a pair of sequences
        sequence1 = DNA('AAAAAAAAGGTGGCCTTTTTTTT')
        sequence1 = DNAIterator([sequence1])
        sequence2 = DNA('AAAAAAAAGGGGCCTTTTTTTT')
        sequence2 = DNAIterator([sequence2])
        observed = nw_align(sequence1, sequence2)

        aligned_sequence1 = DNA('AAAAAAAAGGTGGCCTTTTTTTT')
        aligned_sequence2 = DNA('AAAAAAAAGG-GGCCTTTTTTTT')
        expected = TabularMSA([aligned_sequence1, aligned_sequence2])
```

(continues on next page)

(continued from previous page)

```
        self.assertEqual(observed, expected)
```

First, we import some functions and classes that we'll use in our tests. Then, we define a class, `NWAlignTests`, that inherits from `TestPluginBase`, a class used to facilitate testing of QIIME 2 plugins. `TestPluginBase` has you define a class variable, `package`, defining the submodule that we're working in - we'll come back to how this is used shortly. Finally, we're ready to start defining unit tests.

The first test that I typically define for a function is a test of its default behavior on some very simple input where it's easy for me to determine what the expected outcome should be. In the example here, I'm creating two `skbio.DNA` sequence objects, turning them into `DNAIterators` (remember that that is what `nw_align` expects as input), and then calling `nw_align` on those two inputs. I call the return value from `nw_align` `observed`, because it is my observed output.

I very specifically chose the sequences here because I could tell based on my knowledge of Needleman-Wunsch alignment what the output would be: it's clear that the two sequences differ only in that there appears to have either been an insertion of a `T` character in `sequence1`, or a deletion of a `T` character in `sequence2` since the ancestral sequence. Alternatively, in a case like this, it's also fair game to generate the expected output by calling the underlying function (`skbio.alignment.global_pairwise_align_nucleotide`) directly. This is because we're not testing that `skbio.alignment.global_pairwise_align_nucleotide` does what it's supposed (again, we trust that it is, or we wouldn't be using it). We're only testing that our wrapper generates the output that we would expect from `skbio.alignment.global_pairwise_align_nucleotide`.

After getting my `observed` output, I define my `expected` output (i.e., what `nw_align` should return if it's working as expected). In this case, that's a pairwise alignment of `sequence1` and `sequence2` with a `-` character added where the insertion/deletion event is hypothesized to have occurred.

Finally, we compare our `observed` output to our `expected` output.

An important thing to note here is that I didn't need to load any data from file or use any QIIME 2 artifacts when testing my method. Because my method is just a Python function that I registered with my `Plugin`, I can provide input objects as I would to any other Python function. It is possible to store QIIME 2 Artifacts in the repository and load them for use in the tests, but that gets a little clunky so it's best avoided when possible. For example, you'll often want to test multiple minor variations on the input to test edge cases (i.e., boundary conditions). That's much easier to do if you're working with Python objects as input, rather than if you need to create a whole bunch of different QIIME 2 artifacts and store them in the repository. Storing artifacts in the repository to use as inputs in unit tests can also increase the repository size, and it's not straight-forward to compare how inputs have changed across different revisions of the code.

We can now run the test using `make test` on the command line from your `q2-dwq2` directory. This should look something like the following:

```
$ make test

...
q2_dwq2/tests/test_methods.py::NWAlignTests::test_simple1

...

==== 1 passed, 5 warnings in 0.17s ====
```

At the moment we're not concerned about the warnings that are being reported. We see from this output that we defined one test, and that one test passed. So we're ready to move on.

A second test of our action

All of that said, sometimes you do want to store data that you use in tests in files in the repository (for example, if they are large - in which case it's a pain to store the test data in your unit test Python files). The following unit test illustrates how this can be achieved.

```
    def test_simple2(self):
        # test alignment of a different pair of sequences
        # loaded from file this time, for demonstration purposes
        sequence1 = transform(
            self.get_data_path('seq-1.fasta'),
            from_type=DNAFASTAFormat,
            to_type=DNAIterator)
        sequence2 = transform(
            self.get_data_path('seq-2.fasta'),
            from_type=DNAFASTAFormat,
            to_type=DNAIterator)
        observed = nw_align(sequence1, sequence2)

        aligned_sequence1 = DNA('ACCGGTGGAACCGG-TAACACCCAC')
        aligned_sequence2 = DNA('ACCGGT--AACCGGTTAACACCCAC')
        expected = TabularMSA([aligned_sequence1, aligned_sequence2])

        self.assertNotEqual(observed, expected)
```

In this example, data is loaded from a file path that is relative to the `TestPluginBase.package` variable that we set above. In this case, the data needs to be transformed from a fasta file, which QIIME 2 represents as an instance of `q2-type`'s `DNAFASTAFormat` object to a `DNAIterator`, so it can be provided as input to `nw_align`. Here we use the `qiime2.plugin.util.transform` method to perform the transformation. After loading the files and transforming them, the test looks identical to the previous test case that we defined, except that the `expected` output is different, because the sequences we loaded differ from those used in the `test_simple1` method.

In order for this test to pass, you must actually have the two `.fasta` files that are being loaded in the submodule as specified here (i.e., you should have the files `q2-dwq2/q2_dwq2/tests/data/seq-1.fasta` and `q2-dwq2/q2_dwq2/tests/data/seq-2.fasta` in your Python package).

Save the following text in a new file, `q2-dwq2/q2_dwq2/tests/data/seq-1.fasta`.

```
>example-sequence-1
ACCGGTGGAACCGGTAACACCCAC
```

Save the following text in a new file, `q2-dwq2/q2_dwq2/tests/data/seq-2.fasta`.

```
>example-sequence-2
ACCGGTAACCGGTTAACACCCAC
```

While adding these files, also remove `q2-dwq2/q2_dwq2/tests/data/table-1.biom`, which we're not using any more since we removed the `duplicate_table` action and its tests, using the command:

```
rm q2_dwq2/tests/data/table-1.biom
```

Note

If you initialized a git repository for this plugin when you started working on it, you should instead remove the file with:

```
git rm q2_dwq2/tests/data/table-1.biom
```

After defining this test in your plugin, run the unit tests and confirm that you now have two passing tests.

A few additional tests

Because we want to test that our function generates the results that we would expect from `skbio.alignment.global_pairwise_align_nucleotide`, it's good to check that the parameter values that we provide impact the results in the expected ways. I did this with four additional unit tests, each focused on a different input parameter.

```
    def test_alt_match_score(self):
        s1 = DNA('AAAATTT')
        sequence1 = DNAIterator([s1])
        s2 = DNA('AAAAGGTTT')
        sequence2 = DNAIterator([s2])
        # call with default value for match score
        observed = nw_align(sequence1, sequence2)

        aligned_sequence1 = DNA('--AAAATTT')
        aligned_sequence2 = DNA('AAAAGGTTT')
        expected = TabularMSA([aligned_sequence1, aligned_sequence2])

        self.assertEqual(observed, expected)

        sequence1 = DNAIterator([s1])
        sequence2 = DNAIterator([s2])
        # call with non-default value for match_score
        observed = nw_align(sequence1, sequence2, match_score=10)

        # the following expected outcome was determined by calling
        # skbio.alignment.global_pairwise_align_nucleotide directly. the
        # goal isn't to test that the underlying library code (i.e.,
        # skbio.alignment.global_pairwise_align_nucleotide) is working, b/c
        # I trust that that is already tested (or I wouldn't use it). rather,
        # the goal is to test that my wrapper of it is working. in this case,
        # specifically, i'm testing that passing an alternative value for
        # match_score changes the output alignment
        aligned_sequence1 = DNA('AAAA--TTT')
        aligned_sequence2 = DNA('AAAAGGTTT')
        expected = TabularMSA([aligned_sequence1, aligned_sequence2])

        self.assertEqual(observed, expected)

    def test_alt_gap_open_penalty(self):
        s1 = DNA('AAAATTT')
        sequence1 = DNAIterator([s1])
        s2 = DNA('AAAAGGTTT')
        sequence2 = DNAIterator([s2])
        observed = nw_align(sequence1, sequence2, gap_open_penalty=0.01)

        aligned_sequence1 = DNA('AAAA-T-TT-')
        aligned_sequence2 = DNA('AAAAG-GTTT')
        expected = TabularMSA([aligned_sequence1, aligned_sequence2])

        self.assertEqual(observed, expected)
```

(continues on next page)

(continued from previous page)

```
        sequence1 = DNAIterator([s1])
        sequence2 = DNAIterator([s2])
        observed = nw_align(sequence1, sequence2)

        aligned_sequence1 = DNA('--AAAATTT')
        aligned_sequence2 = DNA('AAAAGGTTT')
        expected = TabularMSA([aligned_sequence1, aligned_sequence2])

        self.assertEqual(observed, expected)

    def test_alt_gap_extend_penalty(self):
        s1 = DNA('AAAATTT')
        sequence1 = DNAIterator([s1])
        s2 = DNA('AAAAGGTTT')
        sequence2 = DNAIterator([s2])
        observed = nw_align(sequence1, sequence2, gap_open_penalty=0.01)

        aligned_sequence1 = DNA('AAAA-T-TT-')
        aligned_sequence2 = DNA('AAAAG-GTTT')
        expected = TabularMSA([aligned_sequence1, aligned_sequence2])

        self.assertEqual(observed, expected)

        sequence1 = DNAIterator([s1])
        sequence2 = DNAIterator([s2])
        observed = nw_align(sequence1, sequence2, gap_open_penalty=0.01,
                            gap_extend_penalty=0.001)

        aligned_sequence1 = DNA('AAAA--TTT')
        aligned_sequence2 = DNA('AAAAGGTTT')
        expected = TabularMSA([aligned_sequence1, aligned_sequence2])

        self.assertEqual(observed, expected)

    def test_alt_mismatch_score(self):
        s1 = DNA('AAAATTT')
        sequence1 = DNAIterator([s1])
        s2 = DNA('AAAAGGTTT')
        sequence2 = DNAIterator([s2])
        observed = nw_align(sequence1, sequence2, gap_open_penalty=0.01)

        aligned_sequence1 = DNA('AAAA-T-TT-')
        aligned_sequence2 = DNA('AAAAG-GTTT')
        expected = TabularMSA([aligned_sequence1, aligned_sequence2])

        self.assertEqual(observed, expected)

        sequence1 = DNAIterator([s1])
        sequence2 = DNAIterator([s2])
        observed = nw_align(sequence1, sequence2, gap_open_penalty=0.1,
                            mismatch_score=-0.1)

        aligned_sequence1 = DNA('-AAA-ATTT')
        aligned_sequence2 = DNA('AAAAGGTTT')
        expected = TabularMSA([aligned_sequence1, aligned_sequence2])

        self.assertEqual(observed, expected)
```

Wrapping up testing

When these tests are all in place (or in the process of putting them in place), you can run them by calling `make test` on the command line. If everything is working as expected, you should see something like the following:

```
$ make test

...

q2_dwq2/tests/test_methods.py::NWAlignTests::test_alt_gap_extend_penalty
q2_dwq2/tests/test_methods.py::NWAlignTests::test_alt_gap_open_penalty
q2_dwq2/tests/test_methods.py::NWAlignTests::test_alt_match_score
q2_dwq2/tests/test_methods.py::NWAlignTests::test_alt_mismatch_score
q2_dwq2/tests/test_methods.py::NWAlignTests::test_simple1
q2_dwq2/tests/test_methods.py::NWAlignTests::test_simple2

...

==== 6 passed, 23 warnings in 1.17s ====
```

If your tests pass, and you can see the action on the command line, you should be in good shape so let's try running the method.

Trying the new action

To run the new action, you'll need two input files, each containing a DNA sequence to align. At this stage, your inputs need to be QIIME 2 artifacts. You should be able to do this with any `FeatureData[Sequence]` artifacts you have access to. If you don't have any, you can use the ones that we used in `test_simple2` by importing.

To do this, change to a temporary directory and copy the two files referenced in `test_simple2` to that directory. You should then be able to run the following commands to import those files:

```
qiime tools import --input-path seq-1.fasta --type "FeatureData[Sequence]" --output-
 ↪path seq-1.qza
qiime tools import --input-path seq-2.fasta --type "FeatureData[Sequence]" --output-
 ↪path seq-2.qza
```

Then, you can apply your new action to these two inputs as follows:

```
qiime dwq2 nw-align --i-seq1 seq-1.qza --i-seq2 seq-2.qza --o-aligned-sequences␣
 ↪aligned-seqs.qza
```

This should create a new output, `aligned-seqs.qza`. Since this is a method, it's generating a new artifact as an output. As always, artifacts aren't intended for human consumption, but rather to be used as input to other QIIME 2 actions or exported for use with other (non-QIIME 2) tools. If you want to take a peek at what's in there, you can export it:

```
qiime tools export --input-path aligned-seqs.qza --output-path aligned-seqs
```

Then, you can view the file contents as you would with any .fasta file. For example:

```
$ cat aligned-seqs/aligned-dna-sequences.fasta
>example-sequence-1
ACCGGTGGAACCGG-TAACACCCAC
>example-sequence-2
ACCGGT--AACCGGTTAACACCCAC
```

So there you have it - a first (real) action in our QIIME 2 plugin. ✓

As a next step, let's make this a little more user-friendly by *adding a* Visualizer that will let us look at the outcome of our pairwise alignment directly, without having to export it from QIIME 2.

An optional exercise

Now that you have this method working, try adding a method for local pairwise alignment of nucleotide sequences using the Smith-Waterman (SW) algorithm. scikit-bio provides an implementation of SW as `skbio.alignment.local_pairwise_align_nucleotide`. Don't forget to write your unit tests!

Throughout the next few chapters, additional exercises will build on this functionality.

1.1.4 Add a first Visualizer

In the last section we created a first method for our plugin which performed pairwise alignment of DNA sequences. We were able to run this to generate an alignment, but we didn't have any way to visualize the result without exporting it from QIIME 2. In this lesson we'll address that by adding a simple *Visualizer* to our plugin which takes a `TabularMSA` artifact as input (which is what our previous action generated as output), and generates a `Visualization` that we can review using QIIME 2 View.

tl;dr

The complete code that I developed to add this visualizer to my plugin can be found here: https://github.com/caporaso-lab/q2-dwq2/commit/1e802ea841ef40a40cfcdf53fca124061fcfccad.

Write the visualizer function

As with adding our alignment method, the first thing we'll do to define a new visualizer is write the underlying Python function. Visualizer functions, at a minimum, take an `output_dir` (directory where output files should be written) as a string as input, but they also generally take one or more QIIME 2 artifacts, as well as metadata, as input. Our action will take the sequence alignment artifact that we generated as input, in addition to the `output_dir` parameter.

The main function of a visualizer is to write some content to `output_dir`. This information will all be packaged as part of the output *Visualization* that QIIME 2 creates when the visualizer is called. When the visualization is viewed by a user using a QIIME 2 result viewer (such as QIIME 2 View), the viewer looks for an `index` file in the directory content created by the visualization, and presents that through the viewer. Most often, the `index` file is an HTML file (`index.html`), though technically other file types are possible. Visualizers are intentionally very flexible. As long as content can be packaged as HTML content and doesn't require a server, it should work ok in the visualizer.

Our task in writing this visualizer is to create some sort of useful, human-readable display of the alignment that is provided as input in an HTML file. Luckily, scikit-bio's `TabularMSA` object has a built-in function for creating a text-based human-readable alignment summary through its `__repr__` function. We'll use that here, even though the representation it creates is a little crude and it is only a summary rather than a display of the full alignment. (Since our goal here is just to write a vizualizer — not actually enable exploration of alignments — this will suffice for our purposes. I'll leave it as an exercise to you at the end of this section to expand on this visualization.)

Start by creating a new file, `_visualizers.py` in your module's top-level directory. For me, this file will be `q2-dwq2/q2_dwq2/_visualizers.py`. Add the following code to that file.

```
import os.path
```

(continues on next page)

(continued from previous page)

```
from skbio.alignment import TabularMSA


def summarize_alignment(output_dir: str, msa: TabularMSA) -> None:
    with open(os.path.join(output_dir, "index.html"), "w") as fh:
        fh.write(_html_template % repr(msa))


_html_template = """
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Alignment summary</title>
    <style>
        body {
            padding: 20px;
        }
        p.alignment {
            font-family: 'Courier New', Courier, monospace;
        }
    </style>
</head>
<body>
    <pre>
%s
    </pre>
</body>
</html>
"""
```

The majority of the code in this file is the `_html_template` variable, which sets up the HTML file and leaves a format specifier (`%s`) for us to insert a string of text. As templates like this get bigger or more complex, it often makes sense to store them in other files that are loaded by your visualizer code, but we'll keep the template in the same file for now to keep it simple.

The other bit of code in here is our viusalizer function. As described above, this takes an `output_dir` (always the first parameter to a visualizer) and our multiple sequence alignment as input, and doesn't `return` anything. As with all Python functions that get registered as plugin actions, our function employs type hints so that QIIME 2 knows what data type should be provided as input for each parameter. In our case, `output_dir` is provided as a string (`str`), `msa` is provided as a `skbio.alignment.TabularMSA`, and the return type is `None`, indicating no return value.

Our visualizer code is short and sweet. It opens a new file, `index.html` in `output_dir`. It then writes the html template to file, replacing the format specifier in the template with the `__repr__` of our `msa`.

That's it for the underlying visualizer function. Let's write a quick test of it before we hook it up to the plugin, to verify that the function works as expected.

Unit testing the visualizer function

I wrote my test of this visualizer in a new file, `q2-dwq2/q2_dwq2/tests/test_visualizers.py`. Create a file for your test in your plugin, and add the following content.

```
import os.path

from skbio.alignment import TabularMSA
from skbio.sequence import DNA

from qiime2.plugin.testing import TestPluginBase

from q2_dwq2._visualizers import summarize_alignment


class SummarizeAlignmentTests(TestPluginBase):
    package = 'q2_dwq2.tests'

    def test_simple1(self):
        aligned_sequence1 = DNA('AAAAAAAAGGTGGCCTTTTTTTT')
        aligned_sequence2 = DNA('AAAAAAAAGG-GGCCTTTTTTTT')
        msa = TabularMSA([aligned_sequence1, aligned_sequence2])

        with self.temp_dir as output_dir:
            summarize_alignment(output_dir, msa)

            with open(os.path.join(output_dir, 'index.html'), 'r') as fh:
                observed = fh.read()

            self.assertIn('AAAAAAAAGGTGGCCTTTTTTTT', observed)
            self.assertIn('AAAAAAAAGG-GGCCTTTTTTTT', observed)
```

Our test again uses `qiime2.plugin.testing.TestPluginBase`, which is good practice for all tests of your QIIME 2 plugin functions as it provides some convenient functionality. We then create a `TabularMSA`, similar to how we created our `expected` values in our `nw_align` tests in the previous section. Then, we use `TestPluginBase`'s `temp_dir` property as an output directory, and call our `summarize_alignment` function providing the `output_dir` and the alignment as input.

Determining what to test with a visualizer is always a little tricker, as we don't want to make the test more fragile than it needs to be. In this case, we do know what the entire HTML file is supposed to look like, so we could compare that as our `expected` to the `observed` value character by character and fail if any characters differ. That could get a bit clunky though, as a lot of that test would just be ensuring that Python correctly wrote a string (our `_html_template`) variable to file, and any changes to our `_html_template` would require changes to the test code as well. Instead, I'm just going to test that the expected aligned sequence strings were correctly written to file. I do that using `assertIn`, which in this case is checking that a given string is `in` another given string (i.e., the first string is a substring of the second string).

Save this file, and run the tests using:

```
make test
```

This will generate a bunch of output, but if everything is working as expected you should see a line like:

```
q2_dwq2/tests/test_visualizers.py .
```

in the output. This indicates that one test ran and passed (indicated by the single `.` character - you'll have one `.` per test that passed) from the `q2_dwq2/tests/test_visualizers.py` file.

If the test didn't pass, take a few minutes to figure out what went wrong, and re-run the test until it passes.

If you get stuck, refer to *the code that I wrote for this section*.

Register your Python function as a plugin action

The last step in defining a visualizer for your plugin is to register the function we just wrote as a visualizer. To do this, we'll go back to the `plugin_setup.py` file.

In that file, you'll first need to import your visualizer function so you can register it by adding the following line to the top of the file:

```
from q2_dwq2._visualizers import summarize_alignment
```

Then, you'll call `plugin.visualizers.register_function`, as follows:

```
plugin.visualizers.register_function(
    function=summarize_alignment,
    inputs={'msa': FeatureData[AlignedSequence]},
    input_descriptions={'msa': 'The multiple sequence alignment to summarize.'},
    parameters={},
    parameter_descriptions={},
    name='Summarize an alignment.',
    description='Summarize a multiple sequence alignment.',
)
```

This is very similar to how we registered our `nw_align` function, except that because this is a `visualizer`, we're calling `plugin.`**`visualizers`**`.register_function`, rather than `plugin.`**`methods`**`.register_function`. We still provide the `function` we want to register, `inputs` and `input_descriptions`, `parameters` and `parameter_descriptions`, a `name`, a `description`, and `citations` if we had any (but we don't here).

If you now save this file, and call `qiime dev refresh-cache`, you should be able to see and use your visualizer. Try out your new visualizer by providing one of the `FeatureData[AlignedSequence]` artifacts that you generated as output from calling your `nw-align` action. To see what it looks like, load the visualizer with QIIME 2 View. You should see something like this at QIIME 2 View:

```
TabularMSA[DNA]
-------------------------
Stats:
sequence count: 2
position count: 25
-------------------------
ACCGGTGGAACCGG-TAACACCCAC
ACCGGT--AACCGGTTAACACCCAC
```

While you're there, also take a minute to review the Provenance of your visualizer using the *Provenance* tab on that page.

An optional exercise

As mentioned above, our visualizer is a little crude and if you try to summarize long alignments (longer than 80 positions) you won't see the full alignment. Spend some time making it look a little nicer, and maybe even expanding this to see the full alignment for longer alignments. For example, color the different nucleotide characters differently, or use colors or other formatting to indicate where there are matches, mismatches, and gaps in the alignment. You don't have to use the `repr` function - you can access other parts of the `TabularMSA` API to display whatever information you'd like.

1.1.5 Add a new Artifact Class

Now that we've built a basic method and a basic visualizer, let's step into another unique aspect of developing with QIIME 2: defining *artifact classes*. *Artifact classes* are closely related to terms you have probably already encountered in the QIIME 2 ecosystem: *semantic types*, *formats*, and *transformers*. For most of the new QIIME 2 developers who I've worked with, defining a new artifact class (and the associated semantic type, format(s), and transformer(s)) is the most obscure step. However it's what gives QIIME 2 a lot of its power (for example, its ability to be accessed through different interfaces and to help users avoid analytic errors) and flexibility (for example, its ability to load the same artifact into different data types, depending on what you want to do with it).

Recall that when we initially built our `nw-align` method, we used artifact classes that were suboptimal because there weren't relevant existing ones. Two questions may arise here. First, as a plugin developer, how do you know what relevant artifact classes are available to you for use in development? And second, how do you add one or more new artifact classes if you need some that don't exist?

We'll start here with a brief *explanation* to set the stage for the work we'll do in this section of the tutorial. Then, we'll address the first question with a very brief *how-to*, because if you don't need to define a new semantic type, that's ideal. And finally, the majority of this section of the tutorial will focus on creating a new artifact class for use in our plugin.

tl;dr

The complete code that I developed to define my new artifact class, including the corresponding semantic type, formats, and transformer, can be found here: https://github.com/caporaso-lab/q2-dwq2/commit/161c8a3a130393d24e5e538e9a622dfef51ead11. The code that I developed to transition my `nw-align` action to use my new artifact class can be found here: https://github.com/caporaso-lab/q2-dwq2/commit/b625b7f0b8194128c5d1c9a5892ce5bcd85ec81b.

Artifact classes

I'm going to begin by defining an *artifact class* as **a kind of QIIME 2 artifact that can exist**. A new artifact class can be registered by a plugin developer by associating a *semantic type* with a *format*. Let's briefly discuss both of those terms.

Semantic types define the meaning of the data - i.e., what it represents. For example, QIIME 2 defines a `Phylogeny[Rooted]` artifact class, and the semantic type associated with the artifact class is `Phylogeny` (i.e., a phylogenetic tree) with a sub-semantic-type `Rooted` (i.e., implying that the phylogenetic tree contains a specified root node). Together, therefore, the `Phylogeny[Rooted]` artifact class can be described as a rooted phylogenetic tree.

Formats describe how the data will be represented inside the artifact when it is serialized for storage (generally meaning written to file, in this context). Continuing with our phylogenetic tree example, multiple different file formats have been defined to represent a rooted phylogenetic tree, including newick and NEXUS, and this is typical for most types of information in bioinformatics (and other fields). Similarly, a single file format can be used to store semantically different information - for example, newick can also be used to store unrooted phylogenetic trees.

This means that formats are inherently independent of semantic types: a semantic type doesn't imply a specific format, and a format doesn't imply a specific semantic type. When a new artifact class is registered by a developer, they associate

a semantic type with a format. This enables QIIME 2 to know what an artifact class is intended to represent, and how it can be read from and written to disk. With this information, an artifact class can exist.

Transformers, which we'll come back to later, can be associated with formats and used to convert (transform) between formats, load formats in data types, and more. Among other things, transformers enable a given artifact class to update its format in new versions of QIIME 2, for example if a more efficient format becomes available, without end-users needing to know that anything changed.

Discovering artifact classes

The main way that plugin developers become aware of the existing artifact classes that are relevant for their plugin is through familiarity from using QIIME 2. For example, if you're planning to add a new method for feature table normalization, you may know that you start with the same feature table artifact class that is used by the `rarefy` action in the `q2-feature-table` plugin. That gives you a lead on the artifact class you're going to use.

On the other hand, if you're searching to see what artifact classes are available, the best approach right now is to call `qiime tools list-types`, which lists the artifact classes that are available. If you do this in your development environment, you should see something like the following:

```
$ qiime tools list-types

Bowtie2Index
        No description

BrackenDB
        No description

DistanceMatrix
        A symmetric matrix representing distances between entities.

FeatureData[AlignedProteinSequence]
        Aligned protein sequences associated with a set of feature
        identifiers. Exactly one sequence is associated with each
        feature identfiier.

FeatureData[AlignedRNASequence]
        Aligned RNA sequences associated with a set of feature
        identifiers. Exactly one sequence is associated with each
        feature identfiier.

...
```

Note

As of this writing (April 2024) you'll see many artifact classes that don't have descriptions. The ability to add descriptions to artifact classes was added relatively recently, and we're working through existing types to add descriptions.

Some of these are self-explanatory and some are a bit opaque. Take a minute to review this list to identify artifact classes that you've used before, and any others that might be relevant in your work.

Note

A how-to article is planned that will provide additional detail on identifying an existing artifact class for use in your plugin. In the meantime, please feel free to reach out through the QIIME 2 Forum Developer Discussion if you're

struggling to identify a relevant semantic type - we know this can be challenging, and we don't mind helping.

Developing a new artifact class

Now that we have that background, lets get into it.

Defining a new semantic type

In this section we're going to define a new artifact class called `SingleDNASequence`, which represents a single DNA sequence. Recall that previously we associated the `FeatureData[Sequence]` artifact class with our inputs to `nw-align`, but since this artifact class is intended to represent collections of sequences (rather than the single sequences that we use as input to `nw-align`) it was just a way to quickly get started. Defining `SingleDNASequence` will allow us to better represent the input to `nw-align`, and to reduce some unnecessary code that we wrote in our action.

Start by creating a new file, `_types_and_formats.py` in your module's top-level directory. For me, this file will be `q2-dwq2/q2_dwq2/_types_and_formats.py`. Add the following code to that file.

```
from qiime2.plugin import SemanticType

SingleDNASequence = SemanticType("SingleDNASequence")
```

That code defines a new semantic type, which can be referred to as `SingleDNASequence`. Defining new semantic types is the easier part of defining new artifact classes.

Defining a new file format

The next thing we'll do is define a file format that we'll use with this semantic type to define our artifact class. We'll call our file format `SingleRecordDNAFASTAFormat`, implying that it is a fasta-formatted file for storing a single DNA sequence record.

Our `SingleRecordDNAFASTAFormat` will be a subclass of QIIME 2's `TextFileFormat` class. The only requirement of our subclass is that it define a method called `_validate_`, and that method should take a validation `level` as input and not return any output. `level` will always be provided as either `max` (the default) or `min`. If a problem is discovered during validation, a `qiime2.plugin.ValidationError` should be raised. If `_validate_` returns without raising a `ValidationError`, that indicates that validation has succeeded.

It's up to you as the format developer to define what happens in the `_validate_` function, and that can range from no validation whatsoever (i.e., just `pass` in the function body; we don't recommend this and *consider it a plugin development anti-pattern*), to extremely detailed validation. The `level` is used to define whether minimal or maximal validation should be performed. The trade-off is that maximal validation can take a long time, slowing down use of this file format (which will likely be percieved as your plugin being slow), but if written well can prevent against invalid data being packaged in this file format. Minimal validation on the other hand can be very quick, but may allow some errors to sneak through. You don't have to do anything differently inside of `_validate_` based on whether a user requests minimal or maximal validation, but it's a good idea to vary what is being done if maximal validation will be slow.

Note

As a practical example of when minimal validation can be helpful, think of the case where large machine-generated fastq files are associated with a QIIME 2 format (e.g., during import). Validating each file in its entirety can take a very long time, and since the fastq files are machine generated, they're (presumably) unlikely to contain errors because the developers of the code that wrote that fastq (presumably) tested that code well.

The less you trust the creator of a file, the more important validation is. If your users are manually creating a file, that's an important case for extensive validation. No offense to your users: creating files can be tedious, and tedious work is error prone when done by humans (we get bored and make mistakes). Computers on the other hand are great at it. So automate everything you can… but I digress.

Tip

One feature to be aware of here, though we won't use it right away, is that since you're defining your format class, you can add additional methods or properties to it that will be convenient for you if/when you work directly with instances of the format in your actions. For example, if you wanted an easy way to get the identifier of the sequence in this object, you could add a `SingleRecordDNAFASTAFormat.get_sequence_id()` method (for example), and then access that in the normal way when you have an instance of this class.

Add the following code to `_types_and_formats.py` (note that I'm building on my `import` statement from the previous code block here):

```
from skbio import DNA
from skbio.io import UnrecognizedFormatError

from qiime2.plugin import SemanticType, TextFileFormat, ValidationError

class SingleRecordDNAFASTAFormat(TextFileFormat):

    def _confirm_single_record(self):
        with self.open() as fh:
            try:
                # DNA.read(..., validate = False) disables checking to ensure
                # that all characters in the sequence are IUPAC DNA characters
                # by scikit-bio.
                # This will be validated independently by _validate_, with user
                # control over how much of the sequence is read during
                # validation, to manage the runtime of validation.
                _ = DNA.read(fh, seq_num=1, validate=False)
            except UnrecognizedFormatError:
                raise ValidationError(
                    "At least one sequence record must be present, but none "
                    "were found."
                )

            try:
                _ = DNA.read(fh, seq_num=2, validate=False)
            except ValueError:
                # if there is no second record, a ValueError should be raised
                # when we try to access the second record
                pass
            else:
                raise ValidationError(
                    "At most one sequence record must be present, but more "
                    "than one record was found."
                )

    def _confirm_acgt_only(self, n_chars):
        with self.open() as fh:
            seq = DNA.read(fh, seq_num=1, validate=False)
```

(continues on next page)

(continued from previous page)

```
            validation_seq = seq[:n_chars]
            validation_seq_len = len(validation_seq)
            non_definite_chars_count = \
                validation_seq_len - validation_seq.definites().sum()
        if non_definite_chars_count > 0:
            raise ValidationError(
                f"{non_definite_chars_count} non-ACGT characters detected "
                f"during validation of {validation_seq_len} positions."
            )

    def _validate_(self, level):
        validation_level_to_n_chars = {'min': 50, 'max': None}
        self._confirm_single_record()
        self._confirm_acgt_only(validation_level_to_n_chars[level])
```

Take a minute to read through this, starting with the `_validate_` function and following the function calls that are made inside of it. What is being done during validation? How is the validation level being used here?

Defining a new directory format

In the previous sections we discussed that QIIME 2 uses file formats to describe how data is organized in artifacts for specific artifact classes. In a QIIME 2 artifact, the relevant data is stored in the `data/` directory. For some artifact classes, including the one we're building here, all of the relevant data is contained in a single file. However in other cases, there may be multiple files and/or subdirectories. Because we're defining the contents of a directory when registering a new artifact class, we actually need to associate a `qiime2.plugin.DirectoryFormat` with our new artifact class.

For simple cases like ours, where there will only be a single file in that directory, QIIME 2 has a helper function, `qiime2.plugin.model.SingleFileDirectoryFormat`, which we can use to create a directory format. Add the following code to `_types_and_formats.py` to define your directory format.

```
from skbio import DNA
from skbio.io import UnrecognizedFormatError

from qiime2.plugin import SemanticType, TextFileFormat, model, ValidationError

SingleRecordDNAFASTADirectoryFormat = model.SingleFileDirectoryFormat(
    'SingleRecordDNAFASTADirectoryFormat', 'sequence.fasta',
    SingleRecordDNAFASTAFormat)
```

This code creates a new object, `SingleRecordDNAFASTADirectoryFormat`. The parameters being provided in the call to `SingleFileDirectoryFormat` are the name of the directory format, what we'd like to call the single file in the directory format, and what format this file will be in. We'll call the file in our directory format `sequence.fasta` (it can be anything, but making it descriptive helps), and it will be of the type we just defined, `SingleRecordDNAFASTAFormat`.

This completes the code you'll need in `_types_and_formats.py`. Compare your code against *mine* to make sure it's functionally identical.

Registering an artifact class

At this stage, we have defined our sematic type and the formats that we'll associated with this semantic type. We'll now move on to registering these, so we can use them.

Making the new type and formats publicly importable

Next, open the `__init__.py` in the top-level directory of your module. For me, this will be `q2-dwq2/q2_dwq2/__init__.py`. Add the following lines to the imports at the top of your file:

```
from ._types_and_formats import (
    SingleDNASequence, SingleRecordDNAFASTAFormat,
    SingleRecordDNAFASTADirectoryFormat)
```

Then add the following lines at the bottom of the file:

```
__all__ = [
    "SingleDNASequence", "SingleRecordDNAFASTAFormat",
    "SingleRecordDNAFASTADirectoryFormat"]
```

This will allow you and others to import your semantic type and formats directly from the module without accessing files that are intended to be private (e.g., by calling `from q2_dwq2 import SingleDNASequence`). This gives you the freedom to reorganize files or file contents internally in your plugin without changing the public-facing API - even if you update the import statements in this file, anyone importing from your code (e.g., plugin developers who are building other plugins that depend on yours) shouldn't need to change their code.

Registering the type, formats, and artifact class

Next, we'll edit our `plugin_setup.py` file to register our new type, our new formats, and our new artifact class. To do this, first we'll import those three objects in `plugin_setup.py`:

```
from q2_dwq2 import (
    SingleDNASequence, SingleRecordDNAFASTAFormat,
    SingleRecordDNAFASTADirectoryFormat)
```

Notice that because of the additions we made in `__init__.py` we import these from our top-level module directly (not from `q2_dwq2._types_and_formats.py`, where they are defined).

Next, we'll call three methods on our `plugin` object as follows:

```
# Register semantic types
plugin.register_semantic_types(SingleDNASequence)

# Register formats
plugin.register_formats(SingleRecordDNAFASTAFormat,
                        SingleRecordDNAFASTADirectoryFormat)

# Define and register new ArtifactClass
plugin.register_artifact_class(SingleDNASequence,
                               SingleRecordDNAFASTADirectoryFormat,
                               description="A single DNA sequence.")
```

The first two should be self-explanatory: we register the new semantic type and the new formats. Each of these calls takes `*args` as input, which means that you can provide all the types and formats that you want to register as arguments to these methods. Each method takes one or more arguments.

The final method call in that block is the one we've been working toward - it's where we associate our new semantic type with a directory format, and provide a description of what this artifact class is for users or other developers who may wish to use it. At this point, your plugin has defined a new artifact class and you should see it in the list of artifact classes if you run the following commands:

```
qiime dev refresh-cache
qiime tools list-types
```

> **Note**
>
> As mentioned earlier, when you add or update actions, types, or formats you'll need to run `qiime dev refresh-cache` to see those modifications through *q2cli*.

Defining and registering a transformer

There are a couple of last things that we need to do before we're ready to use our new artifact class. The first is define how an instance of our artifact class (e.g., an artifact stored in a `.qza` file) can be loaded in the form that we want to use it in inside of our action.

In QIIME 2 jargon, we review to the different ways an artifact can be used inside of an action as different **views of an artifact class**. For example, if we want to recieve the fasta file directly, we can request to *view* the artifact as a `SingleRecordDNAFASTAFormat`. Inside of our action, we could then open that and do whatever we need to with it. This approach of working with file formats or directory formats directly is common, for example, when processing raw sequence data such as collections of demutiplexed sequence reads. This might look like the following in our action definition:

```
# this is just an example - don't put this code in your plugin
def nw_align(seq1: SingleRecordDNAFASTAFormat,
             seq2: SingleRecordDNAFASTAFormat,
             ...
```

In our case, we want to load our sequences into `skbio.DNA` objects, as that's what the `skbio.alignment.global_pairwise_align_nucleotide` function that we're calling takes as input. So, we need to transform our fasta-formatted file into an `skbio.DNA` object, and we do that with a *transformer*.

Create a new file, `_transformers.py`, in your top-level module directory. Mine is called `q2-dwq2/q2_dwq2/_transformers.py`. Add the following code to that file:

```
from skbio import DNA

from q2_dwq2 import SingleRecordDNAFASTAFormat

from .plugin_setup import plugin


# Define and register transformers
@plugin.register_transformer
def _1(ff: SingleRecordDNAFASTAFormat) -> DNA:
    # by default, DNA.read will read the first sequence in the file
    with ff.open() as fh:
        return DNA.read(fh)
```

This code first imports the objects that we want to transform to (`skbio.DNA`) and from (`SingleRecordDNAFASTAFormat`). We then import our `plugin` object from `plugin_setup.py`, which

we'll use to register our transformer. Finally, we define our transformer function and register it with our plugin using a function decorator.

Your transformer function can be called anything you want, but by convention they receive arbitrary names. This is because the transformers are never called directly by users or developers, and because the function signature's type hints unambiguously define what it does. Before we adopted this convention we had a lot of functions with names like `_transform_SingleRecordDNAFASTAFormat_to_DNA`, which was starting to feel silly and in some cases the names were ambiguous (but if you prefer that, there are no issues with naming your transformers that way). Internally, this function does whatever it needs to to convert (or transform) the input object to the output object. In our case, we're opening the file format (`ff`) object provided as input, and reading the first (and only, in this case) sequence from it using `skbio.DNA.read`, and returning the result.

Note

You may have noticed that our artifact class stores data as defined in our `SingleRecordDNAFASTADirectoryFormat` class, but we're working with it here in a `SingleRecordDNAFASTAFormat` object. QIIME 2 automatically creates transformers from directory formats to file formats for single-file directory formats when they are created with `model.SingleFileDirectoryFormat`, as we did above. QIIME 2 also knows how to chain transformers, such that when we attempt to view an instance of our artifact class as `skbio.DNA`, it will first transform from `SingleRecordDNAFASTADirectoryFormat` to `SingleRecordDNAFASTAFormat`, and then transform from `SingleRecordDNAFASTAFormat` to `skbio.DNA`. If you're concerned that a chained transformation will be too slow, you can also define a transformer that skips the intermediate step. In this case, that might have a signature like:

```
_2(df: SingleRecordDNAFASTADirectoryFormat) -> skbio.DNA
```

That won't be needed here however.

Note

If you'd like to be able to view a `SingleDNASequence` artifact as another data type for use in your actions - for example, as a Python string (`str`) - you can define another transformer for it. For example:

```
_3(ff: SingleRecordDNAFASTAFormat) -> str
```

or

```
_4(seq: skbio.DNA) -> str
```

To make this transformer accessible to your plugin, there's just one last thing to do, which is make sure that the code in this file runs when the `plugin` object is created. For this, we go back to `plugin_setup.py`. Add the following line to the imports at the top of that file:

```
import importlib
```

Then, add the following as the last line in your file:

```
importlib.import_module('q2_dwq2._transformers')
```

This will load and run the `_transformers.py` file, ensuring that our transformer is registered after the `plugin` object has been instantiated and our type and formats have been registered.

Unit testing

As always, before we use this code, we're going to test it. At a high-level, there are three things we need to test: the semantic type we defined, the formats we defined, and the transformer we defined. Let's start with the type and formats.

Testing the semantic type and formats

As you probably noticed, there isn't much to a semantic type - we're essentially just defining a name that will be linked to an artifact class. We therefore just want to confirm that the type is registered with the plugin. `qiime2.plugin.testing.TestPluginBase` provides a method for this, `assertRegisteredSemanticType`.

For our formats, most of the action is happening inside of the `_validate_` function, so that's mostly what we're testing. I like to start with confirming that a few valid examples of my format pass validation, and then provide invalid files that are invalid for different reasons to confirm that those files fail validation. It's also a good idea to confirm that your validation level is functioning as expected.

Below is my test code, which lives in `q2-dwq2/q2_dwq2/tests/test_types_and_formats.py`. I added a few new test data files to support these tests, which you can access from *my code on GitHub*.

```
from qiime2.plugin import ValidationError
from qiime2.plugin.testing import TestPluginBase

from q2_dwq2 import (
    SingleDNASequence, SingleRecordDNAFASTAFormat
)


class SingleDNASequenceTests(TestPluginBase):
    package = 'q2_dwq2.tests'

    def test_semantic_type_registration(self):
        self.assertRegisteredSemanticType(SingleDNASequence)


class SingleRecordDNAFASTAFormatTests(TestPluginBase):
    package = 'q2_dwq2.tests'

    def test_simple1(self):
        filenames = ['seq-1.fasta', 'seq-2.fasta', 't-thermophilis-rrna.fasta']
        filepaths = [self.get_data_path(fn) for fn in filenames]

        for fp in filepaths:
            format = SingleRecordDNAFASTAFormat(fp, mode='r')
            format.validate()

    def test_invalid_default_validation(self):
        fp = self.get_data_path('bad-sequence-1.fasta')
        format = SingleRecordDNAFASTAFormat(fp, mode='r')
        self.assertRaisesRegex(ValidationError,
                               "4 non-ACGT characters.*171 positions.",
                               format.validate)

    def test_invalid_max_validation(self):
        fp = self.get_data_path('bad-sequence-1.fasta')
        format = SingleRecordDNAFASTAFormat(fp, mode='r')
        self.assertRaisesRegex(ValidationError,
```

(continues on next page)

(continued from previous page)

```
                                "4 non-ACGT characters.*171 positions.",
                                format.validate,
                                level='max')

    def test_invalid_min_validation(self):
        fp = self.get_data_path('bad-sequence-1.fasta')
        format = SingleRecordDNAFASTAFormat(fp, mode='r')
        # min validation is successful
        format.validate(level='min')
        # but max validation raises an error
        self.assertRaisesRegex(ValidationError,
                                "4 non-ACGT characters.*171 positions.",
                                format.validate,
                                level='max')

        fp = self.get_data_path('bad-sequence-2.fasta')
        format = SingleRecordDNAFASTAFormat(fp, mode='r')
        self.assertRaisesRegex(ValidationError,
                                "4 non-ACGT characters.*50 positions.",
                                format.validate,
                                level='min')
```

Testing the transformer

Next, we'll test our transformer. Here, we should provide a few different valid inputs, and test that they are transformed to the expected output. Generally you should use the `transform_format` action in `qiime2.plugin.testing.TestPluginBase` to access and run the transformer (as opposed to importing the function directly from `_transformers.py`). This also tests that the transformer is registered with the plugin.

The test code that I wrote for this is in `q2-dwq2/q2_dwq2/tests/test_transformers.py`, and follows here:

```
from skbio import DNA

from qiime2.plugin.testing import TestPluginBase

from q2_dwq2 import SingleRecordDNAFASTAFormat


class SingleDNASequenceTransformerTests(TestPluginBase):
    package = 'q2_dwq2.tests'

    def test_single_record_fasta_to_DNA_simple1(self):
        _, observed = self.transform_format(
            SingleRecordDNAFASTAFormat, DNA, filename='seq-1.fasta')

        expected = DNA('ACCGGTGGAACCGGTAACACCCAC',
                       metadata={'id': 'example-sequence-1', 'description': ''})

        self.assertEqual(observed, expected)

    def test_single_record_fasta_to_DNA_simple2(self):
        _, observed = self.transform_format(
            SingleRecordDNAFASTAFormat, DNA, filename='seq-2.fasta')

        expected = DNA('ACCGGTAACCGGTTAACACCCAC',
```

(continues on next page)

(continued from previous page)

```
                        metadata={'id': 'example-sequence-2', 'description': ''})

        self.assertEqual(observed, expected)
```

Write your tests, and then run them with `make test`. You should see something like the following:

```
$ make test
...
====== 14 passed, 23 warnings in 1.51s ======
```

If you have failing tests, work through them to figure out what's wrong. If you get stuck, refer back to my code. At this point, you should have implemented everything in *the first of my commits* associated with this section.

Updating `nw-align` to use the new artifact class

Ok. That was a lot of work. But hopefully you can see that none of the coding is very hard, even if conceptually it's a little challenging as you get started. As we continue to work through the tutorial, I'll point out places where the work we did here provides powerful benefits for our plugin users and for us as plugin developers.

Now let's update our `nw-align` method to use the new artifact class that we defined. As discussed earlier, this will enable us to simplify the code and associated tests. Since we're looking at code changes here, rather than new code, the most convenient view you'll have is GitHub's diff of this commit against the previous. To see this, open the link to the *second of my commits* associated with this section.

The work that we're doing here is transitioning our use of the `FeatureData[Sequence]` artifact class for our new `SingleDNASequence` artifact class, and transitioning our use of the `DNAIterator` view inside `nw-align` to use `skbio.DNA` as our view.

Start by looking at the changes in `test_methods.py`, and adapt your code in the same way. Notice that in the second test case (`test_simple2`), we're using `qiime2.plugin.util.transform` to load files stored in our `tests/data` directory into `skbio.DNA` objects for use in the tests.

> **Note**
>
> Pending a fix for an oversight in `TestPluginBase`, it will be possible to use `qiime2.plugin.testing.TestPluginBase.transform_format` for performing the fasta file to `skbio.DNA` transformation. This will enable tests of actions to use the same machinery used during testing of transformers.

If you run your unit tests now with `make test`, you should get some test failures. That's expected, as we updated the tests but haven't yet updated the code. Let's now update the code, and we'll know we're done when these tests pass.

First, update the method itself, as I did in `q2-dwq2/q2_dwq2/_methods.py`. Here you're telling QIIME 2 to use a different view type inside this function, and then we're removing some of the clunky code that we no longer need. Here's my `nw_align` function after making these changes:

```
def nw_align(seq1: DNA,
             seq2: DNA,
             gap_open_penalty: float = 5,
             gap_extend_penalty: float = 2,
             match_score: float = 1,
             mismatch_score: float = -2) -> TabularMSA:
    msa, _, _ = global_pairwise_align_nucleotide(
        seq1=seq1, seq2=seq2, gap_open_penalty=gap_open_penalty,
        gap_extend_penalty=gap_extend_penalty, match_score=match_score,
```

(continues on next page)

(continued from previous page)

```
        mismatch_score=mismatch_score
    )

    return msa
```

Then, update `plugin_setup.py` to associate our new artifact class with the sequence inputs. Here's what this looks like for me, after I make this change:

```
plugin.methods.register_function(
    function=nw_align,
    inputs={'seq1': SingleDNASequence,
            'seq2': SingleDNASequence},
...
```

After making the changes that I made, all tests should pass when you run `make tests`. Once all tests are passing, run `qiime dev refresh-cache` and call help on your plugin's `nw-align` action. You should see the new types associated with the `seq1` and `seq2` inputs. Refer back to *where we tried out the nw-align action* for the first time. Using those same fasta files (or any others you'd like), adapt the commands in that section to import sequence data into artifacts of our new artifact class, and run `nw-align` on them.

An optional exercise

Try making a new visualizer that will create a visual summary of a `SingleDNASequence` artifact class. Define a transformer to a view type other than `skbio.DNA`, and use that in your visualizer. For example, does another library like BioPython provide an object with a convenient view that you could use here? (Note that you may need to install any other library that you're using here in your development environment.)

1.1.6 Add a Usage Example

If you want others to use your new functionality, it isn't *really* done until you document how to use it. Let's do that now. This will ensure that users know how to use your code, and it will give them something to try after they install your plugin to convince them that it's working. I generally find that I'm the first person to benefit from my documentation, and in some cases I'm also the person who most frequently benefits from my documentation (e.g., if it's code that I write for my own purposes, rather than something I intend to broadly disseminate).

QIIME 2 provides a framework for defining *usage examples* for plugins. Usage examples are defined abstractly, rather than based on a specific QIIME 2 interface, and QIIME 2 interfaces can define *usage drivers* that know how to translate those abstract definitions into instructions for their application through a specific interface. For example, the framework contains a usage driver for the *Python 3 API*, and therefore can turn abstract usage examples into a series of Python 3 commands. *q2cli* contains a usage driver that translates abstract usage examples into a series of shell commands. And q2galaxy contains a usage driver that turns examples into text-based instructions for running them through *Galaxy*. So, by writing a single usage example, your documentatation is targeted toward users with different levels of computational expertise, and as new interfaces become available you don't need to update your usage examples for users to be able to work with your examples through them. This is one way the framework supports our goal of meeting users where they are in terms of their computational experience, and it's one of the big benefits that you as a plugin developer gets by developing with QIIME 2.

In this section of the tutorial, we'll define a usage example for our `nw-align` action.

tl;dr

The full code that I developed for this section can be viewed here: https://github.com/caporaso-lab/q2-dwq2/commit/790c73536a7d0cbf6c4a3f07630c65a79c5d6077.

Defining a usage example for `nw-align`

Usage examples work on actual data that you define when you create the usage example. This allows users to run your usage examples, explore the input and output, and confirm that things work as expected before they try your plugin on their own data. This also allows you to have your usage examples automatically tested every time your run your test code, which lets you make and honor a commitment to your users that the documentation you provide will work, because you can automatically run all of the usage examples you define with a single command after every change you make so you'll be the first to know if anything is broken.

So let's start by defining data for use by our `nw-align` usage example. This is done by creating a function that returns a QIIME 2 artifact that is used as an input in the example.

Define input data for your usage example

Start by creating a top-level file in your module called `_examples.py`. Mine will be called `q2-dwq2/q2_dwq2/_examples.py`. Add the following code, and then we'll work through it.

```
import tempfile

import skbio

import qiime2

from q2_dwq2 import SingleRecordDNAFASTAFormat


def seq1_factory():
    seq = skbio.DNA("AACCGGTTGGCCAA", metadata={"id": "seq1"})
    return _create_seq_artifact(seq)


def seq2_factory():
    seq = skbio.DNA("AACCGCTGGCGAA", metadata={"id": "seq2"})
    return _create_seq_artifact(seq)


def _create_seq_artifact(seq: skbio.DNA):
    ff = SingleRecordDNAFASTAFormat()
    seq.write(str(ff.path))
    return qiime2.Artifact.import_data("SingleDNASequence", ff)
```

Our goal here is to define two "factory" functions - one for each of `nw-align`'s `SingleDNASequence` inputs - that each return an *artifact of class* `SingleDNASequence`. These functions will be used when we define our usage example, and because of how usage example definitions work, these functions can't take any parameters as input. Because both of these functions will do similar work under the hood, I am also creating a helper function that creates a `SingleDNASequence` artifact from an `skbio.DNA` object. That lets me avoid duplicating code.

The factory functions I defined here are `seq1_factory` and `seq2_factory`. Each creates an `skbio.DNA` object, and then passes that to my helper function, `_create_seq_artifact`.

> **Tip**
>
> The type hint in the `_create_seq_artifact` function definition isn't required, but I like to include it to remind myself how this function works when I come back to it in the future. It makes my code more self-documenting.

`_create_seq_artifact` takes an `skbio.DNA` sequence object as input. Internally, it creates a `SingleRecordDNAFASTAFormat` object which is assigned to the variable `ff` (for *file format*). Our sequence is written to `ff.path` (i.e., the file format's `path` object, which we cast to a string) using `seq.write`. In the final step, we use `qiime2.Artifact.import_data`, which allows us to import data in a similar way as if we were calling `qiime tools import` through q2cli (incidentally, `qiime tools import` calls `qiime2.Artifact.import_data`, under the hood). We provide the artifact class that we want to import into, and the file format (`ff`) that we want to import from, and we get a QIIME 2 artifact back. That artifact is returned by the helper function, and in turn is returned by the factory function that called it.

This may feel like a lot of work to define data to use in an example, but it provides a lot of flexibility in how the usage example you define can ultimately be used. For example, it allows some *usage drivers* to actually create these inputs (for example, a usage driver that is going to be used to *test the examples*), while another usage driver can just act as if they were created but not bother taking the time to actually create them (for example, usage drivers that are *displaying examples in command line help* text but not executing them).

Defining the usage example

Next, we define a usage example as a function that takes a `UsageDriver` subclass as input. We often call the input `use`, by convention, but you can call it anything. This function starts by instantiating two sequence artifacts using the factory functions that we just defined. It then defines the relevant action call, which in our case will be to the `dwq2` plugin's `nw_align` action. It also assigns the inputs, and provides a name for the output.

This usage example is using default parameter values, but you could addionally pass parameters to the action in this usage example, or add a second usage example that does that which you also associate with this action.

The following code in my `q2-dwq2/q2_dwq2/_examples.py` file defines the usage example:

```
def nw_align_example_1(use):
    seq1 = use.init_artifact('seq1', seq1_factory)
    seq2 = use.init_artifact('seq2', seq2_factory)

    msa, = use.action(
        use.UsageAction(plugin_id='dwq2',
                        action_id='nw_align'),
        use.UsageInputs(seq1=seq1, seq2=seq2),
        use.UsageOutputNames(aligned_sequences='msa'),
    )
```

Registering the usage example

Finally, we're ready to register the usage example. For this, we'll go back to the `plugin_setup.py` file.

You should first import the new usage example function by adding the following line to your imports at the top of the file:

```
from q2_dwq2._examples import nw_align_example_1
```

Then, in the call to `plugin.methods.register_function`, you should add an `examples` parameter, to which you can provide a dictionary mapping example names to usage example functions. These names (i.e., dictionary keys) will be displayed with the usage example in some interfaces. To do this, adapt your call to `plugin.methods.register_function` as follows.

```
plugin.methods.register_function(
    function=nw_align,
    ...
    citations=[citations['Needleman1970']],
    examples={'Align two DNA sequences.': nw_align_example_1}
)
```

That completes the definition and registration of our `nw-align` usage example.

Displaying usage examples

In this section we'll work through some user-facing commands that allow you or your users to view your usage examples. First, and most straight-forward, is to do this through q2cli. If you call your action with the `--help` parameter, you will now see the usage example at the bottom of the resulting help text.

Command line interface

```
$ qiime dwq2 nw-align --help
Usage: qiime dwq2 nw-align [OPTIONS]

...

Examples:
  # ### example: Align two DNA sequences.
  qiime dwq2 nw-align \
    --i-seq1 seq1.qza \
    --i-seq2 seq2.qza \
    --o-aligned-sequences msa.qza
```

You can test this command by having QIIME 2 write the example data we defined to file using the following q2cli command:

```
$ qiime dwq2 nw-align --example-data usage-example-data/
```

After generating the example data, run the usage example as described in the help text providing the example data as input and confirm that it works as expected.

Python 3 API

As mentioned above, the abstract nature of our usage example definition enables it to be interpreted by different usage drivers, enabling the same example to be presented as it would be used through different interfaces. Open an `ipython` shell in your development environment, and try the following:

```
from qiime2.plugins import dwq2, ArtifactAPIUsage

examples = dwq2.actions.nw_align.examples

for example in examples.values():
    use = ArtifactAPIUsage()
    example(use)
    print(use.render())
```

This should display how to run this example through the Python 3 API. Try it out with the same example data that you generated above (you can use `qiime2.Artifact.load` to load the example files into QIIME 2 artifacts to be provided as input to this call to `nw_align`).

```
import qiime2.plugins.dwq2.actions as dwq2_actions

msa, = dwq2_actions.nw_align(
    seq1=seq1,
    seq2=seq2,
)
```

Automated testing of usage examples

Finally, it's a good idea to have your usage examples run as part of your test suite, as a way to assess if any future changes you make to your code break the usage examples you defined. To do this, create a new test file in your `tests` directory. Mine is called `q2-dwq2/tests/test_examples.py`. Add the following code to that file:

```
from qiime2.plugin.testing import TestPluginBase


class UsageExampleTests(TestPluginBase):
    package = 'q2_dwq2.tests'

    def test_examples(self):
        self.execute_examples()
```

Save the file, and run `make test`. That will now run this additional test, which uses `TestPluginBase.execute_examples` to discover and run all of the usage examples defined in the plugin. You should see output like the following:

```
$ make test

...

===== 15 passed, 25 warnings in 15.06s ======
```

This code tests that the usage examples ran successfully, but importantly it does not test that any output they produce aligns with expected output. It is possible to additionally check the output of these examples, but that doesn't replace the need for unit tests. Unit tests tend to be more expressive and useful for testing your plugin's functionality, while automated usage example testing is a good way to assess the validity of your documentation. If you'd like to learn more about testing specific output that you get from running your usage examples, refer to the *Writing Usage Examples How to* guide.

Writing tutorials

Usage examples provide users with guidelines on the specific commands they can run to use your plugin, but they don't provide a lot of context. It's a good idea to also write tutorials that describe what your plugin is intended to do, how and why to use it, and how to interpret the results. Ideally the tutorial also provides a small data set that can be analysed quickly on a modestly powered laptop computer. I credit a lot of the popularity of QIIME 1 and QIIME 2 to its tutorials and to our support forums.

Writing tutorials is out of scope of this document for now, though we may add a *How to* article in the future that discusses writing and automating testing of tutorials. As a general recommendation though, I highly recommend writing your tutorials using Jupyter Book, which is what *Developing with QIIME 2* is written with. It's very feature rich, easily deployed with GitHub Actions, and it makes it straight-forward to create nice looking documentation. Diataxis is also great reading material on how your tutorials can be structured, and if you get excited about writing documentation (it's fun!) the Write the Docs community is a group of like-minded folks.

Happy documenting!

Optional exercise

Add a usage example for your `summarize` visualizer.

Need some hints?

For your example data, you can use the output generated by the `nw-align` usage example that we created here. If you export that artifact, you can find the data in the format you'll need to create it in your factory function. You can also find the artifact class that you'll need to use in your `qiime2.Artifact.import_data` call using the `qiime tools peek` command.

1.1.7 Add a second transformer

A couple of sections back (in *Add a new Artifact Class*), I noted that adding transformers was an obscure step for a lot of new plugin developers. Let's circle back to that now with the goal of developing a better understanding of the role of transformers in QIIME 2, and also to simplify the code for generating usage examples that we just wrote.

Take a minute to review the helper function we defined in our `_examples.py` file, and try to describe in a sentence or two what that code is doing. Here it is again, for reference:

```
def _create_seq_artifact(seq: skbio.DNA):
    ff = SingleRecordDNAFASTAFormat()
    seq.write(str(ff.path))
    return qiime2.Artifact.import_data("SingleDNASequence", ff)
```

Here's my description of what this is doing, but come up with your own before looking at this.

This code is transforming (or converting) an `skbio.DNA` object into a `q2_dwq2.SingleRecordDNAFASTAFormat` object, and then importing that format into a QIIME 2 artifact.

Transformers in QIIME 2 are designed to handle converstions between objects behind the scenes, so that users don't ever have to think about this, and developers can think about it as infrequently as possible. In this section, we'll do a small refactor of the code we wrote in the previous section.

tl;dr

The code that I wrote for this section can be found here: https://github.com/caporaso-lab/q2-dwq2/commit/93a3098b4e18796e8c33cd35088bf2a3623eed20.

Define a transformer from `skbio.DNA` to `q2_dwq2.SingleRecordDNAFASTAFormat`

The first transformer that we wrote transforms our `q2_dwq2.SingleRecordDNAFASTAFormat` object to an `skbio.DNA` object, so that we can view artifacts of class `SingleDNASequence` as `skbio.DNA` objects when we work with them. As a developer, `skbio.DNA` objects are easier to create and use than `q2_dwq2.SingleRecordDNAFASTAFormat` objects, because they have convenient APIs. Once we have a helper function for creating `q2_dwq2.SingleRecordDNAFASTAFormat` objects from `skbio.DNA` objects, like the `_create_seq_artifact` function we wrote, `q2_dwq2.SingleRecordDNAFASTAFormat` objects are also trivial to create, but it still tends to be more convenient to create and use those via an `skbio.DNA` object since we then don't have to directly deal with reading and writing files. QIIME 2 enables us to define and register functions that convert between object types as transformers, making them universally accessible in deployments where the plugin that defines and registers them is installed.

We can adapt the code from our `_create_seq_artifact` function into a new transformer in our `_transformers.py` file as follows:

```
@plugin.register_transformer
def _2(seq: DNA) -> SingleRecordDNAFASTAFormat:
    ff = SingleRecordDNAFASTAFormat()
    seq.write(str(ff.path))
    return ff
```

If you don't recall exactly what this is doing, review *the text that described this when we defined `_create_seq_artifact`*. The only difference here is that we're returning the `SingleRecordDNAFASTAFormat`, where in `_create_seq_artifact` we imported this into a `qiime2.Aritfact` as well.

This new transformer enables us to adapt our factory functions in `_examples.py` to look like the following:

```
def seq1_factory():
    seq = skbio.DNA("AACCGGTTGGCCAA", metadata={"id": "seq1"})
    return qiime2.Artifact.import_data(
        "SingleDNASequence", seq, view_type=skbio.DNA)


def seq2_factory():
    seq = skbio.DNA("AACCGCTGGCGAA", metadata={"id": "seq2"})
    return qiime2.Artifact.import_data(
        "SingleDNASequence", seq, view_type=skbio.DNA)
```

With this code, we're still importing to a `SingleDNASequence` artifact class, but this time we're doing it directly from an `skbio.DNA` view type. Under the hood, QIIME 2 checks to see if any transfomers are registered that transform a `skbio.DNA` to a `skbio.SingleRecordDNAFASTADirectoryFormat` (the format we *associated with our artifact class*). It finds a transformer from `skbio.DNA` to `skbio.SingleRecordDNAFASTAFormat`, and a transformer from `skbio.SingleRecordDNAFASTAFormat` to `skbio.SingleRecordDNAFASTADirectoryFormat`, so it applies that chain of transformers to import into the `SingleDNASequence` artifact class with the `skbio.DNA` object that we provided. Cool!

At this point, we can delete the `_create_seq_artifact` function from `_examples.py` as we have centralized the functionality for performing the transformation that it did, and we moved the import step into the factories.

Add unit tests of the new transfomer

As always, before this new code is ready for use, we need to write some unit tests. Here are the tests that I wrote in `test_transformers.py`:

```
    def test_DNA_to_single_record_fasta_simple1(self):
        in_ = DNA('ACCGGTGGAACCGGTAACACCCAC',
                  metadata={'id': 'example-sequence-1', 'description': ''})
        tx = self.get_transformer(DNA, SingleRecordDNAFASTAFormat)

        observed = tx(in_)
        # confirm "round-trip" of DNA -> SingleRecordDNAFASTAFormat -> DNA
        # results in an observed sequence that is the same as the starting
        # sequence
        self.assertEqual(observed.view(DNA), in_)

    def test_DNA_to_single_record_fasta_simple2(self):
        in_ = DNA('ACCGGTAACCGGTTAACACCCAC',
                  metadata={'id': 'example-sequence-2', 'description': ''})
        tx = self.get_transformer(DNA, SingleRecordDNAFASTAFormat)

        observed = tx(in_)
        self.assertEqual(observed.view(DNA), in_)
```

Review those to make sure that you understand them, and then copy/paste those into your plugin or write your own. Run `make test` to confirm that everything is working as expected.

1.1.8 Add a first Pipeline

In this chapter we'll add our first *Pipeline* to our plugin. Pipelines allow developers to define workflows composed of Methods and Visualizers that can be run as a single step. Pipelines are a type of *Action* in QIIME 2, like *Methods* and *Visualizers*, but they differ in a few ways.

- Unlike Methods, which exclusively produce one or more *Artifacts* as output, or Visualizers, which exclusively produce one or more *Visualizations* as output, Pipelines create one or more Artifacts *and/or* Visualizations as output.
- To support calling other Actions from within a Pipeline, the function that is registered as a Pipeline takes `qiime2.Artifact` objects as input. This is different than functions registered as Methods and Visualizers which take more typical Python objects as input (e.g., `skbio.DNA` or `pandas.DataFrame`).
- Also in support of calling other Actions from within a Pipeline, functions registered as Pipelines need to interact with a *deployment's Plugin Manager*, while the functions registered as `Methods` and `Visualizers` never interact with a Plugin Manager directly. Functions registered as Pipelines do this by taking an object that enables this communication, called `ctx` (short for *context*) by convention, as their first argument.
- Pipelines provide formal support for parallel computing, unlike Methods or Visualizers which can only provide access to parallel computing support that they define (such as providing access to multi-threaded execution of external applications that they run). As a result, Pipelines are critical for supporting workflows that are computationally expensive in QIIME 2.

In this section, we'll develop our first simple Pipeline which will chain the `nw-align` and `summarize-alignment` *Actions* that we previously wrote together in a new action that produces the alignment and the alignment summary from one command call. This will illustrate how to use Pipelines to simplify common workflows for your users. In subsequent sections of the tutorial, we'll explore developing Pipelines that provide parallel computing support across diverse high-performance computing resource configurations.

tl;dr

The complete code that I developed for this section is available here: https://github.com/caporaso-lab/q2-dwq2/commit/1e601c41b86d98b22f4e16685e868f1c5710f3bf.

Note

Starting with this section, there will be some steps that are required of you that are not described fully in the text, to provide ways to begin applying what you've learned. These are required in that the section of the tutorial you're working on may not work entirely, or may be incomplete for another reason, until they're completed. If you've been doing the *Optional Exercises* at the ends of the sections that have them, these should be straight-forward. For all of these *Required Exercises*, a link that provides a possible solution will be provided so if you get stuck that doesn't prevent you from moving on with the tutorial.

Update the `nw_align` action to avoid duplicating information

The first thing we're going to do in preparation for building this Pipeline is create an object to store the default parameter settings for our `nw_align` function. While not technically required, we'll use those same default settings again in our Pipeline, so we do this to adhere to the **D**on't **R**epeat **Y**ourself (**DRY**) principle of software engineering. This is stated in *The Pragmatic Programmer* [4] as *Every piece of knowledge must have a single, unambiguous, authoritative representation within a system*, and the authors go on to say that this is *one of the most important tools in the Pragmatic Programmer's tool box*.

A bit of the reasoning behind DRY, using our code as an example, is that if default parameter settings for our pairwise alignment functionality are defined in two different places, and we want to change those settings, it's very easy to for us (or someone else maintaining this code in the future) to erroneously make the change in only one of the two places where they are defined. This would result in different output when calling `nw_align` directly versus through our new Pipeline, which would be unexpected behavior that most people would rightly consider to be a bug.

We'll address this by putting the default parameter settings in a `dict`. We can then look up the values in that `dict` anywhere we need them, so they only need to be defined that one time. To achieve this, update the `_methods.py` file, to look like the following:

```
from skbio.alignment import global_pairwise_align_nucleotide, TabularMSA
from skbio import DNA

_nw_align_defaults = {
    'gap_open_penalty': 5,
    'gap_extend_penalty': 2,
    'match_score': 1,
    'mismatch_score': -2
}


def nw_align(
        seq1: DNA,
        seq2: DNA,
        gap_open_penalty: float = _nw_align_defaults['gap_open_penalty'],
        gap_extend_penalty: float = _nw_align_defaults['gap_extend_penalty'],
        match_score: float = _nw_align_defaults['match_score'],
        mismatch_score: float = _nw_align_defaults['mismatch_score']) \
        -> TabularMSA:
    msa, _, _ = global_pairwise_align_nucleotide(
```

(continues on next page)

(continued from previous page)

```
        seq1=seq1, seq2=seq2, gap_open_penalty=gap_open_penalty,
        gap_extend_penalty=gap_extend_penalty, match_score=match_score,
        mismatch_score=mismatch_score
    )

    return msa
```

This implementation is functionally identical to what we had before, but it now allows us to import the `_nw_align_defaults` dictionary and use it elsewhere.

Create `_pipelines.py` and add a Pipeline

Next, let's define the function that we'll register as our Pipeline. Start by creating a new file, `_pipelines.py` in your module's top-level directory. For me, this file will be `q2-dwq2/q2_dwq2/_pipelines.py`. Add the following code to that file.

```
from ._methods import _nw_align_defaults


def align_and_summarize(
        ctx, seq1, seq2,
        gap_open_penalty=_nw_align_defaults['gap_open_penalty'],
        gap_extend_penalty=_nw_align_defaults['gap_extend_penalty'],
        match_score=_nw_align_defaults['match_score'],
        mismatch_score=_nw_align_defaults['mismatch_score']):
    nw_align_action = ctx.get_action('dwq2', 'nw_align')
    summarize_alignment_action = ctx.get_action('dwq2', 'summarize_alignment')

    msa, = nw_align_action(
                    seq1, seq2, gap_open_penalty=gap_open_penalty,
                    gap_extend_penalty=gap_extend_penalty,
                    match_score=match_score, mismatch_score=mismatch_score)
    msa_summary, = summarize_alignment_action(msa)

    return (msa, msa_summary)
```

If you compare this function signature to that of our `nw_align` function, you'll notice they look very similar. This function, `align_and_summarize`, takes all of the same inputs and parameters as `nw_align`, which allows users the same control over how that functionality works as if they had called `nw_align` directly. You don't need to expose these parameters - in some cases you may design a Pipeline that implies a certain parameter choice - but in our case our goal is to make this Pipeline an alternative to calling `nw_align` and `summarize_alignment` back-to-back so it makes sense to provide full access to the functionality.

Notice that `align_and_summarize` accesses the default parameter settings in the same way that `nw_align` does, using the `_nw_align_defaults` dictionary that we created above. So, if we decide at some point that we want to update one of these defaults - maybe we want to set the default `gap_open_penalty` to `4` instead of `5`, we would update that in one place (`_nw_align_defaults`) and it will change the defaults in the two Actions that now use that information. That's DRY in action.

The one new parameter here with respect to `nw_align` is `ctx`, which was mentioned above. `ctx` is an instance of a `qiime2.sdk.Context` object. As a plugin developer, you'll never create one of these objects directly, but your Pipelines may use the `get_action` and `make_artifact` APIs that they provide. A `qiime2.sdk.Context` object will always be the first parameter that is passed to functions that are registered as `Pipelines`, and by convention this parameter is called `ctx`.

One other difference to notice here is that we are not using type hints (such as `seq1: skbio.DNA`) when we define the function that we'll register as our Pipeline. That's because the functions that we register as Pipelines, unlike those that we register as Methods and Visualizers, take Artifacts as inputs. This is because internally they call registered *Actions*, not the functions that are registered as those Actions, and all Actions in QIIME 2 take their inputs as Artifacts. It's not until `seq1` (for example) is passed in to the function registered as our `nw_align` Action (`q2_dwq2._methods.nw_align`) that it is transformed to the `skbio.DNA` object that that function takes as input.

The first thing we're doing in our `align_and_summarize` function is getting those actions that we'll use. These are retrieved using `ctx.get_action`, which takes a plugin name and an action name as input and returns the Action as a function that we can call. I'm assigning these to variables that end in `_action` to indicate that these are registered QIIME 2 Actions, not the underlying functions that are registered as those Actions. The plugin referred to in a call to `ctx.get_action` can either be the plugin you're working on (as in our case here), or any other plugin that your plugin has as a dependency. The action name can be any Method or Visualizer in that plugin. Here we're retrieving the two actions that we want to run in our Pipeline, `nw_align` and `summarize_alignment`, and assigning those to the variables `nw_align_action` and `summarize_alignment_action`.

We then call each of these functions, providing the output of `nw_align_action` (an *artifact of class* `FeatureData[AlignedSequence]`) as the input to `summarize_alignment_action`. `summarize_alignment_action` then returns a *Visualization* as output. Each of our `*_action` functions return a tuple of its *Results*, and in our case each returns only a single Result, which is why the variables that we assign the Results to are followed by commas (as in `msa, = nw_align_action`). That's a shortcut that would be equivalent to `msa = nw_align_action(...)[0]`.

Finally, we return our Pipeline's Results, an Artifact and a Visualization, in a tuple.

Register the Pipeline

Pipeline registration is similar to Methods or Visualizer registration, except that the method used for registration is `plugin.pipelines.register_function`. Because our Pipeline shares many of its inputs with our `nw_align` Method, when we register the `Pipeline` there is another opportunity for duplicated information to make its way into our plugin and violate *DRY*.

Here's the final Pipeline registration code that I ended up with in my `plugin_setup.py` file, after defining some new variables in that file:

```
plugin.pipelines.register_function(
    function=align_and_summarize,
    inputs=_nw_align_inputs,
    parameters=_nw_align_parameters,
    outputs=_align_and_summarize_outputs,
    input_descriptions=_nw_align_input_descriptions,
    parameter_descriptions=_nw_align_parameter_descriptions,
    output_descriptions=_align_and_summarize_output_descriptions,
    name="Pairwise global alignment and summarization.",
    description=("Perform global pairwise sequence alignment using a slow "
                 "Needleman-Wunsch (NW) implementation, and generate a "
                 "visual summary of the alignment."),
    # Only citations new to this Pipeline need to be defined. Citations for
    # the Actions called from the Pipeline are automatically included.
    citations=[],
    examples={}
)
```

As a **required exercise** after adding this code to your `plugin_setup.py` file, define the variables used in this code block in such a way that it avoids the DRY violation mentioned above. If you get stuck, refer to *the code that I added for this section*.

After you're done, you should be able to refresh your cache (`qiime dev refresh-cache`), and then call `--help` on your plugin to see your new `Pipeline` in the list. Try it out using the `SingleDNASequence` Artifacts that you previously passed to the `nw_align` action.

Add tests and documentation

We're of course not done with our new action until we write its tests and documentation. In both cases, this builds off work that we previously did when we defined our Method and Visualizer.

As another **required exercise**, add a unit test and a usage example for this Pipeline. Refer to *the code that I added for this section* if you need a hint.

When you're done, you should be able to run `make test` and see output like the following:

```
$ make test
...
===== 18 passed, 29 warnings in 4.66s =====
```

An optional exercise

In an earlier optional exercise, you may have added a Method for Smith-Waterman alignment, built on the `skbio.alignment.local_pairwise_align_nucleotide`. Adapt your new Pipeline so that it gives the user the option of running global (Needleman-Wunsch) or local (Smith-Waterman) alignment.

Hint

You'll likely define a new parameter in your action that toggles whether global or local pairwise alignment is used. The *Primitive Type* associated with that parameter during your Pipeline registration can be `Str % Choices`, where `Str` and `Choices` are imported from `qiime2.plugin`. You can find examples of how this is used in the `plugin_setup.py` file of the `q2-diversity` plugin. Use this as an opportunity to refer to other plugins as learning examples.

1.1.9 Add a Pipeline with parallel computing support

In this chapter we'll add a second *Pipeline* to our plugin, and then we'll add parallel computing support to that `Pipeline`. This will enable your users to utilize multi-processor computers or multi-node high performance computing infrastructure (e.g., computer clusters) to analyze their data.

tl;dr

1. The complete code that I wrote to add the second `Pipeline` to my plugin can be found here: https://github.com/caporaso-lab/q2-dwq2/commit/d0d5f38ca6d2e8cdc647660db8d1923b048f8e1e.
2. The code that I developed to add parallel computing support to the new `Pipeline` can be found here: https://github.com/caporaso-lab/q2-dwq2/commit/4ed7e01a6da9a10e7ddf1956877cf494740e35cd.
3. Finally, I added a usage example that can be used to compare the performance of serial and parallel runs of the new Pipeline here: https://github.com/caporaso-lab/q2-dwq2/commit/590263ee9bb8c48df09fe62c0e966acfa99f9aff. Because the third commit includes (a small amount) of real data, as opposed to the toy-sized data we've been using in unit tests, it increases the runtime of the test suite considerably (from about 30 seconds to about 6 minutes, on the M3 MacBook Pro that that I'm developing on).

Add a local alignment search `Pipeline`

The first goal of this section is to define a new `Pipeline` that performs a local alignment search and then tabulates those results for the user to view. This is effectively the same goal as the ubiquitous bioinformatics tool, BLAST [5]. The underlying theory, and the implementation that we'll use here, are presented in the *Sequence Homology Searching* chapter of An Introduction to Applied Bioinformatics [3].

Briefly, given one or more *query sequences* and one or more *reference sequences*, a local alignment search uses pairwise sequence alignment to identify the most similar reference sequence for each query sequence. In our implementation, as in the BLAST implementation, we'll allow a query sequence to match a subsequence of a reference sequence (this is useful, for example, if the query sequence represents a fragment of gene, and the reference sequences represent full-length genes). To achieve this, the underlying alignment algorithm that we'll use is Smith-Waterman local pairwise alignment [6], which is presented in the *Pairwise Sequence Alignment* chapter of An Introduction to Applied Bioinformatics.

The output for a typical local alignment search is a table of the results. Depending on what the user requests, this can describe simply the best match in the database for each query (often defined as the reference sequence which obtains the highest scoring pairwise alignment with the query sequence), or a reverse sorted list of the matches, such that for each query sequence the `n` best matches are presented in order from best to worst. Depending on the implementation, some additional information may be provided about each alignment including the percent similarity between the aligned query and reference sequence, the length of the alignment, and the score of the alignment.

To add a local alignment search `Pipeline` to our plugin, we're going to add one new `Method` and one new `Visualizer`. The method will be called `local_alignment_search`, and it will take one or more query sequences and one or more reference sequences as inputs, and it will accept several parameters used to control the behavior of the alignment algorithm and the result tabulation. As an output, it will generate a tabulation of the search results, grouped by query id and sorted in descending order of alignment score. This table will be an Artifact of a new class, `LocalAlignmentSearchResults`, such that these results could be used by another action (e.g., one that associates metadata with the query sequences, such as taxonomic origin or gene function, based on metadata associated with their best matching reference sequences). This means that we'll also create a new format, some new transformers, and a new artifact class in support of the new functionality.

The new visualizer, `tabulate_las_results` (where `las` is an abbreviation of *local alignment search*) will take a `LocalAlignmentSearchResults` artifact as input and will produce a user-friendly view of it.

Finally, the new pipeline, `search_and_summarize`, will link `local_alignment_search` and `tabulate_las_results` together, returning both the `LocalAlignmentSearchResults` data artifact and the human-readable visualization.

Since you've already done all of this type of work before for other functionality in your plugin, we won't go through this in detail. Use this as an opportunity to read and interpret QIIME 2 plugin code. Review the code added in https://github.com/caporaso-lab/q2-dwq2/commit/d0d5f38ca6d2e8cdc647660db8d1923b048f8e1e and add it to your plugin.

Add parallel computing support to `search_and_summarize`

QIIME 2's formal support for parallel computing makes use of Parsl, and enables developers to create parallel `Pipelines` that can run on compute resources ranging from multi-core laptops to multi-node high performance computer clusters. Parallel `Pipelines` follow the *split-apply-combine* strategy. In the *split* stage, input data is divided into smaller data sets, generally of the same type as the input. Then, in the *apply* stage, the intended operation of the `Pipeline` is applied to each of the smaller data sets in parallel. Finally, in the *combine* stage, the results of each *apply* operation are integrated to yield the final result of the operation as a single data set, generally of the same type as the output of each *apply* operation.

In the context of `search_and_summarize`, this can work as follows (see *the flow chart for a visual summary*). In the *split* stage, the query sequences can be divided into roughly equal sized splits of sequences, such that each query sequence appears in only one split. In the *apply* stage, the reference sequences and each split of query sequences can be provided as the input in parallel calls to the `local_alignment_search` method. Each parallel call will result in a tabular output of search results for its split of the query sequences against the full set of reference sequences. This has the effect of reducing the number of query sequences that are searched against the reference in a given call to `local_alignment_search`. Finally, in the *combine* stage, each of the tabular outputs are joined, and the resulting table is sorted and filtered to the top `n` hits per query. This enables the slow step in this workflow - `local_alignment_search` - to be run in parallel on different processors.

Parsl takes care of all of the work of sending each *apply* job out to different processors and monitoring them to see when they're all done. Our work on the plugin development side, after we've already defined the *apply* operation (`local_alignment_search`, in this example), is to define the actions that will perform the *split* and *combine* operations. These operations will be new QIIME 2 `Methods`.

i ***split-apply-combine* flowchart for search and summarize**

Defining a *split* Method

The *split*, *apply*, and *combine* actions are all QIIME 2 Methods, like any others. Our *split* method will build on a function that takes sequences as input, along with a variable defining the number of sequences that should go in each split, and it will result a dictionary mapping an arbitrary split identifier to a split of sequences. This can look like the following:

```
# Store the default values for `split_sequences` in a dict, so we can
# reference them in multiple places.
_split_seqs_defaults = {
    'split_size': 5
}

def split_sequences(
        seqs: DNAIterator,
        split_size: int = _split_seqs_defaults['split_size']) \
        -> DNAIterator:
    result = {i : DNAIterator(split)
              for i, split in enumerate(_batched(seqs, split_size))}
    return result
```

Here, the input DNAIterator is passed to a function, `_batched`, which yields subsets of the input iterator each with `split_size` sequences. (If the number of input sequences isn't a multiple of `split_size`, the last iterator in `result` will have fewer than `split_size` sequences.) The `_batched` function does most of the work of splitting up the sequences here. Referring to the `_methods.py` file in https://github.com/caporaso-lab/q2-dwq2/commit/4ed7e01a6da9a10e7ddf1956877cf494740e35cd, add the `split_sequences` and `_batched` functions to your `_methods.py` file. This is also a good time to write unit tests for your `split_sequences` function. I recommend designing and implementing the unit tests yourself, but you can also refer to the tests that I wrote in `test_methods.py`.

i **Should our sequence splitter take the size of each split or the number of splits to create as input?**

Defining how the splits should be created is a design decision that the developer must consider.

If having the user provide the number of sequences that should go into each split (as we are here), they will need to know the length of `seqs` to make the best decision for the computer where they will run the job. For example, if they have 16 processors available to them, it might make the most sense to have around 15 splits, so they'll need to

know the number of sequences they're providing as input and divide that by 15 (rounding fractional values up to the next whole number).

If instead, we have the user define how many splits they want (e.g., by providing a `num_splits` value), the computer will need to know how many sequences to put in each split, so it will need to know the length of `seqs` to make a good decision. That will require either taking two passes through `seqs` - first to find out how many sequences there are, and second to do the actual splitting - or reading the entirety of `seqs` into memory (e.g., calling `list(seqs)`). The first of these two options can be slow if `seqs` is long, and the second can be problematic if the size of `seqs` is larger than the amount of available RAM.

The best option may be to let the user choose between these two approaches by either providing `split_size` or `num_splits`. If providing `num_splits` will be problematic due to runtime or required memory, they can opt to figure out the `split_size` themselves and provide it.

For the sake of this example I'm going with the simplest option of just requiring the user to provide the `split_size`. As an exercise after completing this section, implement the alternative approach of allowing the user to either provide the `split_size` or `num_splits`.

Registering `split_sequences`

`split_sequences` will be used in our `search_and_summarize` Pipeline, and we therefore need to register it as a new Action on our plugin. This is done in the typical way, and at this point you're probably getting fairly comfortable with this.

The one thing that is needed here, but which we haven't encountered yet in other Actions, is allowing for an arbitrary number of outputs: the individual splits of sequences. The combination of the number of sequences in an input and the user-specified `split_size` will define how many outputs will be generated by a call to this action, so it's not possible for us to determine this when registering the action. We therefore define our output type as a `Collection` of `FeatureData[Sequence]` artifacts, which we annotate as `Collection[FeatureData[Sequence]]` in the `outputs` dictionary. To achieve this, import `Collections` from `qiime2.plugin` at the top of your `plugin_setup.py` file. In your call to `plugin.methods.register_function`, you can then pass:

```
outputs={'splits': Collection[FeatureData[Sequence]]}
```

Using that tip to handle the generation of an arbitrary number of outputs, refer to the other actions that you've registered, and start implementing the registration step without referring to the code in `q2-dwq2`. Once you've done that and have it working, take a look at my call to `plugin.methods.register_function` in the `plugin_setup.py` file of `q2-dwq2` to see if you missed anything that I included (I added the code in this commit: https://github.com/caporaso-lab/q2-dwq2/commit/4ed7e01a6da9a10e7ddf1956877cf494740e35cd).

Defining and registering a *combine* method

Our *combine* method is effectively performing the opposite operation relative to our *split* method. In this case, we need a method that takes multiple `LocalAlignmentSearchResults` Artifacts as input, and returns them in a single, combined `LocalAlignmentSearchResults` Artifact. Let's create a new `combine_las_reports` function.

```
def combine_las_reports(reports: pd.DataFrame) -> pd.DataFrame:
    results = pd.concat(reports.values())
    return results
```

In this function, `reports`, which is a dict of one or more pandas `DataFrames`, are combined using the `pandas.concat` function, which concatenates `DataFrames` in the order in which they're received. The resulting object is a single `DataFrame`, which we'll return from this function.

> **Note**
>
> Something you might notice when looking at this function's signature is that the *view* type for the collection of local alignment search reports (the `reports` variable) is `pd.DataFrame`, not a collection of `DataFrames` as you might expect. QIIME 2 doesn't differentiate between single inputs and collections of inputs in the type annotations of functions to be registered as Actions. This was also apparent above, when `split_sequences` was annotated as returning a `DNAIterator`, which will always be a Python dictionary where `DNAIterator` objects are the values. This is just something to be aware of.

The last remaining things to do before we parallelize `search-and-summarize` is to write unit tests of `combine_las_reports`, and register a `combine-las-reports` Method in the plugin. Complete these steps as an exercise now, and then refer to my code (https://github.com/caporaso-lab/q2-dwq2/commit/4ed7e01a6da9a10e7ddf1956877cf494740e35cd) to check your work.

One thing that I hope is evident from this section is that *split* and *combine* functions tend to be pretty simple. These operations can be more complex in some cases (at some point, I'll add an example of that), but often they're very simple.

Update `search-and-summarize` to use *split* and *combine* Methods

We're now ready to update `search-and-summarize` to run in parallel.

First, we'll add the `split_size` parameter to `search_and_summarize`, so we can pass it through to `split_sequences`.

```
def search_and_summarize(
    ...
    split_size=_split_seqs_defaults['split_size'],
    ...
```

Then, inside the function, we'll get the new Actions we just defined:

```
...
split_action = ctx.get_action('dwq2', 'split_sequences')
combine_action = ctx.get_action('dwq2', 'combine_las_reports')
...
```

Then, we'll apply the *split* action to our input sequences:

```
query_splits, = split_action(query_seqs, split_size=split_size)
```

And next, the interesting bit happens. We start by defining a list (`las_results`) to collect our local alignment search results for each split of the query sequences. Then we iterate over the splits, *applying* the local alignment search method to each split, and appending the results in the `las_results` list that we just created.

```
las_results = []
for q in query_splits.values():
    las_result, = las_action(
        q, reference_seqs, n=n, gap_open_penalty=gap_open_penalty,
        gap_extend_penalty=gap_extend_penalty, match_score=match_score,
        mismatch_score=mismatch_score)
    las_results.append(las_result)
```

Finally we *combine* all of the individual results, and from this point everything proceeds as it did before we added parallel support.

```
las_results, = combine_action(las_results)
```

Under the hood is where the magic happens here. Because we're iterating over a collection of QIIME 2 Artifacts (the `for` loop above) in a QIIME 2 `Pipeline`, each `las_result` object is a proxy for a real QIIME 2 artifact. The jobs to create them are distributed as indicated in the user's *QIIME 2 parallel configuration*. The code continues executing as if these were real Artifacts, not proxy Artifacts, until something is needed from them. At that point, the code will block (i.e., wait) until the proxy Artifacts become real Artifacts, and then continue processing.

Pipeline resumption

A cool feature that you get for free here is *pipeline resumption*. If your Pipeline is interrupted mid-run (for example, because the jobs ran out of alloted time or memory on the shared compute cluster they're running on), users can restart the job and all `Results` that were already computed will be recycled and not need to be computed again. This can save your users a lot of time and frustration, and reduce unnecessary utilization of compute resources and the energy used to power that computation.

The last steps before this is ready to use are to update the parameters to `search_and_summarize` in `plugin_setup.py`, and to add a test of the parallel execution of the Pipeline. Make the necessary changes to `plugin_setup.py`, and then we'll add a new unit test.

Testing the parallel Pipeline

To test that the parallel functionality works as expected, we'll make some adaptations to our test of the Pipeline. Specifically, we'll reuse most of our test of the serial functionality and confirm that when the tests are run in parallel they still work as expected.

There are a lot of changes in the `test_pipelines.py` file in the commit associated with this section, but most of them are just refactoring the code to reuse the input Artifacts and expected test results. First, I added a `setUp` method to the `SearchAndSummarizeTests`. `setUp` is a special method that runs before each individual test function, so it's useful for sharing information across tests. In this new function, the first thing I do is call `super().setUp()`, which calls the `setUp` function on the base class (`TestPluginBase` in this case), if one exists. This ensures that any upstream configuration is still happening. Then, I moved the code from `test_simple1` that accessed the `Pipeline` and created the input `Artifacts` to this function and set all of these as attributes on the `SearchAndSummarizeTests` class. Now I can access these via the `self` variable in the methods of this class. After all of these changes, my `class` definition starts as follows:

```
class SearchAndSummarizeTests(TestPluginBase):
    package = 'q2_dwq2.tests'

    def setUp(self):
        super().setUp()

        self.search_and_summarize_pipeline = \
            self.plugin.pipelines['search_and_summarize']
        query_sequences = [skbio.DNA('ACACTCTCCACCCATTTGCT',
                                     metadata={'id': 'q1'}),
                           skbio.DNA('ACACTCACCACCCAATTGCT',
                                     metadata={'id': 'q2'})]
        query_sequences = DNAIterator(query_sequences)
        self.query_sequences_art = qiime2.Artifact.import_data(
            "FeatureData[Sequence]", query_sequences, view_type=DNAIterator
        )
        reference_sequences = [
```

(continues on next page)

(continued from previous page)

```
            skbio.DNA('ACACTCACCACCCAATTGCT', metadata={'id': 'r1'}),  # == q2
            skbio.DNA('ACACTCTCCACCCATTTGCT', metadata={'id': 'r2'}),  # == q1
            skbio.DNA('ACACTCTCCAGCCATTTGCT', metadata={'id': 'r3'}),
        ]
        reference_sequences = DNAIterator(reference_sequences)
        self.reference_sequences_art = qiime2.Artifact.import_data(
            "FeatureData[Sequence]", reference_sequences, view_type=DNAIterator
        )
```

The next thing I did was define a helper function that compares the observed results to the expected results. Like the creation of the Artifacts above, this code was also moved from `test_simple1`.

```
    def _test_simple1_helper(self, observed_hits, observed_viz):
        expected_hits = pd.DataFrame([
          ['q1', 'r2', 100., 20, 40., 'ACACTCTCCACCCATTTGCT',
                                       'ACACTCTCCACCCATTTGCT'],
          ['q1', 'r3', 95., 20, 35., 'ACACTCTCCACCCATTTGCT',
                                      'ACACTCTCCAGCCATTTGCT'],
          ['q1', 'r1', 90., 20, 30., 'ACACTCTCCACCCATTTGCT',
                                      'ACACTCACCACCCAATTGCT'],
          ['q2', 'r1', 100., 20, 40., 'ACACTCACCACCCAATTGCT',
                                       'ACACTCACCACCCAATTGCT'],
          ['q2', 'r2', 90., 20, 30., 'ACACTCACCACCCAATTGCT',
                                      'ACACTCTCCACCCATTTGCT'],
          ['q2', 'r3', 85., 20, 25., 'ACACTCACCACCCAATTGCT',
                                      'ACACTCTCCAGCCATTTGCT']],
         columns=['query id', 'reference id', 'percent similarity',
                  'alignment length', 'score', 'aligned query',
                  'aligned reference'])
        expected_hits.set_index(['query id', 'reference id'], inplace=True)

        pdt.assert_frame_equal(observed_hits.view(pd.DataFrame), expected_hits)

        # observed_viz is a qiime2.Visualization.
        # access its index.html file for testing.
        index_fp = observed_viz.get_index_paths(relative=False)['html']
        with open(index_fp, 'r') as fh:
            observed_index = fh.read()
            self.assertIn('q1', observed_index)
            self.assertIn('q2', observed_index)
            self.assertIn('r1', observed_index)
            self.assertIn('r2', observed_index)
            self.assertIn('ACACTCACCACCCAATTGCT', observed_index)
            self.assertIn('ACACTCTCCACCCATTTGCT', observed_index)
```

Then, I modified the name of `test_simple1` to `test_simple1_serial`, and adapted it to use the pipeline and artifact attributes, and to call `_test_simple1_helper` to compare the observed and expected results.

```
def test_simple1_serial(self):

    observed_hits, observed_viz = self.search_and_summarize_pipeline(
        self.query_sequences_art, self.reference_sequences_art)
    self._test_simple1_helper(observed_hits, observed_viz)
```

Finally, I defined a new test function `test_simple1_parallel`, which calls the same Pipeline with the same inputs, and compares the observed output to the expected output in the same way as `test_simple1_serial`. The only difference is that in this test `search_and_summarize` is running in parallel instead of serially. This is achieved by

using the `with ParallelConfig()` context manager, and calling the Pipeline using its `.parallel` method.

Add the following import to the top of your `test_pipeline.py` file:

```
from qiime2.sdk.parallel_config import ParallelConfig
```

Then add the following test function:

```
def test_simple1_parallel(self):
    with ParallelConfig():
        observed_hits, observed_viz = \
            self.search_and_summarize_pipeline.parallel(
                self.query_sequences_art, self.reference_sequences_art)
        self._test_simple1_helper(observed_hits, observed_viz)
```

After making all of the changes described here to your `_pipelines.py` and `test_pipelines.py` files, run the test suite with `make test`. If all tests pass, you should be good to go. If not, compare your code to mine (https://github.com/caporaso-lab/q2-dwq2/commit/4ed7e01a6da9a10e7ddf1956877cf494740e35cd) to see what's different.

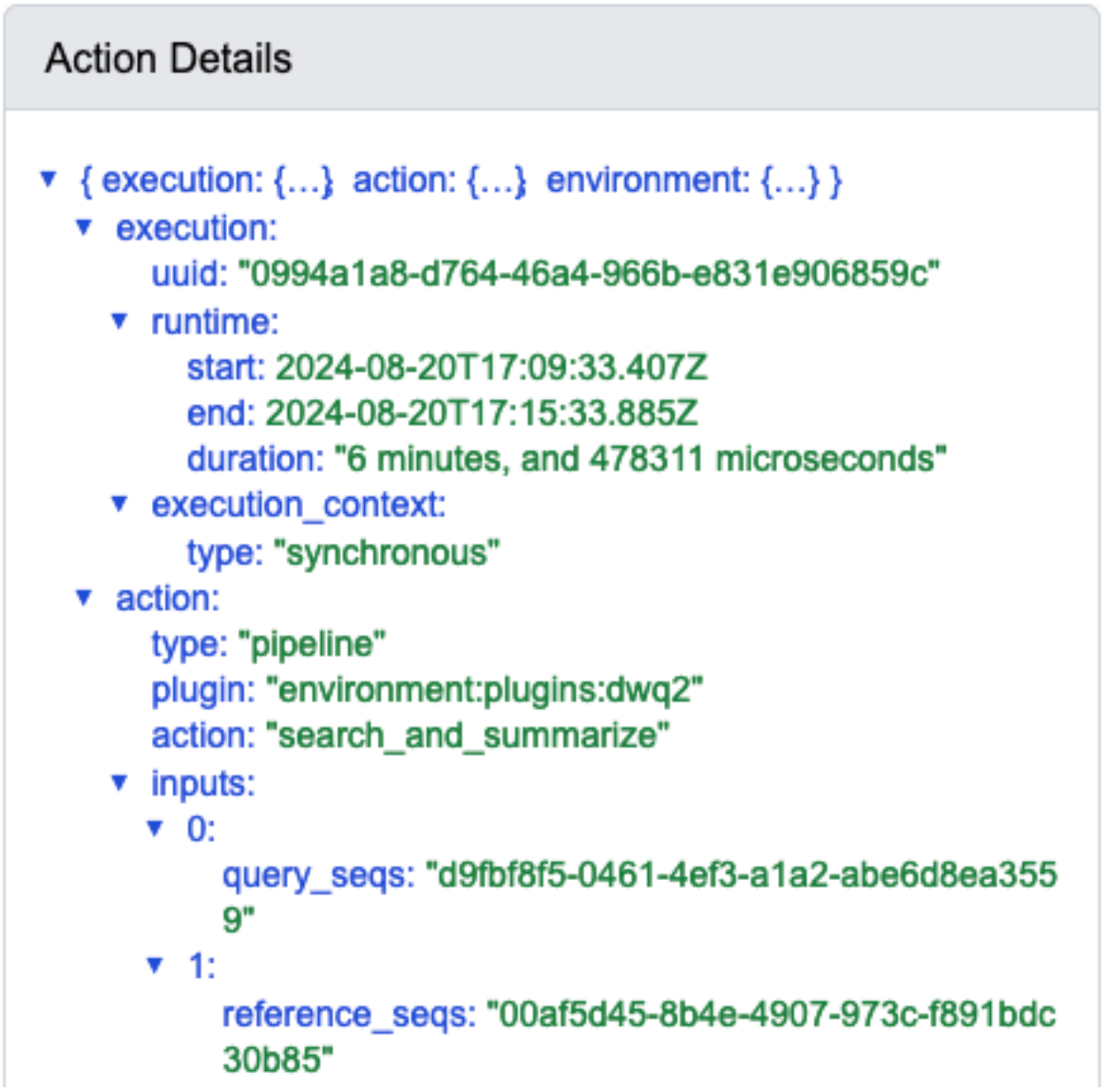
Compare the serial versus parallel run times of the `search-and-summarize`

Now that we've added parallel computing support to our `search-and-summarize` Pipeline, we can try it out to see how it impacts run time. To do this, we need a data set that is big enough that the benefits of parallelization outweigh the overhead associated with parallelization. For example, the splitting of the query sequences and the combining of the search result tables each take some time - if our input data is tiny, that time might be longer than the time to just compute the results serially, so running the code in parallel wouldn't result in a noticeable decrease in runtime (the parallel runtime might even be larger than the serial runtime).

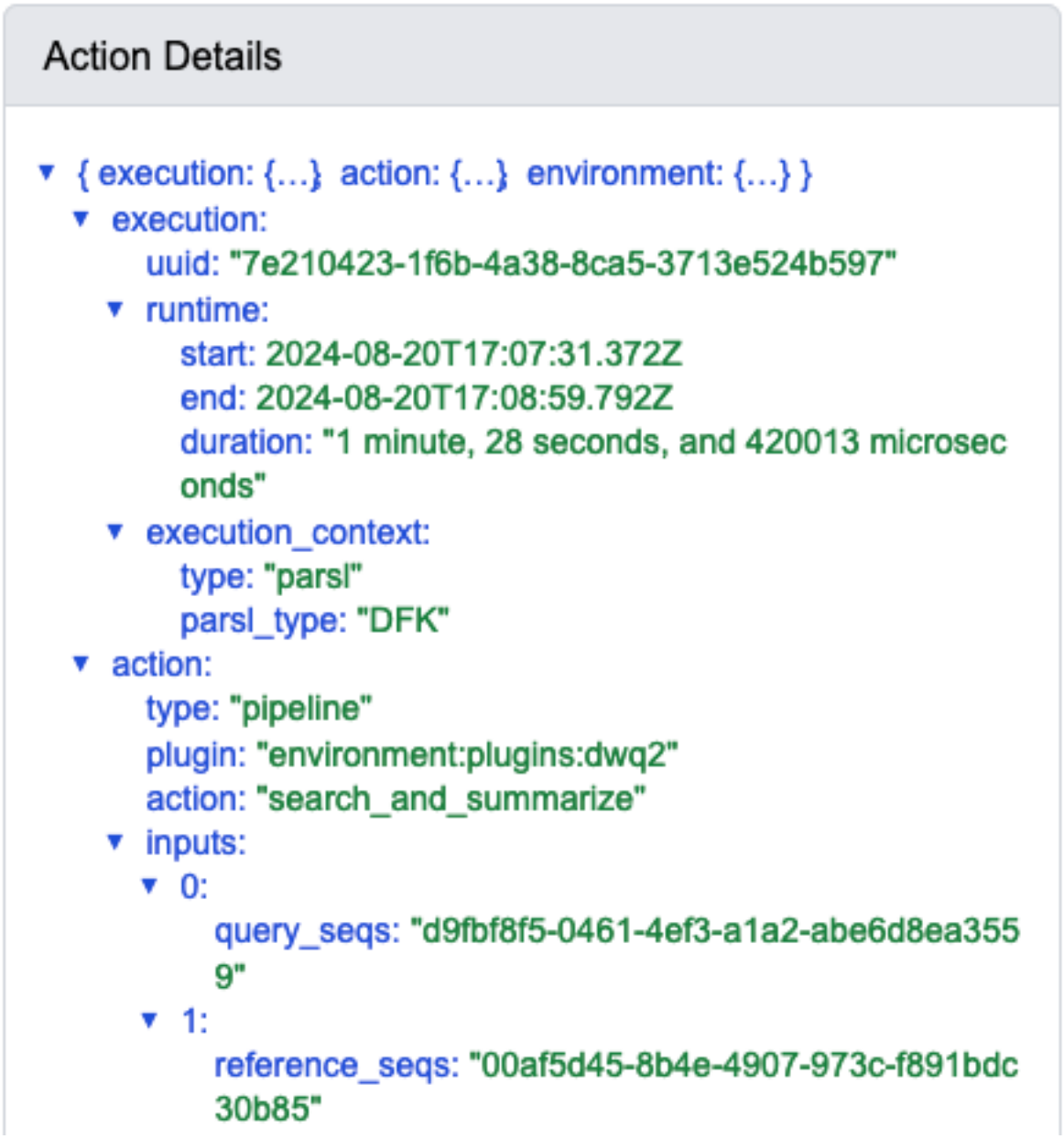
In a third commit associated with this section (https://github.com/caporaso-lab/q2-dwq2/commit/590263ee9bb8c48df09fe62c0e966acfa99f9aff), I added a usage example to our new Pipeline that uses a larger data set. Note that integrating this usage example does considerably increase the runtime of the unit tests. I suggest trying it out as-is, but after experimenting with it you can feel free to reduce the number of query (to 2, for example) and reference sequences (to 3, for example) to reduce the runtime, or just keep it as-is - it's up to you.

Refer to this commit to integrate the usage example into your code. Then, run the usage example as is to get a feel for its serial run time. After that, call the usage example again providing the `--parallel` parameter to get a feel for its parallel run time.

When I do this, I observe the following run times associated with the data provenance of this step:


Action Details
{ execution: {…} action: {…} environment: {…} }
execution:
uuid: "0994a1a8-d764-46a4-966b-e831e906859c"
runtime:
start: 2024-08-20T17:09:33.407Z
end: 2024-08-20T17:15:33.885Z
duration: "6 minutes, and 478311 microseconds"
execution_context:
type: "synchronous"
action:
type: "pipeline"
plugin: "environment:plugins:dwq2"
action: "search_and_summarize"
inputs:
0:
query_seqs: "d9fbf8f5-0461-4ef3-a1a2-abe6d8ea3559"
1:
reference_seqs: "00af5d45-8b4e-4907-973c-f891bdc30b85"

Serial!



Parallel!

1.1.10 Integrate metadata in Actions

We've done a lot so far in q2-dwq2, but we've left out one thing that is common in almost all *real* plugins: the integration of metadata in Actions. In this relatively brief section, we'll enable users to optionally provide metadata associated with reference sequences to the tables generated by the `tabulate-las-results` Visualizer and the `search-and-summarize` Pipeline. This metadata could be just about anything. For example, reference metadata could be taxonomic information about the reference sequences, enabling viewers of the resulting visualization to infer the taxonomic origin of their query sequences from the taxonomy associated with the closest matching reference sequences. (That example is also used in the *Sequence Homology Searching* chapter of An Introduction to Applied Bioinformatics [3], so you can refer there if the idea isn't familar.)

Let's get started.

tl;dr

The code that I wrote for this section can be found here: https://github.com/caporaso-lab/q2-dwq2/commit/153d6a21a2fff54cdf934560ad26832cf9946aff.

The input metadata

In this example, the reference metadata will be a QIIME 2 metadata file that has reference sequence ids as its identifiers, and some number of additional columns containing the actual metadata. For example, the tab-separated text (`.tsv`) metadata file could look like:

```
id        phylum        class genus        species
ref-seq1        Bacteria        Bacillota        Bacillus_A        paranthracis
ref-seq2        Bacteria        Pseudomonadota        Photobacterium        kishitanii
ref-seq3        Bacteria        Actinomycetota        Cryptosporangium        arvum
```

This metadata file will be passed into `tabulate_las_results` along with the `LocalAlignmentSearchResults` artifact that it already takes.

Note

By convention in QIIME 2, all relevant identifiers (in our case, those that could be present in the `LocalAlignmentSearchResults` artifact) should be represented in the metadata, or an error should be raised. However, "extra" identifiers in the metadata that do not show up in the corresponding artifacts (again, the `LocalAlignmentSearchResults` in this example) are allowed.

The goal of this convention is that we want to facilitate users maintaining only a single metadata file. If an upstream artifact - in this case the sequences in the `FeatureData[Sequence]` artifact that is passed into `local-alignment-search` as `reference_seqs` - is ever filtered to remove some sequences, the user shouldn't have to create a filtered version of their metadata file. The reason for this is two-fold. First of all, that would generally be an extra, unnecessary step, so it complicates users' workflows. But, more importantly, that could result in a proliferation of metadata files that need to be kept in sync. This is another application of the *DRY* principle: don't encourage your users to duplicate information represented in their metadata file.

Add an optional input to `tabulate_las_results`

Now that we know what the metadata will look like, let's add a new optional input to our `tabulate_las_results` function. The new input we add will be called `reference_metadata`, and it will be received as a `qiime2.Metadata` object. Our updated function signature in `_visualizers.py` will look like this:

```
def tabulate_las_results(output_dir: str,
                         hits: pd.DataFrame,
                         title: str = _tabulate_las_defaults['title'],
                         reference_metadata: qiime2.Metadata = None) \
                         -> None:
```

If `reference_metadata` is provided, we'll take a few steps to integrate it into `hits` before we write out our HTML file. Otherwise, `tabulate_las_results` will behave exactly as it did before. I added the following `if` block:

```
if reference_metadata is not None:
    reference_metadata = reference_metadata.to_dataframe()
```

(continues on next page)

(continued from previous page)

```
    hits.reset_index(inplace=True)

    metadata_index = reference_metadata.index.name
    metadata_columns = reference_metadata.columns.to_list()
    reference_metadata.reset_index(inplace=True)

    missing_ids = \
        set(hits['reference id']) - set(reference_metadata[metadata_index])
    if len(missing_ids) > 0:
        raise KeyError(
            f"The following {len(missing_ids)} IDs are missing from "
            f"reference metadata: {missing_ids}.")

    hits = pd.merge(hits, reference_metadata,
                    left_on='reference id',
                    right_on=metadata_index,
                    how='inner')

    hits.set_index(['query id', 'reference id'], inplace=True)
    column_order = \
        ['percent similarity', 'alignment length', 'score'] + \
        metadata_columns + ['aligned query', 'aligned reference']
    hits = hits.reindex(columns=column_order)
```

This looks like a lot, but it's really a few simple actions.

1. First, convert the `qiime2.Metadata` object to a `pd.DataFrame` using the built-in method on `qiime2.Metadata`.
2. Then, remove the index from `hits` to prepare it for a `pd.merge` operation.
3. Next, cache the metadata's index name because - importantly - we don't know exactly what this index name will be. The QIIME 2 metadata format allows for a few available options, including `id`, `sample-id`, `feature-id`, and several others, and we don't want to restrict a user to providing any specific one of these. We also cache the list of metadata columns, and remove the index to prepare it for the `pd.merge` operation with `hits`. Remember that we also don't know what metadata columns the user will provide, and in this example we're not putting any restrictions on this.
4. Then, we confirm that all ids that are represented in `hits` are present in `reference_metadata`, and we throw an informative error message if any are missing. Our error message should help a user identify what's wrong, so I chose to indicate how many ids were missing, and provide a list of them.
5. Next, we merge our hits and the reference metadata on the index of `hits` and the index of our metadata. Unlike for the metadata, we *do* know what the index name of `hits` will be because we were explicit about this when we defined our `LocalAlignmentSearchResultsFormat`, so we can refer to it directly. This is an important distinction that differentiates metadata from *File Formats* we define: if we need the flexibility to allow for arbitrary column names, we're generally working with metadata. On the other hand, if our column names are predefined, we should generally be working with a File Format.
6. Then, we prepare the `hits` DataFrame for use downstream. To do this, we first set the MultiIndex. I also sorted the columns such that all of the metadata columns come between the *percent similarity*, *alignment length*, and *score* columns and the *aligned query* and *reference query* columns of the original `hits` DataFrame. When developing, I found this column order to be useful for reviewing the results.

After this, our action proceeds the same as if no metadata was provided.

```
with open(os.path.join(output_dir, "index.html"), "w") as fh:
    fh.write(_html_template % (title, hits.to_html()))
```

Update `search-and-summarize`

We'll also want this option to be available to users of `search-and-summarize`. Since `tabulate-las-results` is already part of the `search-and-summarize` Pipeline, all we need to do in our `_pipelines.py` file is add the new optional parameter (`reference_metadata`), and pass it through in our call to `tabulate_las_results_action`. Try to do that yourself, and refer to my code (https://github.com/caporaso-lab/q2-dwq2/commit/153d6a21a2fff54cdf934560ad26832cf9946aff) as needed.

Update `plugin_setup.py`

Next, we'll need to make the plugin aware of this new parameter when we register the `tabulate-las-results` and `search-and-summarize` actions. Metadata files are provided as *Parameters* of type `qiime2.plugin.Metadata` on action registration.

In `plugin_setup.py`, add `Metadata` to the list of imports from `qiime2.plugin`. Then, add the new parameter and description to the dictionaries we created to house these for `tabulate-las-results`.

```
_tabulate_las_parameters = {'title': Str,
                            'reference_metadata': Metadata}
_tabulate_las_parameter_descriptions = {
    'title': 'Title to use inside visualization.',
    'reference_metadata': 'Reference metadata to be integrated in output.'
}
```

Recall that we reuse these dictionaries when we register `search-and-summarize`, so we only need to add this parameter and description in this one place.

At this point, you should be ready to use this new functionality with your plugin. Open a terminal in a environment where your implementation of `q2-dwq2` is installed, run `qiime dev refresh-cache`, and you should see the new parameter in calls to `qiime dwq2 tabulate-las-results --help` and `qiime dwq2 search-and-summarize --help`.

Add unit tests and update the `search-and-summarize` usage example

Finally, wrap this up by adding new unit tests for the functionality. Do this on your own, and then refer my code (https://github.com/caporaso-lab/q2-dwq2/commit/153d6a21a2fff54cdf934560ad26832cf9946aff) to see how I did it.

In my code for this section, you'll also find a metadata file that I provided which corresponds to the example reference sequences that were provided. Use that file to update the `search-and-summarize` usage example, and then test your code on the command line with the new usage example.

1.1.11 Conclusion

That's the end of the QIIME 2 plugin development tutorial, for now. As of this writing (22 August 2024), this tutorial is still in development, and new content is planned. You can see a list of content planned for the plugin tutorial on our issue tracker here, and more general content planned for the *Plugins* part of *Developing with QIIME 2* here.

A good next step is to create your own plugin. The easiest way to do that is to go back to the first section of the tutorial, *Create your plugin from a template*, and create your new plugin from the cookiecutter template. Start working on adding your methods and visualizers, and don't forget your unit tests and usage examples! Come join the QIIME 2 Forum Developer Discussion if you need help, have feedback on this tutorial, or have more general feedback on developing QIIME 2 plugins.

Thanks for reading and for your interest in QIIME 2 - happy coding!

1.2 How-To Guides

This chapter of the documentation provides instructions for achieving specific goals with your plugin.

> **Note**
>
> If you've never built a QIIME 2 plugin before, and would like step-by-step instructions for building an example plugin before creating your own, you'll find those instructions in the *tutorial on building your first QIIME 2 plugin*. If you're already comfortable with plugin development and are looking for instructions to achieve a specific task, you may find more targeted instructions in the *Plugin Development How-To Guides*.

- *Set up your development environment*
- *Distribute plugins on GitHub*
- *Provide technical support for your users*
- *Maximize compatibility between your plugin(s) and existing QIIME 2 distribution(s)*
- *Facilitating installation of your plugin for users*
- *Automate testing of your plugin*
- *Publicize your QIIME 2 plugins (or other QIIME 2-based tools)*
- *Register a QIIME 2 plugin*
- *Create and register a Method*
- *Create and register a visualizer*
- *Create and register a pipeline*
- *Creating and registering a Transformer*
- *Use Artifact Collections as Action inputs or outputs*
- *How to play nicely with other plugins*
- *How to use Metadata*
- *How to test QIIME 2 plugins*
- *Writing Usage Examples*
- *Defining different Format validation levels*
- *Handling exceptions in parallel Pipelines*

1.2.1 Set up your development environment

This how-to guide describes how to install and set up a QIIME 2 development environment. The development environment you create will be suitable for creating new or contributing to existing QIIME 2 plugins or interfaces, or contributing to the development of the QIIME 2 framework.

> **Warning**
>
> QIIME 2 support for Windows is restricted to Windows Subsystem for Linux (WSL). The QIIME 2 development team doesn't have much experience developing QIIME 2 in the context of Windows. If you're comfortable with creating development environments with WSL, we expect that it should work just fine. If you expect that you might need help

setting up your development environment, we'll be able to provide better assistance if you're developing on a Linux or macOS system.

Install Prerequisites

Miniconda provides the `conda` environment and package manager, and is currently the only supported way to install QIIME 2. Follow the instructions for downloading and installing Miniconda.

After installing Miniconda and opening a new terminal, make sure you're running the latest version of `conda`:

```
conda update conda
```

Install the latest development version of the QIIME 2 "Tiny Distribution"

The QIIME 2 "Tiny Distribution" is a minimal set of QIIME 2 functionality for building and using plugins through the QIIME 2 command line, and is intended for use by developers who want a minimal QIIME 2 environment to work in.

Note

We recommend starting your development with the "Tiny Distribution", unless you specifically need plugins that are installed in other QIIME 2 distributions, such as the "Amplicon" or "Metagenome" distributions, in which case see *Installing other QIIME 2 distributions*.

macOS

```
conda env create -n q2dev-tiny --file https://raw.githubusercontent.com/qiime2/
 ↪distributions/dev/latest/passed/qiime2-tiny-macos-latest-conda.yml
```

Linux

```
conda env create -n q2dev-tiny --file https://raw.githubusercontent.com/qiime2/
 ↪distributions/dev/latest/passed/qiime2-tiny-ubuntu-latest-conda.yml
```

macOS (Apple Silicon)

```
CONDA_SUBDIR=osx-64 conda env create -n q2dev-tiny --file https://raw.
 ↪githubusercontent.com/qiime2/distributions/dev/latest/passed/qiime2-tiny-macos-
 ↪latest-conda.yml
conda config --env --set subdir osx-64
```

Activate the `conda` environment

You can now activate the environment you just created as follows.

```
conda activate q2dev-tiny
```

To test your QIIME 2 environment, run:

```
qiime info
```

You should see something like the following, though the version numbers you'll see will be different:

```
System versions
Python version: 3.8.18
QIIME 2 release: 2023.11
QIIME 2 version: 2023.11.0.dev0+15.g8ac7e3e
q2cli version: 2023.11.0.dev0+12.g7cf7a7a

Installed plugins
types: 2023.11.0.dev0+2.g1827eab

Application config directory
/Users/gregcaporaso/miniconda3/envs/q2dev-tiny/var/q2cli

Getting help
To get help with QIIME 2, visit https://qiime2.org
```

The versions listed here, for QIIME 2, q2cli, and q2-types, are development versions as defined by versioneer, and these indicate that you're working in a QIIME 2 development environment (as opposed to working with a specific release version of QIIME 2).

At this stage you should now have a working development environment - time to start hacking!

Next steps

Building your first plugin

If you're creating your first plugin, you can now move on to *Tutorial: A step-by-step guide to building your first QIIME 2 plugin*.

Contributing to existing plugins

If you want to make changes to the QIIME 2 framework, q2cli, or any existing plugins, follow these steps (for the sake of this example, we will focus on the example of contributing to developing `q2-types`):

```
# Grab the package source from the relevant source repository.
git clone https://github.com/qiime2/q2-types
cd q2-types

# Install any additional build-time dependencies needed for this project.
# Check ci/recipe/meta.yaml in any QIIME 2 repository under the QIIME 2 GitHub
# organization for build or test requirements. For example, see
# https://github.com/qiime2/q2-types/blob/dev/ci/recipe/meta.yaml
conda install pytest flake8
```

(continues on next page)

(continued from previous page)

```
# Install local source in "development mode", and build any package assets.
make dev

# Run package tests to ensure that everything is okay.
make test
```

Installing other QIIME 2 distributions

> **Note**
>
> If you install a distribution other than the "Tiny Distribution", be sure that the environment name in your `conda activate` command (in the example above this was `q2dev-tiny`) matches the value that you provided to the `conda env create` command through the `-n` parameter.

Amplicon distribution

macOS

```
conda env create -n q2dev-amplicon --file https://raw.githubusercontent.com/qiime2/
 ↪distributions/dev/latest/passed/qiime2-amplicon-macos-latest-conda.yml
```

Linux

```
conda env create -n q2dev-amplicon --file https://raw.githubusercontent.com/qiime2/
 ↪distributions/dev/latest/passed/qiime2-amplicon-ubuntu-latest-conda.yml
```

macOS (Apple Silicon)

```
CONDA_SUBDIR=osx-64 conda env create -n q2dev-amplicon --file https://raw.
 ↪githubusercontent.com/qiime2/distributions/dev/latest/passed/qiime2-amplicon-macos-
 ↪latest-conda.yml
conda config --env --set subdir osx-64
```

Metagenome distribution

macOS

```
conda env create -n q2dev-metagenome --file https://raw.githubusercontent.com/qiime2/
 ↪distributions/dev/latest/passed/qiime2-metagenome-macos-latest-conda.yml
```

Linux

```
conda env create -n q2dev-metagenome --file https://raw.githubusercontent.com/qiime2/
↪distributions/dev/latest/passed/qiime2-metagenome-ubuntu-latest-conda.yml
```

macOS (Apple Silicon)

```
CONDA_SUBDIR=osx-64 conda env create -n q2dev-metagenome --file https://raw.
↪githubusercontent.com/qiime2/distributions/dev/latest/passed/qiime2-metagenome-
↪macos-latest-conda.yml
conda config --env --set subdir osx-64
```

1.2.2 Distribute plugins on GitHub

GitHub is a straight-forward way to share your QIIME 2 plugin. This tutorial will walk you through creating a QIIME 2 plugin from template, and then sharing it with installation instructions on GitHub.

This tutorial assumes that you have `git` and the GitHub command line interface installed on your computer. If you don't have these installed, do that now using the GitHub command line interface installation instructions, which should install `git` if you don't already have it installed.

> **Note**
>
> All of the steps illustrated here that use the GitHub command line interface can also be performed through the typical GitHub web interface. The GitHub command line interface is used here as it's easier to document and test the instructions (and it's pretty darn cool).

Template your plugin

Create a new template plugin, according to the instructions in *Create your plugin from a template*. Pay attention to the question about the "target distribution" - this enables templating of installation instructions and test machinery against the distribution of your choice. For example, if you want your plugin to expand upon the `amplicon` distribution, this is where you indicate that.

After completing those steps, return to this page.

Share your plugin on GitHub

Change into *your plugin's top-level directory*. To confirm that a local `git` repository was initialized during the templating process, run:

```
git log
```

You should see that there has been one commit to this repository (or more, if you've done additional work on the plugin and committed those changes).

Next, you'll need to log in to GitHub via the command line interface to authenticate. Run the following command, and follow the instructions:

```
gh auth login
```

After successful authentication, from your plugin's top-level directory run `gh repo create`:

```
gh repo create
```

This will ask a series of questions. As of this writing (24 April 2024), the first question is:

```
? What would you like to do?
  Create a new repository on GitHub from scratch
  Create a new repository on GitHub from a template repository
> Push an existing local repository to GitHub
```

You should select the last option as you're going to push your plugin's local `git` repository to GitHub.

Work through the remaining questions. If you don't know how to answer to a specific question, the default is generally what you'll want to select.

After this process completes, you will have a new GitHub repository. If you navigate to that repository in your web browser you should see that the tests are currently running (or recently completed). If those tests pass, follow the instructions in the `README.md` on your repository to test installation of your plugin on your computer. If that works, you should be ready to share those instructions with others.

Expanding on the install instructions

Users should now be able to install and use your plugin if they're pointed at the `README.md` file. The templated *Installation instructions* in the `README.md` file are intended to be a starting point, and they mention that as a note to readers. You'll almost certainly want to update the install instructions for your plugin as you develop it.

Here are some tips related to updating the installation instructions for your plugin:

- If your plugin requires additional dependencies that can be installed with either conda or pip, you can add those in the environment yaml file that was templated in the `environments/` directory.
- We recommend doing what you can in the `Makefile` in your repository, so that the command `make install` continues to be the mechanism by which your plugin should be installed.
- Be sure to update the `README.md` if you introduce any new constraints (e.g., that your plugin can only be installed on Linux). It's fine to do that, but you should let your users know so they don't get grumpy about your plugin.

If you're ready to start getting users, the next steps are *helping prospective users discover your plugin*, and *supporting them as they use it*.

1.2.3 Provide technical support for your users

You are free to have your users request help on the QIIME 2 Forum for your QIIME 2-based tools. This will generally take place under the *Community Plugin Support* category. We encourage this as it helps to centralize knowledge around the QIIME 2 ecosystem, but you are of course also free to direct users to your email, GitHub issue tracker, or another resource for technical support.

Note

Within the next few months (as of 16 August 2024) we plan to begin offer sub-categories under the *Community Plugin Support* category where plugin developers can run their own mini-forum focused around their plugin. We are

developing eligibility criteria that will likely be focused around developer-responsiveness, automated plugin testing benchmarks, and existence of tested plugin documentation. More on this soon.

If you choose to support your users through the QIIME 2 Forum you should monitor the forum to answer questions from your users. You can search the Discourse community forum for information on configuring your notifications - there are lots of options for this.

Please be aware that the QIIME 2 Forum Moderators will not be able to provide support for your plugins, since they won't necessarily be experts in your tools (and are already tied up providing tech support to the community). If we see questions come in related to your tools the moderators will try to reach you if you seem to have not noticed, but please do monitor for activity related to your tools.

We provide and maintain the platform for supporting users, but it's still ultimately your responsibility to provide user support.

1.2.4 Maximize compatibility between your plugin(s) and existing QIIME 2 distribution(s)

You can build your QIIME 2 tools in your own way. Your new tool doesn't need to live in the QIIME 2 GitHub organization or be part of one of the QIIME 2 distributions developed and maintained in the Caporaso Lab.

If you want your QIIME 2 plugin(s) or other tools to work with existing QIIME 2 *distribution(s)*, your focus should be on maximizing compatibility between your plugin(s) and the relevant QIIME 2 distribution(s). To do this, you can observe the *artifact classes* that are used in the target distribution(s), and make your functionality compatible with those. Avoid defining new artifact classes when you can reuse existing ones, to maximize compatibility and interoperability (as well as reducing your own software development time!). A complete list of artifact classes and formats available in a deployment of QIIME 2 can be accessed with the `qiime tools list-types` and `qiime tools list-formats` commands. (Some are missing documentation - we'd love your help addressing that.) If you do need to create new artifact classes, you can add these in your own plugin(s).

The Caporaso Lab is not taking on new responsibility for distributing plugins right now (i.e., integrating them in the distributions they develop and maintain), but is currently (15 August 2024) developing new mechanisms for helping you share your plugin or other tools (see *Publicize your QIIME 2 plugins (or other QIIME 2-based tools)*) that will ultimately replace the QIIME 2 Library.

You can consider the existing distributions to be foundations that you can build on, or you can create and distribute your own conda metapackages. Some guidance on each of these approaches:

- Your install instructions can indicate that a user should install whichever distribution you depend on (e.g., `tiny`, `amplicon`, or `metagenome`) and then illustrate how to install your plugin(s) in that environment however it makes sense (e.g., `conda` or `pip`). Complete install instructions are drafted for you in the `README.md` of plugins that you build using our template (see *Create your plugin from a template*).
- Alternatively, you can compose and share your own distribution of plugins (e.g., building from the `tiny` distribution) that captures the set of functionality you'd like to share.

Either of these approaches is totally fine, with the following caveats. The former is an easier starting point, and will allow us to provide more troubleshooting assistance for any installation issues that users may encounter. While the latter provides you with more flexibility in your environment construction, our assistance with any install issues or environment conflicts that users may run into will be more limited (and will ultimately be your responsibility to troubleshoot and resolve).

With the above information in mind, refer to *Facilitating installation of your plugin for users* for instructions on how to support either of these approaches.

Getting Feedback on your Plugin

You can request feedback on your plugin as a whole from more experienced QIIME 2 developers by reaching out through the QIIME 2 Forum Developer Discussion. However, be cognizant of the fact that doing code review takes a long time to do well: you should only request this when you feel like you have a final draft of the plugin that you'd like to release, and expect that the reviewer may point out that there is a bunch more work that should be done before you release. Please have others who you work closely with – ideally experienced software developers, and even more ideally experienced QIIME 2 plugin developers – review it first. If you have questions along the way, you can ask those whenever - just be sure to review *Developing with QIIME 2* and search the forum in case your question has already been answered previously.

1.2.5 Facilitating installation of your plugin for users

Installing your plugin on top of an existing QIIME 2 Distribution (recommended)

The easiest way to instruct users to install your plugin in the context of an existing QIIME 2 Distribution is to create a conda environment file that they can use to install a specific distribution of QIIME 2 including your plugin, all while using a single command.

In the top-level directory of your plugin, create the following directory:

```
environment-files/
```

Within this directory, create environment file(s) for current and/or past installable versions of your plugin. You can name them with a pattern like `<package-name>-qiime2-<target-distribution>-<target-epoch>.yml` (for example, `q2-dwq2-qiime2-amplicon-2024.5.yml`).

The contents of your environment file should look something like this:

```
channels:
- https://packages.qiime2.org/qiime2/<target-epoch>/<target-distribution>/released
- conda-forge
- bioconda
dependencies:
  - qiime2-<target-distribution>
  - pip
  - pip:
    - <package-name>@git+https://github.com/<owner>/<repository-name>.git@<target-
↪branch>
```

With the following terms defined:

- `<target-epoch>`: the QIIME 2 epoch that your plugin should be installed under (e.g., `2024.5` or `2024.10`)
- `<target-distribution>`: the QIIME 2 distribution that your plugin should be installed under (e.g., `amplicon`, or `metagenome`)
- `<package-name>`: your plugin's package name (e.g., `q2-dwq2`)
- `<owner>`: the github organization your plugin is hosted under, or your personal github account name
- `<repository-name>`: the name of your plugin repository on GitHub (this often will be the same as your plugin's package name, e.g., `q2-dwq2`)
- `<target-branch>` (optional): the relevant branch that users should be utilizing to install your plugin - if not specified, this will default to your repository's *Default branch* (e.g., `main`). If you don't include this, you should leave off the `@` symbol following `.git`

Note

In the examples provided in this guide, we utilize the `released` channel for our QIIME 2 packages (i.e. `packages.qiime2.org/qiime2/<target-epoch>/<target-distribution>/released`). If you are interested in creating and/or sharing instructions for installing a version of your plugin that relies on one of our development environments (meaning the current development cycle environment prior to an official release version), you'll want to target the `passed` channel of that particular distribution. Here's an example of what this channel might look like (as of Sept. 2024, prior to the 2024.10 release):

```
https://packages.qiime2.org/qiime2/2024.10/amplicon/passed
```

Using the above guidelines, you can provide the following install instructions for your users:

```
conda env create \
 -n <target-epoch>-<package-name> \
 -f https://raw.githubusercontent.com/<owner>/<repository-name>/<target-branch>/
 ↪environment-files/<package-name>-qiime2-<target-distribution>-<target-epoch>.yml
```

Again, you'll fill in the values enclosed in the < and > brackets. As above, `<target-branch>` is the branch that your users should install, but in this case it is required. (Often this will be `main`.)

This method also provides a familiar way for users to install new versions of your plugin. By maintaining release branches on your repository, you can create a new environment file for each release that targets the corresponding release branch.

As an example, your branch structure could look like the following:

```
release-2024.5 # the 2024.5 release of your plugin
release-2024.10 # the 2024.10 release of your plugin
main # your main branch, usually what would be installed for a "development"␣
 ↪installation
```

You could then have environment files and install instructions for these different branches that look like the following (in this example, `amplicon` is the target distribution):

release-2024.5

Environment file: `q2-dwq2-qiime2-amplicon-2024.5.yml`

```
channels:
- https://packages.qiime2.org/qiime2/2024.5/amplicon/released
- conda-forge
- bioconda
dependencies:
  - qiime2-amplicon
  - pip
  - pip:
    - q2-dwq2@git+https://github.com/caporaso-lab/q2-dwq2.git@release-2024.5
```

Install instructions:

```
conda env create \
 -n q2-dwq2 \
 -f https://raw.githubusercontent.com/caporaso-lab/q2-dwq2.git/main/environment-files/
 ↪q2-dwq2-qiime2-amplicon-2024.5.yml
```

release-2024.10

Environment file: `q2-dwq2-qiime2-amplicon-2024.10.yml`

```
channels:
- https://packages.qiime2.org/qiime2/2024.10/amplicon/released
- conda-forge
- bioconda
dependencies:
  - qiime2-amplicon
  - pip
  - pip:
    - q2-dwq2@git+https://github.com/caporaso-lab/q2-dwq2.git@release-2024.10
```

Install instructions:

```
conda env create \
 -n q2-dwq2 \
 -f https://raw.githubusercontent.com/caporaso-lab/q2-dwq2.git/main/environment-files/
 ↪q2-dwq2-qiime2-amplicon-2024.10.yml
```

main (development)

Environment file: `q2-dwq2-qiime2-amplicon-development.yml`

```
channels:
- https://packages.qiime2.org/qiime2/2024.10/amplicon/released
- conda-forge
- bioconda
dependencies:
  - qiime2-amplicon
  - pip
  - pip:
    - q2-dwq2@git+https://github.com/caporaso-lab/q2-dwq2.git
```

Install instructions:

```
conda env create \
 -n q2-dwq2 \
 -f https://raw.githubusercontent.com/caporaso-lab/q2-dwq2.git/main/environment-files/
 ↪q2-dwq2-qiime2-amplicon-development.yml
```

In the above examples, the `main` branch location houses all of the environment files, regardless of which release they're associated with. This is reflected by each `conda env create` command referring to a URL like `https://raw.githubusercontent.com/.../main/environment-files/...`. We recommend having all of your environment files available on a single branch, which makes finding and referencing them easier.

Installing your plugin using the Tiny Distribution and any custom required plugins

If you are working on a plugin that is not compatible with one of our existing distributions but depends on some plugins in those distributions, you can utilize a similar approach to that outlined *above* but with a more customized environment file. As a reminder, while this approach is fairly straightforward to implement, **we don't recommend this if the option presented above is possible for your plugin** as this will be more difficult for us to assist you with and for you to help your users troubleshoot. As long as you are aware of these limitations and wish to proceed in this way, you can follow the steps below.

Start by following the same suggestions presented above for creating an `environment-files/` directory and naming your environment file. We'll put some different content in the environment file(s) this time.

As an example, the contents of an environment file for a plugin that depends on the `q2-feature-table` and `q2-composition` plugins would look something like this:

```
channels:
- https://packages.qiime2.org/qiime2/2024.5/tiny/released
- https://packages.qiime2.org/qiime2/2024.5/amplicon/released
- conda-forge
- bioconda
dependencies:
  - qiime2-tiny
  - q2-feature-table
  - q2-composition
  - pip
  - pip:
    - q2-dwq2@git+https://github.com/caporaso-lab/q2-dwq2.git@release-2024.5
```

In this example, the plugin being developed (`q2-dwq2`) requires `q2-feature-table` and `q2-composition`, but we're assuming that it's not compatible with the entire amplicon distribution. Because this plugin still requires a basic QIIME 2 environment, the `qiime2-tiny` distribution will be installed from the first channel listed. The `q2-feature-table` and `q2-composition` dependencies are not part of the `qiime2-tiny` distribution however, but are a part of the amplicon distribution. Therefore the second channel listed is the `amplicon` channel. We then list the dependencies as `qiime2-tiny` (the `tiny` distribution) and then the two additional plugins. Those are all followed by the installation of the `q2-dwq2` plugin, as in the previous example.

Generally, your customized environment files will be structured as follows:

```
channels:
- https://packages.qiime2.org/qiime2/<target-epoch>/tiny/released
- https://packages.qiime2.org/qiime2/<target-epoch>/<target-distribution>/released
- conda-forge
- bioconda
dependencies:
  - qiime2-tiny
  - <other-plugin-dependency-1>
  - <other-plugin-dependency-2>
  - pip
  - pip:
    - q2-dwq2@git+https://github.com/caporaso-lab/q2-dwq2.git@<target-branch>
```

In this case, `<other-plugin-dependency-1>` and `<other-plugin-dependency-2>` are plugins that are distributed through `<target-distribution>`. Note that if you have plugin dependencies that span multiple distributions, you'll need to include each distribution's channel in your environment file.

1.2.6 Automate testing of your plugin

Automating testing of your plugin is a good way to ensure that you are alerted to issues with your plugin before your users discover them. This How-to guide provides instructions on automating testing of your plugin using GitHub Actions, and assumes that have configured installation of your plugin as described in *Facilitating installation of your plugin for users*.

Important

You will need to adjust the environment file versions in the two Github Actions described below with each new QIIME 2 release. These are specified as `ci-<repository-name>` and `cron-<repository-name>` in the text below. We plan to develop functionality as part of the QIIME 2 Library that will help to automate this process for plugin developers, but as of now (16 August 2024) this process is manual.

Automated Testing using Continuous Integration (CI) and Github Actions (GHA)

The weekly development builds of the QIIME 2 distributions can help you make sure your code stays current with the distribution(s) you are targeting as you can automate your testing against them. *Set up your development environment* will help you install the most recent successful development metapackage build (again, usually weekly, but sometimes the builds fail and take time to debug).

There are a couple of things that we recommend implementing to help you ensure that your plugin remains compatible within the QIIME 2 ecosystem (discussed below).

Configure Continuous Integration (CI) testing

Continuous Integration testing is designed to regularly install and run your plugin's unit tests in the targeted QIIME 2 distributions or your custom distribution.

To implement this, you'll need to create a Github Action (GHA) that will be triggered each time you make a commit to your repository - either through a pull request (PR) or a direct commit to one of your remote branches. Github Actions can be a bit confusing to set up. We recommend the online course, *GitHub Automation for Scientists*, developed by the ITCR Training Network, before moving forward. Once you've read through this (and hopefully played around with a few of the toy examples provided therein), you can start to put together a CI workflow based the examples provided here.

Note

Before creating any GHAs for your plugin, you'll start by creating a top-level directory within your plugin's repository with the following name:

```
.github/workflows/
```

This naming structure is what allows Github to identify any relevant actions or workflows that should be run within your repository, and is where all of your GHA files should be created.

Here is what the basic structure of your GHA will look like:

```
name: ci-<repository-name>
on:
  pull_request:
    branches: ["<target-branch>"]
  push:
    branches: ["<target-branch>"]
```

(continues on next page)

(continued from previous page)

```
jobs:
  ci:
    uses: qiime2/distributions/.github/workflows/lib-community-ci.yaml@dev
    with:
      github-repo: <repository-name>
      env-file-name: <target-epoch>-<package-name>-environment.yml
```

With the bracketed terms defined as:

- `<target-branch>`: the branch that should be used when running the GHA This will typically be your `main` branch, but may differ if you've customized the branch structure of your repository.
- `<repository-name>`: the name of your repository on GitHub
- `<target-epoch>-<package-name>-environment.yml`: the name of your environment file. If you haven't created this yet, refer back to *Facilitating installation of your plugin for users* before continuing.

Your GHA file will be stored under the `.github/workflows/` directory in your repository, and you can use the same name as your Github Action for the filename (e.g., `ci-<repository-name>.yml`). Note that the extension will also be `.yml` (same as your environment file(s)).

After creating this file and pushing it to the main branch of your repository, this GHA should run anytime there is a commit to `<target-branch>` or a pull request against `<target-branch>`.

Configure weekly automated testing

Keeping your package up to date with all of the downstream dependencies can feel like a lot of work and hassle, and it can be. Unfortunately, software is never "done", and that's important to understand if you're distributing software for the community to use. It's going to require maintenance because software is always changing, and the more dependencies your plugin has, the more likely it is that updates to one of your dependencies will necessitate changes to your plugin (e.g., due to an API change in the dependency). Performing automated weekly test builds of your plugin will help you keep your package up to date and alert you as issues arise so you can discover them and address them on your own schedule, before it becomes a problem for your users.

In addition to running your unit tests for each commit and/or pull request against your plugin's `<target-branch>`, we recommend implementing regularly scheduled testing of your plugin against the development environments for the QIIME 2 distributions and/or plugins that it relies on.

The process for this will be very similar to the GHA discussed above. The main differences are that your plugin's environment will be configured with the latest development version of the relevant distribution(s), rather than a specified release version, and that the GHA will be triggered at specific times rather than based on specific events (commits or pull requests). We suggest having these tests run on a weekly basis to make sure you have ample time between QIIME 2 releases to fix any dependency conflicts or issues from code changes that may arise.

Here's the basic structure of the GHA you'll create to initiate these scheduled tests against your target distribution's development environment:

```
name: cron-<repository-name>
on:
  workflow_dispatch: {}
  schedule:
    - cron: 0 0 * * SUN
jobs:
  ci:
    uses: qiime2/distributions/.github/workflows/lib-community-ci.yaml@dev
    with:
```

(continues on next page)

(continued from previous page)

```
      github-repo: <repository-name>
      env-file-name: development-<repository-name>-environment.yml>
```

This GHA file will also be stored under the `.github/workflows/` directory in your repository, and you can use the same name as your Github Action for the filename (e.g., `cron-<repository-name>.yml`).

Relative to the GHA example above, the differences are:

- The trigger for this action (i.e., `on`) is either manual (`workflow_dispatch`) or a schedule (`cron`) (previously it was commits and pull requests). You can utilize the manual trigger under your repository's `actions` tab if you'd like to re-run these scheduled tests sooner than the next scheduled occurrence (if you're troubleshooting a test failure or upstream dependency issue). You can adjust the schedule to any frequency you'd prefer, but we recommend weekly testing to ensure you catch anything that may have fallen out of sync well in advance of the upcoming release. More information on the formatting for cron scheduling can be found here.
- The environment file that is targeted by this action is different than what's used in your CI testing (`development` vs `<target-epoch>`). The idea here is that your CI testing is targeting official release versions of your QIIME 2 environment, while these scheduled tests are targeting the current development environment. In order to support this, you'll need to create a new 'development' environment file with each QIIME 2 release that looks like the following:

```
channels:
- https://packages.qiime2.org/qiime2/<next-epoch>/<target-distribution>/passed
- conda-forge
- bioconda
dependencies:
  - qiime2-<target-distribution>
  - pip
  - pip:
    - <repository-name>@git+https://github.com/<owner>/<repository-name>.git@<target-
 ↪branch>
```

With the bracketed terms defined as:

- `<next-epoch>`: the next QIIME 2 release epoch. QIIME 2 releases are scheduled for the first Wednesday of April and October, so this value is always predictable. For example, if the most recent release was 2024.10, your `<next-epoch>` would be 2025.4.
- `<target-distribution>`: the QIIME 2 distribution that your plugin should be installed under (e.g., `amplicon`, or `metagenome`).
- `<target-branch>` (optional): the branch of your repository that testing will be performed against. If not specified, this will default to your repository's *Default branch* (e.g., `main`). If you don't include this, you should leave off the `@` symbol following `.git`.

You can set up any additional GHAs on your repository that you feel will be beneficial to your plugin and general development workflow. The actions outlined here are what the QIIME 2 developers recommend having to maintain an active and usable plugin.

1.2.7 Publicize your QIIME 2 plugins (or other QIIME 2-based tools)

If you want others to use your QIIME 2-based tools, such as plugins, you have to market them. Ultimately that's your responsibility - **you** built it, **you** know it, **you're** proud of it, so **you need to promote it**.

Here are some ideas on how to market your tools. If you have suggestions on additional ways, we're interested! Please feel free to reach out.

Help others understand how your tool will help them

Everyone is busy. While a lot of us would love to spend time finding and experimenting with cool things that other people are making, ultimately we don't have time and we tend to only invest time in discovering and truly learning the things that are *very obviously going to advance our priorities*. When you're marketing your tool to someone, start by making it clear how it can advance their priorities. This is about them and their goals - not you and your goals. You're selling, and you're hoping they're buying.

Ideally, show them - don't tell them - how your tool will advance their priorities. For example, reach out and ask people if you can run your software on their data to try to help them generate some new insights. If they're interested, get on the same page regarding publication expectations: if you discover something new, could you publish on it now, or could you co-author work that they're preparing? If you're on the same page, analyze their data, see what you see, and help them interpret the findings. If you find something cool, you're off to the races. If not, don't oversell your results - it can backfire. Regardless of how it works, learn from the process and use it to improve your tool and your marketing approach. As your tool helps others with their work, their word-of-mouth and their publications will market your work.

QIIME 2 Library

We are currently (23 April 2024) working on a simplified set of tools for sharing your plugins with the QIIME 2 community, and this will ultimately replace the QIIME 2 Library. Our end goals are to help you get your software discovered by the large QIIME 2 user community, and to help you make it clear how your tools can be installed, used, and cited. It will also enable users to make informed choices about which plugins they want to use by helping them understand if plugins are under active development, if and to what extent they are unit tested, and if they're well-supported by the developers. We expect incremental progress on this over the next year.

Community Contributions on the QIIME 2 Forum

When you create a new QIIME 2-based tool and are ready to share it with the QIIME 2 community, we recommend posting a new topic in the Community Contributions category on the QIIME 2 Forum. This is a way to share information on your tool, including what it is and how people can start using it. For example, you could include information on the project's website, a brief tutorial, and how users should request technical support.

Post a pre-print

Pre-prints are a citable way to get software announcement articles out while peer review is in process. As you probably know, peer review can take *ages* and software changes quickly, so we find that it's worth it to publish software announcement pre-prints that we can immediately begin citing.

That said, software announcement papers are unfortunately not accepted by many pre-print servers. Here are some exceptions to that, based on our experiences.

- arXiv is a pre-print server that does seem to accept software announcement pre-prints (e.g, see their Quantitative Biology category).

- F1000 is another option that we've had luck with. It's not technically a pre-print server, but rather a peer-review journal that publishes your article online once you submit it, and then publishes the peer-reviews, your responses, and the iterations of the paper.

From my (Greg's) perspective, where you publish a pre-print isn't all that important, though more popular pre-print servers can presumably help with visibility of your work. The most important thing is that you're getting a DOI that you can use to uniquely reference your article, and ideally that you can update the pre-print when you update the manuscript. Just confirm that your target journal for the peer-reviewed version of your publication doesn't restrict you from sharing the article on pre-print servers. Most don't, but if it doubt it doesn't hurt to check.

1.2.8 Register a QIIME 2 plugin

This document will describe how to `register` a plugin, allowing this plugin to interact with the QIIME 2 framework.

Overview

There are several high-level steps to registering a QIIME 2 plugin:

1. A QIIME 2 plugin must define one or more Python 3 functions that will be accessible through QIIME.
2. The plugin must be a Python 3 package that can be installed with `setuptools`.
3. The plugin must then instantiate a `qiime2.plugin.Plugin` object and define some information including the name of the plugin and its URL. In the plugin package's `setup.py` file, this instance will be defined as an `entry point`.
4. The plugin must then register its functions as QIIME 2 Actions, which will be accessible to users through any of the QIIME 2 interfaces.
5. Optionally, the plugin could be distributed through Anaconda or pypi as that will simplify installation for QIIME 2 users.

These steps are covered in detail below.

Writing a simple QIIME 2 plugin should be a straightforward process. For example, the `q2-emperor` plugin, which connects Emperor to QIIME 2, is written in a little over 100 lines of code (excluding unit tests and assets). It is a standalone plugin that defines how and which functionality in Emperor should be accessible through QIIME 2. Plugins will vary in their complexity. For example, a plugin that defines a lot of new functionality would likely be quite a bit bigger. q2-diversity is a good example of this. Unlike `q2-emperor`, there is some specific functionality (and associated unit tests) defined in this project, and it depends on several other Python 3 compatible libraries.

Before starting to write a plugin, you should install QIIME 2 and some plugins to familiarize yourself with the system and to provide a means for testing your plugin.

Instantiating a plugin

The next step is to instantiate a QIIME 2 `Plugin` object. This might look like the following:

```
from qiime2.plugin import Plugin
import q2_diversity

plugin = Plugin(
    name='diversity',
    version=q2_diversity.__version__,
    website='https://github.com/qiime2/q2-diversity',
    package='q2_diversity',
```

(continues on next page)

(continued from previous page)

```
    description=('This QIIME 2 plugin supports metrics for calculating '
                 'and exploring community alpha and beta diversity through '
                 'statistics and visualizations in the context of sample '
                 'metadata.'),
    short_description='Plugin for exploring community diversity.',
)
```

This will provide QIIME with essential information about your `Plugin`.

The `name` parameter is the name that users will use to access your plugin from within different QIIME 2 interfaces. It should be a "command-line-friendly" name, so should not contain spaces or punctuation. (Avoiding uppercase characters and using dashes (`-`) instead of underscores (`_`) are preferable in the plugin `name`, but not required).

`version` should be the version number of your package (the same that is used in its `setup.py`).

`website` should be the page where you'd like end users to refer for more information about your package.

`package` should be the Python package name for your plugin.

`description` should give a brief description of this plugin's functionality. This will be displayed when that plugin's help documentation is accessed via the QIIME 2 framework.

`short_description` should give a very brief description of this plugin's functionality. This will be displayed when the QIIME 2 help documentation is accessed.

While not shown in the previous example, plugin developers can optionally provide the following parameters to `qiime2.plugin.Plugin`:

- `citations`: A list of bibtex-formatted citations. These are provided in a separate `citations.bib` file, loaded via the `Citations` API, and accessed by using their bibtex indices as keys. Citations can be listed during plugin or action registration, or both, but will usually only be listed for individual actions unless if a single reference is appropriate for all actions in that plugin. `q2-diversity` has no such plugin-wide citation listed here.
- `user_support_text`: free text describing how users should get help with the plugin (e.g. issue tracker, StackOverflow tag, mailing list, etc.). If not provided, users are referred to the `website` for support. Plugin developers are free to support their plugins on the QIIME 2 Forum, so you can include that URL as the `user_support_text` for your plugin. If you do that, you should get in the habit of monitoring the QIIME 2 Forum for technical support questions.

The `Plugin` object can live anywhere in your project, but by convention it will be in a file called `plugin_setup.py`. You can see a complete working example in q2-dwq2 here.

Defining your plugin object as an entry point

Finally, you need to tell QIIME where to find your instantiated `Plugin` object. This is done by defining it as an `entry_point` in your project's `setup.py` file. In `q2-diversity`, this is done as follows:

```
setup(
    ...
    entry_points={
        'qiime2.plugins': ['q2-diversity=q2_diversity.plugin_setup:plugin']
    }
)
```

The relevant key in the `entry_points` dictionary will be `'qiime2.plugins'`, and the value will be a single element list containing a string formatted as `<distribution-name>=<import-path>:<instance-name>`. `<distribution-name>` is the name of the Python package distribution (matching the value passed for `name`

in this call to `setup`); `<import-path>` is the import path for the `Plugin` instance you created above; and `<instance-name>` is the name for the `Plugin` instance you created above.

1.2.9 Create and register a Method

A `method` accepts some combination of QIIME 2 `artifacts` and parameters as input, and produces one or more QIIME 2 artifacts as output. These output artifacts could subsequently be used as input to other QIIME 2 `methods` or `visualizers`.

Create a function to register as a Method

A function that can be registered as a `Method` will have a Python 3 API, and the inputs and outputs for that function will be annotated with their data types using mypy syntax. mypy annotation does not impact functionality (though the syntax is new to Python 3), so these can be added to existing functions in your Python 3 software project. An example is `q2_diversity.pcoa`, which takes an `skbio.DistanceMatrix` and an `int` as input, and produces an `skbio.OrdinationResults` as output. The signature for this function is:

```
def pcoa(distance_matrix: skbio.DistanceMatrix,
         number_of_dimensions: int = None) -> skbio.OrdinationResults:
```

As far as QIIME is concerned, it doesn't matter what happens inside this function (as long as it adheres to the contract defined by the signature regarding the input and output types). For example, `q2_diversity.pcoa` is making some calls to the `skbio` API, but it could be doing anything, including making system calls (if your plugin is wrapping a command line application), executing an R library, etc.

Register the Method

Once you have a function that you'd like to register as a `Method`, and you've instantiated your `Plugin` object, you are ready to register that function as a `Method`. This will likely be done in the file where the `Plugin` object was instantiated, as it will use that instance (which will be referred to as `plugin` in the following examples).

We register a `Method` by calling `plugin.methods.register_function` as follows (see the original source here).

```
from q2_types import DistanceMatrix, PCoAResults
from qiime2.plugin import Int, Citations

import q2_diversity


citations = Citations.load('citations.bib', package='q2_diversity')


plugin.methods.register_function(
    function=q2_diversity.pcoa,
    inputs={'distance_matrix': DistanceMatrix},
    parameters={
        'number_of_dimensions': Int % Range(1, None)
    },
    outputs=[('pcoa', PCoAResults)],
    input_descriptions={
        'distance_matrix': ('The distance matrix on which PCoA should be '
                            'computed.')
```

(continues on next page)

(continued from previous page)

```
    },
    parameter_descriptions={
        'number_of_dimensions': "Dimensions to reduce the distance matrix to. "
                                "This number determines how many "
                                "eigenvectors and eigenvalues are returned,"
                                "and influences the choice of algorithm used "
                                "to compute them. "
                                "By default, uses the default "
                                "eigendecomposition method, SciPy's eigh, "
                                "which computes all eigenvectors "
                                "and eigenvalues in an exact manner. For very "
                                "large matrices, this is expected to be slow. "
                                "If a value is specified for this parameter, "
                                "then the fast, heuristic "
                                "eigendecomposition algorithm fsvd "
                                "is used, which only computes and returns the "
                                "number of dimensions specified, but suffers "
                                "some degree of accuracy loss, the magnitude "
                                "of which varies across different datasets."
    },
    output_descriptions={'pcoa': 'The resulting PCoA matrix.'},
    name='Principal Coordinate Analysis',
    description=("Apply principal coordinate analysis."),
    citations=[citations['legendrelegendre'],
               citations['halko2011']]
)
```

The values being provided are:

- `function`: The function to be registered as a method.
- `inputs`: A dictionary indicating the parameter name and its `artifact class`, for each input `Artifact`. These artifact classes differ from the data types that you provided in your `mypy_` annotation of the input, as artifact classes describe the data, where the data types indicate the structure of the data. (See *Semantic types, data types, file formats, and artifact classes* for more detail on the difference between data types, semantic types, and artifact classes.) In the example above we're indicating that the table parameter must be a `FeatureTable` of `Frequency` (i.e. counts), and that the `phylogeny` parameter must be a `Phylogeny` that is `Rooted`. Notice that the keys in inputs map directly to the parameter names in `q2_diversity.beta_phylogenetic`.
- `parameters`: A dictionary indicating the parameter name and its type, for each input Parameter. These parameters are primitive values (i.e., non-`Artifacts`). In the example above, we're indicating that the metric should be a string from a specific set (in this case, the set of known phylogenetic beta diversity metrics).
- `outputs`: A list of tuples indicating each output name and its artifact class.
- `input_descriptions`: A dictionary containing input artifact names and their corresponding descriptions. This information is used by interfaces to instruct users how to use each specific input artifact.
- `parameter_descriptions`: A dictionary containing parameter names and their corresponding descriptions. This information is used by interfaces to instruct users how to use each specific input parameter. You should not include any default parameter values in these descriptions, as these will generally be added automatically by an interface.
- `output_descriptions`: A dictionary containing output artifact names and their corresponding descriptions. This information is used by interfaces to inform users what each specific output artifact will be.
- `name`: A human-readable name for the Method. This may be presented to users in interfaces.
- `description`: A human-readable description of the Method. This may be presented to users in interfaces.

- `citations`: A list of bibtex-formatted citations. These are provided in a separate `citations.bib` file, loaded via the `Citations` API, and accessed here by using their bibtex indices as keys.

1.2.10 Create and register a visualizer

A `Visualizer` accepts some combination of QIIME 2 `Artifacts` and parameters as input, and produces exactly one `Visualization` as output. Visualizations are visual representations of analytical results (e.g., plots, statistical results, summary tables) and by definition cannot be used as input to other QIIME 2 methods or visualizers. Thus, visualizers can only produce terminal output in a QIIME 2 analysis.

Create a function to register as a Visualizer

Defining a function that can be registered as a `Visualizer` is very similar to defining one that can be registered as a `Method` with a few additional requirements.

First, the first parameter to this function must be `output_dir`. This parameter should be annotated with type `str`.

Next, at least one `index.*` file must be written to `output_dir` by the function. This index file will provide the starting point for your users to explore the `Visualization` object that is generated by the `Visualizer`. Index files with different extensions can be created by the function (e.g., `index.html`, `index.tsv`, `index.png`), but at least one must be created. You can write whatever files you want to `output_dir`, including tables, graphics, and textual descriptions of the results, but you should expect that your users will want to find those files through your index file(s). If your function does create many different files, an `index.html` containing links to those files is likely to be helpful.

Finally, the function cannot return anything, and its return type should be annotated as `None`.

`q2_diversity.alpha_group_significance` is an example of a function that can be registered as a `Visualizer`. In addition to its `output_dir`, it takes alpha diversity results in a `pandas.Series` and sample metadata in a `qiime2.Metadata` object and creates several different files (figures and tables) that are linked and/or presented in an `index.html` file. The signature of this function is:

```
def alpha_group_significance(output_dir: str,
                             alpha_diversity: pd.Series,
                             metadata: qiime2.Metadata) -> None:
```

Register the Visualizer

Registering `Visualizers` is the same as registering `Methods`, with two exceptions.

First, you call `plugin.visualizers.register_function` to register a `Visualizer`.

Next, you do not provide `outputs` or `output_descriptions` when making this call, as `Visualizers`, by definition, only return a single visualization. Since the visualization output path is a required parameter, you do not include this in an `outputs` list (it would be the same for every `Visualizer` that was ever registered, so it is added automatically).

Registering `q2_diversity.alpha_group_significance` as a `Visualizer` looks like the following (find the original source here):

```
plugin.visualizers.register_function(
    function=q2_diversity.alpha_group_significance,
    inputs={'alpha_diversity': SampleData[AlphaDiversity]},
    parameters={'metadata': Metadata},
    input_descriptions={
        'alpha_diversity': 'Vector of alpha diversity values by sample.'
```

(continues on next page)

(continued from previous page)

```
    },
    parameter_descriptions={
        'metadata': 'The sample metadata.'
    },
    name='Alpha diversity comparisons',
    description=("Visually and statistically compare groups of alpha diversity"
                 " values."),
    citations=[citations['kruskal1952use']]
)
```

See the text describing *registering methods* for a description of these values.

1.2.11 Create and register a pipeline

A `Pipeline` accepts some combination of QIIME 2 `Artifacts` and parameters as input, and produces one or more QIIME 2 artifacts and/or `Visualizations` as output. This is accomplished by stitching together one or more `Methods` and/or `Visualizers` into a single `Pipeline`.

Create a function to register as a Pipeline

Defining a function that can be registered as a `Pipeline` is very similar to defining one that can be registered as a `Method` with a few distinctions.

First, `Pipelines` do not use function annotations and instead receive `Artifact` objects as input and return `Artifact` and/or `Visualization` objects as output.

Second, `Pipelines` must have `ctx` as their first parameter, which provides the following API:

- `ctx.get_action(plugin: str, action: str)`: returns a *sub-action* that can be called like a normal Artifact API call.
- `ctx.make_artifact(type, view, view_type=None)`: this has the same behavior as `Artifact.import_data`. It is wrapped by `ctx` for pipeline book-keeping.

Let's take a look at `q2_diversity.core_metrics` for an example of a function that we can register as a `Pipeline`:

```
def core_metrics(ctx, table, sampling_depth, metadata, n_jobs=1):
    rarefy = ctx.get_action('feature_table', 'rarefy')
    alpha = ctx.get_action('diversity', 'alpha')
    beta = ctx.get_action('diversity', 'beta')
    pcoa = ctx.get_action('diversity', 'pcoa')
    emperor_plot = ctx.get_action('emperor', 'plot')

    results = []
    rarefied_table, = rarefy(table=table, sampling_depth=sampling_depth)
    results.append(rarefied_table)

    for metric in 'observed_otus', 'shannon', 'pielou_e':
        results += alpha(table=rarefied_table, metric=metric)

    dms = []
    for metric in 'jaccard', 'braycurtis':
        beta_results = beta(table=rarefied_table, metric=metric, n_jobs=n_jobs)
        results += beta_results
```

(continues on next page)

(continued from previous page)

```
        dms += beta_results

    pcoas = []
    for dm in dms:
        pcoa_results = pcoa(distance_matrix=dm)
        results += pcoa_results
        pcoas += pcoa_results

    for pcoa in pcoas:
        results += emperor_plot(pcoa=pcoa, metadata=metadata)

    return tuple(results)
```

Registering the Pipeline

Registering `Pipelines` is the same as registering `Methods`, with a few exceptions.

First, we register a `Pipeline` by calling `plugin.pipelines.register_function`.

Second, `visualizations` produced as an output are listed in `outputs` as a `tuple` with `Visualization` as the second value. E.g., `('jaccard_emperor', Visualization)`. A description of this output should be included in `output_descriptions`

Citations do not need to be added for the pipeline unless unique citations are required for the pipeline that are not appropriate for the underlying `Methods` and `Visualizers` that it calls. Citations for these underlying actions are automatically logged in citation provenance for this pipeline.

As an example for registering a `Pipeline`, we can look at `q2_diversity.core_metrics` (find the original source here):

```
plugin.pipelines.register_function(
    function=q2_diversity.core_metrics,
    inputs={
        'table': FeatureTable[Frequency],
    },
    parameters={
        'sampling_depth': Int % Range(1, None),
        'metadata': Metadata,
        'n_jobs': Int % Range(0, None),
    },
    outputs=[
        ('rarefied_table', FeatureTable[Frequency]),
        ('observed_otus_vector', SampleData[AlphaDiversity]),
        ('shannon_vector', SampleData[AlphaDiversity]),
        ('evenness_vector', SampleData[AlphaDiversity]),
        ('jaccard_distance_matrix', DistanceMatrix),
        ('bray_curtis_distance_matrix', DistanceMatrix),
        ('jaccard_pcoa_results', PCoAResults),
        ('bray_curtis_pcoa_results', PCoAResults),
        ('jaccard_emperor', Visualization),
        ('bray_curtis_emperor', Visualization),
    ],
    input_descriptions={
        'table': 'The feature table containing the samples over which '
                 'diversity metrics should be computed.',
    },
```

(continues on next page)

(continued from previous page)

```
    parameter_descriptions={
        'sampling_depth': 'The total frequency that each sample should be '
                          'rarefied to prior to computing diversity metrics.',
        'metadata': 'The sample metadata to use in the emperor plots.',
        'n_jobs': '[beta methods only] - %s' % sklearn_n_jobs_description
    },
    output_descriptions={
        'rarefied_table': 'The resulting rarefied feature table.',
        'observed_otus_vector': 'Vector of Observed OTUs values by sample.',
        'shannon_vector': 'Vector of Shannon diversity values by sample.',
        'evenness_vector': 'Vector of Pielou\'s evenness values by sample.',
        'jaccard_distance_matrix':
            'Matrix of Jaccard distances between pairs of samples.',
        'bray_curtis_distance_matrix':
            'Matrix of Bray-Curtis distances between pairs of samples.',
        'jaccard_pcoa_results':
            'PCoA matrix computed from Jaccard distances between samples.',
        'bray_curtis_pcoa_results':
            'PCoA matrix computed from Bray-Curtis distances between samples.',
        'jaccard_emperor':
            'Emperor plot of the PCoA matrix computed from Jaccard.',
        'bray_curtis_emperor':
            'Emperor plot of the PCoA matrix computed from Bray-Curtis.',
    },
    name='Core diversity metrics (non-phylogenetic)',
    description=("Applies a collection of diversity metrics "
                 "(non-phylogenetic) to a feature table.")
)
```

See the text describing *registering methods* for a description of these values.

1.2.12 Creating and registering a Transformer

Transformers are often short Python functions that convert one file format or data type to another file format or data type. These functions are never directly called by users or developers, so by convention they don't get informative function names (as the annotations of the input and output provide complete detail on what they do).

Here's are two example `transformer` that are defined and registered in `q2-types`:

```
import skbio

from ..plugin_setup import plugin
from .
import LSMatFormat


@plugin.register_transformer
def _1(data: skbio.DistanceMatrix) -> LSMatFormat:
    ff = LSMatFormat()
    with ff.open() as fh:
        data.write(fh, format='lsmat')
    return ff


@plugin.register_transformer
```

(continues on next page)

(continued from previous page)

```
def _2(ff: LSMatFormat) -> skbio.DistanceMatrix:
    return skbio.DistanceMatrix.read(str(ff), format='lsmat', verify=False)
```

These transformers define how an `skbio.DistanceMatrix` object is transformed into an `LSMatFormat` object (the underlying format of the data in a `DistanceMatrix` artifact class, defined here in q2-types, and registered to the `DistanceMatrix` semantic type here). The transformers are registered using the `@plugin.register_transformer` decorator.

1.2.13 Use Artifact Collections as Action inputs or outputs

Commands in QIIME 2 can take collections of Artifacts as singular inputs or return collections of Artifacts as singular outputs. They may also take collections of primitives as single parameters.

Registering an Action that Takes an Input Collection

Input or parameter collections can be in the form of lists or dictionaries. For inputs the type annotation for function registration is the QIIME 2 artifact class(es) of the Artifacts expected. For parameters it is the type of the parameter.

An example of registering collection inputs and parameters is shown below. For a list input, the syntax is `"List[SemanticType]"` and for a dictionary it is `"Collection[SemanticType]"`.

```
dummy_plugin.methods.register_function(
    function=example_function,
    inputs={
        'int_list': List[SingleInt],
        'int_dict': Collection[SingleInt],
    },
    parameters={
        'bool_list': List[Bool],
        'bool_dict': Collection[Bool]
    },
    outputs=[
        ('return', Collection[SingleInt]),
    ],
    name='Example',
    description=('Example collection method')
)
```

In the actual function definition, the type annotation is the view type of the Artifacts and does NOT contain the collection type annotation. The fact that the annotations indicate the view type and not the artifact class is due to the fact that by the time we reach the actual method `int_list` will be a list of integers and not `SingleInt` Artifacts.

```
# Note type annotations are currently just int or bool,
# not dict[int] or list[bool] as you might expect.
def example_function(int_list: int,
                     int_dict: int,
                     bool_list: bool,
                     bool_dict: bool) -> int:
    return int_list.extend(list(int_dict.value()))
```

Warning

The fact that function definitions do not contain the collection type annotations is an implementation detail that may change in the future. We will provide advance warning of this backward incompatible change on the QIIME 2 Forum at least one release prior to the change, if this does end up changing.

Registering an Action that Returns an Output Collection

Returning an output collection works much the same as returning anything else in QIIME 2. Using the same example method as earlier, you register your return as a Collection of the type of Artifact you are returning.

```
dummy_plugin.methods.register_function(
    function=example_function,
    inputs={
        'int_list': List[SingleInt],
        'int_dict': Collection[SingleInt],
    },
    parameters={
        'bool_list': List[Bool],
        'bool_dict': Collection[Bool]
    },
    outputs=[
        ('return', Collection[SingleInt]),
    ],
    name='Example',
    description=('Example collection method')
)
```

The return type annotation on the action itself is still the view type of the Artifacts within the collection.

```
# Note type annotations are currently just int or bool,
# not dict[int] or list[bool] as you might expect.
def example_function(int_list: int,
                     int_dict: int,
                     bool_list: bool,
                     bool_dict: bool) -> int:
    return int_list.extend(list(int_dict.value()))
```

In this instance, the value `ints` that is returned is a list, but it could also have been a dict. The actual QIIME 2 Result you get is a `ResultCollection` object which is essentially a wrapper around a dictionary. If the original return is a list, the `ResultCollection` uses the list indices as keys.

Using Collections

Using Collections with the command line interface (CLI)

In the CLI, output collections require an output path to a directory that does not exist yet. The directory will be created, and the Artifacts in the collection will be written to the directory along with a .order file that lists the order of the Artifacts in the collection.

These collections can then be used as inputs to new actions by simply passing that directory as the input path. You can also create a new directory yourself and place artifacts in it manually to use as an input collection. This directory may or may not have a .order file. If it does not contain a .order file, the artifacts in the directory will be loaded in whatever order the file system presents them in.

De-facto collections of parameters and inputs may also be created on the CLI by simply passing the corresponding argument multiple times. For example, the following will create a collection of foo.qza and bar.qza for the ints input.

```
qiime plugin action --i-ints foo.qza --i-ints bar.qza
```

The collection will be loaded in the order the arguments are presented to the command line in so in this case `[foo, bar]` if ints wants a list or `{'0': foo, '1': bar}` if it wants a dict.

You may also explicitly key the values as follows:

```
qiime plugin action --i-ints foo:foo.qza --i-ints bar:bar.qza
```

As you might imagine, this would look like `{'foo': foo, 'bar': bar}` internally if ints wanted a dict. If ints wanted a list, it would just strip the keys and be `[foo, bar]` again.

Using Collections with the Python API

When working through QIIME 2's Python API, you can pass in a list or a dict and it follows the same rules as the CLI. Internally QIIME 2 will turn it into the collection type it needs. If it needs a dict but you gave it a list it will use list indices as keys. If it needs a list but you gave it a dict, it will strip the keys and make a list of the values.

The `ResultCollection` object

Note

The following content will be moved to the *Reference* chapter as API documentation.

QIIME 2 outputs collections in the form of `ResultCollection` objects. On the CLI, these objects are handled internally, but in the Python API they must be interacted with directly. Fortunately, these objects are very simple.

A `ResultCollection` is basically a simple wrapper around a dictionary that can be referenced through its `collection` attribute.

init

Instantiating a `ResultCollection` object without any arguments will create a `ResultCollection` with an empty dictionary as its collection. Instantiating a `ResultCollection` with a dictionary as its argument will create a `ResultCollection` with that dictionary as its collection. Instantiating a `ResultCollection` with any other iterable will enumerate the iterable and use the indices as keys to the dictionary that is used as the collection.

load

You can load a directory of Artifacts (an output collection from CLI for example) into a `ResultCollection` by calling `ResultCollection.load('path to directory')`. If this directory contains a `.order` file, the Artifacts will be loaded in the order specified in the `.order` file. Otherwise they will be loaded in the order the OS presents them in (not defined by us). The names of the files will be used as the keys to the Artifacts.

save

You can save your `ResultCollection` to disk by calling `ResultCollection.save('path to destination')` where the destination is a directory that does not exist yet. This will save all Artifacts in the collection to .qzas in the directory using their key as their name. It will also create a `.order` file in the directory that lists the keys in the collection in order.

Other than these methods, you may set and read values on a `ResultCollection` just the same as a dictionary, you may also call keys, values, and items on a `ResultCollection` in the same way as a dictionary. The validate method also exists on `ResultCollection` objects and will validate all Artifacts that are part of the collection.

1.2.14 How to play nicely with other plugins

There are several namespace considerations to keep in mind when developing QIIME 2 *plugins* or *actions*.

- *Plugin* names are unique, and collisions are not tolerated by the framework. This means if you want to name your plugin `dada2`, users will not be able to have your plugin **and** the DADA2 plugin installed in the same *deployment* at the same time.
- *Action* names are **not** globally unique, but rather are namespaced within a plugin's domain.
- *Semantic types*, *artifact classes*, *formats*, and *transformers* are registered globally, like plugins, again, if you choose to use an existing name, this will prevent users from deploying both offending plugins concurrently.

There are a few strategies the QIIME 2 developers have employed to work within these limits:

- Most *semantic types*, *artifact classes*, *formats*, and *transformers* are registered in a single *plugin*: q2-types. This prevents circular imports. For many 3rd-party plugin developers, this means most typical types and formats are available within a single import. If you're creating new types in your plugins, avoid re-using type names. Generally types are well established before being added to q2-types, since we commit to backward compability of q2-types (so it's hard to make changes to type names, for example).
- Plugin naming takes one of two approaches, typically:
  - The name matches the source tool being wrapped (e.g. `cutadapt`).
  - The name is some generic descriptor, like `diversity` or `taxa`.
- Action naming typically takes one of two approaches:
  - The name matches a subcommand or method on the source tool being wrapped.
  - The name is a generic descriptor, and typically is written as a verb (but not exclusively).

1.2.15 How to use Metadata

Metadata (the `qiime2.metadata.Metadata` class, internally) allows users to annotate a QIIME 2 *Result* with study-specific values: age, elevation, body site, pH, etc. QIIME 2 offers a consistent API for developers to expose their *Methods* and *Visualizers* to user-defined metadata. For more details about how users might create and utilize metadata in their studies, check out the Metadata In QIIME 2 tutorial.

> **Note**
>
> Refer to the *User Metadata API* for additional information on using metadata in your plugins.

Metadata

Actions may request an entire `Metadata` object to work on. At its core, `Metadata` is just a pandas pd.Dataframe, but the `Metadata` object provides many convenience methods and properties, and unifies the code necessary for handling these data (or metadata). Examples of *Actions* that consume and operate on `Metadata` include:

- q2-longitudinal's `volatility`
- q2-metadata's `tabulate`
- q2-feature-table's `filter-features`
- And many more

Plugins may work with metadata directly, or they may choose to filter, regroup, partition, pivot, etc. - it all depends on the intended outcome relevant to the *method* or *visualizer* in question.

`Metadata` is subject to framework-level validations, normalization, and verification. We recommend familiarizing yourself with this behavior before utilizing `Metadata` in your *Action*. We think having this kind of behavior available via a centralized API helps ensure consistency for all users of `Metadata`.

```
def my_viz(output_dir: str, md: qiime2.Metadata) -> None:
    df = md.to_dataframe()
    ...
```

Metadata Columns

Plugin *Actions* may also request one or more `MetadataColumns` (the `qiime2.metadata.MetadataColumn`, internally) to operate on, a good example of this is identifying which column of metadata contains barcodes, when using q2-demux's `emp-single` or q2-cutadapt's `demux-paired`, for example.

Instances of `MetadataColumn` exist as one of two concrete classes: `NumericMetadataColumn` (`qiime2.metadata.NumericMetadataColumn`) and `CategoricalMetadataColumn` (`qiime2.metadata.CategoricalMetadataColumn`).

By default, QIIME 2 will attempt to infer the type of each metadata column: if the column consists only of numbers or missing data, the column is inferred to be numeric. Otherwise, if the column contains any non-numeric values, the column is inferred to be categorical. Missing data (i.e. empty cells) are supported in categorical columns as well as numeric columns.

```
...
numeric_md_cols = metadata.filter(column_type='numeric')
categorical_md_cols = metadata.filter(column_type='categorical')
...
```

If your *Action* always needs one type of column or another, you can simply register that type in your plugin registration:

```
plugin.methods.register_function(
    ...
    parameters={'metadata': MetadataColumn[Numeric]},
    parameter_descriptions={'metadata': 'Numeric metadata column to '
                            'compute pairwise Euclidean distances from'},
    ...
```

This will ensure that all the necessary type-checking is performed by the framework before these data are passed into the *Action* utilizing it.

Numeric Metadata Columns

Columns that consist only of numeric (or missing) values are eligible for being instantiated as `NumericMetadataColumn` (although these values can be loaded as `CategoricalMetadataColumn`, too).

Categorical Metadata Columns

All types of data columns can be instantiated as `CategoricalMetadataColumn` - values will be cast to strings.

How can the Metadata API Help Me?

The `qiime2.metadata.Metadata` API has many interesting features - here are some of the more commonly utlitized elements amongst the plugins within the Amplicon *Distribution*.

Merging Metadata

Interfaces can allow users to specify more than one metadata file at a time, the framework will handle merging the files or objects `qiime2.metadata.Metadata.merge` prior to handing the final merged set to your *Action*.

Dropping Empty Columns

When working with a single metadata metadata column, plugin code can determine if there are missing values (`qiime2.metadata.MetadataColumn.has_missing_values`), and then subsequently drop those IDs (`qiime2.metadata.MetadataColumn.drop_missing_values`) from the column.

Normalizing TSV Files

By saving (`qiime2.metadata.Metadata.save`) a materialized `Metadata` instance, visualizations that want to provide data exports can do so in a consistent manner (e.g. q2-longitudinal's `volatility`, and the relevant code.

Advanced Filtering

The `filter` (`qiime2.metadata.Metadata.filter_columns`) method can be used to restrict column types, drop empty columns, or remove columns made entirely of unique values.

SQL Filtering

Advanced metadata querying is enabled by SQL-based filtering (`qiime2.metadata.Metadata.get_ids`).

Making Artifacts Viewable as Metadata

By *registering a transformer* from a particular *format* to `qiime2.Metadata`, the framework will allow the *type* represented by that format to be *viewed* as `Metadata` — this can open up all kinds of exciting opportunities for plugins!

```
@plugin.register_transformer
def _1(data: cool_project.InterestingDataFormat) -> qiime2.Metadata:
    df = pd.Dataframe(data)
    return qiime2.Metadata(df)
```

A visualizer for free!

If your *type* is viewable as `Metadata` (as in, the necessary transformers are registered), there is a general-purpose metadata visualization in the q2-metadata plugin called `tabulate`, which renders an interactive (searchable, sortable) table of the metadata in question. Cool!

Generating metadata as output from visualizations

In most cases, if you want to output something that looks like metadata from a QIIME 2 action, you should *assign it an artifact class that has a transformer to `Metadata`*. However in some cases you may want to output actual metadata. In this case, you can create an output for your action of artifact class `ImmutableMetadata`. This will generate an artifact containing the metadata that your function provides as output.

`ImmutableMetadata` artifacts can be *viewed as `Metadata`*, so they can be used anywhere that a typical metadata `.tsv` file can be provided as input in QIIME 2. This includes q2-metadata's `tabulate` visualizer. Additionally, if you want to obtain a `.tsv` file representation of an `ImmutableMetadata` artifact, you can export it.

1.2.16 How to test QIIME 2 plugins

This document is a placeholder at the moment.

Briefly, QIIME 2 provides a test harness to simplify a few of the more repetitive parts of testing. This is the `TestPluginBase` class (`qiime2.plugin.testing.TestPluginBase`).

Examples

Pending full documentation, here are some references where you can see how aspects of QIIME 2 plugins can be tested:

- Formats
- Semantic Types
- Transformers
- Plugin registration - note that testing of plugin registration doesn't require the use of the `TestPluginBase` class.

You can see an example of all of these in one place in the q2-sapienns test suite. You can learn about the q2-sapienns plugin here.

1.2.17 Writing Usage Examples

> **Warning**
>
> This document is not yet formatted for *Developing with QIIME 2*. Some links or references may not work, and updaes are planned to the content. If you run into issues, please let us know here.

QIIME 2 enables you to write *usage examples* that give examples of how plugin actions can be run by your users. It uses dependency injection to generate appropriate usage examples for any interface with a "usage driver". You define and register each usage example once, giving it a parameter that accepts some usage driver. The interface that is running your usage examples will inject one or more of its drivers into your examples, rendering interface-appropriate results.

The API is defined by the Usage class. Individual usage drivers implement the underlying behavior of API functions according to their own needs. As a result, the ExecutionUsage driver will attempt to execute your usage examples, but will disregard *comment*s because it is not a user-facing driver. The ArtifactAPIUsage driver will include your comments as python comments in the rendered usage example, but will not execute your example.

The API is split into two sides - one which allows plugin developers to define usage examples, and one which allows interface developers to write the usage drivers that make those examples go. *In this how-to guide, we will focus exclusively on the plugin-developer facing usage example side of the API.*

In this how-to guide, we will cover:

- *Data factories for usage examples*
- *Defining usage examples*
- *Registering usage examples*
- *Testing usage examples*
- *Trying it out*
- *Comments can provide context*

Data factories for usage examples

Because some drivers actually execute these usage examples, there is an expectation that we provide real data for them. Simple assignment is not possible. Inputs and Metadata must be created by a factory function. This allows many drivers to avoid loading data unnecessarily. Parameter literals may be passed directly, and do not require factories.

This example shows a factory function that returns a `FeatureTable[Frequency]`. We use the Python 3 API to import a `biom.Table` of the appropriate artifact class.

```
# from q2-feature-table/examples.py
import numpy as np
from biom import Table

from qiime2 import Artifact


def ft1_factory():
    return Artifact.import_data(
        'FeatureTable[Frequency]',
        Table(np.array([[0, 1, 3], [1, 1, 2]]),
              ['O1', 'O2'],
              ['S1', 'S2', 'S3']))
```

Defining usage examples

We've created some data, now we'll define a usage example. This is a simple python function with a single parameter (`use` by convention). Interfaces pass their drivers to the example through `use` as described in the introduction. The methods called inside of the function are "public" (non-underscore-prefixed) methods defined in `qiime2.sdk.usage.Usage`. This "Usage API" is common to all Usage drivers, which reimplement the methods to meet their own needs.

```
# also from q2-feature-table/examples.py
def feature_table_merge_example(use):
    feature_table1 = use.init_artifact('feature_table1', ft1_factory)
    feature_table2 = use.init_artifact('feature_table2', ft2_factory)

    merged_table, = use.action(
        use.UsageAction(plugin_id='feature_table',
                        action_id='merge'),
        use.UsageInputs(tables=[feature_table1, feature_table2]),
        use.UsageOutputNames(merged_table='merged_table'),
    )
```

First, we initialize two feature tables. (`ft2_factory` looks a lot like the `ft1_factory` defined above. You'll have to use your imagination on the details.)

We then use a proxy method for invoking an action. The action may or may not *actually* be invoked, depending on implementation details in the usage driver. Beyond ensuring that your example is correct and meaningful, you don't have to worry about this.

Note that `UsageInputs` include both QIIME 2 *Inputs and parameters*. Metadata must be initialized, but primitive parameters (and collections of parameters) may be passed directly. There are examples of this in the identity_with_metadata_column_get_mdc and variadic_input_simple examples in the framework.

Registering usage examples

Like QIIME 2 *Actions*, the usage examples we have defined must be registered in order to be used.

This registration occurs in `plugin_setup.py`, in the `register_function` block for the Action whose usage we are showing.

```
# from q2-feature-table/plugin_setup.py

# we need to import the examples to use them
from .examples import (feature_table_merge_example,
                       feature_table_merge_three_tables_example)

plugin.methods.register_function(
    function=q2_feature_table.merge,
    inputs={'tables': List[i_table]},

    # Skipping ahead to the 'examples' keyword argument
    # Everything else here should look familiar
    ...

    examples={'basic': feature_table_merge_example,
              'three_tables': feature_table_merge_three_tables_example},
)
```

The keys in the `examples` dictionary serve as unique identifiers for the examples themselves. Some drivers (e.g. q2cli) use them to label rendered examples.

Testing usage examples

You might be wondering how to confirm that your examples are working. Great question! Support for usage example testing is available via QIIME 2's *TestPluginBase.execute_examples()* and the *results-and-assertions* exposed by the `UsageVariable` class and optionally implemented in its driver-specific subclasses.

You can test your usage examples by making artifact class and file-contents assertions about the UsageVariables returned by `use.action`. These may be run by any usage driver that cares about them, allowing both local smoke testing ("Can my examples be executed successfully?"), and automated integration testing by interfaces like the QIIME 2 library.

Here, we assert that our results are of the expected type.

```
def observed_features_example(use):
    ft = use.init_artifact('feature_table', ft1_factory)
    # NOTE: we must unpack UsageVariables from the returned UsageOutputs
    # if we wish to use their assertion methods.
    a_div_vector, = use.action(
        use.UsageAction(plugin_id='diversity_lib',
                        action_id='observed_features'),
        use.UsageInputs(table=ft),
        use.UsageOutputNames(vector='obs_feat_vector'))

    a_div_vector.assert_output_type('SampleData[AlphaDiversity]')
```

If we pass the Execution driver into this function, it will execute the example, capturing actual Results. By testing that our output is of the correct type, we can assert the type of the output and in the process confirm that our example runs successfully with the given test data.

The easiest way to do this is with the execute_examples() method on `TestPluginBase`. Including a test case that runs `execute_examples() in your unit tests allows you to smoke test them locally by running *unittest* or *pytest*.

A note on scope:

Usage assertions are intended to allow testing of usage drivers and examples, and make it easy for developers to confirm that their *examples* work. Dedicated unit tests provide much more flexilibity and power, and are the preferred way to confirm that your computational *methods* work properly.

By adding the following to `observed_features_example`, we *could* confirm that our test data produced exactly the expected results when executed, but this hack is clunky, because it's reaching beyond the intended use of this assertion.

```
exp = zip(sample_ids, [1, 1, 2, 2, 3])
for id, val in exp:
    a_div_vector.assert_has_line_matching(
        path='alpha-diversity.tsv',
        expression=f'{id}\t{val}'
    )
```

Asserting correct behavior of QIIME 2 Actions or their underlying python functions will probably result in cleaner and more maintainable tests than attempting to do the same using usage examples.

Trying it out

Now that you've created and registered a usage example and confirmed that it "works", let's see it in action! We'll pretend that we just wrote the `q2-feature-table` usage examples above.

1. Make sure your changes are present in the conda environment. `q2-feature-table` is already installed in my QIIME 2 environment, but the version in the environment came from the latest release, not my code. To include my current changes, I can reinstall by running `pip install -e .` from within the repository's root directory.
2. Confirm my environment is using the right version. Before re-installing, I called `conda list q2-feature-table` to check what version of `q2-feature-table` was installed. Re-running that command now, I see the version has changed, indicating that my conda environment knows about the changes I made.
3. I'll check things out first with `q2cli`, so I need to refresh the cache with `qiime dev refresh-cache`.
4. Finally, I run the specific version of `qiime <plugin name> <action> --help` that I'm curious about.

```
>>> qiime feature-table merge --help
Usage: qiime feature-table merge [OPTIONS]

    Combines feature tables using the `overlap_method` provided.

...

Examples:
# ### example: basic ###
qiime feature-table merge \
    --i-tables feature_table1.qza feature_table2.qza \
    --o-merged-table merged_table.qza
# ### example: three tables ###
qiime feature-table merge \
    --i-tables feature_table1.qza feature_table2.qza feature_table3.qza \
    --p-overlap-method sum \
    --o-merged-table merged_table.qza
```

Note that the unique identifiers we created during example definition and registration (e.g. 'feature_table1.qza', 'basic' and 'three tables', and 'merged_table') show up in our rendered example. Note also that `q2cli`'s usage driver was clever enough to format the commands for `q2cli`, including inferring that this action would produce a `.qza` file named `merged_table`.

If we wanted to see what the Artifact API does with our examples, we would confirm that our conda environment included our code (as above). The cache is a q2cli thing, so we don't need to refresh anything, and we would render the examples manually.

```
>>> from qiime2.plugins import feature_table, ArtifactAPIUsage

>>> # Get the examples
>>> examples = feature_table.methods.merge.examples

>>> for example in examples.values():
>>>     # Create a usage driver instance
>>>     use = ArtifactAPIUsage()
>>>     # Inject the usage driver into the example, returning None
>>>     example(use)
>>>     # display the rendered example
>>>     print(use.render())
```

Which renders the following:

```
from qiime2.plugins.feature_table.methods import merge

merged_table, = merge(
    tables=[feature_table1, feature_table2],
)

from qiime2.plugins.feature_table.methods import merge

merged_table, = merge(
    tables=[feature_table1, feature_table2, feature_table3],
    overlap_method='sum',
)
```

The outcome here shows how we might run the `merge` command in the Artifact API, even including the correct import statement.

Comments can provide context

For complex usage examples, you may want to provide additional context to the user. *usage-annotations* are available to help with this. The linked documentation provides worked examples.

1.2.18 Defining different Format validation levels

Part of defining a new `Format` is defining its `_validate_` function. QIIME 2 provides the ability to perform minimal validation or more extensive validation. Unitl a more detailed How-to guide is developed on this topic, you can refer to the *validation discussion in the Framework Explanation on `Formats`*.

1.2.19 Handling exceptions in parallel Pipelines

In developing parallel computing support in QIIME 2, we tried to minimize the edits that are required to existing Pipelines to enable them to run in parallel. In the code we developed in the *plugin tutorial*, for example, the modifications we made were primarily to support the splitting and combining steps - we didn't add anything to explicitly integrate parallel computing. There is one minor exception to this though.

If you have code that looks like the following in a Pipeline that you want to run in parallel:

```
try:
    result1, result2 = some_action(*args)
except SomeException:
    do.something()
```

You must call `_result()` on the return value from `some_action` in the try/except block:

```
try:
    results = some_action(*args)
    result1, result2 = results._result()
except SomeException:
    do.something()
```

If you do not do this, a parallel run of your Pipeline will most likely crash if `SomeException` is raised.

The reason for this is that when the Pipeline is run in parallel, the return value from `some_action` will be a Future that will eventually resolve into your actual results when the parallel processes complete. Calling `._result()` blocks the main thread and waits for results before proceeding from the try/except block.

If you do not call `_result()` in the try block, the Future will most likely resolve into results after the main Python thread has exited the try/except block. This will lead to the exception not being caught, because it is now actually being raised outside of the try/except.

1.3 Explanations

- *Types of QIIME 2 Actions*
- *The structure of QIIME 2 plugin packages*
- *Semantic types, data types, file formats, and artifact classes*
- *Transformers*

1.3.1 Types of QIIME 2 Actions

A QIIME 2 plugin `action` is any operation that accepts parameters and files (:term:`artifact` or metadata) as input, and generates some type of output. `Actions` are interpreted as "commands" by QIIME 2 interfaces and come in three flavors:

1. A `Method` accepts some combination of QIIME 2 `Artifacts` and `Parameters` as input, and produces one or more QIIME 2 artifacts as output. These output artifacts could subsequently be used as input to other QIIME 2 `Methods` or `Visualizers`. `Methods` can produce intermediate or terminal outputs in a QIIME 2 analysis. For example, the `rarefy` method defined in the `q2-feature-table` plugin accepts a feature table artifact and sampling depth as input and produces a rarefied feature table artifact as output. This rarefied feature table artifact could then be used in another analysis, such as alpha diversity calculations provided by the `alpha` method in `q2-diversity`.
2.

A `Visualizer` is similar to a `Method` in that it accepts some combination of QIIME 2 `Artifacts` and `Parameters` as input. In contrast to a method, a visualizer produces exactly one `Visualization` as output. Visualizations, by definition, cannot be used as input to other QIIME 2 methods or visualizers. Thus, visualizers can only produce terminal output in a QIIME 2 analysis.

3.

A `Pipeline` accepts some combination of QIIME 2 `Artifacts` and `Parameters` as input and produces one or more artifacts and/or visualizations as output. It does so by incorporating one or more `methods` and/or `visualizers` into a single registered `action`.

1.3.2 The structure of QIIME 2 plugin packages

QIIME 2 doesn't put restrictions on how you structure the Python package that contains your plugin, but there are some conventions that experienced developers follow. This *Explanation* article will discuss these conventions in the context of the plugin developed in *Tutorial: A step-by-step guide to building your first QIIME 2 plugin*. Specifically, we'll look at the initial commit in that repository.

> **Tip**
>
> `tree` often isn't installed by default, but you should be able to install it with your preferred package manager (e.g., with `apt-get`, `homebrew`, or whatever you use to install software on your system).

I'm going to use the `tree` command to get a convenient view of all of the files and directories my plugin package after my first commit. Then we'll start from the top and talk about what each of these are.

```
$ tree -a q2-dwq2
q2-dwq2
├── .git
│   └── # many files
├── .github
│   └── workflows
│       └── ci.yml
├── .gitignore
├── LICENSE
```

(continues on next page)

(continued from previous page)

```
├── MANIFEST.in
├── Makefile
├── README.md
├── ci
│   └── recipe
│       └── meta.yaml
├── q2_dwq2
│   ├── __init__.py
│   ├── _methods.py
│   ├── _version.py
│   ├── citations.bib
│   ├── plugin_setup.py
│   └── tests
│       ├── __init__.py
│       ├── data
│       │   └── table-1.biom
│       └── test_methods.py
├── setup.cfg
├── setup.py
└── versioneer.py
```

q2-dwq2

`q2-dwq2` is the top-level directory containing all of the files in the Python package.

q2-dwq2/.git

I am maintaining my plugin code using git for version control, and the `.git` directory contains all of the information for managing the `.git` repository. There are many files in there, and going through them is out-of-scope for this book, but poke through that directory if you're interested. Nothing magical is happening with git repositories: the files in this directory are used by git clients to do all of the cool stuff that git does for us. Don't ever edit files in this directory directly.

q2-dwq2/.github

This directory contains information used by GitHub, and is here because I maintain my git repository for this plugin on GitHub. GitHub will use some files in this directory in special ways if they exist (see here). In my case, this directory contains a single GitHub Action file, `.github/workflows/ci.yml`, that was added when the repository was built from the cookiecutter template. This GitHub Action is what builds the plugin and runs the tests when pulls requests or commits are submitted to GitHub. To learn more about GitHub Actions, see GitHub's documentation. An additional resource that may help is GitHub Automation for Scientists.

q2-dwq2/.gitignore

A file used by git that specifies filename patterns that should be ignored by git (and therefore not included in revision control). This helps keep your repository neat and clean by excluding things like temporary files created by text editors or operating systems. There are lots of examples that you can use or build from here.

q2-dwq2/LICENSE

File containing the software's license. Naming the file this way, and storing it in the top-level directory, enables it to be recognized easily by users or systems (such as GitHub).

q2-dwq2/MANIFEST.in

Used by `setuptools`, along with `setupy.py` and `setup.cfg`, to explicitly define files that should or shouldn't be included in the Python package.

q2-dwq2/Makefile

`make` instructions for building the Python module, running its tests, and more. When you run a command like `make test` or `make dev`, you are applying instructions defined in this file. `make` is a powerful tool, to put it lightly, and it has been around since pre-historic times (i.e., 1976).

q2-dwq2/README.md

The project's readme file. This is often when someone interested in your Python package will first look for information. If you manage your project on GitHub, this will be displayed on the repositories front page.

q2-dwq2/ci

This directory contains information used by the Continuous Integration system, which is used by the GitHub Action workflow referenced in `.github/workflows/ci.yml`. The one file contained here in this case, `ci/recipe/meta.yaml`, provides instructions for how this plugin can be built and tested.

q2-dwq2/q2_dwq2

The top-level module directory. This is where all of files relevant to the use of this code with `Python` are stored. All files not included in this directory can be considered metadata about the Python module.

q2-dwq2/q2_dwq2/__init__.py

A special file whose existance (even if the file is empty) specifies that this directory is a Python module. This will often contain `import` statements that the developer wants to propagate up to be module-level imports, enabling statements like `from qiime2 import Artifact`.

q2-dwq2/q2_dwq2/_methods.py

This is not a required file, but in this plugin it's used to store code for functions that will ultimately be registered as *Methods*. As a plugin grows, it may make sense to consider reorganization such as creating a `_methods` directory that contains files with code for each individual Method.

By convention in Python, files, functions, or objects (or anything else) whose name starts with an _ should be treated as private. In other words, outside of this specific code base, anything named with a leading underscore shouldn't be referenced or used directly. This leaves the developer free to make interface changes (such as renaming the `_methods.py`) file without breaking other people's code.

q2-dwq2/q2_dwq2/_version.py

This is a file created by The Versioneer to assist with creating versions of software from information in the `.git` directory, if it exists. You shouldn't ever edit this file directly.

q2-dwq2/q2_dwq2/citations.bib

This file stores any citations that QIIME 2 will reference for this plugin in BibTeX format. The relative filepath is specified when the Plugin object is initialized, so can be called whatever you'd like.

q2-dwq2/q2_dwq2/plugin_setup.py

By convention, this file is where the QIIME 2 `Plugin` object is instantiated and where actions and other information are registered to the plugin. Again, this file can be called anything and live in other places, but it's pretty standard across plugins at this stage so it's a good idea to just adopt this naming convention in your plugin. (For example, the first thing I typically do when someone sends me their plugin for feedback is read their `plugin_setup.py` file.)

q2-dwq2/q2_dwq2/tests

This directory contains all unit tests for functionality in the plugin. Any associated test data files are generally nested under this directory.

q2-dwq2/q2_dwq2/tests/__init__.py

The file initializing `q2_dwq2/tests` as a submodule in this Python package.

q2-dwq2/q2_dwq2/tests/data

A directory containing any data files that are used in tests.

q2-dwq2/q2_dwq2/tests/test_methods.py

The file containing unit tests of the functionality in the module's `_methods.py` By convention, the naming of test files roughly parallels the naming of the files they are testing.

q2-dwq2/q2_dwq2/setup.cfg

A file containing general configuration information related to the Python module. For example, there is information in here that helps The Versioneer find the inforamtion that it needs to create version numbers.

q2-dwq2/q2_dwq2/setup.py

A special file for the Python pacakge that sets up the Python module. An important component in this file for QIIME 2 is that the `qiime2.plugins` entry point is defined:

```
setup(
    ...
    entry_points={
        "qiime2.plugins": ["q2-dwq2=q2_dwq2.plugin_setup:plugin"]
    },
    ...
```

This allows the QIIME 2 `PluginManager` to load the module and determine if one or mroe QIIME 2 plugins are defined in the module, and if so where they can be imported from. In this case, one plugin is registered (`q2-dwq2`) and it can be imported from `q2_dwq2.plugin_setup` through the variable name `plugin`. If you prefer to not follow the naming convention described above with respect to `plugin_setup.py`, this is where you can let the `PluginManager` know where it should be looking for your plugin(s).

q2-dwq2/q2_dwq2/versioneer.py

Another file created by The Versioneer to assist with creating versions of software from information in the `.git` directory, if it exists.

1.3.3 Semantic types, data types, file formats, and artifact classes

The term *type* is overloaded with a few different concepts. The goal of this *Explanation* article is to disambiguate how it's used in QIIME 2. To achieve this, we'll discuss two ways that it's commonly used, and then introduce a third way that it's used less frequently but which is important to QIIME 2. This document will then conclude with a discussion of *artifact classes*, which is the most relevant of these concepts for new plugin developers.

The three kinds of types that are used in QIIME 2, and which we'll start this explanation with, are **file types**, **data types**, and **semantic types**.

File types (or formats) and data types (or objects)

File types (or formats) refer to what you probably think of when you hear that phrase: the format of a file used to store some data. For example, newick is a file type that is used for storing phylogenetic trees. Files are used most commonly for archiving data when it's not actively in use. Data types refer to how data is represented in a computer's memory (i.e., RAM) while it's actively in use, such as the data structure or object class that a file is loaded into.

For example, if you are adding a root to an unrooted phylogenetic tree, you may use a tool like IQ-Tree. You would provide a path to the file containing the unrooted phylogenetic tree to IQ-Tree, and IQ-Tree would load that tree into some object in the computer's memory to work on it. The object that IQ-Tree uses internally to represent the phylogenetic tree is synonymous with *data type*, as used here. The kind of object that is used is a decision made by the developers of IQ-Tree based on available functionality, efficiency for an operation they plan to carry out, their familiarity with the object, or something else. If IQ-Tree successfully completes the requested rooting operation, it could then write the resulting tree from its internal data type into a new newick-formatted file on the hard disk, and exit.

One thing to notice from this example is that there are at least three *independent* choices being made by the developer regarding *types*: what file type to use as input, what data type to use internally, and what file type to use as output. Users of command line software, like IQ-Tree, shouldn't need to know or care about what data types are used internally by a program. They just need to know what file types are used as input and output. Software developers, on the other hand, should care a lot about what data types are used by their program: choosing an appropriate type can have huge impacts on the performance of the software, for example.

Semantic types

The third *type* that is important in QIIME 2 is the semantic type of data. This is a representation of the *meaning* of the data, which is not necessarily represented by either a file type or a data type. For example, two semantic types used in QIIME 2 are `Phylogeny[Rooted]` and `Phylogeny[Unrooted]`, which are used to represent rooted and unrooted phylogenetic trees, respectively. Both rooted and unrooted trees are commonly described in newick-formatted files, and typically a computer program would need to parse a file to know if the tree it describes is rooted or unrooted. For large trees, this can be a slow operation.

There are some operations, such as rooting a tree, that only make sense to perform on unrooted trees. So, if you have a very large tree that you want to root, you may provide a newick file to a program that will perform that rooting. If you accidentally provide a rooted tree, it may take the program some time to parse the file (say 20 minutes) after which it may fail if it discovers that the tree is already rooted. That sort of delayed notification can be very frustrating as a user, since it's easily missed until a lot of time has passed. For example, I often will start a long-running command on my university cluster computer just before the weekend. I'll typically check on the job for a few minutes, to make sure that it seems to be starting ok. I may then leave, with the hope that the job completes over the weekend and I'll have data to work with on Monday morning. It's very frustrating to come in Monday morning and find out that my job failed just a few minutes after I left on Friday for a reason that I could have quickly addressed had I known in time.

Note

There's actually a worse outcome than a delayed error from a computer program when inappropriate input is provided. When a program fails and provides an error message to the user, whether or not that error message helps the user solve the problem, the program has failed loudly. Something went wrong, and it told the user about it. The program could instead fail quietly. This might happen if the program doesn't realize the input the user provided is inappropriate (e.g., an already rooted tree is provided to a program that roots an unrooted phylogenetic tree), and it runs the rooted tree through its algorithm, misinterprets something because it was provided with the wrong input, and generates an incorrect rooted tree as a result. Quiet failures can be very difficult or impossible for a user to detect, because it looks like everything has worked as expected. Failing quietly is *much* worse than failing loudly: it could waste many hours of your time, and could even lead to you publishing invalid findings.

QIIME 2 semantic types help with this, because they provide information on what the data in a QIIME 2 `.qza` file means without having to parse anything in the `data` directory. All QIIME 2 artifacts are are associated with a semantic type that define what the data they contain means.

Putting it together

There is a many-to-many relationship between file types, data types, and semantic types. It's possible that a given semantic type could be represented on disk by different file types. That's well exemplified by the many different formats that are used to store demultiplexed sequence and sequence quality data. For example, this may be in one a few variants of the fastq format, or in the fasta/qual format. Additionally, data from multiple samples may be contained in one single file or split into per-sample files. Regardless of which of these file formats the data is stored in, the data has the same semantic meaning (in this case represented by the semantic type `SampleData[SequencesWithQuality]`. Similarly, the data type used in memory might differ depending on what operations are to be performed on the data, or based on the preference of the programmer.

QIIME 2 uses the semantic type `FeatureTable[Frequency]` to represent the idea of a feature table that contains counts of features (e.g., bacterial genera) on a per sample basis. Many different actions can be applied to `FeatureTable[Frequency]` artifacts in QIIME 2. When a plugin developer defines a new action that takes a `FeatureTable[Frequency]` as input, they can choose whether to load the table into a `pandas.DataFrame` or `biom.Table` object, which are two different data types. Our example plugin `q2-dwq2` initially defines a action called `duplicate_table` which takes a `FeatureTable[Frequency]` as input, and generates the same as its output. The function registered to this action declares that it will "view" the input `table` as a `pd.DataFrame`, and also return the output as a `pd.DataFrame`.

Each kind of type discussed here represents different information about the data: how it's stored on disk (file type), how it's used by a function (its data type), and what it represents (its semantic type). The motivation for creating QIIME 2's semantic type system was to avoid issues that can arise from providing inappropriate data to actions. The semantic type system also helps users and developers better understand the intent of QIIME 2 actions by assigning meaning to the input and output, and allows for the discovery of new potentially relevant QIIME 2 actions.

Artifact classes

Most of the time, plugin developers are more concerned with *artifact classes*, rather than semantic types and formats directly, though we only recently starting transitioning the language used in our documentation to reflect this. You may still see some outdated usage, such as treating the terms *semantic type* and *artifact class* as synonyms, especially in older video content. Sorry for the confusion!

An artifact class is a kind of QIIME 2 artifact that can exist, and is defined by the association of a semantic type with a format (e.g., this is done in our example plugin when we *create the `SingleDNASequence` artifact class*). Together, these indicate what an artifact is intended to represent, and how its data is stored internally. You can see the artifact classes that your deployment of QIIME 2 is aware of by calling `qiime tools list-types`. Artifact classes (not semantic types) are associated with input and output from QIIME 2 actions.

1.3.4 Transformers

`Transformers` are functions for converting between formats and data types. These transformers are typically defined along with the artifact classes that they are designed to work with, and `q2-types` provides a number of common types and associated transformers. Plugins can define their own transformers.

How are transformers used by a plugin?

`Transformers` are not called directly at any time within a plugin. Transformations are handled by the QIIME 2 framework, as long as the appropriate `transformers` are registered. The framework identifies the *input* `Artifact` source format for the transformation as the format registered to the `Artifact`'s *artifact class*, and it identifies the transformation's destination type based on the functional annotation associated with the input in an `Action`'s registered function. When *output* is generated by a `Method`, the framework identifies the source type for the transformation as the registered function's output annotation, and the destination format for the output `Artifact` from the output's artifact class defined in the `Action`'s registration.

For example, we can see how functional annotations define input and output formats in `q2_diversity.beta_phylogenetic`:

```
def beta_phylogenetic(table: biom.Table,
                      phylogeny: skbio.TreeNode,
                      metric: str)-> skbio.DistanceMatrix:
```

This function requires `biom.Table` and `skbio.TreeNode` objects as input, and produces an `skbio.DistanceMatrix` object as output.

We can examine the first few lines of the `Action` registration for this function to determine the artifact classes of these input and output objects:

```
plugin.pipelines.register_function(
    function=q2_diversity.beta_phylogenetic,
    inputs={'table': FeatureTable[Frequency],
            'phylogeny': Phylogeny[Rooted]},
    parameters={'metric': Str % Choices(beta.phylogenetic_metrics())},
    outputs=[('distance_matrix', DistanceMatrix)],
```

This pair of code examples illustrate that the `biom.Table` object used in `beta_phylogenetic` begins its life as a `FeatureTable[Frequency]` artifact, and the `skbio.TreeNode` comes from a `Phylogeny[Rooted]` artifact. The output `skbio.DistanceMatrix` generated by `beta_phylogenetic` must be coerced to become a `DistanceMatrix` artifact. The QIIME 2 framework takes care of all of those conversions for you, provided the appropriate transformers have been defined and registered (*which you can learn how to do here*).

1.4 References

- *Plugin development anti-patterns*
- *Plugin Development API*
  - *Plugin & Registration*
  - *Formats*
  - *Types*
  - *Citations*

1.4.1 Plugin development anti-patterns

> "An anti-pattern in software engineering, project management, and business processes is a common response to a recurring problem that is usually ineffective and risks being highly counterproductive." Source: Wikipedia – Anti-patterns (Accessed 11 January 2024; last edited 5 December 2023).

This section documents common anti-patterns that we observe in plugin development. Generally speaking, the things documented on this page lead to plugins that, for the most part, work. So why avoid them?

The extra work of making your methods accessible through QIIME 2, including things like defining the artifact classes of your inputs and outputs, pays off in that you get things like cross-interface support and Provenance Replay for your methods for free. You don't have to test these things: we promise they'll work (and if they don't there is probably a bug somewhere that is our responsibility to fix - please do let us know). If you find yourself feeling like you need to apply one of these anti-patterns in your plugin, feel free to reach out through the QIIME 2 Forum Developer Discussion. We'll help you figure out the QIIME-y way to achieve your goal.

Warning

If you adopt the anti-patterns described on this page, the QIIME 2 framework developers explicitly provide no guarantee that your plugins will produce useful, replayable provenance, will be fully accessible (or even minimally functional) across the different interfaces that exist (e.g., Python and R APIs, command line interfaces, and graphical interfaces), or won't fail in other weird ways. We also provide no guarantee that these anti-patterns will continue to work with future versions of the QIIME 2 framework.

If features such as access to your methods through different interfaces or their compatibility with Provenance Replay aren't a priority for your work (e.g., because you are prototyping, or because you have a provenance tracking mechanism external to QIIME 2), the extra effort required to build QIIME 2 plugins may not be worth it.

Providing input or output filepaths as parameters

It's not uncommon for new plugin developers to provide paths to inputs or outputs using `parameter` arguments, rather than `input` or `output` arguments, when defining and registering `Action`. For example:

```
...

def my_action(feature_table: pd.DataFrame,
              taxonomy: pd.DataFrame,
              an_input_filepath: str,
              an_output_filepath: str) -> pd.DataFrame:
    with open(an_input_filepath) as inf:
        ...

    with open(an_output_filepath, 'w') as outf:
        ...
```

(continues on next page)

(continued from previous page)

```
    # return an empty dataframe as the dummy output
    return pd.DataFrame()

...

plugin.methods.register_function(
    function=my_action,
    inputs={'feature_table': FeatureTable[Frequency],
            'taxonomy': FeatureData[Taxonomy]},
    parameters={'an_input_filepath': Str,
                'an_output_filepath': Str},
    outputs=[('dummy_output', FeatureTable[Frequency])],
    input_descriptions={
        'feature_table': 'The input feature table.',
        'taxonomy': 'The input taxonomy.'},
    parameter_descriptions={
        'an_input_filepath': 'The input text file.',
        'an_output_filepath': 'The path where output should be written.'
        },
    output_descriptions={'dummy_output': 'Ignore me.'},
    name='My cool new action.',
    description=("Apply an operation to a feature table."),
    citations=[]
)
```

This approach circumvents the need to associate `an_input_filepath` and `an_output_filepath` with an artifact class (which may need to be defined, if it's a new type). This is convenient for the developer, and under some circumstances QIIME 2 will appear to work ok, but it's problematic for at least a couple of reasons.

First, some QIIME 2 interfaces (e.g., web-based interfaces, such as the QIIME 2 Galaxy interface) won't work correctly. The user would have to type the values for `an_input_filepath` and `an_output_filepath` in text fields, such as `C:\Path\to\my\input.txt` or `/path/to/my/input.txt`, not for example select paths on their system through a file upload/download dialog box, as they would most likely expect when providing files to or receiving files from a web server through their browser. They likely won't know path to type (it would have to be a file on the web server, since the `Action` only knows that this is a string, not that handling as a filepath is needed), and even if they managed to provide the correct input path, they wouldn't have access to the output that is created through the interface that they're interacting with, because it's just being written somewhere on the web server (which probably wouldn't even be allowed). Expecting arbitrary paths to the two Python `open` calls in this example to work on the server would be unreliable at best. In practice, this just won't work.

Next, and more broadly problematic, the entries for `an_input_filepath` and `an_output_filepath` in QIIME 2's data provenance would simply be the paths that were provided when the action was called. This will result in incomplete data provenance for *all* downstream `Results`. There will be no UUID associated with `an_input_filepath` and `an_output_filepath` (so the data they contain couldn't be unambiguously identified by QIIME 2). As a result, the entries in provenance will only be meaningful to provenance consumers (users or machines) that know how to interpret the paths (e.g., what computer the paths are relative to) and are confident that the files at those locations haven't changed since the action was run.

All data that is provided as input to or generated as output from QIIME 2 `Action(s)` should be in the form of QIIME 2 Artifacts. This is essential for ensuring that workflows using your `Action(s)` will be fully reproducible, and that your plugin will be accessible to users with varying levels of computational expertise. These are two of the key benefits of making your methods accessible through QIIME 2 plugins, and are expectations of QIIME 2 users. The cost is going through the upfront work of associating inputs with artifact classes.

Skipping format validation

To save time (either during development, or at run time) plugin developers will sometimes skip implementation of format validation when they create new formats. For example:

```
from qiime2.plugin import model

class MyNewFormat(model.TextFileFormat):
    def _validate_(self, level):
        pass
```

This can lead to arbitrary input files being loaded into artifacts, even if they contain some errors or aren't even remotely the right type of data to be associated with an artifact class. QIIME 2 actions that use an artifact created with this format will assume that validation has already been performed to avoid costly validation processed being applied multiple times to the same data. Sometimes this will work fine, but sometimes it will crash and burn in the hands of your users.

If a user provides invalid data that passes through one of these validators, in the best case QIIME 2 will crash, and most likely with an obscure error message since the `Action(s)` using the data aren't expecting errors in the data (since it has already gone through validation). That frustrates users who may walk away with a negative view of your plugin, or a negative view of QIIME 2 as a whole if they are not aware of the differece between QIIME 2 plugins and QIIME 2. That's bad for everyone. Consider how often you go back to use software (e.g., a phone app) that you tried and decided was buggy.

It's also possible that the user won't get an error message when they provide invalid data, but instead everything will appear to work correctly but in reality generate meaningless results because the input data was invalid (i.e., garbage in → garbage out). In this case, failure to validate could lead to your user being misinformed, which can have major repercussions downstream including retracted publications, missed opportunities for scientific discovery, or even clinical misdiagnoses. Users will likely blame you for these outcomes!

If you're skipping validation to save time during development, consider what your goals are for your plugin development effort. If this is something you'll use only in your own work, and you know the input is going to be valid (e.g., because it's tested elsewhere first), then this may not be a big problem. However if you're planning to distribute your plugin to users, you'll never know where their data is coming from. Performing input validation can save you and your users lots of time and frustration.

If you're skipping validation to reduce run time, you should instead consider allowing for different levels of validation, which is built into QIIME 2's format validation process. You can read about how to use this in *Defining different Format validation levels*. But be aware that validation that is too minimal can lead to all of the same problems as no validation at all.

1.4.2 Plugin Development API

This section details the public API for plugin development. Broadly speaking, everything that is necessary to build a QIIME 2 plugin is available in `qiime2.plugin` or `qiime2.metadata`.

Individual Topics

- *Plugin & Registration*
- *Formats*
- *Types*
- *Citations*
- *Testing*
- *Utilities*
- *Pipeline Context Object*
- *Usage Examples*

Plugin API List

Plugin Object

qiime2.plugin.Plugin	A QIIME 2 Plugin.

Registration

plugin.PluginMethods.register_function	Register a method to the associated plugin.
plugin.PluginVisualizers.register_function	Register a visualizer to the associated plugin.
plugin.PluginPipelines.register_function	Register a pipeline to the associated plugin.
Plugin.register_validator	**Decorator** which registers a validator
Plugin.register_transformer	**Decorator** which registers a transformer to convert data
Plugin.register_formats	Register file formats to the plugin
Plugin.register_views	Register arbitrary views (Python classes or formats) to the plugin
Plugin.register_semantic_types	Register semantic type fragments to the plugin
Plugin.register_semantic_type_to_forma	Connect a semantic type expression to a format.
Plugin.register_artifact_class	Register an artifact class which defines an Artifact

Formats

qiime2.plugin.TextFileFormat	
qiime2.plugin.BinaryFileFormat	
qiime2.plugin.DirectoryFormat	
qiime2.plugin.SingleFileDirectoryForma	
qiime2.plugin.ValidationError	

Types

qiime2.plugin.SemanticType	Create a new semantic type.
qiime2.plugin.Properties	A one or more semantic properties to add to an existing type.
qiime2.plugin.Visualization	The type of a QIIME 2 Visualization.
qiime2.plugin.Bool	A boolean value (`True`/`False`).
qiime2.plugin.Str	A string of Unicode characters (i.e. text).
qiime2.plugin.Int	An integer without any particular bounds.
qiime2.plugin.Float	A 64 bit floating point number.
qiime2.plugin.Threads	The number of logical threads to use (OS threads/CPUs/Cores).
qiime2.plugin.Jobs	The number of jobs to submit as an integer that is greater than zero (exclusive).
qiime2.plugin.Choices	A predicate which defines a set of allowable values.
qiime2.plugin.Range	A predicate which defines a contiguous range of allowable values.
qiime2.plugin.Start	Shorthand to generate a *Range*
qiime2.plugin.End	Shorthand to generate a *Range*
qiime2.plugin.Metadata	Tabular metadata where unique identifiers can be associated with columns.
qiime2.plugin.MetadataColumn	A column of a *qiime2.Metadata*.
qiime2.plugin.Categorical	The categorical variant for *MetadataColumn*.
qiime2.plugin.Numeric	The Numeric variant for *MetadataColumn*.
qiime2.plugin.Set	**Deprecated** - use List or Collection instead
qiime2.plugin.List	An alias for a *Collection* with numeric auto-incrementing keys.
qiime2.plugin.Collection	An ordered set of key-value pairs.
qiime2.plugin.TypeMap	A table of input types which match to output types.
qiime2.plugin.TypeMatch	A trivial *TypeMap* such that every entry maps to itself.

Citations

qiime2.plugin.Citations	A simple subclass of `collections.OrderedDict` but iterates over values instead of keys by default.
qiime2.plugin.CitationRecord	A `collections.namedtuple()` of bibtex entry type and entry fields.

Testing

qiime2.plugin.testing.TestPluginBase	Test harness for simplifying testing QIIME 2 plugins.
qiime2.plugin.testing.assert_no_nans_in_tables	Checks for NaNs present in any of the tables in the indicated file then resets to the head of the file.

Utilities

qiime2.util.duplicate	Alternative to `shutil.copyfile()` this will use `os.link()` when possible.
qiime2.util.redirected_stdio	A context manager to redirect stdio to a new file (if provided).
qiime2.plugin.util.transform	
qiime2.plugin.util.get_available_cores	Finds the number of currently available (logical) cores.

Additional Objects

These objects are not part of the `qiime2.plugin` module, but are commonly used by plugins (and users).

Metadata

qiime2.Metadata	Store metadata associated with identifiers in a study.
qiime2.MetadataColumn	Abstract base class representing a single metadata column.
qiime2.NumericMetadataColumn	A single metadata column containing numeric data.
qiime2.CategoricalMetadataColumn	A single metadata column containing categorical data.

Pipeline Context Object (ctx)

Context.get_action(plugin, action)	Return a function matching the callable API of an action.
Context.make_artifact(type, view[, view_type])	Return a new artifact from a given view.

Usage Examples

Usage.init_artifact	Communicate that an artifact will be needed.
Usage.init_artifact_from_url	Obtain an artifact from a url.
Usage.init_artifact_collection	Communicate that a result collection containing artifacts will be needed.
Usage.init_metadata	Communicate that metadata will be needed.
Usage.init_metadata_from_url	Obtain metadata from a url.
Usage.init_format	Communicate that a file/directory format will be needed.
Usage.import_from_format	Communicate that an import should be done.
Usage.construct_artifact_collection	Return a UsageVariable of type artifact_collection given a list or dict of its members.
Usage.get_artifact_collection_member	Accesses and returns a member of a ResultCollection as a UsageVariable.
Usage.get_metadata_column	Communicate that a column should be retrieved.
Usage.view_as_metadata	Communicate that an artifact should be views as metadata.
Usage.merge_metadata	Communicate that these metadata should be merged.
Usage.comment	Communicate that a comment should be made.
Usage.help	Communicate that help text should be displayed.
Usage.peek	Communicate that an artifact should be peeked at.
Usage.action	Communicate that some action should be performed.
Usage.UsageAction	An object which represents a deferred lookup for a QIIME 2 action.
Usage.UsageInputs	A dict-like mapping of parameters to arguments for invoking an action.
Usage.UsageOutputNames	A dict-like mapping of action outputs to desired names.
UsageOutputs	A vanity class over *qiime2.sdk.Results*.
UsageVariable	A variable which represents some QIIME 2 generate-able value.
UsageVariable.assert_has_line_matching	Communicate that the result of this variable should match a regex.
UsageVariable.assert_output_type	Communicate that this variable should have a given semantic type.

Plugin & Registration

class qiime2.plugin.**Plugin**(*name*, *version*, *website*, *package=None*, *project_name=None*, *citation_text=None*, *user_support_text=None*, *short_description=None*, *description=None*, *citations=None*)

A QIIME 2 Plugin.

An instance of this class defines all features of a given plugin and is instantiated as a module global (i.e. a singleton).

Parameters

- **name** (*str*) – The name of the plugin (hyphens will be automatically replaced for the plugin ID)
- **version** (*str*) – The version of the plugin (this should match the package version)
- **website** (*str*) – A URL to find more information about this plugin
- **package** (*str | None*) – The Python package name of your plugin. This is largely defunct and is set by the entry-point during plugin loading.
- **project_name** (*str | None*) – The external name of the plugin (distinct from the internal `name`, i.e. q2-my-plugin vs my-plugin). Also defunct and set by the entry-point during plugin loading.
- **citation_text** (*Any | None*) – **Deprecated**. Does nothing. Use `citations` instead.
- **user_support_text** (*str | None*) – A message about where to find user support. The default will suggest users visit the QIIME 2 forum.
- **short_description** (*str | None*) – A small (single-line) description to help identify the plugin in a list.
- **description** (*str | None*) – A more complete description of the plugins purpose.
- **citations** (*CitationRecord or list of CitationRecord*) – Citation(s) to associate with a result whenever this plugin is used. Can also use an entire *Citations* object.

Examples

```
>>> plugin = Plugin('my-plugin', __version__, website)
```

register_formats(**formats*, *citations=None*)

Register file formats to the plugin

Parameters

- ***formats** (*TextFileFormat, BinaryFileFormat, or DirectoryFormat*) – Formats which are created or defined by this plugin.
- **citations** (*CitatonRecord or list of CitationRecord*) – Citation(s) to associate with a result whenever this format is used internally. Can also use an entire *Citations* object.

Return type
None

Notes

SingleFileDirectoryFormat() returns a *DirectoryFormat*

register_views (**views*, *citations=None*)

Register arbitrary views (Python classes or formats) to the plugin

Parameters

- ***views** (*type*) – Views which are created or defined by this plugin
- **citations** (*CitationRecord or list of CitationRecord*) – Citation(s) to associate with a result whenever this format is used internally. Can also use an entire *Citations* object.

Return type
None

register_validator (*semantic_expression*)

Decorator which registers a validator

Parameters
semantic_expression (*semantic type expression*) – An expression (may include operators like union (`|`)) which will be compared to an artifact. If the artifact is in the domain of the type expression, it will be transformed into a view that matches the type annotation of the decorated function and that function will be executed.

Returns
A decorator which can be applied to a function which takes `data` as the first argument and `level` as the second. `data` must be annotated with a type which will become the transformed view. `level` should accept the strings `"min"` and `"max"`. It does not necessarily need to change behavior based on `level`, but it is encouraged where it could save the user time.

Return type
decorator

Examples

```
>>> @plugin.register_validator(Foo | Bar)
... def validate_something(data: pd.DataFrame,
...                        level: Literal['min', 'max']):
...     if data.empty:
...         raise ValidationError("This data is empty.")
```

register_transformer (*_fn=None*, ***, *citations=None*)

Decorator which registers a transformer to convert data

This decorator may be used with or without arguments.

Parameters

- **_fn** (*Callable*) – Ignore this parameter as it is the mechanism to allow argumentless decoration
- **citations** (*CitationRecord or list of CitationRecord*) – Citation(s) to associate with a result whenever this transformer is used internally. Can also use an entire *Citations* object.

Returns
A decorator which can be applied to a function which takes a single argument with a type annotation and a single return annotation. This decorated function will then be executed whenever data matching the type annotation exists and there is a need to view that data as the return annotation.

Return type
decorator

Notes

Since the function is entirely defined by the input and output type, the name of the function is usually unimportant and only adds noise. We tend to use `_<number>` as the name, but any other name may be used.

Examples

```
>>> @plugin.register_transformer
... def _0(data: pd.DataFrame) -> CSVFormat:
...     ff = CSVFormat()
...     with ff.open() as fh:
...         data.write_csv(fh)
...     return ff
```

```
>>> @plugin.register_transformer(citations=[
...     citations['baerheim1994effect'],
...     citations['silvers1997effects']
... ])
... def _1(ff: CSVFormat) -> pd.DataFrame:
...     with ff.open() as fh:
...         return pd.read_csv(fh)
```

register_semantic_types(**type_fragments*)

Register semantic type fragments to the plugin

Parameters
***type_fragments** (*semantic types*) – Semantic type fragments to register. If a plugin had defined types for this expression: `BaseType[Variant1 | Variant2]` then `type_fragments` would be `BaseType, Variant1, Variant2`

Return type
None

Examples

```
>>> plugin.register_semantic_types(BaseType, Variant1, Variant2)
```

register_semantic_type_to_format(*semantic_type*, *artifact_format=None*, *directory_format=None*)

Connect a semantic type expression to a format. **Deprecated**

Permits an arbitrary type expression and expands it to all concrete variants which are then associated with the supplied `directory_format`.

As this does not support documentation or examples, it is recommended to use *Plugin.register_artifact_class()* instead, which takes a single concrete expression, but allows for additional specific information.

Parameters

- **semantic_type** (*semantic type expression*) – A semantic type expression which may include operators.
- **artifact_format** (*type*) – **Super deprecated**
- **directory_format** (*subclass of DirectoryFormat*) – A directory format which will define the `/data/` directory of a stored artifact.

Return type
None

register_artifact_class (*semantic_type*, *directory_format*, *description=None*, *examples=None*)

Register an artifact class which defines an Artifact

Parameters

- **semantic_type** (*type expression*) – A *concrete* type expression which will be the type of this artifact class.
- **directory_format** (*subclass of DirectoryFormat*) – A directory format which will define the `/data/` directory of a stored artifact.
- **description** (*str*) – A description of what this artifact will represent.
- **examples** (*dict[str, callable]*) – A dict of example name to usage example functions which take a single argument (the usage driver, a.k.a. `use`). Each function which will demonstrate importing this data when executed.

Return type
None

Examples

```
>>> plugin.register_artifact_class(
...     semantic_type=BaseType[Variant1],
...     directory_format=CSVDirFormat,
...     description="This data represents something important.",
...     examples={
...         'example1': example_function_variant1,
...         'example2': example_function_variant2
...     }
... )
```

Action registration

The following classes exist only on an instantiated *Plugin* object and are generally accessed via `plugin.methods`, `plugin.visualizers`, and `plugin.pipelines`. At this time, `register_function` is the only interesting method for a plugin developer. Otherwise these objects are essentially dictionaries to makes generating interfaces convenient.

class qiime2.plugin.plugin.**PluginMethods**(*plugin*)

Accessed via `plugin.methods`

register_function(*function*, *inputs*, *parameters*, *outputs*, *name*, *description*, *input_descriptions=None*, *parameter_descriptions=None*, *output_descriptions=None*, *citations=None*, *deprecated=False*, *examples=None*)

Register a method to the associated plugin.

Parameters

- **function** (*callable*) – A function which will be called when the user uses this method.
- **inputs** (*dict[str, type expression]*) – A dictionary of function-parameter names to semantic type expressions. The keys of the dictionary must match the parameter names which are on the `function`. Collections and type variables are also permitted so long as they contain semantic types.
- **parameters** (*dict[str, type expression]*) – A dictionary of function parameter names to primitive type expressions. The keys of the dictionary must match the parameter names which are on the `function`. Collections and type variables are also permitted so long as they contain primitive types.
- **outputs** (*dict[str, type expression]*) – A dictionary of named outputs to *concrete* semantic types. The keys of the dictionary will become the parameter names for these outputs and must match the number of annotated outputs on the `function`. Collections and type variables are also permitted so long as they contain semantic types. (In older versions of QIIME 2 this was a list of tuples as dictionaries were not yet ordered.)
- **name** (*str*) – A short description, ideally a human-oriented name or title.
- **description** (*str*) – A longer description of the action.
- **input_descriptions** (*dict[str, str]*) – A dictionary of input names (see `inputs`) to descriptions.
- **parameter_descriptions** (*dict[str, str]*) – A dictionary of parameter names (see `parameters`) to descriptions.
- **output_descriptions** (*dict[str, str]*) – A dictionary of output names (see `outputs`) to descriptions.
- **citations** (`CitationRecord` *or list of* `CitationRecord`) – Citation(s) to associate with a result whenever this action is used. Can also use an entire *Citations* object.
- **deprecated** (*bool*) – Whether this action is deprecated and should be migrated away from.
- **examples** (*dict[str, callable]*) – A dict of example name to usage example functions which take a single argument (the usage driver, a.k.a. `use`). Each function will carry out some example situation for this action when executed.

Return type

None

Examples

```
>>> from qiime2.plugin import Int
>>> plugin.methods.register_function(
...     function=my_method,
...     inputs={
...         'ints': IntSequence1,
...         'optional1': IntSequence1,
...         'optional2': IntSequence1 | IntSequence2
...     },
...     parameters={
...         'num1': Int,
...         'num2': Int
...     },
...     outputs={
...         'output': IntSequence1
...     },
...     name='My cool method',
...     description='It does something very clever and interesting.'
... )
```

class qiime2.plugin.plugin.**PluginVisualizers**(*plugin*)

Accessed via `plugin.visualizers`

register_function(*function*, *inputs*, *parameters*, *name*, *description*, *input_descriptions=None*, *parameter_descriptions=None*, *citations=None*, *deprecated=False*, *examples=None*)

Register a visualizer to the associated plugin.

Parameters

- **function** (*callable*) – A function which will be called when the user uses this method. This function receives an `output_dir` as the first argument.
- **inputs** (*dict[str, type expression]*) – A dictionary of function-parameter names to semantic type expressions. The keys of the dictionary must match the parameter names which are on the `function`. Collections and type variables are also permitted so long as they contain semantic types.
- **parameters** (*dict[str, type expression]*) – A dictionary of function parameter names to primitive type expressions. The keys of the dictionary must match the parameter names which are on the `function`. Collections and type variables are also permitted so long as they contain primitive types.
- **name** (*str*) – A short description, ideally a human-oriented name or title.
- **description** (*str*) – A longer description of the action.
- **input_descriptions** (*dict[str, str]*) – A dictionary of input names (see `inputs`) to descriptions.
- **parameter_descriptions** (*dict[str, str]*) – A dictionary of parameter names (see `parameters`) to descriptions.
- **citations** (`CitationRecord` *or list of* `CitationRecord`) – Citation(s) to associate with a result whenever this action is used. Can also use an entire *Citations* object.
- **deprecated** (*bool*) – Whether this action is deprecated and should be migrated away from.

- **examples** (*dict[str, callable]*) – A dict of example name to usage example functions which take a single argument (the usage driver, a.k.a. `use`). Each function will carry out some example situation for this action when executed.

Notes

Unlike methods and pipelines, there are no registered outputs as every visualizer returns a single `Visualization` output.

Examples

```
>>> plugin.visualizers.register_function(
...     function=my_visualizer,
...     inputs={'ints': IntSequence1 | IntSequence2},
...     parameters={},
...     name='Visualize most common integers',
...     description='Produces a ranked list of integers.',
...     citations=[citations['witcombe2006sword']]
... )
```

class qiime2.plugin.plugin.**PluginPipelines**(*plugin*)

Accessed via `plugin.pipelines`

register_function(*function*, *inputs*, *parameters*, *outputs*, *name*, *description*, *input_descriptions=None*, *parameter_descriptions=None*, *output_descriptions=None*, *citations=None*, *deprecated=False*, *examples=None*)

Register a pipeline to the associated plugin.

Parameters

- **function** (*callable*) – A function which will be called when the user uses this method. This function receives a Context object `ctx` as its first argument.
- **inputs** (*dict[str, type expression]*) – A dictionary of function-parameter names to semantic type expressions. The keys of the dictionary must match the parameter names which are on the `function`. Collections and type variables are also permitted so long as they contain semantic types.
- **parameters** (*dict[str, type expression]*) – A dictionary of function parameter names to primitive type expressions. The keys of the dictionary must match the parameter names which are on the `function`. Collections and type variables are also permitted so long as they contain primitive types.
- **outputs** (*dict[str, type expression]*) – A dictionary of named outputs to *concrete* semantic types. The keys of the dictionary will become the parameter names for these outputs and must match the number of annotated outputs on the `function`. Collections and type variables are also permitted so long as they contain semantic types. (In older versions of QIIME 2 this was a list of tuples as dictionaries were not yet ordered.)
- **name** (*str*) – A short description, ideally a human-oriented name or title.
- **description** (*str*) – A longer description of the action.
- **input_descriptions** (*dict[str, str]*) – A dictionary of input names (see `inputs`) to descriptions.

- **parameter_descriptions** (*dict[str, str]*) – A dictionary of parameter names (see `parameters`) to descriptions.
- **output_descriptions** (*dict[str, str]*) – A dictionary of output names (see `outputs`) to descriptions.
- **citations** (*CitationRecord or list of CitationRecord*) – Citation(s) to associate with a result whenever this action is used. Can also use an entire *Citations* object.
- **deprecated** (*bool*) – Whether this action is deprecated and should be migrated away from.
- **examples** (*dict[str, callable]*) – A dict of example name to usage example functions which take a single argument (the usage driver, a.k.a. `use`). Each function will carry out some example situation for this action when executed.

Examples

```
>>> from qiime2.plugin import Collection
>>> plugin.pipelines.register_function(
...     function=my_pipeline,
...     inputs={'ints': Collection[IntSequence1]},
...     parameters={},
...     outputs={'output': Collection[IntSequence1]},
...     name='Do thing multiple times',
...     description='Takes a collection and returns a better one'
... )
```

Formats

class qiime2.plugin.**TextFileFormat**(*path=None*, *mode='w'*)

class qiime2.plugin.**BinaryFileFormat**(*path=None*, *mode='w'*)

class qiime2.plugin.**DirectoryFormat**(*path=None*, *mode='w'*)

qiime2.plugin.**SingleFileDirectoryFormat**(*name*, *pathspec*, *format*)

class qiime2.plugin.**ValidationError**

Types

Semantic Type

qiime2.plugin.**SemanticType**(*name*, *field_names=None*, *field_members=None*, *variant_of=None*)

Create a new semantic type.

Parameters

- **name** (*str*) – The name of the semantic type: this should match the variable to which the semantic type is assigned.

- **field_names** (*str, iterable of str, optional*) – Name(s) of the fields where member types can be placed. This makes the type a composite type, meaning that fields must be provided to produce realized semantic types. These names will define ad-hoc variant types accessible as *name*.field[*field_names* member].
- **field_members** (*mapping, optional*) – A mapping of strings in *field_names* to one or more semantic types which are known to be members of the field (the variant type).
- **variant_of** (*VariantField, iterable of VariantField, optional*) – Define the semantic type to be a member of one or more variant types allowing it to be placed in the respective fields defined by those variant types.

Returns

There are several (private) types which may be returned, but anything returned by this factory will cause *is_semantic_type* to return True.

Return type

A Semantic Type

Predicates

class qiime2.plugin.**Properties**(**include*, *exclude=()*)

A one or more semantic properties to add to an existing type.

Semantic properties make an existing semantic type "smaller" than it would otherwise be. If a union causes types to become larger, then a property is the conceptual opposite. Either can be used to the same effect, it just depends on what the most natural "starting point" is for the base types (i.e. are the base types small and then union-ed, or are they broad and then subsequently narrowed as needed?)

Parameters

- ***include** (*str*) – Properties that are true. A property is only a subtype of another property if they match. This means that a subtype (with a set of properties `X`) of an expression (with properties `Y`) must have properties such that `X >= Y`. As a consequence, each additional property narrows the type further.
- **exclude** (*tuple[str]*) – Treated as an inverse of `include`. The absence of a property is not sufficient for an excluded property to match. Excluded properties must be explicitly defined on the type. As a consequence, this field is rarely if ever used. It may be deprecated in the future.

Examples

```
>>> from qiime2.plugin import Properties
```

Properties must match:

```
>>> Properties('a') <= Properties('a')
True
```

An empty property is larger than a defined one:

```
>>> # semantic type expressions have implicit empty properties so
>>> # there is no need to ever provide an empty property directly
>>> Properties('a') <= Properties()
True
```

The more properties there are, the narrower the type is:

```
>>> Properties('a', 'b', 'c') <= Properties('a', 'b')
True
>>> Properties('a', 'b', 'c') <= Properties('a', 'c')
True
>>> Properties('a', 'b', 'c') <= Properties('b', 'c')
True
>>> Properties('a', 'b', 'c') <= Properties('a')
True
```

Order does not matter:

```
>>> Properties('a', 'b') <= Properties('a', 'b')
True
>>> Properties('a', 'b') <= Properties('b', 'a')
True
```

Visualization type

qiime2.plugin.**Visualization**

The type of a QIIME 2 Visualization.

This is not a semantic type as it represents a terminal/non-composable output.

An output with this type provides no assurances about the structure of the data as it is meant for human interpretation.

Examples

```
>>> from qiime2.plugin import Visualization
>>> Visualization
Visualization
```

Primitive types

These are types that all QIIME 2 interfaces will recognize and generate user affordances for.

Basic types

qiime2.plugin.**Bool**

A boolean value (`True`/`False`). It can use the predicate *Choices* (but this is only interesting when using *TypeMap*)

Examples

```
>>> from qiime2.plugin import Bool, Choices
```

Normal values:

```
>>> True in Bool
True
>>> False in Bool
True
```

Constrained (for *TypeMap*):

```
>>> False in Bool % Choices(False)
True
>>> True in Bool % Choices(False)
False
```

qiime2.plugin.**Str**

A string of Unicode characters (i.e. text). It can use the predicate *Choices* to create strict enumeration.

Examples

```
>>> from qiime2.plugin import Str, Choices
```

Arbitrary string:

```
>>> "Hello World" in Str
True
```

Enumeration of options:

```
>>> "apple" in Str % Choices("apple", "orange", "banana")
True
>>> "airplane" in Str % Choices("apple", "orange", "banana")
False
```

Warning

Do not use *Str* for filepaths. Not all interfaces have a consistent (or user-navigable) representation of a filesystem. Data should be represented as an artifact which will have a *SemanticType()* and allows the interface to store (and manipulate) your data as it sees fit.

qiime2.plugin.**Int**

An integer without any particular bounds. It can use the predicates *Range*, *Start()*, and *End()*

Examples

```
>>> from qiime2.plugin import Int, Range, Start, End
```

No bounds on the value:

```
>>> -2 in Int
True
```

Integers between 0 (inclusive) and 5 (exclusive):

```
>>> 0 in Int % Range(0, 5)
True
>>> 5 in Int % Range(0, 5)
False
```

Same as above:

```
>>> 0 in Int % (Start(0) & End(5))
True
>>> 5 in Int % (Start(0) & End(5))
False
```

qiime2.plugin.**Float**

A 64 bit floating point number. It can use the predicates *Range*, *Start()*, and *End()*

Examples

```
>>> from qiime2.plugin import Float, Range, Start, End
```

No bounds on the value:

```
>>> -0.2 in Float
True
```

Proportion between 0 (exclusive) and 1 (inclusive):

```
>>> 0.0 in Float % Range(0, 1, inclusive_start=False, inclusive_end=True)
False
>>> 1.0 in Float % Range(0, 1, inclusive_start=False, inclusive_end=True)
True
```

Same as above:

```
>>> 0.0 in Float % (Start(0, inclusive=False) & End(1, inclusive=True))
False
>>> 1.0 in Float % (Start(0, inclusive=False) & End(1, inclusive=True))
True
```

qiime2.plugin.**Threads**

The number of logical threads to use (OS threads/CPUs/Cores).

Valid inputs are an integer that is non-negative or the string `"auto"`. `0` and `"auto"` will indicate that the number of logical threads should be dictated by system resources.

It does not support any predicate expressions.

Examples

```
>>> from qiime2.plugin import Threads
```

Positive integer for the number of logical threads to use:

```
>>> 12 in Threads
True
```

Zero/auto to let the system decide:

```
>>> 0 in Threads
True
>>> "auto" in Threads
True
```

qiime2.plugin.**Jobs**

The number of jobs to submit as an integer that is greater than zero (exclusive).

It does not support any predicate expressions.

Examples

```
>>> from qiime2.plugin import Jobs
```

Positive integer for the number of jobs to use:

```
>>> 20 in Jobs
True
```

Zero is not a valid value:

```
>>> 0 in Jobs
False
```

Predicates

class qiime2.plugin.**Choices**(**choices*)

A predicate which defines a set of allowable values.

Can be used with *Str* and *Bool*.

Parameters

***choices** (*Any*) – The legal values for the base type to use. Any value outside of this enumeration will be considered outside of the domain of the base type.

Examples

```
>>> from qiime2.plugin import Str, Choices
>>> "apple" in Str % Choices("apple", "orange", "banana")
True
>>> "airplane" in Str % Choices("apple", "orange", "banana")
False
```

class qiime2.plugin.**Range**(*[start*, *]end*, *inclusive_start=True*, *inclusive_end=False*)

A predicate which defines a contiguous range of allowable values.

Can be used with *Int* and *Float*.

Parameters

- **[start]** (*number*) – When provided as the first argument, the value will be the start of the range. Will be None (meaning negative infinity) when not provided.
- **end** (*number*) – The end of the range (when provided as the first or second argument). Will be None (meaning positive infinity) when not provided.
- **inclusive_start** (*bool*) – If the start is a part of the range.
- **inclusive_end** (*bool*) – If the end is a part of the range.

Examples

```
>>> from qiime2.plugin import Float, Range
```

A simple proportion without 100%:

```
>>> 0.0 in Float % Range(0, 1)
True
>>> 0.999 in Float % Range(0, 1)
True
>>> 1.0 in Float % Range(0, 1)
False
```

A non-negative value:

```
>>> -1.0 in Float % Range(0, None)
False
>>> 0.0 in Float % Range(0, None)
True
>>> 1.0 in Float % Range(0, None)
True
```

Multiple discontinuous ranges:

```
>>> 0.0 in Float % (Range(0.1, 1, inclusive_end=True)
...                 | Range(10, 100, inclusive_end=True))
False
>>> 3.4 in Float % (Range(0.1, 1, inclusive_end=True)
...                 | Range(10, 100, inclusive_end=True))
False
>>> 1.0 in Float % (Range(0.1, 1, inclusive_end=True)
...                 | Range(10, 100, inclusive_end=True))
```

(continues on next page)

(continued from previous page)

```
True
>>> 100.0 in Float % (Range(0.1, 1, inclusive_end=True)
...                   | Range(10, 100, inclusive_end=True))
True
```

Intersecting ranges (how *Start()* and *End()* work):

```
>>> Range(0, None) & Range(None, 10)
Range(0, 10)
```

See also

Start, *End*

qiime2.plugin.**Start**(*start*, *inclusive=True*)

Shorthand to generate a *Range*

The end of the resulting range is `None` (infinity).

Parameters

- **start** (*number*) – The start of the range
- **inclusive** (*bool*) – Whether the start is a part of the range

Examples

```
>>> from qiime2.plugin import Start
>>> Start(0)
Range(0, None)
```

See also

Range

qiime2.plugin.**End**(*end*, *inclusive=False*)

Shorthand to generate a *Range*

The start of the resulting range is `None` (negative infinity).

Parameters

- **end** (*number*) – The end of the range
- **inclusive** (*bool*) – Whether the end is a part of the range

Examples

```
>>> from qiime2.plugin import End
>>> End(100)
Range(None, 100)
```

See also

Range

Metadata

These primitive types represent tabular metadata, where unique identifiers can be associated with columns. Typically these are used to represent per-sample or per-feature metadata. But there is nothing special about those axes.

qiime2.plugin.**Metadata**

Tabular metadata where unique identifiers can be associated with columns.

This is the type that represents *qiime2.Metadata*.

Examples

```
>>> import pandas as pd
>>> import qiime2
>>> from qiime2.plugin import Metadata
```

Note the distinct module paths:

```
>>> md = qiime2.Metadata(pd.DataFrame([{'num':1, 'cat': 'a'}],
...                                   index=pd.Series(['s1'], name='id')))
>>> md in Metadata
True
```

qiime2.plugin.**MetadataColumn = MetadataColumn[Categorical | Numeric]**

A column of a *qiime2.Metadata*.

Has two variants: *Categorical* and *Numeric*.

Examples

```
>>> import pandas as pd
>>> import qiime2
>>> from qiime2.plugin import Metadata, Categorical, Numeric
>>> md = qiime2.Metadata(pd.DataFrame([{'num':1, 'cat': 'a'}],
...                                   index=pd.Series(['s1'], name='id')))
>>> md
Metadata
--------
1 ID x 2 columns
num: ColumnProperties(type='numeric', missing_scheme='blank')
cat: ColumnProperties(type='categorical', missing_scheme='blank')
...
```

Categorical column:

```
>>> md.get_column('cat') in MetadataColumn[Categorical]
True
>>> md.get_column('num') in MetadataColumn[Categorical]
False
```

Numeric column:

```
>>> md.get_column('num') in MetadataColumn[Numeric]
True
>>> md.get_column('cat') in MetadataColumn[Numeric]
False
```

Any column:

```
>>> md.get_column('cat') in MetadataColumn[Categorical | Numeric]
True
>>> md.get_column('num') in MetadataColumn[Categorical | Numeric]
True
```

qiime2.plugin.**Categorical**

The categorical variant for *MetadataColumn*.

Has no meaning unless used within *MetadataColumn*.

qiime2.plugin.**Numeric**

The Numeric variant for *MetadataColumn*.

Has no meaning unless used within *MetadataColumn*.

Collections

Collections may be used with Semantic, Visualization, and basic Primitive types.

qiime2.plugin.**Set = Set[{type}]**

Deprecated - use List or Collection instead

A set of unique elements without a defined order except when used with a semantic type, in which case, the views are provided to the plugin as a list.

qiime2.plugin.**List = List[{type}]**

An alias for a *Collection* with numeric auto-incrementing keys.

Examples

```
>>> from qiime2.plugin import List, Str
```

A regular list:

```
>>> ['a', 'b', 'c'] in List[Str]
True
```

A collection is also compatible:

```
>>> {'arbitrary_key': 'a'} in List[Str]
True
```

qiime2.plugin.**Collection = Collection[{type}]**

An ordered set of key-value pairs. Compatible with lists and dictionaries.

The keys of a collection are always strings, and the values are defined by the variant provided to the type-field.

Whenever a list is provided, it will be coerced into a dictionary with auto-incrementing integer keys.

Examples

```
>>> from qiime2.plugin import Collection, Str
```

A regular dictionary:

```
>>> {'key1': 'a', 'key2': 'b'} in Collection[Str]
True
```

A list:

```
>>> ['a', 'b'] in Collection[Str]
True
```

Dependent Types

class qiime2.plugin.**TypeMap**(*mapping*)

A table of input types which match to output types.

The TypeMap is best thought of as a table in which QIIME 2 is trying to find a row that matches the user's input. Once found, the row-wise search is terminated and the outputs of that row are bound to the outputs of the action.

So if a TypeMap looked like this:

```
T_paramA, T_paramB, T_out = TypeMap({
    (Bool % Choices(True) , InputTypeA): ResultTypeA
    (Bool % Choices(False), InputTypeA): ResultTypeB
    (Bool % Choices(False), InputTypeB): ResultTypeB
})
```

It could be thought of as this table:

Parameter A	Parameter B	Result
True	InputTypeA	ResultTypeA
False	InputTypeA	ResultTypeB
False	InputTypeB	ResultTypeB

Where if the user provides `True` to Parameter A, they MUST provide `InputTypeA` to Parameter B, and will receive `ResultTypeA`. Otherwise, they may pass `False` to Parameter A, and provide either `InputTypeA` or `InputTypeB`, but will now receive `ResultTypeB`.

Note that when `Parameter B` is given `InputTypeB`, `Parameter A` must be `False` as there is no row in which `True` is simultaneously possible. (Or equivalently, if `True` is given for `Parameter A`, then `Parameter B` must be fixed to `InputTypeA`.)

This can be used to constrain dependent input parameters to a more limited domain than they would otherwise possess if they were treated independently.

If a TypeMap is used exclusively to constrain inputs but does not impact the output in any way, then the convention is to use *Visualization* to indicate a "nonsense" output and that final type variable is ignored (an unbound output variable has no effect so `Visualization` distinguishes the intention from an accidental omission).

It is also possible to define multiple outputs which are dependent on inputs, so long as the value of the dictionary is a tuple. This will result in additional type variables to be used in the output registration.

Parameters

mapping (*dict[tuple[type expressions], tuple[type expressions]]*) – A tuple is not strictly required, so long as there are input and outputs which are enforced by the syntax of a dictionary. In the event a given input tuple's domain overlaps another input tuple, the overlap must be a subset and the smaller branch must come first. Otherwise, the output resolution would be ambiguous (this rule is enforced when the TypeMap is constructed).

Returns

The type variables should be unpacked from the TypeMap and the number will correspond to the number of "columns" in the TypeMap.

Return type

iterable of TypeVarExp

class qiime2.plugin.**TypeMatch**(*listing*)

A trivial *TypeMap* such that every entry maps to itself.

A TypeMatch which looked like this:

```
T = TypeMatch([Foo, Bar, Baz])
```

Is essentially the same as:

```
T_in, T_out = TypeMap({
    Foo: Foo,
    Bar: Bar,
    Baz: Baz
})
```

Except that `T` doubles as both `T_in` and `T_out`.

Parameters

listing (*list[type fragments]*) – A list of type fragments (usually variants). The behavior is similar to a union, but will cause the output type to be the same as the input type.

Returns

A type variable that can be used as a plugin's input **and** output. The output type will then be the same as the input type.

Return type

TypeVarExp

Examples

```
>>> from qiime2.plugin import TypeMatch
>>> from qiime2.core.testing.type import Foo, Bar, Baz, C1
>>> T = TypeMatch([Foo, Bar, Baz])
>>> C1[Foo] <= C1[T]
True
>>> C1[Bar] <= C1[T]
True
>>> C1[Baz] <= C1[T]
True
```

See also

TypeMap

Citations

QIIME 2 can automatically track citations as a user performs an analysis. This is done via *CitationRecord* and lists of them (`list` and *Citations*).

class qiime2.plugin.**Citations**

A simple subclass of `collections.OrderedDict` but iterates over values instead of keys by default.

classmethod load(*path*, *package=None*)

Load a bibtex file from a path (or relative package path)

Parameters

- **path** (*str | PathLike*) – The path of a bibtex file, if *package* is provided, it will be relative to the python package.
- **package** (*str | None*) – The python package to load from.

Return type
Citations

__iter__()

Iterates over the contained *CitationRecord*'s

save(*f*)

Save object as bibtex to a filepath or filehandle

Parameters
f (*str | IO*) – A string (but not `os.PathLike`) or filehandle

Return type
None

class qiime2.plugin.**CitationRecord**(*type*, *fields*)

A `collections.namedtuple()` of bibtex entry type and entry fields.

Parameters

- **type** (*str*) – The bibtex entry type (e.g. 'article')
- **fields** (*dict*) – The individual key-value pairs of the bibtex entry

Testing

class qiime2.plugin.testing.**TestPluginBase**(*methodName='runTest'*)

Test harness for simplifying testing QIIME 2 plugins.

`TestPluginBase` extends `unittest.TestCase`, with a few extra helpers and assertions.

package

The name of the plugin package to be tested. Subclasses **must** define this as a class-attribute.

Type
str

test_dir_prefix

The prefix for the temporary testing dir created by the harness. Optional.

Type
str

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

setUp()

Test runner setup hook.

If overriding this hook in a test, call `__super__` to invoke this method in the overridden hook, otherwise the harness might not work as expected.

tearDown()

Test runner teardown hook.

If overriding this hook in a test, call `__super__` to invoke this method in the overridden hook, otherwise the harness might not work as expected.

get_data_path(*filename*, *result_as_str=True*)

Convenience method for getting a data asset while testing.

Test data stored in the `data/` dir local to the running test can be accessed via this method.

Parameters

- **filename** (*str*) – The name of the file to look up.
- **result_as_str** (*bool, True*) – To retain the original API, by default the result is returned as a string. Setting this to False will return the result as a pathlib.FilePath object.

Returns
filepath – The materialized filepath to the requested test data.

Return type
str (or pathlib.Path if result_as_str != True)

get_transformer(*from_type*, *to_type*)

Convenience method for getting a registered transformer.

This helper deliberately side-steps the framework's validation machinery, so that it is possible for plugin developers to test failing conditions.

Parameters

- **from_type** (*A View Type*) – The *View* type of the source data.
- **to_type** (*A View Type*) – The *View* type to transform to.

Returns
transformer – The registered tranformer from `from_type` to `to_type`.

Return type
A Transformer Function

assertRegisteredSemanticType (*semantic_type*)

Test assertion for ensuring a plugin's semantic type is registered.

Fails if the semantic type requested is not found in the Plugin Manager.

Parameters
semantic_type (*A Semantic Type*) – The *Semantic Type* to test the presence of.

assertSemanticTypeRegisteredToFormat (*semantic_type*, *exp_format*)

Test assertion for ensuring a semantic type is registered to a format.

Fails if the semantic type requested is not registered to the format specified with `exp_format`. Also fails if the semantic type isn't registered to **any** format.

Parameters

- **semantic_type** (*A Semantic Type*) – The *Semantic Type* to check for.
- **exp_format** (*A Format*) – The *Format* to check that the Semantic Type is registed on.

transform_format (*source_format*, *target*, *filename=None*, *filenames=None*)

Helper utility for loading data and transforming it.

Combines several other utilities in this class, will load files from `data/`, as `source_format`, then transform to the `target` view.

This helper deliberately side-steps the framework's validation machinery, so that it is possible for plugin developers to test failing conditions.

Parameters

- **source_format** (*A Format*) – The *Format* to load the data as.
- **target** (*A View Type*) – The *View Type* to transform the data to.
- **filename** (*str*) – The name of the file to load from `data`. Use this for formats that use a single file in their format definition. Mutually exclusive with the `filenames` parameter.
- **filenames** (*list[str]*) – The names of the files to load from `data`. Use this for formats that use multiple files in their format definition. Mutually exclusive with the `filename` parameter.

Returns

- **input** (*A Format*) – The data loaded from `data` as the specified `source_format`.
- **obs** (*A View Type*) – The loaded data, transformed to the specified `target` view type.

execute_examples ()

Runs all usage examples defined in the plugin.

qiime2.plugin.testing.**assert_no_nans_in_tables** (*fh*)

Checks for NaNs present in any of the tables in the indicated file then resets to the head of the file.

Utilities

General Utils

qiime2.util.**duplicate**(*src*, *dst*)

Alternative to `shutil.copyfile()` this will use `os.link()` when possible.

See `shutil.copyfile()` for documention. Only `src` and `dst` are supported. Unlike copyfile, this will not overwrite the destination if it exists.

qiime2.util.**redirected_stdio**(*stdout=None*, *stderr=None*)

A context manager to redirect stdio to a new file (if provided).

Parameters

- **stdout** (*file-like object or file-descriptor (int)*) – The file to redirect stdout to. Does nothing to the process-table if not provided.
- **stderr** (*file-like object or file-descriptor (int)*) – The file to redirect stderr to. Does nothing to the process-table if not provided.

Notes

The current sys.stdout/sys.stderr must be backed by a file descriptor (i.e they cannot be replaced with BytesIO).

Plugin Utils

qiime2.plugin.util.**transform**(*data*, *, *from_type=None*, *to_type*)

qiime2.plugin.util.**get_available_cores**(*n_less=0*)

Finds the number of currently available (logical) cores. Useful for plugins that need to convert a 0 to a concrete number of cores when 0 is not supported by the underlying/called software.

Parameters

n_less (*int*) – The number of cores less than the total number available to request. For example *get_available_cores(n_less=2)* with 10 available cores will return 8.

Returns

The number of cores to be requested.

Return type

int

Pipeline Context Object

The context object is available to pipelines as a required first argument `ctx`. Plugins may use these methods to invoke other registered actions or create artifacts.

Context.**get_action**(*plugin*, *action*)

Return a function matching the callable API of an action. This function is aware of the pipeline context and manages its own cleanup as appropriate.

Context.**make_artifact**(*type*, *view*, *view_type=None*)

Return a new artifact from a given view.

This artifact is automatically tracked and cleaned by the pipeline context.

Usage Examples

This page outlines elements of the Usage API which are used by example authors (and overriden by interface drivers) to describe some example situation in the framework for documentation, testing, or interface generating purposes.

Initializers

These methods prepare some data for use in an example.

Usage.**init_artifact**(*name*, *factory*)

Communicate that an artifact will be needed.

Driver implementations may use this to intialize data for an example.

Parameters

- **name** (*str*) – The canonical name of the variable to be returned.
- **factory** (Callable which returns *qiime2.sdk.Artifact*) – A function which takes no parameters, and returns an artifact. This function may do anything internally to create the artifact.

Returns

This particular return class can be changed by a driver which overrides `usage_variable()`.

Return type

UsageVariable

Examples

```
>>> # A factory which will be used in the example to generate data.
>>> def factory():
...     import qiime2
...     # This type is only available during testing.
...     # A real example would use a real type.
...     a = qiime2.Artifact.import_data('IntSequence1', [1, 2, 3])
...     return a
...
>>> my_artifact = use.init_artifact('my_artifact', factory)
>>> my_artifact
<ExecutionUsageVariable name='my_artifact', var_type='artifact'>
```

Usage.**init_artifact_from_url**(*name*, *url*)

Obtain an artifact from a url.

Driver implementations may use this to intialize data for an example.

Parameters

- **name** (*str*) – The canonical name of the variable to be returned.

- **url** (*str*) – The url of the Artifact that should be downloaded for the example. If a QIIME 2 epoch (e.g., 2022.11) is part of the URL, as might be the case if obtaining an Artifact from docs.qiime2.org, it can be templated in by including *{qiime2.__release__}* in an F-string defining the URL.

Returns
This particular return class can be changed by a driver which overrides `usage_variable()`.

Return type
UsageVariable

`Usage.init_artifact_collection`(*name*, *factory*)

Communicate that a result collection containing artifacts will be needed.

Driver implementations may use this to intialize data for an example.

Parameters

- **name** (*str*) – The canonical name of the variable to be returned.
- **factory** (Callable which returns *qiime2.sdk.ResultCollection*) – A function which takes no parameters, and returns a result collection that contains artifacts. This function may do anything internally to create the result collection.

Returns
This particular return class can be changed by a driver which overrides `usage_variable()`.

Return type
UsageVariable

Examples

```
>>> # A factory which will be used in the example to generate data.
>>> def factory():
...     import qiime2
...     # This type is only available during testing.
...     # A real example would use a real type.
...     a = qiime2.ResultCollection(
...         {'Foo': qiime2.Artifact.import_data('IntSequence1', [1, 2, 3]),
...          'Bar': qiime2.Artifact.import_data('IntSequence1', [4, 5, 6])})
...     return a
...
>>> int_seq_collection = use.init_artifact_collection('int_seq_collection',␣
↪factory)
>>> int_seq_collection
<ExecutionUsageVariable name='int_seq_collection', var_type='artifact_collection'>
```

`Usage.init_metadata`(*name*, *factory*)

Communicate that metadata will be needed.

Driver implementations may use this to intialize data for an example.

Parameters

- **name** (*str*) – The canonical name of the variable to be returned.
- **factory** (Callable which returns *qiime2.Metadata*) – A function which takes no parameters, and returns metadata. This function may do anything internally to create the metadata.

Returns
Variable of type 'metadata'.

Return type
UsageVariable

Examples

```
>>> # A factory which will be used in the example to generate data.
>>> def factory():
...     import qiime2
...     import pandas as pd
...     df = pd.DataFrame({'a':[1, 2, 3]}, index=['a', 'b', 'c'])
...     df.index.name = 'id'
...     md = qiime2.Metadata(df)
...     return md
...
>>> my_metadata = use.init_metadata('my_metadata', factory)
>>> my_metadata
<ExecutionUsageVariable name='my_metadata', var_type='metadata'>
```

`Usage.`**`init_metadata_from_url`**(*name*, *url*)

Obtain metadata from a url.

Driver implementations may use this to intialize example metadata.

Parameters

- **`name`** (*str*) – The canonical name of the variable to be returned.
- **`url`** (*str*) – The url of the Artifact that should be downloaded for the example. If a QIIME 2 epoch (e.g., 2022.11) is part of the URL, as might be the case if obtaining an Artifact from docs.qiime2.org, it can be templated in by including *{qiime2.__release__}* in an F-string defining the URL.

Returns
This particular return class can be changed by a driver which overrides `usage_variable()`.

Return type
UsageVariable

Examples

```
>>> import qiime2
>>> url = ('https://data.qiime2.org/usage-examples/moving-pictures/'
...        'sample-metadata.tsv')
>>> print(url)
https://data.qiime2.org/usage...
>>> md = use.init_metadata_from_url('md', url)
>>> md
<ExecutionUsageVariable name='md', var_type='metadata'>
```

`Usage.`**`init_format`**(*name*, *factory*, *ext=None*)

Communicate that a file/directory format will be needed.

Driver implementations may use this to intialize data for an example.

Parameters

- **name** (*str*) – The canonical name of the variable to be returned.
- **factory** (*Callable which returns a file or directory format.*) – A function which takes no parameters, and returns a format. This function may do anything internally to create the format.
- **ext** (*str*) – The extension to prefer if the format is preserved on disk.

Returns
Variable of type 'format'.

Return type
UsageVariable

Examples

```
>>> # A factory which will be used in the example to generate data.
>>> def factory():
...     from qiime2.core.testing.format import IntSequenceFormat
...     from qiime2.plugin.util import transform
...     ff = transform([1, 2, 3], to_type=IntSequenceFormat)
...
...     ff.validate()  # good practice
...     return ff
...
>>> my_ints = use.init_format('my_ints', factory, ext='.hello')
>>> my_ints
<ExecutionUsageVariable name='my_ints', var_type='format'>
```

Importing

These methods demonstrate how to import an artifact.

`Usage.`**`import_from_format`**(*name*, *semantic_type*, *variable*, *view_type=None*)

Communicate that an import should be done.

Parameters

- **name** (*str*) – The name of the resulting variable.
- **semantic_type** (*str*) – The semantic type to import as.
- **variable** (*UsageVariable*) – A variable of type 'format' which possesses a factory to materialize the actual data to be imported.
- **view_type** (*format or str*) – The view type to import as, in the event it is different from the default.

Returns
Variable of type 'artifact'.

Return type
UsageVariable

Examples

```
>>> # A factory which will be used in the example to generate data.
>>> def factory():
...     from qiime2.core.testing.format import IntSequenceFormat
...     from qiime2.plugin.util import transform
...     ff = transform([1, 2, 3], to_type=IntSequenceFormat)
...
...     ff.validate()  # good practice
...     return ff
...
>>> to_import = use.init_format('to_import', factory, ext='.hello')
>>> to_import
<ExecutionUsageVariable name='to_import', var_type='format'>
>>> ints = use.import_from_format('ints',
...                               semantic_type='IntSequence1',
...                               variable=to_import,
...                               view_type='IntSequenceFormat')
>>> ints
<ExecutionUsageVariable name='ints', var_type='artifact'>
```

See also

init_format

Collections

These methods demonstrate how to manipulate collections.

`Usage.`**`construct_artifact_collection`**(*name*, *members*)

Return a UsageVariable of type artifact_collection given a list or dict of its members.

Parameters

- **`name`** (*`str`*) – The name of the resulting variable.
- **`members`** (*`list or dict`*) – The desired members of the ResultCollection.

Returns

Of type artifact_collection.

Return type

UsageVariable

Examples

```
>>> mapping_1, = use.action(
...     use.UsageAction('dummy_plugin', 'params_only_method'),
...     use.UsageInputs(name='c', age=100),
...     use.UsageOutputNames(out='mapping_1')
... )
>>> mapping_1
<ExecutionUsageVariable name='mapping_1', var_type='artifact'>
>>> collection_1 = use.construct_artifact_collection(
```

(continues on next page)

(continued from previous page)

```
...     'collection_1', {'a': mapping_1, 'b': mapping_1}
... )
>>> collection_1
<ExecutionUsageVariable name='collection_1', var_type='artifact_collection'>
```

Usage.**get_artifact_collection_member**(*name*, *variable*, *key*)

Accesses and returns a member of a ResultCollection as a UsageVariable.

Parameters

- **name** (*str*) – The name of the resulting variable.
- **variable** (UsageVariable) – The UsageVariable of type artifact_collection from which to access the desired member.
- **key** (*str*) – The key of the desired member in the ResultCollection.

Returns

Of type artifact.

Return type

UsageVariable

Examples

```
>>> mapping_2, = use.action(
...     use.UsageAction('dummy_plugin', 'params_only_method'),
...     use.UsageInputs(name='c', age=100),
...     use.UsageOutputNames(out='mapping_2')
... )
>>> mapping_2
<ExecutionUsageVariable name='mapping_2', var_type='artifact'>
>>> collection_2 = use.construct_artifact_collection(
...     'collection_2', {'a': mapping_2, 'b': mapping_2}
... )
>>> collection_2
<ExecutionUsageVariable name='collection_2', var_type='artifact_collection'>
>>> first_member = use.get_artifact_collection_member(
...     'first_member', collection_2, 'a'
... )
>>> first_member
<ExecutionUsageVariable name='first_member', var_type='artifact'>
```

Metadata

These methods demonstrate how to manipulate metadata.

Usage.**get_metadata_column**(*name*, *column_name*, *variable*)

Communicate that a column should be retrieved.

Parameters

- **name** (*str*) – The name of the resulting variable.
- **column_name** (*str*) – The column to retrieve.

- **variable** (UsageVariable) – The metadata to retrieve the column from. Must be a variable of type 'metadata'.

Returns
Variable of type 'column'.

Return type
UsageVariable

Raises
AssertionError – If the variable is not of type 'metadata'.

Examples

```
>>> def factory():
...     import qiime2
...     import pandas as pd
...     df = pd.DataFrame({'column_a':[1, 2, 3]},
...                       index=['a', 'b', 'c'])
...     df.index.name = 'id'
...     return qiime2.Metadata(df)
...
>>> md_for_column = use.init_metadata('md_for_column', factory)
>>> md_for_column
<ExecutionUsageVariable name='md_for_column', var_type='metadata'>
>>> my_column = use.get_metadata_column('my_column', 'column_a',
...                                     md_for_column)
>>> my_column
<ExecutionUsageVariable name='my_column', var_type='column'>
```

See also

init_metadata

Usage.**view_as_metadata**(*name*, *variable*)

Communicate that an artifact should be views as metadata.

Parameters

- **name** (*str*) – The name of the resulting variable.
- **variable** (UsageVariable) – The artifact to convert to metadata. Must be a variable of type 'artifact'.

Returns
Variable of type 'metadata'.

Return type
UsageVariable

Raises
AssertionError – If the variable is not of type 'artifact'.

Examples

```
>>> artifact_for_md, = use.action(
...     use.UsageAction('dummy_plugin', 'params_only_method'),
...     use.UsageInputs(name='c', age=100),
...     use.UsageOutputNames(out='artifact_for_md'))
>>> artifact_for_md
<ExecutionUsageVariable name='artifact_for_md', var_type='artifact'>
>>> metadata = use.view_as_metadata('metadata', artifact_for_md)
>>> metadata
<ExecutionUsageVariable name='metadata', var_type='metadata'>
```

See also

init_artifact, *get_metadata_column*

Usage.**merge_metadata**(*name*, **variables*)

Communicate that these metadata should be merged.

Parameters

- **name** (*str*) – The name of the resulting variable.
- ***variables** (UsageVariable) – Multiple variables of type 'metadata' to merge.

Returns

Variable of type 'metadata'.

Return type

UsageVariable

Raises

AssertionError – If a variable is not of type 'metadata'.

Examples

```
>>> def factory1():
...     import qiime2
...     import pandas as pd
...     df = pd.DataFrame({'a':[0]}, index=['0'])
...     df.index.name = 'id'
...     md = qiime2.Metadata(df)
...     return md
...
>>> def factory2():
...     import qiime2
...     import pandas as pd
...     df = pd.DataFrame({'b':[10]}, index=['0'])
...     df.index.name = 'id'
...     md = qiime2.Metadata(df)
...     return md
...
>>> some_artifact, = use.action(
...     use.UsageAction('dummy_plugin', 'params_only_method'),
...     use.UsageInputs(name='c', age=100),
```

(continues on next page)

(continued from previous page)

```
...     use.UsageOutputNames(out='some_artifact'))
...
>>> md1 = use.init_metadata('md1', factory1)
>>> md2 = use.init_metadata('md2', factory2)
>>> md3 = use.view_as_metadata('md3', some_artifact)
>>> merged = use.merge_metadata('merged', md1, md2, md3)
>>> merged
<ExecutionUsageVariable name='merged', var_type='metadata'>
```

See also

init_metadata, *view_as_metadata*

Annotations

These methods do not return anything, but may be displayed in other ways.

`Usage.`**`comment`**(*text*)

Communicate that a comment should be made.

Default implementation is to do nothing.

Parameters

`text` (*str*) – The inspired commentary.

Examples

```
>>> use.comment("The thing is, they always try to walk it in...")
```

`Usage.`**`help`**(*action*)

Communicate that help text should be displayed.

Default implementation is to do nothing.

Parameters

`action` (UsageAction) – The particular action that should have help-text rendered.

Examples

```
>>> use.help(use.UsageAction('dummy_plugin', 'split_ints'))
```

`Usage.`**`peek`**(*variable*)

Communicate that an artifact should be peeked at.

Default implementation is to do nothing.

Parameters

`variable` (UsageVariable) – A variable of 'artifact' type which should be peeked.

Raises

`AssertionError` – If the variable is not of type 'artifact'.

Examples

```
>>> def factory():
...     import qiime2
...     return qiime2.Artifact.import_data('IntSequence1', [1, 2, 3])
...
>>> a_boo = use.init_artifact('a_boo', factory)
>>> use.peek(a_boo)
```

Actions

These methods invoke a plugin's action.

`Usage.`**`action`**(*action*, *inputs*, *outputs*)

Communicate that some action should be performed.

Parameters

- **`action`** (`UsageAction`) – The action to perform.
- **`inputs`** (`UsageInputs`) – The inputs to provide. These are a map of parameter names to arguments. Arguments may be primitive literals, or variables.
- **`outputs`** (`UsageOutputNames`) – Defines what to name each output variable. The keys much match the action's output signature.

Returns

A wrapper around the usual *`qiime2.sdk.Results`* object. Unpacking this output can be seen in the examples below.

Return type

UsageOutputs

Examples

```
>>> results = use.action(
...     use.UsageAction(plugin_id='dummy_plugin',
...                     action_id='params_only_method'),
...     use.UsageInputs(name='foo', age=42),
...     use.UsageOutputNames(out='bar')
... )
>>> results
UsageOutputs (name = value)
-----------------------------------------------------------
out = <ExecutionUsageVariable name='bar', var_type='artifact'>
```

```
>>> # "out" happens to be the name of this output, it isn't a general
>>> # name for all results.
>>> results.out
<ExecutionUsageVariable name='bar', var_type='artifact'>
```

```
>>> # unpack as an iterator
>>> bar, = results
>>> bar
<ExecutionUsageVariable name='bar', var_type='artifact'>
```

(continues on next page)

(continued from previous page)

```
>>> bar is results.out
True
```

Parameter Objects for `Usage.action`

These three classes define a deferred action that should be taken by some interface driver.

`Usage.`**`UsageAction: Type[`***`UsageAction`***`] = <class 'qiime2.sdk.usage.UsageAction'>`**

class qiime2.sdk.usage.**UsageAction**(*plugin_id*, *action_id*)

An object which represents a deferred lookup for a QIIME 2 action.

One of three "argument objects" used by *Usage.action()*. The other two are *UsageInputs* and *UsageOutputNames*.

Constructor for UsageAction.

The parameters should identify an existing plugin and action of that plugin.

Important

There should be an existing plugin manager by the time this object is created, or an error will be raised. Typically instantiation happens by executing an example, so this will generally be true.

Parameters

- **plugin_id** (*str*) – The (typically under-scored) name of a plugin, e.g. "my_plugin".
- **action_id** (*str*) – The (typically under-scored) name of an action, e.g. "my_action".

Raises

qiime2.sdk.UninitializedPluginManagerError – If there is not an existing plugin manager to define the available plugins.

Examples

```
>>> results = use.action(
...     use.UsageAction(plugin_id='dummy_plugin',
...                     action_id='params_only_method'),
...     use.UsageInputs(name='foo', age=42),
...     use.UsageOutputNames(out='bar1')
... )
>>> results.out
<ExecutionUsageVariable name='bar1', var_type='artifact'>
```

See also

UsageInputs, *UsageOutputNames*, *Usage.action*, *qiime2.sdk.PluginManager*

`Usage.`**`UsageInputs: Type[`***`UsageInputs`***`] = <class 'qiime2.sdk.usage.UsageInputs'>`**

class qiime2.sdk.usage.**UsageInputs**(***kwargs*)

A dict-like mapping of parameters to arguments for invoking an action.

One of three "argument objects" used by *Usage.action()*. The other two are *UsageAction* and *UsageOutputNames*.

Parameters should match the signature of the associated action, and arguments may be *UsageVariable* s or primitive values.

Constructor for UsageInputs.

Parameters

****kwargs** (*primitive or UsageVariable*) – The keys used should match the signature of the action. The values should be valid arguments of the action or variables of such arguments.

Examples

```
>>> results = use.action(
...     use.UsageAction(plugin_id='dummy_plugin',
...                     action_id='params_only_method'),
...     use.UsageInputs(name='foo', age=42),
...     use.UsageOutputNames(out='bar2')
... )
>>> results.out
<ExecutionUsageVariable name='bar2', var_type='artifact'>
```

See also

UsageAction, *UsageOutputNames*, *Usage.action*

Usage.**UsageOutputNames: Type[*UsageOutputNames*] = <class 'qiime2.sdk.usage.UsageOutputNames'>**

class qiime2.sdk.usage.**UsageOutputNames**(***kwargs*)

A dict-like mapping of action outputs to desired names.

One of three "argument objects" used by *Usage.action()*. The other two are *UsageAction* and *UsageInputs*.

All names must be strings.

Note

The order defined by this object will dictate the order of the variables returned by *Usage.action()*.

Constructor for UsageOutputNames.

Parameters

****kwargs** (*str*) – The name of the resulting variables to be returned by *Usage.action()*.

Raises

TypeError – If the values provided are not strings.

Examples

```
>>> results = use.action(
...     use.UsageAction(plugin_id='dummy_plugin',
...                     action_id='params_only_method'),
...     use.UsageInputs(name='foo', age=42),
...     use.UsageOutputNames(out='bar3')
... )
>>> results.out
<ExecutionUsageVariable name='bar3', var_type='artifact'>
```

See also

UsageAction, *UsageInputs*, *Usage.action*

Results and Assertions

The outputs of *Usage.action* are stored in a vanity class *UsageOutputs* which contain *UsageVariable*s. Assertions are performed on these output variables.

class qiime2.sdk.usage.**UsageOutputs**(*fields*, *values*)

A vanity class over *qiime2.sdk.Results*.

Returned by *Usage.action()* with order defined by *UsageOutputNames*.

class qiime2.sdk.usage.**UsageVariable**(*name*, *factory*, *var_type*, *usage*)

A variable which represents some QIIME 2 generate-able value.

These should not be used to represent primitive values such as strings, numbers, booleans, or lists/sets thereof.

UsageVariable.**assert_has_line_matching**(*path*, *expression*, *key=None*)

Communicate that the result of this variable should match a regex.

The default implementation is to do nothing.

Parameters

- **path** (*str*) – The relative path in a result's /data/ directory to check.
- **expression** (*str*) – The regular expression to evaluate for a line within *path*.
- **key** (*str*) – The key to match against a given semantic type if the output is a ResultCollection.

Note

Should not be called on non-artifact variables.

Examples

```
>>> bar, = use.action(
...     use.UsageAction(plugin_id='dummy_plugin',
...                     action_id='params_only_method'),
...     use.UsageInputs(name='foo', age=42),
...     use.UsageOutputNames(out='bar4')
... )
>>> bar.assert_has_line_matching('mapping.tsv', r'foo\s42')
...
>>> # A factory which will be used in the example to generate data.
>>> def factory():
...     import qiime2
...     # This type is only available during testing.
...     # A real example would use a real type.
...     a = qiime2.ResultCollection(
...         {'Foo': qiime2.Artifact.import_data('SingleInt', 1),
...          'Bar': qiime2.Artifact.import_data('SingleInt', 2)})
...     return a
...
>>> int_collection = use.init_artifact_collection('int_collection6', factory)
>>> bar, = use.action(
...     use.UsageAction(plugin_id='dummy_plugin',
...                     action_id='dict_of_ints'),
...     use.UsageInputs(ints=int_collection),
...     use.UsageOutputNames(output='bar5')
... )
>>> bar.assert_has_line_matching('file1.txt', r'1', 'Foo')
>>> bar.assert_has_line_matching('file1.txt', r'2', 'Bar')
>>> bar.assert_has_line_matching('file2.txt', r'1', 'Foo')
>>> bar.assert_has_line_matching('file2.txt', r'2', 'Bar')
```

UsageVariable.**assert_output_type**(*semantic_type*, *key=None*)

Communicate that this variable should have a given semantic type.

The default implementation is to do nothing.

Parameters

- **semantic_type** (*QIIME 2 Semantic Type or str*) – The semantic type to match.
- **key** (*str*) – The key to match against a given semantic type if the output is a ResultCollection.

Note

Should not be called on non-artifact variables.

Examples

```
>>> bar, = use.action(
...     use.UsageAction(plugin_id='dummy_plugin',
...                     action_id='params_only_method'),
...     use.UsageInputs(name='foo', age=42),
...     use.UsageOutputNames(out='bar6')
... )
>>> bar.assert_output_type('Mapping')
...
>>> # A factory which will be used in the example to generate data.
>>> def factory():
...     import qiime2
...     # This type is only available during testing.
...     # A real example would use a real type.
...     a = qiime2.ResultCollection(
...         {'Foo': qiime2.Artifact.import_data('SingleInt', 1),
...          'Bar': qiime2.Artifact.import_data('SingleInt', 2)})
...     return a
...
>>> int_collection = use.init_artifact_collection('int_collection7', factory)
>>> bar, = use.action(
...     use.UsageAction(plugin_id='dummy_plugin',
...                     action_id='dict_of_ints'),
...     use.UsageInputs(ints=int_collection),
...     use.UsageOutputNames(output='bar7')
... )
>>> bar.assert_output_type(semantic_type='SingleInt', key='Foo')
...
```

1.4.3 User Metadata API

This documents the `qiime2.Metadata` API. This may be used by QIIME 2 plugin developers or users of the QIIME 2 Python 3 API.

The `qiime.Metadata` class

class qiime2.**Metadata**(*dataframe*, *column_missing_schemes=None*, *default_missing_scheme='blank'*)

Store metadata associated with identifiers in a study.

Metadata is tabular in nature, mapping study identifiers (e.g. sample or feature IDs) to columns of metadata associated with each ID.

For more details about metadata in QIIME 2, including the TSV metadata file format, see the Metadata Tutorial at https://docs.qiime2.org.

The following text focuses on design and considerations when working with `Metadata` objects at the API level.

A `Metadata` object is composed of zero or more `MetadataColumn` objects. A `Metadata` object always contains at least one ID, regardless of the number of columns. Each column in the `Metadata` object has an associated column type representing either *categorical* or *numeric* data. Each metadata column is represented by an object corresponding to the column's type: `CategoricalMetadataColumn` or `NumericMetadataColumn`, respectively.

A `Metadata` object is closely linked to its corresponding TSV metadata file format described at https://docs.qiime2.org. Therefore, certain requirements present in the file format are also enforced on the in-memory object

in order to make serialized `Metadata` objects roundtrippable when loaded from disk again. For example, IDs cannot begin with a pound character (`#`) because those IDs would be interpreted as comment rows when written to disk as TSV. See the metadata file format spec for more details about data formatting requirements.

In addition to being loaded from or saved to disk, a `Metadata` object can be constructed from a `pandas.DataFrame` object. See the *Parameters* section below for details on how to construct `Metadata` objects from dataframes.

`Metadata` objects have various methods to access, filter, and merge data. A dataframe can be retrieved from the `Metadata` object for further data manipulation using the pandas API. Individual `MetadataColumn` objects can be retrieved to gain access to APIs applicable to a single metadata column.

Missing values may be encoded in one of the following schemes:

'blank'
The default, which treats *None*/*NaN* as the only valid missing values.

'no-missing'
Indicates there are no missing values in a column, any *None*/*NaN* values should be considered an error. If a scheme other than 'blank' is used by default, this scheme can be provided to preserve strings as categorical terms.

'INSDC:missing'
The INSDC vocabulary for missing values. The current implementation supports only lower-case terms which match exactly: 'not applicable', 'missing', 'not provided', 'not collected', and 'restricted access'.

Parameters

- **dataframe** (*pandas.DataFrame*) – Dataframe containing metadata. The dataframe's index defines the IDs, and the index name (`Index.name`) must match one of the required ID headers described in the metadata file format spec. Each column in the dataframe defines a metadata column, and the metadata column's type (i.e. *categorical* or *numeric*) is determined based on the column's dtype. If a column has `dtype=object`, it may contain strings or pandas missing values (e.g. `np.nan`, `None`). Columns matching this requirement are assumed to be *categorical*. If a column in the dataframe has `dtype=float` or `dtype=int`, it may contain floating point numbers or integers, as well as pandas missing values (e.g. `np.nan`). Columns matching this requirement are assumed to be *numeric*. Regardless of column type (categorical vs numeric), the dataframe stored within the `Metadata` object will have any missing values normalized to `np.nan`. Columns with `dtype=int` will be cast to `dtype=float`. To obtain a dataframe from the `Metadata` object containing these normalized data types and values, use `Metadata.to_dataframe()`.
- **column_missing_schemes** (*dict, optional*) – Describe the metadata column handling for missing values described in the dataframe. This is a dict mapping column names (str) to missing-value schemes (str). Valid values are 'blank', 'no-missing', and 'INSDC:missing'. Column names may be omitted.
- **default_missing_scheme** (*str, optional*) – The missing scheme to use when none has been provided in the file or in *column_missing_schemes*.

classmethod load(*filepath*, *column_types=None*, *column_missing_schemes=None*, *default_missing_scheme='blank'*)

Load a TSV metadata file.

The TSV metadata file format is described at https://docs.qiime2.org in the Metadata Tutorial.

Parameters

- **filepath** (*str*) – Path to TSV metadata file to be loaded.

- **column_types** (*dict, optional*) – Override metadata column types specified or inferred in the file. This is a dict mapping column names (str) to column types (str). Valid column types are 'categorical' and 'numeric'. Column names may be omitted from this dict to use the column types read from the file.
- **column_missing_schemes** (*dict, optional*) – Override the metadata column handling for missing values described in the file. This is a dict mapping column names (str) to missing-value schemes (str). Valid values are 'blank', 'no-missing', and 'INSDC:missing'. Column names may be omitted.
- **default_missing_scheme** (*str, optional*) – The missing scheme to use when none has been provided in the file or in *column_missing_schemes*.

Returns
Metadata object loaded from *filepath*.

Return type
Metadata

Raises
MetadataFileError – If the metadata file is invalid in any way (e.g. doesn't meet the file format's requirements).

See also

`save`

property columns

Ordered mapping of column names to ColumnProperties.

The mapping that is returned is read-only. This property is also read-only.

Returns
Ordered mapping of column names to ColumnProperties.

Return type
types.MappingProxyType

property column_count

Number of metadata columns.

This property is read-only.

Returns
Number of metadata columns.

Return type
int

Notes

Zero metadata columns are allowed.

See also

`id_count`

to_dataframe (*encode_missing=False*)

Create a pandas dataframe from the metadata.

The dataframe's index name (`Index.name`) will match this metadata object's `id_header`, and the index will contain this metadata object's IDs. The dataframe's column names will match the column names in this metadata. Categorical columns will be stored as `dtype=object` (containing strings), and numeric columns will be stored as `dtype=float`.

Parameters
encode_missing (*bool, optional*) – Whether to convert missing values (NaNs) back into their original vocabulary (strings) if a missing scheme was used.

Returns
Dataframe constructed from the metadata.

Return type
pandas.DataFrame

get_column (*name*)

Retrieve metadata column based on column name.

Parameters
name (*str*) – Name of the metadata column to retrieve.

Returns
Requested metadata column (`CategoricalMetadataColumn` or `NumericMetadataColumn`).

Return type
MetadataColumn

See also

get_ids

get_ids (*where=None*)

Retrieve IDs matching search criteria.

Parameters
where (*str, optional*) – SQLite WHERE clause specifying criteria IDs must meet to be included in the results. All IDs are included by default.

Returns
IDs matching search criteria specified in *where*.

Return type
set

See also

ids, *filter_ids*, *get_column*

Notes

The ID header (`Metadata.id_header`) may be used in the *where* clause to query the table's ID column.

merge (**others*)

Merge this `Metadata` object with other `Metadata` objects.

Returns a new `Metadata` object containing the merged contents of this `Metadata` object and *others*. The merge is not in-place and will always return a **new** merged `Metadata` object.

The merge will include only those IDs that are shared across **all** `Metadata` objects being merged (i.e. the merge is an *inner join*).

Each metadata column being merged must have a unique name; merging metadata with overlapping column names will result in an error.

Parameters

others (*tuple*) – One or more `Metadata` objects to merge with this `Metadata` object.

Returns

New object containing merged metadata. The merged IDs will be in the same relative order as the IDs in this `Metadata` object after performing the inner join. The merged column order will match the column order of `Metadata` objects being merged from left to right.

Return type

Metadata

Raises

ValueError – If zero `Metadata` objects are provided in *others* (there is nothing to merge in this case).

Notes

The merged `Metadata` object will always have its `id_header` property set to `'id'`, regardless of the `id_header` values on the `Metadata` objects being merged.

The merged `Metadata` object tracks all source artifacts that it was built from to preserve provenance (i.e. the `.artifacts` property on all `Metadata` objects is merged).

filter_ids (*ids_to_keep*)

Filter metadata by IDs.

Parameters

ids_to_keep (*iterable of str*) – IDs that should be retained in the filtered `Metadata` object. If any IDs in *ids_to_keep* are not contained in this `Metadata` object, a `ValueError` will be raised. The filtered `Metadata` object will retain the same relative ordering of IDs in this `Metadata` object. Thus, the ordering of IDs in *ids_to_keep* does not determine the ordering of IDs in the filtered `Metadata` object.

Returns

The metadata filtered by IDs.

Return type
Metadata

See also

get_ids, *filter_columns*

filter_columns (*, *column_type=None*, *drop_all_unique=False*, *drop_zero_variance=False*, *drop_all_missing=False*)

Filter metadata by columns.

Parameters

- **column_type** (*str, optional*) – If supplied, will retain only columns of this type. The currently supported column types are 'numeric' and 'categorical'.
- **drop_all_unique** (*bool, optional*) – If `True`, columns that contain a unique value for every ID will be dropped. Missing data (`np.nan`) are ignored when determining unique values. If a column consists solely of missing data, it will be dropped.
- **drop_zero_variance** (*bool, optional*) – If `True`, columns that contain the same value for every ID will be dropped. Missing data (`np.nan`) are ignored when determining variance. If a column consists solely of missing data, it will be dropped.
- **drop_all_missing** (*bool, optional*) – If `True`, columns that have a missing value (`np.nan`) for every ID will be dropped.

Returns
The metadata filtered by columns.

Return type
Metadata

See also

filter_ids

Metadata columns

class qiime2.**MetadataColumn** (*series*, *missing_scheme='blank'*)

Abstract base class representing a single metadata column.

Concrete subclasses represent specific metadata column types, e.g. `CategoricalMetadataColumn` and `NumericMetadataColumn`.

See the `Metadata` class docstring for details about `Metadata` and `MetadataColumn` objects, including a description of column types.

The main difference in constructing `MetadataColumn` vs `Metadata` objects is that `MetadataColumn` objects are constructed from a `pandas.Series` object instead of a `pandas.DataFrame`. Otherwise, the same restrictions, considerations, and data normalization are applied as with `Metadata` objects.

Parameters

- **series** (*pd.Series*) – The series to construct a column from.

- **missing_scheme** (*"blank", "no-missing", "INSDC:missing"*) – How to interpret terms for missing values. These will be converted to NaN. The default treatment is to take no action.

property name

Metadata column name.

This property is read-only.

Returns
Metadata column name.

Return type
str

property missing_scheme

The vocabulary used to encode missing values

This property is read-only.

Returns
"blank", "no-missing", or "INSDC:missing"

Return type
str

to_series(*encode_missing=False*)

Create a pandas series from the metadata column.

The series index name (`Index.name`) will match this metadata column's `id_header`, and the index will contain this metadata column's IDs. The series name will match this metadata column's name.

Parameters
encode_missing (*bool, optional*) – Whether to convert missing values (NaNs) back into their original vocabulary (strings) if a missing scheme was used.

Returns
Series constructed from the metadata column.

Return type
pandas.Series

See also

to_dataframe

to_dataframe(*encode_missing=False*)

Create a pandas dataframe from the metadata column.

The dataframe will contain exactly one column. The dataframe's index name (`Index.name`) will match this metadata column's `id_header`, and the index will contain this metadata column's IDs. The dataframe's column name will match this metadata column's name.

Parameters
encode_missing (*bool, optional*) – Whether to convert missing values (NaNs) back into their original vocabulary (strings) if a missing scheme was used.

Returns
Dataframe constructed from the metadata column.

Return type
pandas.DataFrame

See also

to_series

get_missing()

Return a series containing only missing values (with an index).

If the column was constructed with a missing scheme, then the values of the series will be the original terms instead of NaN.

get_value(*id*)

Retrieve metadata column value associated with an ID.

Parameters
id (*str*) – ID corresponding to the metadata column value to retrieve.

Returns
Value associated with the provided *id*.

Return type
object

has_missing_values()

Determine if the metadata column has one or more missing values.

Returns
`True` if the metadata column has one or more missing values (`np.nan`), `False` otherwise.

Return type
bool

See also

drop_missing_values, *get_ids*

drop_missing_values()

Filter out missing values from the metadata column.

Returns
Metadata column with missing values removed.

Return type
MetadataColumn

See also

has_missing_values, *get_ids*

get_ids(*where_values_missing=False*)

Retrieve IDs matching search criteria.

Parameters
where_values_missing (*bool, optional*) – If `True`, only return IDs that are associated with missing values (`np.nan`). If `False` (the default), return all IDs in the metadata column.

Returns
IDs matching search criteria.

Return type
set

See also
`ids`, *filter_ids*, *has_missing_values*, *drop_missing_values*

filter_ids (*ids_to_keep*)

Filter metadata column by IDs.

Parameters
ids_to_keep (*iterable of str*) – IDs that should be retained in the filtered `MetadataColumn` object. If any IDs in *ids_to_keep* are not contained in this `MetadataColumn` object, a `ValueError` will be raised. The filtered `MetadataColumn` object will retain the same relative ordering of IDs in this `MetadataColumn` object. Thus, the ordering of IDs in *ids_to_keep* does not determine the ordering of IDs in the filtered `MetadataColumn` object.

Returns
The metadata column filtered by IDs.

Return type
MetadataColumn

See also
get_ids

class qiime2.**NumericMetadataColumn** (*series*, *missing_scheme='blank'*)

A single metadata column containing numeric data.

See the `Metadata` class docstring for details about `Metadata` and `MetadataColumn` objects, including a description of column types and supported data formats.

class qiime2.**CategoricalMetadataColumn** (*series*, *missing_scheme='blank'*)

A single metadata column containing categorical data.

See the `Metadata` class docstring for details about `Metadata` and `MetadataColumn` objects, including a description of column types and supported data formats.

Exceptions

class qiime2.metadata.**MetadataFileError**(*message*, *include_suffix=True*)

Part II

Interfaces

CHAPTER

TWO

INTERFACE DEVELOPMENT

2.1 References

2.1.1 Interface developer API reference

When developing a QIIME 2 interface, you will use APIs defined in the `qiime2.sdk` submodule.

Interface development API

The `PluginManager` Object

`qiime2.sdk.`**`PluginManager`**(*add_plugins=True*)

Inputs and outputs

`qiime2.sdk.`**`Result`**()

Base class for QIIME 2 result classes (Artifact and Visualization).

This class is not intended to be instantiated. Instead, it acts as a public factory and namespace for interacting with Artifacts and Visualizations in a generic way. It also acts as a base class for code reuse and provides an API shared by Artifact and Visualization.

`qiime2.sdk.`**`Results`**(*fields*, *values*)

Tuple class representing the named results of an `Action`.

Provides an interface similar to a `namedtuple` type (e.g. fields are accessible as attributes).

Users should not need to instantiate this class directly.

qiime2.sdk.**Artifact**()

Base class for QIIME 2 result classes (Artifact and Visualization).

This class is not intended to be instantiated. Instead, it acts as a public factory and namespace for interacting with Artifacts and Visualizations in a generic way. It also acts as a base class for code reuse and provides an API shared by Artifact and Visualization.

qiime2.sdk.**Visualization**()

Base class for QIIME 2 result classes (Artifact and Visualization).

This class is not intended to be instantiated. Instead, it acts as a public factory and namespace for interacting with Artifacts and Visualizations in a generic way. It also acts as a base class for code reuse and provides an API shared by Artifact and Visualization.

qiime2.sdk.**ResultCollection**(*collection=None*)

Actions

qiime2.sdk.**Action**()

QIIME 2 Action

qiime2.sdk.**Method**()

QIIME 2 Method

qiime2.sdk.**Visualizer**()

QIIME 2 Visualizer

qiime2.sdk.**Pipeline**()

QIIME 2 Pipeline

Utility functions

qiime2.sdk.**parse_type**(*string*, *expect=None*)

Convert a string into a type expression

Parameters

- **string** (*str*) – The string type expression to convert into a TypeExpression
- **expect** (*{'semantic', 'primitive', 'visualization'}, optional*) – Will raise a TypeError if the resulting TypeExpression is not a member of *expect*.

Return type
type expression

qiime2.sdk.**parse_format**(*format_str*)

qiime2.sdk.**type_from_ast**(*ast*, *scope=None*)

Convert a type ast (from *.to_ast()*) to a type expression.

Parameters

- **ast** (*json compatible object*) – The abstract syntax tree produced by *to_ast* on a type.
- **scope** (*dict*) – A dictionary to use between multiple calls to share scope between different types. This allows type variables from the same type map to be constructed from an equivalent type map. Scope should be shared within a given call signature, but not between call signatures.

Return type
type expression

Exceptions

qiime2.sdk.**ValidationError**()
Common base class for all non-exit exceptions.

qiime2.sdk.**ImplementationError**()
Common base class for all non-exit exceptions.

qiime2.sdk.**UninitializedPluginManagerError**()
Common base class for all non-exit exceptions.

Part III

The Framework

CHAPTER

THREE

FRAMEWORK DEVELOPMENT

This part documents various aspects of the QIIME 2 Framework.

3.1 Explanations

3.1.1 QIIME 2 architecture overview

The goal of this document is to give the reader a high-level understanding of the components of QIIME 2, and how they are inter-related.

At the highest level, there are three kinds of components in QIIME 2:

- The interfaces, which are responsible for translating user intent into action.
- The framework, whose behavior and purpose will be described in further detail below.
- The plugins, which define *all* domain-specific functionality.

Fig. 3.1: **Box and Arrow diagram of QIIME 2 components.** Interfaces only interact with plugins through the framework, which will invoke plugin behavior as needed. Solid arrows are direct dependency. Dash-dotted arrows are a deferred dependency (via entry-point).

The above diagram illustrates the most important restriction of the architecture. Interfaces cannot and *should not* have any particular knowledge about plugins ahead of time. Instead they must request that information from the framework, which provides a high-level description of all of the actions available via SDK (Software Development Kit) objects.

At first glance, this restriction may seem onerous. However, because interfaces cannot communicate directly with plugins, they also never need to coordinate with them. This means interfaces are entirely decoupled from the plugins and more importantly, plugins are always decoupled from interfaces. A developer of a plugin, does not need to concern themselves with providing any interface-specific functionality, meaning development of both plugins and interfaces can be done in parallel. Only changes in the framework itself require coordination between the other two component types. This key constraint coupled with a set of semantically rich SDK objects allows *multiple* kinds of interfaces to be dynamically generated. This allows QIIME 2 to adapt its UI to both the audience and the task at hand.

Detailed Component Diagram

A more complete version of the above figure is found below:

Here we observe that interfaces use a particular sub-component of the framework called the SDK. We also see that one of the interfaces is built into the framework itself (the Artifact API), however it is not any more privileged compared to any of the other interfaces, and none of the other interfaces use it directly.

Looking now at the plugins we see that they use a sub-component of the framework called the Plugin API. This is responsible for constructing and registering the relevant SDK objects for use by interfaces. We also see that plugins can depend on other plugins.

At this point the rough picture of how an interface uses a plugin can be seen. Plugins are loaded by the framework's SDK via an entry-point (more on that later). This in turn causes the plugin code to interact with the Plugin API, which constructs SDK objects. These SDK objects are then introspected and manipulated by any number of Interfaces.

Following A Command Through QIIME 2

To get a better idea of where the responsibility of these components starts and ends, we can look at a sequence diagram describing the execution of an action by a user.

This figure has four components: a User, an Interface, the Framework, and a Plugin. We see first, a User invoking some action with some files and parameters. The Interface receives this and is activated. It locates the plugin and the action requested from the Framework, receiving SDK objects (not shown). Then it loads the provided files as QIIME 2 Artifacts. It is then ready to call the action (an SDK object) with the User's artifacts and parameters.

The Framework then provides some input validation (it is much faster to check that the data provided will work for the action requested than to fail halfway through a very long process, requiring the User to start over). The Framework then identifies what format the data should be in, and invokes relevant code defined by a Plugin (though not necessarily the same one) for converting that data. Finally, with validated input and a compatible format, the data is provided to the plugin to perform whatever task the User intended.

Once finished, the Plugin returns the results and the Framework will again convert that data into a particular format for storage using a Plugin. The Framework then writes that data into an archive (as `/data/`) and records what steps just occurred (in `/provenance/`). This completed archive is now an artifact and is returned to the Interface. The Interface decides to save the artifact to a file and then returns that to the User.

Summary

In this example we see that the activation of each component is strictly nested. It forms a sort of "onion of responsibility" between the component layers. We also note that the Interface waits for the task to finish before becoming inactive; there are other modes of calling actions which are asynchronous and can be used instead. In either case, we see that each component is successively responsible for less tasks which become more specific as we move to the right.

The end result is:

- The Interface need only care about communicating with the User.
- The Plugin need only care about manipulating data to some effect.
- The Framework concerns itself with coordinating the overall effort and recording the data surrounding the action.

Fig. 3.2: **Detailed Box and Arrow diagram of QIIME 2 components.** Solid arrows are a direct dependency. Dash-dotted arrows are a deferred dependency (via entry-point). Dashed rounded boxes surrounding other components indicate a group of like-components. The larger gray box indicates a nested component, containing sub-components. Text within angle-brackets (<>) indicate a Python package/import name.

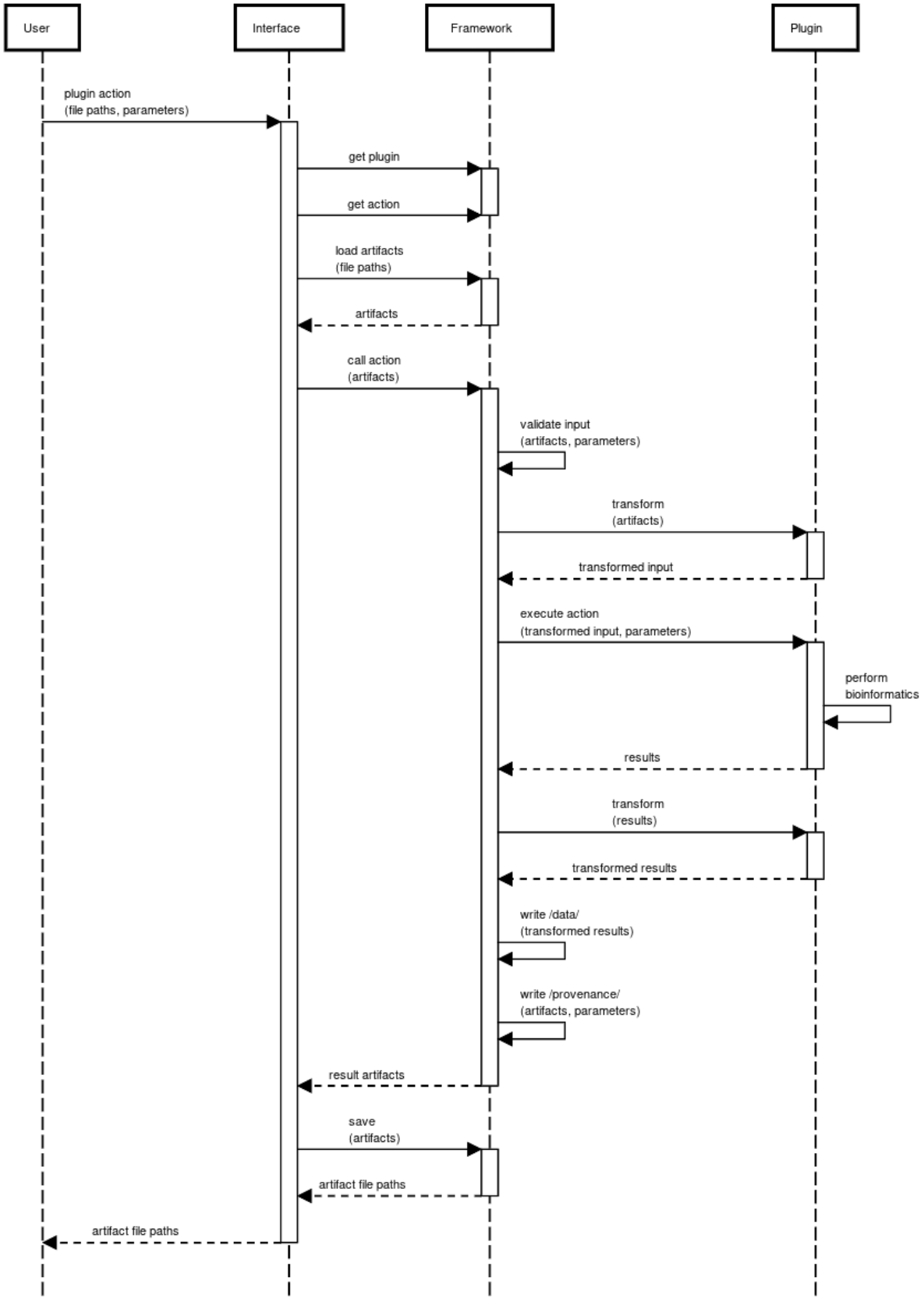


Fig. 3.3: **UML Sequence Diagram of an action-call in QIIME 2** This diagram is read from top to bottom, which indicates the passage of some non-specific amount of time. Components are vertical columns. An activated state of a component is indicated by a narrow box. Components can perform actions either upon other components, or upon themselves. These actions are denoted with a solid arrow pointing at the actor in question. The label indicates what action is performed and, when provided, parenthesis indicate some kind of argument that is provided. Not all arguments are

3.1.2 How Data is Stored

> **Warning**
>
> Some of the discussion in this section uses *semantic type* as a synonym for *artifact class*. While closely related, we're working on clarifying our language to distinguish these ideas. We haven't reviewed/updated this section for this yet.

In any software project, data needs to be stored (or persisted). The way that this is accomplished can impact every facet of the software's design. In order to better demonstrate *why* certain aspects of QIIME 2 exist as they are we will highlight what goals or constraints QIIME 2 has, and then demonstrate how QIIME 2 achieves each goal.

Goals for data storage in QIIME 2

The design of QIIME 2's approach to data storage was motivated by the following goals, which we consider to be contraints. Data should be stored in a way that:

- is accessible long-term (e.g. 20 years), or "future-proof"
- is as convenient as possible to transfer between users
- makes it possible to determine when data is or is not valid as input for a specific analysis
- ease interoperability between tools
- can be extended by plugins
- and include details on its origin and history (provenance), facilitating trust and reproducibility.

These goals directly impact many of the design decisions made in QIIME 2. The solution described in this section seeks to address each of these goals. There are many other ways to solve these problems when one or more of these constraints are lifted, however QIIME 2 chooses these constraints because we believe they are *useful* to the scientist, will allow *composable* software to be developed and reused to advance the state of the art, and support FAIR data principles.

Accessibility and Transferability

QIIME 2 stores all data as a directory structure inside of a ZIP file. There is a *payload* directory named `/data/` where data is stored in a common format. This permits additional *metadata* to be stored alongside the data (in non-`/data/` directories or files).

A directory-based archive is used to store data in a way that is accessible. When a common format exists for a data type (e.g., FASTA format for sequence data) it should be used to be as accessible as possible. When such a format does not exist, it should be stored in plain-text structure that is as self-descriptive as possible. The goals is that a person in 20 years might be able to glance at it, and roughly understand what the purpose of a given document is (assuming file-systems and text-based encodings still make sense in the future).

Because some common formats are paired with others, or reused multiple times to represent multiple entities (e.g. per-sample fastq files), the data that a tool needs is sometimes a *directory* of formats. Alternatively a new format could be invented with new rules (though this would make interoperability difficult).

For these reasons, QIIME 2 stores data as a directory structure. In particular data such as FASTA or newick will be considered the *Payload* which is to be delivered to a tool.

> **Note**
>
> QIIME 2's `.qza` and `.qzv` files are ZIP files with a specific internal structure. You can open these with any standard unzip utility (e.g., WinZip, 7zip, `unzip`), even on systems that don't have QIIME 2 installed.

There is a challenge with using directory structures as a way of storing data. Moving directory structures is inconvenient as they do not exist as a single file. A common way to fix this is to zip a file and extract it at the destination. This is exactly what QIIME 2 does. ZIP files additionally have the advantage of being incredibly well supported by a *wide* array of software. Some software manipulates ZIP files directly (often built into an operating system's graphical interface) and others use ZIP files as a backing structure (such as `.docx` and `.epub`). Because it is so widely used, maintaining the long-term accessibility of data is much more likely.

The following documents provide more detailed information about directory structure and ZIP file used in QIIME 2 `Archives`:

- *Anatomy of an Archive*
- *Archive versions*

Input Validation (Type Checking)

QIIME 2 stores a file named `metadata.yaml` alongside the `/data/` directory. This file contains the *type* of the data, which QIIME 2 can use to validate that a given ZIP file is valid input for a given *action*.

Given that a QIIME 2 `Archive` provides a way to store a *payload* (i.e., data) and a way to move it around, there needs to be a way to *describe* it so that the computer can determine if a given payload is valid input for a given operation. This helps prevent user errors due to accidental misuse of the data, and allows *interfaces* to provide a more complete and rich user interface.

To accomplish this, we need data about the data, or *metadata* (in the general sense, this should not be confused with QIIME 2's sample/feature metadata). If the *payload* is placed in a *subdirectory* then we can store additional files which can contain this *metadata* without needed to worry about filename conflicts with the payload itself. Now QIIME 2 is able to record a type and anything else that may enable the computer (or user) to make a more informed decision about the use of a given piece of data.

See *Semantic Types, Primitives, and Visualizations* to get more information on how types are used and defined in QIIME 2.

Interoperability and Extension

QIIME 2 stores a string called a *Directory Format* in `metadata.yaml` which instructs the computer what the specific layout of `/data/` is. Once this is known, it is possible to convert that data into other formats. *plugins* can define new formats and request data in specific *views*.

Different tools expect different file formats or in-memory data structures. Many of these are *semantically* compatible — in other words, they can carry the same information but in different ways. Another way to state this is that these different formats and data-structures each represent a different *view* of the same data.

If we combine this idea with a *semantic type* we are able to use the abstraction of the type to ignore the view when reasoning about composition of *actions*. While the *semantic type* may be adequate for describing what data is used for, it does not provide a means to structure it (on-disk or in-memory). For this we use a *view*. In particular, when storing data

for later use (or sharing) it is necessary to save it to disk in some way (in particular, we need to store it in `/data/`). We use a *directory format* to accomplish this purpose. Directory formats have a name that is recorded in `metadata.yaml` and defines how `/data/` is to be structured.

See *File Formats and Directory Formats* to learn more about QIIME 2's formats and directory formats.

Provenance Metadata

Inside of each Archive, QIIME 2 stores metadata about how that archive was generated. We call this "provenance". Notably, each Archive contains provenance information about *every* prior QIIME 2 *Action* involved in its creation, from `import` to the most recent step in the analysis.

This provenance information includes type and format information, system and environment details, the Actions performed and all parameters passed to them, and all registered citations.

See *Decentralized retrospective provenance tracking* to learn more about data provenance storage in QIIME 2.

3.1.3 Anatomy of an Archive

QIIME 2 stores data in a directory structure called an *Archive*. These archives are zipped to make moving data simple and convenient.

The directory structure has a single root directory named with a *UUID* which serves as the *identity* of the archive. Additional files and directories present in the archive are described below.

The Most Important File: `metadata.yaml`

In the root of an *archive* directory, there is a file named `metadata.yaml`. This file describes the *type*, the *directory format*, and repeats the *identity* of a piece of data.

An example of this file:

```
uuid: 45c12936-4b60-484d-bbe1-98ff96bad145
type: FeatureTable[Frequency]
format: BIOMV210DirFmt
```

It is possible for `format` to be set as `null` only when `type` is set as `Visualization` (representing a *Visualization (Type)*). This implies that the `/data/` directory (described below) does not have a schema.

Data Goes In `/data/`

Where data is stored, the *payload* of an archive, is in an aptly named `/data/` subdirectory. The structure of this subdirectory depends on the payload.

If the archive is a *visualization*, then the payload is a (possibly interactive) visualization, which is implemented as a small static website (with an `index.html` file and any other assets).

If the archive is an *artifact*, then the payload is determined by the *directory format*.

Provenance Goes In `/provenance/`

In addition to storing data, QIIME 2 stores *metadata* including what actions were performed to generate the current `Result`, what versions of QIIME 2 and other dependencies were used, and what references to cite if the data is used in a publication (for example)

As it relates to the archive structure, the `/provenance/` directory is designed to be self-contained and self-referential. This means that it duplicates some of the information available in the root of the *archive*, but this simplifies the code responsible for tracking and reading provenance.

Fig. 3.4 illustrates this idea.

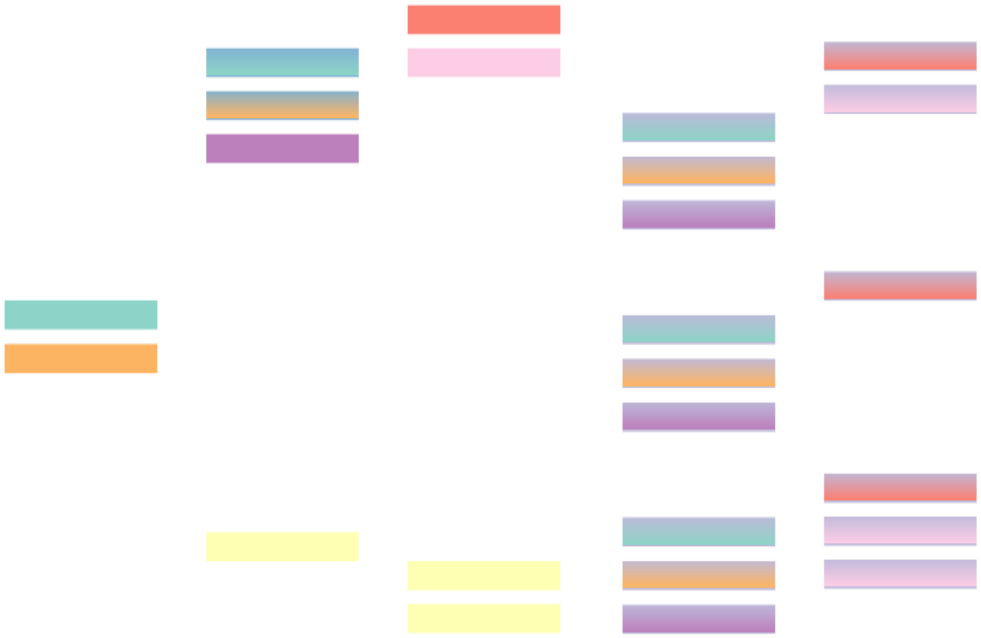


Fig. 3.4: Description of the QIIME 2 archive structure.

Looking closely we see the previously described `/data/` directory and `metadata.yaml` file, in addition to a `VERSION` file (described below), and the `/provenance/` directory in question.

Following the provenance directory, we see that the provenance structure is repeated within the `/provenance/artifacts/` directory. This directory contains the *ancestral provenance* of all *artifacts* used up to this point. Because the structure repeats itself, it is possible to create a new provenance directory by simply adding all input artifacts' `/provenance/` directories into a new `/provenance/artifacts/` directory. Then the `/provenance/artifacts/` directories of the original inputs can be also merged together. Because the directories are named by a *UUID*, we know the *identity* of each ancestor, and if seen twice, can simply be ignored. This simplifies the problem of capturing *ancestral provenance* to one of merging uniquely named file-trees.

Why a ZIP File?

ZIP files are a ubiquitous and well understood format. There is a huge variety of software available to read and manipulate ZIP files.

The ZIP format additonally enables random access of files within the archive, making it possible to read data without extracting the entire contents of the ZIP file (in contrast to a linear archive like TAR). This is very convenient if you need to, for example, pull a single file out of a large ZIP archive containing many files.

> **Note**
>
> `qiime2.core.archive.archiver:_ZipArchive` is the structure responsible for managing the contents of a ZIP file (using `zipfile:ZipFile`).

Rules for identifying an archive

Every QIIME 2 *archive* has the following structure:

A root directory which is named a standard representation of a UUID (version 4), and a file within that directory named `VERSION`.

The *UUID* is the *identity* of the archive, while the `VERSION` file provides enough detail to determine how to parse the rest of the archive's structure.

Within `VERSION` the following text will be present:

> **Note**
>
> The `VERSION` file is intentionally not in YAML, INI, or any other common data serialization or configuration format. This is to discourage the situation where important archive files are reformatted from YAML to another format and VERSION is updated (e.g., for consistency), breaking backwards compatibility.

```
QIIME 2
archive: <integer version>
framework: <version string>
```

`<integer version>` is the version that the archive was saved with. This is used to identify the *schema* of the archive structure, which evolves over time to support new functionality, allowing software to dispatch appropriate parsing logic.

As a historical example, *archive version 0* had no `/provenance/` directory, as QIIME 2's provenance tracking hadn't yet been implemented. The archive version `0` parser therefore doens't look for the `/provenance/` directory. This particular case would be easy enough to check for at runtime, but this versioning scheme allows for more complex differences between archive versions while ensuring that QIIME 2 will always be able to interpret older archives.

> **Note**
>
> These rules are encoded in `qiime2.core.archive.archiver:_Archive`.

The different versions of QIIME 2 Archive Formats are detailed in *Archive versions*.

3.1.4 Semantic Types, Primitives, and Visualizations

> **Warning**
>
> Some of the discussion in this section uses *semantic type* as a synonym for *artifact class*. While closely related, we're working on clarifying our language to distinguish these ideas. We haven't reviewed/updated this section for this yet.

If you're unfamilar with the various ways that the word *type* is used in the context of QIIME 2, we recommend reading *Semantic types, data types, file formats, and artifact classes* before this document. This document provides a deep dive into semantic types, primitives, and visualizations in QIIME 2: in other words, descriptors of the various inputs and/or outputs to QIIME 2 *Actions*.

An Analogy

Suppose we were interested in modeling a system in which there were *hands*, *utensils* and *tasks*. We are trying to determine which *utensils* can be used to complete which *tasks*. Here *utensils* are like QIIME 2 *artifacts* and *tasks* are QIIME 2 *actions*. Imagine then, that this hand (unlike yours or mine) is not intelligent and will grasp anything to carry out its work without further consideration. The *hand* then, represents the computer.

To make this example more concrete, let's suppose we wanted to write a note. We provide to the hand a fork, and ask it to return to us a note. Accepting the fork, the hand blithely tears the paper into shreds and hands you what remains, considering its mission to be complete.

This is not ideal.

Instead, lets give the hand some rules it can follow to determine if it should perform a task with a utensil. For example, we would have prefered it use a pencil when writing a note instead of a fork. To describe this more formally, we might say that a task "write note" requires a pencil and returns a note:

```
write note : pencil -> note
```

Repeating the above situation, we provide a fork to the hand and ask it to write a note. The hand observes that a fork cannot be used to make a note as it is not a pencil, and our paper remains unshredded (though still blank).

However, if we were to write a note with a pen, according to the above rules, we would also be refused, as only pencils are allowed to write notes. Instead of helping us, the type annotation above seems to be in the way.

Finding a way to fix this mismatch while still constraining inputs is the fundamental goal of a type system and is where a lot of complexity arises. There is also a lot of freedom in implementation, some type systems are as strict as the above, and provide little *expressive power*, others provide a great deal (becoming Turing-complete in the process) at the cost of complexity.

Ultimately it is important to remember what the purpose is. A type system should abstract away details that the computer *needs* but which impede a person's comprehension of a system. A good type system should be a compromise between the fuzzy (and indistinct) world of language that people understand, and robust formal systems that computers can use.

Defining a Type

In QIIME 2 there are 3 *kinds* of types, all of which use the same underlying grammar. Only one of these kinds can be extended, the *Semantic Type*. The other two, *Primitive Type* and *Visualization*, are built into the framework.

Semantic types are the only kind that can be extended, so let's start there with our example from before.

To create a type, we use the *SemanticType* factory:

```
Pencil = SemanticType('Pencil')
```

That's it! Let's define some more:

```
Pen = SemanticType('Pen')
Fork = SemanticType('Fork')
Spoon = SemanticType('Spoon')
Chalk = SemanticType('Chalk')
```

To let QIIME 2 know that these new types exist, we'll need to register them on our *plugin object* with *register_semantic_types*:

```
plugin.register_semantic_types(Pencil, Pen, Fork, Spoon, Chalk)
```

Now QIIME 2 is aware of these types and we can use them.

There are only 5 types right now, but imagine we had dozens, it might get a bit hard to keep them all straight. To make it easier for us to talk about them, we can try to group similar types together. Looking at our type, we seem to have two broad categories so far, writing and dining utensils. Let's define some *composite types* to group them:

```
Dining = SemanticType('Dining', field_names=['utensil'],
                      field_members={ 'utensil': (Fork, Spoon) })

Writing = SemanticType('Writing', field_names=['implement'],
                       field_members={ 'implement': (Pen, Pencil, Chalk) })
```

And of course we should register these as well:

```
plugin.register_semantic_types(Dining, Writing)
```

Before explaining what the new parameters are, let's use these and circle back:

```
Writing[Pen]
Writing[Pencil]
Writing[Chalk]

Dining[Spoon]
Dining[Fork]
```

Since we don't have many types, this may look a little silly, but now we can talk about dining and writing utensils as broad groups. What happens if we try to mix these? Let's make some *dining chalk* (gross!):

```
Dining[Chalk]
```

It produces the following exception:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/evan/workspace/qiime2/qiime2/qiime2/core/type/grammar.py", line 68, in _
```

(continues on next page)

(continued from previous page)

```
 ↪__getitem__
    self._validate_field_(*args)
  File "/home/evan/workspace/qiime2/qiime2/qiime2/core/type/semantic.py", line 184,␣
 ↪in _validate_field_
    raise TypeError("%r is not a variant of %r." % (value, varfield))
TypeError: Chalk is not a variant of Dining.field['utensil'].
```

It appears chalk is off the menu. Let's go back over the definition for `Dining`:

```
Dining = SemanticType('Dining', field_names=['utensil'],
                      field_members={ 'utensil': (Fork, Spoon) })
```

Unlike the simpler types, this adds `field_names` and `field_members`, if we were to look at `Dining` on its own:

```
print(Dining)
```

We see:

```
Dining[{utensil}]
```

This `{utensil}` is the *field name*, because `field_names` is a list, we could have more than one, letting us get combinatorical, but that usually isn't necessary.

Looking at `field_members` we see that for the field named `utensil` there are two permitted *variants*: `Fork` and `Spoon`. This is why creating `Dining[Chalk]` didn't work so well, `Chalk` isn't a variant of `Dining`'s field `utensil`.

Extending a Type

Suppose we were satisfied with the above vocabulary of utensils. So much so, we considered ourselves to have described all of the utensils we would ever need. Obviously that isn't going to be true, so there should be a way for other plugins to define new types, while still being able to organize them into our existing hierarchy of labels.

A *seperate plugin* could then define something like this:

```
Knife = SemanticType('Knife', variant_of=[Dining.field['utensil']])
```

Breaking this down, it is similar to some of the earlier invocations of the *`SemanticType`* factory, but there's a new argument for `variant_of` which seems to be providing a list of *fields* from other *composite types*. This means a plugin can extend existing types as needed. In this case, we've suggested that in addition to forks and spoons, there are knives.

We can also create new categories and types that belong to more than one category. Let's create a category for kitchen utensils. A knife has already been defined, but you wouldn't cook with a steak knife, and you wouldn't eat with a chef's knife, so there's more we can add to the knife's story:

```
Kitchen = SemanticType('Kitchen', field_names=['utensil'],
                       field_members={ 'utensil': [Knife] })

Spatula = SemanticType('Spatula', variant_of=[Kitchen.field['utensil']])
PastryBag = SemanticType(
    'PastryBag', variant_of=[Kitchen.field['utensil'], Writing.field['implement']])
```

This creates a new `Kitchen` category, and adds `Knife` as a member. It also adds `Spatula` to `Kitchen` and adds `PastryBag` to both `Kitchen` *and* `Writing`. In case you don't know what a pastry bag is (like me), it's how you would write "Happy Birthday" on a cake. Just as not all knives are the same, not all pastry bags are well suited to writing (some are better for making decorative frosting-flowers).

```
Dining[Knife]
Kitchen[Knife]
Kitchen[Spatula]
Kitchen[PastryBag]
Writing[PastryBag]
```

We should *register these* before we forget:

```
other_plugin.register_semantic_types(Knife, Kitchen, Spatula, PastryBag)
```

Primitive Types

Primitive types are the other main kind of type you'll use in QIIME 2. These closely match their associated data types making them simple to work with, but they also have a few extra tricks up their sleeves to make it possible to automatically generate rich user *interfaces*. The purpose of these types is to explain what kinds of parameters can be provided to an *action*.

There are a few basic types:

- *Int*
- *Bool*
- *Float*
- *Str*

These work essentially as you would expect, an `Int` holds an integer, a `Str` holds a unicode string. They are capitalized to make differentiating them from their Python counterparts (`int` and `str`) simpler.

There are a few collection types:

- *List[{elements}]*
- *Set[{elements}]*

Each of these allows you to provide one of the above basic types to their `{elements}` field.

There are also some metadata types:

- *Metadata* (the type, not the `object`)
- *MetadataColumn[{type}]* which has the following column types (for the `{type}` field):
  - *Numeric*
  - *Categorical*

From these we can construct simple expressions like:

```
Int
List[Int]
Set[Str]
Metadata
MetadataColumn[Numeric]
```

Of course, just writing down a type isn't necessarily useful unless we can *use* it for something. Let's do that now:

```
# These are true:
assert      1 in Int
# These are not:
```

(continues on next page)

(continued from previous page)

```
assert not  "banana" in Int
assert not  0.5 in Int

# True:
assert      "banana" in Str
# Not true:
assert not  1 in Str

# True:
assert      [1, 2, 3] in List[Int]
# Not true:
assert not  ['a', 'b', 'c'] in List[Int]

# True:
assert      ['a', 'b', 'c'] in List[Str]
# Not true:
assert not  [1, 2, 3] in List[Str]
```

While these are all useful constructs, real-world user input must often be constrained to just a few valid strings, or a real number bounded from zero to one. To express these ideas we need a little bit more.

Refining a Type

A *refinement type* is a type that possesses a *predicate* which further constrains the domain of a type. Thats a formal definition anyway. The important piece is the *predicate*, which is a boolean "function" describing whether a given instance is *in* the domain, or *out* of the domain. This means we can *refine* the type to suite our needs.

Suppose we were a graphical interface. A common UI element is a dropdown list containing predetermined choices. We can express that with a primitive type!

Choices

Let's see an example of this, using the `Choices` predicate:

```
# These are Python objects, so we can assign to variables:
dropdown = Str % Choices({'banana', 'apple', 'pear'})

assert      'banana' in dropdown
assert not  'grape' in dropdown
assert not  0.5 in dropdown
```

The `%` operator adds a *predicate* like `Choices` to a type. You can read it as "string modulo choices" in your head if you like. It almost makes sense.

You can see how a graphical interface could inspect this type and automatically generate a dropdown list containing "banana", "apple", and "pear".

Let's try something harder, suppose we wanted to describe some checkboxes, where the choices can be selected at most once, but multiple different choices are allowed:

```
checkboxes = Set[Str % Choices({'banana', 'apple', 'pear'})]

assert      {'banana'} in checkboxes
assert      {'apple', 'banana'} in checkboxes
```

(continues on next page)

(continued from previous page)

```
assert not  {'banana', 'grape'} in checkboxes
assert not  {1, 2, 3} in checkboxes
assert not  'banana' in checkboxes
```

We might read that as "A set of strings modulo the choices of banana, apple, and pear". It is a mouthful, but we've just described an entire UI element in a single line.

Additionally, this is *abstract*, we never actually asked for a checkbox. So the interface can make its own decision about how best to represent this type in its UI. For example a command line interface cannot show checkboxes, but it might have an interactive dialog, or it may just accept multiple arguments for the parameter. A programmatic interface may simply accept a set object instead. It is up to the interface to make the best choice it can. The plugin developer does not need to worry about the representation.

Interface Developer Note

An easy way to transfer (or dispatch on) a type is to use the `.to_ast()` method which will provide a JSON structure describing the type in a machine-friendly representation.

For example:

```
import json

print(json.dumps(checkboxes.to_ast(), indent=2, sort_keys=True))
```

```
{
  "fields": [
    {
      "fields": [],
      "name": "Str",
      "predicate": {
        "choices": [
          "banana",
          "apple",
          "pear"
        ],
        "name": "Choices",
        "type": "predicate"
      },
      "type": "primitive"
    }
  ],
  "name": "Set",
  "predicate": {},
  "type": "collection"
}
```

Range

Another predicate we can use is *Range*:

```
proportion = Float % Range(0, 1, inclusive_end=True)

assert      0 in proportion
assert      0.5 in proportion
assert      1 in proportion
assert not  -1.5 in proportion
assert not  1.5 in proportion
assert not 'banana' in proportion
```

This can be combined with *Int* as well. As before we can nest these kinds of expressions inside of *Set* and *List*.

Semantic Properties

Leaving behind the primitive types and returning the the semantic types, there is a final trick we can use to constrain the semantics of a type. It is to use the *Properties* predicate. This predicate can only be attached to semantic types, so we usually call them semantic properties of the type.

Thinking back to our example involving utensiles, there was a type named:

```
Kitchen[Knife]
```

Suppose we were a plugin that specialized in cutting things, with actions such as filleting fish, paring fruit, etc. To other plugins, the distinction between different kinds of cutlery might be uninteresting. To us, however, *cutting things is what we do*. We wouldn't fillet a fish without a fillet knife. The nomenclature discussed so far lacks that granularity.

In a perfect world, we would extol the virtues of being specific about cutlery, suggesting others adopt a new category `Cutlery[{knife}]` to help better model the world of things-hands-can-use. Building consensus can be slow, though, and you are still interested in inter-operating with other plugins (even if they don't understand why anyone would need more than one kind of knife).

To fix this, you can add a property:

```
Kitchen[Knife % Properties('fillet')]
```

What this means is that you've created a new *subtype* of `Knife` using the label "fillet". There aren't any rules for recognizing a fillet knife, so its something that has to be explicitly attached (but that is the case with all semantic types).

There can additionally be more than one property on a type:

```
Kitchen[Knife % Properties(include=['fillet', 'sharp'])]
Kitchen[Knife % Properties(include=['paring'], exclude=['sharp'])]
```

Now we can describe things like a *sharp fillet knife* or a *dull paring knife*. To illustrate how these are used, we need to talk more about *subtyping*.

Semantic Subtyping

A subtype is some type that is *substitutable* for another. Here's another way to think about it: the domain of the subtype exists *entirely within* the domain of the supertype. Anywhere you could use a supertype, a subtype will suffice.

There are two ways to create this relation: with a semantic property (described above), or with a *union operator*: `|`. In order to use a subtyping relation, we also need an operator to test the relation, for that we can use <= and >= (which matches the Python `set` API).

Let's try it out:

```
assert      Spoon <= Spoon  # is spoon a subtype of spoon?
assert      Spoon >= Spoon  # is spoon a supertype of spoon?
assert not  Fork <= Spoon   # is fork a subtype of spoon?
assert not  Fork >= Spoon   # is fork a supertype of spoon?
```

Here we have the makings of equality and inequality. We see that any instance of a `Spoon` can be substituted wherever a `Spoon` is required (which is obvious enough), and we also see that a `Fork` will not do, when a `Spoon` is needed (soup comes to mind).

Unions

Of course, this subtyping relationship isn't very interesting, let's use the union operator to *construct a supertype*:

```
assert      Spoon <= Spoon | Fork
assert      Fork <= Spoon | Fork
# The relationship has direction:
assert not  Spoon >= Spoon | Fork
assert not  Fork  >= Spoon | Fork
# And of course, unrelated things are not equal
assert not  Knife <= Spoon | Fork
assert not  Knife >= Spoon | Fork
```

Using this mechanism we can define actions that accept a broad range of types, while still being specific about which types are known to work. Also instead of using `A <= B <= A` to test equality, we can use `.equals` (the operator is reserved for hash-equality).

We can also evaluate more sophisticated expressions:

```
assert not  Dining[Knife].equals(Kitchen[Knife])
assert      Dining[Knife] <= Kitchen[Knife] | Dining[Knife]
# Union types also have subtyping relations:
assert      Writing[Pencil] | Writing[Pen] <= Writing[Pencil] | Writing[Pen] |␣
 ↪Writing[Chalk]
# or more concisely:
assert      Writing[Pencil | Pen] <= Writing[Pencil | Pen | Chalk]
```

In QIIME 2, subtyping and equality are *extensional*, meaning that the order and form do not matter, only the meaning.

In other words, these expressions are the same:

```
assert  (Writing[Pencil] | Writing[Pen]).equals(Writing[Pen | Pencil])
assert  Writing[Pencil | Pen].equals(Writing[Pen | Pencil])
```

Properties

Let us return now to the other way of constructing a subtyping relation, the *semantic property*. We had the following definitions which we'll assign to a variable, since they are lengthy:

```
sharp_fillet = Kitchen[Knife % Properties(include=['fillet', 'sharp'])]
dull_paring  = Kitchen[Knife % Properties(include=['paring'], exclude=['sharp'])]
```

How should these relate to a plain `Kitchen[Knife]`? Well, because we've added information about the knife, we've *refined* the domain, and so we have a *subtype*. In other words, our fancy knifes can be used wherever a normal knife can be used. The way to think about this is we haven't created something new, paring knives and fillet knives were always in the set of `Kitchen[Knife]`, but until we added the property we were unable to distinguish them.

```
assert  sharp_fillet <= Kitchen[Knife]
assert  dull_paring  <= Kitchen[Knife]
```

Additionally, the combination is still a smaller domain than the domain of all kitchen knives:

```
assert  sharp_fillet | dull_paring <= Kitchen[Knife]
```

What is most important is that an action that needs something specific can avoid receiving an over-general type. For example, consider this action:

```
sharpen knife : Kitchen[Knife % Properties(exclude=['sharp'])
    -> Kitchen[Knife % Properties(include=['sharp'])
```

This rather intuitively swaps the property of not-being sharp for the property of being sharp. We can see how the subtyping relation allows the action to enforce this:

```
assert      dull_paring  <= Kitchen[Knife % Properties(exclude=['sharp'])]
assert not  sharp_fillet <= Kitchen[Knife % Properties(exclude=['sharp'])]
```

One consequence of this is that an unadorned type like `Kitchen[Knife]` is not known to be either sharp or dull (remember it is actually supertype of both of these).

```
# Can't substitute any-old knife for a dull one, some of them are sharp.
assert not  Kitchen[Knife] <= Kitchen[Knife % Properties(exclude=['sharp'])]
```

As a matter of practice, it would probably be easier for everyone if "sharpen knife" were to just re-sharpen the already-sharp knife.

Intersections

There is another kind of type known as the intersection type. Currently QIIME 2 implements this only in a very limited way. The idea is that you might have an instance that is simultanously many different types. For example, a *spork* is both a fork and a spoon (and good at neither).

Nonetheless, someday you might write something like this:

```
# This doesn't work yet
Spork = Fork & Spoon

assert  Spork <= Fork
assert  Spork <= Spoon
```

As you can see, the relationship is inverted from a union. Why bring this up, if the above isn't implemented? First, this syntax would be a convenient way to describe *compound artifacts*, where a lot of data is bundled up nicely in a single zip file. Second, this is how semantic properties work.

When you are dealing with multiple semantic properties, each property is *intersected* with the others, meaning that an artifact that has multiple properties associated with it is considered to have each one. This means these expressions are the same:

```
Knife % Properties(['fillet', 'sharp'])
# is the same as:
(Knife % Properties('fillet')) & (Knife % Properties('sharp'])
# if `&` was implemented
```

It also means that this is true:

```
assert  Knife % Properties(['fillet', 'sharp']) <= Knife % Properties(['fillet']) <=␣
↪Knife
```

The more information we add, the more specific our knife (and the smaller our domain).

3.1.5 File Formats and Directory Formats

Formats in QIIME 2 are on-disk representations of data. When associated with an *artifact class*, they define how data are stored in or read from the `data/` directory of *artifacts*.

QIIME 2 doesn't have much of an opinion on how data should be represented or stored, so long as it *can* be represented or stored in an *archive*.

File Formats

The simplest `Formats` are the `TextFileFormat` (`qiime2.plugin.TextFileFormat`) and the `BinaryFileFormat` (`qiime2.plugin.BinaryFileFormat`). These formats represent a single file with a fixed on-disk representation.

Validation

Both types of `FileFormat` support validation. This is (typically) a small bit of code that is run when initially loading a file from an *archive* that allows the framework to ensure that the the data contained within the *archive* at least *looks* like its declared *type*. This works very well for on-the-fly loading and saving, and goes a long way to preventing corrupt or invalid data from persisting. The one "gotcha" here is that in order to keep things quick, we typically recommend that "minimal" validation occurs over a limited subset of the file (e.g. the first 10 records in a FASTQ file). Because of this, formats allow for multiple levels of sniffing to be defined. As of this writing (March 2024) there currently there are two validation levels: `min` and `max`.

Here we provide an example of a `TextFileFormat` definition, with a focus on the `_validate_` function.

```
class IntSequenceFormat(TextFileFormat):
    """
    A sequence of integers stored on new lines in a file.
    To make validation more interesting, values in the list can be any integer as long
    as that integer is not equal to the previous value plus 3
    (i.e., `line[i] != (line[i-1]) + 3`).
    """
    def _validate_n_ints(self, n):
```

(continues on next page)

(continued from previous page)

```
        with self.open() as fh:
            previous_val = None
            for idx, line in enumerate(fh, 1):
                if n is not None and idx >= n:
                    # we have passed the min validation level,
                    # so bail out
                    break
                try:
                    val = int(line.rstrip('\n'))
                except (TypeError, ValueError):
                    raise ValidationError(
                        f"Line {idx} contains {val}, but must be an integer.")
                if previous_val is not None and previous_val + 3 == val:
                    raise ValidationError(
                        f"Value on line {idx} is 3 more than the value on "
                        f"line {idx-1}.")
                previous_val = val

    # `_validate_` is exposed through the public method `validate`.
    def _validate_(self, level):
        record_map = {'min': 5, 'max': None}
        self._validate_n_ints(record_map[level])

format_instance = IntSequenceFormat(temp_dir.name, mode='r')
format_instance.validate()  # Shouldn't error!
```

In the `IntSequenceFormat` example, when `validate` is called with `level='min'`, `_validate_` will check the first 5 records. Otherwise, when `level='max'`, it will check the entire file.

Astute observers might notice that the method defined in the `IntSequenceFormat` is called `_validate_`, but the method called on the `format_instance` was `validate`. This is because defining format validation is optional (although highly recommended!).

Warning

We consider skipping validation all together when defining formats to be a plugin development anti-pattern. For more information, see *Skipping format validation*.

Every format has a `validate` method available to interfaces (for performing ad-hoc validation). The framework will check for the presence of a `_validate_` method on the format in question, and if it exists it will include that method as part of more general validations that the framework will perform. The aim here is that the framework is capable of ensuring common basic patterns, like presence of required files, while the `_validate_` method is the place for the format developer to declare any special "business" logic necessary for ensuring the validity of their format.

Text File Formats

The `TextFileFormat` (`qiime2.plugin.TextFileFormat`) is for creating text-based formats (e.g. FASTQ, TSV, etc.). An example of one of these formats is the `DNAFASTAFormat`, used for storing FASTA data.

Binary File Formats

The `BinaryFileFormat` (`qiime2.plugin.BinaryFileFormat`) is for creating binary formats (e.g. BIOM, gzip, etc.). An example of one of these formats is the `FastqGzFormat`, a format for storing gzipped FASTQ files.

Directory Formats

While many formats can accurately be described using a single file, many formats exist that require the presence of more than one file present together as a set. QIIME 2 allows more than one `FileFormat` to be combined together as a `DirectoryFormat` (`qiime2.plugin.DirectoryFormat`).

Fixed Layouts

Some directory layouts can be accurately described with a fixed number of members. An example of this is the `EMPPairedEndDirFmt`. This directory format is always composed of three `FastqGzFormat` files: one for the forward reads (`forward.fastq.gz`), one for the reverse reads (`reverse.fastq.gz`), and one for the barcodes (`barcodes.fastq.gz`). The underlying `FastqGzFormat` is defined once — it doesn't need to know about the sematic difference between biological reads and barcode reads, unlike the `EMPPairedEndDirFmt` which must be able to differentiate these.

```
class EMPPairedEndDirFmt(model.DirectoryFormat):
    forward = model.File(r'forward.fastq.gz', format=FastqGzFormat)
    reverse = model.File(r'reverse.fastq.gz', format=FastqGzFormat)
    barcodes = model.File(r'barcodes.fastq.gz', format=FastqGzFormat)
```

The component files of this `DirectoryFormat` are defined using the `File` (`qiime2.plugin.model.File`) class.

Variable Layouts

While some layouts are accurately described with a fixed set of members, others are highly variable, preventing formats from accurately knowing how many files to expect in its *payload*. An example of this kind of format are any of the demultiplexed file formats — when sequences are demultiplexed there is one (or two) files per sample, but how many samples are there? One study might have 5 samples, while another has 5000. For these situations the `DirectoryFormat` (`qiime2.plugin.DirectoryFormat`) can be configured to watch for set pattern of filenames present in its *payload*. An example of this is the `CasavaOneEightSingleLanePerSampleDirFmt` class, which stores demultiplexed sequence data in files named with a pattern used by Illumina's Casava v1.8 software.

```
class CasavaOneEightSingleLanePerSampleDirFmt(model.DirectoryFormat):
    sequences = model.FileCollection(
        r'.+_.+_L[0-9][0-9][0-9]_R[12]_001\.fastq\.gz',
        format=FastqGzFormat)

    @sequences.set_path_maker
    def sequences_path_maker(self, sample_id, barcode_id, lane_number,
```

(continues on next page)

(continued from previous page)

```
                            read_number):
    return '%s_%s_L%03d_R%d_001.fastq.gz' % (sample_id, barcode_id,
                                             lane_number, read_number)
```

Single File Directory Formats

Currently QIIME 2 requires that all formats registered to an *artifact class* be a directory format. For those cases, there exists a factory for quickly constructing directory layouts that contain *only a single file*. This requirement might be removed in the future, but for now it is a necessary evil (and also isn't too much extra work for format developers).

```
DNASequencesDirectoryFormat = model.SingleFileDirectoryFormat(
    'DNASequencesDirectoryFormat', 'dna-sequences.fasta', DNAFASTAFormat)
```

Associating Formats with a Type

Formats on their own aren't of much use. It is only once they are associated with a *semantic type* to define an *artifact class* that things become interesting. Artifact classes define the data that can be provided as input or generated as output by QIIME 2 `Actions`. An example of this can be seen in the registration of the `SampleData[PairedEndSequencesWithQuality]` artifact class.

3.1.6 Decentralized retrospective provenance tracking

QIIME 2 provides automatic, integrated, and decentralized tracking of analysis metadata, including information about the host system, the computing environment, Actions performed, parameters passed, primary sources cited, and more [7]. We describe all of this information about *how an analysis was performed to produce a result* as the result's **provenance**.

The notion of a QIIME 2 *Result* is central here. Whenever an *Action* is performed on some data with QIIME 2, the framework captures relevant metadata about the action and environment in a *Result* object, which is backed by an *Archive*. If that Result is saved as a `.qza` or `.qzv`, the captured provenance data persists within that zip archive. *Provenance Goes In /provenance/* contains a detailed discussion of the file structure which holds provenance metadata.

This is a **decentralized** approach to provenance capture: every QIIME 2 Result's provenance is packaged with the Result itself. This prevents disassociation of a Result with its provenance, for example when a simpler analysis outcome (e.g., a `.png` file containing a graph) is emailed to a colleague. It also prevents accidental mis-association of a Result with the wrong provenance, or with inaccurate or outdated provenance records. For example, with *centralized approaches to provenance tracking*, like saving scripts or computational notebooks, those records may be updated. If a result was generated prior to a script update, and the result is inadvertantly not updated, a subsequent user or viewer of that result may have inaccurate information about how it was created. In contrast, the provenance captured within a QIIME 2 Archive will always describe the way that Archive was actually created.

QIIME 2's approach is also a form of **retrospective** provenance capture. With a *prospective* provenance capture approach, a script, work plan, or computational notebook are used to document what will be done. Referring to the example from the previous paragraph, if the script is updated but not run to recreate a result of interest (e.g., because an older version of the result was already shared with a colleague, and they forget to download the new version) the script won't accurately represent the provenance of that result, and it will be hard or impossible to discover that. Retrospective provenance, on the other hand, is captured when the result is generated, and therefore documents what *actually occurred*, supporting a more reliable recording of an analysis.

> **Note**
>
> *When and if* `Results` are saved as `.qza` or `.qzv` ZIP archives is dependent on the interface that is being used to run QIIME 2. For example, results are saved automatically by `q2cli` (QIIME 2's command line interface) every time a user runs a command. Results must be saved manually by users of the Python 3 API, if they want a `.qza` or `.qzv` ZIP archives (but importantly, provenance will be complete even if some intermediate results are not saved). This allows API users to reduce I/O, and keeps things simpler for CLI users.

Why Capture Provenance Data?

QIIME 2's provenance capture gives users, developers, support teams, manuscript reviewers, and research consummers valuable tools for documentation, study validation and reproduction, analysis transparancy, software maintenance and repair, and technical support.

Among the benefits of this model are:

- Analyses are fully reproducible.
- Analyses self-document, reducing the reliance on the (possibly incomplete or incomprehensible) notes of the individual who ran the analysis. For example, q2view produces directed provenance graphs, like this one, that allow any consummer of that result to understand exactly how it was created.
- QIIME 2 Artifacts bring their citations with them.
- Methods-section text could theoretically be generated from a collection of QIIME 2 Artifacts. (If you're interested in contributing that functionality, please feel free to get in touch - large language models (LLMs) should make this easier than ever!)
- Analyses are automatically replay-able [7], meaning that you can generate new code from existing results.
- In the unlikely event of a data integrity bug, problematic combinations of hardware, environment, `Actions`, and parameters can be investigated effectively by users, developers, or technical support providers. Impacted results can be programatically identified, and could be programatically correctable in some cases.
- By capturing provenance metadata at the level of Actions and Results, QIIME 2 provenance is both host- and interface-agnostic. In other words, a QIIME 2 analysis can be performed across various host systems, using whatever interfaces the user prefers, without compromising the validity of the analysis or the provenance. The `Result` of every step in the analysis contains its own unique history.

What Provenance Data is Captured?

In order to focus on provenance data, we will consider a relatively simple example QIIME 2 Archive structure, with limited non-provenance content. In Fig. 3.5 the outer *UUID* directory of this *Artifact* holds the data it produced in a `data` directory (see *Data Goes In /data/*), and a few "clerical" files (see *Anatomy of an Archive*). Here we focus on the `provenance/` directory.

> **Note**
>
> Importantly, the *data* from the parent Results are not included in the current Result's provenance: only their provenance *metadata*. Including their data would result in massive file sizes and duplication of data.

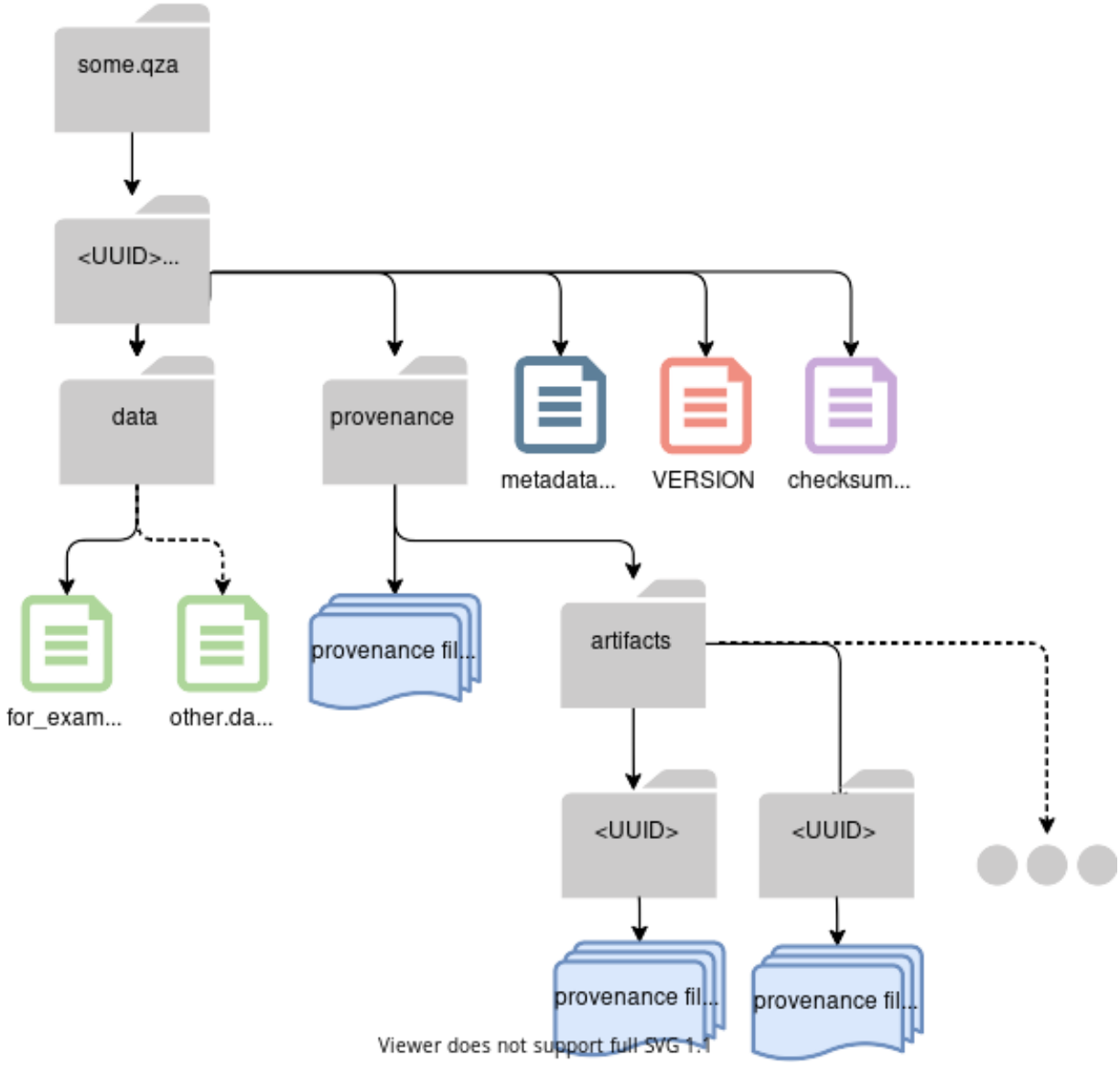


Fig. 3.5: Simplified representation of the files within one Archive.

In Fig. 3.5, we use a blue "multiple-files" icon to represent the collection of provenance data associated with one single QIIME 2 action. When this icon appears directly within `provenance/` the files describe the "current" *Result*. All remaining icons appear within the `artifacts/` subdirectory. These file collections describe all "parent" Results used in the creation of the current Result, and are housed in directories named with their respective UUIDs.

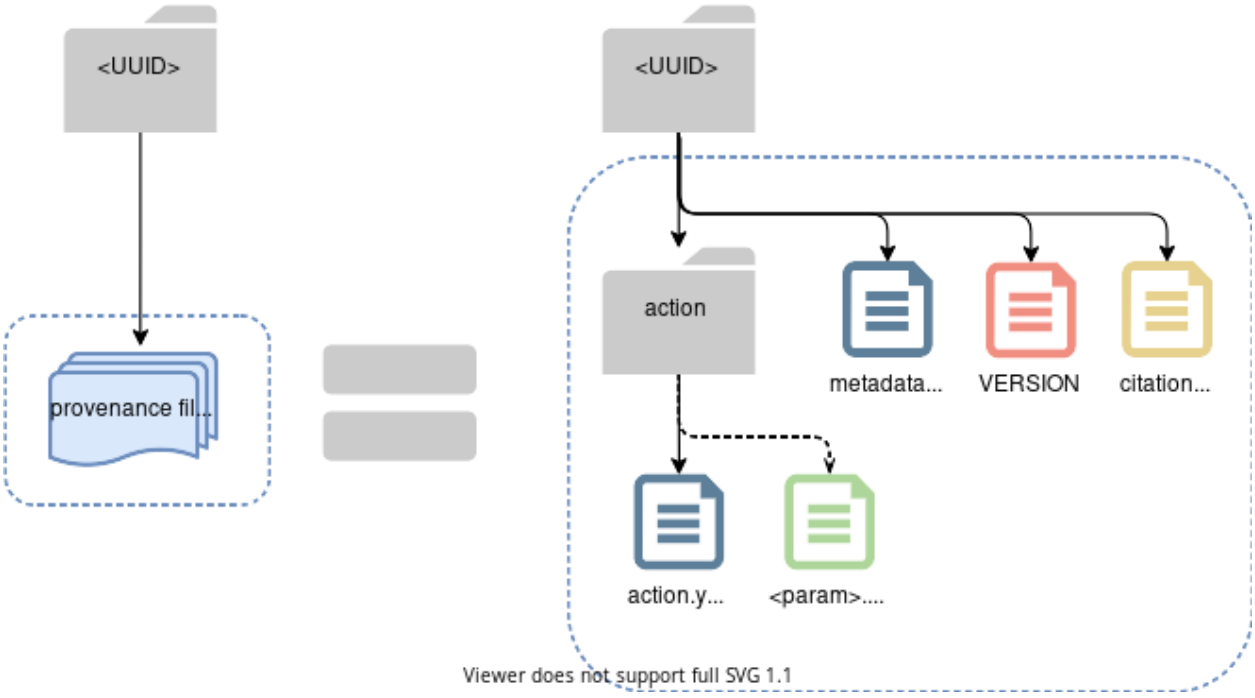


Fig. 3.6: A detail indicating how we abbreviate one action's provenance records in Fig. 3.5.

With the exception of the current Result (whose provenance lives in `provenance/`, every Action is captured in a directory titled with the Action's *UUID* Fig. 3.6.

That directory contains:

- `VERSION` (see *Rules for identifying an archive*)
- `metadata.yaml` (see *The Most Important File: metadata.yaml*)
- `citations.bib`: All citations related to the run Action, in bibtex format. (This includes "passthrough" citations like those registered to transformers, regardless of the plugin where they are registered.)
- `action/action.yaml`: A YAML description of the Action and its environment (i.e., the stuff we care most about).
- `action/metadata.tsv` or other data files (optional): Data captured to provide additional Action context.

The `action.yaml` file

Here we'll do a deep dive into the contents of a sample Visualization's `action.yaml`. The `action.yaml` files are broken into three top-level sections, in this order:

- `execution`: the Execution ID and runtime of the Action that created this Result
- `action`: Action type, plugin, action, inputs, parameters, etc.
- `environment`: a description of the system and the Python environment where this action was executed

The sample Visualization we're referring to can be viewed here. That link will open directly to the *Provenance* tab where you can click on the bottom square in that provenance graph (not the circle within the square!) to cross-reference the information provided here.

The `execution` block

High-level information about this action and its run time.

```
execution:
    uuid: 3611a0c1-e5c5-4308-ac92-ebb5968ebafb
    runtime:
        start: 2021-04-21T14:42:16.469998-07:00
        end: 2021-04-21T14:42:21.080381-07:00
        duration: 4 seconds, and 610383 microseconds
```

- Datetimes are formatted as ISO 8601 timestamps.
- The `uuid` field captured here is a UUID V4 representing this *Execution*, and *not the Result* it produced.

Unique IDs

There are many elements of provenance that require unique IDs, to help us keep track of different aspects of an analysis. Archive provenance has separate Result and Execution IDs (the UUIDs in `metadata.yaml` and `action.yaml` respectively). This allows us to manage the common case where one Action produces multiple Results.

Artifacts produced by QIIME 2 Pipelines have an additional `alias-of` UUID, allowing interfaces to display provenance in terms of Pipelines (rather than displaying all of the pipeline's "nested" inner Actions). This enables a view of provenance that better reflects the user experience of pipelines, displaying them as single blocks, rather than as the full chain of inner Actions which the user generally does not specify directly.

Terminal pipeline Results are redundant "aliases" of "real" Results nested within the pipeline. The `alias-of` UUID in the terminal/"alias" Result points to this "real" inner result. Further details are provided in *Pipeline Provenance*.

The `unweighted_unifrac_emperor.qzv` described here has three different IDs:

- The Result UUID, in `metadata.yaml`, which is unique to this Result.
- The Execution UUID, in `action.yaml`'s `execution` block, which is unique to this Action's current execution, and present in all Archives produced during this Actions's current execution. All Results from a given run of an `Action` (`core-metrics-phylogenetic`, in our current example) share this ID.
- The `alias-of` UUID, in `action.yaml`'s `action` block, which is the Result UUID of the "inner" Visualization created during a `Pipeline` execution that is aliased by this Result. (Remember that a *Pipeline* is a type of *Action*.)

We chose to use v4 UUIDs for our unique IDs, but there is nothing special about them that couldn't be handled by a different unique identifier scheme. They're just IDs.

The `action` block

Details about the action, including action and plugin names, inputs and parameters

```
action:
    type: pipeline
    plugin: !ref 'environment:plugins:diversity'
    action: core_metrics_phylogenetic
    inputs:
    -   table: 34b07e56-27a5-4f03-ae57-ff427b50aaa1
    -   phylogeny: a10d5d44-62c7-4322-afbe-c9811bcaa3e6
    parameters:
```

(continues on next page)

(continued from previous page)

```
    - sampling_depth: 1103
    - metadata: !metadata 'metadata.tsv'
    - n_jobs_or_threads: 1
  output-name: unweighted_unifrac_emperor
  alias-of: 2adb9f00-a692-411d-8dd3-a6d07fc80a01
```

- The `type` field describes the *type of the Action*: a *Method*, *Visualizer*, or *Pipeline*.
- The `plugin` field describes the plugin which registered the Action, details about which can be found in `action.yaml`'s `environment:plugins` section. `!ref` is a custom YAML tag defined here. Generally, these custom tags provide a way to express a structure not easily described by basic YAML.
- `inputs` lists the registered names of all *inputs* to the Action, as well as the UUIDs of the passed inputs. Note the distinction between inputs and parameters.
- `parameters` lists registered parameter names, and the user-passed (or selected default) values.
- `output-name` is the name assigned to this Action's output *at registration*, which can be useful when determining which of an Action's multiple outputs a file represents. (This does not capture the user-passed filename, because file names are interface specific and may not always be relevent - for example, when using the Python API, files may not exist for inputs at the time of action execution.)
- `alias-of` is an optional field, present if the Action is the terminal result of a QIIME 2 *Pipeline*. This value is the UUID of the "inner" result which this pipeline result aliases.

The `environment` block

The environment block is a description of the computing environment in which this Action was run. It is not uncommon for QIIME 2 analyses to be run through multiple user interfaces, on multiple systems. For this reason, per-Action logging of system characteristics is useful.

- `platform`: The operating system and version used to run the Action. For virtual machines (VMs), this is the client Operating System.
- `python`: python version details, as captured by `sys.version`.
- `framework`: Details about the QIIME 2 version used to perform this Action.
- `plugin`: The QIIME 2 plugin, its version, and registered source web site.
- `python-packages`: Package names and version numbers for all packages in the global `working_set` of the active Python distribution, as collected by pkg_resources.

Warning

QIIME 2 currently captures only Python package information in the environment block, and therefore doesn't collect complete information about the environment. We plan to expand this to include all relevant packages in the environment regardless of language. See GitHub issue #587 if you are interested in contributing to this effort.

```
environment:
  platform: macosx-10.9-x86_64
  python: |-
      3.8.8 | packaged by conda-forge | (default, Feb 20 2021, 16:12:38)
      [Clang 11.0.1 ]
  framework:
```

(continues on next page)

(continued from previous page)

```
        version: 2021.4.0
        website: https://qiime2.org
        citations:
        - !cite 'framework|qiime2:2021.4.0|0'
    plugins:
        diversity:
            version: 2021.4.0
            website: https://github.com/qiime2/q2-diversity
    python-packages:
        zipp: 3.4.1
        xopen: 1.1.0

        ...

        q2-dada2: 2021.4.0
        q2-composition: 2021.4.0
        q2-alignment: 2021.4.0

        ...

        alabaster: 0.7.12
```

Pipeline Provenance

As discussed in *the Unique IDs note above*, *Pipeline* provenance is more complex than the provenance of other Actions. Most Pipelines wrap one or more *Methods* or *Visualizers*. Pipeline users are often concerned primarily with ease of use and interpretation, rather than the fine-grained details of the Actions "nested" within the Pipeline. With this in mind, provenance viewing interfaces may choose to abstract away nested Actions, displaying only the Pipeline used to run those Actions.

QIIME 2 View works in this way, and the simple graph shown in our provenance example is the result of hiding ten nested Actions from view behind the two pipelines that use them. The user sees only five of the fifteen captured Results, each of which they ran themselves. Because the bottom two are pipelines, this view simply but completely represents the provenance of the Archive being viewed.

This is possible because QIIME 2 captures redundant "terminal" pipeline outputs that alias the "real" pipeline outputs nested in provenance. These terminal outputs are of the same *artifact class* as the Results they alias, but capture provenance details at the scope of the Pipeline, rather than at the scope of the Method or Visualizer they alias.

Pipeline provenance example

This Artifact's root-level `metadata.yaml`, which can be accessed by clicking the circle within the bottom box of our example provenance DAG, tells us it's a Visualization:

```
uuid: 87058ae3-e168-4e2f-a416-81b130d538c3
type: Visualization
format: null
```

Next, clicking on the box (rather than the circle within the box) lets us view the `action.yaml` contents. The root-level `action.yaml` file tells us that this is the (terminal) result of a Pipeline (and not of a Visualizer). The `inputs` are the UUIDs passed by the user to the Pipeline. The `parameters`, too, are linked to the Pipeline, and not to the nested Visualizer. Finally, we see the `alias-of` key, whose value is the UUID of the nested "real" Visualization aliased by this terminal output.

```
action:
    type: pipeline
    plugin: !ref 'environment:plugins:diversity'
    action: core_metrics_phylogenetic
    inputs:
    -   table: 34b07e56-27a5-4f03-ae57-ff427b50aaa1
    -   phylogeny: a10d5d44-62c7-4322-afbe-c9811bcaa3e6
    parameters:
    -   sampling_depth: 1103
    -   metadata: !metadata 'metadata.tsv'
    -   n_jobs_or_threads: 1
    output-name: unweighted_unifrac_emperor
    alias-of: 2adb9f00-a692-411d-8dd3-a6d07fc80a01
```

If we were to extract this `.qzv` (e.g., by opening it with an unzip utility), we could use this `alias-of` UUID to drill down into `provenance/artifacts/2adb9.../` to find the `action.yaml` of the aliased Visualization. There, the action type would be a `visualizer`, and the `inputs` and `parameters` would be those passed to the *Visualizer* within the Pipeline. Notably, neither the Visualizer node shown below, nor its PCoA input, are visible in the provenance graph linked above, because they are neither inputs to nor terminal outputs from the containing `Pipeline`.

```
action:
    type: visualizer
    plugin: !ref 'environment:plugins:emperor'
    action: plot
    inputs:
    -   pcoa: 93224813-ed5d-42b5-a983-cd4015db31da
    parameters:
    -   metadata: !metadata 'metadata.tsv'
    -   custom_axes: null
    -   ignore_missing_samples: false
    -   ignore_pcoa_features: false
    output-name: visualization
```

Pipeline provenance take-aways

- All Results used in producing an Archive are captured in that Archive's provenance, including "inner" pipeline results. Each Result has its own normal provenance directory.
- "Terminal" pipeline outputs are aliases, mirroring inner Actions.
- Different terminal outputs from the same pipeline will (generally) have different alias-of's, *because they are aliasing different inner nodes*. For example, a Visualization aliases a Visualization, while the terminal PCoA results point to the inner PCoA results, and these inner Results have different UUIDs.
- Pipelines may wrap pipelines, so an arbitrary number of levels of nesting and aliasing are possible. Tools that aim to work with nested provenance will likely have to traverse from the terminal node. The traversal algorithm is discussed here. (Want to port that content over? Get in touch!).
- Inner "nested" pipeline Results are normal Results, and may be used as inputs to other nested Actions, or may be aliased by terminal pipeline results. The only "special" thing happening with pipelines is the aliasing of terminal pipeline Results.

3.1.7 Garbage Collection

This page does not offer any particularly deep or useful answers, but seeks to serve as a warning that this part of QIIME 2 is *difficult* to understand. If you think you have a better or more robust way to manage resource allocation, we would love to hear from you!

Because QIIME 2 *Artifacts* are really directory structures, synchronizing filesystem state with memory can be difficult. It is further excacerbated by the lifetimes of the objects being unkown. For example viewing an artifact can produce anything from a in-memory object to an entirely new directory structure.

Currently, QIIME 2 chooses to tie the filesystem state directly with an objects lifetime in memory. This means that instead of allocating a location and automatically destroying it with a static lifetime (e.g. RAII/context manager) it is handled dynamically. It is up to the Python garbage collector to invoke a destructor that cleans the filesystem when appropriate. This pushes the issue off from managing lifetimes, to ensuring that destruction occurs and that the data *should* be destroyed (e.g. it is not user-input).

The objects responsible for managing filesystem paths (and destroying them) are located in `qiime2.core.path`.

Additionally it is not always the case that the garbage collecter *will* call a destructor. For example in a multiprocessing context, `sys._exit` is called instead of `sys.exit`, meaning that any filesystem objects created in the child-process will not be cleaned up. Exceptions are another way in which normal cleanup can become confounded. Fortunately there is an additional object `qiime2.sdk.context:Context` which can be used to juggle this information and can provide a context-manager for when a static lifetime is known.

3.1.8 Metaprogramming

The framework uses a significant amount of runtime metaprogramming in which properties and methods are swapped out or constructed at the last minute based on plugin information. This list shows some of the aspects of the Python language that are used to achieve this:

- decorators (used everywhere, `qiime2.sdk.action:Action` is really just a decorator-object)
- descriptor protocol (used in `qiime2.core.util:LateBindingAttribute`)
- import hooks (used by Artifact API)
- metaclasses (used by `qiime2.plugin.model.directory_format`)
- eval (`qiime2.skd.util:parse_type` and `decorator` package for signature rewriting)

3.2 References

- *Archive versions*

3.2.1 Archive versions

The structure of QIIME 2 *archives* has evolved as QIIME 2 has been developed. This section describes each historical version of the QIIME 2 Archive format, and may be useful to interface developers whose code depends on guarantees made by that format (source code).

Version-agnostic format guarantees

Though there is significant variability in the format of QIIME 2 Archives across archive versions, all archive versions share some common traits.

These shared characteristics, defined in the `_Archive` class in `qiime2/core/archive/archiver.py`, must be consistent across all formats over time as they allow archive versions to be checked, and archives with different formats to be dispatched to the appropriate version-specific parsers.

All QIIME 2 Archives have:

- a directory named with the Archive UUID, directly under the archive root at `/<UUID>/`, and
- a file `/<UUID>/VERSION` within that directory, formatted as shown below.

Fig. 3.7 illustrates the Archive file system.

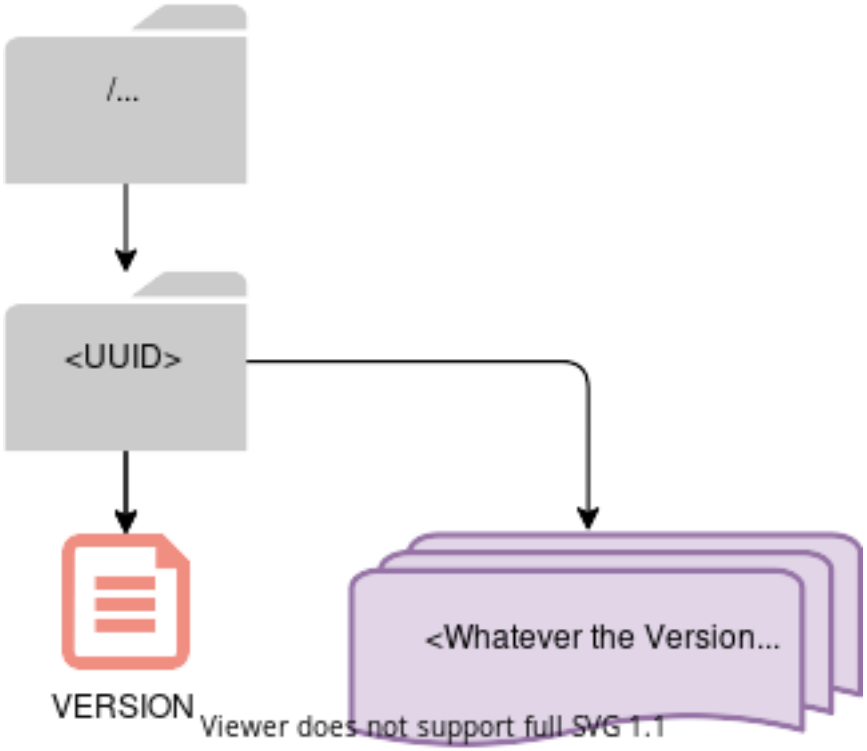


Fig. 3.7: Box and arrow diagram of the guaranteed components of an archive.

The VERSION file format is described *above*.

Archive Version 0

Version 0 was the original QIIME 2 Archive format, and there aren't many V0 Archives "in the wild". V0 Archives were produced by alpha versions of the QIIME 2 framework, and were superseded in framework version 2.0.6 on 24 October 2016.

- *Result* data files are written in the directory `/<UUID>/data`
- Result UUID, semantic type, and format information are saved in `/<UUID>/metadata.yaml`.
- The `ArchiveFormat` class in `v0.py` offers convenience methods for loading and parsing `metadata.yaml` files.

Fig. 3.8 illustrates the format of a Version 0 Archive.

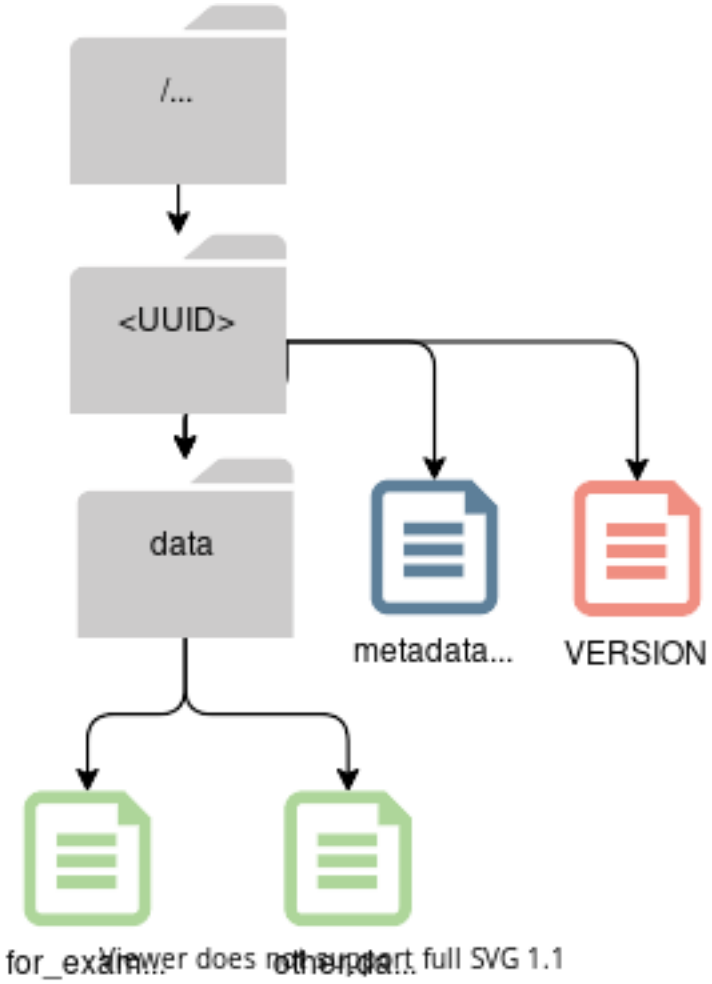


Fig. 3.8: Box and arrow diagram of a v0 archive.

Archive Version 1

Released in QIIME 2 version 2.0.6, commit `bdc8aed`, Version 1 Archives introduce decentralized provenance tracking to QIIME 2. ArchiveFormat V1 inherits all traits of v0, modifying its `__init__()` and `write()` methods only to add provenance capture.

> **Note**
>
> All `ArchiveFormat` versions subclass their predecessor. For example, the `ArchiveFormat` in `v1.py` inherits from the `ArchiveFormat` in `v0.py`, etc. This makes it easier for humans to interpret the version history.

Provenance data is stored in the directory `/<UUID>/provenance/`. Specifically, `metadata.yaml`, `action.yaml` and `VERSION` files are captured for the current Result and each of its ancestors. Each Result's `action.yaml` file and associated data artifacts (e.g. sample metadata) are stored in an `action` directory alongside that Result's `VERSION` and `metadata.yaml`. Considered together, we can describe these as "provenance files". This structure is illustrated in Fig. 3.9.

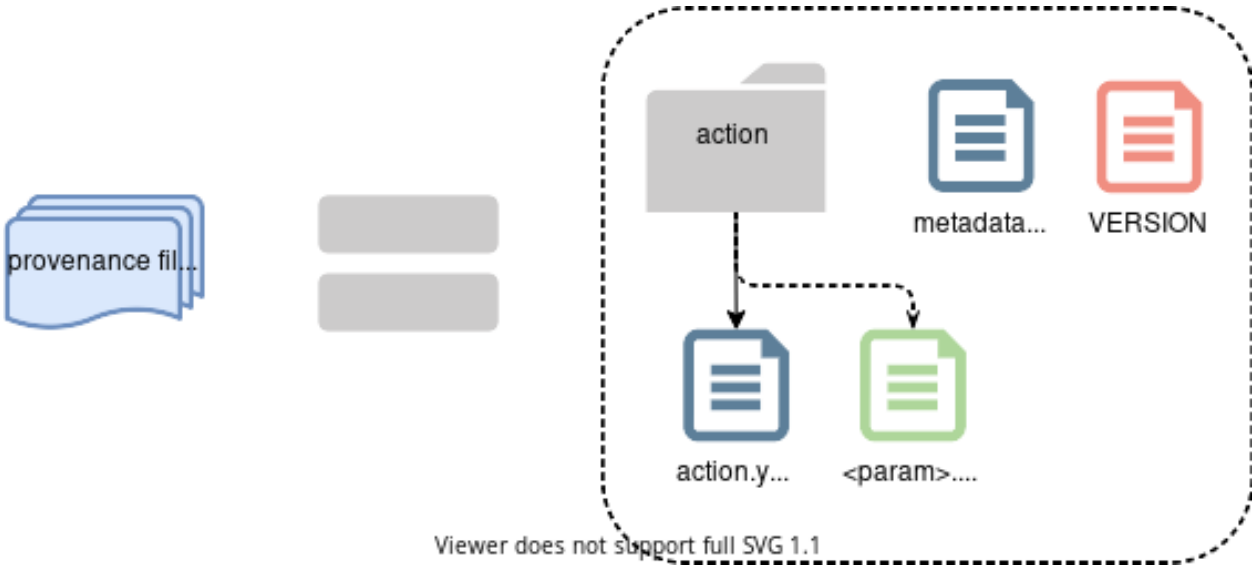


Fig. 3.9: Box and arrow diagram of the provenance files in a v1 archive. The blue "multiple files" icon represents all provenance files associated with a single action (e.g. the current action, or one of its ancestors).

The structure of V1 Archives as a whole is illustrated in Fig. 3.10.

Note

V0 Archives do not capture provenance data. As a result, if a V0 artifact is an ancestor to a V1 (or greater) artifact, it is possible for the `action.yaml` to list Artifact UUIDs which are not present in the `artifacts` directory.

Archive Version 2

Released across QIIME 2 versions 2017.9 (changelog) and 2017.10 (changelog), the directory structure of this format is identical to v1, but the `action.yaml` file has changed.

In commit `4389a0b` the Version 2 `ArchiveFormat` adds an `output-name` key to the `action` section of `action.yaml` (unless the action type is `import`), assigning it the output name registered to the relevant action. Prior to this change, if one action returned multiple artifacts of the same *artifact class*, it was not possible to differentiate between them using provenance alone.

In commit `e072706`, it adds provenance support for *Pipelines*, adding the `alias-of` key to the `action` section of `action.yaml`.

Archive Version 3

Released in QIIME 2 version 2017.12 (changelog), commit `684b8b7`, the directory structure of this format is identical to v1 and v2.

With this release, QIIME 2 Actions are able to take variadic arguments, allowing users to pass collections of Artifacts (`List`s and `Set`s). A YAML representer has been added so that `action.yaml` can represent `Set`s of Artifact inputs. These will show up in `action.yaml` as custom `!set` tags.

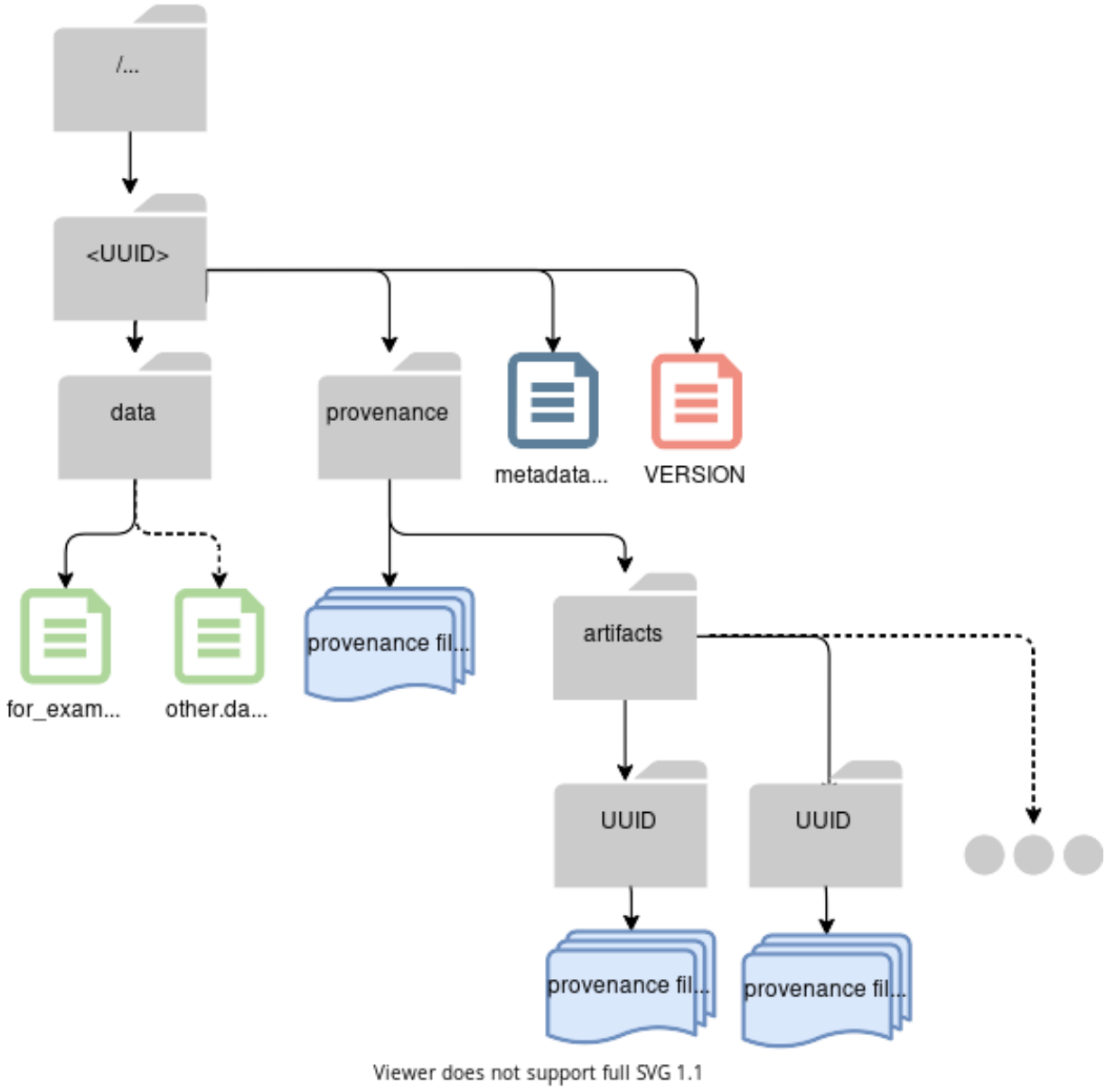


Fig. 3.10: Box and arrow diagram of a v1 archive.
Provenance files for the current Result are stored in `/<UUID>/provenance/`. Provenance files for each ancestor Result are stored in directory at `/<root_UUID>/provenance/artifacts/<ancestor_UUID>/`.

Archive Version 4

Released in QIIME 2 version 2018.4 (changelog), commit `00a294c`, this format adds citations to the directory format, adds a `transformers` section to `action.yaml`, and aligns the structure of `environment:framework` (also in `action.yaml`) to match the structure of `environment::plugins::<some_plugin>`.

Whenever an `Action` is run, its registered citations are captured. When saved, they are written to a `citations.bib` file inside the Archive's `provenance` directory. Citations for all of the current Result's ancestors are stored in their respective directories (e.g. `/<root_UUID>/provenance/artifacts/<ancestor_UUID>/citations.bib`).

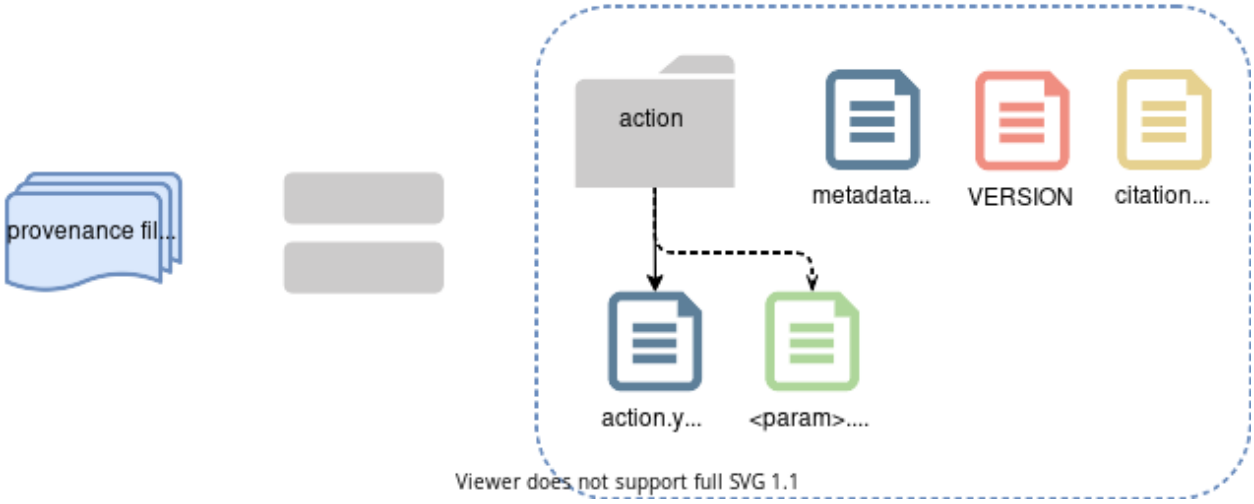


Fig. 3.11: Box and arrow diagram of the provenance files in a v4 archive.

The overall directory structure remains identical to a v1 archive (Fig. 3.9).

Result-specific citation tags are also written to the `transformers` and `environment` sections of the `action.yaml` files, for the current Result and for all ancestors with registered citations. A new custom `!cite '<citation key>'` tag is use to support this in YAML.

A `transformers` section is added between the `action` and `environment` sections of `action.yaml`. Because Pipelines do not use transformers, transformers will be recorded only for *Methods*, *Visualizers*, and when importing data (such as with `qiime tools import`). It looks like the following:

```
transformers:
inputs:
    demultiplexed_seqs:
    -   from: SingleLanePerSamplePairedEndFastqDirFmt
        to: SingleLanePerSamplePairedEndFastqDirFmt
output:
-   from: q2_types.feature_data._transformer:DNAIterator
    to: DNASequencesDirectoryFormat
    plugin: !ref 'environment:plugins:types'
```

`environment::framework` was previously only a version string, and is now structured identically to each plugin action's `software_entry`, with version, website, and citation sections:

```
framework:
    version: 2019.10.0
```

(continues on next page)

(continued from previous page)

```
    website: https://qiime2.org
    citations:
    - !cite 'framework|qiime2:2019.10.0|0'
plugins:
    fragment-insertion:
        version: 2019.10.0
        website: https://github.com/qiime2/q2-fragment-insertion
        citations:
        - !cite 'plugin|fragment-insertion:2019.10.0|0'
        ...
```

Archive Version 5

Released in QIIME 2 version 2018.11 (changelog) commit `f95f324`, this format version adds archive checksums to the directory structure.

A new, md5sum-formatted checksum file has been added at `/<root_UUID>/checksums.md5`, with one md5sum and one filename on each line. For a more detailed specification, see the QIIME 2 Pull Request #414.

`checksums.md` looks like the following:

```
5a7118c14fd1bacc957ddf01e61491b7  VERSION
333fd63a2b4a102e58e364f37cd98b74  metadata.yaml
4373b96f26689f78889caeb1fbb94090  data/faith_pd-cat1.jsonp

...

7a40cff7855daffa28d4082194bdf60e  provenance/artifacts/f6105891-2c00-4886-b733-
↪6dada99d0c81/metadata.yaml
ae0d0e26da5b84a6c0722148789c51e0  provenance/artifacts/f6105891-2c00-4886-b733-
↪6dada99d0c81/action/action.yaml
```

Fig. 3.12 illustrates the V5 Archive.

Archive Version 6

Full documentation for Archive Version 6 is pending. The source code can be found here.

Archive Version 7

Archive Version 7 development is currently being planned.

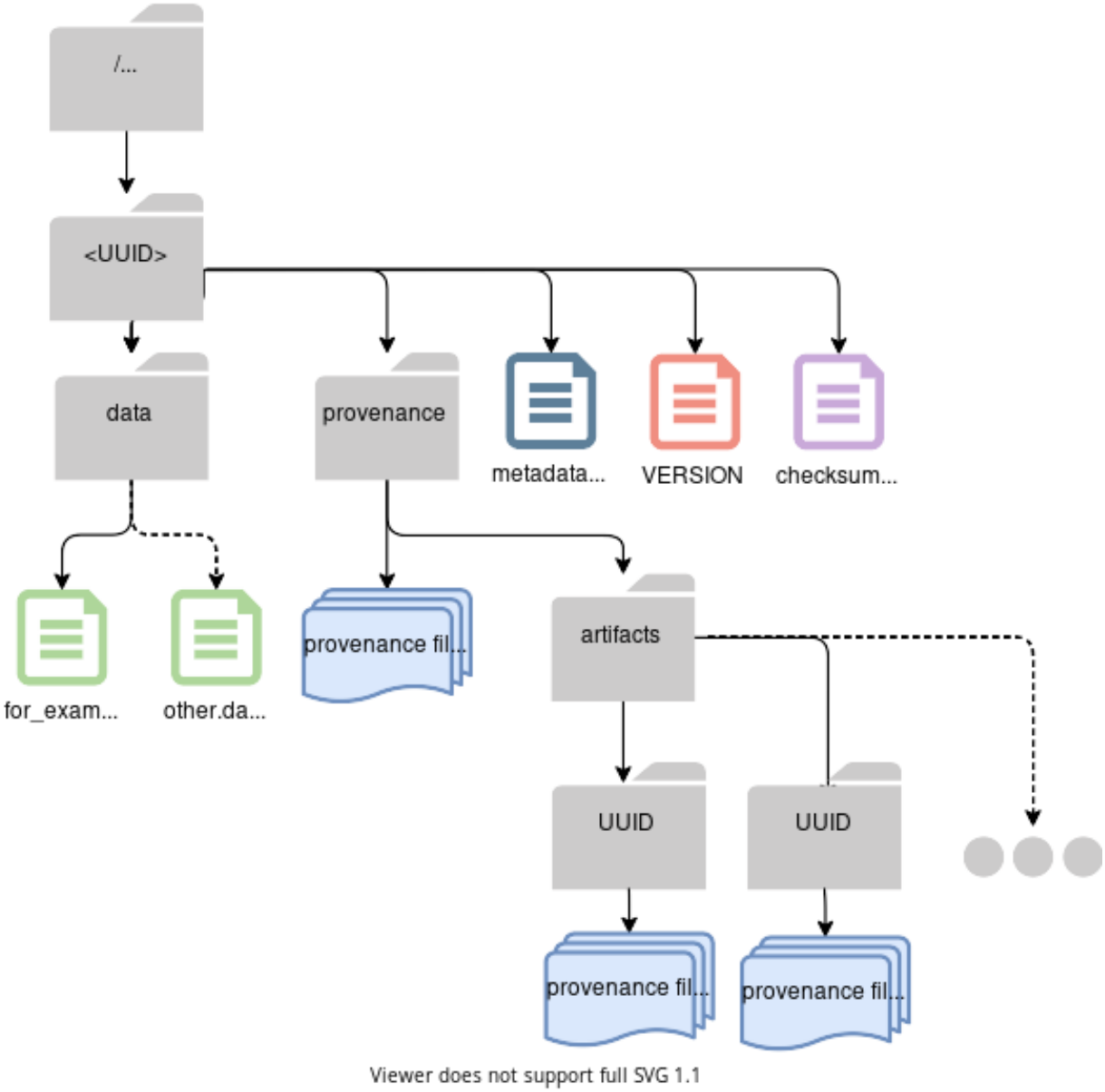


Fig. 3.12: Box and arrow diagram of a v5 archive.

Part IV

Documentation

CHAPTER

FOUR

DOCS DEVELOPMENT

4.1 User documentation

As of the time of this writing (12 January 2023) the QIIME 2 user documentation is in a state of transition. This document presents ideas about where we're planning to go with the user documentation in the future and then covers how to contribute to the current user documentation (https://docs.qiime2.org)

4.1.1 Plans for refactoring of user documentation

We are moving from a single source of documentation (the current content at https://docs.qiime2.org) to resources for **cross-distribution QIIME 2 documentation**, **within-distribution QIIME 2 reference material**, and **data set specific usage documentation**.

We expect that the **cross-distribution user documentation** will cover topics including:

- building and use a cache, which will expand on the content here
- parallel support, which will expand on the *content here*
- provenance replay/viewing, which will expand on the content here
- using the interfaces
- recycling old results
- installing old versions of QIIME 2 (why, why not, and how)
- discussion of distributions (why are there multiple? how to deal with that)
- importing/exporting
- viewing .qzvs

Distribution specific reference material will likely be the equivalent of our current plugin/action pages. These will render all usage examples associated with actions as well, to avoid the need for things like the "filtering tutorial" (which isn't really a tutorial, but rather a list of different approaches for filtering data). Ideally generation of distribution-specific reference documentation will be fully automated, such that these docs can be built from any QIIME 2 environment using the `PluginManager`.

Usage tutorials, like the Moving Pictures tutorial, will be dataset specific and may cross distributions (like the Cancer Microbiome Intervention Tutorial). This will facilitate the transition from replayed provenance to tutorial, ideally helping to blur the line between replayed provenance and a tutorial over time. To support this, we will likely add a usage driver

that generates usage driver source code (for example, select `View Source (qiime2.sdk)` from a multi-interface tutorial, so this can template documentation.

4.1.2 Contributing to the current user documentation

> **Warning**
>
> These instructions are a little sparse at the moment and don't cover things like working on your own fork or branch of the documentation. We do recommend forking and working on your own change-specific branch.

First, install the most recent release version of the QIIME 2 Amplicon Distribution, or *create a QIIME 2 Amplicon Distribution development environment*. Switch to that new environment.

Then, clone the QIIME 2 User documentation repository and install its requirements.

```
git clone https://github.com/qiime2/docs.git
cd docs
pip install -r requirements.txt
```

Build the documentation in `preview` mode to confirm that the build works before making your changes.

> **Note**
>
> Building the documentation in `preview` mode will avoid running all of the QIIME 2 steps covered in the tutorials during the build of the documentation, which will be vastly faster than doing a complete build (i.e., `make html`) of the documentation.

```
make preview
```

Make your edits, and re-build and view them. Iterate on this process until you're done.

```
make preview
cd build/preview && python -m http.server ; cd -
```

The above command will launch a web server on your computer. Open http://localhost:8000/ in your browser to view the documentation as hosted on that local web server.

When you're ready, submit a pull request in the usual way.

4.2 Developer documentation

Developing with QIIME 2 is being authored by Greg Caporaso and Evan Bolyen.

Contributions from others are welcomed and acknowledged via the project's GitHub contributors page in *Acknowledgements*. At the moment, while we're still laying the groundwork, we're accepting only specific contributions.

If you have suggestions or feedback we'd love to hear from you.

4.2.1 Finding docstring sources

For API documentation, the source of documentation for code entities will be defined by their respective docstrings. In many cases, the linkcode extension will provide a link (via `[source]`) to the source code of these entities. However in certain cases, (such as plugin types), autodoc can only find docstrings at the provided module path (and linkcode is unable to resolve a source). This means that these docstrings will be found in `qiime2/plugin/__init__.py` (or wherever the module provided to autodoc indicates).

Part V

Continuous Integration

CHAPTER

FIVE

DISTRIBUTION DEVELOPMENT

Note

This Part of DWQ2 is in early development, and currently provides references to relevant content in other Parts of *Developing with QIIME 2*. We're open to feedback on the ideas presented here, and some technical details may change as the ideas develop.

You don't need approval from existing QIIME 2 developers to create and distribute your own tools, such as plugins or interfaces, that build on QIIME 2. This is the beauty of *plugin-based, interface-agnostic architecture*: new functionality or new ways to use that functionality can be introduced by anyone, removing bottlenecks or gatekeepers between developers and users. Customized QIIME 2 distributions can also be created and shared by anyone.

As of this writing (23 April 2024) we are redesigning our approach to helping you share and promote your QIIME 2-based tools. *Distribute plugins on GitHub* provides an example of how you can currently share your plugin with users, and we are actively working to expand that functionality to make it more useful for developers and users. We also discuss more general ideas on sharing QIIME 2-based tools in *Publicize your QIIME 2 plugins (or other QIIME 2-based tools)*.

As our support in this area improves, this Part of *Developing with QIIME 2* will be expanded to describe new functionality.

Part VI

Back Matter

CHAPTER

SIX

BACK MATTER

- *Glossary*
- *List of works cited*
- *Index*

6.1 Glossary

Action
A generic term to describe a *Method*, *Visualizer*, or *Pipeline*. Actions accept parameters and/or *Artifacts* and/or *Metadata*) as input, and generate one or more *Results* as output.

Archive
The directory structure of a QIIME 2 *Result*. Contains *at least* a root directory (named by *UUID*) and a `VERSION` file within that directory.

Artifact
A QIIME 2 *Result* that contains data to operate on.

Artifact class
A kind of *Artifact* that can exist. This is defined by a plugin developer by associating a *semantic type* with a *directory format* when registering an artifact class.

Artifact API
See *Python 3 API*.

Conda metapackage
A metapackage is a package with no files, only metadata. They are typically used to collect several packages together into a single package via dependencies. (source)

Deployment
An installation of QIIME 2 as well as zero-or-more *interfaces* and *plugins*. The collection of interfaces and plugins in a deployment can be defined by a *distribution* of QIIME 2.

Directory Format
An object that is a subclass of `qiime2.plugin.DirectoryFormat`. A Directory Format represents a particular layout of a directory that contains files and/or arbitrarily nested sub-directories, and defines how the contents must be structured.

Distribution
A collection of QIIME 2 plugins that are installed together through a single *conda metapackage*. These are generally grouped by a theme. For example, the *amplicon distribution* provides a collection of plugins for analysis of microbiome amplicon data, while the *metagenome distribution* provides a collection of plugins for analysis of

microbiome shotgun metagenomics data. When a distribution is installed, that particular installation of QIIME 2 is an example of a *deployment*.

DRY
An acronym of *Don't Repeat Yourself*, and a critical principle of software engineering. For more information on DRY and software engineering in general, see Thomas and Hunt [4]. The Thomas and Hunt [4] content on DRY is available in a free example chapter here.

File Format
An object which subclasses either `qiime2.plugin.TextFileFormat` or `qiime2.plugin.BinaryFileFormat`. File formats define the particular format of a file, and define a process for validating the format.

Format
See *file format* and *directory format*.

Framework
The engine of orchestration that enables QIIME 2 to function together as a cohesive unit.

Galaxy
Galaxy is a browser-based graphical interface used to access bioinformatics (and other data science tools) without having to write command line or other code. QIIME 2 provides a Galaxy interface to support access to plugins through a web browser.

Identifier
A unique value that denotes an individual sample or feature.

Identity
Distinguishes a piece of data. QIIME 2 does not consider a rename (like UNIX `mv`) to change identity, however re-running a command, would change identity.

Input
Data provided to an *action*. Can be an *artifact* or *metadata*.

Interface
A user-interface responsible for coordinating user-specified intent into *framework*-driven action.

Metadata
Columnar data for annotating additional values to existing data. Operates along Sample IDs or Feature IDs.

Method
A method accepts some combination of QIIME 2 *artifacts* and *parameters* as *input*, and produces one or more QIIME 2 artifacts as *output*.

Output
Objects returned by an *action*. Can be *artifact(s)* or *visualization(s)*.

Pairwise sequence alignment

1. (noun) A hypothesis about which positions in a pair of biological sequences (i.e., a DNA, RNA, or protein sequence) were derived from a common ancestral sequence position.

2. (verb) The process of generating a pairwise sequence alignment (noun). For additional detail, see the *Pairwise Sequence Alignment* chapter of *An Introduction to Applied Bioinformatics* [3].

Parameter
A value that alters the behavior of an *action*.

Payload
Data that is meant for primary consumption or interpretation (in contrast to *metadata* which may be useful retrospectively, but is not primarily useful).

Pipeline
A pipeline accepts some combination of QIIME 2 *artifacts* and *parameters* as *input*, and produces one or more QIIME 2 *artifacts* and/or *visualizations* as *output*.

Plugin
A discrete module that registers some form of additional functionality with the *framework*, including new *methods*, *visualizers*, *formats*, or *transformers*.

Plugin Manager
An instance of the `qiime2.sdk.PluginManager` object. This object provides access to all plugins, actions, artifact classes, and transformers that are registered to all plugins in a given *deployment*, and are therefore central to the functioning of interfaces.

Primitive Type
A *type* that is used to communicate parameters to an *interface*. These are predefined by the *framework* and cannot be extended.

Provenance
In the context of QIIME 2, provenance or data provenance refers to the history of how a given *result* was generated. Provenance information describes the host system, the computing environment, Actions performed, parameters passed, primary sources cited, and more.

Provenance Replay
The QIIME 2 functionality that enables new executable code to be generated from an existing QIIME 2 *result's provenance*. For additional detail, refer to [7].

Python 3 API
When *the Python 3 API* is referred to in the context of QIIME 2, this refers to the interface that allows users to work with QIIME 2 plugins and actions natively in Python 3 (for example in a Jupyter Notebook environment). This was formerly referred to as the Artifact API.

q2cli
q2cli is the original (and still primary, as of March 2024) command line interface for QIIME 2.

Result
A generic term for either a *Visualization* or an *Artifact*.

Semantic Type
An identifier that is used to describe what some data is intended to represent, and when and where they can be used. When associated with a *directory format*, the combination defines an *artifact class*. These types may be extended by *plugins*.

Single-Use Plugin (SUP)
A plugin that is intended for one specific use case, such as generating figures for a single manuscript, as opposed to a plugin that is intended for general widespread usage.

tl;dr
"Too long; didn't read." In other words, a quick summary of the content that follows.

Transformer
A function registered on the *framework* capable of converting data in one *format* into data of another *format*.

Type
A term that is used to represent several different ideas in QIIME 2, and which is therefore ambiguous when used on its own. More specific terms are *file type*, *semantic type*, and *data type*. See *Semantic types, data types, file formats, and artifact classes* for more information.

UUID
Universally Unique IDentifier, in the context of QIIME 2, almost certainly refers to a *Version 4* UUID, which is a randomly generated ID. See this RFC or this wikipedia entry for details.

View
A particular representation of data. This includes on-disk formats and in-memory data structures (objects).

Visualization
A QIIME 2 *Result* that contains an interactive visualization.

Visualization (Type)
The *type* of a *visualization*. There are no subtyping relations between this type and any other (it is a singleton) and cannot be extended (because it is a singleton).

Visualizer
A visualizer accepts some combination of QIIME 2 *artifacts* and *parameters* as *input*, and produces exactly one *visualization* as *output*.

6.2 List of works cited

6.3 Index

BIBLIOGRAPHY

[1] Daniele Procida. Diátaxis documentation framework. URL: https://diataxis.fr/.

[2] S B Needleman and C D Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Mol. Biol.*, 48(3):443–453, March 1970.

[3] J Gregory Caporaso. An introduction to applied bioinformatics, 2nd edition. 2021. URL: https://readiab.org.

[4] David Thomas and Andrew Hunt. *The Pragmatic Programmer: your journey to mastery, 20th Anniversary Edition*. Addison-Wesley Professional, September 2019.

[5] S F Altschul, W Gish, W Miller, E W Myers, and D J Lipman. Basic local alignment search tool. *J. Mol. Biol.*, 215(3):403–410, October 1990.

[6] T F Smith and M S Waterman. Identification of common molecular subsequences. *J. Mol. Biol.*, 147(1):195–197, March 1981.

[7] Christopher R Keefe, Matthew R Dillon, Elizabeth Gehret, Chloe Herman, Mary Jewell, Colin V Wood, Evan Bolyen, and J Gregory Caporaso. Facilitating bioinformatics reproducibility with QIIME 2 provenance replay. *PLoS Comput. Biol.*, 19(11):e1011676, November 2023.