onice_conversion

Release 0.1.0

onice, sneakers-the-rat

Apr 03, 2022

USER GUIDE

CHAPTER

ONE

CONTRIBUTING

1.1 Adding an Example Notebook

Sphinx doesn't allow references to files that are outside the source directory ('/docs' in our case), so we have to use the nbsphinx_link package to include them in our documentation.

Say we have some example notebook located at `/examples/wehr/wehr-nick.ipynb` , and we want to refer to it from the .rst file located at `/docs/examples/wehr/wehr.rst` . We would create an `.nblink` file at `/docs/examples/wehr/wehr-nick.ipynb` like:

```
{
    "path": "../../../examples/wehr/wehr-nick.ipynb"
}
```

that references the `.ipynb` file relative to the directory that the `.nblink` file is in.

In `/docs/examples/wehr/wehr.rst` , we would then include the notebook using a `toctree` directive like:

```
.. toctree:

   wehr-nick
```

CHAPTER

TWO

SMEAR LAB EXAMPLES

2.1 Reese

Example presented at ONICE meeting 2022-04-01.

2.1.1 Reese

Conversion Example

Presented at ONICE meeting 2022-04-01. Sample dataset to be uploaded separately

Demonstrates reading trialwise behavior data for odor concentration experiment, including

- Making an `pynwb.NWBFile` with a `pynwb.file.Subject` description
- Trialwise data using `pynwb.file.NWBFile.add_trial()`
- Spatial data from pose tracking using `pynwb.behavior.Position`
- Generic Timeseries data for analog sniff signal using `pynwb.base.TimeSeries`
- Writing an NWBFile with `pynwb.NWBHDF5IO`

```

import numpy as np
import os
from datetime import datetime
from pynwb import NWBFile
from pynwb import TimeSeries
from pynwb.behavior import Position
from pynwb import NWBHDF5IO
from pynwb.file import Subject

data_dir = "./session data/"
save_dir = "./NWB1.5.1/"

subjects = os.listdir(data_dir)

for subject in subjects:
     subject_dir = data_dir + subject + "/"
     subject_number = subject[-4:]
     experiments = os.listdir(subject_dir)
```

(continues on next page)

(continued from previous page)

```

    for experiment in experiments:
        experiment_dir = subject_dir + experiment + "/"
        sessions = os.listdir(experiment_dir)
        if experiment == 'ARHMM': #leave out ARHMM data for now
            continue

        frame_data_exists = False
        trial_data_exists = False
        sniff_signal_exists = False

        for session in sessions:
            session_dir = experiment_dir + session + "/"
            session_number = session[-2:]
            if session_number == 'nt': #skip pre-implant data
                continue

            #Neurodata Without Borders file settings
            nwb_description = "Experiment type: " + experiment + ", " + subject + ", "␣
↪+ session
            print(nwb_description)
            nwb_identifier = subject_number + '_' + experiment + '_' + session_number
            io = NWBHDF5IO(save_dir + nwb_identifier + '.nwb', mode='w')

            dateinfo = "2021-01-01, 01:00:00"
            session_date_info = datetime.strptime(dateinfo, "%Y-%m-%d,%H:%M:%S")
            if os.path.exists(session_dir + "notes.txt") == True:
                with open(session_dir + "notes.txt") as f:
                    notes = f.readlines()
                    for line in range(0,len(notes)):
                        if 'Date' in notes[line]:
                            dateinfo = notes[line]
                            dateinfo = dateinfo[6:26]
                            if dateinfo[14] == '-':
                                session_date_info = datetime.strptime(dateinfo, "%Y-%m-
↪%d: %H-%M-%S")
                            else:
                                session_date_info = datetime.strptime(dateinfo, "%Y-%m-
↪%d, %H:%M:%S")

            if os.path.exists(session_dir + "trial_params.txt") == True:
                #sampled by trial
                trial_data_exists = True
                trial_params = np.genfromtxt(session_dir + "trial_params.txt",delimiter␣
↪= ',',skip_header=0)
                concentration_level = trial_params[:,0]
                stimulus_side = trial_params[:,1]
                chosen_side = trial_params[:,2]
                trial_start = trial_params[:,3]
                trial_end = trial_params[:,4]

            if os.path.exists(session_dir + "frame_params_wITI.txt") == True: #sampled␣
↪at 80 Hz
```

(continues on next page)

(continued from previous page)

```
                frame_data_exists = True
                frame_params = np.genfromtxt(session_dir + "frame_params_wITI.txt",
↪delimiter = ',',skip_header=0)
                nose_x = frame_params[:,0]; nose_y = frame_params[:,1]
                head_x = frame_params[:,2]; head_y = frame_params[:,3]
                body_x = frame_params[:,4]; body_y = frame_params[:,5]
                frame_msec = frame_params[:,7]

            if os.path.exists(session_dir + "sniff.bin") == True:
                #sampled at 800 Hz
                sniff_signal_exists = True
                sniff_signal = np.fromfile(session_dir + "sniff.bin",dtype = 'float')

            #create a session-specific neurodata without borders file
            subject_info = Subject(age=None, description=None,
                                   genotype=None, sex=None, species='Mouse', subject_
↪id=subject,
                                   weight=None, date_of_birth=None, strain='B6')
            nwbfile = NWBFile(session_description=nwb_description,  #required
                              identifier=nwb_identifier,  # required
                              session_start_time=session_date_info,  #required
                              subject = subject_info,
                              session_id=session, #optional
                              file_create_date= session_date_info)  #optional

            if trial_data_exists == True:
                nwbfile.add_trial_column(name='level', description='the level of odor␣
↪concentration stimulus presented')
                nwbfile.add_trial_column(name='side', description='which side of the␣
↪assay the correct stimulus is presented on')
                nwbfile.add_trial_column(name='chosen', description='which side of the␣
↪assay the mouse chose (correct or incorrect)')

                for trial in range(0,len(concentration_level)):
                    nwbfile.add_trial(start_time=trial_start[trial],
                                      stop_time=trial_end[trial],
                                      level=concentration_level[trial],side=stimulus_
↪side[trial],
                                      chosen=chosen_side[trial])

            if frame_data_exists == True:
                tracking = Position() #create a position container for tracking data
                tracking.create_spatial_series(name='nose x-position',
                                               data=nose_x,
                                               timestamps=frame_msec,
                                               reference_frame='session start')
                tracking.create_spatial_series(name='nose y-position',
                                               data=nose_y,
                                               timestamps=frame_msec,
                                               reference_frame='session start')
                tracking.create_spatial_series(name='head x-position',
                                               data=head_x,
```

(continues on next page)

(continued from previous page)

```
                                                timestamps=frame_msec,
                                                reference_frame='session start')
                    tracking.create_spatial_series(name='head y-position',
                                                data=head_y,
                                                timestamps=frame_msec,
                                                reference_frame='session start')
                    tracking.create_spatial_series(name='body x-position',
                                                data=body_x,
                                                timestamps=frame_msec,
                                                reference_frame='session start')
                    tracking.create_spatial_series(name='body y-position',
                                                data=body_y,
                                                timestamps=frame_msec,
                                                reference_frame='session start')

                if sniff_signal_exists == True:
                    sniff = TimeSeries(name='sniff_signal', data=sniff_signal, unit='V',␣
↪starting_time=0.0, rate=0.00125)
                    nwbfile.add_acquisition(sniff)

                io.write(nwbfile)
                io.close()
```

Opportunities for Reuse

Some opportunities to make this code able to be integrated into this package, and opportunities to learn a bit of Python!

Breaking into Functions

Most analysis code in science is written as scripts — code that is usually written in the "root" of the document, usually not enclosed within functions or classes (or in the case of matlab, a single function per file). Scripts can be a very direct way of approaching a problem, but they can be difficult to reuse and maintain.

One first step in making code a bit more resuable is to break the code into separate functions. There are a lot of ways of thinking about what makes a "good" function, but in my opinion a good function is one that (among other things)

- **Does "one"(ish) thing**, or, has a well-defined purpose. Functions should be short – it's always possible to combine multiple functions in a larger "wrapper" function.
- Allows **alternative behavior** with arguments using optional arguments with **reasonable defaults.** A function should encapsulate a particular operation that you might need to do multiple times, so rather than copying code with minor modifications, a function should offer the caller optionoal arguments that change its behavior. The function should *require* as few things as it needs to perform the option, so optional arguments should have default values that make the function do what one would expect it to do (ie. what its documentation says it does)
- Has no **side effects,** and conversely doesn't rely on anything outside of the function. A function should never change its surrounding environment (unless that's specifically what it is for): it shouldn't change the working directory, implicitly make or load files, etc. It also shouldn't rely on other variables/objects in the surrounding environment that aren't explicitly passed in as arguments. Functions with side effects are very brittle, as they need to be carefully orchestrated with other functions, making their functionality linked and thus harder to maintain because any change in one function affects the others. Instead, functions should require what they need with arguments, and return their result. Instead of saving a file at the end of a long analysis script, return the result of the analysis and then use a separate saving function that takes the data and a path to save it.

Among many other considerations…

To start breaking into functions, it's always a good idea to make a brief outline of what we need our code to do! In this case, we

- Analyze data from multiple `subjects`, which have biographical data contained in `pynwb.file.Subject` object.
- Each of which has multiple `experiments`
- Which in turn have multiple `sessions` – each of which has its own nwb file.
- Each session can contain
  - `notes.txt`, which contained the date of the experiment
  - `trial_params.txt`, which describes the metadata for each of the sniff trials.
  - `sniff.bin`, a binary file containing analog sniffing data
- After loading, the data is then saved using one of several objects or methods described at the top of this page.

That's a reasonable place to start breaking up the code! We can start by breaking up the nested for loops by putting them in separate functions that look like this…

```
import os

def convert_session(experiment_dir:str, session:str):
    # ... code to conver session
    pass

def convert_experiment(subject_dir:str, experiment:str):
    experiment_dir = subject_dir + experiment + "/"
    sessions = os.listdir(experiment_dir)
    for session in sessions:
        convert_session(experiment_dir, session)

def convert_subject(data_dir: str, subject:str):
    subject_dir = data_dir + subject + "/"
    subject_number = subject[-4:]
    experiments = os.listdir(subject_dir)
    for experiment in experiments:
        if experiment == 'ARHMM':
            continue

        convert_experiment(subject_dir, experiment)
```

but even those are likely to be too "large" — the `convert_session` function would effectively have the rest of the code in the script! It's also a good idea for both readability and reusability to encapsulate other separable operations in functions.

So we can also break up each of the three different types of data we have into their own functions. For example, the position data could look like this:

```
from typing import List, Tuple
import numpy as np
from dataclasses import dataclass

from pynwb.behavior import Position
from pynwb import NWBFile, ProcessingModule
```

(continues on next page)

(continued from previous page)

```
@dataclass
class Frame_Data:
    name: str
    """
    What to call this spatial series
    """
    position: np.ndarray
    """
    Value of the position in (unit)
    """
    timestamps: np.ndarray
    """
    Array of timestamps (ms) for each position
    """

    def add_series(self, position:Position, reference_frame:str='session_start') ->␣
↪Position:
        """
        Method to add this data series to a pynwb Position object.
        """
        position.create_spatial_series(
          name=self.name,
          data=self.position,
          timestamps=self.timestamps,
          reference_frame=reference_frame
        )
        return position

def load_frame_data(data_path:str, delimiter:str = ',', skip_header:int=0) -> List[Frame_
↪Data]:
    """
    Load an individual session's frame data from a file into a list of Frame_Data␣
↪containers
    """
    frame_params = np.genfromtxt(
        data_path,
        delimiter = delimiter,
        skip_header=skip_header)

    names = ['nose_x', 'nose_y', 'head_x', 'head_y', 'body_x', 'body_y']
    position_series = []
    for i, name in enumerate(names):
        position_series.append(
          Frame_Data(name, frame_params[:,i], frame_params[:,7]))

    return position_series

def add_frame_data(nwbfile: NWBFile, frame_data:List[Frame_Data],
                   module_name:str="position", description:str='') -> Tuple[Position,␣
↪ProcessingModule]:
    """
```

(continues on next page)

(continued from previous page)

```
    Load the data from a given frame data file and add it to an NWB IO file.

    Creates a :class:`~pynwb.base.ProcessingModule`

    Args:
        nwbfile: File to add to!
        frame_data (List[:class:`.Frame_Data`]): see :func:`.load_frame_data`
        module_name (str): Name to give to the created ProcessingModule
        description (str): Description to give to the created ProcessingModule

    References:
        https://pynwb.readthedocs.io/en/stable/tutorials/general/file.html?
↪highlight=Position#spatial-series-and-position
    """
    position = Position()
    for data_series in frame_data:
        position = data_series.add_series(position)

    position_module = nwbfile.create_processing_module(
      name=module_name, description=description
    )

    position_module.add(position)

    return position, position_module
```

Which we would use like::

```
>>> frame_data = load_frame_data(data_path)
>>> position, position_module = add_frame_data(nwbfile, frame_data)
```

This is really nice because we have made a relatively general `Frame_Data` class and `add_frame_data` that doesn't depend on the particularity of the data at hand – only the `load_frame_data` has information that's unique to us! (names and how to load and index the data frame). So someone else could then extend our functions by adding their own loading function without needing to rewrite the rest!

By writing docstrings as we go (and using types and type hints, which we'll cover another time!!), we help keep track of what everything does, so this function would have its documentation rendered like:

onice_conversion.**add_frame_data**(*nwbfile: NWBFile*, *frame_data: Frame_Data*, *module_name: str*, *description: str*)

Load the data from a given frame data file and add it to an NWB IO file.

Creates a `ProcessingModule`

Args: nwbfile (NWBFile) : File to add to! frame_data (List[`Frame_Data`]) : see `load_frame_data()` module_name (str) : Name to give to the created ProcessingModule description (str) : Description to give to the created ProcessingModule

References: https://pynwb.readthedocs.io/en/stable/tutorials/general/file.html?highlight=Position#spatial-series-and-position

The next thing we would start doing is structuring our code into separate modules: i/o operations, processing operations, and so on, but that's for another time!

Warnings

todo

(warn about using default values)

```
import os
from datetime import datetime
import warnings

dateinfo = "2021-01-01, 01:00:00"
session_date_info = datetime.strptime(dateinfo, "%Y-%m-%d,%H:%M:%S")
if os.path.exists(session_dir + "notes.txt") == True:
    # get date from file
    pass
else:
    warnings.warn(f'Couldnt get data from notes.txt, using default date {dateinfo}')
```

pathlib!

todo

show use of pathlib vs os.path

```
from pathlib import Path

# for example
data_dir = Path().home() / 'my_data'
text_files = data_dir.glob('**/*.txt')
data_dir.exists()
my_file = data_dir / 'myfile.txt'
my_file2 = my_file.with_stem(my_file.stem + "_two")
# myfile_two.txt
```

CHAPTER

THREE

WEHR LAB EXAMPLES

The following section was generated from `examples/wehr/wehr-ira.ipynb`

3.1 Yavorska

We'll be using an example dataset that can be downloaded from:

[ ]:

The following section was generated from `examples/wehr/wehr-nick.ipynb`

3.2 Sattler

Dataset: https://www.dropbox.com/sh/4adrgjsee60vcvj/AADJ-hbes1uHg3FE0et69sy5a?dl=1

Say we have these files…

[1]:
```
# first handle imports..
from pathlib import Path
from pprint import pprint

from onice_conversion import NWBConverter
from onice_conversion import spec
```

[2]:
```
# we've symlinked the example data folder to the cwd for this example
base_path = Path().cwd()  / '2021-02-26_17-19-10_mouse-0232'

data_files = [str(path.relative_to(base_path)) for path in base_path.glob("**/*")]
pprint(sorted(data_files))
```

```
['.DS_Store',
 '2021-02-26_17-19-12_mouse-0232',
 '2021-02-26_17-19-12_mouse-0232/103_ADC1.continuous',
 '2021-02-26_17-19-12_mouse-0232/103_ADC2.continuous',
 '2021-02-26_17-19-12_mouse-0232/103_ADC3.continuous',
 '2021-02-26_17-19-12_mouse-0232/103_ADC4.continuous',
 '2021-02-26_17-19-12_mouse-0232/103_ADC5.continuous',
 '2021-02-26_17-19-12_mouse-0232/103_ADC6.continuous',
 '2021-02-26_17-19-12_mouse-0232/103_ADC7.continuous',
```

(continues on next page)

(continued from previous page)

```
 '2021-02-26_17-19-12_mouse-0232/103_ADC8.continuous',
 '2021-02-26_17-19-12_mouse-0232/103_AUX1.continuous',
 '2021-02-26_17-19-12_mouse-0232/103_AUX2.continuous',
 '2021-02-26_17-19-12_mouse-0232/103_AUX3.continuous',
 '2021-02-26_17-19-12_mouse-0232/103_CH1.continuous',
 '2021-02-26_17-19-12_mouse-0232/103_CH10.continuous',
 '2021-02-26_17-19-12_mouse-0232/103_CH11.continuous',
 '2021-02-26_17-19-12_mouse-0232/103_CH12.continuous',
 '2021-02-26_17-19-12_mouse-0232/103_CH13.continuous',
 '2021-02-26_17-19-12_mouse-0232/103_CH14.continuous',
 '2021-02-26_17-19-12_mouse-0232/103_CH15.continuous',
 '2021-02-26_17-19-12_mouse-0232/103_CH16.continuous',
 '2021-02-26_17-19-12_mouse-0232/103_CH17.continuous',
 '2021-02-26_17-19-12_mouse-0232/103_CH18.continuous',
 '2021-02-26_17-19-12_mouse-0232/103_CH19.continuous',
 '2021-02-26_17-19-12_mouse-0232/103_CH2.continuous',
 '2021-02-26_17-19-12_mouse-0232/103_CH20.continuous',
 '2021-02-26_17-19-12_mouse-0232/103_CH21.continuous',
 '2021-02-26_17-19-12_mouse-0232/103_CH22.continuous',
 '2021-02-26_17-19-12_mouse-0232/103_CH23.continuous',
 '2021-02-26_17-19-12_mouse-0232/103_CH24.continuous',
 '2021-02-26_17-19-12_mouse-0232/103_CH25.continuous',
 '2021-02-26_17-19-12_mouse-0232/103_CH26.continuous',
 '2021-02-26_17-19-12_mouse-0232/103_CH27.continuous',
 '2021-02-26_17-19-12_mouse-0232/103_CH28.continuous',
 '2021-02-26_17-19-12_mouse-0232/103_CH29.continuous',
 '2021-02-26_17-19-12_mouse-0232/103_CH3.continuous',
 '2021-02-26_17-19-12_mouse-0232/103_CH30.continuous',
 '2021-02-26_17-19-12_mouse-0232/103_CH31.continuous',
 '2021-02-26_17-19-12_mouse-0232/103_CH32.continuous',
 '2021-02-26_17-19-12_mouse-0232/103_CH4.continuous',
 '2021-02-26_17-19-12_mouse-0232/103_CH5.continuous',
 '2021-02-26_17-19-12_mouse-0232/103_CH6.continuous',
 '2021-02-26_17-19-12_mouse-0232/103_CH7.continuous',
 '2021-02-26_17-19-12_mouse-0232/103_CH8.continuous',
 '2021-02-26_17-19-12_mouse-0232/103_CH9.continuous',
 '2021-02-26_17-19-12_mouse-0232/Continuous_Data.openephys',
 '2021-02-26_17-19-12_mouse-0232/TT0.spikes',
 '2021-02-26_17-19-12_mouse-0232/TT1.spikes',
 '2021-02-26_17-19-12_mouse-0232/TT2.spikes',
 '2021-02-26_17-19-12_mouse-0232/TT3.spikes',
 '2021-02-26_17-19-12_mouse-0232/TT4.spikes',
 '2021-02-26_17-19-12_mouse-0232/TT5.spikes',
 '2021-02-26_17-19-12_mouse-0232/TT6.spikes',
 '2021-02-26_17-19-12_mouse-0232/TT7.spikes',
 '2021-02-26_17-19-12_mouse-0232/all_channels.events',
 '2021-02-26_17-19-12_mouse-0232/messages.events',
 '2021-02-26_17-19-12_mouse-0232/messages_bak.events',
 '2021-02-26_17-19-12_mouse-0232/notebook.mat',
 '2021-02-26_17-19-12_mouse-0232/settings.xml',
 '2021-02-26_17-19-12_mouse-0232/stimlog.txt',
 'Sky_mouse-0232_2021-02-26T17_19_10.csv',
```

(continues on next page)

(continued from previous page)

```
 'Sky_mouse-0232_2021-02-26T17_19_10.mp4',
 'TTL_mouse-0232_2021-02-26T17_19_10.csv']
```

Which compose a dataset of

- Continuous extracellular ephys data recorded by open ephys
- Spikes sorted by Kilosort
- Stimulus information from some custom behavioral software
- Raw video of the behaving animal.

Different parts of the metadata are

- Encoded in the file paths
- embedded in a .mat file
- and a .txt file
- and a .csv file

We'll use our fancy new tools in three steps:

1. Add metadata with NWBConverter.add_metadata
2. Add nwb-conversion-tools interfaces to common data formats with .add_interface
3. Add base pynwb container types with .add_container

The first step is to create our converter object, which will store the abstract representation of our data format and handle the conversion to NWB:

[3]:
```
converter = NWBConverter(base_path)
```

3.2.1 Add Metadata!

The first step is to add general file-level metadata about the experiment, the researcher, etc. We can see what fields are available/expected from NWB by default with our converter!

It's a little verbose, so for the purpose of keeping this notebook readable we'll just print the names of the 'NWBFile' metadata container

[4]:
```
sorted([field['name'] for field in converter.base_nwb_metadata['NWBFile']])
```

[4]:
```
['data_collection',
 'electrodes',
 'epoch_tags',
 'epochs',
 'experiment_description',
 'experimenter',
 'file_create_date',
 'identifier',
 'institution',
 'invalid_times',
 'keywords',
 'lab',
 'notes',
```

(continues on next page)

(continued from previous page)

```
 'pharmacology',
 'protocol',
 'related_publications',
 'session_description',
 'session_id',
 'session_start_time',
 'slices',
 'source_script',
 'source_script_file_name',
 'stimulus_notes',
 'subject',
 'surgery',
 'sweep_table',
 'timestamps_reference_time',
 'trials',
 'units',
 'virus']
```

Static Metadata

The simplest metadata is static metadata that you don't expect to change across all instances of this data format. We can call `add_metadata` with a dictionary of static metadata, in this case nested within the `'NWBFile'` container.

[5]:
```
converter.add_metadata({
    'NWBFile': {
        'institution': "University of Oregon",
        'lab': 'Wehr'
    }
})
```

Metadata from paths - the spec module

This package relies heavily on its `.spec` module, which gives us tools to express where data is stored in different forms.

One common pattern is to specify some metadata in file and directory names. In this case the subject ID is encoded in several of the paths. We will use that to start adding metadata for the other default container in nwb, `'Subject'` which has field names:

[6]:
```
sorted([field['name'] for field in converter.base_nwb_metadata['Subject']])
```

[6]:
```
['age',
 'date_of_birth',
 'description',
 'genotype',
 'sex',
 'species',
 'subject_id',
 'weight']
```

Let's use this filename (it doesn't matter which, as long as it will be present in all datasets you're applying this converter to):

`Sky_mouse-0232_2021-02-26T17_19_10.csv`

The subject id `0232` is embedded, and lucky for us so is the experiment start time! We can specify that to the converter like this:

[7]:
```
our_first_spec = spec.Path(
    'Sky_mouse-{Subject[subject_id]}_{NWBFile[session_start_time]}.csv'
)
```

Note how we replaced the parts of the string we want to parse out with `{bracketed}` terms – these define what to call the variables we extract. We can give nested names (ie. to conform to the container structure of NWB files) using `[]` square brackets.

We can preview what the output of our spec object will look like by calling its `parse` method with the directory to look in:

[8]:
```
our_first_spec.parse(base_path)
```

[8]:
```
{'Subject': {'subject_id': '0232'},
 'NWBFile': {'session_start_time': '2021-02-26T17_19_10'}}
```

Metadata in Files

Another common pattern is to store metadata in one or several structured files, like `.json`, `.csv`, `.mat`, and so on. No prob. A lot of our metadata in this case is located in the `notebook.mat` file.

We can use one of our helper functions to preview what's in it:

[9]:
```
mat_meta = spec.external_file.load_clean_mat(
    list(base_path.glob('**/notebook.mat'))[0]
)
mat_meta['nb']
```

[9]:
```
{'user': 'Molly',
 'mouseID': '0232',
 'Depth': 'unknown',
 'datapath': 'Z:\\lab\\djmaus\\Data\\Molly',
 'activedir': '\\\\wehrrig4\\d\\lab\\djmaus\\Data\\Molly\\2021-02-26_17-19-10_mouse-0232\
↪\2021-02-26_17-19-12_mouse-0232',
 'LaserPower': 'unknown',
 'mouseDOB': 'age unknown',
 'mouseSex': 'sex unknown',
 'mouseGenotype': 'genotype unknown',
 'Drugs': 'none',
 'notes': array([], dtype='<U1'),
 'Reinforcement': 'none'}
```

We can add metadata from the file using the `Mat` object, which in this case needs us to specify the `key` separately. Since we don't really care about the rest of the path, it might change, and there should only be one notebook, we can just glob away the rest of the path as well

Say for example, we want to get the experimenter's name

[10]:
```
mat_spec = spec.Mat(
```

(continues on next page)

(continued from previous page)

```
    path='**/notebook.mat', # 2 **s mean we can glob recursively
    key="user", # hold up on the nested ones for this,
    field = ('nb', 'user')
)
mat_spec.parse(base_path)
```

[10]:
```
{'user': 'Molly'}
```

3.2.2 Add Interfaces!

We have some open ephys data here! It's described by the

[11]:
```
converter.add_interface('recording', 'open_ephys')
```

```
Source Schema for ABCMeta
-------------------------
{'additionalProperties': True,
 'properties': {'folder_path': {'description': 'Path to directory containing '
                                               'OpenEphys files.',
                                'format': 'directory',
                                'type': 'string'}},
 'required': ['folder_path'],
 'type': 'object'}
-------------------------
```

[12]:
```
converter.add_interface(
    'recording', 'open_ephys',
    spec.Glob(
        key="folder_path",
        format="*mouse*",
        only_dirs=True
    )
)
```

3.2.3 Run the conversion!!

[13]:
```
# converter.run_conversion(nwbfile_path='nwbfile.nwb')
```

Here we have two mighty fine example of how to convert wehrlab data to NWB!

3.3 Nick

This example has multiple video files as well as electrophysiology!

3.4 Ira

CHAPTER

FOUR

NWB CONVERTER

Classes:

NWBConverter(*args, **kwargs)	ONICE extension to nwb_conversion_tools.NWBConverter

Functions:

_monkeypatch_spikeextractors()	To make *NWBConverter.hail_mary()* work, we have to override some __del__ methods in spikeextractors that throw errors on incomplete __init__ calls

class onice_conversion.nwbconverter.**NWBConverter**(**args*, ***kwargs*)

Bases: nwb_conversion_tools.nwbconverter.NWBConverter

ONICE extension to nwb_conversion_tools.NWBConverter

Parameters

- **base_dir** (pathlib.Path) – The base directory of the source data, from which all paths are relative. If not provided at initialization, must be provided when calling *run_conversion()*
- **source_data** (*dict*) – Old style source_data dictionary, kept for compatibility

Methods:

add_container([container_name, spec])	Add a
hail_mary([base_dir, interface_type])	Just try every interface on every file and see what instantiates.
add_interface([interface_type, device_name, ...])	Add a recording interface
add_metadata(spec)	**Parameters** **spec** (*BaseSpec, dict*) -- if an object that inherits from BaseSpec, then the keys and values of metadata
convert_many(expt_paths[, out_fns])	
from_json(json_path[, hook, base_dir])	Reconstitute a parameterized converter from a json file created by NWBConverter.to_json()
get_conversion_options_schema()	Compile conversion option schemas from each of the data interface classes.
get_metadata()	Auto-fill as much of the metadata as possible.
get_metadata_schema()	Compile metadata schemas from each of the data interface objects.
get_source_schema()	Compile input schemas from each of the data interface classes.
run_conversion([metadata, nwbfile_path, ...])	Run the NWB conversion over all the instantiated data interfaces.
to_json([output_path, mode])	Save the converter parameterization from add_metadata() and add_interface() to a .json file, for use with from_json() to recreate conversion objects :)

Attributes:

base_nwb_metadata	Return descriptions for the basic file-level metadata container objects, `'NWBFile', 'Subject'`
conversion_options_schema	
metadata	
metadata_schema	
source_schema	

add_container(*container_name: Optional[str] = None*, *spec: Optional[*onice_conversion.spec.base_spec.BaseSpec*] = None*, ***kwargs*)

Add a

Parameters

- **container_type** (*str*)
- **container_name** (*str*)
- **spec** (`BaseSpec`) – Spec object declaring the metadata for the container
- ****kwargs** – stored as static metadata and passed to container

Returns:

property base_nwb_metadata: Dict[str, tuple]

Return descriptions for the basic file-level metadata container objects, `'NWBFile'`, `'Subject'`

Returns dict of tuples of parameter spec for each container type

hail_mary(*base_dir: Optional[pathlib.Path] = None*, *interface_type: Optional[str] = None*)

Just try every interface on every file and see what instantiates.

Parameters

- **base_dir** (*directory to peruse. if none, then the base_dir provided on init is used.*)
- **interface_type** (*if provided, only try interfaces of this type*)

Returns (interface object, path (relative to base_dir), parameter key that was used, and the instantiated object itself)

Return type tuple of:

add_interface(*interface_type: Optional[str] = None*, *device_name: Optional[str] = None*, *spec: Optional[nwb_conversion_tools.spec.base_spec.BaseSpec] = None*, ***kwargs*)

Add a recording interface

Specify interface either with an interface type and name, or else give the class itself as `interface_class`. If both are present, use the class.

Everything afterwards

Parameters

- **interface_type** (*str*) – Type of interface, like 'recording' – a name of a package in `interfaces`
- **device_name** (*str*) – Name of specific interface, matching the interfaces `device_name`
- **spec** (*BaseSpec*) – Metadata specifier to parameterize interface object
- **kwargs** – kwargs passed to data interface.

add_metadata(*spec: Union[nwb_conversion_tools.spec.base_spec.BaseSpec, str]*)

Parameters **spec** (*BaseSpec, dict*) – if an object that inherits from *BaseSpec*, then the keys and values of metadata are resolved by the object: ie. the keys are the value of *BaseSpec.specifies* and *BaseSpec.parse()* returns a dictionary of keys and values.

if dictionary, assumes static metadata (unchanged across multiple sessions/experiments) otherwise, use spec to resolve Either a string representing a "top-level" metadata property, or a tuple of nested metadata properties like ('NWBFile', 'experimenter')

property conversion_options_schema

convert_many(*expt_paths: list*, *out_fns: Optional[list] = None*, **args*, ***kwargs*)

classmethod from_json(*json_path: Union[str, pathlib.Path, dict]*, *hook: Optional[Callable] = None*, *base_dir: Optional[Union[str, pathlib.Path]] = None*) → nwb_conversion_tools.nwbconverter.NWBConverter

Reconstitute a parameterized converter from a json file created by *NWBConverter.to_json()*

Parameters

- **json_path** (*str, pathlib.Path, dict*) – Path to the .json file, or else the already-loaded dict
- **hook** (*callable*) – Optional callable to use with json.load's `object_hook`

- **base_dir** (*str, pathlib.Path*) – Optional, instantiate the converter with a `base_path`

Return type Reconstituted Converter!

`get_conversion_options_schema()`

Compile conversion option schemas from each of the data interface classes.

`get_metadata()`

Auto-fill as much of the metadata as possible. Must comply with metadata schema.

`get_metadata_schema()`

Compile metadata schemas from each of the data interface objects.

`get_source_schema()`

Compile input schemas from each of the data interface classes.

`property metadata`

`property metadata_schema`

`run_conversion`(*metadata: Optional[dict] = None*, *nwbfile_path: Optional[str] = None*, *overwrite: Optional[bool] = False*, *nwbfile: Optional[pynwb.file.NWBFile] = None*, *conversion_options: Optional[dict] = None*, *base_dir: Optional[pathlib.Path] = None*)

Run the NWB conversion over all the instantiated data interfaces.

Parameters

- **metadata** (*dict*)
- **nwbfile_path** (*str, optional*) – Location to save the NWBFile, if save_to_file is True. The default is None.
- **overwrite** (*bool, optional*) – If True, replaces any existing NWBFile at the nwbfile_path location, if save_to_file is True. If False, appends the existing NWBFile at the nwbfile_path location, if save_to_file is True. The default is False.
- **nwbfile** (*NWBFile, optional*) – A pre-existing NWBFile object to be appended (instead of reading from nwbfile_path).
- **conversion_options** (*dict, optional*) – Similar to source_data, a dictionary containing keywords for each interface for which non-default conversion specification is requested.

Returns **nwbfile** – the created NWBFile

Return type NWBFile

`property source_schema`

`to_json`(*output_path: Optional[Union[str, pathlib.Path]] = None*, *mode: str = 'w'*) → dict

Save the converter parameterization from *add_metadata()* and *add_interface()* to a .json file, for use with *from_json()* to recreate conversion objects :)

Parameters

- **output_path** (*str, pathlib.Path*) – Path to write .json file to. if `None`, don't save, just return dict.
- **mode** (*str*) – Write mode, default: 'w'

Return type dict created and saved

`onice_conversion.nwbconverter._monkeypatch_spikeextractors()`

To make *NWBConverter.hail_mary()* work, we have to override some __del__ methods in spikeextractors that throw errors on incomplete __init__ calls

CHAPTER

FIVE

SPEC

5.1 Path Spec

Classes:

Path(format, *args, **kwargs)	Specify a metadata variable embedded in a path using `parse`
Paths(format, *args, **kwargs)	Like `spec.Path` but allows multiple values for a single key
Glob(key, format[, only_dirs])	Sort of the opposite of *Path* -- specify some path given some metadata values

class onice_conversion.spec.path.**Path**(*format: str*, **args*, ***kwargs*)

Bases: *onice_conversion.spec.base_spec.BaseSpec*

Specify a metadata variable embedded in a path using `parse`

See the parse documentation for more details, but briefly, to specify the metadata variables `subject_id == 'jonny'` and `session_id = '001'` in a file path `data/recordings/jonny_spikes_001.spikes`, one would use a `format == 'data/recordings/{subject_id}_spikes_{session_id}.spikes`. Additional options like specifying a format for the values, etc. can be found in the parse documentation.

Raises an exception if multiple matching values are found in *Path._parse()* , this is the singular version, and if there are multiple matches that means it's mis-specified To allow multiple matches, try *Paths*

Parameters **retype** (*Callable (optional)*)

Methods:

_parse_dir(base_path)	First part of Path._parse() , given a base directory and parser, return a list of dicts of matching keys found.
_parse(base_path[, metadata])	Parse metadata stored in some path name relative to
_expand_named_fields(named_fields)	Convert nested key specs like key[key2] into nested dicts
_full_name()	Returns the full module and class name of an object, eg.
_get_init_args()	introspect object and get all arguments passed on __init__
children()	Generator for iterating over children (added)
parse(base_path[, metadata])	Parse all parameters from self and child _parse() methods, combining into single dictionary
to_dict()	Get a dictionary description of this spec object, of the form.

Attributes:

parent	
specifies	Which metadata variables are specified by this Spec object and its children

_parse_dir(*base_path: Union[str, pathlib.Path]*) → list

First part of Path._parse() , given a base directory and parser, return a list of dicts of matching keys found.

_parse(*base_path: Union[str, pathlib.Path]*, *metadata: Optional[dict] = None*) → dict

Parse metadata stored in some path name relative to using the parser created by `format`.

If the input path is not absolute, it is made absolute relative to `base_path` so that it matches `format`

Raises a AmbiguityError if multiple matches for a single key are found, and a `ValueError` if zero matches are found.

Parameters **base_path** (pathlib.Path) – Path to _parse!!!

Return type dict of metadata params

_expand_named_fields(*named_fields*)

Convert nested key specs like key[key2] into nested dicts

borroed from: https://github.com/r1chardj0n3s/parse/blob/0477aa58673cd957c19d377e029347ce72c08b1b/parse.py#L944

_full_name()

Returns the full module and class name of an object, eg. `nwb_conversion_tools.spec.external_file.JSON`

Return type str

_get_init_args()

introspect object and get all arguments passed on __init__

depends on introspecting up frames so should only be called *during* the top-level __init__ of the base class :)

Return type dict of argument names and params

children() → Iterable[*onice_conversion.spec.base_spec.BaseSpec*]

Generator for iterating over children (added)

property parent: *onice_conversion.spec.base_spec.BaseSpec*

parse(*base_path: pathlib.Path*, *metadata: Optional[dict] = None*) → dict

Parse all parameters from self and child *_parse()* methods, combining into single dictionary

Parameters

- **base_path** (*Path*) – The base path we compute the spec'd value from!
- **metadata** (*dict*) – other metadata used by the parsing function, usually passed in *NWBConverter.run_conversion()*

property specifies: Tuple[str, ...]

Which metadata variables are specified by this Spec object and its children

Return type tuple of strings

to_dict() → dict

Get a dictionary description of this spec object, of the form:

```
{
    'module': self.__module__,
    'class': type(self).__name__,
    'kwargs': self._init_args,
    'children': [ ... same structure as top-level without children list ...]
}
```

That allows a spec to be reconstituted with `from_dict()`

Return type dict of initialization parameters, as described above

class onice_conversion.spec.path.**Paths**(*format: str*, **args*, ***kwargs*)

Bases: *onice_conversion.spec.path.Path*

Like `spec.Path` but allows multiple values for a single key

Parameters **retype** (*Callable (optional)*)

Methods:

_expand_named_fields(named_fields)	Convert nested key specs like key[key2] into nested dicts
_full_name()	Returns the full module and class name of an object, eg.
_get_init_args()	introspect object and get all arguments passed on __init__
_parse_dir(base_path)	First part of *Path._parse()* , given a base directory and parser, return a list of dicts of matching keys found.
children()	Generator for iterating over children (added)
parse(base_path[, metadata])	Parse all parameters from self and child *_parse()* methods, combining into single dictionary
to_dict()	Get a dictionary description of this spec object, of the form.

Attributes:

parent	
specifies	Which metadata variables are specified by this Spec object and its children

_expand_named_fields(*named_fields*)

Convert nested key specs like key[key2] into nested dicts

borroed from: https://github.com/r1chardj0n3s/parse/blob/0477aa58673cd957c19d377e029347ce72c08b1b/parse.py#L944

_full_name()

Returns the full module and class name of an object, eg. `nwb_conversion_tools.spec.external_file.JSON`

Return type str

_get_init_args()

introspect object and get all arguments passed on __init__

depends on introspecting up frames so should only be called *during* the top-level __init__ of the base class :)

Return type dict of argument names and params

_parse_dir(*base_path: Union[str, pathlib.Path]*) → list

First part of *Path._parse()* , given a base directory and parser, return a list of dicts of matching keys found.

children() → Iterable[*onice_conversion.spec.base_spec.BaseSpec*]

Generator for iterating over children (added)

property parent: *onice_conversion.spec.base_spec.BaseSpec*

parse(*base_path: pathlib.Path*, *metadata: Optional[dict] = None*) → dict

Parse all parameters from self and child *_parse()* methods, combining into single dictionary

Parameters

- **base_path** (*Path*) – The base path we compute the spec'd value from!
- **metadata** (*dict*) – other metadata used by the parsing function, usually passed in *NWBConverter.run_conversion()*

property specifies: Tuple[str, ...]

Which metadata variables are specified by this Spec object and its children

Return type tuple of strings

to_dict() → dict

Get a dictionary description of this spec object, of the form:

```
{
    'module': self.__module__,
    'class': type(self).__name__,
    'kwargs': self._init_args,
    'children': [ ... same structure as top-level without children list ...]
}
```

That allows a spec to be reconstituted with `from_dict()`

Return type dict of initialization parameters, as described above

class onice_conversion.spec.path.**Glob**(*key: str*, *format: str*, *only_dirs: bool = False*, **args*, ***kwargs*)

Bases: *onice_conversion.spec.base_spec.BaseSpec*

Sort of the opposite of *Path* – specify some path given some metadata values

Replaces any named format variables in *{brackets}*, and then globs any ``'*'``s

Parameters

- **key** (*str*) – The key that will define what's returned from Parse
- **format** (*str*) – A globlike format string to match files within the base directory, eg to match `parentdir_335092/some_file_250269287.bin` we might use `"parentdir_*/some_file_*.bin"`

  Can also use previously defined metadata, eg to replace some part of the file with `subject_id`, use `"parentdir_{subject_id}/"` etc.
- **only_dirs** (*bool*) – Only match directories, not files (default: False)
- ***args** ()
- ****kwargs** ()

Methods:

__init__(key, format[, only_dirs])	**Parameters** • **key** (*str*) -- The key that will define what's returned from Parse
_parse(base_path[, metadata])	Find a path by first replacing *{format_strings}* with variables from the passed metadata dict and then globbing over any '*'
_expand_named_fields(named_fields)	Convert nested key specs like key[key2] into nested dicts
_full_name()	Returns the full module and class name of an object, eg.
_get_init_args()	introspect object and get all arguments passed on __init__
children()	Generator for iterating over children (added)
parse(base_path[, metadata])	Parse all parameters from self and child *_parse()* methods, combining into single dictionary
to_dict()	Get a dictionary description of this spec object, of the form.

Attributes:

parent	
specifies	Which metadata variables are specified by this Spec object and its children

`__init__`(*key: str*, *format: str*, *only_dirs: bool = False*, **args*, ***kwargs*)

Parameters

- **key** (*str*) – The key that will define what's returned from Parse
- **format** (*str*) – A globlike format string to match files within the base directory, eg to match `parentdir_335092/some_file_250269287.bin` we might use `"parentdir_*/some_file_*.bin"`

  Can also use previously defined metadata, eg to replace some part of the file with `subject_id`, use `"parentdir_{subject_id}/"` etc.
- **only_dirs** (*bool*) – Only match directories, not files (default: False)
- ***args** ()
- ****kwargs** ()

`_parse`(*base_path: Union[str, pathlib.Path]*, *metadata: Optional[dict] = None*) → dict

Find a path by first replacing *{format_strings}* with variables from the passed metadata dict and then globbing over any '*'

This class ensures a single path is returned, and raises an *AmbiguityError* otherwise. To return multiple paths, use `Globs`

Parameters

- **base_path**
- **metadata**

`_expand_named_fields`(*named_fields*)

Convert nested key specs like key[key2] into nested dicts

borroed from: https://github.com/r1chardj0n3s/parse/blob/0477aa58673cd957c19d377e029347ce72c08b1b/parse.py#L944

`_full_name`()

Returns the full module and class name of an object, eg. `nwb_conversion_tools.spec.external_file.JSON`

Return type str

`_get_init_args`()

introspect object and get all arguments passed on __init__

depends on introspecting up frames so should only be called *during* the top-level __init__ of the base class :)

Return type dict of argument names and params

`children`() → Iterable[*onice_conversion.spec.base_spec.BaseSpec*]

Generator for iterating over children (added)

`property parent: onice_conversion.spec.base_spec.BaseSpec`

`parse`(*base_path: pathlib.Path*, *metadata: Optional[dict] = None*) → dict

Parse all parameters from self and child *_parse()* methods, combining into single dictionary

Parameters

- **base_path** (*Path*) – The base path we compute the spec'd value from!

- **metadata** (*dict*) – other metadata used by the parsing function, usually passed in *NWBConverter.run_conversion()*

property specifies: Tuple[str, ...]

Which metadata variables are specified by this Spec object and its children

Return type tuple of strings

to_dict() → dict

Get a dictionary description of this spec object, of the form:

```
{
    'module': self.__module__,
    'class': type(self).__name__,
    'kwargs': self._init_args,
    'children': [ ... same structure as top-level without children list ...]
}
```

That allows a spec to be reconstituted with `from_dict()`

Return type dict of initialization parameters, as described above

5.2 External File Spec

Specify metadata that's in a separate, external file from the standard format files

Classes:

BaseExternalFileSpec(path, key, field[, cache])	**Parameters** • **self**
JSON([hook])	Load a field from a .json file.
Mat([simplified])	**Parameters** • **simplified** (*bool*) -- Whether we attempt to simplify the matlab struct into lists
YAML(path, key, field[, cache])	**Parameters** • **self**

Functions:

load_clean_mat(filename)	Load a matlab *.mat* file as python lists, dictionaries, and numpy arrays rather than the sort-of hard to work with numpy record arrays.

class onice_conversion.spec.external_file.**BaseExternalFileSpec**(*path: pathlib.Path*, *key: str*, *field: Union[str, Tuple[str, ...]]*, *cache: bool = True*, **args*, ***kwargs*)

Bases: *onice_conversion.spec.base_spec.BaseSpec*

Parameters

- **self**
- **path** (path relative to base_dir that is passed in *_parse()*)
- **key**
- **field**
- **cache** (*bool*) – if True, store loaded file in *loaded_files* dictionary to prevent re-load if another spec needs it.
- **kwargs**

Attributes:

loaded_files	
parent	
specifies	Which metadata variables are specified by this Spec object and its children

Methods:

__init__(path, key, field[, cache])	**Parameters** • **self**
_load_file(path)	Load the file and return it as a nested dictionary of dictionaries or tuples
_sub_select(loaded_file)	Use `field` to select from the loaded_file
_expand_named_fields(named_fields)	Convert nested key specs like key[key2] into nested dicts
_full_name()	Returns the full module and class name of an object, eg.
_get_init_args()	introspect object and get all arguments passed on __init__
children()	Generator for iterating over children (added)
parse(base_path[, metadata])	Parse all parameters from self and child *_parse()* methods, combining into single dictionary
to_dict()	Get a dictionary description of this spec object, of the form.

loaded_files = {}

__init__(*path: pathlib.Path*, *key: str*, *field: Union[str, Tuple[str, ...]]*, *cache: bool = True*, **args*, ***kwargs*)

Parameters

- **self**
- **path** (path relative to base_dir that is passed in *_parse()*)
- **key**

- **field**
- **cache** (*bool*) – if True, store loaded file in `loaded_files` dictionary to prevent re-load if another spec needs it.
- **kwargs**

abstract _load_file(*path: pathlib.Path*) → dict

Load the file and return it as a nested dictionary of dictionaries or tuples

such that it can be indexed by successively slicing with `field`

Parameters

- **self**
- **key** (*str*) – name of the property that will be returned
- **path**

_sub_select(*loaded_file: dict*) → Any

Use `field` to select from the loaded_file

Parameters **loaded_file**

_expand_named_fields(*named_fields*)

Convert nested key specs like key[key2] into nested dicts

borroed from: https://github.com/r1chardj0n3s/parse/blob/0477aa58673cd957c19d377e029347ce72c08b1b/parse.py#L944

_full_name()

Returns the full module and class name of an object, eg. `nwb_conversion_tools.spec.external_file.JSON`

Return type str

_get_init_args()

introspect object and get all arguments passed on __init__

depends on introspecting up frames so should only be called *during* the top-level __init__ of the base class :)

Return type dict of argument names and params

children() → Iterable[*onice_conversion.spec.base_spec.BaseSpec*]

Generator for iterating over children (added)

property parent: *onice_conversion.spec.base_spec.BaseSpec*

parse(*base_path: pathlib.Path*, *metadata: Optional[dict] = None*) → dict

Parse all parameters from self and child `_parse()` methods, combining into single dictionary

Parameters

- **base_path** (*Path*) – The base path we compute the spec'd value from!
- **metadata** (*dict*) – other metadata used by the parsing function, usually passed in `NWBConverter.run_conversion()`

property specifies: Tuple[str, ...]

Which metadata variables are specified by this Spec object and its children

Return type tuple of strings

`to_dict()` → dict

Get a dictionary description of this spec object, of the form:

```
{
    'module': self.__module__,
    'class': type(self).__name__,
    'kwargs': self._init_args,
    'children': [ ... same structure as top-level without children list ...]
}
```

That allows a spec to be reconstituted with `from_dict()`

Return type dict of initialization parameters, as described above

class onice_conversion.spec.external_file.**JSON**(*hook: Optional[Callable] = None*, **args*, ***kwargs*)

Bases: *onice_conversion.spec.external_file.BaseExternalFileSpec*

Load a field from a .json file. see base class for docs

Parameters

- **hook** (*Optionally, include some callable function to use as the fallback*) – object loader hook (see `object_hook` argument in `json.load` for more information)
- **args** (passed to *BaseExternalFileSpec*)
- **kwargs**

Methods:

__init__([hook])	Load a field from a .json file.
_expand_named_fields(named_fields)	Convert nested key specs like key[key2] into nested dicts
_full_name()	Returns the full module and class name of an object, eg.
_get_init_args()	introspect object and get all arguments passed on __init__
_sub_select(loaded_file)	Use `field` to select from the loaded_file
children()	Generator for iterating over children (added)
parse(base_path[, metadata])	Parse all parameters from self and child *_parse()* methods, combining into single dictionary
to_dict()	Get a dictionary description of this spec object, of the form.

Attributes:

loaded_files	
parent	
specifies	Which metadata variables are specified by this Spec object and its children

`__init__`(*hook: Optional[Callable] = None*, **args*, ***kwargs*)

Load a field from a .json file. see base class for docs

Parameters

- **hook** (*Optionally, include some callable function to use as the fallback*) – object loader hook (see `object_hook` argument in `json.load` for more information)
- **args** (passed to *BaseExternalFileSpec*)
- **kwargs**

_expand_named_fields(*named_fields*)

Convert nested key specs like key[key2] into nested dicts

borroed from: https://github.com/r1chardj0n3s/parse/blob/0477aa58673cd957c19d377e029347ce72c08b1b/parse.py#L944

_full_name()

Returns the full module and class name of an object, eg. `nwb_conversion_tools.spec.external_file.JSON`

Return type str

_get_init_args()

introspect object and get all arguments passed on __init__

depends on introspecting up frames so should only be called *during* the top-level __init__ of the base class :)

Return type dict of argument names and params

_sub_select(*loaded_file: dict*) → Any

Use `field` to select from the loaded_file

Parameters **loaded_file**

children() → Iterable[*onice_conversion.spec.base_spec.BaseSpec*]

Generator for iterating over children (added)

loaded_files = {}

property parent: *onice_conversion.spec.base_spec.BaseSpec*

parse(*base_path: pathlib.Path*, *metadata: Optional[dict] = None*) → dict

Parse all parameters from self and child *_parse()* methods, combining into single dictionary

Parameters

- **base_path** (*Path*) – The base path we compute the spec'd value from!
- **metadata** (*dict*) – other metadata used by the parsing function, usually passed in *NWBConverter.run_conversion()*

property specifies: Tuple[str, ...]

Which metadata variables are specified by this Spec object and its children

Return type tuple of strings

to_dict() → dict

Get a dictionary description of this spec object, of the form:

```
{
    'module': self.__module__,
    'class': type(self).__name__,
    'kwargs': self._init_args,
```

(continues on next page)

(continued from previous page)

```
    'children': [ ... same structure as top-level without children list ...]
}
```

That allows a spec to be reconstituted with `from_dict()`

Return type dict of initialization parameters, as described above

class onice_conversion.spec.external_file.**Mat**(*simplified: bool = True*, **args*, ***kwargs*)

Bases: *onice_conversion.spec.external_file.BaseExternalFileSpec*

Parameters

- **simplified** (*bool*) – Whether we attempt to simplify the matlab struct into lists and dicts, or just take the base output from `scipy.io.loadmat()`
- ***args** () – Passed to superclass
- ****kwargs** () – Passed to superclass

Methods:

__init__([simplified])	**Parameters** • **simplified** (*bool*) -- Whether we attempt to simplify the matlab struct into lists
_sub_select(loaded_file)	Calls *BaseExternalFileSpec._sub_select()*, but then unstacks all *len == 1* numpy arrays so that the *field* arg can be like *('sessionInfo', 'session')* rather than *('sessionInfo', 'session', 0, 0, 0, 0, 0, 0)*
_expand_named_fields(named_fields)	Convert nested key specs like key[key2] into nested dicts
_full_name()	Returns the full module and class name of an object, eg.
_get_init_args()	introspect object and get all arguments passed on __init__
children()	Generator for iterating over children (added)
parse(base_path[, metadata])	Parse all parameters from self and child *_parse()* methods, combining into single dictionary
to_dict()	Get a dictionary description of this spec object, of the form.

Attributes:

loaded_files	
parent	
specifies	Which metadata variables are specified by this Spec object and its children

__init__(*simplified: bool = True*, **args*, ***kwargs*)

Parameters

- **simplified** (*bool*) – Whether we attempt to simplify the matlab struct into lists and dicts, or just take the base output from `scipy.io.loadmat()`
- ***args** () – Passed to superclass
- ****kwargs** () – Passed to superclass

_sub_select(*loaded_file: dict*) → Any

Calls *BaseExternalFileSpec._sub_select()*, but then unstacks all *len == 1* numpy arrays so that the *field* arg can be like *('sessionInfo', 'session')* rather than *('sessionInfo', 'session', 0, 0, 0, 0, 0, 0)*

Parameters **loaded_file**

_expand_named_fields(*named_fields*)

Convert nested key specs like key[key2] into nested dicts

borroed from: https://github.com/r1chardj0n3s/parse/blob/0477aa58673cd957c19d377e029347ce72c08b1b/parse.py#L944

_full_name()

Returns the full module and class name of an object, eg. `nwb_conversion_tools.spec.external_file.JSON`

Return type str

_get_init_args()

introspect object and get all arguments passed on __init__

depends on introspecting up frames so should only be called *during* the top-level __init__ of the base class :)

Return type dict of argument names and params

children() → Iterable[*onice_conversion.spec.base_spec.BaseSpec*]

Generator for iterating over children (added)

loaded_files = {}

property parent: *onice_conversion.spec.base_spec.BaseSpec*

parse(*base_path: pathlib.Path*, *metadata: Optional[dict] = None*) → dict

Parse all parameters from self and child *_parse()* methods, combining into single dictionary

Parameters

- **base_path** (*Path*) – The base path we compute the spec'd value from!
- **metadata** (*dict*) – other metadata used by the parsing function, usually passed in *NWBConverter.run_conversion()*

property specifies: Tuple[str, ...]

Which metadata variables are specified by this Spec object and its children

Return type tuple of strings

to_dict() → dict

Get a dictionary description of this spec object, of the form:

```
{
    'module': self.__module__,
    'class': type(self).__name__,
```

(continues on next page)

(continued from previous page)

```
    'kwargs': self._init_args,
    'children': [ ... same structure as top-level without children list ...]
}
```

That allows a spec to be reconstituted with `from_dict()`

Return type dict of initialization parameters, as described above

class onice_conversion.spec.external_file.**YAML**(*path: pathlib.Path*, *key: str*, *field: Union[str, Tuple[str, ...]]*, *cache: bool = True*, **args*, ***kwargs*)

Bases: *onice_conversion.spec.external_file.BaseExternalFileSpec*

Parameters

- **self**
- **path** (path relative to base_dir that is passed in *_parse()*)
- **key**
- **field**
- **cache** (*bool*) – if True, store loaded file in *loaded_files* dictionary to prevent re-load if another spec needs it.
- **kwargs**

Methods:

__init__(path, key, field[, cache])	**Parameters** • **self**
_expand_named_fields(named_fields)	Convert nested key specs like key[key2] into nested dicts
_full_name()	Returns the full module and class name of an object, eg.
_get_init_args()	introspect object and get all arguments passed on __init__
_sub_select(loaded_file)	Use `field` to select from the loaded_file
children()	Generator for iterating over children (added)
parse(base_path[, metadata])	Parse all parameters from self and child *_parse()* methods, combining into single dictionary
to_dict()	Get a dictionary description of this spec object, of the form.

Attributes:

loaded_files	
parent	
specifies	Which metadata variables are specified by this Spec object and its children

__init__(*path: pathlib.Path*, *key: str*, *field: Union[str, Tuple[str, ...]]*, *cache: bool = True*, **args*, ***kwargs*)

Parameters

- **self**
- **path** (path relative to base_dir that is passed in *_parse()*)
- **key**
- **field**
- **cache** (*bool*) – if True, store loaded file in *loaded_files* dictionary to prevent re-load if another spec needs it.
- **kwargs**

_expand_named_fields(*named_fields*)

Convert nested key specs like key[key2] into nested dicts

borroed from: https://github.com/r1chardj0n3s/parse/blob/0477aa58673cd957c19d377e029347ce72c08b1b/parse.py#L944

_full_name()

Returns the full module and class name of an object, eg. `nwb_conversion_tools.spec.external_file.JSON`

Return type str

_get_init_args()

introspect object and get all arguments passed on __init__

depends on introspecting up frames so should only be called *during* the top-level __init__ of the base class :)

Return type dict of argument names and params

_sub_select(*loaded_file: dict*) → Any

Use `field` to select from the loaded_file

Parameters **loaded_file**

children() → Iterable[*onice_conversion.spec.base_spec.BaseSpec*]

Generator for iterating over children (added)

loaded_files = {}

property parent: *onice_conversion.spec.base_spec.BaseSpec*

parse(*base_path: pathlib.Path*, *metadata: Optional[dict] = None*) → dict

Parse all parameters from self and child *_parse()* methods, combining into single dictionary

Parameters

- **base_path** (*Path*) – The base path we compute the spec'd value from!
- **metadata** (*dict*) – other metadata used by the parsing function, usually passed in *NWBConverter.run_conversion()*

property specifies: Tuple[str, ...]

Which metadata variables are specified by this Spec object and its children

Return type tuple of strings

to_dict() → dict

Get a dictionary description of this spec object, of the form:

```
{
    'module': self.__module__,
    'class': type(self).__name__,
    'kwargs': self._init_args,
    'children': [ ... same structure as top-level without children list ...]
}
```

That allows a spec to be reconstituted with `from_dict()`

Return type dict of initialization parameters, as described above

onice_conversion.spec.external_file.**load_clean_mat**(*filename: str*) → dict

Load a matlab *.mat* file as python lists, dictionaries, and numpy arrays rather than the sort-of hard to work with numpy record arrays.

Credit to https://stackoverflow.com/a/29126361/13113166

Parameters **filename** (*str*) – filename of .mat to load

Returns dict

Specify where your metadata is within a directory.

An extension of the spec module I started in the nwb-conversion-tools package, rebuilt here.

In reseracher-specific data formats, metadata is tucked away in a thousand unpredictable places. The spec module is intended to give you the means of expressing where it is.

If it's embedded within some path name, try `spec.Path`,

If it's embedded in some .mat file, try `spec.Mat`

Functions:

parse_nested_spec(spec, base_dir)	

onice_conversion.spec.**parse_nested_spec**(*spec*, *base_dir*)

5.3 Base Spec

Classes:

BaseSpec([retype])	Base class for specification objects.

Functions:

from_dict(spec_dict)	Reconstitute a spec object from a dict created by *BaseSpec.to_dict()*

class onice_conversion.spec.base_spec.**BaseSpec**(*retype: Optional[Callable] = None*, **args*, ***kwargs*)

Bases: abc.ABC, *onice_conversion.utils.IntrospectionMixin*

Base class for specification objects.

Abstract, should not be instantiated on its own.

Parameters **retype** (*Callable (optional)*)

Methods:

__init__([retype])	**Parameters** **retype** (*Callable (optional)*)
parse(base_path[, metadata])	Parse all parameters from self and child *_parse()* methods, combining into single dictionary
_parse([base_path, metadata])	All Specs should instantiate a _parse method that returns a dictionary of metadata variable keys and values. eg::.
children()	Generator for iterating over children (added)
to_dict()	Get a dictionary description of this spec object, of the form.
_expand_named_fields(named_fields)	Convert nested key specs like key[key2] into nested dicts
_full_name()	Returns the full module and class name of an object, eg.
_get_init_args()	introspect object and get all arguments passed on __init__

Attributes:

specifies	Which metadata variables are specified by this Spec object and its children
parent	

__init__(*retype: Optional[Callable] = None*, **args*, ***kwargs*)

Parameters **retype** (*Callable (optional)*)

parse(*base_path: pathlib.Path*, *metadata: Optional[dict] = None*) → dict

Parse all parameters from self and child *_parse()* methods, combining into single dictionary

Parameters

- **base_path** (*Path*) – The base path we compute the spec'd value from!
- **metadata** (*dict*) – other metadata used by the parsing function, usually passed in *NWBConverter.run_conversion()*

abstract _parse(*base_path=None*, *metadata: Optional[dict] = None*) → dict

All Specs should instantiate a _parse method that returns a dictionary of metadata variable keys and values. eg:

```
>>> BaseSpec().parse()
{ 'subject_id': 'jonny' }
```

The typical use is to be able to specify some metadata values that are contained ***somewhere*** relative to a directory of data, so the passed argument should typically be that directory.

property specifies: Tuple[str, ...]

Which metadata variables are specified by this Spec object and its children

Return type tuple of strings

property parent: *onice_conversion.spec.base_spec.BaseSpec*

children() → Iterable[*onice_conversion.spec.base_spec.BaseSpec*]

Generator for iterating over children (added)

to_dict() → dict

Get a dictionary description of this spec object, of the form:

```
{
    'module': self.__module__,
    'class': type(self).__name__,
    'kwargs': self._init_args,
    'children': [ ... same structure as top-level without children list ...]
}
```

That allows a spec to be reconstituted with `from_dict()`

Return type dict of initialization parameters, as described above

_expand_named_fields(*named_fields*)

Convert nested key specs like key[key2] into nested dicts

borroed from: https://github.com/r1chardj0n3s/parse/blob/0477aa58673cd957c19d377e029347ce72c08b1b/parse.py#L944

_full_name()

Returns the full module and class name of an object, eg. `nwb_conversion_tools.spec.external_file.JSON`

Return type str

_get_init_args()

introspect object and get all arguments passed on __init__

depends on introspecting up frames so should only be called *during* the top-level __init__ of the base class :)

Return type dict of argument names and params

onice_conversion.spec.base_spec.**from_dict**(*spec_dict: dict*) → *onice_conversion.spec.base_spec.BaseSpec*

Reconstitute a spec object from a dict created by *BaseSpec.to_dict()*

Parameters **spec_dict** (*dict*) – A dictionary created by `BaseSpect.to_dict()`

Return type The reconstituted spec object!

CHAPTER

SIX

PYNWB CONTAINERS

Tools for making an abstract interface to work with pynwb Containers

Functions:

get_container([container_name])	Get and list pyNWB containers by name.
get_container_schema(container)	Get argument schema for a pyNWB container.

onice_conversion.containers.**get_container**(*container_name: Optional[str] = None*) → Union[List[pynwb.core.NWBContainer], pynwb.core.NWBContainer]

Get and list pyNWB containers by name.

If called with no arguments, returns all container objects. Otherwise return the container named `'container_name'`.

Eg. get `pynwb.file.NWBFile` by calling with `'NWBFile'`

Parameters **container_name** (*str, None*) – if None, return all containers. Otherwise return container by name

Returns list of Containers, or Container itself.

onice_conversion.containers.**get_container_schema**(*container: Union[pynwb.core.NWBContainer, str]*) → Tuple[dict]

Get argument schema for a pyNWB container.

Parameters **container** (`pynwb.NWBContainer`, str) – Either the container itself, or a string to call *get_container()* with

Returns tuple of dicts that describe each parameter

Classes:

ContainerMixin()	Mixin to give pynwb containers the classmethods expected by nwb-conversion-tools

class onice_conversion.containers.container.**ContainerMixin**

Bases: `object`

Mixin to give pynwb containers the classmethods expected by nwb-conversion-tools

CHAPTER

SEVEN

UTILS

Utility functions used internally across the library

Exceptions:

AmbiguityError	Exception type for when *onice_conversion.spec* modules give ambiguous results

Classes:

IntrospectionMixin()	Mixin to allow objects to become aware of all the arguments they were called with on initialization

Functions:

_recurse_subclasses(cls[, leaves_only])	Given some class, find its subclasses recursively
_recursive_import(module_name)	Given some path in a python package, import all modules beneath it
_gather_list_of_dicts(a_list)	Gather a list of dictionaries like.
_recursive_dedupe_dicts(a_dict[, raise_on_dupes])	Deduplicate a list of dicts.

exception onice_conversion.utils.**AmbiguityError**

Bases: Exception

Exception type for when *onice_conversion.spec* modules give ambiguous results

class onice_conversion.utils.**IntrospectionMixin**

Bases: object

Mixin to allow objects to become aware of all the arguments they were called with on initialization

Call *_get_init_args()* in the __init__ method of any object that inherits from this mixin :)

Methods:

_get_init_args()	introspect object and get all arguments passed on __init__
_full_name()	Returns the full module and class name of an object, eg.

_get_init_args()

introspect object and get all arguments passed on __init__

depends on introspecting up frames so should only be called *during* the top-level __init__ of the base class :)

Return type dict of argument names and params

_full_name()

Returns the full module and class name of an object, eg. `nwb_conversion_tools.spec.external_file.JSON`

Return type str

onice_conversion.utils.**_recurse_subclasses**(*cls*, *leaves_only=True*) → list

Given some class, find its subclasses recursively

See: https://stackoverflow.com/a/17246726/13113166

Parameters

- **leave_only** (*bool*) – If True, only include classes that have no further subclasses,
- **if False, return all subclasses.**

Returns list of subclasses

onice_conversion.utils.**_recursive_import**(*module_name: str*) → List[str]

Given some path in a python package, import all modules beneath it

Parameters **module_name** (*str*) – name of module to recursively import

Returns list of all modules that were imported

onice_conversion.utils.**_gather_list_of_dicts**(*a_list: list*) → Dict[str, list]

Gather a list of dictionaries like:

```
[{'key1':'val1'}, {'key1':'val2'}, {'key1':'val3'}]
```

to a dict of lists like:

{'key1': ['val1', 'val2', 'val3']}

onice_conversion.utils.**_recursive_dedupe_dicts**(*a_dict*, *raise_on_dupes=True*)

Deduplicate a list of dicts.

Optionally raise an exception if duplicates are found, otherwise call `set` and unwrap singletons and return

Parameters **a_dict** (*of dicts*)

Returns **dict**

Return type deduplicated dictionary

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

O

INDEX

Symbols

T

Y