Flywheel Tools

Release 0.3.0

Mar 03, 2022

Contents:

`FlywheelTools` is a suite of software tools for curating your data into BIDS on Flywheel. It's comprised of 2 parts:

`fw-heudiconv`, which is a Python-based tool kit for curating BIDS data on the Flywheel platform, and `flaudit`, which is a Flywheel project auditor.

Full documentation at readthedocs

License: BS3-D

CHAPTER 1

Contents

1.1 BIDS & FlywheelTools Explained

FlywheelTools consists of two tools — `fw-heudiconv` and `flaudit`.

Flywheel HeuDiConv (`fw-heudiconv`) is based on the popular `heudiconv` software, "flexible DICOM converter for organizing brain imaging data into structured directory layouts" [source]. Like `heudiconv`, `fw-heudiconv` makes use of a user-defined **heuristic** — a discrete set of rules — to standardise naming conventions within the user's project directory into BIDS. This is particularly useful in Flywheel, where curation, pre-processing, and analyses are all automated in the context of BIDS.

BIDS stands for Brain Imaging Data Structure, and is a standard format for organizing and storing brain imaging data. It's an organisational standard that makes it easy to share, collaborate on, and analyse your data. Among many other details, BIDS prioritizes having imaging scans located in a nested directory structure, where the top level is the subject label, followed by the imaging session label, followed by the modality of measurement. Each imaging file (typically NIfTI format) must have an associated *.json* dictionary, known as a sidecar, that stores the image's metadata. Additionally, filenames must adhere to a `{key}-{value}.{extension}` naming convention, where key-value pairs are separated by underscores _. This makes files very easy to parse by eye, hand, or machine, and if a BIDS dataset is valid, it can be used as seamless input into a myriad of BIDS-ready processing and analysis pipelines, officially called BIDS apps.

Learn more about BIDS here.

The goal of `fw-heudiconv` is to provide researchers with a tool to flexibly and reproducibly curate their Flywheel datasets into BIDS, that is as powerful as it is easy to use.

When you're done, you can use `flaudit` to audit your data and make sure everything went as planned. `flaudit` is a containerized gear on Flywheel that loops over the data in your project, and collects data about the sequences that exist in the project, if and how they were curated into BIDS, and if analysis gears were run on them. One important feature is the inclusion of a *template subject*. When specified, `flaudit` will use that subject's data as a gold standard and compare each other subject to them, highlighting:

- If the subject has collected the same scanning sequences as the template
- If the subject has been curated into BIDS identically to the template

- If the subject has run the same analysis gears as the template. This comparison is sensitive to gear versions, so you can be sure your subjects ran the same version of algorithms as the template.

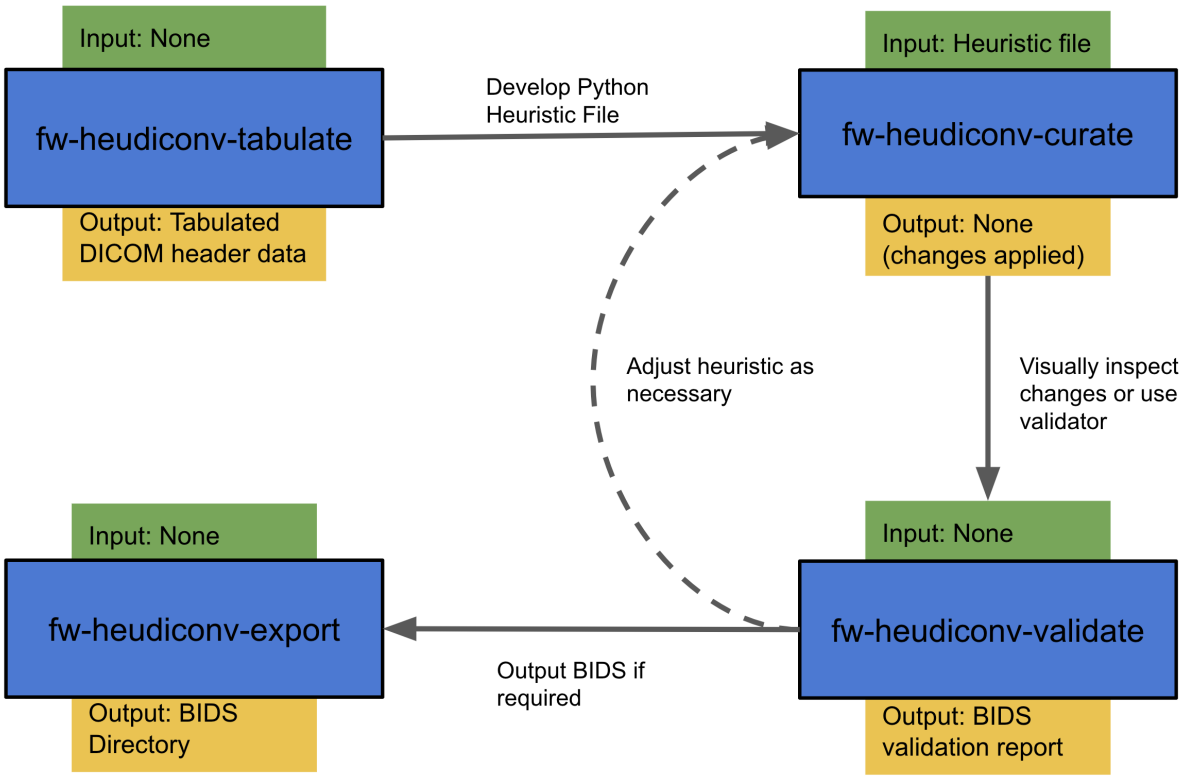
The output of the gear is an HTML report that can be opened in your web browser, as well as the accompanying CSVs that generated it, in case you want to explore the data further.

1.1.1 General Workflow

In `fw-heudiconv`, the general workflow is as follows:

1. Extract DICOM header information from your data with the `tabulate` tool
2. Craft a heuristic that can: a) create BIDS-valid filenames, paths, and metadata b) parse the DICOMs to decide which NIfTIs are assigned to which BIDS names
3. Test & apply these changes on Flywheel with the `curate` tool
4. Adjust the heuristic as necessary and repeat testing and application
5. Validate and/or export your BIDS-valid data (`validate` or `export` tools)

The workflow is illustrated below:



This workflow can also be accomplished on Flywheel through the GUI without the need for command line tools. Each of the tools is documented in the next section.

1.2 Installation

`fw-heudiconv` can be run in the Flywheel GUI as a gear, or locally using the Command Line Interface distributed using *pip*. `flaudit` can *only* be run in the Flywheel GUI.

Note: FlywheelTools are intended for use with a Flywheel site. At the University of Pennsylvania, our site is available at upenn.flywheel.io.

To use locally, follow instructions below to set up your system for using `fw-heudiconv` on your machine:

Estimated time: 15 minutes

1.2.1 Install & start up Miniconda

First, get a package management system. Recommended is miniconda (conda): **Conda quickly installs, runs and updates packages and their dependencies.**

https://docs.conda.io/en/latest/miniconda.html.

You can check if you have this successfully by going to the terminal and doing:

```
$ which conda [macOS]
```

1.2.2 Start a virtual environment

Use miniconda to create a virtual environment, a restricted workspace where your programs and processes can operate without affecting everything on your computer. Create an environment called flywheel, in the terminal:

```
$ conda create -n flywheel anaconda python=3
```

At the prompt for which packages to install, type y and hit enter. It's better to have them all, and they will not take up a lot of space on your machine:

```
:
:
:
$ wurlitzer          pkgs/main/osx-64::wurlitzer-1.0.2-py37_0
$ xlrd               pkgs/main/osx-64::xlrd-1.2.0-py37_0
$ xlsxwriter         pkgs/main/noarch::xlsxwriter-1.1.8-py_0
$ xlwings            pkgs/main/osx-64::xlwings-0.15.8-py37_0
$ xlwt               pkgs/main/osx-64::xlwt-1.3.0-py37_0
$ xz                 pkgs/main/osx-64::xz-5.2.4-h1de35cc_4
$ yaml               pkgs/main/osx-64::yaml-0.1.7-hc338f04_2
$ zeromq             pkgs/main/osx-64::zeromq-4.3.1-h0a44026_3
$ zict               pkgs/main/noarch::zict-1.0.0-py_0
$ zipp               pkgs/main/noarch::zipp-0.5.1-py_0
$ zlib               pkgs/main/osx-64::zlib-1.2.11-h1de35cc_3
$ zstd               pkgs/main/osx-64::zstd-1.3.7-h5bba6e5_0


$ Proceed ([y]/n)?
```

Activate your environment, so that any packages you install or use stay restricted to this project:

```
$ source activate flywheel
```

1.2.3 Download `fw-heudiconv` from pip

The `fw-heudiconv` code is hosted on pip: **pip is a standard package-management system used to install and manage software packages written in Python**

Pip should be installed with your new environment, but you can ensure you have it by running:

```
$ which pip
```

Now, use pip to install `fw-heudiconv`:

```
$ pip install fw-heudiconv
```

1.2.4 Download the Flywheel SDK & CLI

You will need to download the flywheel software development kit in order to use `fw-heudiconv`. Follow the instructions here to install, or run:

```
$ pip install flywheel-sdk
```

The flywheel CLI allows `fw-heudiconv` (or any other program you write) to communicate with Flywheel's database. Follow their instructions here to download and login.

Once installed and logged in, you should see your username when you run the following:

```
$ fw status
$ You are currently logged in as Tinashe Tapera to https://upenn.flywheel.io
```

1.2.5 Updating `fw-heudiconv`

If you already have `fw-heudiconv` and wish to update to the latest version, just run:

```
$ pip install --upgrade fw-heudiconv
```

1.2.6 Appendix — `fw-heudiconv-validate`

`fw-heudiconv-validate` is a convenience tool that wraps the official Bids Validator and pipes the output of `fw-heudiconv-export` to it. It's most useful for validating Flywheel data through a gear on the GUI.

To use `fw-heuduiconv-validate` on your local machine, you need to install node.js. This is not necessary, however, and instead you are welcome to use `fw-heudiconv-validate` on your Flywheel site, or, use `fw-heudiconv-export` to export data first, and then use the official Bids Validator available here.

1.3 Quick start guide

1.3.1 `fw-heudiconv` at the Command Line

1. Make sure you have the Flywheel CLI and SDK installed. Note that this is `flywheel-sdk` and **NOT** `flywheel`.
2. Download the package from pip with `pip install fw-heudiconv`.

3. Tabulate your DICOM header info with `fw-heudiconv-tabulate --project <MY_PROJECT> --path <OUTPUT_PATH>`.
4. Design your *heuristic* file in Python.
5. **Curate your dataset** on Flywheel into BIDS: `fw-heudiconv-curate --project <MY_PROJECT> --heuristic <MY_HEURISTIC_FILE>`. Use the flag `--dry_run` to test your heuristic.

To **export the dataset** to your machine: `fw-heudiconv-export --project <MY_PROJECT> --path <OUTPUT_DIR>`.

Use the `--subject <SUBJECT>` and/or `--session <SESSION>` flags to use the tool on specific subject/session labels.

1.3.2 `fw-heudiconv` as a Flywheel Gear

1. Have the *Flywheel Heudiconv* gear installed on your Flywheel instance.
2. Design your heuristic file in Python and upload it to your project.
3. Run the *Flywheel Heudiconv* gear (accessible through *Run Gear -> Analysis Gear*), using your heuristic file as the input.

To **list the sequence information** for your dataset: In the "Gear Configuration" window, type "Tabulate" in the "Action" field, which will leave a sequence info table (.tsv) in the gear's outputs.

To **curate the dataset** into BIDS: In the "Gear Configuration" window, type "Curate" in the "Action" field, which will curate the dataset for BIDS. Click the "dry run" button to test your heuristic first (the gear log will print out all the changes without applying them).

To **export the dataset** into BIDS for downloading to your machine: In the "Gear Configuration" window, type "Export" in the "Action" field, which will leave an exported BIDS dataset in the gear's outputs. Click the "dry run" button to test your output first (the gear log will print out the expected directory structure of an export).

1.3.3 `flaudit` as a Flywheel Gear

1. Have the *flaudit* gear installed on your Flywheel instance.
2. Select a *template subject* – an ideal subject who you know has been curated correctly and has had all the required preprocessing gears run on them. Input this subject ID in the `Template` field in the config.
3. Hit Run!

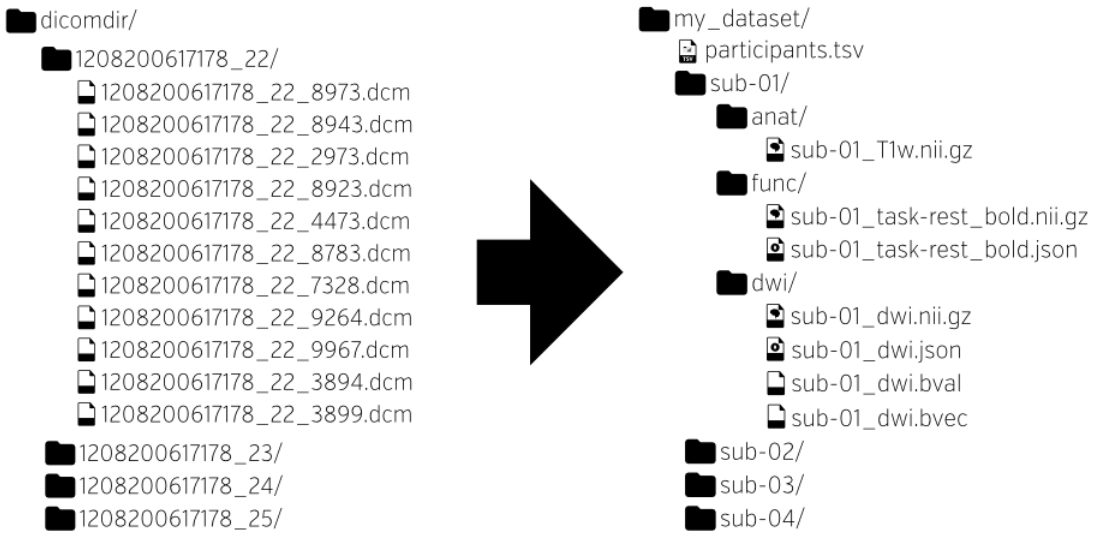
1.4 Step-By-Step Guide

Let's walk through an example of how to curate some simple data on Flywheel. For this tutorial, you will at the very least need access to Flywheel and a text editor.

1.4.1 Step 1: Understanding Your Dataset in the Context of BIDS

Before you can curate the dataset into BIDS, it's important to be able to predict how your dataset should look in BIDS. If you don't know what BIDS is, check the official readthedocs.

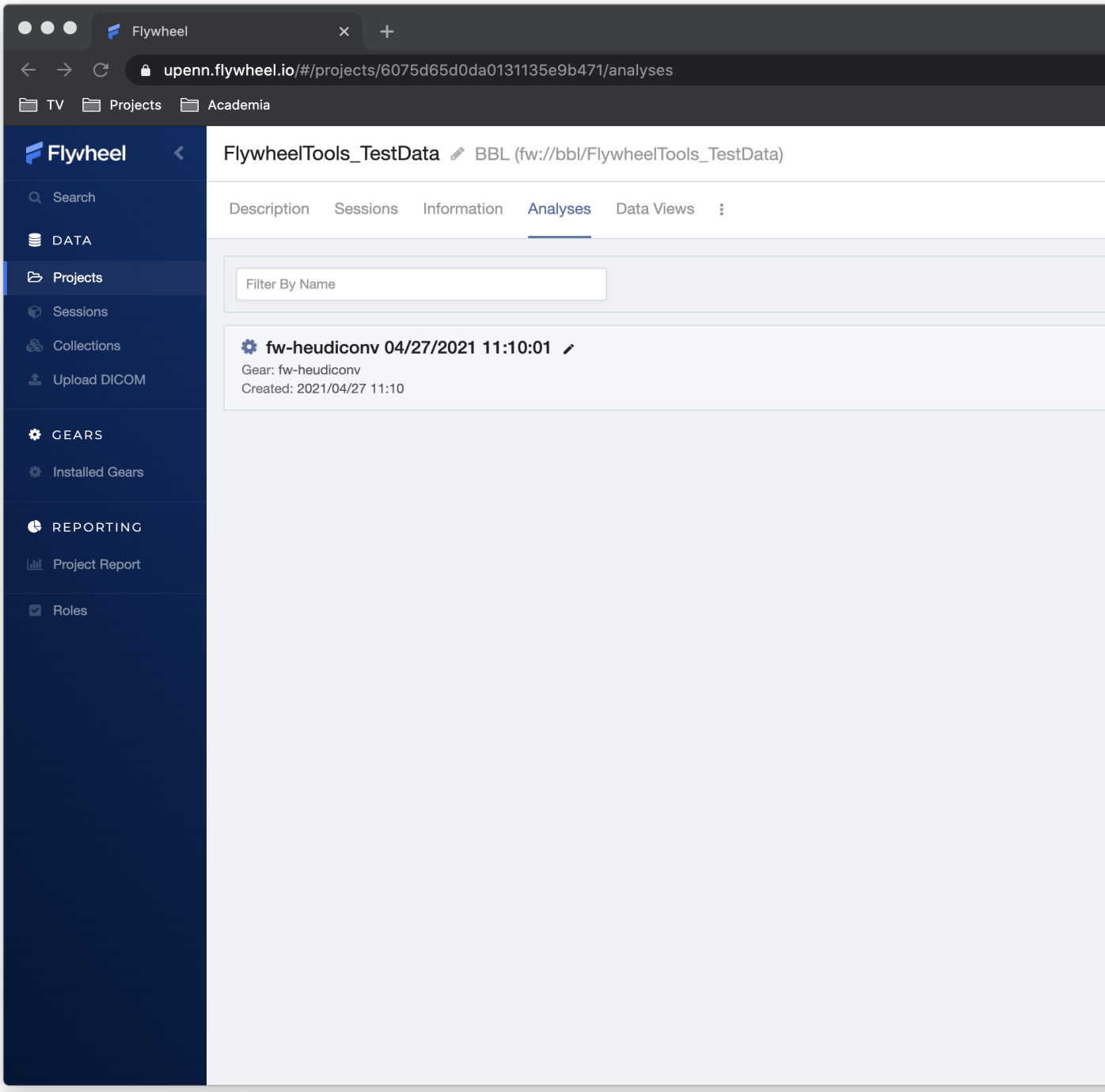
Our goal here will be to map DICOMs to NIfTIs named correctly in BIDS, including the directory structure, correct metadata sidecars, and fieldmap files:



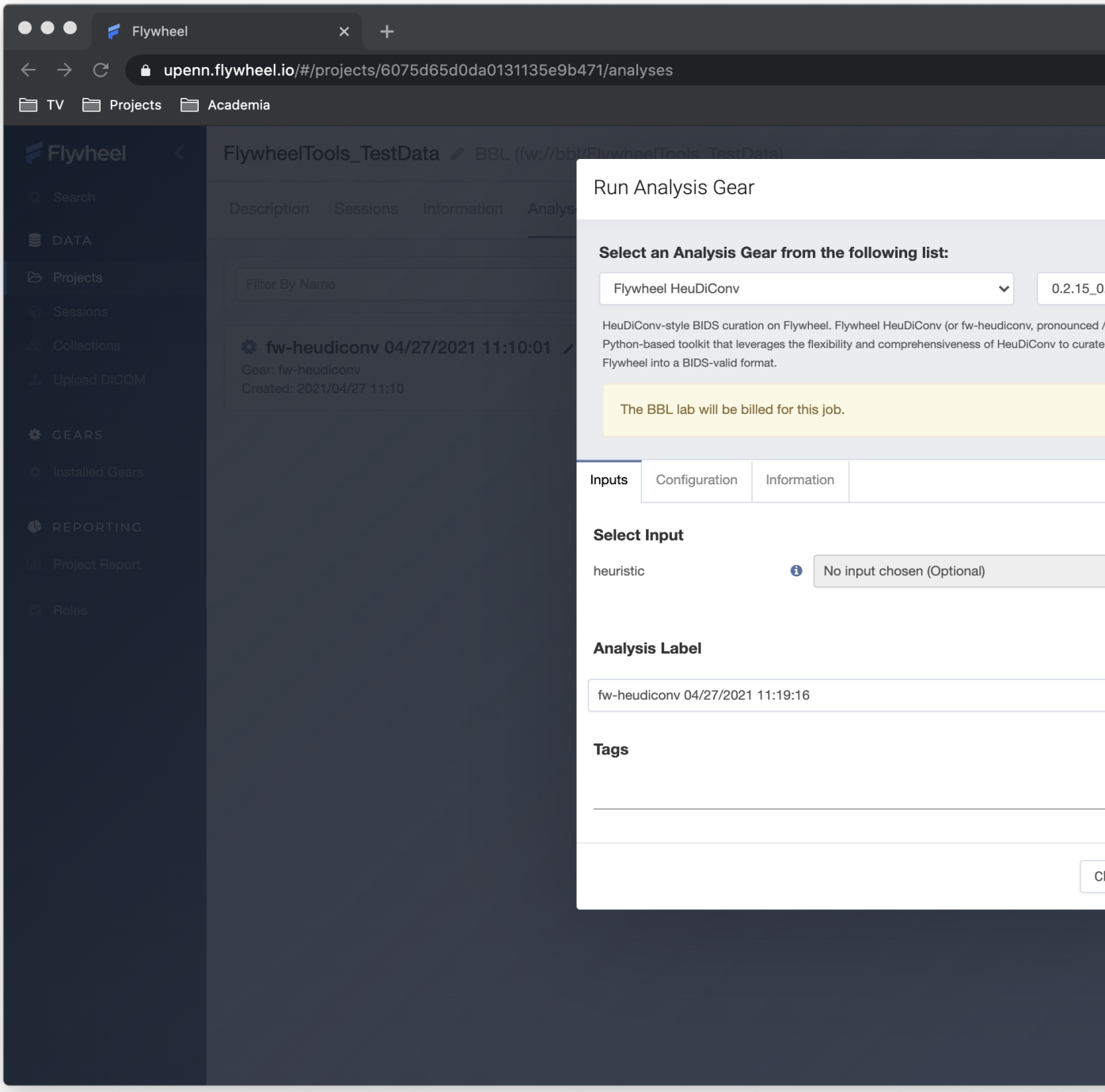
To start, we need to figure out what we can use to create this "mapping". In `fw-heudiconv` curation, this mapping is called a heuristic, and we'll use the DICOMs' header information to create rules for this mapping. To extract this information, we will use `fw-heudiconv-tabulate` to generate a *seqinfo table*.

1.4.2 `fw-heudiconv-tabulate`

In the Flywheel GUI, navigate to your project, and select the "Analysis" tab:



Click the "Run Analysis Gear" button, which will drop down the analysis box. In this box, select `Flywheel HeuDiConv` as the gear to run the analysis.



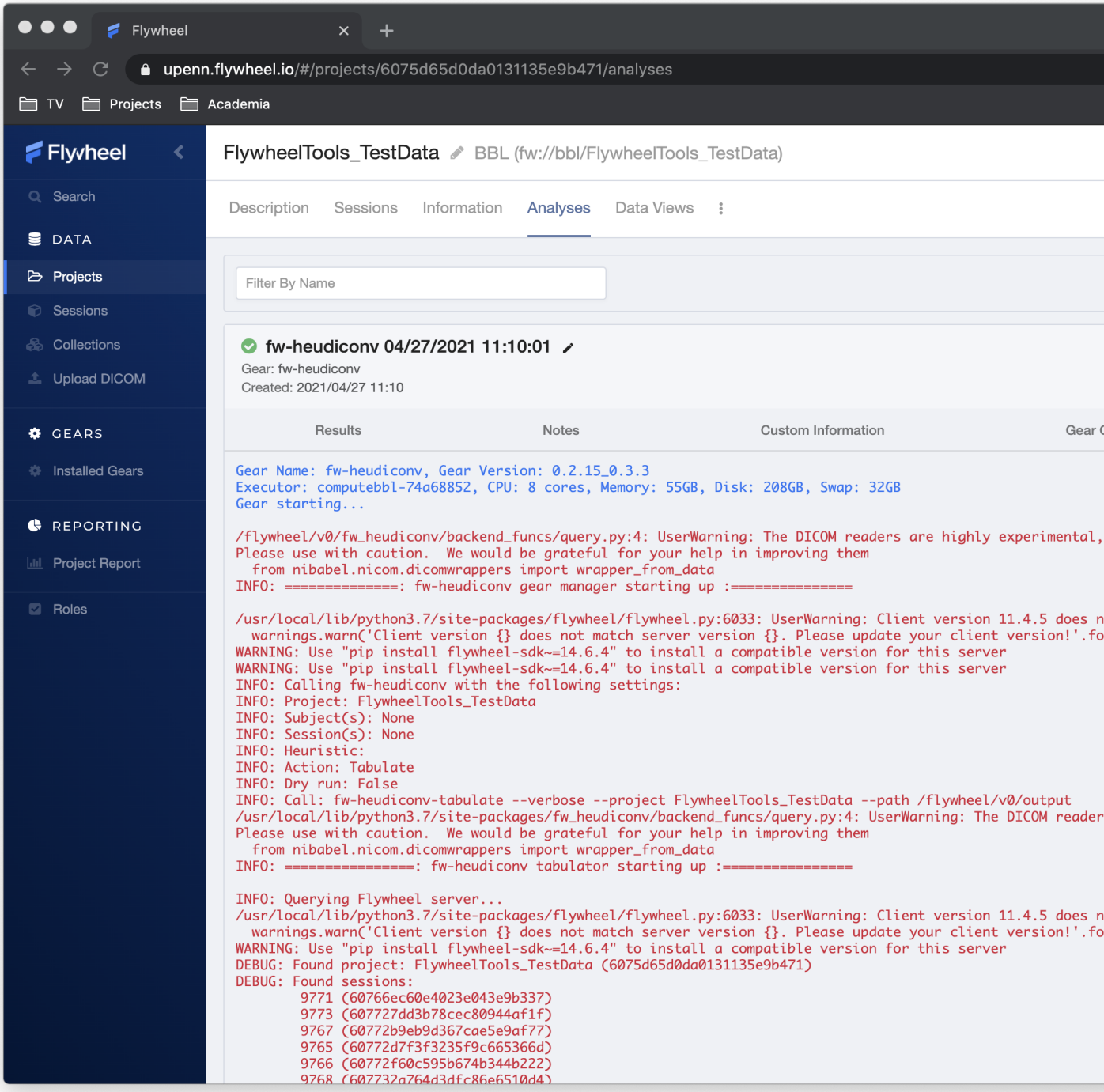
From here, click the "Configuration" tab (there are no inputs required at this stage). This will allow you to set the configuration for the gear. Under "Action", select "Tabulate", and make sure to *uncheck* `dry_run`. When ready, hit "Run Gear"!

The same can be accomplished at the command line, with this command:

```
fw-heudiconv-tabulate --project FlywheelTools_TestData --path MY/OUTPUT/DIRECTORY/
```

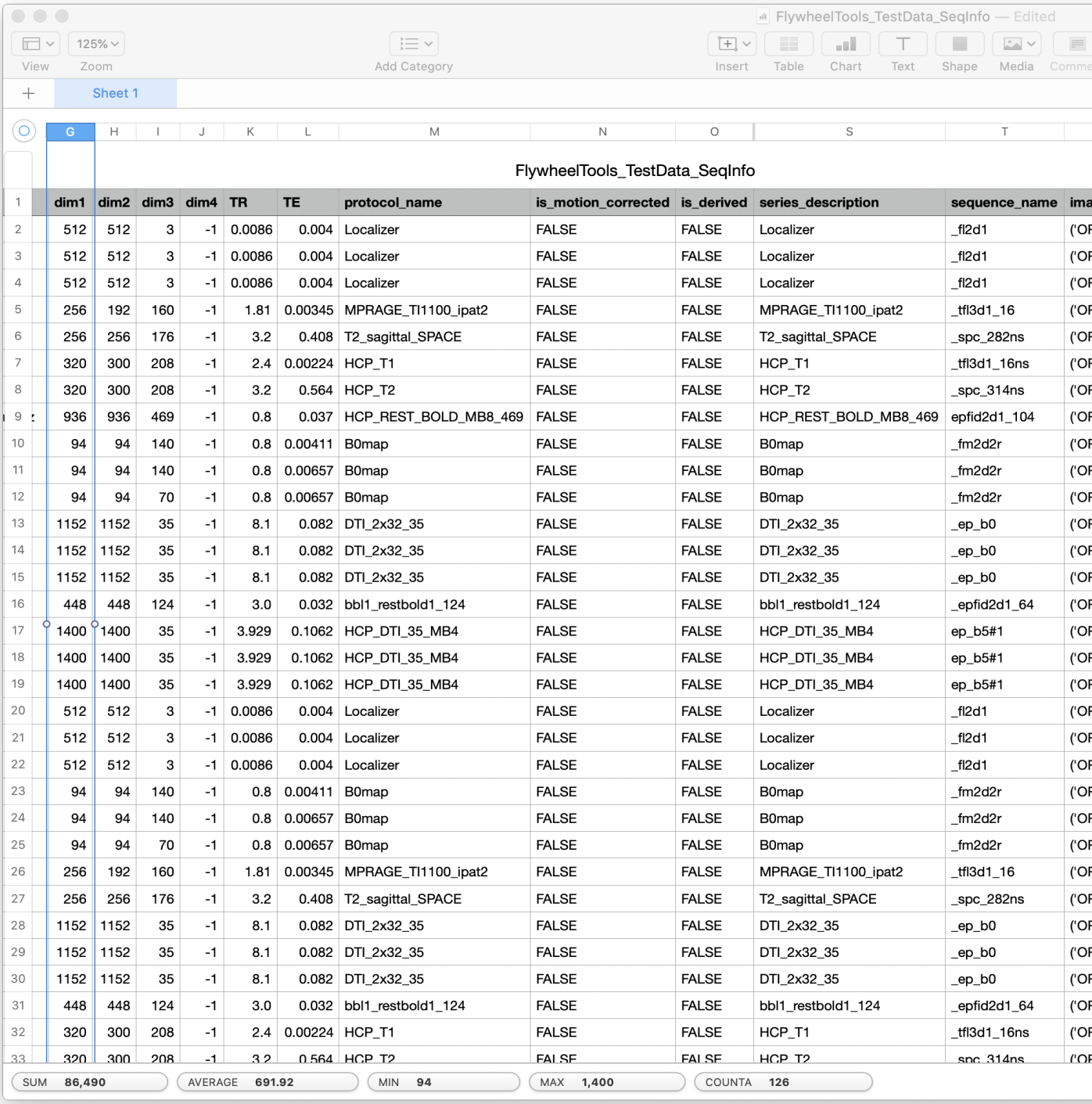
The Output

You should now see an analysis object appear in the GUI. This analysis object is associated with the project, since we started it at the project level. If a blue gear is spinning, the gear is still running (this can include virtual machine initialization and shut down time); a red X means it failed, but a green check means success! You should be able to check the "Gear Logs" in the analysis object to read through `stdout` (all the commands and outputs) as the gear ran.



In the Results section of the analysis, Flywheel zips all the data it was instructed to save as outputs – in this case, the

result of our tabulation. Download this file and unzip it, afterwhich you can open it in your table viewer or text editor of choice.



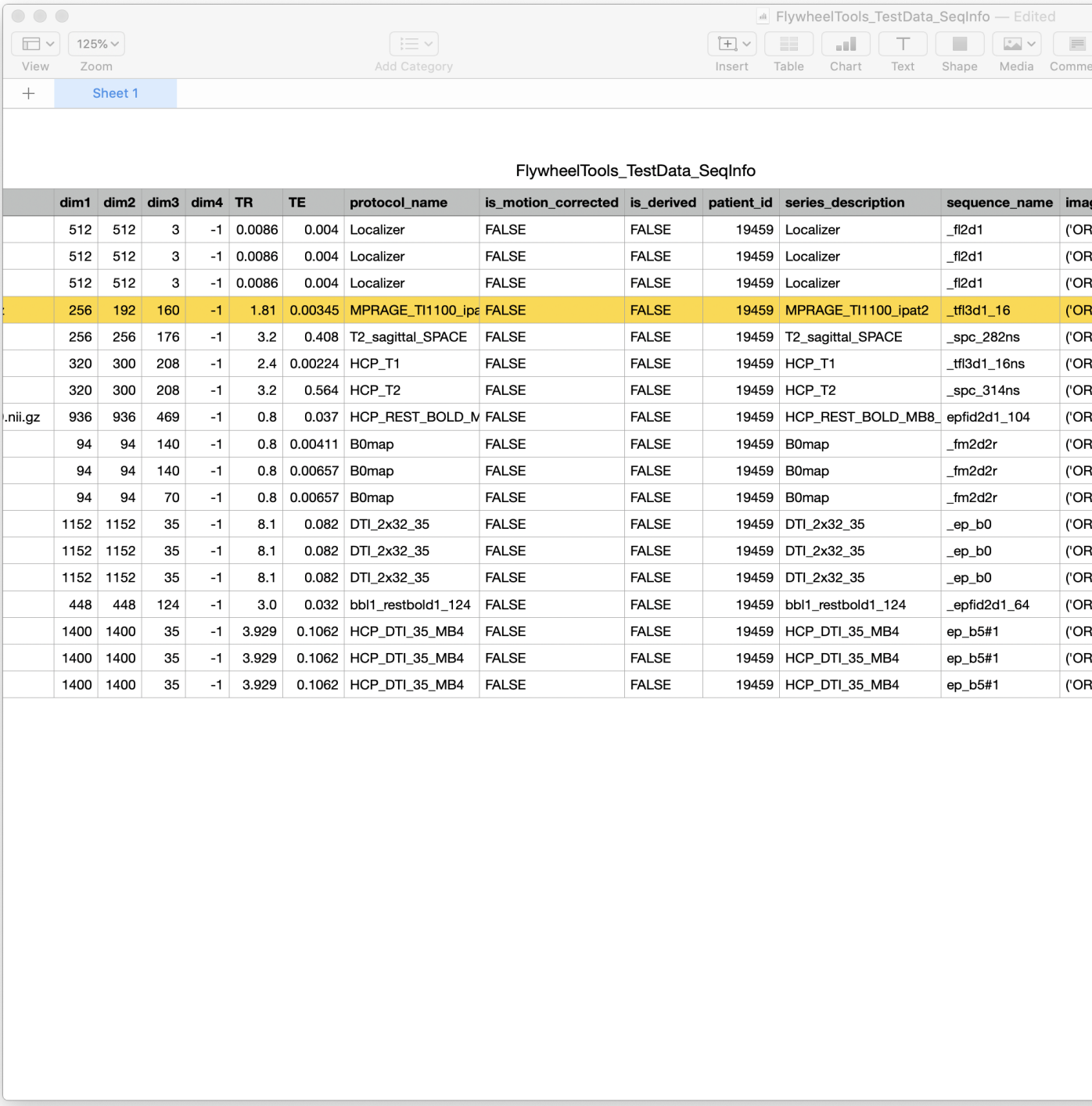
FlywheelTools_TestData_SeqInfo

dim1	dim2	dim3	dim4	TR	TE	protocol_name	is_motion_corrected	is_derived	series_description	sequence_name	ima
512	512	3	-1	0.0086	0.004	Localizer	FALSE	FALSE	Localizer	_fl2d1	('OR
512	512	3	-1	0.0086	0.004	Localizer	FALSE	FALSE	Localizer	_fl2d1	('OR
512	512	3	-1	0.0086	0.004	Localizer	FALSE	FALSE	Localizer	_fl2d1	('OR
256	192	160	-1	1.81	0.00345	MPRAGE_TI1100_ipat2	FALSE	FALSE	MPRAGE_TI1100_ipat2	_tfl3d1_16	('OR
256	256	176	-1	3.2	0.408	T2_sagittal_SPACE	FALSE	FALSE	T2_sagittal_SPACE	_spc_282ns	('OR
320	300	208	-1	2.4	0.00224	HCP_T1	FALSE	FALSE	HCP_T1	_tfl3d1_16ns	('OR
320	300	208	-1	3.2	0.564	HCP_T2	FALSE	FALSE	HCP_T2	_spc_314ns	('OR
936	936	469	-1	0.8	0.037	HCP_REST_BOLD_MB8_469	FALSE	FALSE	HCP_REST_BOLD_MB8_469	epfid2d1_104	('OR
94	94	140	-1	0.8	0.00411	B0map	FALSE	FALSE	B0map	_fm2d2r	('OR
94	94	140	-1	0.8	0.00657	B0map	FALSE	FALSE	B0map	_fm2d2r	('OR
94	94	70	-1	0.8	0.00657	B0map	FALSE	FALSE	B0map	_fm2d2r	('OR
1152	1152	35	-1	8.1	0.082	DTI_2x32_35	FALSE	FALSE	DTI_2x32_35	_ep_b0	('OR
1152	1152	35	-1	8.1	0.082	DTI_2x32_35	FALSE	FALSE	DTI_2x32_35	_ep_b0	('OR
1152	1152	35	-1	8.1	0.082	DTI_2x32_35	FALSE	FALSE	DTI_2x32_35	_ep_b0	('OR
448	448	124	-1	3.0	0.032	bbl1_restbold1_124	FALSE	FALSE	bbl1_restbold1_124	_epfid2d1_64	('OR
1400	1400	35	-1	3.929	0.1062	HCP_DTI_35_MB4	FALSE	FALSE	HCP_DTI_35_MB4	ep_b5#1	('OR
1400	1400	35	-1	3.929	0.1062	HCP_DTI_35_MB4	FALSE	FALSE	HCP_DTI_35_MB4	ep_b5#1	('OR
1400	1400	35	-1	3.929	0.1062	HCP_DTI_35_MB4	FALSE	FALSE	HCP_DTI_35_MB4	ep_b5#1	('OR
512	512	3	-1	0.0086	0.004	Localizer	FALSE	FALSE	Localizer	_fl2d1	('OR
512	512	3	-1	0.0086	0.004	Localizer	FALSE	FALSE	Localizer	_fl2d1	('OR
512	512	3	-1	0.0086	0.004	Localizer	FALSE	FALSE	Localizer	_fl2d1	('OR
94	94	140	-1	0.8	0.00411	B0map	FALSE	FALSE	B0map	_fm2d2r	('OR
94	94	140	-1	0.8	0.00657	B0map	FALSE	FALSE	B0map	_fm2d2r	('OR
94	94	70	-1	0.8	0.00657	B0map	FALSE	FALSE	B0map	_fm2d2r	('OR
256	192	160	-1	1.81	0.00345	MPRAGE_TI1100_ipat2	FALSE	FALSE	MPRAGE_TI1100_ipat2	_tfl3d1_16	('OR
256	256	176	-1	3.2	0.408	T2_sagittal_SPACE	FALSE	FALSE	T2_sagittal_SPACE	_spc_282ns	('OR
1152	1152	35	-1	8.1	0.082	DTI_2x32_35	FALSE	FALSE	DTI_2x32_35	_ep_b0	('OR
1152	1152	35	-1	8.1	0.082	DTI_2x32_35	FALSE	FALSE	DTI_2x32_35	_ep_b0	('OR
1152	1152	35	-1	8.1	0.082	DTI_2x32_35	FALSE	FALSE	DTI_2x32_35	_ep_b0	('OR
448	448	124	-1	3.0	0.032	bbl1_restbold1_124	FALSE	FALSE	bbl1_restbold1_124	_epfid2d1_64	('OR
320	300	208	-1	2.4	0.00224	HCP_T1	FALSE	FALSE	HCP_T1	_tfl3d1_16ns	('OR
320	300	208	-1	3.2	0.564	HCP_T2	FALSE	FALSE	HCP_T2	_spc_314ns	('OR

Next, we're going to use this table to curate one of the subjects. Fortunately, in the table viewer, we can use a filter to only show data from one subject. Here, we pick subject 019465 using the `patient_id` column.

1.4.3 Developing a Simple Heuristic

To start, open up any text editor, such as Notepad or TextEdit. We're going to start by curating the anatomical T1w image, whose DICOM is highlighted here:



FlywheelTools_TestData_SeqInfo

	dim1	dim2	dim3	dim4	TR	TE	protocol_name	is_motion_corrected	is_derived	patient_id	series_description	sequence_name	imag
	512	512	3	-1	0.0086	0.004	Localizer	FALSE	FALSE	19459	Localizer	_fl2d1	('OR
	512	512	3	-1	0.0086	0.004	Localizer	FALSE	FALSE	19459	Localizer	_fl2d1	('OR
	512	512	3	-1	0.0086	0.004	Localizer	FALSE	FALSE	19459	Localizer	_fl2d1	('OR
	256	192	160	-1	1.81	0.00345	MPRAGE_TI1100_ipa	FALSE	FALSE	19459	MPRAGE_TI1100_ipat2	_tfl3d1_16	('OR
	256	256	176	-1	3.2	0.408	T2_sagittal_SPACE	FALSE	FALSE	19459	T2_sagittal_SPACE	_spc_282ns	('OR
	320	300	208	-1	2.4	0.00224	HCP_T1	FALSE	FALSE	19459	HCP_T1	_tfl3d1_16ns	('OR
	320	300	208	-1	3.2	0.564	HCP_T2	FALSE	FALSE	19459	HCP_T2	_spc_314ns	('OR
.nii.gz	936	936	469	-1	0.8	0.037	HCP_REST_BOLD_M	FALSE	FALSE	19459	HCP_REST_BOLD_MB8_	epfid2d1_104	('OR
	94	94	140	-1	0.8	0.00411	B0map	FALSE	FALSE	19459	B0map	_fm2d2r	('OR
	94	94	140	-1	0.8	0.00657	B0map	FALSE	FALSE	19459	B0map	_fm2d2r	('OR
	94	94	70	-1	0.8	0.00657	B0map	FALSE	FALSE	19459	B0map	_fm2d2r	('OR
	1152	1152	35	-1	8.1	0.082	DTI_2x32_35	FALSE	FALSE	19459	DTI_2x32_35	_ep_b0	('OR
	1152	1152	35	-1	8.1	0.082	DTI_2x32_35	FALSE	FALSE	19459	DTI_2x32_35	_ep_b0	('OR
	1152	1152	35	-1	8.1	0.082	DTI_2x32_35	FALSE	FALSE	19459	DTI_2x32_35	_ep_b0	('OR
	448	448	124	-1	3.0	0.032	bbl1_restbold1_124	FALSE	FALSE	19459	bbl1_restbold1_124	_epfid2d1_64	('OR
	1400	1400	35	-1	3.929	0.1062	HCP_DTI_35_MB4	FALSE	FALSE	19459	HCP_DTI_35_MB4	ep_b5#1	('OR
	1400	1400	35	-1	3.929	0.1062	HCP_DTI_35_MB4	FALSE	FALSE	19459	HCP_DTI_35_MB4	ep_b5#1	('OR
	1400	1400	35	-1	3.929	0.1062	HCP_DTI_35_MB4	FALSE	FALSE	19459	HCP_DTI_35_MB4	ep_b5#1	('OR

In-depth knowledge of these functions is not necessary for this tutorial, but see *The Heuristic File* if you want to understand each of them in earnest. First, copy and paste the `create_key()` function into a new file in your text

editor:

```
def create_key(template, outtype=('nii.gz',), annotation_classes=None):
    if template is None or not template:
        raise ValueError('Template must be a valid format string')
    return template, outtype, annotation_classes
```

Next, use this function to create a BIDS name for the T1w NIfTI you want:

```
def create_key(template, outtype=('nii.gz',), annotation_classes=None):
    if template is None or not template:
        raise ValueError('Template must be a valid format string')
    return template, outtype, annotation_classes

# Create Keys
t1w = create_key(
   'sub-{subject}/ses-{session}/anat/sub-{subject}_ses-{session}_T1w')
```

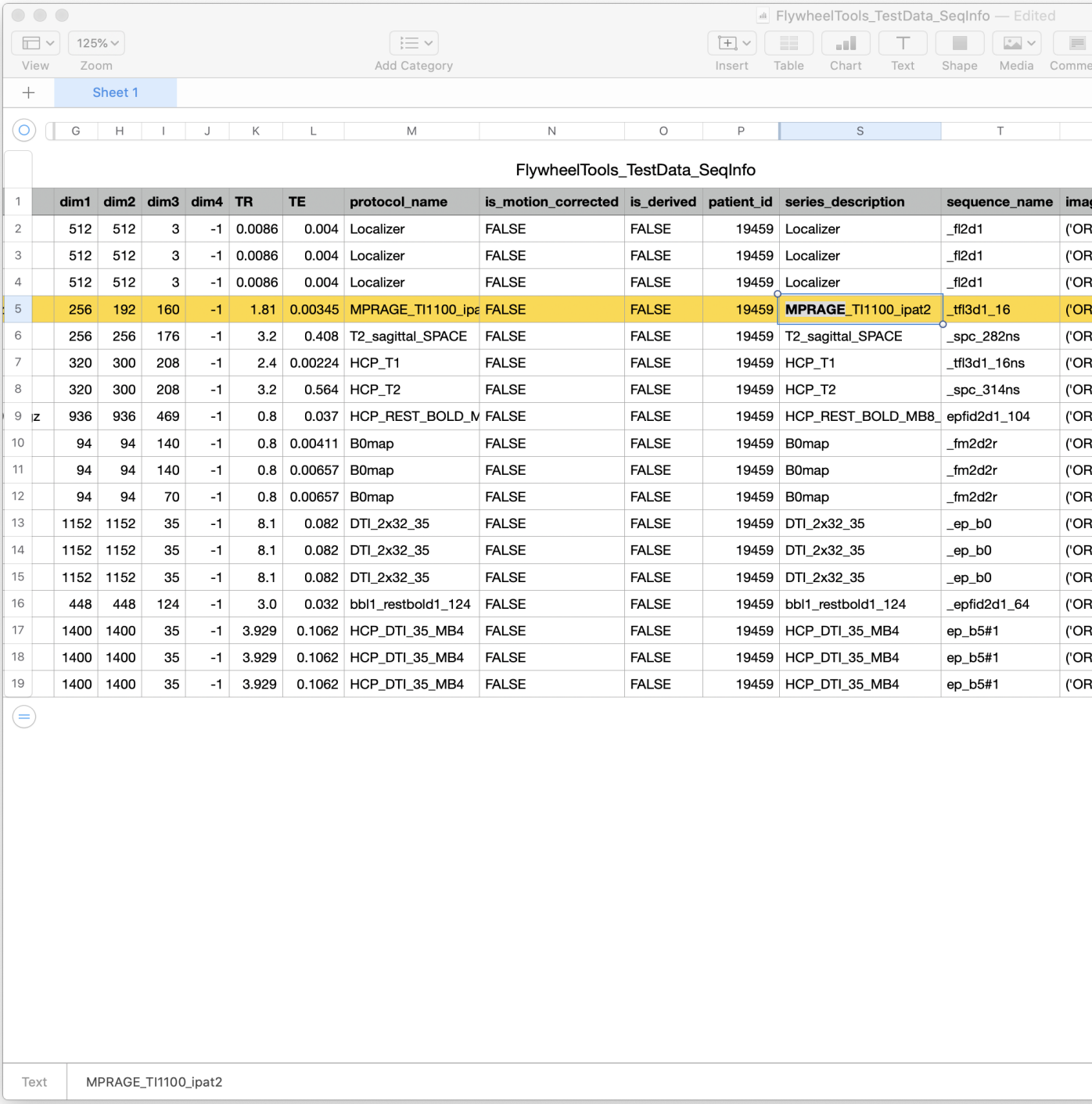
When `fw-heudiconv` runs this heuristic, there will exist a variable called `t1w`, and it will have the string specifying the BIDS file name and path for a T1w (relative to the BIDS root). The next step is making sure that the DICOM we selected will be assigned to this variable. The next function we will use to do that is the `infotodict` function:

```
def infotodict(seqinfo):

    info = {
      t1w: []
    }

    for s in seqinfo:
        if "MPRAGE" in s.series_description:
            info[t1w].append(s.series_id)

    return info
```

After the function is defined with `def`, we create the `info` object – a Python dictionary with one key, `t1w`, and an empty list. Our goal is to populate this dictionary with the list of DICOMs who belong to the `t1w` key.

The input to this function, `seqinfo`, is *each row from your seqinfo table*. So looping over the object `seqinfo` gives you access to each row of your table, where the variables in the table are accessed using Python.

In our example above, we access `series_description` and use Python logic to check if it contains the string `MPRAGE`. We know our T1w is the only one that has this string:



FlywheelTools_TestData_SeqInfo — Edited

FlywheelTools_TestData_SeqInfo

	dim1	dim2	dim3	dim4	TR	TE	protocol_name	is_motion_corrected	is_derived	patient_id	series_description	sequence_name	imag
2	512	512	3	-1	0.0086	0.004	Localizer	FALSE	FALSE	19459	Localizer	_fl2d1	('OR
3	512	512	3	-1	0.0086	0.004	Localizer	FALSE	FALSE	19459	Localizer	_fl2d1	('OR
4	512	512	3	-1	0.0086	0.004	Localizer	FALSE	FALSE	19459	Localizer	_fl2d1	('OR
5	256	192	160	-1	1.81	0.00345	MPRAGE_TI1100_ipa	FALSE	FALSE	19459	MPRAGE_TI1100_ipat2	_tfl3d1_16	('OR
6	256	256	176	-1	3.2	0.408	T2_sagittal_SPACE	FALSE	FALSE	19459	T2_sagittal_SPACE	_spc_282ns	('OR
7	320	300	208	-1	2.4	0.00224	HCP_T1	FALSE	FALSE	19459	HCP_T1	_tfl3d1_16ns	('OR
8	320	300	208	-1	3.2	0.564	HCP_T2	FALSE	FALSE	19459	HCP_T2	_spc_314ns	('OR
9	936	936	469	-1	0.8	0.037	HCP_REST_BOLD_M	FALSE	FALSE	19459	HCP_REST_BOLD_MB8_	epfid2d1_104	('OR
10	94	94	140	-1	0.8	0.00411	B0map	FALSE	FALSE	19459	B0map	_fm2d2r	('OR
11	94	94	140	-1	0.8	0.00657	B0map	FALSE	FALSE	19459	B0map	_fm2d2r	('OR
12	94	94	70	-1	0.8	0.00657	B0map	FALSE	FALSE	19459	B0map	_fm2d2r	('OR
13	1152	1152	35	-1	8.1	0.082	DTI_2x32_35	FALSE	FALSE	19459	DTI_2x32_35	_ep_b0	('OR
14	1152	1152	35	-1	8.1	0.082	DTI_2x32_35	FALSE	FALSE	19459	DTI_2x32_35	_ep_b0	('OR
15	1152	1152	35	-1	8.1	0.082	DTI_2x32_35	FALSE	FALSE	19459	DTI_2x32_35	_ep_b0	('OR
16	448	448	124	-1	3.0	0.032	bbl1_restbold1_124	FALSE	FALSE	19459	bbl1_restbold1_124	_epfid2d1_64	('OR
17	1400	1400	35	-1	3.929	0.1062	HCP_DTI_35_MB4	FALSE	FALSE	19459	HCP_DTI_35_MB4	ep_b5#1	('OR
18	1400	1400	35	-1	3.929	0.1062	HCP_DTI_35_MB4	FALSE	FALSE	19459	HCP_DTI_35_MB4	ep_b5#1	('OR
19	1400	1400	35	-1	3.929	0.1062	HCP_DTI_35_MB4	FALSE	FALSE	19459	HCP_DTI_35_MB4	ep_b5#1	('OR

Text MPRAGE_TI1100_ipat2

So we *append* the `series_id` value of that row (the unique identifier of the DICOM) to the list of files that should be named this way – the `t1w` key. The heuristic at this point should look like this:

```
def create_key(template, outtype=('nii.gz',), annotation_classes=None):
    if template is None or not template:
        raise ValueError('Template must be a valid format string')
```

(continues on next page)

(continued from previous page)

```
    return template, outtype, annotation_classes

# Create Keys
t1w = create_key(
   'sub-{subject}/ses-{session}/anat/sub-{subject}_ses-{session}_T1w')

# loop over the seqinfo table
def infotodict(seqinfo):

    # the dictionary of keys and list of files they correspond to
    info = {
      t1w: []
    }

    # loop over each row of your seqinfo table
    for s in seqinfo:

        # if the series description contains "MPRAGE",
        # add the DICOM identifier to the dictionary

        if "MPRAGE" in s.series_description:
            info[t1w].append(s.series_id)

        # a print line to tell us T1w was not found
        print("This seqinfo is not the MPRAGE:", s.series_description)
    return info
```

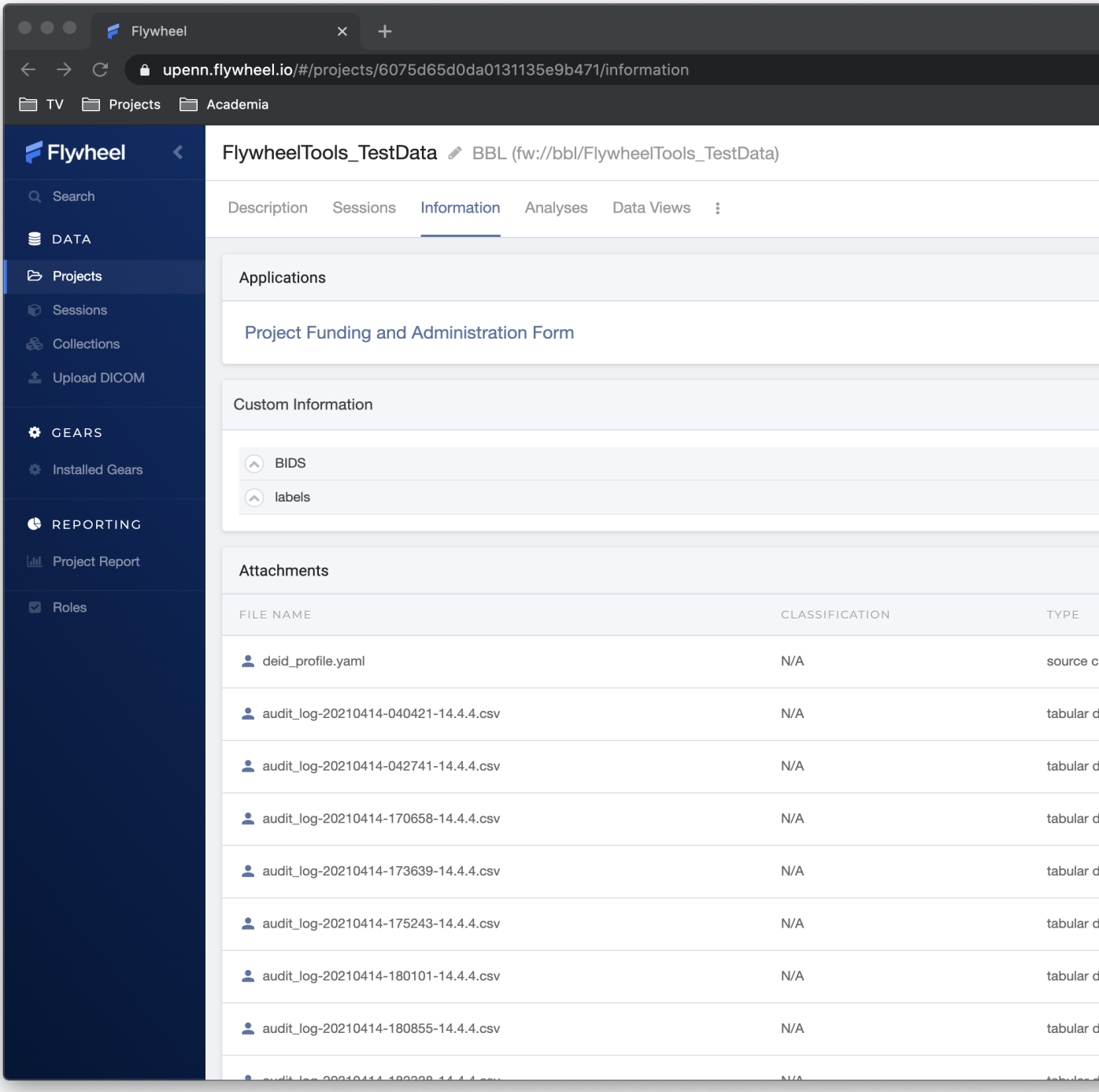
Note: A good habit for debugging is to print out the `seqinfo` rows that did not meet any tests and haven't been assigned to a key

Save this file as `my_test_heuristic.py` – we're going to use it in the next section to curate the T1w image!

1.4.4 Curating The First Image with `fw-heudiconv`

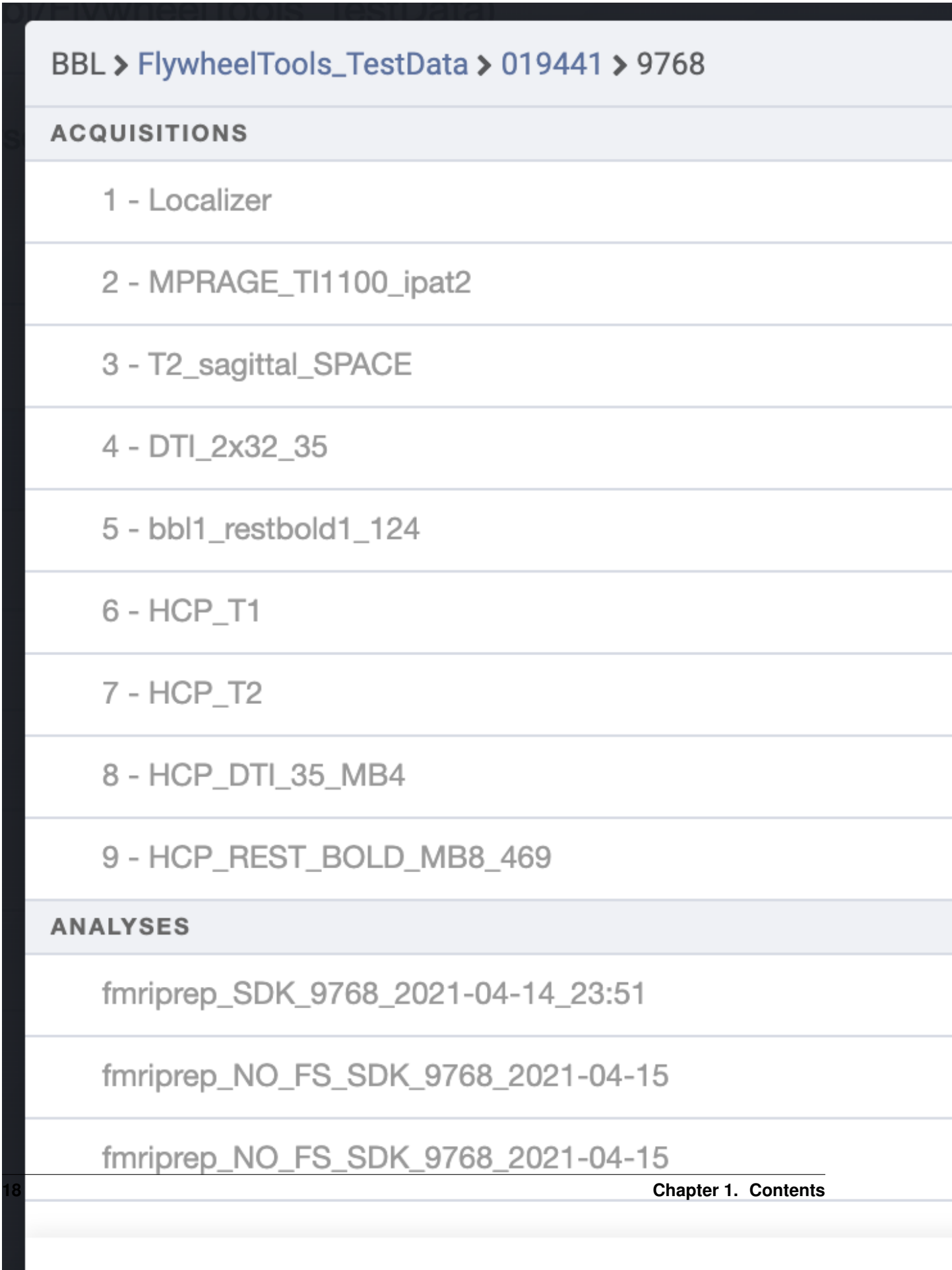
The first step to curating the data is to upload this file to the Flywheel project. Although files can be attached to any object, we recommend attaching this to the project so that all other projects can access it easily.

In the Flywheel GUI, access the "Information" tab of the project, and upload your heuristic file using the "Upload Attachment" button:



Now, we're going to launch a gear on a *single session*. Pick a session from the subject we've been developing on (019465 – in this case, the session is 9793). Gears run from the session level by default, though it is possible to launch them from a subject.

In the top right, click "Run Gear". As before, a dropdown should appear for you to select the `Flywheel HeuDiConv` gear from the *Analysis Gears* list. This time, though, select an input — in the *heuristic* input box, click "Select Input". Here' you'll be presented with a drop down to let you pick *which object* to look for input files. The hierarchy at the top shows that it's looking at the current session and acquisitions:


BBL > FlywheelTools_TestData > 019441 > 9768
ACQUISITIONS
1 - Localizer
2 - MPRAGE_TI1100_ipat2
3 - T2_sagittal_SPACE
4 - DTI_2x32_35
5 - bbl1_restbold1_124
6 - HCP_T1
7 - HCP_T2
8 - HCP_DTI_35_MB4
9 - HCP_REST_BOLD_MB8_469
ANALYSES
fmriprep_SDK_9768_2021-04-14_23:51
fmriprep_NO_FS_SDK_9768_2021-04-15
fmriprep_NO_FS_SDK_9768_2021-04-15

Instead of this, click the *Project label* to select files attached to your project, and select your heuristic. In the "Configuration" tab, select "Curate" under the "action" option. You can leave the "dry_run" box checked – we will uncheck it after this test run.

When you're ready, hit "Run Gear"! Take note of the analysis' name to refer back to later.

To monitor progress of the gear, click on the session's "Provenance" tab. A grey pause symbol indicates that the job is queued, a blue cog indicates that it is running, and a green check or red triangle indicates that the gear has finished, successfully or unsuccessfully, respectively; refresh the page to update the status of running gears.

At the command line, this achieved with the following command:

```
fw-heudiconv-curate --project FlywheelTools_TestData --heuristic PATH/TO/DIRECTORY/my_
↪test_heuristic.py --subject 019465 --session 9793
```

The Output

Next you can navigate to the output of the gear to see what happened. From within the session, click on either the "Analysis" (for strictly any analysis gears that have run) or "Provenance" (for a listing of *any* kind of gear that has operated on the session) and navigate to your recent `fw-heudiconv` analysis. From here, select "View Log".

First, there are virtual machine instructions, stuff Flywheel uses to let us know what kind of virtual machine the gear ran in. Then, we have initialization instructions from `fw-heudiconv`'s "gear manager":

```
Gear Name: fw-heudiconv, Gear Version: 0.2.15_0.3.3
Executor: computebbl-31f3d27f, CPU: 8 cores, Memory: 55GB, Disk: 208GB, Swap: 32GB
Gear starting...

INFO: ==============: fw-heudiconv gear manager starting up :===============

INFO: Calling fw-heudiconv with the following settings:
INFO: Project: FlywheelTools_TestData
INFO: Subject(s): ['019465']
INFO: Session(s): ['9793']
INFO: Heuristic: /flywheel/v0/input/heuristic/my_test_heuristic.py
INFO: Action: Curate
INFO: Dry run: True
INFO: Call: fw-heudiconv-curate --verbose --project FlywheelTools_TestData --dry-run -
↪-subject 019465 --session 9793 --heuristic /flywheel/v0/input/heuristic/my_test_
↪heuristic.py
INFO: =================: fw-heudiconv curator starting up :=================
```

Pay attention to the `Call` directive; this prints the command line equivalent of what is running in the gear.

Next, the actual `fw-heudiconv` outputs. We see that `fw-heudiconv` first attempts to load your heuristic and then lists out all of your `seqinfo` objects (the rows from the table in the first part) with the `series_description` first, followed by other columns:

```
INFO: Loading heuristic file...
INFO: Heuristic loaded successfully!
INFO: Querying Flywheel server...
DEBUG: Found project: FlywheelTools_TestData (6075d65d0da0131135e9b471)
DEBUG: Found sessions:
    9793 (607732a764d3dfc86e6510d4)
INFO: Applying heuristic to 9793 (1/1)...
DEBUG: Found SeqInfos:
Localizer:
```

(continues on next page)

(continued from previous page)

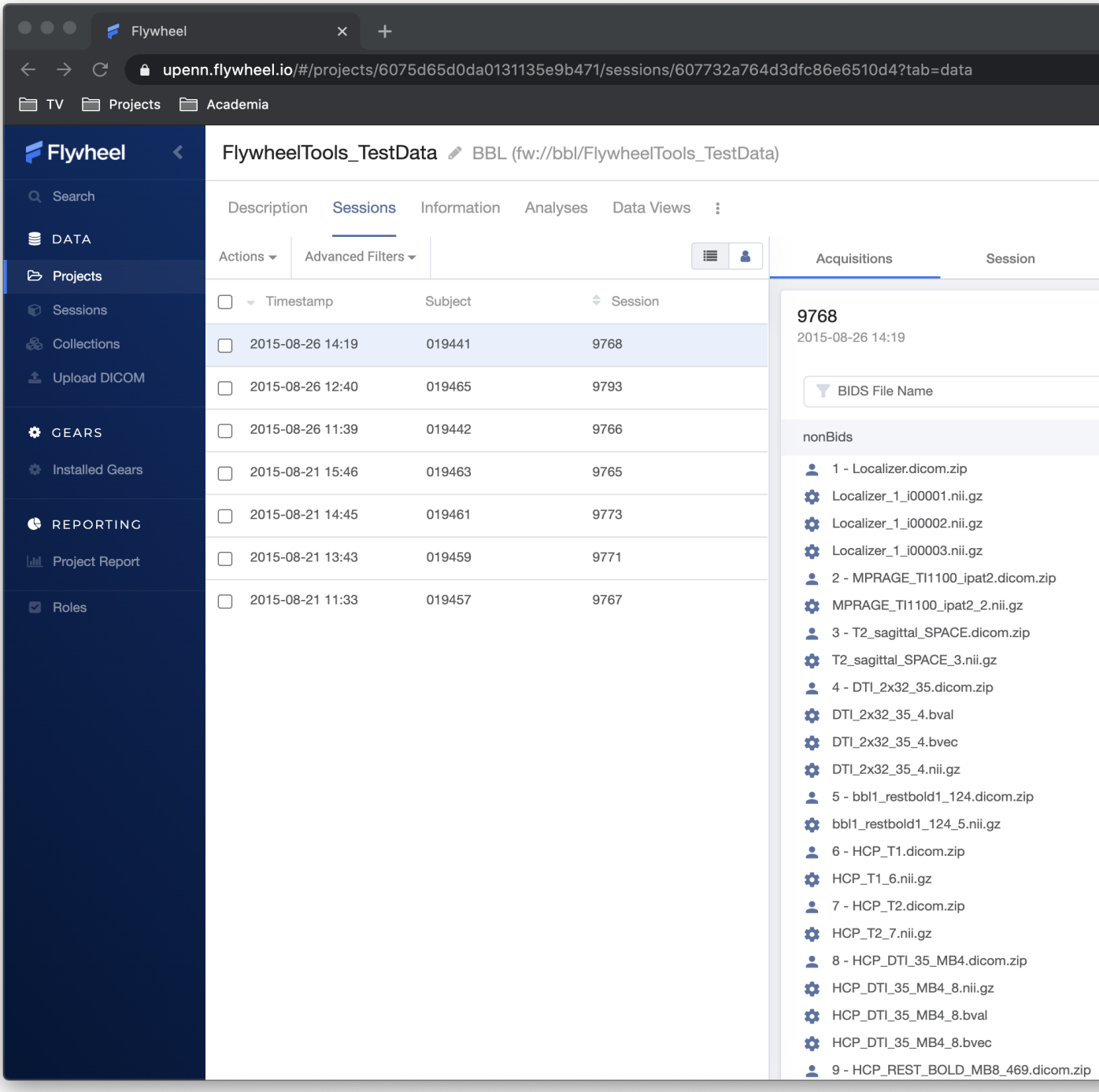
```
            [TR=0.0086 TE=0.004 shape=(512, 512, 3, -1) image_type=('ORIGINAL',
↪'PRIMARY', 'M', 'ND', 'NORM')] (607732a75b936738e644aee8)
MPRAGE_TI1100_ipat2:
              [TR=1.81 TE=0.00345 shape=(256, 192, 160, -1) image_type=('ORIGINAL',
↪'PRIMARY', 'M', 'ND', 'NORM')] (607732a8b9d367cae5e9b0c5)
:
:
:
```

This is good, as we can confirm the table data with the `seqinfo` object the tool is using to curate your data. For example, we know there's a DICOM with the series description `MPRAGE_TI1100_ipat2`, and we searched specifically for the string `MPRAGE`. So, did we catch this `seqinfo`?

```
HCP_REST_BOLD_MB8_469:
  [TR=0.8 TE=0.037 shape=(936, 936, 469, -1) image_type=('ORIGINAL', 'PRIMARY', 'M',
↪'MB', 'ND', 'MOSAIC')] (607732a864d3dfc86e6510d7)

DEBUG:
MPRAGE_TI1100_ipat2_2.nii.gz
    sub-019465_ses-9793_T1w.nii.gz -> sub-019465/ses-9793/anat/sub-019465_ses-9793_
↪T1w.nii.gz
INFO: Done!
INFO: ====================: Exiting fw-heudiconv curator :====================
```

Excellent! Now we know that the NIfTI from this DICOM, MPRAGE_TI1100_ipat2_2.nii.gz, will be mapped to a file named `sub-019465_ses-9793_T1w.nii.gz`, and the path is listed there too. Additionally, we get printouts of the `seqinfo` objects that didn't get caught by our logic.

To see our BIDS data before curation, go to the session view and click the "BIDS View" toggle; there should be no BIDS data:



Now that we know it works, we can run it again with the "dry_run" box unchecked to apply the changes. The only difference should be that the log lets you know the changes are being applied:

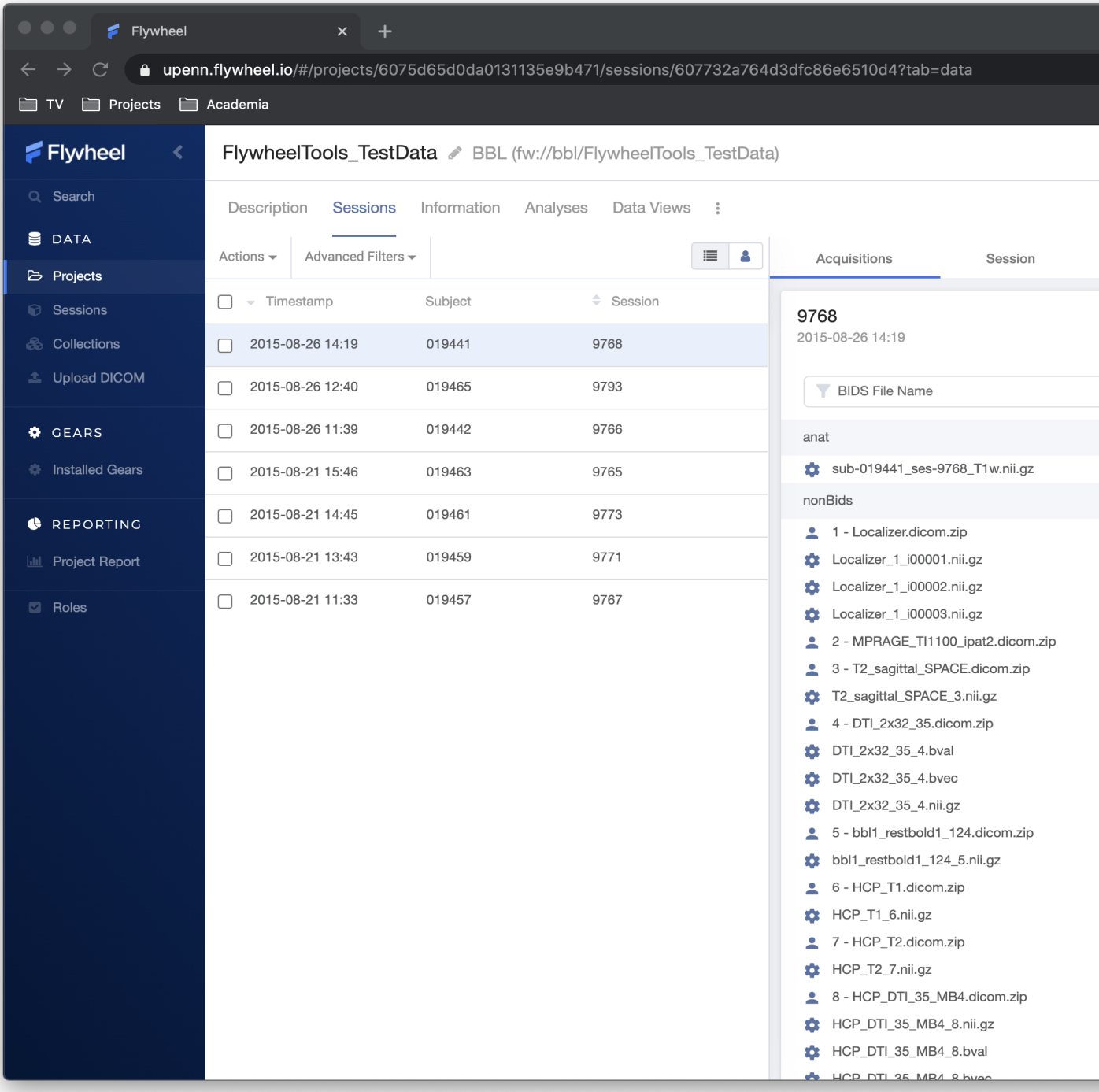
```
HCP_REST_BOLD_MB8_469:
  [TR=0.8 TE=0.037 shape=(936, 936, 469, -1) image_type=('ORIGINAL', 'PRIMARY', 'M',
↪'MB', 'ND', 'MOSAIC')] (607732a864d3dfc86e6510d7)
```

(continues on next page)

(continued from previous page)

```
INFO: Applying changes to files...
DEBUG:
MPRAGE_TI1100_ipat2_2.nii.gz
sub-019465_ses-9793_T1w.nii.gz -> sub-019465/ses-9793/anat/sub-019465_ses-9793_T1w.
↪nii.gz
INFO: Done!
INFO: ==================: Exiting fw-heudiconv curator :==================
```

Now, in the session view, hit the "BIDS View" toggle:



We've successfully curated one of our images into BIDS!

1.4.5 Adding More Images

By now, it should be clear that as the tool loops over the rows in the `seqinfo` table, you can add all sorts of logic to capture additional `seqinfo` objects and assign them to keys you create. Below, we edit the heuristic and add a key for the BOLD data in our project, and use similar logic to assign data to the key:

```
def create_key(template, outtype=('nii.gz',), annotation_classes=None):
    if template is None or not template:
        raise ValueError('Template must be a valid format string')
    return template, outtype, annotation_classes

# Create Keys

# anatomical
t1w = create_key(
   'sub-{subject}/ses-{session}/anat/sub-{subject}_ses-{session}_T1w')

# fMRI scans
rest_bbl = create_key(
  'sub-{subject}/{session}/func/sub-{subject}_{session}_task-rest_acq-BBL_bold')

# loop over the seqinfo table
def infotodict(seqinfo):

    # the dictionary of keys and list of files they correspond to
    # now contains two scans
    info = {
      t1w: [], rest_bbl: []
    }

    # loop over each row of your seqinfo table
    for s in seqinfo:

        # if the series description contains "MPRAGE",
        # add the DICOM identifier to the dictionary

        if "MPRAGE" in s.series_description:
            info[t1w].append(s.series_id)

        elif "bbl1_restbold" in protocol:
            info[rest_bbl].append(s.series_id)

        # a print line to tell us T1w was not found
        print("This seqinfo is not the MPRAGE or rs-fMRI:", s.series_description)
    return info
```

But what if we have a fieldmap? Not only do we need to name it correctly, but we also have to make sure it points to the BOLD data. We can do this quite flexibly using the special `IntendedFor` keyword. This keyword is set outside of the `infotodict` for loop and makes use of existing keys:

```
def create_key(template, outtype=('nii.gz',), annotation_classes=None):
    if template is None or not template:
        raise ValueError('Template must be a valid format string')
    return template, outtype, annotation_classes

# Create Keys

# anatomical
t1w = create_key(
   'sub-{subject}/ses-{session}/anat/sub-{subject}_ses-{session}_T1w')

# fMRI scans
rest_bbl = create_key(
```

(continues on next page)

(continued from previous page)

```
  'sub-{subject}/{session}/func/sub-{subject}_{session}_task-rest_acq-BBL_bold')

# fieldmaps
b0_phase = create_key(
   'sub-{subject}/{session}/fmap/sub-{subject}_{session}_phasediff')
b0_mag = create_key(
   'sub-{subject}/{session}/fmap/sub-{subject}_{session}_magnitude{item}')


# loop over the seqinfo table
def infotodict(seqinfo):

    # the dictionary of keys and list of files they correspond to
    # now contains two scans
    info = {
      t1w: [], rest_bbl: [], b0_mag: [], b0_phase: []
    }

    # loop over each row of your seqinfo table
    for s in seqinfo:

        # if the series description contains "MPRAGE",
        # add the DICOM identifier to the dictionary

        if "MPRAGE" in s.series_description:
            info[t1w].append(s.series_id)

        elif "bbl1_restbold" in s.series_description:
            info[rest_bbl].append(s.series_id)

        elif "B0map" in s.series_description and "M" in s.image_type:
            info[b0_mag].append(s)
        elif "B0map" in s.series_description and "P" in s.image_type:
            info[b0_phase].append(s)


        # a print line to tell us T1w was not found
        print("Protocol not found!:", s.series_description)
    return info

IntendedFor = {
    b0_phase: [
        '{session}/func/sub-{subject}_{session}_task-rest_acq-BBL_bold.nii.gz'
    ],
    b0_mag: [
        '{session}/func/sub-{subject}_{session}_task-rest_acq-BBL_bold.nii.gz'
    ]
}
```

Notice that in this heuristic, we use the special `{item}` keyword in the key for the magnitude fieldmaps. This keeps us from having to write multiple keys. The keyword is iterated over automatically within the NIfTIs for this acquisition. Additionally, we access the `image_type` column when differentiating between the magnitude and phase fieldmaps.

Lastly, the `IntendedFor` keyword: it's a dictionary, like `info`, containing the keys for each of our fieldmaps, and the values for each key are the files we expect this fieldmap to correct. In this case we must specifically list the files out. Update this heuristic upload it to Flywheel, and try out curation with it.

1.4.6 Wrapping Up

In this walkthrough, you should have learned how to curate your data with `fw-heudiconv`, starting with discovering data in your DICOM headers, crafting a heuristic for a single T1w image, and then applying it to a session. Next, we went over how to add more images, including how to point fieldmaps to BOLD scans. Now, you're ready to investigate more functionality of `fw-heudiconv`. Take a look at the *The Heuristic File* page for an in-depth look at what more features fw-heudiconv curation can offer, and the tips page for inspiration on how to come up with more creative solutions.

1.5 The Heuristic File

BIDS curation of data on Flywheel is implemented through the use of a heuristic file. Like the name implies, a heuristic is a set of simple and efficient rules that, for our purposes, will help map DICOM header info to a BIDS-valid filename.

The heuristic's rules are defined in a Python file which is used as input to the curate command line tool `fw_heudiconv.cli.curate`. Using Python, it's possible to accomplish a wide variety of logical operations to define these relationships, but in order to communicate with Flywheel, `fw-heudiconv` expects a few reserved functions and data structures. These functions are documented below.

1.5.1 How `fw-heudiconv` Uses a Heuristic

Once `fw-heudiconv` has parsed arguments and filtered out the target sessions to curate, `fw-heudiconv` then gathers all of the DICOM header information in a session's acquisitions. In the program, we call these objects `seqinfo` objects. The program loops over each of these `seqinfo` objects and tests each one to see if the heuristic has defined a BIDS filename for a `seqinfo` of this type. If so, it adds a reference to this `seqinfo` to a special internal list. At the end of the checks, `fw-heudiconv` goes through the list of references, adding BIDS metadata to each of the NIfTIs the references point to.

1.5.2 Heuristic Functions

This heuristic demonstrates all of the functionalities available in fw-heudiconv data curation.

Mandatory functions

There are two mandatory functions that are expected in a heuristic. The first is the `create_key()` function. This function allows the heuristic to define BIDS- valid filenames for each scan type and category you expect to find. Once defined, you can then assign keys to variables to be used in the next mandatory function.

`create_key`(*template*, *outtype=('nii.gz',)*, *annotation_classes=None*)

Create a BIDS key

Use this function to create a BIDS key with keywords to be populated at runtime. Keys **must** be BIDS valid and have the full BIDS path; the file extension is **not** required. Available keywords are as follows:

{subject}	The subject label
{session}	The session label
{item}	An iterator to be used *within* an acquisition

Example:

```
>>> t1w = create_key('sub-{subject}/{session}/anat/sub-{subject}_{session}_T1w')
>>> t1w
('sub-{subject}/{session}/anat/sub-{subject}_{session}_T1w', ('nii.gz',), None)
>>> rest_mb = create_key('sub-{subject}/{session}/func/sub-{subject}_{session}_
↪task-rest_acq-multiband_bold')
>>> rest_mb
('sub-{subject}/{session}/func/sub-{subject}_{session}_task-rest_acq-multiband_
↪bold', ('nii.gz',), None)
```

The next mandatory function is `infotodict()`. This function does the heavy lifting — it loops over the `seqinfo` objects, and uses boolean logic in each to decide if it is going to be assigned to a BIDS key.

infotodict (*seqinfo*)

Heuristic evaluator for mapping seqinfos to BIDS filenames

A function for defining the boolean logic that determines how to map a seqinfo to a key made with *create_key()*. The `seqinfo` object has a number of attributes that can be tested in boolean logic; when a match is found, the `series_id` attribute is added to a list that tracks the matches.

All usable attributes are listed as columns in the output of the tabulate tool (for example, all DICOMs have a `series_description`, which shows up as a column in the output of `fw-heudiconv-tabulate`; you can access this attribute using `s.series_description`).

The return object **must** be a dictionary where each *key* is a `key` variable already earlier defined in the namespace, and the corresponding *value* is a list of `series_id`.

We find that the easiest way to accomplish this (and debug iteratively) is with the use of a for-loop.

Parameters **seqinfo** – a `fw-heudiconv` seqinfo object, enumerating DICOM metadata as attributes

Returns dictionary – a dictionary of keys and a list of seqinfo series IDs that match the key

Example:

```
>>> t1w = create_key('sub-{subject}/{session}/anat/sub-{subject}_{session}_T1w')
>>> t1w
('sub-{subject}/{session}/anat/sub-{subject}_{session}_T1w', ('nii.gz',), None)
>>> rest_mb = create_key('sub-{subject}/{session}/func/sub-{subject}_{session}_
↪task-rest_acq-multiband_bold')
>>> rest_mb
('sub-{subject}/{session}/func/sub-{subject}_{session}_task-rest_acq-multiband_
↪bold', ('nii.gz',), None)
```

```
>>> def infotodict(seqinfo):
...     info = {t1w:[], rest_mb:[]}
...     for s in seqinfo:
...         protocol = s.protocol_name.lower()
...         if "mprage" in protocol:
...             info[t1w].append(s.series_id)
...         elif "rest" in protocol:
...             info[rest_mb].append(s.series_id)
...         else:
...             print('Series {} not found!'.format(protocol_name))
...     return info
```

Optional variables

There are optional variables you can use to hardcode metadata into the BIDS sidecar or define fieldmap intentions (`MetadataExtras` and `IntendedFor`).

`MetadataExtras = {}`

Special variable defining metadata to hardcodeinto the BIDS sidecar.

Use this variable to define metadata that you want to hardcode into the BIDS sidecar. For example, we could use this to hardcode the EchoTime for phase fieldmaps, or for use in ASL, we can use this to hardcode metadata that sometimes isn't extracted by `dcm2niix`.

This variable **must** be a dictionary, where the *key* is a `key` variable already earlier defined in the namespace, and the *value* is itself a dictionary of metadata.

Example (we've already defined keys `b0_phase` and `asl` with `create_key`):

```
>>> MetadataExtras = {
    b0_phase: {
        "EchoTime1": 0.00412,
        "EchoTime2": 0.00658
    },
    asl: {
        "PulseSequenceType": "3D_SPRIAL",
        "PulseSequenceDetails" : "WIP" ,
        "LabelingType": "PCASL",
        "LabelingDuration": 1.8,
        "PostLabelingDelay": 1.8,
        "BackgroundSuppression": "Yes",
        "M0":10,
        "LabelingSlabLocation":"X",
        "LabelingOrientation":"",
        "LabelingDistance":2,
        "AverageLabelingGradient": 34,
        "SliceSelectiveLabelingGradient":45,
        "AverageB1LabelingPulses": 0,
        "LabelingSlabThickness":2,
        "AcquisitionDuration":123,
        "BackgroundSuppressionLength":2,
        "BackgroundSuppressionPulseTime":2,
        "VascularCrushingVenc": 2,
        "PulseDuration": 1.8,
        "InterPulseSpacing":4,
        "PCASLType":"balanced",
        "PASLType": "",
        "LookLocker":"True",
        "LabelingEfficiency":0.72,
        "BolusCutOffFlag":"False",
        "BolusCutOffTimingSequence":"False",
        "BolusCutOffDelayTime":0,
        "BolusCutOffTechnique":"False"
    }
}
```

`IntendedFor = {}`

Special variable mapping fieldmaps to scans.

Use this variable to define which files your fieldmaps are intended to correct for. You do this by using the fieldmap keys defined with `create_key`, and a list of filenames where the keywords `{subject}`,

`{session}` and others are used for ambiguity. `fw-heudiconv` will check for each file and try to map IntendedFor's appropriately.

This variable **must** be a dictionary, where the *key* is a `key` variable already earlier defined in the namespace, and the *value* is a list of filename templates.

Example (we've already defined `b0_phase`, `b0_phase`, `pe_rev` with `create_key`):

```
>>> IntendedFor = {
    b0_phase: [
        '{session}/func/sub-{subject}_{session}_task-rest_acq-multiband_bold.nii.
↪gz',
        '{session}/func/sub-{subject}_{session}_task-rest_acq-singleband_bold.nii.
↪gz',
        '{session}/func/sub-{subject}_{session}_task-fracback_acq-singleband_bold.
↪nii.gz',
        '{session}/func/sub-{subject}_{session}_task-face_acq-singleband_bold.nii.
↪gz'
    ],
    b0_mag: [
        '{session}/func/sub-{subject}_{session}_task-rest_acq-multiband_bold.nii.
↪gz',
        '{session}/func/sub-{subject}_{session}_task-rest_acq-singleband_bold.nii.
↪gz',
        '{session}/func/sub-{subject}_{session}_task-fracback_acq-singleband_bold.
↪nii.gz',
        '{session}/func/sub-{subject}_{session}_task-face_acq-singleband_bold.nii.
↪gz'
    ],
    pe_rev: [
        '{session}/dwi/sub-{subject}_{session}_acq-multiband_dwi.nii.gz',
    ]
}
```

`Replace*` functions

There are optional functions that assist with Flywheel-specific data manipulation. The first of these is the `ReplaceSubject()` and `ReplaceSession()` functions, which can be used to manipulate the label of a Flywheel object before it is inserted into a BIDS filename (for example, to remove leading zeroes). These functions are expected to have a string as input (the Flywheel label) and the return object to be a string of your making. These functions *don't* affect the source data objects on Flywheel, only the metadata BIDS fields.

`ReplaceSubject`(*label*)

Manipulate the BIDS subject label

Use this function to define how to manipulate a subject's label on Flywheel into a BIDS valid <subject> value

Parameters **`label`** (*string*) – the Flywheel subject label

Returns string – the manipulated string label

Example – stripping leading zeroes from a subject label:

```
>>> def ReplaceSubject(label):
...     return label.lstrip('0')
>>> ReplaceSubject('01234')
'1234'
```

ReplaceSession(*label*)

Manipulate the BIDS session label

Use this function to define how to manipulate a session's label on Flywheel into a BIDS valid <session> value

Parameters **label** (*string*) – the Flywheel session label

Returns string – the manipulated string label

Example – enforcing all sessions are labelled 01:

```
>>> def ReplaceSession(label):
...     return '01'
>>> ReplaceSession('SomeSession')
'01'
```

Attach* functions

Then there are the `AttachToProject()` and `AttachToSession()` functions, which are used to dynamically generate and upload BIDS metadata files, like participant or event files. We've found these functions useful for generating and uploading ASL context files, but can be used for any dynamic file attachment purpose, so long as the data can be parsed into a raw text string.

AttachToSession()

Attach BIDS data files to a project at the session level

Use this function to dynamically generate files and upload them to the BIDS project at the session level. The filename **must** be BIDS valid. Examples include the `events.tsv` file or the `aslcontext.tsv` file.

This function takes no input but **must** return a dictionary (or list of dictionaries) with three parts:

1. `name`: the BIDS filename, with optional keywords for formatting (e.g. `{subject}`).
2. `data`: the data to upload, which **must** be in literal string format.
3. `type`: the file MIMEType; see Link here for available types.

Returns dictionary – the dictionary containing BIDS data

Example – creating an ASL context file from scratch to attach to each session:

```
>>> def AttachToSession():
...     attachment1 = {
...         'name': '{subject}/{session}/perf/{subject}_{session}_aslcontext.tsv',
...         'data': '\n'.join(['Control', 'Label', 'Control', 'Label']),
...         'type': 'text/tab-separated-values'
...     }
...     return attachment1
```

```
>>> AttachToSession()
{'name': '{subject}/{session}/perf/{subject}_{session}_aslcontext.tsv', 'data':
↪'Control\nLabel\nControl\nLabel', 'type': 'text/tab-separated-values'}
```

AttachToProject()

Attach BIDS data files to a project at the session level

Use this function to dynamically generate files and upload them to the BIDS project at the session level. The filename **must** be BIDS valid. Examples include the README or CHANGES file.

This function takes no input but **must** return a dictionary (or list of dictionaries) with three parts:

1. `name`: the BIDS filename, with optional keywords for formatting (e.g. `{subject}`).
2. `data`: the data to upload, which **must** be in literal string format.
3. `type`: the file MIMEType; see Link here for available types.

Returns dictionary – the dictionary containing BIDS data

Example – Adding a README:

```
>>> def AttachToSession():
...     attachment1 = {
...         'name': 'README',
...         'data': 'This is my BIDS dataset',
...         'type': 'text/plain'
...     }
...     return attachment1
```

```
>>> AttachToSession()
{'name': 'README', 'data': 'This is my BIDS dataset', 'type': 'text/plain'}
```

1.5.3 A Real Example

In all, a heuristic file could look like this:

```
import os

def create_key(template, outtype=('nii.gz',), annotation_classes=None):
    if template is None or not template:
        raise ValueError('Template must be a valid format string')
    return template, outtype, annotation_classes

# Create Keys
t1w = create_key(
   'sub-{subject}/{session}/anat/sub-{subject}_{session}_T1w')
t2w = create_key(
   'sub-{subject}/{session}/anat/sub-{subject}_{session}_T2w')
dwi = create_key(
   'sub-{subject}/{session}/dwi/sub-{subject}_{session}_acq-multiband_dwi')

# Field maps
b0_phase = create_key(
   'sub-{subject}/{session}/fmap/sub-{subject}_{session}_phasediff')
b0_mag = create_key(
   'sub-{subject}/{session}/fmap/sub-{subject}_{session}_magnitude{item}')
pe_rev = create_key(
    'sub-{subject}/{session}/fmap/sub-{subject}_{session}_acq-multiband_dir-j_epi')

# fmri scans
rest_mb = create_key(
   'sub-{subject}/{session}/func/sub-{subject}_{session}_task-rest_acq-multiband_bold
↪')
rest_sb = create_key(
   'sub-{subject}/{session}/func/sub-{subject}_{session}_task-rest_acq-singleband_bold
↪')
fracback = create_key(
```

(continues on next page)

(continued from previous page)

```
    'sub-{subject}/{session}/func/sub-{subject}_{session}_task-fracback_acq-singleband_
↪bold')
face = create_key(
    'sub-{subject}/{session}/func/sub-{subject}_{session}_task-face_acq-singleband_bold
↪')

# ASL scans
asl = create_key(
    'sub-{subject}/{session}/perf/sub-{subject}_{session}_asl')
asl_dicomref = create_key(
    'sub-{subject}/{session}/perf/sub-{subject}_{session}_acq-ref_asl')
m0 = create_key(
    'sub-{subject}/{session}/perf/sub-{subject}_{session}_m0')
mean_perf = create_key(
    'sub-{subject}/{session}/perf/sub-{subject}_{session}_mean-perfusion')


def infotodict(seqinfo):

    last_run = len(seqinfo)

    info = {t1w:[], t2w:[], dwi:[], b0_phase:[],
            b0_mag:[], pe_rev:[], rest_mb:[], rest_sb:[],
            fracback:[], asl_dicomref:[], face:[], asl:[],
            m0:[], mean_perf:[]}

    def get_latest_series(key, s):
    #    if len(info[key]) == 0:
        info[key].append(s.series_id)
    #    else:
    #        info[key] = [s.series_id]

    for s in seqinfo:
        protocol = s.protocol_name.lower()
        if "mprage" in protocol:
            get_latest_series(t1w,s)
        elif "t2_sag" in protocol:
            get_latest_series(t2w,s)
        elif "b0map" in protocol and "M" in s.image_type:
            info[b0_mag].append(s.series_id)
        elif "b0map" in protocol and "P" in s.image_type:
            info[b0_phase].append(s.series_id)
        elif "topup_ref" in protocol:
            get_latest_series(pe_rev, s)
        elif "dti_multishell" in protocol and not s.is_derived:
            get_latest_series(dwi, s)

        elif s.series_description.endswith("_ASL"):
            get_latest_series(asl, s)
        elif protocol.startswith("asl_dicomref"):
            get_latest_series(asl_dicomref, s)
        elif s.series_description.endswith("_M0"):
            get_latest_series(m0, s)
        elif s.series_description.endswith("_MeanPerf"):
            get_latest_series(mean_perf, s)

        elif "fracback" in protocol:
```

(continues on next page)

(continued from previous page)

```
            get_latest_series(fracback, s)
        elif "face" in protocol:
            get_latest_series(face,s)
        elif "rest" in protocol:
            if "MB" in s.image_type:
                get_latest_series(rest_mb,s)
            else:
                get_latest_series(rest_sb,s)

        elif s.patient_id in s.dcm_dir_name:
            get_latest_series(asl, s)

        else:
            print("Series not recognized!: ", s.protocol_name, s.dcm_dir_name)
    return info

MetadataExtras = {
    b0_phase: {
        "EchoTime1": 0.00412,
        "EchoTime2": 0.00658
    },
    asl: {
        "PulseSequenceType": "3D_SPRIAL",
        "PulseSequenceDetails" : "WIP" ,
        "LabelingType": "PCASL",
        "LabelingDuration": 1.8,
        "PostLabelingDelay": 1.8,
        "BackgroundSuppression": "Yes",
        "M0":10,
        "LabelingSlabLocation":"X",
        "LabelingOrientation":"",
        "LabelingDistance":2,
        "AverageLabelingGradient": 34,
        "SliceSelectiveLabelingGradient":45,
        "AverageB1LabelingPulses": 0,
        "LabelingSlabThickness":2,
        "AcquisitionDuration":123,
        "BackgroundSuppressionLength":2,
        "BackgroundSuppressionPulseTime":2,
        "VascularCrushingVenc": 2,
        "PulseDuration": 1.8,
        "InterPulseSpacing":4,
        "PCASLType":"balanced",
        "PASLType": "",
        "LookLocker":"True",
        "LabelingEfficiency":0.72,
        "BolusCutOffFlag":"False",
        "BolusCutOffTimingSequence":"False",
        "BolusCutOffDelayTime":0,
        "BolusCutOffTechnique":"False"
    }
}

IntendedFor = {
    b0_phase: [
        '{session}/func/sub-{subject}_{session}_task-rest_acq-multiband_bold.nii.gz',
        '{session}/func/sub-{subject}_{session}_task-rest_acq-singleband_bold.nii.gz',
```

(continues on next page)

(continued from previous page)

```
        '{session}/func/sub-{subject}_{session}_task-fracback_acq-singleband_bold.nii.
↪gz',
        '{session}/func/sub-{subject}_{session}_task-face_acq-singleband_bold.nii.gz'
    ],
    b0_mag: [],
    pe_rev: [
        '{session}/dwi/sub-{subject}_{session}_acq-multiband_dwi.nii.gz',
    ]
}

def ReplaceSubject(label):
    return label.lstrip("0")


def ReplaceSession(label):
    return label.lstrip("0")


def AttachToSession():

    # example: uploading a json file
    import json

    adict = {
        "id": "04",
        "name": "foo",
        "scan": "blah"
    }

    json_object = json.dumps(adict, indent = 4) # json.dumps() returns a string!

    attachment1 = {
        'name': 'jsonexample.json',
        'data': json_object,
        'type': 'application/json'
    }

    return attachment1


def AttachToProject():

    # example: uploading a single CHANGES file

    attachment1 = {
        'name': 'CHANGES',
        'data': 'This is a CHANGES file!',
        'type': 'text/plain'
    }

    return attachment1
```

1.6 Usage

`fw-heudiconv` can be run either from the command line, or on Flywheel as a gear. See below for command line instructions.

1.6.1 Tabulate

Tabulate DICOM header info from a project on Flywheel

```
usage: fw-heudiconv-tabulate [-h] --project PROJECT [--path PATH]
                             [--subject SUBJECT [SUBJECT ...]]
                             [--session SESSION [SESSION ...]] [--verbose]
                             [--dry-run] [--unique | --no-unique]
                             [--api-key API_KEY]
```

Named Arguments

--project	The project in flywheel
--path	Path to download .tsv file
	Default: "."
--subject	The subject label(s)
--session	The session label(s)
--verbose	Print ongoing messages of progress
	Default: False
--dry-run	Don't apply changes
	Default: False
--unique	Default: False
--no-unique	Default: True
--api-key	API Key

1.6.2 Curate

Use a heudiconv heuristic to curate data into BIDS on flywheel

```
usage: fw-heudiconv-curate [-h] --project PROJECT --heuristic HEURISTIC
                           [--subject SUBJECT [SUBJECT ...]]
                           [--session SESSION [SESSION ...]] [--verbose]
                           [--dry-run] [--api-key API_KEY]
```

Named Arguments

--project	The project in flywheel
--heuristic	Path to a heudiconv-style heuristic file
--subject	The subject label(s)

--session	The session label(s)
--verbose	Print ongoing messages of progress
	Default: False
--dry-run	Don't apply changes
	Default: False
--api-key	API Key

1.6.3 Export

Export BIDS-curated data from Flywheel

```
usage: fw-heudiconv-export [-h] --project PROJECT [--path PATH]
                           [--subject SUBJECT [SUBJECT ...]]
                           [--session SESSION [SESSION ...]]
                           [--folders FOLDERS [FOLDERS ...]]
                           [--attachments ATTACHMENTS [ATTACHMENTS ...]]
                           [--dry-run] [--destination DESTINATION]
                           [--directory-name DIRECTORY_NAME]
                           [--api-key API_KEY] [--verbose]
```

Named Arguments

--project	The project in flywheel
--path	The target directory to download [DEPRECATED. PLEASE USE <DESTINATION> INSTEAD]
--subject	The subject(s) to export
--session	The session(s) to export
--folders	The BIDS folders to export
	Default: ['anat', 'dwi', 'fmap', 'func', 'perf']
--attachments	Only download attachment files matching these names
--dry-run	Don't apply changes (only print the directory tree to the console)
	Default: False
--destination	Path to destination directory
	Default: "."
--directory-name	Name of destination directory
	Default: "bids_directory"
--api-key	API Key
--verbose	Print ongoing messages of progress
	Default: False

1.6.4 Validate

Validate BIDS-curated data on Flywheel. A simple wrapper around the original BIDS Validator https://github.com/bids-standard/bids-validator

```
usage: fw-heudiconv-validate [-h] [--directory DIRECTORY]
                             [--project PROJECT [PROJECT ...]]
                             [--subject SUBJECT [SUBJECT ...]]
                             [--session SESSION [SESSION ...]] [--verbose]
                             [--tabulate TABULATE] [--api-key API_KEY]
```

Named Arguments

--directory	Temp space used for validation
	Default: "."
--project	The project on Flywheel
--subject	The subject(s) on Flywheel to validate
--session	The session(s) on Flywheel to validate
--verbose	Pass on <VERBOSE> flag to bids-validator
	Default: False
--tabulate	Directory to save tabulation of errors
	Default: "."
--api-key	API Key

1.6.5 Clear

Go nuclear: clear BIDS data from Flywheel

```
usage: fw-heudiconv-clear [-h] --project PROJECT [PROJECT ...]
                          [--subject SUBJECT [SUBJECT ...]]
                          [--session SESSION [SESSION ...]] [--verbose]
                          [--dry-run] [--api-key API_KEY]
```

Named Arguments

--project	The project in flywheel
--subject	The subject label(s)
--session	The session label(s)
--verbose	Print ongoing messages of progress
	Default: False
--dry-run	Don't apply changes
	Default: False
--api-key	API Key

1.6.6 flaudit

`flaudit` runs as a gear on Flywheel. See the Quick Start guide for usage.

1.7 Tips & Tricks: Curating Creatively

Because `fw-heudiconv` is built in Python, you have access to anything Python can do when you build your heuristic (as long as you use the special functions and data structures). Here, we show a few fun ways we've used Python to solve a few tricky heuristic challenges.

1.7.1 Dynamically Replacing Subject/Session Labels

It might be useful to dynamically replace a Flywheel subject's label with some other label in BIDS — for example, in the event that you need to withhold personally identifying information from a BIDS output you share, but still keep the original Flywheel subject's label, for consistency. Well this can be accomplished in the `Replace*()` functions using a DataFrame with `pandas`. If you're running `fw-heudiconv` from disk, you can read in a file at the same time that the heuristic is parsed:

```
def ReplaceSubject(label):

    import pandas as pd

    df = pd.read_csv('DeIdentifiedNames.csv')
```

And then filter your DataFrame as necessary:

```
def ReplaceSubject(label):

    import pandas as pd

    df = pd.read_csv('DeIdentifiedNames.csv')
    target = df[(df.first_name == "Jason")]
    replacement = target['new_ID'].values[0]

    return str(replacement)
```

1.7.2 DataFrames to Strings

In order to use the `AttachTo*()` function, your data needs to be converted into a string. To attach a data-frame object, use the following steps:

```
def AttachToSession():

    # example: uploading multiple files -- a json, and a TSV
    import json

    adict = {
        "id": "04",
        "name": "foo",
        "scan": "blah"
    }
```

(continues on next page)

(continued from previous page)

```
    json_object = json.dumps(adict, indent = 4)

    attachment1 = {
        'name': 'jsonexample.json',
        'data': json_object,
        'type': 'application/json'
    }
    import pandas as pd
    raw_data = {'first_name': ['Jason', 'Molly', 'Tina', 'Jake', 'Amy'],
        'last_name': ['Miller', 'Jacobson', 'Ali', 'Milner', 'Cooze'],
        'age': [42, 52, 36, 24, 73],
        'preTestScore': [4, 24, 31, 2, 3],
        'postTestScore': [25, 94, 57, 62, 70]}
    df = pd.DataFrame(raw_data, columns = ['first_name', 'last_name', 'age',
↪'preTestScore', 'postTestScore'])

    attachment2 = {
        'name': '{subject}/{session}/perf/{subject}_{session}_aslcontext.tsv',
        'data': df.to_csv(index=False, sep='\t'), # .to_csv() with no file argument␣
↪returns a string!
        'type': 'text/tab-separated-values'
    }

    # this is also an opportunity to demonstrate how to attach multiple files -- just␣
↪use a list!
    return [attachment1, attachment2]
```

1.7.3 Arterial Spin Labelling Data

ASL is a BIDS protocol proposal that is fast on its way to being accepted into the official BIDS spec, but is still being reviewed and updated. At present, ASL in BIDS requires a special kind of events file, the *aslcontext* file. This is a TSV file not unlike the *events.tsv* file given for BOLD task data, but is used in this case to denote the order of label vs. control in the volumes. The file might look like this:

For this purpose, we can use the `AttachToSession()` function. You *could* do as above and read in a file on disk *within* the function, but you could be even cleverer and instead dynamically create this file:

```
def AttachToSession():

    NUM_VOLUMES=10
    data = ['control', 'label'] * NUM_VOLUMES
    data = '\n'.join(data)
    data = 'volume_type\n' + data # the data is now a string; perfect!

    output_file = {

      'name': '{subject}_{session}_aslcontext.tsv',
      'data': data,
      'type': 'text/tab-separated-values'
    }

    return output_file
```

This could be especially useful if you don't want to rely on external data files to curate your project. You can find out the correct number of LABEL-CONTROL pairs from the DICOM header info found in the output of

`fw-heudiconv-tabulate`, which will also help you hard code the extra ASL metadata and insert it into the `MetadataExtras` variable.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Index